

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 la sequenza contiene due o più elementi, la sequenza viene suddivisa 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

Vediamo lo pseudocodice dell'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 MERGE

- Possiamo fondere due sequenze ordinate $A = \langle a_0, \dots, a_{n-1} \rangle$ e $B = \langle b_0, \dots, b_{m-1} \rangle$ 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 e i viene incrementato di 1. Se a_i è maggiore di b_j , b_j viene inserito alla fine della sequenza output 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, \dots, a_n alla fine della sequenza output; se $j \leq m$ trasferiamo uno dopo l'altro gli elementi b_j, \dots, 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 conseguenza 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

Vediamo lo pseudocodice dell'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 Merge 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 algoritmo 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:

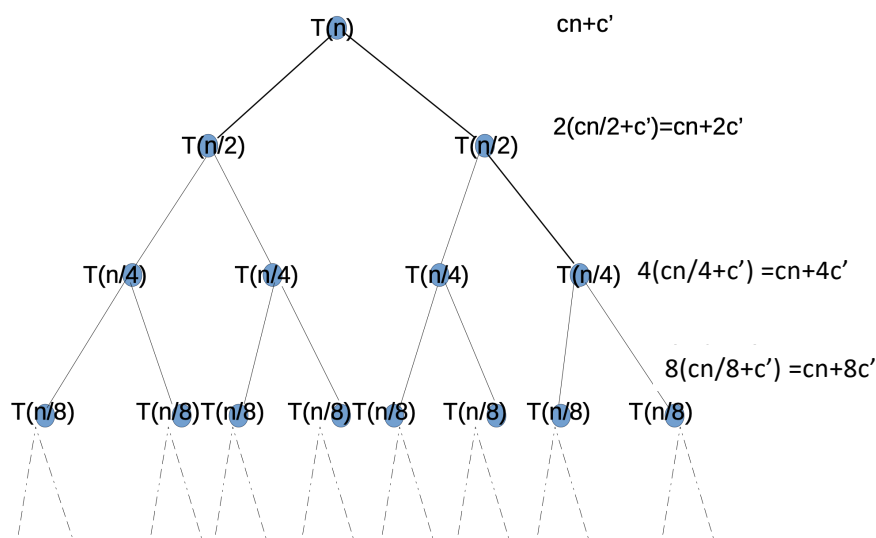
$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq c \\ \alpha T(n/\beta) + d(n) + r(n) & \text{altrimenti} \end{cases}$$

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 suoi 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 cn
- sommando su tutti i livelli $\sum_{i=0}^{\log_2 n - 1} (cn + 2^i c') + c_0 n = cn \log_2 n + (2^{\log_2 n} - 1)c' + c_0 n = cn \log_2 n + c'n - c' + c_0 n$
 $\rightarrow T(n) = \Theta(n \log n)$

TEMPO DI ESECUZIONE DI MERGESORT

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

- Iteriamo la ricorrenza

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

- Quante volte dobbiamo iterare la ricorrenza per raggiungere il caso base?
- Ogni volta che applichiamo la ricorrenza il valore dell'argomento di T viene dimezzato 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
- 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

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

- Per $i = \log n$ abbiamo

$$T(n) \leq T(1) + c \log n \leq c_0 + c \log n = O(\log n).$$

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 $T(1) + c = c_0 + c$ per cui basta scegliere $c' \geq c + c_0$.
- Passo induttivo: Supponiamo che per $2, \dots, n-1$ 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 $i = 1, 2, \dots$
- 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 $O(\log 2^{h-1}) = O(h) = O(\log x)$
- ⑥ In totale $O(\log x)$: $h = \lceil \log x \rceil$ domande fatte per individuare l'intervallo $[2^{h-1} + 1, 2^h]$ e $h-1$ domande per cercare x in $[2^{h-1} + 1, 2^h]$.

array ordinato $A = \langle a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 \rangle$

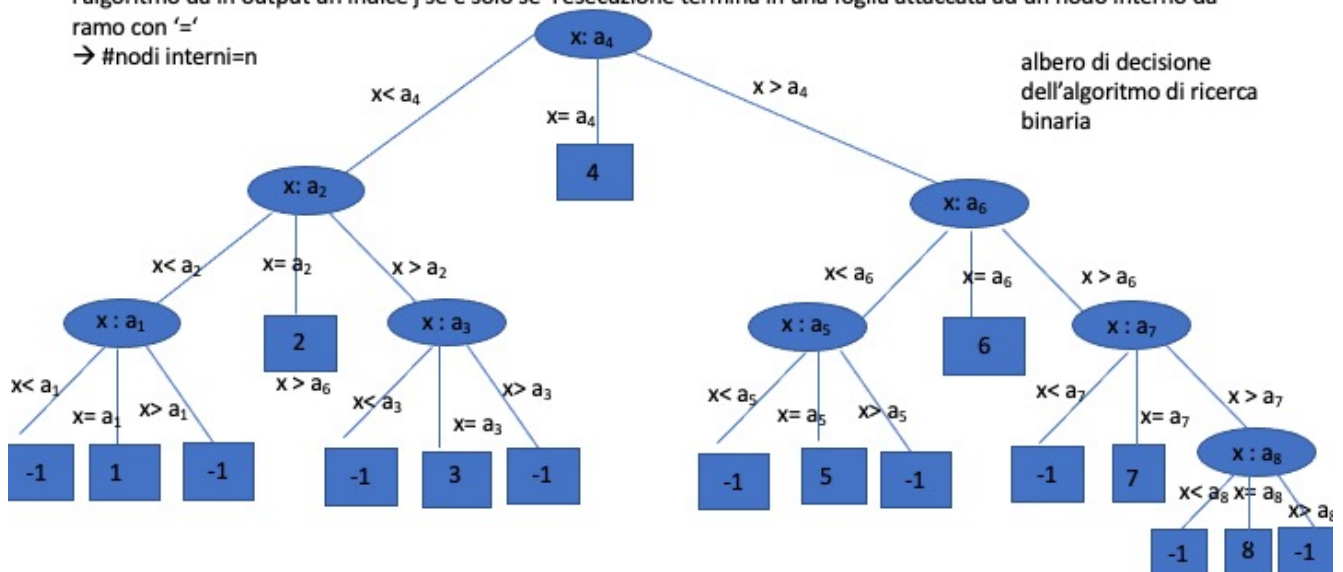
albero di decisione descrive tutte le possibili sequenze di confronti che potrebbe fare l'algoritmo.

nell'albero di decisione i nodi interni rappresentano i confronti, mentre le foglie rappresentano gli output prodotti. \rightarrow altezza albero=numero confronti nel caso pessimo

l'algoritmo dà in output un indice j se e solo se l'esecuzione termina in una foglia attaccata ad un nodo interno da ramo con '='

\rightarrow #nodi interni= n

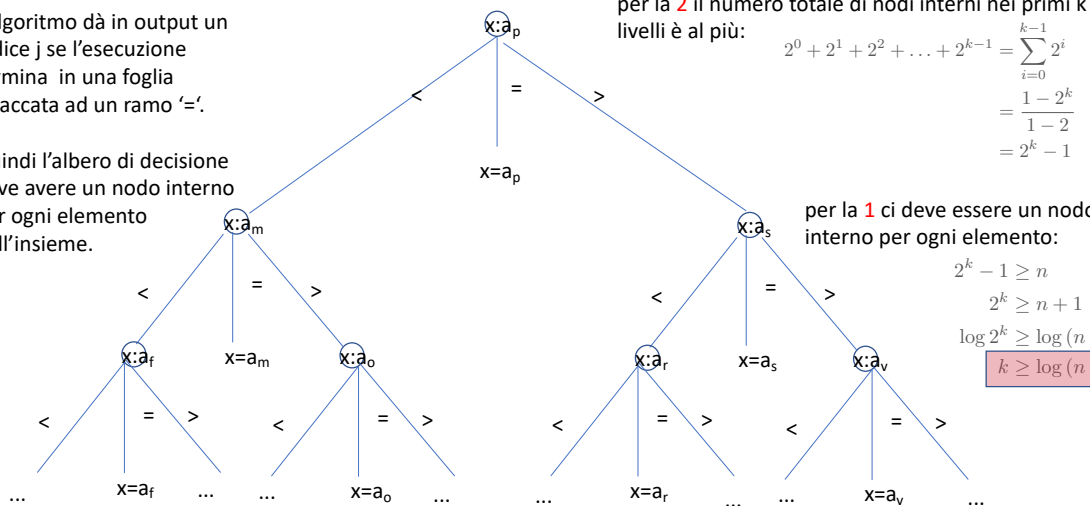
albero di decisione dell'algoritmo di ricerca binaria



LOWER BOUND SULLA RICERCA SU UN INSIEME ORDINATO

1 L'algoritmo dà in output un indice j se l'esecuzione termina in una foglia attaccata ad un ramo '='.

Quindi l'albero di decisione deve avere un nodo interno per ogni elemento dell'insieme.



per la **2** il numero totale di nodi interni nei primi k livelli è al più:

$$2^0 + 2^1 + 2^2 + \dots + 2^{k-1} = \sum_{i=0}^{k-1} 2^i = \frac{1 - 2^k}{1 - 2} = 2^k - 1$$

per la **1** ci deve essere un nodo interno per ogni elemento:

$$2^k - 1 \geq n$$

$$2^k \geq n + 1$$

$$\log 2^k \geq \log(n + 1)$$

$$k \geq \log(n + 1)$$

2 Ogni nodo interno "produce" al più due nodi interni per cui ci sono al più 2^i nodi interni (confronti) a livello i . NB: i nodi nell'ultimo livello sono foglie ciascuna delle quali è attaccata ad un ramo '=' (elemento trovato) o ad un ramo '<' o '>' (elemento non trovato).

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

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 $a[sx, dx]$:
 - scambia gli elementi $a[px]$ e $a[dx]$, se $px \neq dx$
 - usa due indici i e j per scandire il segmento: i parte da sx e va verso destra e j parte da $dx - 1$ e va verso sinistra fino a quando $i \leq j$
 - ogni volta che si ha $a[i] > pivot$ e $a[j] < pivot$, scambia $a[i]$ con $a[j]$ 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:  $sx \leq i, j \leq dx$ >
2   temp = a[j]; a[j] = a[i]; a[i] = temp;
```

ANALISI DI DISTRIBUZIONE

- ① per stimare il tempo richiesto dal while esterno dobbiamo stimare il numero di iterazioni eseguite complessivamente dei due while interni.
- ② numero totale di iterazioni del primo while interno = numero confronti tra un elemento $a[i]$ con il pivot
- ③ numero totale di iterazioni del secondo while interno = numero confronti tra un elemento $a[j]$ con il pivot
- ④ dopo ogni confronto di $a[i]$ con il pivot o viene incrementato i (nel while stesso o nell'if). Fa eccezione solo il caso in cui $i = j$ e $a[i] > a[\text{dx}]$.
- ⑤ dopo ogni confronto di $a[j]$ con il pivot o viene decrementato j (nel while stesso o nell'if)
- ⑥ per i due punti precedenti si ha che il numero totale di confronti sul totale di tutte le iterazioni dei due for interni è minore o uguale di $1 +$ numero di volte in cui viene incrementato i + il numero di volte in cui viene decrementato j .
- ⑦ dal momento che il while esterno termina quando $i = j + 1$ allora il numero totale di volte in cui viene incrementato i più il numero di volte in cui viene decrementato j è $n - 1$. Di conseguenza il numero totale di confronti è al più n così come pure il numero totale di iterazione dei due while interni.
- ⑧ tempo $O(n)$

ANALISI DI QUICKSORT MEDIANTE RELAZIONE DI RICORRENZA

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

- 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:

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

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

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

- Nella sommatoria sostituiamo j con $k = n - j$ e otteniamo

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

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 (escluso il pivot) di dimensione rispettivamente pari circa a $(m-1)(\frac{1}{d})$ e $(m-1)(1 - \frac{1}{d})$, con $d > 1$ costante. Diciamo "circa" perché in realtà per un segmento occorre prendere la parte intera superiore e per l'altra quella inferiore.
- 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 (privato del pivot) in due parti di dimensione pari rispettivamente a circa $\frac{1}{\beta}$ e $1 - \frac{1}{\beta}$ della dimensione del segmento, dove β è una costante positiva.
- Il tempo richiesto è sicuramente non più grande di quello che sarebbe richiesto se una tale suddivisione si verificasse per ogni chiamata ricorsiva su input maggiori di β . Per input di dimensione minore o uguale di β ci mettiamo nel caso peggiore, cioè quello in cui un segmento è vuoto e l'altro contiene tutti gli elementi diversi dal pivot. Il tempo sarà comunque limitato da una costante che indichiamo con c_1 .

EFFICIENZA DEL QUICKSORT RANDOMIZZATO: INTUIZIONE

- Omettiamo le parti intere inferiori e superiori. Si può dimostrare che ciò non influisce sul comportamento asintotico della ricorrenza.
- Consideriamo quindi la relazione di ricorrenza

$$T(n) \leq \begin{cases} T((n-1)/\beta) + T((n-1)(1-1/\beta)) + cn & \text{se } n > \beta \\ c_1 & \text{per } n \leq \beta \end{cases}$$

Vogliamo dimostrare che questa relazione di ricorrenza ha soluzione $O(n \log n)$ per qualsiasi costante $\beta > 1$.

- Possiamo assumere che $\beta \neq 2$ in quanto abbiamo già visto che in quel caso $T(n) = O(n \log n)$. Possiamo inoltre assumere senza perdere di generalità che $\beta > 2$, cioè che il primo segmento sia più piccolo del secondo.
- Dimostriamo con il metodo della sostituzione che $T(n) = O(n \log n)$. Per far ciò dimostreremo per induzione che esiste una costante $c' > 0$ per cui $T(n) \leq c' n \log n$ per ogni $n \geq \beta$.
- Base induzione: per $n = \beta$, si ha $T(\beta) \leq c_1$. Perché sia $T(\beta) \leq c'(\beta \log \beta)$ basta quindi scegliere c' tale che $c' \geq c_1/(\beta \log \beta)$.

EFFICIENZA DEL QUICKSORT RANDOMIZZATO: INTUIZIONE

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

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

- Perché risulti $T(n) \leq c'n \log n$ basta imporre $-(c'/\beta) \log(\beta-1) + c \leq 0$ che è soddisfatta per $c' \geq c\beta/(\log(\beta-1))$
- Quindi dobbiamo scegliere

$$c' = \max\{c_1/(\beta \log \beta), c\beta/(\log(\beta-1))\}.$$

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.

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 per cui il costo è $\leq c_0$ per una certa costante c_0 positiva.
Se l'indice restituito da *Distribuzione*($a, sinistra, pivot, destra$) è uguale a $r - 1$, l'algoritmo termina. Il costo in questo caso è dato dal costo lineare di *Distribuzione* più il costo costante delle altre istruzioni per cui il costo totale è $\leq c_1 n$, dove $c_1 > 0$ è una certa costante.
- Passo ricorsivo: Il costo in questo caso è dato dal costo lineare di *Distribuzione* più il costo costante delle altre istruzioni e il costo della chiamata ricorsiva sul segmento degli elementi minori del pivot **oppure** in quello degli elementi maggiori del pivot. Il costo in questo caso è quindi al più pari a cn (per una certa costante $c > 0$) più il costo della chiamata ricorsiva.

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

$$T(n) \leq \begin{cases} c_0 & \text{se } n = 1 \\ c_1 n & \text{se } n > 1 \text{ e } r_p = r - 1 \\ \max\{T(r_p - 1), T(n - r_p)\} + cn & \text{altrimenti} \end{cases}$$

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:

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

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

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

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.

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

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 $O(n)$.