Appunti Algoritmi e Strutture Dati

Eugenio Militerno

01/06/2024

L'esame di ASD è uno di quelli che richiede *uno studio intenso ed accurato, comprendendo tante nozioni, tanti teoremi e richiedendo un grande ragionamento, spesso abbastanza complesso.*

Questi appunti **non sono sufficienti** a superare l'esame, ma servono come ripetizione e fungono da base per lo studio.

Gli appunti sono strutturati *seguendo le slide*, e possono mancare di alcune nozioni o piccole parti di argomenti, ma coprono tutto il programma.

Roadmap di supporto alla ripetizione: Roadmap ASD

Sommario

1 Notazioni Asintotiche

Le notazioni asintotiche servono ai programmatori per **definire e predire l'andamento dell'algoritmo in base alla taglia e alla forma dell'input**. In questo modo, si può definire e predire il comportamento di un algoritmo al variare dei dati di input che l'algoritmo avrà.
Ad esempio, non sappiamo se un algoritmo che ordina 5 dati, impiegandoci 10s, ordinerà 10 dati impiegandoci 20s, perchè:

1. Dipende da come sono ordinati inizialmente i dati
2. Dipende da come l'algoritmo confronta ed ordina i dati
3. Dipende da quali e quanti sono i dati.

Per questo motivo si usano le **Notazioni Asintotiche**, che sono 3:

1. Notazione O
2. Notazione Ω
3. Notazione Θ

Andiamo ora a vederle nel dettaglio.

1.1 Notazione O

La notazione O definisce **il limite superiore della funzione esaminata**. La dicitura $f(x) = O(g(x))$ indica che *i valori assunti da $g(x)$ saranno sempre maggiori dei valori assunti da $f(x)$,* ***a parità di x***

1.2 Notazione Ω

La notazione Ω definisce **il limite inferiore della funzione esaminata**. La dicitura $f(x) = \Omega(g(x))$ indica che *i valori assunti da $g(x)$ saranno sempre inferiori ai valori assunti da $f(x)$,* ***a parità di x.***

1.3 Notazione Θ

La notazione Θ definisce **il limite superiore ed inferiore della funzione esaminata**. La dicitura $f(x) = \Theta(g(x))$ indica che *i valori assunti da $c_1 * g(x)$ saranno sempre superiori ai valori assunti da $f(x)$, ed i valori assunti da $c_2 * g(x)$ saranno sempre inferiori ai valori assunti da $f(x)$,* ***a parità di x,*** $\exists(c_1, c_2) > 0$.

2 Proprietà fra le Notazioni

Le notazioni asintotiche godono di diverse proprietà:

1. Moltiplicazione per costante :
$f(n) = O(g(n)) \implies \lambda * f(n) = O(\lambda * g(n)) = O(g(n)), \forall \lambda > 0$

2. Additivita':
$f(n) = O(h(n)), g(n) = O(i(n)) \implies f(n) + g(n) = O(h(n)) + O(i(n)) = O(h(n) + i(n))$

3. Moltiplicazione:
$f(n) = O(h(n)), g(n) = O(i(n)) \implies f(n) * g(n) = O(h(n)) * O(i(n)) = O(h(n) * i(n))$

4. Transitivita':
$f(n) = O(g(n)) \wedge g(n) = O(h(n)) \implies f(n) = O(i(n))$

Tali proprieta' sono valide per *tutte* le notazioni.

3 Serie Numeriche

3.1 Serie Aritmetica - Gauss Bambino

$$\sum_{i=1}^{n} i = \frac{1}{2}\frac{n(n+1)}{2} = \Theta(n^2)$$

3.2 Serie dei Quadrati

$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)+(2n+1)}{6}\Theta(n^3)$$

3.3 Serie dei Cubi

$$\sum_{k=0}^{n} k^3 = \frac{n^2(n+2)^2}{4} = \Theta(n^4)$$

3.4 Serie Armonica

$$\sum_{k=1}^{n} \frac{1}{k} = \log_e(n) + O(1) = O(\log(n))$$

3.5 Serie Geometrica

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$$

3.6 Serie Geometrica Infinita

$$\sum_{k=1}^{\infty} x^k = \frac{1}{1-x}$$

3.7 Serie Geometrica Generale

$$\sum_{k=1}^{n} k * x^k = \frac{x}{(1-x)^2}$$

4 Analisi degli Algoritmi

4.1 Operazioni Basilari

L'analisi degli algoritmi si fonda su alcune nozioni di base, che definiamo **Operazioni Basilari**. Queste operazioni costano poco, in termini di tempo, perche' sono operazioni eseguibili in modo facile e rapido da parte del processore. Esse sono:

1. Assegnazione di un valore
   $a = 1$
2. Confronto fra valori
   $a > b$
3. Spostamento di un puntatore
   $i++$
4. Addizione e Moltiplicazione **fra numeri piccoli**
   $1 + 1 * 2$

Queste operazioni, per convenzione, hanno costo $O(1)$. Esse infatti vengono considerate come *eseguibili immediatamente* dal compilatore, nonostante richiedano comunque un piccolo tempo (per le operazioni di **Fetch, Decode, Execute**).

4.2 Cicli

I cicli hanno costo pari al **Costo delle operazioni svolte al loro interno**, per **Il numero di volte che vengono eseguite**. Dunque, basta svolgere il calcolo $\sum n * c_i$, dove n e' il numero di operazioni totali, mentre c_i e' il costo di ciascuna operazione. Per fare un esempio:

```
int a=0;
for(int i=0;i<4;i++){
    a++;
}
```

Questo codice verra' eseguito 4 volte, per i che va da 0 a 3. Ad ogni iterazione, aumentiamo di 1 il valore di a, che inizialmente e' 0. Ergo, deduciamo che: $T(n) = 4 * O(1) = O(4)$, che e' costante, *in quanto abbiamo un numero finito di operazioni*, e possiamo dire che $T(n) = O(1)$. Chiaramente, **se non sapessimo quante volte viene eseguito il ciclo for**, dovremmo dire che $T(n) = n * O(1) = O(n)$.

Stesso ragionamento vale per il ciclo **while**, dove abbiamo una condizione d'uscita/di arresto, che viene valutata a **TRUE** solo dopo n iterazioni, ma spesso non sappiamo quanto valga n.
Il ciclo **DO/WHILE** invece esegue prima il codice, e poi controlla se la condizione e' rispettata o meno, quindi esegue le stesse operazioni del ciclo **WHILE**, ma una volta in piu' (*ovvero, viene eseguito* ***almeno*** *una volta*).

4.3 Costrutti Decisionali

I costrutti decisionali sono molto semplici da capire: **il loro costo equivale al costo dell'opzione piu' costosa, piu' 1 per il confronto**. Questo vale per ciascun costrutto, per cui *se abbiamo piu' cicli innestati*,dovremo ragionare costrutto per costrutto. Per fare un esempio:

```
int a=0;
int b=1;
if(a>b){
    cout << "a e' maggiore di b" << endl;
}else{
    cout << "b e' maggiore di a" << endl;
}
```

In questo codice, abbiamo un if ed un else, quindi abbiamo due costrutti che costano $O(1)$ ciascuno. Non verranno mai eseguiti entrambi insieme, dunque avremo al massimo un controllo. Mentre all'interno di ciascuno, troviamo una stampa, che costa $O(1)$. Dunque avremo: $T(n) = 1 + O(1) + O(1) = 3 * O(1) = O(1)$.

4.4 Altri fattori

Quando si analizza un algoritmo, bisogna tenere conto anche di altri fattori: **taglia** e **forma** dell'input.
La *Taglia* e' la dimensione del problema iniziale, ovvero, in parole povere, il *quanto e' grande*; mentre la *Forma* e' la struttura del problema, ovvero, in parole povere, il *come e' strutturato*.
Potremmo dire che l'ordinamento di dati e' sempre uguale, e quindi avra' sempre lo stesso costo, usando lo stesso algoritmo, ma bisogna ragionare anche sul fatto che **esistono diversi tipi di dati**, che **i dati stessi potrebbero avere un ordinamento iniziale diverso**, oppure che **potrebbero esserci dati che non possono essere ordinati naturalmente**. Ad esempio, ordinare **delle stringhe** e' diverso dall'ordinare **delle struct**, come lo e' dall'ordinare **dei float**.

Per la **taglia** dell'input, solitamente ci riferiremo al **Criterio di Costo Uniforme**, ovvero intendiamo come taglia *il numero di elementi che compongono l'input del problema*, mentre per la **forma** analizzeremo tre casi: **Peggiore,Medio e Migliore**, in base alla struttura dell'input

5 Ricorrenze

Le ricorrene esprimono l'andamento, e quindi la *complessita'* di un algoritmo, mediante il riferimento alla complessita' **dei loro sottoproblemi**. Le ricorrenze sono infatti **ricorsive**, indi per cui vengono usate (*e generate*) solamente sugli algoritmi che si richiamano ricorsivamente sui sottoproblemi per risolverli.
Le ricorrenze sono della forma:
$T(n) = a * T(\frac{n}{b}) + f(n)$ dove $a, b > 1 \wedge f(n)$ *nota*
Esistono 4 metodi per risolvere le ricorrenze:

1. Metodo **Iterativo**
2. Metodo di **Sostituzione**
3. Metodo dell'**Albero di Ricorsione**
4. **Master Theorem**

che adesso vediamo nel dettaglio.

5.1 Metodo Iterativo

Il metodo Iterativo consiste nello **srotolare** la ricorrenza, fino a riconoscere una *successione* di valori, che possiamo ricondurre quindi ad una **serie nota**, oppure riusciamo a risolverla e ottenerne il risultato.
Un esempio e' il seguente:

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + 1 \\ &= T(\frac{n}{4}) + 1 + 1 \\ &= T(\frac{n}{2^k}) + \sum(1) \end{aligned} \tag{1}$$

Tale ricorrenza si fermera' quando $\frac{n}{2^k} = 1$, cioe' quando $k = log(n)$, e dunque avremo che la sua soluzione e' $T(n) = O(logn)$.

Vediamo adesso un altro esempio:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + n \\ &= 2(2T(\frac{n}{4} + \frac{n}{2}) + n \\ &= 4T(\frac{n}{4}) + 2n \\ &= ... \\ &= 2^k * T(\frac{n}{2^k}) + k * n \end{aligned} \tag{2}$$

La ricorrenza si arresterà quando $2^k = 1 \implies k = \log(n)$. Quindi avremo che $T(n) = 2^{\log(n)}T(1) + n\log(n)$, che implica $T(n) = n * O(1) + n\log(n)$, e dunque $T(n) = O(n\log(n))$

5.2 Metodo di Sostituzione

Il metodo di Sostituzione si basa sull'**indovinare una soluzione approssimativa alla ricorrenza**, in base a *ricorrenze gia' svolte, o viste*, e dimostrarne la veridicita' mediante calcoli matematici.
In termini matematici, dobbiamo dimostrare che $\exists c > 0, n_0 : T(n) =< soluzione >$, dove *soluzione* puo' essere $O(g(n)) \vee \Theta(g(n)) \vee \Omega(g(n))$
Un esempio e' il seguente:
$T(n) = T(n/2) + n$
Suppongo $T(n) = O(n * \log(n))$, dato che assomiglia alla ricorrenza del Merge Sort.

$$\begin{aligned} T(n) &= O(n\log(n))T(n) \\ &\leq c * n * \log(\frac{n}{2}) + n \\ &\leq c * n * (\log(n) - \log(2)) + n \\ &\leq c * n * (\log(n) - 1) + n \\ &\leq c * n * \log(n) - c * n + n \\ &\leq c * n * \log(n) + n(1 - c) \end{aligned} \tag{3}$$

Affinche' questa disequazione sia vera, deve valere $1 - c > 0$. Dimostrabile facilmente per $c < 1$, ma dovendo essere $c > 0 \implies 0 < c < 1$, che conferma la nostra ipotesi.

Vediamo un altro esempio:

$$T(n) = T(\frac{n}{2}) + n \tag{4}$$

Ipotizziamo che vale $T(n) = O(n)$
si ha che:

$$\begin{aligned} T(n) &\leq c\frac{n}{2} + n \\ &\leq n * (\frac{c}{2} + 1) \end{aligned} \tag{5}$$

Questa disequazione è valida solo se $c \geq 2$

5.3 Metodo dell'Albero di Ricorsione

Il metodo dell'albero richiede di svolgere **l'albero della ricorsione** della ricorrenza, e, una volta costruito tutto l'albero, permette di calcolare il costo della ricorrenza in base al numero di nodi ed il loro costo, **che chiaramente poi andra' verificata mediante *il metodo di Sostituzione***.
NOTA BENE:
Un albero si dice:

> **BILANCIATO** se, *per ogni livello* k, la somma dei costi di ciascun nodo al livello k e' costante.
>
> **SBILANCIATO** altrimenti.

Esiste un trucco per capire subito il risultato della ricorrenza. Andiamo a vedere subito un esempio:

$$T(n) = 2T(\frac{n}{2}) + \log(n) \tag{6}$$

Notiamo subito che la $f(n) = \log(n)$. Il log non e' un polinomio, e dunque la somma dei suoi valori nei vari livelli non avra' mai lo stesso valore, ergo **l'albero e' sbilanciato**.
Se invece di $\log(n)$ avessimo avuto n, ovvero la seguente ricorrenza: $T(n) = 2T(\frac{n}{2}) + n$, avremmo facilmente notato che **ad ogni livello, la somma dei costi sarebbe rimasta costante**, e quindi *l'albero sarebbe stato bilanciato*. **Chiaramente, bisogna anche fare attenzione al numero di rami dell'albero**. Potremmo anche avere $T(n) = 2T(n/3) + n$, che però non è bilanciato, in quanto avremmo *3 sottoproblemi, ciascuno di dimensione* $\frac{n}{3}$, e due rami. La somma del livello risulta quindi $\frac{n}{3} * 2 \neq n$, per cui l'albero **è sbilanciato**.
Interviene quindi qui la *famosa* **Regola del Pollice**:

> 1. Se l'albero di ricorsione e' **Bilanciato**, il costo della ricorrenza e' $T(n) = \Theta(f(n) * \log(n))$
> 2. Se l'albero di ricorsione e' **Sbilanciato**, il costo della ricorrenza e' $T(n) = \Theta(f(n))$

Questa regola e' facilmente dimostrabile svolgendo il classico calcolo del costo dell'albero: $T(n) = \sum c_i * n_i$, ovvero somma del prodotto di numero di nodi del livello per costo di ciascuno.

5.4 Master Theorem

Il Master Theorem e' un teorema che ci fornisce direttamente la soluzione ad una ricorrenza, previa verifica di alcune condizioni.
Il MT risolve le ricorrenze della forma: $T(n) = a * T(\frac{n}{b}) + f(n)$.
Esistono 4 casi:

1. $f(n) = O(n^{\log_b(a)-\epsilon}) \implies T(n) = \Theta(n^{\log_b(a)}), \iff \epsilon > 0$
2. $f(n) = \Theta(n^{\log_b(a)}) \implies T(n) = \Theta(n^{\log_b(a)} * \log(n))$
3. $f(n) = \Omega(n^{\log_b(a)+\epsilon}) \wedge \exists c < 1 : a * f(\frac{n}{b}) \leq c * f(n) \implies T(n) = \Theta(f(n))$
4. (**BONUS**) $f(n) = \Theta(f(n) * \log^k(n)) \implies T(n) = \Theta(f(n) * \log^{k+1}(n))$

La risoluzione della ricorrenza dipende quindi unicamente dal comportamento di $n^{\log_b(a)}$ in relazione alla $f(n)$, ed eventualmente ad un termine $log^k(n)$, se la differenza di grandezza fra $f(n)$ e $n^{\log_b(a)}$ *non è polinomiale*
Vediamo alcuni esempi:

$$T(n) = 3T(\frac{n}{9}) + n \tag{7}$$

Secondo il MT, abbiamo $a = 3$ e $b = 9$, quindi $n^{\log_b(a)} = n^{\log_9(3)} = n^{\frac{1}{2}}$
Facendo quindi il paragone:

$$\begin{aligned} f(n) &= \Theta(n^{\log_b(a)}) \\ &= \Theta(n^{\frac{1}{2}}) \end{aligned} \tag{8}$$

Siamo quindi nel 3° caso, dobbiamo dimostrare che

$$\begin{aligned} & a * (f(\frac{n}{b})) \\ & \leq c * f(n) \\ & \implies 3(\frac{n}{9}) \\ & \leq c * n \\ & \implies \frac{n}{3} \\ & \leq c * n \\ & \implies c \\ & \geq \frac{1}{3} \end{aligned} \tag{9}$$

6 Algoritmi di Ordinamento

Gli algoritmi di ordinamento si dividono in base a:

1. **Complessita'**
2. **Funzionamento**
3. **Strutture ausiliarie**

6.1 Algoritmi Polinomiali

I primi algoritmi che andremo a vedere sono gli algoritmi che operano in tempo **Polinomiale**, ovvero un tempo $T(n) = O(n^k)$. Essi sono:

1. **Bubble Sort**
2. **Insertion Sort**
3. **Quick Sort**

Questi algoritmi *scorrono tutto l'input, non sfruttano alcuna proprieta' dell'input ne' di strutture dati*, e hanno complessita' temporale che **dipende molto dalla forma dell'input stesso**.

6.1.1 Bubble Sort

Il Bubble e' l'algoritmo di ordinamento *meno efficiente di tutti*, ed infatti e' anche il piu' odiato e il meno usato. Esso scorre tutto l'array, e confronta **uno per uno** tutti gli elementi, ordinandoli nel caso in cui *un elemento ed il successivo siano in posizioni sbagliate*.

```
void bubbleSort(int arr)
    for (i<-1 to n)
        for (j<-1 to n)
            if (arr[j] > arr[j + 1])
                swap(arr[j], arr[j + 1])
```

La complessita' di questo algoritmo e' $T(n) = O(n^2)$, per ognuno dei 3 casi.

6.1.2 Insertion Sort

L'Insertion e' **di poco migliore** rispetto al Bubble, perche' a differenza del precedente, non scorre **sempre due volte l'array**, ma *puo' fermarsi prima di fare un doppio scorrimento* (Ciò è dovuto dal fatto che viene usato il *while*, che non itera un numero di volte prefissate). Chiaramente, quanti elementi scorre per volta **dipende dalla forma dell'array di input**.

```
void insertionSort(int arr[])
    for (i<-2 to n)
        key<-arr[i]
        j<-i - 1;
        while (j>=1 AND arr[j] > key)
            arr[j + 1] = arr[j]
            j = j - 1
        arr[j + 1] = key;
```

L'algoritmo controlla ciascun elemento dell'array, e per ciascuno controlla se gli elementi **precedenti** sono maggiori. In tal caso, *sposta a destra* tutti gli elementi maggiori della key (ovvero dell'elemento analizzato), e poi inserisce key nel primo spazio vuoto che trova.
E' importante notare che **questo algoritmo non effettua scambi, ma piuttosto *sposta ciascun elemento finche' non trova la posizione appropriata per l'elemento key***.
La complessita' di questo algoritmo e':

Caso Migliore: $T(n) = O(n)$

Caso Medio: $T(n) = O(n^2)$

Caso Peggiore: $T(n) = O(n^2)$

La complessita' nel caso migliore, ovvero quando **gli elementi dell'input sono gia' ordinati**, e' $O(n)$ perche' **non verra' mai eseguito il while**.

6.1.3 Complessita'

1.	For j ←2 to Length(A)	$c_1 n$
2.	Key ←A[j]	$c_2(n-1)$
3.	i← j-1	$c_3(n-1)$
4.	While (i> 0 and A[i] > key)	c_4
5.	Do A[i+1]←A[i]	c_5
6.	i←i-1	c_6
7.	A[i+1]←key	$c_7(n-1)$

Caso migliore:
Se l'input e' gia' ordinato, il loop del WHILE non verra' mai eseguito, portando ad una complessita' di:

$$T(n) = c_1 * n + (c_2 + c_3 + c_7) * (n-1) = O(n)$$

Caso Peggiore:
Il caso peggiore si ha quando l'input e' *ordinato in modo decrescente*. In questo caso la complessita' e' pari a:

$$T(n) = P(n) + c_4 * \sum_{j=2}^{n} j + (c_5 + c_6) * \sum_{j=2}^{n} (j-1)$$

Ma si ha che:

$$\sum_{j=2}^{n} j + 1 - 1 = \sum_{j=1}^{n} (j-1) = \frac{n(n+1)}{2} - 1 \text{[\textbf{Sommatoria di Gauss Bambino}]}$$

$$\sum_{j=2}^{n} (j-1) = \sum_{j=2}^{n} j - \sum_{j=2}^{n} 1 = \frac{n(n+1)}{2} - 1 - n + 1 = \frac{n(n-1)}{2}$$

Avremo quindi che: $T(n) = P(n) + c_4 * (\frac{n(n+1)}{2}) + (c_5 + c_6) * \frac{n(n-1)}{2} = P(n) + \Theta(n^2) = \Theta(n^2)$
Caso Medio:
Il caso medio si ha quando l'array e' parzialmente ordinato, oppure se gli elementi sono disposti in ordine casuale. In questo caso vale:

$$T(n) = P(n) + c_4 * \sum_{j=2}^{n} \frac{j}{2} + (c_5 + c_6) * \sum_{j=2}^{n} \frac{j-1}{2}$$

Ma si ha che:

$$\frac{1}{2} \sum_{j=2}^{n} j + 1 - 1 = \frac{1}{2} (\frac{n(n+1)}{2} - 1)$$

$$\frac{1}{2}\sum_{j=2}^{n} j - 1 = \sum_{j=2}^{n} j - \sum_{j=2}^{n} 1 = \frac{1}{2} * \frac{n(n-1)}{2}$$

Che risulta comunque $P(n) + \Theta(n^2) = \Theta(n^2)$

INVARIANTE DI CICLO

All'inizio di ciascuna iterazione del ciclo **for**, il sottoarray $A[1 \ldots j-1]$ e' ordinato, ed e' formato *dagli stessi elementi che erano originariamente in* $A[1 \ldots j-1]$

INIZIALIZZAZIONE

Prima della prima iterazione ($j = 2$), osserviamo che $A[1 \ldots j-1] = A[1]$, che per definizione e' ordinato, ed e' l'unico elemento originariamente presente in $A[1]$.

CONSERVAZIONE

Il corpo del for opera *spostando ogni elemento di una posizione verso destra*, finche' non trovera' la posizione appropriata per $A[j]$. Il sottoarray $A[1 \ldots j]$ e' ordinato, ed e' formato dagli stessi elementi che originariamente erano in $A[1 \ldots j]$. Percio', l'incremento di j per la successiva iterazione **preserva l'invariante**.

CONCLUSIONE

La condizione che determina la conclusione del for e' $j > A.length = n$. Poiche' ogni iterazione aumenta j di 1, alla fine del ciclo si ha che $j = n + 1$.
Sostituendo j con $n + 1$ nella formulazione dell'invariante, otteniamo che $A[1 \ldots n]$ e' formato dagli elementi *ordinati* che originariamente erano presenti in $A[1 \ldots n]$, che pero' era l'array originale, e dunque l'array e' ordinato. **Pertanto, l'algoritmo e' corretto**.

6.1.4 Quick Sort

Il QuickSort e' l'algoritmo che *generalmente* ha le prestazioni migliori fra quelli basati su ordinamento. E' abbastanza efficiente per tutti i tipi di input, ma ha la pecca di **non essere stabile**, ovvero *gli elementi dell'input possono non comparire nello stesso ordine nell'output*, e di avere **complessita' quadratica nel caso medio/peggiore**.

```
void QuickSort(int a[], int p, int r)
    if(p<r)
        int q<-Partition(a,p,r)
        QuickSort(a,p,q)
        QuickSort(a,q+1,r)

int Partition(int a[],int p, int r)
    int x = a[p]
    int i = p-1
    int j = r+1
    while(true)
        repeat
            i<-i+1
        until(a[i]<=x)
        repeat
            j<-j-1
        until(a[j]>=x)
        if(i<j)
            swap(a[i],a[j])
        else
            return j
```

La complessita' del QuickSort e':

Caso peggiore: $T(n) = O(n^2)$

Caso medio: $T(n) = O(n^2)$

Caso migliore: $T(n) = 2T(n/2) + \Theta(n) = O(n * \log(n))$

Si ha complessita' $O(n * \log(n))$ per ogni partizione a proporzionalita' costante (99 a 1, 5 a 1,...)

INVARIANTE DI CICLO

All'inizio di ogni iterazione del ciclo si ha che:

1. Se $p \leq k \leq i \implies A[k] \leq x$
2. Se $i+1 \leq k \leq j-1 \implies A[k] > x$
3. Se $k = r \implies A[k] = x$

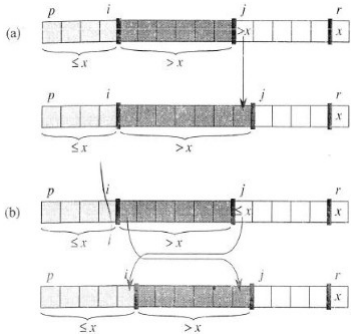
Gli indici fra j ed $r-1$ non rientrano in alcuno di questi tre casi, e non hanno una particolare relazione con il pivot x.

INIZIALIZZAZIONE

Prima della prima iterazione del ciclo, $i = p-1$ e $j = p$. Non ci sono valori fra p ed i, ne' fra $i+1$ e $j-1$, quindi le prime due condizioni sono verificate.
La terza condizione e' assicurata dalla riga 1.

CONSERVAZIONE

Ci sono due casi da considerare, a seconda del test della riga 4:



Quando $A[j] > x$, si ricade nel caso a. L'unica azione e' *incrementare* j. Dopo l'incremento, la condizione 2 e' soddisfatta per $A[j-1]$ e tutte le altre posizioni non cambiano.
Nell'altro caso, vengono scambiati $A[i]$ ed $A[j]$, viene incrementato i, e poi j. Dopo lo scambio, abbiamo che $A[i] \leq x$ e la condizione 1 e' soddisfatta, ma si ha anche che $A[j-1] > x$, in quanto l'elemento spostato e' maggiore di x.

CONCLUSIONE

Alla fine del ciclo $j = r$. Pertanto, ogni posizione dell'array si trova in uno dei tre insiemi descritti dall'invariante, e noi abbiamo ripartito i valori dell'array in tre insiemi:

1. I valori minori o uguali ad x
2. I valori maggiori di x

3. L'elemento x

6.2 Valore Atteso

In teoria della probabilita' il *valore atteso* di una *variabile aleatoria*, e' un numero, indicato con $E[x]$. Esso e' dato dalla **somma dei possibili valori** di tale variabile, ognuno **moltiplicato per la probabilita' che sia assunto**, ovvero e' **la media ponderata dei possibili risultati**.
In matematica, si scrive che *il valore atteso di una variabile aleatoria* ***continua*** *che ammette come funzione di probabilita'* $f(x)$ e':

$$E[x] = \int_{-\infty}^{+\infty} x * f(x)dx$$

Nel caso di una variabile aleatoria **discreta**, che ha funzione di probabilita' p_i, si ha:

$$E[x] = \sum_{i=1}^{+\infty} x_i * p_i$$

La media e' un caso particolare del valore atteso, che si ottiene quando il numero di valori e' N, e la probabilita' per ogni valore e' $p_i = \frac{1}{N}$

6.2.1 Proprieta' Valore Atteso

COSTANTI

Il valore atteso di una variabile aleatoria **costante** e' la costante stessa:
$E[c] = c$

LINEARITA'

$$E[\alpha * X + \beta * Y] = \alpha * E[X] + \beta * E[Y]$$

α, β costanti e X, Y variabili aleatorie

6.2.2 Tempo d'esecuzione Atteso

QuickSort e Randomized-Quicksort sono diverse solo nel modo in cui viene scelto il pivot, per cui analizzeremo Partition e Quicksort supponendo che ogni volta il pivot sia scelto in modo casuale.
Il tempo d'esecuzione della procedura Quicksort e' dominato dal tempo d'esecuzione della procedura Partition. Ogni volta che viene selezionato un pivot, esso non sara' mai incluso nelle successive chiamate ricorsive di Quicksort e Partition.
Ci saranno quindi, *al piu'*, n chiamate ricorsive di Partition. Una chiamata a Partition impiega $O(1)$ piu' un tempo proporzionale al numero di iterazioni del ciclo for (*rr 3-6*).
Ogni iterazione di questo ciclo effettua un confronto tra il pivot ed un elemento dell'array, pertanto vale:

LEMMA

Se X e' il numero di confronti svolti nella riga 4 di Partition nell'intera esecuzione di Quicksort, su un array di n elementi, allora il tempo d'esecuzione di Quicksort e' $O(n + X)$

Non calcoleremo *esattamente* X, ma deriveremo un limite superiore sul numero **globale** dei confronti. Abbiamo pertanto necessita' di capire *quando viene effettuato un confronto fra due elementi dell'array.*
Rinominiamo $z_1, \ldots, z_n$ gli elementi dell'array, dove z_i e' l'i-simo elemento piu' piccolo. Definiamo $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ l'insieme degli elementi compresi fra z_i e z_j.
Utilizzando una variabile indicatrice, definiamo $X_{ij} = I\{z_i \; confrontato \; con \; z_j\}$.
Stiamo considerando *tutti i confronti fatti dall'algoritmo*, **non solo durante un'iterazione o una chiamata di** Partition.
Poiche' ogni coppia viene confrontata al piu' una sola volta, il numero totale dei confronti e' dato da:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

Prendendo i valori attesi ambo i lati, si ha:

$$E[X] = E[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$$

Dove $E[X_{ij}]$ e' dato dalla **probabilita' con cui z_i sia confrontato con z_j**.
La nostra analisi suppone che **ogni pivot sia scelto in modo casuale ed indipendente**. E' quindi utile riflettere su *quando due elementi non sono confrontati.*
In generale, ogni volta che viene scelto un pivot x, con $z_i < x < z_j$, sappiamo che z_i e z_j *non saranno confrontati in un istante successivo.*

Se viene scelto z_i come pivot, prima di qualsiasi elemento di Z_{ij}, sara' confrontato *con tutti gli elementi tranne che con se stesso.*
Quindi z_i e z_j saranno confrontati **se e solo se il primo termine ad essere scelto e'** z_i **o** z_j. Poiche' Z_{ij} ha $j - i + 1$ elementi, la probabilita' che qualsiasi elemento sia scelto per primo come pivot e': $\frac{1}{j-i+1}$.
Pertanto, la probabilita' di confronto fra z_i e z_j e': $E[X_{ij}] = \frac{2}{j-i+1}$.
Sostituendo, otteniamo:

$$E[X] = \sum_{i=1}^{n} -1 \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$E[X] = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

$$E[X] < \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} < \sum_{i=1}^{n-1} \log(n)$$

$$E[X] = \sum_{i=1}^{n-1} O(\log(n)) = O(n \log(n))$$

Pertanto, il RANDOMIZED-QUICKSORT ha un tempo d'esecuzione atteso di $O(n \log(n))$, quando i valori degli elementi sono **distinti**.

6.3 Algoritmi Logaritmici

Gli algoritmi Logaritmici sono quelli che non hanno complessita' polinomiale, ma solamente logaritmica. Essi sono:

1. HeapSort
2. Merge Sort

6.3.1 Heap Sort

L'HeapSort e' un algoritmo di ordinamento che opera **basandosi sulla struttura dati Heap**. L'heap e' un **Albero Binario Completo**, e rispetta, *per ogni nodo*, la proprieta':

$$1 \text{ - } \textbf{Max-Heap} : A[Parent(i)] \geq A[i]$$
$$2 \text{ - } \textbf{Min-Heap} : A[Parent(i)] \leq A[i]$$

Negli Heap, abbiamo le seguenti funzioni per calcolare padre e ciascuno dei figli, di ogni nodo: $Parent(i) = \frac{i}{2}$, $Left(i) = 2 * i + 1$, $Right(i) = 2 * i + 2$.
Per creare un Heap (*per semplicita', assumiamo di lavorare con un Max-Heap*), si dispone delle seguenti funzioni:

```
Build_Max_Heap(int A[])
    A.heapsize<-A.length
    for(i<-A.length/2 downto 1)
        Max_Heapify(A,i)
```

INVARIANTE DI CICLO

All'inizio di ogni iterazione del ciclo FOR, ogni nodo $i+1, \ldots, n$ e' la **radice di un Max-Heap**.

INIZIALIZZAZIONE

Prima della prima iterazione del ciclo, $i = \lfloor \frac{n}{2} \rfloor$. Ogni nodo $\lfloor \frac{n}{2} \rfloor + 1, \ldots, n$ e' una foglia, e quindi **una radice di un Max-Heap *banale***.

CONSERVAZIONE

Per verificare che ogni iterazione conserva l'invariante di ciclo, notiamo che *i figli del nodo i hanno una numerazione **piu' alta di** i*. Per l'invariante di ciclo, sono entrambi radici di Max-Heap. E' la condizione richiesta affinche' la chiamata a MAX-HEAPIFY renda il nodo i la radice di un Max-Heap.
Inoltre, la chiamata MAX-HEAPIFY *preserva la proprieta' che tutti i nodi* $i+1, \ldots, n$ *siano radici di Max-Heap.*
La diminuzione di i nell'aggiornamento del ciclo ristabilisce l'invariante per la successiva iterazione.

CONCLUSIONE

Alla fine del ciclo, $i = 0$. Per l'invariante di ciclo, ogni nodi $1, 2, \ldots, n$ e' la radice di un Max-Heap, ed in particolare, **lo e' il nodo 1**.

```
Max_Heapify(int A[],int i)
    int l<-Left(i)
    int r<-Right(i)
    if(A[i]< A[l] && l<=A.heapsize)
        max<-l
    else
        max<-i
    if(A[i]<A[r] && r<=A.heapsize)
        max<-r
    if(max!=i)
        swap(A[max],A[i])
        Max_Heapify(A,i)
```

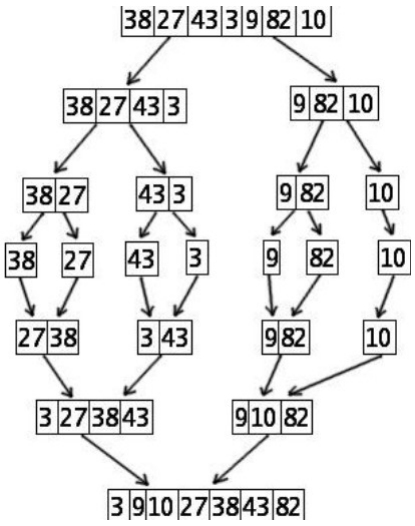
La Max-Heapify è la funzione che *ordina l'array, in base alle proprietà dell'Heap.* Inizialmente, per ogni nodo, otteniamo il *figlio sinistro* e il *figlio destro.* Successivamente andiamo a confrontare il padre con i figli, e se devono essere scambiati, segniamo come *max* il nodo con chiave maggiore (*l'indice dell'elemento maggiore nell'array*).
Infine, se il massimo non è il nodo i, si procede a scambiare i due valori, e *si richiama la* $Max_Heapify$ *sul nodo scambiato* i, per verificare che le proprietà dell'Heap vengano rispettate.
La $Build_Max_Heap$ si limita a *costruire un Max-Heap* dato un array. Inizia da **metà** dell'array, perchè oltre la metà dell'array avremo nodi foglia, e per ciascuno dei nodi *fino alla radice*, richiama la Max-Heapify.
Il costo della Max-Heapify e' $T(n) = T(2/3n) + \Theta(1) = \Theta(n * \log(n))$, mentre quello della Build-Max-Heap e' $O(n)$.
Vediamo adesso l'HeapSort:

```
HeapSort(int A[], int size){
    Build_Max_Heap(A)
    for(j=A.size downto 1)
        swap(A[1],A[j])
        A.heapsize<-A.heapsize-1
        Max_Heapify(A,1)
```

L'HeapSort procede a costruire il Max-Heap mediante la *Build˙Max˙Heap*, e successivamente procede in questo modo:
1) Scambia il primo elemento con il j-simo, perchè il massimo nel Max-Heap si trova nel 1° elemento.
2) Riduce l'heapsize di A, perchè operiamo su un array di dimensione ridotta (*avendo scambiato il primo e l'ultimo elemento,non dobbiamo ordinare tutto l'array, ma un elemento in meno*)

3) Richiama $Max_Heapify$ sul primo elemento, per far rispettare le proprietà del Max-Heap.
La complessita' dell'HeapSort e' $T(n) = O(n * \log(n))$, invocando **Max_Heapify** n volte

6.3.2 Merge Sort



Il Merge opera **suddividendo il problema iniziale in sottoproblemi**, finche' non riduce il problema ad un'istanza singola (*ovvero, un singolo elemento*). In quel momento, riordina l'array, inserendo elemento per elemento, *in base a quale e' minore*, ed abbiamo quindi l'intero array ordinato.

```
merge(arr[],left,mid,right)
    n1<-mid-left+1
    n2<-right-mid
    L<-new array[n1]
    R <-new array[n2]
    i<-0,j<-0
    for(i<-1 to n1)
        L[i]=A[left+i-1]
    for(j<-1 to n2)
        R[j]=A[mid+j]
    L[n1]<- +INF
    R[n2]<- +INF
    i<-1
    j<-1
    for(k<-left to right)
        if L[i] <= R[j]
            A[k]<-L[i]
            i++
        else
            A[k]<-R[j]
```

```
            j++
void mergeSort(int arr[], int temp[], int left, int right)
    if (left < right)
        mid<-[(left+right)/2]
        Merge-Sort(A,left,mid)
        Merge-Sort(A,mid+1,right)
        Merge(A,left,mid,right)
    }
}
```

Dividendo ogni volta l'array a meta', otteniamo una sequenza di sotto-array, che puo' essere riscritta come **un albero**, dove la radice e' l'array iniziale, e ogni figlio e' l'array con dimensione del padre divisa per due. Essendo un albero, e dividendo la dimensione ogni volta per due, otteniamo che le foglie, ovvero *gli array di elementi singoli*, saranno a profondita' $2^n = 1$, che, applicando il log, diventa $log_2(n)$. Chiaramente, il MergeSort viene invocato n volte, per cui la complessita' sara' $T(n) = O(n * \log(n))$.
La complessita' del Merge Sort, *essendo un algoritmo* ***ricorsivo***, va espressa mediante una **equazione di ricorrenza**, ovvero la seguente: $T(n) = 2T(\frac{n}{2} + 1) = O(n * \log(n))$

INVARIANTE DI CICLO

All'inizio di ogni iterazione del for, il sottoarray $A[p \dots k-1]$ contiene, *ordinati*, i $k - p$ elementi piu' piccoli di $L[1 \dots n_1 + 1]$ e $R[1 \dots n_2 + 1]$. Inoltre, $L[i]$ ed $R[j]$ sono *i piu' piccoli elementi dei rispettivi array* ***non ancora copiati in A***.

INIZIALIZZAZIONE

Prima dell'iterazione di ogni ciclo, abbiamo $k = p$, quindi il sottoarray $A[p \dots k-1]$ *e' vuoto*, e contiene i $k - p = 0$ elementi piu' piccoli di L ed R. Inoltre, poiche' $i = j = 1$, $L[i]$ ed $R[j]$ sono i piu' piccoli elementi degli array che non sono stati ancora copiati in A.

CONSERVAZIONE

Supponiamo che $L[i] \leq R[j]$, quindi $L[i]$ e' l'elemento piu' piccolo non ancora copiato in A.
Poiche' $A[p \dots k-1]$ contiene i $k - p$ elementi piu' piccoli, dopo la copia in A avremo in L i $k - p + 1$ elementi piu' piccoli. Incrementando k ed i si ristabilisce l'invariante di ciclo per la prossima iterazione.
Il ragionamento e' simile se $R[j] > L[i]$, sono le rr16-17 a garantire la conservazione dell'invariante.

CONCLUSIONE

Alla fine del ciclo, $k = r + 1$. Per l'invariante di ciclo, il sottoarray $A[p \dots k-1]$ contiene i $k - p = r - p + 1$ elementi ordinati che sono piu' piccoli di L e di R. Insieme gli array contengono $r - p + 3$ elementi. Tutti, **tranne le sentinelle**, sono stati copiati in A.

6.4 Algoritmi Lineari

Gli algoritmi lineari hanno una complessita' di tempo **lineare**, ovvero $T(n) = \Theta(n)$. Essi infatti non operano per confronti, e quindi vengono meno al **Teorema Ordinamento per Confronti**, che ci dice che *qualsiasi algoritmo di ordinamento basato su confronti ha complessita' pari a* $\Omega(n * \log(n))$.
Gli algoritmi Lineari sono:

1. **Bucket Sort**
2. **Counting Sort**
2. **Radix Sort**

6.4.1 Bucket Sort

Il Bucket Sort ordina gli elementi **suddividendoli in *bucket***, ognuno raggruppante gli elementi che rientrano nella *categoria* identificata dal bucket. Un esempio potrebbe essere quello di *Ordinare numeri decimali*, in cui **ogni numero rientra nel *bucket* identificato dal primo numero decimale**. Successivamente, ordina ciascun bucket con un algoritmo *stabile*, ed infine combina ogni bucket in ordine crescente.

```
void BucketSort(arr[])
    buckets[n]<-new array of buckets
    for (i<-1 to n)
        bucketIndex<-n * arr[i];
        buckets[bucketIndex]<-arr[i]
    for (i<-1 to n)
        //Ordina i bucket con un algoritmo
    index<-0;
    for (i<-1 to n)
        for (j<-0 to buckets[i].size
            arr[index] = buckets[i][j]
```

L'algoritmo BucketSort ha complessita' $T(n) = \Theta(n) + \sum_{i=0}^{n-1}(O(n_i^2))$, dove:

n_i^2 e' la complessita' per ordinare ciascun bucket tramite Insertion Sort

$\Theta(n)$ e' la complessita' per suddividere A in ciascun bucket

n_i indica ciascun bucket (*ciascun "elemento" del BucketSort*)

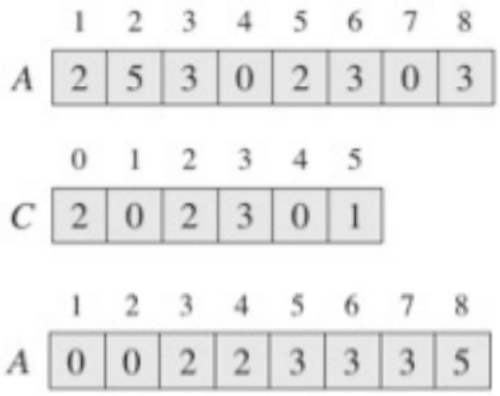
Calcolandoci il valor medio, mediante il *valore atteso*, otteniamo la complessita' dell'algoritmo, ovvero $T(n) = \Theta(n)$.

6.4.2 CountingSort

Il Counting Sort e' un algoritmo che *sfrutta il numero di occorrenze di ciascun elemento* ***come indice per l'ordinamento***. Ogni volta che un elemento distinto appare, si aumenta l'occorrenza appropriata, che poi verra' usata come indice per riposizionare l'elemento stesso al suo posto nell'array di output.

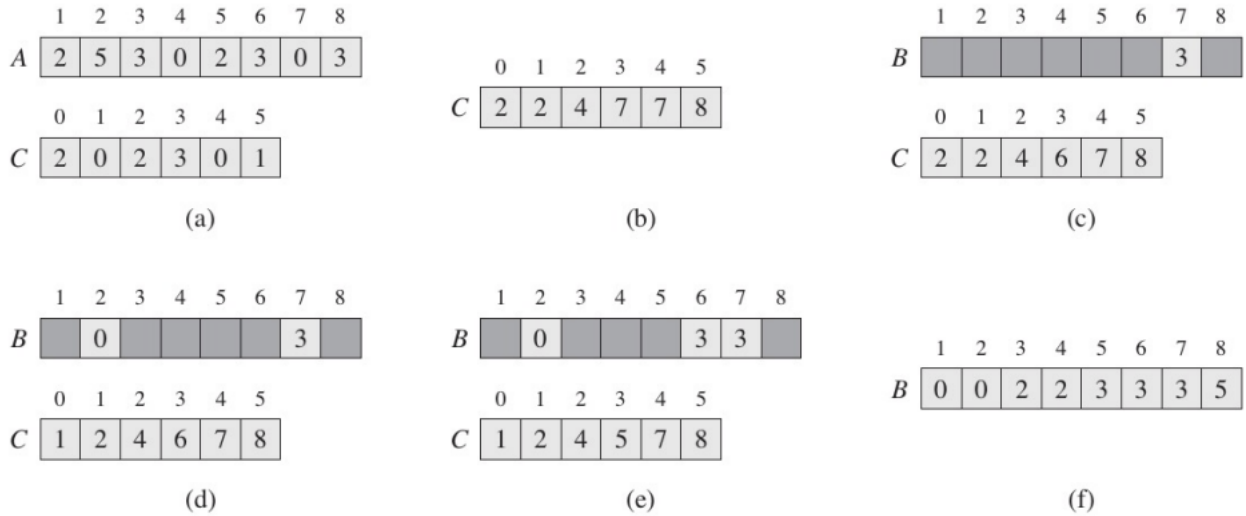
```
void CountingSort(A,dim){
    int min = A[1];
    int max = A[1];
    for(i=2 to dim){
        if(A[i]<min)
            min<-A[i]
        else if(A[i]>max)
            max<-A[i]
    }
    dimC<-max-min+1;
    C<-new array[dimC];
    for(i<-1 to dimC)
        C[i]<-0;
    }
    for(i<-1 to dim){
        C[A[i]]<-C[A[i]]+1;
    }
    k<-1
    for(i<-1 to dimC)
        while(C[i]>0){
            A[k]<-C[i]
            k<-k+1
            i<-i-1
```

Vediamo ora come funziona il CountingSort. Usiamo quest immagine:



Inizialmente, l'array C e' inizializzato a 0. Per ogni elemento di A, scorriamo l'array C e aumentiamo il valore *in posizione pari* ***al valore dell'elemento A[i]***. In questo modo, avremo il **count** di ciascun elemento, **in modo ordinato**.
Notiamo infatti che, preso il primo elemento di $A[]$, ovvero 2, nell'array $C[]$ avremo un incremento di 1 nella posizione di indice 2. Stessa cosa accadra' per il 2 in posizione 5, e quindi nell'array $C[]$ manteniamo l'informazione che esistono due 2 nell'array iniziale. Notiamo anche che non esistono 1 nell'array $A[]$, cosi' come non esistono 4, infatti **il valore nelle loro posizioni e' zero**.
Una volta creato e completato l'array C, lo scorriamo tutto, e per ogni elemento inseriamo in $A[i]$ il valore segnato **dall'indice di** $C[]$, *tante volte quant e' il valore* $C[i]$.

Vediamo adesso un'immagine piu' comprensibile:



Notiamo l'esecuzione del Counting Sort, per ogni iterazione, e possiamo quindi capire che *per ogni elemento di A,* ***C mantiene l'informazione sulle occorrenze e gli ordini****, mentre B* ***e' l'array ordinato, di output****.*

6.4.3 Radix Sort

Il Radix Sort e' un algoritmo di ordinamento che ordina **record con chiavi multiple**, ad esempio la **data**. Il Radix Sort ordina *prima le cifre meno significative*, e poi procede verso quelle piu' significative.
Il codice e' molto semplice:

```
Radix-Sort(A,d)
    for (i<-1 to d)
        //metodo di ordinamento stabile su cifra i
```

Ovvero, richiamiamo un altro algoritmo **stabile** per ordinare le singole cifre/lettere/parti di data, e quindi avremo un ordinamento totale sul record. La complessita' **media** del Radix Sort dipende dall'algoritmo di ordinamento che richiamiamo sulle parti del record da ordinare. Ad esempio, se vogliamo ordinare 106 numeri a 3 cifre, con il radix sort si effettua per ogni dato 3 chiamate al counting sort.

7 Programmazione Dinamica

La programmazione dinamica è una tecnica di programmazione che, rispetto alla programmazione D.E.I., considera le soluzioni ai sottoproblemi precedentemente trovate. La programmazione dinamica opera cercando di **trovare la soluzione migliore associando uno score a tutte le soluzioni possibili**. Tale score va *massimizzato* o *minimizzato*.
La programmazione dinamica è applicabile solo se si verifica che:

1. Il problema possiede la **sottostruttura ottima**
2. I sottoproblemi sono **ripetuti**
3. Lo spazio dei sottoproblemi ha **complessità polinomiale**

Vediamo ora nel dettaglio tali condizioni.

7.1 Sottostruttura Ottima

Un problema possiede la sottostruttura ottima se **la soluzione ottima del problema contiene la soluzione ottima dei sottoproblemi**.
Ciò significa che, se dovessimo risolvere un problema N, andiamo a dividerlo nei suoi sottoproblemi $N_1, ..., N_n$, e per ciascuno cerchiamo la soluzione migliore, detta appunto *ottima*.
Una volta che tale soluzione è trovata per ciascun sottoproblema, possiamo combinare le varie soluzioni, per ottenere la soluzione al problema originario N, e tale soluzione sarà la soluzione **ottima** del problema iniziale.

7.2 Sottoproblemi Ripetuti

Nella programmazione dinamica, si necessita che i problemi vengano ripetuti, cioè che *il problema si ri-presenti, **con taglia più piccola**, all'interno dei sottoproblemi*. Ciò significa che, dovendo risolvere un problema N, se lo dividiamo nei sottoproblemi $N_1, ..., N_n$, la forma del sottoproblema N_i è simile (*se non uguale*) a quella del problema originale N.

7.3 Complessità Polinomiale

L'ultima condizione della PD è che **i sottoproblemi siano risolvibili in tempo polinomiale**.
Questo perchè, se non fossero risolvibili in tempo polinomiale, *il costo per risolvere un sottoproblema sovrasterebbe l'intera complessità dell'algoritmo* per la risoluzione

del problema originale (totale), e quindi quel sottoproblema sarebbe da **scartare**, in quanto sarebbe più problematico e difficile degli altri da risolvere.

8 Algoritmi PD

8.1 Fibonacci

Un esempio classico di algoritmo di Programmazione Dinamica è **l'algoritmo per il calcolo dei numeri della sequenza di Fibonacci**.
Ricordiamo che $F[i] = F[i-1] + F[i-2]$, con $F[0] = F[1] = 1$.
Volendo implementare un algoritmo **ricorsivo**, avremmo:

```
Fibonacci(i)
    if(i=0 OR i=1)
        return 1
    num <- Fibonacci(i-1)+Fibonacci(i-2)
    return num
```

Il costo di questo algoritmo è dato dalla ricorrenza $T(n) = 1.6T(n-1)$, che come soluzione ha $\Theta(1.6^n)$. Tale complessità rende arduo calcolare i numeri di Fibonacci per valori molto grandi di i.
Un algoritmo di PD per risolvere lo stesso problema è il seguente:

```
Num_Fibo(i)
    for j<-1 to MAX_NUM
        Kn_Fib[j]<-0
    Fibonacci(i)


Fibonacci(i)
    if(Kn_Fib[i]!=0)
        return Kn_Fib[i]
    if(i=1 OR i=0)
        return 1
    Kn_Fib[i]<-Fibonacci(i-1)+Fibonacci(i-2)
    return Kn_Fib[i]
```

Vediamo adesso come funziona questo algoritmo:
L'algoritmo inizia creando un vettore (*o una matrice*), in cui salveremo **i numeri di Fibonacci già trovati**.
Ogni qual volta la funzione viene richiamata, controlla prima di tutto se *il valore ricercato è già salvato*, ed in tal caso lo ritorna. Altrimenti, lo calcola, **e lo salva all'interno della table**.

In questo modo, possiamo **eliminare il calcolo dei numeri di Fibonacci che sono già stati calcolati in precedenza**, semplificando la complessità dell'algoritmo in toto.

8.2 LCS

La **Longest Common String** è la sottostringa di dimensione massima, *composta dai caratteri comuni alle due stringhe X e Y*. In termini più rigorosi:
Date due sequenze X e Y, Z è la LCS di X e Y se $Z \in CS \wedge \nexists\ W \in CS : |W| > |Z|$.

8.2.1 Caratterizzazione sottostruttura ottima

Data una sequenza $X = (x_1, ..., x_n)$, $X(i)$ è la sottosequenza $(x_1, ..., x_i)$.Se $X =$ "$ABCDE$", si ha che:

X(1) = "A"

X(2) = "AB"

X(3) = "ABC"

eccetera.
Date le due sequenze $X = (x_1, ..., x_m)$ e $Y = (y_1, ..., y_n)$, sia $Z = (z_1, ..., z_k)$.

Se $x_m = y_n \implies z_k = x_m = y_n$, e $z_{k-1} = LCS(X, Y)$

Se $x_m != y_n\ e\ z_k != x_m \implies z_k = LCS(X_{m-1}, Y_n)$

Se $x_m != y_n\ e\ z_k != y_n \implies z_k = LCS(X_m, Y_{n-1})$

8.3 LCS Progr Dinamica

Nella LCS basata su Programmazione Dinamica, andiamo a definire una tabella $n * m$, dove $|X| = m \ \wedge\ |Y| = n$. Ogni cella $c[i][j]$ della tabella è la lunghezza della LCS fra X_i e Y_j.
Definiamo quindi:

$$c[i][j] = \begin{cases} 0 \iff i = 0 \vee j = 0 \\ c[i-1][j-1] + 1 \iff x_i = y_j \\ max(c[i-1][j], c[i][j-1]) \iff else \end{cases} \tag{10}$$

In questo modo, gestiamo tutti i possibili casi dei valori delle celle di c, in base ai caratteri delle due stringhe.
Ricordiamo che:

1. $c[i][j] = 0 \iff i = 0 \ \vee \ j = 0$

2. $x_i = y_j \implies$ prendiamo il valore **in alto a sx** della cella attuale, e lo aumentiamo di uno

3. se $x_i \neq y_j \implies$, prendiamo il massimo fra il valore della cella in alto e quella a sinistra

Vediamo quindi lo pseudocodice per la LCS in Programmazione Dinamica

```
LCS(X,m,Y,n)
    for(i<-1 to m)
        c[i,0]<-0
    for(j<-1 to n)
        c[0,j]<-0
    for(i<-2 to m)
        for(j<-2 to n)
            if(X[i]=Y[j])
                c[i][j]=c[i-1][j-1]+1
            else
                c[i][j]=max(c[i-1][j],c[i][j-1])
    return c
```

8.4 Distanza di Editing

La Distanza di Editing è un problema simile alla LCS. Essa infatti è data **dal numero di operazioni necessarie a trasformare una stringa in un'altra**, usando operazioni come *Inserimento, Cancellazione Sostituzione* di un carattere.
La distanza di editing si concretizza nel seguente esempio:
Date le stringhe S_1 ="*Eugenio*" e S_2 ="*Militerno*", quante operazioni di **inserimento, rimozione o modifica** servono per trasformare S_1 in S_2? Vi sono due metodi per calcolare la distanza di editing: la distanza di **Hamming** e quella di **Levenshtein**.

8.4.1 Distanza di Levenshtein

La distanza di Levenshtein fra due stringhe $S = (S_1, ..., S_N)$ e $T = (T_1, ..., T_N)$ è definita nel seguente modo:
Siano S_i e T_j i caratteri corrispondenti all'i-simo di S e al j-simo di T. La loro dis-

tanza di editing è data da:

$$d[i][j] = min \begin{cases} d[i-1][j] \\ d[i][j-1] \\ d[i-1][j-1] + (S_i \neq T_j) \\ 0 \iff i, j = 0 \end{cases} \tag{11}$$

Secondo queste operazioni, e usando le due stringhe S_1 =*"Eugenio"* e S_2 =*"Militerno"*, si ottiene la distanza 7, data dalla seguente tabella:

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	1	2	3	4	5	6	7	8	9
2	2	2	2	3	4	5	6	7	8	9
3	3	3	3	3	4	5	6	7	8	9
4	4	4	4	4	4	5	5	6	7	8
5	5	5	5	5	5	5	6	6	6	7
6	6	6	5	6	5	6	6	7	7	7
7	7	7	6	6	6	6	7	7	8	7

8.5 Prodotto di Sequenza di Matrici

Il problema del prodotto di una sequenza di matrici è quello di minimizzare le moltiplicazioni svolte.
Un prodotto si dice **completamente parentesizzato** se *consiste di una sola matrice*, o *di un prodotto di matrici completamente parentesizzato.*
Consideriamo un esempio: avendo quattro matrici, A_1, A_2, A_3, A_4, potremo avere uno fra **cinque** modi di parentesizzare il prodotto, come riportato nell'immagine:

$$\Big(A_1\big(A_2(A_3A_4)\big)\Big)$$

$$\big((A_1A_2)(A_3A_4)\big)$$

$$\Big(\big((A_1A_2)A_3\big)A_4\Big)$$

$$\Big(\big(A_1(A_2A_3)\big)A_4\Big)$$

$$\Big(A_1\big((A_2A_3)A_4\big)\Big)$$

La ricerca della migliore parentesizzazione è importante per il calcolo dei prodotti, perchè può avere un impatto notevole sul calcolo dei prodotti.
Il problema della moltiplicazione di matrici è impostato in questo modo:
Data una sequenza di matrici $A_1, \ldots, A_n$, dove ogni matrice A_i ha dimensioni $p_{i-1} * p_i i = 1, \ldots, n$, si deve determinare lo schema di parentesizzazione completa del prodotto $A_1 * \ldots * A_n$ che **minimizza il numero di prodotti scalari**.
Indichiamo con $P(n)$ il numero di parentesizzazioni alternative di una sequenza di n matrici. Per $n = 1$, abbiamo una sola matrice, ed un solo schema di parentesizzazione.
Quando $n \geq 2$, un prodotto di matrici completamente parentesizzato è dato **dal prodotto di due sottoprodotti di matrici parentesizzati**, e la suddivisione fra i prodotti può avvenire fra la k-sima e la $k+1$-sima matrice, per $k = 1, ..., n-1$.
Quindi, $P(n)$ è dato da :

$$P(n) = \begin{cases} 1 \iff n = 1 \\ \sum_{k=1}^{n-1} P(k) * P(n-k) \iff n \geq 2 \end{cases} \tag{12}$$

La risoluzione di tale ricorrenza è data da $P(n) = C(n-1)$, dove $C(n)$, detto **Numero catalano**, è dato da: $C(n) = \frac{1}{n+1} * \binom{2n}{n} = \Omega(\frac{4^n}{n^{\frac{3}{2}}})$

8.5.1 Caratterizzazione Sottostruttura Ottima

Sia $A_{i,j}$ la matrice risultato del prodotto, con una parentesizzazione ottima, delle matrici $A_i, ..., A_j$.
Esiste un indice $k = i, ..., j-1$ tale che è A_k è il prodotto fra $A_{i...k}$ e $A_{k+1,j}$. Anche $A_{i...k}$ e $A_{k+1,j}$ possiedono parentesizzazione ottima.
Infatti, supponendo **per assurdo** che esiste una parentesizzazione (*con costo inferiore*) $A^*_{i...k}$ diversa, allora $A^*_{i...k}A_{k+1,j}$ avrebbe un costo inferiore a A_{ij}, che **è assurdo**. Analogamente, si dimostra che anche $A_{k+1,j}$ è ottima.
In altre parole, *ogni soluzione ottima al problema della parentesizzazione contiene le soluzioni ottime dei sottoproblemi.* Il prossimo passo è **definire ricorsivamente il valore di una soluzione ricorsiva**.
Scegliamo come sottoproblemi i problemi di determinare il costo minimo di una parentesizzazione di $A_iA_{i+1}...A_j$. Sia m il numero minimo di prodotti necessari a calcolare il prodotto. Per il problema principale, il costo per calcolare $A_{1...n}$ sarà $m[1, n]$.
Se abbiamo $i = j$, il problema è banale, *essendo formato da un'unica matrice* A_{ii}, quindi non occorre calcolare alcun prodotto scalare. In tal caso, $m[i, i] = 0, \forall i = 1, ..., n$.
Per calcolare $m[i, j]$, quando $i < j$, il costo sarà ottenuto sfruttando la struttura della parentesizzazione ottima. Supponiamo che la parentesizzazione ottima $A_{i...j}$ si suddivida in due parti $A_{i...k}$ e $A_{k+1,j}$, dove $i \leq k < j$.
Quindi, il costo è dato dalla **somma del costo delle due matrici** $A_{i...k}$ e $A_{k+1,j}$, ossia $m[i, k]$ e $m[k+1, j]$, più il costo per moltiplicare $A_{i...k}$ e $A_{k+1,j}$, ossia $p_{i-1} * p_k * p_j$ colonne. Quindi avremo che $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} * p_k * p_j$.
Questa equazione presuppone che k sia conosciuto, ma in realtà k può assumere $j - i$ valori diversi. Pertanto, bisogna esaminarli tutti, per ottenere il migliore.
Quindi, l'equazione di ricorrenza per il costo minimo è:

$$m[i, j] = \begin{cases} 0 \iff i = j \\ min(m[i, k] + m[k + 1, j] + p_{i-1} * p_k * p_j) \iff i < j \end{cases} \tag{13}$$

I valori di $m[i, j]$ non ci forniscono tutte le informazioni per ricostruire la soluzione ottima. Pertanto, definiamo $s[i, j]$ come **il valore di k in cui è stato suddiviso il prodotto** $A_{i...j}$ **per ottenere una parentesizzazione ottima**, ossia *il k per cui* $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} * p_k * p_j$.
Vediamo quindi il codice:

```
Recursive-Matrix-Chain(p,i,j)
    if i=j return 0
    m[i,j]<-INF
    for k<-i to j-1
        q<-Recursive-Matrix-Chain(p,i,k)+Recursive-Matrix-Chain(p,k+1,j)+p_i-1*p_k*p_
    if q<m[i,j]
        m[i,j]<-q
    return m[i,j]
```

La cui ricorrenza è:

$$T(n) \geq \begin{cases} 1 \iff n = 1 \\ 1 + \sum_{k=1}^{n-1}(T(k) + T(n-k) + 1) \iff n > 1 \end{cases} \tag{14}$$

Risolvendo tale ricorrenza, si ha che $T(n) = \Omega(2^n)$.
Il numero dei problemi distinti è però basso, ovvero $\Theta(n^2)$.

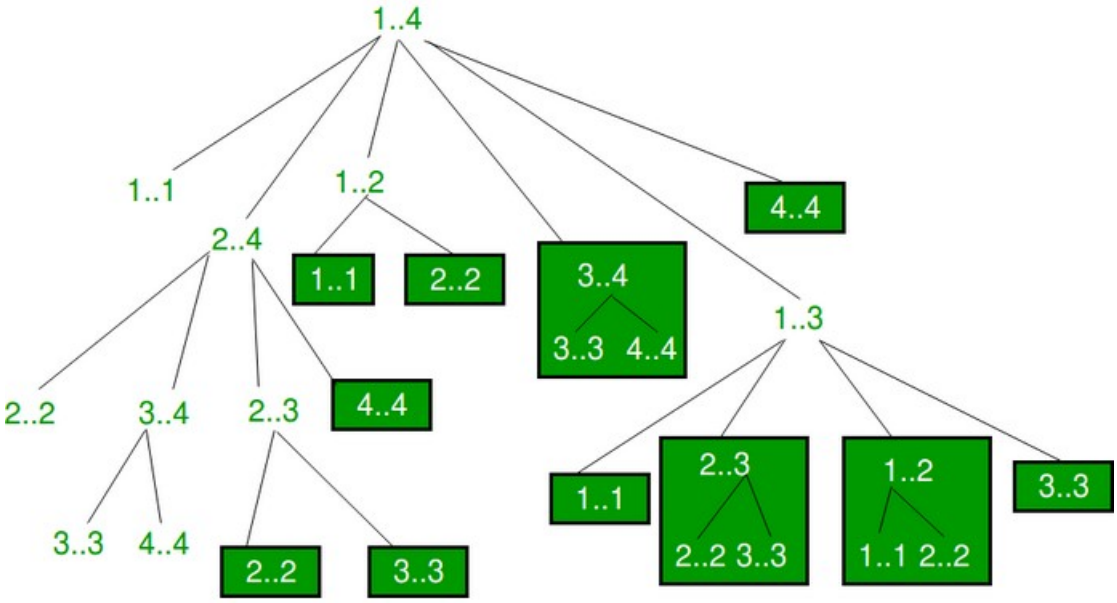
8.6 Matrix-Chain-Order

La procedura MCO assume che la matrice A_i abbia p_{i-1} righe e p_i colonne, e l'input è una sequenza $p = p_0, ..., p_n$.
La procedura usa una tabella $m[n][n]$ per memorizzare i costi $m[i][j]$, e una tabella $s[n][n]$, che registra *l'indice del costo ottimo nel calcolo di* $m[i][j]$.

```
Matrix-Chain-Order(p)
    n<-p.length-1
    //Siano m[1,...,n][1,...,n] e s[1,...,n][2,...,n]
    for i<-1 to n
        m[i,i]<-0
    for l<-2 to n
        for i<-1 to n-l+1
            j<-i+l-1
            m[i,j]<-INF
            for i<-1 to j-1
                q<-m[i,k]+m[k+1,j]+p_i-1*p_k*p_j
                if q<m[i,j]
                    m[i,j]<-q
                    s[i,j]<-k
```

La complessità di questo algoritmo è $O(n^3)$.

Un esempio della **MCO** è dato usando una matrice, identificata dal vettore $\{1, 2, 3, 4, 3\}$. Tale matrice produce un output:



che sono le possibili parentesizzazioni della matrice $\{1 \dots 4\}$. Il numero totale di parentesizzazione per la matrice $\{1, 2, 3, 4, 3\}$ è 30.
Notiamo che la dimensione della matrice in alto è 4 *perchè stiamo cercando le parentesizzazioni della matrice* $1 \dots 5$*,che* ***andrà suddivisa almeno in due matrici****, al massimo in una matrice* $1 \dots 4$ *e* $1 \dots 1$

9 Memoization

La memoization è una tecnica che fa uso di una tabella per salvare i valori già calcolati della risoluzione dei sottoproblemi. Quando bisogna risolvere un sottoproblema, si controlla prima se è già stato risolto, ed in tal caso si ritorna il valore salvato, altrimenti si calcola e si salva il risultato nella tabella.

```
Memoized-Matrix-Chain(p)
    n<-p.length-1
    Sia m[1...n,1...n]
    for i<-1 to n
        for j<-1 to n
            m[i,j]<-INF
    return LookUpChain(m,p,1,n)

LookUpChain(m,p,i,j)
    if m[i,j]<INF
        return m[i,j]
    if i=j m[i,i]<-0
    else
        for k<-i to j-1
            q<-LookUpChain(m,p,i,k)+LookUpChain(m,p,k+1,j)+p_i-1*p_k*p_j
            if q<m[i,j]
                m[i,j]<-q
    return m[i,j]
```

Questo algoritmo ha complessità $O(n^3)$, che è molto minore dell'iniziale $\Omega(2^n)$

10 Programmazione Greedy

La programmazione Greedy e' simile alla programmazione dinamica, nel senso che bisogna dimostrare alcune proprieta' del problema che si vuol risolvere, per poterlo risolvere con questa tecnica di programmazione. Inoltre, il risultato della programmazione dinamica e' la massimizzazione o la minimizzazione di alcuni valori.
Bisogna infatti dimostrare che *il problema gode della sottostruttura ottima*, ma anche che **il problema gode della greedy choice**, ovvero *selezionando la soluzione ottima localmente, per ogni sottoproblema, si otterra' la soluzione ottima del problema originale.*

10.1 Greedy Activity Selector

Il problema del GAS chiede di trovare le **attivita' mutualmente compatibili**, ovvero attivita' che iniziano dopo la terminazione delle precedenti.
Supponiamo di avere un insieme S di attivita' s_i, ciascuna con un tempo di inizio s_i ed un tempo di fine f_i. Il problema GAS richiede di *massimizzare il numero di attivita' del set S per cui $s_i \geq f_{i-1}$*. In sostanza, per ogni attivita' s_i, che appartiene all'insieme semiaperto $[s_i, f_i)$, bisogna trovare un'attivita' s_j appartenente all'intervallo semiaperto $[s_j, f_j)$ tale che $s_j \geq f_i$.
Fornendo un esempio, vediamo questa tabella:

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	9
6	5	9
7	6	10
8	8	11
9	8	12
10	2	14
11	12	16

Inizialmente, il set di output (*ovvero il set che racchiude le attivita' compatibili*) e' vuoto ($A = \emptyset$). Un metodo semplice per controllare quali attivita' sono compati-

bili, e' quello di **ordinarle per s_i crescente**. Si puo' usare un algoritmo di ordinamento, ma il metodo piu' rapido (*e meno costoso*) e' quello di usare **una coda di priorita'**, eventualmente basata su *heap o fibonacci heap*. Questo ci garantisce di estrarre sempre l'attivita' con s_i minore nel set S di attivita', e dunque la *successiva* temporalmente.
Una volta estratta l'attivita' s_i, andiamo a verificare (*se non e' la prima*), che $s_i \geq f_{i-1}$, e in tal caso la inseriamo nel set di output A ($A = A \cup a_i$). Facendo questo per ogni attivita', al termine di ciascuna presente nel set originale S, otterremo che *nel set A avremo tutte le attivita' mutualmente compatibili*, mentre le restanti rimarranno nel set S.

10.1.1 Sottostruttura ottima

Il problema del GAS possiede sottostruttura ottima, in quanto *ogni sottoproblema e' una ripetizione del problema originario, ma con taglia minore*. Notiamo infatti che
$\forall i = 1, ..., n\ P(i) = \{a_i \in A : s_i \geq f_{i-1}, a_{i-1} \in A\} \implies P(i-1) = \{a_{i-1} \in A : s_{i-1} \geq f_{i-2}, a_{i-1} \in A\}$. In poche parole, ogni volta che inseriamo un'attivita' a_i all'interno del set A, stiamo *riducendo la taglia del successivo sottoproblema*, e stiamo ripetendo il problema iniziale, ovvero **stiamo cercando un'attivita' mutualmente compatibile a quelle presenti nel set A_i, ovvero il set A ottenuto dopo i iterazioni**.

10.1.2 Complessita'

L'equazione di ricorrenza del GAS e' la seguente:

$$c[i,j] = \begin{cases} 0 \iff S_{i,j} = \emptyset \\ max_{a_k \in S_{i,j}}(c[i,k] + c[k,j] + 1) \iff S \neq \emptyset \end{cases} \tag{15}$$

dove $c[i,j]$ e' la dimensione di una soluzione ottima di $S_{i,j}$.
Questa equazione di ricorrenza ci fa capire che *per ogni sottoproblema $S_{i,j}$, il sottoproblema generato dalla selezione della soluzione ottima al passo i, j diventa* ***il massimizzare il numero di attivita' mutualmente compatibili con il set S trovato in quel momento***, *ovvero trovare il massimo numero di attivita' tali che a_k e' mutualmente compatibile con $S_{i,j}$, e per cui* ***le restanti attivita'*** $a_k, ..., a_j$ ***mutualmente compatibili siano massime in numero***.

10.1.3 Teorema della Greey Choice

Consideriamo un sottoproblema non vuoto S_k, e sia a_m l'attivita' in S_k che ha il primo tempo di fine. Allora, l'attivita' a_m e' inclusa in qualche sottoinsieme mas-

simo di attivita' mutualmente compatibili di S_k.
DIMOSTRAZIONE

Supponiamo che A_k sia un sottoinsieme massimo di attivita' mutualmente compatibili di S_k e sia a_j l'attivita' in A_k con il minor tempo di fine.
Se $a_j = a_m$, abbiamo dimostrato che a_m e' utilizzata in qualche sottoinsieme massimo di attivita' mutualmente compatibili di S_k.

Se $a_j \neq a_m$, sia $A'_k = \{A_k - \{a_j\} \cup \{a_m\}\}$ l'insieme A_k con la sostituzione di a_m con a_j. Le attivita' in A'_k sono disgiunte (*perche' le attivita' di A_k sono disgiunte, ed abbiamo sostituito l'attivita' che finisce prima (a_j), con a_m* ***tale che*** $f_m \leq f_j$).
Poiche' $|A'_k| = |A_k|$, A'_k e' un insieme massimo di attivita' mutualmente compatibili di S_k che include a_m, che sarebbe la a_j scelta nel teorema.

10.1.4 Codice

```
GreedyActivitySelector(s,f)
    //Attivita' Ordinate per s_i
    n<-s.length
    A<-a_1
    k<-1
    for m<-2 to n
        if(s[m]>=f(k])
            A<- A U a[m]
            k<-m
    return A
```

Il codice presuppone he l'insieme delle attivita' sia **gia' ordinato**. Altrimenti, va usata una *Coda di Priorita'*, oppure un *Heap* per estrarre sempre quella con s_i minore.
L'algoritmo confronta s_i e f_j, e nel caso siano compatibili, inserisce a_i all'interno del set A.

10.2 SJF

Lo **Shortest Job First** è un algoritmo di *Scheduling*, che *ordina le attività in base a tempo di completamento crescente*. In questo modo, avremo una lista di attività da svolgere, ciascuna con tempo maggiore della successiva.
Un algoritmo del genere è utile per minimizzare *il tempo d'attesa* dell'utente fra le varie task. Infatti, se ordiniamo tutte le attività per tempo crescente, andremo a minimizzare il tempo totale d'attesa, sommando sempre attività che richiedono un tempo crescente.
In questo modo, avremo il minimo distacco fra ogni attività, per cui non vi sarà molto spesso un momento in cui non vi è un'attività in corso, e quindi un'attesa.

```
SJF(S)
    //Coda di priorità
    Q<-0
    foreach(si in S)
        INSERT(Q,si)
    totalTime<-0
    while(Q NOT EMPTY)
        //Estraggo l'attività con tempo minore
        ti<-EXTRACT-MIN(Q)
        totalTime<-totalTime+ti
    return totalTime
```

Questo algoritmo richiede $T(n) = O(n \log(n))$, e ritorna il tempo totale d'attesa minimizzato, per il set S di attività.

11 Zaino

Il problema dello zaino e' un problema di **massimizzazione**, in cui si cerca di massimizzare il valore degli elementi presenti all'interno dello zaino.
Il problema e' posto nel seguente modo:
Ci sono n oggetti. L'oggetto i-esimo pesa w_i chili e vale v_i dollari.Un ladro vuol riempire uno zaino con oggetti in modo da massimizzare il valore complessivo del carico ma di non superare un peso fissato di W chili.
Esistono due tipi di algoritmi per lo zaino: lo **Zaino 0-1** e lo **Zaino Frazionario**.
Le differenze fra i due sono:

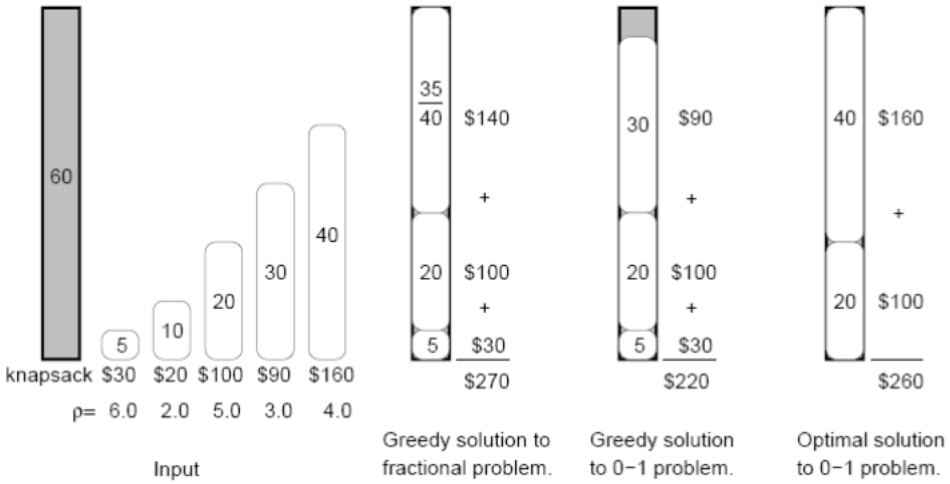
1. Lo zaino 0-1 limita la scelta in base al peso dell'elemento e alla capacita' totale. Se la capacita' e' maggiore o uguale al peso dell'oggetto, puoi prenderlo, altrimenti dovrai lasciarlo. Lo zaino frazionario invece consente di prendere una **parte** dell'oggetto, pari alla frazione trasportabile, ovvero la capacita' rimanente.

2. Lo zaino 0-1 **non gode della greedy choice**, per cui andra' usata una tecnica di programmazione dinamica, mentre lo zaino frazionario gode di tale scelta, **e quindi andremo sempre a scegliere la soluzione localmente ottima**.

.

11.1 Fractional Knapsack

Il problema dello zaino frazionario e' risolvibile usando la programmazione Greedy. Cio' significa prendere l'oggetto k tale che $\frac{v_k}{w_k}$ e' massimo. Vediamo un esempio con la differenza con lo zaino 0-1



Notiamo che, usando lo zaino 0-1, *abbiamo la soluzione ottima che vale 260 dollari*. Se usiamo, **erroneamnte**, la programmazione dinamica dello zaino frazionario per *il problema 0-1*, abbiamo una soluzione con valore minore (220 dollari). Invece, la soluzione migliore e' rappresentata dall'utilizzo della **tecnica greedy** sul problema **frazionario**, che comporta un valore totale di 270 dollari.
Questo significa che, *in base al problema sottoposto*, bisogna sempre usare l'algoritmo corretto.
Chiaramente, la soluzione greedy massimizza il valore ottenuto, ma non significa che **va usata sempre**, perche' e' applicabile **solo ed univocamente quando vale la Greedy Choice**.
L'algoritmo dello zaino frazionario **presuppone che gli elementi siano già pre-ordinati**, in base al rapporto $\frac{v_k}{w_k}$, ovvero valore/peso. In tal modo, avremo sempre prima gli oggetti che valgono di più, a parità di peso. Inoltre, seleziona una quantità $X[k]$, che è la quantità dell'oggetto k da prendere, e un $Value$ che è il valore totale dello zaino.
La procedura ritorna quindi $Value$ e X, che ci permettono di dire *quanti soldi siamo riusciti a racimolare*, e soprattutto *quali oggetti (**o frazioni di oggetti**) bisogna prendere.*

```
Fractional_Knap(V,W,X,TOT)
    for k<-1 to N
        X[k]<-0
    Value<-0.0
    Capacity<-TOT
    k<-1
    while(k<=N AND Capacity>0)
        if(W[k]<=Capacity)
            Value <- Value + V[k]
            Capacity <- Capacity - W[k]
            X[k]<-1.0
        else
            X[k]<-Capacity/W[k]
            Value<-Value+X[k]*V[k]
            Capacity<-0.0
        k<-k+1
```

Essendo gli elementi ordinati per $\frac{v_k}{w_k}$, possiamo semplicemente *iterare* su ciascuno, e selezionare sempre **la prima frazione possibile che può entrare nello zaino**. *Chiaramente, essa può anche essere 1.0*, nel caso in cui il peso dell'oggetto sia minore della capacità attuale dello zaino.
In questo modo, andremo sempre a selezionare l'oggetto con valore massimo possibile, e quindi andremo a **massimizzare il valore dello zaino**.

11.2 0-1 Knapsack

Il problema dello zaino 01 non è risolvibile con la programmazione Greedy, perchè **non gode della Greedy Choice**. Dobbiamo quindi risolverlo con la *programmazione dinamica*.

11.2.1 Sottostruttura Ottima

Denotiamo son $S(i, w)$ il sottoproblema in cui il ladro può scegliere gli oggetti nell'insieme $\{a_1, ...a_i\}$, $\forall i = 1, ..., n$, per realizzare il furto di maggior valore, compatibile con una capacità residua dello zaino pari a $1 \leq w \leq W$.
Osserviamo che i casi banali sono:

1 - $S(0, W)$, ossia se non vi sono oggetti (*l'appartamento è vuoto*)

2 - $S(i, 0)$, ossia se *lo zaino è pieno*

In entrambi i casi, il risultato sarebbe di 0, in quanto non potremo prendere elementi (*non ci sono o non ci entrano*)
Sia $S(i, w)$ un sottoproblema **non banale** $(i, w > 0)$. Esaminando il peso w_i dell'i-simo oggetto, abbiamo che esistono diversi casi:

1. $w_i > W$: L'oggetto non può essere preso, risolviamo il sottoproblema $S(i - 1, w)$
2. $w_i \leq W$: Abbiamo due possibili strade:
   2.1 Prendiamo a_i e risolviamo $S(i - 1, w - w_i)$
   2.2 Ignoriamo a_i e risolviamo $S(i - 1, w)$

Andremo sempre a scegliere l'opzione (fra 2.1 e 2.2) che ci permette di ottenere il valore maggiore. In ogni caso, una soluzione ottima per $S(i, w)$ deve contenere una soluzione per uno dei due sottoproblemi $S(i - 1, w)$ o $S(i - 1, w - w_i)$.

Andiamo a dimostrare il **primo caso**, cioè $a_i \neq A(i, w)$

$A(i, w)$ conterrà dunque una soluzione $A(i - 1, w)$ per il problema $S(i - 1, w)$.
Supponiamo per assurdo che $A(i, w)$ *non sia ottima*. Allora, deve esistere una soluzione alternativa $A^*(i - 1, w)$ per $S(i - 1, w)$ il cui valore è maggiore di quello di $A(i - 1, w)$. Ma in questo caso si avrebbe:
$v(A(i, w)) = v(A(i - 1, w)) < v(A^*(i - 1, w)) = v(A^*(i, w))$
Ciò **è un assurdo**, perchè $A(i, w)$ è una soluzione ottima al sottoproblema $S(i, w)$.
Andiamo ora a vedere il **secondo caso**

Sia $A(i, w)$ una soluzione ottima $S(i, w)$ con $i, j > 0$. Consideriamo il caso $A(i, w) = \{a_i\} \cup A(i - 1, w - w_i)$, con $A(i - 1, w - w_i)$ che è una soluzione ottima per il sottoproblema $S(i - 1, w - w_i)$.
Dimostriamo ora che la nostra soluzione è ottima:
Se **per assurdo** $A(i - 1, w - w_i)$ non fosse ottima, allora esisterebbe una $A * (i - 1, w - w_i)$ per *lo stesso sottoproblema*, il cui valore sarebbe maggiore di quello di $A(i - 1, w - w_i)$.
E' pertanto possibile **costruire una nuova soluzione** del problema principale $A * (i, w) = \{a_i\} \cup A * (i - 1, w - w_i)$, il cui valore è maggiore rispetto a quello della soluzione principale.
Infatti, si avrebbe: $v(A^*(i, w)) = v_i + v(A^*(i-1, w-w_i)) > v_i + v(A(i-1, w-w_i)) \implies v(A^*(i, w)) > v(A(i, w))$.
Questo **è un assurdo**, in quanto contraddice che $A(i, w)$ *sia una soluzione ottima del problema* $S(i, w)$, e quindi non può esistere una soluzione con valore maggiore.

11.2.2 Numero Sottoproblemi

Nel problema dello zaino, abbiamo W_n sottoproblemi, della forma $S(i, w), \forall\ 1 \leq i \leq n\ \ \wedge 1 \leq w \leq W$.
Il problema iniziale è dato da $S(n, W)$, mentre i casi banali sono W sottoproblemi della forma $S(0, w)$ ed n problemi della forma $S(i, 0)$.
In totale abbiamo quindi $W * n + n + W = O(W * n)$ sottoproblemi.

11.2.3 Calcolo Soluzione Ottima

Sia $v(i, w)$ il valore della soluzione ottima del sottoproblema $S(i, w)\ , \forall i = 0, ..., n \wedge w = 0, ..., n$. Dati i casi banali $S(0, w)$ e $S(i, 0)$, abbiamo che, *per tutti i casi non banali*:

$$v(i, w) = \begin{cases} v(i-1, w) \iff w_i > w \\ max\{v(i-1, w), v(i-1, w-w_i)\} \iff w_i \leq w \end{cases} \tag{16}$$

Ovvero, andiamo a selezionare sempre **l'opzione dal massimo guadagno**. Ciò comporta che *se l'oggetto può entrare nello zaino, lo prendiamo*, e risolviamo il sottoproblema ottenuto prendendo l'oggetto, diminuendo la capacità, ed aumentando il valore dello zaino.
Altrimenti, risolviamo il sottoproblema *che si ottiene dal sottoproblema rimuovendo l'oggetto a_i dall'insieme degli oggetti presenti.*

Il codice per lo **Zaino 0/1** e' il seguente:

```
0_1Knapsack(int W,weights,values,N){
    dp[N][W]<-new table
    for (i<-1 to N)
        for (w<-1 to W)
            dp[i][w]<-0;
    for (i<-1 to n)
        for (w<-1 to W)
            if (weights[i-1] <= w)
                dp[i][w]<-max(dp[i-1][w], dp[i-1][w - weights[i-1]] + values[i-1])
            else
                dp[i][w]<-dp[i-1][w];
    return dp[n][W];
```

12 Codici di Huffman

I codici di Huffman definiscono *una tecnica diffusa ed efficiente per la compressione di dati*. Usando questa tecnica, si ottiene un risparmio dal 20% al 90%.
Supponiamo di voler comprimere un file di 100000 caratteri, $\{E, F, G, H, I\}$, le cui frequenze sono $\{40\%, 5\%, 25\%, 10\%, 20\%\}$.
Huffman usa un **codice prefisso** a *lunghezza variabile*. Ciò significa che *ogni lettera è codificata in un modo diverso*, ma con la proprietà che **nessuna stringa di compressione è suffisso di un'altra**.
Il codice di Huffman funziona in questo modo:
Si assegna una stringa corta ai caratteri molto frequenti, *così da ottenere un risparmio di bit*, e si assegna **la stringa più lunga al carattere meno frequente**.
Usando l'esempio precedente, otterremo una situazione del genere:

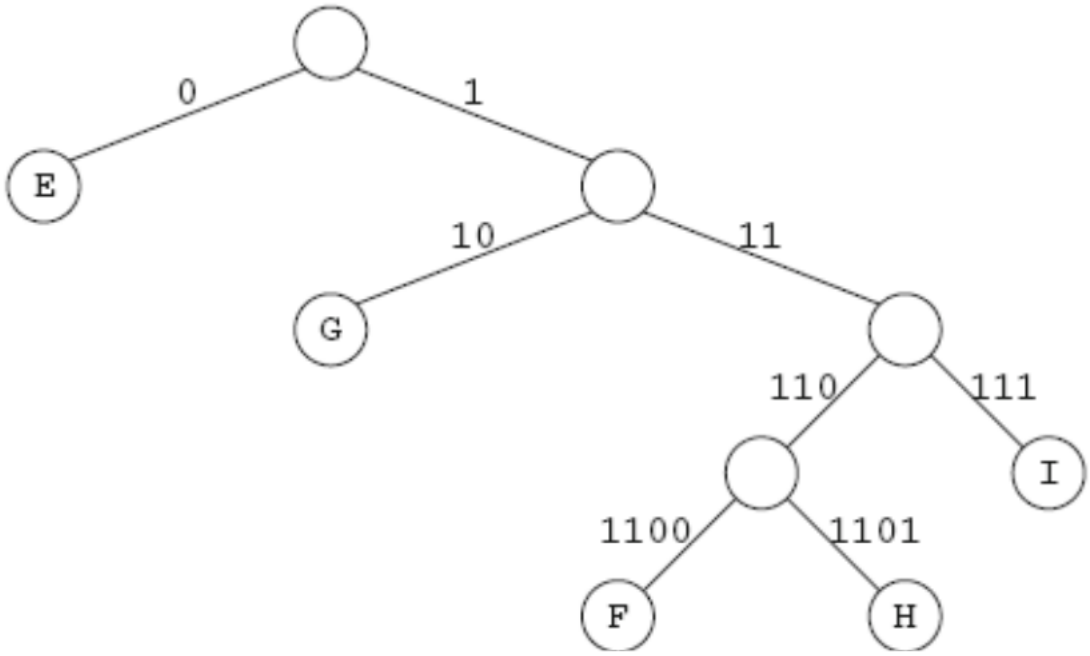


Table 3.1 Frequency of each character in the file

	E	*F*	*G*	*H*	*I*
Frequency	40%	5%	25%	10%	20%
Fix length code	000	001	010	011	100
Huffman code	0	1100	10	1101	111

Notiamo subito due cose:
1-Il carattere E, più frequente, ha un codice di un solo bit, per cui si utilizzeranno

40000 bit per codificarlo, mentre il carattere F,meno frequente, ha un codice di 4 bit, per cui si utilizzeranno 20000 bit per codificarlo.
2-Ogni carattere è ottenibile scorrendo l'albero e quindi si va a **comporre la stringa di compressione**. Inoltre, alcuni nodi non hanno caratteri, ma sono semplicemente *nodi frequenza*, che ci consentono di avere la somma delle fequenze dei caratteri e poterci muovere nell'albero.
In questo modo, rispetto ad una codifica a codice fisso, che impiegherebbe 300000 bit, la codifica a codice prefisso di Huffman impiega solamente 210000 bit, con un risparmio del 30%.

Per ogni carattere c dell'alfabeto C (*i caratteri su cui viene costruito Huffman*), indichiamo con $f(c)$ la sua frequenza nel file, e con $d_t(c)$ la sua profondità nell'albero, che è anche **la lunghezza della parola in codice per il carattere** c.
Il numero di bit per codificare un file è quindi pari a:
$B(T) = \sum_{c \in C} f(c) * d_T(c)$, che è il costo dell'albero T.

```
Huffman(C)
    n<-|C|
    Q<-C
    for i<-1 to n-1
        z<-ALLOCATE_NODE()
        x<-Left[z]<-EXTRACT-MIN(Q)
        y<-Right[z]<-EXTRACT-MIN(Q)
        f[z]<-f[x]+f[y]
        INSERT(Q,z)
    return EXTRACT-MIN(Q)
```

Con una coda di priorità basata su Heap, il costo dell'algoritmo è di $O(n * \log n)$.
NOTA BENE : ritorniamo l'extract-min perchè avremo il nodo finale, con frequenza 100 dell'albero, ovvero **la radice**, con cui potremo visitare o scorrere **l'intero albero**.

12.1 Lemma Sottostruttura Ottima

Sia C un alfabeto con frequenza $f[c] \forall c \in C$. Siano poi x, y due caratteri in C con frequenza minima. Sia C' il nuovo alfabeto con x, y rimossi ed il nuovo carattere z aggiunto, tale che $C' = C - \{x, y\} \cup \{z\}$.
Definiamo f per C' come per C, con la differenza che $f[z] = f[x] + f[y]$.
Sia T' un albero qualsiasi, che rappresenta *un codice prefisso per l'alfabeto* C'. Allora l'albero T, ottenuto da T' sostituendo il nodo foglia z con un nodo interno che ha x e y come figli, rappresenta un **codice prefisso ottimo per C**.

DIMOSTRAZIONE
Per ogni $c \in C - \{x, y\} \cup \{z\}$, si ha $d_T(c) = d'_T(c)$, e quindi $f(c) * d_T(c) = f(c) * d_{T'}(c)$.

Poichè $d_T(x) = d_T(y) = d_{T'}(z) + 1$, si ha:

$f[x]d_T(x) + f[y]d_T(y) = (f[x] + f[y])(d_{T'}(z) + 1) =$
$f[z]d_{T'}(z) + (f[x] + f[y])$

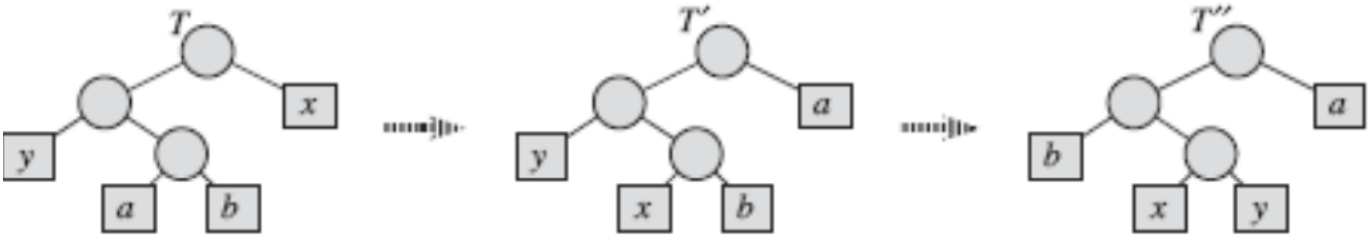
Dalla quale si deduce che il costo $B(T)$ dell'albero T può essere espresso in termini del costo $B(T')$ dell'albero T', ovvero:
$B(T) = B(T') + f[x] + f[y]$

Dimostriamo ora il lemma **per assurdo**.
Supponiamo che T *non sia un codice prefisso ottimo*. Allora deve esistere un albero T'' tale che $B(T'') < B(T)$. L'albero T'' ha x e y come foglie sorelle.“ Sia T''' l'albero T'' con il padre comune di x e y sostituito da una foglia z con frequenza $f[z] = f[x] + f[y]$. Allora:
$B(T''') = B(T'') - f[x] - f[y]$
$< B(T) - f[x] - f[y]$
$< B(T')$
Il che **è un assurdo**, in quanto contraddice l'ipotesi che T' *sia un codice prefisso oddimo per* C'.

12.2 Lemma Greedy Choice

Sia C un alfabeto dove ogni carattere $c \in C$ ha frequenza $f[c]$. Siano x, y due caratteri in C con le frequenze minori.
Allora, esiste un codice prefisso ottimo per C in cui le parole in codice per x ed y **hanno la stessa lunghezza**, e *differiscono solo per l'ultimo bit.*

DIMOSTRAZIONE L'idea è quella di prendere un albero T che rappresenta un *codice prefisso* ***ottimo***, e modificarlo, in modo da creare **un altro albero**, che rappresenta *un altro codice prefisso ottimo*, in modo che i caratteri x e y appaiano *come foglie sorelle*, con profondità massima.
Se questa trasformazione sarà possibile, allora i codici avranno la stessa lunghezza, e differiranno solo per l'ultimo bit.



Siano a e b due caratteri che sono *foglie sorelle* con profondità massima in T. Supponiamo anche che valga: $f[a] \leq f[b] \ \wedge \ f[x] \leq f[y]$.
Poichè $f[a]$ e $f[b]$ sono minime, si ha che $f[x] \leq f[a] \ \wedge f[y] \leq f[b]$.
Ora, si scambiano le posizioni di a e x, per ottenere T', e successivamente le posizioni di b ed y, per ottenere T''.
La differenza di costo fra T e T' è:
$B(T) - B(T') = \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c)$

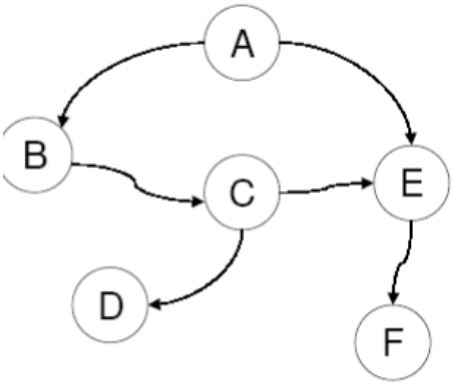
$= f[x]d_T(x) + f[a]d_T(a) - f[x]d_{T'}(x) - f[a]d_{T'}(a)$
$= f[x]d_T(x) + f[a]d_T(a) - f[x]d_T(a) - f[a]d_T(x)$
$= (f[a] - f[x])(d_T(a) - d_T(x)) \geq 0$
Il primo termine è ≥ 0 perchè x è una foglia con frequenza minima. Il secondo è ≥ 0 perchè a è una foglia con profondità massima in T.
Analogamente, lo scambio di y e b non aumenta il costo, quindi $B(T') - B(T'') \geq 0$.
Quindi $B(T'') \leq B(T')$ ma, *essendo T un albero ottimo*, $B(T'') \leq B(T)$.
Quindi, T'' è un albero ottimo dove x e y figurano come foglie sorelle con profondità massima.

13 Grafi

Un grafo e' una struttura dati composta da **Vertici**, ovvero nodi con un valore *chiave* e altri parametri, che identificano la *posizione* all'interno del grafo, e da **Archi**, che sono i *collegamenti* fra i vari vertici.
Un grafo si dice **orientato** se, per ogni arco $\{u, v\} \in G.E, u, v \in G.V$, se la coppia e' **ordinata**; **non orientato** altrimenti.
Qualsiasi grafo puo' essere implementato in due modi:

1. **Lista di adiacenza** : *ogni nodo ha una lista di nodi cui e' collegato*
2. **Matrice di adiacenza** : *le adiacenze sono rappresentate dagli* 1 *in una matrice*, che e' detta appunto *di adiacenza*.

Diremo, inoltre, che :
1 - Se v e' *collegato* a w, allora diremo che v e' **adiacente** a w
2 - Se $\{u, v\}$, il nodo u e' **parent** di v (*nel caso di un orientato, dipende dall'ordine*)
3 - Il **cammino** di un grafo e' l'insieme dei nodi $v_0, ..., v_n$ tali che $\{v_i, v_j\}$ sono archi
4 - La **lunghezza** di un cammino e' *il numero degli archi* del cammino
5 - Un **cammino semplice** e' un cammino in cui tutti i nodi *tra* u *e* v sono distinti



ABCD = cammino semplice

Lungh(ABCD) = 3

13.1 Algoritmi di Visita

Per *visita*, intendiamo **un insieme di cammini *con origine nello stesso vertice***
In un grafo, esistono **due** tipi di visita:

1. **Breadth First Search** : Visita in *Ampiezza*, visita tutti i nodi alla stessa distanza e poi quelli a distanza maggiore

2. **Depth First Search** : Visita in *Profondita'*, visita prima le adiacenze e poi i singoli nodi.

13.1.1 BFS

L'algoritmo di BFS, dato un vertice s detto **sorgente**, visita **tutti i nodi a distanza k** e poi successivamente **tutti i nodi a distanza k+1** dal nodo s.
Per ogni nodo, calcola la *distanza, ovvero il minimo numero di archi necessari a raggiungerlo*, e genera **l'albero Breadth-First**, radicato in s, che contiene *tutti i nodi raggiungibili*.
L'algoritmo opera su grafi **orientati e non orientati**, ed opera in questo modo:

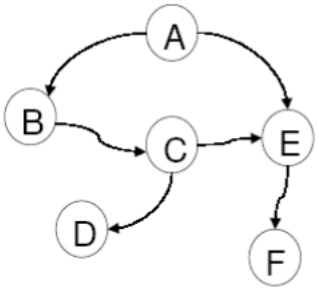
1. All'inizio, colora tutti i nodi di bianco, tranne s che e' grigio

2. Ogni nodo scoperto viene colorato di grigio, e viene calcolata la sua distanza

3. Il nodo viene colorato di nero se tutti i nodi adiacenti sono stati scoperti.

Vediamo con il seguente esempio un'applicazione della BFS:

ABECFD



Notiamo che B e C non vengono scoperti *simultaneamente*, ma *essendo B ed E raggiungibili **con un solo arco** da A, saranno scoperti prima di C*.
Inoltre, il nodo D e' scoperto per ultimo, perche' **il suo parent e' C, che viene scoperto dopo E**.

In questo modo, possiamo dire che D *e' raggiungibile passando per piu' archi* rispetto a F, e dunque verra' posto dopo nella visita (*nell'albero BF*).
L'algoritmo opera quindi con una **coda**, che mantiene i nodi in ordine **FIFO**, che funziona in questo modo:

1. Si inserisce il nodo s, ovvero la radice, nella coda
2. Finche' vi e' un nodo nella coda, lo si estrae
3. Tale nodo viene visitato, e si inseriscono nella coda le sue adiacenze

Andiamo quindi a vedere il codice effettivo dell'algoritmo:

```
BFS(G,s)
    foreach (u in G.V)
        u.c<-WHITE
        u.d<-0
        u.p<-NIL
    s.d<-0
    Q<-empty queue
    ENQUEUE(Q,s)
    while(Q not empty)
        u<-EXTRACT-MIN(Q)
        foreach(v in u.adj)
            if(v.c=WHITE)
                v.c<-GRAY
                v.d<-u.d+1
                v.p<-u
                ENQUEUE(Q,v)
        u.c<-BLACK
```

Notiamo che, per ogni nodo nella coda, *si esplorano prima tutti i nodi adiacenti*, per cui **l'algoritmo visita tutti i nodi a stessa distanza, per poi passare ai nodi con distanza successiva**.
La complessita' dell'algoritmo di BFS e' somma delle operazioni sulla coda, *ciascuna richiedente un costo* $O(1)$, ma che svolte per V volte vengono a costare $O(V)$, e le operazioni di controllo sulla lista di adiacenza, che vengono fatte per tutti i vertici, ovvero tutti gli archi, e quindi costano $\Theta(E)$. Dunque, la complessita' e' $\Theta(V+E)$.

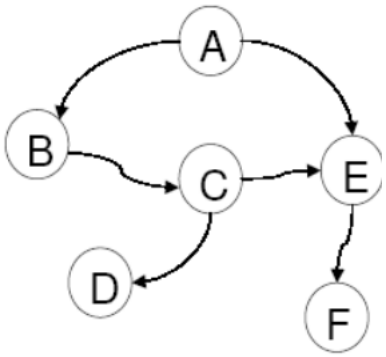
13.1.2 DFS

La visità in profondità (**Depth-First Search**) visita il grafo *andando il più possibile in profondità*, prima di proseguire oltre. Gli archi vengono scoperti **a partire**

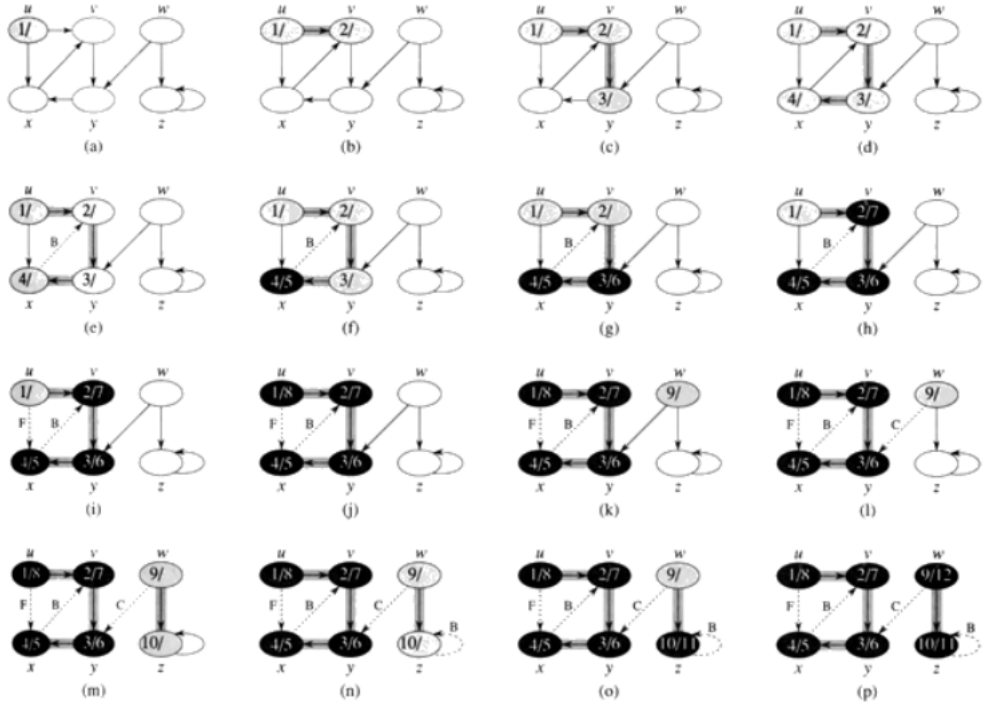
dall'ultimo vertice scoperto che abbia ancora vertici non visitati connessi.
Se rimangono dei vertici non raggiungibili dalla sorgente attuale, uno di questi diventa la nuova sorgente e viene ripetuto l'algoritmo.
L'algoritmo crea un albero **Depth-First**, o meglio una **Foresta DFS**, perchè l'algoritmo può essere eseguito da più sorgenti.
Inizialmente, tutti i nodi sono bianchi, perchè *non sono stati scoperti*. Diventano grigi *appena vengono scoperti*, e diventano neri *quando la loro visita è terminata*, cioè quando sono stati esplorati tutti i nodi adiacenti.
La DFS associa ad ogni vertice v due etichette: $v.d \wedge v.f$, che simboleggiano *l'istante di scoperta e l'istante di fine visita* del nodo. I valori di $v.d$ e $v.f$ appartengono all'intervallo $[1, 2|V|]$, in quanto ogni vertice *puo' essere scoperto e visitato completamente una sola volta*, e si avra' sempre che $u.d > u.f$

```
DFS(G)
    foreach u in G.V
        u.c<-WHITE
        u.p<-NIL
    time<-0
    foreach u in G.V
        if(u.c=WHITE)
            Visit(u)

Visit(u)
    time<-time+1
    u.d<-time
    u.c<-GRAY
    foreach v in u.adj
        if v.c=WHITE
            v.p<-u
            Visit(v)
    u.c<-BLACK
    u.f<-time<-time+1
```

ABCDEF



Il funzionamento step-by-step della DFS e' il seguente:



13.2 Cammini Minimi

Definiamo adesso il **cammino minimo**, $\delta(u, v)$ come *il minimo numero di archi in un cammino qualsiasi da un vertice u ad un vertice v*. Se non esiste alcun cammino tra i due vertici, $\delta(u, v) = \infty$.

13.2.1 Lemma Cammino Minimo

Se $G = (V, E)$ e' un grafo e $s \in V$ e' un vertice arbitrario, allora per qualsiasi arco $(u, v) \in E$ si ha che $\delta(s, v) \leq \delta(s, u) + 1$.

Dimostrazione Se u e' raggiungibile da s, allora anche v lo e'. Pertanto, il cammino minimo da s a v non potra' mai essere maggiore di $\delta(s, u) + 1$, dato che u e v sono collegati da un singolo arco (u, v), che quindi ha **distanza unitaria**.
Viceversa, se u non e' raggiungibile da s, allora $\delta(s, u) = \infty$, e la disuguaglianza rimane valida, perche' si avrebbe $\delta(s, u) = \infty \wedge \delta(s, v) = \infty \implies \infty \leq \infty$.

13.2.2 Lemma Limite Superiore

Sia $G = (V, E)$ un grafo e supponiamo che venga eseguita una **BFS** su tale grafo, da un vertice sorgente $s \in G.V$. Al termine della procedura, per ogni vertice $v \in G.V$, il valore di $d[v]$ soddisfa la relazione $\delta(s, v) \leq d[v]$.

DIMOSTRAZIONE

Si dimostra per induzione che $\delta(s, v) \leq d[v]$, per ogni $v \in G.V$.
Il caso base dell'induzione è quello in cui la sorgente s è stata inserita nella coda.
L'ipotesi è verificata, perchè $\delta(s, s) = 0 < \delta(s, v) \forall\ v \in G.V - \{s\}$.
Per il passaggio induttivo, consideriamo un nodo **bianco** v che viene scoperto durante la visita del nodo u. L'ipotesi induttiva implica che $\delta(s, u) \leq d[u]$.
Ma si ha, per ipotesi induttiva e per il *Lemma 1*, che $d[u] = d[u] + 1 \geq \delta(s, u) + 1 \geq \delta(s, v)$. Il vertice v sarà inserito nella coda e non sarà più reinserito *visto che viene colorato di grigio.*
Pertanto, il valore di $d[v]$ non cambierà più, e l'ipotesi resterà valida.

13.2.3 Lemma 3 [DA FINIRE]

13.3 Algoritmi per cammini minimi

13.3.1 Algoritmo di Dijkstra

L'algoritmo di Dijkstra effettua una visita sul grafo, **rilassando gli archi**, e trova i cammini minimi *dalla sorgente s a qualsiasi altro nodo.* L'algoritmo prende in input il grafo G ed il nodo s, e, *utilizzando una coda di priorita'*, estrae il vertice con distanza minore, se necessario rilassa l'arco, e poi controlla le sue adiacenze.
In tal modo, avremo che le distanze fra i nodi vengono aggiornate in base agli archi ed ai loro pesi. **NOTA BENE: Dijkstra funziona solamente con grafi con funzione peso** $w : \mathbb{R}^+ -> \mathbb{R}$.

```
Dijkstra(G,s)
    foreach (u in G.V)
        v.d<-INF
        v.p<-NIL
    s.d<-0
    Q<-empty queue
    ENQUEUE(Q,s)
    while(Q not empty)
        u<-EXTRACT-MIN(Q)
        foreach(v in u.adj)
            if(v.d>u.d+w(u,v))
```

```
v.d<-u.d+w(u,v)
v.p<-u
```

13.3.2 Algoritmo di Bellman-Ford

L'algoritmo di Bellman-Ford e' simile a quello di Dijkstra, con due differenze:

1. Opera su grafi con $w : \mathbb{R} - > \mathbb{R}$, quindi *anche negativi*
2. E' utile anche per trovare **cicli di peso negativo**

```
Bellman-Ford(G,s)
    foreach (u in G.V)
        v.d<-INF
        v.p<-NIL
    s.d<-0
    for i<-1 to |V|-1
        foreach({u,v} in G.E)
            if(v.d>u.d+w(u,v))
                v.d<-u.d+w(u,v)
                v.p<-u
    foreach({u,v} in G.E)
        if(v.d>u.d+w(u,v))
            return TRUE
        else
            return FALSE
```

L'algoritmo restituisce un valore booleano che indica *se esiste o meno un ciclo di peso negativo* ***raggiungibile dalla sorgente***. Infatti, il ritorno di FALSE indica che **non esistono cicli negativi**, in quanto *abbiamo applicato correttamente la procedura di rilassamento a tutti gli archi.* Se invece ritorna TRUE, **sappiamo che esiste un ciclo di peso negativo**, perche' *l'algoritmo ha trovato almeno un altro modo per ridurre il costo per raggiungere v, e quindi effettivamente* ***rilassare l'arco*** $\{u, v\}$, per qualche $u, v \in G.V$.
NOTA BENISSIMO: sulle slide ritorna FALSE se trova tale ciclo, ma per me, e per la mia logica, andrebbe ritornato TRUE. Quindi, scegliete il vostro modo di scrivere l'algoritmo, ma spiegategli perche' lo fate in quel modo!

13.4 Sottografo dei Predecessori

Il *sottografo dei predecessori* di una visita **DFS** e' il grafo $G_\pi = (V, E_\pi)$, dove $E_\pi = \{(\pi(v), v) : v \in V \ \wedge \ \pi[v] \neq NIL\}$.
Il sottografo forma una **foresta DF** composta da *vari alberi DF*, ognuno radicato nella sorgente da cui inizia la visita della DFS. Gli archi vengono quindi detti **Archi d'Albero**.

13.5 Classificazione degli Archi

L'algoritmo di DFS puo' essere usato per classificare gli archi che si incontrano nel grafo. Ad esempio, si ha che *Un grafo orientato e' aciclico solo se una visita in profondita' non genera* ***archi all'indietro***.
Possiamo definire quattro tipi di archi in un grafo:

1. Archi d'albero: Sono gli archi *della foresta DF*. L'arco $\{u, v\}$ e' d'albero se $v.c = WHITE$, ovvero se il nodo v e' scoperto per la prima volta durante la visita al nodo u adiacente

2. Archi all'indietro: Sono gli archi $\{u, v\}$ che collegano u *ad un antenato* v *in un albero DF*. I cappi, che possono presentarsi nei grafi **orientati**, *sono considerati archi all'indietro*. Sostanzialmente, il nodo v e' stato scoperto prima dell'attuale visita ad u, quindi si ha che $v.c = GRAY$

3. Archi in avanti: Sono gli archi $\{u, v\}$ che collegano u ad un *discendente* v in un albero DF. Avremo quindi che $u.d < v.d$, perche' il *time* in cui e' stato scoperto u e' minore del *time* in cui e' stato scoperto v, che e' un nodo nero. Quindi si ha che $u.d < v.d \& v.c = BLACK$

4. Archi trasversali: Sono tutti gli altri tipi di archi, quindi un arco $\{u, v\}$ con $u.c = v.c = BLACK$ con $u.d \geq v.d$

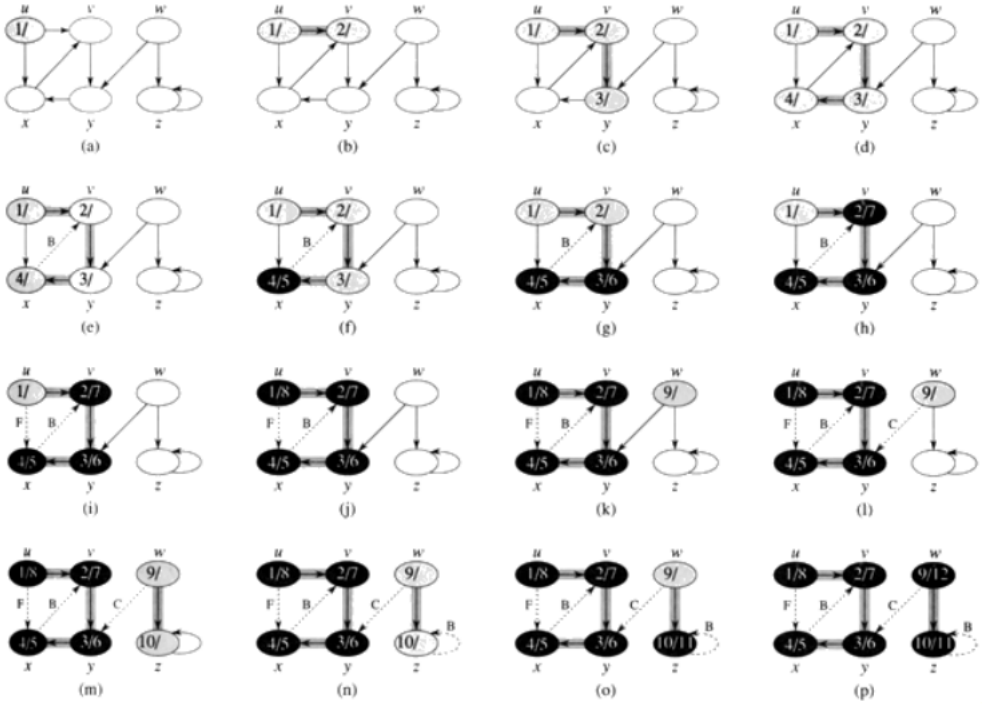
13.6 Componenti Connesse

Una componente connessa di un grafo $G = (V, E)$ e' un sottografo connesso massimo di G: $H = (V^1, E^1), V^1 \subseteq V \wedge E^1 \subseteq E$. Ciascun vertice appartiene ad un'unica componente connessa, e allo stesso modo ogni arco appartiente ad una sola componente connessa. Un grafo e' detto **Fortemente Connesso** se *contiene un cammino minimo orientato da u a v, per ogni coppia di vertici* $\{u, v\}$. Le componenti **Fortemente Connesse** sono i sottografi massimi.
L'algoritmo per trovare le componenti fortemente connesse utilizza il grafo **trasposto** $G^T = (V, E^T)$, in cui andremo a selezionare ciascun arco, ma **nel senso opposto**. Quindi, avendo nel grafo G un arco $\{u, v\}$, nel grafo G^T avremo un arco $\{v, u\}$.
Il seguente algoritmo calcola le componenti fortemente connesse in un tempo $\Theta(V + E)$:

```
SCC(G)
    //DFS Iniziale
    for each vertex u in G:
        color[u] = WHITE
    time = 0
    stack = [] //Stack per le SCC
    for each vertex u in G:
        if color[u] == WHITE:
            DFS_FirstPass(G, u, stack)
    GT = Transpose(G)
    for each vertex u in GT:
        color[u] = WHITE
    while stack is not empty:
        u = stack.pop()
        if color[u] == WHITE:
            current_component = []
            DFS_SecondPass(GT, u, current_component)
    //return current_component

DFS_FirstPass(G, u, stack):
    //DFS classica
    color[u] = GRAY
    for each vertex v in Adj[u]:
        if color[v] == WHITE:
            DFS_FirstPass(G, v, stack)
    color[u] = BLACK
    stack.push(u)

DFS_SecondPass(GT, u, current_component):
    //DFS sul trasposto
    color[u] = GRAY
    current_component.append(u)
    for each vertex v in Adj[GT, u]:
        if color[v] == WHITE:
            DFS_SecondPass(GT, v, current_component)
    color[u] = BLACK

//Semplice function per la trasposizione
Transpose(G):
    GT = new Graph()
```

```
    for each vertex u in G:
        for each vertex v in Adj[u]:
            GT.add_edge(v, u)
    return GT
```

Il procedimento dell'algoritmo e' il seguente:



13.7 Operazioni su Insiemi Disgiunti

Un insieme **Disgiunto** e' una struttura dati che mantiene una collezione $S = \{S_1, ..., S_k\}$, dove ogni insieme e' identificato da un **rappresentante**, che possiede un **grado**. Il rappresentante rende identificabile univocamente l'insieme d'appartenenza del dato i, mentre il grado rappresenta *il numero di dati all'interno dell'insieme* i.
Ogni elemento disgiunto e' rappresentato da un oggetto. Indicando con x un oggetto, vogliamo supportare le seguenti operazioni:

13.7.1 Make-Set

L'operazione di *Make-Set* crea un nuovo set partendo da un singolo oggetto, che sara' anche il rappresentante del set stesso. Dato che gli insiemi sono disgiunti, x non potra' trovarsi in altri set.

```
Make-Set(x)
    S<-new Set
    PUSH(S,x)
    x.rank<-1
    x.p<-NIL
```

13.7.2 Union

La *Union* unisce due set, ponendo come rappresentante l'oggetto con rank maggiore fra i rappresentanti dei set.

```
Union(X,Y)
    rootX<-Find-Set(X)
    rootY<-Find-Set(Y)
    if(rootX!=rootY)
        if(rootX.rank>=rootY.rank)
            rootY.p<-rootX
            rootX.rank<-rootX.rank+1
        else
            rootX.p<-rootY
            rootY.rank<-rootY.rank+1
```

13.7.3 Find-Set

L'operazione di *Find-Set* trova il rappresentante del set su cui e' chiamata

```
Find-Set(X)
    while(x.p!=NIL)
        x<-x.p
    return x
```

13.8 Componenti Connesse

Le componenti connesse di un grafo sono quindi trovabili mediante questa procedura:

```
SCC(G)
    foreach(v in G.V)
        Make-Set(v)
    foreach(u,v in G.E)
        if(Find-Set(u)!=Find-Set(v))
            Union(u,v)
```

14 Minimum Spanning Tree

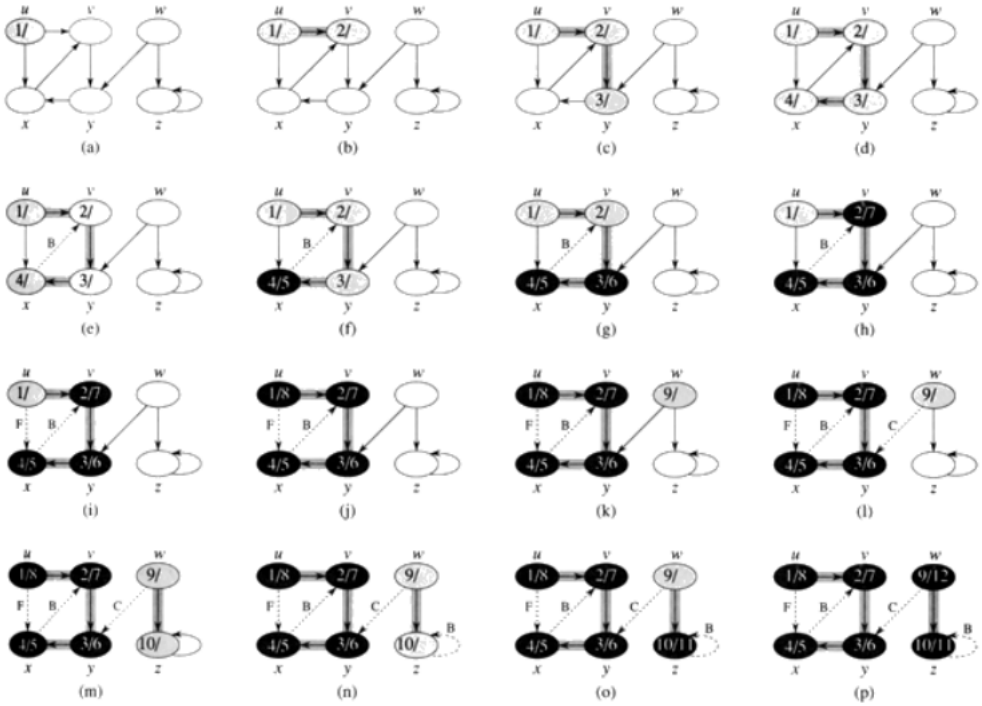
Dato un grafo connesso non orientato $G = (V, E)$, un **albero ricoprente massimo** (o *MST)*) di G e' un sottografo *che e ' un albero* e ricopre tutti i suoi nodi, avente **peso minore**, se assegnati pesi ai suoi archi.
Ogni grafo non orientato possiede una **Minimum Spanning Tree Forest**, ovvero una foresta che contiene *tutti i suoi MST* delle componenti connesse.
Esistono due algoritmi per il calcolo degli MST, ovvero gli algoritmi di 22.4 **Kruskal** e di **Prim**.
Definiamo quindi l'**arco sicuro** come un arco $\{u, v\} : A \cup \{u, v\}$ e' ancora parte di qualche albero di connessione minimo. Per caratterizzare gli archi sicuri, bisogna introdurre alcune definizioni:

1. **Taglio**: Insieme $(S, V - S)$ di un grafo $G = (V, E)$ che e' una *partizione di V in due insiemi disgiunti.*

2. Un arco $\{u, v\}$ **attraversa il taglio** se $u \in S \ \wedge \ v \in V - S$.

3. Un taglio **rispetta un insieme di archi A** se *nessun arco di A attraversa il taglio.*

4. Un arco che attraversa un taglio e' **leggero** nel taglio se *il suo peso e' minimo fra i pesi degli archi che attraversano il taglio*

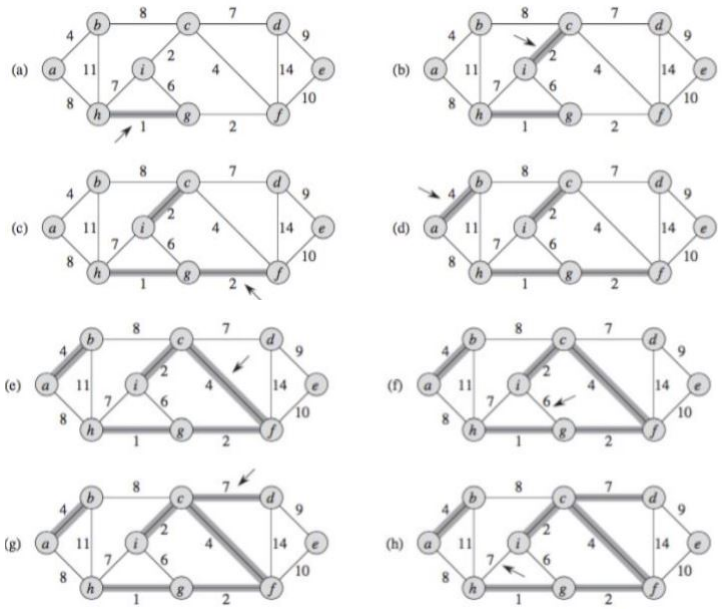

u v w
x y z
(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)
(m)
(n)
(o)
(p)

14.1 Kruskal

L'algoritmo di Kruskal e' uno degli algoritmi che trova l'MST di un grafo non orientato pesato.

```
KRUSKAL(G)
    A<-empty set
    foreach(v in G.V)
        Make-Set(v)
    Sort(E) //Ordina gli archi in ordine di peso non decrescente
    foreach(u,v in G.E)
        if(Find-Set(u)!=Find-Set(v))
            A = A U u,v
            Union(u,v)
    return A
```

Vediamo quindi il funzionamento dell'algoritmo:



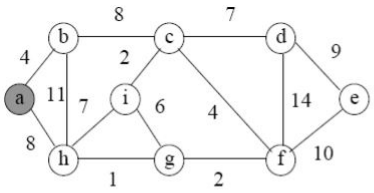
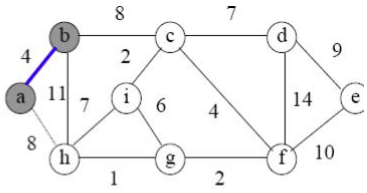
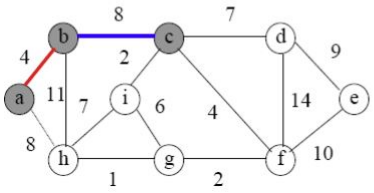
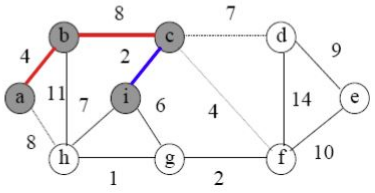
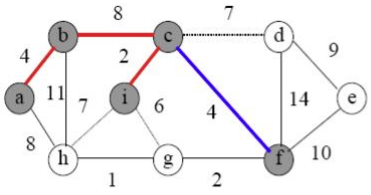
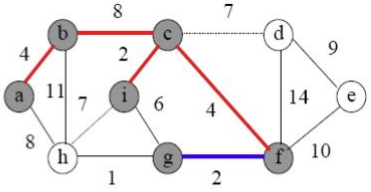
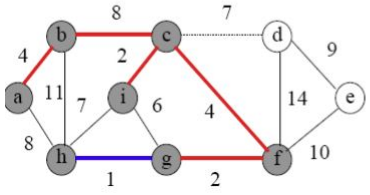
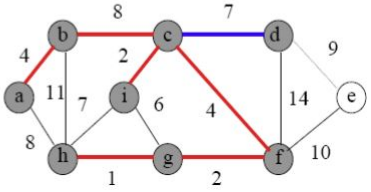
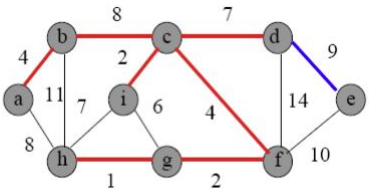
Per ogni nodo, l'algoritmo controlla se ciascun nodo fa parte della stessa componente connessa. Nel caso in cui faccia parte di una componente connessa diversa, le unisce e aggiunge l'arco u,v all'insieme A. Alla fine dell'esecuzione, l'insieme A contiene le componenti connesse del grafo, e quindi rappresenta l'MST.
La complessita' dell'algoritmo di Kruskal e' $O(E * \log(V))$

14.2 Prim

L'algoritmo di Prim, come quello di Kruskal, trova l'MST di un grafo G, pero' procede **mantenendo A in un singolo albero**.
Procede partendo dalla radice r, e cresce finche' non ricopre tutti i vertici. Ad ogni passo viene aggiunto un *arco leggero* che collega un vertice in VA con un vertice in $V - VA$, dove VA **e' l'insieme dei nodi raggiunti da archi in A**.

```
PRIM(G,r)
    Q<-G.V
    foreach(u in G.V-r)
        u.key<-INF
        u.p<-NIL
    r.key=0
    while(Q not empty)
        u<-EXTRACT-MIN(Q)
        foreach(v in u.adj)
            if(v in Q AND w(u,v)<key)
                v.p<-u
                u.key<-w(u,v)
```

L'algoritmo opera usando una coda di priorita', che seleziona sempre il nodo con chiave minore fra quelle disponibili, e controlla se per ogni arco uscente, vi e' un qualche arco che connette u a v, **avente peso minore**, e quindi lo impostiamo, per l'MST.
Ad ogni iterazione, i nodi che compongono l'MST vengono gradualmente rimossi dalla queue (**mediante l'operazione di** $EXTRACT - MIN(Q)$), finche' la coda stessa non sara' vuota. Ogni nodo estratto vedra' la sua *key* aggiornata, ed in tal modo non entrera' piu' nel *while*.

Il costo dell'algoritmo di Prim e' $O(E \log(V))$, asintoticamente pari a quello di Kruskal

15 Backtracking

Il backtracking e' una tecnica di programmazione applicata a:

1.Problemi **Decisionali**

2.Problemi **di Ricerca**

3.Problemi **di Ottimizzazione**

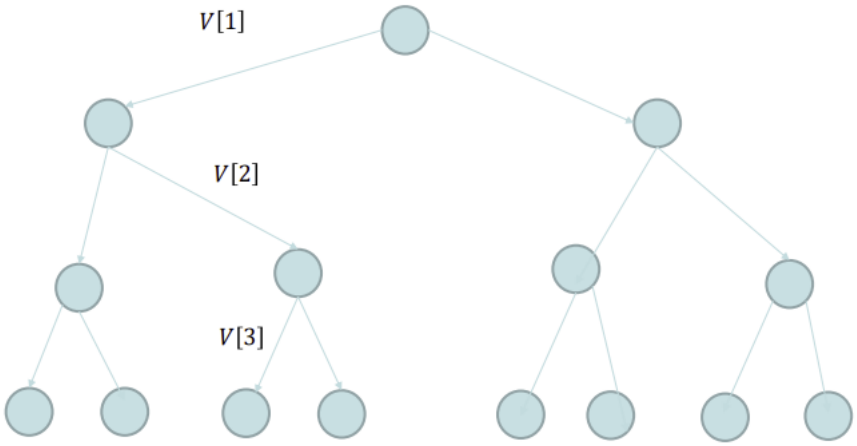
Il backtracking si basa sul concetto di **soluzione ammissibile per un problema**, ovvero una soluzione che rispetta alcuni criteri o vincoli.

15.1 Organizzazione Generale

Una soluzione viene rappresentata come un vettore $V[1, ..., N]$, il cui contenuto degli elementi $V[i]$ e' preso da un insieme S di scelte dipendente dal problema iniziale. Ad ogni passo, partiamo da una soluzione parziale $V[1, .., k]$, in cui $k \geq 1$ decisioni sono gia' state prese.
Se $V[1, ..., k]$ e' una soluzione ammissibile, la *processiamo*, ovvero eseguiamo le operazioni che dobbiamo fare per il problema (stampa, ritorno,..).
Se invece $V[1, ..., k]$ non e' una soluzione ammissibile, possiamo **ampliarla** con una delle possibili scelte in $V[1, ..., k + 1]$ oppure **cancelliamo l'elemento** k (*backtrack*) e ripartiamo dalla soluzione ammissibile $V[1, .., k - 1]$.

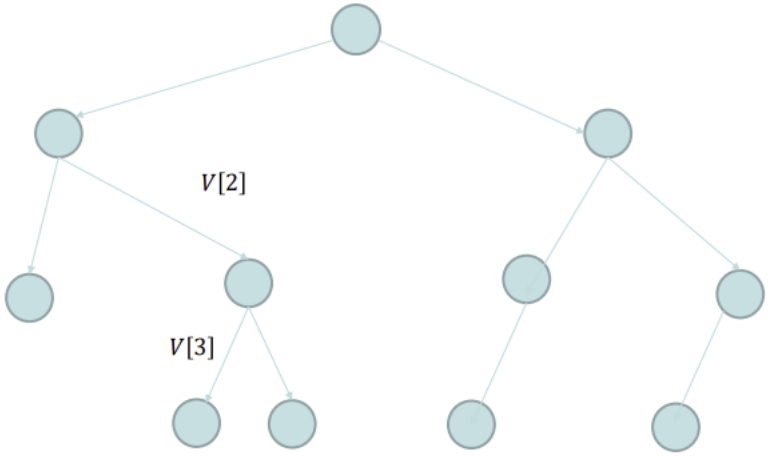
15.2 Albero delle decisioni

L'albero delle decisioni e' l'albero che rappresenta le vie che percorriamo durante l'esecuzione dell'algoritmo, per trovare la soluzione al problema.
Esso rappresenta lo **spazio di ricerca**, ovvero lo spazio in cui andiamo a cercare la soluzione al problema iniziale. La radice identifica la **soluzione parziale vuota**; i nodi interni le **soluzioni parziali**; le foglie sono **le soluzioni ammissibili**, che combinate daranno la soluzione integrale al problema.



15.2.1 Pruning

Il pruning e' l'eliminazione delle soluzioni *non ammissibili* ad un problema S. L'eliminazione di tali soluzioni permette di avere solo le soluzioni ammissibili, e quindi di poter ricavare la soluzione ammissibile all'intero problema.



15.3 Approccio Backtracking

Esistono due approcci per la programmazione Backtracking:

1. Ricorsivo : Lavora mediante **una visita in profondita'** nell'albero delle scelte, ed e' *basata su un approccio ricorsivo*

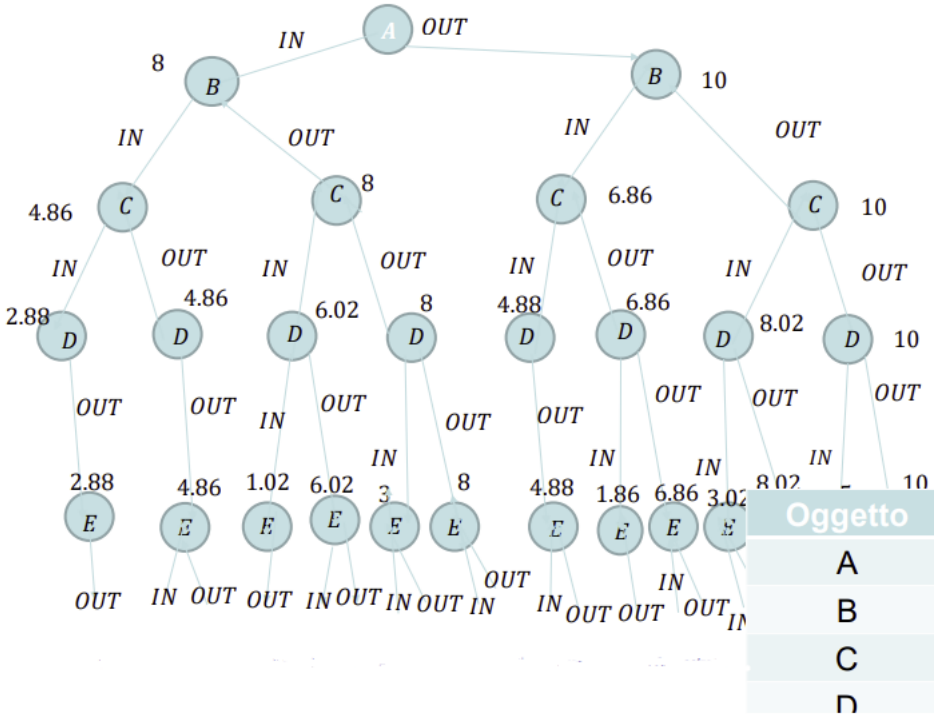
2. Iterativo : Utilizza un approccio **Greedy**, eventualmente *tornando sui propri passi*

15.3.1 0_1Knapsack-Backtracking

Avendo il seguente problema dello zaino 0_1:

Oggetto	Peso	Valore
A	2.00 Kg	40 $
B	3.14 Kg	50 $
C	1.98 Kg	100 $
D	5.00 Kg	95 $
E	3.00 Kg	30 $

La soluzione, mediante Backtracking ed Albero delle Decisioni, sara' la seguente:



16 Branch-And-Bound

La B&B e' una tecnica di programmazione per risolvere **problemi di Ottimizzazione**. Consiste nell'*esplorare in maniera intelligente* tutte le possibili soluzioni del problema di ottimizzazione, ovvero *l'albero delle soluzioni*.
Per esplorare in maniera intelligente l'albero, la B&B ha al suo interno un meccanismo di previsione, detto **LookAhead**, che gli consente di capire se *il ramo sottostante puo' portare ad una soluzione migliore di quella corrente*. Se cio' e' vero, lo percorre, altrimenti *effettua il pruning* rinunciando a percorrerlo.

16.1 Upper Bound

Per implementare il meccanismo di previsione, la B&B utilizza un limite superiore, detto **Upper Bound**, ossia *determina e stabilisce un limite superiore al valore che la soluzione puo' assumere*. Durante l'esecuzione, la tecnica del B&B effettua il **rilassamento**, per aggiornare tale limite.

16.2 Esecuzione

La tecnica del B&B organizza l'attivita' di ricerca in modo che *i nodi dell'albero siano espansi uno alla volta*. Ad ogni nodo interno e' associato l'upper bound della soluzione, per cui **e' espanso solo se ha il potenziale di ottenere una soluzione migliore**, ed in tal caso e' chiamato **attivo**. Il potenziale e' ottenuto calcolando *l'upper bound sul valore delle soluzioni contenute nel sottoalbero, radicato nel nodo interno*.
Le parti del B&B sono la **Branching**, ovvero *l'esplorazione del sottoalbero radicato nel nodo attivo*, e la **Bounding**, ovvero *stimare l'upper bound sul valore delle soluzioni nel sottoalbero radicato nel nodo attivo*.
L'albero delle soluzioni puo' essere esplorato usando:

1. DFS : **soluzioni accettabili costruite prima**
2. BFS : **fare pruning presto**
3. Best-First-Search : **gli insiemi con migliori upper bound sono esplorati prima**

16.3 Esempio Zaino 01

Riferendoci al problema precedente, dello zaino 01:

Oggetto	Peso	Valore
A	2.00 Kg	40 $
B	3.14 Kg	50 $
C	1.98 Kg	100 $
D	5.00 Kg	95 $
E	3.00 Kg	30 $

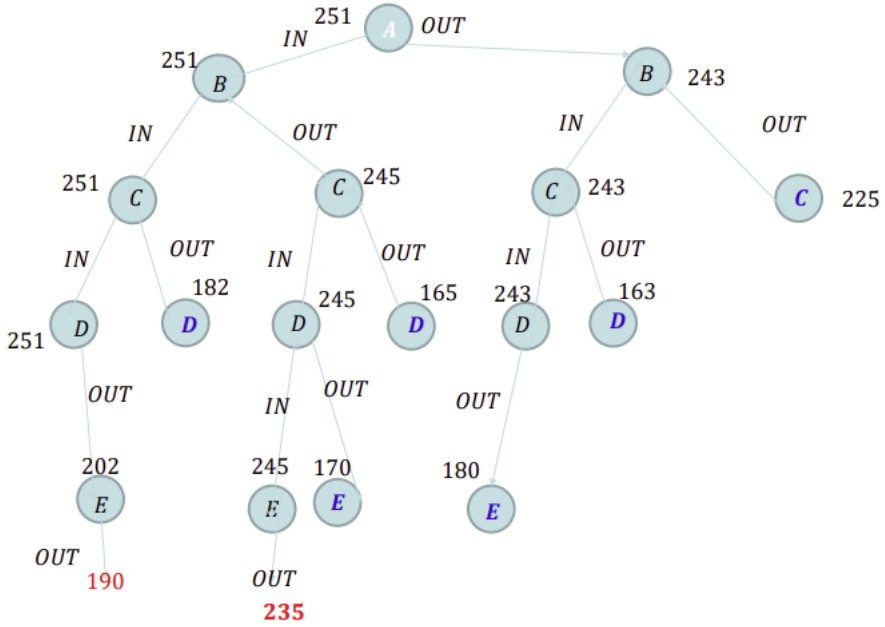
Dobbiamo inizialmente stimare l'**Upper Bound**, che e' *il valore che otterremmo con lo zaino frazionario usando la tecnica* ***Greedy***:
$100\$ + 40\$ + 95\$ + \frac{10-8.98}{3.14} * 50\$ = 251.24\$$

Scegliendo i pesi: $8.02Kg/6.02Kg/1.02Kg$

$40\$ + 100\$ + 95\$ + \frac{1.02}{3} * 30\$ = (235 + 10.2)\$ = 245.2\$$

$100\$ + 95\$ + 30\$ = 225\$$

Avremo quindi tale soluzione:



17 Automi a Stati Finiti

Gli Automi a Stati Finiti sono modelli di molte categorie di HW e SW, definiscono l'andamento rispetto alla ricerca di una soluzione di un algoritmo.
Un automa riceve un **input** quando si trova in uno **stato**, e mediante una **funzione di trasformazione**, dallo stato iniziale si puo' ritrovare in un altro stato, detto **stato accettante**. La sequenza di input che porta l'automa in tale stato e' considerata valida.

17.1 Alfabeto

E' un insieme di simboli non vuoti, indicato con la lettera Σ. Qualche esempio di Alfabeto e' l'alfabeto binario $\Sigma = \{0, 1\}$, l'insieme delle lettere $\Sigma = \{a, b, .., z\}$.

17.2 Stringa

Una stringa e' una sequenza finita di simboli, scelti da un alfabeto. Alcuni esempi sono dei numeri rappresentati in binario, come 0110, dall'alfabeto $\Sigma = \{0, 1\}$, oppure delle parole, come *ciao*, dall'alfabeto $\Sigma = \{a, b, .., z\}$. La stringa vuota e' una stringa composta da nessun simbolo, e puo' essere scelta da *qualunque alfabeto*.

17.3 Potenze di un alfabeto

La **potenza** di un alfabeto, indicata con Σ^k, e' l'insieme delle stringhe di lunghezza k tratte dall'alfabeto Σ. Alcuni esempi sono $\Sigma^2\{0, 1\} = \{00, 01, 10, 11\}$, $\Sigma^3\{0, 1\} = \{000, 001, 010, 011, 100, 101, 110, 111\}$.
La potenza Σ^0 e' la stringa nulla ϵ

17.3.1 Insieme delle stringhe

L'insieme di tutte le stringhe di un alfabeto, di qualunque potenza, e' l'insieme $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup ...$

17.4 Linguaggio

Il linguaggio di un Automa a Stati Finiti e' **l'insieme delle stringhe scelte da Σ^* accettate dall'automa**, ovvero tutte quelle stringhe che, date in input all'automa, lo portano in uno degli stati accettanti. Si ha che: **Se Σ e' un alfabeto e $L \subseteq \Sigma^*$, allora L e' un linguaggio su Σ.** Notare che L **non deve per forza contenere tutti i simboli di** Σ.

17.5 Stella di Kleene

La **Stella di Kleene**, o *chiusura*, di un linguaggio L, e' il linguaggio $L^* = \{\{\epsilon\} \cup L \cup L^2 \cup ....\}$

18 Definizione formale di Problema

Se Σ e' un alfabeto, e L e' un linguaggio su Σ, il problema L e':
Data una stringa w in Σ^*, decidere se $w \in L$

18.1 Automa a Stati Finiti Deterministico - DFA

Un **DFA** e' un automa che consiste delle seguenti componenti:

1. Un insieme **finito** Q di stati
2. Un insieme **finito** Σ di simboli
3. Una **funzione di stato** $\delta : Q\ x\ \Sigma -> Q$
4. Uno stato **iniziale** q_0
5. Un insieme di stati **finali** accettanti $F \subseteq Q$

18.1.1 Funzione di Transizione Estesa

La **Funzione di Transizione Estesa** e' una funzione che prende in input uno stato iniziale q_0 e una stringa w, e restituisce uno stato p, ossia *lo stato che l'automa raggiunge quando, dallo stato q_0, elabora la sequenza di input w*

18.2 Linguaggio DFA

Il linguaggio di un DFA e' definito nel seguente modo:
Dato un DFA $A = (Q, \Sigma, \delta, q_0, F)$, il linguaggio di tale DFA, indicato con $L(A)$ e' definito da:
$L(A) = \{w \mid \hat{\delta}(q_0, w) \in F\}$
In pratica, il linguaggio di un DFA e' l'insieme delle stringhe che, partendo da uno stato inziale, portano l'automa in uno dei suoi stati accettanti, dandogli in input la sequenza dei caratteri della stringa, che appartengono tutti al suo alfabeto.

18.3 Automa a Stati Finiti Non Deterministico - NFA

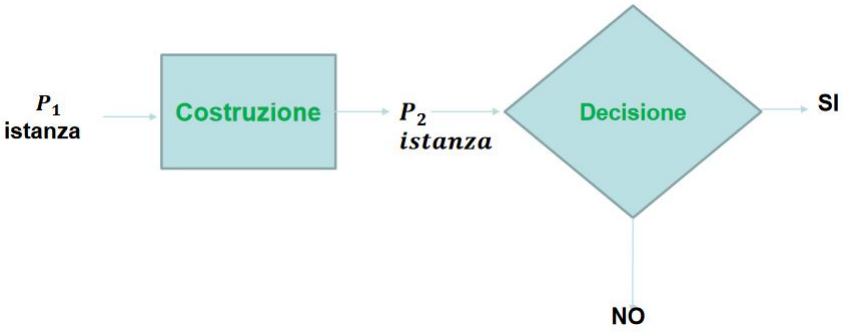
Un **NFA** e' un automa che, presa in input una sequenza di simboli del suo linguaggio, **non raggiunge un unico stato, ma puo' raggiungerne diversi, racchiusi in un insieme**.
Formalmente: $A = (Q, \Sigma, q_0, F, \delta : Q\ x\ \Sigma -> P \subseteq Q)$.
Definiamo il **Linguaggio** di un NFA come:
Dato un **NFA** $A = (Q, \Sigma, \delta, q_0, F)$, si ha che $L(A) = \{w | \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$

18.4 Problema Decidibile

Se un problema ha un algoritmo che ci dice sempre in maniera corretta se un'istanza del problema riceve la risposta *si* o *no*, allora il problema e' **decidibile**. Altrimenti, e' detto **indecidibile**.
In parole povere, un problema e' decidibile se *esiste un algoritmo che, preso in input il problema,* ***riesca a decidere se la soluzione e' coretta oppure no***.
Un problema viene detto **indecidibile** se *non puo' essere risolto da alcun computer*, mentre viene detto **astratto** un problema basato sulla *relazione* ***binaria*** *sull'insieme I delle* ***istanze*** *del problema e l'insieme S delle* ***soluzioni*** *del problema.*
In pratica, un problema astratto e' un problema logico in cui si ricerca la soluzione in base ad una relazione logica fra ciascuna istanza ed una sua possibile soluzione (*se A, allora B...*).

18.5 Riducibilita' di un Problema

Un problema P_1 e' detto **riducibile** *in un altro problema* P_2 se *esiste una funzione f che trasforma* ***un'istanza di*** P_1 *in* ***un'istanza di*** P_2.



Nell'esempio, l'istanza del problema P_1 viene costruita, e la stringa w del problema viene convertita in una stringa w' per il problema P_2. Se P_2 **accetta** la stringa w', allora significa che $w' \in L(P_2) \implies w \in L(P_1)$.

18.6 Tesi di Church

La **Tesi di Church-Turing**, che *e' indimostrabile*, assume che **qualunque modo generale di computare permette di computare** ***solo cio' che la macchina di Turing e' in grado di calcolare***.
In parole povere, non esistono problemi risolvibili da compilatori che non possono essere risolti dallo schema della Macchina di Turing.

19 Problemi P-NP

I problemi che si affrontano durante un corso di ASD sono detti *polinomiali*, in quanto **possono essere risolti in tempo polinomiale** ($O(n^k)$).
Vengono detti **trattabili** tali problemi, mentre i problemi *non risolvibili in tempo polinomiale sono detti* ***intrattabili***.
Esiste poi una classe di problemi, detti **NP**, che comprende tutti i problemi la cui soluzione *non e' calcolabile in tempo polinomiale* ($O(a^n), O(e^n), ...$).
Tra le due classi di problemi, ve ne e' una *il cui stato non e' conosciuto*. Si tratta dei problemi **NP-Completi**, per cui **non e' ancora stato trovato un algoritmo con tempo polinomiale che risolva un problema NPC**, ne' si e' riusciti a fornire **un limite inferiore con tempo *superpolinomiale* per risolvere un problema NPC**.

19.0.1 Classe P

La **Classe di Complessita' P** e' definita come:
L'insieme dei problemi di decisione concreti risolvibili *in tempo polinomiale*.
Si dice che una funzione $f : \{0,1\}* \to \{0,1\}*$ e' **calcolabile in tempo polinomiale** se *esiste un algoritmo A che,* ***dato come input un*** $x \in \{0,1\}*$*, produce come output* $f(x)$ ***in tempo polinomiale***.
Definiamo il **linguaggio accettato**, o **riconosciuto**, da un algoritmo A come l'insieme: $L = \{x \in \{0,1\}* : A(x) = 1\}$. Se $A(x) = 0$, diremo che A **rifiuta la stringa** x.
Un linguaggio L e' **deciso** da A se **ogni stringa binaria in** L **e' *accettata* da** A **e ogni stringa binaria non in** L **e' *rifiutata* da** A.

19.0.2 Algoritmo di Verifica

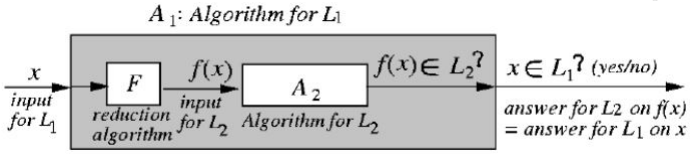
Si definisce **Algoritmo di Verifica** un algoritmo A a due argomenti, dove il primo e' una stringa x di input, e l'altro e' **una stringa binaria** y, detta **certificato**.
Si dice che un algoritmo A a due argomenti **verifica** una stringa x di input se **esiste un certificato** $y : A(x, y) = 1$.
Il linguaggio verificato da un algoritmo di verifica A e' $L = \{x \in \{0,1\}*, y \in \{0,1\}* : A(x, y) = 1\}$. Intuitivamente, l'algoritmo verifica solo le x per cui esiste una y tale che $A(x, y) = 1$. Per qualunque $x \notin L$, **non esiste** y **che dimostri che** $x \in L$.

19.1 Classe NP

La **Classe di Complessita' NP** (*tempo polinomiale non deterministico*) e' la classe **degli algoritmi che *possono essere verificati* in tempo polinomiale**. Un linguaggio $L \in NP \iff \exists L = \{x \in \{0,1\}*, y : |y| = O(|x|^c) : A(x,y) = 1\}$. In tal caso, si dice che **l'algoritmo A verifica il linguaggio L in tempo polinomiale**.
Bisogna tenere a mente che **Se un linguaggio appartiene alla classe P, apparterra' anche alla classe** NP ($L \in P \implies L \in NP$)
. Infatti, se esiste un algoritmo A che in tempo **polinomiale** decide il linguaggio L, allora e' immediato trasformare A in un algoritmo di verifica:
*Bastera' infatti far si' che A **ignori** qualunque certificato gli venga fornito*, per cui **accettera' solo le** $x \in L$, e rifiutera' le altre. Quindi, abbiamo dimostrato che $P \subseteq NP$.

20 Riducibilita' dei Problemi

Come detto in precedenza, *si puo' ridurre un'istanza di un problema P_1 ad un'istanza di un problema P_2* se **una qualsiasi istanza di P_1 puo' essere riformulata come un'istanza di P_2, e *la soluzione di P_2 fornisce una soluzione all'istanza di P_1***.
Si dice che un linguaggio L_1 e' **riducibile in tempo polinomiale** ad un linguaggio L_2, e si scrive $L_1 \leq_p L_2$, se **esiste unna funzione calcolabile in tempo polinomiale** $f : \{0,1\}* \to \{0,1\}* : \forall x \in L_1 f(x) \in L_2$. Cio' significa che **l'output di f(x) e' accettato come input per** L_1.
Se $L_1, L_2 \subseteq \{0,1\}*$ sono linguaggi tali che $L_1 \leq_p L_2 \implies L_2 \in P \implies L_1 \in P$.



Per un input $x \in \{0,1\}*$, A_1 usa F per trasformare x in $f(x)$, e quindi usa A_2 per verificare se $f(x) \in L_2$. L'output di A_2 e' il valore fornito come output da A_1.

21 Classe NPC

La classe dei problemi NP-Completi e' stata introdotta nel 1970, e racchiude tutti quei problemi Q per cui valgono **entrambe** le seguenti proprieta':

$$Q = \begin{cases} L \in NP \\ L' \leq_p L, \forall L' \in NP \end{cases} \tag{17}$$

Se un linguaggio L soddisfa la proprieta' 2, ma non la 1, e' detto **NP-Hard**.

21.1 Relazione P-NP

Se un qualsiasi problema NP-C e' risolvibile in tempo polinomiale, allora $P = NP$. Viceversa, se un qualsiasi problema NP-C non e' risolvibile in tempo polinomiale, nessun problema NP-C lo sara'.
Si dimostra cio' su due vie:
Se $L \in P$ ed $L \in NP - C$, allora $\forall L' \in NP - CL' \leq_p L$, per *definizione di NP-C*. Si ha quindi che $L \in P \implies L' \in P$.
Al contrario, supponiamo $\exists L \in NP : L \notin P$. Supponiamo **per assurdo** che $\exists L' \in NP : L' \in P$. Allora si avrebbe $L \leq_p L'$, da cui segue $L \in P$, che contraddice l'ipotesi iniziale.

21.2 Teorema NP-C e NP-H

Se L e' un linguaggio tale che $L' \leq_p L$, per qualche $L' \in NPC$, allora L e' **NP-Hard**. Inoltre, se $L \in NP$, allora $L \in NP - C$.

22 Algoritmi C++ (*2° scritto*)

22.1 Grafi

I grafi sono una struttura dati che racchiude un insieme di **vertici** e **archi**.
I nodi sono strutture che racchiudono diverse informazioni, come : *padre*,*colore*,*distanza*,...
Gli archi collegano due vertici, e il loro ordine è importante a seconda se il grafo è **orientato** o meno.
Più in generale, si può implementare un grafo in questo modo:
Queste le librerie necessarie, della **STL** (*Standard Template Library*)

```
#include <iostream>
#include <queue>
#include <vector>
#include <list>
#include <queue>
#include <map>
#include <unordered_map>
using namespace std;
```

Esse infatti ci permettono di *utilizzare* ***in modo efficiente*** *le strutture dati per realizzare le adiacenze dei nodi, i collegamenti degli archi, la coda e creare la struttura del grafo stesso.*

```
//Enum per gestire i colori del nodo
enum COLOR{WHITE,GRAY,BLACK};

class Grafo{
private:
    //Struct per gestire i nodi
    struct Nodo{
        int key,d,f;
        COLOR c;
        Nodo* p;
        list<Nodo> adj;
        Nodo(int k=0):key(k){
            this->c=WHITE;
            this->p=nullptr;
        }
    };
    //Struct per gestire gli archi
    struct Arco{
        int to,from;
        Arco(int t,int f):to(t),from(f){}
    };
    int V,E;
    //Unordered map per gestire i grafi NON ORIENTATI
    unordered_map<int,Nodo> nodi;
    vector<Arco> archi;
public:
    Grafo(){}
    void AddNodo(int k){
        Nodo n = Nodo(k);
        nodi[k] = n;
        //DEBUG: cout << "Nodo con chiave: " << k << " aggiunto" << endl;
    }
    void AddArco(int s,int e){
        if(nodi.find(s)!=nodi.end() && nodi.find(e)!=nodi.end()){
            Arco a = Arco(s,e);
            nodi[s].adj.push_back(nodi[e]);
            nodi[e].adj.push_back(nodi[s]);
            archi.emplace_back(a);
        }
        else{
```

```
            cout<<"Almeno uno dei nodi non presente"<<endl;
        }
    }
    void BFS(int s);
    void DFS();
    void Visit(Nodo& u, int& time);
    void prt_graph_BFS();
    void prt_graph_DFS();
    ~Grafo(){}
};

void Grafo::BFS(int s){
    Nodo start = Nodo(nodi[s]);
    queue<Nodo> q;
    for(auto it=nodi.begin();it!=nodi.end();++it){
        it->second.c = WHITE;
    }
    start.d=0;
    start.c = GRAY;
    q.push(start);
    while(!q.empty()){
        Nodo u = q.front();
        q.pop();
        for(auto it2=u.adj.begin();it2!=u.adj.end();it2++){
            Nodo v = *it2;
            if(v.c==WHITE){
                v.c = GRAY;
                q.push(v);
            }
        }
        u.c = BLACK;
    }
}

void Grafo::DFS(){
    int time=0;
    for(auto& it : nodi){
        it.second.c=WHITE;
        it.second.d=it.second.f=time;
        it.second.p=nullptr;
```

```
    }
    for(auto& it : nodi){
        if(it.second.c==WHITE){
            Visit(it.second,time);
        }
    }
}
void Grafo::Visit(Nodo& u,int& time){
    time+=1;
    u.d=time;
    u.c=GRAY;
    for(auto it=u.adj.begin();it!=u.adj.end();it++){
        if(it->c==WHITE){
            Visit(*it,time);
            it->p=&u;
        }
    }
    u.c=BLACK;
    time+=1;
    u.f=time;
    //DEBUG : cout << "Nodo: " << u.key << ".d= " << u.d <<", "<< u.key <<" .f= " <<
}

void Grafo::prt_graph_BFS() {
    cout << "Nodi del grafo e adiacenze:" << endl;
    for (const auto& n : nodi) {
        cout << "Nodo " << n.first << ": ";
        for (const auto& adj : n.second.adj) {
            cout << adj.key << " ";
        }
        cout << endl;
    }
    cout << endl;
    cout << "Archi del grafo:" << endl;
    for (const auto& a : archi) {
        cout << a.from << " -> " << a.to << endl;
    }
}

void Grafo::prt_graph_DFS() {
```

```
    for(const auto& n : nodi){
        cout << "Nodo: " << n.second.key << " D: " << n.second.d << " F: " << n.secon
    }
    cout << "Archi del grafo:" << endl;
    for (const auto& a : archi) {
        cout << a.from << " -> " << a.to << endl;
    }
}

int main(){
    Grafo G = Grafo();
    //Inserimento dei nodi nel grafo
    G.AddNodo(0);
    G.AddNodo(1);
    G.AddNodo(2);
    G.AddNodo(3);
    G.AddNodo(4);
    //Inserimento degli archi nel grafo
    G.AddArco(0,2);
    G.AddArco(0,1);
    G.AddArco(1,2);
    G.AddArco(2,4);
    G.AddArco(1,3);
    G.AddArco(4,2);
    G.AddArco(3,2);
    G.AddArco(0,3);
    cout<<endl;
    //Run della BFS
    cout << "BFS sul grafo, inizio col nodo 0" << endl;
    G.BFS(0);
    cout << endl;
    G.prt_graph_BFS();
    cout << endl;
    cout << "DFS sul grafo" << endl;
    G.DFS();
    cout << endl;
    G.prt_graph_DFS();
    return 0;
}
```

22.2 Esercizi sui grafi

22.2.1 Contare numero cicli

```
int Grafo::DFS(){
    int time=0,cycles=0;
    for(auto& it : nodi){
        it.second.c=WHITE;
        it.second.d=it.second.f=time;
        it.second.p=nullptr;
    }
    for(auto& it : nodi){
        if(it.second.c==WHITE){
            Visit(it.second,time,cycles);
        }
    }
    return cycles;
}
int Grafo::Visit(Nodo& u,int& time,int& cycles){
    time+=1;
    u.d=time;
    u.c=GRAY;
    for(auto it=u.adj.begin();it!=u.adj.end();it++){
        if(it->c==WHITE){
            Visit(*it,time);
            it->p=&u;
        }else{
            if(v!=u){
                cycles++;
            }
        }
    }
    u.c=BLACK;
    time+=1;
    u.f=time;
    return cycles;
}
```

22.2.2 Ordinamento Topologico

L'ordinamento topologico è un ordinamento che ordina i nodi **in base al tempo** $n.f$ **decrescente**.
L'ordinamento utilizza uno **stack**, per mantenere i nodi, e vengono inseriti in ordine di fine visita, appena dopo l'istruzione $u.f = time$.
L'implementazione fa riferimento alla stessa classe vista nella sezione 22.1

```
void Grafo::topologico(){
    int time=0;
    stack<Nodo*> topoSort = stack<Nodo*>();
    for(auto& it : nodi){
        if(it.second.c==WHITE)
            //DEBUG: cout << "Nodo: " << it.second.key << ", eseguo la visita" << end
            topoVisit(&it.second,topoSort,time);
    }
    cout << "Stampo lo stack" << endl;
    prt_topo(topoSort);
}
void Grafo::topoVisit(Nodo* u,stack<Nodo*>& s,int& time){
    u->c=GRAY;
    u->d=time;
    time+=1;
    for(auto& it : u->adj){
        if((*it).c==WHITE)
            //DEBUG: cout << "Nodo: " << it.key << " bianco, eseguo la visita" << end
            topoVisit(it,s,time);
    }
    u->c=BLACK;
    u->f=time;
    time+=1;
    s.push(u);
}
void Grafo::prt_topo(stack<Nodo*>& s){
    while(!s.empty()){
        Nodo* topNode = s.top();
        s.pop();
        cout << "Nodo: " << topNode->key << ", n.d= " << topNode->d << ", n.f= " << t
    }
}
int main(){
```

```
    Grafo G = Grafo();
    //Aggiunta dei nodi
    G.AddNodo(0);
    G.AddNodo(1);
    G.AddNodo(2);
    G.AddNodo(3);
    G.AddNodo(4);
    G.AddNodo(5);
    //Aggiunta degli archi
    G.AddArco(5,2);
    G.AddArco(5,0);
    G.AddArco(4,0);
    G.AddArco(4,1);
    G.AddArco(2,3);
    cout << endl;
    //Esecuzione del sort topologico
    G.topologico();
    return 0;
}
```

22.2.3 Dijkstra/Bellman Ford

Gli algoritmi di **Dijkstra** e di **Bellman Ford** eseguono entrambi il *rilassamento* degli archi, ovvero *riducono la distanza del nodo v al valore* ***della somma degli archi necessari a raggiungerlo***.
La differenza fra gli algorimti è che **Bellman Ford lavora con pesi anche negativi**, e *restituisce FALSE se esiste un ciclo di peso negativo.*
Entrambi questi algoritmi fanno riferimento alla classe Grafo riportata qui 22.1

```
//Function di Dijkstra per cammini minimi e rilassamento
void Grafo::Dijkstra(int s){
    //Creo la function di compare che confronta le distanze dei nodi
    auto compare = [](const Nodo* a, const Nodo* b){return a->d>b->d;};
    //Creo la priority_queue usando la compare che ho implementato
    priority_queue<Nodo*,vector<Nodo*>,decltype(compare)> pq(compare);
    //Inizializzazione dei nodi
    for(auto& it : nodi){
        it.second.d=INT_MAX;
        it.second.p=nullptr;
    }
    //Impongo che il nodo da cui parto abbia distanza nulla, per poterlo estrarre com
    nodi[s].d=0;
    pq.push(&nodi[s]);
    //Ciclo finchè la coda non è vuota
    while(!pq.empty()){
        Nodo* u = pq.top();
        pq.pop();
        //Per ogni arco uscente da u (tutte le adiacenze)
        for(auto& [v,peso] : u->adj){
            //Se posso rilassare un arco, lo faccio, ed imposto il padre di v
            if(v->d > u->d+peso){
                v->d = u->d+peso;
                v->p = u;
                pq.push(v); //Inserisco v in coda per poterlo estrarre successivament
            }
        }
    }
}
//Function di Bellman-Ford per verificare la presenza di cicli di peso negativo
bool Grafo::BellmanFord(int s){
    //Inizializzazione dei nodi
```

```
    for(auto& [_,nodo]: nodi){
        nodo.d=INT_MAX;
        nodo.p=nullptr;
    }
    //Impongo che il nodo da cui parto abbia distanza nulla, per poterlo estrarre com
    nodi[s].d=0;
    //Rilassamento degli archi n-1 volte
    for(int i=0;i<nodi.size()-1;i++){
        //Itero su tutti gli archi del grafo
        for(const auto& [u,v,p] : archi){
            //Se posso rilassare un arco, lo faccio
            if(nodi[v].d>nodi[u].d+p){
                nodi[v].d=nodi[u].d+p;
                nodi[v].p=&nodi[u];
            }
        }
    }
    //Per ogni arco
    for(auto& it : archi){
        int u=it.from,v=it.to,w=it.w;
        //Se trovo che un arco è ancora rilassabile, significa che nel grafo c'è un c
        // di peso negativo, perchè continuerebbe ad essere rilassabile all'infinito
        if(nodi[u].d+w<nodi[v].d){
            return false;
        }
    }
    return true;
}
```

22.2.4 Algoritmo di Prim

L'algoritmo di Prim costruisce il **Minimum Spanning Tree** dal grafo G, ovvero l'albero che comprende tutti i nodi del grafo, con costo minimo.
L'algoritmo seguente fa sempre riferimento alla classe: 22.1

```
void Grafo::Prim(int r){
    auto compare=[](pair<Nodo*,int> a,pair<Nodo*,int> b){
        return a.second>b.second;
    };
    priority_queue<pair<Nodo*,int>,vector<pair<Nodo*,int>>,decltype(compare)> queue(c
    for(auto& [_,nodo] : nodi){
        nodo.key=INT_MAX;
        nodo.p=nullptr;  // Aggiungi questa riga
    }
    unordered_map<Nodo*,bool> inMST;
    nodi[r].key=0;
    queue.push({&nodi[r],0});
    while(!queue.empty()){
        Nodo* u = queue.top().first;
        queue.pop();
        if(inMST[u]) continue;
        inMST[u]=true;
        for(auto& [v,w] : u->adj){
            if(!inMST[v] && w<v->key){
                v->p=u;
                v->key=w;
                queue.push({v,w});
            }
        }
    }
    prt_MST(inMST);
}

void Grafo::prt_MST(const unordered_map<Nodo*,bool>& inMST) {
    cout << "Stampa dell'MST" << endl;
    int totW = 0;
    for (const auto& [k, n] : nodi) {
        if (n.p != nullptr && inMST.at((Nodo*)&n)){
            int edgeW = 0;
            for (const auto& [v, w] : n.adj) {
```

```
                if (v == n.p) {
                    edgeW = w;
                    break;
                }
            }
            totW += edgeW;
            cout << "Arco: (" << n.key << "," << n.p->key << "), w: " << edgeW << end
        }
    }
    cout << "Peso totale dell'MST: " << totW << endl;
}
```

22.3 Esercizi d'Esame - Tracce vecchie

```
//              MAGGIO 2023
/*
Usando il paradigma della programmazione ad oggetti ed linguaggio C++ 11 progettare e
    di caricare dati di un grafo non orientato pesato G contenuti nel file GRP.txt.

Il file testo contiene nel primo rigo due interi separati da uno spazio che indicano,
    rispettivamente, il numero di nodi ed il numero di archi.
I successivi righi contengono ciascuno tre numeri, separati da uno spazio,
    per indicare il nodo sorgente, nodo destinazione ed il peso di ogni arco.

Dotare la classe di un metodo BFS(s) che scrive nel file OUT.txt, per ogni nodo,
    la distanza dal nodo s ed il proprio predecessore. (pt. 18)
Dotare la classe dei metodi PRIM() e PRINT_MST() per calcolare e stampare a video il
Dotare la classe del metodo IS_BINARY(MST) che restituisca TRUE se MST è un albero bi
*/
//Inclusione librerie
#include <iostream>
#include <fstream>
#include <unordered_map>
#include <list>
#include <vector>
#include <queue>
using namespace std;

//enum per dare il colore ai nodi, più semplice
enum COLOR{WHITE,GRAY,BLACK};
//classe generale per gli algoritmi richiesti
class Grafo{
private:
    //struct privata per gestire i nodi
    struct Nodo{
        int key,d,f,peso;
        COLOR c;
        Nodo* p;
        bool inMST;//se il nodo fa parte dell'MST
        list<pair<Nodo,int>> adj;
        Nodo(int k=0) : key(k) {}
    };
```

```
    //struct privata per gestire gli archi
    struct Arco{
        int from,to,w;
        bool orient;
        Arco(int f,int t,int w,bool orie) : from(f),to(t),w(w){}
    };
    //strutture che mantengono informazioni su nodi e grafi
    unordered_map<int,Nodo> nodi;
    vector<pair<Arco,int>> archi;
public:
    //prototipi delle function
    Grafo(){}
    void AddNodo(int k);
    void AddArco(int s,int e,int w, bool orient);
    void graph_init(string path);
    void BFS(int start);
    void Prim(int r);
    void Print_MST();
    bool isBinary(int r);
};
//Uso il costruttore per creare il nodo e lo inserisco nella map
void Grafo::AddNodo(int k){
    Nodo tmp = Nodo(k);
    nodi[k]=tmp;
}
//Creo l'arco e aggiungo i nodi alle adiacenze solo se esistono
void Grafo::AddArco(int s,int e,int w,bool orient=false){
    if(nodi.find(s)!=nodi.end()&&nodi.find(e)!=nodi.end()){
        Arco a = Arco(s,e,w,false);
        nodi[s].adj.push_back(nodi[e],w);
        //Inserisco l'altra direzione se il grafo non è orientato
        if(!orient)
            nodi[e].adj.push_back(nodi[s],w);
        archi.emplace_back(make_pair(a,w));
    }else{
        cout << "Almeno un nodo fra " << s << " e " << e << " non presente nel grafo"
    }
}
//Function che inizializza il grafo dato un file txt
void Grafo::graph_init(string path){
```

```
    ifstream myfile(path);
    if(!myfile.is_open()){
        cout << "Errore di apertura del file " << path << endl;
        return;
    }
    //leggo il numero di vertici ed archi
    int V,E;
    myfile >> V >> E;
    //per ciascun arco, leggo u,v,w
    for(int i=1;i<=V;i++)
        AddNodo(i);
    int u,v,w;
    for(int i=0;i<E;i++){
        if(myfile >> u >> v >> w)
            AddArco(u,v,w);
        else{
            cout << "Errore lettura arco: " << endl;
            return;
        }
    }
    myfile.close();
}
//function di Breadth-First Search
void Grafo::BFS(int s){
    //coda per mantenere i nodi
    queue<Nodo*> pq;
    //Inizializzo i nodi del grafo
    for(auto& [k,n] : nodi){
        n.c=WHITE;
        n.p=nullptr;
        n.d=INT_MAX;
    }
    nodi[s].d=0;
    pq.push(&nodi[s]);
    //Itero finchè ci sono nodi nella queue (non ancora scoperti)
    while(!pq.empty()){
        Nodo* u = pq.front();
        pq.pop();
        //Scorro le adiacenze di u
        for(auto& [v,w] : u->adj){
```

```
            Nodo* nodoV = &nodi[v.key];
            if(nodoV->c==WHITE){
                nodoV->d=u->d+1;
                nodoV->p=u;
                nodoV->c=GRAY;
                pq.push(nodoV);
            }
        }
        u->c=BLACK;
    }
    //Scrivo sul file di output
    cout << "BFS eseguita, scrivo sul file..." << endl;
    ofstream output("output.txt");
    for(auto& [key,node] : nodi){
        //Se il nodo è quelli iniziale, scrivo NIL al posto del predecessore
        if(node.p==nullptr){
            output << "NIL->" << node.key << endl;
        }else{
            output << node.p->key << "->" << node.key << endl;
        }
    }
}
//Function per l'algoritmo di Prim
void Grafo::Prim(int r){
    //Function specifica per comparare i pesi
    auto compare = [](const Nodo* a,const Nodo* b){return a->peso>b->peso;};
    priority_queue<Nodo*,vector<Nodo*>,decltype(compare)> pq(compare);
    //Inizializzo nodi e i loro pesi
    for(auto& [key,node] : nodi){
        node.peso=INT_MAX;
        node.p=nullptr;
        node.inMST=false;
    }
    nodi[r].peso=0;
    pq.push(&nodi[r]);
    //Itero finchè la coda non è vuota
    while(!pq.empty()){
        Nodo* u = pq.top();
        pq.pop();
        if(u->inMST) continue;
```

```
        u->inMST=true;
        for(auto& [v,w] : u->adj){
            Nodo& nodoV = nodi[v.key];
            //Se il nodo non è nell'MST, e possiamo rilassarne il peso
            if(!nodoV.inMST && w<nodoV.peso){
                nodoV.p=u;
                nodoV.peso=w;
                pq.push(&nodoV);
            }
        }
    }
    //Sommiamo i pesi dei nodi che appartengono all'MST
    long long totW=0;
    for(auto& [key,node] : nodi){
        if(node.inMST && node.p!=nullptr)
            totW+=node.peso;
    }
    cout << "Peso dell'MST: " << totW << endl << "Stampo l'MST" << endl;
    Print_MST();
}
//Stampa dell'MST e scrittura sul file (Non richiesta)
void Grafo::Print_MST(){
    for(auto& [key,node] : nodi){
        if(node.inMST){
            if(node.p==nullptr)
                cout << "Radice: " << node.key<<endl;
            else
                cout << node.p->key << "->" << node.key << endl;
        }
    }
    //Apro il file in scrittura
    ofstream Prim_output("prim_output.txt");
    if(!Prim_output.is_open())
        return;
    for(auto& [key,node] : nodi){
        if(node.inMST){
            if(node.p==nullptr)
                Prim_output << "Radice: " << node.key<<endl;
            else
                Prim_output << node.p->key << "->" << node.key << endl;
```

```
        }
    }
    Prim_output.close();
}
//Function che ritorna se l'MST è binario (ha max due figli per nodo)
bool Grafo::isBinary(int r){
    for(auto& [k,n] : nodi){
        //Se adj ha size >3, significa che abbiamo almeno 3 nodi adiacenti (figli)
        if(n.adj.size()>3) return false;
    }
    return true;
}

int main(){
    Grafo G = Grafo();
    string path;
    cout << "Inserisci nome del file: ";
    cin >> path;
    if(path.length()>=4 && path.substr(path.length()-4)==".txt")
        G.graph_init(path);
    else{
        path+=".txt";
        G.graph_init(path);
    }
    int s;
    cout << "Inserisci il nodo da cui partire per la BFS:"<<endl;
    cin >> s;
    G.BFS(s);
    cout << "Procedo con l'MST" << endl;
    cout << "Inserisci la radice dell'MST:";
    cin >> s;
    G.Prim(s);
    bool res = G.isBinary(1);
    if(res)
        cout << "L'MST e binario" << endl;
    else
        cout << "L'MST non e binario" << endl;
    cout << "Termino..."<<endl;
    return 0;
}
```

```
/*
GIUGNO 2023
    ordinamento topologico (18pt) (DFS)
    ciclo hamiltoniano (6 pt)
    contare i cicli (6 pt)
*/
//Librerie
#include <iostream>
#include <fstream>
#include <list>
#include <vector>
#include <unordered_map>
#include <cstring>
#include <stack>
using namespace std;

//Enum per gestire i colori dei nodi
enum COLOR{WHITE,GRAY,BLACK};

//Classe generale per gli algoritmi richiesti
class Grafo{
private:
    //Struct privata per gestire i Nodi
    struct Nodo{
        int key,d,f;
        Nodo* p;
        COLOR c;
        list<Nodo> adj;
        Nodo(int k=0) : key(k){}
    };
    //Struct privata per gestire gli Archi
    struct Arco{
        int from,to;
        Arco(int s,int e) : from(s),to(e){}
    };
    //Strutture che mantengono informazioni su nodi e archi
    unordered_map<int,Nodo> nodi;
    vector<Arco> archi;
public:
    Grafo(){}
```

```
    //Prototipi
    void AddNodo(int k);
    void AddArco(int s,int e);
    void graph_init(string path);
    void DFS();
    void Visit(int u,int& time,stack<Nodo*>& s,int& cnt);
    void prt_topo(stack<Nodo*>& s);
    void output(stack<Nodo*>& s);
    bool trovaCicloHamiltoniano();
    bool trovaCicloHamiltonianoUtil(Nodo* u,vector<bool>& visitato,vector<int>& path,
};

void Grafo::AddNodo(int k){
    Nodo n = Nodo(k);
    nodi[k]=n;
}

void Grafo::AddArco(int s,int e){
    if(nodi.find(s)!=nodi.end()&&nodi.find(e)!=nodi.end()){
        Arco a = Arco(s,e);
        nodi[s].adj.push_back(nodi[e]);
        nodi[e].adj.push_back(nodi[s]);
        archi.emplace_back(a);
    }else{
        cout << "Almeno uno dei nodi fra " << s << " e " << e << " non trovato" << en
        return;
    }
}

//Function per la DFS e l'ordinamento topologico
void Grafo::DFS(){
    int cnt=0;
    stack<Nodo*> s;
    //Inizializzazione dei nodi
    for(auto& [key,node] : nodi){
        node.c=WHITE;
        node.d=node.f=0;
        node.p=nullptr;
    }
    int time=0;
```

```
    for(auto& [key,node] : nodi){
        if(node.c==WHITE)
            Visit(node.key,time,s,cnt);
    }
    //Stampa nel file di output e a schermo
    cout << "Procedo a creare il file di output" << endl;
    //Utilizzo una copia dello stack perchè andrò a svuotarlo e non funzionerebbe su
    stack<Nodo*> tmp = s;
    output(tmp);
    cout << "Stampo a video l'ordinamento" << endl;
    prt_topo(s);
    if(cnt>0){
        cout << "Trovati: " << cnt << " cicli" << endl;
    }else{
        cout << "Non sono presenti cicli nel grafo" << endl;
    }
}

//Function di visita del Topologico
void Grafo::Visit(int u,int& time, stack<Nodo*>& s,int& cnt){
    //Trovo il nodo del grafo corretto (struct nodo)
    Nodo& nodoU = nodi[u];
    nodoU.d=time;
    time++;
    nodoU.c=GRAY;
    for(auto& v : nodoU.adj){
        if(v.c==WHITE){
            v.p=&nodoU;
            Visit(v.key,time,s,cnt);
        //Se il nodo v è grigio, abbiamo un ciclo
        }else if(v.c==GRAY)
            cnt++;
    }
    nodoU.c=BLACK;
    time++;
    nodoU.f=time;
    //Inserisco u nello stack dopo aver finito la visita
    s.push(&nodoU);
}
//Function di stampa per l'ordinamento topologico
```

```
void Grafo::prt_topo(stack<Nodo*>& s){
    while(!s.empty()){
        Nodo* u = s.top();
        s.pop();
        cout << "Nodo: " << u->key << ": (d) " << u->d << ", (f) " << u->f << endl;
    }
}
//Function di scrittura dell'output del topologico
void Grafo::output(stack<Nodo*>& s)    ofstream output("output.txt");    while(!s.emp

//Function di inizializzazione del grafo dal file path
void Grafo::graph_init(string path){
    ifstream input(path);
    if(!input.is_open()){
        cout << "Errore nell'apertura del file: " << path << endl;
    }
    int V,E;
    input >> V >> E;
    for(int i=1;i<=V;i++)
        AddNodo(i);
    int u,v,w;
    for(int i=0;i<E;i++)
        if(input >> u >> v >> w)
            AddArco(u,v);
        else
            cout << "Errore durante la lettura degli archi" << endl;
    input.close();
}

//Function che ritorna TRUE se vi è un ciclo Hamiltoniano
bool Grafo::trovaCicloHamiltonianoUtil(Nodo* u, vector<bool>& visitato, vector<int>&
    //Utilizzo path per avere il percorso del ciclo
    path[pos] = u->key;
    //Segno u come visitato
    visitato[u->key] = true;
    //Se siamo alla fine dei nodi (G.V)
    if (pos == nodi.size() - 1) {
        //Scorro le adiacenze dell'ultimo nodo
        for (auto& v : u->adj) {
            //Se l'ultimo nodo è adiacente al primo
```

```
            if (v.key == path[0]){
                //Lo inserisco nella coda e stampo il path
                path[pos + 1] = v.key;
                cout << "Ciclo Hamiltoniano trovato: ";
                for (int i = 0; i <= pos + 1; i++)
                    cout << path[i] << " ";
                cout << endl;
                return true;
            }
        }
        //Altrimenti, eseguo il backtrack, non essendoci un C.H.
        visitato[u->key] = false; // Backtrack
        return false;
    }
    //Richiamo la function sulle adiacenze di u
    for (auto& v : u->adj){
        if (!visitato[v.key]) {
            if (trovaCicloHamiltonianoUtil(&v, visitato, path, pos + 1))
                return true;
        }
    }
    //Se u non porta a cicli hamiltoniani, effettuo il backtrack
    visitato[u->key] = false; // Backtrack
    return false;
}
//Function principale per la ricerca del C.H.
bool Grafo::trovaCicloHamiltoniano() {
    vector<bool> visitato(nodi.size() + 1, false);
    vector<int> path(nodi.size() + 1, -1);
    //Richiamo la Util sui nodi del grafo
    for (auto& [key, node] : nodi) {
        if (trovaCicloHamiltonianoUtil(&node, visitato, path, 0))
            return true;
    }
    return false;
}

int main(){
    Grafo g = Grafo();
    string path;
```

```
    cout << "Inserisci il nome del file: ";
    cin >> path;
    if(path.length()<4||path.substr(path.length()-4)!=".txt")
        path += ".txt";
    g.graph_init(path);
    cout << "Runno l'ordinamento topologico" << endl;
    g.DFS();
    bool hasCicloHamiltoniano = g.trovaCicloHamiltoniano();
    cout << "Il grafo " << (hasCicloHamiltoniano ? "ha" : "non ha") << " un ciclo ham
    return 0;
}
```

22.4 Esercizi sugli ABR

```
/*
Implementare una struttura dati ABR che permetta di caricare le coppie <chiave,valore
    nel file ABR.txt secondo la preorder (L'ordine in cui appaiono è root,root->l,roo
Dotare la classe di un metodo per scrivere nel file OUT.txt l'output della visita pre
*/
//Librerie
#include <iostream>
#include <fstream>
#include <cstring>
#include <vector>
using namespace std;

//Semplice classe per il nodo
class Nodo{
private:
    int key;
    string val;
    Nodo *p,*l,*r;
public:
    Nodo(int k,string val) : key(k),val(val){}
    int getKey(){return key;}
    string getVal(){return val;}
    Nodo* getP(){return p;}
    Nodo* getL(){return l;}
    Nodo* getR(){return r;}
    void setP(Nodo* n){p=n;}
```

```
    void setL(Nodo* n){l=n;}
    void setR(Nodo* n){r=n;}
};
//Semplice classe per l'ABR
class ABR{
private:
    Nodo *root;
public:
    ABR(){root=nullptr;}
    //Prototipi
    Nodo* insertPreorder(vector<int>& chiavi,vector<string>& valori,int& idx);
    void preorder(Nodo* root);
    void preorderFile(Nodo* root,ofstream& outFile);
    Nodo* getRoot()return root;
    void setRoot(Nodo* r)root=r;
    void tree_init(string path);
};
//Function specifica per l'inserimento preorder
Nodo* ABR::insertPreorder(vector<int>& chiavi,vector<string>& valori, int& idx){
    //Se abbiamo superato i nodi da inserire, ritorno nullptr
    if(idx>=chiavi.size())  return nullptr;
    //Creo la radice
    Nodo *root = new Nodo(chiavi[idx],valori[idx]);
    idx++;
    //Per ogni nodo successivo, imposto in ordine il figlio Sx e Dx
    root->setL(insertPreorder(chiavi,valori,idx));
    root->setR(insertPreorder(chiavi,valori,idx));
    //Ritorno la radice dell'ABR
    return root;
}
//Function specifica per la scrittura della preorder su file
void ABR::preorderFile(Nodo* root,ofstream& outFile){
    if(root==nullptr) return;
    //Scrivo la coppia (chiave-valore)
    outFile << root->getKey() << " - " << root->getVal() << endl;
    //Invoco ricorsivamente la function sui figli
    preorderFile(root->getL(),outFile);
    preorderFile(root->getR(),outFile);
}
```

```
void ABR::preorder(Nodo* root){
    if(root==nullptr) return;
    cout << root->getVal() << " ";
    preorder(root->getL());
    preorder(root->getR());
}

//Function che inizializza l'ABR dal file txt
void ABR::tree_init(string path){
    ifstream input(path);
    if(!input.is_open()){
        cout << "Errore nell'apertura del file: " << path << endl;
        return;
    }
    int chiave;
    string valore;
    vector<int> chiavi;
    vector<string> valori;
    //Mantengo chiavi e valori nei vector
    while(input>>chiave>>valore){
        chiavi.push_back(chiave);
        valori.push_back(valore);
    }
    input.close();
    int idx = 0;
    //Creo l'albero nodo per nodo e ritorno la radice
    root = insertPreorder(chiavi,valori,idx);
}

int main(){
    ABR *A = new ABR();
    string path;
    cout << "Inserisci nome file: ";
    cin >> path;
    if(path.length()<4||path.substr(path.length()-4)!=".txt")
        path+=".txt";
    A->tree_init(path);
    A->preorder(A->getRoot());
    ofstream out("ABR_output.txt");
    A->preorderFile(A->getRoot(),out);
```

```
    delete A;
    return 0;
}
```

23 Algoritmi di Ordinamento - Codice C++

23.1 Algoritmi di Complessita' Polinomiale

23.1.1 Bubble Sort

```
void BubbleSort(int a[],int dim){
    for(int i=0;i<dim-1;i++){
        for(int j=0;j<dim-i-1;j++){
            if(a[j]>a[j+1]){
                swap(a[j],a[j+1]);
            }
        }
    }
}
```

23.1.2 Insertion Sort

```
void InsertionSort(int a[],int dim){
    for(int i=1;i<dim-1;i++){
        int key=a[i];
        int j=i-1;
        while(j>=0&&a[j]>key){
            a[j+1]=a[j];
            j--;
        }
        a[j+1]=key;
    }
}
```

23.1.3 Quick Sort

```
int Partition(int a[], int p, int r){
    int x=a[p];
    int i=p-1;
    int j=r+1;
    while(1){
        do{
            i++;
        }while(a[i]>=x);
        do{
            j--;
        }while(a[j]<=x);
        if(i<j){
            swap(a[i],a[j]);
        }else{
            return j;
        }
    }
}
void QuickSort(int a[],int p,int r){
    if(p<r){
        int q = Partition(a,p,r);
        QuickSort(a,p,q);
        QuickSort(a,q+1,r);
    }
}
```

23.2 Algoritmi di Complessita' Linearitmica

23.2.1 Heap Sort

```
int Left(int i) {
    return 2 * i + 1;
}
int Right(int i) {
    return 2 * i + 2;
}
int Parent(int i) {
    return (i - 1) / 2;
}
void MaxHeapify(int arr[], int n, int i) {
    int largest = i;
    int left = Left(i);
    int right = Right(i);
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
   }
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }
    if (largest != i) {
        swap(arr[i], arr[largest]);
        MaxHeapify(arr, n, largest);
    }
}

void BuildMaxHeap(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--) {
        MaxHeapify(arr, n, i);
    }
}

void HeapSort(int arr[], int n) {
    BuildMaxHeap(arr, n);
    for (int i = n - 1; i >= 0; i--) {
        swap(arr[0], arr[i]);
        MaxHeapify(arr, i, 0);
    }
```

```
}
```

23.2.2 Merge Sort

```
void Merge(int a[],int l,int m,int r){
    n1=m-l+1;
    n2=r-m;
    int *L = new int[n1+1];
    int *R = new int[n2+1];
    int i,j;
    for(i=0;i<n1;i++){
        L[i]=a[l+i-1];
    }
    for(j=0;j<n2;j++){
        R[j]=a[m+j];
    }
    L[n1]=INT_MAX;
    R[n2]=INT_MAX;
    i=1;j=1;
    for(int k=l;k<r;k++){
        if(L[i]<=R[j]){
            a[k]=L[i++];
        }else{
            a[k]=R[j++];
        }
    }
}

void MergeSort(int a[],int l,int r){
    if(l<r){
        int m = (l+r)/2;
        MergeSort(a,l,m);
        MergeSort(a,m+1,r);
        Merge(a,l,m,r);
    }
}
```

23.3 Algoritmi di Complessita' Lineare

23.3.1 Bucket Sort

```
void BucketSort(float a[], int n){
    vector<float> buckets[n];
    for(int i=0;i<n;i++){
        int bucket_idx = n*a[i];
        buckets[bucket_idx].push_back(a[i]);
    }
    for(int i=0;i<n;i++){
        sort(buckets[i].begin(),buckets[i].end());
    }
    int idx = 0;
    for(int i=0;i<n;i++){
        for(size_t j=0;j<buckets[i].size();j++){
            a[idx++]=buckets[i][j];
        }
    }
}
```

23.3.2 Counting Sort

```
void CountingSort(int a[],int dim){
    //Trovo il max tramite la function max_element() di algorithm
    int max = *max_element(a,a+dim);
    vector<int> cnt(max+1,0);
    for(int i=0;i<dim;i++){
        cnt[a[i]]++;
    }
    int idx=0;
    for(int i=0;i<=max;i++){
        while(cnt[i]>0){
            a[idx++]=i;
            int[i]--;
        }
    }
}
```

23.3.3 Radix Sort

```
//CountingSort basato su cifre
void countingSort(int arr[], int n, int exp){
    vector<int> output(n); // Array di output
    vector<int> count(10, 0); // Array di conteggio, inizializzato a 0
    // Memorizza il conteggio delle occorrenze in count[]
    for (int i = 0; i < n; i++) {
        count[(arr[i] / exp) % 10]++;
    }
    // Cambia count[i] in modo che count[i] contenga ora la posizione dell'elemento i
    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }
    // Costruisci l'array di output
    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }
    // Copia l'array di output in arr[], così che arr[] contenga ora i numeri ordinat
    for (int i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}

void RadixSort(int a[],int n){
    //Prendo il max tramite la function max_element() di algorithm
    int max = *max_element(a,a+n);
    //Algoritmo stabile su cifra d
    for(int exp=1;max/exp>0;exp*=10){
        CountingSort(a,n,exp);
    }
}
```