



Definizioni

Limite superiore

$0 \leq T(n) = O(f(n))$ se esistono due costanti $c > 0$ e $n_0 \geq 0$ tali che per tutti gli $n \geq n_0$ si ha che $T(n) \leq c \cdot f(n)$.

Limite inferiore

$0 \leq T(n) = \Omega(f(n))$ se esistono due costanti $c > 0$ e $n_0 \geq 0$ tali che per tutti gli $n \geq n_0$ si ha che $T(n) \geq c \cdot f(n)$.

Limite esatto

$T(n)$ è $\Theta(f(n))$ se $T(n)$ è sia $O(f(n))$ che $\Omega(f(n))$.

Componente connessa

Una componente connessa è un sottoinsieme di vertici tale che per ciascuna coppia di vertici (u, v) esiste un percorso tra u e v .

Connettività forte

I nodi u e v sono mutualmente raggiungibili se c'è un percorso da u a v e anche un percorso da v ad u . Un grafo in cui ogni coppia di nodi è mutualmente raggiungibile si dice fortemente connesso.

DAG

Un DAG è un grafo direzionato che non contiene cicli direzionati.

Grafo Bipartito

Un grafo non direzionato è bipartito se l'insieme di nodi può essere partizionato in due sottoinsiemi X e Y , tali che ciascun arco del grafo ha una delle estremità in X e una in Y .

Ordinamento Topologico

Un ordinamento topologico di un grafo direzionato $G = (V, E)$ è un'etichettatura dei nodi v_1, v_2, \dots, v_n tale che se G contiene l'arco (v_i, v_j) si ha $i < j$. Praticamente, se c'è l'arco (u, v) , allora u precede v nell'ordinamento.

Correttezza di Dijkstra

Teorema: sia G un grafo in cui per ogni arco e è definita una lunghezza $l_e \geq 0$. Per ogni nodo $u \in S$, il valore $d(u)$ calcolato dall'algoritmo di Dijkstra è la lunghezza del percorso più corto da s a u .

Dimostrazione: per dimostrare questo teorema, usiamo l'induzione sulla cardinalità di S , l'insieme dei nodi esplorati. Il caso base è quando $|S| = 1$, cioè quando S contiene solo il nodo sorgente s . In questo caso, banalmente, la distanza $d(s)$ sarà 0, poiché il percorso più corto da s a s è ovviamente 0. Per il passo induttivo, supponiamo che, per i primi k nodi nell'insieme S , l'algoritmo abbia già calcolato correttamente le distanze. Aggiungiamo un nuovo nodo v , quindi $|S| = k + 1$. Dobbiamo dimostrare, quindi, che Dijkstra ha scelto v con la distanza corretta, che quindi $d(v)$ sia, di fatto, la distanza minima da s a v . Noi sappiamo che Dijkstra sceglie il nodo v con la distanza minima tra quelli non

esplorati, quindi $d(v) = \min\{d(u) + l_{uv}\}$. Consideriamo un qualsiasi altro percorso P da s a v e chiamiamo (x, y) il primo arco di questo percorso che esce da S . Dividiamo il percorso in due parti: P' è il sottopercorso da s a x e (x, y) è l'arco che collega x a y . Per ipotesi induttiva, sappiamo che $d(x)$ è già la distanza minima da s a x , quindi qualsiasi percorso non può essere più corto di quello scelto da Dijkstra, quindi $l(P') \geq d(x)$. Aggiungendo (x, y) , quindi, si avrà che $l(P') + l(x, y) \geq d(x) + l(x, y)$. Quindi concludiamo che Dijkstra ha scelto v poiché $d(v) \leq d(y)$, per ogni altro nodo y non ancora esplorato.

Earliest Finish Time è ottimo

1Teorema: l'algoritmo greedy basato sulla strategia *Earliest Finish Time* è ottimo.

Dimostrazione: per dimostrare questa asserzione, conviene dimostrare due cose. La prima è che il tempo di fine dei job selezionate dall'algoritmo greedy non è mai peggiore del tempo di fine dei corrispondenti job della soluzione ottima. Quindi, per ogni job i_r , scelto da greedy, il tuo tempo di fine $f(i_r)$ è minore o uguale al tempo di fine dei job j_r nella soluzione ottima. La seconda cosa che andremo a dimostrare è che greedy seleziona almeno lo stesso numero di job della soluzione ottima. Usiamo l'induzione sul numero r di job selezionati per dimostrare che $f(i_r) \leq f(j_r)$. Il caso base è quando $r = 1$, cioè quando abbiamo un solo job. Banalmente, greedy e la soluzione ottima, devono per forza scegliere entrambe quello. Per il passo induttivo, supponiamo che per $r - 1$, valga già $f(i_{r-1}) \leq f(j_{r-1})$. Consideriamo il prossimo job j_r nella soluzione ottima. Per definizione della compatibilità, j_r inizia dopo che j_r è terminato, quindi $f(j_{r-1}) \leq s(j_r)$. Ma, per ipotesi induttiva, $f(i_{r-1}) \leq f(j_{r-1})$, quindi $f(i_{r-1}) \leq s(j_r)$. Questo vuol dire che il job j_r è compatibile con tutti i job i_1, i_2, \dots, i_{r-1} selezionati da greedy. Supponiamo, ora, per assurdo, che greedy selezioni meno job rispetto alla soluzione ottima. Siano

k i job selezionati da greedy e m i job selezionati dall'ottimo, con $k < m$.

Supponiamo, quindi, che $f(i_k) \leq f(j_k)$. Questo implica che il job successivo della soluzione ottima, j_{k+1} , deve iniziare dopo che tutti i job i_1, i_2, \dots, i_k greedy sono terminati. Quindi, j_{k+1} è compatibile con tutti i job già selezionati dal greedy. Ma questo significa che greedy avrebbe potuto selezionare j_{k+1} , poiché compatibile con tutto ciò che è già stato scelto. Questo è un assurdo, perché

greedy seleziona sempre tutte le attività compatibili disponibili. Concludiamo, quindi, che greedy rappresenta la soluzione ottima.

Inversione

Un'inversione, in uno scheduling S , è una coppia di job i e j tali che $d_i < d_j$, ma j viene eseguito prima di i .

Idle Time

Un idle time è un momento in cui la risorsa non viene utilizzata.

Teorema Inversioni e Idle-Time

Teorema: in uno scheduling senza inversioni e senza *idle time*, job con la stessa scadenza vengono eseguiti uno dopo l'altro.

Dimostrazione: consideriamo due job i e j che hanno la stessa scadenza, cioè $d_i = d_j = d$ e supponiamo, senza perdita di generalità, che i venga eseguito prima di j . Supponiamo per assurdo che tra i e j venga eseguito un altro job q con una scadenza diversa da d . Esistono due possibilità. Nella prima, $d_q > d$, ma eseguire q prima di completare j costituisce un'inversione e questo contraddice la proprietà di assenza di inversioni. Nel secondo caso, $d_q < d$, significa che il job i viene eseguito prima di q , ma q ha una scadenza ravvicinata e ci troveremo di fronte ad un'altra inversione. Quindi, nessun job con scadenza diversa da d può essere eseguito tra i e j .

Interval Scheduling

Input: un'istanza del problema di Interval Scheduling consiste in un insieme di job di cui si conosce l'istante di inizio s_j e l'istante di fine f_j e può essere eseguito solo un job per volta.

Obiettivo: vogliamo eseguire tutti i job nel minor tempo possibile. Diremo che due job i e j sono compatibili se $f_i \leq s_j$ oppure $f_j \leq s_i$, quindi l'obiettivo è trovare un sottoinsieme di cardinalità massima di due job a due a due compatibili.

Partizionamento di Intervalli

Input: un'istanza del problema consiste in un insieme di n intervalli $[s_1, f_1], [s_2, f_2], \dots, [s_j, f_j]$ dove s_j rappresenta il tempo di inizio e f_j il tempo di fine. Ad ogni risorsa d può essere assegnato al massimo un job per volta.

Obiettivo: vogliamo far eseguire tutte le attività col minor numero possibile di risorse, facendo sì che ad ogni risorsa venga assegnato un job per volta.

Minimizzazione dei ritardi

Input: un'istanza del problema consiste in un insieme di job di cui si conosce il tempo di esecuzione t_j e la relativa scadenza di ogni singolo job, rappresentata da d_j . Se un job inizia ad un istante s_j , allora $f_j = s_j + t_j$.

Obiettivo: vogliamo trovare uno scheduling che garantisca il minimo ritardo massimo $L = \max\{l_j\}$. Per il ritardo di un job intendiamo $l = \max\{0, f_j - d_j\}$.

Ottimalità del Taglio del Tubo

Idea: per l'ottimalità del taglio del tubo, proponiamo un algoritmo dove vengono ordinate le lunghezze dei segmenti in ordine non decrescente, ottenendo $lung'[1], lung'[2], \dots, lung'[n]$. Tagliamo dall'inizio, fino a che non possiamo ottenere altri segmenti, ovvero finché la somma delle lunghezze dei segmenti non supera L . Quindi, l'algoritmo greedy seleziona i segmenti più corti per primi. Vogliamo dimostrare che la strategia greedy garantisce una soluzione ottima, basandosi sulla tecnica dello scambio. L'idea è che, se esiste una soluzione ottima o_1, o_2, \dots, o_q , possiamo progressivamente trasformarla nella soluzione greedy g_1, g_2, \dots, g_p , mantenendo la stessa qualità nella soluzione.

Dimostrazione: supponiamo che la strategia greedy produca g_1, g_2, \dots, g_p segmenti, mentre la soluzione ottima produca o_1, o_2, \dots, o_q segmenti. Supponiamo che fino ad un certo indice j , le due soluzioni abbiano selezionato gli stessi segmenti, cioè $g_1 = o_1, g_2 = o_2, \dots, g_j = o_j$. Vogliamo dimostrare che è possibile sostituire il segmento o_{j+1} con g_{j+1} nella soluzione ottima senza peggiorare la soluzione. Nel caso in cui $g_{j+1} = o_{j+1}$, la dimostrazione è già

conclusa, poiché le due soluzioni coincidono per $j + 1$ scelte. Invece, nel caso in cui $g_{j+1} \neq o_{j+1}$, sappiamo che $g_{j+1} \leq o_{j+1}$ perché l'algoritmo greedy sceglie sempre i segmenti più corti disponibili. Pertanto, se sostituiamo o_{j+1} con g_{j+1} , la parte restante del tubo sarà più grande di quella che si aveva prima della sostituzione. Questo implica che le scelte successive $o_{j+2}, o_{j+3}, \dots, o_q$ sono ancora valide, poiché rimane sufficiente tubo per completare il taglio. Ne segue, quindi, che la soluzione greedy proposta è ottima.

Interval Scheduling Pesato

Input: un'istanza del problema consiste in un insieme di job j , di cui si conosce il tempo di inizio s_j , il tempo di fine f_j e il relativo peso v_j .

Obiettivo: vogliamo trovare il sottoinsieme di job compatibili con il massimo peso totale.

OPT(j): è il valore della soluzione ottima per l'istanza del problema, costituita dalle j richieste con i j tempi di fine più piccoli.

$$OPT(j) = \begin{cases} 0 & \text{se } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j - 1)\} & \text{altrimenti} \end{cases}$$

1. se $j = 0$, significa che non abbiamo nessun job.
2. Possono verificarsi due casi:
 - a. nel primo caso, la soluzione include il job j col suo peso v_j , quindi l' OPT è uguale a $v_i + OPT(p(j))$.
 - b. nel secondo caso, la soluzione non include il job j , quindi l' OPT è uguale a $OPT(j - 1)$.

SubsetSum

Input: un'istanza del problema consiste in un insieme di job ognuno dei quali richiede tempo $w_i > 0$ e un limite W al tempo di utilizzo del processore.

Obiettivo: vogliamo trovare un sottoinsieme S degli n job tali che $\sum_{i \in S} w_i$ sia

quanto più grande possibile, con il vincolo $\sum_{i \in S} w_i \leq W$.

OPT(i, w): è il valore della soluzione ottima per i job $1, \dots, i$ con limite w sul tempo di utilizzo del processore.

$$OPT(i, w) =$$

$$\begin{cases} 0 & \text{se } i = 0 \\ OPT(i - 1, w) & \text{se } w_i > w \\ \max\{OPT(i - 1, w), w_i + OPT(i - 1, w - w_i)\} & \text{altrimenti} \end{cases}$$

1. se $i = 0$, banalmente non abbiamo nessun job.
2. se $w_i > w$, significa che il job i non può far parte della soluzione ottima perché richiede più tempo di quello a disposizione.
3. altrimenti, possono verificarsi due casi.
 - a. nel primo caso, la soluzione ottima non include il job i e, quindi, passiamo al precedente, con $OPT(i - 1, w)$.
 - b. nel secondo caso, la soluzione ottima include il job i col suo peso, quindi $w_i + OPT(i - 1, w - w_i)$.

Problema dello zaino

Input: un'istanza del problema consiste in un insieme di n oggetti, ognuno dei quali con peso $w_i > 0$ e valore $v_i > 0$ e un limite di peso W .

Obiettivo: vogliamo riempire lo zaino in modo da massimizzare il valore totale degli oggetti inseriti, senza eccedere con il limite W .

OPT(i, w): è il valore della soluzione ottima per gli oggetti $1, \dots, i$ con limite di peso w .

$$OPT(i, w) =$$

$$\begin{cases} 0 & \text{se } i = 0 \\ OPT(i - 1, w) & \text{se } w_i > w \\ \max\{OPT(i - 1, w), v_i + OPT(i - 1, w - w_i)\} & \text{altrimenti} \end{cases}$$

1. se $i = 0$, banalmente non abbiamo nessun oggetto.
2. se $w_i > w$, significa che l'oggetto i non può far parte della soluzione ottima perché richiede più peso di quello a disposizione.
3. altrimenti, possono verificarsi due casi:
 - a. nel primo caso, la soluzione ottima non include l'oggetto i , quindi passiamo al precedente, con $OPT(i - 1, w)$.
 - b. nel secondo caso, la soluzione ottima include l'oggetto i col suo valore, quindi $v_i + OPT(i - 1, w - w_i)$.

Minimum Coin Change

Input: un'istanza del problema consiste in un insieme di monete $v_1 < v_2 < \dots < v_n$ e una somma di denaro in banconote V .

Obiettivo: vogliamo calcolare il minimo numero di monete richieste per cambiare la somma di denaro V .

OPT(i, v): è il minimo numero di monete per cambiare una banconota di valore v quando abbiamo a disposizione monete di valore v_1, v_2, \dots, v_i .

$$OPT(i, v) = \begin{cases} 0 & \text{se } v = 0 \\ V & \text{se } i = 1 \\ OPT(i - 1, v) & \text{se } v_i > v \\ \min\{OPT(i - 1, v), 1 + OPT(i, v - v_i)\} & \text{altrimenti} \end{cases}$$

1. se $v = 0$, il valore della banconota è 0, quindi sono richieste, di fatto, 0 monete.
2. se $i=1$, abbiamo soltanto la moneta dal valore 1, quindi per cambiare il valore di v , servirà esattamente V .
3. se $v_i > v$, significa che la moneta v_i ha un valore più grande di v , quindi non fa parte della soluzione ottima.
4. altrimenti, possono verificarsi due casi:

- a. la moneta i non fa parte della soluzione ottima, quindi passiamo al valore precedente.
- b. la moneta i fa parte della soluzione ottima, quindi aggiungiamo 1 al numero di monete richieste per ottenere il valore rimanente $v - v_i$.
Aggiungiamo 1 poiché useremo una moneta in più rispetto alla soluzione ottima.

Bellman-Ford

Input: un'istanza del problema consiste in un grafo $G = (V, E)$, dove V è l'insieme dei nodi ed E l'insieme degli archi e un nodo destinazione t .

Obiettivo: vogliamo calcolare i cammini minimi dal nodo destinazione t verso tutti gli altri nodi $v \in V$.

OPT(i, v): è la lunghezza del cammino più corto p per andare da v a t , che consiste in al più i archi.

$$OPT(i, v) = \begin{cases} 0 & \text{se } v = t \\ \infty & \text{se } i = 0 \\ \min_{(v,w) \in E} \{ OPT(i-1, w) + c_{vw} \} & \text{altrimenti} \end{cases}$$

1. se $v = t$, banalmente, nodo di destinazione e nodo di partenza coincidono, quindi non serve percorrere nessun arco, quindi 0.
2. se $i = 0$, significa che abbiamo a disposizione 0 passi, quindi non possiamo muoverci, quindi avremo un costo infinito.
3. altrimenti, il costo minimo per raggiungere il nodo v , con i passi, è determinato considerando tutti i nodi w adiacenti a v . Quindi scegliamo il minimo tra il costo per arrivare a w più il costo dell'arco (v, w) .

Sottosequenza comune più lunga

Input: un'istanza del problema consiste in due stringhe x e y , composte dai caratteri x_1, x_2, \dots, x_m e y_1, y_2, \dots, y_n .

Obiettivo: vogliamo trovare la sottosequenza comune più lunga z , composta dai caratteri z_1, z_2, \dots, z_k che è presente in entrambe le stringhe nell'ordine relativo, ma non necessariamente consecutiva.

OPT(i, j): è la lunghezza della sottosequenza comune più lunga a x_1, x_2, \dots, x_i e y_1, y_2, \dots, y_j .

$$OPT(i, j) = \begin{cases} 0 & \text{se } i = 0 \text{ || } j = 0 \\ OPT(i - 1, j - 1) + 1 & \text{se } x_i = y_j \\ \max\{OPT(i - 1, j), OPT(i, j - 1)\} & \text{altrimenti} \end{cases}$$

1. se $i = 0$ o $j = 0$, significa che la sottosequenza comune è vuota, poiché abbiamo, di fatto, una delle stringhe che non ha caratteri.
2. se $x_i = y_j$ significa che c'è un match tra i caratteri delle due stringhe e che, quindi, il carattere fa parte della sottosequenza comune più lunga e lo includiamo andando a sommare 1 alla sua diagonale.
3. altrimenti, non c'è un match, quindi i caratteri non fanno parte della soluzione ottima, quindi andremo a prendere il massimo tra l'elemento precedente di x e quello di y .

MergeSort

Input: l'algoritmo prende in input un array, la parte destra e la parte sinistra

Obiettivo: vogliamo ordinare un vettore di numeri in ordine crescente, dividendo quest'ultimo in parti sempre più piccole, per poi unirli tra di loro.

$$T(n) = \begin{cases} c_0 & \text{se } n \leq 1 \\ 2T(\frac{n}{2}) + cn + c' & \text{altrimenti} \end{cases}$$

Questa relazione di ricorrenza descrive il tempo di esecuzione dell'algoritmo al variare della dimensione dell'input n . Il primo caso è il caso base ed indica che l'array, essendo composto da un elemento, è già ordinato e non è necessario fare alcuna operazione, quindi il tempo impiegato è costante. Poi, abbiamo il caso ricorsivo, dove l'array di dimensione n viene diviso in due sottosequenze di

lunghezza $\frac{n}{2}$ ed esso viene chiamato ricorsivamente due volte. cn , invece, rappresenta il costo della procedura di fusione, che richiede tempo lineare n . c' , invece, rappresenta il costo di operazioni aggiuntive, come il calcolo degli indici centrali o il controllo delle condizioni di terminazione.

Ricerca Binaria Ricorsiva

Input: l'algoritmo prende in input l'array A , un elemento da cercare k e gli indici di sinistra e destra.

Obiettivo: vogliamo trovare, all'interno di una sequenza ordinata, un determinato elemento.

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \text{ || } k = cen \\ T\left(\frac{n}{2}\right) + c & \text{altrimenti} \end{cases}$$

Questa relazione di ricorrenza descrive il tempo di esecuzione dell'algoritmo al variare della dimensione dell'input n . Il primo caso è il caso base. Se l'array ha un solo elemento, allora si può fare il confronto con k direttamente, oppure se k si trova subito nella posizione centrale, allora la ricerca può terminare. Anche qui, il tempo è costante. Ogni volta che si effettua una chiamata ricorsiva, la dimensione dell'input si dimezza e bisogna aggiungere un costo aggiuntivo relativo al calcolo dell'indice centrale e del confronto, anch'esso costante.

QuickSort

Input: l'algoritmo prende in input l'array A , e gli indici di destra e di sinistra.

Obiettivo: vogliamo ordinare l'array.

$$T(n) \leq \begin{cases} c_0 & \text{per } n \leq 1 \\ T(r-1) + T(n-r) + cn & \text{altrimenti} \end{cases}$$

Questa relazione di ricorrenza descrive il tempo di esecuzione dell'algoritmo al variare della dimensione dell'input n . Il primo caso è il caso base. Se l'array ha un elemento, significa che è già ordinato ed è richiesto sempre tempo costante. Nel caso ricorsivo, l'array viene diviso in due metà attorno al pivot r , mentre cn

rappresenta il tempo costante che ci vuole per la fase di distribuzione, in cui l'array viene scomposto intorno al pivot.

QuickSelect

Input: l'algoritmo prende in input l'array A, l'indice di sinistra e destra e il rango dell'elemento che si vuole cercare

Obiettivo: vogliamo trovare l'elemento nella posizione del rango.

$$T(n) \leq \begin{cases} c_0 & \text{per } n = 1 \\ c_1 n & \text{per se } n > 1, r_p = r \\ \max\{T(r_p - 1), T(n - r_p)\} + cn & \text{altrimenti} \end{cases}$$

Questa relazione di ricorrenza descrive il tempo di esecuzione dell'algoritmo al variare della dimensione dell'input n . Il primo caso è il caso base. Se l'array ha un elemento, non serve effettuare ulteriori operazioni e questo ho tempo costante c_0 . Se il pivot scelto si trova esattamente nella posizione r , allora la distribuzione restituisce direttamente il risultato, senza effettuare operazioni aggiuntive. Se il pivot non si trova nella posizione r , l'algoritmo continua a cercare in una delle due sequenze divise dalla distribuzione, quindi il tempo di esecuzione totale è il massimo tempo necessario per una delle due metà, più il tempo costante della distribuzione.