Progettazione di algoritmi

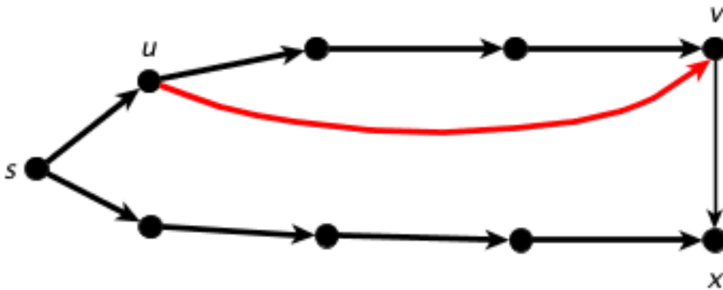
Breve discussione delle soluzioni della prova intermedia

Le soluzioni sono solamente abbozzate, non sono soluzioni complete.

Esercizio [DFS vs BFS] L'albero *T* non può essere stato prodotto da una BFS. Solo il nodo *c* può essere un nodo di partenza di una DFS che produce *T*. Da *f* o da *g* o da *a* gli alberi DFS devono contenere l'arco {*c*, *d*} mentre da *d* l'albero DFS deve contenere l'arco {*c*, *f*}.

Esercizio [Tarjan] Il grafo ha due compomenti una composta solamente dal nodo di partenza *a* e l'altra composta da tutti gli altri nodi. La componente grande è quella determinata prima e la sua c-root è o il nodo *b* o il nodo *i*.

Esercizio [update] L'algoritmo non è corretto come si vede dal seguente controesempio:



Esercizio [ciclo rosso] Si eliminano tutti i nodi blu e poi si applica l'algoritmo per trovare cicli al grafo rimasto.

Esercizio [cut] Sia *T* un qualsiasi MST di *G*. Se *T* non contiene l'arco *e*, c'è un ciclo *C* in $T \cup \{e\}$. Siccome l'arco *e* attraversa il taglio (*V*', *V* - *V*'), c'è almeno un arco *e*' del ciclo *C* che attraversa il taglio. L'arco *e*' può essere sostituito ad *e*, dato che $p(e) \leq p(e')$.

Esercizio [connessioni sicure] Rappresentiamo la rete tramite un grafo *G* i cui nodi sono i computer e vi è un arco tra due nodi se vi è un collegamento tra i corrispondenti computer. Inoltre, il peso di un arco è 0 se il corrispondente collegamento è sicuro ed è 1 altrimenti. Allora, il peso di un cammino è uguale al numero di archi insicuri attraversati dal cammino. Quindi il problema può essere risolto applicando Dijkstra a partire dal nodo che corrisponde al computer centrale *C*. L'algoritmo ha complessità $O((n + m)\log n)$.

Esercizio [treni] Il primo algoritmo non è corretto, basta considerare i tempi {1, 1.5, 2, 2.5, 3.5}. Il secondo è corretto e si può usare lo schema generale per ottenere una dimostrazione di correttezza.

Divide et Impera

Una delle prime tecniche algoritmiche che si incontrano è *Divide et Impera*. La si vede all'opera in algoritmi di ordinamento, come *quick-sort* e *merge-sort*, e nell'algoritmo di *ricerca binaria*. La tecnica quindi dovrebbe essere già abbastanza conosciuta ed anche per questo la trattazione sarà più breve rispetto a quella di altre tecniche.

La tecnica Divide et Impera cerca di "spezzare" l'istanza di un problema in istanze più piccole in modo tale che le soluzioni per queste sotto-istanze agevolino la costruzione della soluzione dell'istanza originale. Gli algoritmi che usano questa tecnica si prestano naturalmente ad essere implementati in modo ricorsivo. Ad esempio, ricordiamo che nell'algoritmo quick-sort l'array di input è diviso in due parti spostando nella prima parte tutti gli elementi minori di un elemento perno e nella seconda parte tutti quelli maggiori, poi l'algoritmo è chiamato ricorsivamente su entrambe le parti per ordinarle ottenendo così l'ordinamento dell'intero array. In questo caso, l'algoritmo fa un lavoro (lo spostamento degli elementi a seconda che siano minori o maggiori del perno) prima delle chiamate ricorsive. In altri casi, come nel merge-sort, viene fatto un lavoro dopo aver ottenuto le soluzioni delle sotto-istanze per costruire a partire da quest'ultime la soluzione dell'istanza di input. In questi algoritmi l'istanza di input è

partizionata in due parti (in generale, in due o più parti) e l'algoritmo è poi applicato ricorsivamente su ognuna delle parti, ma ci sono casi, come la ricerca binaria, in cui è sufficiente conoscere la soluzione di una sola delle parti e quindi l'algoritmo è applicato ricorsivamente solo su una di queste.

Quali sono i vantaggi offerti dalla tecnica Divide et Impera? Di solito, quando la tecnica è applicabile, non è difficile vedere *come* applicarla. Questo perchè, tipicamente, è facile intuire come si potrebbe spezzare l'istanza. Inoltre, è abbastanza facile, una volta che un algoritmo di Divide et Impera è stato ideato, sia analizzarne la correttezza che la complessità. In particolare, la complessità di un algoritmo Divide et Impera, essendo un algoritmo ricorsivo, può essere determinata risolvendo un'opportuna relazione di ricorrenza relativa alla funzione $T(n)$ che dà il massimo tempo di calcolo dell'algoritmo su istanze di dimensione n. Dovrebbero essere già noti i metodi più comuni per risolvere tali ricorrenze, tuttavia ricordiamo uno degli strumenti più utili, il cosidetto *Master Theorem*, che risolve relazioni del seguente tipo:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad \text{dove } a \geqslant 1 \text{ e } b > 1$$

- Se $f(n) = \Theta(n^c)$ con $c < \log_b a$, allora

$$T(n) = \Theta(n^{\log_b a})$$

- Se $f(n) = \Theta(n^{\log_b a} \log^k n)$ per $k \geqslant 0$, allora

$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$$

- Se $f(n) = \Theta(n^c)$ con $c > \log_b a$, allora

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

Ad esempio, per il merge-sort la relazione di ricorrenza è

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

dove n è la dimensione dell'array di input. Dal Master Theorem si ottiene che $T(n) = \Theta(n\log n)$ (secondo caso con $a = b = 2$ e $k = 0$). Per la ricerca binaria la ricorrenza è

$$T(n) = T(n/2) + O(1)$$

e dal Master Theorem concludiamo che $T(n) = \Theta(\log n)$ (secondo caso con $a = 1$, $b = 2$, $k = 0$).

Il problema del massimo sottovettore

Ora vedremo un problema che si è presentato come versione semplificata di un problema di analisi statistica di immagini negli anni '70 del secolo scorso. Il problema è conosciuto con il nome *Maximum subarray problem* e consiste nel trovare, dato un vettore A di numeri reali, un sottovettore (una sequenza di elementi consecutivi del vettore) la cui somma degli elementi è massima. Si assume che il sottovettore vuoto abbia somma zero. Il problema

è interessante perché pur essendo molto semplice ci permetterà di vedere un'evoluzione di algoritmi via via più efficienti che passando per la tecnica Divide et Impera arriverranno fino alla Programmazione dinamica. Lo vedremo infatti anche quando parleremo di quest'ultima tecnica.

È chiaro che il problema è banale se i valori sono tutti nonnegativi o sono tutti nonpositivi. Il problema non è banale solo se il vettore di input ha valori positivi e valori negativi. Ecco un esempio:

Un algoritmo diretto per il problema consiste nel considerare la somma di tutti i possibili sottovettori e prenderne il massimo (per semplicità descriviamo l'algoritmo che ritorna solamente il massimo senza indicazione del sottovettore):

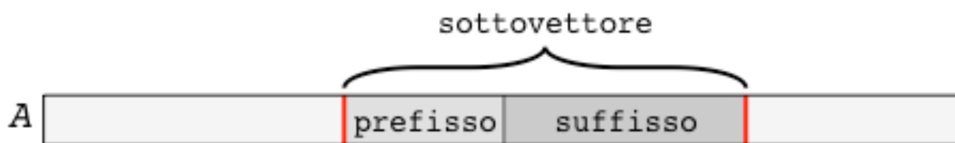
```
MS1(A: vettore di n numeri reali)
    max <- 0            /* somma di un sottovettore vuoto */
    FOR i <- 1 TO n DO
        FOR j <- i TO n DO
            sum <- 0
            FOR k <- i TO j DO
                sum <- sum + A[k]
            IF sum > max THEN max <- sum
    RETURN max
```

È molto inefficiente, i tre cicli `FOR` nidificati implicano che la complessità è $\Theta(n^3)$. Possiamo migliorarlo osservando che la somma di un sottovettore `[i…j]` è uguale alla somma del sottovettore `[i…j-1]` più `A[j]`. Quindi l'algoritmo migliorato è:

```
MS2(A: vettore di n numeri reali)
    max <- 0
    FOR i <- 1 TO n DO
        sum <- 0
        FOR j <- i TO n DO
            sum <- sum + A[j]
            IF sum > max THEN max <- sum
    RETURN max
```

La complessità di `MS2` è $\Theta(n^2)$. Possiamo fare di meglio? Cerchiamo di applicare la tecnica Divide et Impera. La prima cosa che viene in mente è di dividere a metà il vettore, trovare il massimo in entrambe le metà e poi calcolare il massimo delle somme dei sottovettori a cavallo delle due metà. Alla fine ritorniamo il massimo dei tre massimi. Sicuramente l'algoritmo è corretto perché considera tutti i possibili sottovettori. Infatti, un qualsiasi sottovettore o è contenuto nella prima metà del vettore di input o nella seconda o è cavallo delle due. Ma, è più efficiente di `MS2`? La risposta dipende da come effettuiamo il calcolo delle somme dei sottovettori a cavallo. Se lo facciamo in modo diretto cioè, per ogni coppia di indici `i`, `j` con `i` nella prima metà e `j` nella seconda, ci calcoliamo la somma del sottovettore `[i…j]`, questo richiede tempo almeno $O(n^2)$, anche se usiamo la tecnica dell'algoritmo `MS2` dato che il numero di sottovettori a cavallo è ordine di n^2. Ovviamente questo implica che il nostro algoritmo Divide et Impera avrebbe complessità $O(n^2)$ (terzo caso del Master Theorem con $a = b = 2$ e $c = 2$).

Possiamo migliorare il calcolo del massimo delle somme dei sottovettori a cavallo delle due metà? Osserviamo che ogni tale sottovettore è formato da una parte nella prima metà del vettore, che chiamiamo il *prefisso*, e un'altra nella seconda metà, che chiamiamo il *suffisso*:



Un sottovettore di somma massima deve essere tale che il suo prefisso ha somma massima tra tutti i prefissi e il suo suffisso ha somma massima tra tutti i suffissi. Quindi basterà calcolare la somma massima dei prefissi e quella dei suffissi, la somma delle due sarà la somma massima tra tutti sottovettori a cavallo della metà.

```
MS3(A: vettore di numeri reali, a,b: indici di A)
    IF a = b THEN
        IF A[a] > 0 THEN RETURN A[a]
        ELSE RETURN 0
    ELSE
        m <- (a + b)/2       /* parte intera inferiore */
        max1 <- MS3(A, a, m)
        max2 <- MS3(A, m+1, b)
        pre <- 0
        sum <- 0
        FOR i <- m DOWNTO a DO
            sum <- sum + A[i]
            IF sum > pre THEN pre <- sum
        suf <- 0
        sum <- 0
        FOR i <- m+1 TO b DO
            sum <- sum + A[j]
            IF sum > suf THEN suf <- sum
        max <- max1
        IF max2 > max THEN max <- max2
        IF pre + suf > max THEN max <- pre + suf
        RETURN max
```

La prima chiamata sarà `MS3(A, 1, n)`. La relazione di ricorrenza per il tempo di calcolo di `MS3` è $T(n) = 2T(n/2) + O(n)$, infatti ora il calcolo del sottovettore massimo tra quelli a cavallo richiede solamente tempo lineare. Il secondo caso del Master Theorem con $a = b = 2$ e $k = 0$, ci dice che la complessità dell'algoritmo basato sulla tecnica Divide et Impera è $\Theta(n\log n)$. Un bel miglioramento rispetto a $\Theta(n^2)$ di `MS2`.

Esercizio [salto]

In un vettore V di n interi, chiamiamo *salto* un indice i, $1 \leq i < n$, tale che $V[i] < V[i+1]$.

- Dato un vettore V di $n \geq 2$ interi tale che $V[1] < V[n]$, provare che V ha almeno un salto.
- Descrivere un algoritmo che, dato un vettore V di $n \geq 2$ interi tale che $V[1] < V[n]$, trovi un salto in tempo $O(\log n)$.