Il paradigma "divide et impera"

La tecnica algoritmica del "divide et impera" consiste nel

- decomporre il problema in un piccolo numero di sotto-problemi, ciascuno dei quali è dello stesso tipo del problema originale ma è definito su un insieme di dati più piccolo rispetto a quello iniziale;
- risolvere **ricorsivamente** ciascun sotto-problema fino a che non si arriva a risolvere sotto-problemi di taglia così piccola da poter essere risolti direttamente (senza effettuare ulteriori chiamate ricorsive);
- combinare le soluzioni dei sotto-problemi al fine di ottenere una soluzione al problema di partenza.

Ordinamento per fusione: MergeSort

L'algoritmo MergeSort ordina in modo non decrescente una sequenza di numeri. L'idea dell'algoritmo è descritta di seguito.

- Se l'array contiene due o più elementi, l'array viene suddiviso in due parti ciascuna delle quali contiene circa la metà degli elementi
- Le due sottosequenze vengono ordinate ricorsivamente.
- Una volta ordinate, le due sottosequenze vengono fuse in un'unica sequenza ordinata.

MergeSort

Descriviamo l'algoritmo MergeSort che ordina un array. L'algoritmo riceve in input un array e due interi che delimitano la parte di array che si desidera ordinare. Inizialmente invochiamo MergeSort con *sinistra* uguale a 0 e *destra* uguale al numero di elementi dell'array -1.

```
1  MergeSort( a, sinistra, destra ):
2    IF (sinistra < destra) {
3      centro = (sinistra+destra)/2;
4      MergeSort( a, sinistra, centro );
5      MergeSort( a, centro+1, destra );
6      Merge( a, sinistra, centro, destra );
7    }
```

Per calcolare il tempo di esecuzione $T(n)$ dobbiamo tener conto del

- tempo per decomporre il problema in due sottoproblemi : O(1) in quanto occorre solo calcolare il centro
- tempo per eseguire le due chiamate ricorsive: $T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil)$
- tempo per fondere le due sequenze: ?

L'algoritmo Fusione

- Possiamo fondere due sequenze ordinate $A =< a_1, \ldots, a_n >$ e $B =< b_1, \ldots, b_m >$ in modo da formare un'unica sequenza ordinata in tempo lineare in $n + m$.
- L'idea dell'algoritmo è il seguente:
  1. Scandiamo gli elementi delle due sequenze da sinistra verso destra utilizzando l'indice i per A e l'indice j per B.
  2. Fino a che $i \leq n$ e $j \leq m$, confrontiamo a_i con b_j. Se a_i è minore o uguale di b_j, a_i viene inserito alla fine della sequenza output o, nel caso di un array, nella prima cella libera e i viene incrementato di 1. Se a_i è maggiore di b_j, b_j viene inserito alla fine della sequenza output o, nel caso di un array, nella prima cella libera e j viene incrementato di 1.
  3. Al termine del ciclo precedente se $i \leq n$ trasferiamo uno dopo l'altro gli elementi $a_i, \ldots, a_n$ alla fine della sequenza output; se $j \leq m$ trasferiamo uno dopo l'altro gli elementi $b_j, \ldots, b_m$ alla fine della sequenza output.

L'algoritmo Fusione

- Ogni volta che eseguiamo un confronto tra un elemento di A ed uno di B, viene incrementato uno tra i due indici i e j. Di consequenza l'algoritmo effettua al più $n + m$ confronti.
- Sia $k \leq n + m$ il numero totale di confronti effettuati dall'algoritmo. Al termine di questi confronti, la sequenza output conterrà k elementi e in una delle due sequenze ci saranno $n + m - k$ elementi che dovranno essere trasferiti nella sequenza output.
- Il tempo totale per fondere le due sequenze ordinate è quindi lineare in $k + (n + m - k) = n + m$.

Merge: algoritmo Merge

Descriviamo l'algoritmo Merge che fonde due segmenti adiacenti di un array.

- Il primo segmento parte dalla locazione di indice sx e finisce nella locazione di indice cx
- il secondo segmento parte dalla locazione di indice $cx + 1$ e finisce nella locazione di indice dx

```
1  Merge( a, sx, cx, dx ):
2    i = sx; j = cx+1; k = 0;
3    WHILE ((i <= cx) && (j <= dx)) {
4      IF (a[i] <= a[j]) {
5        b[k] = a[i]; i = i+1;
6      } ELSE {
7        b[k] = a[j]; j = j+1;
8      }
9      k = k+1;
10   }
11   FOR ( ; i <= cx; i = i+1, k = k+1)
12     b[k] = a[i];
13   FOR ( ; j <= dx; j = j+1, k = k+1)
14     b[k] = a[j];
15   FOR (i = sx; i <= dx; i = i+1)
16     a[i] = b[i-sx];
```

Analisi dell'algoritmo MergeSort

Ora che sappiamo qual è il tempo di esecuzione dell'algoritmo MergeSort, possiamo completare l'analisi dell'algoritmo MergeSort. Indichiamo con $T(n)$ il suo tempo di esecuzione per un array input di n elementi. Il tempo $T(n)$ è dato da

- tempo per decomporre il problema in due sottoproblemi : $\Theta(1)$,
- tempo per eseguire le due chiamate ricorsive: $T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil)$,
- tempo per fondere le due sequenze: $cn = \Theta(n)$.

Si ha quindi $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + cn + c' = O(?)$.

Relazioni di ricorrenza

- Quando un algoritmo contiene una o più chiamate ricorsive a sé stesso, il suo tempo di esecuzione può essere spesso descritto da una *relazione di ricorrenza*.
- Una relazione di ricorrenza consiste in un'uguaglianza o in una disuguaglianza che descrive una funzione in termini dei suoi valori su input più piccoli.
- Esempio:

$$f(n) = \begin{cases} a & \text{se } n \leq 2 \\ 2f(n/3) + 4n & \text{altrimenti} \end{cases}$$

Relazioni di ricorrenza

- Vediamo come si scrive la relazione di ricorrenza che descrive il tempo di esecuzione $T(n)$ di un algorimo basato sulla tecnica del divide et impera per un input di dimensione n.

- Se la dimensione n del problema è minore di una certa costante c, l'algoritmo risolve direttamente il problema (senza effettuare chiamate ricorsive)

$$T(n) \leq c_0, \text{ per una certa costante } c_0 \ .$$

- Per $n > c$, il problema viene suddiviso in sottoproblemi: supponiamo che il problema venga suddiviso in α sottoproblemi, ognuno di dimensione n/β
- L'algoritmo viene invocato ricorsivamente per risolvere ciascuno di questi α sottoproblemi
- Le α soluzioni per questi sottoproblemi vengono ricombinate per ottenere la soluzione al problema originario.

Relazioni di ricorrenza

- Supponiamo che l'algoritmo impieghi al più tempo $d(n)$ per suddividere il problema di partenza in α sottoproblemi.
- Supponiamo che l'algoritmo impieghi al più tempo tempo $r(n)$ per ricombinare le soluzioni degli α sottoproblemi.
- Il tempo di esecuzione $T(n)$ per $n > c$ può essere descritto dalla relazione:

$$T(n) \leq \alpha T(n/\beta) + d(n) + r(n)$$

- Quindi possiamo scrivere la relazione di ricorrenza:

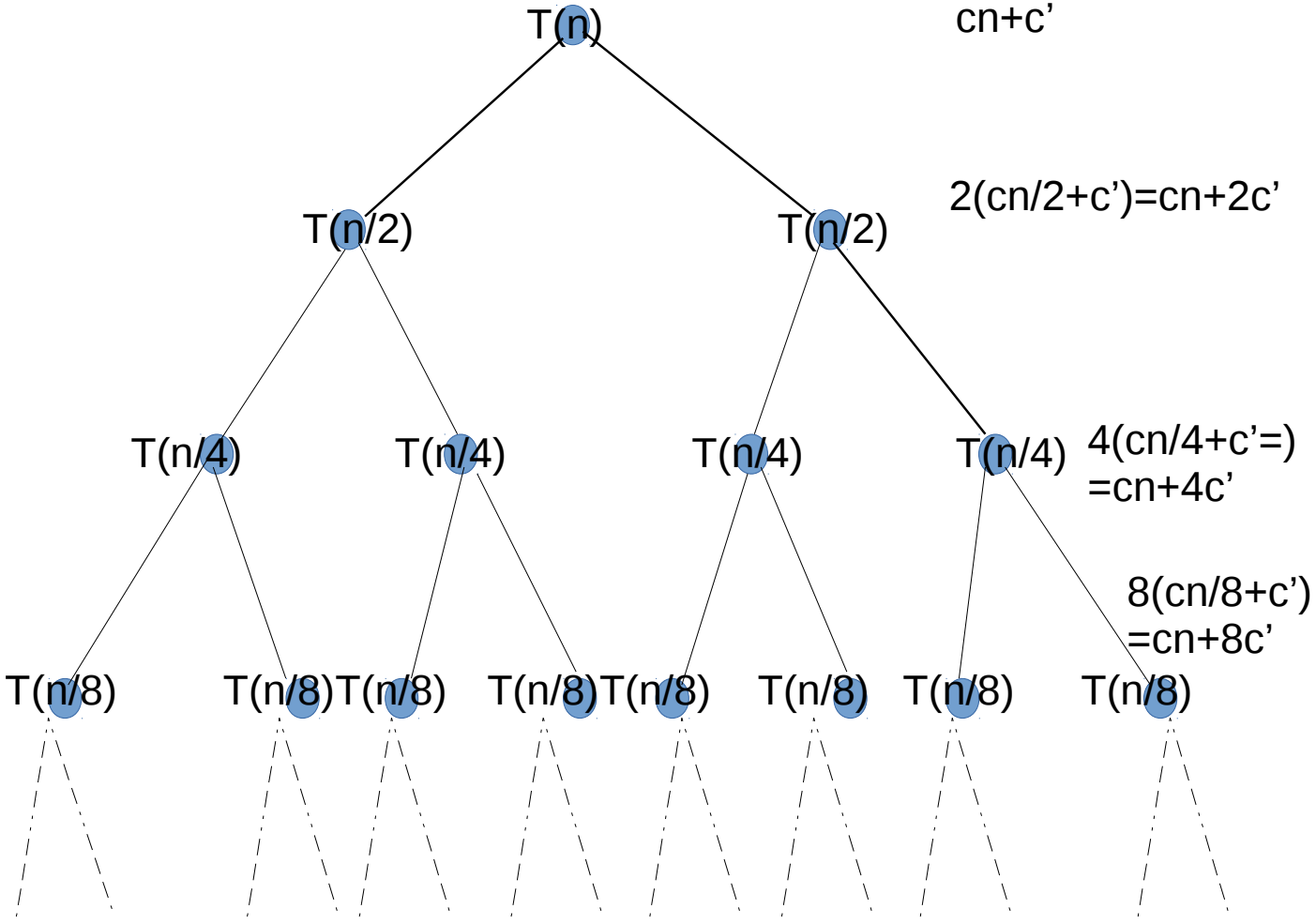
$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq c \\ \alpha T(n/\beta) + d(n) + r(n) & \text{altrimenti} \end{cases}$$

Tempo di esecuzione di MergeSort

- L'algoritmo MergeSort decompone il problema in due sottoproblemi di dimensione $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$ rispettivamente e impiega tempo costante per la decomposizione (deve semplicemente computare l'indice centrale in modo da individuare la fine e l'inizio dei due segmenti da ordinare) e tempo lineare per ricombinare le soluzioni dei sue sottoproblemi (deve fondere i due segmenti ordinati).
- Nell'analisi per semplicità assumiamo che n sia una potenza di 2 in modo che ogni chiamata ricorsiva divida il segmento su cui opera in due segmenti di uguale grandezza.
- Quindi

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

Tempo di esecuzione di MergeSort



L’ultimo livello contiene n foglie ciascuna delle quali rappresenta il tempo per risolvere il problema su un input di dimensione 1. In totale il lavoro richiesto da queste n chiamate ricorsive e` $c_0 n$

$\log_2 n+1$ livelli -→sommando su tutti i livelli $cn \log n + c_0 n + (2^{\log n+1}-1)c'$

$= cn \log n + c_0 n + c'n - c'$ -->$T(n)=\Theta(n \log n)$

Tempo di esecuzione di MergeSort

- Dimostriamo che il tempo di esecuzione è $\Theta(n \log n)$.
- Iteriamo la ricorrenza

$$\begin{aligned} T(n) &= c' + cn + 2T(n/2) = c' + cn + 2(c' + cn/2 + 2T(n/4)) \\ &= (1+2)c' + 2cn + 4T(n/4) = (1+2)c' + 2cn + 4(c' + cn/4 + 2T(n/8)) \\ &= (1+2+4)c' + 3cn + 8T(n/8) \\ &= \ldots = (1+2+4+8+\ldots+2^{i-1}) + icn + 2^i T\left(\frac{n}{2^i}\right) \end{aligned}$$

- Quante volte dobbiamo iterare la ricorrenza prima di raggiungere il caso base?
- Ogni volta che applichiamo la ricorrenza la dimensione dell'input viene dimezzata per cui l'i-esima volta che applichiamo la ricorrenza l'argomento della funzione T diventa $\frac{n}{2^i}$. Raggiungiamo il caso base quando $\frac{n}{2^i} \leq 1$ e cioè non appena $2^i \geq n$. Ne consegue che ci fermiamo dopo che abbiamo applicato la ricorrenza $\log n$ volte.
- Dopo aver applicato la ricorrenza $\log n$ volte si ha

$$T(n) = c'(2^{\log n} - 1) + cn \log n + 2^{\log n} T(1) = c'n - c' + cn \log n + nc_0.$$

- Abbiamo dimostrato che $T(n) = \Theta(n \log n)$

Ricerca binaria: versione ricorsiva

```
1  RicercaBinariaRicorsiva( a,k,sinistra,destra ):
2    IF (sinistra > destra) {
3      RETURN -1;
4    }
5    c = (sinistra+destra)/2;
6    IF (k == a[c]) {
7      RETURN c;
8    }
9    IF (sinistra==destra) {
10     RETURN -1;
11   }
12   IF (k <a[c]) {
13     RETURN RicercaBinariaRicorsiva( a,k,sinistra,c-1 );
14   } ELSE {
15     RETURN RicercaBinariaRicorsiva( a,k,c+1,destra );
16   }
```

Paradigma divide et impera

1. **Caso base**: Il segmento in cui stiamo effettuando la ricerca contiene al più un elemento oppure abbiamo trovato l'elemento al centro del segmento (righe 2–10)
2. **Decomposizione**: per decomporre occorre vedere se k è minore o maggiore di $a[c]$ (riga 11)
3. **Ricorsione e ricombinazione**: di fatto non occorre nessun lavoro di ricombinazione (righe 12–16)

Analisi mediante relazione di ricorrenza

- Se il segmento all'interno del quale stiamo cercando contiene al più un elemento oppure l'elemento cercato è quello centrale, allora l'algoritmo esegue un numero costante di operazioni $\leq c_0$.
- Altrimenti, il tempo richiesto è pari a una costante c più il tempo richiesto dalla ricerca dell'elemento in un segmento di dimensione al più pari alla metà di quello attuale.
  Il tempo totale di esecuzione $T(n)$ su un array di n elementi verifica la relazione di ricorrenza:

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

- Applicando iterativamente la ricorrenza si ha

$$T(n) \leq T(n/2) + c \leq T(n/4) + c + c \leq \ldots \leq T(\frac{n}{2^i}) + ci$$

- Per $i = \log n$ abbiamo

$$T(n) \leq T(1) + c \log n = c_o + c \log n = O(\log n).$$

Analisi mediante relazione di ricorrenza

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

- Risolviamo la relazione di ricorrenza con il metodo della sostituzione.
- Intuizione ci suggerisce che $T(n) = O(\log n)$. Dimostriamo questo limite con l'induzione. Dimostremo che $T(n) \leq c' \log n$ per una certa costante $c' > 0$ e per ogni $n \geq 2$.
- Base dell'induzione: per $n = 2$ si ha tempo minore o uguale ad una certa costante C per cui basta scegliere $c' \geq C$.
- Passo induttivo: Supponiamo che per $2, \ldots, n-1$ il limite superiore sia verificato. Si ha quindi che $T(n/2) \leq c' \log(n/2)$.Di conseguenza

$$T(n) \leq T(n/2) + c \leq c' \log(n/2) + c = c' \log n - c' + c$$

- Affinché risulti $T(n) \leq c' \log n$ basta scegliere $c' \geq c$.
- Abbiamo quindi dimostrato che $T(n) \leq c' n$ per ogni $n' \geq 2$ e $c' = \max\{C, c\}$

Paradigma della ricerca binaria

Viene usato in diverse situazioni: per esempio, indovinare un numero positivo x con domande del tipo "$x \leq b$?", per un certo b

1. Chiedi se il numero intero x è $\leq 2^i$ per $i = 1, 2, \ldots$
2. Fermati non appena la risposta è *sì*.
3. Sia h l'indice in corrispondenza del quale otteniamo *sì* come risposta. Ovviamente si ha che $2^{h-1} < x \leq 2^h$ e di conseguenza $\log x \leq h < \log x + 1$
4. Effettua ricerca binaria nell'intervallo $[2^{h-1} + 1, 2^h]$
5. Intervallo contiene 2^{h-1} interi per cui ricerca binaria nell'intervallo richiede tempo $O(\log 2^{h-1}) = O(h - 1) = O(\log x)$
6. In totale $O(\log x)$ per le $h = \lceil \log x \rceil$ domande fatte per individuare l'intervallo $[2^{h-1} + 1, 2^h]$ più tempo $O(\log x)$ per trovare l'elemento nell'intervallo.

Lower bound sulla ricerca

Sia A un qualunque algoritmo di ricerca che usa confronti tra coppie di elementi: A deve discernere tra $n+1$ situazioni (l'elemento cercato appare in una delle n posizioni dell'insieme oppure non appare nell'insieme)

- A esegue dei confronti, ognuno dei quali dà luogo a tre possibili risposte in $[<, >, =]$
- Se A effettua t confronti ci sono 3^t possibili sequenze di risposte.
- Dopo t confronti di elementi, l'algoritmo A può discernere al più 3^t situazioni.
- Poiché le situazioni da discernere sono $n+1$, deve valere $3^t \geq n+1$.
- Ne deriva che occorrono $t \geq \log_3(n+1) = \Omega(\log n)$ confronti: ciò rappresenta un limite inferiore per il problema della ricerca per confronti.

Conseguenza: l'algoritmo di ricerca binaria è asintoticamente ottimo.

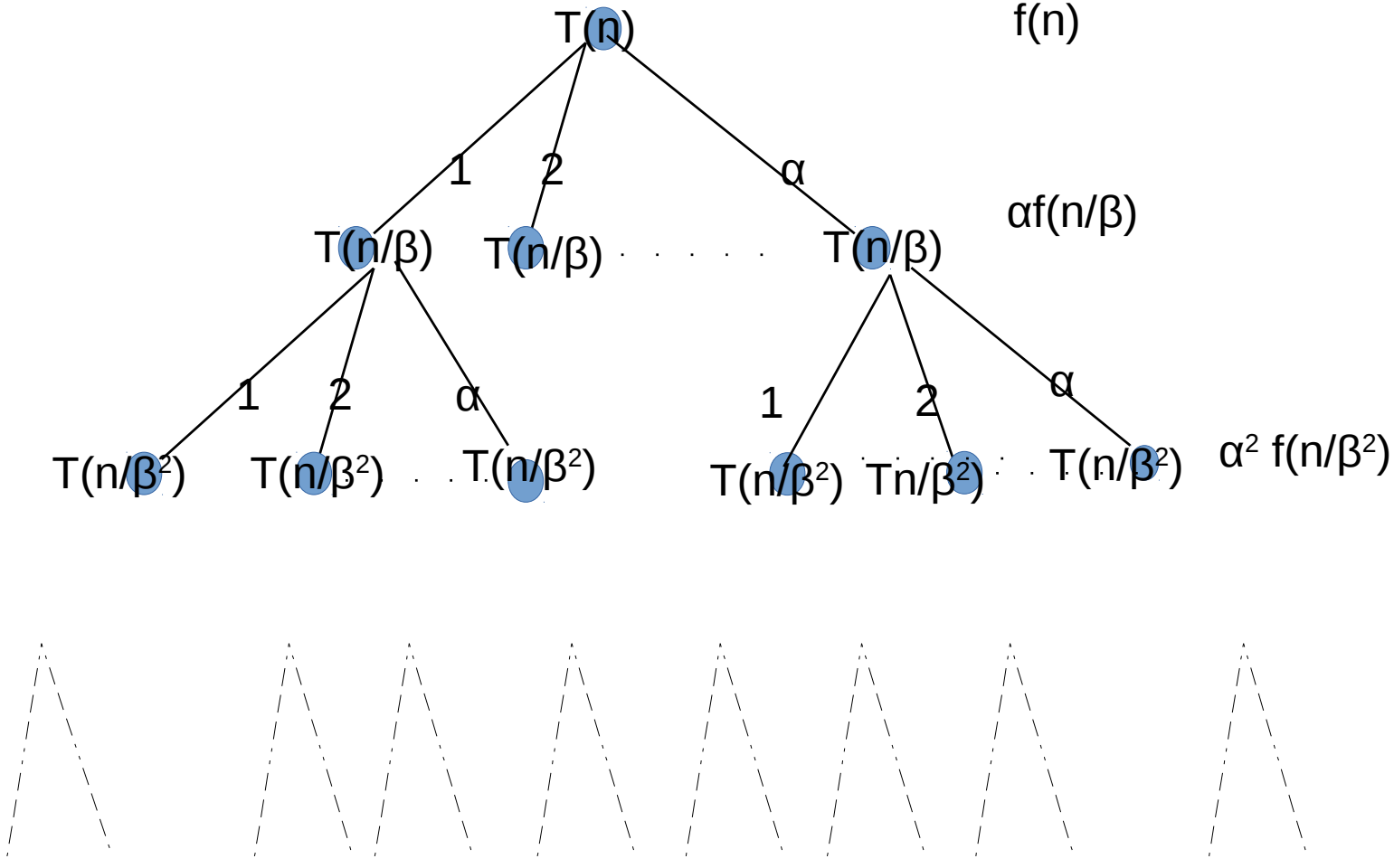
Soluzione delle relazioni di ricorrenza

Consideriamo la seguente relazione di ricorrenza:

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq c \\ \alpha T(n/\beta) + f(n) & \text{altrimenti} \end{cases}$$

Nel caso in cui $T(n)$ sia la funzione che scaturisce dall'analisi di un algoritmo basato sul paradigma del Divide et Impera, $f(n)$ è il tempo per il lavoro di suddivisione e di ricombinazione. In altre parole, $f(n) = d(n) + r(n)$.Per stimare $T(n)$, assumiamo per semplicità che n sia una potenza di β.

Soluzione delle relazioni di ricorrenza



Sia h l'altezza dell'albero (h+1 livelli). Per i<h, Il tempo di esecuzione per tutte le chiamate ricorsive a livello i e` al piu` $\alpha^i f(n/\beta^i)$.

Soluzione delle relazioni di ricorrenza

- Il numero di livelli dell'albero è $\lceil \log_\beta n/c \rceil + 1$ e ciascun nodo sul livello $\lceil \log_\beta n/c \rceil$ corrisponde al tempo $T(n/\beta^{\lceil \log_\beta n/c \rceil}) \leq T(c) \leq c_0$. Il tempo totale per eseguire le $\alpha^{\lceil \log_\beta n/c \rceil}$ chiamate ricorsive in quest'ultimo livello è quindi $\leq \alpha^{\lceil \log_\beta n/c \rceil} c_0$.

- Abbiamo visto che per $i < \lceil \log_\beta n/c \rceil$, il tempo per eseguire tutte le chiamate sul livello i è $\alpha^i f(n\beta^i)$.

- Sommando su tutti i livelli si ha

$$T(n) \leq \alpha^{\lceil \log_\beta n/c \rceil} c_0 + \sum_{i=0}^{\lceil \log_\beta n/c \rceil - 1} \alpha^i f(n/\beta^i).$$

Soluzione delle relazioni di ricorrenza

- Vogliamo stimare la funzione

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq c \\ \alpha T(n/\beta) + f(n) & \text{altrimenti} \end{cases}$$

  quando la funzione $f(n)$ è limitata da $c' n^k$, dove c' e k sono due costanti tali che $k \geq 0$, $c' > 0$ ($f(n)$ polinomiale).

- Da quanto ottenuto nella slide precedente si ha che

$$\begin{aligned} T(n) &\leq \alpha^{\lceil \log_\beta n/c \rceil} c_0 + \sum_{i=0}^{\lceil \log_\beta n/c \rceil - 1} \alpha^i f(n/\beta^i) \\ &\leq \alpha^{\lceil \log_\beta n/c \rceil} c_0 + \sum_{i=0}^{\lceil \log_\beta n/c \rceil - 1} \alpha^i c (n/\beta^i)^k \end{aligned}$$

Soluzione delle relazioni di ricorrenza

- Abbiamo visto che se $f(n) \leq c' n^k$, dove c' e k sono due costanti tali che $k \geq 0$, $c' > 0$, allora

$$T(n) \leq \alpha^{\lceil \log_\beta n/c \rceil} c_0 + c' n^k \sum_{i=0}^{\lceil \log_\beta n/c \rceil - 1} (\alpha/\beta^k)^i.$$

- consideriamo i 2 seguenti casi:
- $\alpha = \beta^k$: In questo caso si ha

$$\alpha^{\lceil \log_\beta n/c \rceil} c_0 = (\beta^k)^{\lceil \log_\beta n/c \rceil} c_0 < (\beta^k)^{\log_\beta (n/c)+1} c_0 = \beta^k (n/c)^k c_0 = O(n^k)$$

e

$$c' n^k \sum_{i=0}^{\lceil \log_\beta n/c \rceil - 1} (\alpha/\beta^k)^i = c' n^k \sum_{i=0}^{\lceil \log_\beta n/c \rceil - 1} 1 = c n^k \lceil \log_\beta n/c \rceil = O(n^k \log_\beta n).$$

Quindi $T(n) = O(n^k) + O(n^k \log_\beta n) = O(n^k \log_\beta n) = O(n^k \log n)$.

Soluzione delle relazioni di ricorrenza

- $\alpha \neq \beta^k$: In questo caso si ha

$$\begin{aligned} \alpha^{\lceil \log_\beta n/c \rceil} c_0 &< c_0 \alpha^{\log_\beta n/c + 1} = c_0 \alpha \cdot \alpha^{\log_\beta n/c} = c_0 \alpha \cdot \alpha^{\log_\alpha (n/c) \log_\beta \alpha} \\ &= c_0 \alpha (n/c)^{\log_\beta \alpha} = O(n^{\log_\beta \alpha}), \end{aligned} \tag{1}$$

e

$$c' n^k \sum_{i=0}^{\lceil \log_\beta n/c \rceil - 1} (\alpha/\beta^k)^i = c' n^k \cdot \frac{(\alpha/\beta^k)^{\lceil \log_\beta n/c \rceil} - 1}{(\alpha/\beta^k) - 1}. \tag{2}$$

Soluzione delle relazioni di ricorrenza

- Consideriamo i due sottocasi:
  - Caso $\alpha < \beta^k$:

$$\frac{(\alpha/\beta^k)^{\lceil \log_\beta n/c \rceil} - 1}{(\alpha/\beta^k) - 1} < \frac{1}{1 - (\alpha/\beta^k)} = \frac{\beta^k}{\beta^k - \alpha} < \beta^k = O(1).$$

Si ha quindi che la suddetta relazione insieme alla (1) e alla (2) della slide precedente implicano:

$$T(n) \leq O(n^{\log_\beta \alpha}) + n^k O(1) = O(n^k + n^{\log_\beta \alpha}).$$

Si noti che $\alpha < \beta^k$ implica $\log_\beta \alpha < k$ e di conseguenza si ha

$$T(n) = O(n^k + n^{\log_\beta \alpha}) = O(n^k).$$

  - Caso $\alpha > \beta^k$:

$$\begin{aligned} \frac{(\alpha/\beta^k)^{\lceil \log_\beta (n/c) \rceil} - 1}{(\alpha/\beta^k) - 1} &< \frac{(\alpha/\beta^k)^{\log_\beta (n/c)+1} - 1}{(\alpha/\beta^k) - 1} = \frac{(\alpha/\beta^k)(\alpha/\beta^k)^{\log_\beta (n/c)} - 1}{(\alpha/\beta^k) - 1} \\ &= O((\alpha/\beta^k)^{\log_\beta (n/c)}) = O((\alpha/\beta^k)^{\log_{\alpha/\beta^k}(n/c) \log_\beta (\alpha/\beta^k)}) \\ &= O((n/c)^{\log_\beta (\alpha/\beta^k)}) = O(n^{\log_\beta (\alpha) - k}) \end{aligned}$$

Si ha quindi che la suddetta relazione insieme alla (1) e alla (2) della slide precedente implicano:

$$T(n) \leq O(n^{\log_\beta \alpha}) + c' n^k O(n^{\log_\beta (\alpha) - k}) = O(n^{\log_\beta \alpha} + n^k n^{\log_\beta (\alpha) - k}) = O(n^{\log_\beta \alpha}).$$

Soluzione delle relazioni di ricorrenza

- Abbiamo stimato la funzione

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq c \\ \alpha T(n/\beta) + c' n^k & \text{altrimenti,} \end{cases}$$

  dove c' e k sono due costanti tali che $k \geq 0$, $c' > 0$.
- Abbiamo provato

$$T(n) = \begin{cases} O(n^k) & \text{se } \alpha < \beta^k \\ O(n^k \log n) & \text{se } \alpha = \beta^k \\ O(n^{\log_\beta \alpha}) & \text{se } \alpha > \beta^k \end{cases}$$

- **Esempi:** Nel caso di MergeSort $\alpha = 2$, $\beta = 2$ e $k = 1$. Si ha $\alpha = \beta^k$ e quindi $T(n) = O(n^k \log n) = O(n \log n)$.
  Nel caso dell'algoritmo per la ricerca binaria $\alpha = 1$, $\beta = 2$ e $k = 0$. Si $\alpha = \beta^k$ e quindi $T(n) = O(n^k \log n) = O(\log n)$.

Ordinamento per distribuzione

L'algoritmo di ordinamento per distribuzione (*quicksort*) opera nel modo seguente.

Decomposizione: se la sequenza ha almeno due elementi, scegli un elemento **pivot** e dividi la sequenza in due sotto-sequenze in modo tale che la prima contenga elementi minori o uguali al pivot e la seconda gli elementi maggiori o uguali del pivot.

Ricorsione: ordina ricorsivamente le due sotto-sequenze.

Ricombinazione: non occorre fare alcun lavoro.

```
1  QuickSort( a, sinistra, destra ):
2
3    IF (sinistra < destra) {
4      scegli pivot nell'intervallo [sinistra...destra];
5      indiceFinalePivot = Distribuzione(a, sinistra, pivot, destra);
6      QuickSort( a, sinistra, indiceFinalePivot-1 );
7      QuickSort( a, indiceFinalePivot+1, destra );
8    }
```

Distribuzione

- Data la posizione `px` del pivot in un segmento $a[\texttt{sx}, \texttt{dx}]$:
  - scambia gli elementi $a[\texttt{px}]$ e $a[\texttt{dx}]$, se $\texttt{px} \neq \texttt{dx}$
  - usa due indici `i` e `j` per scandire il segmento: `i` parte da `sx` e va verso destra e `j` parte da $\texttt{dx} - 1$ e va verso sinistra fino a quando $\texttt{i} \leq \texttt{j}$
  - ogni volta che si ha $a[\texttt{i}] > \texttt{pivot}$ e $a[\texttt{j}] < \texttt{pivot}$, scambia $a[\texttt{i}]$ con $a[\texttt{j}]$ e poi riprende la scansione
  - alla fine della scansione posiziona il pivot nella sua posizione corretta

Ordinamento per distribuzione

```
1  Distribuzione( a, sx, px, dx ):
2    IF (px != dx) Scambia( px, dx );
3    i = sx;
4    j = dx-1;
5    WHILE (i <= j) {
6      WHILE ((i <= j) && (A[i] <= A[dx]))
7        i = i+1;
8      WHILE ((i <= j) && (A[j] => A[dx]))
9        j = j-1;
10     IF (i < j) Scambia( i, j ); i=i+1,j=j-1;
11   }
12   IF (i != dx) Scambia( i, dx );
13   RETURN i;
```

```
1  Scambia( i, j ):                                  ⟨pre: sx ≤ i, j ≤ dx⟩
2    temp = a[j]; a[j] = a[i]; a[i] = temp;
```

Analisi di Distribuzione

- per stimare il tempo richiesto dal while esterno dobbiamo stimare il numero di iterazioni eseguite complessivamente dei due while interni.
- (numero di volte che incrementiamo `i`)+ (numero di volte che decrementiamo `j`)=$n-1$
- numero totale di iterazioni del primo while interno = numero confronti tra un elemento a[`i`] con il pivot
- numero totale di iterazioni del secondo while interno = numero confronti tra un elemento a[`j`] con il pivot
- dopo ogni confronto di a[`i`] con il pivot o viene incrementato `i` (nel while stesso o nell'if) o si esce dal while esterno
- dopo ogni confronto di a[`j`] con il pivot o viene decrementato `j` (nel while stesso o nell'if) o si esce dal while esterno
- numero totale di confronti con il pivot è quindi al più $n-1$ (in realtà si può vedere che è proprio $n-1$)
- la somma del numero di iterazioni del primo while interno e del numero di iterazioni del secondo while interno ' quindi al più $n-1$.
- tempo $O(n)$

Analisi di QuickSort mediante relazione di ricorrenza

Relazione di ricorrenza per il tempo $T(n)$ di esecuzione dell'algoritmo.

- Caso base: $T(n) \leq c_0$ per $n \leq 1$.
- Passo ricorsivo: sia r il rango dell'elemento pivot. Ci sono $r-1$ elementi a sinistra del pivot e $n-r$ elementi a destra, per cui $T(n) \leq T(r-1) + T(n-r) + cn$.

Analisi di QuickSort mediante relazione di ricorrenza

caso pessimo

- Il pivot è tutto a sinistra ($r = 1$) oppure tutto a destra ($r = n$). In entrambi i casi, la relazione diventa $T(n) \leq T(n-1) + T(0) + cn \leq T(n-1) + c'n$ per un'opportuna costante c'
- Applichiamo iterativamente la relazione di ricorrenza:

$$T(n) \leq T(n-1)+c'n \leq T(n-2)+c'(n-1)+c'n \leq \ldots \leq T(n-i)+\sum_{j=0}^{i-1} c'(n-j).$$

- Sostitutendo $i = n-1$ nell'espressione più a destra, otteniamo

$$T(n) \leq T(1)+\sum_{j=0}^{n-2} c'(n-j) = c_0+\sum_{j=2}^{n} c'j = c_0+c'(n+1)n/2-c' = O(n^2),$$

Analisi di QuickSort mediante relazione di ricorrenza

caso ottimo

- La distribuzione è bilanciata ($r = n/2$)), la ricorsione avviene su ciascuna metà
- In questa situazione, il costo è simile a quella dell'ordinamento per fusione.
- Possiamo dimostrare che il costo è di $O(n \log n)$ tempo

Efficienza del QuickSort randomizzato: intuizione

- Affinché QuickSort abbia tempo di esecuzione $O(n \log n)$ non è necessario che ogni volta il pivot sia l'elemento centrale ma è sufficiente che una frazione costante degli elementi risulti minore o uguale del pivot.
- Sia m la dimensione del segmento di array da ordinare in una certa chiamata ricorsiva. Supponiamo che il segmento venga suddiviso in due segmenti (incluso il pivot) di dimensione $m(\frac{1}{d})$ e $m(1 - \frac{1}{d})$, con $d > 1$ costante.
- Ovviamente quanto più sono diverse le lunghezze dei due segmenti (d molto piccolo o molto grande) tanto peggiore è il comportamento dell'algoritmo.
- Supponiamo che la chiamata ricorsiva in cui la suddivisione risulta più sbilanciata, suddivida il segmento da ordinare in due parti di dimensione pari rispettivamente a $\frac{1}{\beta}$ e $1 - \frac{1}{\beta}$, con $\beta > 1$ costante, della dimensione del segmento di partenza.
- Il tempo richiesto è sicuramente non più grande di quello che sarebbe richiesto se una tale suddivisione si verificasse ad ogni chiamata ricorsiva.
- Vale quindi la relazione di ricorrenza
  $T(n) \leq T(n/\beta) + T(n(1 - 1/\beta)) + cn$ per $n > 1$ e $T(n) \leq c_0$ per $n \leq 1$.
- Questa relazione ha soluzione $O(n \log n)$ per qualsiasi costante $\beta > 1$.

Efficienza del QuickSort randomizzato: intuizione

- Ci sono quindi molte possibili scelte del pivot che fanno in modo che l'algoritmo si comporti bene.
- Questo ci suggerisce che scegliere il pivot in modo random (con distribuzione di probabilità uniforme) porta con buona probabilità a scegliere un pivot "ben posizionato" e cioè un pivot che suddivide il segmento da ordinare nel modo descritto in precedenza e ad avere un tempo di esecuzione $O(n \log n)$.
- Si può dimostrare formalmente che il QuickSort randomizzato ha tempo di esecuzione medio $O(n \log n)$.

Selezione per distribuzione

Problema: selezione dell'elemento con rango r in un array a di n elementi distinti.

- Si vuole evitare di ordinare a
- NB: Il problema diventa quello di trovare il minimo quando $r = 1$ e il massimo quando $r = n$.

Osservazione: la funzione `Distribuzione` permette di trovare il rango del pivot, posizionando tutti gli elementi di rango inferiore alla sua sinistra e tutti quelli di rango superiore alla sua destra.

Possiamo modificare il codice del quicksort procedendo ricorsivamente nel *solo* segmento dell'array contenente l'elemento da selezionare.

La ricorsione ha termine quando il segmento è composto da un solo elemento.

Selezione per distribuzione

```
1  QuickSelect( a, sinistra, r, destra ):
2    IF (sinistra == destra) {
3      RETURN a[sinistra];
4    } ELSE {
5      scegli pivot nell'intervallo [sinistra...destra];
6      indiceFinalePivot = Distribuzione(a, sinistra, pivot, destra);
7      IF (r-1 == indiceFinalePivot) {
8        RETURN a[indiceFinalePivot];
9      } ELSE IF (r-1 < indiceFinalePivot) {
10       RETURN QuickSelect( a, sinistra, r, indiceFinalePivot-1 );
11     } ELSE {
12       RETURN QuickSelect( a, indiceFinalePivot+1, r, destra );
13     }
14   }
```

Analisi di QuickSelect mediante relazione di ricorrenza

- Caso base:
  Se il segmento sul quale opera l'algoritmo contiene un solo elemento allora l'algoritmo esegue un numero costante di operazioni $\leq c_0$;
  Se l'indice restituito da *Distribuzione*$(a, \mathit{sinistra}, \mathit{pivot}, \mathit{destra})$ è uguale a $r-1$ (in altre parole se il pivot ha rango r), l'algoritmo termina e il suo costo è dominato dal costo di Distribuzione che esegue $\leq c_1 n$ operazioni, dove c_1 è una certa costante.
- Passo ricorsivo: il numero di operazioni eseguite è al più pari a cn (c costante) più il numero di operazioni richieste per effettuare la selezione nel segmento degli elementi minori o uguali del pivot **oppure** in quello degli elementi maggiori o uguali del pivot.

Analisi di QuickSelect mediante relazione di ricorrenza

Relazione di ricorrenza per il tempo $T(n)$ di esecuzione dell'algoritmo.
Indichiamo con r_p il rango del pivot

- Caso base:
  $T(n) \leq c_0$ per $n \leq 1$ e
  $T(n) \leq c_1 n$ se $r_p = r$.
  In entrambi i casi $T(n) \leq c_1 n$.
- Passo ricorsivo: Ci sono $r_p - 1$ elementi a sinistra del pivot e $n - r_p$ elementi a destra, per cui $T(n) \leq \max\{T(r_p - 1), T(n - r_p)\} + cn$.

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

Analisi di QuickSelect mediante relazione di ricorrenza

caso pessimo

- Il pivot è tutto a sinistra ($r_p = 1$) e $r > r_p$ oppure tutto a destra ($r_p = n$) e $r < r_p$. In entrambi i casi, la relazione diventa $T(n) \leq T(n-1) + cn$.
- Applichiamo iterativamente la relazione di ricorrenza:

$$T(n) \leq T(n-1)+cn \leq T(n-2)+c(n-1)+cn \leq \ldots \leq T(n-i)+\sum_{j=n-i+1}^{n} c\,j.$$

- Sostitutendo $i = n-1$ nell'ultima disequazione, otteniamo

$$T(n) \leq T(1) + \sum_{j=2}^{n} c\,j \leq c_0 + \sum_{j=2}^{n} c\,j = c_0 + c\,n(n+1)/2 - c = O(n^2).$$

Analisi di QuickSelect mediante relazione di ricorrenza

caso ottimo

- L'elemento di rango r è proprio il pivot ($r_p = r$), per cui si esce dalla procedura senza effettuare la ricorsione e si ha che $T(n) = O(n)$.
- Il caso ottimo si verifica anche quando ad ogni chiamata ricorsiva viene dimezzata la lunghezza del segmento in cui effettuare la selezione.

$$T(n) \leq T(n/2) + cn \leq T(n/4) + c(n/2) + cn \leq \ldots \leq T(\frac{n}{2^i}) + \sum_{j=0}^{i-1} c\frac{n}{2^j}.$$

Dopo $\log n$ applicazioni della relazione di ricorrenza otteniamo

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{2^{\log n}}\right) + \sum_{j=0}^{\log n - 1} c\frac{n}{2^j} = T(1) + cn \sum_{j=0}^{\log n - 1} \frac{1}{2^j} \\ &\leq c_0 + cn\left(\frac{1 - 1/2^{\log n}}{1/2}\right) = c_0 + 2cn(1 - 1/n) = O(n) \end{aligned}$$

Efficienza del QuickSelect randomizzato: intuizione

- Per il QuickSelect, vale un discorso analogo a quello fatto per il QuickSort
- Ci sono molte possibili scelte del pivot che fanno in modo che l'algoritmo si comporti bene.
- Scegliendo il pivot in modo random (con distribuzione di probabilità uniforme) è probabile che si scelga un pivot "ben posizionato" e cioè un pivot tale che esiste una costante $a > 1$ per cui una frazione $1/a$ degli elementi sono minori o uguali del pivot e una frazione $1 - 1/a$ degli elementi sono maggiori o uguali del pivot.
- Si può dimostrare formalmente che il QuickSelect randomizzato ha tempo di esecuzione medio $O(n)$.

Moltiplicazione di interi

- Algoritmo che usiamo comunemente ha tempo di esecuzione $O(n^2)$, dove n e` il numero di cifre di ciascun numero

```
      2345 x
      5382 =

      ____
      4690
    18760
    7035
  11725

___________
  12620790
```

Moltiplicazione veloce di interi

Ogni numero intero w di n cifre può essere scritto come $10^{n/2} \times w_s + w_d$

- w_s indica il numero formato dalle $n/2$ cifre più significative di w
- w_d denota il numero formato dalle $n/2$ cifre meno significative.

Ad esempio 124100 può essere scritto come $10^3 \times 124 + 100$

Per moltiplicare due numeri x e y, vale l'uguaglianza

$$\begin{aligned} x\,y &= (10^{n/2}\,x_s + x_d)(10^{n/2}\,y_s + y_d) \\ &= 10^n\,x_s\,y_s + 10^{n/2}(x_s\,y_d + x_d\,y_s) + x_d\,y_d \end{aligned}$$

Decomposizione: se x e y hanno almeno due cifre, dividili come numeri x_s, x_d, y_s e y_d aventi ciascuno la metà delle cifre.

Ricorsione: calcola ricorsivamente le moltiplicazioni $x_s\,y_s$, $x_s\,y_d$, $x_d\,y_s$ e $x_d\,y_d$.

Ricombinazione: combina i numeri risultanti usando l'uguaglianza riportata sopra.

Moltiplicazione veloce di interi

- l'algoritmo esegue quattro moltiplicazioni di due numeri di $n/2$ cifre (ad un costo di $T(n/2)$), e tre somme di due numeri di n cifre (a un costo $O(n)$)
- la moltiplicazione per il valore 10^k può essere realizzata spostando le cifre di k posizioni verso sinistra e riempiendo di 0 la parte destra
- il costo della decomposizione e della ricombinazione è cn

Vale la relazione di ricorrenza

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

Moltiplicazione veloce di interi

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

Assumiamo per semplicità $n = 2^k$ per un certo k e applichiamo iterativamente la relazione di ricorrenza:

$$\begin{aligned} T(n) &\leq cn + 4T(n/2) \leq cn + 4(cn/2 + 4T(n/2^2)) = cn + 2cn + 4^2 T(n/2^2) \\ &\leq cn + 2cn + 4^2(cn/2^2 + 4T(n/2^3)) = cn + 2cn + 2^2 cn + 4^3 T(n/2^3) \\ &\leq \dots \\ &\leq cn + 2cn + 2^2 cn + \dots + 2^{i-1} cn + 4^i T(n/2^i) \\ &= cn \sum_{j=0}^{i-1} 2^j + 4^i T(n/2^i) = cn2^i - cn + 4^i T(n/2^i) \end{aligned}$$

Ponendo $i = k = \log_2 n$ si ha $T(n) \leq cn^2 - cn + n^2 T(1) = O(n^2)$.

Moltiplicazione veloce di interi

- È possibile progettare un algoritmo più veloce?
- Abbiamo visto che $x\,y = 10^n\,x_s\,y_s + 10^{n/2}(x_s\,y_d + x_d\,y_s) + x_d\,y_d$.
- Osserviamo che sommando e sottraendo $x_s y_s + x_d y_d$ a $x_s y_d + x_d y_s$ si ha

$$\begin{aligned} x_s y_d + x_d y_s &= x_s y_d + x_d y_s + x_s y_s + x_d y_d - x_s y_s - x_d y_d \\ &= x_s y_s + x_d y_d + (x_s y_d + x_d y_s - x_s y_s - x_d y_d) \end{aligned}$$

- Poiché $x_s y_d + x_d y_s - x_s y_s - x_d y_d = -(x_s - x_d) \times (y_s - y_d)$ allora possiamo scrivere

$$x_s y_d + x_d y_s = x_s y_s + x_d y_d - (x_s - x_d) \times (y_s - y_d)$$

- quindi il valore $x_s y_d + x_d y_s$ può essere calcolato facendo uso di $x_s y_s$, $x_d y_d$ e $(x_s - x_d) \times (y_s - y_d)$
- Quindi per computare il prodotto xy sono necessarie tre moltiplicazioni e non più quattro come prima

Moltiplicazione veloce di interi

Si ha quindi la relazione di ricorrenza

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

Assumiamo per semplicità $n = 2^k$, per un certo k, e applichiamo iterativamente la relazione di ricorrenza:

$$\begin{aligned}
T(n) &\leq cn + 3T(n/2) \leq cn + 3(cn/2 + 3T(n/2^2)) = cn + (3/2)cn + 3^2 T(n/2^2) \\
&\leq cn + (3/2)cn + 3^2(cn/2^2 + 3T(n/2^3)) = cn + (3/2)cn + (3/2)^2 cn + 3^3 T(n/2^3) \\
&\leq \dots \\
&\leq cn + (3/2)cn + (3/2)^2 cn + \dots + (3/2)^{i-1} cn + 3^i T(n/2^i) \\
&= cn \sum_{j=0}^{i-1} (3/2)^j + 3^i T(n/2^i) = cn \left(\frac{(3/2)^i - 1}{3/2 - 1} \right) + 3^i T(n/2^i) \\
&= 2cn((3/2)^i - 1) + 3^i T(n/2^i) = 2cn(3/2)^i - 2cn + 3^i T(n/2^i)
\end{aligned}$$

Continua nella prossima slide

Ponendo $i = k = \log_2 n$ si ha

$$
\begin{aligned}
T(n) &\leq 2cn(3/2)^{\log_2 n} - 2cn + 3^{\log_2 n} T(1) \\
&= 2cn \left(2^{\log_2(3/2)}\right)^{\log_2 n} - 2cn + \left(2^{\log_2 3}\right)^{\log_2 n} T(1) \\
&= 2cn \left(2^{\log_2 n}\right)^{\log_2(3/2)} - 2cn + \left(2^{\log_2 n}\right)^{\log_2 3} T(1) \\
&= 2cn\, n^{\log_2(3/2)} - 2cn + n^{\log_2 3} T(1) \\
&= 2cn\, n^{\log_2 3 - 1} - 2cn + n^{\log_2 3} T(1) \\
&= 2cn^{\log_2 3} - 2cn + n^{\log_2 3} T(1) \\
&\leq 2cn^{\log_2 3} - 2cn + n^{\log_2 3} c_0 \\
&= O(n^{\log_2 3}) = O(n^{1,585})
\end{aligned}
$$

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 $T(n) = O(n^{\log_2 3}) = O(n^{1,585})$

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 $T(n)= O(n)$

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

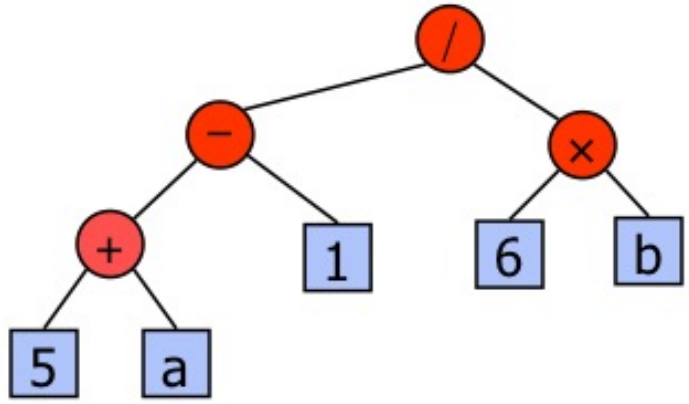
- **posticipata** (*postorder*):

```
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 rapprentata da un albero binario

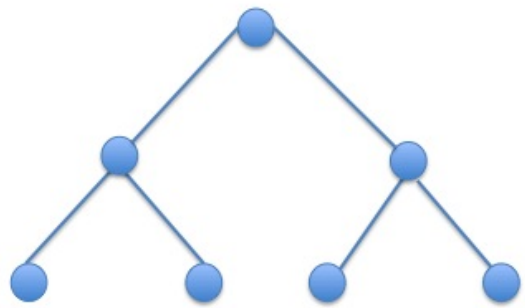
```
1  Valuta( u ):
2    IF (u==null) {
3      RETURN null;
4    }
5    IF (u.sx == null && u.dx==null) {
6      RETURN u.dato;
7    } ELSE {
8      valSinistra=Valuta( u.sx );
9      valDestra= Valuta( u.dx );
10     ris= Calcola(u.dato,valSinistra ,valDestra);
11     RETURN ris;
12   }
```

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



Algoritmo per verificare se un albero binario è completamente bilanciato

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 profondita' 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 profondita' 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 esserre 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   }
```