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;
- 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      Fusione( 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 in tempo lineare nel numero totale di elementi. Per il momento ignoriamo il fatto che le due sequenze sono memorizzare in due segmenti adiacenti di un array.
- L'idea dell'algoritmo è il seguente:
  1. Confrontiamo i due elementi più piccoli delle due sequenze. Siccome le due sequenze sono ordinate i due elementi più piccoli si trovano all'inizio delle due sequenze.
  2. L'elemento più piccolo tra i due confrontati viene rimosso dalla sequenza in cui si trova e inserito alla fine della sequenza output.
  3. Eseguiamo i due passi precedenti fino a che una delle due sequenze si svuota. A questo punto aggiungiamo gli elementi rimasti nell'altra sequenza alla fine della sequenza output, nello stesso ordine in cui erano disposti in origine.

L'algoritmo Fusione

- Il tempo dipende dal numero di volte che eseguiamo i passi 1 e 2 e dal numero di elementi da spostare al passo 3.
- Ogni volta che eseguiamo i passi 1 e 2, cancelliamo un elemento di una delle due sequenze. Di conseguenza, non possiamo eseguire i passi 1 e 2 più di $m - 1$ volte. Se eseguissimo questi due passi m volte, sicuramente entrambe le sottosequenza rimarrebbero senza elementi. Ciò è impossibile perchè noi smettiamo di applicare il passi 1 e 2 non appena una delle due sottosequenze si svuota. Chiamiamo k il numero di esecuzioni dei passi 1 e 2. Ovviamente, per quanto detto $k < m$.
- Se i passi 1 e 2 vengono eseguiti k volte allora vuol dire che una delle due sequenze è vuota e l'altra contiene $m - k$ elementi e il il passo 3 richiede di spostare $m - k$ elementi.

Fusione: algoritmo Fusione

Descriviamo l'algoritmo Fusione 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  Fusione( 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 Fusione

- L'algoritmo ha tempo di esecuzione $O(m)$, dove m è il numero totale di elementi da fondere.
  - Una singola 'esecuzione del corpo di ciascuno dei cicli presenti nell'algoritmo richiede tempo $O(1)$
  - Ad ogni iterazione del ciclo di while viene incrementato almeno uno tra i e j e il ciclo termina quando sono stati scanditi tutti gli elementi di uno dei due segmenti. Di conseguenza il corpo del while viene iterato al più un numero di volte pari ad m-1. Chiamiamo k il numero di iterazioni del corpo del while. Ovviamente $k < m$.
  - Viene eseguito solo uno dei due cicli di for successivi. Tale ciclo viene iterato un numero di volte pari al numero di elementi che non sono stati copiati nell'array b nel ciclo precedente. Il numero di questi elementi è pari a $m - k$.
  - L'ultimo ciclo di for viene iterato un numero di volte pari al numero totale m di elementi dei due segmenti.

Analisi dell'algoritmo MergeSort

Ora che sappiamo qual è il tempo di esecuzione dell'algoritmo Fusione, 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 : $O(1)$,
- 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: $cn = O(n)$.

Si ha quindi $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + cn = O(?)$.

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

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

Tempo di esecuzione di MergeSort

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

$$\begin{aligned} T(n) &\leq cn + 2T(n/2) \leq cn + 2(cn/2 + 2T(n/4)) = 2cn + 4T(n/4) \\ &\leq 2cn + 4(cn/4 + 2T(n/8)) = 3cn + 8T(n/8) \\ &\leq \ldots \leq icn + 2^i T\left(\frac{n}{2^i}\right) \end{aligned}$$

- 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

$$T(n) \leq cn \log n + 2^{\log n} T(1) \leq cn \log n + 2^{\log n} c_0 = cn \log n + c_0 n = O(n \log n).$$

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

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

La ricerca binaria in un array ordinato richiede $O(\log n)$ passi.

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 2^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.

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 );
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

- (numero di volte che incrementiamo `i`)+ (numero di volte che decrementiamo `j`)=$n-1$
- ogni confronto di a[`i`] con il pivot porta a incrementare `i` (o prima o dopo aver effettuato lo scambio)
- ogni confronto di a[`j`] con il pivot porta a decrementare `j` (o prima o dopo aver effettuato lo scambio)
- numero di confronti con il pivot è quindi al più $n-1$ (in realtà si può vedere che è proprio $n-1$)
- 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

Analisi di QuickSort mediante relazione di ricorrenza

caso medio

- Tempo $O(n \log n)$ in quanto nelle chiamate ricorsive si alternano situazioni in cui gli elementi si distribuiscono in modo bilanciato a sinistra e a destra del pivot e situazioni in cui gli elementi si distribuiscono in modo non bilanciato. L'algoritmo è veloce in pratica.