

---

# **PROGRAMMAZIONE E ALGORITMICA II**

## **SEMESTRE**

---

**Corso C**

**Autore**

Giuseppe Acocella

2023/24

<https://github.com/Peenguino>

# Contents

<b>1 Strutture Dati</b>	<b>4</b>
1.1 Strutture Dati Lineari . . . . .	4
1.2 Stack (Pila) . . . . .	5
1.3 Queue (Coda) . . . . .	6
1.4 Linked List Doppia . . . . .	7
1.4.1 Esempi d'applicazione . . . . .	10
1.5 Strutture Dati Non Lineari . . . . .	13
1.6 Alberi - Definizione . . . . .	13
1.7 Alberi Binari di Ricerca . . . . .	15
1.7.1 Operazioni su ABR . . . . .	15
1.8 2-3 Alberi . . . . .	17
1.8.1 Lemma su n(nodi), f(foglie) e h(altezza) . . . . .	17
1.8.2 Operazioni su 2-3 Alberi . . . . .	19
1.9 Heap . . . . .	22
1.9.1 Implementazione su Array . . . . .	22
1.9.2 Operazioni su Heap . . . . .	23
1.9.3 maxHeapify . . . . .	23
1.9.4 Heapsort . . . . .	26
1.10 Hashmap . . . . .	27
1.10.1 Tavole ad indirizzamento diretto . . . . .	27
1.10.2 Funzione Hash . . . . .	28
1.10.3 Collisione . . . . .	28
1.10.4 Chaining . . . . .	28
1.10.5 Operazioni e Costi - Chaining . . . . .	29
1.10.6 Funzioni Hash per interi . . . . .	30
1.10.7 Open Addressing . . . . .	31
<b>2 Sorting Lineare</b>	<b>33</b>
2.1 Counting Sort . . . . .	33
2.2 Radix Sort . . . . .	34
2.3 Ordinamento Stabile . . . . .	35
<b>3 Programmazione Dinamica</b>	<b>35</b>
3.1 Fibonacci . . . . .	36
3.2 Rod Cutting . . . . .	37
3.3 LCS . . . . .	39
3.4 Edit Distance . . . . .	41
3.5 Zaino 0-1 . . . . .	43
3.6 Greedy . . . . .	45
3.6.1 Zaino Frazionario . . . . .	45
3.6.2 Scheduling di attività . . . . .	46
3.6.3 Dimostrazione Soluzione Ottima del Greedy . . . . .	47

<b>4 Grafi</b>	<b>48</b>
4.1 Definizioni e caratteristiche . . . . .	48
4.2 Attraversamento di Grafo . . . . .	49
4.2.1 BFS - Breadth First Search . . . . .	49
4.2.2 DFS - Depth First Search . . . . .	52
4.2.3 Topological Sort . . . . .	54
4.2.4 Algoritmo di Dijkstra . . . . .	55
4.2.5 Algoritmo di Bellman Ford . . . . .	58
<b>5 Complessità Computazionale</b>	<b>60</b>
5.1 Modelli Computazionali . . . . .	60
5.2 Classi Computazionali . . . . .	60
5.2.1 Problemi Indecidibili . . . . .	61
5.2.2 Problemi Decidibili . . . . .	62
5.3 Problemi e Notazione . . . . .	62
5.3.1 Definizione Formale di Problema . . . . .	62
5.3.2 Tipologie di Problemi . . . . .	62
5.3.3 Classi di Complessità . . . . .	63
5.4 P vs NP . . . . .	64
5.4.1 SAT semplice . . . . .	64
5.4.2 Certificato . . . . .	65
5.4.3 P vs NP - Congettura . . . . .	66
5.5 Mappatura di Riduzione . . . . .	66
5.5.1 NP - Arduo/Completo . . . . .	66
5.5.2 Esempio di Riduzione Clique - SAT . . . . .	66

# 1 Strutture Dati

Definiamo ed elenchiamo varie Strutture Dati che ci permettono di rappresentare degli insiemi dinamici.

**Insieme Dinamico** Un Insieme Dinamico a differenza di Insiemi Statici hanno la caratteristica di poter essere modificati nel tempo. Rappresenteremo quindi ogni **elemento** dell'insieme come **oggetto** con dei relativi **attributi**. Perchè è necessario l'utilizzo di oggetti e attributi? Per poter sviluppare determinati insiemi dinamici sarà necessario **puntare** agli elementi successivi. In questo modo lavoreremo sempre per riferimento, puntando gli elementi tra di loro. Un esempio sono le linked list.

**Operazioni su Insiemi Dinamici** Abbiamo due categorie di operazioni, ovvero le operazioni di **query** e le operazioni di **modifica**:

1. Operazioni di Query: Le operazioni di Query "interrogano" la struttura dati, permettendo l'accesso ai suoi dati stessi. Sono delle operazioni di Query ad esempio SEARCH, MINIMUM, MAXIMUM, PREDECESSOR e SUCCESSOR.
2. Operazioni di Modifica: Le operazioni di Modifica, appunto, modificano la struttura stessa e i dati al suo interno. Sono delle operazioni di modifica ad esempio INSERT e DELETE.

## 1.1 Strutture Dati Lineari

Le strutture dati lineari sono caratterizzate da un "ordinamento" tra i vari suoi elementi, o meglio, sarà presente una relazione di successore e predecessore per ogni elemento.

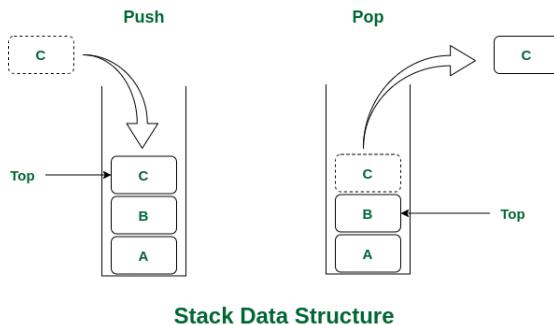
**Premessa sugli Array** Per poter permettere una prima implementazione di Pile e Code utilizzeremo degli Array. Ricordiamo alcune caratteristiche degli Array:

1. Organizzazione Logica: L'organizzazione logica degli array è **contigua**. Dunque gli elementi sono in posizioni una successiva all'altra.
2. Organizzazione Fisica: I valori sono memorizzati ad indirizzi **contigui**. In molti linguaggi infatti, per accedere ad un Array, si fa riferimento al primo elemento e successivamente si "somma al primo indice" la quantità necessaria per fare in modo che si possa accedere all'elemento desiderato.
3. Costo delle operazioni: Possiamo quindi analizzare come il costo delle operazione su Array cambi in base a che tipo di operazione stiamo effettuando, se di **Query** o di **Modifica**. Infatti, facendo l'esempio nel quale volessimo rimuovere il terzo elemento da un array di 8 elementi (**Modifica**). Tutti gli elementi successivi a quello rimosso dovranno essere spostati, richiedendo quindi un costo  $O(n)$ . Invece se volessimo accedere ad un elemento specifico (**Query**), basterebbe conoscere la sua posizione, richiedendo un costo  $O(1)$ .

## 1.2 Stack (Pila)

Gli stack sono strutture dati elementari caratterizzate dallo schema **LIFO**, ovvero Last In First Out. Potremmo implementare gli Stack su degli Array, ma andremmo ad incorrere naturalmente negli errori di **UNDER/OVER-FLOW**.

La pila infatti, se implementata su array, sarà gestita da un attributo *S.top* che indicherà l'elemento aggiunto più di recente. Le operazioni dunque di **Push** e **Pop** saranno gestite dallo spostamento di questo indice.



```
function emptyStack(S)
{
    if(S.top==0)
        return true
    else
        return false
}

function push(S,x)
{
    S.top=S.top+1
    S[S.top]=x
}

function pop(S)
{
    if(emptyStack)
        console.log("error UNDERFLOW")
    else
    {
        x=S[S.top]
        S.top=S.top-1
        return x
    }
}
```

Figure 1: Rappresentazione in pseudo-JS delle PILE su Array

### 1.3 Queue (Coda)

Le Code sono strutture dati caratterizzate dallo schema **FIFO**, ovvero First In First Out. Possiamo implementare le Code su Array, e come per le Pile potremmo incorrere nella problematica di **UNDER/OVER-FLOW**. A differenza delle Pile, le Code utilizzano due attributi, ovvero **Q.head** e **Q.tail**, dove *Q.head* punta al primo elemento, ovvero quello da "più tempo" nella pila, mentre *Q.tail* punta la posizione successiva all'elemento aggiunto più di recente.

```
function enqueue(Q,x)
{
    Q[Q.tail]=x
    if(Q.tail==Q.length)
        Q.tail=0
    else
        Q.tail=Q.tail+1
}

function dequeue(Q)
{
    x=Q[Q.head]
    if(Q.head==Q.length)
        Q.head=0
    else
    {
        Q.head=Q.head+1
        return x
    }
}
```

## 1.4 Linked List Doppia

Una linked list è una collezione di oggetti ognuno con tre attributi:

1. Key: Attributo chiave, elemento effettivo quindi dell'oggetto in questione.
2. Prev: Attributo puntatore, punta infatti all'elemento (oggetto) precedente.
3. Next: Attributo puntatore, punta infatti all'elemento (oggetto) successivo.



Ogni TRIPPA è dunque un oggetto con relativi attributi, definiamo quindi anche una LIST HEAD ed una LIST TAIL, determinate dall'assenza rispettivamente di un Prev o di un Succ.

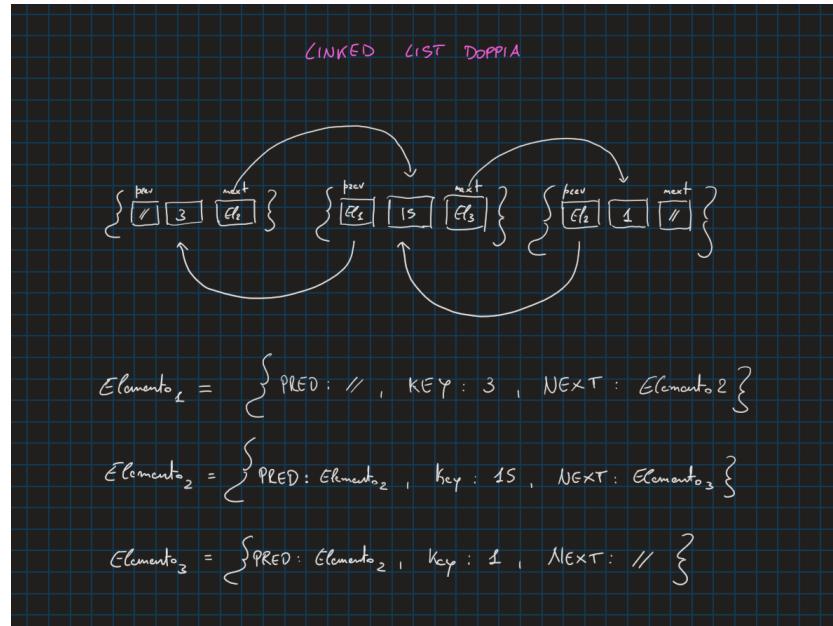
**Operazioni di Modifica e Costi** Notiamo che se vogliamo aggiungere un qualsiasi elemento in una qualsiasi posizione basterà cambiare i riferimenti di Prev e Next degli elementi tra cui vogliamo inserire il nuovo elemento. Questo provoca un costo  $O(1)$  in ogni tipologia di modifica, anche il **DELETE**.

```
function insertList(L,x)
{
    x.next=L.head
    //il successivo dell'elemento da aggiungere è l'attuale testa
    if(L.head!=nil) //se la lista non è vuota
        L.head.pred=x
    //allora il predecessore della testa è l'elemento da aggiungere
    L.head=x //la testa viene settata all'attuale elemento
    L.head.pred=nil //essendo la testa, il predecessore è settato a nil
}
```

**Operazioni di Query e Costi** A differenza degli Array, queste strutture dati non sono logicamente contigue, dunque per accedere ad un elemento qualsiasi della lista sarà necessario "attraversare" tutti quelli ad esso precedenti. Questo provocherà un costo  $O(n)$  per le operazioni di Query.

```
function insertList(L,x)
{
    x.next=L.head
    //il successivo dell'elemento da aggiungere è l'attuale testa
    if(L.head!=nil) //se la lista non è vuota
        L.head.pred=x
    //allora il predecessore della testa è l'elemento da aggiungere
    L.head=x //la testa viene settata all'attuale elemento
    L.head.pred=nil //essendo la testa, il predecessore è settato a nil
}
```

**Rappresentazione di Elementi della Lista come Oggetti** Ecco una rappresentazione dove, ispirandoci alla sintassi di oggetto in JS, mostriamo ogni elemento come un oggetto con tre attributi, di cui 2 sono puntatori al precedente o al successivo. Attenzione, gli attributi puntatore non puntano agli attributi key dell'oggetto puntato, ma esattamente all'oggetto.



**Gestione della Sentinella nelle Linked List** Per poter semplificare le operazioni agli estremi della lista, quindi le operazioni sui null, si potrebbe utilizzare un elemento nullo, ovvero la **sentinella**, impostata come precedente alla testa e successivo alla coda. Questo ci permette di evitare vari controlli durante le procedure di aggiunta e rimozione di elementi dalla lista.

**Operazioni su liste con Sentinella** Mostriamo quindi le operazioni eseguite senza il controllo agli estremi delle liste. Ogni riferimento alla testa della lista verrà visto come successivo della sentinella quindi  $L.nil.next$ .

```
//delete in una lista con sentinella

function deleteList1(L,x)
{
    x.prev.next=x.next
    x.next.prev=x.prev
}

//ricerca in una lista con sentinella

function searchInList1(L,k)
{
    x=L.nil.next
    while (x!=nil && x!=k)
    {
        x=k.next
    }
    return x //se l'elemento non è presente ritorna nil
}

//inserimento in una lista con sentinella

function insertList1(L,x)
{
    x.next=L.nil.next
    L.nil.next.prev=x
    L.nil.next=x
    x.prev=L.nil
}
```

#### 1.4.1 Esempi d'applicazione

Mostriamo degli esempi di esercizi che utilizzano strutture dati lineari per ottimizzare le operazioni:

**Notazione Polacca Inversa** La notazione polacca inversa è una forma che permette di annullare ogni tipo di ambiguità ponendo gli operandi prima dell'operatore. Potremmo immaginarci gli operandi come "argomenti" posti prima dell'operazione che li elaborerà. Mostriamo un esempio.

$$(((2 + 3) * 7) - 5) \quad (1)$$

2 3 + 7 \* 5 -

Come possiamo implementare un algoritmo che legga questa notazione e che la valuti?

Descriviamo passo passo le operazioni da eseguire:

1. Leggo da sinistra verso destra l'espressione ed eseguo un *push()* nella pila quando trovo degli operandi
  2. Quando trovo un operatore eseguo due *pop()* nella pila, valutando quell'operazione
  3. Una volta eseguita l'operazione, eseguo un *push()* alla pila con il risultato ottenuto
  4. Reitero fino a quando non rimane un solo elemento nella pila, ovvero il risultato.
- Ovviamente stiamo considerando di aver valutato l'intera espressione.

Mostriamo dunque un'implementazione in JavaScript del seguente algoritmo, assumendo l'utilizzo della seguente classe:

```
1 constructor(){
2     this.top=0
3     this.data=[]
4 }
5 isEmpty(){
6     return this.top==0?true:false
```

```
1 push(element){
2     this.top++
3     this.data[this.top]=element
4 }
5 pop(){
6     if(!this.top<=1)
7     {
8         this.top=this.top-1
9         return this.data[this.top+1]
10    }
11    else
12        return "UNDERFLOW ERROR"
13 }
```

```

1  function npi (espressione)
2  {
3      let pilal = new Stack();
4      let opr1,opr2,res
5      for(let i of espressione)
6      {
7          switch(i)
8          {
9              case "+":
10                 opr2=pilal.pop()
11                 opr1=pilal.pop()
12                 res= Number(opr1)+Number(opr2)
13                 pilal.push(""+res)
14                 break;
15
16              case "-":
17                 opr2=pilal.pop()
18                 opr1=pilal.pop()
19                 res= Number(opr1)-Number(opr2)
20                 pilal.push(""+res)
21                 break;
22
23              case "*":
24                 opr2=pilal.pop()
25                 opr1=pilal.pop()
26                 res= Number(opr1)*Number(opr2)
27                 pilal.push(""+res)
28                 break;
29
30              case "/":
31                 opr2=pilal.pop()
32                 opr1=pilal.pop()
33                 res= Number(opr1)/Number(opr2)
34                 pilal.push(""+res)
35                 break;
36
37              default:
38                  pilal.push(i)
39          }
40      }
41      return pilal.data[1]
42 }

```

**Visita Livelli albero con coda** Possiamo mostrare un altro esempio di utilizzo di strutture dati lineari. Infatti la visita per livelli di un albero k-ario in questo caso viene implementata grazie ai seguenti passi:

1. Valuta la radice, carica quindi nella coda i suoi figli ed effettua un *dequeue()* sulla testa che contiene l'attuale radice per poter proseguire, accumuliamo quindi le varie teste delle code che vengono rimosse
2. Reiteriamo questa operazione su tutti gli elementi della coda fino al raggiungimento della coda vuota

Dunque in questo caso abbiamo utilizzato questa classe per l'implementazione delle code:

```

1   class Queue
2 {
3     constructor()
4   {
5       this.head=0
6       this.tail=0
7       this.data=[]
8   }
9
10    isEmpty(){
11      if(this.tail==this.head)
12        return true
13      else
14        return false
15   }
16
17
18    enqueue(element){
19      this.data[this.tail]=element
20      this.tail=this.tail+1
21   }
22
23    dequeue(){
24      this.head=this.head+1
25      return this.data[this.head-1]
26   }
27 }
```

Mostriamo quindi l'algoritmo in JavaScript. (La corrente implementazione non è ottimale, infatti è in ordine  $O(n^2)$ .

```

1
2 function visitaLivelli(tree)
3 {
4     let res=[];
5     let coda1 = new Queue();
6     coda1.enqueue(tree)
7
8     while (!coda1.isEmpty())
9     {
10         let elementoElab = coda1.dequeue()
11         res.push(elementoElab.val)
12         if(elementoElab.s)
13         {
14             for(let j in elementoElab.s)
15                 coda1.enqueue(elementoElab.s[j])
16         }
17     }
18
19     return res
20 }
```

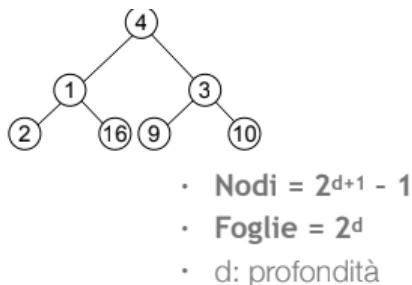
## 1.5 Strutture Dati Non Lineari

Fino ad ora abbiamo descritto strutture come Array, Liste, Pile e Code, dove ha senso chiederci quale sia l'elemento successivo e quello precedente. In questo capitolo descriveremo invece strutture come Alberi o Tavole Hash che non sono lineari.

## 1.6 Alberi - Definizione

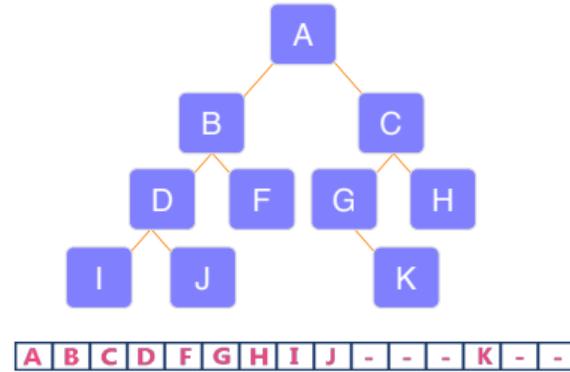
Possiamo definire gli alberi come una struttura dati gerarchica, caratterizzata appunto tra "relazioni" tra nodi. Elenchiamo alcuni tratti caratteristici di questa struttura dati:

1. Radice: Nodo senza padre.
2. Nodi Fratelli: Nodi che hanno lo stesso padre.
3. Profondità di Nodo: Numero di antenati del nodo stesso.
4. Altezza dell'albero: Profondità massima di un nodo nell'albero.



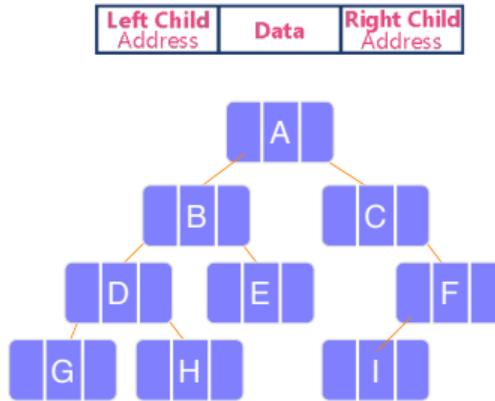
**Visite Alberi e Possibili Implementazioni** Una volta denotata la struttura "logica" di un albero, ci chiediamo come questi si possano implementare effettivamente. Mostriamo degli esempi:

1. Alberi implementati su Array: Questa implementazione è la più semplice, e porta diversi svantaggi.



Infatti, anche se le operazioni di accesso sono dirette, il *search()* sarebbe in ordine  $O(n)$  e saremmo costretti ad investire spazio inutilmente per le celle che devono indicare i "nil" dei nodi senza figli.

2. Alberi implementati su Linked List



Questo tipo di implementazione non spreca spazio, e ci permette anche di modificare abbastanza facilmente l'albero grazie all'utilizzo dei riferimenti. (Questa implementazione è molto simile a quella che si utilizza solitamente per gli alberi binari in JavaScript).

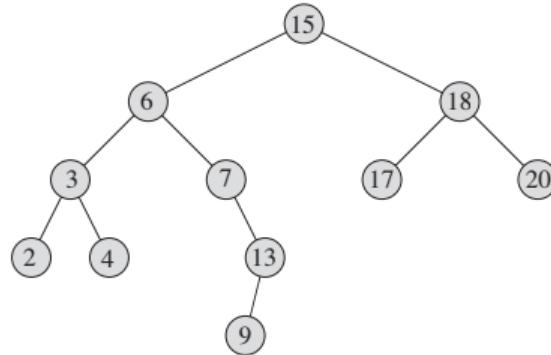
**Tipologie di Visite di Alberi Binari** Descriviamo prima di tutto le possibili visite che si possono effettuare su un albero binario tenendo in considerazione il nodo, il figlio sx e il figlio dx.

1. Visita Anticipata (PreOrder): NODO - FIGLIO SX - FIGLIO DX
2. Visita Simmetrica (InOrder): FIGLIO SX - NODO - FIGLIO DX
3. Visita Asimmetrica (PostOrder): FIGLIO SX - FIGLIO DX - NODO

## 1.7 Alberi Binari di Ricerca

Gli Alberi Binari di Ricerca (ABR) sono caratterizzati dall'ordine dei dati al suo interno. Per essere definito come tale, infatti, un albero binario di ricerca deve rispettare queste 2 condizioni:

1. Esiste una relazione di ordinamento totale tra i valori al suo interno, dunque è sempre possibile stabilire chi precede l'altro e viceversa.
2. Per ogni nodo e relativo valore, alla sua sinistra appariranno solo valori inferiori, e a destra solo valori superiori.



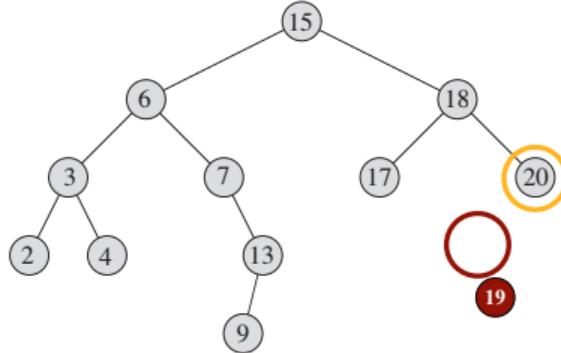
### 1.7.1 Operazioni su ABR

Le operazioni di Query e di Modifica notiamo che sono caratterizzate dal bilanciamento dell'albero stesso. Mostriamo qualche esempio.

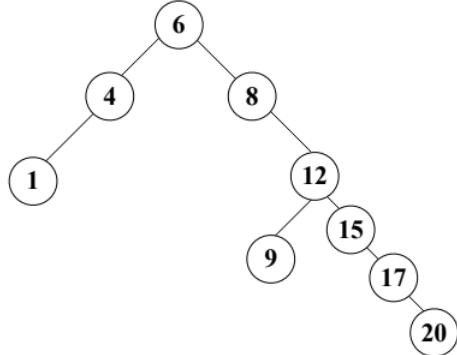
**Ricerca su ABR** La ricerca si baserà sull'ordinamento dell'albero stesso. Dunque, partendo dalla radice, se il valore cercato è più grande di quello attuale, allora si cercherà nel sottoalbero destro, altrimenti in quello sinistro. Reiterando questo procedimento ritroveremo il numero cercato, se presente. Costo dell'operazione?  $O(h)$ .

**Max/Min su ABR** In base a se stiamo cercando massimo o minimo, andremo a cercare arbitrariamente e rispettivamente a destra o a sinistra. Costo?  $O(h)$ .

**Inserimento in ABR** Un nuovo nodo verrà sempre inserito come foglia, ma il suo inserimento dovrà sempre rispettare la seconda condizione che caratterizza gli ABR. Costo?  $O(h)$ .



**Altezza, Bilanciamento ed Efficienza** Notiamo che tutte le operazioni mostrate precedentemente, dipendevano fortemente dall'altezza dell'albero stesso. Non possiamo però affermare che l'altezza rispetterà sempre un ordine logaritmico. Se ponessimo il caso in cui i dati inseriti siano già stati ordinati, allora staremmo sviluppando un albero completamente sbilanciato, tendendo quasi ad una lista.



Notiamo quindi che per preservare i vantaggi stabiliti da un albero binario di ricerca, è necessario mantenere una condizione semi-bilanciata, che assomigli il più possibile ad un albero binario completo. Esistono dei metodi di bilanciamento diretti, che redistribuiscono i valori periodicamente per assicurarne la corretta distribuzione, ma questo problema viene risolto completamente nei 2-3 Alberi.

## 1.8 2-3 Alberi

Questa struttura dati risolve la questione del bilanciamento degli ABR grazie ad una sorta di compromesso. Esprimiamo le condizioni che caratterizzano questa struttura:

1. Tutti i nodi hanno almeno 2 figli e massimo 3.
2. Tutti i cammini radice-foglia hanno la stessa lunghezza.

### 1.8.1 Lemma su $n(\text{nodi})$ , $f(\text{foglie})$ e $h(\text{altezza})$

definiamo questo lemma con queste due condizioni:

1. Prima condizione:

$$2^{h+1} - 1 \leq n \leq \frac{(3^{h+1} - 1)}{2} \quad (2)$$

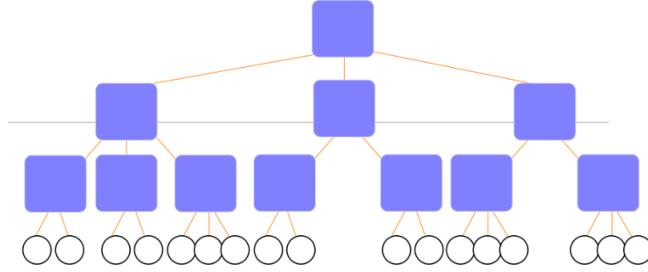
2. Seconda condizione:

$$2^h \leq f \leq 3^h \quad (3)$$

Questo quindi ci permette di sviluppare una dimostrazione per induzione che ci permette di affermare che in ogni caso il costo sarà  $\Theta(\log n)$ .

**Dimostrazione per Induzione** Definiamo passo passo la dimostrazione:

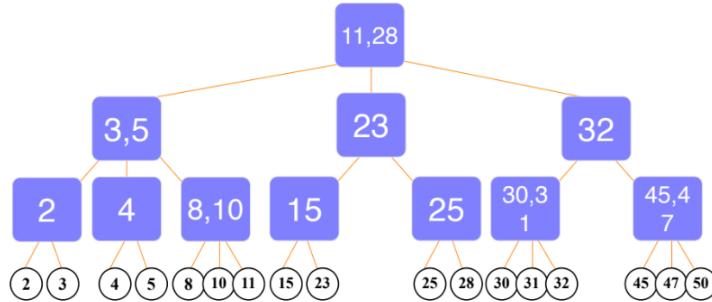
1. Caso Base: Considerando un albero di altezza  $h = 0$  staremmo semplicemente ipotizzando un caso banale che ci permette di assicurare che la proprietà sia vera per  $h=0$
2. Caso induttivo: Assumiamo che  $P(n)$  sia vera e verifichiamo che sia vera anche  $P(n + 1)$ . Sappiamo che se validiamo quest'implicazione, allora sarà valida per ogni  $n$ . Per poter validare l'implicazione, consideriamo un  $T$  albero con altezza  $h + 1$ , allora:
  - (a) Sia  $T'$  un albero ottenuto sottraendo a  $T$  l'ultimo livello
  - (b) Siano  $f'$  e  $n'$  rispettivamente foglie e nodi di  $T'$
  - (c) Per ipotesi induttiva avremo  $2^{h+1} - 1 \leq n' \leq \frac{(3^{h+1} - 1)}{2}$  e  $2^h \leq f' \leq 3^h$
  - (d) Allora dato che ogni foglia in  $T'$  avrà tra due e tre figli in  $T$  avremo che  $2 * 2^h \leq f \leq 3 * 3^h$  ovvero  $2^{h+1} \leq f \leq 3^{h+1}$
  - (e) Dato che  $n = n + f'$  abbiamo completato la dimostrazione



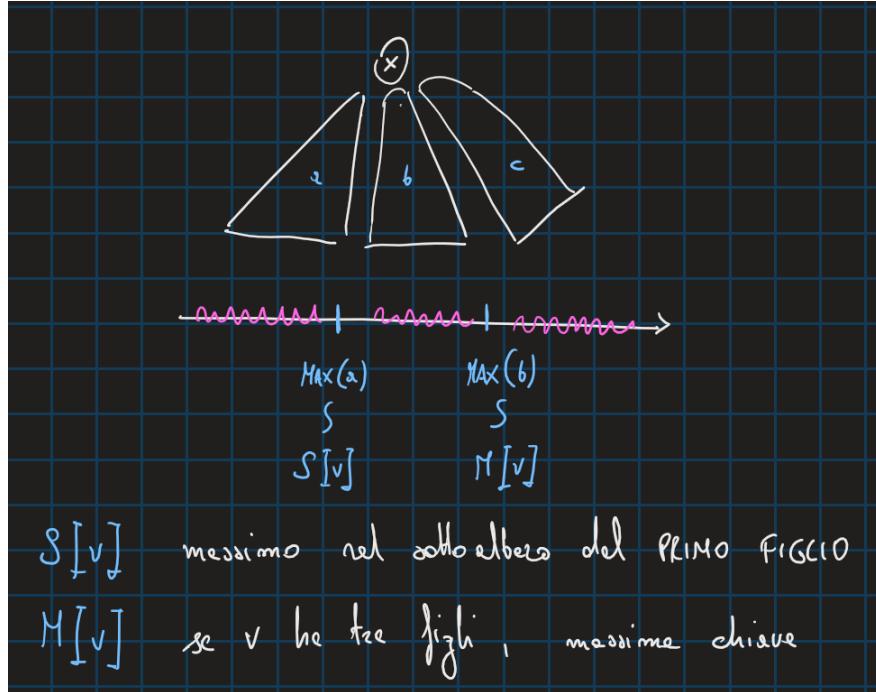
**Ordine e Distribuzione Dati nei 2-3 Alberi** Durante l'inserimento dei dati, la struttura del 2-3 Albero "indirizza" il valore in base all'intervallo a cui deve appartenere. Infatti i dati dell'albero verranno distribuiti in "intervalli" spezzati dai valori  $S[v]$  e  $M[v]$ .

1.  $S[v]$  indica l'elemento maggiore del sottoalbero sinistro
2.  $M[v]$  indica l'elemento minore del sottoalbero destro

Utilizzeremo questi due elementi per "spezzare" i dati ordinati all'interno dell'albero per suddividere in "intervalli" i dati stessi.



Questa distribuzione ci permette di avere una struttura sempre bilanciata. Il vantaggio del bilanciamento della struttura è che  $h = \Theta(\log n)$ . Ricordiamo infatti che tutte le operazioni definite su questa struttura e sull'ABR dipendono fortemente dall'altezza. Dunque è nostro interesse "forzare" l'altezza ad ordine logaritmico.



### 1.8.2 Operazioni su 2-3 Alberi

Dato che questo tipo di struttura permette di ottimizzare tutti i costi delle operazioni di query/modifica, si presenta la necessità di descrivere bene il mantenimento di questa struttura e le sue proprietà.

**Search su 2-3 Alberi** Descriviamo lo pseudocodice sottostante:

Notiamo subito che il tipo di ricerca che stiamo effettuando è ricorsiva. In un albero 2-3, ogni gruppo di dati ordinati, può essere suddiviso in 2 o 3 intervalli. Allora effettuiamo questi controlli

1. Controllo che la radice non sia una foglia, quindi sto controllando il caso base della ricorsione
  - (a) Controllo che il valore cercato non sia nella radice
2. Stabilisco, utilizzando gli elementi  $S[v]$  e  $M[v]$ , quale sia l'intervallo da controllare:
  - (a) Quello sinistro o destro se la radice ha due figli
  - (b) Il primo, il secondo o il terzo se la radice ha tre figli

```
1  function search(radice ,valore)
2  {
3      if(radice == una foglia)
4      {
5          if(valore == chiave(radice))
6              return chiave(radice)
7          else
8              return null
9      }
10     else
11     {
12         v_i=figlio(radice)
13         if(valore < S[radice])
14         {
15             search(v1 ,valore)
16         }
17         else
18         {
19             if(radice ha due figli || valore < M[radice])
20             {
21                 search(v2 ,valore)
22             }
23             else
24             {
25                 search(v3 ,valore)
26             }
27         }
28     }
29 }
30 }
```

**Insert/Delete e Split** Ipotizziamo di dover aggiungere un nuovo valore all'albero 2-3 tra le foglie. Abbiamo 2 possibili casi:

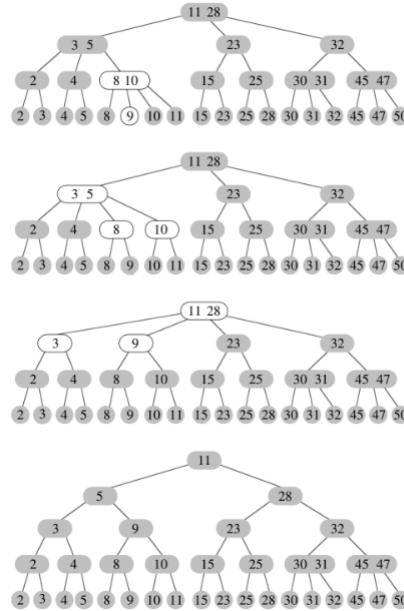
1. Trovato il potenziale nodo padre, questo ha 2 figli, quindi possiamo aggiungerlo senza particolari problemi
2. Trovato il potenziale nodo padre, questo ha 3 figli, dunque aggiungendo una foglia andremmo a violare la proprietà 2-3. Risulta quindi necessaria una procedura che ristabilizzi la condizione 2-3 dopo aver creato un nodo padre con 4 figli

**Descrizione della procedura di Split** Descriviamo dunque lo split, considerando 3 elementi principali, ovvero:

1. Nodo  $v$  padre
2. Nodo  $w$  nuovo
3. Nodo  $u$  aggiunta

Una volta identificati questi elementi principali, possiamo descrivere a passi l'algoritmo, sapendo che l'obiettivo è quello di collocare correttamente il nodo  $u$  aggiunta.

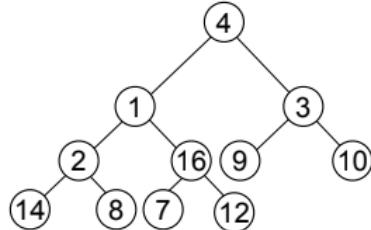
1. Identificate le foglie del nodo  $v$  padre, le sue due foglie con chiave minima diventano figlie di nodo  $w$  nuovo
2. Le due rimanenti foglie di nodo  $v$  padre restano a  $v$
3. Il nodo  $w$  nuovo viene attaccato al padre del nodo  $v$  padre
4. Se il padre di nodo  $v$  padre adesso ha 3 figli va bene, altrimenti dobbiamo richiamare la funzione split anche su padre di  $v$



## 1.9 Heap

Un heap è caratterizzato dalla presenza di due proprietà fondamentali:

1. **Proprietà Strutturale di Albero Quasi Completo**, ovvero un albero completo, tranne l'ultimo livello dove le foglie si distribuiscono in maniera non bilanciata tra sinistra e destra



2. **Proprietà di Max/Min Heap**, ovvero la radice di ogni sottoalbero è il massimo o il minimo tra i nodi del sottoalbero stesso. E' detto maxHeap quando la radice è il massimo del sottoalbero e minHeap quando la radice è il minimo del sottoalbero.

### 1.9.1 Implementazione su Array

Dato che l'albero non avrà "buchi", l'implementazione naturale dell'heap è su array, dato che la parte vuota sarà quella sulla destra. Distribuiamo quindi i valori dei nodi per livelli nell'array.

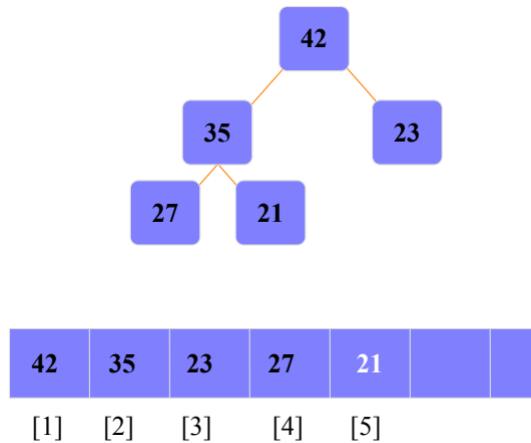


Figure 2: Implementazione su array con indici da 1

**Indici Padre/Figlio** Notiamo che esprimendo in questi termini l'heap, è possibile determinare in questo modo le relazioni esistenti tra i nodi:

1.  $indicePadre = \text{floor}(indiceElemento/2)$
2.  $indiceFiglioSx = 2 * indicePadre$
3.  $indiceFiglioDx = (2 * indicePadre) + 1$

### 1.9.2 Operazioni su Heap

Definiamo varie operazioni su questa struttura dati, come quella per mantenere la condizione di maxHeap o quella per costruire un maxHeap a costo lineare, ma prima è necessario ricordare queste due concetti fondamentali:

1. L'**altezza** di un nodo si considera dal basso verso l'alto
2. La **profondità** di un nodo si considera dall'alto verso il basso

### 1.9.3 maxHeapify

Aggiungendo un nodo o modificando in qualche modo i dati all'interno dell'heap potremmo alterarne la condizione di maxHeap. Dobbiamo costantemente controllare quindi che questa condizione sia rispettata affinché si possano sfruttare tutti i vantaggi generati dall'Heap.

Descriviamo dunque a passi l'algoritmo di maxHeapify:

1. Si assume che i sottoalberi sx e dx correnti rispettino la condizione di maxHeap
2. Si controlla dunque se l'attuale radice rompe la condizione
3. In caso affermativo, si esegue uno swap con il maggiore tra i figli sx e dx
4. Si ricontrolla, sull'albero la cui radice è stata swappata che sia rispettata la condizione di maxHeap
5. In caso negativo si richiama ricorsivamente la funzione maxHeapify sul sottoalbero in questione

Definiamo quindi le funzioni *parent(i)*, *left(i)*, *right(i)*:

```
1 parent(i)
2     return [i/2]
```

```
1 left(i)
2     return 2i
```

```
1 right(i)
2     return 2i+1
```

Adesso quindi possiamo descrivere lo pseudocodice della funzione maxHeapify

**Pseudocodice di maxHeapify** Definiamo in pseudocodice il mantenimento della condizione di maxHeap e successivamente ne analizziamo il costo ricorsivo:

```

1 maxHeapify(A, i)
2 {
3     l=left(i)
4     r=right(i)
5     if(l<=A.heapSize && A[l] > A[i])
6     {
7         max=l
8     }
9     else
10    {
11        max=i
12    }
13    if(r<=A.heapSize && A[r] > A[max])
14    {
15        max=r
16    }
17    if(max!=i)
18    {
19        A[i]=A[max]
20        maxHeapify(A,max)
21    }
22 }
```

Volendo discutere della complessità di questo algoritmo potremmo notare che la parte non ricorsiva ha costo  $\Theta(1)$ , è quindi necessario analizzare il costo della chiamata ricorsiva. Il caso peggiore è quello in cui l'ultima riga dell'albero è piena esattamente a metà, dunque la complessità è esprimibile come:

$$T(n) = T\left(\frac{2n}{3}\right) + \Theta(1) \quad (4)$$

Risolvendo con il caso 2 Master Theorem, si ricava che  $T(n) = O(\log n) = O(h)$

**buildMaxHeap** Se costruissimo un maxHeap inserendo in maniera adeguata le chiavi in questione, provocheremmo un costo in tempo  $O(n \log n)$ . Possiamo invece chiamare la funzione appena costruita che manterrà la condizione di maxHeap ad ogni inserimento

```

1 buildMaxHeap(A)
2 {
3     n = A.length
4     for(i = floor(n/2) , i>=1, i--)
5     {
6         maxHeapify(A,i,n)
7     }
8 }
```

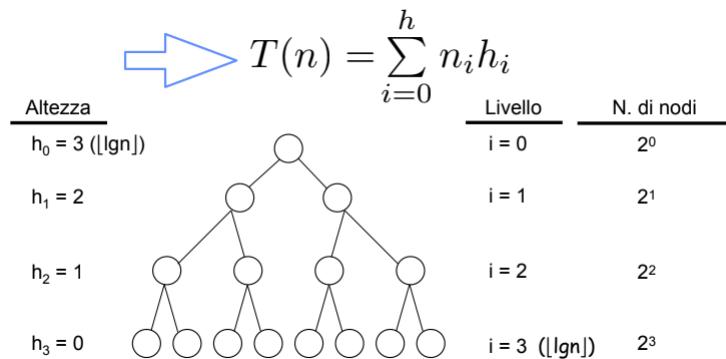
Stiamo quindi applicando *buildMaxHeap* dal basso verso l'alto, per permettere che la condizione di maxHeap sia verificata. Questo metodologia è detta **"bottom-up"**, appunto dal basso verso l'alto.

## buildMaxHeap: Correttezza e Costi

1. Analizziamo la **correttezza** per invarianti dell'algoritmo **buildMaxHeap**:

- (a) **Inizio:** Considerando  $i = \lfloor n/2 \rfloor$ , tutti i nodi indicizzati con un qualsiasi  $i_1 > i$  saranno delle foglie, rispetteranno quindi banalmente la condizione di maxHeap
- (b) **Mantenimento:** I figli del nodo  $i$ , sono numerati come  $> i$ , dunque saranno radice di maxHeap inferiori. Questo rende legale la chiamata a maxHeapify. Dato quindi che è vera nel nodo  $i$  allora sarà vera anche nei nodi  $i - 1$
- (c) **Terminazione:** Tutti i nodi da 1 a  $n$  sono quindi radice di un maxHeap

2. Analisi del **costo** dell'algoritmo **buildMaxHeap**:



$$h_i = \text{altezza heap radicato al livello } i$$

$$n_i = \text{numero di nodi al livello } i$$

$$\begin{aligned}
 T(n) &= \sum_{i=0}^h n_i h_i && \text{Costo di HEAPIFY al livello } i * \text{numero nodi a quel livello} \\
 &= \sum_{i=0}^h 2^i (h - i) && \text{Sostituisci valori di } n_i \text{ e } h_i \\
 &= \sum_{i=0}^h \frac{h-i}{2^{h-i}} 2^h && \text{Moltiplica per } 2^h \text{ num. e denom., e scrivi } 2^i \text{ come } \frac{1}{2^{i-h}} \\
 &= 2^h \sum_{k=0}^h \frac{k}{2^k} && \text{Cambio variabile: } k = h - i \\
 &\leq n \sum_{k=0}^{\infty} \frac{k}{2^k} && \text{La somma è più piccola di quella a } \infty \text{ e } h = \lg n
 \end{aligned}$$

Notiamo che il costo di *maxHeapify* è proporzionale all'altezza del nodo nel quale lo si sta applicando. Dunque possiamo rendere generico il calcolo e, dopo vari passaggi algebrici e portando fuori  $n$  dalla sommatoria, possiamo rifarci ad una serie notevole che converge. Dunque il costo di *buildMaxHeap*, applicando in bottom-up la funzione *maxHeapify* avrà costo  $O(n)$ .

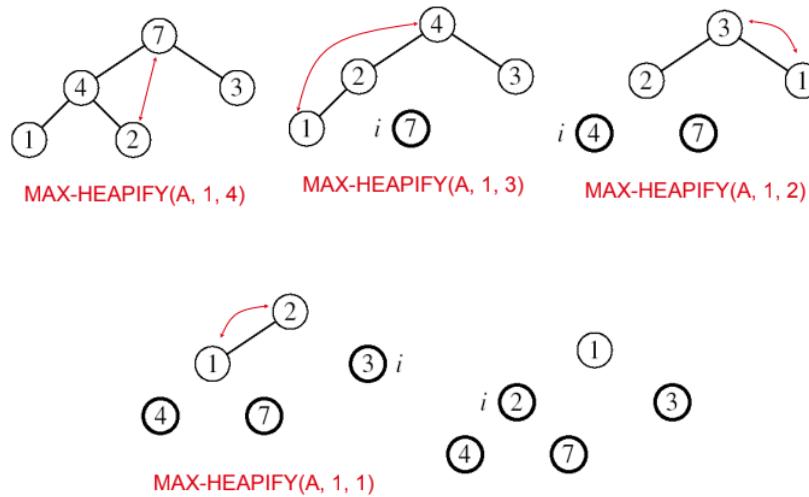
#### 1.9.4 Heapsort

L'Heapsort è un sort per confronti ottimale per tempo e spazio. A differenza dei sort visti fino ad ora infatti, è in place e ha costo  $O(n \log n)$ . Descriviamo il suo procedimento.

1. Considerando come esempio un *maxHeap*, noi sappiamo che questo sarà implementato su array e la radice conterrà l'elemento massimo di tutto l'heap
2. Possiamo dunque sostituire il primo elemento del vettore (**radice**) con l'ultimo
3. Escludiamo con gli indici l'ultimo elemento dell'array
4. Ricostituiamo un *maxHeap* sui rimanenti elementi dell'array
5. Reiterando questa procedura otterremo un array ordinato in senso crescente, dato che come proprietà abbiamo utilizzato quella del *maxHeap*

Ipotizziamo un array:

$$A = [7, 4, 3, 1, 2] \quad (5)$$



Dunque, dopo l'applicazione dell'heapSort, l'array sarà:

$$A = [1, 2, 3, 4, 7] \quad (6)$$

```

1  heapSort(A){
2      let tmp
3      buildMaxHeap()
4      for(i=A.length;i>0;i--){
5          tmp=A[0]
6          A[0]=A[i]
7          A[i]=tmp
8          maxHeapify(A,1)
9      }
10 }
```

## 1.10 Hashmap

Una Hashmap è una struttura dati che per ogni chiave  $k \in U$  universo delle chiavi, calcola il valore hash  $h(k)$  che sarà la posizione nella tabella in cui sarà inserita la chiave  $k$  stessa.

Prima però introduciamo un concetto simile ma più semplice:

### 1.10.1 Tavole ad indirizzamento diretto

Se considerassimo un universo delle chiavi  $U$  ragionevolmente piccolo, e un  $m$  grandezza della tabella non troppo grande, allora potremmo discutere le operazioni effettuabili su questa struttura.

1. Ricerca: Costo  $\Theta(1)$

```
1     directAddressSearch(T, k)
2     {
3         return T[k]
4     }
5
```

2. Inserimento: Costo  $\Theta(1)$

```
1     directAddressInsert(T, x)
2     {
3         T[x.key]=x
4     }
5
```

3. Cancellazione: Costo  $\Theta(1)$

```
1     directAddressDelete(T, k)
2     {
3         T[x.key]=nil
4     }
5
```

Questa pratica ha un ovvia debolezza: Se l'universo delle chiavi  $U$  è troppo grande, sarebbe impossibile generare una tabella di dimensioni  $|U|$ . Dunque vogliamo una struttura che sia in grado di simulare queste prestazioni in tempo, ma che allo stesso tempo gestisca in maniera più attuabile lo spazio.

### 1.10.2 Funzione Hash

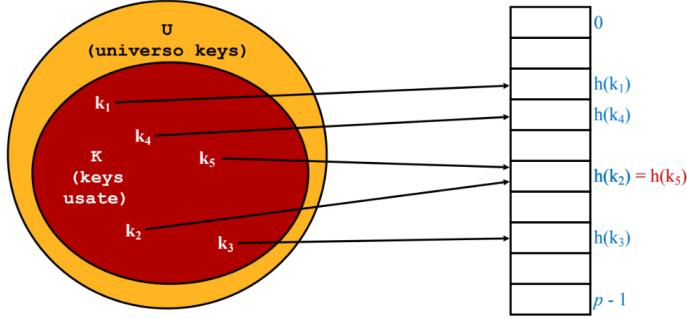


Figure 3: Hashmap

Notiamo gli elementi fondamentali elencandoli:

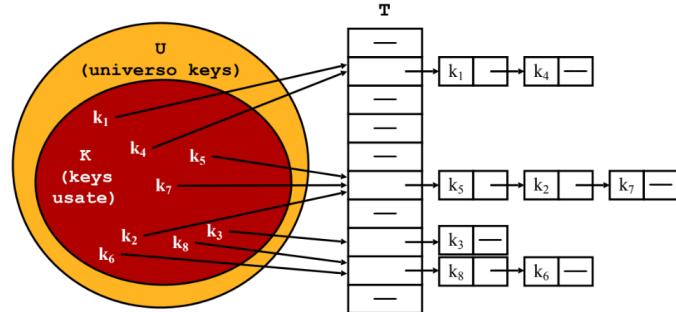
1.  $U$ : universo delle chiavi
2.  $K$ : insieme delle chiavi realmente utilizzate
3.  $h(k)$ : funzione hash  $h : U \rightarrow (0, 1, \dots, m - 1)$  che appunto, data una chiave  $k \in U$  elabora il suo valore, calcolando la posizione nella quale verrà inserita nella tabella
4. Tabella: Array di dimensione  $m$

### 1.10.3 Collisione

Notiamo, nell'immagine 3, che il valore hash delle chiavi  $k_2$  e  $k_5$  è lo stesso. Dunque quelle due chiavi nella stessa posizione causeranno una collisione. Esistono diversi metodi per risolvere la collisione. Mostreremo **chaining** e **open addressing**.

### 1.10.4 Chaining

Quando due chiavi  $k_i$  e  $k_j$  hanno lo stesso valore hash, allora l'ultimo dei due viene aggiunto in testa alla lista alla quale sta puntando la posizione  $h(k_i) = h(k_j)$ . Dunque gestiamo la collisione dando la possibilità di poter aggiungere più elementi in una sola posizione.



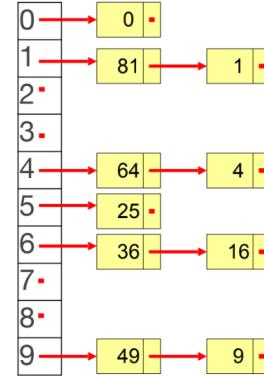
**Esempio di chaining** In questo caso, notiamo che stiamo considerando la tabella come array di liste concatenate. Dunque viene risolta la collisione tra chiavi.

Tabella hash è un array di liste collegate

Chiavi da inserire: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

**Note:**

- Elementi sono associati alle chiavi
- Hash function usata:  $h(k) = k \bmod m$



### 1.10.5 Operazioni e Costi - Chaining

Consideriamo le tre operazioni principali, ovvero  $search(x)$ ,  $delete(x)$  e  $insert(x)$ .

**Load Factor** Il load factor è il rapporto tra  $n$  numero di elementi nella tabella hash e  $m$  grandezza della tabella.

$$\lambda = \frac{n}{m} \quad (7)$$

Rappresenta quindi la **media** di elementi contenuti in ogni posizione della tabella.

**Caso peggiore del Chaining** Nella peggiore delle ipotesi, tutti i dati vengono inseriti in una sola posizione della tabella, dunque il costo della ricerca sarà  $O(n)$  dato che sarà necessario scorrere la lista, sia in caso di successo che di insuccesso.

**Hashing uniforme semplice** E' necessario dunque assumere che ogni chiave ha la stessa probabilità di essere mappata in una delle posizioni della tabella.

**Lunghezza media della lista** Dato che abbiamo affermato che  $\lambda$  fosse equivalente alla media di elementi presenti in ogni posizione della tabella, allora assumiamo che la lunghezza della lista  $T[h(k)] = \lambda$ .

**Costo Ricerca senza successo** Una volta aver assunto tutte le proprietà elencate sopra, analizziamo quali passi vengono eseguiti nel caso di una ricerca senza successo.

1. Si accede alla lista, costo:  $O(1)$
2. La si scorre tutta, costo:  $O(\lambda)$

Dunque la **ricerca fallita** impiega tempo  $\Theta(1 + \lambda)$ .

**Delete e Insert nel caso medio** Analizziamo anche queste due operazioni:

1. Delete: Bisogna trovare l'elemento e rimuoverlo, dunque il costo sarà simile a quello della ricerca nel caso medio:  $\Theta(1 + \lambda)$ . Nel caso di liste doppiamente collegate allora il costo sarebbe  $O(1)$ .
2. Insert: L'inserimento costerà semplicemente  $O(1)$  dato che aggiungeremo la chiave in testa alla nostra lista.

#### 1.10.6 Funzioni Hash per interi

Esempi di funzioni hash per interi, analizzandone le caratteristiche:

**Division Method** Si sceglie questa funzione hash:

$$h(k) = (k)mod(m) \quad (8)$$

In genere è consigliato evitare determinati valori di  $m$ , infatti spesso si sceglie un  $m$  primo non troppo vicino ad una potenza di 2.

**Multiplication Method** Ulteriore metodo per generare una funzione hash per interi:

1. Si moltiplica la chiave  $k$  per una costante  $A$  e si sottrae la parte sua parte frazionaria

$$kA - \text{partefrazionaria}(ka) \quad (9)$$

2. Si moltiplica questo valore per  $m$  e si prende il floor dell'espressione

$$h(k) = \text{floor}(m((kA)modulo1)) \quad (10)$$

### 1.10.7 Open Addressing

L'Open Addressing è un altro metodo che cerca di risolvere le problematiche causate dalla collisione di chiavi con stesso valore hash.

Ipotizzando dunque di voler inserire in una tabella  $T$  una chiave  $k_2$  con lo stesso valore hash  $h(k)$  di  $k_1$ , l'Open Addressing gestirà questa collisione con il **probing**, ovvero avanzarà di  $n$  posti nella tabella, dando un criterio a quella  $n$  in base al tipo di probing selezionato. Elenchiamo i tre tipi principali di probing:

- 1. Probing Lineare:** Il probing lineare è una tecnica che permette il controllo della posizione con indice  $h(k)$  e in caso la posizione fosse piena, controlla in senso lineare le posizioni successive. I tentativi sono formalmente indicati come  $f(i)$ , dunque la funzione finale di hash  $h(k)$  sarà la somma delle due funzioni  $h'(k)$  hash effettivo e  $f(i)$ , dunque:

$$h(k) = h'(k) + f(i) \quad (11)$$

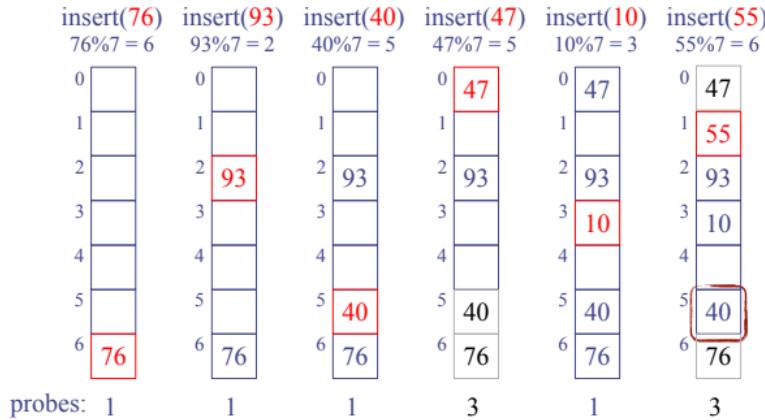


Figure 4: Con probes indichiamo i correnti tentativi

Questa tecnica però soffre una conseguenza diretta della linearità proposta, ovvero il **clustering primario**. Pian piano infatti, ad ogni tentativo d'inserimento andremo a generare dei "blocchi" di dati che rallenteranno le operazioni, dato che saremo costretti a scorrerli tutti prima di poter trovare la posizione desiderata.

**Costo Ricerca fallita** Intuitivamente, si fa sempre il primo tentativo, e successivamente ti tenta con probabilità  $\lambda$  il successivo, dunque il costo sarà:

$$\frac{1}{1 - \lambda} \quad (12)$$

Dunque il probing rende bene se  $\lambda \leq 0.5$ . Ricordiamo che  $\lambda$  rappresenta anche in questo caso il **load factor**. Questo costo andrebbe però dimostrato formalmente\*

**2. Probing Quadratico:** Simile al probing linare, semplicemente la funzione sommata che "skippava" elementi non sarà lineare ma quadratica. In formula:

$$f(i) = i^2 \quad (13)$$

$$h(k) = h'(k) + f(i) \quad (14)$$

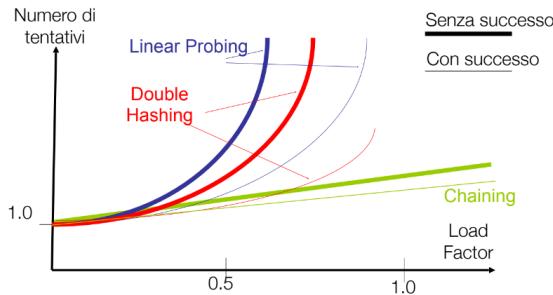
Questo tipo di probing è quindi caratterizzato dal fatto che, considerando un indice  $i$  di probing, invece di incrementarlo e tentare di nuovo, lo incrementa e considera la posizione al quadrato. Allo stesso modo del probing lineare, questo tipo di soluzione presenta la problematica del **clustering secondario**, che influenza meno negativamente la soluzione rispetto al tipo primario.

**3. Double Hashing:** La funzione sommata è un valore hash calcolato su una seconda funzione hash data. Dunque in formula:

$$f(i) = i * h_2(k) \quad (15)$$

$$h(k) = h'(k) + f(i) \quad (16)$$

Per far in modo che la tabella venga utilizzata per intero, il valore  $h_2(k)$  deve essere coprimo di  $m$  che rappresenta la dimensione della tabella. Due numeri sono coprimi se l'unico divisore che hanno in comune è 1.



	Ricerca senza Successo	Ricerca con Successo
Chaining	$1 + \lambda$	$1 + \lambda$
Open Addressing (hashing uniforme)	$1 / (1 - \lambda)$	$\frac{1}{\lambda} \ln \frac{1}{1-\lambda}$

	Load Factor consigliato
Chaining	$\lambda$ costante
Probing lineare o quadrattico	$\lambda \leq 0.5$ (semi-piena)
Double Hashing	$\lambda \leq 0.5$ (semi-piena)

## 2 Sorting Lineare

In questo capitolo mostreremo dei tipi di sorting che ordinano in maniera differente dal confronto, e definiremo delle proprietà come la stabilità, analizzandone i costi.

### 2.1 Counting Sort

Il Counting Sort è un tipo di ordinamento che non sfrutta i confronti, ma ipotizza che i dati inseriti siano interi contenuti nel range  $[0, k]$ . Dunque in questo modo non sarà necessario confrontare ogni dato con gli altri, infatti ogni numero contiene già la sua "posizione" dell'output ordinato.

Forniamo un esempio pratico per rendere più semplice l'applicazione del counting sort. Per implementare questo sort avremo bisogno di:

1. A, array di input, dove  $len = A.length$

A	0	1	2	3	4	5	6	7
	2	5	3	3	3	2	3	1

2. B, array di output, che avrà la stessa  $len$  di A

B	0	1	2	3	4	5	6	7
	?	?	?	?	?	?	?	?

3. C, array contatore, che ci permetterà di conservare le occorrenze di ogni elemento nell'array

C	0	1	2	3	4	5
	0	1	2	4	0	1

4. k, range dove sono contenuti gli interi dell'array di input

$$k = 5 \tag{17}$$

**Perchè non utilizzare sempre il Counting Sort?** Questo tipo di sort dipende fortemente dal  $K$  range su cui si distribuiscono i dati. Dunque in caso di cifre intere è molto vantaggioso, ma se stessimo lavorando su range più ampi non sarebbe per nulla vantaggioso.

```

1 function countingSort(A,range)
2 {
3     let C=[],B=[]
4     //inizializzazione del contatore C
5     for(let i=0;i<range;i++){
6         C[i]=0
7     }
8
9     //conteggio occorrenze di elementi di A su contatore C
10    for(i=0;i<A.length;i++){
11        C[A[i]]++
12    }
13
14    //conteggio degli elementi minori o uguali
15    for(i=1;i<range;i++){
16        C[i]=C[i]+C[i-1]
17    }
18
19    //caricamento dell'array di output
20    for(i=A.length;i>=0;i--){
21        B[C[A[i]]]=A[i]
22        C[A[i]]=C[A[i]]-1
23    }
24
25    return B
26}

```

Questo tipo di sorting è detto **stabile** perchè, in caso di parità di valore, imposta l'ordine originale dell'array di input. Questo tipo di implementazione dunque mantiene un ordine tra gli elementi anche nelle varie iterazioni, questo torna utile nel caso di ordinamenti come il Radix Sort.

## 2.2 Radix Sort

Il Radix Sort è un ordinamento che si basa sulla gestione delle cifre significative, dalla più piccola alla più grande. Quando ordiniamo ad esempio le unità non abbiamo nessuna particolare condizione da rispettare, ma passando ad esempio alle decine, dobbiamo tener conto dell'ordine generato dalla precedente iterazione in caso di decina dello stesso valore. Questo, espresso formalmente, vuol dire che necessita di un **ordinamento stabile**.

<b>1094</b>	<b>1120</b>	<b>1120</b>	<b>1094</b>	<b>125</b>
<b>986</b>	<b>1094</b>	<b>125</b>	<b>1120</b>	<b>234</b>
<b>234</b>	<b>234</b>	<b>234</b>	<b>125</b>	<b>986</b>
<b>125</b>	<b>125</b>	<b>986</b>	<b>234</b>	<b>1094</b>
<b>1120</b>	<b>986</b>	<b>1094</b>	<b>986</b>	<b>1120</b>

### 2.3 Ordinamento Stabile

Fino ad ora abbiamo analizzato vari tipi di ordinamento, ma quali di questi mantengono, a parità di "chiave", l'ordine originale? *MergeSort*, *CountingSort* e *InsertionSort* mantengono l'ordine originale a parità di chiave, dunque risultano **stabili**.

```
1 RadixSort(A, d)
2 {
3     for (i=0; i < d; i++)
4         StableSort(A) sulla cifra i
5 }
```

## 3 Programmazione Dinamica

La Programmazione Dinamica è un particolare paradigma che memorizza le soluzioni parziali su una **tabella di supporto**. Questo approccio è utilizzato solitamente per l'ottimizzazione di problemi.

**Sottoproblemi ottimi ed dipendenza** Per poter utilizzare questo paradigma è necessario che il problema originale sia divisibile in **sottoproblemi comuni dipendenti**. Potremmo ad esempio menzionare la ricerca binaria in un array ordinato come controesempio: Quando in un array ordinato, decidiamo di esplorare il sottoarray destro o sinistro, queste due "partizioni" di problema originale sono del tutto indipendenti. Accade esattamente il contrario nel caso dell'approccio dinamico sui **Numeri di Fibonacci**.

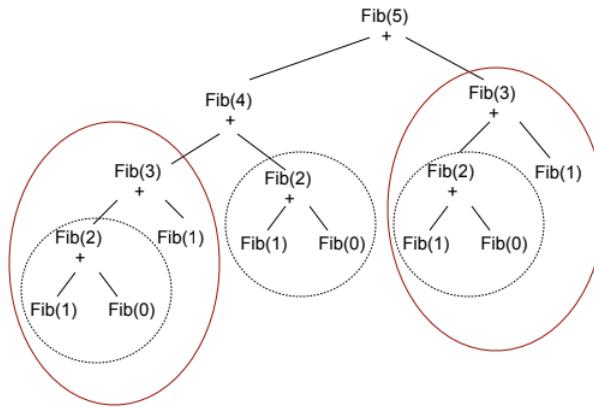
- Divide et impara
  - Partiziona il problema in sotto-problemi **indipendenti**
  - **Risolvi i sotto-problemi ricorsivamente**
  - **Combina le soluzioni per risolvere il problema originale**
- Programmazione Dinamica
  - Partiziona il problema in sotto-problemi (**non indipendenti**)
  - **Risolvi i sotto-problemi ricorsivamente**
  - **Combina le soluzioni per risolvere il problema originale**

### 3.1 Fibonacci

**Approccio Top-Down** Analizziamo il costo di Fib ricorsivo, definendo semplicemente in pseudocodice le definizioni ricorsive e il caso base dei numeri di Fibonacci.

```

1 Fib( n ) { // n >= 0
2     if (n <= 1)
3         RETURN n;
4     else
5         RETURN Fib(n-1) + Fib(n-2);
6 }
```



Possiamo dunque notare i sottoproblemi comuni che sono cerchiati. Dunque svolgendo questo problema ricorsivamente andremo a sviluppare più volte calcoli che abbiamo già effettuato.

**Approccio Bottom-Up** Proviamo ad eseguire lo stesso problema ma dalle foglie dell'albero alla radice, caricando ogni risultato in un array di supporto  $F$ .

```

1 Fib( n ) { // n >= 0
2     F[0] = 0;
3     F[1] = 1;
4     for (k = 2; k <= n; k = k + 1)
5         F[k] = F[k-1] + F[k-2];
6     RETURN F[n];
7 }
```

### 3.2 Rod Cutting

Definiamo il problema del taglio della corda, dove, avendo appunto una corda di una determinata lunghezza, e conoscendo i prezzi di ogni possibile taglio, dobbiamo determinare quale sia il modo migliore per tagliarla e venderla.

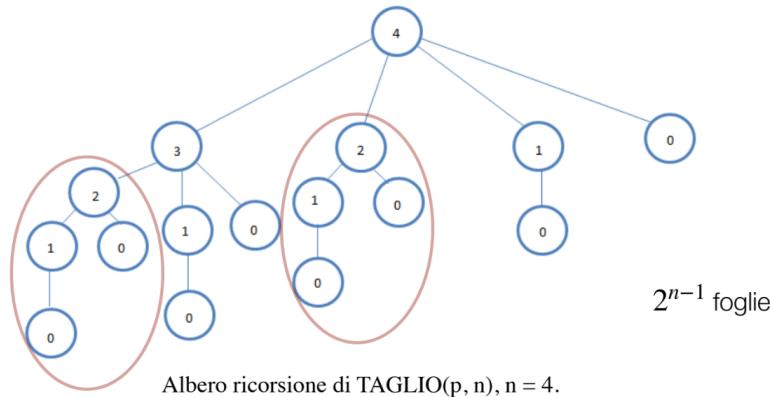
**Brute Force** Il primo metodo che potrebbe essere pensato per un problema del genere è quello di considerare tutti i possibili tagli e ne prende il massimo. Costo di quest'operazione?

$$O(2^{n-1}) \quad (18)$$

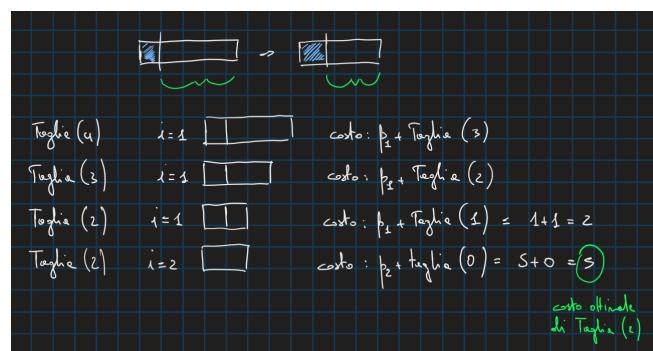
Vogliamo dunque trovare delle soluzioni che siano accessibili in termini di costo in tempo.

**Ricorsione Top Down** Consideriamo la corda in questione, e consideriamo solo una prima parte, richiamando ricorsivamente la funzione sul resto della corda.

$$r_n = \max\{p_i + r_{n-i} : 1 \leq i \leq n\} \quad (19)$$



Anche in questo caso però, notiamo dall'albero sviluppato che eseguiamo più volte gli stessi calcoli, rendendo molto meno efficiente questo algoritmo ricorsivo in top down. Perchè questo?. Perchè anche in questo caso, ad esempio, per una corda di lunghezza 3, stiamo "verificando" il costo migliore di ogni sua possibile composizione. Verificheremo quindi prima la corda lunga una unità con le possibilità restituite dal resto della corda, dato che verrà richiamata la funzione ricorsivamente sul resto. Costo di queste operazioni?  $O(2^n)$



**Soluzione Bottom Up** Come per Fibonacci, possiamo pensare di caricare una tabella che memorizzi i valori delle foglie dell'albero precedentemente rappresentato. Dunque consideriamo il massimo per ogni segmento, ma quando calcoleremo i sottosegmenti, sarà possibile accedere al loro miglior prezzo semplicemente accedendo alla tabella precedentemente caricata.

Definiamo gli elementi principali:

### 1. Tabella $r$

Definiamo la tabella  $r$  dove l'indice sarà definito dalla lunghezza del sottosegmento e conterrà il costo ottimale di ogni possibile lunghezza dei sottosegmenti.

$$r = [0, 1, 5, 8, 0] \quad (20)$$

### 2. Definizione generale del problema di ottimizzazione

$$r_n = \max\{p_i + r_{n-i} : 1 \leq i \leq n\} \quad (21)$$

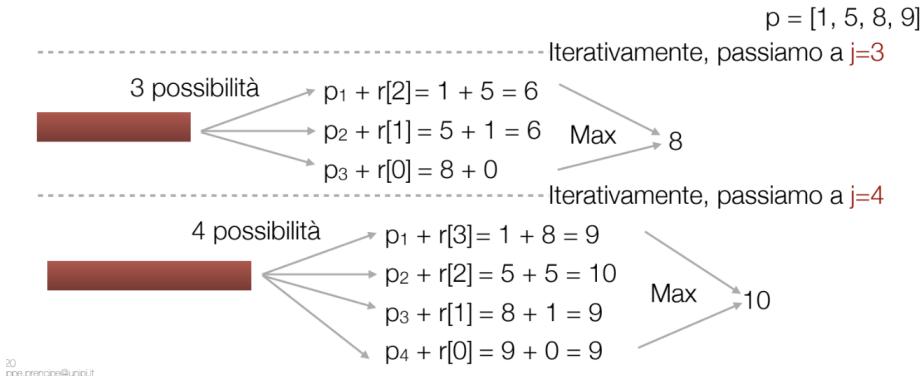
### 3. Pseudocodice dell'approccio in bottom up

```

1 TAGLIO-Dinamico(p,n)
2   r[0..n]
3   r[0]=0
4   for j=1 to n
5     q= -Infinity
6     for i=1 to j
7       if (q < p[i]+r[j-i]
8         q= p[i]+r[j-i]
9     r[j]=q
10  return r

```

### 4. Esempio di istanza della soluzione



20  
ppr.principe@unipi.it

Il tempo dunque richiesto passerà da esponenziale a lineare, sacrificando lo spazio dedicato alla tabella di appoggio. Costi?

$O(n)$  in tempo e  $O(m)$  in spazio ipotizzando  $m$  come grandezza della tabella d'appoggio.

### 3.3 LCS

Con LCS intendiamo *LongestCommonSubsequence*, dunque considerando due stringhe in input consideriamo la sottosequenza comune (non consecutiva) tra le due. Dunque, analogamente ai problemi analizzati precedentemente ci saranno vari metodi proposti, come *brute force* e *Ricorsione TopDown* e successivamente verranno elencati i vantaggi e le caratteristiche dell'approccio della *Programmazione dinamica*.

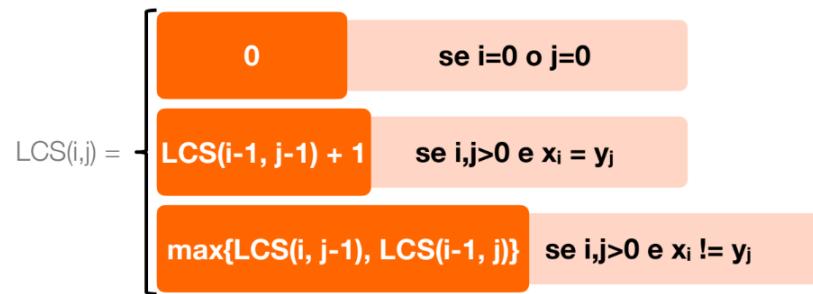
**Brute Force** Date due sequenze in input  $A$  e  $B$ , allora

$$LCS(A, B) = L(m, n) \quad m = |A| \quad e \quad n = |B| \quad (22)$$

Dunque si controlla se ogni sottosequenza in  $A$  sia o meno sottosequenza in  $B$ .

Bisogna però controllare tutte le sottosequenze, si impiega dunque tempo  $O(n2^m)$ .

**Ricorsione Top Down** Con la ricorsione Top Down possiamo considerare i prefissi delle sequenze stesse, dunque definiamo e commentiamo la ricorsione formale:



1. Caso Base: Gli indice sono entrambi a 0
2. Primo Caso Ricorsivo: Gli indici sono entrambi superiori a 0 e i due caratteri delle sottosequenze correnti sono uguali, allora l' $LCS$  viene incrementata e entrambi gli indici vengono spostati verso sinistra.
3. Secondo Caso Ricorsivo: Si prende il massimo delle due chiamate ricorsive a LCS scorrendo solo la prima verso sinistra o solo la seconda verso sinistra.

Siamo dunque riusciti a trovare dei sottoproblemi dipendenti. Dunque impostiamo il calcolo con tabella di Programmazione Dinamica. In questo caso sarà una matrice.

**Calcolo Bottom Up** Dedichiamo a questo calcolo una matrice di dimensione  $n \times m$ . Analizziamo la definizione ricorsiva formale e le operazioni che esegue di conseguenza sulla matrice

$$L(i, j) = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ L(i - 1, j - 1) + 1 & \text{se } i, j > 0 \text{ e } a[i - 1] = b[j - 1] \\ \max \{ L(i, j - 1), L(i - 1, j) \} & \text{se } i, j > 0 \text{ e } a[i - 1] \neq b[j - 1] \end{cases}$$

	0	1	2	j	n
0	0	0	0	0	0
1	0				
2	0				
i	0				
0					
m	0				

Ipotizzando di essere nella cella rossa, e di dover inserire un valore, avremo due possibilità:

- Se i due caratteri analizzati coincidono, si inserisce il valore della cella blu, posizionata in alto a destra rispetto alla corrente, incrementandone il suo valore di 1.
- Se i due caratteri analizzati sono diversi, si inserisce il massimo tra le due caselle rappresentate in giallo.

Dunque questa rappresenta l'interpretazione informale della sopracitata definizione ricorsiva formale, dove abbiamo omesso il caso base che permette l'impostazione di prima riga e prima colonna a 0.

**Esempio d'applicazione** Mostriamo un esempio pratico con due stringhe e relativa sottosequenza comune più lunga.

$$S_1 : < A, B, C, B, D, A > \quad S_2 : < B, D, C, A, B, A > \quad (23)$$

	0	1	2	3	4	5	6
Y <sub>j</sub>	B	D	C	A	B	A	
0	x <sub>i</sub>	0	0	0	0	0	0
1	A	0	0	0	1	-1	1
2	B	0	1	-1	-1	1	2
3	C	0	1	1	2	-2	2
4	B	0	1	1	2	2	3
5	D	0	1	2	2	3	3
6	A	0	1	2	2	3	4
7	B	0	1	2	2	3	4

La lunghezza dell'LCS sarà nella casella  $A[n][m]$ . Se volessimo ricostruire la sottosequenza in questione dovremmo seguire le frecce contrassegnate nella tabella, stampando il contenuto delle celle contrassegnate con una freccia diagonale.

### 3.4 Edit Distance

Vogliamo considerare la distanza massima tra due stringhe. Ci sono due possibili casi:

Consideriamo di poter inserire spazi allo scopo di allineare le sequenze

1. Caratteri uguali equivalgono a distanza 0
2. Caratteri diversi, o carattere confrontato con spazio equivale a 1

**Formulazione Ricorsiva** Descriviamo il criterio del riempimento della matrice:

$$M[i,j] = \min \{ M[i, j-1] + 1 \\ M[i-1, j] + 1 \\ M[i-1, j-1] + p(i,j) \}$$

Complessità:  $\Theta(nm)$

	0	1	2	3	4	5	6	
Y <sub>j</sub>	L	A	B	B	R	O		
0	x <sub>i</sub>	0	1	2	3	4	5	6
1	A	1						
2	L	2						
3	B	3						
4	E	4						
5	R	5						
6	O	6						

Ipotizzando che la corrente casella analizzata sia quella in basso a destra tra quelle evidenziate, imposteremo il suo valore in questo modo:

1. Se i due caratteri sono diversi si seleziona il minore tra i tre attorno alla corrente casella incrementandolo di 1
2. Se i due caratteri sono uguali, il corrente viene settato come il valore della casella in diagonale rossa

Ipotizzando due stringhe, e la tabella di Edit Distance completamente riempita seguendo la formulazione ricorsiva menzionata sopra, possiamo ricostruire anche il suo allineamento. Mostriamo un esempio commentato nella pagina successiva.

**Esercizio commentato su Edit Distance** Possiamo definire l'allineamento partendo dall'alto della matrice e "seguendo" le "frecce" che indicano quale casella ha provocato l'inserimento della successiva.

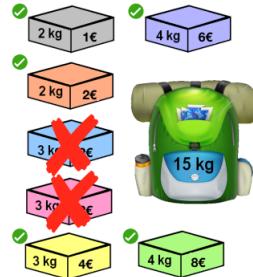
$\begin{matrix} \varnothing & C & E & N & Z & A \\ \varnothing & 0 & 1 & 2 & 3 & 4 & 5 \\ S & 1 & \swarrow & 2 & 3 & 4 & 5 \\ T & 2 & 2 & \swarrow & 3 & 4 & 5 \\ A & 3 & 3 & 3 \uparrow & 3 & 4 & 5 \\ N & 4 & 4 & 4 & 3 & 4 & 5 \\ 2 & 5 & 5 & 4 & 3 & 4 & \\ A & 6 & 6 & 5 & 4 & 3 & \end{matrix}$	$\begin{matrix} C & E & - & N & Z & A \\ S & T & A & N & Z & A \\ 1 & 1 & 1 & 0 & 0 & 0 = 3 \end{matrix}$
---	---

1. **Freccia obliqua:** caratteri di riga e colonna vanno inseriti in entrambe le stringhe che stiamo producendo
2. **Freccia orizzontale:** carattere della riga va inserito nella corrispondente stringa, mentre carattere della colonna non va inserito, sostituendolo con uno spazio
3. **Freccia verticale:** carattere della colonna va inserito nella corrispondente stringa, mentre carattere della riga non va inserito, sostituendolo con uno spazio

Una volta effettuati i passaggi raggiungendo l'ultima casella della matrice in basso a destra, possiamo definire che due caratteri diversi (o carattere spazio), confrontando le due stringhe prodotte, incrementeranno la distanza di edit di 1, mentre caratteri uguali non la incrementano.

### 3.5 Zaino 0-1

Il problema dello **zaino intero** è un esempio di approccio alla programmazione dinamica. Questo è differente dai problemi già mostrati, dato che è il primo che non si rifà ai sottoproblemi precedenti in maniera diretta. Introduciamo il problema:



**Caratteristiche del problema** Abbiamo uno zaino che può accogliere degli oggetti, con un limite di capienza in peso. Dati dunque degli oggetti con valore e peso caratteristici, l'obiettivo sarà quello di massimizzare il valore contenuto nello zaino.

$$\begin{aligned}
 & \text{Dato } \{I_1, I_2, I_3\} \text{ e' } S_3 = \{I_1, I_2, I_3\} \quad \text{con peso/valore:} \\
 & \text{Dunque } S_4 \text{ e' } S_4 = \{I_1, I_3, I_4\} \\
 & \qquad \qquad \qquad \downarrow \\
 & \qquad \qquad \qquad (S_3, I_4) \\
 & \text{soltuzione ottima:} \\
 & \text{Data delle parte innanzitutto} \\
 & \text{dopo aver sottratto il} \\
 & \text{peso del nuovo oggetto} \\
 & \text{la soluzione ottima in } S_{k+1} \text{ potrebbe} \\
 & \text{non contenere la sua diretta} \\
 & \text{sottosoluzione ottima}
 \end{aligned}$$

In questa immagine mostriamo come, la soluzione indicata con  $S_3$  indica l'insieme ottimo di tre elementi in quanto a massimizzazione di valore. Dunque definiremo  $S_3$  come soluzione ottima a 3 elementi. Notiamo però che  $S_4$  oppure un generico  $S_{k+1}$  potrebbe non contenere la precedente soluzione. Infatti, tenendo sempre la massimizzazione del valore come obiettivo, si nota che rimuovendo  $I_3$  da  $S$  ed aggiungendo  $I_4$  otterremo la soluzione ottima per 4, dunque  $S_4$ .

**Sintesi** La soluzione ottima in  $S_{k+1}$  potrebbe non contenere la sua diretta precedente sottosoluzione  $S_k$

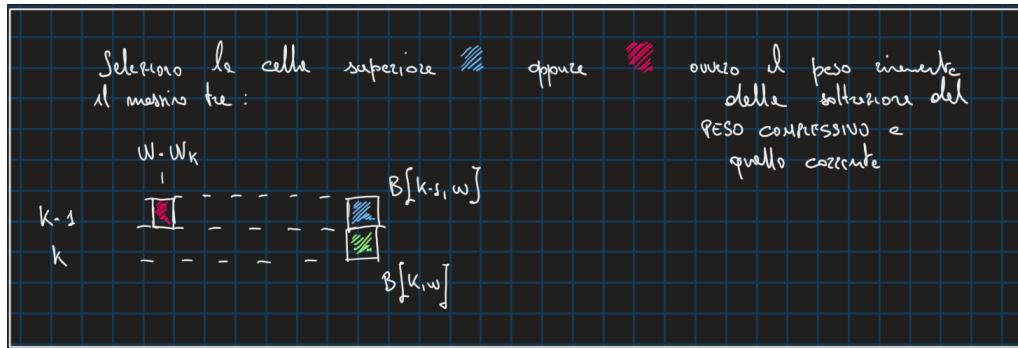
**Forza Bruta** La soluzione in forza bruta del problema dello zaino intero porterebbe ad una complessità  $2^n$ , dato che sarebbe necessario controllare tutte le possibili composizioni. Proviamo ad ottimizzare questo costo.

**Definizione Ricorsiva :**

$$B[k, w] = \begin{cases} B[k-1, w] & \text{se } w_k > w \\ \max\{B[k-1, w], B[k-1, w - w_k] + v_k\} & \text{altrimenti} \end{cases}$$

La definizione ricorsiva si divide dunque in due casi:

1. Il peso dell'oggetto è maggiore del peso complessivo trasportabile, dunque considero i  $k$ -esimi meno uno elementi, ovvero gli elementi presenti senza il corrente.
2. Cerco il massimo tra la composizione senza il corrente e la composizione più il corrente, rimuovendo un elemento dello stesso peso del corrente.



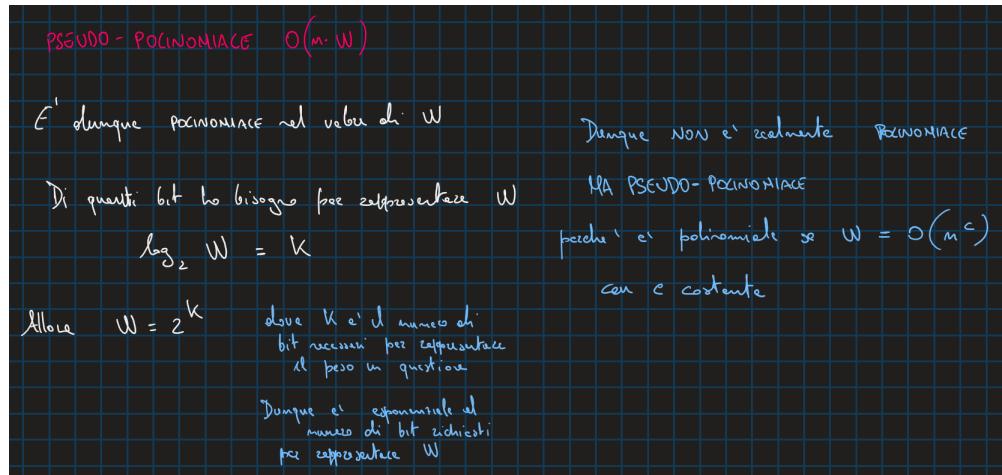
**Esempio d'istanza del problema** Mostriamo un istanza del problema Zaino 0-1 su matrice, dati quattro oggetti con (peso,valore) (2,3) (3,4) (4,5) (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

La tabella va letta in questa maniera: La riga indica quanti possibili elementi **in ordine** stiamo considerando nella combinazione. Il peso invece indica quanto può occupare la combinazione considerata. Dunque, facendo degli esempi:

1. Riga 1: Considero solo il primo elemento da peso 0 a 5
2. Riga 2: Considero le combinazioni tra primo e secondo elemento da peso 0 a 5
3. Riga 3: Considero le combinazioni tra primo, secondo e terzo elemento da peso 0 a 5...

**Analisi del costo - Zaino 0-1 e Programmazione Dinamica** Il costo dell'approccio dinamico per questo problema sarà  $O(n * W)$ , definita come **pseudo-polinomiale**. Perchè pseudo? Perchè per la prima volta non stiamo considerando solo il numero di elementi, ma anche una loro qualità, in questo caso il peso. Sarà dunque necessario valutare quanti bit saranno necessari per rappresentare il peso in questione.



In questa figura infatti mostriamo come potrebbe diventare persino esponenziale rispetto a  $k$ , ovvero i bit necessari per rappresentare il peso in questione.

### 3.6 Greedy

Con Greedy intendiamo una strategia basata sull'ottimo locale, nella speranza che coincida con l'ottimo globale. Non sempre infatti la strategia Greedy coincide con l'ottimo globale della soluzione, ma lo farà negli esempi proposti, dunque **Zaino Frazionario** e **Scheduling di Attività**.

#### 3.6.1 Zaino Frazionario

Mostriamo l'esempio della strategia Greedy sulla versione frazionaria dello zaino. Consideriamo dunque il rapporto costo/peso delle frazioni in questione, dando priorità a questo durante il riempimento dello zaino con le varie frazioni.



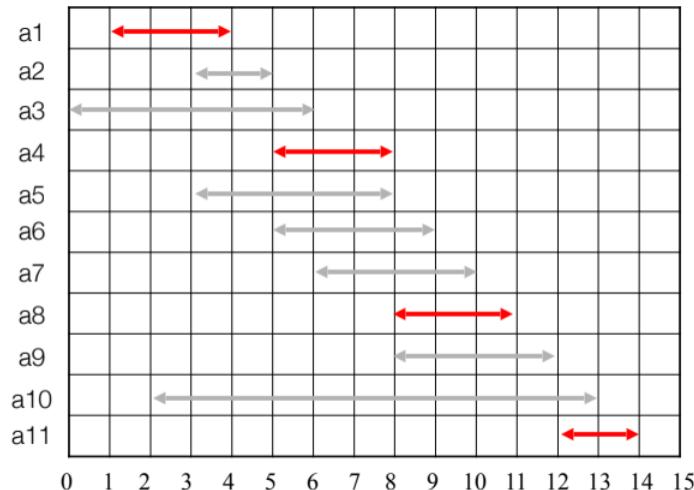
**Simulazione selezione oggetti e costi** Dato dunque il rapporto costo peso, stabiliamo un ordinamento tra gli oggetti dati su questo criterio e selezioniamo man mano in ordine decrescente gli oggetti dal rapporto maggiore a quello minore. Stiamo dunque dando priorità al massimo corrente. Dunque i costi saranno  $\Theta(n)$  se gli oggetti sono già ordinati in base al rapporto dato, altrimenti il costo corrisponderà a  $\Theta(n \lg n)$ .

### 3.6.2 Scheduling di attività

Lo scheduling di attività si pone come ottimo esempio per l'applicazione di una strategia Greedy. Assumiamo che, dato un insieme di attività, il nostro obiettivo sarà quello di determinare la combinazione che ci permetta di inserire quante più attività possibili.

Tutto l'algoritmo della scelta in Greedy dello scheduling si basa su questi passi:

1. Scelgo un'attività da cui cominciare
2. Scarto tutte le attività il cui inizio è compreso nell'intervallo della prima attività (scarto le attività incompatibili)
3. Scelgo tra le rimanenti l'attività che finisce prima tra le attività compatibili (applicazione della strategia Greedy)
4. Reitero il procedimento fino alla conclusione di attività



Dato questo esempio, è necessario stabilire **quando** la strategia Greedy porta alla soluzione ottima globale.

### 3.6.3 Dimostrazione Soluzione Ottima del Greedy

Per dimostrare che una soluzione greedy sia ottima anche globalmente è necessario dimostrare:

1. Il problema ha una **sotto-struttura ottima**, dunque possiamo approcciare il sotto-problema in maniera simile
2. Viene soddisfatta la **greedy-choice property**, ovvero
  - (a) Dimostrare che il greedy porta alla soluzione ottima
  - (b) Soluzione greedy che procede in **top-down**

**Dimostrazione di esistenza soluzione ottima** Dimostriamo che il Greedy coincide con la soluzione ottima,

#### Elementi Fondamentali

1.  $A$  soluzione ottima
2.  $S$  insieme di attività da schedulare
3.  $a_1$  scelta Greedy
4.  $a_k$  primo elemento dell'ottimo  $A$

#### Confronto di primo elemento e Scelta Greedy

1. Dato  $A$ , il suo primo elemento sarà  $a_k$ , e la scelta effettuata dal Greedy sarà  $a_1$ . Abbiamo due possibilità:
  - (a)  $a_1 = a_k$  Scelta Greedy coincide con primo elemento di  $A$  ottimo.
  - (b) Scelgo una nuova soluzione  $B$ , dato che *fine* di  $a_1$  è minore o uguale a *fine* di  $a_k$ , allora sostituiamo  $a_k$  con  $a_1$ . Notiamo che il numero di attività in  $B$  coincide con il numero di attività in  $A$ . Dunque anche  $B$  è ottimo, dato che l'obiettivo corrente è quello di massimizzare il numero di attività.

#### Resto degli elementi e induzione

1. Se  $A$  è ottimo per  $S$ , allora  $A'$  sarà ottimo per  $S'$ , dove  $S'$  è l'insieme degli elementi appartenenti ad  $S$  tali che ogni tempo d'inizio delle attività in  $S'$  sia successivo alla fine dell'attività  $a_1$ .
2. La validità in questo contesto di  $S'$  è dimostrabile per assurdo, dato che estendendo un ipotetico  $B'$  ed aggiungendo ad esso l'attività  $a_1$  avremmo generato una soluzione  $B$  per  $S$  più grande dell'ottimo  $A$ .

## 4 Grafi

Definiamo un grafo come una coppia  $G = (V, E)$ , dove  $V$  è l'insieme dei vertici e  $E$  l'insieme degli archi.

### 4.1 Definizioni e caratteristiche

**Grafo Denso/Sparso** Un grafo è detto:

1. Sparso :  $|E| \approx |V|$
2. Denso :  $|E| \approx |V|^2$

Ricordiamo anche che il massimo numero di archi per un grafo è  $|V|^2$ .

**Componente connessa** Presi due nodi  $v, u$  questi fanno parte della stessa componente connessa se sono raggiungibili con un cammino da  $v$  a  $u$ .

**Self-Loop/Cappio** In un grafo orientato, un arco che va da un generico nodo  $u$  a  $u$  è definito Self-Loop.

**Rappresentazione in matrice di adiacenza** Un grafo potrebbe utilizzare come supporto di memorizzazione una matrice che utilizza come indici gli stessi nodi e pone 1 nella cella corrispondente ad un arco esistente e 0 dove non esiste un arco tra i nodi presi in considerazione.

	0	1	2	3	4	5
0	1	1				1
1				1		1
2			1			
3				1	1	1
4						
5				1	1	

Questo è solo un tipo di possibile supporto di memorizzazione per i grafi. Questi ultimi infatti potrebbero essere memorizzati con liste di adiacenza.

**Grafo Pesato** Un grafo pesato associa ad ogni arco un elemento. Solitamente questi elementi sono interi, dato che diamo per scontato che esista un'algebra ed un ordinamento su questi ultimi. Ma stabilendo queste due caratteristiche su altri oggetti, potremmo associare ad ogni arco una qualsiasi etichetta, non solo numeri.

**Cammino** Un cammino è un insieme di nodi connessi, tali che partendo da un nodo  $v$  è possibile raggiungere il nodo  $u$ .

**Spanning Tree** Uno Spanning Tree, detto anche Albero di Copertura, è un albero che "certifica" la connessione tra nodi. Dunque rappresenta l'albero minimo di connessione tra i nodi di un grafo.

## 4.2 Attraversamento di Grafo

Definiamo l'attraversamento di un grafo  $G = (V, E)$  con vertice  $s \in V$  detto *sorgente* l'esplorazione di ogni vertice raggiungibile nel grafo dal vertice  $s$ .

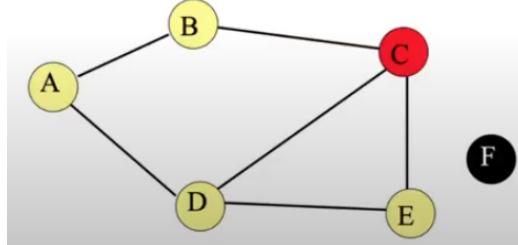


Figure 5: In questo caso la sorgente è il nodo C, F è l'unico nodo non raggiungibile dalla sorgente

**Shortest Path** Dati due nodi  $u, v$ , uno shortest path è un cammino minimo (contenente il numero minimo di archi) tra i due nodi. Solitamente, contestualizzato all'attraversamento di grafi, uno dei due nodi è settato alla sorgente. Ricordiamo che i cammini minimi in un grafo non sono unici, ma potrebbero esisterne diversi.

**Distanza dalla sorgente** I cammini minimi tengono considerazione solo del numero di archi tra i due nodi in questione,  $\delta(s, v)$  come distanza tra la sorgente ed un nodo dato.

### 4.2.1 BFS - Breadth First Search

La BFS è la visita in ampiezza di un grafo. Questo tipo di visita sfrutta come struttura dati di supporto una **coda**. Viene dunque messa in coda la sorgente da cui si parte e successivamente si valuta, aggiungendo il suo vicinato . Dunque "valutare" corrisponde ad inserire in coda il vicinato di un nodo. Ricordiamo che il vicinato di un nodo è l'insieme di nodi a cui è connesso il nodo in questione.

**Versione 1/Versione 2** Facendo riferimento alle slide del corso, vengono mostrati due diversi modi di implementare la BFS. La prima tra queste implementazioni non gestiva il caso in cui fosse presente un ciclo nel grafo. Dunque si implementa la seconda versione che grazie alla gestione "a colori" permette di valutare lo stato di un nodo:

1. Colore Bianco: Il nodo in questione non è mai stato inserito in coda, quindi non è mai stato visitato.
2. Colore Grigio: Il nodo in questione è stato inserito in coda, ma ancora non è stato valutato.
3. Colore Nero: Il nodo è stato inserito in coda ed è stato anche rimosso dalla coda stessa essendo stato valutato.

**Analisi del costo in tempo** Analizziamo il costo dell'algoritmo proposto nelle slide.

Il costo in tempo è scandito dalla visita delle liste di adiacenza che stiamo ipotizzando come tipo di memorizzazione del grafo. Per ogni vertice, bisogna valutare tutta la relativa lista di adiacenza, dunque la lista visiterà al massimo ogni arco, è scandita una sola volta ogni lista di adiacenza. E lo farò per ogni nodo. Dunque il costo sarà:

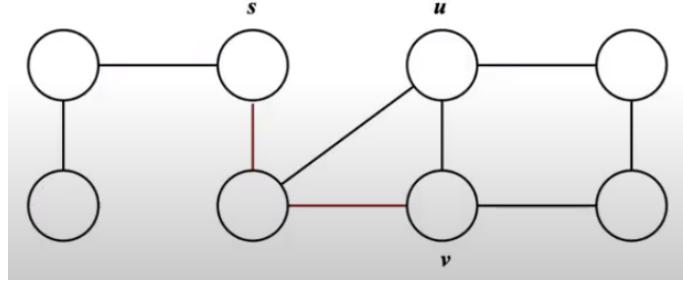
$$O(|V| + |E|)$$

**Analisi del costo in spazio** Dobbiamo memorizzare tutti i nodi, e la coda, nella peggiore delle ipotesi, dovrà mantenere tutti i nodi, e non può crescere più del numero di nodi stesso. Dunque lo spazio sarà:

$$O(|V|) \quad (24)$$

**Shortest Path Distance e BFS** Considerando un generico nodo  $v$  e la sorgente  $s$ , la BFS permette il calcolo della distanza minima tra  $v$  ed  $s$ , come  $\delta(s, v)$ . Questo si basa sulla proprietà tale che dato  $(u, v)$

$$\delta(s, v) \leq \delta(s, u) + 1 \quad (25)$$



Dunque la gestione del minimo numero di archi tra sorgente e nodo è di natura induttiva.

### Caratteristiche BFS

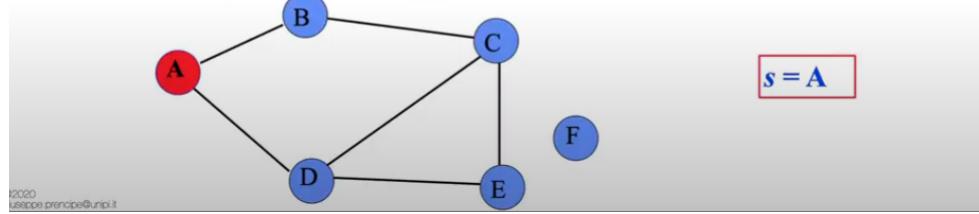
1. Trova il cammino di lunghezza minima
2. In grafi non pesati, trova il costo ottimo
3. In grafi pesati, non trova il costo ottimo

**BFS e relativo Spanning Tree** Abbiamo definito uno Spanning Tree come un albero che certificava la connessione tra nodi. La BFS permette la costruzione di uno Spanning Tree su un grafo, scegliendo tra i possibili shortest path sul criterio della visita in ampiezza.

---

Un *albero breadth-first* di un grafo  $G = \langle V, E \rangle$  con *sorgente*  $s$ , è un *albero*  $G' = \langle V', E' \rangle$  tale che:

- $G'$  è un *sottografo* del grafo sottostante  $G$
- $v \in V'$  se e solo se  $v$  è raggiungibile da  $s$
- per ogni  $v \in V'$ , il percorso da  $s$  a  $v$  è *minimo*



**Sottografo dei predecessori** Sottografo che certifica la connessione definito su sottoinsiemi di vertici e archi.

Dato  $G = \langle V, E \rangle$  e un nodo sorgente  $s$ , il *sottografo dei predecessori* è  $G_{pred} = \langle V_{pred}, E_{pred} \rangle$ , dove:

$$V_{pred} = \{v \in V : v.p \neq \text{Nil}\} \cup \{s\}$$

$$E_{pred} = \{(v.p, v) \in E : v \in V_{pred} - \{s\}\}$$

---

Gli archi del sottografo sono detti *tree edges*. Dunque ogni nodo, tranne la sorgente avrà esattamente un predecessore.

**Operazioni su BFS** Alcune operazioni, come la stampa degli shortest path, data la sorgente  $s$  e un nodo generico  $v$ , si basa sull'albero prodotto dalla BFS, e visitando i nodi richiesti. Dunque molte operazioni simili si basano sull'utilizzo dei dati calcolati durante la costruzione dell'albero minimo.

#### 4.2.2 DFS - Depth First Search

La DFS è la visita in profondità di un grafo. Questo tipo di visita sfrutta come struttura dati una **pila**. Viene naturale implementare ricorsivamente questo tipo di visita data la pila dei record di attivazione della funzione ricorsiva in questione che gestirebbe automaticamente la struttura dati.

Depth-First Search

*Tempo di DFS = O(V+E)*

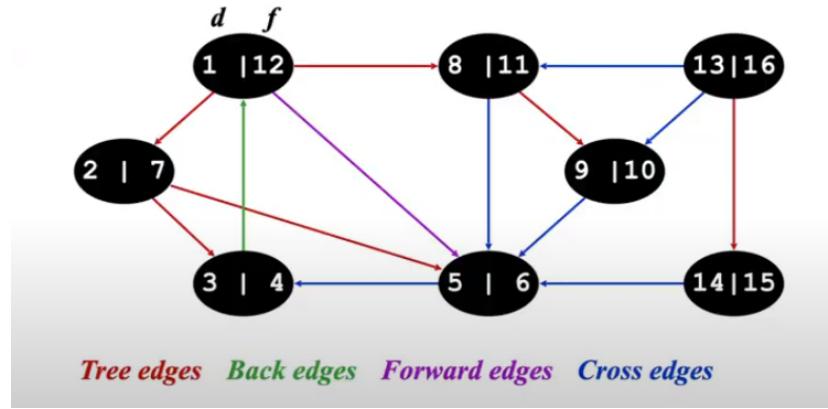
```

DFS(G)
{
    for each vertex u ∈ V
    {
        u.color = BIANCO;
        u.p = NIL;
    }
    time = 0;
    for each vertex u ∈ V
    {
        if (u.color == BIANCO)
            DFS_Visit(u);
    }
}

DFS_Visit(u)
{
    u.color = GRIGIO;
    time = time+1;
    u.d = time;
    for each v ∈ u.adj
    {
        if (v.color == BIANCO)
            v.p = u
            DFS_Visit(v);
    }
    u.color = NERO;
    time = time+1;
    u.f = time;
}

```

**Visita in DFS e tipi di archi** La DFS presenta quattro possibili tipi di archi:



- DFS introduce una importante distinzione tra gli archi del grafo iniziale
  - **Tree edge**: viene incontrato un nuovo vertice (bianco)
  - **Back edge**: da un discendente a un antenato
  - **Forward edge**: da un antenato a un discendente
    - Non un tree edge
    - Da un nodo **GRIGIO** a un nodo **NERO**

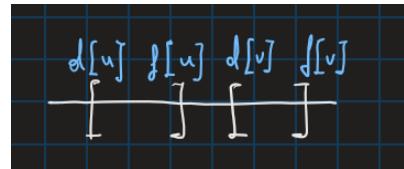
Queste classificazioni degli archi nella DFS permette una visualizzazione grafica dei casi previsti dal *Parenthesis Theorem* che formalizza il concetto che sta dietro al "colore" dei vertici, sfruttando degli intervalli.

**Parenthesis Theorem** Per ogni  $u, v$ , indichiamo

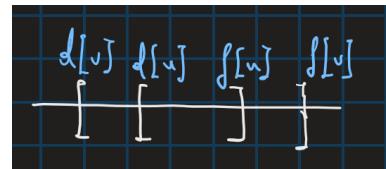
1.  $d[u] = f[u]$  l'intervallo del nodo  $u$ .
2.  $d[v] = f[v]$  l'intervallo del nodo  $v$ .

Dovrà dunque essere valida esattamente una tra queste condizioni:

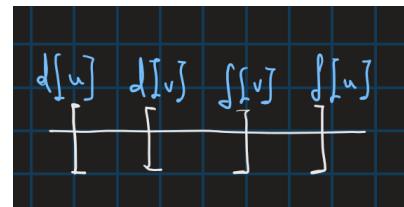
1. Gli intervalli  $d[u] - f[u]$  e  $d[v] - f[v]$  sono **completamente disgiunti**, che corrisponde al fatto che né  $u$  né  $v$  sono discendenti uno dell'altro.



2. L'intervallo  $d[u] - f[u]$  è **completamente contenuto** nell'intervallo  $d[v] - f[v]$ , dunque  $u$  è discendente di  $v$ .



3. L'intervallo  $d[v] - f[v]$  è **completamente contenuto** nell'intervallo  $d[u] - f[u]$ , dunque  $v$  è discendente di  $u$ .



#### 4.2.3 Topological Sort

Un ordinamento topologico è un esempio di ordinamento che risolve delle **dipendenze** tra elementi. Possiamo dunque estrapolare un possibile ordinamento topologico da un grafo grazie all'utilizzo di una **DFS** e di una **PILA**. E' però necessaria una specifica condizione, ovvero:

1. Esiste un Ordinamento Topologico  $\Leftrightarrow$  il Grafo è un DAG

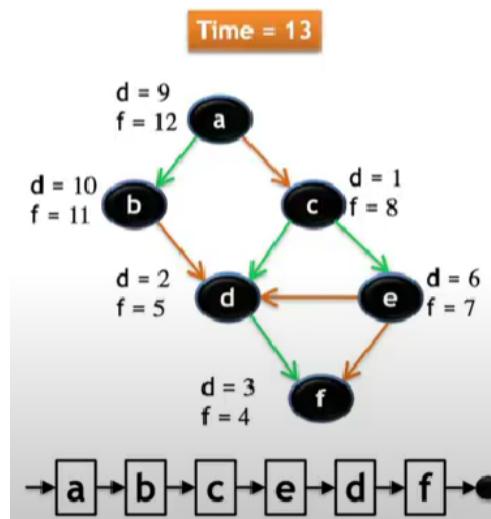
L'obiettivo dell'ordinamento topologico è dunque quello di stabilire se esiste o meno "la dipendenza" tra gli elementi. A causa di questo non può essere presente un ciclo nel grafo, perchè staremmo affermando che  $A$  dipende da  $B$  e che contemporaneamente  $B$  dipende da  $A$ . Questo non è possibile ed è definito come **deadlock**.

**Implementazione con Pila e DFS** Descriviamo l'algoritmo che implementa l'ordinamento topologico in passi:

1. Eseguo un *DFS* su grafo  $G$ , calcolando dunque i tempi di fine per ogni vertice  $v$  rispetto ad una sorgente data
2. Quando un vertice generico  $v$  viene del tutto valutato viene inserito in una pila
3. Restituisco la pila generata durante la valutazione di tutti i vertici

Questa implementazione sfrutta dunque la proprietà *LIFO* della pila per rappresentare uno dei possibili ordinamenti topologici dato un grafo.

**Costo del Topological Sort** L'unica differenza tra queste operazioni ed una visita in *DFS* è l'inserimento nella pila che costa  $O(1)$ , dunque asintoticamente costerà quanto la visita in *DFS*, quindi  $O(|n| + |m|)$ .



#### 4.2.4 Algoritmo di Dijkstra

L'Algoritmo di Dijkstra ha l'obiettivo di determinare un insieme  $S$  in un grafo pesato, dove sono stati "salvati" i valori della distanza in peso di ogni nodo dalla sorgente  $s$  data.

L'intero algoritmo si basa sull'utilizzo di una coda di priorità e dell'operazione di *relaxation*. Descriviamo in passi l'algoritmo:

- 1. Inizializzazione:** Si setta la distanza dalla sorgente  $s$  per ogni nodo del grafo  $G$ . Dunque ci sono due possibili casi:

- (a) **Il nodo corrisponde alla sorgente:** La sua distanza dalla sorgente sarà settata a 0.
- (b) **Il nodo non corrisponde alla sorgente:** La sua distanza dalla sorgente sarà settata ad  $\infty$ .

- 2. Iterazione sulla Min Coda:** Si seleziona dalla coda l'elemento con distanza minore, e si rilassano gli adiacenti, iterando su questo criterio fino a quando la coda non è vuota. Dunque descrivendo dei passi generici:

- (a) Si seleziona il primo nodo con priorità maggiore nella coda, ovvero la sorgente, avendo distanza 0 dalla sorgente stessa.
- (b) Selezionare vuol dire effettuare l'operazione di *relax* sui nodi adiacenti, seguendo la priorità impostata dalla coda. Selezionare un nodo permette la sua rimozione dalla coda.
- (c) L'operazione di *relax* vuol dire verificare se la potenziale distanza attuale dalla sorgente è migliore di quella già associata al nodo stesso. Dunque si effettua uno scambio se la distanza nuova è più breve.

```

1 Dijkstra(G, s) {
2     Q = empty vertex priority queue;
3     for each v in V {
4         if (v == s) v.d = 0
5         else v.d = infinity;
6         v.p = nil;
7         add v to Q with priority v.d;
8     }
9     while (Q != empty) {
10        u = vertex with min priority in Q;
11        for each v in u.adj {
12            alt = u.d + weight(u,v);
13            if (alt < v.d) {
14                v.d = alt;
15                v.p = u;
16                decrease_priority(Q,v,alt)
17            }
18        }
19    }
20 }
```

**Analisi del costo** Analizziamo il costo dell'algoritmo di Dijkstra implementato con coda di min priorità:

```

Dijkstra(G, s) {
    Q = empty vertex priority queue;
    for each v in V {
        if (v == s) v.d = 0
        else v.d = infinity;
        v.p = nil;
        add v to Q with priority v.d; |O(log |V|)
    }
    while (Q != empty) {
        u = vertex with min priority in Q; |O(log |V|)
        for each v in u.adj {
            alt = u.d + weight(u,v);
            if (alt < v.d) {
                v.d = alt;
                v.p = u;
                decrease_priority(Q,v,alt) |O(log |V|)
            }
        }
    }
}

```

$O(|V|)$

$O(|V| \log |V|)$

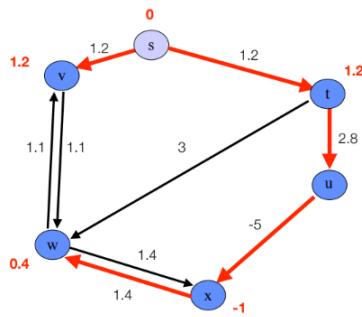
$O(|E|)$

$O(|V| \log |V| + |E| \log |V|)$

Il costo finale dell'algoritmo sarà dunque

$$O(|V| + |E| \log(|V|)) \quad (26)$$

**Commento su cicli di peso negativi** Questo algoritmo presenta un problema, si infrangerebbe infatti la proprietà **monotona** che permette l'accumulo e il calcolo di nuove distanze su quelle precedenti. Questo problema viene risolto nell'algoritmo proposto da Bellman Ford, a costo di una complessità in tempo maggiore.



Con **pesi negativi**, una soluzione esiste sempre?

Se esistesse un **ciclo di costo negativo** raggiungibile da s?

Il costo minimo sarebbe  $-\infty$ !

soluzione ottima

**Dimostrazione per Invariante** : Dimostriamo per invariante l'algoritmo di Dijkstra

Rendiamo nota la proprietà che vogliamo che sia rispettata alla fine dell'esecuzione dell'algoritmo stesso:

$$PROP1 : \forall v \in V \mid v.d = \delta(s, v) \quad (27)$$

Dunque che ogni vertice  $v$  abbia in  $v.d$  la lunghezza dello shortest path da  $s$  a  $v$ .

1. **Inizializzazione:**  $PROP1$  è valida nella sorgente, assumendo che

$$\delta(s, s) = 0$$

2. **Mantenimento:**

- (a) Consideriamo un vertice visitato  $u$  e uno non visitato  $v$  con priorità massima, influirebbe in modo meno pesante possibile sulla somma
- (b) Assumiamo allora che  $v.d = u.d + weight(u, v)$
- (c) Per dimostrare che la scelta del nodo  $v$  sia la migliore effettuabile, scegliamo per assurdo un altro nodo che arrivi in maniera migliore a  $v$ , dunque un ipotetico nodo  $z$  per assurdo. Allora abbiamo due sottocasi:
  - i. Se  $z$  **non visitato**,  $z.d > v.d$ , allora abbiamo una contraddizione, dato che avevamo affermato che la scelta migliore sul peso sarebbe stata quella di  $v$
  - ii. Se  $z$  **visitato**, allora avremmo  $v.d = z.d + weight(z, v)$ , ma questo causa contraddizione dato che avevamo stabilito che  $v.d = u.d + weight(u, v)$

3. **Conclusione:** Alla fine dell'algoritmo termina l'estrazione dalla coda, dunque non ha senso controllare ulteriori condizioni se non durante l'esecuzione stessa.

#### 4.2.5 Algoritmo di Bellman Ford

Mostriamo il relativo pseudocodice e successivamente descriviamolo:

```

1 Bellman-Ford(G, s) {
2     for each v in V {
3         if (v == s) v.d = 0
4         else v.d = infinity;
5         v.p = nil;
6     }
7     for (i=1, i<|V|, i++) {
8         for each (u,v) in E {
9             alt = u.d + weight(u,v);
10            if (alt < v.d) {
11                v.d = alt;
12                v.p = u;
13            }
14        }
15    }
16    for each (u,v) in E {
17        if (v.d > u.d + weight(u,v))
18            return false
19    }
20    return true
21 }
```

Questa implementazione ha lo scopo di ritornare *true* se esiste nel grafo dato un **ciclo di peso negativo**, altrimenti ritorna *false*. Possiamo descrivere le sue parti fondamentali:

##### 1. Inizializzazione:

```

1 for each v in V {
2     if (v == s) v.d = 0
3     else v.d = infinity;
4     v.p = nil;
5 }
```

Si setta la distanza dalla sorgente di ogni nodo a  $\infty$  e la sorgente stessa a 0.

## 2. Ciclo e Rilassamento per ogni nodo:

```
1 for (i=1, i<|V|, i++) {  
2     for each (u,v) in E {  
3         alt = u.d + weight(u,v);  
4         if (alt < v.d) {  
5             v.d = alt;  
6             v.p = u;  
7         }  
8     }  
9 }
```

Per ogni nodo tranne la sorgente, si tenta il rilassamento calcolando  $ALT$  ovvero la potenziale nuova distanza dalla radice per ogni arco del nodo selezionato. Se  $ALT$  è migliore del corrente  $v.d$  allora  $v.d$  viene sostituito con  $ALT$ .

## 3. Controllo di cicli negativi:

```
1 for each (u,v) in E {  
2     if (v.d > u.d + weight(u,v))  
3         return false  
4     }  
5     return true  
6 }
```

Controlla la presenza di cicli negativi all'interno del grafo, iterando su ogni arco.

## 5 Complessità Computazionale

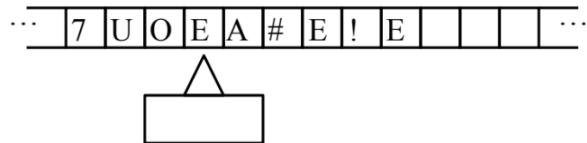
Per introdurre l'argomento, è prima necessario stabilire quale sia il modello computazionale di riferimento attuale.

### 5.1 Modelli Computazionali

**Modello RAM** L'astrazione dei moderni calcolatori è caratterizzata dalle seguenti caratteristiche:

1. Ogni cella di memoria può contenere una quantità di dati finita.
2. Impiega lo stesso tempo per accedere ad ogni cella di memoria.
3. Le principali operazioni che può effettuare sono **lettura** e **scrittura** in 1 unità di tempo.

**Macchina di Turing** Potremmo rappresentare un modello computazionale alternativo che è però in grado di effettuare le stesse operazioni del modello **RAM** moderno:



Questo modello si basa su una *testina*, un *processore* ed un *programma*. In 1 unità di tempo può:

1. Leggere o scrivere nella cella corrente
2. Spostarsi (destra o sinistra) oppure rimanere fermo.

Il punto di questa introduzione è quello di mostrare che modelli diversi possono effettuare simili operazioni. Dunque volendo discutere del tempo di esecuzione di un programma, sarebbe necessario considerare troppe variabili se non volessimo concentrarci solo sull'efficienza della soluzione proposta ma di un efficienza sistematica.

### 5.2 Classi Computazionali

In questa sezione analizzeremo non più la **complessità di algoritmi** ma la **complessità di problemi**. Distinguiamo algoritmi e problemi:

1. **Complessità Algoritmo:** Dato un problema e proposta una soluzione algoritmica a quel problema, il costo sarà determinato dall'organizzazione e dalle strutture dati d'appoggio scelte.
2. **Complessità Problema:** Dato un problema, e proposto un algoritmo generico  $A$ , questo termina? Riusciamo a trovare degli algoritmi di risoluzione al problema che terminano? Questo stabilisce la classificazione **DECIDIBILE / INDECIDIBILE**.

### 5.2.1 Problemi Indecidibili

Un problema  $P$  è detto **indecidibile** se non esiste un algoritmo generico  $A$  risolutivo che termina.

**Problema della fermata** Impostiamo l'analisi di un problema indecidibile:

1. **Osservazioni necessarie:**

- (a) Una sequenza di simboli può essere interpretata come dato oppure come programma.
- (b) Un programma può essere dato in input ad un altro programma (come i compilatori con gli algoritmi).

2. **Ipotesi:** Ipotizziamo un algoritmo  $\text{Termina}(A, D)$  che può restituire in tempo finito uno di questi esiti:

- (a)  $A$  (algoritmo al primo argomento), **termina** con input  $D$  (al secondo argomento)
- (b)  $A$  (algoritmo al primo argomento), **va in loop** con input  $D$  (al secondo argomento).

3. **Layer aggiuntivo:** consideriamo un algoritmo *paradosso*, e diamo in input a *paradosso* l'algoritmo stesso, basandoci sul fatto che un algoritmo può ricevere in input  $D$  un altro algoritmo.

4. **Contraddizione:** Stabilendo l'algoritmo paradosso in questo modo:

```
1 Paradosso(A) {
2     if Termina(A, A)
3         LOOP()
4     else
5         FALSE
6 }
```

Siamo allora riusciti ad impostare la seguente contraddizione:

$$\text{Paradosso(Paradosso)} \text{ termina} \Rightarrow \text{Paradosso(Paradosso)} \text{ non termina} \quad (28)$$

5. **Conclusione:** Con questo siamo riusciti a concludere che il problema in questione appartiene alla **classe degli indecidibili**.

### 5.2.2 Problemi Decidibili

I problemi **decidibili** sono dunque caratterizzati dall'essere risolvibili da un algoritmo che termina. E' però necessario in questo caso stabilire due nuove classificazioni:

1. **Problemi Trattabili:** Si può provare che questi problemi siano risolvibili da un algoritmo in tempo **polinomiale**.
2. **Problemi Intrattabili:** Non esiste un algoritmo con complessità polinomiale in grado di risolverlo. Sono contenuti anche i presunti intrattabili, dei quali si ipotizza che non esista un algoritmo di soluzione in input in tempo **polinomiale**.

Lo studio delle classi di complessità analizzerà a fondo questo tipo di problemi. Da ora in avanti dunque verranno analizzati problemi appartenenti a questa classificazione.

## 5.3 Problemi e Notazione

Definiamo altre classificazioni di problemi e notazione generale

### 5.3.1 Definizione Formale di Problema

Un problema generico  $P$  è:

$$P = I \times S \quad (29)$$

1.  $I$ : Insieme delle istanze in ingresso del problema.
2.  $S$ : Insieme delle soluzioni del problema.

Possiamo anche immaginare un *predicato*<sub>1</sub> che data un'istanza  $x \in I$  ed una soluzione  $s \in S$ , restituisce 1 se  $(x, s) \in P$ . Questo sarà utile nei **problemi di decisione**.

### 5.3.2 Tipologie di Problemi

Classifichiamo ulteriormente i problemi in base alla risposte richieste:

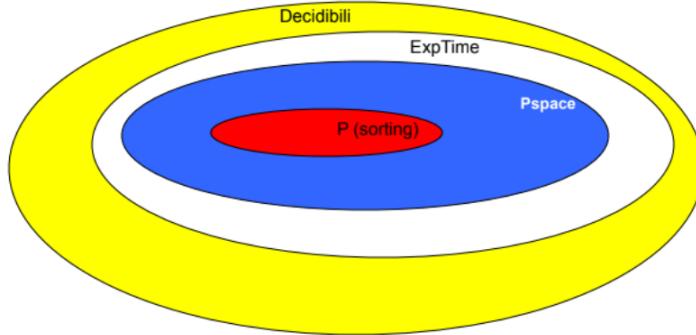
1. **Problemi di decisione:** Richiedono una risposta binaria in uscita.  
Dunque richiedono delle istanze positive tali che il *predicato*<sub>1</sub> ritorni 1 o 0.
2. **Problemi di ricerca:** Richiedono una soluzione  $s \in S$  tale che  $(x, s) \in P$ .
3. **Problemi di ottimizzazione:** Particolare problema di ricerca si cerca infatti una soluzione  $s \in S$  tale che  $(x, s) \in P$  tale che  $s$  sia la soluzione migliore.

Essendo dunque che un problema di *ottimizzazione* è un particolare tipo di problema di *ricerca*, allora possiamo stabilire che il problema di ottimizzazione è difficile **almeno quanto** quello relativo di ricerca.

### 5.3.3 Classi di Complessità

Stabiliamo dunque delle classi in base alle caratteristiche del problema decisionale generico preso in analisi. Enumeriamo le classi in questione:

1. **ExpTime**: Classe dei problemi risolvibili in **tempo esponenziale** rispetto ad un entità di dimensione  $n$  in input.
2. **Pspace**: Classe dei problemi risolvibili richiedendo uno **spazio polinomiale** rispetto ad un entità di dimensione  $n$  in input.
3. **P**: Classe dei problemi risolvibili richiedendo uno **tempo polinomiale** rispetto ad un entità di dimensione  $n$  in input.



**Esempi di Problemi appartenenti ad ExpTime** : Possiamo descrivere alcuni problemi che non sono risolvibili se non con algoritmi in tempo esponenziale analizzando le definizioni ricorsive richiamate negli algoritmi. Ne è un esempio **La Torre di Hanoi**.

```

hanoi(n,A,B,C):
    if (n ==1)
        sposta da A a C;
    else{
        hanoi(n-1,A,C,B);
        sposta da A a C;
        hanoi(n-1,B,A,C);
    }

```

- $h(1) = 1$
- $h(N) = 2h(N-1) + 1, N > 1$

Figure 6: Definizione ricorsiva dell'algoritmo proposto al problema della Torre di Hanoi

Ricordiamo che il problema di hanoi su 3 pioli si basa su 3 regole, ovvero:

1. Si può muovere un solo disco alla volta
2. Non si può spostare un disco più grande su uno più piccolo
3. Una mossa corrisponde allo spostamento di un disco

## 5.4 P vs NP

Ricapitoliamo gli elementi presi in considerazione e descriviamo di nuovi per rappresentare il seguente problema:

$$P = NP \text{ o } P \neq NP \quad (30)$$

1. **P**: Classe dei problemi a cui riusciamo ad assegnare un algoritmo risolutivo caratterizzato da un tempo polinomiale.
2. **NP**: Classe dei problemi di cui riusciamo solo a trovare un **certificato** in tempo polinomiale, ma non ancora un algoritmo risolutivo caratterizzato da un tempo polinomiale. NP sta per "Non Deterministic Polynomial", dato che è risolvibile in tempo polinomiale solo da un modello non deterministico, ovvero dato uno stesso input svariate volte, questo non garantisce che l'output sia sempre lo stesso.

### 5.4.1 SAT semplice

Determiniamo le caratteristiche del problema:

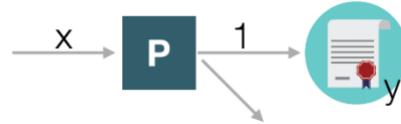
1. **Descrizione Problema**: Data una formula proposizionale (non quantificata, o almeno che non comprende il quantificatore universale) vogliamo stabilire se sia **soddisfacibile** o meno.
2. **Tempo e Spazio richiesti**: Considerando di aver già escluso il quantificatore universale, dobbiamo per ogni variabile, considerare che questa può essere 0 oppure 1. Dunque il tempo richiesto dalla soluzione che stiamo proponendo sarà  $O(2^n)$  su spazio  $O(n)$ . Non esistono ancora altre soluzioni che abbiano tempo inferiore.

$$\exists x, \forall y, \exists z, \forall w: (x \vee \bar{y} \vee z) \wedge (\bar{x} \vee w) \wedge y$$

In questo caso dunque, rifacendoci alla definizione di problema decisionale e di *predicato*<sub>1</sub>, cerchiamo un'istanza del problema che soddisfa una determinata proprietà, impostando un certificato.

### 5.4.2 Certificato

Enumeriamo gli elementi principali che caratterizzano i certificati:



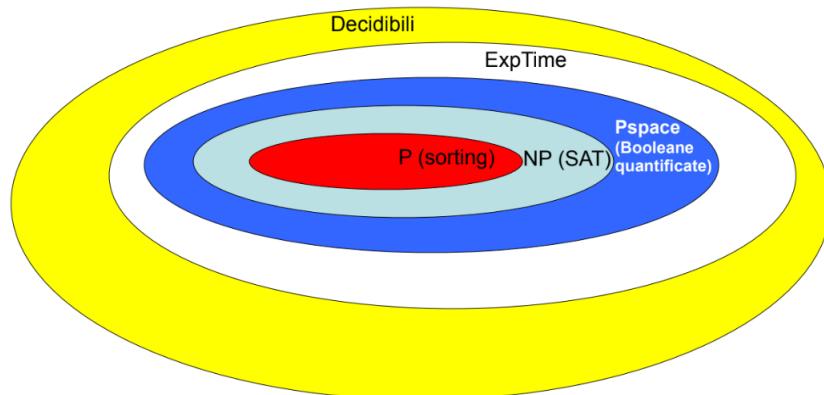
1. **P**: Proprietà che vogliamo soddisfare dipendente dal problema
2. **X**: Istanza del problema data in input alla proprietà *P*
3. **Y**: Certificato prodotto, ovvero un oggetto *Y* che può certificare che *X* soddisfa la proprietà *P*.

Contestualizzando ai problemi descritti prima, potremmo fare degli esempi di certificati:

1. Certificato per il SAT Semplice: Assegnare dei valori alle variabili che rendano la proposizione vera. **Certificato ottenuto dunque in tempo polinomiale, detto Certificato Semplice.**
2. Certificato per la connessione di un grafo: L'albero di copertura. **Certificato Semplice.**
3. Certificato per il SAT con quantificatore universale? In questo caso il "per ogni" non ci permette di assegnare dei valori alle variabili, dato che sarà necessario verificarle tutte. Dunque questo è un caso di **Certificato Difficile**.

Dunque i problemi di **Classe NP** sono caratterizzati da un certificato ottenibile in tempo **polinomiale** a cui ancora **non è stata trovata** una soluzione algoritmica in tempo **polinomiale**.

Dunque la nuova gerarchia sarà:



### 5.4.3 P vs NP - Congettura

Abbiamo dunque definito come  $P \subset NP$  ma non si sa per certo se quell'espressione sia propria o impropria. Non è dunque stato dimostrato che la classe  $NP$  non collassi sulla classe  $P$ , ma si congettura questo fenomeno. Basterebbe che solo determinati problemi  $NP$  appartengano a  $P$  per fare in modo che tutti gli altri siano anche essi in  $P$ , grazie alla **mappatura di riduzione**.

## 5.5 Mappatura di Riduzione

La riduzione è una mappatura che permette a determinati problemi appartenenti a  $NP$  di essere ridotti ad altri problemi di  $NP$ .

Questi problemi verso cui viene effettuata la riduzione sono detti **Problemi NP - Ardui**, definiamoli meglio.

### 5.5.1 NP - Arduo/Completo

**NP - Arduo** Un problema decisionale  $P_1$  è detto NP-ARDUO se ogni problema  $P \in NP$  è riducibile polinomialmente a  $P_1$ .

**NP - Completo** Un problema  $P$  è detto NP-COMPLETO se:

1.  $P \in NP$
2.  $P$  è NP-ARDUO

Detto questo, abbiamo gli strumenti per poter dire che se un problema  $P_1$  è NP-Completo, allora basterebbe dimostrare che  $P \in P_1$  per dimostrare che anche tutti gli altri appartengono a  $P$  effettuando la mappatura di riduzione in  $P_1$ .

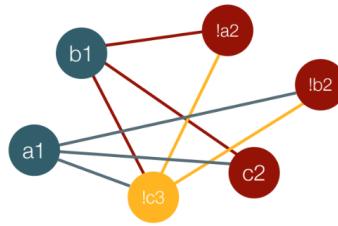
### 5.5.2 Esempio di Riduzione Clique - SAT

Ipotizziamo un problema  $P_1$  su grafi, ovvero: "Dato un grafo  $G$  e un intero  $K$ , esiste una clique di  $K$  nodi?". Per dimostrare l'appartenenza di questo problema  $P_1$  agli NP-Ardui e successivamente agli NP-Completi effettueremo una mappatura dal SAT-3, assumendo che sia NP-Completo, al problema attuale  $P_1$ . Illustriamo le fasi della dimostrazione:

**1. Inizializzazione mappatura:**

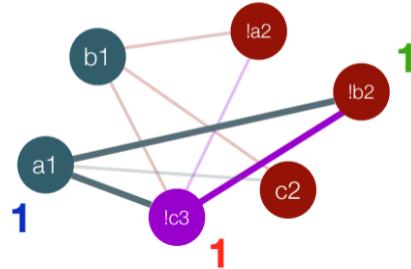
- (a) Ad ogni variabile della clausola corrisponde un vertice nel grafo
- (b) Due vertici sono adiacenti solo se:
  - i. Appartengono a clausole diverse
  - ii. Possono essere veri contemporaneamente

$$F = (a \vee b) \wedge (\neg a \vee \neg b \vee c) \wedge \neg c$$



**2. Verifica condizione:** Dopo aver costruito la mappa vogliamo mostrare che  $F$  è soddisfacibile se e solo se  $G$  contiene una clique di  $k$  nodi.

$$F = (1 \vee 0) \wedge (0 \vee 1 \vee 0) \wedge 1$$



Abbiamo dunque fatto in modo che istanze positive della clique trovate in tempo polinomiale venissero mappate ad istanze positive del SAT-3. Abbiamo dunque dimostrato che il problema appartiene agli NP-Ardui.