Algoritmi e strutture Dati

Marini Mattia

1^o semestre 2^o anno

Indice

1 Analisi complessità

Definizione 1: *Criterio di costo logaritmico*

La taglia dell'input è il *numero di bit* necessari per rappresentarlo

Definizione 2: *Criterio di costo uniforme*

La taglia dell'input è il *numero di elementi* di cui è costituito l'input

Definizione 3: *Notazione O, Ω, Θ*

- O: limita funzione *dall'alto*:

$$\exists c > 0, \exists m \in \mathbb{N} : \forall n \geq m, f(n) \leq c \cdot g(n)$$

- Ω: limita funzione *dal basso*:

$$\exists c > 0, \exists m \in \mathbb{N} : \forall n \geq m, f(n) \geq c \cdot g(n)$$

- Θ: limita funzione *dal entrambe le parti*:

$$\exists c_1, c_2 > 0, \exists m \in \mathbb{N} : \forall n \geq m, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Operativamente, per dimostrare che $f(x) = O(g(x))$ oppure $\Omega(g(x))$, quando f e g sono polinomi

- Limite superiore O: alzo gli esponenti minori del grado di f:

$$\begin{aligned} f(n) &= a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 \\ &\leq a_k n^k + |a_{k-1}| n^{k-1} + \ldots + |a_1| n + |a_0| \\ &\leq a_k n^k + |a_{k-1}| n^k + \ldots + |a_1| n^k + |a_0| n^k \quad \forall n \geq 1 \\ &= (a_k + |a_{k-1}| + \ldots + |a_1| + |a_0|) n^k \\ &\overset{?}{\leq} c n^k \end{aligned}$$

  che è vera per $c \geq (a_k + |a_{k-1}| + \ldots + |a_1| + |a_0|) > 0$ e per $m = 1$.

- Limite inferiore Ω: alzo gli esponenti minori del grado di f:

$$\begin{aligned} f(n) &= a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 \\ &\geq a_k n^k - |a_{k-1}| n^{k-1} - \ldots - |a_1| n - |a_0| \\ &\geq a_k n^k - |a_{k-1}| n^{k-1} - \ldots - |a_1| n^{k-1} - |a_0| n^{k-1} \quad \forall n \geq 1 \\ ? &\geq d n^k \end{aligned}$$

  che è vera se:

$$d \leq a_k - \frac{|a_{k-1}|}{n} - \frac{|a_{k-2}|}{n} - \ldots - \frac{|a_1|}{n} - \frac{|a_0|}{n} > 0 \Leftrightarrow n > \frac{|a_{k-1}| + \ldots + |a_0|}{a_k}$$

Nota come in questo caso alziamo gli esponenti solo a $k-1$ e non k in quanto, così facendo otteniamo che $d \leq h(x)$ dove $h(x)$ p monotona decrescente. Per questo motivo esisterà per forza un d che la limita dall'alto

Definizione 4: *Notazione o, ω*

- o: funzione limitata dall'alto:

$$\forall c \; \exists m : f(n) < cg(n), \forall n > m$$

- ω funzione limitata dal basso:

$$\forall c \; \exists m : f(n) > cg(n), \forall n > m$$

la differenza rispetto alle notazioni in definizione 1 sta nel $\forall c$

Utilizzando il concetto di limite, date due funzioni $f(n)$ e $g(n)$ si possono fare le seguenti affermazioni:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = o(g(n))$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \neq 0 \Rightarrow f(n) = \Theta(g(n))$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow f(n) = \omega(g(n))$$

Si noti che:

$$f(n) = o(g(n)) \Rightarrow f(n) = O(g(n))$$
$$f(n) = \omega(g(n)) \Rightarrow f(n) = \Omega(g(n))$$

1.1 Proprietà di O, Ω, Θ

1. Dualità
2. Eliminazione delle costanti
3. Somma
4. Prodotto
5. Simmetria
6. Transitività

Dimostrazioni:

1. Dualità: è sufficiente girare la formula data dalla definizione $O()$, ribattezzando la nuova costante $c' = \frac{1}{c}$

2. Eliminazione delle costanti: moltriplico per una nuova costante arbitraria a e ribattezzo $c' = ac$
3. Somma: ricordarsi che

$$f_1(n) + f_2(n) \leq \max(c_1, c_2)\, 2 \max(g_1(n), g_2(n))$$

4. Prodotto: come con somma, è immediato
5. Simmetria: è immediato dalla dualità
6. Transitività: dalla catena di $\leq$ ottengo che $f(n) \leq c_1 c_2 h(n)$

1.2 Dimostrazione complessità

Ci sono 3 metodi principali per l'analisi della complessità di un algoritmo:

1.2.1 Analisi a livelli, o metodo dell'albero

1. Srotola albero/espandi espressione e capisci la forma che ha.
2. Di solito si ottiene una sommatoria alla fine.
3. Risolvere sommatoria in forma chiusa secondo successioni geometriche. Le più comuni sono:

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

1.2.2 Analisi per tentativi (induzione)

- Bisogna tirare a indovinare la complessità dell'algoritmo e poi dimostrare per induzione che è vera per ogni n
- Se il tentativo è sbagliato ci sono due opzioni:
  - La costante c trovata dipende da n, quindi non è una costante
  - L'equazione finale è impossibile
- Il tentativo può essere giusto però possiamo comunque fallire nella dimostrazione. In questo caso ritentare con un *limite più stretto*. Ad esempio se abbiamo tentato con $O(n^2)$ possiamo tentare con $O(n^2 - bn)$
- Può infine darsi che *non si riesca a dimostrare il caso base* con $n = 1$. In tal caso si può dimostrare gli altri per $n = 2, \ldots, n = k$, assicurandosi che siano coperti tutti i casi per $n < k$

1.2.3 Metodo dell'esperto (master theorem)

Definizione 5: *Master theorem*

Siano a e b costanti intere tali che $a \geq 1$ e $b \geq 2$, e c, β costanti reali tali che $c > 0$ e $\beta \geq 0$. Sia $T(n)$ data dalla relazione di ricorrenza:

$$T(n) = \begin{cases} aT(n/b) + cn^\beta & n > 1 \\ d & n \leq 1 \end{cases}$$

Posto $\alpha = \frac{\log a}{\log b} = \log_b a$, allora:

$$T(n) = \begin{cases} \Theta(n^\alpha) & \alpha > \beta \\ \Theta(n^\beta \log n) & \alpha = \beta \\ \Theta(n^\beta) & \alpha < \beta \end{cases}$$

Per ricordarti, pensa al fattore $\frac{\log(a)}{\log(b)}$ come la velocità con cui "esplodono" le chiamate ricorsive.
Se questa è alta ($> \beta$) allora questa prevale e la complessità è $\Theta\left(n^\alpha\right)$.
In caso contrario prevale il fattore n^β quindi la complessità è $\Theta\left(n^\beta\right)$

La dimostrazione procede innanzitutto considerando la ricorrenza lineare generica

$$T(n) = aT(n/b) + cn^\beta$$

Supponiamo inoltre di avere n potenza di b, cioè $n = b^k$ per qualche $k \in \mathbb{N}$. In questo modo ci semplifichiamo la vita, in quanto "srotolando" la ricorrenza otteniamo un albero completo. Cerchiamo di calcolare il costo usando il medoto dell'albero descritto in sezione 1.2.1

Liv.	Dim.	Costo chiam.	N. chiamate	Costo livello
0	b^k	$cb^{k\beta}$	1	$cb^{k\beta}$
1	b^{k-1}	$cb^{(k-1)\beta}$	a	$acb^{(k-1)\beta}$
2	b^{k-2}	$cb^{(k-2)\beta}$	a^2	$a^2cb^{(k-2)\beta}$
...	...	...	...	...
i	b^{k-i}	$cb^{(k-i)\beta}$	a^i	$a^icb^{(k-i)\beta}$
...	...	...	...	...
$k-1$	b	cb^β	a^{k-1}	$a^{k-1}cb^\beta$
k	1	d	a^k	da^k

Dunque, possiamo arrivare alla conclusione che la somma di tutti i livelli, ossia la *complessitò complessiva* dell'algoritmo, è data da:

$$T\left(n\right) = a^k d + cb^{k\beta} \sum_{i=0}^{k-1} \left(\frac{a}{b^\beta}\right)^i$$

Prima di partire dobbiamo fare due importanti osservazioni :

$$a^k = n^\alpha \qquad\qquad a = b^\alpha$$

Caso 1 : $\alpha > \beta$

Ne segue che $q = b^{\alpha-\beta} > 1$:

$$\begin{aligned}
T(n) &= dn^\alpha + cb^{k\beta}\sum_{i=0}^{k-1} q^i \\
&= n^\alpha d + cb^{k\beta}\left[\frac{q^k - 1}{q-1}\right] && \text{Serie geometrica finita} \\
&\leq n^\alpha d + cb^{k\beta}\frac{q^k}{q-1} && \text{Disequazione} \\
&= n^\alpha d + \frac{cb^{k\beta}a^k}{b^{k\beta}}/(q-1) && \text{Sostituzione } q \\
&= n^\alpha d + ca^k/(q-1) && \text{Passi algebrici} \\
&= n^\alpha[d + c/(q-1)] && a^k = n^\alpha, \text{raccolta termini}
\end{aligned}$$

Dunque

$$T(n) = \Theta(n^\alpha)$$

Ho usato il fatto che $q > 1$ nel passaggio evidenziato in rosso. Pensa così:

$$\frac{q^k - 1}{q-1} \quad \text{sempre } > 0$$

mentre

$$\frac{q^k}{q-1}\begin{cases} > 0 \text{ se } q > 1 \\ < 0 \text{ se } q < 1 \end{cases}$$

Caso 2 : $\alpha = \beta$

Ne segue che: $q = b^{\alpha-\beta} = 1$

$$\begin{aligned}
T(n) &= dn^\alpha + cb^{k\beta}\sum_{i=0}^{k-1} q^i \\
&= n^\alpha d + cn^\beta k && q^i = 1^i = 1 \\
&= n^\alpha d + cn^\alpha k && \alpha = \beta \\
&= n^\alpha(d + ck) && \text{Raccolta termini} \\
&= n^\alpha[d + c\log n/\log b] && k = \log_b n
\end{aligned}$$

Dunque

$$T(n) = \Theta(n^\alpha \log n)$$

Caso 3 : $\alpha < \beta$ Ne segue che: $q = b^{\alpha-\beta} < 1$

$$
\begin{aligned}
T(n) &= dn^\alpha + cb^{k\beta}\sum_{i=0}^{k-1} q^i \\
&= n^\alpha d + cb^{k\beta}[(q^k - 1)/(q-1)] && \text{Serie geometrica finita} \\
&= n^\alpha d + cb^{k\beta}[(1 - q^k)/(1-q)] && \text{Inversione} \\
&\leq n^\alpha d + cb^{k\beta}[1/(1-q)] && \text{Disequazione} \\
&= n^\alpha d + cn^\beta/(1-q) && b^k = n
\end{aligned}
$$

Dunque

$$T(n) = \Theta(n^\beta)$$

1.2.4 Versione generalizzata

Definizione 6: *Master theorem generalizzato*

Sia $a \geq 1$, $b > 1$, $f(n)$ asintoticamente positiva, e sia

$$T(n) = \begin{cases} aT(n/b) + f(n) & n > 1 \\ d & n \leq 1 \end{cases}$$

Sia $\alpha = \frac{\log(a)}{\log(b)} \log_b a$. Sono dati tre casi:

Caso	complessità
$\exists \epsilon > 0 : f(n) = O(n^{\alpha-\epsilon})$	$T(n) = \Theta(n^\alpha)$
$f(n) = \Theta(n^\alpha)$	$T(n) = \Theta(f(n)\log n)$
$\exists \epsilon > 0 : f(n) = \Omega(n^{\alpha+\epsilon}) \wedge$	
$\exists c : 0 < c < 1, \exists m \geq 0 :$	$T(n) = \Theta(f(n))$
$af(n/b) \leq cf(n), \forall n \geq m$	

1.3 Ricorrenze lineari

Definizione 7: *Ricorrenze lineari*

Siano $a_1, a_2, \ldots, a_h$ costanti intere non negative, con h costante positiva, c e β costanti reali tali che $c > 0$ e $\beta \geq 0$, e sia $T(n)$ definita dalla relazione di ricorrenza:

$$T(n) = \begin{cases} \sum_{1 \leq i \leq h} a_i T(n-i) + cn^\beta & n > m \\ \Theta(1) & n \leq m \geq h \end{cases}$$

Posto $a = \sum_{1 \leq i \leq h} a_i$, allora:

- $T(n)$ è $\Theta(n^{\beta+1})$, se $a = 1$.
- $T(n)$ è $\Theta(a^n n^\beta)$, se $a \geq 2$.

2 Analisi ammortizzata

L'analisi ammortizzata è utile nel caso in cui la complessità di un algoritmo dipende dallo "stato" di una struttura dati associata. Ci sono 3 metodi:

- Metodo dell'aggregazione : si calcola il costo totale di n operazioni e si divide per n.
- Metodo degli accantonamenti : alle operazioni vengono assegnati i *costi ammortizzati*, che possono essere maggiori/minori del loro costo effettivo
- Metodo del potenziale Lo stato del sistema viene rappresentato da una *funzione di potenziale*

2.0.1 Contatore binario (metodo dell'aggregazione)

Immaginiamo di avere un contatore binario con l'algoritmo `increment` che . Questo algoritmo avrà complessità che dipende molto dal numero di 1 che deve scorrere

Algoritmo: *Increment*

```
void increment(bool[] A, int k):
    i ← 0;
    while i < k and A[i] = 1 do
        A[i] ← 0;
        i ← i + 1;
    if i < k then
        A[i] ← 1;
```

Nota come il bit i -esimo viene modificato ogni 2^i incrementi. Per questo il costo di n operazioni consecutive è di:

$$\text{Costo totale } : \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \leq n \sum_{i=0}^{k-1} \frac{1}{2^i} \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

Dunque il costo ammortizzato secondo il metodo dell'aggregazione è di:

$$\text{Costo ammortizzato } : \frac{T(n)}{n} \leq \frac{2n}{n} = O(1)$$

2.0.2 Contatore binario (metodo degli accantonamenti)

Possiamo stimare il costo necessario per una singola iterazione dell'algoritmo. Sapredo dunque che il costo ammortizzato può essere maggiore o minore del costo effettivo. Su n iterazioni tuttavia, sappiamo che il coto effettivo di queste sarà sempre minore

Nel caso del contatore binario possiamo assegnare un costo ammortizzato di 2 per ogni operazione. In questo modo, ogni bit impostato a 1 ha "immagazzinato" un'operazione bonus che può essere usata per riportare il bit a 0, nel momento in cui devo "scorrere i bit" a sinistra. Quindi ho che:

$$\sum_{i=0}^{n} c_i \leq \sum_{i=0}^{n} a_i = 2n$$

2.0.3 Contatore binario (metodo del potenziale)

In questo caso definiamo una funzione $\Phi(S)$ che in modo simile al metodo degli accantonamenti 2.0.2 tiene traccia di quanto lavoro possiamo fare prima di eccedere la complessità stimata.

$$a_i = c_i + \Phi(S_i) - \Phi(S_{i-1})$$

siccome

$$A = \sum_{i=1}^{n} a_i = C + \Phi(S_n) - \Phi(S_0)$$

so che se $\Phi(S_n) - \Phi(S_0) \geq 0 \rightarrow$ il costo ammortizzato è un limite superiore del costo reale

Nel caso del contatore binario:

- $\Phi(S)$ è il numero di bit settati a 1
- Chiamando t il numero di bit impostati a 1 prima del primo 0, abbiamo che

$$a_i = \underbrace{t+1}_{c_i} + \underbrace{1-t}_{\Delta\Phi} = 2$$

- Siccome $\Phi(S_n) - \Phi(S_0) \geq 0$ (ci sono più 1 nello stato finale che in quello iniziale), allora vale che

$$T(n) = O(2n) = O(n)$$

2.1 Vettori dinamici

L'analisi ammortizzata è fondamentale per la comprensione dei vettori dinamici.

2.1.1 Ingrandimento

Ho due strategie per implemetarli:

- Strategia del raddoppiamento : quando il vettore è pieno, raddoppio la sua dimensione (o moltiplichi per fattore specificato). Risolvendo con formula 3

$$\text{Costo } n \text{ inserimenti } = T(n) = n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j = O(n)$$

  la complessità ammortizzata è quindi:

$$\frac{O(n)}{n} = O(1)$$

- Strategia dell'ingrandimento : quando il vettore è pieno, aggiungo un numero fisso di elementi. Dato d il fattore di ingrandimento e risolvendo la sommatoria con *formula di Gauss*

$$\text{Costo } n \text{ inserimenti } = T(n) = n + \sum_{j=0}^{\lfloor n/d \rfloor} d \cdot j = O\left(n^2\right)$$

  la complessità ammortizzata è quindi:

$$\frac{O(n^2)}{n} = O(1)$$

2.1.2 Cancellazione

Definiamo con

$$\alpha = \frac{\text{dimensione}}{\text{capacità}}$$

il fattore di carico.

- Impostiamolo a $\frac{1}{4}$
- (1/2 non va bene in quanto se continuiamo a inserire/rimuovere a size/2 dobbiamo continuare a riallocare il vettore)

Impostiamo una funzione di potenziale tale per cui:

- Vale 0 quando $\alpha = \frac{1}{2}$
- Vale `dim` quando $\alpha = \frac{1}{4}$ o $\alpha = 1$. Questo significa che posso ripagare allocazione

$$\Phi = \begin{cases} 2 \cdot \text{ dim } - \text{ capacità} & \alpha \geq \frac{1}{2} \\ \frac{\text{capacità}}{2} - \text{ dim} & \alpha \leq \frac{1}{2} \end{cases}$$

- $\alpha = \frac{1}{2}$ (dopo espansione/contrazione) $\Rightarrow \Phi = 0$
- $\alpha = 1$ (prima di espansione) $\Rightarrow$ dim $=$ capacità $\Rightarrow \Phi =$ dim
- $\alpha = \frac{1}{4}$ (prima di contrazione) $\Rightarrow$ capacità $= 4 \cdot$ dim $\Rightarrow \Phi =$ dim

3 Alberi

3.1 Alberi di ricerca binaria

Ho le seguenti operazioni:

3.1.1 Stampa

Itero ricorsivamente su ogni nodo, partendo dalla radice, e stampo il valore del nodo. Ho 3 opzioni:

- Inorder : stampo il sottoalbero sinistro, il nodo corrente, e il sottoalbero destro. In questo modo ottengo i nodi in ordine crescente (l - p - r)
- Preorder : stampo il nodo corrente, il sottoalbero sinistro, e il sottoalbero destro. In questo modo ottengo i nodi in ordine di visita (p - l - r)
- Postorder : stampo il sottoalbero sinistro, il sottoalbero destro, e il nodo corrente. In questo modo ottengo i nodi in ordine di visita, ma dopo aver visitato i figli (l - r - p)

3.1.2 Ricerca

- Se `curr.val == n.val` ho trovato il nodo
- Se `curr.val == nil` NON ho trovato il nodo
- Se `n.val < curr.val` chiamo ricorsivamente a sinistra
- Se `n.val > curr.val` chiamo ricorsivamente a destra

3.1.3 Inserimento

Prendo un nodo `new_node` e:

- Se `new_node.val < v.val` chiamo ricorsivamente a sinistra
- Se `new_node.val > v.val` chiamo ricorsivamente a destra
- Altrimenti, se `v.val == nil`, sono in una foglia e inserisco nodo

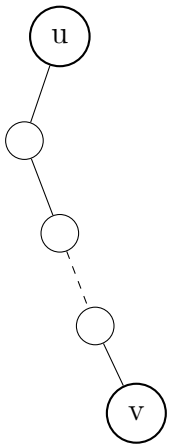
3.1.4 Successore-predecessore

Voglio trovare il nodo che viene subito prima/dopo al nodo `n` nell'ordinamento. Innanzitutto trovo il nodo `n` con la tecnica specificata in 3.1.2. Supponiamo di cercare il *successore*. Distinguo i due casi

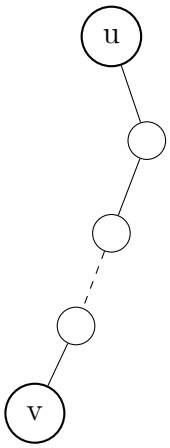
- Il figlio destro di `n` è diverso da `nil`: il successore è il nodo con il valore minimo del sottoalbero destro di `n`
- Il figlio destro di `n` è `nil`:
  - Risalgo per i parents finche non trovo un nodo che è un *figlio sinistro*

– Il parent di questo nodo è il *successore*

Il ragionamento è del tutto analogo per il predecessore, ma devo considerare gli alberi a sinistra. Dunque è del tutto speculare



u = successor (v)



u = predecessor (v)

3.1.5 Rimozione

Trovo nodo da rimuovere e distinguo in 2 casi:

- Se il nodo ha 0 figli, lo rimuovo semplicemente
- Se il nodo ha solo un figlio, posso attaccarlo al parent del nodo rimosso (*shortcut*)
- Se il nodo ha 2 figli, allora:
  - Noto che il successore è per forza nel sottoalbero destro. Se il sottoalbero destro esiste allora non serve ricercare il successore nei parents
  - Trovo il successore `s`. Nota che il successore non può avere un figlio sinistro perché deve essere il minimo del sottoalbero destro
  - Se il successore ha un figlio destro, lo attacco al parent del successore (*shortcut*)
  - Sostituisco il valore del nodo da rimuovere con il valore del successore

3.2 Alberi red-black

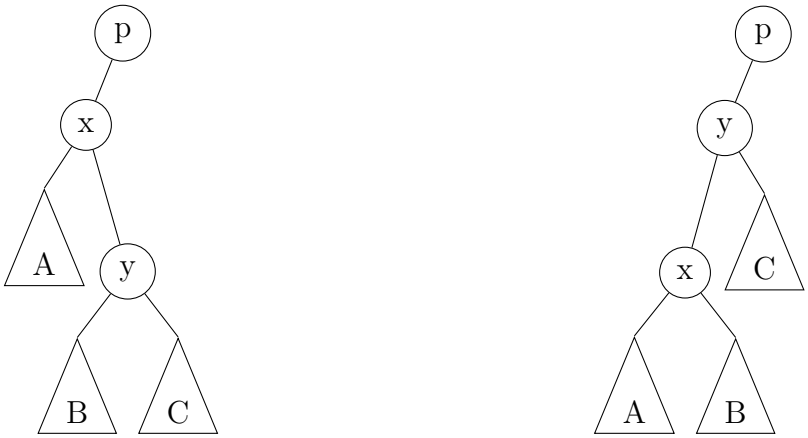
Gli alberi red black sono alberi binari il cui scopo è *mantenere il bilanciamento dell'albero* dopo ogni inserimento. In realtà, non è garantito che l'albero sia sempre completamente bilanciato, ma la differenza di altezza può essere al più $\log(n)$. In un albero red-black ogni nodo è di colore rosso o nero.

In un RBT vigono le seguenti proprietà:

- La radice è nera
- Ogni foglia è nera ed è un "nodo nil"
- Se un nodo è rosso, allora entrambi i suoi figli sono neri
- Ogni cammino da un nodo a una foglia ha lo stesso numero di nodi neri (altezza nera)

3.2.1 Inserimento

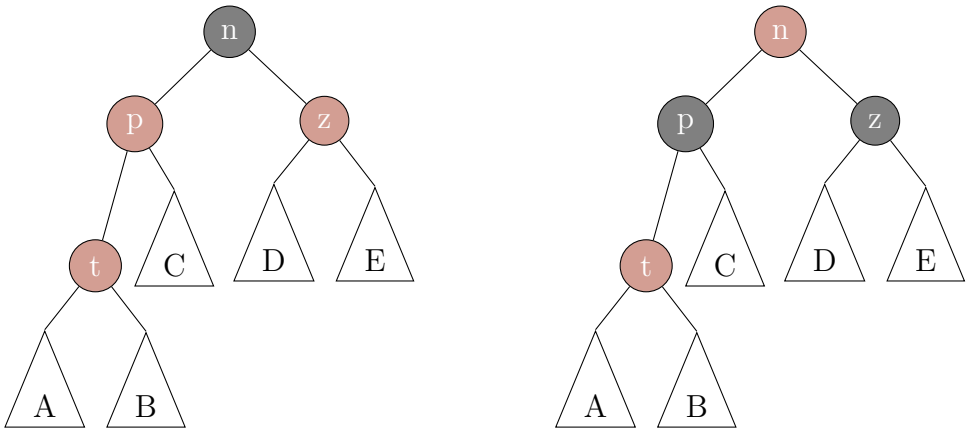
Per capire l'inserimento, bisogna definire le operazioni di rotazione:



Nota come dopo la rotazione, il sorting dei nodi non cambia ([A, x, B, y, C, p])

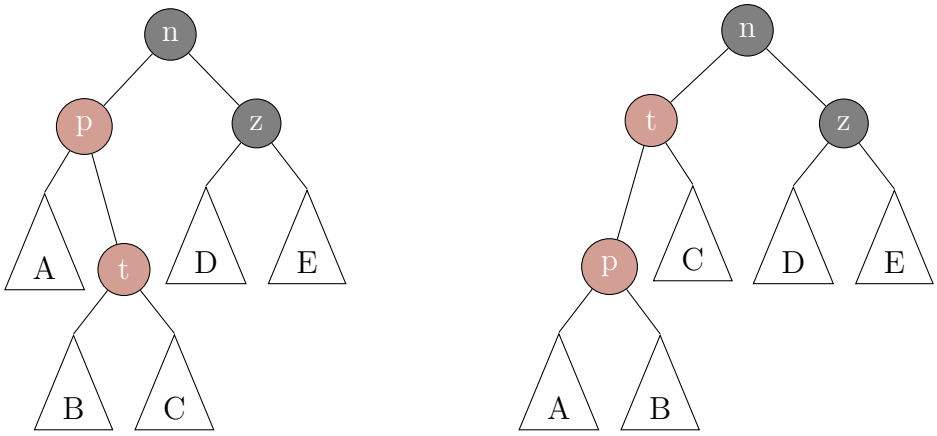
Nell'inserimento, si imposterà il colore del nuovo nodo a rosso (non posso violare vincolo altezza nera), e fixo ricorsivamente le violazioni di vincoli che si possono creare. In particolare ci sono 7 casi:

- Caso1 : nodo non ha padre: zero problemi, cambio colore in nero
- Caso2 : nodo ha padre nero: zero problemi, nessun vincolo è violato
- Caso3 : t rosso, p rosso, z rosso. Semplicemente cambio colore di p e z



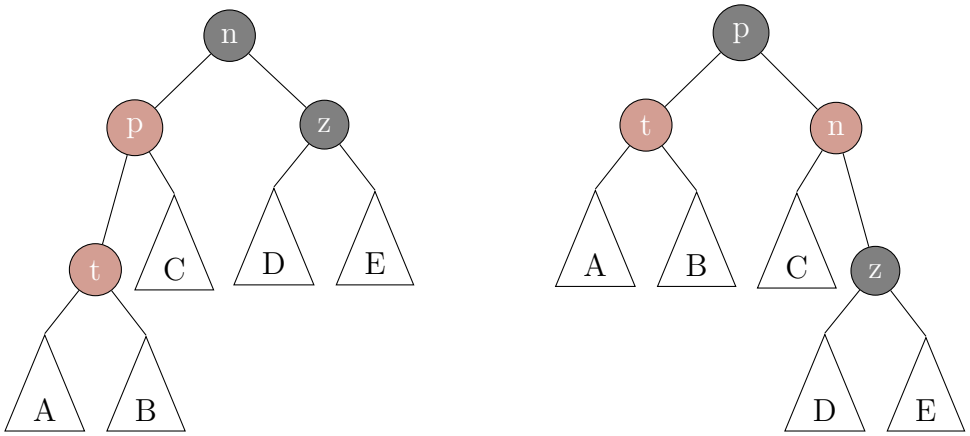
  $t = n$

- Caso4/5: (simmetrici): t rosso, p rosso, z nero. Rotazione a sinistra attorno a p



$t = p$

- Caso6/7: (simmetrici): t rosso, p rosso, z nero. Nodi rossi a sinistra. Rotazione a sinistra attorno a n. Si arriva a questo caso dal precedente



$t =$ nil . Ho finito, questa situazione non viola nessun vincolo

3.2.2 Eliminazione

Per eliminare un nodo, posso seguire la stessa logica descritta in sezione 3.1.5. Nota inoltre che se il nodo da eliminare è rosso homeno problemi. Posso avere problemi solo nel caso in cui

- Il nodo ha 2 figli
- Il successore è nero

3.2.3 Teoremi vari alberi rb

Teorema 1: *Numbero nodi interni albero RB*

In un albero RB, un sottoalbero con radice x ha almeno

$$n \geq 2^{bh(x)} - 1$$

nodi interni (nodi non foglie `nil`)

Dimostrazione:

- Caso base: u è una foglia `nil` $\rightarrow 0$ nodi interni

$$n > 2^{bh(u)} - 1 = 2^0 - 1 = 0$$

- Passo induttivo: $h > 1$

$$n \geq 2 \cdot \left(2^{bh(u)-1} - 1\right) + 1 = 2^{bh(u)} - 1$$

  Il $+1$ finale deriva dal nodo corrente, nel caso in cui sia nero. Nel caso in cui sia rosso ho un'ipotesi ancora meno stringente

Teorema 2: *Quantità nodi neri albero RB*

In un albero RB, almeno la metà dei nodi in un cammino dalla radice ad una figlia sono neri

Dimostrazione: al massimo posso avere un nodo rosso e uno nero alternati, altrimenti violerei i vincoli. Quindi il numero dei nodi neri e sempre $\leq bh\,(u)$

Teorema 3: *Differenza di altezza in albero RB*

In un albero RB, dati due cammini dalla radice a due foglie, non è possibile che uno sia più lungo del doppio dell'altro.

Dimostrazione: nel caso limite avro un cammino tutto nero lungo x. Per teorema 3.2.3 non posso avere più nodi rossi che neri. Quindi al massimo avro altrettanti x nodi rossi in un altro cammino

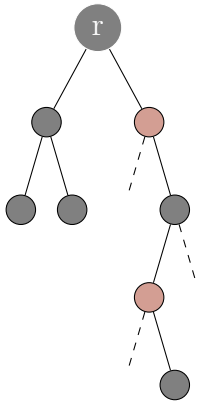
Teorema 4: *Altezza massima alberi RB*

L'altezza di un albero RB con n nodi *interni* è al più $2\log\,(n+1)$

Dimosttazione:

- Versione informale:
  - Devo stimare il numero di nodi minimo data un'altezza h.
  - L'albero in cui ho meno nodi possibili, a parità di altezza è l'albero in cui ho:

* Altezza massima: $h = 2bh$, nodi rossi e neri alternati
* Il resto dell'albero è costituito da soli nodi neri (minimizzando il numero di nodi)



– Il caso in cui ho meno nodi possibile è il caso in cui ho un albero di soli nodi neri, ad eccezione del percorso con i nodi alternati. Come ogni albero, questo ha:

$$n > 2^{bh(r)} - 1$$

nodi. Nota che è $2^{bh(r)}$ e non $2^{bh(r)-1}$ in quanto non considero i nodi foglia che sono `nil`

– Sostituend bh e girando la formula ottengo

$$n \geq 2^{bh(r)} - 1 = 2^{\frac{h}{2}} - 1$$

ossia

$$h \leq 2 \log (n + 1)$$

∘ Versione formale: mi baso su questi due teoremini: 1, 2

$$\begin{aligned} n \geq 2^{bh(r)} - 1 &\Leftrightarrow n \geq 2^{h/2} - 1 \\ &\Leftrightarrow n + 1 \geq 2^{h/2} \\ &\Leftrightarrow \log(n + 1) \geq h/2 \\ &\Leftrightarrow h \leq 2 \log(n + 1) \end{aligned}$$

4 Grafi

4.0.1 Kosaraju

L'algoritmo serve a trovare le componenti fortemente connesse di un grafo diretto. Gli step sono i seguenti:

∘ Calcola sort topologico del grafo

∘ Calcola grafo trasposto

∘ Lancia dfs in ordine inverso di finish time e assegna un id diverso ad ogni chiamata della bfs

In particolare, questo funziona perché:

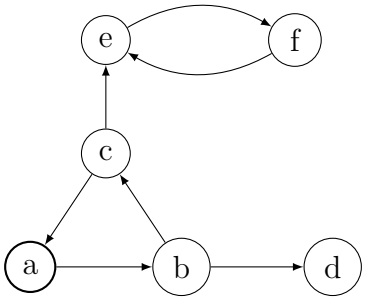
- Il grafo di condensazione[1] è un DAG (). Se non lo fosse per assurdo si creerebbe un scc più grande creata dal ciclo fra le scc
- Avendo due scc C e C' collegare da un edge $u- > v$ e chiamando s e s' il primo nodo visitato appena entrari in a visitare la scc, allora

  $$\text{finish\_time}(s) > \text{finish\_time}(s')$$

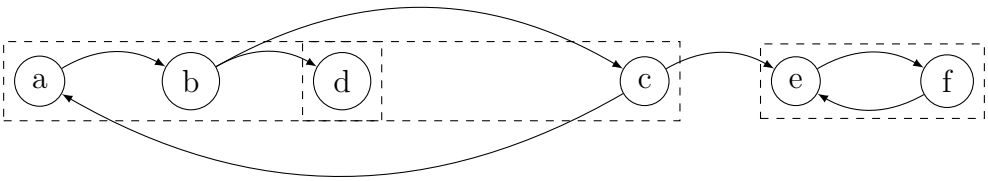
  ossia s viene prima di s' nel sort topologico
- Quindi facendo il grafo trasposto, le scc rimangono connesse
- Scorrendo i nodi in ordine di toposort, il primo nodo non visitato su cui verra chiamata la bfs è sempre il primo nodo da cui si è enttrati a visitare la scc

L'idea è che se faccio il sorting delle scc in base al primo nodo visitato di ognuna nella toposort del grafo originale, ottengo il toposort corretto del grafo di consensazione

Non è detto che le componenti del condensation graph siano adiacenti nel toposort.



Ad esempio potrei visitare come $a - b - c - a - c - e - f - e - c - b - d - b - a$, il che corrisponderebbe ad un toposort di $f - e - c - d - b - a$



5 Hashing

5.1 Definizioni

[1] Grafo in cui ogni SCC's è condensata in un nodo

Definizione 8: *Funzione hash*

Dati:

- $\mathcal{U}$: universo delle chiavi
- m: dimensione della struttura dati che immagazzina i valori (*hash-table*)
- n: numero di chiavi inserite

Una funzione hash è una funzione dall'universo delle chiavi $\mathcal{U}$ ad un valore intero.

$$h : \mathcal{U} \to \{0, 1, \ldots, m-1\}$$

Definizione 9: *Funzione hash perfetta*

Una funzione hash è detta *perfetta* se non genera collisioni:

$$\forall k_1, k_2 \in \mathcal{U} : k_1 \neq k_2 \Rightarrow h(k_1) \neq h(k_2)$$

Definizione 10: *Uniformità semplice*

Una funzione di hash gode di uniformità semplice se applicandola su un valore estratto a caso la probabilità che finisca nell'i-esimo slot è di $\frac{1}{m}$. In altri termini, ogni slot ha la stessa probabilità di essere selezionato dalla funzione di hash.

Formalmente

- Sia $P(k)$ la probabilità che una chiave k sia inserita in tabella
- Sia $Q(i)$ la probabilità che una chiave finisca nella cella i

$$Q(i) = \sum_{k \in \mathcal{U} : h(k) = i} P(k)$$

Una funzione hash h gode di uniformità semplice se:

$$\forall i \in [0, \ldots, m-1] : Q(i) = 1/m$$

5.2 Funzioni hash stringhe

Definiamo innanzitutto le seguenti operazioni:

- ord(c) : valore ordinale binario del carattere c in qualche codifica
- bin (k) : rappresentazione binaria della chiave k, concatenando i valori binari dei caratteri che lo compongono
- int(b): valore numerico associato al numero binario b
- int(k) = int(bin(k))

5.2.1 Estrazione

Seleziono solo certi bit in una stringa, ad esempio ultimi. Facile collisione, vedi `Coperto`, `Roberto`

5.2.2 Xor

Divido la rappresentazione binaria della parola in gruppi di x bit, e ne eseguo lo `xor` bit a bit (Il quale corrisponde alla somma)

5.2.3 Metodo della divisione

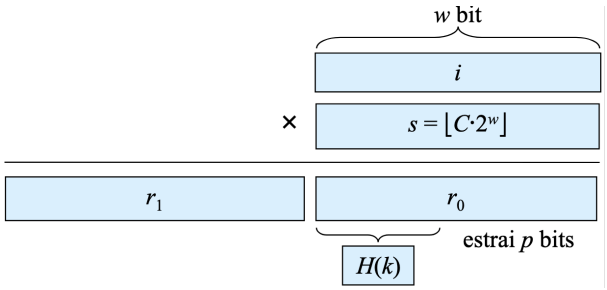
- m dispari, meglio se numero primo
- $H(k) = \text{int}(k) \mod m$

5.2.4 Metodo della moltiplicazione

- m qualsiasi, meglio se potenza di 2
- C costante reale, $0 < C < 1$
- Sia $i = \text{int}(k)$
- $H(k) = \lfloor m \cdot (C \cdot i - \lfloor C \cdot i \rfloor) \rfloor$

Nota come di base questo coincida con prendere la parte decimale di $C \cdot i$, ossia un numero "abbastanza casuale" compreso tra 0 e 1.

Implementativamente, si può calcolare $H(k)$ in modo efficiente:



Questo coincide con

- Considerare la costante C senza la virgola ($s = C \cdot 2^w$, ossia shifto a sinistra di w bit)
- Eseguire la moltiplicazione
- I w bit più significativi contengono il risultato intero della moltiplicazione
- I w bit meno significativi contengono il risultato dopo la virgola
- Questo è vero perché dovrei shiftare nuovamente a destra il risultato, se avessi moltiplicato usando la virgola

5.3 Liste di trabocco

n	Numero di chiavi memorizzati in tabella hash
m	Capacità della tabella hash
$\alpha = n/m$	Fattore di carico
$I(\alpha)$	Numero medio di accessi alla tabella per la ricerca di una chiave non presente nella tabella (ricerca con insuccesso)
$S(\alpha)$	Numero medio di accessi alla tabella per la ricerca di una chiave presente nella tabella (ricerca con successo)

In una hash map con liste di trabocco, il *valore atteso* della lunghezza delle liste di trabocco è pari al fattore di carico $\alpha = n/m$. In media quindi il costo è di

- Ricerca senza successo: $\theta(1) + \alpha$
- Ricerca con successo: $\theta(1) + \frac{\alpha}{2}$

5.4 Indirizzamento aperto

Con l'indirizzamento aperto, ogni valore è salvato nello stesso vettore. Se lo slot scelto è già in uso, si tenta il prossimo

Definizione 11: *Sequenza di ispezione*

Una sequenza secondo la quale si esaminano gli slot

Definizione 12: *Hashing uniforme*

La situazione ideale prende il nome di hashing uniforme, in cui ogni chiave ha la stessa probabilità di avere come sequenza di ispezione una qualsiasi delle $m!$ permutazioni di $[0, \dots, m-1]$

Abbiamo diverse possibilità nella scelta delle sequenze di ispezione:

- Ispezione lineare: $H(k,i) = (H_1(k) + h \cdot i) \mod m$
  - Problema di agglomerazione primaria: la cella dopo una sequenza di lunghezza i ha probabilità $\frac{(i+1)}{m}$ di essere estratta
- Ispezione quadratica: $(H(k,i) = H_1(k) + h \cdot i^2) \mod m$
- Doppio hash: $H(k,i) = (H_1(k) + i \cdot H_2(k)) \mod m$

Nota che se utilizzi indirizzamento libero bisogna distinguere fra inserimento e lookup. In particolare distinguiamo fra valori `nil`, ossia mai inseriti e `deleted`, ossia rimossi

- Inserimento: scorriamo sequenza di ispezione ed inseriamo appena troviamo un valore `nil` o `deleted`

- Lookup: scorriamo finche incontriamo un valore `nil` o il valore cercato. Sui valori `deleted` andiamo avanti siccome potrebbe essere stato inserito più tardi nella sequenza di ispezione

6 Divide et impera

6.1 Torri di hanoi

Un algoritmo classico divide et impera è quello delle torri di hanoi. Si hanno 3 torri e si vuole spostare n dischi dalla prima all'ultima. Ogni disco non può essere impilato su un disco più piccolo

Chiamiamo le torri `A, B, C`, a partire da SX. L'algoritmo procede così:

- Sposto $n-1$ dischi da `A` a `B`
- Sposto il disco rimanente da `A` a `C`
- Sposto $n-1$ dischi rimasti da `B` a `C`

Ecco l'algoritmo in pseudo codice:

Algoritmo 1: *Torri di hanoi*

```
void hanoi(n, src, dest, middle):
    if n = 1 then
        print src → dest;
    else
        hanoi(n − 1, src, middle, dest);
        print src → dest;
        hanoi(n − 1, middle, dest, src);
```

6.2 Algoritmo di Strassen

L'idea è che possiamo esprimere il prodotto matriciale come segue, dividendo la matrice in 4:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \quad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

$$C = \begin{bmatrix} A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1} & A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2} \\ A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1} & A_{2,1} \times B_{1,2} + A_{2,2} \times B_{2,2} \end{bmatrix}$$

Così facendo ottengo la ricorrenza:

$$T(n) = 8T\left(\frac{n}{2}\right) + n^3$$

In realtà posso eseguire i prodotti in modo intelligente, risparmiandone uno, ottenendo quindi

$$T(n) = 7T\left(\frac{n}{2}\right) + n^2 \approx n^{2.81}$$

7 Strutture dati speciali

7.1 Heap

Abbiamo bisogno di 2 funzioni:

- `heapbuild`: costruisce un heap
- `heaprestore`: dato un heap con una possibile violazione nella radice, ripristina le proprietà di min/max heap

7.1.1 Heap restore

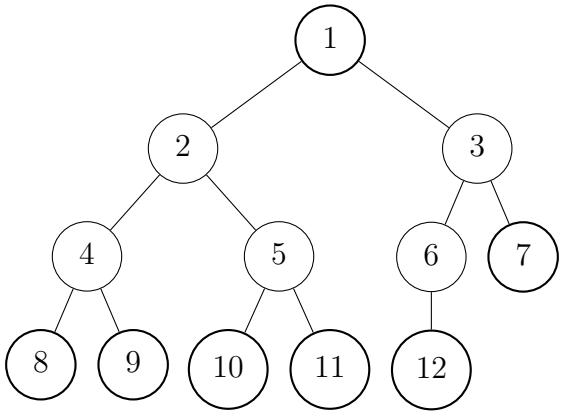
Posso avere una violazione nella radice di un heap. Controllo il nodo root `r`

- Se *r.val* è maggiore di entrambi i figli non violo nulla
- Se `r.val` non è maggiore di entrambi i figli allora lo scambio con il figlio maggiore. Chiamo ricorsivamente sul sottoalbero del figlio maggiore (l'altro sottoalbero rimane inalterato)

7.1.2 Heap build

Innanzitutto nota che

- Sia $A[1 \dots n]$ un vettore da ordinare
- Tutti i nodi $A[\lfloor n/2 \rfloor + 1 \dots n]$ sono foglie dell'albero e quindi heap contenenti 1 elemento



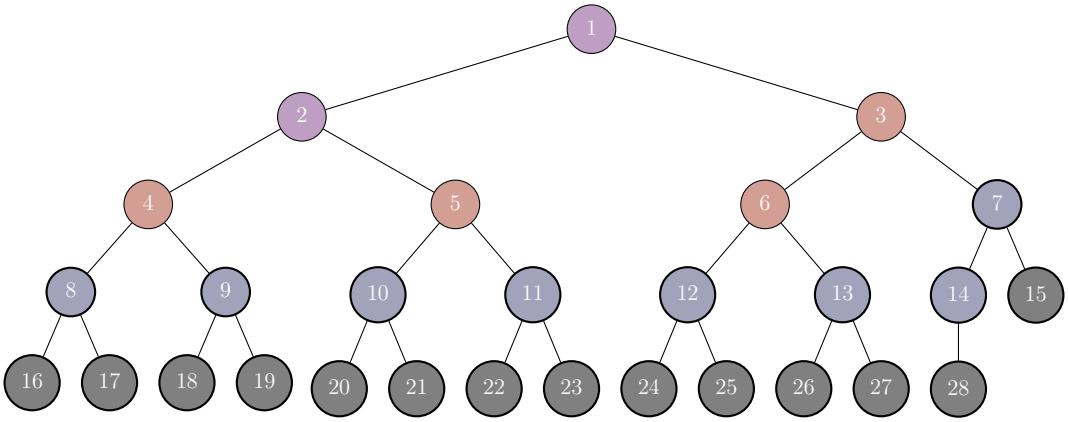
Il fatto che gli ultimi $\frac{n}{2}$ elementi dell'array siano foglie è dato dal fatto che dato un nodo di indice i allora $\frac{i}{2}$ è suo padre Dunque, la procedura `heapBuild()`

- attraversa i restanti nodi dell'albero, a partire da $\lfloor n/2 \rfloor$ fino ad 1
- esegue `maxHeapRestore()` su ognuno di essi

Algoritmo 2: *Headp build*

```
heapBuild(Item [] A, int n):
    for i = ⌊n/2⌋ downto 1 do
        maxHeapRestore(A, i, n);
```

Verrebbe naturale affermare che la complessità è $\Theta(n \log(n))$, ma è in realtà $\Theta(n)$. Ora l'idea è di dividere per livelli il calcolo della complessità. In particolare dividiamo i nodi in base alla loro altezza



Chiamo h l'altezza dell'albero. Nota che per il livello i, a partire dall'ultimo (quello più in basso), abbiamo $\frac{n}{2}$ nodi e per ognuno di essi dobbiamo eseguire al più 1 operazione. Più in generale:

Livello (dal basso)	Numero operazioni	Numero di nodi
1	1	$n/2$
2	2	$n/4$
$\vdots$	$\vdots$	$\vdots$
i	i	$n/(2^i)$

Quindi il costo totale è dato da:

$$T(n) = \sum_{i=1}^{\lfloor \log(n) \rfloor} \frac{n}{2^i} \cdot i = n \sum_{i=1}^{\lfloor \log(n) \rfloor} \left(\frac{1}{2}\right)^i i \leq n \sum_{i=1}^{\infty} \left(\frac{1}{2}\right)^i i$$

Questa è una successione geometrica:

$$\sum_{k=1}^{\infty} k a^k = \frac{a}{(1-a)^2}$$

dunque

$$T(n) \leq n \sum_{i=1}^{\infty} \left(\frac{1}{2}\right)^i i = n \cdot \frac{1/2}{(1/2)^2} = 2n = O(n)$$

7.1.3 Heap insert

Semplicemente prende un elemento e lo mette nell'ultima posizione dell'array, quindi l'ultimo figlio a destra nell'ultimo livello. Ricorsivamente chiama heap restore sul parent

7.1.4 Heap remove min

Swappa l'elemento nella root e l'ultimo elemento a destra nell'ultimo livello. Chiama `heap restore` sulla root

7.2 Priority queue

Una priority queue usa un heap come descritto in sezione 7.1 per fornire accesso in tempo costante all'elemento maggiore/minore.

7.2.1 Decremento/incremento priorità

- Se è una minqueue e la prirorità aumenta, oppure ho una maxqueue e la priorità diminuisce è sufficiente chiamare `heap_restore` sul nodo coinvolto
- Se è una minqueue e la priorità diminuisce, oppure ho una maxqueue e la priorità aumenta, devo chiamare ricorsivamente `heaprestore` su padre, come nell'inserimento descritto in sezione 7.1.3

Tipo coda	incremento	decremento
minqueue	heaprestore	recurse up
maxqueue	recurse up	heaprestore

7.3 Merge find set

7.3.1 Realizzazione con liste

- Il primo elemento della lista contiene il rappresentante della componente
- Ogni elemento della lista contiene un puntatore al rappresentante
- Per unire le liste

Dunque `find()` ha complessità $O(1)$ e `merge()` ha complessità $O(n)$. Tuttavia su n operazioni, `merge()` ha complessità ammortizzata di $O(n)$

7.3.2 Realizzazione con alberi

- Il rappresentante è la radice. La radice ha un self loop
- Per trovare rappresentante risalgo fino a radice
- Per fare merge attacco radice di un albero a radice dell'altro

Dunque `merge()` ha complessità $O(1)$ e `find()` ha complessità $O(n)$. Si possono usare euristiche per ridurre la complessità di find a $O(\log n)$

7.3.3 Euristica sul peso (liste)

Attacco sempre lista più corta a lista più lunga. Complessità di `merge` diventa $O(\log(n))$

7.3.4 Euristica sul rango (alberi)

Attacco sempre albero con altezza massima minore a quello con altezza massima maggiore. Dunque:

- Se `rango(t1) == rango(t2)` allora rango aumenta di 1
- Il rango rimane uguale a `max(rango(t1), rango(t2))` se `rango(t1) != rango(t2)`

Quindi visto che posso effettuare al massimo $\log(n)$ operazioni di merge, il rango massimo è $\log(n)$. Formalmente procedo per induzione e dimostro che $n > 2^{\text{rank}}$, dove `rank` è l'altezza massima dell'albero:

- Caso base : nel nodo singolo rank $= 0 \to 1 \geq 2^0$
- Induzione caso 1: $\text{rank}[x] > \text{rank}[y]$
  - Il rango finale è pari a $\text{rank}[x]$
  - Per induzione, il numero di nodi è:
$$n \geq 2^{\text{rank}[x]} + 2^{\text{rank}[y]} \geq 2^{\text{rank}[x]}$$
- Induzione caso 2: $\text{rank}[x] = \text{rank}[y]$
  - Il rango finale è pari a $\text{rank}[x] + 1$
  - Per induzione, il numero di nodi è:
$$n \geq 2^{\text{rank}[x]} + 2^{\text{rank}[y]} \geq \text{rank}[x] = 2 \cdot 2^{\text{rank}[x]} \geq 2^{\text{rank}[x]+1}$$
    come volevasi dimostrare

7.3.5 Euristica di compressione dei cammini

L'idea è di appiattire l'albero attaccando i nodi direttamente alla radice, facendo si che `find()` abbia complessità costante

Algoritmo	find()	merge()
Liste	$O(1)$	$O(n)$
Alberi	$O(n)$	$O(1)^+$
Liste + euristica sul peso	$O(1)$	$O(\log n)^*$
Alberi + euristica sul rango	$O(\log n)$	$O(1)^+$
Alberi + euristica sul rango + compressione cammini	$O(1)^*$	$O(1)$

8 Programmazione dinamica

8.1 Donimo

Quanti modi ho di disporre tasselle di domino in una scacchiera $2 \times n$?

8.1.1 Soluzione

- Salvo in $dp[i]$ il numero di combinazioni che ci sono per un rettangolo $2 \times i$
- Ho due opzioni:
  - Metto 2 tessere in orizzontale, allora $dp[i] = dp[i-2]$
  - Metto 1 tessera in verticale, allora $dp[i] = dp[i-1]$
- Quindi $dp[i] = dp[i-1] + dp[i-2]$
- La soluzione è $\text{Fib}(n)$

8.2 Hateville

Ho un vettore di prezzi. Se prendo un prezzo $v[i]$ non posso prendere $v[i-1]$ e $v[i+1]$. Trova prezzo massimo

8.2.1 Soluzione

- Salvo in $dp[i]$ il prezzo massimo che posso ottenere con i vicini $\leq i$
- Ho due opzioni:
  - Non prendo $v[i]$, allora il prezzo è $dp[i-1]$
  - Prendo $v[i]$, allora il prezzo è $dp[i-1] + v[i]$

8.3 Zaino

Zaino ha capacità C, ho n pezzi di peso $w[i]$ e profitto $p[i]$. Trova profitto massimo

8.3.1 Soluzione

- Crea matrice $n \times C$ in cui si salva $dp[i][j]$ il profitto massimo che si può ottenere con i pezzi $\leq i$ e capacità $\leq j$
- Ho due opzioni:
  - Prendo pezzo (i, j), allora il prezzo migliore è $dp[i-1][j - w[i]] + p[i]$
  - Non lo prendo, allora il prezzo è $dp[i-1][j]$
- Posso ottimizzare lo spazio tenendo salvato solo due righe della matrice, la i e la $i-1$

8.4 Zaino umbound

Vedi zaino, solo che non c'è limite al numero di oggetti che uno puo prendere

8.4.1 Soluzione

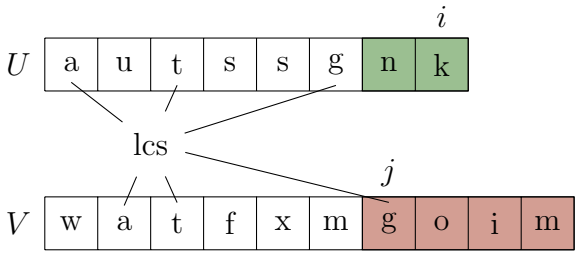
- Vettore dp in cui salvo in i il profitto massimo per uno zaino grande i
- Per ogni peso item x, il profitto massimo è $p[x] + dp[i - w[x]]$
- $dp[i]$ è il massimo fra tutti i valori trovati al punto 2

8.5 LCS

Date due stringhe U e T, trova la sottosequenza massimale. Una sottosequenza è una stringa che si ottiene da un'altra selezionandone solo alcuni caratteri (non necessariamente contigui, ma mantenendone l'ordine).

8.5.1 Soluzione

- Tabella dp con U su un lato e T sull'altro. In $dp[i][j]$ salvo la lunghezza della LCS fra la sottostringa $U[0, i]$ e $T[0, j]$
- Ho due opzioni:
  - $U[i] = T[j]$, allora $dp[i][j] = dp[i-1][j-1] + 1$ (aggiungo un carattere alla LCS più corta di 1)
  - $U[i] \neq T[j]$ allora $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$. Vedi immagine



Per migliorare la soluzione, se i caratteri sono diversi, devo aggiungere un carattere che sia nell'insieme dei caratteri dopo l'ultimo carattere comune. Quindi ho che

- A T, devo aggiungere un carattere che appartiene all'insieme rosso
- A U, devo aggiungere un carattere che appartiene all'insieme verde

Chiaramente la cosa è asimmetrica, per questo devo controllare $dp[i-1][j]$ e $dp[i][j-1]$

Dimostrazione formale : dobbiamo dimostrare che date due parole $U(u_1, \ldots, u_i)$ e $V(v_1, \ldots v_j)$ e $X(x_1, \ldots x_k)$ allora

- Se $u_i = v_j$ allora

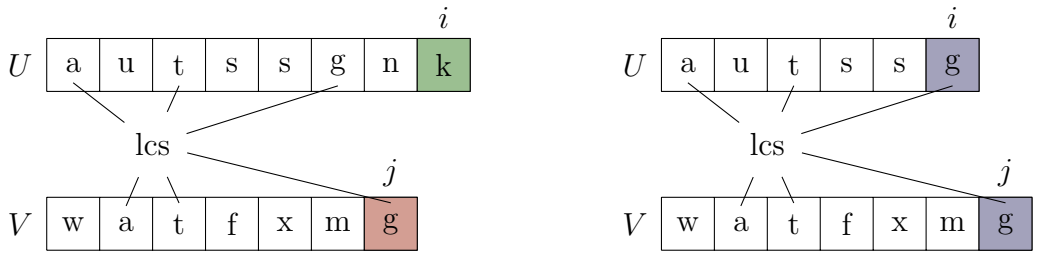
$$u_i = v_j = x_k$$
$$X(K-1) \in \mathcal{LCS}(U(i-1), V(j-1))$$

- Se $u_i \neq v_j$ e $x_k \neq u_i$ allora

$$X \in \mathcal{LCS}(U(i-1), V)$$

- Se $u_i \neq v_j$ e $x_k \neq v_j$ allora

$$X \in \mathcal{LCS}(U, V(j-1))$$



8.6 Occorrenza k approssimata

Data una stringa t e una p, diciamo che la distanza k di p da t è il numero minimo di *inserimenti, eliminazioni e scambi* che dobbiamo fare in t per far si che $t == p$.

$$t = \text{"scempio"}\ , \quad p = \text{esempio} \ \rightarrow k = 2$$

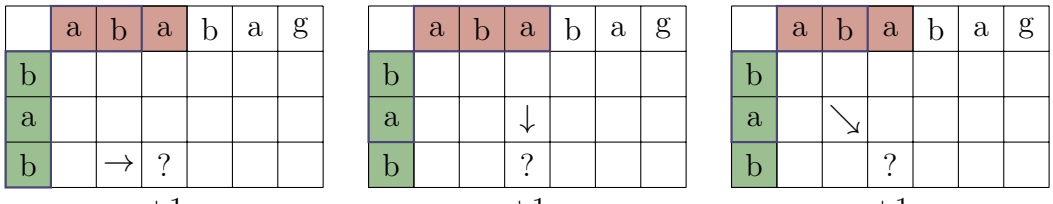
ad esempio, scambiando la "s" e "c" di *scempio* in "e" ed "s" rispettivamente

Il problma sta nel trovare in un testo t, la distanza minima di un pattern p da una sua qualsiasi sottostringa.

Ciò equivale a trovare quanti inserimenti, rimozioni e scambi devo fare nel testo per far si che il pattern diventi una sua sottostringa

8.6.1 Soluzione

- Inizializza matrice che ha p in verticale e t in orizzontale
- In $dp[i][j]$ si salva *il minor valore di k per far si che* $p[0,i]$ *sia sottostringa di* $t[0,j]$ *che finisca in* j
- Se $p[i] == t[j]$ allora non serviranno altre mosse per riportare la soluzione di $dp[i-1][j-1]$ alla soluzione corrente
- Se $p[i] \neq t[j]$ allora posso fare 3 cose:



Fai coincidere bab con ab e poi elimina a

Fai coincidere ba con aba e poi aggiungi b

Fai coincidere ba con ab e poi cambia a in b

La soluzione migliore è data dal minimo valore nell'ultima riga della tabella

Nota che la prima riga e la prima colonna vanno riempite rispettivamente con $[0, \ldots, 0]$ e $[1, 2, \ldots, n-1, n]$. Questo ha senso in quanto:

- Per far si che il pattern vuoto sia sottostringa di t non serve alsona mossa ($[0, \dots, 0]$)
- Per far sic che un pattern di lunghezza k sia sottostringa del testo vuoto è necessario aggiungere i k caratteri del pattern ($[1, 2, \dots, n-1, n]$)

8.7 Prodotto di catena di matrici

Si vuole fare il prodotto matriciale tra $[A_1, A_2, \dots, A_{n-1}, A_n]$. Il prodotto matriciale gode di proprietà associativa. Si trovi la parentizzazione che riduce al minimo il numero di moltiplicazioni scalari totali da compiere

Ad esempio, avendo $[A, B, C, D]$, posso parentizzare come segue:

$$[(A \cdot B) \cdot (C \cdot D)], \quad [A \cdot (B \cdot C) \cdot D], \quad [A \cdot (B \cdot (C \cdot D))]$$

e cosi via. Questo funziona in quanto per moltiplicare delle matrici bisogna assicurarsi che queste siano compatibili. Il numero di colonne della prima deve essere uguale al numero di righe della seconda. Ad esempio, indicando con $[\text{righe}, \text{colonne}]$ una matrice, una serie che può essere moltiplicata è la seguente:

$$[4, 5] \cdot [5, 2] \cdot [2, 10] \cdot [10, 7] \rightarrow [4, 5, 2, 10, 7]$$

Nota che la dimensione di ogni matrice può essere salvata in un vettore c in cui c_i contiene il numero di colonne della matrice i, che corrisponde al numero di righe della matrice $i+1$. Quindi il numero di moltiplicazioni necessarie per eseguire $A_i \times A_j$ sarà:

$$c_i \cdot (\cdot c_{i-1} \cdot c_j)$$

- c_i: numero di moltiplicazioni per calcolare una cella
- $(\cdot c_{i-1} \cdot c_j)$: dimensione della matrice risultante

8.7.1 Soluzione

- Creo matrice `dp` come seguen:

	1	2	3	4	5	6
1	0					
2	-	0				
3	-	-	0			
4	-	-	-	0		
5	-	-	-	-	0	
6	-	-	-	-	-	0

- In `dp[i][j]` salvo il minor numero di moltiplicazioni necessarie per moltiplicare le matrici fra `i` e `j`
- Costruisco matrice scorrento in diagonale a partire dalla diagonale più vicina alla diagonale principale. Il numero minore è dato dal numero minore date due parentizzazioni, ad esempio se ho

$$[A_3, A_4, A_5, A_6]$$

dovro tentare con

$$[(A_3) \cdot (A_4, A_5, A_6)], \quad [(A_3, A_4) \cdot (A_5, A_6)], \quad [(A_3, A_4, A_5) \cdot (A_6)]$$

- Il risultato finale si trova in `dp[1][n]`, dove `n` è il numero di matrici
- Per ricostruire la parentizzazione, posso salvarmi in una tabella `last[i][j]` l'indice a cui ho "spezzato la parentizzazione". Poi posso ricostruirla ricorsivamente come segue:

Algoritmo 3: *Find minimum parenthesization*

```
printPar(int last[][], int i, int j):
    if i == j then
        print "A["; print i; print "]";
    else
        print "(";
        printPar(last, i, last[i][j]);
        print ".";
        printPar(last, last[i][j] + 1, j);
        print ")";
```

Algorithm 1: Print optimal parenthesization

8.8 Intervalli pesati

Vengono dati n intervalli aperti $[a_1, b_1[, [a_2, b_2[, \dots [a_n, b_n[$. Ogni intervalli ha un valore w_i. Trovare il valore massimo che si può ottenere selezionando intervalli non sovrapposti.

8.8.1 Soluzione

- Ordina intervalli per tempo di fine
- Definisco la funzione `pred(i)`, che ritorna il *predecessore* di un intervallo, ossia il primo intervallo che ha tempo di fine minore del tempo di inizio di i
- Creo vettore `dp` che salva in `i` il valore massimo ottenibile con gli intervalli fino ad `i` compreso
- Itero su intervalli. Per ciascun intervallo `i` posso:
  - Selezionarlo: in questo il valore massimo ottenibile è dato da `dp[pred(i)] + w_i` a
  - Non selezionarlo: in questo caso il valore massimo è uguale al precedente `dp[i-1]`

Complessità: $O(n \log n)$

9 Algoritmi grafi più avanzati

9.1 Teorema di Bellman

Definizione 13: *Albero di copertura*

Dato un grafo $G = (V, E)$ non orientato e connesso, un albero di copertura è un sottoinsieme $T = (V, E_t)$ tale che:

- T è un albero
- T contiene tutti i vertici di G
- $E_t \in E$

In pratica un albero che contiene tutti i vertici del grafo G. Ci possono essere più aleri di copertura

Definizione 14: *Albero di copertura ottimo*

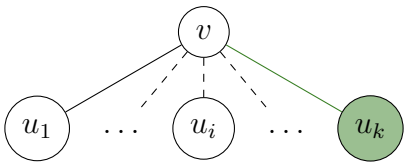
Un albero di compertura ottimo è un albero di copertura T tale che la somma dei pesi degli archi di T è minima

Teorema 5: *Teorema di Bellman*

Un albero *dei cammini minimi* T è ottimo se e solo se:

$$d[v] = d[u] + w(u, v) \text{ per ogni arco } \in E_t$$
$$d[v] \leq d[u] + w(u, v) \text{ per ogni arco } \in E$$

Il modo più facile per interpretare graficamente è il seguente:



In questo caso va pensato come: per ogni nodo, per ogni arco che vi entra, la distanza minima per arrivare a quel nodo è data da la distanza minima di uno dei suoi neighboors + il peso dell'arco per arrivare a v. Da tutti gli altri archi avrò per forza distanze maggiori. Nell'albero di copertura minimo avrò solo gli archi con cui raggiongo il nodo con distanza minore

Mia dimostrazione

- T ottimo $\rightarrow$ condizioni di Bellman:
  - Se T è ottimo, allora
$$d^T[x] = d[x] \quad \forall x$$

– Per costruzione, se $(u, v) \in T$ allora

$$d^T[v] = d^T[u] + w(u, v)$$
$$d[v] = d[u] + w(u, v)$$

questo è vero per quanto detto a punto 9.1 a

– Se per assurdo non fosse vero che $d[v] \leq d[u] + w(u, v)$ allora

$$d[v] > d[u] + w(u, v)$$
$$d^T[v] > d^T[u] + w(u, v)$$

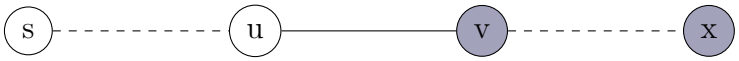
quindi T non sarebbe ottimale, assurdo

∘ Condizioni di Bellman $\rightarrow T$ ottimo:

– Se T per assurdo non fosse ottimo, allora esisterebbe un nodo x tale che

$$d^T[x] > d[x]$$

$$d[u] = d^T[u] \qquad d[v] < d^T[v]$$



– Dunque so che

$$\begin{aligned} d^T[v] &= d^T[u] + w(u, v) \\ &= d[u] + w(u, v) \\ &= d[v] \end{aligned}$$

quindi $d^T[v] = d[v]$, il che contraddice $d[v] < d^T[v]$, ossia il fatto che la distanza di v sia migliorabile

9.2 Dijkstra

Algoritmo 4: *Dijkstra*

Input: Graph G, starting node s
Output: Shortest path tree T, distance array d

```
void shortestPath (Graph G, Node s):
    PriorityQueue Q = PriorityQueue(); Q.insert(s, 0);
    while not Q.isEmpty() do
        u = Q.deleteMin();
        b[u] = false;
        foreach v ∈ G.adj(u) do
            if d[u] + G.w(u, v) < d[v] then
                if not b[v] then
                    Q.insert(v, d[u] + G.w(u, v));
                    b[v] = true;
                else
                    Q.decrease(v, d[u] + G.w(u, v));
                T[v] = u;
                d[v] = d[u] + G.w(u, v);
    return (T, d);
```

Algorithm 2: Dijkstra algorithm

L'algoritmo funziona perchè ho la certezza che un nodo, quando viene estratto (`Q.deleteMin()`), abbia la lunghezza minore. Per questo non verra mai reinserito.

- Ogni nodo viene inserito solo una volta
- Ogni nodo quando viene estratto ha la distanza minore

Tipo algoritmo	Complesità	Note
Funzione `min`	$O(V^2)$	nota 1
Min priority queue	$O(E \log V)$	nota 2
Fibonacci heap	$O(E + V \log V)$	nota 3

- Nota 1 V chiamate a `min` a costo $O(V)$
- Nota 2 E chiamate a `decrease` a costo $O(\log V)$
- Nota 3 V chiamate a `find_min` a costo $O(\log V)$ e E chiamate a `decrease` a costo *ammortizzato* $O(1)$

Operazione	#	Vettore	Priority queue	Fibonacci Heap
`delete min`	$O(n)$	$O(n)$	$O(\log(n))$	$O(\log(n))$
`insert`	$O(n)$	$O(1)$	$O(\log(n))$	$O^*(1)$
`decrease prio`	$O(m)$	$O(1)$	$O(\log(n))$	$O^*(1)$
Totale		$O(n^2)$	$O(m \log(v))$	$O(m + n \log(n))$

9.3 Bellman-Ford

Algoritmo 5: *Bellman Ford*

Input: Graph G, starting node s
Output: Shortest path tree T, distance array d

```
void shortestPath (Graph G, Node s):
    Queue Q = Queue(); Q.enqueue(s);
    while not Q.isEmpty() do
        u = Q.dequeue();
        b[u] = false;
        foreach v ∈ G.adj(u) do
            if d[u] + G.w(u, v) < d[v] then
                if not b[v] then
                    Q.enqueue(v);
                    b[v] = true;
                T[v] = u;
                d[v] = d[u] + G.w(u, v);
    return (T, d);
```

Algorithm 3: Bellman-Ford algorithm

- L'idea è che ogni k-esima volta che svuoto la coda sto trovo i cammini minimi di lunghezza (in archi) di al più k
- Un cammino semplice (senza cicli) può avere al più $v - 1$ nodi
- Ripetendo il rilassamento per ogni edge $v - 1$ volte ho trovato tutti i cammini semplici di lunghezza minima
- Runnando una seconda volta l'algoritmo di belman ford può succedere che vi siamo dei nodi che migliorano ancora. Questo significa che questo nodi sono in un ciclo di peso negativo

Complesità
$O(EV)$

9.4 Dag shortest path with negative wheights optimization

I cammini minimi in un DAG sono sempre ben definiti; anche in presenza di pesi negativi, non esistono cicli, dunque è sufficiente rilassare gli archi una volta sola in ordine topologico

Algoritmo 6: *Dag shortest path with negative wheights optimization*

Input: Graph G, starting node s
Output: Shortest path tree T, distance array d

```
void shortestPath (G, s):
    int [] d = new int[1...G.n] ;          ▷ d[u] è la distanza da s a u
    int [] T = new int[1...G.n] ;   ▷ T[u] è il padre di u nell'albero T
    foreach u ∈ G.V() − {s} do
        T[u] = nil; d[u] = +∞;
    T[s] = nil; d[s] = 0;
    Stack S = topsort(G);
    while not S.isEmpty() do
        u = S.pop();
        foreach v ∈ G.adj(u) do
            if d[u] + G.w(u, v) < d[v] then
                T[v] = u;
                d[v] = d[u] + G.w(u, v);
    return (T, d);
```

Algorithm 4: Dag shortest path with negative wheights optimization

Complessità	Note
$O(V+E)$	nota 1

- Nota 1: $O(V+E)$ per topo sort e $O(E)$ per rilassare ogni arco

9.5 Floyd Warshall

Se volessimo calcolare i cammini minimi fra tutti i nodi di un grafo, potremmo usare l'algoritmo di Dijkstra per ogni nodo. Questo ha complessità $O(V^2 \log V)$. L'algoritmo di *Floyd Warshall* si comporta un po meglio in questo caso

Definizione 15: *Cammino k-vincolato*

Un cammino $k-$vincolato è un cammino fra due nodi x e y tale per cui la distanza è minima e vengono usati solo i nodi con indice $0 \dots k$. Si indica con

$$p^k_{x,y}$$

L'algoritmo di Floyd-Warshall usa questa definizione per calcolare con programmazione dinamica tutte le coppie di distanze fra ogni vertice.

- Creo matrice `DP` in cui `DP[i][j]` è salvata la distanza minima fra i vertici da `i` e `j`
- Calcolo di volta in volta la distanza k-vincolata secondo la seguente logica:

– Se conosco la distanza migliore $k-1$ vincolata, allora nel calcolo della k vincolata posso includere il nuovo vertice k oppure no. Quindi devo scegliere il risultato migliore che ottengo

$$\underbrace{(\text{da } \texttt{x} \text{ a } \texttt{k})}_{\text{k-1 vincolato}} - - (\texttt{k}) - - \underbrace{(\text{da } \texttt{k} \text{ a } \texttt{y})}_{\text{k-1 vincolato}} \qquad \text{usando } \texttt{k}$$

$$\underbrace{(\text{da } \texttt{x} \text{ a } \texttt{y})}_{\text{k-1 vincolato}} \qquad \text{non usando } \texttt{k}$$

Dunque possiamo calcolare la matrice con formula ricorsiva

```
dp[i][j] = max{dp[i][j], dp[x][k] + dp[k][y]}
```

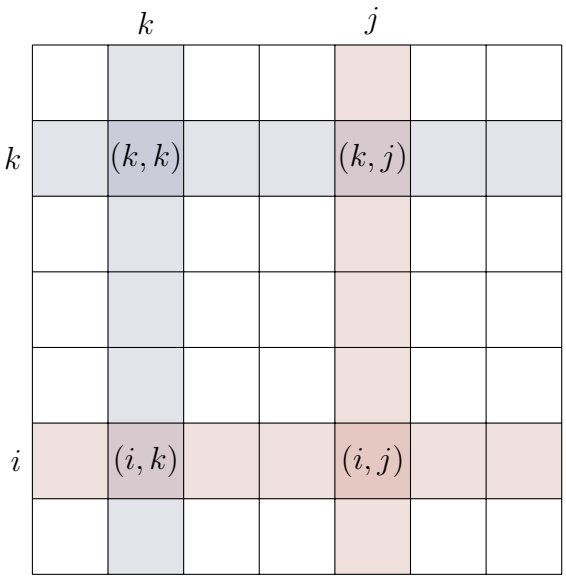
Algoritmo 7: *Floyd-Warshall*

Input: Graph G
Output: Shortest path matrix d, predecessor matrix T

```
(int[][], int[][]) floydWarshall (Graph G):
    for k = 1 to G.n do
        foreach u ∈ G.V() do
            foreach v ∈ G.V() do
                if d[u][k] + d[k][v] < d[u][v] then
                    d[u][v] = d[u][k] + d[k][v];
                    T[u][v] = T[k][v];
    return (d, T);
```

Algorithm 5: Floyd-Warshall algorithm

Nota che non è necessario tenere salvato in una matrice separata il livello $k-1$ in quanto



L'idea è che la colonna e la riga k non cambiano. Siccome devo prendere il minore tra `dp[i][k] + dp[k][k]` e `dp[i][k]`, le celle non cambieranno mai. Nel calcolo di una nuova cella, farò riferimento solo alla cella stessa e alle riga/colonna k-esima, dunque posso utilizzare la stessa tabella in quanto faccio riferimento solo ai valori calcolati in $k-1$

Per quanto riguarda i cicli negativi, come nell'algoritmo di Bellman-Ford descritto ins ezione 9.3, se dopo aver eseguito l'algoritmo di Floyd-Warshall la matrice delle distanze contiene un valore negativo sulla diagonale, allora il grafo contiene un ciclo negativo. In questo caso posso runnare una seconda iterazione dell'algoritmo, mettendo segnando tutte le posizioni che vengono migliorate.

$$m[i][j] \text{ migliora } \Leftrightarrow \text{ esiste ciclo negativo in path da } i \text{ a } j$$

9.6 Chiusura transitiva

Definizione 16: *Chiusura transitiva*

La chiusura transitiva di un grafo $G = (V, E)$ è un grafo $G^*(V, E^*)$ in cui

$$(u, w) \in E^* \Leftrightarrow v \text{ è raggiungibile da } u$$

Possiamo applicare l'algoritmo di *Floyd-Warshall* per calcolare la chiusura transitiva.

```
dp[i][j] = dp[i][j] or (dp[x][k] and dp[k][y])
```

L'algoritmo è praticamente uguale, solo che anzichè salvare il peso salviamo un `booleano` per tracciare i nodi raggiungibili

10 Greedy

10.1 Insime indipendente di intervalli

Abbiamo visto il problema degli intervalli pesati. Possiamo avere la versione greedy in cui non abbiamo un peso ma dobbiamo massimizzare il numero degli intervalli.

10.1.1 Soluzione

- Ordiniamo gli intervalli per tempo di fine
- Prendiamo ogni volta l'intervallo con tempo di fine minore.
- Scorriamo gli intervalli finche non ne troviamo uno con tempo di inizio $>$ del tempo di fine dell'ultimo intervallo selezionato

Dimostrazione:

- Sia S' una soluzione ottima per un intervallo
- Sia m l'intervallo con tempo di fine minore e m' l'intervallo con tempo di fine minore in S'

- Allora $(S' - \{m'\}) \cup \{m\}$ è una soluzione in quanto ha la stessa cardinalità ed è accettabile in quanto $b_m < b_{m'}$

10.2 Coin change

Devi dare un resto R e hai a disposizione una serie di tagli di monete $t[c]$. Hai a disposizione infinite monete per ogni taglio. Trova il numero minore di monete necessarie per dare un resto

10.2.1 Soluzione

- Prendo di volta in volta la moneta con taglio maggiore e ne sottraggo fino a quando ciò che avanza è minore del taglio stesso
- Ripeto finché non ho riempito il resto

Algoritmo 8: *Money change*

Input: Array di monete t, numero di monete n, valore da cambiare R
Output: Array x con la soluzione greedy del problema del cambio

```
int[] moneyChange (int [] t, int n, int R):
    ▷ Ordina le monete in modo decrescente
    int [] x = new int[1...n];
    for i = 1 to n do
        x[i] = ⌊R/t[i]⌋;
        R = R − x[i] · t[i];
    return x;
```

Algorithm 6: Coin change greedy per sistemi canonici

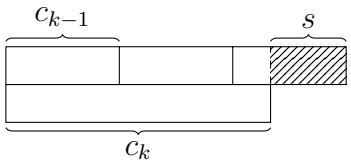
Nota bene! L'algoritmo funziona solo se i tagli di monete a disposizione consistono in un *sistema canonico*. Pensa ad esempio a `t = [5,4,1], R=8`, dove la soluzione ottima è `S=[4,4]`, mentre l'algoritmo greedy ritornerebbe `S=[5,1,1,1]`

L'algoritmo per verificare che un sistema sia canonico procede così:

- Un sistema con `t[0]=1` e `len(t)<3` è sempre canonico (ogni numero è multiplo di 1)
- Un sistema con `len(t)>2`. Dato $C_1 = [c_1, \dots c_{i-1}]$, $C_2 = [c_1, \dots c_i]$ è canonico se la soluzione greedy per un valore ben preciso di n ottengo una soluzione greedy migliore o uguale con C_2 rispetto che con C_1. Questo valore è così dfinito

$$n = \left\lceil \frac{t[i]}{t[i-1]} \right\rceil \cdot t[i-1]$$

Nota come n è sicuramente il valore più grande di c_i che può essere risolto con il numero minore di coins in $c[0, \dots, i-1]$.



Per questo preciso valore esiste un teorema che afferma che

$$C_2 \text{ canonico } \Leftrightarrow \texttt{greedy}(C_2, n) \ \leq \ \texttt{greedy}(C_1, n)$$

Posso quindi verificare con il seguente algoritmo se un sistema è canonico

Algoritmo 9: *Verifica sistema canonico di monete*

Input: Array di monete *coins*
Output: **true** se il sistema è canonico, **false** altrimenti

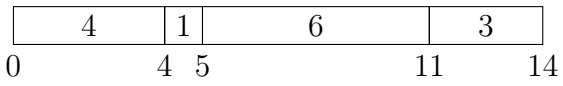
for $i = 2$ **to** `len` $(coins)$ **do**
  $n = \lceil coins[i]/coins[i-1] \rceil$;
  $m = coins[i-1] \cdot n - coins[i]$;
  if `greedy`$(coins, m, i-2) \geq n$ **then**
    return false
return `True`

Algorithm 7: Verifica Sistema Canonico

`greedy(coins, m, k)` ritorna la soluzione trovata dall'algoritmo greedy coin change, limitandosi ad utilizzare le prime `k` monete

10.3 Task scheduling

Abbiamo un processore che deve eseguire n task ognuna di lunghezza $t\,[i]$. Trova l'odrine che minimizza il tempo di completamento medio, dove il tempo di completamento di una task è il tempo a cui una task finisce di eseguire. Ad esempio:



il tempo di completamento di questa disposizione è

$$\frac{4+5+11+14}{4} = \frac{34}{4} = 8.5$$

10.3.1 Soluzione

- Ordino le task per lunghezza crescente

Prendendo una permutazione ottima in cui a indice m si trova la task più corta, posso scambiarla con il primo elemento ottenendo che:

- Per $i > m$ le task hanno lo stesso tempo di completamento rispetto a prima dello scambio
- Per $i < m$ le task hanno tempo di completamento $\leq$ rispetto a prima dello scambio

Quindi scambiando posso migliorare la soluzione

10.4 Zaino frazionario

Abbiamo un array di pesi e pressi rispettivamente $w[i]$ e $p[i]$ e una capacità C. Trovare la quantità di ciascun oggetto per massimizzare il profitto senza eccedere la capacità. La quantità può essere frazionaria

10.4.1 Soluzione

- Ordino per *profitto specifico descrescente* (`p[i]/w[i]`)
- Prendo quanto più possibile di ogni oggetto, a partire da quello con profitto specifico più alto

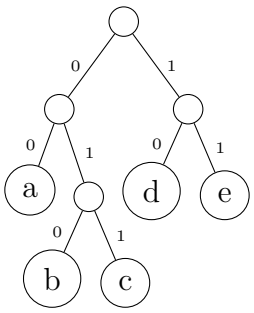
Informalmente posso dire che se l'oggeto con il peso specifico maggiore occupa x, non ha senso riempire questo spazio in parte con "oro" e il resto con argento, ma conviere riempire tutto con oro

10.5 Compressione di Huffman

Idea di base:

- Un file può essere rappresentato come stream di caratteri, dove ogni carattere ha una codifica binaria (es `a = 01100001`)
- Possiamo associare a sequenze che si ripetono spesso dei codici più corti, così da risparmiare spazio (es `a = 01100001` $\rightarrow$ `001`)

Per identificare un carattere si possono usare degli alberi di parsing come segue:



a	b	c	d	e
00	010	011	10	11

Algoritmo 10: *Huffmann*

parti dalla radice;
while *file non è finito* **do**
  leggi un bit;
  if *bit è zero* **then**
    vai a sinistra;
  else
    vai a destra;
  if *nodo foglia* **then**
    stampa il carattere;
    torna alla radice;

Algorithm 8: Algoritmo di decodifica

Siccome i caratteri possono essere codificati a lunghezza variabile in base alla frequenza con cui compaiono si possono costruire alberi di parsing più o meno efficienti:

Caratteri	a	b	c	d	e	f	Dim.
Frequenza	45%	13%	12%	16%	9%	5%	
ASCII	01100001	01100010	01100011	01100100	01100101	01100110	$8n$
Codifica 1	000	001	010	011	101	101	$3n$
Codifica 2	0	101	100	111	1101	1100	$2.24n$

Tabella 1: Tabella comparativa delle codifiche

Costo totale: $(0.45 \cdot 1 + 0.13 \cdot 3 + 0.12 \cdot 3 + 0.16 \cdot 3 + 0.09 \cdot 4 + 0.05 \cdot 4) \cdot n = 2.24n$

Note come nel caso della codifica 3, devo scegliere un insieme di stringhe binarie che non condividono alcun prefisso, altrimenti si rischierebbe di avere un conflitto durante il riconoscimento, ad esempio:

$$a = 0 \quad b = 1 \quad c = 11$$

la stringa

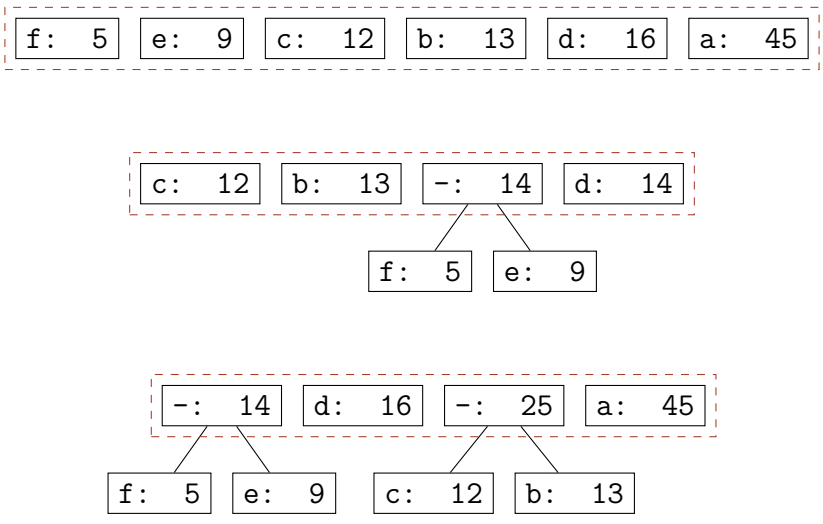
```
0111
```

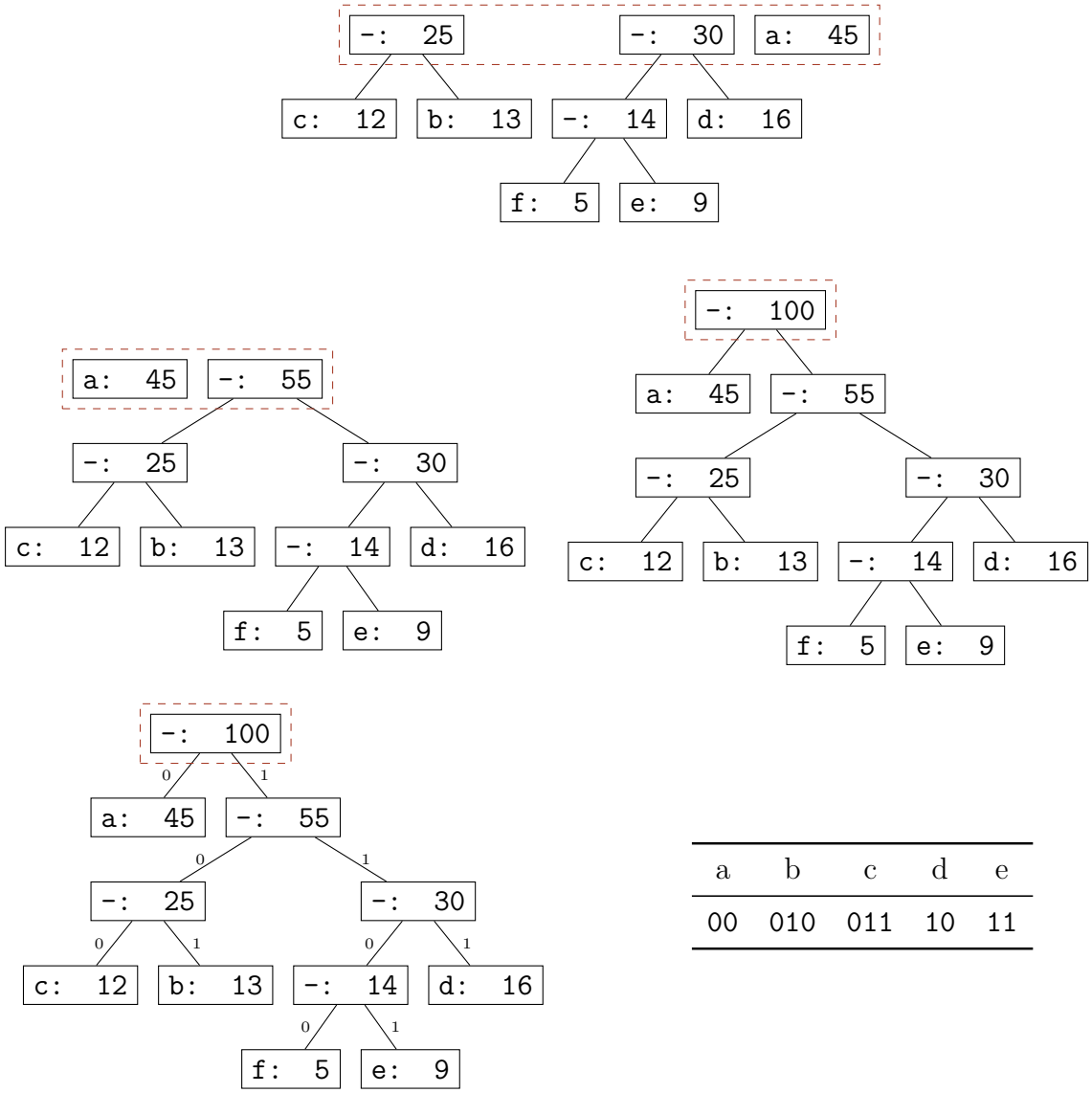
ha interpretazione ambigua

10.5.1 Costruzione albero di parsing ideale

Per costruire un albero di parsing ottimale si può procedere così:

- Creare una lista di nodi foglia, ognuno con carattere e il relativo numero di occorrenze nel file
- Rimuovere i due nodi con frequenze minori f_x, f_y
- Creare un nodo padre con etichetta `"-"` e frequenza $f_x + f_y$
- Collegare i due nodi rimossi con il nuovo nodo
- Aggiungere il nodo così creato all'insieme





a	b	c	d	e
00	010	011	10	11

Figura 1: Albero di parsing finale

10.5.2 Dimostrazione

- Consideriamo una soluzione ottima `T`
- Consideriamo il carattere `x` con *frequenza più bassa* $\rightarrow f[x]$ minima
- Consideriamo il carattere `a` con *profondità massima* $\rightarrow d_T[a]$ massima
- Consideriamo l'albero `T'` ottenuto scambiando `x` con `a`

- Dimostriamo che il costo finale di `T'` è $\leq$ di quello di `T`:

$$\begin{aligned} C(f,T) - C(f,T') &= \sum_{c\in\Sigma} f[c]d_T(c) - \sum_{c\in\Sigma} f[c]d_{T'}(c) \\ &= \big(f[x]d_T(x) + f[a]d_T(a)\big) - \big(f[x]d_{T'}(x) + f[a]d_{T'}(a)\big) \\ &= \big(f[x]d_T(x) + f[a]d_T(a)\big) - \big(f[x]d_T(a) + f[a]d_T(x)\big) \\ &= \big(f[a] - f[x]\big)\big(d_T(a) - d_T(x)\big) \\ &\geq 0 \end{aligned}$$

quindi

$$C(f,T) - C(f,T') \geq 0 \rightarrow C(f,T') \leq C(f,T)$$

10.6 Alberi di copertura minimali

Definizione 17: *Albero di copertura*

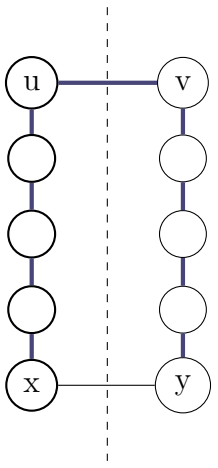
Un albero di copertura di un grafo $G = (V, E)$ è un sottoinsieme di E che contiene tutti i nodi di V la cui somma dei pesi è minima

Teorema 6: *Archi sicuri per minimum spanning tree*

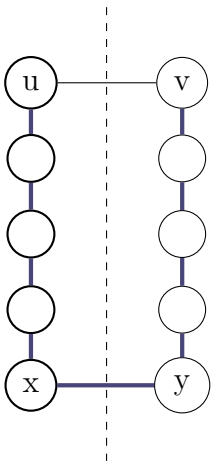
Dato un insieme di archi A contenuto in qualche albero di copertura e un taglio che rispetti A, se (u, v) è un arco leggero che attraversa il taglio, allora (u, v) è *sicuro per* A

Dimostrazione: considera

- T: un albero di copertura minimo.
- A: un sottoinsieme degli archi di T
- $t = (S, V - S)$ un taglio che rispetti A
- (u, v) un arco leggero a cavallo di t
- (x, y) l'arco di T che attraversa il taglio



T



T'

Dove i nodi con contorno grossi costituiscono A e gli edge blue costituiscono appartengono all'albero di copertura

- Se $(u, v) \in T$ allora è per forza sicuro
- Se $(u, v) \notin T$ allora procedo così:
  - Considero un taglio che sia trapassato da u, v
  - Esisterà un arco (x, y) che attraversa il taglio
  - Se considero $T' = T - \{x, y\} + \{u, v\}$
    - T' è più leggero di T in quanto $w(u, v) \leq w(x, y)$ ((u, v) è leggero)
    - T è più leggero di T' in quanto è *minimum spanning tree*

10.6.1 Algoritmo di kruskal

Secondo quanto dimostrato nel teorema sugli archi sicuri, posso costruire un albero di copertura minimo in questo modo:

- Ogni nodo appartiene inizialmente a un gruppo diverso
- Scorro gli edge in ordine di peso crescente
- Se l'edge connette due nodi che appartengono a gruppi diversi allora posso unire il gruppo. Altrimenti non aggiungo nulla, se no introdurrei un ciclo
- L'albero viene registrato come `set<edges>`

Algoritmo 11: *Kruskal MSPT*

```
Set < Edge > kruskal (Edge[] A, int n, int m):
    Set T = Set();
    MFSet M = Mfset(n);
    ▷ ordina gli edge per peso crescente
    int count = 0;
    int i = 1;
    ▷ Termina quando l'albero ha n − 1 archi o non ci sono più
      archi
    while count < n − 1 and i ≤ m do
        if M.find(A[i].u) ≠ M.find(A[i].v) then
            M.merge(A[i].u, A[i].v);
            T.insert(A[i]);
            count = count + 1;
        i = i + 1;
    return T;
```

Algorithm 9: Kruskal's Algorithm

Fase	Volte	Costo
Inizializzazione	1	$O(n)$
Ordinamento	1	$O(m \log m)$
Operazioni `find()`,`merge()`	$O(m)$	$O(1)^{(*)}$

Quindi la complessità dell'algoritmo di Kruskal è:

$$O\left(m \log m\right)$$

Nota che $O\left(m \log\left(m\right)\right) = O\left(m \log\left(n^2\right)\right) = O\left(m \log\left(n\right)\right)$

10.6.2 Algoritmo di Prim

A differenza dell' algoritmo di Kruskal, l'algoritmo di Prim usa una *priority queue* e ritorna l'albero di copertura come *vettore dei padri*

- Mantengo un unico albero T
- Creo taglio fra vertici contenuti dell'albero e altri vertici: $t = (T, V - A)$
- Aggiungo ad ogni iterazione un arco leggero di t

Algoritmo 12: *Prim MSPT*

```
int [] prim(Graph G, Node r):
    PriorityQueue Q = MinPriorityQueue();
    PriorityItem [] pos = new PriorityItem[1...G.n];
    int[] p = new int[1...G.n];
    foreach u ∈ G.V() \ {r} do
        pos[u] = Q.insert(u, +∞);
    pos[r] = Q.insert(r, 0);
    p[r] = 0;
    while not Q.isEmpty() do
        Node u = Q.deleteMin();
        pos[u] = nil;
        foreach v ∈ G.adj(u) do
            if pos[v] ≠ nil and w(u, v) < pos[v].priority then
                Q.decrease(pos[v], w(u, v));
                p[v] = u;
    return p;
```

Algorithm 10: Prim's Algorithm

Fase	Volte	Costo
Inizializzazione	1	$O(n \log n)$
`deleteMin()`	$O(n)$	$O(\log n)$
`decreasePriority()`	$O(m)$	$O(\log n)$

Quindi la complessità dell'algoritmo di Prim è:

$$O(m \log n)$$

11 Rete di flussi - Fork-Folkerson

Ho un grafo che rappresenta una rete di *"manichette"* d'acqua, ossia un grafo orientato tale che:

- Nodo sorgente $s \in V$
- Nodo pozzo $t \in V$
- Funzione di capacità: $c : V \times V \to \mathbb{R} \geq 0$ tale che $(x, y) \notin E \Rightarrow c(x, y) = 0$. Di fatto il peso degli archi indica quanta acqua può passare

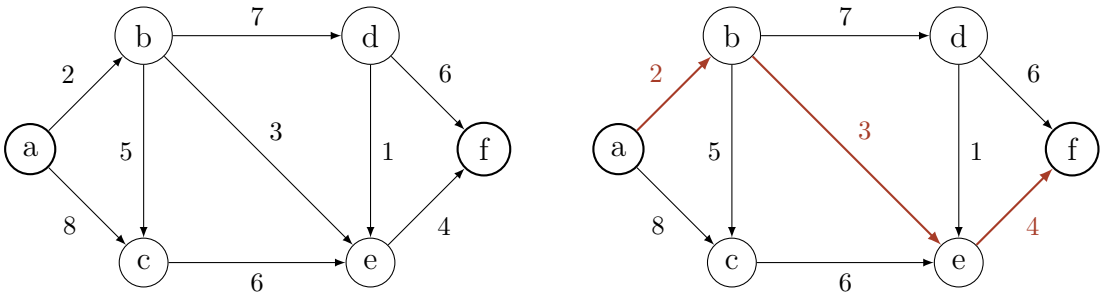
11.0.1 Flusso

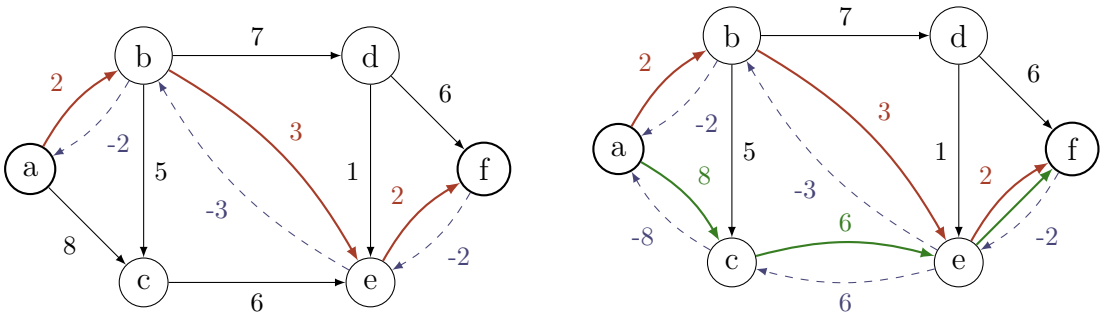
Il flusso è una funzione che indica lo scorrimento di acqua in ogni manichetta e gode di 3 proprietà:

- Vincolo sulla capacità: non può passare può acqua della portata di una manichetta
- Antisimmetria: $f(x, y) = -f(y, x)$. Di fatto aggiungo un arco al contrario da quale posso far passare l'acqua per *"correggere gli errori"*, ossia evito di mandare acqua per un percorso in favore di un altro
- Conservazioe del flusso: il flusso totale entrante su di ogni nodo è uguale al flusso uscente, ad eccezione di s e t

L'algoritmo consiste in :

- Trovo percorso da s a t e delle manichette che passo prendo il prezzo minore
- Sottraggo questo peso a tutti gli archi del percorso e lo aggiungo agli archi inversi (i quali sono inizialmente inizializzati a zero)
- Ripeto algoritmo finchè non è più possibile trovare percorso sa s a t





Nota come, quando passo su di un arco inverso è come dire

Ok, evita di mandarmi questa data quantità di acqua, tanto io manderò la stessa quantità al nodo alla quale tu la mandavi. Allo stesso modo, ti trovero un percorso che arrivi a t cosisshè tu possa far confluire la quantità di acqua che hai evitato di mandarmi

11.0.2 Complessità

Algoritmo	Complessità
Ford-Fulkerson	$O(E \lvert f* \rvert)$
Edmonds-Karp	$O(VE^2)$

Nota che la complessità compare solo E per il costo delle BFS/DFS siccome nel problema delle reti di flusso il grafo è per forza di cose connesso e dunque $V = O(V)$

Definizione 18: *Flusso per un taglio*

Dato un taglio $T = (S, T)$, il *flusso netto* per esso è definito come

$$\sum_{x \in S, y \in T} f(x, y)$$

11.0.3 Teorema flusso per un taglio

Teorema 7: *Flusso per un taglio*

Dato un *qualsiasi* taglio $T(S, T)$ di una rete di flusso e una funzione di flusso f, allora il flusso per il taglio è uguale al flusso complessivo:

$$\sum_{x \in S, y \in T} f(x, y) = |f|$$

Dimostrazione informale:

- Il flusso per un taglio è dato dal peso degli archi del flusso "a cavallo" del taglio stesso

$$\sum_{x \in S, y \in T} f(x, y)$$

- Per l'antisimmetria, il flusso per il taglio è dato dalla somma di ogni arco che parte da un vertice $\in S$ e arriva in un qualsiasi altro $\in V$, in quanto gli archi fra due vertici $\in S$ si controbilanciano (*antisimmetria*)

$$\sum_{x \in S, y \in V} f(x, y)$$

- Posso ora spezzare questa quantità nella somma degli archi che parono dalla sorgente s e quelli che partono da $S - \{s\}$.

$$\underbrace{\sum_{x \in s - \{s\}, y \in V} f(x, y)}_{\text{archi che partono da } S - \{s\}} + \underbrace{\sum_{y \in V} f(s, y)}_{\text{archi che partono da } s}$$

- Ora ho che per la conservazione del flusso, la somma degli archi uscenti da ciascun nodo (salvo $\{s, t\}$) è pari a zero. Dunque

$$\sum_{x \in s - \{s\}, y \in V} f(x, y) = 0$$

e so anche che per definizione:

$$\sum_{y \in V} f(s, y) = |f|$$

11.0.4 Teorema capacità taglio minimo

Teorema 8: *Capacità del taglio*

Il flusso massimo è limitato superiormente dalla capacità del taglio minimo, ovvero il taglio la cui capacità è minore fra tutti i tagli.

Dimostrazione:

- Per il teorema riguardo il flusso per un taglio, sappiamo che il flusso complessivo è pari al flusso che attraversa un qualsiasi taglio
- E' facile capire inoltre che il flusso per un taglio è limitato superiormente dalla capacità di quel taglio, in quando per ogni arco che attraversa il taglio ho che

$$f(x, y) \leq c(x, y)$$

- Prendo il taglio con capacità minore $\rightarrow$ il flusso non può essere maggiore di questa capacità

11.1 Dimostrazione correttezza

Teorema 9: *Flusso massimo, cammini aumentanti e taglio*

Le seguenti affermazioni sono equivalenti:

- f è il flusso *massimo*
- Non esistono cammini aumentanti
- $|f|$ è uguale alla capacità del *taglio minimo*

L'equivalenza si dimostra circolarmente:

$$1 \Rightarrow 2, \quad 2 \Rightarrow 3, \quad 3 \Rightarrow 1$$

1. $1 \Rightarrow 2$
   - Se esistessero cammini aumentanti allora sarebbe assurdo che il flusso fosse massimo
2. $2 \Rightarrow 3$
   - Prendo insieme di nodi raggiungibili da s e lo chiamo S. Considero taglio $T = (S, V - S)$
   - Tutti gli edge a cavallo del taglio devono essere saturati, altrimenti significa che potrei raggiungere altri nodi da s
   - Se un taglio è saturato, questo deve per forza essere il taglio minimo, altrimenti esisterebbe un taglio più piccolo che violerebbe il vincolo sulla capacità
3. $3 \rightarrow 1$
   - Il flusso è limitato superiormente dalla capacità di un qualsiasi taglio
   - Se $|F|$ è uguale al taglio minimo allora f è per forza massimo

11.2 Dimostrazione complessità Edmonds-Karp

11.2.1 Teorema monotonia

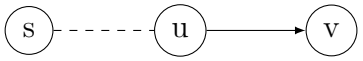
Teorema 10: *Monotonia distanze in iterazioni edmonds-karp*

Le distanze minime di un nodo v dalla sorgente s ($d(v)$) possono solo aumentare dopo ogni volta che aggiorniamo il flusso tramite un cammino aumentante

Dimostrazione:

- Chiamo G il grafo prima di essere aggiornato e G' il grafo dopo l'aggiornamento
- Suppongo per assurdo che esistano dei nodi con distanza minima minore in G' rispetto che G

- Chiamo v il nodo con distanza minore fra questi. Considero il seguente shortest path:



- Considero che, siccome v è il nodo che ha diminuito la distanza *di distanza minima*, allora u non può aver diminuito la sua distanza:

$$d_{\text{dopo}}(u) \geq d_{\text{prima}}(u)$$

- Dimostro che l'arco (u, v) *non* può appartenere a G in quanto

$$\begin{aligned} d_{\text{prima}}(v) &\leq d_{\text{prima}}(u) + 1 \\ &\leq d_{\text{dopo}}(u) + 1 \\ &= d_{\text{dopo}}(v) \end{aligned}$$

  quindi $d_{\text{prima}}(v) \leq d_{\text{dopo}}$, il che contradddice l' ipotesi

- Dunque $(u, v) \notin G$ e $(u, v) \in G'$. L'unico modo perché questo accada è che l'arco (u, v) sia stato "riattivato" percorrendolo al contrario (*con uno shortest path*). Considero dunque

$$\begin{aligned} d_{\text{prima}}(v) &= d_{\text{prima}}(u) - 1 \\ &\leq d_{\text{dopo}}(u) - 1 \\ &= (d_{\text{dopo}}(v) - 1) - 1 \end{aligned}$$

  quindi ancora una volta ottengo che $d_{\text{prima}}(v) \leq d_{\text{dopo}} - 2$, il che contradddice l' ipotesi

11.2.2 Dimostrazione complessità edmon karps

Edmonds-Karp usa BFS. Così facendo si può dimostrare che aumento il flusso massimo VE volte:

- Considero arco (x, y). Ogni volta che rimuovo cammino aumentante degli archi si "spengono"
- Dimostro che un arco critico (x, y) (che fa da collo di bottiglia) aumenta la sua distanza in archi di almeno 2 prima di ritornare critico
  - Prima di rimuovere cammino aumentante distanza da y = distanza da $x + 1$
  - Dopo averlo rimosso, per "riaccendere" arco devo ripercorrerlo al contrario
  - Prendendo percorso in cui ripercorro arco al contrario
  - Quindi $d_{\text{ dopo spegnimento }}(x) = d_{\text{ dopo spegnimento }}(y) + 1$
  - Visto che le distanze NON possono diminure ho che

$$\begin{aligned} d_{\text{ dopo spegnimento }}(x) &= d_{\text{ dopo spegnimento }}(y) + 1 \\ &\geq d_{\text{ prima spegnimento }}(y) + 1 \\ &= \left(d_{\text{ prima spegnimento }}(x) + 1\right) + 1 \end{aligned}$$

- Ho dimostrato quindi che la distanza in archi aumenta almeno di 2 ogni volta che un arco diventa critico
- Dato (x, y) con $d(x) \leq d(y)$ allora $d(x)$ è al massimo $V - 2$
- La distanza MINIMA di un nodo non può aumentare all'infinito, quindi, dato che aumenta di 2 ogni volta, ogni arco può diventare critivo al più $\frac{(V-2)}{2}$ volte, ossia $O(V)$
- Siccome ci sono $O(E)$ archi, al più posso aumentare $O(VE)$ volte

12 Backtracking

Solitamente i problemi che richiedono una soluzione *brute-force* sono in qualche modo riconducibili a problemi di *combinatoria*. In particolare molti problemi di backtracking sono risolvibili con versioni modificate dei due seguenti algoritmi

12.1 Generete all subsets

Algoritmo 13: *Generate all subsets*

```
void print_subsets_rec (Item set, Item item, int index):
    if index == 0 then
        print_set (set, choiches);
        return;
    choiches[index] ← false;
    print_subsets_rec (set, choiches, index - 1);
    choiches[index] ← true;
    print_subsets_rec (set, choiches, index - 1);
void print_subsets (Item set):
    choices ← empty array;
    for i ← 1 to #set do
        choices[i] ← false;
    print_subsets_rec (set, choices, #set);
```

Algorithm 11: Print Subsets of a Set (Recursive)

12.2 Generete all permutations

Algoritmo 14: *Generate all permutations*

```
void print_perm_rec (Item v, Item choices, int index):
    if index == 0 then
        print_array (choices);
        return;
    for i ← 1 to #v do
        choices[index] ← v[i];
        print_perm_rec (v, choices, index - 1);
void print_perm (Item v):
    choices ← empty array;
    for i ← 1 to #v do
        choices[i] ← false;
    print_perm_rec (v, choices, #v);
```

Algorithm 13: Print All Permutations of a Set

12.3 K-sottoinsiemi

Per questo algoritmo possiamo usare la versione naive, meno efficiente, modificando l'algoritmo dei sottoinsiemi

Algoritmo 15: *K-subsets naif*

```
void print_subsets_rec (Item set, Item item, int index, int k):
    if index == 0 and count_ones(choices) == k then
        print_set (set, choiches);
        return;
    choiches[index] ← false;
    print_subsets_rec (set, choiches, index - 1);
    choiches[index] ← true;
    print_subsets_rec (set, choiches, index - 1);
```

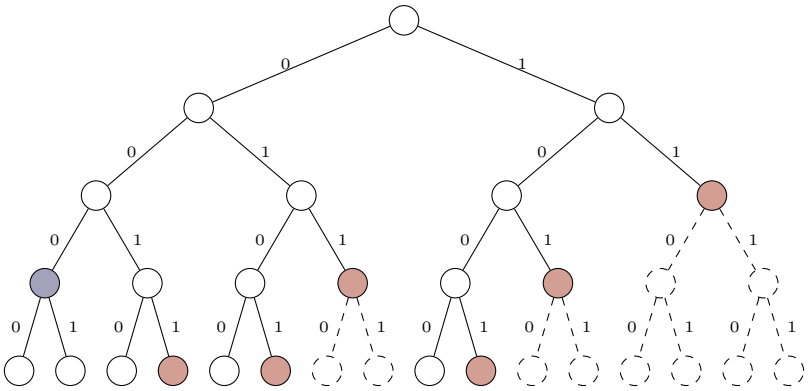
Algorithm 14: Print Subsets of a Set (Recursive)

Possiamo utilizzare una verisone molto più ottimizzata utilizzando un pruning:

Algoritmo 16: *K-subsets con pruning*

```
void kssRec (int n, int missing, int[] S, int i):
    if missing == 0 then
        processSolution (S, i − 1);
        return;
    else if i ≤ n and 0 < missing ≤ n − (i − 1) then
        foreach c ∈ {0, 1} do
            S[i] ← c;
            kssRec (n, missing − c, S, i + 1);
```

Algorithm 15: kssRec (Recursive Subset Sum Algorithm)



Utilizzando il *pruning*, andiamo a risparmiare un sacco di salcolo (guarda i nodi tratteggiati). In particolare

12.4 Subset sum

Dato un insieme di interi positivi `A` e un numero intero positivo `k`, ritornare `true` se esiste una combinazione di elementi di `A` che sommati danno `k`

Algoritmo 17: *Subset sum*

```
bool ssRec (int[] A, int n, int missing, int[] S, int i):
    if missing == 0 then
        processSolution (S, i − 1) ;        ▷ Stampa gli indici della soluzione
        return true;
    else if i > n or missing < 0 then
        ▷ Terminati i valori o somma eccessiva
        return false;
    else
        foreach c ∈ {0, 1} do
            S[i] ← c;
            if ssRec (A, n, missing − A[i] · c, S, i + 1) then
                return true;
        return false;
```

Algorithm 16: Recursive Subset Sum with Solution Printing

`missing` tiene conto della somma mancante da riempire per arrivare a `k`.

12.5 Problema delle 8 regine

Supponiamo di avere `n` regine da disporre in una griglia quadrata `nxn`. Una regina "minaccia" un'altra regine sa sono sulla stessa riga, colonna o diagonale. Il problema consiste nel disporre le regine in modo tale che nessuna di esse possa "prendere" un'altra. Vediamo un'ottimizzazione alla volta

- Numeriamo posizioni scacchiera da 1 a 64. Ogni posizione può avere una regina oppure no.

$$\text{Complessità: } 2^{n^2} = 2^{64} = \approx 1.84 \cdot 10^{19}$$

- Visto che dobbiamo piazzare solo 8 regine, possiamo considerare un vettore `v[1...8]` con `v[i] = posizione i-esima regina`. Ogni cella ha 64 possibilità, quindi:

$$\text{Complessità: } \left(n^2\right)^n = 64^8 = \approx 2.81 \cdot 10^{14}$$

- Evito di considerare permutazioni di schieramenti. Quando piazzo la regina i, considero solo le posizioni $>$ rispetto alla posizione della regina $i-1$. Dato che per ogni schieramento non calcolo le permutazioni il costo è di

$$\text{Complessità: } \frac{(n^2)^n}{n!} = \frac{64^8}{8!} \approx 6.98 \cdot 10^9$$

- Considero che ogni colonna della scacchiera contiene esattamente 1 regina, altrimenti ci sarebbero regine che si minacciano. Ognuna delle n colonne ha n possibili posizioni:

$$\text{Complessità: } (n)^n = 8^8 \approx 1.67 \cdot 10^7$$

- Se piazzo una regina in una riga j, allora non potrò piazzare nessuna delle regine successive in quella colonna. Quindi ogni colonna le posizioni possibili diminuiscono:

Complessità: $n! = 8! = 40320$

12.6 Sudoku

Algoritmo 18: *Sudoku*

```
bool sudoku (int[][] S, int i):
    if i == 81 then
        processSolution (S, n) ;    ▷ Process the completed solution
        return true;
    int x = i  mod 9;
    int y = ⌊i/9⌋;
    Set C = moves(S, x, y);
    int old = S[x][y];
    foreach c ∈ C do
        S[x][y] = c;
        if sudoku (S, i + 1) then
            return true;
    S[x][y] = old;
    return false;

Set moves (int[][] S, int x, int y):
    Set C = Set();
    if S[x][y] ≠ 0 then
        C.insert(S[x][y]) ;                    ▷ Pre-inserted number
    else
        ▷ Check for conflicts
        for c = 1 to 9 do
            if check (S, x, y, c) then
                C.insert(c);
    return C;
bool check (int[][] S, int x, int y, int c):
    ▷ Check if c can be inserted in cell (x, y)
```

Algorithm 17: Sudoku Solver

Per comodità numero celle da 0 a 80. Di base scorro ogni cella, se questa è vuota provo ad inserire un qualsiasi numero. Se sono arrivato all'ultima cella (numero 80) e ho eseguito un'ulteriore chiamata ricorsiva, allora ho trovato una soluzione.

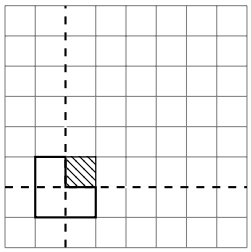
12.7 Puzzle di triomini

Immaginiamo di avere una griglia `n x n` con $n = 2^n$. Una posizione della griglia è identificata da un buco. Disponiamo solo di pezzi a "L" detti *triomini*

Riempire la griglia con i *triomini*

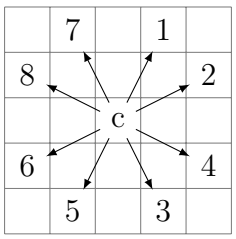
Questo problema potrebbe sembrare un problema di backtracking, ma è molto più semplice.

- L'idea è che si può dividere la griglia in 4
- Rimarrà dunque un buco in una delle 4 porzioni.
- Metto triomino in modo tale che sia a cavallo delle 3 porzioni senza buco
- Ho ottenuto ora 3 quadrati con lato `n/2` con un posto occupato. Posso iterare ricorsivamente fino ad aver riempito tutta la griglia



12.8 Knight tour

Considerando che le mosse che può fare un cavallo sono le seguenti:



si trovi un percorso di un cavallo su di una scacchiera che visiti ogni sua cella al massimo 1 volta.

- Salvo in `dp[i][j]` il passo a cui è stata visitata la cella `(i,j)` o 0 se non è stata ancora visitata
-
- Per ogni posizione ho al massimo 8 opzioni, ossia il numero di mosse possibili per un cavallo

Algoritmo 19: *Knight tour*

```
bool knightTour (int[][] S, int i, int x, int y):
    ▷ Se i = 64, ho fatto 63 mosse e ho completato un tour
      (aperto)
    ;
    if i == 64 then
        processSolution (S);
        return true;
    Set C = moves(S, x, y);
    foreach c ∈ C do
        S[x][y] = i;
        if knightTour (S, i + 1, x + m_x[c], y + m_y[c]) then
            return true;
        S[x][y] = 0;
    return false;
```

Algorithm 18: Knight's Tour (Backtracking)

Questo algoritmo esegue circa $8^{63} \approx 7.84 \cdot 10^{55}$. Tantini

12.9 Inviluppo convesso

Dato un insimee di punti sul piano cartesiano, trovare il più piccolo poligono convesso che li contenga tutti e quanti.

12.9.1 Algoritmo Naive

- Considero una retta per due punti a, b
- Se tutti i restanti punti sono dalla stessa parte della retta allora per forza a e b appartengono al poligono ($O(n)$)
- Ripeto per ogni coppia di punti ($O(n^2)$)

Complessità $O(n^3)$

12.9.2 Algoritmo di Jarvis

Un approccio più intelligente sta nell'immaginarsi di "arrotolare" un bastoncino intorno all'ammasso di punti.

- Considero il punto più a sinistra p
- Considero l'inclinazione di ogni retta passante per p e x ($O(n)$)
- Prendo la retta con angolo *minore rispetto alla verticale*
- Setto t al punto della retta selezionata prima e itero ($O(h)$, dove h è il numero di vertici del poligono finale)

Complessità $O(nh)$, dove h è il numero di vertici del poligono finale.

12.9.3 Algoritmo di Graham

- Seleziono punto con y minore. Lo chiamo p_1
- Ordino punti in base all'angolo che forma la retta per p_1 p rispetto all'orizzontale
- Inserisco p_1 e p_2 in uno stack.Chiamo p il punto corrente. Considero la retta formata fra gli ultimi due punti inseriti nello stack
  - Se p_1 e p stanno nello stesso semipiano delineato dalla retta inserisco p nello stack
  - Altrimenti elimino punti dallo stack e ripeto il controllo finché questo non è vero

Siccome ogni punto viene inserito e rimosso al più una volta dallo stack, il costo dell'algoritmo è $O(n)$. Tuttavia i punti vanno ordinati, dunque il costo finale è $O(n \log n)$.

Algoritmo 20: *Grahams inviluppo convesso*

```
Stack graham (point[] p, int n):
    Stack S = Stack();
    S.push(p_1); S.push(p_2);
    for i = 3 to n do
        while not same_side (S.top(), S.top2(), p_1, p_i) do
            S.pop();
        S.push(p_i);
    return S;
```

Algorithm 19: Graham's Scan Algorithm (Convex Hull - Stack Implementation)

13 Algoritmi probabilitstici

Possiamo distinguere tra due tipi di algoritmi probabilistici:

- Algoritmi di Montecarlo: il risultato è corretto solo una % delle volte
- Algoritmi di Las Vegas: il risultato è corretto sempre, ma il tempo di esecuzione è probabilistico

13.1 Test di primalità

Dato in input un numero n, determinare se p primo.

13.1.1 Varsione naif

Per ogni numero fino alla $\sqrt{n}$ controllo se è divisibile

Algoritmo 21: *Prime check naif*

```
bool isPrimeNaif (int n):
    for i = 2 to ⌊√n⌋ do
        if n/i == ⌊n/i⌋ then
            return false;
    return true;
```

Algorithm 20: Prime Check (Naive Implementation)

13.1.2 Versione di fermat

Il piccolo teorema di Fermat afferma che:

Teorema 11: *Piccolo teorema di Fermat*

Se un numero n è primo, allora:

$$\forall b \in [2, n) \quad b^{n-1} \mod n = 1$$

Algoritmo 22: *Prime check Fermat*

```
bool isPrimeFermat (int n):
    for i = 1 to k do
        b = random(2, n − 1);
        if b^{n−1} mod n ≠ 1 then
            return false;
    return true;
```

Algorithm 21: Probabilistic Prime Check (Fermat's Test)

Se questo algoritmo ritorna

- `false` allora n è sicuramente composto (*non primo*)
- `true` allora n può essere sia primo che non

Non ho garanzie sulla probabilità con cui azzecco il risultato

13.1.3 Algoritmo di Miller-Rabin

Teorema 12: *Teorema di Miller-Rabin*

Se n è primo e ho:

1. $b \in [2, n)$
2. $n - 1 = m \cdot 2^v$, m dispari.

alora per ogni b:

1. $\text{mcd}(n, b) = 1$
2. $b^m \mod n = 1 \vee \exists i, 0 \leq i < v : b^{m \cdot 2^i} \mod n = n - 1$

Nota come posso scrivere $m \cdot 2^v$ in binario. v è il numero di *trailing zeros* se scriviamo il numero in binario

$$\underbrace{1001011}_{m}\underbrace{000}_{v}$$

Di base per controllare mi basta verificare queste due condizioni. Se trovo un valore di b (*testimone*) per cui non è rispettata la 1 oppure non esiste un valore di i per cui è rispettata la 2 sono *sicuro* che il numero non sia primo. Queste sono

condizioni necessarie ma non sufficienti per la primalità

Algoritmo 23: *Prime check Miller-Rabin*

```
bool isPrime (int n):
    for i = 1 to k do
        b = random(2, n − 1);
        if isComposite(n, b) then
            return false;
    return true;
```

Algorithm 22: Probabilistic Prime Check

`isComposite(n,b)` esegue un check secondo il teorema descritto qui.

Si può dimostrare che se n è composto si esistono almento $\frac{3}{4} \cdot (n-1)$ testimoni. Quindi ho $\frac{1}{4}$ di possibilità di sbagliare. Eseguendo k passate ho $\frac{1}{4}^k$ possibilità di errare, ossia pochissime

Complessità: $O(k \log^2 n \log\log n \log\log\log n)$ (*fanculo la dimostrazione*)

13.2 Espressione polinomiale nulla

Teorema 13: *Annullamento polinomiale*

Se ogni v_i è un valore intero compreso casuale fra 1 e $2d$, dove d è il grado del polinomio, allora la probabilità di errore non supera $1/2$.

L'idea è di valutare k volte il polinomio in valori presi a caso.

- Se *si annulla*: il polinomio può essere identicamente nullo oppure no
- Se *non si annulla*: il polinomio *non* può essere identicamente nullo
- Probabilità di errore $\left(\frac{1}{2}\right)^k$

13.3 Bloom filter

Struttura dati probabilistica che funziona come hashmap ma eseguendo `contains(i)`:

- Se `i` *non è contenuto* → ritorna `false`
- Se `i` *è contenuto* → ritorna `true o false`

Le funzioni sono le seguenti:

- `insert(i)`: ho n funzioni hash. Per ogni funzione setto a 1 il bit corrispondente
- `contains(i)`: ritorna `true` se tutti i bit corrispondendi alle funzioni hash sono 1, false altrimenti

Si può dimostrare che la efficienza di questa struttura segue quanto descritto da queste formule:

- Dati n oggetti, m bit, k funzioni hash, la probabilità di un falso positivo è pari a:

$$\epsilon = \left(1 - e^{-kn/m}\right)^k$$

- Dati n oggetti e m bit, il valore ottimale per k è pari a:

$$k = \frac{m}{n} \ln 2$$

- Dati n oggetti e una probabilità di falsi positivi ϵ, il numero di bit m richiesti è pari a:

$$m = -\frac{n \ln \epsilon}{(\ln 2)^2}$$

13.4 Selezione naif

Algoritmo 24: *Selezione naif*

```
int naifSelect (Item[] A, int n, int k):
    sort (A);                                   ▷ O(n log n)
    return A[k];
```

Algorithm 23: Selection naif

Questo approccio ha complessità $O(n \log n)$

13.5 Selezione simil heapsort

Algoritmo 25: *Selezione simil Heap-sort*

```
int heapSelect (Item[] A, int n, int k):
    buildHeap (A);                                   ▷ O(n)
    for i = 1 to k − 1 do
        deleteMin (A, n);                            ▷ O(log)
    return deleteMin (A, n);
```

Algorithm 24: Heap Selection

Questo approccio ha complessità $O\left(n + k \log n\right)$

13.6 Selezione probabilistica

Posso avere un algoritmo probabilistico in stile *quicksort*.

- Scelgo un pivot a caso
- Metto alla sua sinistra tutti gli elementi $<$ pivot
- Chiamo ricorsivamente *solo* sulla parte di mio interesse (*è qua la differenza rispetto al quicksort*, che invece chiama l'algoritmo su entrambe)

Algoritmo 26: *Selezione probabilistica*

```
Item selection (Item[] A, int start, int end, int k):
    if start == end then
        return A[start];
    else
        int j = pivot(A, start, end);
        int q = j − start + 1;
        if k == q then
            return A[j];
        else if k < q then
            return selection (A, start, j − 1, k);
        else
            return selection (A, j + 1, end, k − q);
```

Algorithm 25: Selection Algorithm

La funzione `pivot` prende gli elementi in `[start, end]`, e mette tutti gli elementi `< v[start]` a sinistra e i rimanenti a destra. Di base usa `a[start]` come pivot e sposta gli elementi

13.6.1 Complessità

Assumiamo di dover sempre chiamare ricorsivamente `select` sulla porzione di vettore più lunga(caso pessimo). Assumiamo che `pivot()` restituisca con la stessa probabilità una qualsiasi posizione j del vettore A

$$T(n) = n + \underbrace{\frac{1}{n}\sum_{q=1}^{n} T(\max\{q-1, n-q\})}_{\text{media su tutti i casi possibili}}$$

Ad esempio, su $n = 4$ posso avere i seguenti modi di spezzare il vettore e la chiamata ricorsiva sarebbe effettuata sulla porzione in rosso:

$$\{0, 4\}, \{1, 3\}, \{2, 2\}, \{3, 1\}, \{4, 0\}$$

Quindi ogni chiamata è limitata superiormente dalla somma fino a $\frac{n}{2}$ diviso n

$$T(n) \leq n + \frac{1}{n}\sum_{q=\lfloor \frac{n}{2} \rfloor}^{n-1} 2T(q)$$

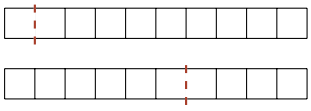
Posso procedere per sostituzione sostituzione

$$\begin{aligned}
T(n) &\leq n + \frac{1}{n}\sum_{q=\lfloor n/2 \rfloor}^{n-1} 2 \cdot cq \leq n + \frac{2c}{n}\sum_{q=\lfloor n/2 \rfloor}^{n-1} q && \text{Sostituzione, raccolgo } 2c \\
&= n + \frac{2c}{n}\left(\sum_{q=1}^{n-1} q - \sum_{q=1}^{\lfloor n/2 \rfloor - 1} q\right) && \text{Sottrazione prima parte} \\
&= n + \frac{2c}{n} \cdot \left(\frac{n(n-1)}{2} - \frac{(\lfloor n/2 \rfloor)(\lfloor n/2 \rfloor - 1)}{2}\right) \\
&\leq n + \frac{2c}{n} \cdot \left(\frac{n(n-1)}{2} - \frac{(n/2-1)(n/2-2)}{2}\right) && \text{Rimozione limite inferiore} \\
&= n + \frac{c}{n} \cdot \frac{(n^2 - n - (1/4n^2 - 3/2n + 2))}{1} \\
&= n + c/n \cdot \left(3/4n^2 + 1/2n - 2\right) \\
&\leq n + c/n \cdot \left(3/4n^2 + 1/2n\right) = n + 3/4cn + 1/2cn && \text{Vera per } c \geq 6 \text{ e } n \geq 1
\end{aligned}$$

Quindi la complessità media è $O(n)$

13.7 Selezione deterministica

L'idea è che se ho la certezza di scegliere il pivot in modo tale che non taglio solo il primo elemento, la formula ricorsiva è di



$T(n) = T(n-1)$

$T(n) = T\left(\frac{n}{k}\right)$

L'idea di base è che scegliendo un pivot che è la mediana fra le mediane dei gruppi da 5, ho la garanzia di avere $\frac{7}{10}n$ degli elementi a destra/a sinistra del pivot. Quindi non spezzero mai in stile $T(n) = T(n-1)$. La dimostrazione è questa:

- Ogni mediana di un blocco di 5 ha alla sua sinistra 3 interi minori (*inclusa la mediana stessa*)
- La mediana delle mediane ha alla sua sinistra esattamente $\frac{n}{5}/2$ mediane per lo stesso ragionamento di prima
- Quindi alla sinistra della mediana delle mediane ci devono essere

$$\text{elementi minori} = 3 \cdot \frac{n}{5}/2 = \frac{3}{10}n$$

- Assumo di selezionare sempre la parte peggiore, ossia la porzione larga $\frac{7}{10}n$

Quindi, siccome ad ogni chiamata ricorsiva avrò $T(n) = T\left(\frac{3}{7}n\right)$, l'algoritmo è lineare per il master theorem

Algoritmo 27: *Selezione deterministica*

```
Item select (Item[] A, int start, int end, int k):
    ▷ Sotto una certa dimensione trovo mediana tramite sort in O(1)
    if end − start + 1 ≤ 10 then
        InsertionSort (A, start, end) ▷ Versione con indici inizio/fine
        return A[start + k − 1];

    ▷ Divide array in sottovett di dim 5 e calcola la mediana
    int[] M = new int[1 ... ⌈n/5⌉];
    for i = 1 to ⌈n/5⌉ do
        M[i] = median5(A, start + (i - 1) · 5, end);

    ▷ Individua la mediana delle mediane e usala come perno
    Item m = select(M, 1, ⌈n/5⌉, ⌈⌈n/5⌉/2⌉);          ▷ Chiamata 1
    int j = pivot(A, start, end, m) ;      ▷ Usa m come elemento pivot

    ▷ Calcola l'indice q di m in [start...end]
    int q = j − start + 1;
    if q == k then
        return m;
    else if q < k then
        return select (A, start, q − 1, k);          ▷ Chiamata 2
    else
        return select (A, q + 1, end, k − q);        ▷ Chiamata 3
```

Algorithm 26: Selezione Deterministica

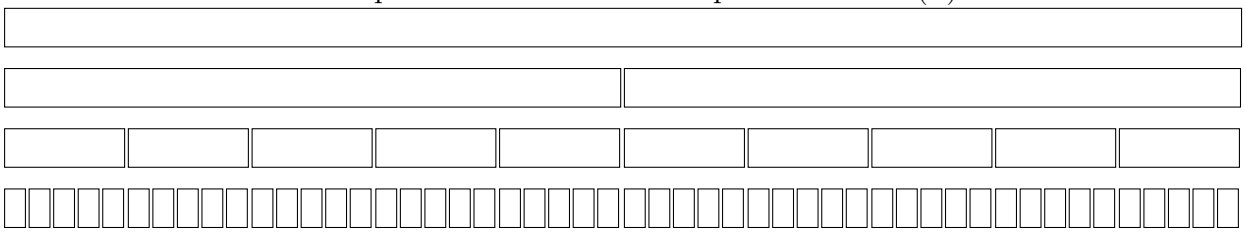
Complessità:

$$T(n) = \underbrace{T\left(\frac{n}{5}\right)}_{\text{chiamata 1}} + \underbrace{T\left(\frac{7}{10}n\right)}_{\text{chiamata 2 o 3}} + \underbrace{\frac{11}{5}n}_{\text{mediane pivot }*}$$

* il fattore deriva da

- $6 \cdot \frac{n}{5}$ per trovare mediane di blocchetti da 5
- n per eseguire l'operazione di `pivot` sull'array originale

Risolvendo la ricorrenza si può verificare che la complessità è di $O(n)$



14 Problemi np completi

Possiamo distinguere tra 3 tipi di problemi:

- Problemi di *ottimizzazione*: trova la soluzione migliore
- Problemi di *ricerca*: trova una possibile soluzione
- Problemi di *decisione*: verifica se una soddisfa certi prerequisiti

Nota bene che

Si sa risolve efficientemente un problema di *ottimizzazione*

$\Rightarrow$

Si sa risolve efficientemente un problema di *decisione*

14.1 Riduzione polinomiale

Definizione 19: *Riduzione polinomiale*

Se posso risolvere un problema P_1 con l'algoritmo risolutivo di un problema P_2, convertendo l'input/output di P_1 in tempo *polinomiale* si dice che P_1 è riducibile polinialmente a P_2

Vediamo ora una serie di problemi NP complessi che possono essere ridotti fra di loro

14.2 Colorazione dei grafi

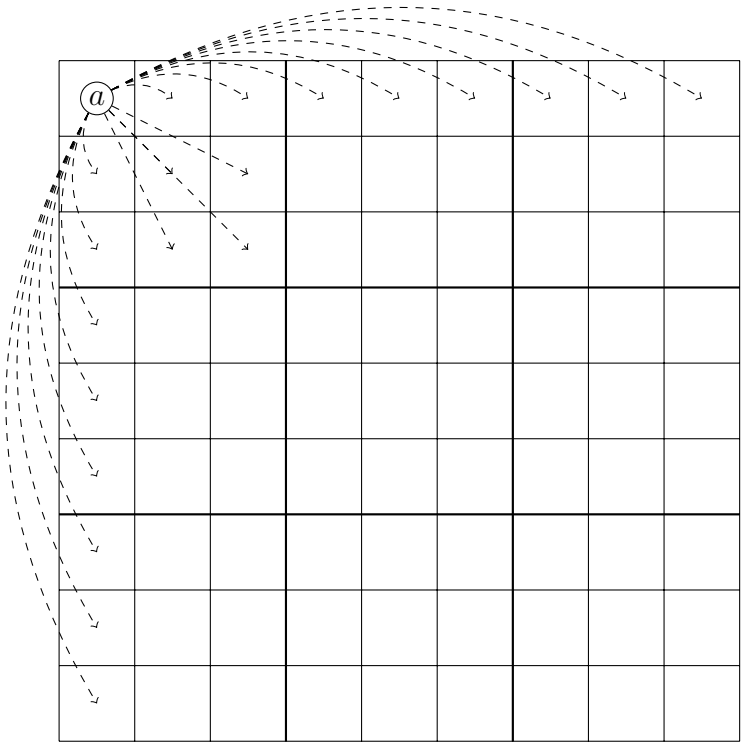
Dato un grafo non orientato colorare i suoi vertici con k colori in modo che non ci siano vertici adiacenti con lo stesso colore

- *Ottimizzazione*: trovare il numero minore di colori k
- *Decisione*: esiste una colorazione valida per k

14.3 Sudoku

Data una griglia $n^2 x n^2$ riempirla con numeri da $1 \dots n^2$ secondo le regole del sudoku

Il prolema del sudoku è *riducibile polinomialmente* al problema di colorazione dei grafi.



Supponendo di costruire un grafo in cui ogni cella del sudoku e collegata con le colonne e con gli elementi della cella interna, facendo il coloring del grafo non posso dare lo stesso colore (numero) a questi elementi

14.4 Insieme indipendente e vertex cover

Definizione 20: *Insieme indipendente*

Dato un grafo non orientato, un *insieme indipendete* è un insieme di vertici per cui non ce nessun *arco* fra due nodi di questo insieme

Definizione 21: *Vertex cover*

Dato un grafo non oeirnetato, una *vertex cover* è un insieme di vertici tale che ogni arco del grafo ha almeno un estremo in questo insieme

Teorema 14: *Dualità insieme indipendente e vertex cover*

Se S è un insieme indipendente e V l'indieme di vertici del grafo, allora $V - S$ è una vertex cover

14.4.1 Dimostrazione

- Sia S un insieme indipendente. Allora ogni altro arco o ha 1 estremo in $V - S$ se l'altro è in S, oppure li ha entrambi
- Sia S una vertex cover. Se $V - S$ non fosse indipendere allora ci sarebbe un arco con entrambi gli estremi in $V - S$. Questo arco tuttavia avendo entrambi gli estremi in $V - S$, non ne ha in S, quindi S non può essere vertex cover

14.5 Formule booleane in forma normale congiuntiva

Definizione 22: *Formule booleane in forma normale congiuntiva*

Una formula booleana in forma normale congiunta è un'insieme di letterali (che possono esser `true` o `false`). I letterali sono uniti da `and`($\wedge$) fra gruppi di `or` ($\vee$)

$$(x \vee \overline{y} \vee z) \wedge (\overline{x} \vee w) \wedge y$$

Il problema principale è quello della *satisfiability*:

- *Satisfiability*: esiste una combinazioni di valori che rendono la formula vera
- *3-satisfiability*: ogni clausula è data da esattamente 3 letterali (es $x \vee \overline{z} \vee z$)

Teorema 15: *Riducibilità polinomiale satisfiability*

Il problema di satisfiability è riducibile polinomialmente come segue:

$$\text{SAT} \leq_p \text{3-SAT} \leq_p \text{INDEPENDENT-SET} \leq_p \text{VERTEX-COVER}$$

14.5.1 Dimostrazione

SAT $\leq_p$ 3-SAT

- Se la clausola è più lunga di tre elementi, si introduce una nuova variabile e si divide la clausola in due:

  $$(a \vee b \vee c \vee d) \equiv (a \vee b \vee z) \wedge (\overline{z} \vee c \vee d)$$

  l'idea è che una variabile z fa da "switch", accendendo il blocco a sinistra o il blocco a destra
- Se la clausola è più corta di tre elementi, si fa "padding":

  $$(a \vee b) \equiv (a \vee a \vee b)$$

3-SAT $\leq_p$ INDEPENDENT-SET

- Se costruisco un grafo come mostrato qui
  - Collego ogni letterale con ogni corrispondente negato

- Collego ogni letterale con ogni letterale della stessa clausula

- L'idea è che se seleziono un letterale (lo rendo `true`), non posso più prenderne nella stessa clausula e non posso avverare i suoi negati

- Se ne trovo k, che sono per forza in k clausule separate, ho avverato ogni clausula

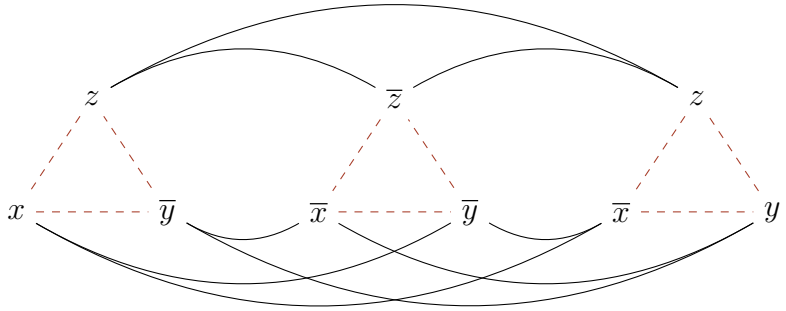


Figura 2: 3-sat graph

14.6 Classi di problemi

Definizione 23: *Classi di complessità*

Data una qualunque funzione $f(n)$, chiamiamo:

- $\mathbb{TIME}(f(n))$ l'insieme dei problemi decisionali risolvibili da un algoritmo che lavora in tempo $O(f(n))$

- $\mathbb{SPACE}(f(n))$ gli insiemi dei problemi decisionali risolvibili da un algoritmo che lavora in spazio $O(f(n))$.

Definizione 24: *Classi* $\mathbb{P}, \mathbb{PSPACE}$

Classe $\mathbb{P}$
La classe $\mathbb{P}$ è la classe dei problemi decisionali risolvibili in tempo polinomiale nella dimensione n dell'istanza di ingresso:

$$\mathbb{P} = \bigcup_{c=0}^{\infty} \mathbb{TIME}(n^c)$$

Classe $\mathbb{PSPACE}$
La classe $\mathbb{PSPACE}$ è la classe dei problemi decisionali risolvibili in spazio polinomiale nella dimensione n dell'istanza di ingresso:

$$\mathbb{PSPACE} = \bigcup_{c=0}^{\infty} \mathbb{SPACE}(n^c)$$

Nota che se un problema è $\mathbb{PSPACE}$ allora è anche $\mathbb{P}$. In qualche modo dovro "iterare" tutta la soluzione

Definizione 25: *Certificato*

Informalmente, un certificato è una "soluzione ammissibile" la cui ottimalità può essere dimostrata in tempo polinomiale.

Formalmente, dato un problema decisionale R e un suo input I che si sa essere vero, un certificato è un insieme di dati che permettono di provare che $(i, \texttt{true}) \in R$

Ad edempio un assegnamento di verità alle variabili della formula SAT

Definizione 26: *Classe* $\mathbb{NP}$

L'insieme di tutti i problemi che ammettono un certificato verificabile in tempo polinomiale.

Non tutti i certificati sono verificabili in tempo polinomiale. Ad esempio se prendo il problema di satisfiability e lo estendo con i quantificatori universali (invece di chiedermi se esista un valore di x che la avvera, mi chiedo se sia vera *per ogni* $\forall x$).

Definizione 27: *Problema NP-arduo (NP-hard)*

Un problema decisionale R si dice *NP-arduo* se ogni problema $Q \in \text{NP}$ è riducibile polinomialmente a R ($Q \leq_p R$).

Definizione 28: *Problema NP-completo (NP-complete)*

Un problema decisionale R si dice *NP-completo* se appartiene alla classe *NP* ed è *NP-hard*.

Dimostrare che un problema è $NP - completo$ è difficilissimo. Ad oggi abbiamo solo:

Teorema 16: *Teorema di Cook-Levin*

Cook Levin ha dimostrato che il problema di *SAT* $\mathbb{NP}$-completo

Definizione 29: *Dimensione di un problema*

La dimensione di un problema è il numero di bit necessari a rappresentare un'istanza del problema.

- d: *dimensione*, ossia numero di bit per rappresentare l'intero problema
- $\#$: valore *numerico* dell'intero *più grande* in input

Definizione 30: *Problema fortemente/debolmente NP-completo*

Sia R_{pol} la *versione ridotta* di un problema, ottenuta limitando la dimensione dell'intero maggiore # superiormente:

$$\# = O\left(n^c\right)$$

allora

- Se R_{pol} rimane $\mathbb{NP}$-completo, allora R è detto *fortemente NP-completo.*
- Se R_{pol} diventa di complessità polinomiale $\mathbb{P}$, allora R è detto *debolmente NP-completo.*

L'idea intuitiva è che se limitando la velocità con cui cresce l'input riusciamo ad ottenere complessità polinomiale, il problema è debolmente *NP-completo*

14.6.1 Esempio base

Supponi di avere un problema che prende in input un singolo numerale $k = 2^t$, con complessità $O\left(k\right)$. Se esprimo la complessità in funzione del *numero di bit* necessari per rappresentare l'input, allora

$$\text{complessità} = O\left(2^k\right)$$

se limito superiormente la dimensione dei numerali (k) il problema diventa polinomiale. In particolare, la dimensione dei numerali deve essere al più polinomiale nella dimensione del problema

$$k = O\left(t^c\right) \Rightarrow \text{complessità} = O\left(k\right) = O\left(O\left(t^c\right)\right) = O\left(t^c\right)$$

14.6.2 Subset sum

Complessità $O\left(nk\right)$. Debolmente *NP-completo*

se $k = O\left(n^c\right)$ allor

$$\text{Complessità} = O\left(n^c \cdot n\right) = O\left(n^{c+1}\right)$$

14.6.3 Clique

Complessità $O\left(n^2\right)$. Fortemente *NP-completo*

Limitare superiormente k non ha effetto in quanto è gia limitata superiormente. Se $k > n$ per forza la risposta è `true`

14.6.4 Esempi debolmente vs fortemente NP-completo

- *Subset sub*: limitando la dimensione di $k \to k = O\left(n^c\right)$ (la somma da raggiongere) ottengo complessità polinomiale

$$\text{Complessità} = O\left(nk\right) = O\left(n \cdot O\left(n^c\right)\right) = O\left(n^{c+1}\right)$$

- *Clique (cricca)*: limitare l'input k non ha effetto in quanto è già limitato superiormente da n (se $k > n$ per forza non vi sono soluzioni). La versione ridotta del problema è dunque uguale a quella non ridotta i

15 Soluzioni a problemi intrattabili

Spesso dobbiamo accontentarci di una soluzione approssimata per un problema di ottimizzazione.

Definizione 31: *$\alpha(n)$ approssimazione di un problema*

Un problema si dice essere $\alpha(n)$ approssimato se la soluzione dista al massimo $\alpha(n)$ dalla soluzione ottimale. Detto $c(n)$ il costo della soluzione trovata e $c(x^*)$ il costo della soluzione ideale, allora

$$c(x) \leq c(x^*) \cdot \alpha(n)$$

15.1 Bin packing

Supponiamo di avere un set di `n` oggetti ognuno con un peso intero `A[i]`. Abbiamo a disposizione scatole con capienza `k`. Trovare il numero minimo di scatole necessarie

15.1.1 Approccio first fit

Scorro ogni oggetto in `A` e lo metto nella prima scatola che può contenerlo. Siccome ho che:

- Nel migliore dei casi ho le scatole tutte completamente piene
- Nel peggiore del casi ho le scatole vuote "appena meno" che metà
- La soluzione approssimata è al più 2 vole più costosa di quella ottima

$$\alpha(n) = 2$$

Si potrebbe dimostrare un limite più stretto

$$N < \frac{17}{10}N^* + 2$$

Se ordiniamo gli oggetti in modo decrescente si ottiene (*non dimostrato*)

$$N < \frac{11}{9}N^* + 4$$

15.2 TSP con disuguaglianze triangolari

Possiamo modificare il problema del commesso viaggiatore imponendo che per ogni arco (i, j)

$$w(i, j) \leq w(i, k) + w(k, j)$$

l'idea è che se ho una strada da i a j, non mi converrrà mai andare da i a k e poi da k a j, quindi è un problema sensato dal punto di vista della realtà.

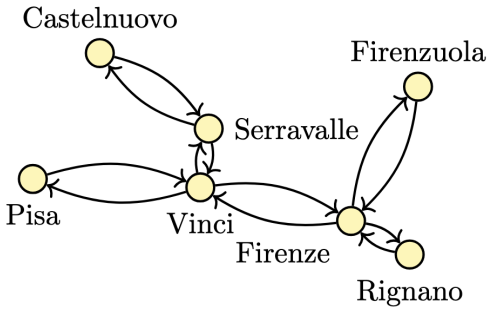
15.2.1 Soluzione 2-approssimata

Servono due piccoli teoremini:

- Un albero di copertura minimo mst ha sempre peso inferiore al ciclo Hamiltoniano minore π.
  - Supponiamo per assurdo che $w(\pi) \leq w(mst)$
  - Togliendo un arco a π ottengo un albero di copertura con peso minore del msp, insensato
- Un grafo che rispetta la disuguaglianza triangolare è *completo*. Dimostrabile facilmente per induzione

Quindi procedo così

- Calcolo albero di copertura minimo
- Prendo un percorso che percorre ogni arco di questo albero 2 volte. Questo ha costo $2 \cdot c(mst)$



- Parto da un nodo a caso. Seguo archi dell'albero fino a quanto tornerei su nodo visitato.
- Ogni volta che finisco su un nodo visitato lo salto e continuo a seguire gli archi dell'albero
- Per i due teoremini visti prima,

$$c(\pi) < 2 \cdot c(mst) < 2 \cdot c(\pi^*)$$

dove

$$\underbrace{c(\pi) < 2 \cdot c(mst)}_{\text{per disuguaglianza triangolare}} \qquad \underbrace{2 \cdot c(mst) < 2 \cdot c(\pi^*)}_{\text{per teorema mst}}$$

Nota che è dimostrabile che non esiste un'approssimazione migliore di 2 per TSP generico. Per Δ-tsp è possibile un'approssimazione di $\frac{3}{2}$

15.3 TSP euristico

15.3.1 Shortest edge first

- Ordino archi in modo decrescente
- Per ogni arco, lo prendo se e solo se
  - Non forma un ciclo
  - I suoi estremi sono nodi con selected out degree < 2, ossia che siano connessi ad al più un edge fra quelli selezionati
- Alla fine connetto i 2 nodi con selected out degree 1

Complessità: $O(n^2 \log n)$

15.3.2 Nearest neighbor

- Parto da un nodo a caso
- Seleziono la prossima città non visitata più vicina

Complessità: $O(n^2)$

15.4 TSP branch and bound

Possiamo applicare una strategia branch and bound con un pruning aggressivo per risolvere il TSP-problem. L'idea del branch and bound è quella di

- Tengo traccia globalmente della soluzione migliore `minCost` scoperta fin'ora
- Ad ogni step di decisione, calcolo il limite inferiore del prezzo necessario per concludere la soluzione. Lo chiamo `lb` (*lower bound*).
- Continuo a generare le combinazioni del sotoalbero se e solo se `lb < minCost`

Quindi il trucco sta nel calcolare il limite inferiore del costo di un albero (`lb`). In tsp possiamo

- Nella soluzione parziale S abbiamo un path.
- Dobbiamo passare per tutti i rimanenti nodi.
- Nel migliore dei casi, passerò per ogni nodo, entrando per l'arco di peso minore e uscendo per l'arco di peso minore. Chiamo questa quantità `transfer[h]`
- A questo costo c'è da aggiungere il costo per uscire dall'ultimo nodo di S e entrare nel suo primo. Il costo totale del *lower bound* sarà dunque:

$$\text{lb}(d, S, i) = \text{cost}[i] + \left\lceil \frac{\text{out} \ + \sum_{h \notin S} \text{transfer}[h] + \ \text{last}}{2} \right\rceil$$

16 Dimostrazioni da sapere

- Master Theorem
- Analisi ammortizzata vettori dinamici
- Limite altezza alberi red black
- Complessità costruzione Heap
- Limite altezza utilizzando euristica sul rango
- Correttezza LCS
- Compressione di huffman
- Teorema di Bellman i
- Selezione probabilistica i