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.

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.

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

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*

```
Merge( a, sx, cx, dx ):
  i = sx; j = cx+1; k = 0;
  WHILE ((i <= cx) && (j <= dx)) {
    IF (a[i] <= a[j]) {
      b[k] = a[i]; i = i+1;
    } ELSE {
      b[k] = a[j]; j = j+1;
    }
    k = k+1;
  }
  FOR ( ; i <= cx; i = i+1, k = k+1)
    b[k] = a[i];
  FOR ( ; j <= dx; j = j+1, k = k+1)
    b[k] = a[j];
  FOR (i = sx; i <= dx; i = i+1)
    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:

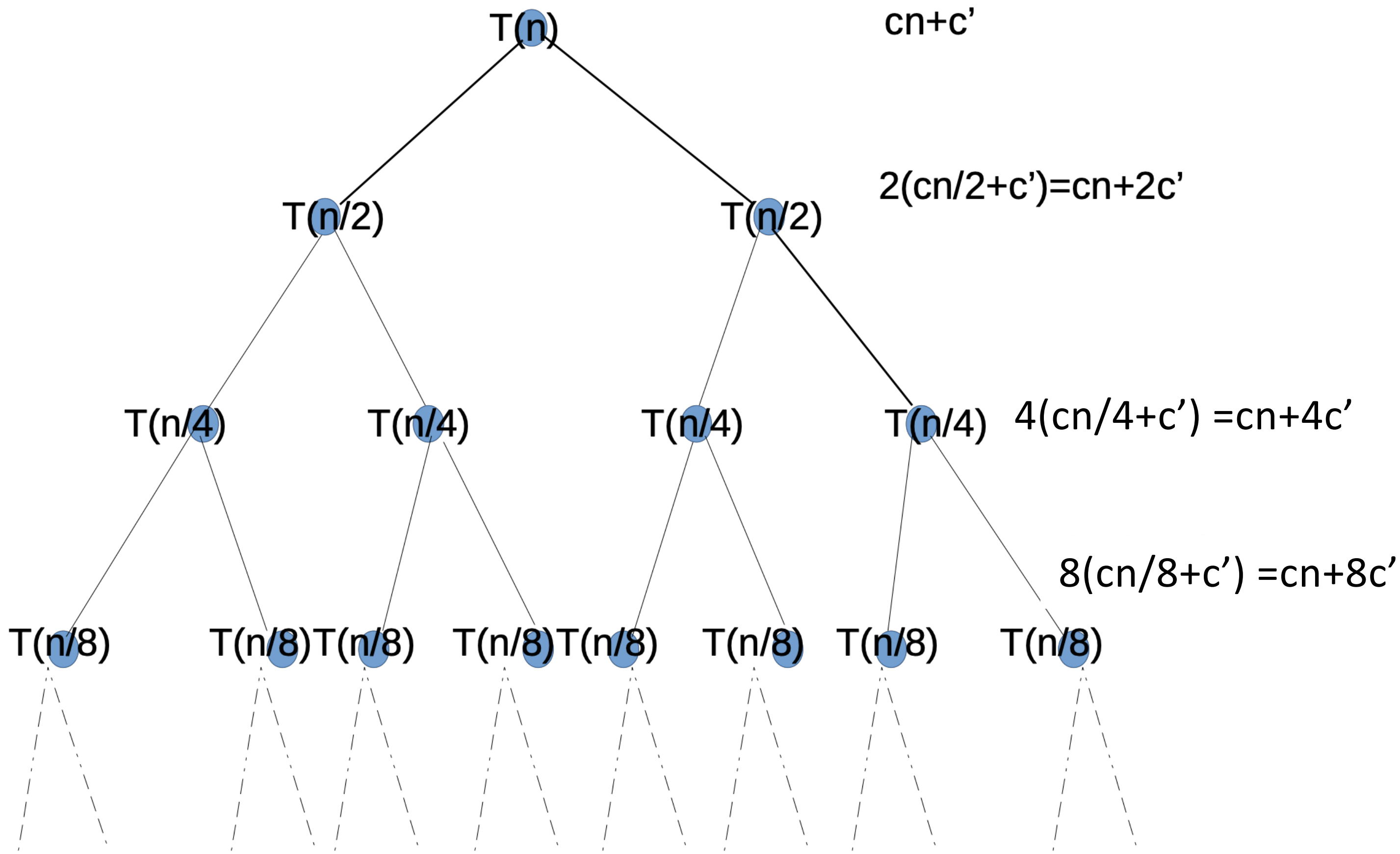
$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq c \\ \alpha T(n/\beta) + d(n) + r(n) & \text{altrimenti} \end{cases}$$

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



- $\log_2 n+1$ livelli: nodi di profondità 0, nodi di profondità 1, ..., nodi di profondità log n
- Il costo totale associato al livello dei nodi di profondità $i \leq \log_2 n$ è $cn+2^i c'$
- 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 è $c_0 n$
- sommando su tutti i livelli $\sum_{i=0}^{\log_2 n-1} (cn+2^i c')+c_0 n = cn\log_2 n+(2^{\log_2 n}-1)c'+c_0 n = cn\log_2 n+c'n-c'+c_0 n$

→ T(n)=Θ(nlogn)

Tempo di esecuzione di MergeSort

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

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

- 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

```
RicercaBinariaRicorsiva( a,k,sinistra,destra ):
  IF (sinistra > destra) {
    RETURN -1;
  }
  c = (sinistra+destra)/2;
  IF (k == a[c]) {
    RETURN c;
  }
  IF (sinistra==destra) {
    RETURN -1;
  }
  IF (k <a[c]) {
    RETURN RicercaBinariaRicorsiva( a,k,sinistra,c-1 );
  } ELSE {
    RETURN RicercaBinariaRicorsiva( a,k,c+1,destra );
  }
```

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
2. **Decomposizione**: per decomporre occorre calcolare l'indice centrale c e vedere se k è minore o maggiore di $a[c]$
3. **Ricorsione e ricombinazione**: di fatto non occorre nessun lavoro di ricombinazione

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

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

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 di $c + c_0$ per cui basta scegliere $c' \geq c + c_0$.
- 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 $c' = \max\{c + c_0, c\} = c + c_0$.

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.
- Affinché l'algoritmo individui correttamente x nell'insieme input o decida correttamente che x non è presente, ognuno dei possibili n+1 output deve essere associato ad una delle 3^t sequenze di risposte (altrimenti certi output non verrebbero mai prodotti) e ciascuna delle 3^t sequenze di risposte deve essere associata ad al più uno degli $n+1$ possibili output (altrimenti ci sarebbero più output possibili per una stessa sequenza di risposte).
- Deve quindi 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.

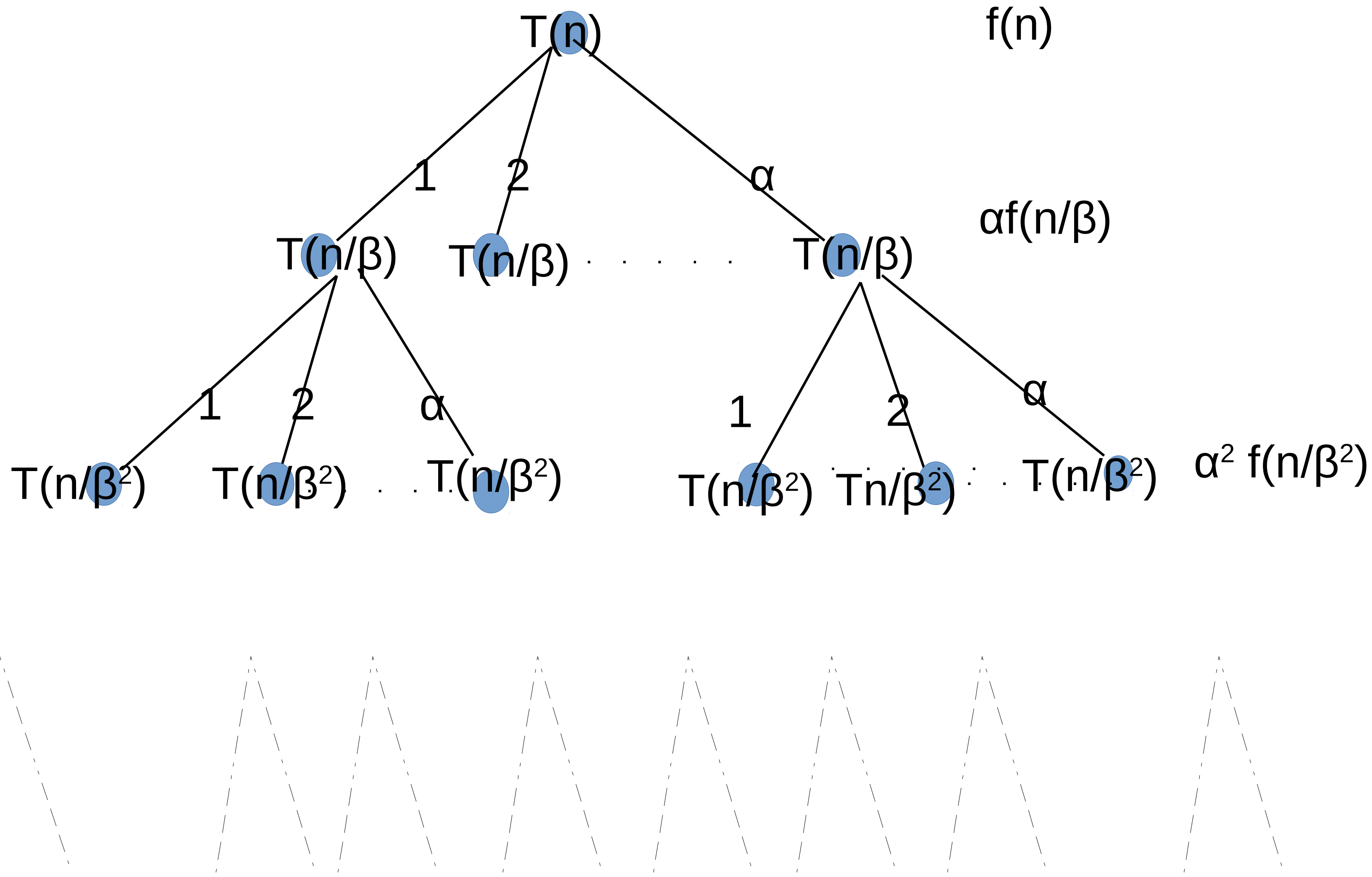
Conseguenza: l'algoritmo di ricerca binaria è asintoticamente ottimo.

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

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

- 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

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

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

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

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

- $\alpha \neq \beta^k$: 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:

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

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.

```
QuickSort( a, sinistra, destra ):

  IF (sinistra < destra) {
    scegli pivot nell'intervallo [sinistra…destra];
    indiceFinalePivot = Distribuzione(a, sinistra, pivot, destra);
    QuickSort( a, sinistra, indiceFinalePivot-1 );
    QuickSort( a, indiceFinalePivot+1, destra );
  }
```

Distribuzione

- Data la posizione $\texttt{px}$ 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 $\texttt{i}$ e $\texttt{j}$ per scandire il segmento: $\texttt{i}$ parte da $\texttt{sx}$ e va verso destra e $\texttt{j}$ 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

```
Distribuzione( a, sx, px, dx ):
  IF (px != dx) Scambia( px, dx );
  i = sx;
  j = dx-1;
  WHILE (i <= j) {
    WHILE ((i <= j) && (A[i] <= A[dx]))
      i = i+1;
    WHILE ((i <= j) && (A[j] => A[dx]))
      j = j-1;
    IF (i < j) Scambia( i, j ); i=i+1,j=j-1;
  }
  IF (i != dx) Scambia( i, dx );
  RETURN i;
```

```
Scambia( i, j ):                          ⟨pre: sx ≤ i,j ≤ dx⟩
  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 $T(n) \leq T(r - 1) + T(n - r) + cn$.

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

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

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

- Dimostriamo con il metodo di sostituzione che la relazione di ricorrenza

$$T(n) \leq \begin{cases} T(n/\beta) + T(n(1-1/\beta)) + cn & \text{se } n > 1 \\ c_0 & \text{per } n \leq 1 \end{cases}$$

  ha soluzione $O(n \log n)$ per qualsiasi costante $\beta > 1$.
- Dimostriamo per induzione che esiste una costante c' per cui $T(n) \leq c' n$ per ogni $n \geq 2$
- Base induzione: per n=2, T(n)$\leq 2c_0 + c_1$ (la costante c_1 limita il tempo di esecuzione del codice che precede le due chiamate ricorsive e che comprende il tempo costante per eseguire Distribuzione su un segmento di lunghezza 2 e le altre istruzioni che precedono le due chiamate ricorsive). Perché sia $T(2) \leq c' 2 \log 2 = 2c'$ basta quindi scegliere c' tale che $2c' \geq 2c_0 + c_1$, cioè $c' \geq c_0 + c_1/2$.

Efficienza del QuickSort randomizzato: intuizione

- Passo induttivo. Supponiamo vera la disuguaglianza per $2, \ldots, n-1$. Si ha

$$
\begin{aligned}
T(n) &\leq T(n/\beta) + T(n(1-1/\beta)) + cn \\
&\leq c'(n/\beta)\log(n/\beta) + c'(n(1-1/\beta)\log(n(1-1/\beta)) + cn \\
&= c'(n/\beta)(\log(n/\beta) - \log(n(1-1/\beta))) + c'n\log(n(1-1/\beta)) + cn \\
&= -c'(n/\beta)\log(\beta-1) + c'n\log(n(1-1/\beta)) + cn \\
&\leq -c'(n/\beta)\log(\beta-1) + c'n\log n + cn
\end{aligned}
$$

- Perché sia $T(n) \leq c'n\log n$ basta imporre $-(c'/\beta)\log(\beta-1) + c \leq 0$ che è soddisfatta per $c' \geq c\beta/(\log(\beta-1))$
- Quindi dobbiamo scegliere $c' = \max\{c_0 + c_1/2, c\beta/(\log(\beta-1))\}$.

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.

```
QuickSelect( a, sinistra, r, destra ):
  IF (sinistra == destra) {
    RETURN a[sinistra];
  } ELSE {
    scegli pivot nell'intervallo [sinistra…destra];
    indiceFinalePivot = Distribuzione(a, sinistra, pivot, destra);
    IF (r-1 == indiceFinalePivot) {
      RETURN a[indiceFinalePivot];
    } ELSE IF (r-1 < indiceFinalePivot) {
      RETURN QuickSelect( a, sinistra, r, indiceFinalePivot-1 );
    } ELSE {
      RETURN QuickSelect( a, indiceFinalePivot+1, r, destra );
    }
  }
```

- 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 $Distribuzione(a, sinistra, pivot, destra)$ 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 $O(n)$.