Progettazione di algoritmi

Discussione dell'esercizio [palindroma]

Dobbiamo trovare un algoritmo efficiente che data una stringa *s* di *n* caratteri trova la più lunga sottostringa di *s* che sia palindroma.

È chiaro che un algoritmo esaustivo dovrebbe considerare tutte le possibili sottostringhe che sono $\Theta(n^2)$ e per ognuna verificare se è o meno palindroma e quindi avrebbe complessità $O(n^3)$. Ma con la Programmazione Dinamica speriamo di fare di meglio. Essendo un problema relativo a una sequenza o stringa, proviamo a considerare come sotto-problemi quelli relativi ai prefissi:

$$P[i] = \text{massima lunghezza di una palindroma che è una sottostringa di } s[1 \cdots i]$$

Il caso base è ovviamente $P[1] = 1$. Per calcolare $P[i]$ dobbiamo considerare i due casi possibili: o la più lunga palindroma di $s[1 \cdots i]$ non comprende l'elemento $s[i]$ oppure lo comprende. Nel primo caso $P[i] = P[i\text{-}1]$, ma nel secondo caso dovremmo trovare la più lunga palindroma di $s[1 \cdots i]$ che termina nella posizione i. Però la tabella P non ci dà informazioni per determinare tale lunghezza in modo efficiente. Calcolandola direttamente otterremmo un algoritmo di complessità cubica e noi vorremo fare di meglio. Allora modifichiamo i sotto-problemi da considerare in modo che sia facile calcolare la più lunga palindroma che termina in una determinata posizione:

$$P[i] = \text{massima lunghezza di una palindroma che termina nella posizione } i \text{ di } s$$

Come calcolare $P[i]$? Per prima cosa, possiamo osservare che $P[i] \le P[i\text{-}1] + 2$. Infatti, se fosse $P[i] > P[i\text{-}1] + 2$ allora la sottostringa $s[i\text{-}P[i]+2 \cdots i\text{-}1]$ sarebbe una palindroma che termina in i-1 di lunghezza i - 1 - (i - $P[i]$ + 2) + 1 = $P[i]$ - 2 > $P[i\text{-}1]$, contraddizione. Quindi se $s[i] = s[i\text{-}P[i\text{-}1]\text{-}1]$ allora abbiamo che $P[i] = P[i\text{-}1] + 2$. Ma se $s[i] \neq s[i\text{-}P[i\text{-}1]\text{-}1]$, come facciamo a calcolare $P[i]$? Non si riesce a trovare un modo efficiente. Dobbiamo cambiare i sotto-problemi da considerare. Quando i prefissi falliscono si può tentare con le sottostringhe. Essendo le soluzioni possibili proprio delle sottostringhe, conviene definire i sotto-problemi nel modo seguente: per $1 \le i \le j \le n$,

$$P[i,j] = \begin{cases} true & \text{se la sottostringa } s[i \cdots j] \text{ è palindroma} \\ false & \text{altrimenti} \end{cases}$$

Quello che ci interessa è

$$\max\{j - i + 1 \mid 1 \leqslant i \leqslant j \leqslant n \wedge P[i,j] = true\}$$

I casi base sono $P[i, i] = true$, per $i = 1,\ldots,n$. Vediamo come calcolare $P[i, j]$ quando j - $i \ge 1$. Una condizione necessaria affinché $P[i, j]$ sia *true* è che $s[i] = s[j]$. Se j - i = 1, cioè $j = i + 1$, quella condizione è anche sufficiente. Se invece j - $i \ge 2$, perchè sia $P[i, j] = true$ deve anche essere $P[i+1, j\text{-}1] = true$. Per $1 \le i \le j \le n$,

$$P[i,j] = \begin{cases} true & \text{se } j - i \leqslant 1 \text{ e } s[i] = s[j] \\ P[i+1, j-1] & \text{se } j - i > 1 \text{ e } s[i] = s[j] \\ false & \text{altrimenti} \end{cases}$$

Il programma che calcola la tabella P deve procedere dalle sottostringhe più corte verso quelle più lunghe in modo simile al calcolo della tabella del Matrix Chain Multiplication Problem:

```
PAL(s: stringhe lunga n)
    P: tabella nxn
    FOR i <- 1 TO n DO P[i, i] <- true
    max <- 1
    FOR k <- 1 TO n DO
        FOR i <- 1 TO n - k DO
            j <- i + k
            IF s[i] = s[j] THEN
                IF k = 1 THEN
                    P[i, j] <- true
                ELSE
                    P[i, j] <- P[i+1, j-1]
```

```
            ELSE
                P[i, j] <- false
            IF P[i, j] AND k ≥ max THEN
                max <- k + 1
    RETURN max
```

L'algoritmo ha complessità $O(n^2)$. Si può facilmente modificare il programma `PAL` cosicché ritorni anche gli estremi della massima sottostringa palindroma. Basta registrare, ogni volta che `max` viene aggiornato, anche gli estremi della sottostringa che ha migliorato il valore di `max` .

Spazi di ricerca

Durante il corso abbiamo incontrato vari esempi di problemi difficili, cioè problemi per cui non si conoscono algoritmi efficienti capaci di risolverli. Per questo tipo di problemi si conoscono solamente algoritmi esaustivi che generano tutte le soluzioni possibili alla ricerca di una soluzione ottima. Ad esempio, per il problema Ricoprimento tramite Nodi (Vertex Cover), si tratterebbe di generare tutti i possibili sottoinsiemi di nodi e per ognuno controllare se è un ricoprimento. Per il problema del Ciclo Hamiltoniano si dovrebbero generare tutte le possibili permutazioni dei nodi del grafo e per ognuna controllare se è un ciclo. Per la 3-Colorazione di un grafo dovremmo generare tutte le possibili colorazioni con 3 colori dei nodi e per ognuna controllare se è una colorazione ammissibile. Per il problema dello Zaino (Knapsack), dovremmo generare tutti i possibili sottoinsiemi degli oggetti e per ognuno controllare se ha peso non eccedente la capacità e in tal caso confrontare il valore con quello più alto finora trovato.

Alla base degli algoritmi esaustivi c'è quindi la generazione di tutte le possibili strutture di un certo tipo: sottoinsiemi, permutazioni, 3-colorazioni, ecc. Per la maggior parte dei problemi non è necessario generare proprio tutte le soluzioni possibili. Ad esempio, durante la generazione delle permutazioni per trovare un Ciclo Hamiltoniano possiamo incontrare il caso che l'inizio di una permutazione, diciamo 1,4,2,…, già preclude la possibilità che possa produrre un ciclo perchè nel grafo non c'è l'arco (4, 2). Così possiamo evitare di generare tutte le permutazioni che iniziano con 1,4,2. In questi casi si dice che è stato effettuato un **taglio** dello **spazio di ricerca** di tutte le soluzioni possibili. I miglioramenti, a volte molto grandi, che possono essere apportati ad un algoritmo esaustivo sono di questo tipo, cioè sono tagli dello spazio di ricerca. I tagli possono essere basati su considerazioni e idee molto semplici, ma comunque efficaci, o su algoritmi piuttosto complessi. Tuttavia, non importa quanto siano raffinati i metodi di taglio, gli algoritmi esaustivi hanno uno scheletro portante che è la generazione di tutte le soluzioni. Inoltre, mentre i metodi di taglio sono strettamenti legati al particolare problema, le strutture che devono essere generate per rappresentare le soluzioni possibili sono di solito comuni a tantissimi problemi. Ad esempio, i sottoinsiemi sono le strutture che rappresentano le soluzioni di tanti problemi come Ricoprimento tramite Nodi e Zaino. Per queste ragioni ci concentreremo, almeno per adesso, su come generare strutture semplici come sottoinsiemi, permutazioni e affini che rappresentano però le soluzioni possibili di moltissimi problemi.

Backtracking

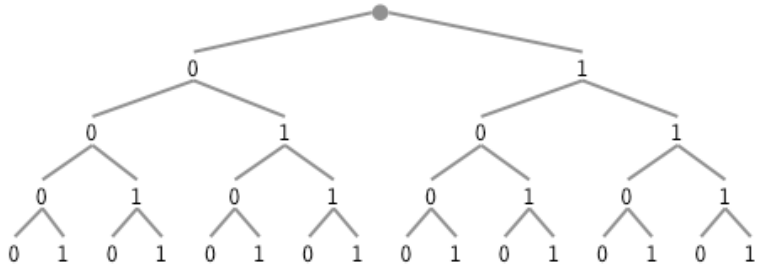
Come possiamo generare tutte le strutture di un certo tipo, ad esempio tutti i sottoinsiemi di un dato insieme? Ovviamente potrebbero esserci molti modi diversi di farlo che potrebbero essere però ad-hoc per un particolare tipo e non essere adattabili ad altri tipi. Tuttavia c'è un metodo generale che può essere facilmente adattato per generare tutti i principali tipi di strutture. Affinché un tipo di struttura sia generabile con tale metodo è sufficiente che ogni struttura di quel tipo sia rappresentabile con una sequenza $(a_1,a_2,\ldots,a_n)$ dove gli a_i sono elementi di un certo dominio A. Per semplicità assumeremo che la lunghezza n delle sequenze sia uguale per tutte le strutture. Vediamo alcuni esempi.

- Ogni sottoinsieme B dell'insieme $\{1,2,\ldots,n\}$ può essere rappresentato tramite una sequenza $(b_1,b_2,\ldots,b_n)$ dove $b_i \in \{0,1\}$. Il significato è chiaro, $b_i = 1$ se e solo se $i \in B$.
- Ogni 3-colorazione di un grafo con nodi $\{1,\ldots,n\}$ può essere rappresentata da una sequenza $(c_1,c_2,\ldots,c_n)$ dove $c_i \in \{$ r , g , b $\}$ è il colore assegnato al nodo i.
- Ogni permutazione di $(1,2,\ldots,n)$ è rappresentata tramite una sequenza $(p_1,p_2,\ldots,p_n)$ dove $p_i \in \{1,\ldots,n\}$ e $p_i \neq p_j$ se $i \neq j$.
- Ogni k-sottoinsieme B di $\{1,2,\ldots,n\}$, cioè un sottoinsieme di cardinalità k, può essere rappresentato con una sequenza $(b_1,b_2,\ldots,b_n)$ dove $b_i \in \{0,1\}$ e il numero di b_i uguali a 1 è k.

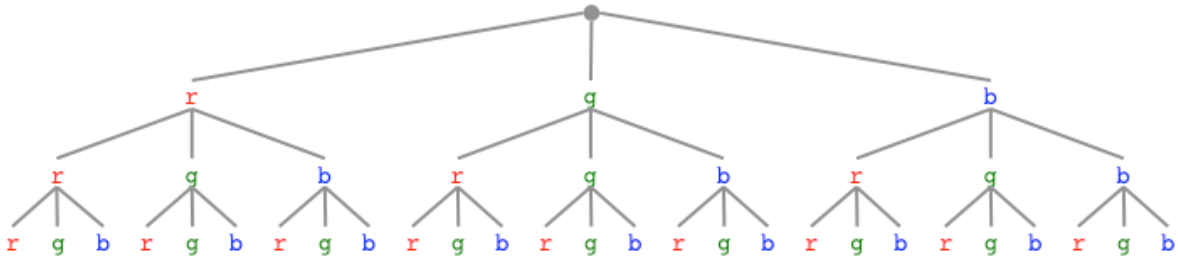
La generazione delle sequenze $(a_1,a_2,\ldots,a_n)$ parte dalla sequenza vuota e via via estende una sequenza parziale $(a_1,a_2,$

…,a_i) con un nuovo elemento ammissibile fino ad ottenere una sequenza completa e poi torna indietro sulle sequenze parziali per eventualmente estenderle con altri elementi ammissibili. La generazione termina quando si ritorna alla sequenza vuota. Questo modo di procedere è anche descrivibile come la visita in profondità dell'albero i cui nodi corrispondono ai prefissi delle sequenze da generare. La radice è la sequenza vuota, le foglie corrispondono alle sequenze complete e un nodo interno ha figli che corrispondono ai prossimi prefissi. Nel rappresentare tali alberi conviene etichettare ogni nodo con l'ultimo elemento del prefisso a cui corrisponde. Così il prefisso o la sequenza completa che corrisponde a un nodo è data dalle etichette dei nodi del cammino dalla radice a quel nodo.

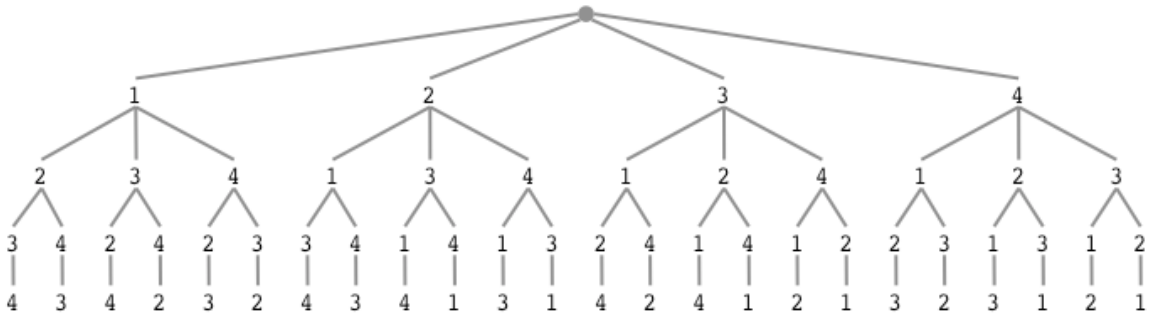
Vediamo alcuni esempi. L'albero per i sottoinsiemi di {1,2,3,4}:



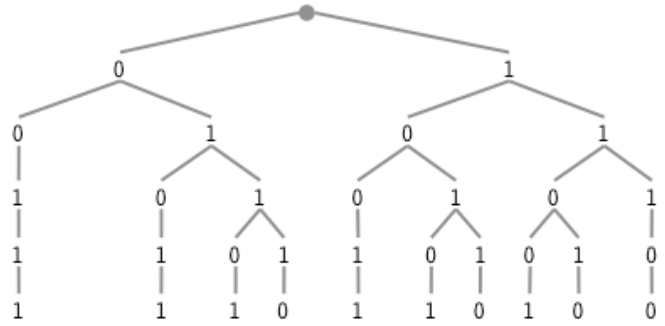
L'albero delle 3-colorazioni di un grafo di 3 nodi:



L'albero delle permutazioni di (1,2,3,4):



Infine, l'albero dei 3-sottoinsiemi di {1,2,3,4,5}:

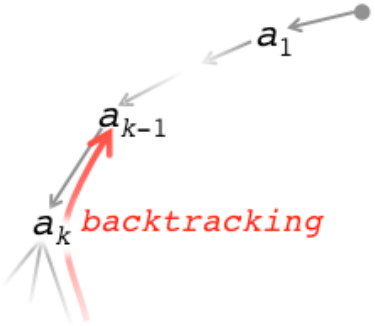


Quindi il metodo generale per generare tali tipi di strutture consiste nella visita in profondità (ovvero in preordine) dell'albero delle corrispondenti sequenze. Questa visita può essere convenientemente descritta tramite una procedura

ricorsiva che effettua la **visita di un albero implicito**. Lo schema generale della visita può essere descritto come segue:

```
GEN(n: lunghezza sequenze, h: lunghezza prefisso, S: sequenza)
    IF h = n THEN
        OUTPUT S  /* Stampa la sequenza S o elabora S */
    ELSE
        E <- elementi che possono estendere il prefisso S[1···h]
        FOR ogni x in E DO
            S[h+1] <- x
            GEN(n, h+1, S)      /* Genera tutte le sequenze con prefisso S[1···h+1] */
```

La prima chiamata è `GEN(n, 0, S)`. Questo metodo è chiamato **Backtracking**. Il nome proviene da quello che fa la procedura `GEN` quando per un prefisso di lunghezza *h* termina di considerare le possibili estensioni (gli elementi in `E`): *torna indietro* (cioè backtrack) al prefisso di lunghezza *h*-1.



La complessità della generazione non può essere inferiore al numero $\boldsymbol{S}(n)$ di tutte le sequenze generate, quindi la complessità è almeno $\Omega(\boldsymbol{S}(n))$. Ma generalmente è superiore a causa del lavoro svolto durante la visita dei nodi dell'albero. Ritorneremo su questo punto dopo aver visto alcuni esempi.

Generare sottoinsiemi, colorazioni, permutazioni e combinazioni

Come primo esempio di applicazione dello schema `GEN` consideriamo la generazione di tutti i sottoinsiemi di {1,…,*n*}:

```
SUBSETS(n: cardinalità, h: lunghezza prefisso, S: sottoinsieme)
    IF h = n THEN
        OUTPUT S
    ELSE
        FOR ogni b in {0, 1} DO
            S[h+1] <- b
            SUBSETS(n, h+1, S)
```

Il caso delle 3-colorazioni di un grafo con *n* nodi è molto simile:

```
COL3(n: numero nodi, h: lunghezza prefisso, S: colorazione)
    IF h = n THEN
        OUTPUT S
    ELSE
        FOR ogni c in {r, g, b} DO
            S[h+1] <- c
            COL3(n, h+1, S)
```

Per le permutazioni di (1,2,…,*n*) c'è qualche difficoltà dovuta al fatto di dover trovare gli elementi che possono estendere un prefisso di permutazione *S*[1···*h*]. Gli elementi che possono estendere *S*[1···*h*], cioè quelli che possono essere posti in *S*[*h*+1], sono esattamente gli elementi in {1,…,*n*} che non appaiono già in *S*[1···*h*]. Un modo molto semplice, anche se non il più efficiente, di determinare tali elementi è di usare un array *E* tale che *E*[*i*] = 1 se *i* è in

$S[1\cdots h]$ e 0 altrimenti.

```
PERM(n: lunghezza permutazione, h: lunghezza prefisso, S: permutazione)
    IF h = n THEN
        OUTPUT S
    ELSE
        E: array di dimensione n, inizializzato a 0
        FOR i <- 1 TO h DO
            E[S[i]] <- 1
        FOR i <- 1 TO n DO
            IF E[i] = 0 THEN
                S[h+1] <- i
                PERM(n, h+1, S)
```

Osserviamo che se $h < n$, viene sempre effettuata almeno una chiamata ricorsiva. Quindi, da ogni nodo visitato dell'albero si arriva sempre a una foglia che corrisponde ad una sequenza completa, cioè una permutazione.

Passiamo ora ai k-sottoinsiemi di {1,..,n}. Un k-sottoinsieme è anche chiamato una *combinazione* di n elementi presi k alla volta. Il numero di 1 delle sequenze generate deve sempre essere uguale a k, quindi per determinare i valori da assegnare a $S[h+1]$ dobbiamo controllare quali sono ammissibili. Sia ℓ il numero di 1 in $S[1\cdots h]$. Se $\ell = k$, allora l'unico valore ammissibile è 0. Se invece $\ell < k$, allora bisogna vedere se le successive $n - (h + 1)$ posizioni sono sufficienti per arrivare a k 1. Quindi, se $k - \ell \leq n - (h + 1)$, allora sono ammissibili sia 0 che 1, altrimenti è ammissibile solamente il valore 1. Per ragioni di efficienza conviene che il numero di 1 dei prefissi sia mantenuto durante la visita:

```
COMB(n: cardinalità insieme, k: cardinalità sottoinsieme, ℓ: numero di 1, h: lung. prefisso,
         S: k-sottoinsieme)
    IF h = n THEN
        OUTPUT S
    ELSE
        IF ℓ ≥ k - n + h + 1 THEN
            S[k+1] <- 0
            COMB(n, k, ℓ, h+1, S)
        IF ℓ < k THEN
            S[k+1] <- 1
            COMB(n, k, ℓ+1, h+1, S)
```

La prima chiamata è `COMB(n, k, 0, 0, S)`. Si osservi che da ogni nodo visitato dell'albero con $h < n$ viene effettuata sempre almeno una chiamata ricorsiva e quindi le foglie sono esattamente i nodi con $h = n$, cioè quelli che corrispondono ai k-sottoinsiemi.

Complessità della generazione

Come abbiamo già notato la complessità della generazione di un numero $\boldsymbol{S}(n)$ di sequenze non può essere inferiore a $\Omega(\boldsymbol{S}(n))$. Però generalmente è superiore perché il numero di nodi visitati può essere di molto superiore al numero delle sequenze generate e in ogni nodo visitato viene effettuato un certo lavoro. Per calcolare la complessità di un algoritmo di generazione basato sulla tecnica di Backtracking è utile il seguente risultato:

Sia n la lunghezza delle sequenze da generare, sia $O(f(n))$ il tempo richiesto dalla visita di un nodo interno e sia $O(g(n))$ il tempo richiesto dalla visita di una foglia. Se vengono visitati esclusivamente nodi che portano a sequenze generate, il tempo richiesto dalla visita è

$$\mathrm{O}((nf(n) + g(n))\mathbf{S}(n))$$

dove $\boldsymbol{S}(n)$ è il numero di sequenze generate.

Dimostrazione. Poiché un nodo dell'albero viene visitato solo se porta a sequenze generate, l'albero effettivamente visitato ha come foglie le $\boldsymbol{S}(n)$ sequenze generate. Ogni nodo interno visitato appartiene ad almeno uno dei cammini radice-foglia ciascuno dei quali ha lunghezza al più n. Quindi i nodi interni visitati sono al più $n\boldsymbol{S}(n)$. Il costo della visita è dato dalla somma dei costi di visita dei nodi interni e delle foglie. Da quanto detto il costo è $\mathrm{O}(n\boldsymbol{S}(n)f(n) + \boldsymbol{S}(n)g(n))$.

Applichiamo questo semplice risultato agli esempi che abbiamo visto. Per tutti gli esempi possiamo assumere che il costo di visita di una foglia sia O(n) (se vogliamo anche solo stampare la sequenza lunga n impieghiamo tempo O(n)), cioè O($g(n)$) = O(n). Si osservi che se il tempo impiegato dalla visita delle foglie è Θ(n) nessun algoritmo di generazione può impiegare meno di Ω($n\boldsymbol{S}(n)$).

Consideriamo l'esempio dei sottoinsiemi, la visita di un nodo interno nella procedura `SUBSETS` prende tempo O(1), cioè O($f(n)$) = O(1). Quindi l'algoritmo di generazione impiega tempo O($n\boldsymbol{S}(n)$) e siccome in questo caso $\boldsymbol{S}(n) = 2^n$, il tempo è O($n2^n$). Un ragionamento del tutto simile vale anche per le 3-colorazioni e il tempo impiegato dall'algoritmo `3COL` è O($n\boldsymbol{S}(n)$). Siccome le 3-colorazioni di n nodi sono 3^n, il tempo è O($n3^n$).

Per le permutazioni, il tempo impiegato per la visita di un nodo interno è O(n) e se applichiamo il risultato otteniamo O($n^2\boldsymbol{S}(n)$) = O($n^2n!$). Ma una analisi un po' più raffinata porta a un limite inferiore. Il numero totale dei nodi dell'albero può essere determinato contandoli livello per livello a partire dalla radice: 1, n, $n(n-1)$, $n(n-1)(n-2)$,… In generale al livello h ci sono $n!/(n-h)!$ nodi. Quindi il numero totale di nodi è

$$\sum_{h=0}^{n} \frac{n!}{(n-h)!} = n! \sum_{h=0}^{n} \frac{1}{(n-h)!} = n! \sum_{j=0}^{n} \frac{1}{j!} = n! \left(1 + \sum_{j=1}^{n} \frac{1}{j!}\right) \leqslant n! \left(1 + \sum_{j=0}^{n-1} \frac{1}{2^j}\right) = n! \left(1 + \frac{1-(1/2)^n}{1-1/2}\right) \leqslant 3n!$$

Perciò il tempo che l'algoritmo `PERM` impiega è O($nn!$) dato che il numero totale dei nodi, sia interni che foglie, è limitato da $3n!$ e per ognuno la visita costa O(n).

L'algoritmo `COMB` che genera i k-sottoinsiemi di {1,…,n} impiega tempo O(1) per visitare un nodo interno, quindi in totale prende tempo

$$\mathrm{O}(n\mathbf{S}(n)) = \mathrm{O}\left(n\binom{n}{k}\right)$$

Quindi in tutti gli esempi che abbiamo visto il tempo di generazione è asintoticamente ottimale essendo limitato da O($n\boldsymbol{S}(n)$) e per quanto detto sopra non può essere inferiore a Ω($n\boldsymbol{S}(n)$).

Esercizio [cassaforte]

Sappiamo che la combinazione di una cassaforte è composta da una sequenza di n cifre decimali (cioè `0` , `1` , `2` ,…, `9`) la cui somma ha valore k. Descrivere un algoritmo che, dati n e k, genera tutte le possibili combinazioni con questa proprietà. L'algoritmo deve avere complessità O($nC(n, k)$), dove $C(n, k)$ è il numero di combinazioni possibili.