Sottosequenza di somma massima di un array di numeri

Dato un array a di n numeri positivi e negativi trovare la sottosequenza di numeri consecutivi la cui somma è massima. N.B. Se l'array contiene solo numeri positivi, il massimo si ottiene banalmente prendendo come sequenza quella di tutti i numeri dell'array; se l'array contiene solo numeri negativi il massimo si ottiene prendendo come sottosequenza quella formata dalla locazione contenente il numero più grande .

- I soluzione: Per ogni coppia di indici (i, j) con $i \leq j$ dell'array computa la somma degli elementi nella sottosequenza degli elementi di indice compreso tra i e j e restituisci la sottosequenza per cui questa somma è max.
- Costo della I soluzione: $O(n^3)$ perché

$$\begin{aligned}
\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}(j-i+1) &= \sum_{i=0}^{n-1}\sum_{k=1}^{n-i} k = \sum_{i=0}^{n-1}(n-i+1)(n-i)/2 \\
&= \sum_{i=0}^{n-1}((n-i)^2/2+(n-i)/2) = \sum_{a=1}^{n}(a^2/2+a/2) \\
&= \sum_{a=1}^{n} a^2/2 + \sum_{a=1}^{n} a/2 \\
&= 1/2(n(n+1)(2n+1)/6) + 1/2(n(n+1)/2) = \Theta(n^3).
\end{aligned}$$

Sottosequenza di somma massima di un array di numeri

- II soluzione Osserviamo che la somma degli elementi di indice compreso tra i e j può essere ottenuta sommando $a[j]$ alla somma degli elementi di indice compreso tra i e $j-1$. Di conseguenza, per ogni i, la somma degli elementi in tutte le sottosequenze che partono da i possono essere computate con un costo totale pari a $\Theta(n-i)$. Il costo totale è quindi

$$\sum_{i=0}^{n-1} \Theta(n-i) = \sum_{i=1}^{n} \Theta(i) = \Theta(\sum_{i=1}^{n} i) = \Theta(n^2)$$

Sottosequenza di somma massima di un array di numeri

- III soluzione: Divide et Impera
  Algoritmo A:
  1. Se $i = j$ viene restituita la sottosequenza formata da $a[i]$
  2. Se $i < j$ si invoca ricorsivamente $A(i, (i+j)/2)$ e $A((i+j)/2+1, j)$: la sottosequenza cercata o è una di quelle restituite dalle 2 chiamate ricorsive o si trova a cavallo delle due metà dell'array
  3. La sottosequenza di somma massima tra quelle che intersecano entrambe le metà dell'array si trova nel seguente modo:
     - si scandisce l'array a partire dall'indice $(i+j)/2$ andando a ritroso fino a che si arriva all'inizio dell'array sommando via via gli elementi scanditi: ad ogni iterazione si confronta la somma ottenuta fino a quel momento con il valore max s_1 delle somme ottenute in precedenza e nel caso aggiorna il max s_1 e l'indice in corrispondenza del quale è stato ottenuto.
     - si scandisce l'array a partire dal'indice $(i+j)/2+1$ andando in avanti fino a che o si raggiunge la fine dell'array sommando gli elementi scanditi: ad ogni iterazione si confronta la somma ottenuta fino a quel momento con il valore max s_2 delle somme ottenute in precedenza e nel caso aggiorna il max s_2 e l'indice in corrispondenza del quale è stato ottenuto.
     - La sottosequenza di somma massima tra quelle che intersecano le due metà dell'array è quella di somma $s_1 + s_2$.
  4. L'algoritmo restituisce la sottosequenza massima tra quella restituita dalla prima chiamata ricorsiva, quella restituita dalla seconda chiamata ricorsiva e quella di somma $s_1 + s_2$

Sottosequenza di somma massima di un array di numeri

- Tempo di esecuzione dell'algoritmo Divide et Impera

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

Il tempo di esecuzione quindi è $O(n \log n)$.

Sottosequenza di somma massima di un array di numeri

- IV soluzione: Chiamiamo s_j la sottosequenza di somma massima che tra quelle che terminano in j. Si ha $s_{j+1} = \max\{s_j + a[j+1], a[j+1]\}$. Questo valore si calcola in tempo costante per ogni j. L'algoritmo calcola questi valori per ogni j e prende il massimo degli n valori computati. Il tempo dell'algoritmo quindi è $O(n)$.

Esempi di relazioni di ricorrenza della forma $T(n) \leq \alpha T(n/\beta) + n^k$

- Ricerca binaria

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \text{ oppure } k \text{ è l'elemento centrale} \\ T(n/2) + c & \text{altrimenti} \end{cases}$$

Si ha $\alpha = 1, \beta = 2$, $k = 0$.

Siccome $\alpha = \beta^k$, siamo nel secondo caso e si ha

$$T(n) = O(n^k \log n)) = O(\log n).$$

Esempi di relazioni di ricorrenza della forma $T(n) \leq \alpha T(n/\beta) + n^k$

Nell'ordinamento per fusione,

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ 2T(n/2) + cn & \text{altrimenti} \end{cases}$$

Quindi,

- $\alpha = 2$, $\beta = 2$ e $k = 1$
- siamo nel caso $\alpha = \beta^k$ e quindi $T(n) = O(n^k \log n) = O(n \log n)$.

Esempi di relazioni di ricorrenza della forma $T(n) \leq \alpha T(n/\beta) + n^k$

- Moltiplicazione veloce di interi: primo algoritmo

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ 4T(n/2) + cn & \text{altrimenti} \end{cases}$$

Applicazione del risultato provato:

  - si ha che $\alpha = 4$, $\beta = 2$ e $k = 1$
  - $\alpha > \beta^k$, quindi si applica il terzo caso e si ha $T(n) = O(n^{\log_2 4}) = O(n^2)$

- Moltiplicazione veloce di interi: secondo algoritmo

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 1 \\ 3T(n/2) + cn & \text{altrimenti} \end{cases}$$

Applicando il risultato dimostrato,

  - si ha che $\alpha = 3$, $\beta = 2$ e $k = 1$
  - $\alpha > \beta^k$, quindi si applica il terzo caso e si ha $T(n) = O(n^{\log_2 3}) = O(n^{1,585})$

Sommatorie utili

- $$\sum_{i=1}^{n} i = n(n+1)/2$$
- $$\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$$
- $$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1} \text{ per } a \neq 1$$
- $$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \text{ per } 0 < a < 1.$$

Divide et impera su alberi

- **Caso base**: per `u` = `null` o una foglia
- **Decomposizione**: riformula il problema per i sottoalberi radicati nei figli di u.
- **Ricombinazione**: ottieni il risultato con `Ricombina`

```
1  Decomponibile(u):
2    IF (u == null) {
3      RETURN valore base;
4    } ELSE {
5      i=0;
       FOR( ciascun figlio f di u ){

6      risultatiFigli[i] = Decomponibile(f);
7      i=i+1 }
8      RETURN Ricombina(risultatiFigli);
9    }
```

La ricombinazione del risultati della chiamate ricorsive sui figli potrebbe essere effettuata anche nel for man mano che vengono ottenuti i risultati delle chiamate sui figli.

Divide et impera su alberi binari

- **Caso base**: per `u` = `null` o una foglia
- **Decomposizione**: riformula il problema per i sottoalberi radicati nei figli `u.sx` e `u.dx`
- **Ricombinazione**: ottieni il risultato con `Ricombina`

```
1 Decomponibile(u):
2   IF (u == null) {
3     RETURN valore base;
4   } ELSE {
5     risultatoSX = Decomponibile(u.sx);
6     risultatoDx = Decomponibile(u.dx);
7     RETURN Ricombina(risultatoSX, risultatoDx);
8   }
```

Analisi dell'algoritmo Decomponibile

- Assumiamo che il tempo per la decomposizione e la ricombinazione sia costante
- Se escludiamo il tempo impiegato per le chiamate ricorsive, l'algoritmo impiega tempo $O(1+ c_v)$, dove c_v è il numero di figli di v
- Se cominciamo la visita dal nodo w, l'algoritmo viene invocato su tutti i discendenti di w

→Tempo totale = $\sum_{v \in T_w} O(c_v + 1) = O(|T_w|)$

- La visita di tutto l'albero richiede tempo $O(|\mathbf{T}|)$
- Se l'albero ha n nodi la visita richiede tempo $T(n)= O(n)$

Analisi dell'algoritmo Decomponibile

- Nell'analisi precedente abbiamo usato il fatto che $\sum_{v \in T_w} c_v = |T_w| - 1$.
- È facile vedere che vale questa uguaglianza in quanto ogni nodo di T_w, eccezion fatta per la radice w, è figlio di un unico nodo v dell'albero T_w e quindi viene contato esattamente una volta in quella sommatoria.

Analisi dell'algoritmo Decomponibile per un albero binari mediante relazione di ricorrenza

La funzione $T(n)$ che esprime il tempo di esecuzione dell'algoritmo Decomponibile su un albero binario con n nodi può essere descritta dalla seguente relazione di ricorrenza, dove $r-1 \geq 0$ è il numero di nodi del sottoalbero sinistro.

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

Dimostriamo per induzione che $T(n) \leq c'n$ per $n \geq 1$ e per una costante $c' > 0$. In altre parole $T(n) = O(n)$.

- Base: $T(1) \leq c_0$ implica $T(1) \leq c'$ se si sceglie $c' \geq c_0$.
- Passo induttivo: Assumiano $T(m) \leq c'm$ per ogni $1 \leq m < n$ e dimostriamo $T(n) \leq c'n$.

  Applichiamo relazione di ricorrenza: $T(n) \leq T(r-1) + T(n-r) + c$.

  L'ipotesi induttiva implica $T(r-1) \leq c'(r-1)$ e $T(n-r) \leq c'(n-r)$.

  Si ha quindi $T(n) \leq c'(r-1) + c'(n-r) + c = c'n - c'r + c$.

  Affinché risulti $T(n) \leq c'n$ basta scegliere c' in modo che $c'r \geq c$ cioè $c' \geq c/r$. Non sappiamo quanto vale r ma sappiamo che $r \geq 1$ per cui basta scegliere $c' \geq c$.
- Dalla base dell'induzione e dal passo induttivo, sappiamo che basta scegliere $c' = \max\{c_0, c\}$ affinché valga $T(n) \leq c'n$ per $n \geq 1$.

Algoritmi ricorsivi su alberi: dimensione

Calcolo della dimensione d = numero di nodi

- Caso base: albero vuoto $\Rightarrow d = 0$
- Caso induttivo: $d = 1+$ dimensione del sottoalbero sinistro $+$ dimensione del sottoalbero destro

```
1  Dimensione( u ):
2    IF (u == null) {
3      RETURN 0;
4    } ELSE {
5      dimensioneSX = Dimensione( u.sx );
6      dimensioneDX = Dimensione( u.dx );
7      RETURN dimensioneSX + dimensioneDX + 1;
8    }
```

Se si vuole conoscere la dimensione di tutto l'albero, si invoca Dimensione con u uguale alla radice

Algoritmi ricorsivi su alberi: altezza

Calcolo dell'altezza h di un nodo:

- caso base per `null` $\Rightarrow h = -1$
- passo induttivo: $h = 1+$ massima altezza dei figli

```
1  Altezza( u ):
2    IF (u == null) {
3      RETURN -1;
4    } ELSE {
5      altezzaSX = Altezza( u.sx );
6      altezzaDX = Altezza( u.dx );
7      RETURN max( altezzaSX, altezzaDX ) + 1;
8    }
```

Per calcolare l'altezza dell'albero, si invoca Altezza con `u` uguale alla radice

Visita di un albero binario: inorder

- **simmetrica** (*inorder*):

```
1  Simmetrica( u ):
2    IF (u != null) {
3      Simmetrica( u.sx );
4      elabora(u);
5      Simmetrica( u.dx );
6    }
```

  $O(n)$ tempo per n nodi

Visita di un albero binario: preorder

- **anticipata** (*preorder*):

```
1  Anticipata( u ):
2    IF (u != null) {
3      elabora(u);
4      Antiticipata( u.sx );
5      Antiticipata( u.dx );
6    }
```

  $O(n)$ tempo per n nodi

Visita di un albero binario: postorder

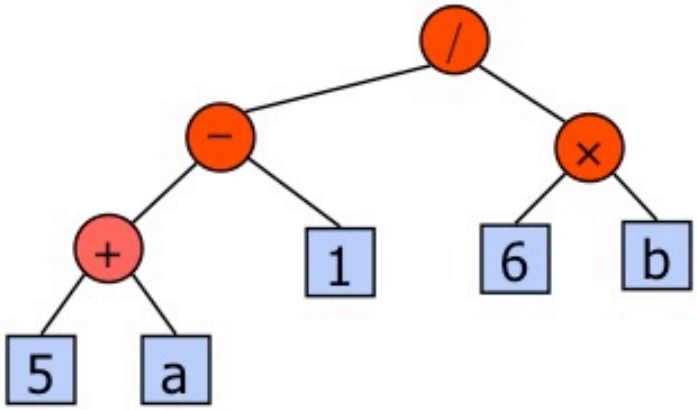
- **posticipata** (*postorder*):

```
1  Posticipata( u ):
2    IF (u != null) {
3      Posticipata( u.sx );
4      Posticipata( u.dx );
5      elabora(u);
6    }
```

  $O(n)$ tempo per n nodi

Esempio dell'uso delle visite: valutazione dell'espressione aritmetica rappresentata da un albero binario

- Albero binario associato ad una espressione:
  - Nodi interni: operatori
  - Nodi esterni: operandi
- Esempio: $((5 + a) - 1) / (6 \times b)$



Uso della visita postorder per valutare l'espressione aritmetica rappresentata da un albero binario

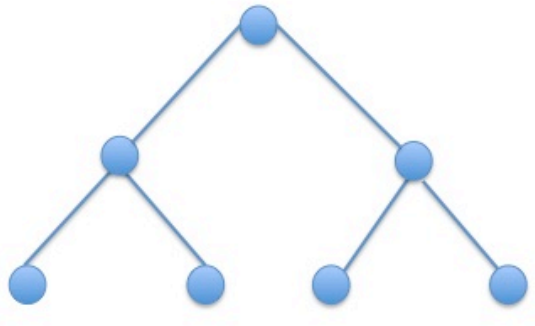
```
1  Valuta( u ):
2    IF (u==null) {
3      RETURN null;
4    }
5    IF (u.sx == null && u.dx==null) {
6      RETURN u.dato;
7    } ELSE {
8      valSinistra=Valuta( u.sx );
9      valDestra= Valuta( u.dx );
10     ris= Calcola(u.dato,valSinistra ,valDestra);
11     RETURN ris;
12   }
```

- La funzione Calcola invocata su *u.dato*, *valSinistra* e *valDestra*, applica l'operatore memorizzato nel nodo interno *u* ai valori *valSinistra* e *valDestra*.
- N.B.: la condizione del primo if è soddisfatta (*u* è null) solo se inizialmente la funzione Valuta è invocata su null. Se inizialmente Valuta è invocata su un nodo $u \neq null$ allora la condizione del primo if non sarà mai soddisfatta perché quando è invocata su una foglia, la funzione restituisce il contenuto della foglia.

Algoritmo per verificare se un albero binario è completamente bilanciato

Definizioni:

- Albero binario **proprio**: ogni nodo interno ha sempre due figli non vuoti
- Albero **completamente bilanciato**: albero proprio con tutte le **foglie** alla **stessa profondità**
  Esempio:



Algoritmo per verificare se un albero binario è completamente bilanciato

- Def. ricorsiva di albero completamente bilanciato:
  - Un albero binario vuoto è completamente bilanciato
  - Una albero binario con almeno un nodo è completamente bilanciato se e solo se il sottoalbero destro e il sottoalbero sinistro della radice sono completamente bilanciati e hanno la stessa altezza (per convenzione, un albero vuoto ha altezza -1)
- N.B. In un albero completamente bilanciato l'altezza dell'albero corrisponde alla profondità di tutte le foglie
- Indichiamo con $T(u)$ il sottoalbero di T radicato in u
- Risolviamo un problema più generale per $T(u)$, calcolandone anche l'altezza oltre che a dire se è completamente bilanciato o meno
- La ricorsione restituisce una coppia (booleano, intero)
- Tempo di risoluzione: $O(n)$ tempo per n nodi

```
1  CompletamenteBilanciato( u ):
2    IF (u == null) {
3      RETURN <TRUE, -1>;
4    } ELSE {
5      <bilSX,altSX> = CompletamenteBilanciato( u.sx );
6      <bilDX,altDX> = CompletamenteBilanciato( u.dx );
7      bil = bilSX && bilDX && (altSX == altDX);
8      altezza = max(altSX, altDX) + 1;
9      RETURN <bil,altezza>;
10   }
```

Algoritmi ricorsivi su alberi: profondità di un nodo

- La radice ha profondità 0
- I figli della radice hanno profondità pari a 1, e così via
- Un nodo ha profondità p ha i figli a profondità $p + 1$

Versione iterativa dell'algoritmo per calcolare la profondità di un nodo `u`

```
p = 0;
WHILE (u.padre != null) {
  p = p + 1;
  u = u.padre;
}
```

Definizione ricorsiva di profondità di un nodo:

- La radice ha profondità 0
- I nodi diversi dalla radice hanno profondità pari alla profondità del padre + 1

Versione ricorsiva dell'algoritmo per calcolare la profondità di un nodo `u`

```
1 Profondita( u ):
2   IF (u.padre==null) {
3     RETURN 0;
4   }
5   RETURN profondita(u.padre)+1;
```

Trasmissione dell'informazione tra chiamate ricorsive

- **postorder** : l'informazione è trasferita dalle foglie alla radice
  – la soluzione del problema per $T(\mathtt{u})$ può essere ottenuta dalla soluzioni dei sottoproblemi per $T(\mathtt{u.sx})$ e $T(\mathtt{u.dx})$
- **passaggio dei parametri** : informazione passata attraverso i parametri dalla radice alle foglie
  – la soluzione del problema per $T(\mathtt{u})$ può essere ottenuta utilizzando l'informazione raccolta dalla radice fino al nodo u

Esempio: stampa la profondità di tutti i nodi

```
1 Profondita( u, p ):
2   IF (u != null) {
3     PRINT profondità di u è pari a p;
4     Profondita( u.sx, p+1 );
5     Profondita( u.dx, p+1 );
6   }
```

Il parametro p indica la profondità del nodo u. Se vogliamo stampare le profondità di tutti i nodi dobbiamo invocare la funzione con u uguale all'indirizzo della radice dell'albero e $p = 0$.

Algoritmo per trovare i nodi cardine

Trasferiamo informazione simultaneamente dalle foglie alla radice e dalla radice verso le foglie combinando i due approcci della slide precedente

- Nodo u è cardine se e solo se $\text{profondita}(u) = \text{altezza}(T(u))$

```
1  Cardine( u, p ):
2    IF (u == null) {
3      RETURN -1;
4    } ELSE {
5      altezzaSX = Cardine( u.sx, p+1 );
6      altezzaDX = Cardine( u.dx, p+1 );
7      altezza = max( altezzaSX, altezzaDX ) + 1;
8      IF (p == altezza) PRINT u.dato;
9      RETURN altezza;
10   }
```