Appunti di

Algoritmi e Strutture Dati

dalle lezioni di Linda Pagli e Roberto Grossi

Francesco Baldino

Secondo semestre a.a. 19/20

v1.0.1

Indice

Disclaimer

- Questi appunti DOVREBBERO essere completi, ma non ne sono sicuro al 100%. Se volete avvisarmi di errori o di parti mancanti, potete avvisarmi alla mia mail di dipartimento: baldino@mail.dm.unipi.it
- Questi appunti seguono approssimativamente l'ordine delle lezioni del corso. Alcuni argomenti sono stati anticipati o posticipati di poco per rendere più organizzata la presentazione degli argomenti.
- Alcune parti potrebbero essere spudoratamente copiate da Wikipedia poiché non riuscivo ad esprimerle in modo migliore.
- Continuo ripetutamente a dimenticarmi che in LaTeXdevo essere in math mode per poter scrivere i simboli $<$ e $>$. Se vi capita di vedere dei ¡ o ¿, sappiate che ¡ $=<$ e ¿ $=>$. Per favore segnalatemeli che li correggo.

Prerequisiti

Per seguire questi appunti è consigliabile aver studiato per il corso di Fondamenti di Programmazione, o avere comunque un'infarinatura generale di programmazione.

Contenuto

In questi appunti vengono riportate solo le lezioni tenute da Linda Pagli, con alcune aggiunte delle (poche) lezioni teoriche tenute da Roberto Grossi. Il contenuto delle esercitazioni tenute da Roberto Grossi non è riportato.

1 Lezione Introduttiva - 25/02/2020

1.1 Algoritmi

Che cos'è un algoritmo? Un algoritmo è una sequenza di passi "elementari" atti alla soluzione di un problema. Il tipo di problema da risolvere non deve necessariamente riguardare la matematica o l'informatica perché si parli di algoritmo: un algoritmo risolve problemi di natura arbitraria.
Il primo algoritmo noto e formalizzato è un algoritmo per la moltiplicazione tra due interi, basato esclusivamente sulla moltiplicazione e divisione per 2. Questo algoritmo, giunto a noi tramite il papiro di Rhind, prende il nome di Moltiplicazione Egizia.

1.2 Moltiplicazione Egizia

La proprietà che questo algoritmo sfrutta è la seguente:

$$A \times B = \begin{cases} \lfloor A/2 \rfloor \times 2B & \text{se A è pari} \\ \lfloor A/2 \rfloor \times 2B + B & \text{se A è dispari} \end{cases}$$

Supponiamo di avere in input due interi positivi A, B con A > B e vogliamo come output P = A×B.
Consideriamo il seguente algoritmo:
[Il seguente algoritmo, come tutti gli algoritmi in queste dispense, è scritto in "pseudocodice", ovvero un linguaggio non formale ma utile per spiegare a parole e formule il funzionamento dell'algoritmo].

```
MoltEgizia(A, B)
{
    P = 0;
    while(A != 0)
    {
        if(A == dispari)
        {
            P = P + B;
        }
        A = A/2;    // La maggiorparte dei linguaggi di
                    //  programmazione fanno automaticamente la
                    //  divisione arrotondata per difetto quando
                    //  si dividono due interi
        B = B*2;

    }
    return P;
}
```

Proviamolo su un esempio pratico, volendo calcolare 25×17:

A	B	P
25	17	0+17 = 17
12	34	17
6	68	17
3	136	17+136 = 153
1	272	153+232 = 425

La moltiplicazione egizia è usata anche nei computer poiché le operazioni di divisione per 2 (per difetto) e moltiplicazione per due si semplificano notevolmente quando si salvano i numeri in binario (base 2). Infatti:

- moltiplicazione per 2 = aggiungi uno zero alla fine del numero
- divisione per 2 per difetto = tronca l'ultima cifra del numero

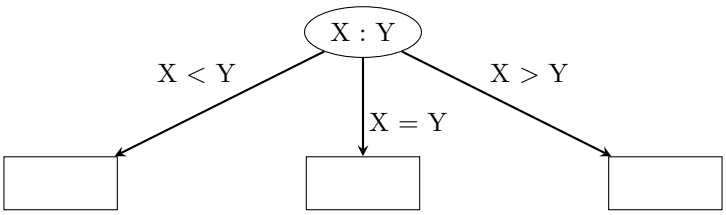
Si può dimostrare che il numero di operazioni da svolgere nel caso in cui A e B siano numeri di n cifre ciascuno è un $O(n^2)$ (che, circa, vuol dire che è una quantità che cresce al più velocemente tanto quanto n^2 (a meno di costante); spiegheremo meglio questa notazione in seguito).

1.3 Problema delle 12 monete

Problema

Ci sono 12 monete nelle quali è possibile che ci sia una moneta di peso diverso, che può essere più leggera o più pesante delle altre. Si dispone di una bilancia a piatti con la quale è possibile paragonare il peso di due gruppi di monete. Stabilire con al più 3 pesate se c'è effettivamente una moneta dal peso diverso, qual è e se è più leggera o più pesante.

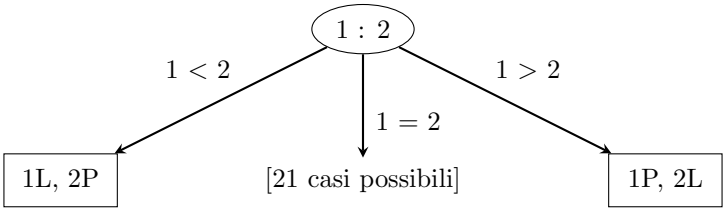
I risultati possibili sono 25 che indicheremo con 1L (corrispondente al caso in cui c'è una moneta diversa, è la 1 ed è più leggera delle altre), 1P (corrispondente al caso in cui c'è una moneta diversa, è la 1 ed è più pesante delle altre), 2L, 2P, ..., 12L, 12P, 0 (dove 0 vuol dire che non esistono monete dal peso diverso dalle altre).
Supponendo di confrontare due gruppi di monete X e Y, posso visualizzare i possibili risultati in questo modo:



Ogni pesata permette di distinguere 3 casi diversi, quindi con n pesate riesco a distinguere al più 3^n situazioni diverse. Per poter risolvere il problema devo riuscire a distinguere i 25 risultati possibili, quindi serve che $3^n \geq 25$, ovvero $n \geq 3$. Vale a dire che 3 è un limite inferiore al numero di pesate necessarie a risolvere il problema in generale.

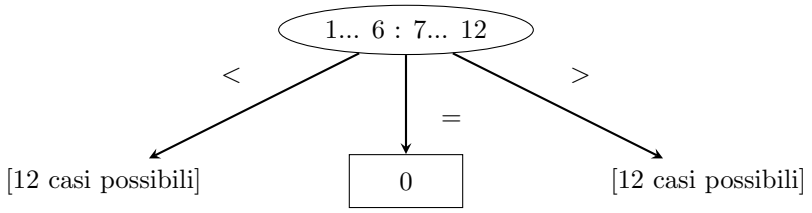
Strategie

Cerchiamo ora qualche strategia di risoluzione del problema.
Supponiamo di utilizzare la prima pesata per confrontare solo due monete tra di loro. Allora



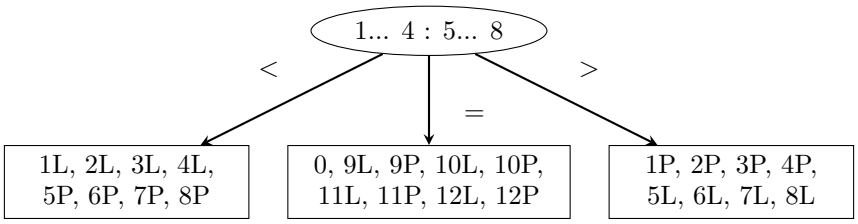
L'albero ottenuto non è bilanciato (ovvero il numero di casi possibili non è equidistribuito sui rami), quindi nel caso 1 e 2 risultassero uguali servirebbero in totale 4 pesate. Questa strategia non va bene.

Supponiamo di confrontare le prime 6 monete con le altre 6. Otterremmo



Anche in questo caso l'albero non è bilanciato e se i pesi dei due gruppi risultassero diversi servirebbero, anche in questo caso, 4 pesate.

Proviamo a pesare gruppi di 4:



In questo caso l'albero è ben bilanciato. Continuando con questa strategia del bilanciamento dei rami è possibile risolvere il problema con sole 3 pesate

2 Lezione Introduttiva - 27/02/2020

2.1 Indecidibilità e Teorema di Turing

Il Teorema di Turing [1] afferma che esistono problemi non decidibili, ovvero problemi per i quali non esistono algoritmi in grado di dare soluzioni in tempo finito su tutte le istanze del problema.
Un esempio di problema indecidibile, che può servire come accenno di dimostrazione, è il "problema della fermata" (o Halting Problem).

2.2 Problema della fermata

Ci si pone il seguente problema: dato un algoritmo A che prende come input D, è possibile determinare a priori in tempo finito se l'algoritmo si arresta?
Spieghiamo con un esempio concreto dando una "soluzione" per la congettura di Goldbach [2]. Cominciamo prima con il seguente algoritmo Primo(n) che determina se il numero n è un numero primo o meno.

```
Primo(n)
{
    f = 2;
    while(n % f != 0)
        f = f + 1;
    return (f == n);
}
```

Esibiamo ora un algoritmo che esibisce, qualora esista, un controesempio alla congettura di Goldbach.

```
Controesempio()
{
    n = 2;
    controesempio = TRUE;
    do
    {
        n = n + 2;
        controesempio = TRUE;
        for(p = 2; p <= n - 2; p++)
        {
            q = n-p;
```

[1] Alan Turing: 1912-1954, matematico Britannico, uno dei padri dell'informatica
[2] Congettura di Goldbach: "Ogni numero pari maggiore di 2 può essere scritto come somma di due numeri primi (che possono essere anche uguali)"

```
            if(Primo(q) && Primo(p)) controesempio = FALSE;
        }
    } while(!controesempio)
    return n;
}
```

Il motivo per cui abbiamo esibito questo algoritmo è per rendere chiaro un esempio di algoritmo per il quale non sia ovvio il fatto che termini o meno. Infatti, per le nostre conoscenze attuali, non sappiamo se esista un controesempio alla congettuda di Goldbach, e quindi non possiamo sapere se questo algoritmo termina.
Ritorniamo al problema di prima: dato un algoritmo A che prende come input D, è possibile determinare a priori in tempo finito se l'algoritmo si arresta?
La risposta è no, perché il problema della fermata è un problema indecidibile, ovvero non esiste nessun algoritmo tale che per ogni A e per ogni D è in grado di dire se A si arresta o meno.

Proviamo un abbozzo di dimostrazione per assurdo:

2.3 "Dimostrazione" dell'indecidibilità del problema della fermata

Supponiamo che esista un algoritmo Termina(A, D) che prende in input un algoritmo A(D) e l'input D di A e restituisca TRUE se l'algoritmo termina in tempo finito, false se l'algoritmo non si arresta in tempo finito.
Consideriamo ora un ulteriore algoritmo Paradosso(A) che prende come input a sua volta un algoritmo A:

```
Paradosso(A)
{
    while(Termina(A, A)
    {
    }
}
```

Per quanto sciocco l'algoritmo appena mostrato sia, esso rimane comunque un valido algoritmo. Osserviamo che Termina(A, A) è ben definito: sia perché nel caso in cui lo utilizzeremo adesso, A prende come input un algoritmo (e quindi A stesso è un input valido), sia perché un algoritmo può essere tradotto in una sequenza di caratteri o numeri, e può quindi essere dato come input generico ad un altro algoritmo.

Consideriamo ora Paradosso(Paradosso) (oppure P(P) abbreviando Paradosso = P). P(P) termina?

- P(P) termina $\Rightarrow$ Termina(P, P) = TRUE $\Rightarrow$ L'algoritmo non esce mai dal ciclo while $\Rightarrow$ P(P) non termina

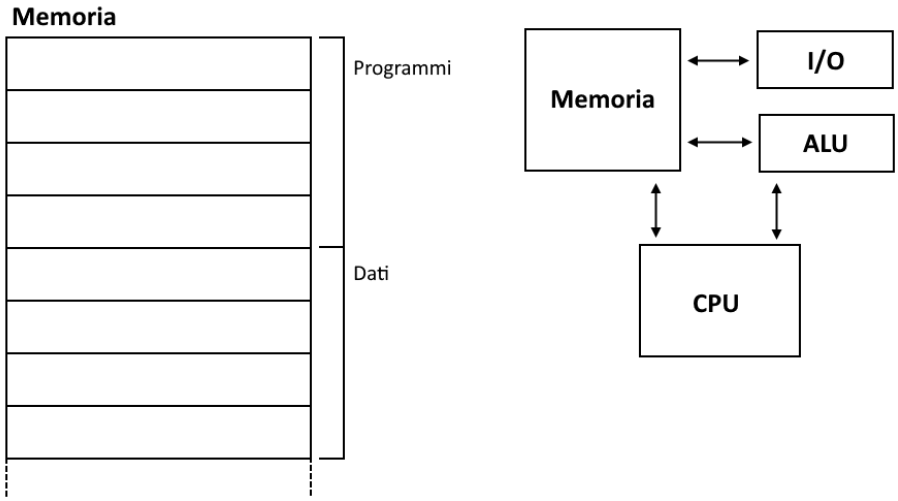
- P(P) non termina ⇒ Termina(P, P) = FALSE ⇒ L'algoritmo non entra mai nel ciclo while ⇒ P(P) termina

Arriviamo ad un assurdo, quindi non esiste tale algoritmo Termina(A, D).

3 03/03/2020

3.1 Modello RAM e Word Model

Per studiare la complessità in tempo e memoria utilizzata degli algoritmi, dobbiamo introdurre un modello semplificato di computer per poter fare delle assunzioni su come l'algoritmo verrà svolto (e quanto complesso sarà svolgere le operazioni "base").
Introduciamo quindi il modello RAM (Random Access Memory) di Von Neumann con i due seguenti diagrammi (abilmente ricreati in paint):



Nel modello RAM, i programmi e i dati vengono salvati nella memoria in celle, e il costo richiesto per accedere ad una qualsiasi di queste celle è $\Theta(1)$ (spiegeremo in seguito cosa vuol dire questa notazione, per adesso interpretatela come volesse dire che il tempo impiegato per accedere ad una qualsiasi di queste celle sia costante e non dipendente dalla cella che vogliamo).
Nel diagramma a destra le frecce indicano quali moduli comunicano con quali, dove "I/O" è l'unità di input e output (Input Output Unit), "ALU" è l'unità aritmetica e logica (Arithmetic-Logic Unit) e "CPU" è l'unità centrale di elaborazione (Central Processing Unit).
Nel modello RAM, le operazioni di base sono:

- **Operazioni aritmetiche** (+, -, *, /)
- **Operazioni di confronto** (==, !=, <, >)

- **Operazioni logiche** (AND, OR, XOR, NOT, ...)
- **Operazioni di trasferimento** (da e alla memoria)
- **Operazioni di controllo** (salti di programma)

Tutte queste operazioni richiedono costo costante $\Theta(1)$
Un'ulteriore ipotesi che aggiungeremo al modello RAM è che ogni dato che considereremo possa essere contenuto in una sola cella di memoria. Con questa ipotesi, il modello considerato prende il nome di Word Model.

Un modello alternativo (che non considereremo a meno che specificato) è il Bit Model, dove ogni cella di memoria rappresenta un singolo bit. La differenza principale tra i due modelli è nel tempo richiesto per compiere alcune operazioni. Ad esempio, sommare due numeri a n cifre (binarie) nel Word Model richiede una sola operazione, mentre nel Bit Model ne richiede n.

3.2 Complessità di un algoritmo (O, Ω, Θ)

Per poter valutare la qualità di un algoritmo, poniamoci l'obiettivo di poter valutare due delle principali caratteristiche di un algoritmo: il tempo richiesto e lo spazio richiesto.
Formalizziamo questi due termini:

- **Tempo**: numero di operazioni di base svolte dall'algoritmo, in funzione della dimensione n dei dati.
- **Spazio**: quantità di celle di memoria addizionali utilizzate dall'algoritmo in funzione della dimensione n dei dati.

Per dimensione n dei dati si intende in modo generico "quanto grandi" sono i dati, e può riferirsi ad esempio al valore assoluto di un numero dato in input, alla dimensione di un array dato in input, o altro.
Il tempo e lo spazio richiesto da un algoritmo, anche fissata la dimensione n dei dati, può dipendere da input a input. Per valutare la qualità di un algoritmo si considerano i seguenti "casi":

- **Caso pessimo**: il valore massimo di tempo e spazio richiesto su tutti gli input possibili (a parità di dimensione n).
- **Caso medio**: il valore medio di tempo e spazio richiesto su tutti gli input possibili (a parità di dimensione n).
- **Caso ottimo**: il valore minimo di tempo e spazio richiesto su tutti gli input possibili (a parità di dimensione n).

In generale il valore più utile è quello relativo al caso medio, anche se più difficile da calcolare. Il caso pessimo risulta comunque utile per poter dare un limite superiore al tempo e spazio richiesto. Il caso ottimo è tendenzialmente inutile.

Introduciamo ora la notazione O, Ω, Θ, inventata da Donald Knuth[3]

Definizione: Date due funzioni $f(n), g(n)$, vale che $f(n) \in O(g(n))$ (e si dice che f(n) "è un o grande di g(n)", o impropriamente $f(n) = O(g(n))$) se $\exists\ c, n_0$ costanti positive tali che $\forall n \geq n_0$ vale $f(n) \leq c \cdot g(n)$

Esempio: $f(n) = 7n^2 + 5n - 12 \Rightarrow f(n) = O(n^2)$

O fornisce un limite superiore al tasso di crescita di f.

Definizione: Date due funzioni $f(n), g(n)$, vale che $f(n) \in \Omega(g(n))$ (e si dice che f(n) "è un omega di g(n)", o impropriamente $f(n) = \Omega(g(n))$) se $\exists\ c, n_0$ costanti positive tali che $\forall n \geq n_0$ vale $f(n) \geq c \cdot g(n)$

Ω fornisce un limite inferiore al tasso di crescita di f.

Definizione: Date due funzioni $f(n), g(n)$, vale che $f(n) \in \Theta(g(n))$ (e si dice che f(n) "è theta di g(n)", o impropriamente $f(n) = \Theta(g(n))$) se vale contemporaneamente che $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$

Esempio:

- $f(n) = 3n^2 log(n) + 2n^4$ è $\Theta(n^4)$ ma non $\Theta(n^5)$. Infatti $f(n)$ è $O(n^5)$ ma non $\Omega(n^5)$
- $f(n) = 3n^3 log^2(n) + 4n log^3(n)$ è $\Theta(n^3 log^2(n))$
- $f(n) = 4n^{10} + 5n^2 + 6n - 2 = O(n^{10})$

Diciamo che un problema è "facile" se il migliore algoritmo risolutivo è del tipo $\Theta(n^k)$ (oppure $O(n^k)$), e si dice che è di complessità polinomiale.

Diciamo che un problema è "difficile" se il migliore algoritmo risolutivo è del tipo $\Theta(k^{g(n)})$ oppure $\Theta(n^{f(n)})$, e si dice che è di complessità esponenziale.

Analizziamo ora un problema per utilizzare nell'atto pratico le notazioni appena introdotte:

3.3 (O, Ω, Θ) in pratica

Problema

Input: a[0... n-1] array di n interi (sia positivi che negativi)
Output: il valore della somma del sottoarray a$[l...r]$ di somma massima (con $0 \leq l \leq r \leq n-1$)

[3]Donald Knuth: 1938, inventore del LaTeX, autore di "The art of Computer Programming"

Soluzione 1

```
Soluzione1(a)
{
    max = a[0];
    for(i = 0; i < n; i++)
    {
        for(j = i; j< n; j++)
        {
            sum = 0;
            for(k = i; k <= j; k++)
            {
                sum = sum + a[k];
            }
            if(sum > max) max = sum;
        }
    }
    return max;
}
```

Osserviamo che tutti e tre i cicli for vengono ripetuti sicuramente meno di n volte ciascuno. Inoltre, l'operazione di somma è $\Theta(1)$. Quindi il ciclo for più interno richiederà $O(n \cdot \Theta(1)) = O(n)$ operazioni, quello a metà $O(nO(n)) = O(n^2)$ operazioni e quello esterno $O(n^3)$ operazioni. L'algoritmo ha quindi una complessità di $O(n^3)$. Proviamo ora a dare un limite inferiore. Il numero di operazioni realmente richieste è:

$$\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}(j-i+1) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-i} j \geq \sum_{i=0}^{n-1}\frac{(n-i)^2}{2} = \sum_{j=1}^{n}\frac{j^2}{2} = \Omega(n^3)$$

Quindi questa soluzione è $\Theta(n^3)$

Soluzione 2

```
Soluzione2(a)
{
    max = a[0];
    for(i = 0; i < n; i++)
    {
        sum = 0;
        for(j = i; j< n; j++)
        {
            sum = sum + a[j];
            if(sum > max) max = sum;
        }
    }
```

```
    return max;
}
```

Con ragionamenti analoghi a quelli fatti per la soluzione 1, otteniamo che questa soluzione è $\Theta(n^2)$

Soluzione 3

Una soluzione ancora migliore può essere ottenuta studiando alcune proprietà del sottoarray cercato. In particolare, vale sia che tutti i prefissi del sottoarray (ovvero i sotto-sottoarray della forma a$[l...k]$ con $k < r$) sono a somma positiva, e che a$[l-1]$ deve essere un numero negativo (se $l \neq 0$). Se così non fosse, infatti, si potrebbe togliere il prefisso a somma negativa per ottenere un sottoarray di somma maggiore, oppure aggiungere al sottoarray anche a$[l-1]$ per ottenere un sottarray di somma maggiore.
Utilizziamo questi fatti per costruire un algoritmo migliore.

```
Soluzione3(a)
{
    max = a[0];
    sum = 0;
    for(i = 0; i < n; i++)
    {
        if(sum > 0) sum = sum + a[i];
        else sum = a[i];

        if(sum > max) max = sum;
    }
    return max;
}
```

E' facile vedere che questo algoritmo è $\Theta(n)$, molto migliore a $\Theta(n^2)$ o $\Theta(n^3)$.

4 10/03/2020

4.1 Il sorting

Spesso gran parte del tempo di calcolo è "sprecato" nella ricerca di dati. Se i dati sono ordinati, la ricerca richiede molto meno tempo. Diventa quindi interessante riuscire ad avere dati ordinati. Poniamoci quindi il seguente problema:

Sorting:

Input: $a_0, ..., a_{n-1}$ array di interi
Output: $a_{i0} \leq ... \leq a_{in-1}$ Vediamo ora alcune soluzioni, che prendono il nome di algoritmi di ordinamento.

4.2 Insertion Sort

L'algoritmo si basa sullo scorrere l'array da sinistra a destra, ordinando pian piano un sottoinsieme, inserendo l'elemento corrente nel sottoinsieme già ordinato.

Esempio:

$$5\ 9\ 18\ 21 \mid 7\ 4\ 15...$$
$$\downarrow$$
$$5\ 7\ 9\ 18\ 21 \mid 4\ 15...$$

dove l'elemento considerato è quello a destra della sbarretta e la sbarretta determina il sottoinsieme ordinato (ovvero il sottoarray alla sua sinistra).
L'algoritmo in pseudocodice è

```
InsertionSort(a)
{
    for(i = 1; i < n; i++)
    {
        prox = a[i];
        j = i;
        while(j > 0 AND a[j-1] > prox)
        {
            a[j] = a[j-1];
            j--;
        }
        a[j] = prox;
    }
}
```

Il ciclo più interno è un $O(n)$, quindi il tutto è $O(n^2)$. Possiamo dire che è $\Theta(n^2)$? No.
Infatti, se l'array è già ordinato (caso ottimo), la complessità è $\Theta(n)$.
Nel caso pessimo, invece, l'array è decrescente, nel qual caso la complessità è $\Theta(n^2)$.
In generale l'algoritmo è $O(n^2)$ e $\Omega(n)$.

Mostriamo ora la correttezza dell'algoritmo. Per farlo, dobbiamo trovare una proprietà invariante vera all'inizio, che rimanga vera ad ogni iterazione e che implichi, alla fine, l'ordinamento dell'array.

Proprietà invariante: all'i-esima iterazione, a[0... i-1] è ordinato.

- a[0] è ordinato.
- all'inizio della i+1-esima iterazione a[0...i-1] è ordinato, e dopo l'inserimento a[0... i] è ordinato.
- all'n-esima iterazione tutto l'array è ordinato.

4.3 SelectionSort

Il selection sort è un altro algoritmo "semplice" di ordinamento. All'iterazione i-esima si cerca il minimo di a[i... n-1] e lo si posiziona in posizione a[i]. I passi da compiere sono:

- **Passo 1:** Trova il minimo tra n elementi.
- **Passo 2:** Trova il minimo tra n-1 elementi.

...

- **Passo n-1:** Trova il minimo tra 2 elementi.

Osserviamo in oltre che per trovare il minimo tra k elementi generici ci vogliono k-1 confronti, e non esistono caso ottimo o caso pessimo.
In tutti i casi i confronti da fare sono $(n-1)+...+2+1 = \Theta(n^2)$.
Quindi il Selection Sort è $\Theta(n^2)$ mentre l'Insertion Sort è $O(n^2)$.
In pseudocodice diventa:

```
SelectionSort(a)
{
    for(i = 0; i < n; i++)
    {
        min = i;
        for(j = i+1; j < n; j++)
        {
            if(a[j] < a[min]) min = j;
        }
        if(min != i)
        {
            temp = a[i];
            a[i] = a[min];
            a[min] = temp;
        }
    }
}
```

Esistono algoritmi migliori? L'operazione da considerare su cui calcolare un limite inferiore è il confronto tra due elementi.

4.4 Merge Sort - Divide et Impera

Introduciamo il concetto di Divide et Impera, "Divide and Conquer", come paradigma di costruzione di algoritmi.

Divide et Impera, passi base:

- **Dividi** il problema da risolvere in sottoproblemi più piccoli,

- **Risolvi** i sottoproblemi ricorsivamente (o direttamente se abbastanza piccoli),
- **Combina** le soluzioni dei sottoproblemi per ottenere la soluzione del problema originale.

Forti di questo nuovo paradigma, introduciamo un nuovo algoritmo di sorting: il Merge Sort.
Il Merge Sort si basa sui seguenti passi:

- **Dividi**: dividi l'array di lunghezza n in due sottoarray di lunghezza $n/2$,
- **Risolvi**: se l'insieme è di un elemento è già ordinato, altrimenti applica ricorsivamente l'algoritmo,
- **Combina**: fai la fusione dei due sottoinsiemi ordinati di $n/2$ elementi ciascuno per ottenere l'insieme ordinato.

In pseudocodice diventa:

```
MergeSort(a, sx, dx)
{
    if(sx < dx)
    {
        cx = (sx + dx)/2;
        MergeSort(a, sx, cx);
        MergeSort(a, cx+1, dx);

        Merge(a, sx, cx, dx);
    }
}
```

Abbiamo utilizzato la funzione ausiliaria Merge(a, sx, cx, dx), la quale inserisce i valori dei due sottoarray (a[sx... cx] e a[cx+1... dx]) in un array di lunghezza uguale alla somma delle due lunghezze (b[0... dx-sx]), scorrendo entrambi i sottoarray da sinistra e inserendo i valori in modo ordinato. In pseudocodice è:

```
Merge(a, sx, cx, dx)
{
    i = sx;
    j = cx+1;
    k = 0;
    while(i <= cx AND j <= dx) {
        if(a[i] <= a[j])
        {
            b[k] = a[i];
            i++;
        }
        else
```

```
        {
            b[k] = a[j];
            j++;
        }
        k++;
    }
    while(i <= cx) {
        b[k] = a[i];
        i++;
        k++;
    }
    while(j <= dx) {
        b[k] = a[j];
        j++;
        k++;
    }
    for(i = sx; i <= dx; i++) a[i] = b[i-sx];
}
```

Osserviamo che la funzione Merge è $\Theta(n)$. Per calcolare invece la complessità di MergeSort, possiamo esprimere la sua complessità in modo ricorsivo. Detta T(n) la quantità di operazioni svolte da MergeSort su un array di lunghezza n, vale:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{altrimenti} \end{cases}$$

Iterando la scrittura su $T(n/2)$ otteniamo la forma generica

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + i\Theta(n)$$

e supponendo $n = 2^k$ con k intero, vale

$$T(n) = 2^k T(1) + k\Theta(n) = 2^{log_2 n}\Theta(1) + log_2 n\Theta(n) = \Theta(n \cdot log_2 n) = \Theta(n \cdot log(n))$$

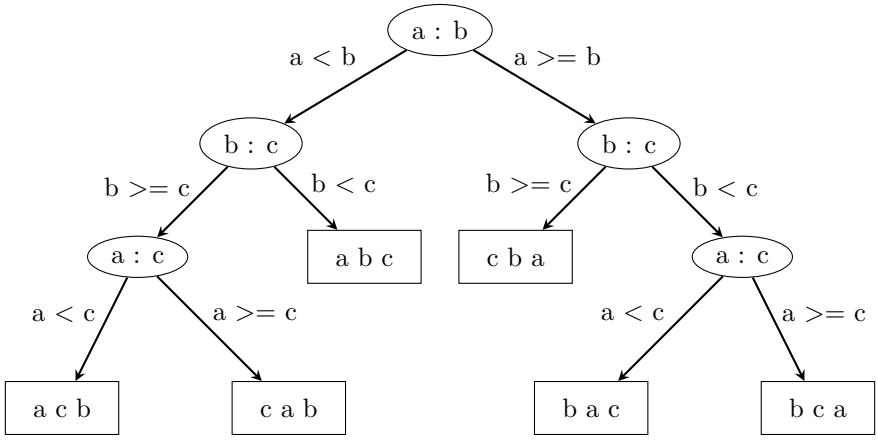
Quindi MergeSort è $\Theta(n \cdot log(n))$ tempo. Il "lato negativo" è che MergeSort usa, a differenza degli algoritmi visti in precedenza, spazio aggiuntivo non costante. Merge Sort è infatti $O(n)$ spazio.
Vedremo in seguito il teorema principale per le equazioni di ricorrenza per calcolare velocemente la complessità di algoritmi ricorsivi.

Abbiamo visto il Selection Sort ($\Theta(n^2)$), l'Insertion Sort ($O(n^2)$) e il Merge Sort ($\Theta(n \cdot log(n))$). E' possibile fare di meglio? Proveremo a determinare un limite inferiore nella prossima lezione.

5 12/03/2020

5.1 Limiti inferiori per il sorting

Cerchiamo un limite inferiore al problema del sorting.
Per il problema delle 12 monete abbiamo usato un albero di decisione con l'operazione della pesata per stabilire un limite inferiore al numero di pesate necessarie.
L'operazione base che considereremo è il confronto tra elementi.
Supponiamo di avere solo 3 elementi da confrontare, a b c. Consideriamo il seguente diagramma di confronti:



Consideriamo ora il problema generale, con n elementi da ordinare. La prima domanda a cui rispondere è quante sono le possibili soluzioni. Poiché non so come sono disposti gli elementi, la soluzione potrebbe essere una qualsiasi permutazione di n elementi, quindi $|S| = n!$
C'è poi da decidere come deve essere fatto l'albero di decisione. L'albero deve essere un albero binario il più bilanciato possibile, quindi con 2^0 elementi alla radice, 2^1 elementi al livello 1, ..., 2^i foglie al livello i-esimo.
Vogliamo che ci siano almeno tante foglie quante soluzioni possibili, quindi

$$
\begin{aligned}
&2^i \geq n! = |S| \\
&\quad \Downarrow \\
&i \geq log(n!) \approx \\
&\quad \approx log\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) = \\
&\quad = n \cdot log(n) - n \cdot log(e) + log(\sqrt{2\pi n}) = \\
&\quad = \Theta(n \cdot log(n))
\end{aligned}
$$

dove si è utilizzata l'approssimazione di Stirling per il fattoriale[4].
Abbiamo quindi dimostrato che per qualsiasi algoritmo di sorting basato sul

[4] Approssimazione di Stirling: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, per $n \to +\infty$

confronto di elementi, il numero di confronti è $\Omega(n \cdot log(n))$.
Poiché abbiamo trovato Merge Sort, che è $\Theta(n \cdot log(n))$ e abbiamo dimostrato che il problema di sorting è $\Omega(n \cdot log(n))$, il Merge Sort è un algoritmo ottimale in tempo (ma non in spazio).

5.2 Tecniche per limiti inferiori

1. Dimensione Input/Output
2. Eventi Contabili
3. Albero di decisione
4. Metodo dell'oracolo/avversario

Volendo trovare un limite inferiore per il sorting con il metodo (1) otterrei che il sorting è $\Omega(n)$, mentre con il metodo (3) abbiamo ottenuto $\Omega(n \cdot log(n))$.
Proviamo ad utilizzarli con altri problemi. Qual è il limite inferiore del problema di elencare tutte le permutazioni di n elementi? Utilizzando il metodo (1) ho che la dimensione dell'output è n!, quindi il problema è $\Omega(n!)$. Il metodo (1) è facile da usare ma funziona solo con i problemi ovvi.
Proviamo con un altro problema.

Problema

Trovare il massimo (o equivalentemente il minimo) elemento di un array.

Utilizziamo il metodo (2) contando il numero di confronti da fare.
Per ogni confronto si confrontano due elementi, e tutti gli elementi vanno confrontati almeno una volta, quindi servono almeno n/2 confronti.
Tuttavia vale che ogni elemento che non è il massimo deve "perdere" in almeno un confronto. Questo è l'evento contabile. Poiché gli elementi non massimi sono n-1, occorrono almeno n-1 confronti.
Poiché abbiamo un algoritmo che usa n-1 confronti (scansionando l'array da sinistra a destra) questo è un algoritmo ottimo.
Un altro algoritmo ottimo è l'algoritmo del Torneo, basato sul Divide et Impera.

```
Torneo(a, sx, dx)
{
    if(sx == dx) return a[sx];
    else
    {
        cx = (sx + dx)/2;
        m1 = Torneo(a, sx, cx);
        m2 = Torneo(a, cx+1, dx);

        return max(m1, m2);
    }
}
```

Per la quantità di confronti fatti, vale (assumendo senza perdita di generalità che n $= 2^i, i \in \mathbb{N}$)

$$
\begin{aligned}
C(1) &= 0 \\
C(n) &= 2C\left(\frac{n}{2}\right) + 1 = \\
&\dots \\
&= 2^i C\left(\frac{n}{2^i}\right) + \sum_{k=0}^{i-1} 2^k = \\
&= \sum_{k=0}^{i-1} 2^k = \\
&= 2^i - 1 = \\
&= n - 1
\end{aligned}
$$

quindi anche l'algoritmo del torneo è un algoritmo ottimo.

Il valore n-1 (ovvero il numero di confronti necessari) si poteva ottenere anche in un altro modo. Possiamo immaginare il "torneo" come un albero binario il più bilanciato possibile (o completamente bilanciato nel caso di $n = 2^i$) dove si comincia mettendo tutti i valori nelle foglie, e pian piano si riempiono i nodi interni facendo "scontrare" ogni nodo con il nodo con cui condivide il padre, e scrivendo nel padre il valore del nodo vincente (e iterando si otterrà in cima il valore massimo). Il numero di confronti da fare è quindi uguale al numero dei nodi interni dell'albero, che (come dimostreremo più avanti) è, nel caso di un albero binario completamente bilanciato con n foglie, n-1 nodi interni.
L'algoritmo "banale" e quello del torneo sono entrambi ottimali, ma quello banale utilziza un ciclo for che può risultare più efficiente di chiamate ricorsive. Tuttavia, l'algoritmo del torneo lascia spazio a confronti in parallelo.

Per analizzare il metodo dell'oracolo/avversario, consideriamo un problema leggermente diverso dal precedente.

Problema

Trovare il primo e secondo elemento massimo di un array.

Con l'algoritmo del torneo, si ottiene il massimo e il "secondo classificato". Ma il secondo classificato è necessariamente il secondo massimo?
Non per forza. Se per caso il massimo e il secondo massimo si incontrassero alla prima "partita", il secondo massimo non sarebbe mai arrivato in finale. Il secondo massimo potrebbe essere uno qualsiasi delle persone che hanno perso con il massimo. Il vero secondo, per arrivare in finale, deve stare nella "metà dei nodi" (ovvero, considerando la visualizzazione utilizzata prima con un albero binario che rappresenta tutto il torneo, l'albero può essere diviso in una metà

destra e una metà sinistra considerando il sottoalbero destro e il sottoalbero sinistro) dove non sta il primo massimo, e ha quindi $\frac{n}{2(n-1)} \approx \frac{1}{2}$ di probabilità di arrivare in finale. Questa ingiustizia fu osservata per la prima volta da Lewis Carrol[5].
Cerchiamo algoritmi funzionanti per il primo e il secondo massimo:

Algoritmo banale

Scansiona l'array per il massimo, togli il massimo e trova il nuovo massimo. I confronti da fare sono n-1 + n-2 = 2n-3.
Si può trovare anche un algoritmo tramite Divide et Impera, che però porta a 3n-2 confronti.

Algoritmo ottimo

L'algoritmo ottimo è suggerito dall'algoritmo del torneo, ovvero creare un secondo torneo con quelli che hanno perso con il massimo. Sicuramente il secondo massimo si troverà nell'insieme dei valori che hanno perso in uno scontro contro il massimo, poiché sicuramente il secondo massimo non può vincere (quindi deve perdere con qualcuno) e non può perdere con altri valori che non siano il massimo (poiché in quanto secondo massimo, vince contro tutti i valori che non sono il massimo). Chiamiamo questo algoritmo l'algoritmo del doppio torneo.
Il numero di perdenti con il campione (ovvero il numero di partecipanti al secondo torneo) è uguale alla profondità dell'albero, quindi C(n) = n-1 + log(n)-1 = n + log(n) - 2.

[Premessa: nella parte che segue di dispense, si cercherà di utilizzare il metodo dell'oracolo per stabilire un limite inferiore al numero di confronti necessario per risolvere il problema del primo e secondo massimo, far vedere che questo limite è proprio n + log(n) - 2 e mostrare quindi che l'algoritmo del doppio torneo è ottimo. Ora, a me francamente il ragionamento fatto non torna. Mi spiego: i ragionamenti che seguono sono logici e non dicono nulla di falso. La parte secondo me fallace è la conclusione. In breve, il metodo dell'oracolo è un procedimento per cui, analizzando un algoritmo, si decide che ad ogni confronto (o generico momento con più di un output possibile), l'output considererato sarà quello che fornisce meno informazione possibile (questa scelta dell'output si dice "intervento dell'oracolo"), riconducendoci quindi al caso pessimo. In questo modo capiremo che tecniche utilizzare per limitare il caso pessimo e ne seguirà, nel nostro caso, che il numero di incontri che il massimo deve fare è log(n). La parte che non mi convince è come questo (ovvero mostrare che un algoritmo che trova il massimo con log(n) incontri è anche quello che minimizza il caso pessimo, e che quindi l'algoritmo del (singolo) torneo sia ottimale) implichi che la scelta di fare due tornei dia l'algoritmo ottimale. Chiaramente quello che segue dimostra che l'algoritmo mostrato (ovvero con tornei sviluppati come un

[5]Lewis Carrol: 1832-1898, fu uno scrittore, matematico, fotografo, logico e prete anglicano britannico dell'età vittoriana

albero binario perfettamente bilanciato) sia ottimale tra gli algoritmi che fanno 2 tornei, ma sinceramente non vedo perché non debba esistere un altro algoritmo completamente diverso che usa meno confronti. Chi mi garantisce che non esiste qualche proprietà del secondo minimo che mi permette di trovarlo in, che ne so, n + log(n) - 7 confronti? Se qualcuno ha le idee più chiare a riguardo, per favore contattatemi]

Stabiliamo un limite inferiore a questo problema con il metodo (4).
Servono n-1 confronti per stabilire il primo massimo, e questo già lo sappiamo. Quanti ne servono per stabilire il secondo?
Considero M(j) = numero di giocatori risultati perdenti (non solo in scontro diretto, anche perdenti in quanto giocatori sconfitti da giocatori che poi hanno perso contro il massimo) dal campione dopo j incontri. Voglio mostrare che M(j) $= 2^j - 1$. L' "intervento dell'oracolo" consiste nel considerare sempre l'opzione peggiore in ogni confronto che si fa, per ricondurci e studiare il caso pessimo.
Considero il confronto x:y. Poiché so che l'oracolo farà sì che il risultato di questo confronto sia quello che mi garantisce meno informazione possibile, voglio che l'informazione sia "bilanciata" tra i due possibili output. All'atto pratico, voglio che, se sto confrontanto x e y, sia noto che sia x che y sono maggiori di altri k elementi ciascuno, poiché se x fosse maggiore di k elementi e y maggiore di h elementi con k << h, l'informazione non sarebbe bilanciata e l'oracolo mi farebbe ricevere la quantità di informazione minore possibile.
Con un albero bilanciato, però, vale per M(j) la seguente equazione ricorsiva: M(j) = 2M(j-1) + 1. Infatti, dopo ogni incontro, il numero di giocatori risultati perdenti contro il massimo sarà uguale alla quantità di giocatori risultati perdenti al massimo prima dell'incontro, più la quantità di giocatori risultati perdenti allo sfidante del massimo di questo incontro (che per quanto abbiamo detto deve essere uguale a M(j-1) anche lui, altrimenti l'informazione sarebbe sbilanciata), +1 ovvero il giocatore sfidante del massimo di questo incontro (che giustamente ha perso).
Con questa equazione ricorsiva e con il valore di partenza M(0) = 0 otteniamo facilmente M(j) $= 2^j - 1$.
Per essere sicuri che quello considerato sia il massimo, deve valere M(j) $\geq$ n-1, da cui j $\geq$ log(n).
Quindi il massimo deve fare almeno log(n) incontri, e log(n) sono gli elementi perdenti dal campione in modo diretto, dunque un algoritmo che organizza un torneo con log(n) elementi è ottimale.

6 17/03/2020

6.1 Equazioni di ricorrenza

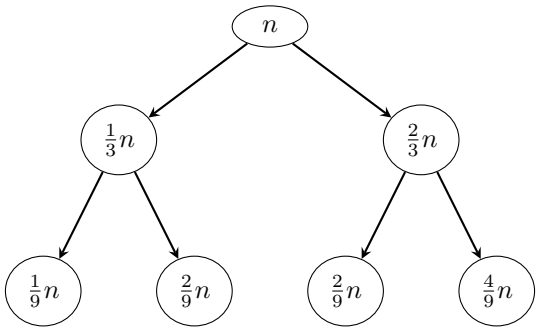
Vediamo ora alcuni metodi per calcolare delle formule generiche per delle equazioni di ricorrenza per la complessità degli algoritmi.

1. Metodo iterativo

2. Albero di ricorsione

Il primo metodo è anche quello che abbiamo già utilizzato in passato, e consiste nello sviluppare iterativamente la formula ricorsiva finché non si riesce a costruire una formula generica per l'iterazione i-esima, e continuare finché non si arriva al caso base.

Vediamo adesso invece un esempio del secondo metodo.
Sia T(n) = T(n/3) + T(2n/3) + n. Costruiamo il seguente albero di ricorsione:



dove il numero nel cerchietto rappresenta quel "+n", ovvero il numero di operazioni compiute ad ogni passo ricorsivo non dipendente dalla ricorsione, con:

$$
\begin{aligned}
&n = n \\
&\frac{1}{3}n + \frac{2}{3}n = n \\
&\frac{1}{9}n + \frac{2}{9}n + \frac{2}{9}n + \frac{4}{9}n = n \\
&...
\end{aligned}
$$

quindi ad ogni livello vengono svolte n operazioni.

Quanti livelli sono in tutto? L'albero ricorsivo continua finiché non trova il caso base, ovvero T(1). La condizione di arresto quindi è che il ramo dal valore più alto (quello tutto a destra) raggiunga il valore 1, ovvero deve valere che $n \cdot (2/3)^i = 1$, quindi

$$i = log_{3/2}(n) = \Theta(log(n))$$

Allora vale che il costo totale di T è il numero di operazioni ad ogni livello per il numero di livelli. Poiché è possibile che alcuni livelli verso la fine non siano completi (i rami a sinistra tendenzialmente si arrestano prima dei rami a destra, perché hanno valori più piccoli) sappiamo che ad ogni livello il numero di operazioni è $O(n)$ (infatti, se il livello è completo la somma delle operazioni da fare vale n, mentre se non è completo è $< n$, quindi in generale vale che la somma è $O(n)$).
Allora vale che $T(n) = O(n)\Theta(log(n)) = O(n \cdot log(n))$

6.2 Teorema dell'esperto

Il teorema fondamentale (o principale) delle equazioni di ricorrenza, o teorema dell'esperto, dice che: sia

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq n_0 \\ aT(\frac{n}{b}) + f(n) & \text{altrimenti} \end{cases}$$

per un certo n_0, con $a \geq 1$, $b > 1$ e dove n/b vuol dire $\lfloor n/b \rfloor$ oppure $\lceil n/b \rceil$. Supponiamo che le condizioni iniziali richiedano tempo costante, e che possano quindi essere ignorate. Si confrontano ora le seguenti due quantità:

$$n^{log_b(a)} \text{ e } f(n)$$

Caso 1 se vale che:

$$\exists \varepsilon > 0 \text{ tc } f(n) = O\left(n^{log_b(a)-\varepsilon}\right)$$

allora

$$T(n) = \Theta\left(n^{log_b(a)}\right)$$

Caso 2 se vale che:

$$f(n) = \Theta\left(n^{log_b(a)}\right)$$

allora

$$T(n) = \Theta\left(n^{log_b(a)} \cdot log(n)\right)$$

Caso 3 se vale che:

$$\exists \varepsilon > 0 \text{ tc } f(n) = \Omega\left(n^{log_b(a)+\varepsilon}\right)$$

$$\text{e } \exists c \in [0,1] \text{ tc } af\left(\frac{n}{b}\right) \leq cf(n) \text{ [Condizione di regolarità]}$$

allora

$$T(n) = \Theta(f(n))$$

Vediamo ora alcuni esempi:

- Sia $T(n) = 9T\left(\frac{n}{3}\right) + n$
  allora $f(n) = n$ e $n^{log_3 9} = n^2$.
  Vale $f(n) = O\left(n^{2-\varepsilon}\right)$ per $\varepsilon \leq 1$.
  Siamo quindi nel caso 1, quindi vale $T(n) = \Theta(n^2)$
- Sia $T(n) = T\left(\frac{2}{3}n\right) + 1$
  allora $f(n) = n^0$ e $n^{log_{3/2} 1} = n^0$.
  Siamo quindi nel caso 2, quindi vale $T(n) = \Theta(log(n))$

- Sia $T(n) = 3T\left(\frac{n}{4}\right) + n \cdot log(n)$
  allora $f(n) = n \cdot log(n)$ e $n^{log_b a} = n^{log_4 3} \approx n^{0.79}$.
  Vale $f(n) = \Omega\left(n^{0.79+\varepsilon}\right)$ per $\varepsilon < 0.21$
  Se valesse anche la condizione di regolarità, saremmo nel caso 3. Verifichiamo che vale la condizione di regolarità:

  $$af\left(\frac{n}{b}\right) \leq cf(n)$$

  ovvero

  $$3\frac{n}{4} \cdot log\left(\frac{n}{4}\right) \leq c \cdot n \cdot log(n)$$

  che è vero per $c \geq 3/4$.
  Allora $T(n) = \Theta(n \cdot log(n))$

- Sia $T(n) = 2T\left(\frac{n}{2}\right) + n \cdot log(n)$
  allora $f(n) = n \cdot log(n)$ e $n^{log_2 2} = n$.
  Esiste ε tale che

  $$f(n) \stackrel{?}{=} \Omega\left(n^{1+\varepsilon}\right)$$

  No! Quindi non si applica il teorema.

Un altro metodo che può funzionare per trovare formule generali ad equazioni di ricorsioni è, come sempre, il trial and error.

6.3 Dimostrazione del teorema dell'esperto

$$\begin{aligned}
T(n) &= aT\left(\frac{n}{b}\right) + f(n) = \\
&= a^2T\left(\frac{n}{b^2}\right) + af\left(\frac{n}{b}\right) + f(n) = \\
&... \\
&= a^iT\left(\frac{n}{b^i}\right) + a^{i-1}f\left(\frac{n}{b^{i-1}}\right) + ... + f(n) = \\
&... \\
&= a^{log_b n}\Theta(1) + \sum_{i=0}^{log_b n - 1} a^i f\left(\frac{n}{b^i}\right) = \\
&= \Theta\left(a^{log_b n}\right) + \sum_{i=0}^{log_b n - 1} a^i f\left(\frac{n}{b^i}\right)
\end{aligned}$$

Da questa formula dovrebbe essere facile ricavare i risultati dei vari casi.

6.4 Esempi pratici

Torneo

Tornando all'algoritmo del torneo, possiamo vedere che $T(n) = 2T(n/2) + \Theta(1)$ è del primo caso, quindi vale $T(n) = \Theta(n)$.

Moltiplicazione nel modello RAM Bit Model

Come algoritmo di moltiplicazione abbiamo visto ad inizio corso la moltiplicazione egizia. Un altro algoritmo per la moltipicazione potrebbe essere la traduzione in codice della moltiplicazione "carta e penna", ovvero la moltiplicazione in colonna. Entrambe risultano essere $\Theta(n^2)$. Riusciamo a trovare un algoritmo migliore con il Divide et Impera?
Siano A, B interi di n cifre. Sia

$$A = A_1 10^{n/2} + A_2,\ B = B_1 10^{n/2} + B_2$$

$$\Rightarrow AB = A_1 B_1 10^n + (A_1 B_2 + A_2 B_1) 10^{n/2} + A_2 B_2$$

dove A_1, A_2, B_1, B_2 sono numeri di n/2 cifre.
Posso quindi sviluppare il seguente algoritmo:

```
Molt(A, B, N)
{
    if(n==1) return A*B;
    else
    {
        // dividi A e B in A1, A2, B1 e B2
        [A1, A2] = Dividi(A, N);
        [B1, B2] = Dividi(B, N);
        // la funzione Dividi non verrà esplicitata

        x = Molt(A1, B1, n/2);
        z = Molt(A2, B2, n/2);
        y = Molt(A1, B2, n/2) + Molt(A2, B1, n/2);

        return x*(10^n) + y*(10^(n/2)) + z;
    }
}
```

Allora $T(n) = 4T(n/2) + \Theta(n) \Rightarrow T(n) = \Theta(n^2)$ tramite il teorema dell'esperto.

Osservando però che $(A_1 + A_2) * (B_1 + B_2) = A_1 B_1 + A_1 B_2 + A_2 B_1 + A_2 B_2$ e in particolare $A_1 B_2 + A_2 B_1 = (A_1 + A_2) * (B_1 + B_2) - A_1 B_1 - A_2 B_2$, poiché conosco già i valori di $A_1 B_1$ e $A_2 B_2$, posso ottimizzare e fare una moltiplicazione in meno per calcolare y.

7 19/03/20

7.1 Quicksort

Supponiamo diavere un array A di lunghezza N contenente soltanto 0 e 1. Lo scopo è di ordinare A mediante scambi del contenuto di due posizioni.
Introduciamo questa funzione ausiliaria che semplificherà il codice:

```
Scambia(A, i, j)
{
    temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}
```

Osservazione: senza essere costretti ad utilizzare scambia, basterebbe contare il numero di 1 (o di 0) e sovrascrivere A con prima tutti k zeri e poi n-k uni.

Utilizzando Scambia, utilizziamo due puntatori i, j, e descriviamo brevemente un algoritmo che fa al caso nostro:

- scorro i da sinistra finché non trovo un 1
- scorro j da destra finché non trovo uno 0
- Scambia(i, j)
- ripeto e mi fermo quanto i $\geq$ j

Osserviamo che l'invariante è che in un qualsiasi momento, a sinistra di i ci sono solo zeri e a destra di j ci sono solo uni, quindi quanto i = j (o i $\geq$ j in base a come si scrive l'algoritmo e quando si fa il controllo), l'array è ordinato.

Tenendo a mente l'algoritmo appena descritto, occupiamoci adesso di un altro caso, ovvero di un array di interi su cui vogliamo operare una distribuzione o partizione. Ciò vuol dire scegliere un elemento, detto pivot, e riorganizzare l'array in modo che sia composto da tre blocchi: il primo, a sinistra, composto solo da elementi minori del pivot; il secondo, centrale, contenente solo il pivot; il terzo, a destra, contenente solo elementi maggiori del pivot.
Prendiamo il nostro array originale e applichiamo l'algoritmo descritto precedentemente trattando tutti gli elementi minori del pivot come degli zeri, e tutti gli elementi maggiori del pivot come uni.
Descriviamo ora un algoritmo che applica la distribuzione su un sottoarray di A (array di interi), in modo da poterlo poi applicare su sottosezioni di A sempre più piccole:

```
Distribuzione(A, sx, px, dx)
{
    if(px != dx) Scambia(A, px, dx);

    i = sx;
    j = dx;
    while(i <= j)
    {
        while((i <= j) AND (A[i] <= A[dx])) i++;
        while((i <= j) AND (A[j] >= A[dx])) j--;
```

```
        if(i < j) Scambia(A, i, j);
    }

    if(i != dx) Scambia(A, i, dx);
                // Non l'avevamo richiesto esplicitamente
                //  all' inizio, ma tornerà comodo ritornare la
                //  posizione finale del pivot nel sottoarray,
                //  che a fine dell'algoritmo sarà in posizione
                //  i (dopo essere stato messo da parte in
    return i;   //  posizione dx)
}
```

Applicando questo algoritmo otterremo un array A[0... n-1] il cui sottoarray A[sx... dx] è distribuito secondo il valore iniziale di A[px].
Ciò suggerisce uno schema di Divide et Impera.

```
QuickSort(A, sx, dx)
{
    if(sx < dx)
    {
        [scegli px pivot in {sx... dx}]
        // Discuteremo la scelta del pivot tra poco, per adesso
        //  limitiamoci a sceglierlo in qualche modo
        rango = Distribuzione(A, sx, px, dx);
        QuickSort(A, sx, rango-1);
        QuickSort(A, rango+1, dx);
    }
}
```

Osserviamo che Distribuzione è O(n) in tempo e $\Theta(1)$ in spazio. Il costo del QuickSort sarà, quindi, T(n) = T(rango-1) + T(n-rango) + O(n)
Idealmente ci piacerebbe avere rango $\approx$ n/2, così da avere T(n) = 2T(n/2) + O(n), ma non c'è garanzia di ciò. Anzi, nei casi pessimi di rango = 1 o rango = n, l'algoritmo diventa $O(n^2)$. Il costo medio di QuickSort è, in realtà, $O(n \cdot log(n))$.
Introduciamo ora un concetto utile per cercare di ottenere quanto più spesso possibile il caso medio.

7.2 Randomization

Per quanto QuickSort sia $O(n \cdot log(n))$, il caso pessimo è $O(n^2)$ e in base a come sono organizzati i dati con cui stiamo lavorando (che non sono necessariamente puramente casuali) è possibile che ci si imbatta più spesso nel caso pessimo di quanto non lo si faccia con dati puramente casuali. Introduciamo quindi un'idea potente: usiamo la scelta di un numero casuale per sfuggire provatamente al caso sfavorevole.

Con il supporto della funzione random(), che restituisce un numero casuale in $[0, 1]$ scelto in modo uniforme, e round(), che arrotonda un numero non intero all'intero più vicino, riscriviamo il QuickSort nel seguente modo:

```
QuickSortRandomized(A, sx, dx)
{
    if(sx < dx)
    {
        px = sx + round((dx - sx)*random());
        rango = Distribuzione(A, sx, px, dx);
        QuickSortRandomized(A, sx, rango-1);
        QuickSortRandomized(A, rango+1, dx);
    }
}
```

Poiché scegliamo il pivot in modo casuale e non sempre allo stesso modo, il costo medio di questo algoritmo è $O(n \cdot log(n))$ indipendentemente da come sono distribuiti i dati in input. Mostriamolo:

7.3 Dimostrazione costo medio di QuickSort randomizzato

Per analizzare QS randomizzato, prendiamo A' = A ordinato. Diciamo A' = $z_1 < z_2 < ... < z_n$.

Osservazioni:

1. se z_i e z_j vengono confrontati, uno dei due è un pivot.
2. per ogni i, j, vale che z_i e z_j sono confrontati al più una volta.
3. se z_i e z_j sono nella stessa porzione da ordinare, allora tale porzione contiene almeno $j - i + 1$ elementi, ovvero $z_i, z_{i+1}, ..., z_j$

Sia X_{ij} una variabile indicatrice tale che

$$X_{ij} = \begin{cases} 1 & \text{se } z_i \text{ e } z_j \text{ sono confrontati} \\ 0 & \text{altrimenti} \end{cases}$$

Osservazione: presa una generica variabile indicatrice Y tale che la probabilità che faccia 1 sia p $\in [0, 1]$, allora il suo valore atteso è $E[Y] = 1 \cdot p + 0 \cdot (1 - p) = p$.

Utilizzando l'osservazione 1 e l'osservazione 2 otteniamo che la quantità definita come

$$X = \sum_{i<j} X_{ij}$$

è uguale al numero di coppie di elementi confrontati.
Inoltre vale che $\mathbb{P}[X_{ij} = 1] \leq \frac{2}{j-i+1}$ poiché uno tra z_i e z_j deve essere un pivot

e la probabilità di scegliere uno dei due come pivot è al massimo $\frac{2}{j-i+1}$ dato che almeno gli elementi $z_i, z_{i+1}, ..., z_j$ sono tutti nella partizione da ordinare e il pivot è scelto in modo casuale.

Il costo dell'algoritmo è O(n + X), e poiché X è una variabile aleatoria, il costo medio è O(n + $E[X]$). Calcoliamo $E[X]$:

$$E[X] = \sum_{i<j} E[X_{ij}] \leq \sum_{i<j} \frac{2}{j-i+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = O(n \cdot log(n))$$

Quindi il costo medio è O(n + n · log(n)) = O(n · log(n)).

8 24/03/2020

8.1 Moltiplicazione di Matrici

Esibiamo un altro esempio di Divide et Impera.
Siano A e B matrici $n \times n$. Diciamo A = (a_{ij}), B = (b_{ij}) e C = A×B = $(c_i j)$ dove $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$. Il numero di moltiplicazione necessarie per ottenere C in questo modo è $\Theta(n^3)$.
Cosa si può dire del limite inferiore? Dobbiamo produrre n^2 elementi distinti, quindi sicuramente (utilizzando il metodo della dimensione dell'output per la determinazione di un limite inferiore) sicuramete è $\Omega(n^2)$. Supponiamo senza perdita di generalità che n = 2^i con i intero. Siano

$$A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right), B = \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right), C = \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right)$$

dove:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Questa scomposizione suggerisce la struttura del Divide et Impera. Consideriamo il seguente algoritmo:

```
MM(A, B, n)
{
    if(n==1) return [a * b];
    else
    {
        [A11, A12, A21, A22] = Scomponi(A);
        [B11, B12, B21, B22] = Scomponi(B);
        // Il modo in cui si scompongono A e B nelle sottomatrici
```

```
        //  dipende da come sono implementate le matrici nel
        //  linguaggio utilizzato. Immaginiamo di poterle
        //  semplicemente scommporre così, per comodità

        C11 = MM(A11, B11, n/2) + MM(A12, B21, n/2);
        C12 = MM(A11, B12, n/2) + MM(A12, B22, n/2);
        C21 = MM(A21, B11, n/2) + MM(A22, B21, n/2);
        C22 = MM(A21, B12, n/2) + MM(A22, B22, n/2);

        // Allo stesso modo in cui immaginiamo una funzione
        //  Scomponi, immaginiamo anche una funzione Ricomponi
        return Ricomponi(C11, C12, C21, C22);
    }
}
```

Vale che T(n) = 8T(n/2) + $\Theta(n^2)$. Utilizzanto il teorema dell'esperto otteniamo:
$n^{log_b a} = n^3$ e $f(n) = \Theta(n^2)$
$n^2 = O(n^{3-\varepsilon})$ con $\varepsilon \leq 1$.
Quindi siamo nel caso 1, per cui T(n) = $\Theta(n^3)$. Non abbiamo migliorato il costo dell'algoritmo.

Il primo algoritmo in grado di migliorare il costo fu scoperto da Strassen[6]

Algoritmo di Strassen

consideriamo:
$X_1 = (A_{11} + A_{22})(B_{11} + B_{22})$
$X_2 = (A_{21} + A_{22})B_{11}$
$X_3 = A_{11}(A_{12} + B_{22})$
$X_4 = A_{22}(B_{21} - B_{11})$
$X_5 = (A_{11} + A_{12})B_{22}$
$X_6 = (A_{21} + A_{22})(B_{11} + B_{12})$
$X_1 = (A_{12} + A_{22})(B_{21} + B_{22})$
da cui otteniamo
$C_{11} = X_1 + X_4 - X_5 + X_7$
$C_{12} = X_3 + X_5$
$C_{21} = X_2 + X_4$
$C_{22} = X_1 + X_3 - X_2 + X_6$

Allora vale che $T(n) = 7T(\frac{n}{2}) + 18c\frac{n^2}{4} = 7T(\frac{n}{2}) + \Theta(n^2)$, quindi $T(n) = \Theta(n^{log_2 7}) \approx \Theta(n^{2.81\cdots})$[7].

[6] Volker Strassen: 1936, matematico Tedesco

[7] Non è attualmente l'algoritmo migliore. Nel tempo si sono seguiti l'algoritmo di Coppersmith–Winograd con un costo di $O(n^{2.375477})$ del 1990, migliorato nel 2010 da Andrew Stothers con un costo di $O(n^{2.374})$, migliorato nel 2011 da Virginia Vassilevska Williams con un

8.2 3-Partition

Prima di cominciare con i primi esempi di strutture dati, vediamo velocemente un miglioramento del QuickSort.

Problema

Sia A un array di n elementi tra 0, 1, 2. Vogliamo in output il vettore ordinato senza usare contatori ma solo scambiando elementi (in modo analogo a come abbiamo fatto prima di vedere QuickSort).

Immaginiamo un algoritmo descritto dal seguente passo generico.
Supponiamo che all'inizio del passo generico l'array sia così composto:
[0 | 1 $|a_j|$ $*$ | 2]

dove il primo segmento (di tutti zeri) va dall'indice 0 all'indice i (inclusi), il secondo segmento (di tutti uni) va dall'indice i+1 all'indice j-1 (inclusi), in posizione j abbiamo l'elemento a_j considerato in questo passo, da j+1 a k-1 (inclusi) abbiamo la parte di array non ancora smistata, e da k a n-1 (inclusi) l'ultimo segmento, composto da soli due.

Se A[j] = 1, incremento j.
Se A[j] = 0, incremento i, scambio A[i] con A[j] e poi incremento j.
Se A[j] = 2, decremento k e scambio A[j] con A[k].

Termino quando j = k.
In pseudocodice diventa:

```
Ordina012(A)
{
    i = -1;
    j = 0;
    k = n;
    while(j < k)
    {
        if(A[j] == 0)
        {
            i++;
            Scambia(A, i, j);
            j++;
        }
        else if(A[j] == 2)
        {
            k--;
            Scambia(A, j, k);
```

costo di $O(n^{2.3728642})$, migliorato nel 2014 da François Le Gall con un costo di $O(n^{2.3728639})$. Non sappiamo ancora nulla sul limite inferiore

```
        }
        else j++;
    }
}
```

Il costo di questo algoritmo è $\Theta(n)$.
Miglioriamo ora il QuickSort. Per ogni pivot, dividiamo l'array in tre segmenti: il primo con elementi tutti minori del pivot, il secondo con elementi uguali al pivot, il terzo con elementi maggiori. Definiamo un algoritmo 3Partition(A, p, r) con il quale scriveremo una nuova versione di QuickSort che risulterà molto più veloce nel caso in cui nell'array ci siano molte ripetizioni:

```
3Partition(A, p, r)
{
    i = p-1;
    j = p;
    k = r;
    x = A[r];
    while(j < k)
    {
        if(A[j] < x)
        {
            i++;
            Scambia(A, i, j);
            j++;
        }
        else if(A[j] > x)
        {
            k--;
            Scambia(A, j, k);
        }
        else j++;
    }
    Scambia(A, j, r);
    return (i, j);
}
```

Il passo generico del quicksort migliorato sarà quindi prima ordinare un sottoarray, e poi chiamare ricorsivamente solo sul segmento sinistro e sul segmento destro del sottoarray, lasciando stare il segmento centrale che è già ordinato.

8.3 Strutture Dati

Vediamo ora alcuni tipi di strutture dati.
Una "coda semplice" è un qualsiasi tipo di array FIFO (First In First Out, ovvero il primo elemento inserito è anche il primo ad essere estratto. Si oppone al LIFO, Last In First Out dove l'ultimo elemento inserito è il primo ad essere

estratto).
In una coda semplice le operazioni che si richiede di poter fare sono:

1. isEmpty: controlla se la coda è vuota
2. First: restituisce il primo elemento
3. Enqueue: aggiunge un elemento alla coda
4. Dequeue: restituisce e rimuove il primo elemento

Nella coda semplice tutte queste operazioni sono $\Theta(1)$.
Una "coda con priorità" è una struttura dati più complessa della coda semplice, che permette di associare ad ogni elemento nella coda la priorità con cui vogliamo che venga estratto. La struttura quindi non è più di tipo FIFO, ma tale che l'elemento estratto con First o Dequeue è l'elemento con priorità massima. Le operazioni richieste sono sempre le stesse, e il costo dipende da come si decide di implementare la coda di priorità. Vediamone alcuni:

	Array non ordinato	Array ordinato	Lista
(2)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
(3)	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
(4)	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Nella prossima lezione vedremo un altro modo di creare una coda di priorità.

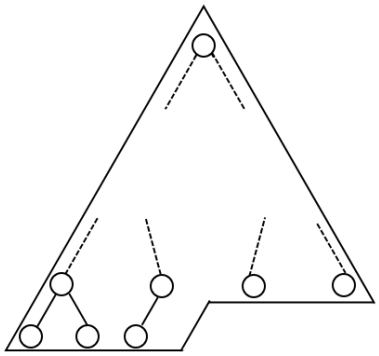
9 26/03/20

9.1 Heap

Un altro modo per implementare una coda di priorità è l'heap (dall'inglese, "mucchio"). L'heap è un "albero binario completo addossato a sinistra". Ovvero, detto n il numero di nodi, allora vale che:

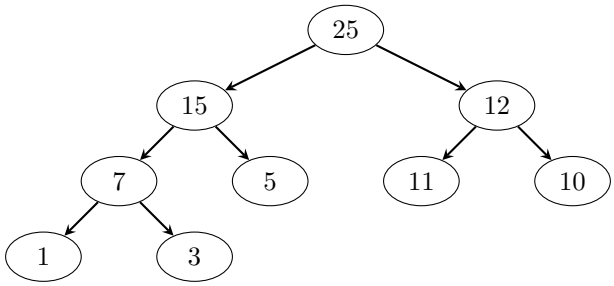
- se $n = 2^i - 1$, allora l'albero è completo.
- se $n \neq 2^i - 1$, allora si avrà un albero completo fino al penultimo livello, e poi le foglie tutte adossate a sinistra

Spiegandolo con un'immagine, un albero binario completo addossato a sinistra è il seguente:



Nota: Ogni elemento della coda, ovvero ogni nodo dell'albero, sarà una coppia della forma e = (valore, priorità), dove valore è il valore degli elementi che vogliamo ordinare (di tipo intero, delle stringhe, quello che si vuole) e priorità è la priorità associata a quel valore.
Gli heap si distinguono in max heap e min heap. Nel max heap ogni elemento è maggiore di tutti gli elementi nel suo sottoalbero (ovvero il sottoalbero di cui lui è radice). In questo caso la radice conterrà l'elemento con priorità massima. Nel min heap accade il contrario.

Esempio:



(In questo esempio i nodi, anche se normalmente sarebbero coppie (valore, priorità), vengono rappresentati esclusivamente come la loro priorità, poiché il valore del nodo è assolutamente indipendente dalla priorità e non fornisce nessuna informazione sulla struttura dell'heap)
Poiché l'albero è adossato a sinistra, può essere facilmente salvato in un array immaginando di leggere i nodi livello per livello da sinistra a destra (quindi in questo caso l'array che rappresenta l'heap sarebbe [25, 15, 12, 7, 5, 11, 10, 1, 3]). La struttura permette comunque di spostarci da padre a figlio. Infatti:

- Dato un nodo in posizione i, il suo figlio sinistro sarà in posizione 2i+1, quello destro in posizione 2i+2
- Dato un nodo in posizione i, suo padre sarà in posizione $\lfloor \frac{i-1}{2} \rfloor$

Si definisce come "altezza" di un albero la distanza massima tra la radice e una foglia.

Proposizioni: Dato un albero binario completo di altezza h, vale:

- N_h = numero dei nodi = $2^{h+1} - 1$
- F_h = numero delle foglie = 2^h
- I_h = numero dei nodi interni = $2^h - 1$

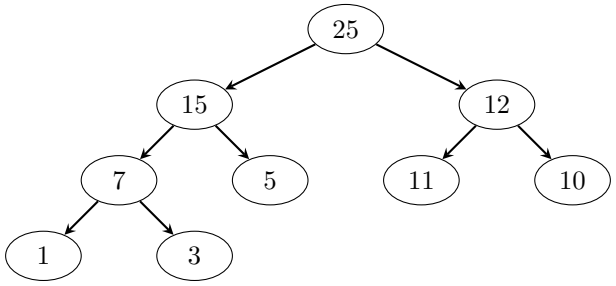
Dimostrazioni: Per induzione su h:
Per il passo base h = 1 tutto ok.
Per il passo induttivo, consideriamo un albero binario completo di altezza h. Allora il sottoalbero di altezza h-1 è nell'hp induttiva, quindi ha $2^h - 1$ nodi. Allora il numero dei nodi interni dell'albero di altezza h è $I_h = 2^h - 1$. Vale poi che $F_h = 2F_{h-1} = 2 \cdot 2^{h-1} = 2^h$ e infine $N_h = I_h + F_h = 2^{h+1} - 1$. □
In particolare, un albero binario completo di n nodi ha altezza $log_2(n+1) - 1 = \Theta(log(n))$.

Studiamo ora la relazione tra l'altezza e il numero di nodi in un heap.
Un heap di altezza h ha almeno 2^h nodi (ovvero $2^h - 1$ nodi del sottoalbero completo di altezza h-1 più una sola foglia al livello h), e al più $2^{h+1} - 1$ (nel caso di un albero binario completo di altezza h). Allora $2^h \leq n < 2^{h+1}$. Quindi

$$h \leq log_2 n < h + 1 \Rightarrow \begin{cases} h \leq log_2 n \\ h > log_2 n - 1 \end{cases} \Rightarrow h = \lfloor log_2 n \rfloor = \Theta(log(n))$$

9.2 Enqueue, Dequeue e RiorganizzaHeap

Le operazioni svolgibili su un heap sono le stesse 4 richieste per una generica coda con priorità, quindi isEmpty, First (o Max), Enqueue e Dequeue. L'heap è strutturato per realizzare efficientemente le (2), (3) e (4).
Prendiamo l'heap di esempio di prima (salvato nell'array [25 | ... | 3] di lunghezza 9):



Supponiamo di voler inserire un nuovo elemento di priorità 20. L'algoritmo di enqueue è così strutturato: prima si aggiunge il 20 nell'ultima posizione

(ovvero nella "prima foglia mancante" partendo da sinistra, in questo caso il figlio sinistro di 5), poi si confronta con il padre:

- Se l'elemento aggiunto è minore o uguale del padre, allora è rispettata la condizione di Max Heap, e lo si lascia lì. Abbiamo finito.
- Se l'elemento aggiunto è maggiore del padre, si scambiano i due elementi. Si ripete considerando come "nuovo elemento" quello appena "sollevato" di livello (ovvero messo nella posizione di padre).

Il numero massimo di confronti che si fanno nell'enqueue è uguale all'altezza dell'albero (e accade quando l'elemento nuovo è anche l'elemento di priorità massima e quindi "risale" tutto l'albero, confronto dopo confronto). Poiché l'altezza dell'albero è $\Theta(log(n))$, anche la complessità di Enqueue è $\Theta(log(n))$. Vediamo ora Enqueue in pseudocodice. Il significato di alcune righe di codice non risulterà subito ovvissimo, le spiegheremo subito dopo.

```
Enqueue(e)
{
    VerificaRaddoppio();
    heaparray[heapsize] = e;
    heapsize++;
    RiorganizzaHeap(heapsize-1);
}
```

Il primo commento da fare è su heaparray: questo è l'array dove fisicamente vengono salvati i nodi dell'albero.
Il secondo commento da fare è su VerificaRaddoppio(): la versione in pseudocodice di questo algoritmo verrà mostrato in seguito; per adesso limitiamoci a spiegare cos'è e perché lo utilizziamo.
In generale, aggiungere o togliere un elemento da un array, se si richiede che l'array contenga solo gli elementi necessari e non posizioni vuote "fittizie", è $\Theta(n)$. Poiché stiamo cercando di ottenere un algoritmko $\Theta(log(n))$ dobbiamo cercare di smorzare questo costo.
Il modo per farlo è "spalmare" il suo costo su più chiamate dell'algoritmo enqueue. Con "spalmarlo" si intende farlo solo una volta ogni k chiamate dell'algoritmo enqueue, così che il suo costo diventi $\Theta(n/k)$ (ovviamente con k dipendente da n e non costante, altrimenti non si avrebbe nessun guadagno).
Il modo in cui riusciamo a spalmare il costo di ridimensionamento dell'array è avendo un array con alcune posizioni finali vuote, e salvando in un'altra variabile (in questo caso heapsize) qual è la dimensione del sottoarray che contiene effettivamente dei valori (N.B: poiché questo sottoarray comincia dalla prima posizione dell'array, heapsize può anche essere visto come la quantità di elementi contenuti, visto che di fatto è la stessa cosa).
In questo modo, finché abbiamo ancora qualche posizione fittizia libera possiamo aggiungere elementi in $\Theta(1)$ (poiché basta sovrascrivere l'elemento dell'array). Nel momento in cui si finiscono le posizioni fittizie allora si raddoppia la dimensione dell'array. In questo modo, il costo dell'incremento di dimensione (che

abbiamo detto essere $\Theta(n)$) viene applicato solo una volta ogni $n/2$ volte, quindi anche se il costo della singola chiamata può essere $\Theta(n)$, il costo medio è $\approx \Theta(n/(n/2)) = \Theta(1)$ (avremo una dimostrazione più formale di ciò in seguito).
La funzione VerificaRaddoppio() fa esattamente quello che abbiamo appena detto: verifica se nell'array heaparray (che sarà di una certa dimensione 2^k e non di dimensione heapsize) ci sono ancora posizioni vuote libere. Se ci sono, non fa niente. Se non ci sono, raddoppia la dimensione di heaparray.
Ciò fatto, l'algoritmo aggiunge (come spiegato) il nuovo elemento in ultima posizione, incrementa la dimensione dell' "array effettivo" di 1 e riorganizza la struttura dell'heap facendo riscalare l'ultimo elemento. Anche lo pseudocodice di RiorganizzaHeap verrà mostrato in seguito e non ora.

Vediamo ora l'eliminazione del massimo.
L'algoritmo di dequeue è così strutturato: prima si rimuove il nodo radice, che contiene l'elemento di priorità massima. Poi si sposta l'ultimo elemento (la foglia più a destra dell'ultimo livello) al posto della radice. Dopo di che si comincia a riorganizzare l'heap. Si prende l'elemento appena sposto (inizialmente la ex-ultima foglia) e lo si confronta con i figli

- Se l'elemento considerato è maggiore di entrambi i figli, allora la struttura del Max heap è rispettata. Si termina qui.
- Se l'elemento considerato è minore di almeno uno dei due figli, si scambiano il padre e il maggiore dei figli. Si reitera prendendo come elemento considerato l'ex-Padre.

Il numero di confronti massimo avviene nel caso in cui l'ultima foglia, "scendendo" dalla radice, percorra tutta l'altezza dell'albero. Quindi anche questo algoritmo è $\Theta(log(n))$.
Vediamolo in pseudocodice:

```
Dequeue()
{
    if(!isEmpty)
    {
        max = heaparray[0];

        heaparray[0] = heaparray[heapsize-1];
        heapsize--;
        RiorganizzaHeap(0);

        VerificaDimezzamento();

        return max;
    }
}
```

In questo pseudocodice, VerificaDimezzamento svolge una funzione analoga a quella di VerificaRaddoppio nell'enqueue: se la dimensione dell'array effettivo(heaparray[0... heapsize-1]) scende sotto una certa soglia (tendenzialmente n/4. in seguito spiegheremo perché proprio questo valore), si dimezza la dimensione dell'array totale.
Vediamo ora RiorganizzaHeap.

RiorganizzaHeap

Input: un heap dove solo l'elemento i-esimo è fuoriposto.
Output: heap riordinato.

Nel caso di enqueue, l'elemento fuoriposto era quello appena aggiunto (in posizione heapsize-1). Nel dequeue era l'elemento spostato in radice (in posizione 0).
Chiamiamo "bottom-up" il procedimento per cui (nell'enqueue) un elemento in basso fuoriposto risale l'heap. Chiamiamo "top-down" il procedimento per cui (nel dequeue) un elemento in alto fuoriposto scende l'heap. Vediamo ora RiorganizzaHeap in pseudocodice:

```
RiorganizzaHeap(i)
{
    //--- bottom-up ---//
    while((i > 0) AND (heaparray[i] > heaparray[Padre(i)]))
    {
        Scambia(heaparray, i, Padre(i));
        i = Padre(i);
    }

    //--- top-down ---//
    while((FiglioSinistro(i) < heapsize) AND
          (i != MigliorePadreFigli(i)))
    {
        migliore = MigliorePadreFigli(i);
        Scambia(heaparray, i, migliore);
        i = migliore;
    }
}
```

In questo pseudocodice abbiamo supposto di disporre delle seguenti funzioni elementari:

- Padre(i): ritorna l'indice del padre di i
- FiglioSinistro(i): ritorna l'indice del figlio sinistro di i
- MigliorePadreFigli(i): ritorna l'indice del nodo con priorità maggiore tra i (il padre), il suo figlio sinistro e il suo figlio destro

Le condizioni (i > 0) e (FiglioSinistro(i) < heapsize) controllano che l'elemento che si sta spostando non abbia percorso tutto l'albero, poiché in quel caso sicuramente sarebbe nella posizione giusta (non avendo altri elemento sopra/sotto con cui confrontarsi) e bisogna terminare l'algoritmo.

9.3 HeapSort

Possiamo usare un heap per sviluppare un nuovo algoritmo di sorting.
Definiamo il seguente algoritmo. HeapSort:

1. Costruisco un maxheap con gli n elementi dell'array
2. Comincio con k = n
3. Scambio la radice con l'elemento in posizione k-1
4. Chiamo RiorganizzaHeap considerando solo l'array dei primi k-1 elementi
5. Decremento k e ritorno a (2)

In questo modo ad ogni ciclo il primo elemento (ovvero l'elemento di priorità massima) viene messo in fondo all'heap considerato, e lì viene lasciato fisso poiché RiorganizzaHeap considererà solo gli elementi prima di lui. Così facendo si aggiungono, in ordine, gli elementi di priorità massima. A fine algoritmo l'array conterrà gli elementi in priorità crescente.
Il costo di creazione iniziale dell'heap è di $O(n \cdot log(n))$, poiché vengono chiamati n enqueue da $O(log(n))$ ciascuno. Il costo delle riorganizzazioni è $O(n \cdot log(n))$, quindi il costo totale è anch'esso $O(n \cdot log(n))$.
Vediamo ora l'heapsort in pseudocodice:

```
HeapSort(heaparray)
{
    heapsize = 0;
    for(i = 0; i < n; i++)
        Enqueue(heaparray(i));

    while(heapsize > 0)
    {
        Scambia(heaparray, 0, heapsize-1);
        heapsize--;
        RiorganizzaHeap(0);
    }
}
```

HeapSort è un algoritmo ottimo sia in termini di tempo che in termini di spazio. Infatti, in tempo è $O(n \cdot log(n))$, che abbiamo mostrato essere un limite inferiore per il sorting. In spazio è invece $\Theta(1)$ poiché non usa spazio aggiuntivo ma solo l'array fornito inizialmente, sovrascrivendolo pian piano (si dice che fa tutto "in loco").

10 31/03/2020

10.1 Ricerca nell'heap

Supponiamo di avere un array organizzato in un maxheap.
Poniamoci il problema della ricerca in un heap. Ricerca(a,k):

- caso ottimo: k > max, allora l'elemento cercato non esiste, $\Theta(1)$
- caso pessimo: k < nodo per ogni nodo visitato, $\Theta(n)$

Quindi ha complessità $O(n)$, come la ricerca sequenziale, a differenza di estrazione e inserzione che sono $O(log(n))$.
Un problema leggermente diverso che possiamo porci è la ricerca del minimo in un maxheap. Osservando che il minimo deve necessariamente trovarsi su una foglia (non può essere padre di nessuno perché dovrebbe essere maggiore di qualcuno, ma è il minimo) possiamo limitare gli elementi tra cui cercare a solo le foglie.

Proposizione: il numero di foglie in un heap è $\lceil \frac{n}{2} \rceil$.
Dimostriamolo per induzione.
Per il passo base n = 2 è banale.
Consideriamo passo induttivo. Quando aggiungiamo un elemento ad un heap si genera una nuova foglia, e possiamo distinguere in due casi:

- La nuova foglia è un figlio sinistro. Allora n è dispari e n+1 è pari. Il numero totale di foglie non cambia, poiché anche ne stiamo aggiungendo una, il padre della nuova foglia era anche lui una foglia prima dell'aggiunta, e ora non lo è più. Aggiungendo e togliendo una foglia il totale non cambia. Inoltre, se n è dispari, $\lceil \frac{n}{2} \rceil = \lceil \frac{n+1}{2} \rceil$, quindi torna
- La nuova foglia è un figlio destro. Allora n è pari, e per ragionamenti analoghi a quelli fatti prima il numero di foglie incrementa di 1. Inoltre, poiché n è pari, $\lceil \frac{n+1}{2} \rceil = \lceil \frac{n}{2} \rceil + 1$, quindi torna. □

Osserviamo, inoltre, che poiché il numero di nodi interni in un heap è $\lfloor \frac{n}{2} \rfloor$ e poiché nell'array in cui viene salvato l'heap compaiono prima tutti i nodi interni e poi le foglie, $\lfloor \frac{n}{2} \rfloor$ è anche la posizione della prima foglia, quindi possiamo definire in modo molto semplice la "limitazione alle foglie" della ricerca. Basterà infatti cercare solo nel sottoarray heaparray[$\lfloor \frac{n}{2} \rfloor$... n-1].

Problema

Verificare se un array di n elementi sia organizzato come un maxheap.

Sappiamo che questa cosa accade se per ogni nodo interno vale che esso è maggiore di entrambi i figli. Scriviamo quindi un algoritmo in pseudocodice:

```
IsHeap(A, n)
{
    for(i = 0; i < floor(n/2); i++)
    {
        if(A[i] < A[2i + 1]) return FALSE;
        else if((2i+2 < n) AND (A[i] < A[2i + 2])) return FALSE;
    }
    return TRUE;
}
```

Osserviamo che la prima condizione dell' elseif() è necessaria poiché non è detto che un nodo interno abbia necessariamente un figlio destro (mentre non era necessaria per il primo if poiché ogni nodo interno ha necessariamente un figlio sinistro).
Questo algoritmo è $O(n)$.

Problema

Dato un array, esibire il k-esimo elemento in ordine di grandezza.

Vediamo ora qualche soluzione del problema.
Premettiamo che esiste una procedura, detta BuildHeap, che permette di costruire un heap con costo $O(n)$.

Soluzione 1

Una soluzione può essere, come abbiamo visto poco fa, ordinare l'array con HeapSort e poi selezionare il k-esimo elemento, A[k]. Sia utilizzando BuildHeap, sia costruendo l'heap con n enqueue, il costo è di $O(n \cdot log(n))$.

Soluzione 2

Un'altra soluzione può essere costruire un minheap e fare k estrazioni. In questo modo, la k-esima estrazione restituirà il k-esimo elemento in ordine di grandezza. Il costo(utilizzando BuildHeap) è di $O(n + k \cdot log(n))$.

Soluzione 3

E' possibile raggiungere la complessità $O(n \cdot log(k))$. L'idea è quella di costruire un maxheap di k elementi prendendo i primi k elementi dell'array (con costo $O(k)$ grazie a BuildHeap o costo $O(k \cdot log(k))$ con l'inserzione).
A questo punro, per ogni $i > k$ confrontio A[i] con il massimo dell'heap costruito, e:

- se A[i] > max, scartiamo A[i].

- altrimenti si fa prima dequeue e poi enqueue(A[i]).

Alla fine l'heap contiene i k elementi più piccoli dell'array, con il k-esimo elemento in ordine di grandezza nella radice. Il costo di questa soluzione è $O(k + n \cdot log(k)) = O(n \cdot log(k))$. Vediamo lo pseudocodice di questa soluzione:

```
Selection(A, k)
{
    H = nuovoarray(k);
    for(i = 0; i < k; i++)
        Enqueue(H, A[i]);

    for(i = k; i < n; i++)
    {
        if(A[i] < H[0])
        {
            Dequeue(H);
            Enqueue(H, A[i]);
        }
    }

    return H[0];
}
```

Questa soluzione torna particolarmente utile nel caso in cui i dati arrivino man mano, nel quale caso si parla di Modello Streaming.

Osserviamo che è possibile, con una modifica del QuickSort randomizzato, ottenere una soluzione la problema del k-esimo elemento con costo medio $O(n)$.

Aggiungiamo brevemente, prima di concludere, che un albero binario non è l'unico modo per strutturare un heap. Un altro modo per strutturare un heap è, ad esempio, un maxheap ternario, ovvero usando un maxheap basato su un albero ternario invece di un albero binario. In questo caso ci troveremmo a lavorare, invece che con figli sinistri e figli destri, con primi, secondi e terzi figli. Il vantaggio di un heap ternario è che, sebbene la complessità delle operazioni resti sempre logaritmica, l'altezza dell'albero risulta $log_3 n$ invece di $log_2 n$, quindi sempre migliore di un fattore costante.

10.2 Array dinamici/di dimensione variabile

Concludiamo l'heap vedendo l'implementazione di un array di dimensione variabile, con VerificaRaddoppio() e VerificaDimezzamento(), che abbiamo utilizzato per l'heap.
Sia A il nostro array, $d = \dim(A)$, ovvero la lunghezza totale di A, e sia n la quantità di valori memorizzati in A, (nel caso dell'heap è l'heapsize). Vale sicuramente $n \leq d$.

Vediamo l'implementazione di inserimento e eliminazione in un array di dimensione variabile.

Inserzione:

- se $n + 1 \leq d$, allora c'è posto per inserire il nuovo elemento. Inserisce l'elemento e basta
- altrimenti, raddoppiamo A. Ovvero, allochiamo un vettore B di dimensione $2 \times d$, copiando A in B[0... d-1], chiamando poi free(A) e sovrascrivendo A = B

Eliminazione:
Rimuovi l'elemento e decrementa n di 1. Se $n \leq d/4$ dimezza l'array, altrimenti non fare niente.

Osserviamo che la soglia sotto la quale bisogna andare per far scattare il dimezzamento nell'eliminazione è $d/4$ e non $d/2$, sebbene la dimensione finale dell'array diventa $d/2$ e non $d/4$. Il motivo di ciò è che nello sfortuito caso in cui n si trovi ad oscillare molto frequentemente sopra e sotto una stessa potenza di 2, si raddoppierebbe e dimezzerebbe in continuazione l'array, perdendo il vantaggio della complessità ammortizzata su più chiamate.
Mettendo invece la soglia a $d/4$ ci si assicura che tra un dimezzamento e un raddoppiamento passi almeno un certo numero di chiamate di inserimento, così da poter trarre vantaggio dal costo ammortizzato.

Vediamo ora lo pseudocodice:

```
VerificaRaddoppio()
{
    if(n==d)
    {
        b = nuovoarray(2*d);
        for(i = 0; i < n; i++)
            B[i] = A[i];
        d = 2*d;
        free(A);
        A = B;
    }
}

VerificaDimezzamento()
{
    if((d > 1) AND (n==floor(d/4)))
    {
        b = nuovoarray(d/2);
        for(i = 0; i < n; i++)
            B[i] = A[i];
```

```
        d = d/2;
        free(A);
        A = B;
    }
}
```

Entrambi gli algoritmi costano $O(n)$, rendendo quindi l'inserzione e l'eliminazione nell'heap $O(log(n) + n) = O(n)$. E' interessante però studiare il costo ammortizzato dell'inserzione e eliminazione, ovvero lo studio del costo in n chiamate.

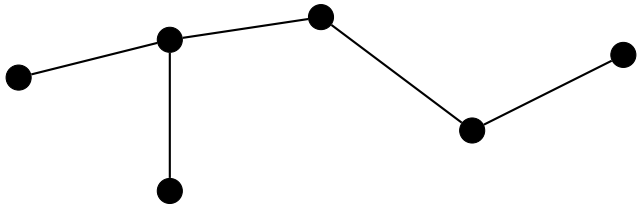
Teorema: n operazioni di inserimento e cancellazione in un array dinamico costano $O(n)$ (quindi virtualmente ogni operazione costa $\Theta(1)$).

Dimostriamolo: dopo un raddoppio abbiamo n = d+1 elementi in un array di dimensione $2 \times d$, quindi servono almeno altre n-1 inserzioni per un nuovo raddoppio.
Dopo un dimezzamento ci sono n = d/4 elementi, quindi ci vogliono almeno altre n/2 cancellazioni per un nuovo dimezzamento. I costi $O(n)$ sono quindi ammortizzati sulle $\Omega(n)$ operazioni che lo hanno "causato". □
Il costo ammortizzato è quindi $\Theta(1)$.

11 02/04/2020

11.1 Alberi come strutture dati

Si dice "albero" (generico) un grafo connesso privo di cicli. Un albero ha quindi, necessariamente, n nodi e n-1 archi. Un esempio di albero libero è:

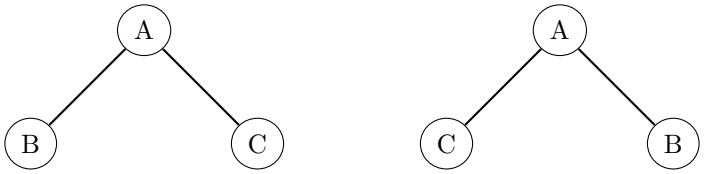


Gli alberi che considereremo saranno tutti alberi con radice (ovvero con un nodo che verrà considerato "sorgente"). Fissata un nodo come radice, gli altri vengono disposti per livelli in base alla distanza (intesa come numero di archi da percorrere). Come imparerete a scoprire, molta della terminologia relativa agli alberi viene dalla botanica e dagli alberi genialogici.

11.2 Alberi binari

Cominciamo introducendo gli alberi ordinati. Si dice albero ordinato un albero dove si tiene conto dell'ordine in cui vengono scritti i figli di un nodo. In

particolare, prendendoli come alberi ordinati, i due seguenti alberi non sono uguali:



La definizione di albero binario è una definizione ricorsiva. Si dice che T è un albero binario se:

- T è vuoto, oppure
- T è composto da tre sottoinsiemi:
  - radice
  - un sottoalbero sinistro, anche lui binario
  - un sottoalbero destro, anche lui binario

Osservazione: La definizione di albero binario si generalizza agli alberi k-ari (ternari, quaternari etc...)

Gli alberi binari tornano particolarmente comodi per salvare dati in memoria. A differenza degli alberi ordinati generici, gli alberi binari hanno nodi con un numero fissato di figli, il che permette di salvarli comodamente in memoria, ad esempio con il seguente struct:

$$\text{nodo u} \begin{cases} \text{u.dato} \\ \text{u.sx} \\ \text{u.dx} \end{cases}$$

Per un albero binario, conoscendo solo il puntatore alla radice, non è nota a priori la quantità di nodi. Scriviamo il seguente algoritmo che, prendendo in input il puntatore alla radice, restituisce il numero di nodi dell'albero:

```
Dimensione(u)
{
    if(u == null) return 0;
    else
    {
        dimSx = Dimensione(u.sx);
        dimDx = Dimensione(u.dx);
        return dimSx + dimDx + 1;
    }
}
```

Calcoliamo la complessità di questo algoritmo.
Verrebbe spontaneo dire che $T(n) = 2T(\frac{n}{2}) + \Theta(1)$. Tuttavia non c'è garanzia che l'albero si divida a metà su ogni nodo. Quello che sappiamo è che $T(n) = T(n_s) + T(n_d) + \Theta(1)$, ma è un problema che non sappiamo risolvere. Poiché vale però che $n_s + n_d + 1 = n$, un'ipotesi che vale la pena avanzare è che $T(n) = \Theta(n)$. Effettivamente, vale che $T(n) = \Theta(n)$, e si verifica facilmente per induzione forte.

Esibiamo invece un algoritmo per calcolare l'altezza di un albero binario conoscendo solo il puntatore alla radice:

```
Altezza(u)
{
    if(u == null)
        return -1;
    else
        return max(Altezza(u.sx), Altezza(u.dx)) + 1;
}
```

Un altro problema che ci si può porre parlando di alberi è quello di elencare tutti i nodi di un albero come una successione di elementi. Gli algoritmi atti a ciò vengono definiti algoritmi di visita, e sono generalmente ricorsivi, composti da tre momenti:

1. Visita/Stampa del valore del nodo correntemente analizzato
2. Chiamata ricorsiva sulla radice del sottoalbero sinistro
3. Chiamata ricorsiva sulla radice del sottoalbero destro.

L'ordine in cui queste parti vengono eseguite (in particolare, il momento in cui avviene la visita del nodo rispetto alle due chiamate ricorsive) determina tre algoritmi di visita diversi:

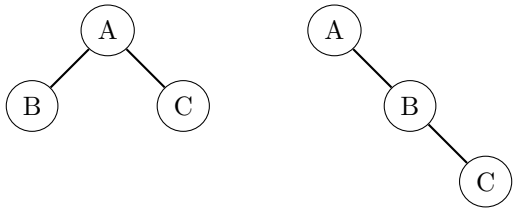
- Visita anticipata (Preorder traversal[8]): (1) → (2) → (3)
- Visita simmetrica (Inorder traversal): (2) → (1) → (3)
- Visita posticipata (Postorder traversal): (2) → (3) → (1)

Scriviamo in pseudocodice l'algoritmo per la visita anticipata. Quelli per le altre due visite sono del tutto analoghe, e si limitano ad invertire l'ordine delle righe di codice:

[8] A lezione si è parlato di "Previsit", "Invisit" e "Postvisit" invece di "Preorder traversal" etc... Tuttavia, cercando su internet non ho trovato nessun risultato relativo a binary tree previsit, e ovunque cercassi si parlava di "Preorder traversal" etc... Nel dubbio, riporto la notazione supportata dall'internet.

```
Anticipata(u)
{
    if(u != NULL)
    {
        print(u.dato);
        Anticipata(u.sx);
        Anticipata(u.dx);
    }
}
```

In modo analogo alla complessità di Dimensione(u), si dimostra che tutte e tre le visite hanno complessità $\Theta(n)$.
In generale, data solo una stringa relativa ad una sola visita, non è possibile ricostruire l'albero di partenza (ad esempio è molto facile esibire alberi binari diversi con la stessa stringa di visita anticipata, come i seguenti due che hanno entrambi come stringa di visita anticipata "A B C")



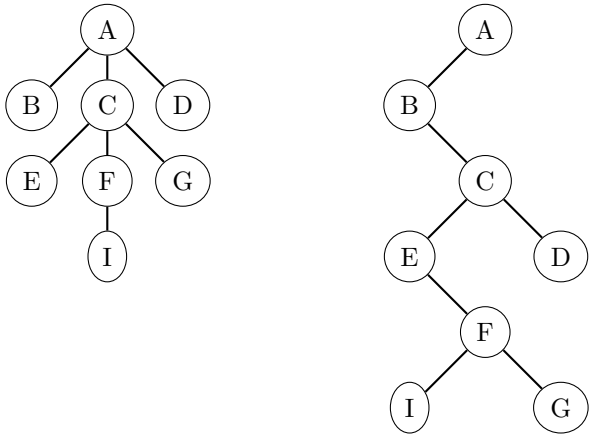
Tuttavia, conoscendo almeno la visita simmetrica e un'altra visita è possibile ricostruire l'albero[9]

12 07/04/2020

12.1 Trasformare alberi ordinati in alberi binari

Esiste una corrispondenza biunivoca tra gli alberi ordinati e gli alberi binari. In particolare, esiste un algoritmo che associa ad ogni albero ordinato un univoco albero binario, e viceversa. Il corrispondente binario di un albero ordinato T si chiama immagine binaria di T.
L'algoritmo è il seguente: sia T' l'immagine binaria di T. Si fissa come radice di T' la radice di T. Poi, per ogni nodo n' in T' si fissa come figlio sinistro il primo figlio del nodo n (in T), e ogni figlio i-esimo di n diventa figlio destro (in T') dell'(i-1)-esimo figlio di n (in T). Ad esempio, l'albero binario a destra è l'immagine binaria dell'albero ordinato a sinistra

[9] L'algoritmo non si è visto a lezione. Se volete lo trovate qua: https://www.geeksforgeeks.org/if-you-are-given-two-traversal-sequences-can-you-construct-the-binary-tree/



Osserviamo che vale che le dimensioni di un albero e della sua immagine binaria sono uguali. Tuttavia, ciò non vale per l'altezza.
Un'altra proprietà che viene conservata è la visita anticipata degli alberi. Infatti, per entrambi gli alberi la visita anticipata è "A B C E F I G D".
La visita posticipata non viene conservata. Osserviamo che per l'albero ordinato non è ben definita la visita simmetrica, poiché non è fisso il numero di figli per nodo.
Vale però che la visita posticipata dell'albero ordinato corrisponde alla visita simmetrica dell'albero binario (in questo caso entrambe "B E I F G C D A").

12.2 Alberi Binari di Ricerca (ABR)

Gli alberi binari si prestano molto a rendere efficienti quelle che vengono definite le "operazioni di dizionario". Con operazioni di dizionario si intendono:

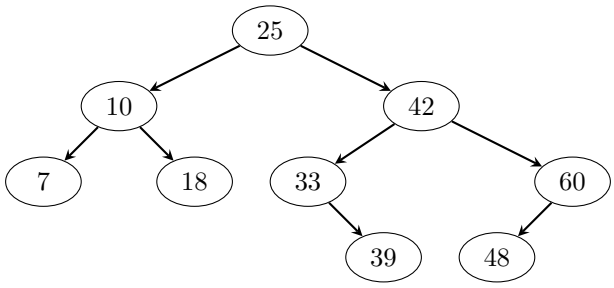
- Ricerca(k)
- Inserzione(k)
- Cancellazione(k)

applicate su una struttura dati generica contenente elementi distinti dalle chiavi k.
Se si vogliono usare le operazioni di dizionario su un albero binario, può tornare utile strutturare questo albero in un modo particolare per velocizzare le operazioni. Introduciamo quindi gli alberi binari di ricerca.

Si dice un albero binario di ricerca un albero binario tale che ogni nodo abbia la seguente proprietà:

- Le chiavi degli elementi del sottoalbero sinistro del nodo sono tutte minori della chiave del nodo
- Le chiavi degli elementi del sottoalbero destro del nodo sono tutte maggiori della chiave del nodo

Il motivo per cui si può preferire un albero di ricerca binaria ad un array ordinato (implementato con la ricerca binaria) è che la struttura fissa dell'array non è molto incline alle sequenze di inserzioni.
Un esempio di albero binario di ricerca è il seguente:



Scriviamo ora le operazioni di dizionario su un albero binario in pseudocodice:

```
RicercaABR(u, k)
{
    if(u == NULL) return NULL;
    else
    {
        if(u.dato == k) return u;
        else if(u.dato > k) return RicercaABR(u.sx, k);
        else return RicercaABR(u.dx, k);
    }
}
```

Si può facilmente dimostrare che la complessità di ricerca è $O(h)$, dove h è l'altezza dell'albero.

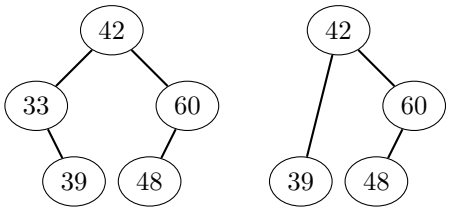
```
InserimentoABR(u, k)
{
    if(u == NULL)
    {
        u = nuovonodo(u);
        u.sx = NULL;
        u.dx = NULL;
        u.dato = k;
    }
    else if(k < dato)
        u.sx = Inserisci(u.sx, k);
    else
        u.dx = Inserisci(u.dx, k);

    return u;
}
```

Anche la complessità di inserimento è $O(h)$.

Per l'algoritmo di cancellazione, invece di esibire uno pseudocodice, elencheremo i passi dell'algoritmo. Supponiamo di voler cancellare l'elemento k. Allora:

- Se k è foglia, basta eliminarlo e mettere NULL al puntatore (dx o sx) del padre.
- Se k è un elemento con un puntatore NULL, si fa un "bypass". Ovvero si prende il puntatore del padre di k che punta a k e lo si sovrascrive con il puntatore di k non NULL. Visivamente, un bypass significa passare dall'albero a sinistra all'albero a destra (volendo cancellare k = 33):



- Se k ha entrambi i puntatori diversi da NULL, va sostituito con uno degli unici due nodi che possono sostituirlo: il precedente e il successivo (in ordine di chiavi).
  - Successivo: è il minimo del sottoalbero destro. Ha necessariamente il puntatore sinistro uguale a NULL (altrimenti si avrebbe nello stesso sottoalbero destro un elemento minore di lui), quindi può essere facilmente eliminato (al più con un bypass) e sovrascritto al posto di k. Un albero così rispetta le richieste di un albero binario di ricerca: infatti, detto m il valore del minimo del sottoalbero destro, vale k < m poiché m sta nel sottoalbero destro, e per ogni nodo del sottoalbero destro n vale che n.dato > m perché è il minimo. Allora scrivendo m al posto di k, avremo che tutto il sottoalbero destro ha nodi maggiori di m, e tutto il sottoalbero sinistro ha nodi n con n.dato < k < m, quindi minori di m. Vengono quindi rispettate le richieste per un albero binario di ricerca.
  - Precedente: è il massimo del sottoalbero sinistro. Si agisce in modo analogo e speculare al caso del successivo.

Anche la complessità di cancellazione è $O(h)$.

Altre operazioni facili da compiere su un albero ABR sono le seguenti:

- Ricerca di minimo o massimo. Per trovare il minimo basta "andare sempre a sinistra", ovvero percorrere l'albero nodo per nodo, partendo dalla radice, prendendo sempre i puntatori sx finché non si trova un nodo con

puntatore sx NULL. Il costo è $O(h)$. Per il massimo, la procedura è analoga ma "andando a destra".

- Ordinamento, ovvero restituire la lista ordinata dei nodi dell'albero. Questo, per un ABR, equivale alla visita simmetrica. Il costo è $O(n)$.
- Ricerca del precedente e del successivo di k (in ordine di chiavi). Consideriamo il caso del successivo (il precedente risulterà del tutto analogo). Detto u il nodo che contiene la chiave k, consideriamo i due caso:
  - se u.dx != NULL, allora mi basta considerare il minimo del sottoalbero destro.
  - se u.dx == NULL, allora l'elemento successivo (in ordine di chiavi) a k è, nel percorso che abbiamo fatto per raggiungere u dalla radice, l'ultimo elemento per cui siamo scesi a sinistra.

  Anche in questo caso la complessità è $O(h)$.

Abbiamo visto che molti degli algoritmi basati sugli alberi ABR hanno come complessità $O(h)$, dove h è l'altezza dell'albero. Questo però non ci dice direttamente a quanto corrisponde la complessità dell'algoritmo in funzione del numero di nodi n.
Sappiamo che nel caso di un albero bilanciato, $h \approx log(n)$, il che renderebbe gli algoritmi di complessità logaritmica, che sarebbe un risultato molto buono.
Tuttavia, non ci sono garanzie che un albero binario di ricerca sia bilanciato. Nel caso pessimo, infatti, un albero binario di ricerca può degenerare in una lista (immaginando ad esempio di inserire i dati in ordine crescente), risultando così in $h = n$ e trasformando gli algoritmi in complessità lineare, decisamente peggiore di una complessità logaritmica. L'unica considerazione che possiamo fare sull'altezza di un albero ABR è $log(n) \leq h \leq n$ (possiamo dire, però che almeno al caso medio l'altezza è $O(log(n))$).

Ci piacerebbe poter disporre di una struttura simile all'ABR che cerca però di rimanere il più bilanciata possibile, per poter sfruttare al meglio la complessità logaritmica degli algoritmi visti. Nella prossima lezione introdurremo un tipo di alberi binari di ricerca che risolve esattamente questo problema.

13 09/04/2020

13.1 Alberi binari di ricerca bilanciati in altezza

Abbiamo visto che, per quanto riguarda l'altezza h di un albero in funzione del numero dei nodi n, vale

$$log(n) \leq h \leq n$$

Dove si ottiene altezza logaritmica nel caso di un albero completamente bilanciato, e altezza lineare nel caso di un albero degenere.

Una strategia per ovviare al problema potrebbe essere semplicemente di implementare un normale albero ABR con l'accortezza di ribilanciare l'albero ogni volta che chiamiamo un'inserimento o una cancellazione. Così facendo, però, otterremmo degli algoritmi di inserimento e cancellazione di complessità $O(n)$, perdendo i vantaggi che ottenevamo con un albero ABR.
La soluzione è quella di tenere l'albero "abbastanza" bilanciato, dove questo "abbastanza" verrà definito con più criterio in seguito.

Alcuni tipi di struttura dati che riescono a mantenere il bilanciamento sono i seguenti:

- Alberi 2-3 (2-3 tree), che sono un caso particolare di B-alberi (B-tree).
- Alberi rossoneri (Red-black tree)
- Alberi AVL (Adelson-Velsky[10]- Landis[11])

In questo corso tratteremo solo quest'ultimo tipo di alberi binari di ricerca bilanciati.

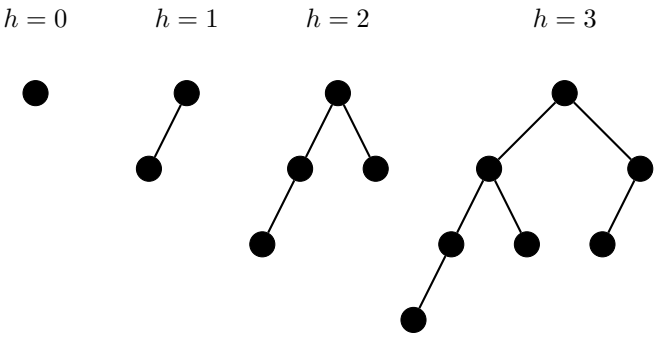
13.2 Alberi AVL

Per ogni nodo, indichiamo con h_s l'altezza del suo sottoalbero sinistro, e con h_d l'altezza del suo sottoalbero destro.
Un albero AVL è un albero binario di ricerca tale che per ogni nodo vale

$$|h_s - h_d| \leq 1$$

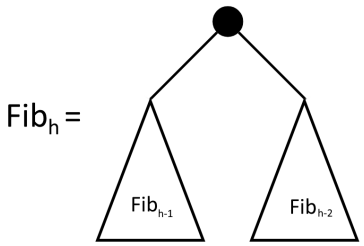
Mantenere questo tipo di bilanciamento è molto più facile di un bilanciamento completo. Tuttavia, questo tipo di bilanciamento garantisce un'altezza $h = O(log(n))$ e buone prestazioni.

Dimostriamo che vale $h = O(log(n))$.
Vediamo quanto possono essere "sbilanciati" gli alberi AVL. Ovvero, fissata un'altezza h, vediamo qual è l'albero AVL di altezza h il più sbilanciato possibile.



[10] Georgy Maximovich Adelson-Velsky: 1922-2014, fu un matematico e informatico Sovietico e Israeliano
[11] Evgenii Mikhailovich Landis: 1921-1997, fu un matematico Sovietico

Osserviamo che più in generale, l'h-esimo albero più sbilanciato può essere definito ricorsivamente nel seguente modo:



Questi alberi rappresentano il caso pessimo, poiché raggiungiamo una data altezza con il numero minimo di nodi. Prendono il nome di alberi di Fibonacci, per via della struttura ricorsiva simile a quella della successione di Fibonacci.

Sia n_h = numero di nodi in Fib_h. Osserviamo che vale la formula ricorsiva $n_h = n_{h-1}+n_{h-2}+1$. Confrontiamo ora n_h con F_h (la successione di Fibonacci). Vale

h	0	1	2	3	4	5	6	7
n_h	1	2	4	7	12	20	33	54
F_h	0	1	1	2	3	5	8	13

Si dimostra per induzione che vale la seguente formula: $n_h = F_{h+3}-1$. Sappiamo poi che per la successione di Fibonacci vale:

$$F_h = \frac{\phi^h - (1-\phi)^h}{\sqrt{5}}, \text{ con } \phi = \frac{1+\sqrt{5}}{2}$$

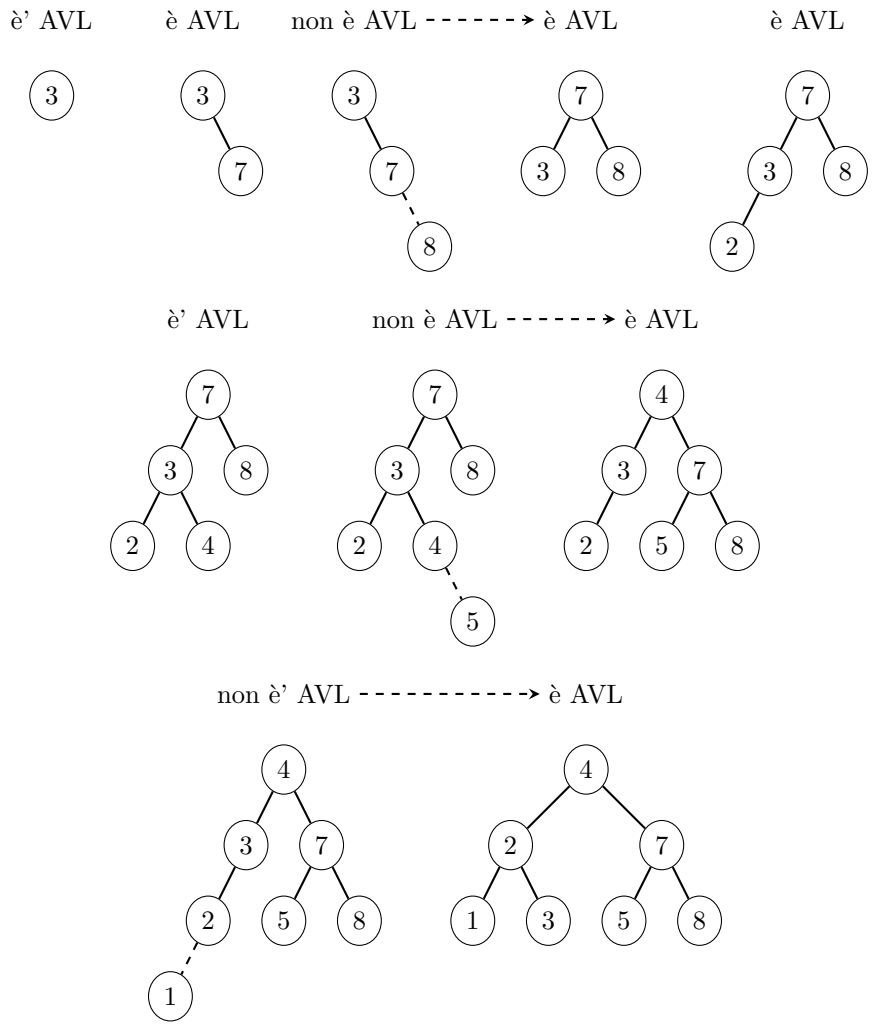
Si può dimostrare allora che $\exists c > 1$ tale che $\forall h$ vale $n_h = F_{h+3} - 1 \geq c^h$. Preso quindi un generico albero AVL di altezza h e detto n il suo numero di nodi, vale $n \geq n_h \geq c^h \Rightarrow n \geq c^h \Rightarrow h \leq log_c n \Rightarrow h = O(log(n))$.
Volendo fare i conti si otterrebbe $h \leq 1.4 \cdot log_2 n + \widetilde{c}$, con $\widetilde{c} \in \mathbb{R}$.

Quindi tutte le operazioni che per gli alberi binari di ricerca costano $O(h)$ si realizzano, sugli alberi AVL, in $O(log(n))$.

13.3 Mantenere il bilanciamento AVL dopo inserimento

E' possibile che, preso un albero AVL, subito dopo aver inserito un nodo (con lo stesso criterio con cui lo inseriamo in un albero ABR), l'albero sia sbilanciato, ovvero non rispetti più la proprietà degli alberi AVL. In questo caso, abbiamo bisogno di un algoritmo per poter ribilanciare l'albero nel modo più efficiente possibile. Facciamo un esempio pratico.

Partiamo da un albero AVL vuoto (che è bilanciato in modo banale) e diamogli questa sequenza di inserimenti: 3, 7, 8, 4, 2, 5, 1. Costruiamo l'albero AVL:

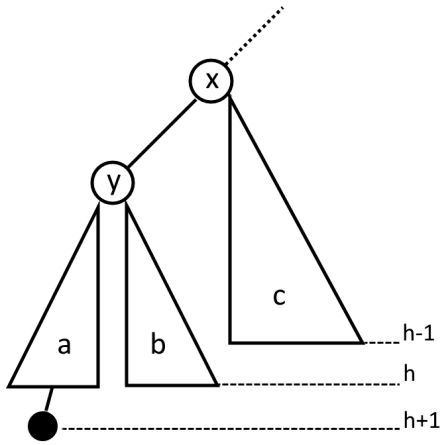


Gli algoritmi che abbiamo appena utilizzato per rendere AVL gli alberi sbilanciati si chiamano Rotazione DD (il primo), Rotazione SD (il secondo) e Rotazione SS (il terzo) (ed esiste come ultimo tipo di rotazione, la Rotazione DS). Il nodo che viene sbilanciato viene detto nodo critico (che nell'ordine erano i nodi 3, 7 e di nuovo 3). E' possibile che, con un inserimento, vengano in realtà sbilanciati più di un nodo: in questo caso il nodo critico attorno al quale avviene la rotazione è quello a profondità maggiore. Questo risolve anche le criticità superiori. Si osservi che dopo una rotazione (nel caso di sbilanciamento dovuto ad inserimento) l'albero riprende l'altezza precedente allo sbilanciamento, e gli effetti dell'inserimento e sbilanciamento non si avvertono al di sopra del nodo critico.

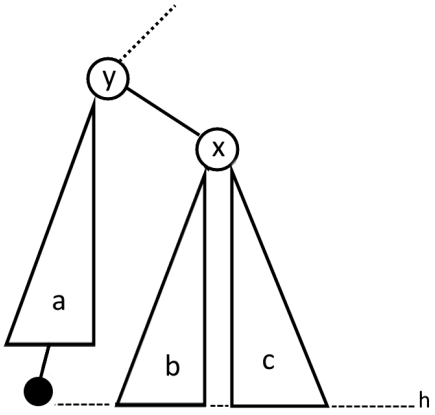
13.4 Rotazioni dopo inserimento

Rotazione semplice SS (e DD)

Tratteremo il caso di Rotazione SS. La Rotazione DD è uguale e speculare. Supponiamo che subito dopo un inserimento, l'albero sia sbilanciato (con x nodo critico) nel seguente modo: l'altezza del sottoalbero sinistro di x è di 2 maggiore all'altezza del suo sottoalbero destro, e (detto y il figlio sinistro di x) il sottoalbero sinistro di y ha altezza maggiore del sottoalbero destro di y.



Il modo in cui si svolge la rotazione è decisamente più facile da mostrare in figura che da spiegare a parole. Il risultato finale della Rotazione SS è il seguente:

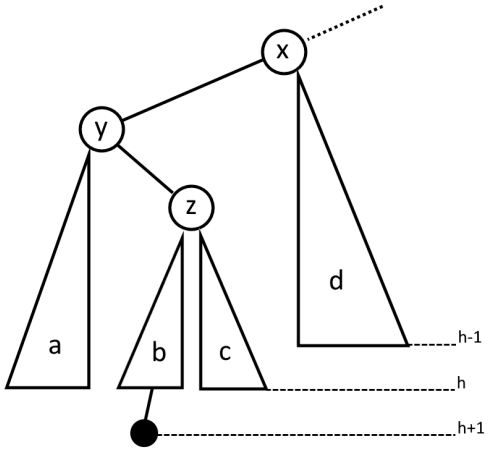


Osserviamo che poiché stiamo solo cambiando i valori di un numero fissato di puntatori, la complessità di questo algoritmo è $\Theta(1)$.

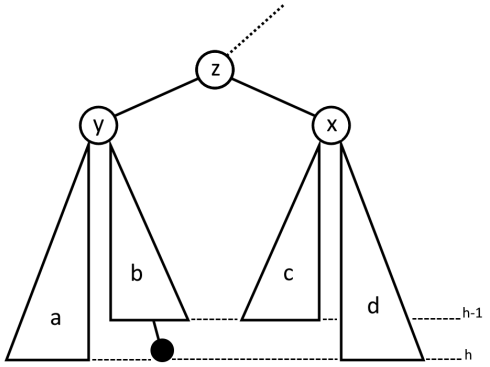
Leggermente più complicata è, invece, la Rotazione SD.

Rotazione doppia SD (e DS)

Allo stesso modo, tratteremo solo la Rotazione SD, poichè la DS è uguale e speculare.
A differenza di prima, supponiamo che l'albero sia sbilanciato in modo che il sottoalbero di y con altezza maggiore sia quello destro. Osserviamo che, anche se nel seguente diagramma (detto z il figlio destro di y) il sottoalbero di z di altezza maggiore è quello sinistro, l'algoritmo si applica uguale identico anche nel caso in cui il sottoalbero di altezza maggiore di z sia quello destro.
Graficamente la situazione è la seguente:



Come prima, un diagramma vale più di mille parole per spiegare questo algoritmo. Lo stato finale raggiunto dalla Rotazione SD è il seguente:



Anche in questo caso nella configurazione finale, l'altezza totale dell'albero è uguale a quella precedente all'inserimento, e il costo totale è $\Theta(1)$.

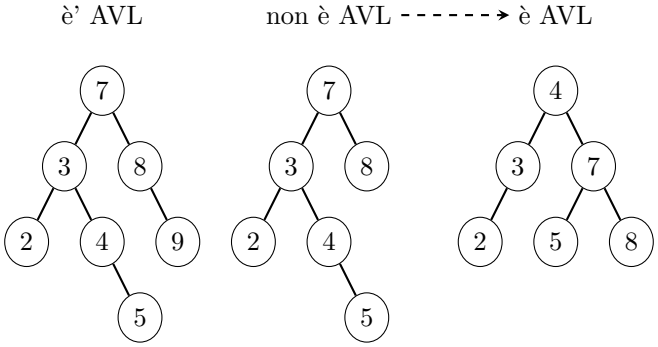
Abbiamo visto che sistemare l'albero con delle rotazioni conoscendo il nodo critico è un'operazione (a livello di complessità in tempo) facile. La domanda spontanea da porsi è: come si fa a determinare il nodo critico? I metodi possibili sono due:

- Memorizzare o calcolare l'altezza dei sottoalberi per ogni nodo
- Assegnare ad ogni nodo un "fattore di bilancia", che valga:
  - un valore negativo se il sottoalbero sinistro ha altezza maggiore del sottoalbero destro
  - 0 se le altezze dei due sottoalberi sono uguali
  - un valore positivo se il sottoalbero destro ha altezza maggiore del sottoalbero sinistro

  Durante il percorso di inserimento possiamo memorizzare l'ultimo nodo che potrebbe essere sbilanciato dopo l'inserimento e aggiustare poi i fattori di bilancia. Questo procedimento può essere svolto con un costo di $O(log(n))$.

13.5 Rotazioni dopo cancellazione

I tipi di rotazioni da svolgere dopo la cancellazione di un nodo sono gli stessi da svolgere dopo un inserimento. Ad esempio (volendo cancellare il nodo 9)



C'è però un'importante differenza: poiché è possibile che dopo la cancellazione l'altezza del sottoalbero di cui il nodo critico è radice diminuisca di uno, è possibile che dopo il ribilanciamento si sbilanci il padre del nodo critico. Non basta necessariamente, come per l'inserimento, una sola rotazione per ribilanciare l'albero, ma è necessario un numero di rotazioni pari a $O(log(n))$.

Vediamo quindi ora i costi totali di inserimento e cancellazione negli alberi AVL. Poiché, come abbiamo mostrato, le rotazioni costano $\Theta(1)$, i costi delle operazioni sono:

- Inserimento: $O(log(n))$ + 1 rotazione = $O(log(n))$
- Cancellazione: $O(log(n)) + O(log(n))$ cancellazioni = $O(log(n))$

14 21/04/2020

14.1 Dizionari

Le strutture dati su cui si compiono le operazioni di dizionario che abbiamo visto in precedenza si chiamano, come uno potrebbe aspettarsi, "dizionari". Fino ad adesso abbiamo visto, come semplificazione, dizionari i cui elementi erano solo chiavi senza valori associati (ad esempio negli alberi binari, i nodi contenevano solo il valore secondo il quale veniva ordinato). In generale, però, un dizionario contiene elementi coppia composti da una chiave e da un dato (e si assume che tutte le chiavi siano a due a due distinte).
Come abbiamo visto, le operazioni principali che si svolgono su un dizionario sono:

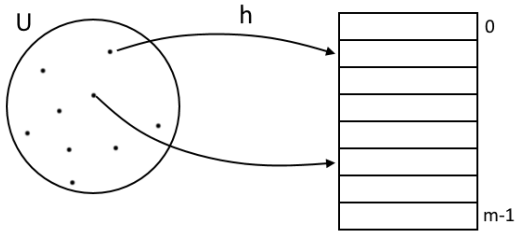
- Ricerca(k)
- Inserimento(e)
- Cancellazione(e)

dove e = {k, v} è un elemento generico del dizionario, composto da una chiave k e un dato v.
Per semplicità di rappresentazione, spesso vedremo ancora gli elementi di un dizionario come solo la loro chiave (come abbiamo fatto fin'ora).
Vediamo ora altri modi per strutturare un dizionario.

14.2 Tabelle Hash

Abbiamo visto che il limite inferiore sulla ricerca per confronti è di $\Omega(log(n))$ confronti, cioè è impossibile sviluppare una ricerca basata sui confronti più efficiente di così.
Proviamo qundi a sviulppare un algoritmo di ricerca non basato sui confronti.

L'idea che utilizzeremo è quella di supporre di avere un insieme U di tutte le chiavi possibili ("universo delle chiavi"), un array in memoria, e una funzione (detta hash) che prende in input una chiave e restituisce un indice dell'array. Allora, per inserire nella struttura dati un elemento di chiave k, lo salveremo in posizione $h(k)$ e volendo cercare un elemento di chiave k basterà prendere l'elemento in posizione $h(k)$. Così facendo, l'unica complessità di ricerca e inserimento sarà dovuta alla complessità della funzione hash (che può essere (quasi) piccola a piacere e non è limitata inferiormente da $log(n)$).



Si suppone che nell'universo delle chiavi, tutte le chiavi siano equiprobabili.
Può accadere, però, che prese due chiavi $k_i \neq k_j$, valga $h(k_i) = h(k_j)$. In questo caso si parla di "collisione".
Per creare una tabella hash, quindi, dobbiamo risolvere alcuni problemi:

- determinare a priori la dimensione m della tabella da utilizzare
- scegliere una funzione hash adeguata
- determinare una strategia per risolvere le collisioni

La dimensione m della tabella è strettamente legata alla funzione hash utilizzata.

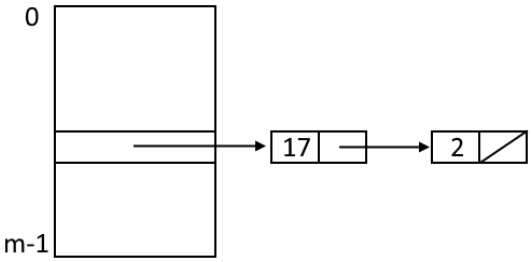
Osservazione: le chiavi possono essere sia numeriche che alfanumeriche. Nel caso di chiavi alfanumeriche ci si può ricondurre a chiavi numeriche considerando la loro codifica ASCII.
Le tabelle hash vengono comunemente usate, ad esempio, dai compilatori per gestire la tabella dei simboli (cioè i nomi delle variabili).
Una famosa funzione hash è la funzione SHA1, utilizzata per tabelle hash, sistemi crittografici, per calcolare fingerprint e per i sistemi peer to peer.
Vediamo ora alcuni metodi per creare una funzione di hash:

- Fissata la dimensione della tabella m, $h(k) = k \bmod m$
- Fissato $m = 2^r$, possiamo prendere $h(k) =$ un valore a r bit in funzione di k. Per cercare di avere valori di $h(k)$ il più uniformemente distribuiti possibili possiamo prendere come $h(k)$ gli r bit centrali di k scritto in base 2. Un altra tecnica che si può utilizzare per cercare di scorrelare il più possibile le chiavi dai valori della funzione hash è quella di dividere k in n blocchi da r bit ciascuno (detti $r_1, ..., r_n$) e scrivere $h(k) = (...((r_1 \text{ XOR } r_2) \text{ XOR } r_3)...) \text{ XOR } r_n$.

Idealmente ci piacerebbe avere che $k_i \neq k_j \Rightarrow h(k_i) \neq h(k_j)$. Nella realtà ci dobbiamo però aspettare, avendo dati casuali, di avere delle collisioni. Vediamo ora alcuni metodi per la risoluzione di collisioni.

14.3 Chaining (liste di trabocco, liste di concatenazione)

Nel chaining, ogni cella di memoria della tabella contiene una lista (o meglio, un puntatore a lista). Così facendo, in caso di collisione si possono mettere gli elementi con chiavi che collidono nella stessa lista.



Per studiare meglio il funzionamento del chaining, definiamo i seguenti parametri (che torneranno utili anche per i prossimi metodi):

- n = numero totale di elementi salvati
- m = dimensione della tabella (numero di celle della tabella)
- $\alpha = \frac{n}{m}$, detto "fattore di carico" (nel caso del chaining, corrisponde a quanti elementi in media sono salvati per lista)

Definiamo ora un algoritmo di ricerca su una tabella hash di questo tipo.
Ricerca(k):

1. calcolo di $h(k)$
2. accesso alla lista che ha come puntatore la cella di indice $h(k)$
3. ricerca nella lista

Per calcolare la complessità di questo algoritmo dobbiamo studiare il suo caso pessimo. Il caso pessimo è quello in cui tutti e gli n elementi salvati vengono salvati in una sola lista. La complessità è quindi $O(n)$ (c'è da dire che in generale le tabelle hash funzionano male nel caso pessimo).
Per quanto riguarda il caso medio, invece, bisogna considerare la lunghezza media della generica lista: abbiamo detto che la lunghezza media corrisponde a $\alpha = \frac{n}{m}$, quindi la complessità al caso medio corrisponde a $\Theta(1+\alpha)$, che diventa costante per opportuni valori di α.
Definiamo ora gli algoritmi di inserimento e cancellazione per il chaining.
Inserimento(k):

1. calcolo di $h(k)$
2. inserimento di k in fondo alla lista di puntatore $h(k)$ controllando che k non sia già presente (oppure, per mantenere il costo non proporzionale ad α, se già sappiamo che k non è presente possiamo aggiungerlo in testa alla lista)

Cancellazione(k):

1. calcolo di $h(k)$
2. cancellazione nella lista

In entrambi i casi, il caso pessimo ha costo $O(n)$ e come caso medio $\Theta(1+\alpha)$, che è costante per opportuni valori di α.

14.4 Open Hash con scansione lineare

Nell'Open Hash tutti gli elementi vengono memorizzati nella tabella (a differenza delle liste di poco fa). Data una chiave, per decidere in quale cella bisognerà mettere l'elemento, si stabiliscono delle "prove" per dove collocare l'elemento. Ovvero, con una funzione si decide una successione di indici $h(k, 0)$, $h(k, 1)$, ..., $h(k, m-1)$ e si posiziona l'elemento nel primo di questi indici che corrisponde ad una cella non ancora occupata. Due esempi di funzioni per l'Open Hash sono le seguenti:

- $h(k, i) = (h(k) + i) \bmod m$, detta scansione lineare di passo 1
- $h(k, i) = (h(k) + qi) \bmod m$ (con MCD(q, m) = 1), detta scansione lineare di passo q

Vediamo ora un esempio pratico: supponiamo di voler salvare in una tabella hash un insieme di vegetali. Consideriamo una tabella di dimensione $m = 21$ e una funzione hash $h(k)$ che assegna a k (nome del vegetale) la posizione nell'alfabeto della sua iniziale. Teniamo a mente che come funzione hash la $h(k)$ che stiamo considerando è pessima, ma torna comodo per fare esempi semplici e concentrarsi sulla gestione dei conflitti. Diamo ora una lista di inserimenti, indicando con una freccia il passaggio alla prova successiva qualora l'indice calcolato fosse già occupato. Per semplicità scriviamo $h(k)$ al posto di $h(k, 0)$.

1. h(Zucca) = 20, ok.
2. h(Banana) = 1, ok.
3. h(Baobab) = 1 $\rightarrow$ h(Bobab, 1) = 2, ok.
4. h(Pomodoro) = 13, ok.
5. h(Palma) = 13 $\rightarrow$ h(Palma, 1) = 14, ok.
6. h(Peperone) = 13 $\rightarrow$ h(Peperone, 1) = 14 $\rightarrow$ h(Peperone, 2) = 15, ok.
7. h(Zibibbo) = 20 $\rightarrow$ h(Zibibbo, 1) = 0, ok.
8. h(Zenzero) = 20 $\rightarrow$ ... $\rightarrow$ h(Zenzero, 4) = 3, ok.

La tabella finale ottenuta è la seguente

0	Zibibbo
1	Banana
2	Baobab
3	Zenzero
...	
13	Pomodoro
14	Palma
15	Peperone
...	
20	Zucca

La scansione lineare porta però con sé alcuni problemi. Vediamoli esaminando gli algoritmi di ricerca, inserimento e cancellazione.

Per la ricerca di k basterà seguire la successione delle prove finché non lo troviamo. Inoltre, se ad un certo punto troviamo una cella vuota, sappiamo che la ricerca è fallita e possiamo fermarci. Fin qui nessun problema.

Per l'inserimento, si segue la successione delle prove di k e lo colloco alla prima cella vuota. Il problema di questo metodo è che favorisce la creazione di aglomerati sempre più grandi, che aumentano la complessità di inserimento. Per capire meglio perché tendono a crearsi aglomerati, mettiamo a confronto le probabilità che una chiave finisca in una cella isolata o che finisca subito dopo un aglomerato, espandendolo.

- se i è una cella isolata, vale $\mathbb{P}[\text{chiave } k \text{ vada in } i] = \mathbb{P}[h(k) = i] = \frac{1}{m}$ (supponendo che le chiavi siano equidistribuite).
- se i è una cella che segue un aglomerato (che comincia nella cella $j < i$), vale $\mathbb{P}[\text{chiave } k \text{ vada in } i] = \mathbb{P}[h(k) = j] + \mathbb{P}[h(k) = j + 1] + ... + \mathbb{P}[h(k) = i] = \frac{i-j+1}{m}$, poiché in uno qualsiasi dei casi $h(k) = j$, ..., $h(k) = i$, la prima cella libera trovata sarebbe la i.

La cancellazione porta ulteriori problemi. Cancellando un elemento isolato, non accade nulla di male poiché stiamo solo liberando uno spazio. Cancellando invece un elemento a metà di un aglomerato stiamo effettivamente spezzando quell'aglomerato.
In questo caso, supponendo di star cancellando l'elemento nella cella i, se avessimo salvato un elemento di chiave k tale che $h(k) < i$ ma k fosse stato salvato in una posizione $j > i$ poiché tutte quelle prima erano occupate, cancellare l'elemento i renderebbe introvabile l'elemento di chiave k poiché prima di arrivare alla cella j troveremmo la cella i vuota, che era la nostra condizione di arresto.

Per ovviare a questo problema potremmo introdurre una variabile di "marcatura", ovvero una variabile che verrà utilizzata per segnare le celle cancellate come non veramente "vuote". Modificando leggermente gli algoritmi di ricerca

e inserimento rispetto a come li abbiamo definiti, riusciamo ad ottenere una cancellazione che non rende alcuni elementi introvabili (dicendo alla ricerca di controllare anche oltre le celle vuote ma marcate). Così facendo, però, un grande numero di cancellazioni porterebbe ad un grande numero di celle vuote ma marcate, che peggiora molto la complessità di inserimento e ricerca, rendendo il loro tempo medio $\frac{1}{1-\alpha}$, per cui bisogna cercare di tenere α abbastanza piccolo (ad esempio sotto il 90%).

Un altro modo di risolvere il problema della cancellatura, senza passare per la marcatura, è quello di "risistemare" tutti i valori spostandoli nelle posizioni giuste "aggiornate". Ovvero:

1. cancella l'elemento
2. considera tutti gli elementi che lo seguono nell'aglomerato e spostali nella cella vuota se il loro indirizzo hash è minore o uguale a quello della cella vuota (ricorsivamente)

Formalizziamo questo algoritmo. Distinguiamo la cancellazione all'interno di un aglomerato in due casi:

- Caso 1: l'eliminazione avviene in un blocco "al centro" della tabella, ovvero non a cavallo tra la fine e l'inizio. L'algoritmo, allora, si svolge così:
  1. rimuove la cella in posizione p, rendendo p vuota
  2. esamina gli elementi che seguono nell'aglomerato, ricalcolando gli hash
  3. se trova un elemento la cui posizione dovrebbe essere q $\leq$ p, sposta questo elemento nella cella p.
  4. la nuova cella libera è quella da cui abbiamo spostato quest'ultimo elemento. Ripeti da (2)
- Caso 2: l'eliminazione avviene in un blocco a cavallo degli estremi. Concettualmente, il modo in cui l'algoritmo si svolge è lo stesso, ma bisogna gestire bene il modo in cui si confrontano gli indici q e p nel caso in cui si trovino nella parte di aglomerato adiacente all'inizio della tabella o nella parte adiacente alla fine.
  Più precisamente, immaginiamo che i sia l'indice che utilizziamo per scorrere gli elementi che seguono nell'aglomerato, e supponiamo di aver scoperto che in posizione i, c'è un elemento di hash q che va riposizionato.
  Prima, per riposizionare, controllavamo che valesse che q $\leq$ p, nel qual caso avremmo spostato l'elemento di posizione i in posizione p.
  In questo caso, questo controllo non basta. Infatti, è possibile che l'indice i scorra fino ad arrivare oltre la fine della tabella e tornando all'inizio. In questo caso, è possibile che ci sia un elemento in posizione i il cui indirizzo hash sia q che viene dopo p nell'aglomerato, ma tale che q $\leq$ p poiché gli

elementi nella parte di aglomerato adiacente all'inizio della tabella hanno sempre indici minori rispetto a quelli nella parte adiacente alla fine.
Bisogna essere più precisi su come e quando bisogna risistemare elementi. In particolare, l'elemento in posizione i va risistemato se e solo se ci troviamo in uno di questi tre casi:

- q, p e i si trovano tutti nella stessa parte di aglomerato. Perché ciò accada, deve valere $q \leq p < i$. Osserviamo che tenere conto di questo caso risolve anche il caso 1 visto prima (poiché in quel caso, visto che l'aglomerato non va mai oltre la fine, vale sempre $p < i$. Nell'implementazione dell'algoritmo, quindi, potremo semplicemente implementare il check ($q \leq p < i$) per tenere conto sia del caso 1, sia di questo sottocaso del caso 2).
- i è andato oltre la fine della tabella, tornando all'inizio, mentre p e q sono entrambi nella parte di aglomerato adiacente alla fine. Perché accada ciò, deve valere $i < q \leq p$.
- i e p sono andati oltre la fine, q no. Perché accada ciò, deve valere $p < i < q$.

Vediamo ora lo pseudocodice:

```
HashCanc(A, k)
{
    p = h(k);
    while(A[p] != k && A[p] != NULL)
        p = (p+1)%m;

    if(A[p] == NULL)
        return NULL;    // L'elemento non era presente
    else
    {
        A[p] = NULL;
        i = (p+1)%m;
        while(A[i] != NULL)
        {
            q = h(A[i]);
            if((q <= p < i) OR (i < q <= p) OR (p < i < q)
            {
                A[p] = A[i];
                A[i] = NULL;
                p = i;
            }
            i = (i+1)%m;
        }
    }
}
```

15 23/04/2020

15.1 Numero medio di accessi nell'Open Hash

Nella scorsa lezione abbiamo detto che nell'open hahs il numero medio di accessi per ricerca e inserimento è $\frac{1}{1-\alpha}$. Dimostriamolo.
Mettiamo nelle ipotesi che la sequenza di inserimento sia casuale, ovvero che la sequenza di scansione sia una qualsiasi delle $m!$ permutazioni delle celle.

Osservazione: quest'ipotesi è falsa in generale nella scansione lineare, nella quale le sequenze di scansioni diverse sono m (poiché la sequenza con cui vedremo gli elementi dipende solo dal punto in cui partiamo, procedendo poi in modo lineare). Si verifica però sperimentalmente che il numero medio di accessi è molto vicino a $\frac{1}{1-\alpha}$.
Sia $T(n,m) =$ il numero di accessi per inserire la $n-$esima chiave.
Sicuramente vale $T(0,m) = 1$. Consideriamo ora il generico caso $n-$esimo:

- $\mathbb{P}$[la chiave $n-$esima trova la casella libera] $= \frac{m-n}{m}$, nel qual caso serve un solo accesso.
- $\mathbb{P}$[la chiave $n-$esima non trova la casella libera] $= \frac{n}{m}$, nel qual caso servono $1 + T(n-1, m-1)$ accessi.

Allora il valore di $T(n,m)$, ovvero il valore atteso (cioè medio) di numero di accessi necessari, è

$$T(n,m) = 1 \cdot \frac{m-n}{n} + (1 + T(n-1, m-1)) \cdot \frac{n}{m}$$

da cui

$$T(n,m) = \begin{cases} 1 & \text{se } n = 0 \\ 1 + \frac{n}{m} T(n-1, m-1) & \text{altrimenti} \end{cases}$$

Mostriamo ora per induzione su n che $T(n,m) \leq \frac{m}{m-n} = \frac{1}{1-\alpha}$.
Per il passo base $n = 0$ ok.
Per il passo induttivo, vale

$$\begin{aligned} T(n,m) &\leq 1 + \frac{n}{m} T(n-1, m-1) \leq \\ &\leq 1 + \frac{n}{m} \frac{m-1}{(m-1)-(n-1)} \leq \\ &\leq 1 + \frac{n}{m} \frac{m}{m-n} = \\ &= \frac{m}{m-n} = \\ &= \frac{1}{1-\alpha} \end{aligned}$$

Ad esempio, se abbiamo $\alpha = \frac{9}{11}$ (ovvero, se riusciamo a mantenere α costante indipendentemente da n), abbiamo che $T(n,m) \leq \frac{11}{2} = 5.5 = \Theta(1)$.

Vediamo ora qualche altro tipo di scansione per l'Open Hash.

15.2 Open Hash con scansione quadratica

Fondamentalmente, la scansione quadratica funziona allo stesso modo della scansione lineare. La differenza è che la lunghezza del salto di indici ad ogni prova aumenta in modo quadratico. In pratica vale

$$h(k,i) = (h(k) + ai^2 + bi + c) \bmod m$$

Vediamo un paio di esempi:

- L1: $h(k,i) = (h(k) + i^2) \bmod m$, con m primo
- L2: $h(k,i) = (h(k) + \frac{i^2}{2} + \frac{i}{2}) \bmod m$, con $m = 2^s$

Osserviamo che L2 è totale, ovvero la successione di prove prima o poi esamina tutte le celle della tabella[12]. L1, invece, non colpisce tutte le celle, poiché dopo $\frac{m+1}{2}$ tentativi si torna sulle stesse posizioni (infatti, se ad esempio $h(k) = 0$, andiamo a colpire solo le celle i cui indici sono dei quadrati mod m, e per Algebra sappiamo che i quadrati mod m sono $\frac{m+1}{2}$. Per $h(k) \neq 0$ sono semplicemente dei traslati dei quadrati, quindi sono ancora solo $\frac{m+1}{2}$).
Anche in questo caso, come nella scansione lineare, le sequenze di scansione possibili sono solo m, ma poiché il passo non è costante la scansione quadratica tende a sparpagliare di più le chiavi.

15.3 Doppio Hash

Nel doppio hash si svincola la legge di scansione dall'indirizzo hash, nel seguente modo:

$$h(k,i) = (h(k) + i(h'(k) + 1)) \bmod m$$

dove $h(k)$ e $h'(k)$ sono due funzioni hash distinte.
Vediamo un paio di esempi:

- L3: m primo, con $h(k) = k \bmod m$, $h'(k) = k \bmod m - 1$
- L4: $m = 2^s$, con $h(k) = s$ bit estratti da k, $h'(k) = 2(s-1)$ bit estratti da k

In questi casi si ottengono m^2 sequenze di scansione, a divverenza delle scansioni lineari.

Vediamo ora due esempi pratici di inserimento con L1 e L3.

[12] La dimostrazione di questo fatto non è stata fatta a lezione, ma per completezza verrà messa alla fine di questi appunti. Vi avviso: l'unica dimostrazione che sono riuscito a fare non è una bella dimostrazione

L1

Consideriamo $m = 11$, $h(k) = k \bmod m$ e questa sequenza di inserimento: 43, 22, 31, 22, 31, 4, 15, 28, 17, 86, 60.

- $h(43) = 10$, ok.
- $h(22) = 0$, ok.
- $h(31) = 9$, ok.
- $h(4) = 4$, ok.
- $h(15) = 4 \to h(15, 1) = 4 + 1^2 = 5$, ok.
- $h(28) = 6$, ok.
- $h(17) = 6 \to h(17, 1) = 6 + 1^2 = 7$, ok.
- $h(86) = 9 \to h(86, 1) = 9 + 1^2 = 10 \to h(86, 2) = 9 + 2^2 = 2$ (mod m, si intende), ok.
- $h(60) = 5 \to h(60, 1) = 5 + 1^2 = 6 \to h(60, 2) = 5 + 2^2 = 9 \to h(60, 3) = 5 + 3^2 = 3$, ok.

La tabella finale ottenuta è la seguente

22		86	60	4	15	28	17		31	43

L3

Consideriamo $m = 11$, e la stessa sequenza di inserimento: 43, 22, 31, 22, 31, 4, 15, 28, 17, 86, 60.

- $h(43) = 10$, ok.
- $h(22) = 0$, ok.
- $h(31) = 9$, ok.
- $h(4) = 4$, ok.
- $h(15) = 4 \to h(15, 1) = 4 + \cdot(5 + 1) = 10 \to h(15, 2) = 4 + 2 \cdot (5 + 1) = 5$ (sempre inteso mod m), ok.
- $h(28) = 6$, ok.
- $h(17) = 6 \to h(17, 1) = 6 + 1 \cdot (7 + 1) = 3$, ok.
- $h(86) = 9 \to h(86, 1) = 9 + 1 \cdot (6 + 1) = 5 \to h(86, 2) = 9 + 2 \cdot (6 + 1) = 1$, ok.
- $h(60) = 5 \to h(60, 1) = 5 + 1 \cdot (0 + 1) = 6 \to h(60, 2) = 5 + 2 \cdot (0 + 1) = 7$, ok.

La tabella finale ottenuta è la seguente

22	86		17	4	15	28	60		31	43

16 28/04/2020

16.1 Programmazione Dinamica

Con programmazione dinamica si intende un paradigma per la creazione di algoritmi simile al Divide et Impera, ovvero basato sulla suddivisione del problema in sottoproblemi più facili da risolvere. La differenza principale tra il Divide et Impera e la programmazione dinamica è che se nel Divide et Impera i sottoproblemi erano divisi "a compartimenti stagni", ovvero ogni sottoproblema era gestito, richiamato e utilizzato da un solo sovraproblema, nella programmazione dinamica i sottoproblemi possono essere condivisi tra più sovraproblemi.
In generale, un algoritmo di programmazione dinamica è definito in 4 passi:

1. Uguale caratterizzazione della struttura del problema generale e dei sottoproblemi
2. Definizione dei sottoproblemi elementari
3. Regola ricorsiva
4. Strategia di memorizzazione delle soluzioni parziali

La programmazione dinamica può essere applicata a problemi la cui soluzione ottima può essere ricavata dalle soluzioni ottime dei sottoproblemi.
Un esempio può essere calcolare l'n-esimo numero di Fibonacci. L'algoritmo più banale che può venire in mente è:

```
Fib(n)
{
    if(n <= 1) return n;
    else return Fib(n-1) + Fib(n-2);
}
```

Questo però non è un buon algoritmo, poiché moltissimi numeri vengono calcolati più volte del necessario. L'algoritmo che abbiamo appena esibito fornisce una soluzione a complessità esponenziale ad un problema che può essere risolto in modo lineare. Cerchiamo di migliorarlo.

```
Fib2(n)
{
    F = array(n);
    F[0] = 0;
    F[1] = 1;
    for(i = 2; i < n; i++)
    {
        F[i] = F[i-1] + F[i-2];
    }
    return F[n];
}
```

Questo algoritmo è ottimo in tempo (infatti è $\Theta(n)$), ma non in spazio. Lo spazio aggiuntivo utilizzato da questo algoritmo è $\Theta(n)$, ma non c'è nessun bisogno di salvare in memoria tutta la tabella. Miglioriamo ancora con il seguente algoritmo:

```
FibMigliore(n)
{
    if(n <= 1) return n;
    else
    {
        a = 1;
        b = 0;
        for(i = 2; i < n; i++)
        {
            c = a+b;
            b = a;
            a = c;
        }

        return a;
    }
}
```

Con la programmazione dinamica abbiamo risolto il problema in $\Theta(n)$ tempo e $\Theta(1)$ spazio.

16.2 Longest Common Subsequence (LCS)

Un altro esempio di problema che può essere risolto con la programmazione dinamica è il problema della Longest Common Subsequence.
Supponiamo di avere due sequenze (di caratteri, di interi, o in generale di variabili confrontabili) a e b di lunghezze rispettivamente m < n. Si chiede di trovare la sottosuccessione di caratteri (non necessariamente consecutivi) comune a entrambe le successioni. Ad esempio, date:
a: A D C A A B,
b: D A A B D C D A A C A C C B A,
la LCS (esibita in b ma comune ad entrambe) è
D (A) A B (D C) D (A A) C A C C (B) A.
Le LCS hanno un'applicazione importante sia in bioinformatica, dove vengono usate per analizzare sequenze di DNA, sia in alcuni ambiti nel campo della sicurezza informatica.
Proviamo a sviluppare un algoritmo in programmazione dinamica per risolvere questo problema.
(Anticipo che in questo algoritmo utilizzeremo array con indici che partono da 1 invece che da 0. So che va contro lo standard utilizzato fino ad adesso, ma in questo caso torna più comodo così).

1. Definiamo il problema generale come LCS(a[1... m], b[1... n]) e definiamo come sottoproblemi lo stesso problema chiesto su prefissi di a e b. La forma generale del sottoproblema è LCS(a[1... i], b[1... j]).

2. Come sottoproblemi elementari consideriamo il caso in cui una delle due sequenze sia vuota. LCS(∅, b[1... j]) = 0, LCS(a[1... i], ∅) = 0.

3. (Introduciamo un leggero abuso di notazione: diciamo LCS(i, j) = LCS(a[1... i], b[1... j]))
   Supponiamo di aver risolto LCS(i-1, j-1). Aggiungiamo alle sequenze anche a[i] e b[j]. A questo punto consideriamo:

   - se a[i] = b[j], allora LCS(i, j) = LCS(i-1, j-1) + 1.
   - se sono diversi, considero il massimo tra LCS(i-1, j) e LCS(i, j-1).

   Quindi

$$LCS(i,j) = \begin{cases} 0 & \text{se i} = 0, \text{ j} = 0 \\ LCS(i-1, j-1) + 1 & \text{se a[i]} = \text{b[j]} \\ max\{LCS(i-1, j), LCS(i, j-1)\} & \text{altrimenti} \end{cases}$$

4. Usiamo un array L[0... m, 0... n] per memorizzare le soluzioni parziali.

Facciamo un esempio pratico. Sia a: AMICA, b: MATEMATICA. Allora m = 5, n = 10. Partiamo con la nostra tabella L (m+1)x(n+1) vuota, e cominciamo a riempirla per righe. Nelle celle corrispondenti ad un caso elementare mettiamo uno 0. Per gli altri casi, affidiamoci alla formula ricorsiva definita prima. Supponiamo, come esempio per mostrare il procedimento, di aver riempito la tabella fino al caso i = 3, j = 8 (escluso):

	∅	M	A	T	E	M	A	T	I	C	A
∅	0	0	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1	1	1
M	0	1	1	1	1	2	2	2	2	2	2
I	0	1	1	1	1	2	2	2			
C											
A											

In questo caso, poiché vale che a[3] = b[8], in posizione (3, 8) scriveremo LCS(2, 7) + 1. Poiché in posizione (2, 7) avevamo salvato 2, scriveremo 3. Passando al caso successivo, ovvero il caso i = 3, j = 9, vale che a[3] ≠ b[9]. Allora in posizione (3, 9) scriveremo il massimo tra LCS(2, 9) e LCS(3, 8). In questo caso il massimo è LCS(3, 8) = 3, quindi in posizone (3, 9) scriveremo 3.
Procediamo allo stesso modo fino a completare tutta la tabella. La tabella completa è

	∅	M	A	T	E	M	A	T	I	C	A
∅	0	0	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1	1	1
M	0	1	1	1	1	2	2	2	2	2	2
I	0	1	1	1	1	2	2	2	3	3	3
C	0	1	1	1	1	2	2	2	3	4	4
A	0	1	2	2	2	2	3	3	3	4	5

Completata la tabella, in posizione (m, n) avremo la lunghezza della LCS cercata. Per ricostruire la sottosequenza cercata, una volta riempita tutta L, possiamo utilizzare il seguente algoritmo, scritto in pseudocodice:

```
EstraiLCS(L, m, n)
{
    i = m;
    j = n;
    k = L[m, n],
    w = nuovoarray(k);
    while((i > 0) AND (j > 0))
    {
        if(a[i] == b[j])
        {
            w[k] = a[i];
            k--;
            i--;
            j--;
        }
        else if(L[i, j] == L[i, j-1]) i--;
        else j--;
    }

    return w;
}
```

Spiegato a parole, prendiamo la tabella e partiamo dall'estremo in basso a destra. A questo punto ci spostiamo nella tabella, ripetendo questi passaggi fino ad arrivare o in cima o a sinistra:

1. Se ci troviamo in una posizione corrispondete a due lettere uguali, aggiungiamo questa lettera in testa alla sottosequenza cercata, e ci spostiamo nella tabella in diagonale in alto a sinistra.

2. Altrimenti, vuol dire che l'elemento della tabella che stiamo considerando è uguale o a quello subito sopra, o a quello a sinistra (in quanto uguale al massimo di questi due). Ci spostiamo allora sull'elemento uguale, senza aggiungere lettere alla sottosequenza che stiamo costruendo.

L'algoritmo che abbiamo esibito per trovare la LCS è $\Theta(n \cdot m)$ tempo. Per la complessità in spazio c'è un'osservazione ulteriore da fare: se l'obiettivo era

ottenere la sottosequenza, allora bisogna memorizzare tutta la tabella per poi poterla ripercorrere, e in questo caso lo spazio aggiuntivo è $\Theta(n \cdot m)$; se l'obiettivo invece era sapere solo la lunghezza della sottosequenza, non c'è davvero bisogno di salvare in memoria tutta la tabella, ma solo le ultime due righe (o ultime due colonne se procediamo a riempirla per colonne), e in questo caso lo spazio aggiuntivo utilizzato è $\Theta(n)$ (o $\Theta(m)$ se procediamo per colonne).

17 30/04/2020

17.1 Principio di ottimalità

Abbiamo esibito un algoritmo in programmazione dinamica come soluzione al problema dell'LCS. Per poter applicare la programmazione dinamica, però, bisogna prima dimostrare il principio di ottimalità, ovvero che la soluzione ottima può essere ricavata dalle soluzioni ottime dei sottoproblemi. Dimostriamolo.

Teorema: Siano $x = x_1...x_n$, $y = y_1...y_m$, $z = z_1...z_k = \text{LCS}(x, y)$. Allora vale che:

1. se $x_n = y_m$, allora $z_k = x_n = y_m$ e $z_1...z_{k-1} = \text{LCS}(x_1...x_{n-1}, y_1...y_{m-1})$
2. se $x_n \neq y_m$, allora $z_k \neq x_n$ implica $z_1...z_k = \text{LCS}(x_1...x_{n-1}, y)$
3. se $x_n \neq y_m$, allora $z_k \neq y_m$ implica $z_1...z_k = \text{LCS}(x, y_1...y_{m-1})$

Dimostrazione:

1. Supponiamo per assurdo che $z_k \neq x_n = y_m$. Allora $z_1...z_k x_n$ sarebbe una sottosequenza comune più lunga di z, che è assurdo poiché z era la più lunga. Ne segue immediatamente che $z_1...z_{k-1} = \text{LCS}(x_1...x_{n-1}, y_1...y_{m-1})$.
2. Se per assurdo non fosse che $z_1...z_k = \text{LCS}(x_1...x_{n-1}, y)$, poiché $z_1...z_k$ è comunque una sottosequenza comune a $x_1...x_{n-1}$ e y dovrebbe valere che $\text{LCS}(x_1...x_{n-1}, y)$ è più lunga di z. Ma allora anche $\text{LCS}(x, y)$ sarebbe più lunga di z, da cui un assurdo.
3. Analogo al punto (2) □

17.2 Problema di allineamento ottimo (edit distance)

Siano $x = x_1...x_n$, $y = y_1...y_m$ due sequenze con $n \approx m$. Vogliamo "allineare" x e y (ovvero inserire spazi vuoti in entrambe le sequenze per farle coincidere elemento per elemento il più possibile) con il minor numero di errori possibili, dove con errore intendiamo:

1. Mismatch: $\widetilde{x_i} \neq \widetilde{y_i}$, dove $\widetilde{x}$ e $\widetilde{y}$ sono le sequenze modificate.
2. Inserimento: si allinea un blank (carattere vuoto) della sequenza $\widetilde{x}$ con un carattere della sequenza $\widetilde{y}$

3. Cancellazione: si allinea un blank della sequenza $\widetilde{y}$ con un carattere della sequenza $\widetilde{x}$

Tutti e tre gli errori hanno lo stesso peso.

Applichiamo la programmazione dinamica per risolvere il problema:

1. Definiamo come problema generale $\text{ED}(n, m) = \text{ED}(x, y)$. Definiamo come sottoproblema, lo stesso problema posto sui prefissi di x e y. La forma generale del sottoproblema generico è $\text{ED}(i, j) = \text{ED}(x_1...x_i, y_1...y_j)$.

2. Definiamo come problemi elementari i casi in cui una delle due stringhe è vuota. $\text{ED}(\varnothing, j) = j$ (j errori dovuti a j inserimenti), $\text{ED}(i, \varnothing) = i$ (i errori dovuti a i cancellazioni).

3. Dobbiamo determinare una regola ricorsiva. Facciamo un esempio. Dati x = ALBERO, y = LABBRO, proviamo prima ad allinearli così come sono:

$x =$	A	L	B	E	R	O	
$y =$	L	A	B	B	R	O	
	X	X	-	X	-	-	distanza = 3

   dove con distanza intendiamo la somma totale degli errori. Proviamo con un altro allineamento:

$x =$	A	L		B	E	R	O
$y =$		L	A	B	B	R	O
	X	-	X	-	X	-	-	distanza = 3

   Per costruire la nostra formula ricorsiva, definiamo prima la seguente variabile indicatrice:

$$p_{ij} = \begin{cases} 0 & \text{se } x_i = y_j \\ 1 & \text{se } x_i \neq y_j \end{cases}$$

   Definiamo allora la seguente formula ricorsiva:

$$ED(i, j) = min \begin{cases} ED(i-1, j-1) + p_{ij} \\ ED(i, j-1) + 1 \\ ED(i-1, j) + 1 \end{cases}$$

   dove il primo dei tre possibili risultati rappresenta la scelta di un (possibile) mismatch, mentre gli altri due rappresentano la scelta di piazzare un blank.

4. Per salvare i risultati di $\text{ED}(i, j)$ utilizziamo una tabella L[0... n, 0... m].

Come fatto per il problema dell'LCS, riempiamo la tabella per righe o per colonne fino a completarla. Nell'esempio di x =ALBERO, y =LABBRO, la tabella risultante sarebbe:

	∅	L	A	B	B	R	O
∅	0	1	2	3	4	5	6
A	1	1	1	2	3	4	5
L	2	1	2	2	3	4	5
B	3	2	2	2	2	3	4
E	4	3	3	3	3	3	3
R	5	4	4	4	4	3	4
O	6	5	5	5	5	4	5

Il costo in tempo è $\Theta(n \cdot m)$. Analogamente a quanto visto per il problema dell'LCS, se vogliamo soltanto sapere la distanza, il costo in spazio è $\Theta(n)$ (o $\Theta(m)$). Se invece vogliamo poter ricostruire l'allineamento, il costo in spazio è $\Theta(n \cdot m)$.

17.3 Problema 0-1 Knapsack (o dello zaino, della bisaccia, o del ladro)

Il nome del problema deriva dalla sua spiegazione "informale": l'obiettivo è quello di cercare di inserire in uno zaino quanta più refurtiva possibile in modo che il peso non ecceda il carico massimo dello zaino, massimizzando il profitto.
La versione formale del problema è la seguente. Supponiamo di avere n oggetti $s_1, ..., s_n$, dove ad ogni oggetto $s_i = (p_i, v_i)$ è associato un peso p_i (intero positivo) e un valore v_i, e sia W il carico massimo trasportabile (un intero positivo). L'obiettivo è quello di determinare $S \subseteq \{1, ..., n\}$ tale che $\sum_{i \in S} v_i$ sia massimo e $\sum_{i \in S} p_i \leq W$.
Questo problema è un problema "difficile" (NP-Hard[13]).
Vediamo un esempio

	s_1	s_2	s_3
Peso	10	6	6
Valore	5	4	4

con $W = 12$, nel quale caso la soluzione è $S = \{2, 3\}$.
Proviamo a risolverlo con un algoritmo greedy. Il paradigma greedy (dall'inglese "goloso/avido") è una strategia di creazione di algoritmi dove si cerca di risolvere un problema globale prendendo ripetutamente la soluzione più "proficua".

Algoritmo greedy 1: In questo caso, proviamo a selezionare un oggetto alla volta prendendo di volta in volta quello di valore massimo, finché abbiamo posto nello zaino.
E' facile verificare, però, che questo algoritmo non da la soluzione ottimale. Applicato sull'esempio fatto restituisce come soluzione $S = \{5\}$, che non è la risposta cercata.

[13]Spiegheremo verso la fine del corso cosa ciò voglia dire. Per adesso vi basta sapere che NP-Hard vuol dire (molto approssimativamente) che è a complessità esponenziale

Algoritmo greedy 2: Proviamo invece a selezionare di volta in volta l'oggetto con il miglior rapporto $\frac{\text{valore}}{\text{peso}}$, finché abbiamo posto nello zaino.
Anche in questo caso, però, la soluzione restituita è $S = \{5\}$, che non è la risposta cercata.
Osserviamo che il motivo per cui il secondo algoritmo fornito non funziona è perché gli oggetti sono indivisibili (rappresentato anche dallo "0-1" nel nome del problema: per ogni oggetto, o prendi tutto (1) o non prendi niente (0), e non puoi prenderne una frazione). Nella variante del problema in cui di ogni oggetto si può prendere anche una quantità frazionaria, il secondo algoritmo fornisce la soluzione ottimale.

Proviamo ora a scrivere una soluzione al problema 0-1 Knapsack in programmazione dinamica. Strutturiamo i quattro passi di un algoritmo in programmazione dinamica:

1. Definiamo come problema generale $Z(n, W)$, ovvero il problema posto su tutti gli n oggetti e con una soglia massima di peso W. Definiamo come sottoproblema $Z(i, j)$ (con $0 \leq i \leq n$, $0 \leq j \leq W$), ovvero il problema posto sui primi i oggetti con una soglia massima di peso j.

2. Definiamo come problemi elementari i casi in cui uno tra i e j sia zero, ovvero $Z(i, 0) = 0$, $Z(0, j) = 0$.

3. Come regola ricorsiva, consideriamo il seguente fatto: $Z(i, j)$ è uguale al massimo tra il risultato di mettere i nell'insieme della soluzione e non mettere i nell'insieme della soluzione. Quindi

$$Z(i,j) = max \begin{cases} Z(i-1, j-p_i) + v_i & \text{corrispondente all'inserire } i \\ Z(i-1, j) & \text{corrispondente al non inserisce } i \end{cases}$$

   dove ovviamente bisogna tenere conto della prima opzione solo se $j - p_i \geq 0$

4. Salviamo i risultati di $Z(i, j)$ in una tabella T[0... n, 0... W].

Facciamo un esempio con i seguenti oggetti

	s_1	s_2	s_3
Peso	1	2	3
Valore	60	100	120

e con $W = 5$. Prendiamo la tabella vuota e, come fatto per altri problemi, rempiamola per righe. Supponiamo di aver riempito la tabella fino al caso $j = 3, i = 2$ (escluso).

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100			
3						

In questo caso, prendere l'oggetto $i = 2$ corrisponde a $Z(1, 3-2) + 100 = 160$. Non prenderlo invece corrisponde a $Z(1, 3) = 60$. Quindi, poiché dobbiamo prendere il massimo dei due, in posizione $j = 3, i = 2$ scriveremo 160. La tabella completa è

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3	0	60	100	160	180	220

Il costo di questo algoritmo, sia in termini di tempo che di spazio, è $\Theta(n \cdot W)$. Viene allora spontaneo chiedersi perché all'inizio avessimo definito 0-1 Knapsack come un problema "difficile", e quindi a complessità esponenziale. La risposta è che in realtà la soluzione trovata non è a costo polinomiale, ma pseudopolinomiale. Con complessità pseudopolinomiale si intende una complessità che sembra polinomiale rispetto alla dimensione dell'input, ma è in realtà esponenziale. Per spiegare meglio cosa si intende con pseudopolinomiale, consideriamo i seguenti due casi:

- Supponiamo che un algoritmo prenda in input un array di dati di lunghezza n, e di voler calcolare il costo dell'algoritmo rispetto alla dimensione dei dati passati in input. La dimensione effettiva in memoria dei dati in input cresce linearmente rispetto ad n, poiché la dimensione dell'array cresce linearmente con n. Allora il costo dell'algoritmo andrà calcolato rispetto a n.

- Supponiamo che un algoritmo prenda in input un numero W, e di voler calcolare il costo dell'algoritmo rispetto alla dimensione dei dati passati in input. La dimensione effettiva dei dati in input cresce logaritmicamente rispetto a W, poiché ho bisogno solo di $log(W)$ bit per rappresentare W in memoria. Allora il costo dell'algoritmo andrà calcolato rispetto a $log(W)$

In questi termini, la complessità reale dell'algoritmo è $\Theta(n \cdot W) = \Theta(n \cdot k^{log(W)})$ dove $log(W)$ è la quantità che realmente ci interessa. L'algoritmo allora ha complessità esponenziale.

Un'altra soluzione al problema 0-1 Knapsack è usare la brute force, ovvero controllare tutte le possibili soluzioni per vedere quali sono valide, e tra quelle valide prendere quella a somma massima. Algoritmicamente parlando, i passi da compiere sono:

1. Generare tutti i sottoinsiemi possibili di $\{1, ..., n\}$

2. Per ogni sottoinsieme, controllare se la somma dei pesi è minore di W e se la somma dei valori è maggiore dell'ultimo massimo che abbiamo salvato. Se valgono entrambi, salvo questa soluzione come massimo.

Calcolare e controllare il peso e il valore di ogni sottoinsieme è un'operazione che costa $\Theta(n)$. Per calcolare il costo totale dell'algoritmo bisogna contare quanti sottoinsiemi possibili ci sono. I sottoinsiemi possibili sono $|\mathcal{P}(\{1, ..., n\}| = 2^n$ (dove con $\mathcal{P}(\{1, ..., n\}$ si intende l'insieme delle parti di $\{1, ..., n\}$). Un altro modo per calcolare il numero di sottoinsiemi possibili è quello di prendere un albero binario di decisione e strutturarlo in modo che ogni bivio a livello i rappresenti se comprendere o meno l'i-esimo elemento nel sottoinsieme. Così facendo, le foglie rappresenteranno tutti e soli i possibili sottoinsiemi (per ogni foglia sappiamo che elementi considerare e che elementi escludere ripercorrendo la strada verso la radice e vedendo che direzione era stata presa ad ogni bivio). Quindi il numero di sottoinsiemi è uguale al numero di foglie di un albero binario completo di altezza n, ovvero 2^n.
La complessità dell'algoritmo brute force è quindi $\Theta(n \cdot 2^n)$. Questo algoritmo può essere leggermente migliorato, escludendo i sottoalberi che hanno come radice un nodo ottenuto da un percorso la cui somma dei pesi già supera il carico massimo, ma la complessità resta comunque esponenziale.

Abbiamo visto quindi due algoritmi: uno in programmazione dinamica, con costo esponenziale rispetto a W, uno in brute force, con costo esponenziale rispetto a n. Nel caso dovessimo realmente calcolare la soluzione di un problema 0-1 Knapsack possiamo usare uno o l'altro algoritmo in base a quale è più piccolo tra n e W.

18 05/05/2020

18.1 Grafi

Introduciamo un po' di nomenclatura:

Un grafo è una coppia G(V, E) di insiemi, dove V = $\{v_1, ..., v_n\}$ rappresenta l'insieme dei nodi, e E $\subseteq V \times V$, E = $\{(v_i, v_j), ...\}$ (con $|E| = m \leq n^2$) è un insieme di coppie di vertici che rappresentano gli spigoli o archi del grafo.

Un grafo può essere:

- Orientato, ovvero dove le coppie di E sono coppie ordinate. In questo caso si parla di archi (arcs) e nodi (nodes).
- Non orientato, dove le coppie di E non sono coppie ordinate. In questo casi si parla di spigoli (edges) e vertici (vertices).

Un grafo (orientato o meno) si dice connesso se vale che per ogni coppia di nodi esiste un percorso di archi che li connette. Nel caso di un grafo orientato, gli archi considerati per costruire il percorso devono essere nella giusta "direzione": tendenzialmente si indica come primo elemento della coppia il nodo di partenza dell'arco, e come secondo il nodo di arrivo; nel costruire il percorso bisogna sempre percorrere gli archi in questo verso.

In questo corso tratteremo principalmente grafi connessi.

Preso un grafo connesso, se vale che per ogni coppia di nodi c'è al più un arco, allora vale che $n - 1 \leq m \leq \frac{n(n-1)}{2}$.
Se $m = O(n)$ allora il grafo si dice sparso. Se $m = O(n^2)$ allora il grafo si dice denso. Chiaramente per poter parlare di grafo sparso o denso non si può considerare un singolo grafo, poiché $O(n)$ e $O(n^2)$ riguardano tassi di crescita. Ci si riferisce infatti ad una serie di grafi o un modello di grafo in cui si riesce a scrivere m in funzione di n. Volendo fare considerazioni su un singolo grafo, si può dire che il grafo è sparso se m è più vicino (in ordine di grandezza) a n, denso se è più vicino a n^2.

Nelle stesse ipotesi, se vale anche che $m = \frac{n(n-1)}{2}$, ovvero m è il massimo possibile, il grafo viene detto "Clique".

Un grafo può essere pesato, ovvero può essere un grafo in cui gli archi, oltre a indicare la connessione tra due nodi hanno anche associato un valore reale. In questo caso l'espressione del grafo diventa G(V, E, W), dove W è una funzione W: E $\rightarrow \mathbb{R}$ che associa ad ogni arco il suo peso.

Se $\exists(v_i, v_j)$ arco appartenente a E (oppure $\{v_i, v_j\}$ spigolo appartenente a E, nel caso di un arco non orientato), si dice che i nodi v_i e v_j sono adiacenti. Inoltre, si dice che l'arco (v_i, v_j) è incidente in v_i e v_j.

Si dice grado di $v \in V$ il numero di archi incidenti in v

Si dice cammino una successione di archi tali che preso un arco e il suo successivo, il nodo di arrivo del primo è il nodo di partenza del secondo. Si dice lunghezza di un cammino la somma dei pesi degli archi che attraversa. Se il grafo non è pesato, la lunghezza del cammino è il numero di archi attraversati.

Si dice ciclo un cammino che inizia e termina sullo stesso nodo.

Si dice distanza tra due nodi la lunghezza del cammino più corto che li congiunge

Si dice sottografo di G(V, E) un grafo G(V', E') tale che V' $\subseteq$ V e E' $\subseteq$ E $\cap$ (V'$\times$V').

Ora che abbiamo introdotto tutta questa notazione, siamo pronti a parlare di grafi da un punto di vista algoritmico e come strutture dati.

18.2 Rappresentazione di un grafo in memoria

I metodi principali per salvare in memoria un grafo sono due: le matrici di adiacenza e le liste di adiacenza.

Matrici di adiacenza

Con matrice di adiacenza si intende una matrice $n \times n$ (dove n è il numero di nodi del grafo) che contiene tutte le informazioni su quali nodi sono adiacenti e quali no. In particolare, nel caso di un grafo non pesato, l'entrata di coordinate (i, j) sarà 0 se i nodi v_i, v_j non sono adiacenti, 1 altrimenti. Nel caso di un grafo pesato, alla coordinata (i, j) ci sarà (se esiste) il peso dell'arco che unisce v_i e v_j. Osserviamo che nel caso in cui il grafo non sia orientato basta in realtà salvare solo la metà triangolare superiore (o inferiore) della matrice, poiché simmetrica. Il costo in spazio di questa struttura è $\Theta(n^2)$. Il costo in tempo per elencare i nodi adiacenti ad un dato nodo è $\Theta(n)$.

Liste di adiacenza

Con lista di adiacenza si intende un array di lunghezza n dove ogni cella è una lista (o meglio, puntatore a lista), dove la lista all'indice i-esimo contiene tutti gli indici dei nodi adiacenti al nodo v_i.
Il costo in spazio di questa struttura è $\Theta(n+m)$. Il costo al caso pessimo quindi è $\Theta(n^2)$, ma si preferisce lavorare con le liste di adiacenza perché in generale il costo è $O(n^2)$.
Un altro motivo per preferire la rappresentazione con liste di adiacenza è che il costo per elencare i nodi adiacenti ad un nodo è $\Theta(numerodinodiadiacenti) \leq \Theta(n)$.
Un'ipotesi importante che bisogna avere per poter lavorare con le liste di adiacenza è che il grafo sia statico (ovvero che non cambi il numero di nodi e non cambino gli archi) e che ci sia al più un arco per coppia di nodi.

19 07/05/2020

19.1 Visita BFS (Breadth-First Search)

Il primo tipo di visita di un grafo che vedremo è la BFS (Breadth-First Search, o ricerca in ampiezza), che si applica sui grafi non pesati. Nella BFS, fissato un nodo di partenza, vengono visitati prima tutti i nodi a distanza 1, poi tutti i nodi a distanza 2, e così via. Sarebbe, per intenderci, l'equivalente di una visita per livelli di un albero.

Per implementare questo algoritmo si utilizzano tre strutture di appoggio:

- Una coda semplice (quindi FIFO) Q, che conterrà i nodi da visitare nell'ordine in cui andranno visitati. Verrà aggiornata volta per volta togliendo il nodo che stiamo visitando e aggiungendo i nodi adiacenti al nodo che stiamo visitando che non abbiamo già visitato.

- un array di booleani visitato[] di dimensione uguale alla quantità di nodi, che indicherà se un nodo è già stato visitato o meno, per evitare di rimanere incastrati in loop infiniti in caso di presenza di cicli nel grafo.

- un array di booleani inCoda[] di dimensione uguale alla quantità di nodi, che indicherà se un nodo è già stato inserito nella coda di nodi da visitare, per evitare di inserire più di una volta un nodo nella coda prima che venga visitato.

Il primo nodo che aggiungeremo alla coda è il nodo sorgente. A questo punto, iterativamente si estrarrà un nodo dalla coda, lo si esaminerà e si aggiungerà alla coda tutti i nodi adiacenti che non sono già stati aggiunti alla coda o visitati. Così facendo, con il primo nodo aggiungeremo alla coda tutti i nodi a distanza 1. Con il primo nodo a distanza 1 aggiungeremo alla coda alcuni nodi a distanza 2, ma prima di arrivare ad analizzare un nodo a distanza 2 avremo analizzato tutti i nodi a distanza 1. Poiché abbiamo analizzato tutti i nodi a distanza 1 e aggiunto tutti i vicini che non erano già stati visitati o aggiunti in coda, nella coda avremo tutti e soli i nodi a distanza 2. Il discorso è analogo per i nodi a distanza 3, 4 e così via. Così facendo, i nodi verranno visitati in ordine di distanza.
Supponiamo di avere a disposizione n, il numero di nodi, e Adj[], la lista di adiacenza che contiene il grafo. Scriviamo adesso la visita BFS in pseudocodice (dove s è il nodo sorgente scelto):

```
BFS(s)
{
    Q = nuovacoda();
    raggiunto = nuovoarray(boolean, n);
    inCoda = nuovoarray(boolean, n);

    for(i = 0; i < n; i++)
    {
        raggiunto[i] = false;
        inCoda[i] = false;
    }

    Enqueue(Q, s);
    inCoda[s] = true;

    while(!isEmpty(Q))
    {
        u = Dequeue(Q);
        raggiunto[u] = true;

        // Qui visiti il nodo. Se necessario, stampa
        //  il nodo, o fai quello che devi fare mentre
        //  lo analizzi

        for(x = Adj[u]; x != null; x = x.succ)
        {
            v = x.dato;
```

```
            if(!inCoda[v])
            {
                Enqueue(Q, v);
                inCoda[v] = true;
            }
        }
    }
}
```

Il costo in spazio di BFS è $\Theta(n)$. Infatti, abbiamo bisogno di

- una coda, lunga al più n,
- due array di booleani, lunghi n.

In termini di tempo, il costo è diviso in:

- $\Theta(n)$ per il ciclo for iniziale
- $l_0 + l_1 + ... + l_{n-1}$ per il ciclo while (dove l_i è il numero di archi incidenti all'i-esimo nodo, e rappresenta il numero di iterazioni compiute dal ciclo for più interno per il nodo $i - esimo$). Poiché vale $l_0 + ... + l_{n-1} = 2m$ (oppure m se il grafo è orientato), il costo del while è $\Theta(m)$.

Il costo totale in termini di tempo è quindi $\Theta(n + m)$.

La visita BFS induce uno spanning tree (o albero di copertura), ovvero un albero che contiene tutti i vertici del grafo e come archi ha un sottoinsieme degli archi del grafo, che prende il nome di albero BFS. Nell'albero BFS la radice è il nodo sorgente e ogni nodo ha come padre il nodo che lo ha aggiunto alla coda mentre veniva visitato.
L'albero BFS da per ogni nodo v il percorso minimo da v a s. Con una leggera modifica, quindi, questo algoritmo può essere utilizzato per calcolare la distanza tra due nodi. Infatti, vale il seguente teorema:

Teorema: La distanza tra v e s è uguale alla profondità di v nell'albero BFS con sorgente in s.

Nel caso di un grafo non orientato, la BFS può essere utilizzata per verificare che esso sia connesso. Nel caso invece di un grafo orientato, la BFS può essere utilizzata per verificare quali nodi siano raggiungibili da s.
Inoltre, eseguendo la BFS da tutte le sorgenti possibili (in tempo $\Theta(n(n+m))$) è possibile calcolare il diametro del grafo, che è definito come la distanza massima tra due nodi, $d = max_{u,v \in V}\{dist(u, v)\}$.

Nel caso in cui il grafo non sia salvato su una lista di adiacenza ma su un altro tipo di struttura dati, è comunque possibile eseguire una visita BFS leggermente diversa. In questo caso faremo uso di un dizionario D dove salveremo i nodi

che abbiamo già aggiunto alla coda Q. Ipotiziamo anche di avere una funzione Adj(u) che restituisce una lista con tutti i nodi adiacenti a u (funziona anche se Adj(u) restituisce un array o altro, ma con la lista è più facile scrivere il codice). In pseudocodice diventa:

```
BFSExplore(s)
{
    Q = nuovacoda();
    D = nuovodizionario();
    Enqueue(Q, s);
    while(!isEmpty(Q))
    {
        u = Dequeue(Q);
        if(Ricerca(D, u) == NULL)
        {
            Inserimento(D, u);
            for(x = Adj(u); x != NULL; x = x.succ)
            {
                v = x.dato;
                Enqueue(Q, v);
            }
        }
    }
}
```

Il costo di questa procedura dipende da come viene implementato il dizionario (di cui utiliziamo solo inserimento e ricerca):

- Se D è implementato con una tabella hash, il costo al caso medio è $\Theta(n+m)$.
- Se D è implementato con alberi AVL, ricerca e inserimento hanno costo $O(log(n))$. Quindi il costo dell'algoritmo è $O((n+m)log(n))$.

19.2 Visita DFS (Depth-First Search)

Un altro tipo di visita di grafi possibile è la visita DFS (Depth-first search, o ricerca in profondità). Se la BFS era l'equivalente della visita per livelli degli alberi per i grafi, la DFS è l'equivalente della visita anticipata.
La DFS non prevede un nodo sorgente, e visita tutti i nodi del grafo, anche in presenza di componenti sconnesse (ovvero insiemi di nodi tali per cui non è possibile raggiungere un insieme dall'altro attraverso gli archi del grafo). Per fare ciò, la DFS utilizza una funzione di appoggio ricorsiva, detta DFS-Visit, che definiremo dopo.
Ipotizziamo anche qui di avere un grafo salvato in una lista di adiacenza Adj[], e di disporre di n il numero di nodi. L'algoritmo in pseudocodice è:

```
DFS(Adj)
{
    raggiunto = nuovoarray(n);
    for(i = 0; i < n; i++)
        raggiunto[i] = false;

    for(s = 0; s < n; s++)
        if(!raggiunto[s]) DFS-Visit(s);
}
```

Scriviamo ora lo pseudocodice di DFS-Visit (dove ipotiziamo che l'array creato nella funzione DFS sia una variabile globale, o comunque una variabile leggibile e modificabile da questa funzione esterna):

```
DFS-Visit(u)
{
    raggiunto[u] = true;
    for(x = Adj[u]; x != NULL; x = x.succ)
    {
        v = x.dato;
        if(!raggiunto[v]) DFS-Visit(v);
    }
}
```

Osserviamo che per questo algoritmo non serve una coda di nodi da visitare, perché la struttura ricorsiva memorizza l'ordine dei nodi da visitare in maniera "implicita".
C'è da aggiungere che se il grafo considerato è connesso, la prima chiamata di DFS-Visit() visita, con le chiamate ricorsive, tutti i nodi del grafo, e non c'è quindi bisogno di utilizzare la funzione DFS(). In generale, preso un generico nodo s di un grafo connesso e posto come il nodo di indice 0, chiamare DFS(Adj) e DFS-Visit(s) avrà lo stesso risultato.
Se il grafo è connesso, anche la DFS induce un albero, detto albero DFS. Questo albero permette una classificazione degli archi del grafo, dividendoli nelle seguenti quattro categorie:

- Un arco viene detto arco dell'albero se compare anche nell'albero DFS.
- Un arco viene detto arco all'indietro (backward) se connette un nodo ad un altro nodo suo antenato nell'albero DFS.
- Un arco viene detto arco in avanti (forward) se connette un nodo ad un suo discendente che non sia suo figlio.
- Un arco viene detto arco di attraversamento se connette due nodi che non sono uno antenato dell'altro.

Nei grafi non orientati non c'è distinzione tra gli archi forward e gli archi backward, e vengono entrambi classificati come backward.

20 12/05/2020

20.1 Ordinamento topologico

Consideriamo un grafo orientato che non presenta cicli (o DAG, Directed Acyclic Graph). Un ordinamento topologico consiste in un ordinamento globale dei nodi del grafo che sia compatibile con gli archi del grafo, intendendo gli archi come relazione d'ordine parziale. In pratica, definita la relazione d'ordine parziale "$v_i < v_j$ se esiste un cammino da v_i a v_j", l'obiettivo è quello di assegnare ad ogni vertice un indice univoco in $\{1, ..., n\}$ tale che non esistano due vertici u, v con $\eta(u) < \eta(v)$ e $u > v$
Definiamo nodi sorgente tutti quei nodi che non hanno archi entranti e nodi pozzo tutti quei nodi che non hanno archi uscenti. Poiché non ci sono cicli, in ogni DAG esiste almeno un nodo pozzo.

Per definire un ordinamento topologico dobbiamo definire una funzione $\eta : V \to \{1, ..., n\}$ che rispetti le proprietà di ordinamento spiegate prima. Una volta fatto ciò, sviluppiamo il seguente algoritmo:

1. Si inizializza un contatore che parte da n-1.
2. Da un nodo s si fa una DFS leggermente modificata per trovare un nodo pozzo z,
3. Si assegna $\eta(z)$ uguale al valore del contatore,
4. Si eliminano gli archi entranti in z.
5. Si decrementa il contatore e si ripete dal punto 2.

Utilizzeremo quindi: un array raggiunto[0... n-1] di booleani, una variabile contatore e un array eta[0... n-1] di interi, dove assegneremo all'indice i il valore $\eta(v_i)$ (supponendo di aver etichettato già i nodi con indici da 0 a n-1). Tutte e tre le variabili saranno globali, o quanto meno accessibili da entrambe le funzioni di cui avremo bisogno.
Scriviamo ora la procedura in due funzioni:

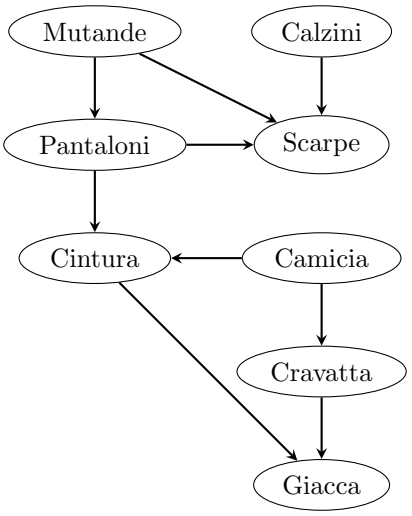
```
OrdinamentoTopologico(Adj)
{
    raggiunto = nuovoarray(boolean, n);
    eta = nuovoarray(int, n);
    for(i = 0; i < n; i++)
        raggiunto[i] = false;

    contatore = n-1;

    for(s = 0; s < n; s++)
        if(!raggiunto[s]) DFS-Ordina(s);
}
```

```
DFS-Ordina(u)
{
    raggiunto[u] = true;
    for(x = Adj[u]; x != NULL; x = x.succ)
    {
        v = x.dato;
        if(!raggiunto[v]) DFS-Ordina(v);
    }
    eta[u] = contatore;
    contatore--;
}
```

Il costo, analogamente a DFS, è di $\Theta(n)$ in spazio e $\Theta(n+m)$ in tempo.
Supponendo di avere un DAG di operazioni da svolgere, dove ogni nodo rappresenta una azione da fare e se da u parte un arco verso v vuol dire che u deve essere svolta prima di v, si può utilizzare un ordinamento topologico mettere in una lista le operazioni da svolgere in modo che vengano rispettate le richieste sugli ordini.
Ad esempio, supponiamo di avere questo grafo che indica con quale priorità vanno indossati gli indumenti quando ci si veste:



Applicando un ordinamento topologico si potrebbe ottenere il seguente ordinamento: Camicia, Cravatta, Calzini, Mutande, Pantaloni, Scarpe, Cintura, Giacca.
Osserviamo che questo non era l'unico ordinamento topologico possibile, ma uno dei vari ottenibili da questo DAG.

20.2 Algoritmo di Dijkstra

Abbiamo visto che la BFS può essere utilizzata per calcolare il cammino di distanza minima. Ciò si applica, però, solo su grafi non pesati, dove con distanza

si intende il numero di archi. Per calcolare il cammino minimo tra due nodi in un grafo G(V, E, W) pesato possiamo utilizzare l'algoritmo di Dijkstra[14].
L'algoritmo di Dijkstra prende in input un grafo e un nodo sorgente, e restituisce per ogni nodo del grafo il cammino minimo dalla sorgente al nodo. All'atto pratico, Dijkstra restituisce un array di puntatori, uno per ogni nodo del grafo, dove ad ogni nodo viene associato il puntatore al nodo a lui precedente nel cammino suo minimo. Così facendo, seguendo i puntatori, è possibile ricostruire il cammino minimo per ogni nodo.
Il motivo per cui salvare solo il puntatore al nodo precedente funziona per restituire il cammino minimo è che tutti i sottocammini di un cammino minimo sono anch'essi cammini minimi. In pratica, se $s, v_{i1}, v_{i2}, ..., v_{ik-1}, v_{ik}, v$ è il cammino minimo di v, allora necessariamente $s, ..., v_{ik-1}, v_ik$ è il cammino minimo di v_{ik}.
L'algoritmo di Dijkstra trova molte applicazioni, quali ad esempio il problema del routing dei messaggi sulla rete: partendo da un grafo pesato che rappresenta la rete, bisogna stabilire il percorso minimo per fare arrivare un messaggio da un nodo di partenza ad un nodo di arrivo.
L'algoritmo di Dijkstra utilizza le seguenti strutture di appoggio:

- Un array pred[0... n-1] (inizializzato a -1) dove in indice i verrà assegnato l'indice del nodo precedente al nodo i nel suo cammino minimo.
- Un array dist[0... n-1] (inizializzato a $+\infty$ ovunque tranne in indice s (sorgente), dove viene inizializzato a 0) che volta per volta aggiornerà le distanze calcolate tra la sorgente e i nodi.
- Una coda con priorità (un minheap, nel nostro caso) che conterrà i nodi da visitare elencati in ordine di distanza, la cui priorità verrà aggiornata qualora si trovasse un nuovo modo per raggiungere quel nodo più velocemente.

In pseudocodice diventa:

```
Dijkstra(s)
{
    pred = nuovoarray(n);
    dist = nuovoarray(n);
    PQ = nuovoheap();
    for(u = 0; u < n; u++)
    {
        pred[u] = -1;
        dist[u] = +inf;
    }
    dist[s] = 0;
    for(u = 0; u < n; u++)
    {
        elemento = nuovoelemento();
```

[14]Edsger Dijkstra: 1930-2002, fu un informatico Olandese

```
        elemento.peso = dist[u];
        elemento.dato = u;
        Enqueue(PQ, elemento);
    }
    while(!isEmpty(PQ))
    {
        e = Dequeue(PQ);
        v = e.dato;
        for(x = Adj[v]; x != NULL; x = x.succ)
        {
            u = x.dato;
            if(dist[u] > dist[v] + x.peso)
            {
    // Se entriamo in questo if vuol dire che abbiamo trovato un
    //  percorso per arrivare a u migliore dell'ultimo che
    //  avevamo salvato. Allora sovrascrivi la distanza con
    //  quella migliorata e aggiorna la priorità di u nell'heap

                dist[u] = dist[v] + x.peso;
                pred[u] = v;
                DecreaseKey(PQ, u, dist[u]);
            }
        }
    }
}
```

dove DecreaseKey(PQ, u, dist[u]) accede all'elemento u nella coda PQ, aggiorna il valore della distanza al nuovo valore e poi ristruttura l'heap mantenendolo ordinato.
[Ora, francamente io non capisco perché l'algoritmo visto a lezione abbia quel ciclo for che aggiunge tutti i nodi alla coda. Le implementazioni che trovo su internet, ad esempio, non hanno quel ciclo for. Uno potrebbe dire che è perché, visto che stiamo facendo una visita del grafo e tendenzialmente vogliamo visitarlo tutto, così facendo potremo visitare anche le componenti non connesse con quella del nodo sorgente nel grafo. Questa cosa però non può funzionare, poiché se un nodo non è connesso al nodo sorgente, non esiste nessun cammino che possa arrivarci, quindi la sua distanza rimarrà sempre $+\infty$. Ciò causa una serie di problemi: primo, tutti i nodi non connessi al nodo sorgente avranno sempre la stessa priorità, impedendo all'algoritmo di funzionare in un modo sensato; secondo, che staremmo effettivamente calcolando (nell'if più interno) un valore più infinito (che in programmazione si intende con INT_MAX o comunque il massimo valore raggiungibile dal tipo di variabile che state usando), cosa che per come abbiamo scritto il codice causerebbe problemi di overflow, roba non bella. Se qualcuno può illuminarmi sul perché esiste quel ciclo for, per favore contattatemi.]
Il minheap viene utilizzato per esaminare ogni volta il nodo più vicino alla

sorgente tra quelli non ancora esaminati.
Sia Dequeue() che DecreaseKey() hanno costo $O(log(n))$, senza tenere conto del costo per la ricerca del nodo u. In un heap, però, la ricerca ha costo lineare al caso pessimo. Tuttavia, conoscendo a priori gli elementi che stanno nell'heap, possiamo usare un array posheap[0... n-1] che contiene la posizione nell'heap di tutti i nodi.
Analizziamo la complessità dell'algoritmo:

- Per fare l'inizializzazione vengono fatti n inserimenti, quindi il costo è $\Theta(n \cdot log(n))$.
- Per il ciclo while c'è un Dequeue che viene eseguito n volte. Tuttavia, la sua complessità è superata dal ciclo for, che compie l_i iterazioni (dove l_i è il numero di nodi adiacenti al nodo i) per ogni nodo i, quindi in totale compie m iterazioni (o $2m$ se il grafo non è orientato).

Resta quindi da determinare la complessità della gestione del posheap e delle DecreaseKey.
Grazie al posheap, l'accesso al nodo i nella coda PQ è il costo di calcolare PQ[posheap[i]], e ha quindi costo costante $\Theta(1)$. Grazie all'accesso costante, il costo di DecreaseKey è $\Theta(log(n))$.
Il posheap costa $\Theta(n)$ per essere costruito all'inizio, ma questa complessità è superata dal costo della costruzione dell'heap. Inoltre, quando faccio un Dequeue o un DecreaseKey, per aggiornare il posheap vado a confrontare e eventualmente scambiare un elemento con il padre fino a raggiungere, al più, la radice. Quindi il costo per mantenere aggiornato il posheap è $O(log(n))$ per ogni poerazione di Dequeue o DecreaseKey.

La complessità totale dell'algoritmo è quindi $\Theta(n \cdot log(n))$ (costo di inizializzazione) + $\Theta(m \cdot log(n))$ (m iterazioni totali del ciclo for più interno moltiplicato per il costo del DecreaseKey) = $\Theta(n \cdot log(n) + m \cdot log(n)) = \Theta(m \cdot log(n))$ (poiché m > n).
Utilizzando strutture dati più complesse come il FibonacciHeap (che non tratteremo) è possibile ridurre la complessità a $\Theta(nlog(n) + m)$.

Osserviamo che questo algoritmo segue il paradigma greedy: infatti, la soluzione globale viene ottenuta compiendo passo dopo passo la scelta ottima "locale" (ovvero prendendo il percorso più rapido nodo per nodo).

L'algoritmo di Dijkstra funziona solo nel caso di archi a pesi positivi. Per pesi negativi, esiste l'algoritmo di Bellman-Ford.

21 14/05/2020

21.1 Minimal Spanning Tree (MST)

Sia G(V, E, W) un grafo pesato con pesi positivi. Vogliamo trovare un albero T(V, E') tale che E' $\subseteq$ E, e che $\sum_{e \in E'} W(e)$ sia minimo. Questo albero viene detto Minimal Spanning Tree, o MST.
Poiché non abbiamo una sorgente prefissata, dobbiamo solo trovare l'insieme di archi di peso minimo che connette i vertici del grafo, senza creare cicli.
Osserviamo che se il grafo non è pesato, poiché il numero di archi sarebbe sempre $n - 1$ (dove ricordiamo $n = |V|$), ogni albero va bene, e possiamo scegliere ad esempio l'albero BFS o l'albero DFS.
Prima di parlare del prossimo teorema, diamo una definizione di "taglio".

Definizione: un taglio (cut) è un sottoinsieme di archi C $\subseteq$ E tale che G'(V, E\C) è sconnesso, ovvero la cui rimozione "sconnette" G.

Teorema: Dato G(V, E, W) e T(V, E'), allora T è un MST per G se vale che $\forall e \in E$:

1. $e \in E' \iff \exists$ un taglio in G che comprende e, ed e è l'arco di peso minimo in questo taglio (detta "condizione di taglio"),
2. $e \notin E' \iff \exists$ un ciclo in G che comprende e, ed e è l'arco di peso massimo di questo ciclo (detta "condizione di ciclo").

Esistono due algoritmi per la risoluzione di questo problema (entrambi degli anni '50): l'algoritmo di Kruskal[15] e l'algoritmo di Prim[16]-Jarnik[17]. Entrambi sono algoritmi greedy.

21.2 Algoritmo di Kruskal

Per l'algoritmo di Kruskal, prima vengono ordinati gli archi del grafo in ordine crescente di peso. Dopodiché si esamina ogni arco in ordine aggiungendoli con criterio a E', e per ogni arco (u, v):

- se u e v sono già collegati in E', u e v non viene scelto (poiché chiuderebbe un ciclo),
- se u e v non sono già collegati, viene aggiunto (u, v) a E'.

L'algoritmo comincia quindi con T(V, $\varnothing$) con n nodi isolati, per poi costruire componenti connesse ogni volta che si aggiunge un arco, arrivando ad avere una sola componente connessa.
Questo algoritmo usa le seguenti strutture di appoggio:

[15] Joseph Bernard Kruskal: 1929-2010, fu uno statistico e matematico Statunitense
[16] Robert Clay Prim: 1921, è un matematico e informatico Statunitense
[17] Vojtěch Jarník: 1897-1970, fu un matematico Ceco

- Un minheap PQ degli archi, ordinati tramite il loro peso.
- Una struttura dati che contenga l'informazione sulle componenti connesse, che possa fare le seguenti operazioni in modo efficiente:
  - Set: dato un nodo x dice a quale componente connessa appartiene (che può essere facilmente realizzato con un array Set[0... n-1] dove in ogni posizione è indicato a quale componente connessa appartiene il relativo nodo).
  - Union: unisce le due componenti connesse di due nodi u e v.
- Una lista doppia (ovvero una lista dove ogni elemento possiede sia un puntatore all'elemento successivo, sia un puntatore all'elemento precedente) MST che memorizza gli archi della soluzione.

Vediamo ora lo pseudocodice per l'algoritmo di Kruskal:

```
Kruskal()
{
    PQ = nuovoheap();
    MST = nuovadoppialista();
    for(u = 0; u < n; u++)
    {
        for(x = Adj[u]; x != NULL; x = x.succ)
        {
            v = x.dato;
            elemento = nuovoelemento();
            elemento.dato = (u, v); // Inteso come arco
            elemento.peso = x.peso;
            Enqueue(PQ, elemento);
        }
    }
    while(!isEmpty(PQ))
    {
        elemento = Dequeue(PQ);
        (u, v) = elemento.dato;
        if(Set(u) != Set(v))
        {
            Unisci(Set(u), Set(v));
            Inserisci(MST, (u,v));
        }
    }
}
```

La complessità dell'algoritmo dipende pesantemente dalla complessità delle operazioni Unione e Set. Un modo efficiente per realizzarli è il seguente: creiamo un array di liste. Ogni lista rappresenterà una componente connessa. La cella in

posizione i dell'array conterrà un puntatore alla lista che rappresenta la componente connessa che contiene il nodo i. A questo punto, ottenere la componente connessa per ogni nodo richiede tempo costante. Per unire due componenti connesse si uniscono le liste mettendone una in coda, e si sovrascrivono i puntatori dell'array che puntavano alla lista messa in coda per puntare alla nuova testa della lista (formatasi dall'unione). Se facciamo sì che ogni volta che uniamo due liste aggiungiamo quella più corta a quella più lunga, possiamo dire che il puntatore associato ad ogni nodo va sovrascritto solo se il nodo viene spostato in una lista lunga almeno il doppio di quella di partenza. Poiché la lunghezza massima per una lista è n, ogni nodo può essere spostato di lista al più $O(log(n))$ volte. Allora il costo di Unione (ammortizzato su tutti gli elementi) è $O(log(n))$ per ogni elemento. Così facendo, il costo totale dell'algoritmo è $O(m \cdot log(m))$ (di inizializzazione) $+O(n \cdot log(n))$ (per il while) $\approx O((m+n)log(m))$

22 21/05/2020

22.1 P e NP

Tratteremo adesso di problemi "decisionali", ovvero problemi la cui risposta è TRUE/FALSE. Tratteremo solo questo tipo di problemi sia perché sono sufficienti ai nostri scopi, sia perché spesso i problemi di ottimo sono riducibili a problemi decisionali. Definiamo le seguenti classi di problemi:

- Classe P: classe dei problemi decisionali risolubili in tempo polinomiale
- Classe NP: classe dei problemi decisionali verificabili in tempo polinomiale (cioè che data una potenziale soluzione si riesce a dire in tempo polinomiale se questa sia effettivamente una soluzione o meno)

Sappiamo sicuramente che P $\subseteq$ NP, ma non sappiamo se valga l'uguale o se l'inclusione sia un'inclusione stretta.

22.2 Riducibilità Polinomiale

Siano $\mathcal{P}_1$ e $\mathcal{P}_2$ due problemi decisionali. Diciamo che "$\mathcal{P}_1$ si riduce a $\mathcal{P}_2$" (e si indica con $\mathcal{P}_1 \to \mathcal{P}_2$, oppure $\mathcal{P}_1 \alpha \mathcal{P}_2$, oppure $\mathcal{P}_1 \leq_p \mathcal{P}_2$, o con altre notazioni non meglio specificate) se ogni istanza x di $\mathcal{P}_1$ si può trasformare in tempo polinomiale in un'istanza x' di $\mathcal{P}_2$ tale che:

$$\mathcal{P}_2(x') = TRUE \iff \mathcal{P}_1(x) = TRUE$$

Osservazione: Se $\mathcal{P}_1 \to \mathcal{P}_2$, $\mathcal{P}_2$ è almeno difficile quanto $\mathcal{P}_1$. Se vale anche $\mathcal{P}_2 \to \mathcal{P}_1$, i due problemi si dicono equivalenti.

Osservazione: La riducibilità è transitiva (ovvero se $\mathcal{P}_1 \to \mathcal{P}_2 \to \mathcal{P}_3$, allora $\mathcal{P}_1 \to \mathcal{P}_3$)

22.3 NP-Complettezza e Problema SAT

Diciamo che un problema $\mathcal{P}$ è NP-Completo se $\mathcal{P} \in$ NP e se ogni problema in NP si riduce a $\mathcal{P}$.

Il primo problema che fu scoperto essere NP-Completo è il Problema SAT (soddisfacibilità o satisfiability).
Il problema SAT prende in input un insieme $X = \{x_0, ..., x_{n-1}\}$ di variabili booleane e un insieme $C = \{c_0, ..., c_{m-1}\}$ di "clausole", dove ogni clausola è un insieme di letterali (e ogni letterale è una delle variabili booleane di X, presa così com'è o negata) concatenati con l'operazione logica OR (ad esempio potrebbe essere $c_0 = x_2$ OR NOT(x_5)).
Con questi input, costruiamo la seguente formula $F = c_0$ AND ... AND c_{m-1} (F in questa forma viene detta in "forma normale congiuntiva").
Definito F, possiamo enunciare il problema: esiste un valore dei letterali per cui la formula F risulti TRUE?

Esempio: (utilizzando la notazione AND = $\wedge$, OR = $\vee$, NOT(x_i) = $\neg x$)
Sia $X = \{x_0, x_1, x_2\}$, $C = \{(x_0 \vee x_1), (\neg x_0 \vee \neg x_1 \vee x_2), (x_2)\}$.
La formula risultante è $F = (x_0 \vee x_1) \wedge (\neg x_0 \vee \neg x_1 \vee x_2) \wedge (x_2)$.
In questo caso, esiste un valore dei letterali per cui F è TRUE, ed è x_0 = TRUE, x_1 = FALSE, x_2 = TRUE.
Come abbiamo detto, SAT è NP-Completo. Per dimostrare ciò esiste il teorema di Cook[18]-Levin[19]:

Teorema: SAT è NP-Completo
Sicuramente vale che SAT $\in$ NP, poiché data una possibile soluzione, si verifica in tempo lineare (rispetto alla lunghezza della formula) se è soluzione o meno. Verificare che ogni problema di NP si riduce a SAT è più complicato, e non lo dimostreremo in questo corso. Vediamo però un esempio di problema che si riduce a SAT.

3-Color

Dato un grafo, l'obiettivo di 3-Color è di colorare i vertici del grafo utilizzando 3 colori in modo che vertici adiacenti non siano dello stesso colore. Mostriamo che si riduce a SAT.
Sicuramente, per 3-Color, vale il punto (1), poiché possiamo controllare per ogni nodo il suo colore e il colore dei suoi vicini, che costa $O(n^2)$. Vediamo che Cominciamo definendo la variabile booleana a_{ij} che rappresenterà se i nodi i e j sono adiacenti. Definiamo poi le variabili r_i, g_i, b_i che rappresenteranno il colore del nodo i-esimo (rosso/giallo/blu).
Queste saranno le variabili aleatorie che daremo in input al problema di SAT che vogliamo creare. Vogliamo poi creare una formula che risulti TRUE se e

[18] Stephen Cook: 1939, è un matematico e informatico statunitense
[19] Leonid Levin: 1948, è un informatico sovietico-americano

solo se il grafo ammette una 3-Colorazione.
Prima di tutto, poiché le variabili a_{ij} sono, appunto, variabili, dobbiamo creare una formula che risulti TRUE se e solo se il grafo rappresentato dagli archi definiti con gli a_{ij} è effettivamente il nostro grafo di partenza. Definiamo quindi

$$A_{ij} = \begin{cases} a_{ij} & \text{se nel nostro grafo di partenza esisteva l'arco } (i,j) \\ \neg a_{ij} & \text{altrimenti} \end{cases}$$

Prendendo tutti questi A_{ij} concatenati con l'operazione AND otterremo una formula che risulta TRUE se e solo se le variabili a_{ij} descrivono il grafo originale. Creiamo ora una formula che sia vera se solo se le variabili r_i, g_i, b_i descrivono una colorazione ammissibile per il nodo i (ovvero ha uno e un solo colore). Tale condizione può essere espressa come

$$B_i = (r_i \wedge \neg g_i \wedge \neg b_i) \vee (\neg r_i \wedge g_i \wedge \neg b_i) \vee (\neg r_i \wedge \neg g_i \wedge b_i)$$

Dopodiché, scriviamo una condizione per ogni coppia di nodi i, j che risulti vera solo se le variabili descrivono questi due nodi come non "in conflitto" in termini di colori adiacenti. Sappiamo che se l'arco esiste, allora i due nodi adiacenti non possono avere lo stesso colore. Quindi o l'arco non esiste, o devono avere colori diversi (in particolare o uno è rosso e l'altro no, o uno è giallo e l'altro no, o uno è verde e l'altro no). Possiamo quindi scrivere questa condizione come

$$C_{ij} = \neg a_{ij} \vee (r_i \wedge \neg r_j) \vee (g_i \wedge \neg g_j) \vee (b_i \wedge \neg b_j)$$

Possiamo quindi esprimere la seguente formula che è soddisfacibile se e solo se il grafo ammette una 3-colorazione:

$$\Phi_G = \bigwedge_{0 \leq i,j < n} A_{ij} \wedge \bigwedge_{0 \leq i < n} B_i \wedge \bigwedge_{0 \leq i,j < n} C_{ij}$$

Questa formula non è in forma normale congiuntiva, ma può essere trasformata in una formula normale congiuntiva in tempo polinomiale. Quindi, 3-Color può essere ridotto a SAT

Vediamo ora alcuni problemi NP-Completi.

Osservazione: $\mathcal{P}$ è NP-Completo se:

1. $\mathcal{P} \in$ NP
2. SAT $\rightarrow \mathcal{P}$

Infatti dalla (1), con il teorema di Cook-Levin si deduce che $\mathcal{P} \rightarrow$ SAT, e con questo fatto unito alla (2) si deduce che $\mathcal{P} \cong$ SAT, che è NP-Completo per il teorema di Cook-Levin.
Per il punto (2) non è necessario usare SAT, basta in realtà usare un qualsiasi problema che è noto essere NP-Completo.

3-SAT

Una variante del problema SAT è il problema 3-SAT, che si formula allo stesso modo ma con la condizione aggiunta che ogni clausola può essere composta al più da 3 letterali. Vale che anche 3-SAT è NP-Completo. Se invece si pone come limite 2 letterali, il problema diventa polinomiale.

Clique (decisionale)

Dato un grafo G(V, E) e dato un intero k, esiste un sottografo completo (ovvero dove ogni coppia di nodi è adiacente) con un numero di nodi maggiore o uguale a k?
Considereremo in realtà un problema un po' più semplice (K-Clique), in cui ci chiediamo se esiste un sottografo completo con k vertici.

1. Verifichiamo che vale K-Clique $\in$ NP. Sia G un grafo e b una stringa binaria di n elementi, tale che $b_i = 1$ se i sta nella soluzione proposta. E' possibile verificare che b sia effettivamente una soluzione in tempo polinomiale? Sì, con il seguente algoritmo (dove il grafo è rappresentato da una matrice di adiacenza G):

   ```
   VerificaClique(G, b)
   {
       num = 0;
       for(i = 0; i < n; i++)
           if(b[i]) num++;

       if(num != k) return FALSE;

       for(i = 0; i < n; i++)
       {
           for(j = 0; j < n; j++)
           {
               if(b[i] AND b[j])
                   if(A[i, j] == 0) return FALSE;
           }
       }
       return TRUE;
   }
   ```

   Questo algoritmo, molto banale, è $\Theta(n^2)$, quindi K-Clique $\in$ NP.

2. Mostriamo che 3-SAT $\rightarrow$ K-Clique. Data un'istanza generica di 3-SAT, creiamo un grafo partendo dalle sue clausole. Per ogni clausola c_i, e per ogni letterale x contenuto nella clausola c_i, creeremo un nodo che avrà come etichetta (x, i). Per gli archi, colleghiamo i nodi (x, i) e (y, j) solo se $i \neq j$ (ovvero i letterali appartenevano a due clausole diverse) e $x \neq \neg y$ (ovvero x e y possono essere vere contemporaneamente).

Dimostriamo che 3-SAT è soddisfacibile se e solo se il grafo creato ha una clique di dimensione k = numero delle clausole.
[⇒] Se SAT è soddisfacibile, allora esistono k letterali (uno per clausola) che possono essere veri contemporaneamente. La clique cercata è quella che ha per vertici i letterali appena citati.
[⇐] Se c'è la clique, i vertici di essa appartengono a clausole diverse e possono essere veri contemporaneamente, quindi SAT è soddisfacibile.

Si osservi che K-Clique è NP-Completo per valori di k che sono $\Theta(n)$. Per valori costanti di k al variare della quantità di nodi, non è NP-Completo.

I problemi NP-Completi sono i problemi più difficili all'interno della classe NP. Esistono anche problemi "open", ovvero problemi che nessuno è ancora riuscito a dimostrare che siano NP-Completi. Ad esempio, il problema della fattorizzazione in numeri primi è open.
Per i problemi aperti c'è speranza di trovare soluzioni polinomiali, mentre per i problemi NP-Completi sembra molto più difficile. Infatti, per via della riducibilità:

- se si trova una soluzione polinomiale per un solo problema NP-Completo, si trova una soluzione efficiente per tutti i problemi NP-Completi,
- se si trova un limite inferiore esponenziale per uno, allora sono tutti esponenziali.

La speranza che P = NP è bassa.

Come si può verificare che un problema sia NP-Completo?

- Cercandolo su Wikipedia
- Cercandolo su libri simil-enciclopedie per i problemi NP-Completi (come "Computers and Intractability", M.Garey[20] & D.S.Johnson[21])
- Dimostrandolo, ma può essere parecchio difficile

22.4 0-1 Knapsack è NP-Hard

Un problema si dice NP-Hard se il suo problema equivalente decisionale è NP-Completo. Per far vedere che 0-1 Knapsack è NP-Hard bisogna quindi far vedere che 0-1 Knapsack decisionale è NP-Completo.
Dato un input di 0-1 Knapsack e un intero k, il problema decisionale è: esiste una soluzione la cui somma dei valori sia maggiore o uguale a k?

1. Sia b una stringa binaria di oggetti selezionati. Per b si calcola:
   - V = somma dei valori,
   - P = somma dei pesi,

[20] Michael Garey: 1945, informatico statunitense
[21] David Stifler Johnson: 1945-2016, fu un informatico statunitense

e si controlla se P $\leq$ peso_max e V $\geq$ k. Il costo è lineare.

2. Mostriamo che il problema Partition[22] si può ridurre a 0-1 Knapsack decisionale. Il problema Partition è il seguente: dato A $= \{a_0, ..., a_n\}$ esiste A' $\subseteq$ A tale che $\sum_{a_i \in A'} a_i = \sum_{a_i \in A \setminus A'} a_i = \frac{1}{2} \sum_{a_i \in A} a_i$?
   Ad esempio, con A $= \{9, 7, 5, 4, 1, 2, 3, 8, 4, 3\}$ basta prendere A' $= \{9, 7, 5, 2\}$.
   Per far vedere che Partition si riduce a 0-1 Knapsack decisionale, mostriamo che Partition si riduce ad un caso particolare (e non generale) di 0-1 Knapsack decisionale ("tecnica della riduzione").
   Consideriamo un problema 0-1 Knapsack in cui per ogni oggetto $p_i = v_i$ e consideriamo k = W $= \frac{1}{2} \sum p_i$.
   Allora Partition $\rightarrow$ 0-1 Knapsack ridotto $\rightarrow$ 0-1 Knapsack decisionale. Quindi 0-1 Knapsack decisionale è NP-Completo e 0-1 Knapsack è NP-Hard

22.5 Quando i problemi NP sono troppo difficili per i nostri gusti

Talvolta può capitare di dover risolvere un problema la cui soluzione ottima è difficile da trovare. In questo caso, possiamo accontentarci di una soluzione "buona" (ma non ottima) che sia facile da trovare. Le tre strategie principali sono:

- algoritmi approssimati, in cui la soluzione ottenuta non è lontana da quella ottima (ad esempio un algoritmo greedy che fa scelte ottime locali),
- algoritmi probabilistici, in cui la soluzione ottenuta è ottima con alta probabilità,
- algoritmi euristici, in cui si cercano algoritmi locali semplici e si mostra con simulazioni statistiche la bontà del metodo in molti casi.

[22] che è NP-Completo

23 Appendici

23.1 L2 è totale - (15.2)

Come scritto nelle note, il fatto che la funzione di hash che abbiamo chiamato L2, definita come

$$h(k,i) = \left(h(k) + \frac{i^2}{2} + \frac{i}{2}\right) \bmod m, \text{ con } m = 2^s,\ s \in \mathbb{N}$$

sia totale non è stato dimostrato a lezione. Per completezza riporto l'unica dimostrazione che sono riuscito a fare per questa proposizione. Avviso che non è una bella dimostrazione, la cui idea di base è relativamente semplice (un'induzione più la generalizzazione di un pattern) che diventa però un po' astratta e a caso se non è chiaro cosa si sta facendo, ovvero quale sia l'idea di base. Qualora trovaste dimostrazioni più eleganti vi prego di farmelo sapere così da poterle sostituire a questa (citandovi nel caso).

Proposizione: L2 è totale.

Questa proposizione vuol dire:

Proposizione: $h(k,i) = \left(h(k) + \frac{i^2}{2} + \frac{i}{2}\right) \bmod m$ colpisce tutte le celle al variare di i.

Poiché il fattore $h(k)$ è solo una costante che trasla gli indici delle celle considerate al variare di i, vale che la proposizione è vera per ogni $h(k)$ se è vera almeno per $h(k) = 0$. Possiamo quindi limitarci a dimostrare la seguente proposizione.

Proposizione: Ogni classe di equivalenza modulo m ha un rappresentante della forma $\frac{i^2}{2} + \frac{i}{2}$ con $i \in \mathbb{N}$.

Dimostreremo in realtà una proposizione più forte di questa, che è:

Proposizione: Ogni classe di equivalenza modulo m ha un rappresentante della forma $\frac{i^2}{2} + \frac{i}{2}$ con $i \in \{0, ..., 2^s - 1\}$.

Dimostrazione: La dimostrazione di questa proposizione si fa per induzione su s. Il caso base $s = 1$ è banale, poiché per $i = 0$ si ottiene la classe di 0 e per $i = 1$ si ottiene la classe di 1.
Per capire bene l'idea del passo induttivo (che faccio un po' fatica a formalizzare astrattamente) vediamo prima un caso particolare: proviamo a dimostrare $s = 4$

sapendo $s = 3$. Indichiamo per semplicità $f(i) = \frac{i^2}{2} + \frac{i}{2} = \frac{i(i+1)}{2}$ e consideriamo i seguenti valori:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$f(i)$	0	1	3	6	10	15	21	28	36	45	55	66	78	91	105	120
$f(i) \bmod 8$	0	1	3	6	2	7	5	4	4	5	7	2	6	3	1	0
$f(i) \bmod 16$	0	1	3	6	10	15	5	12	4	13	7	2	14	11	9	8

Il primo pattern che osserviamo è che i valori di $f(i) \bmod 2^3$ per $i = 0, ..., 2^3 - 1$ sono "uguali e riflessi" (ovvero sono gli stessi ma appaiono in ordine inverso) dei valori di $f(i) \bmod 2^3$ per $i = 2^3, ..., 2^4 - 1$. Chiameremo questa "proprietà 1".
Il secondo pattern che osserviamo è che, passando da mod 2^3 a mod 2^4, non solo i valori o sono gli stessi o sono gli stessi $+ 2^3$ (questo infatti è vero in generale[23]), ma per ogni indice i da 0 a $2^3 - 1$, detto j il suo indice "uguale e riflesso" (ovvero l'unico indice tale che $f(i) \equiv f(j) \bmod 2^3$, ovvero $j = 2^4 - 1 - i$),

$$f(i) \mod 2^4 = f(i) \mod 2^3 \Rightarrow f(j) \mod 2^4 = 2^3 + (f(j) \mod 2^3)$$

e

$$f(i) \mod 2^4 = 2^3 + (f(i) \mod 2^3) \Rightarrow f(j) \mod 2^4 = f(j) \mod 2^3$$

Detto a parole, $f(i) \equiv f(j) \mod 2^3$ e passando all'esponente successivo e guardando i rappresentanti delle classi di equivalenza di $f(i)$ e $f(j)$, uno dei due rappresentanti rimarrà uguale e l'altro aumenterà di 2^3.
Per spiegare questo passaggio si è fatto un po' un abuso di notazione, avendo trattato "$f(i) \mod 2^3$" come un numero intero. Con "$n \mod k$" si intendeva il rappresentante canonico della classe di equivalenza di k modulo k, ovvero l'intero r tale che $0 \leq r < k$ e $r \equiv n \mod k$.
La dimostrazione effettiva sarà più formale e non userà abusi di notazione.
In ogni caso, chiameremo la proprietà evidenziata da questo pattern "proprietà 2".

Spieghiamo ora il passo induttivo sa s a $s+1$ supponendo che sia la proprietà 1 che la proprietà 2 siano vere per ogni s. La dimostrazione della validità di queste due proprietà verrà data subito dopo.

$\underline{s \Rightarrow s+1}$**:**
Gli interi $\{0, ..., 2^s - 1\}$ sono i rappresentanti standard di tutte classi di equivalenza modulo 2^s. Indichiamo questo insieme con $\mathbb{K}(2^s)$.
Sappiamo che tutti i rappresentanti standard delle classi di equivalenza modulo

[23]Per ogni intero n, detto $r_{n,k}$ il rappresentante standard della classe di equivalenza di n modulo k (cioè $r_{n,k} \equiv n \bmod k$ e $0 \leq r_{n,k} < k$) vale sempre una delle due equivalenze: $r_{n,2k} = r_{n,k}$ oppure $r_{n,2k} = r_{n,k} + k$

2^{s+1} sono o interi in $\mathbb{K}(2^s)$ oppure della forma $k + 2^s$ con k intero in $\mathbb{K}(2^s)$. Possiamo quindi dire che

$$\mathbb{K}(2^{s+1}) = \bigcup_{k \in \mathbb{K}(2^s)} \{k, k + 2^s\}$$

Consideriamo ora gli indici $i \in \{0, ..., 2^s - 1\}$. Indichiamo con $r(n, k)$ il rappresentante standard della classe di equivalenza di n modulo k (ovvero, se volete, il resto della divisone Euclidea di n per k).
Per ipotesi induttiva sappiamo che $\{r(f(i), 2^s) \mid i \in \{0, ..., 2^s - 1\}\} = \mathbb{K}(2^s)$ (poiché tutte le classi di equivalenza hanno un rappresentante della forma $f(i)$ con $i \in \{0, ..., 2^s - 1\}$).

Per la proprietà 1 vale che

$$\{r(f(i), 2^s) \mid i \in \{0, ..., 2^s - 1\}\} = \{r(f(j), 2^s) \mid j = 2^{s+1} - i - 1,\ i \in \{0, ..., 2^s - 1\}\}$$

Passiamo ora al modulo 2^{s+1}. Consideriamo l'insieme $A = \{r(f(i), 2^{s+1}) \mid i \in \{0, ..., 2^{s+1} - 1\}\}$. Per concludere l'induzione dobbiamo dimostrare che $A = \mathbb{K}(2^{s+1})$.
Cominciamo dividendo A nell'unione dei due seguenti insiemi:

$$A = \{r(f(i), 2^{s+1}) \mid i \in \{0, ..., 2^s - 1\}\} \cup \{r(f(i), 2^{s+1}) \mid i \in \{2^s, ..., 2^{s+1} - 1\}\}$$

Osserviamo che, a costo di inserire un nuovo indice j, il secondo insieme dell'unione può essere riscritto come

$$\{r(f(j), 2^{s+1}) \mid j = 2^{s+1} - i - 1,\ i \in \{0, ..., 2^s - 1\}\}$$

Risulta ovvio adesso che gli elementi del primo dei due insiemi che stiamo unendo sono in corrispondenza biunivoca con gli elementi del secondo dei due insiemi, facendo corrispondere ad ogni $r(f(i), 2^{s+1})$ l'elemento $r(f(j), 2^{s+1})$ con $j = 2^{s+1} - i - 1$.
Per la proprietà 2 sappiamo che, per ogni $i \in \{0, ..., 2^s - 1\}$ e $j = 2^{s+1} - i - 1$

$$r(f(i), 2^{s+1}) = r(f(i), 2^s) \Rightarrow r(f(j), 2^{s+1}) = r(f(j), 2^s) + 2^s$$

e

$$r(f(i), 2^{s+1}) = r(f(i), 2^s) + 2^s \Rightarrow r(f(j), 2^{s+1}) = r(f(j), 2^s)$$

Vale quindi, per i e j come sopra, che

$$\{r(f(i), 2^{s+1}), r(f(j), 2^{s+1})\} = \{r(f(i), 2^s), r(f(i), 2^s) + 2^s\}$$

Quindi, raggruppando gli elementi corrispondenti, vale

$$A = \bigcup_{i=0}^{2^s - 1} \{r(f(i), 2^{s+1}), r(f(j), 2^{s+1})\} = \bigcup_{i=0}^{2^s - 1} \{r(f(i), 2^s), r(f(i), 2^s) + 2^s\}$$

Per ipotesi induttiva, al variare di i da 0 a $2^s - 1$, gli $r(f(i), 2^s)$ corrispondono a tutti i rappresentanti standard delle classi di equivalenza modulo 2^s. Possiamo quindi riscrivere l'unione come

$$A = \bigcup_{k \in \mathbb{K}(2^s)} \{k, k + 2^s\}$$

e per quanto detto all'inizio del passo induttivo, vale $A = \mathbb{K}(2^{s+1})$ □

Dimostriamo ora che per ogni $s \in \mathbb{N}$ valgono sia la proprietà 1 che la proprietà 2, in modo da rendere valida la dimostrazione. Per farlo, definiamo bene cosa intendiamo con "Proprietà 1" e "Proprietà 2" che abbiamo accennato nella spiegazione "a parole".

Proprietà 1: per ogni $s \in \mathbb{N}$ e per ogni $i \in \{0, ..., 2^s - 1\}$, detto $j = 2^{s+1} - i - 1$, vale $f(i) \equiv f(j) \mod 2^s$, ovvero $f(j) - f(i) = h2^s$ con $h \in \mathbb{Z}$.

Dimostrazione:

$$\begin{aligned} f(j) - f(i) &= f(2^{s+1} - i - 1) - f(i) = \\ &= \frac{(2^{s+1} - i - 1)(2^{s+1} - i)}{2} - \frac{(i)(i+1)}{2} = \\ &= \frac{2^{2(s+1)} - i2^{s+1} - i2^{s+1} + i^2 - 2^{s+1} + i}{2} - \frac{i^2 + i}{2} = \\ &= 2^{2s+1} - i2^{s+1} - 2^s + \frac{i^2 + i}{2} - \frac{i^2 + i}{2} = \\ &= 2^{2s+1} - i2^{s+1} - 2^s = \\ &= (2^{s+1} - 2i - 1)2^s, \text{ con } (2^{s+1} - 2i - 1) \in \mathbb{Z} \end{aligned}$$

□

Riutilizzeremo in futuro l'uguaglianza appena dimostrata:

$$f(j) = f(i) + (2^{s+1} - 2i - 1)2^s$$

Per la proprietà 2 introduciamo prima una notazione. Abbiamo già definito $r(n, k)$ come il resto della divisione Euclidea di n per k. Definiamo $q(n, k)$ come il quoziente della divisione Euclidea di n per k. Vale quindi che $n = r(n, k) + q(n, k)k$, con $0 \leq r(n, k) < k$ e, per l'esistenza della divisione Euclidea in $\mathbb{Z}$, $r(n, k)$ e $q(n, k)$ esistono e sono unici.

Osserviamo ora una proposizione: per ogni n, k, vale che

- $r(n, 2k) = r(n, k) \iff q(n, k)$ è pari
- $r(n, 2k) = r(n, k) + k \iff q(n, k)$ è dispari

La dimostrazione di questa proposizione è semplice:

- Se $r(n,2k) = r(n,k)$ allora
  $n = r(n,2k) + q(n,2k)2k = r(n,k) + (q(n,2k)\cdot 2)k$ e per unicità di $q(n,k)$ vale $q(n,k) = 2q(n,2k)$, quindi è pari.
  Se $q(n,k) = 2h$ allora
  $n = r(n,k) + q(n,k)k = r(n,k) + 2hk = r(n,k) + (h)2k$ e per unicità di $r(n,2k)$ vale $r(n,2k) = r(n,k)$.

- Se $r(n,2k) = r(n,k) + k$ allora
  $n = r(n,2k) + q(n,2k)2k = r(n,k) + k + (q(n,2k)\cdot 2)k = r(n,k) + k + (2q(n,2k)+1)k$ e per unicità di $q(n,k)$ vale $q(n,k) = 2q(n,2k)+1$, quindi è dispari.
  Se $q(n,k) = 2h+1$ allora
  $n = r(n,k) + q(n,k)k = r(n,k) + (2h+1)k = r(n,k) + k + (h)2k$ e per unicità di $r(n,2k)$ vale $r(n,2k) = r(n,k) + k$.

Dimostrata questa proposizione possiamo ora ridefinire e dimostrare la proprietà 2.

Proprietà 2: per ogni $s \in \mathbb{N}$ e per ogni $i \in \{0,...,2^s-1\}$, detto $j = 2^{s+1} - i - 1$, vale $q(f(i),2^s)$ pari $\iff$ $q(f(j),2^s)$ dispari (e viceversa $q(f(i),2^s)$ dispari $\iff$ $q(f(j),2^s)$ pari).

Dimostrazione 2:

$$\begin{aligned} f(j) &= f(i) + (2^{s+1} - 2i - 1)2^s = \\ &= r(f(i),2^s) + q(f(i),2^s)2^s + (2^{s+1} - 2i - 1) = \\ &= r(f(i),2^s) + 2^s \cdot (q(f(i),2^s) + 2^{s+1} - 2i - 1) \end{aligned}$$

Così facendo (oltre ad ottenere un'altra dimostrazione, per unicità di $r(f(j),2^s)$, dell'identità $r(f(i),2^s) = r(f(j),2^s)$), per unicità di $q(f(j),2^s)$ abbiamo che

$$q(f(j),2^s) = q(f(i),2^s) + 2^{s+1} - 2i - 1$$

ovvero $q(f(i),2^s)$ e $q(f(j),2^s)$ differiscono di un intero dispari. Allora vale che $q(f(i),2^s)$ pari $\iff$ $q(f(j),2^s)$ dispari (e viceversa $q(f(i),2^s)$ dispari $\iff$ $q(f(j),2^s)$ pari).

□