Argomenti avanzati

Cammini minimi

Cammini minimi

- Si vuole ottenere il cammino minimo tra due nodi (es. problema distanze tra stazioni ferroviarie)

Ingresso: G = (V, E) orientato, funz. di peso w : E $\rightarrow$ R

- Peso di un cammino: $p = \langle v_0, v_1, \ldots, v_k \rangle$

$$W(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- peso cammino minimo da u a v:

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \xrightarrow{p} v\}, \text{se esiste cammino p da u a v} \\ \infty, \text{altrimenti} \end{cases}$$

Cammini minimi (2)

- Ok pesi negativi
  - affinché il cammino minimo resti ben definito non si ammettono cicli negativi! (come mai?)
- Inoltre calcoliamo i cammini minimi da un nodo fissato *s* (detto *sorgente*) (anch'esso fornito in ingresso)

Uscita: $\forall v \in V$:

- $d[v] = \delta(s,v)$
  - all'inizio $d[v] = \infty$
  - viene progressivamente ridotto, però sempre con $d[v] \geq \delta(s,v)$
  - d[v] viene anche detto *stima di cammino minimo* (dalla sorgente s a v)
  - Pi[v] = predecessore di v nel cammino min da s
    - se non esiste: NIL

Rilassamento

Rilassamento di un lato (u,v):

- se d[v] > d[u]+w(u,v) allora

  d[v] := d[u]+w(u,v) e

  Pi[v] := u

- (cioè vediamo che "costa di meno" passare per il lato (u,v))

```
RELAX(u, v, adj, d, Pi)
if d[v] > d[u] + adj[u][v]
  d[v] := d[u] + adj[u][v]
  Pi[v] := u
```

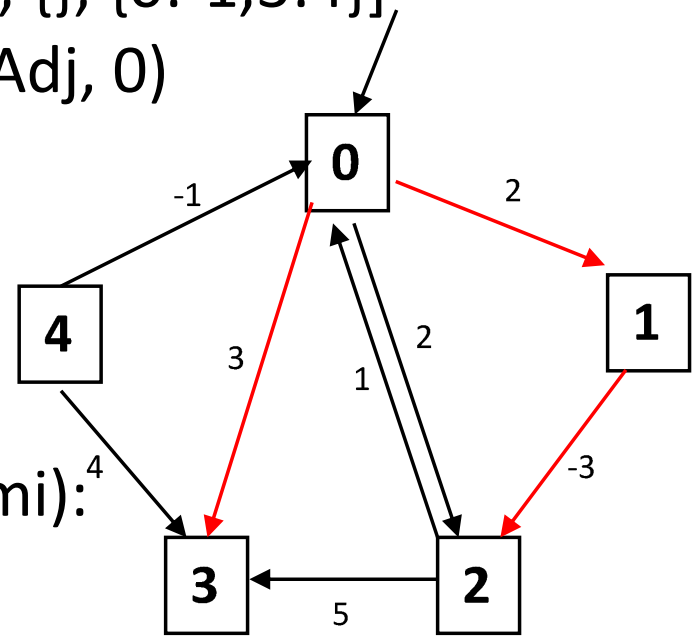
Algoritmo di Bellman-Ford

```
BELLMAN-FORD(adj, s) {
  V := vettore dei nodi di adj
  alloca d e Pi (dimensione adj.length)
  for i:=0 to adj.length-1
    d[i]:= ∞; Pi[i]:=NIL
  d[s]:=0
  for i:=1 to adj.length-1 // ripete |V|-1 volte
    for each u in V
      for each v in adj[u]
        RELAX(u, v, adj, d, Pi)
  return (d, Pi)
}
```

In pratica rilasso, un passo alla volta, partendo da s. Ad ogni passo avanzo di un passo nei miei cammini. Al |V|-1-esimo passo sicuramente avrò toccato tutti i nodi raggiungibili, perché tale è la massima lunghezza di un cammino aciclico (chiaramente non funziona se ci sono cicli negativi…)

Esempio

Adj = [{1:2, 2:2, 3:3}, {2:-3}, {0:1, 3:5}, {}, {0:-1,3:4}]
Sia (d, Pi) il risultato di bellmanFord(Adj, 0)



Distanze (4 è irraggiungibile):
d: [0,2,-1,3, ∞]

Predecessori (codifica cammini minimi):
Pi: [NIL, 0, 1, 0, NIL]

Costo: ciclo su tutte le adiacenze nidificato in un ciclo su |V|; `RELAX` ha costo costante => Θ(|V| |E|)

Programmazione dinamica

Programmazione dinamica

- Come la tecnica divide-et-impera, si basa sull'idea di scomporre il problema in sottoproblemi, risolverli e ricombinarne le soluzioni
  - si applica però quando i problemi non sono indipendenti, cioè condividono dei sottoproblemi
  - quando si risolve un sottoproblema comune, si mette la soluzione in una tabella, per riutilizzarla in seguito
    - il termine “programmazione” qui non si riferisce alla codifica in linguaggi di programmazione, ma al fatto che è una tecnica tabulare
- Programmazione dinamica spesso usata per problemi di ottimizzazione
  - la “soluzione” è un ottimo del sottoproblema
    - un problema potrebbe avere più soluzioni ottime

Passi tipici di programmazione dinamica

- caratterizzare la struttura delle soluzioni ottimali
- definire ricorsivamente il valore di una soluzione ottimale del problema
- calcolare una soluzione ottimale in modo bottom-up
  - dai sottoproblemi più semplici a quelli più difficili, fino al problema originario
- costruire una soluzione ottimale del problema richiesto

Problema: taglio delle aste

- Il prezzo di un'asta di acciaio dipende dalla sua lunghezza
- problema: date delle aste di lunghezza *n* che posso tagliare in pezzi più corti, devo trovare il modo ottimale di tagliare le aste per massimizzare il ricavo che posso derivare dalla vendita delle aste
- il ricavo massimo lo potrei avere anche non tagliando l'asta, e vendendola intera
- Esempio di tabella dei prezzi:

lunghezza i	1	2	3	4	5	6	7	8	9	10
prezzo p_i	1	5	8	9	10	17	17	20	24	30

lunghezza i	1	2	3	4	5	6	7	8	9	10
prezzo p_i	1	5	8	9	10	17	17	20	24	30

- per esempio, un'asta di lunghezza 4 può essere tagliata in tutti i modi seguenti (tra parentesi il prezzo):

  [4](9), [1,3](9), [2,2](10), [3,1](9), [1,1,2](7), [1,2,1](7), [2,1,1](7), [1,1,1,1](4)

  - il taglio ottimale in questo caso è unico, ed è [2,2]
- Data un'asta di lunghezza *n*, ci sono 2^{n-1} modi di tagliarla
  - ho *n*-1 punti di taglio possibili
  - se indico con una sequenza di *n*-1 0 e 1 la decisione di tagliare o no ai vari punti di taglio, ogni sequenza corrisponde ad un numero binario, e con *n*-1 cifre binarie posso rappresentare fino a 2^{n-1} valori
    - per esempio, per un'asta lunghezza 4, la decisione di non tagliare è data da 000; la decisione di tagliare solo a metà è data da 010, ecc.
- Chiamiamo r_n il ricavo massimo ottenibile dal taglio di un'asta di lunghezza *n*
- per esempio, dati i prezzi della tabella di cui sopra abbiamo r_4 = 10, mentre r_{10} = 30 (derivante da nessun taglio)

Sottostruttura ottima

- Per un qualunque n, la forma di r_n è del tipo r_i+r_{n-i}
  - a meno che l'ottimo preveda di non tagliare l'asta; in questo caso abbiamo $r_n = p_n$, il prezzo dell'asta intera
- In altre parole,
  $r_n = \max(p_n, r_1+r_{n-1}, r_2+r_{n-2}, \ldots, r_{n-1}+r_1)$
- Quindi, l'ottimo è dato dalla somma dei ricavi ottimi derivanti dalle 2 semiaste ottenute tagliando l'asta in 2
  - l'ottimo incorpora cioè i 2 ottimi delle soluzioni dei 2 sottoproblemi
- Per forza è così: se non fosse vero che r_i ed r_{n-i} sono gli ottimi dei rispettivi sottoproblemi, allora, sostituendo per esempio ad r_i una soluzione ottima del taglio di un'asta di lunghezza i otterremmo un ricavo totale $< r_n$, che non potrebbe essere più ottimo

- Quando la soluzione di un problema incorpora le soluzioni ottime dei suoi sottoproblemi, che si possono risolvere indipendentemente, diciamo che il problema ha una *sottostruttura ottima*
- Riformulando l'espressione dell'ottimo r_n, inoltre, possiamo fare dipendere r_n dall'ottimo di un solo sottoproblema:
  - r_n = prezzo del primo pezzo tagliato + taglio ottimo della restante asta, cioè $r_n = p_i + r_{n-i}$
    - ciò vale anche nel caso particolare in cui l'asta non va tagliata; in questo caso è $r_n = p_n + r_0$, con $r_0 = 0$
- Quindi, $r_n = \max_{1 \le i \le n}(p_i + r_{n-i})$

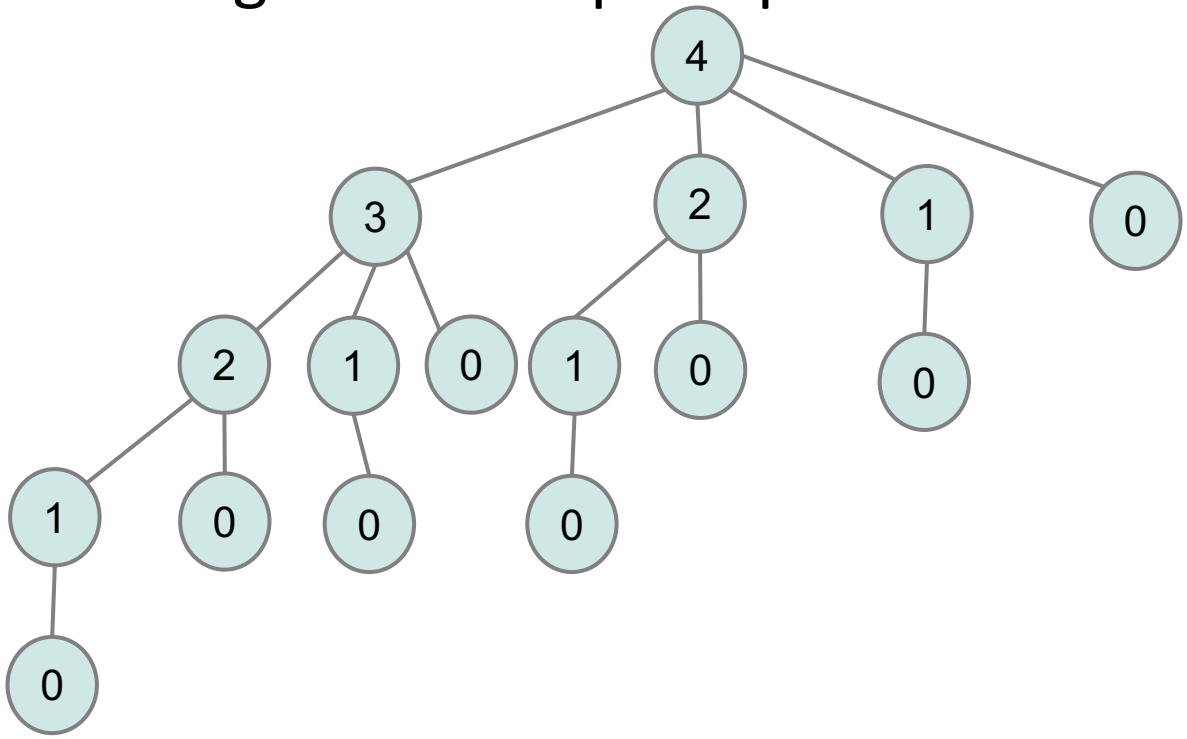
Algoritmo ricorsivo

- Applicando l'espressione ricorsiva appena vista della soluzione del problema del taglio delle aste otteniamo il seguente algoritmo

```
CUT-ROD(p,n)
if n = 0
  return 0
q := -∞
for i := 1 to n
  q := max(q, p[i] + CUT-ROD(p,n-i))
return q
```

- Complessità temporale: $T(n) = c + \sum_{j=0}^{n-1} T(j)$

- si può verificare che è $T(n) = \Theta(2^n)$
- Il tempo di esecuzione è così alto perché gli stessi problemi vengono risolti più e più volte:



Algoritmo di programmazione dinamica

- Usando un po' di memoria extra, si riesce a migliorare di molto il tempo di esecuzione
  - addirittura diventa polinomiale
  - trade-off spazio-temporale: aumento la complessità spaziale, riducendo quella temporale
- Idea: memorizzo il risultato dei sottoproblemi già calcolati, e quando li reincontro, invece di ricalcolarli, mi limito ad andare a prendere il risultato dalla tabella
  - risolvo ogni problema distinto una volta sola
  - il costo diventa polinomiale se il numero di problemi distinti da risolvere è polinomiale, e la risoluzione dei singoli problemi richiede tempo polinomiale

Approcci top-down e bottom-up per la programmazione dinamica

- Nel metodo **top-down**, comincio a risolvere il problema di dimensione *n*, e ricorsivamente vado a risolvere i sottoproblemi via via più piccoli; aumento però l'insieme dei parametri passati con una tabella nella quale memorizzo i risultati già calcolati
  - prima di lanciare la ricorsione sul problema più piccolo, controllo nella tabella se non ho già calcolato la soluzione
  - questa tecnica va sotto il nome di **memoization**
- Nel metodo bottom-up, parto dai problemi più piccoli, e li risolvo andando in ordine crescente di dimensione; quando arrivo a risolvere un problema di dimensione *i*, ho già risolto tutti i problemi di dimensioni < *i*

Versione memoized di CUT-ROD

```
MEMOIZED-CUT-ROD(p, n)
crea un nuovo array r[0..n]
for i := 0 to n
  r[i] := -∞
return MEMOIZED-CUT-ROD-AUX(p,n,r)

MEMOIZED-CUT-ROD-AUX(p,n,r)
if r[n] ≥ 0
  return r[n]
if n = 0
  q := 0
else q := -∞
  for i := 1 to n
    q := max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p,n-i,r))
r[n] := q
return q
```

Complessità di `MEMOIZED-CUT-ROD`

- `MEMOIZED-CUT-ROD` ha complessità $T(n) = \Theta(n^2)$: ogni sottoproblema è risolto una volta sola, ed il ciclo 6-7 fa n iterazioni per risolvere un problema di dimensione n
- quindi si fanno in tutto $n + (n-1) + (n-2) + \ldots + 1$ iterazioni

Bottom-up `CUT-ROD`

- Gli algoritmi `CUT-ROD` visti fino ad ora restituiscono il massimo ricavo, ma non il modo in cui l'asta va tagliata
- teniamo traccia non solo del massimo, ma del modo di effettuare il taglio con l'array s
  - s[j] mi dice quale è la lunghezza del primo pezzo nel taglio ottimale di un'asta di lunghezza j
- Esempio di risultato di un'esecuzione:

i	0	1	2	3	4	5	6	7	8	9	10
p_i	0	1	5	8	9	10	17	17	20	24	30
r[i]	0	1	5	8	10	13	17	18	22	25	30
s[i]	0	1	2	3	2	2	6	1	2	3	10

```
BOTTOM-UP-CUT-ROD(p, n)
crea r[0..n] e s[0..n]
r[0] := 0
for j := 1 to n
  q := -∞
  for i := 1 to j
    if q < p[i] + r[j-i]
      q := p[i] + r[j-i]
      s[j] := i
  r[j] := q
return (r,s)
```

è facile vedere che BOTTOM-UP-CUT-ROD *ha complessità* $T(n) = \Theta(n^2)$ *a causa dei 2 cicli annidati*

```
PRINT-CUT-ROD-SOLUTION(p, n)
(r,s) := BOTTOM-UP-CUT-ROD(p,n)
while n > 0
  print s[n]
  n := n - s[n]
```

Algoritmi golosi

Algoritmi golosi

- Per quanto con la programmazione dinamica un sottoproblema non venga risolto più di una volta, comunque occorre analizzare diverse soluzioni per decidere quale è l'ottimo
- A volte però non serve provare tutte le soluzioni: è dimostrabile che una sola può essere quella ottima
- Questo è esattamente quel che succede negli algoritmi “golosi” (*greedy*)

Ottimi locali e ottimi globali

- In generale, gli algoritmi golosi si muovono per “ottimi locali”:
  - se vedono davanti a loro una strada “promettente”, la prendono senza farsi troppi problemi (in questo senso sono golosi)
- Ovviamente questo può portare a soluzioni non ottime a livello globale, ma in alcuni casi si riesce a dimostrare che si raggiunge l'ottimo
- Spesso vengono usati in problemi difficili, in cui l'ottimo locale rappresenta una “buona approssimazione” dell'ottimo globale

Il problema della scelta delle attività

- n attività $a_1, a_2, ..., a_n$ usano la stessa risorsa
  - es: lezioni da tenere in una stessa aula
- Ogni attività a_i ha un tempo di inizio s_i ed un tempo di fine f_i con $s_i < f_i$
- a_i occupa la risorsa nell'intervallo temporale semiaperto $[s_i, f_i)$
- a_i ed a_j sono compatibili se $[s_i, f_i)$ e $[s_j, f_j)$ sono disgiunti
- voglio scegliere il massimo numero di attività compatibili
  - supponiamo che le attività $a_1, a_2, ..., a_n$ siano ordinate per tempo di fine non decrescente $f_1 \leq f_2 \leq ... \leq f_n$ altrimenti le ordiniamo in tempo $O(n \log n)$

Esempio

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- Insieme di attività compatibili: $\{a_3, a_9, a_{11}\}$
- massimo numero di attività compatibili: 4
  - un esempio: $\{a_2, a_4, a_9, a_{11}\}$

Soluzione

- Possiamo risolvere il problema con un algoritmo di programmazione dinamica
- Definiamo S_{ij} come l'insieme delle attività che iniziano dopo la fine di a_i e terminano prima dell'inizio di a_j
  - quindi che sono compatibili con a_i (e quelle che non terminano dopo a_i) e con a_j (e quelle che iniziano non prima di a_j)
  - $S_{ij} = \{a_t \in S : f_i \leq s_t < f_t \leq s_j \}$
- Chiamiamo A_{ij} un insieme massimo di attività compatibili in S_{ij}
  - supponiamo che a_k sia una attività di A_{ij}, allora A_{ij} è della forma: $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$
    - è fatto dell'ottimo del sottoproblema S_{ik} più l'ottimo del sottoproblema S_{kj}
      - se così non fosse, allora potrei trovare un insieme di attività più grande di A_{ik} (resp. A_{kj}), in S_{ik}, il che vorrebbe dire che A_{ij} non è un ottimo di S_{ij} (assurdo)
    - questa è dunque una sottostruttura ottima

- Memorizziamo in una tabella c la dimensione dell'ottimo del problema A_{ij}, cioè $c[i, j] = |A_{ij}|$
- Allora abbiamo che $c[i, j] = c[i, k] + c[k, j] + 1$
- Se non sappiamo che la soluzione ottima include l'attività a_k, dobbiamo provare tutte le attività in S_{ij}, cioè

$$c[i, j] = \begin{cases} 0 & \text{se } S_{ij} = \emptyset \\ \max\limits_{a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & \text{se } S_{ij} \neq \emptyset \end{cases}$$

- esercizio per casa: scrivere gli algoritmi di programmazione dinamica con memoization e con tecnica bottom-up

Algoritmo goloso

- E' inutile però provarle tutte per risolvere il problema S_{ij}: è sufficiente prendere l'attività a_1 che finisce per prima in S_{jj}, e risolvere il problema S_{kj}, con k prima attività in S_{ij} che inizia dopo la fine di a_1
  - se chiamiamo S_k l'insieme di tutte le attività che iniziano dopo la fine di a_k, cioè $S_k = \{a_t \in S : f_k \leq s_t\}$, dopo che abbiamo preso a_1, ci rimane da risolvere il solo problema S_1
- Abbiamo il seguente risultato:
  dato un sottoproblema S_k, se a_m è l'attività che finisce per prima in S_k, a_m è inclusa in qualche sottoinsieme massimo di attività mutuamente compatibili di S_k
  - supponiamo che A_k sia un sottoinsieme massimo di S_k, e chiamiamo a_j l'attività che finisce per prima in A_k; allora o $a_j = a_m$, oppure $f_m \leq f_j$, e se sostituisco a_j con a_m in A_k ho ancora un sottoinsieme massimo A'_k di S_k
- Quindi, per risolvere il problema di ottimizzazione mi basta ogni volta scegliere l'attività che finisce prima, quindi ripetere l'operazione sulle attività che iniziano dopo quella scelta

Versione ricorsiva

- *s* e *f* sono array con, rispettivamente, i tempi di inizio e di fine delle attività
- *k* è l'indice del sottoproblema S_k da risolvere (cioè l'indice dell'ultima attività scelta)
- *n* è la dimensione (numero di attività) del problema originario

```
RECURSIVE-ACTIVITY-SELECTOR(s,f,k,n)
m := k + 1
while m ≤ n and s[m] < f[k]
  m := m + 1
if m ≤ n
  return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s,f,m,n)
else return ∅
```

Versione iterativa

```
GREEDY-ACTIVITY-SELECTOR(s,f)
n := s.length
A := {a_1}
k := 1
for m := 2 to n
  if s[m] ≥ f[k]
    A := A ∪ {a_m}
    k := m
return A
```

- Si assume anche qui che le attività siano ordinate in modo crescente rispetto ai tempi di fine
- Entrambe le versioni hanno complessità $\Theta(n)$, in quanto considerano ogni attività una volta sola

Complessità e non determinismo

Complessità e non determinismo

- Esiste una sorta di “classe universale di complessità”?
  - cioè esiste una qualche funzione di complessità T(n) tale che tutti i problemi risolvibili impiegano al più T(n)?
- Il non determinismo può cambiare la complessità di soluzione dei problemi?
  - in primis, come si definisce la complessità di un modello non deterministico?
- Cominciamo con alcune definizioni, per fissare le idee

DTIME e DSPACE

- Data una funzione T(n), indichiamo con DTIME(T) l'insieme dei problemi tali che esiste un algoritmo che li risolve in tempo T(n)
- Più precisamente:
  - problema = riconoscimento di un linguaggio
    - per semplicità, consideriamo i linguaggi ricorsivi
  - algoritmo = macchina di Turing
- Riformulando: DTIME(T) (risp. DSPACE(T)) è la classe (l'insieme) dei linguaggi (ricorsivi) riconoscibili in tempo (risp. spazio) T mediante macchine di Turing *deterministiche* a k nastri di memoria

Un primo risultato

- Data una funzione totale e computabile T(*n*), esiste un linguaggio ricorsivo che non è in DTIME(T)
  - c'è quindi una *gerarchia* di linguaggi (problemi) definita sulla base della complessità temporale deterministica
  - una cosa analoga vale per DSPACE, e per le computazioni nondeterministiche (NTIME ed NSPACE)
  - a proposito...

Computazioni non deterministiche

- Data una macchina di Turing non deterministica M, definiamo la sua complessità temporale $T_M(x)$ per riconoscere la stringa x come la lunghezza della computazione più breve tra tutte quelle che accettano x
  - $T_M(n)$ poi è (nel caso pessimo) il massimo tra tutti i $T_M(x)$ con
    $|x| = n$
- Quindi NTIME(T) è la classe (l'insieme) dei linguaggi (ricorsivi) riconoscibili in tempo T mediante macchine di Turing ***non*** *deterministiche* a k nastri di memoria

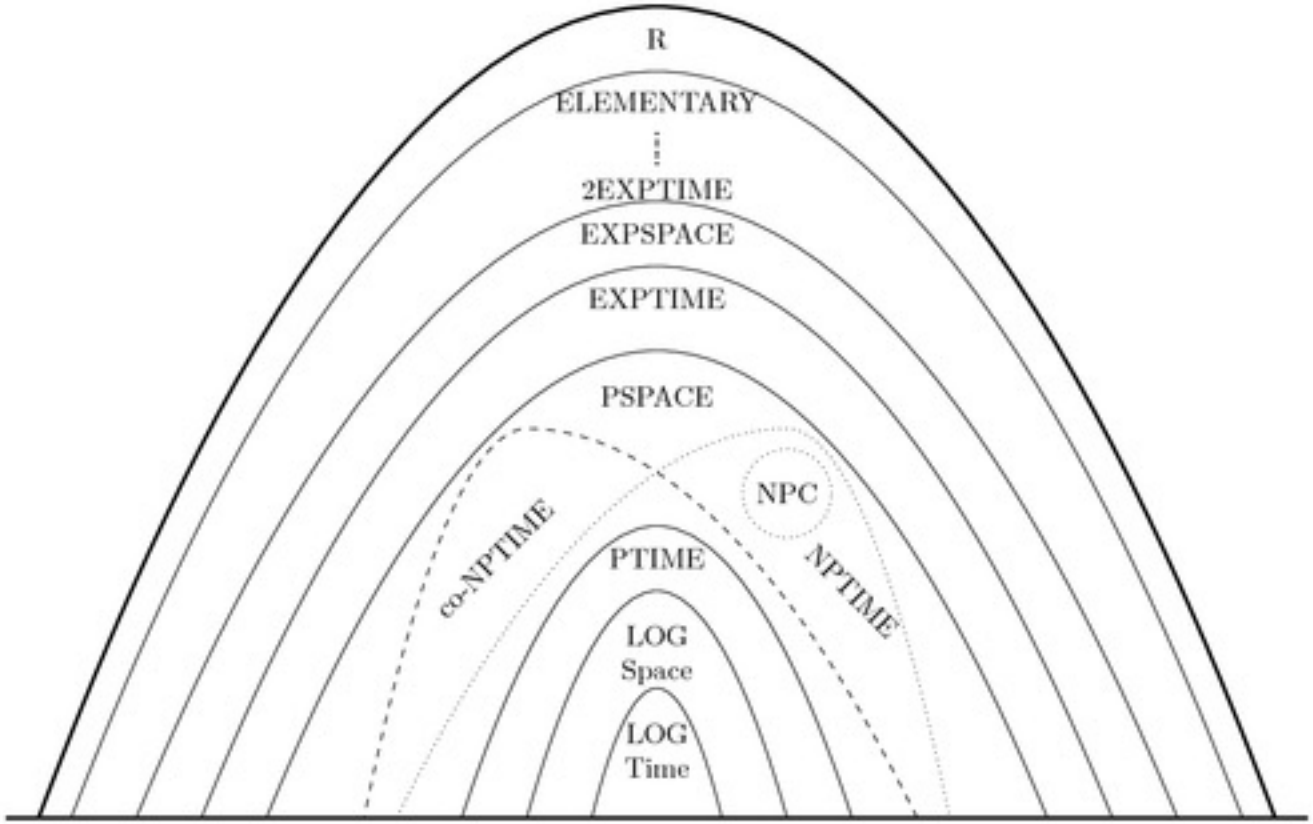
- Moltissimi problemi si risolvono in modo molto naturale mediante meccanismi non deterministici (per esempio, trovare un cammino in un grafo che tocca tutti i nodi)...
- ... però gli attuali meccanismi di calcolo sono deterministici
- se riuscissimo a trovare una maniera “poco onerosa” per passare da una formulazione non deterministica ad una deterministica, moltissimi problemi interessanti potrebbero essere risolti (in pratica) in modo (teoricamente) efficiente
  - però spesso abbiamo notato una “esplosione” nel passaggio da un meccanismo ND ad uno D (quando i 2 meccanismi sono equipotenti, come è peraltro il caso delle MT)
    - per esempio, esplosione del numero degli stati nel passare da NFA a FSA deterministici

Relazione tra DTIME e NTIME

- Sarebbe utile poter determinare, date certe interessanti famiglie di funzioni di complessità, se la classe dei problemi risolvibili non cambia nel passare da computazioni deterministiche a quelle non deterministiche
  - in altri termini, se DTIME($\mathcal{F}$) = NTIME($\mathcal{F}$) per certe famiglie $\mathcal{F}$={T_i}, di funzioni
- Una fondamentale classe di problemi:

  $P = \cup_{i \geq 1} \mathrm{DTIME}(O(n^i))$
  - convenzionalmente, questi sono considerati i problemi "trattabili"
- Similmente: $NP = \cup_{i \geq 1} \mathrm{NTIME}(O(n^i))$
- Altre classi interessanti di problemi: PSPACE, NPSPACE, EXPTIME, NEXPTIME, EXPSPACE, NEXPSPACE

Gerarchia di complessità



- Vedi anche lo "zoo" delle complessità: https://complexityzoo.net/Complexity_Zoo

P vs NP

- La domanda: P = NP?
  - probabilmente no, ma non si è ancora riusciti a dimostrarlo
- Alcuni esempi di problemi della classe NP:
  - Soddisfacibilità di formule di logica proposizionale (SAT): data una formula F di logica proposizionale, esiste un assegnamento dei valori alle lettere proposizionali che compaiono in F tale che F sia vera?
    - detto in altro modo: F ammette un modello?
  - Circuito hamiltoniano (HC): dato un grafo G, esiste un cammino in G tale che tutti i nodi del grafo sono toccati una ed una sola volta prima di tornare al nodo di partenza?

Riduzione in tempo polinomiale e completezza

- Un linguaggio (problema) L_1 è riducibile in tempo polinomiale ad un altro linguaggio L_2 se e solo se esiste una MT deterministica (traduttrice) con complessità in P che per ogni *x* produce una stringa $\tau(x)$ tale che $\tau(x) \in L_2$ se e solo se $x \in L_1$
- Se $\mathcal{L}$ è una classe di *linguaggi*, diciamo che un linguaggio L (che non è detto che debba essere in $\mathcal{L}$) è **$\mathcal{L}$-difficile** rispetto alle riduzioni in tempo polinomiale se e solo se, per ogni $L' \in \mathcal{L}$, L' è riducibile in tempo polinomiale a L
  - cioè se risolvere L (determinare se una stringa *x* appartiene ad L o no) è almeno tanto difficile quanto risolvere un qualunque linguaggio in $\mathcal{L}$
- Un linguaggio L è **$\mathcal{L}$-completo** se è $\mathcal{L}$-difficile ed è in $\mathcal{L}$

Conclusioni…

- Se si trovasse un problema NP-completo che è risolvibile in tempo polinomiale (da una MT deterministica!), allora avremmo P = NP
- Dualmente, se si trovasse un problema NP-completo che non è risolvibile in tempo polinomiale, allora avremmo $P \subset NP$
- SAT è NP-difficile
  - quindi è NP-completo (teorema di Cook-Levin)
  - si mostra codificando le computazioni di una generica MT non deterministica M (con complessità polinomiale) in SAT, in modo che M accetta una stringa *x* se e solo se una opportuna formula *s* è soddisfacibile
- HC è anch'esso NP-difficile (e NP-completo)
  - NP-completezza di HC si mostra riducendo SAT a HC
- Moltissimi altri problemi sono NP-completi...

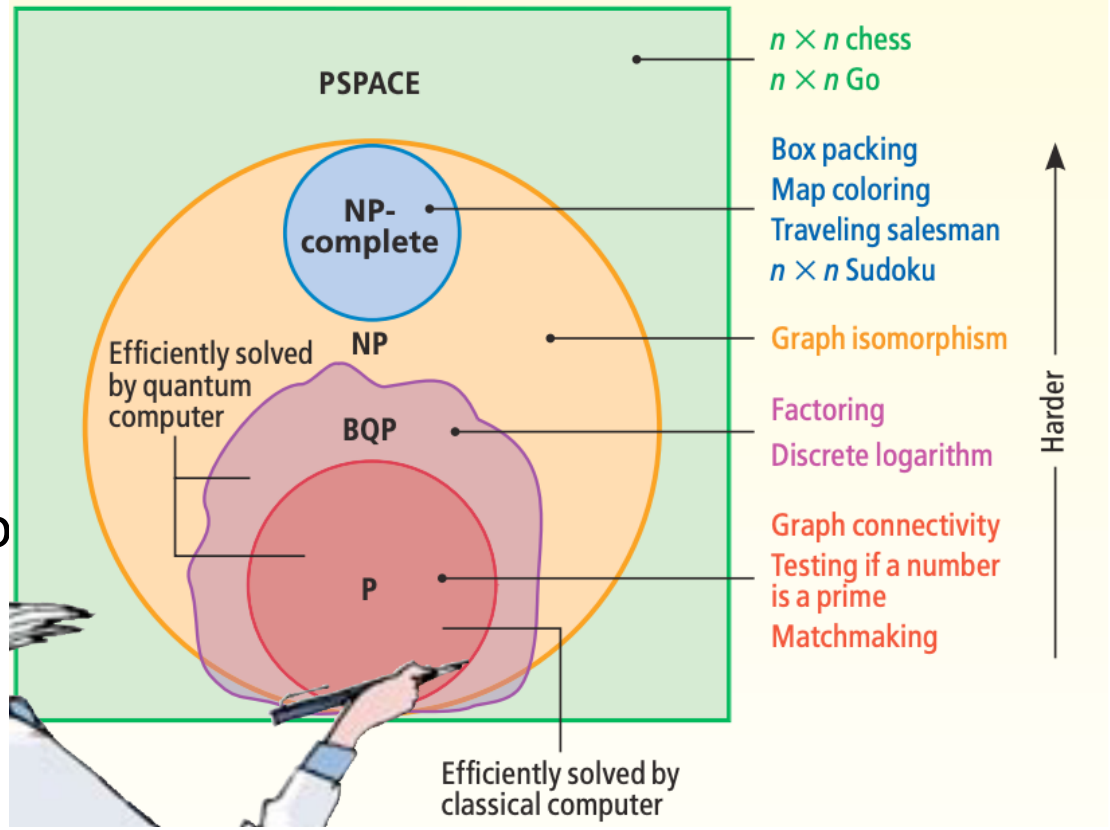
Computer quantistici

Qubit e informatica quantistica

- Basato su fenomeni di meccanica quantistica, quali sovrapposizione degli stati e entanglement
- Diversamente dal bit, il *qubit* può non solo trovarsi a 0 o 1, ma anche in sovrapposizione tra i due stati
- Obiettivo della **supremazia quantistica**: risolvere problemi difficili molto più efficientemente che con i computer classici
  - Risultati recenti (ottobre 2019): *Quantum supremacy using a programmable superconducting processor* https://www.nature.com/articles/s41586-019-1666-5#Abs1
    - Google: risolto in 200 secondi un problema che richiederebbe 10000 anni con un computer classico
    - IBM: se configurato bene, ci metterebbe 2.5 giorni, non 10000 anni!
  - Esperimenti più recenti sembrano confermare la supremazia quantistica

Quali problemi?

- I computer quantistici possono risolvere efficientemente i problemi NP-difficili?
- Non esattamente…
- Shor (1994) ha però sviluppato un algoritmo quantistico per la fattorizzazione dei numeri interi in tempo polinomiale
  - Probabilmente né NP-completo né in P
- NB: verificare *se* un numero è primo è in P



BQP: Bounded-error Quantum Polynomial time
I confini di questa classe non sono chiari