* Descrivi la differenza tra problema e algoritmo

Un problema descrive il “cosa”, ad esempio il problema dei cammini minimi. Un algoritmo

descrive il “come”, ad esempio l’algoritmo di Dijkstra per risolvere il problema dei cammini minimi. Un algoritmo è una funzione parziale dall’input I all’insieme delle soluzioni S, e A(i) è

una specifica soluzione del problema P.

* Cosa sono i problemi trattabili? E gli intrattabili?

I problemi trattabili sono risolvibili in tempo polinomiale (solitamente <= n^3) mentre gli

intrattabili sono risolvibili solo esponenzialmente.

* Cos’è la correttezza? Come si prova per algoritmi ricorsivi e iterativi?

La correttezza specifica che un algoritmo produca una soluzione corretta per il problema da

esso descritto. Per algoritmi ricorsivi si prova per induzione e per algoritmi iterativi si utilizza

la tecnica dell’invariante di ciclo.

* Parla dell’andamento asintotico, di omega, theta e omicron

O: g(n) è O(f(n)) se E c>0 e n0>=0 | g(n) <= cf(n) Vn>=n0 (g cresce al più come un mult. di f)

Ω: g(n) è Ω(f(n)) se E c>0 e n0>=0 | g(n) >= cf(n) Vn >=n0 (// almeno come un sottomult. di f)

θ: g(n) è θ(f(n)) se E c1,c2>0 e n0>=0 | c1f(n) <= g(n) <= c2f(n) Vn >= n0 (// esattam. come f)

* Quali costanti dovrei usare affinché c1(f(n)+g(n)) <= max(f(n), g(n)) <= c2(f(n)+g(n))

Posto c2=1 ho che max(x,y)<=(x+y): allora pongo c1=½ perchè la media è sempre <= dei due

* Parla di caso migliore, peggiore e medio, della loro definizione e fai un esempio.

Per molti algoritmi è impossibile stabilire un’unica complessità, poiché essa può variare in

base all’input; in questi casi si introducono le seguenti definizioni:

Tworst(n) = max{t(i) | dim(i) = n},Tbest(n) = min{t(i) | dim(i) = n}, Tavg(n) = avg{t(i) | dim(i) = n}

Ad esempio l’insertion sort ha un caso peggiore di θ(n^2) → O(n^2) per def., e un caso

migliore di θ(n) → Ω(n) per def. (solitamente il caso peggiore risulta quello più interessante

perchè fornisce garanzia che Texec non sarà maggiore di quel t prevedibile, mentre il medio

può risultare significativo quando la distribuzione delle probabilità riflette la situazione reale).

* Confronta i casi migliore, peggiore e medio tra alg. deterministici e randomizzati

A differenza dei A deterministici, nei randomizzati si ha il concetto di Expected Running Time

(media dei tempi di computazione su input di stessa dim, dipendenti dalla distribuzione dell’I).

Ad es. QS ha caso peggiore O(n^2) mentre LVQS ha E[t] nel caso peggiore di O(nlogn), ossia

può sempre richiedere O(n^2) nel caso peggiore del caso peggiore ma mediamente

richiederà O(nlogn).

* Definisci upper e lower bound di un problema e fornisci degli esempi.

Un problema ha complessità O(f(n)) se esiste un algoritmo di complessità O(f(n)) che lo risolve (risolubile in tempo non maggiore di f(n)). Un problema, invece, ha complessità di Ω(f(n) se è dimostrato che tutti gli algoritmi che lo risolvono hanno complessità Ω(f(n)) (cioè è impossibile risolverlo in meno di f(n)). Un esempio è il problema dell’ordinamento per confronti, che ha limite superiore e inferiore coincidenti di θ(nlogn). Senza conoscere, ad es., il merge sort, una persona potrebbe pensare un lim\_sup di n^2 conoscendo insertion sort e un lim\_inf di n osservando che vanno confrontati almeno una volta tutti gli elementi.

* Che differenza c’è tra problema aperto e chiuso? Un esempio di gap algoritmico?

Un problema aperto è tale se sussiste un gap algoritmico (ossia non si sono trovati dei lim\_sup e lim\_inf coincidenti). Viceversa un problema chiuso (come ad esempio quello dell’ord. per confronti). Un esempio di problema aperto è il SAT (soddisfacib. form. bool)

* Come si può chiudere un gap?

Si può chiudere dall’alto (trovando un algoritmo migliore e abbassando quindi il lim\_sup) o dal basso (dimostrando un lim\_inf più alto). I due metodi non sono mutuamente esclusivi, come nell’esempio del merge sort.

* Fornisci un esempio per i seguenti tipi di algoritmi: trattabili chiusi, trattabili con gap, presumibilmente intrattabili, intrattabili (dimostrati), insolubili (dimostrati)

In ordine: ord. per confronti - molt. tra matrici - SAT - Hanoi - Halting Problem

* Induzione aritmetica vs forte. Quale ci interessa? Fai un esempio per entrambe.

Arit: P(0) & P(n) → P(n+1) / Forte(ci interessa): P(0) & P(n) Vn<m → P(m)

Un es. classico di ind. aritmetica è la progr. arit. (sommat. da 1 a n = [n(n+1)]/2).

Un es. di ind. forte è quella usata per dimostrare la definizione di altezza di un albero

* Definisci scopo, complessità e relazione di ricorrenza della binary search ricorsiva

Lo scopo della bin\_Search è, tramite la tecnica del Divide Et Impera, di trovare un elemento all’interno di una sequenza ordinata dividendo ad ogni passo in due sottosequenze e cercando in quella corretta. La complessità risulta essere θ(logn) e la relazione di ricorrenza:

T(1) = k & T(n) = 1 + T(n/2) → da cui poter dim. la complessità ponendo n = 2^k

* Definisci scopo, complessità e le diverse relazioni di ricorrenza del quick sort

Lo scopo del QS è quello di ordinare una sequenza partizionandola ad ogni passo sulla base di un pivot scelto in diversi metodi. Nel caso migliore e medio QS ha complessità O(nlogn) e nel caso peggiore O(n^2), con le seguenti relazioni di ricorrenza:

generale: T(1) = 1 & T(n) = n + T(r) + T(n-r-1)

peggiore: T(1) = 1 & T(n) = n + T(n-1) + T(0) = n + T(n-1)

migliore: T(1) = 1 & T(n) = n + 2T(n/2) (come il merge sort)

* Definisci pseudocodice, complessità e relazione di ricorrenza delle Torri di Hanoi

Hanoi(n, from, to, aux)

if n = 1: move(from, to)

else: Hanoi(n-1, from, aux, to); move(from, to); Hanoi(n-1, aux, to, from)

La complessità risulta O(2^n) dalla relazione di ricorrenza: T(1) = 1 & T(n) = 2T(n-1) + 1

* Quali sono i 5 punti chiave dell’invariante di ciclo?

1) vale all’inizio (Pre → Inv) 2) alla fine garantisce Post (Inv & !B → Post) 3) si preserva ad

ogni iterazione 4) a ogni it. il valore di una funzione di terminazione t decresce strettamente

5) t è limitata inferiormente

* Definisci correttezza parziale vs totale, quale ci interessa?

Parziale: se prima vale Pre e il ciclo termina allora vale Post

Totale (ci interessa): se prima vale Pre ALLORA //

* Definisci un’invariante per la binary search iterativa e discutine la validità

x ∈ a[0..n-1] sse x ∈ a[inf..sup] con t = sup - inf

1) inf = 0 & sup = n-1 OK 2) x !∈ a[0..n-1] (el. non trovato) 3) x costruz. dei 3 casi

4) e 5): la distanza sup-inf decresce ad ogni passo (posso aggiungere a Inv).

* I 3+2 punti dell’Inv implicano la correttezza del ciclo rispetto a Pre/Post? Perchè?

Trovare l’invariante mi fornisce un metodo per la costruzione di un algoritmo corretto. Trovando un’Inv per cui valgano correttamente i vari punti chiave, si prova (abbiamo assunto che l’algoritmo termini) che eseguendo C n volte con n>=0, quando n=0 vale Inv & !B e dunque Post; notando che non può non terminare perchè essendo t limitata inferiormente troveremmo un assurdo (continuerebbe a decrescere il valore di qualcosa di limitato).

* Definisci invariante e pseudocodice del problema della bandiera nazionale olandese

Inv ipotizzando R|U|W|B: R(1, U-1) & W(W, B-1) & B(B, N) & U <= W con t = W - U

Pre: U = 1 & W = B = N+1 e Post: R(1, W-1) & W(W, B-1) & B(B, N)

while(U < W): case Colour(W-1):

R: swap(W-1, U) U++

W: W--

B: swap(W-1, B-1) W-- B--

* Definisci, riguardo ai grafi:

grado(n) per non orientati e in/out per orientati: #archi incidenti - #archi entranti/uscenti

max archi dati n nodi: !or: [n(n+1)] = O(n^2) e or: n^2

sommatoria gradi: !or: 2m e or(IN) = or(OUT) = m

multigrafi: G con + archi da un nodo all’altro

cammino, catena, semplice, ciclo: sequenza di n nodi da u0 a un t.p.c esiste un arco (ui, ui+1) per ognuno di essi - catena se !or. - semplice se gli n sono tutti diversi a meno, eventualmente, di u0 e un - ciclo se u0 = un

connesso, fortemente connesso: !or: non esistono nodi isolati - or: esiste cammino da - per tutti i nodi (debolmente connesso se il corrispondente !or sarebbe connesso)

sottografo indotto: dato G’ ⊆ G, dei nodi V’ ⊆ V tengo tutti gli archi

albero libero: grafo !or, connesso, aciclico (senza concetto di radice)

spanning tree (albero ricoprente): sottografo di G con tutti i nodi che è anche albero libero

* Elenca le complessità per lista di archi, lista di adiacenza (conviene) e matrice di adiacenza

Spazio: n+m n+m(or) n^2

Grado: m grado(u) n

Adiacenti: m grado(u) n

Esiste arco m min(g(u),g(v)) 1  
+ Nodo: 1 1 n^2

+ Arco: 1 1 1

- Nodo: m m n^2

- Arco: m g(u) + g(v) 1

* Quali visite per grafi esistono? Che frangia usano? Che complessità hanno?

BFS (Ampiezza) che utilizza una coda e DFS (Profondità) che per la versione iterativa utilizza una pila. Hanno entrambe una complessità di n+m (n per marcatura dei nodi ed m per le operazioni eseguite per ogni arco sugli adiacenti).

* Effettua una visita BFS su un grafo, esplicitando la coda [📦](https://emojipedia.org/package/)
* Effettua una visita iterativa DFS su un grafo, esplicitando lo stack [📦](https://emojipedia.org/package/)
* Effettua una visita DFS ricorsiva su un grafo non connesso [📦](https://emojipedia.org/package/)
* Che problema risolve Dijkstra? Che complessità ha con e senza heap?

Dijkstra risolve il problema dei cammini minimi (G con somma dei pesi minima) da un nodo src a tutti gli altri nodi del grafo G pesato (senza pesi negativi). La complessità risulta essere in tutto O((n+m)logn): n per via dei cicli di aggiunta delle distanze all’array dist e di aggiunta dei nodi all’Heap - m per le operazioni di aggiornamento svolte per ogni arco - logn per le operazioni di getMin e changePriority svolte sull’Heap. In caso di grafo denso, usando l’Heap, ho uno svantaggio (O(n^2 logn) che è peggio di O(n^2) senza Heap). Se uso Liste ordinate ho getMin costante ma getElem lineare → n^2 e viceversa con liste non ordinate.

* Effettua un tracing di Dijkstra esplicitando array dei padri e heap [📦](https://emojipedia.org/package/)
* Parla dell’Inv e della prova di correttezza di Dijkstra

L’idea dell’algoritmo è di tenere le distanze provvisorie che alla fine saranno le distanze definitive. Ad ogni passo estraggo il nodo con dist minima e aggiorno (se migliori) le distanze adiacenti, utilizzando un Heap come frangia. Da queste osservazioni derivano:

Inv1: Vu non in H dist[u] = d(u) + Inv2: Vu in H dist[u] = costo camm. min. da s a u (oppure inf)

Pre: dist[s] = 0 e dist[u] = inf Vu != s / Post: Vu dist[u] = d(u) / t : #H (card.)

La Inv2 (ripristinata con l’aggiornamento delle distanze dopo il controllo) serve a far valere Inv1 → si prova che dato π cammino min. per nodi neri, esso è il minimo in assoluto perchè presi π1 e π2 alternativi, se #π1 >= #π a maggior ragione #π1 + #π2 >= #π

* Che problema risolvono Prim e Kruskal? Su che grafi? Complessità di Prim?

Dato un G non orientato e connesso (se connesso sicuramente esiste min. span. tree) si ottiene un sottografo con tutti i nodi, connesso, acicilico e con somma dei pesi degli archi minima (un min spanning tree). La complessità di Prim è uguale a quella di Dijkstra.

* Cosa sono le union-find? Complessità di Kruskal? Come migliora col path compression?

Le U-F sono delle strutture dati che risultano utili per implementare Kruskal, sussitendo una corrispondenza 1:1 tra i set dell’U-F e i sottoinsiemi della foresta ricoprente che viene costruita dall’algoritmo. Possono esistere delle varianti di implementazione:

Quick find (alberi sempre alti 1) con F costante e U lineare: rendo figli di root1 tutti gli altri nodi

Quick union con F logn e U costante (ipotizzando semplice collegamento di root1 a root2)

Union by Size: si mantiene l’albero con + nodi memorizzando la size di ogni set

Union by Rank: si sceglie l’albero più alto (necessario adattamento in caso di Path Compres.)

→ Scegliendo UbS o UbR si ottiene l’altezza dell’albero sempre O(logn)

Path Compression: ad ogni find si rendono figli della root i nodi incontrati (Union implica Find → complessità ammortizzata di n mkSet + 2m find + n-1 union: “quasi” O(m+n) cioè O((m+n)log\*n). Da ciò si ottiene, poiché il G è connesso (m >= n-1) e quindi O(m+n) viene assorbito in O(m), una complessità di O(mlogm).

* Effettua un tracing di Prim esplicitando lo spanning tree [📦](https://emojipedia.org/package/)
* Effettua un tracing di Kruskal esplicitando lo spanning tree e qualche operazione U-F [📦](https://emojipedia.org/package/)
* Parla dell’Inv e della prova di correttezza di Prim e di Kruskal (simili, si riporta dunque Prim)

Dato T l’albero costruito fino a quel momento: Pre: tutti i nodi in H / Post: T è m.a.r di G

Inv1: T ⊆ qualche m.a.r + Inv2: Vu in H dist[u] = dist tra u e T = costo min. di arco tra u e un nodo nero (o inf.) → Si preserva perchè, una volta preso (y,u), se si ipotizza non essere appartenente a T si deve allora prendere una tripla di archi (y,x), (x,z), (z,u) che attraversino la frontiera (con x nero e z in H), ma T+ con i nuovi archi richiede un collegamento in più rispetto a T con (y,u), ottenendo un assurdo.

* Cos’è l’ordinamento topologico? Che grafi prende in input?

Un ord. top. è un ordine totale dei nodi che preservi gli archi, data una relazione di raggiungibilità U<V su un DAG (se esiste un cammino da U a V) da cui si costruisce un ordine parziale.

* C’è un algoritmo dummy per topsort? é corretto?

<cerco nodo src (per cui non esistono archi entranti)> → <lo elimino>

è corretto perchè supponendo per assurdo che non esista (ad ogni passo) un nodo src u0, deve esistere un u1 src da cui ci si arriva, e così all’infinito o formando un ciclo (è DAG!)

* Esponi come lavorano i due algoritmi topsort e discutine correttezza e complessità

Un primo algoritmo, per evitare di cancellare i nodi, alla loro estrazione in qualità di nodi src ne diminuisce l’IN-degree degli adiacenti. L’algoritmo inserisce in prima istanza gli indegree in un array, costruisce una sequenza S con i nodi e una sequenza ord inizialmente vuota per ospitare l’ordine topologico. Per ogni nodo della sequenza lo aggiunge in fondo a ord diminuendo indegree[v] per ogni arco (u,v) ad esso collegato (se indegree[v] = 0 v viene reinserito all’interno della sequenza S). La complessità è di O(n+m).

Un algoritmo alternativo risulta in una DFS con l’aggiunta di un timestamp: parto da A e seguo il path fino a Z contando al numeratore all’andata e al denominatore al ritorno (sse non esistono altri archi uscenti da quel nodo). Anche per questo alg. la complessità è di O(n+m).

Esso è corretto perchè se i due nodi sono collegati, se da u va in v, v finisce prima di u (end[v] < end[u]), se invece visitiamo prima v: DFS(G,u) non viene chiamata perchè è aciclico, e dunque di nuovo end[v] < end[u].

* Cosa sono le componenti fortemente connesse? Discutine proprietà e correttezza

Una cfc è un insieme di nodi mutuamente raggiungibile massimale (stanno sullo stesso ciclo). Si definisce una relazione di equivalenza u <-> v da cui deriva la definizione di cfc come classe di eq. e dunque del grafo quoziente Gq come G aciclico contenente le cfc. Si nota che: 1) cfc è pozzo/src se lo è in Gq (end(c) = max{end(u)|u in c}) 2) dato cfc pozzo sono raggiungibili da esso tutti e soli i nodi di quella cfc **3)** cfc src diventa pozzo in Gt(trasposto).

L’algoritmo dunque esegue una dfs con timestamp su G per ottenere un ord.topologico, che viene letto al contrario per eseguire una seconda dfs sul grafo trasposto, ottenendo a ogni passo le cfc. La complessità è di O(n+m) (per 3 volte si eseguono porzioni di complessità n+m, cioè le due DFS e la trasposizione → 3(n+m)). Per la prova di correttezza si prova la correttezza di DFS (corrispondente al punto 1) e si aggiungono i punti 2) e 3).

* Effettua un tracing dell’algoritmo per le cfc comprendente topsort. Complessità? [📦](https://emojipedia.org/package/)
* Cos’è la programm. dinamica? Punti in comune e differenze e col Divide et Impera?

La programmazione dinamica è uno schema algoritmico. In comune col Divide et Impera ha la suddivisione in sottoproblemi di dimensione minore, cioè sono entrambi schemi induttivi. Tuttavia, la programmazione dinamica è basata su un approccio bottom up, rispetto al top-down del DeT dove devo ricalcolare più volte lo stesso sottoproblema. Questo approccio implica tipicamente la memorizzazione delle soluzioni intermedie al fine del loro riutilizzo per sottoproblemi “più alti”. Essa risulta dunque vantaggiosa se i sottoproblemi servono più volte, anche se con un approccio bottom up potrei trovarmi a calcolare sottoproblemi non strettamente necessari ai miei fini.

* Parla dell’esempio di fibonacci e delle 3 possibilità di realizzazione

L’algoritmo della successione di Fibonacci è realizzabile in 3 modi differenti. Il modo più classico è con la funzione ricorsiva: f(n) = if(n <= 1) ret 1; else return fb(n-1) + fb(n-2);

Si nota però che i sottoproblemi vengono ricalcolati più volte ottenendo dunque una complessità esponenziale con relazione di ricorrenza: T(n) = T(n-1) + T(n-2) + O(1), che maggiorando diventa T(n) > 2T(n-2)+1 → O(2^(n/2)). Un modo alternativo, sfruttando la programmazione dinamica, è quello di utilizzare un array ausiliario: f[i] = f[i-1] + f[i-2], ottenendo in questo modo una complessità lineare grazie al riutilizzo dei sottoproblemi.

Esiste in realtà anche una soluzione ricorsiva efficiente basata sulla costruzione di una lista di coppie dal basso (in stile funzionale): fb(n) = let <f(n-1),f(n)> = fb(n-1) in <f(n),f(n)+f(n-1)>

* Che problema risolve il Longest Common Subsequence? Che complessità?

Un esempio di programmazione dinamica si trova nell’algoritmo di LCS con complessità temporale e spaziale O(m\*n), anche se è possibile ottimizzare la spaziale tagliando le parti di matrice non utili. LCS è il problema di trovare la sottosequenza di lunghezza maggiore comune a due sequenze (ad esempio tra POPI e PINO → PO o PI a seconda di da dove inizio). Il suddetto algoritmo si basa sull’analisi per ogni lettera Li di tutte le altre lettere Lj, ponendo +1 sulla matrice se Li=Lj (con annesso “puntatore” in alto a sinistra), e ponendo il numero in alto (con “puntatore” in alto) se esso è maggiore del numero a sinistra e viceversa

L’algoritmo è realizzabile anche esponenzialmente con il triviale alg. brute force, con relazione di ricorrenza: T(0,m) = T(n,0) = 1 & T(n,m) = T(n-1,m) + T(n, m-1) + 1 nel caso peggiore → exp come Hanoi).

* Effettua un tracing di LCS tra le parole POPI e PINO [📦](https://emojipedia.org/package/)
* Parla di Floyd-Warshall. Su che grafi opera? Complessità temporale e spaziale?

Dato un G orientato e pesato, F-W trova i cammini minimi (semplici) per ogni coppia di nodi. I pesi possono essere anche negativi ma non possono esserci cicli negativi (poiché creerebbero un ciclo infinitamente migliore). Uso la definizione di cammino K-vincolato, cioè che usa solo i nodi da 1 a k, con gli estremi esclusi dal vincolo. Dato dk(x,y) il costo minimo k-vincolato da x a y e dn(x,y) il costo finale: uso D (matrice distanze) e π (matrice predecessori per trovare l’albero dei cammini minimi e per trovare il cammino da x a y con l’algoritmo shortest path). Posso utilizzare una matrice unica e non per forza k matrici diverse perché dato un cammino x-k “k-1 vincolato” al massimo ci può essere k, ma in questo caso un cammino da k a k (non potendo avere cicli negativi) è per forza >=0 e dunque sempre > di quello senza k. La complessità temporale è O(n^3) e (con ottimizz.) e O(n^2) spaziale.

* Effettua un tracing parziale di F-W [📦](https://emojipedia.org/package/)
* Esponi la teoria dell’NP completezza nei suoi punti generali (P, NP, NP-C, P=?NP)

Dividiamo i problemi in tre classi: P (risolvibili in tempo polinomiale), NP (risolvibili in tempo non polinomiale o alternativamente non deterministic pol. time) e NP-C (se problemi “hard” (difficili) e appartenenti alla classe NP). Si nota che NP-C ⊆ NP e P ⊆ NP, ma non si è ancora dimostrato se P = NP. Trovare un algoritmo polinomiale per un problema in NP-C proverebbe che P = NP, mentre dimostrare che per un problema di NP-C non esiste un algoritmo pol. proverebbe che P != NP (ne basta uno in entrambi i casi!).

* Definisci un problema formalmente e classificalo in: “decisione, ricerca, ottimizzazione” e in “astratto, concreto”. Perchè serve una codifica concreta? Che proprietà ha la f di codifica?

Un problema è definibile come sottoinsieme del prodotto cartesiano tra input e insieme delle soluzioni (p ⊆ I x S), mentre l’algoritmo (A(i) sol. per p) è definibile come una funzione da I a S. Un problema di decisione, a cui si cerca sempre di ricondursi, è definibile come una funzione da I a {T, F}. Un problema di ricerca trova 1 soluzione, mentre un problema di ottimizzazione trova una soluzione minima o massima. Ci interessa che un problema (consideriamo il p di decisione) NON sia risolvibile, perché se quello di decisione non lo è, a maggior ragione neanche quelli di ricerca e ottimizz. lo saranno (se so trovare il m.a.r so anche trovare un m.a.r < 100). Fin ora abbiamo sempre parlato di problemi astratti, ma quando si parla di algoritmo polinomiale si parla di p(n) per una certa n = dim(I), che però non è unicamente definibile per ogni tipologia di input. S’introduce dunque un problema concreto, definibile tramite una funzione di codifica c in stringa binaria {0,1}\*, che deve essere iniettiva e totale (voglio poter codificare tutto). Si definisce pc(x) = T sse p(i) = T e c(i) = x, ossia un p concreto su x str binaria è vero se p astratto per i è vero e la funzione di codifica da i fornisce x (si definisce F su x che non sono codifiche di nessuna i (infatti c non è surgettiva) → spuria)

* Definisci formalmente NP e NP-C. Che relazioni hanno? Cos’è la funzione di riduzione?

p ∈ NP sse E alg. di verifica Av | Ec. Av(i, c) = T sse p(i) = T (Esiste algoritmo di verifica polinomiale, o alternativamente, esiste un algoritmo polinomiale non deterministico (ad esempio la generazione casuale di un certificato, con successivo tentativo di match, che ovviamente può produrre risultati errati).

Per esprimere la relazione “più difficile di” tra NP e NP-C definiamo una funzione f di riduzione che sia corretta e calc. pol. che ci permetta di dire che se Q è risolvibile pol → anche p lo è (principio di riuso del sw). Diciamo dunque p <=(p) Q (p riducib. a Q) sse E f : Ip → Iq (concreti) | calcolab. pol. (NB: anche lo spazio è pol. (y.len non può essere exp se t.pol)

Ap: [ -x-(→Af–f(x)=y→Aq(y∈?Q) → T/F] (componendo Aq(pol. x ip.) e Af(f di rid.) rimango pol)

* Parla del problema SAT e della sua estensione coi quantificatori. SAT <= Q? Con che f?

SAT è il problema della soddisfacibilità di formule booleane, che esprimiamo in CNF (forma normale congiuntiva) nella sua versione base, ma che possiamo estendere con i quantificatori (+ difficile di SAT, il primo è in NP e il secondo in NP-C). Ci vogliamo chiedere se esiste un’assegnazione alle variabili che rende vera la formula, tuttavia laddove è molto facile determinare la validità di un’assegnazione data (certificato) con una semplice funzione per casi eval, è molto difficile determinarne la validità “da zero” (se non, apparentemente, provando tutte le possibilità). Per dimostrare che SAT <= Q(SAT coi quantificatori) si utilizza la f di immersione (non serve trasformare nulla perché è sottoinsieme e dunque se una formula ∈ SAT, ∈ anche a Q). SAT è stato storicamente il primo problema di cui è stata dimostrata l’appartenenza a NP-C (VQ∈NP Q <=p SAT)

* Come si prova che un problema appartiene a NP-C? Serve?

Si deve 1) provare che p ∈ NP e 2) provare che VQ∈NP Q <=p p (se conosco un altro p ∈ NP-C basta dare VQ Q <=p(f) p’(SAT) <=p(g) p (ossia la composizione gof). Provare che un p ∈ NP-C è utile perché so che sarà molto difficile trovare un A pol. → tento di ridurre, usare tecniche alternative, ecc..

* SAT <= 3SAT? E 2SAT?

Nonostante riduca il caso generale al solo N=3, 3SAT rimane in NP-C (2SAT invece ∈ P). Lo si può provare aggiungendo 3 variabili y1,y2,y3 e “condensando” la formula SAT in 3SAT tramite l’espansione inversa (sostituzione a passi successivi → CNF <= 3CNF)

* 3SAT <= CLIQUE?

Clique è il problema di trovare, dato G !or, il sottoins. denso massimo, per il quale cioè ho 1 arco per ogni coppia di nodi. In relazione con questo problema, una formula

è soddisfacibile sse il grafo associato alla stessa (var = nodi e arco tra due nodi se sono in 2 clausole diverse e non sono l’uno l’opposto dell’altro) ha clique di dimensione x.

Oppure, per ragionamento inverso, se esiste clique (con nodo Vclausola) allora, facendo venire T il literal, rende la formula vera → ergo 3SAT e CLIQUE ∈ entrambi a NP-C

* Fai altri esempi di problemi NP-C

insieme indip. in G !or. (v’ clique sse ins.ind. in G’ (archi dove non ci sono e viceversa) -

commesso viaggiatore (ciclo hamilt.) - zaino (k ogg → peso/rendimento) -

copertura nodi(sottoinsieme di archi min. che copra tutti i nodi) - colorazione (probl. confini)

* Ottimo lavoro soldato :D

Grazie me stesso

NB: i [📦](https://emojipedia.org/package/) sono da svolgere su foglio/lavagna, e dunque non si riportano nelle risposte!

SECONDA PARTE

* Definisci gli algoritmi di tipo Las Vagas e quelli di tipo Monte Carlo  
  Un algoritmo Las Vegas è un tipo di algoritmo non deterministico, o randomizzato, che una volta fissato un input restituisce sempre lo stesso output. La randomizzazione è nel tempo che l’algoritmo impiega per la sua esecuzione, cioè il tempo t è una variabile casuale.  
  Un algoritmo Monte Carlo, invece, è un algoritmo randomizzato in cui non sono sicuro di ottenere ogni volta la risposta corretta, ed è dunque necessario effettuare un certo numero di run per poter essere sicuro ad una certa percentuale di aver trovato il risultato voluto.
* Definisci la proprietà di linearità per l’espettazione e inseriscila nel contesto del corso  
  L’espettazione di una somma di variabili casuali è la somma delle espettazioni. Abbiamo usato questo risultato per dimostrare l’esempio del guardarobiere distratto, in cui risulta molto più semplice calcolare la sommatoria delle espettazioni (N \* 1/N = 1) piuttosto che l’espettazione della somma (funzione a più variabili).
* Parla di LVQS, definisci lo pseudocodice e la relazione sui confronti I(i, j)  
  L’LVQS ha lo stesso pseudocodice del QS con previo campionamento da S con prob. uniforme (shuffle), poi vengono formati S< e S>= e viene chiamato LVQS ricorsivamente sulle due sottosequenze per poi concatenarli dopo le chiamate.  
  La relazione sui confronti I(i, j) è di 2 casi favorevoli (2 confronti sui pivot delle sottoseq.) su j - i + 1 casi possibili, ottenuto da delle osservazioni sui confronti (esempio sul quaderno). Se si approssima la sommatoria di 2/(j-i+1) ad un integrale si può ottenere E[scambi] = O(nlogn).
* Ricorda la proprietà vista sull’intersezione delle probabilità e inseriscila nel contesto del corso  
  P(ABC) = P(A|BC)/P(BC) \* P(B|C)/P(C) \* P(C); utilizzata per la dim della p di MinCut
* Cosa descrive il problema del taglio minimo? Cita un algoritmo deterministico  
  Descrive il problema di trovare il minimo numero di tagli agli archi di un grafo necessario a rompere la connessione dello stesso. Un algoritmo deterministico consiste nel partire da un nodo sorgente, arrivare al penultimo nodo salvandomi il taglio da fare verso l’ultimo condensando il collegamento (ad es. se ho tagliato 2 archi verso E e F utilizzerò un doppio collegamento verso E,F). Ripeto la procedura sino a terminare gli archi salvandomi il minimo, oppure sino ad incontrare un taglio pari a 1 (minimo assoluto trovabile).
* Descrivi l’algoritmo randomizzato Monte Carlo per il Min Cut. Con che p ottengo il taglio min?  
  Campiono un arco da G con p uniforme → Unisco i vertici, elimino gli archi che univano u e v e mantengo gli archi che incidevano su u e v (il grado dei nodi non scende) → Ripeto fino a che rimangono due vertici → Il min cut candidato sarà il #archi che collegano questi due ultimi vertici. Ottengo il risultato giusto se durante l’esecuzione non campiono archi appartenenti al min cut. Ad es. dati n vertici e |taglio minimo| = k, abbiamo almeno (nk)/2 archi (altrimenti esisterebbe almeno un vertice di grado minore di k i cui archi costituiscono un taglio minimo di dim. < k) e p(beccarlo) <= 2/n (perché casi fav.(k)/casi poss.(nk/2) → dunque p(non beccarlo) >= 1 - 2/n. Per le prove ripetute faccio la produttoria e ottengo 2/n(n-1) dunque p(ottenere taglio minimo) > 2/n^2 (dunque la p di NON ottenerlo al crescere di m tende a 0).
* Effettua qualche passo di tracing sui due tipi di algoritmi per il min cut  
  [📦](https://emojipedia.org/package/)
* Descrivi il problema dell’accordo bizantino in generale e cita la sua utilità  
  L’accordo bizantino, utile per la gestione dei contratti nei sistemi distribuiti, descrive il problema del trovare da parte dei processi affidabili un consenso (su una certa maggioranza) nonostante la presenza di processi inaffidabili (nella realtà difettosi) che tentano di remare contro.
* Spiega gli elementi: n, t, b(i), consenso v, validità, maj, tally riguardo all’accordo bizantino  
  n: #totale di processi (ad esempio n >= 3t+1)  
  t: processi inaffidabili (ad es. 1)

b(i): bit inviato dal processo i al termine di ogni round  
v: consenso raggiunto quando tutti i processi affidabili hanno raggiunto la soglia (es: 2t+1)  
maj: bit con > occorrenza ricevuti dal processo x

tally: #occorrenze del bit maggioritario del processo x

* Descrivi elementi, relazioni e pseudocodice del protocollo MC per l’accordo bizantino  
  Per ogni processo affidabile j ho un input b0(j) e un output b(j) = v con p = 1/2

while T // in 1 round ho ½ di p di aver trovato l’accordo, in 2 ne ho ¾ , in 3 ⅞ , ecc  
 trasmetto b(j); ricevo i bit; maj(j) ← maj; tally(j) ← tally;

if(tally(j) >= 2t+1) b(j) = maj(j) // se lo tolgo violo il vincolo di validità

else if T b(j) = 1 else b(j) = 0 // se c’è maj ampia di un tipo non può esserci dell’altro

* Descrivi elementi, relazioni e pseudocodice del protocollo LV per l’accordo bizantino  
  n = 8t+1 / affid. = 7t+1 / t = 1 / L = floor(n/2)+t+1 = 5t+1 / H = L + t = 6t+1

La doppia soglia depotenzia l’azione degli inaffidabili, poiché la scelta della soglia invalida con p = ½ la loro azione (nell’es. max turni = 3 e E[turni] = 2). Se uno supera la soglia > tutti superano la <. Se uno non supera la < nessuno supera la >, sennò sono tutti compresi tra < e >. Se ci sono due maj diverse nessuno supera L. Ovviamente se l’accordo è pregresso non violo il vincolo di validità.  
while T

trasmetto b(j); ricevo gli altri; maj(j) ← maj; tally(j) ← tally;

if T soglia = H else L

if tally(j) >= soglia b(j) = maj(j) else b(j) = 0

if tally(j) >= 7t+1 b(j) = maj(j)

* Definisci la congruenza come relazione d’equivalenza e fai un esempio  
  a = b(mod n) con a,b ∈ Z è rel. riflessiva, simmetrica e transitiva.  
  8 = 5(mod 3) perché 8/3 ha resto 2 e anche 5/3. Oppure 8-5=3 (3\*1(multiplo))
* Cos’è la coprimalità? Definisci Zn, Z\*n e gli inversi moltiplicativi. Fai esempi  
  Due numeri a, b sono coprimi se MCD(a,b) = 1, cioè se non hanno fattori in comune (ad esempio 15 e 8 sono coprimi tra loro). Un gruppo Zn rispetto alla somma contiene le classi di resto con rappresentanti da 0 a n-1 (cardinalità n). Un gruppo Z\*n contiene n-1 classi (da 1 a n-1) se n è primo, e < n-1 altrimenti. In un gruppo Z\*n sono presenti coloro che hanno un inverso moltiplicativo, cioè i coprimi. Ad esempio in Z\*7 l’inverso di 2 è 4 perché 2\*4 = 8 = 1(mod7), mentre in Z\*8 = {1,3,5,7} ogni elemento è inverso di se stesso.
* Con p primo, che soluzioni ha x2=1(mod p)? E binom(p, i)? Perchè?  
  Se p primo x2=1(mod p) ha come soluzioni x = 1 e x = -1 = p - 1.  
  Si verifica sapendo che la congruenza è se “x2 - 1” è multiplo di p (x2=0(mod p)) ottengo x2 - 1 = (x-1)(x+1) → sse x = +-1.  
  Se p primo, inoltre, bin(p, i) = 0 (mod p) ~= kp perché il p rimane e verrà moltiplicato da qualcosa (ad esempio bin(5,2) = 10 = (p \* k) = (5 \* 2))
* Dato a non divisibile per p: dimostra per induzione il piccolo teor. di Fermat ap-1 = 1 (mod p)  
  ap = a (mod p) → per induzione con p.base (a = 1): 1 = 1(mod p) OK  
  p.indutt: dato np = n (mod p) → (n+1)p = (1+n)p = 1p + bin(p,1)n + bin(p,2)n2 +..+np

= (dato bin(p, i) = 0 (mod p)): 1 + 0 +...+ 0 + n = n + 1

* Come posso vedere un n dispari? Come sviluppo tale risultato affinché sia utile per Miller-R?  
  Dato n = q \* 2s + 1, prendiamo a = {2, …, n-2}:

an-1 - 1 = a(q\*2^s)-1 = *{(a2 - 1) = (a+1)(a-1)}* = (a(q\*2^s-1)+1)(a(q\*2^s-1)-1) = … = (aq+1)(aq-1)

* Spiega idea e pseudocodice di Miller Rabin, svolgendo anche un esempio  
  L’idea è quella di decomporre il numero in forma q \* 2s + 1, campionare uniformemente tra {2, … n-2}, calcolare aq(mod n) per poi risalire da destra verso sinistra quadrando per s passi fino ad arrivare ad an-1. Nel primo passo, se le soluzioni sono +-1 si ritorna “probabilmente primo”. In ogni passo è necessario controllare se il polinomio si annulla: se si ottiene -1 si ritorna “probabilmente primo”. Se alla fine si sono sempre ottenuti valori diversi da +-1 si ritorna con sicurezza “composto”. Le possibili sequenze ottenibili sono:

“-1, 1, 1…” (una volta ottenuto -1 esco, procedendo otterrei tutti 1) o analogamente “1, 1, 1…”

“?, ?, …, -1, 1” (ottengo val. diversi, una volta ottenuto -1 esco, procedendo otterrei tutti 1).

(“?, … ?” → composto)

* Cosa sono testimoni e bugiardi? Fai esempio di 7 e di 65 per “a”=5 e “a”=8  
  **7** = 3 \* 21 + 1 → q = 3 e s = 1 → consideriamo a = 2 - a = 5 (6 è banale) e ripetiamo s=1 volte  
  2: 23 = 8 = 1 (prob. primo) / 3: 33 = 27 = 1 (prob. primo) / 4 : 64 = 1 / 5 : 125 = 1 → prob primo  
  **65** = 1 \* 26 + 1 → q = 1 e s = 6 → proviamo con a = 5 e troviamo che è testimone (perché MCD (5, 65) = 5 → un non coprimo non è invertibile)  
  → proviamo con a = 8 (e dunque 64 perchè si va a coppie) ho un bugiardo (il primo!)
* Perchè la probabilità di fallire (avere un “a” bugiardo) è minore della metà?  
  Perchè per Lagrange il massimo sottogruppo proprio ha cardinalità uguale al massimo divisore della cardinalità di Z (ad es. in Z48 ho max card. 24), dunque i bugiardi sono al più la metà in quanto sottogruppo proprio dello Z\*n → la p di fallire (bugiardo) è < ½ e così via per la geometrica. Nell’esempio di 65 ho i bugiardi 8, 57, 18, 47, (1, 64) MA ho 14 e 51 testimoni nonostante appartengano al sottogruppo dei bugiardi (perché per essere sottogruppo devono esserci gli inversi dunque inserisco 14 e 51 ottenuti dal prodotto di bugiardi). Importante notare che esiste comunque sempre almeno un testimone fuori dal sottogruppo dei bugiardi.
* Costo di Miller Rabin?  
  Il costo è O(k(logn)3) dato il log come misura della lunghezza di un intero. Esiste un alg. deterministico con O(k(logn)6) e dunque risulta ancora efficiente usare MC Miller Rabin.
* La moltiplicazione matriciale è onerosa? Si può migliorare? Si può trovare un’alternativa?  
  Si, ha una complessità di n3, nonostante sia migliorabile riducendo (ad esempio in una matrice 2x2) da 8 a 7 le moltiplicazioni da effettuare. Questa osservazione è alla base di un algoritmo ricorsivo che richiede O(n log2 7 ) = O(n 2.807) moltiplicazioni. Negli anni l’esponente è stato ripetutamente abbassato. Al momento l’algoritmo ottimale richiede O(n 2.372) moltiplicazioni, ma la verifica di una sua corretta implementazione è estremamente complicata. Esiste un algoritmo galattico di circa n2.4, che però data la grande costante a cui è associato assume senso solo in termini asintotici.
* Descrivi l’algoritmo di MC Matrix Multiplic. Verif. In quali casi “falso il risultato”? P(k) su k run?  
  Esiste un algoritmo di verifica di uguaglianza che sfrutta la proprietà associativa. Invece che effettuare la moltiplicazione (onerosa) tra A e B, si campiona un vettore r di 0 e 1 e si effettuano le seguenti operazioni per verificare se AB = C (A(Br) = Cr?):

si effettua B\*r ottenendo un primo vettore s, per poi moltiplicare A\*s ottenendo t →  
→ si moltiplica C per t verificando se u = t → se si probabilmente AB = C, sennò diversi!  
Falso il risultato in due casi:

- perdendo informazioni in caso di zeri nel vettore

- sommando per addendi differenti ma che producono lo stesso risultato  
La probabilità che, date AB != C, risultino però uguali, è <= ½ → eseguendo k volte otteniamo una ottima p pari a 1 - 1/2k

* Definisci la stima dei numeri primi, il numero fattori primi di un numero e la fingerprint

La stima dei numeri primi pi(x) <= a x è pari a x/ln(x), infatti lim to inf pi(x)/(x/ln(x)) = 1  
I fattori primi di un numero < 2l sono al più l → mando 2ln(l) invece che 2l  
Dato p tra 2 e l2 e n < 2l la fingerprint è f(n) = n (mod p)

* Descrivi l’algoritmo di fingerprint. Con quale probabilità “falso” il risultato?  
  Date due stringhe binarie a e b, campiono p tra 2 e stringa.size2, calcolo la fingerprint “f(i) = i (mod p)” vedendo a e b come interi, e se il valore assoluto della differenza tra le due fingerprint è uguale a 0 (mod p) sono probabilmente uguali.  
  Dato che i fattori primi di |a-b| <= 2l sono al più l, e dato che la stima dei numeri primi è pari a x/ln(x) → l2/2ln(l), ponendo casi favorevoli / casi possibili ottengo p ~= 2ln(l) / l
* Come si risolve un problema di ottimizzazione? Cos’è l’algoritmo del simplesso?  
  Un problema di ottimizzazione (lineare perchè ci limitiamo a casi lineari) si risolve ponendo uguale a zero la derivata della funzione obiettivo. In caso la funzione obiettivo sia composta da più vincoli si vuole risolvere un problema di programmazione lineare, trovando una regione ammissibile e utilizzando l’algoritmo del simplesso trovando il vertice del poligono della regione che massimizza la funzione obiettivo. Si nota che la maggior parte dei vincoli potrebbero risultare ridondanti e che, inoltre, per trovare il primo vertice si può utilizzare l’algoritmo del simplesso stesso identificando il vertice se il valore dell’obiettivo non è negativo, per poi procedere spostandomi lungo le rette dei vincoli dei vincoli nella direzione dove O è massima (nell’esempio fatto: somma tra coordinate cartesiane)
* Risolvi l’esercizio proposto a lezione sulla programmazione lineare (f O = x1 + x2)  
  Dati A, B due macchinari e x1, x2 due beni:

40m A + 20m B per produrre x1 / 20m A + 60m B per produrre x2

In magazzino: x25 x1 e x90 x2

Pianificazione: 40 h per A, 30 h per B / con alla fine x75 x1 e x95 x2

→ V1 = 40x1 + 20x2 <= (40 \* 60) / V2 = 20x1 + 60x2 <= 1800  
→ V3 = x1 >= (75 - 25) = 50 / V4 = x2 >= (95 - 90) = 5  
Proietto i 4 vincoli, trovo la regione ammissibile, parto da un vertice e mi muovo lungo i vertici trovando che quello che massimizza la f obiettivo è quello in alto a destra.

* Descrivi l’algoritmo LV Incremental LP[📦](https://emojipedia.org/package/)
* Parla della teoria dei giochi con riferimenti all’esempio proposto a lezione  
  Preso come esempio un gioco di carta/forbice/sasso tra due giocatori R e C che vogliono entrambi fare il loro meglio per vincere. Se immaginiamo una matrice dal punto di vista di R, i valori positivi saranno le vittorie di R (somma che C deve pagare) mentre i negativi le sconfitte di R. Nell’esempio, il migliore degli scenari peggiori è quello in cui si perde meno considerando strategie pure e non miste (VR = max(i) min(j) Mi,j)  
  Se Vr = Vc il gioco ha soluzione, ed è uguale al payoff V e le strategie sono ottimali.
* Definisci Vr, Vc, il teorema di Von Neumann e il valore di un gioco  
  - Vr = max(i) min(j) Mi,j - Vc = min(j) max(i) Mi,j

- disuguaglianza fondamentale: max(i) min(j) Mi,j <= min(j) max(i) Mi,j

- teorema di Von Neumann: ciò che non si può ottenere con una strategia fissa lo si può  
con una mista → se esiste mista esiste strategia random per cui il gioco ha valore)

- Somma zero = descrive una situazione in cui il guadagno o la perdita di un partecipante è perfettamente bilanciato da una perdita o un guadagno di un altro partecipante.

(- Se il gioco ha valore (e a somma zero) a lungo andare vanno pari.)  
Ad esempio, se C gioca una strategia mista q = (⅓ ⅓ ⅓ )T la moltip. Mq tra la matrice e la strategia è = (-⅓ ⅔ -⅓ )T e dunque R massimizza il payoff con la strategia pura p = (0 1 0)T → moltiplicando pTMq = Vr = ⅔ (valore atteso payoff R)

* Caratterizza la teoria con righe = input x alg. / colonne = possibili alg. det. / M(i,j): t esecuz.  
  Per il teorema di Yao, fissata una distribuzione p degli input, il tempo di esecuzione atteso del miglior algoritmo deterministico è un limite inferiore per il tempo di esecuzione atteso di un qualunque algoritmo randomizzato. Dunque per ottenere tale limite inferiore possiamo scegliere una distribuzione delgi input p conveniente e studiare la complessità dell’algoritmo deterministico ottimo su p. (minA E[T(Ip, A)] <= maxI E[T(I, Aq)]) : max perché caso peggiore
* Parla dell’albero di un gioco e fai un esempio con AND/OR  
  Considerato un gioco in cui non esiste il pareggio si può costruire un modello ad albero per il gioco, in cui gli archi rappresentano le mosse da una configurazione all’altra. Dato un k e d mosse un albero T(d, k) ha d2k foglie a distanza 2k dalla radice, i nodi interni a distanza pari dalla radice sono di tipo MIN e viceversa e il valore del gioco è un numero associato a ogni foglia. Il problema della valutazione del gioco consiste nel determinare il valore della radice risalendo l’albero da sinistra verso destra. Se i valori possibili per le foglie sono 0 e 1 MAX = OR e AND = MIN, e per la valutazione si seguono le usuali definizioni degli operatori booleani. Nel caso binario con radice AND, ad esempio, avremo (caso base con k = 1):

- se AND = 1: ½ (1 + 2) + ½ (1 + 2) = 3 - se AND = 0: ½ (2) + ½ (2 + 2) = 3

→ passo induttivo: dimostro che <= 3k

- lim\_inf ottenibile con Depth First Pruning (vedi es del NOR), in cui non esploro sottoalberi senza info utile → il limte inf lascia spazio per migliorare rispetto al costo sublineare del tempo di esecuzione atteso (inferiore comunque al caso peggiore di un alg deterministico)?

NO, perché nel modello non ho considerato la dipendenza tra i nodi