Algoritmi e strutture dati

Algoritmi di ordinamento

Tecniche differenti:

- **Ordinamento interno**: dati da ordinare in memoria centrale *(quelli che studieremo noi)*
  - Accesso diretto agli elementi
- **Ordinamento esterno**: dati da ordinare in memoria di massa
  - Accesso a blocchi di dati. Accedere a questi dati è più lento in quanto prelevati da periferiche

Ordinamento interno

Si ordinano array di strutture complesse, come oggetti o record (`struct` in Go e in C)

Un particolare *campo* di queste strutture è scelto come *chiave* per l'ordinamento.

- Per semplicità negli esempi verranno ordinati array di interi

Stabilità

Un algoritmo di ordinamento è detto stabile se preserva l'ordine relativo tra record con la medesima chiave

$$\begin{array}{cccc} \text{cane} & \text{gatto} & \text{ape} & \longrightarrow & \text{cane} & \text{ape} & \text{gatto} \\ 10 & 12 & 10 & & 10 & 10 & 12 \end{array}$$

Complessità degli algoritmi di sort

Studieremo principalmente algoritmi di ordinamento basati su confronti tra chiavi

Spazio: considereremo la memoria utilizzata in aggiunta all'array da ordinare (inclusa la memoria sullo stack nel caso di algoritmi ricorsivi)
Tempo: stimeremo la complessità in tempo di questi algoritmi, in funzione della lunghezza del vettore da ordinare, calcolando prima di tutto il numero di confronto tra chiavi.

- Utilizzeremo il numero di confronti perchè: sono le operazioni più costose utilizzate da questi algoritmi; se ciascun confronto viene effettuato in tempo costante il numero di confronti fornisce una stima del tempo di calcoli; in generale il calcolo può essere stimato moltiplicando il numero di confronti per il tempo utilizzato da ciascun confronto

Tecniche di ordinamento interno

Queste tecniche sono basate su confronti

Algoritmi elementari

- per selezione `selectionSort`
- per inserimento `insertionSort`
- a bolle `bubbleSort`

Questi algoritmi utilizzano $\Theta(n^2)$ confronti

Algoritmi avanzati

- per fusione `mergeSort`
- veloce `quickSort`
- basato su "heap" `heapSort`

Questi algoritmi utilizzano $\Theta(n \log n)$ confronti

- Non è possibile ordinare tramite i confronti utilizzando meno di $n \log n$ confronti
- L'unica eccezione è il quick sort, che nel caso peggiore utilizza $\Theta(n^2)$ confronti
  - Nonostante ciò è uno degli algoritmi di sort più utilizzato poiché questi casi sono molto rari

Sono tutti algoritmi *in loco*, cioè non necessitano di strutture ausilarie, eccetto il `mergeSort` e, per certi aspetti, il `quickSort`. Non tutti questi algoritmi sono inoltre stabili.

Selection sort

```
ALGORITMO selectionSort(Array A[0..n-1])
    FOR k <- 0 TO n-2 DO
    |   //ricerca la posizione dell'elemento minimo in A[k..n-1]
    |   m <- k
    |   FOR j <- k+1 TO n-1 DO
    |   |   IF A[j] < A[m] THEN
    |   |       m <- j
    |   scambia A[m] con A[k]
```

- Si consideri l'array $[\ 10_0\ 10_1\ 5\ ]$. Utilizzando questo algoritmo per ordinarlo si arriva a $[\ 5\ 10_1\ 10_0\ ]$.
  Di conseguenza questo algoritmo **non è stabile**.

Numero di confronti

Alla k-esima iterazione del ciclo principale si fanno nel ciclo interno $n-k-1$ confronti

$$\sum_{k=0}^{n-2} \#\text{ confronti dell'iterazione } k = \sum_{k=0}^{n-2}(n-k-1) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \Theta(n^2)$$
$$\downarrow$$
$$n-k-1=i$$

Spazio: è un algoritmo che funziona in loco, di conseguenza lo spazio che utilizza è $O(1)$

Insertion sort

```
ALGORITMO insertionSort(Array A[0..n-1])
    FOR k <- 1 TO n-1 DO
    //inserisci A[k] al posto giusto tra A[0]..A[k-1]
    |   x <- A[k]
    |   j <- k-1
    |   WHILE j >= 0 AND A[j] > x DO                        [1]
    |   |   A[j+1] <- A[j]
    |   |   j <- j-1
    |   A[j+1] <- x
```

- Quando tornando indietro si trova un elemento uguale, non lo si sposta. Questo è un algoritmo **stabile**.
- Si ricerca andando a ritroso, ossia da destra verso sinistra, la posizione dell'elemento x nell'array. Nel mentre che si fa ciò gli si crea anche il posto andando a spostare tutti gli elementi.

Nota: se ci si muove da sinistra verso destra si trova la posizione, ma poi bisogna spostare tutti gli elementi dopo per creare il posto. Facendolo invece da destra verso sinistra, nel ciclo while si trova la posizione dell'elemento da spostare e nel frattempo gli si crea il posto perché man mano li confronto e li sposto.

Numero di confronti

Si prenda in considerazione l'iterazione k [1]: nel caso peggiore si fanno k confronti, nel caso migliore si fa un confronto.

Caso peggiore

All'iterazione k nel caso peggiore si fanno k confronti. Il caso peggiore si ha quando l'array è ordinato al contrario.

$$\#\text{ confronti nel caso peggiore} = \sum_{k=1}^{n-1} k = \frac{(n-1)n}{2} = \Theta(n^2)$$

Caso migliore

Il caso migliore lo si trova quando l'array è già ordinato: in questo caso all'iterazione k si fa un solo confronto.

$$\#\text{ confronti nel caso migliore} = \sum_{k=1}^{n-1} 1 = n-1$$

Spazio: è un algoritmo che funziona in loco, di conseguenza lo spazio che utilizza è $O(1)$

Bubble sort

```
ALGORITMO bubbleSort(Array A[0..n-1])
    i <- 1
    DO
    |   scambiato <- false
    |   FOR j <- 1 TO n-i DO
    |   |   IF A[j] < A[j-1] THEN
    |   |   |   scambia A[j] e A[j-1]
    |   |   |   scambiato <- true
    |   i <- i+1
    WHILE scambiato AND i < n
```

Il bubble sort è un algoritmo di ordinamento **stabile**.

Numero di confronti

Caso migliore

Nel caso migliore (ossia in un array già ordinato) questo algoritmo fa $n-1$ confronti: $\Theta(n)$

Caso peggiore

Nel caso peggiore, all'i-esima iterazione si fanno nel ciclo for $n-1$ confronti

$$\#\text{ confronti nel caso peggiore} = \sum_{i=1}^{n-1}(n-i) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2)$$

Spazio: è un algoritmo che funziona in loco, di conseguenza lo spazio che utilizza è $O(1)$

Merge

Dati due array B e C ordinati in maniera non decrescente, costruire un array X ordinato in modo non decrescente contenente tutti gli elementi di B e di C

```
ALGORITMO merge(Array B[0..l_b-1], Array C[0..l_c-1]) -> Array
    sia X[0 .. l_b + l_c - 1] un array
    i_1 <- 0, i_2 <- 0, k <- 0

    WHILE i_1 < l_b AND i_2 < l_c DO // ciclo fino a quando nessuno dei due array è finito
    |   IF B[i_1] <= C[i_2] THEN
    |   |   X[k] <- B[i_1]
    |   |   i_1 <- i_1 + 1
    |   ELSE
    |   |   X[k] <- C[i_2]
    |   |   i_2 <- i_2 + 1
    |   k <- k + 1

    IF i_1 < l_b THElN // se B non è finito
    |   FOR j <- i_1 TO l_b - 1 DO
    |   |   X[k] <- B[j]
    |   |   k <- k + 1
    ELSE // se C non è finito
    |   FOR j <- i_2 TO l_c - 1 DO
    |   |   X[k] <- C[j]
    |   |   k <- k + 1

    RETURN X
```

Prestazioni

Spazio: array aggiuntivo per il risultato e tre variabili, di conseguenza è costante.
Numero di confronti: sia $n = l_B + l_C$, nel caso peggiore si fanno $n-1$ confronti.

Merge tramite indici

Dato un array A, siano B e C due sue porzioni contigue, entrambe ordinate in maniera crescente. Si utilizza un array di supporto X per ordinare le due porzioni, per poi ricopiarlo in A

```
PROCEDURA merge(Array A, intero i, intero m, intero f, Array X)
    i_1 <- i, i_2 <- m, k <- 0

    WHILE i_1 < m AND i_2 < f DO
    |   IF B[i_1] <= C[i_2] THEN
    |   |   A[k] <- A[i_1]
    |   |   i_1 <- i_1 + 1
    |   ELSE
    |   |   X[k] <- A[i_2]
    |   |   i_2 <- i_2 + 1
    |   k <- k + 1

    IF i_1 < m THElN    // se la prima porzione non è finita
    |   FOR j <- i_1 TO m - 1 DO
    |   |   X[k] <- A[j]
    |   |   k <- k + 1
    ELSE                // se la seconda porzione non è finita
    |   FOR j <- i_2 TO f - 1 DO
    |   |   X[k] <- A[j]
    |   |   k <- k + 1

    FOR k <- 0 TO f - i - 1 DO
        A[i+k] <- X[k]
```

Prestazioni

Spazio: tre variabili, di conseguenza è costante.
Numero di confronti: sia $n = l_B + l_C$, nel caso peggiore si fanno $n - 1$ confronti.

Merge sort

Concetto di base: Array $A[0..n-1]$

Se $n \leq 1$ allora A è già ordinato, altrimenti:

- si divide A in due parti della stessa lunghezza
- si ordinano le due parti separatamente
- si fondono i risultati in un array ordinato

Nel merge sort il processo di divisione è semplice, mentre la ricombinazione (il merge) è complessa. Il merge sort è un algoritmo di ordinamento **stabile**.

Implementazione tramite array

```
ALGORITMO mergeSort(Array A[0..n-1])
    IF n > 1 THEN
    |   m <- n/2
    |   B <- A[0..m-1] //prima metà di A
    |   C <- A[m. .n-1] //seconda metà di A
    |   //ordino le due parti ricorsivamente
    |   mergeSort(B)
    |   mergeSort(C)
    |   A <- merge(B, C)  //combino le due soluzioni
```

Equazione di ricorrenza

$$C(n) = \begin{cases} C(\lfloor \frac{n}{2} \rfloor) + C(\lceil \frac{n}{2} \rceil) + C_{\text{merge}}(n) & \text{se } n > 1 \\ 0 & \text{altrimenti} \end{cases}$$

$\overset{\text{mergeSort(B)}}{\downarrow} \quad \overset{\text{mergeSort(C)}}{\downarrow} \quad \overset{\text{merge(B, C)}}{\downarrow}$

dove $C(n)$ è il $\#$ totale di confronti di `mergeSort` per un array di lunghezza n

Per n pari:

$$C(n) = \begin{cases} 0 & \text{se } n = 1 \\ 2C\left(\frac{n}{2}\right) + n - 1 & \text{altrimenti} \end{cases}$$

$$\begin{aligned} C(n) &= 2C\left(\frac{n}{2}\right) + n - 1 = 2\left[2C\left(\frac{n}{2^2}\right) + \frac{n}{2} - 1\right] + n - 1 = 2^2 C(\frac{n}{2^2}) + n - 2 + n - 1 = \\ &= 2^2\left[2C\left(\frac{n}{2^3}\right) + \frac{n}{2^2} - 1\right] + n - 2 + n - 1 = 2^3 C(\frac{n}{2^3}) + n - 2^2 + n - 2^1 + n - 2^0 = \\ &= \cdots = 2^k C\left(\frac{n}{2^k}\right) + n - 2^{k-1} + n - 2^{k-2} + \cdots + n - 2^1 + n - 2^0 = \\ &= 2^k C\left(\frac{n}{2^k}\right) + kn - \sum_{i=0}^{k-1} 2^i = 2^k C\left(\frac{n}{2^k}\right) + kn - 2^k + 1 = \\ &= nC(1) + n\log_2 n - n + 1 = n\log_2 n - n + 1 \end{aligned}$$

$\downarrow$

$\frac{n}{2^k} = 1$ per $n = 2^k$, cioè $k = \log_2 n$; $\quad nC(1) = 0$

Per n potenza di 2:

$$C(n) = n\log_2 n - n + 1 = \Theta(n \log n)$$

In generale:

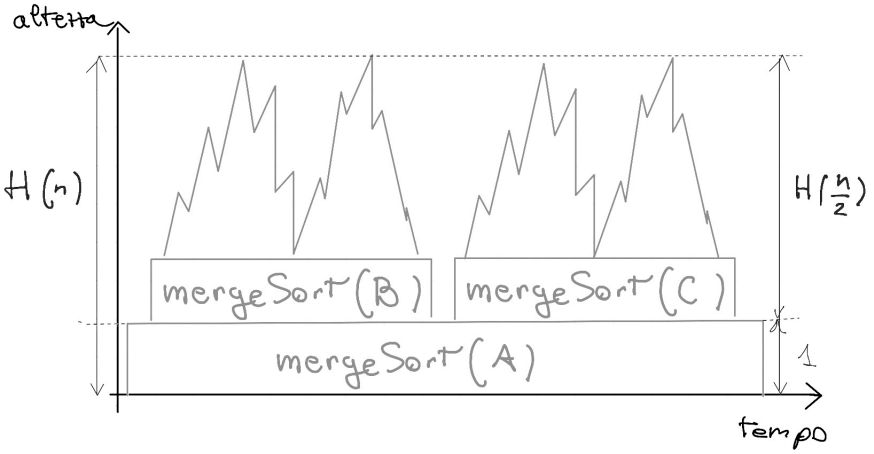
$\exists N$ potenza di 2 tale che $n \leq N \leq 2n$

$$\begin{aligned} C(n) \leq C(N) &= N\log_2 N - N + 1 < 2n \cdot \log_2 2n - 2n + 1 = \\ &\quad \downarrow \qquad\qquad\quad \downarrow \quad \downarrow \\ &\quad N \text{ potenza di } 2 \qquad n \leq N \leq 2n \quad 1 + \log_2 n \\ &= 2n(1 + \log_2 n) - 2n + 1 = 2n\log_2 n + 1 \end{aligned}$$

$$C(n) \leq 2n\log_2 n + 1 = \Theta(n \log n)$$

Complessità spaziale

Trattandosi di un algoritmo ricorsivo, viene utilizzato lo stack per ricorsione. Sia $H(n)$ l'altezza dello stack.



Inizialmente sullo stack troviamo un record di attivazione per `mergeSort(A)`. Dopo aver calcolato i due array, viene aggiunto sullo stack un record di attivazione per `mergeSort(B)`. Quando `mergeSort(B)` termina, il suo record di attivazione viene distrutto, e successivamente l'algoritmo richiama `mergeSort(C)`. Viene creato un nuovo record di attivazione che verrà caricato riutilizzando lo spazio precedentemente liberato.

L'altezza dello stack comprenderà quindi il record per A, e successivamente il maggiore tra quelli per B e per C:

$$H(n) = \begin{cases} 1 & \text{se } n = 1 \\ 1 + \max(H(\lfloor \frac{n}{2} \rfloor), H(\lceil \frac{n}{2} \rceil)) & \text{se } n > 1 \end{cases}$$

Per n pari $H(n) = \begin{cases} 1 & \text{se } n = 1 \\ 1 + H(\frac{n}{2}) & \text{se } n > 1 \end{cases}$

Per n potenza di 2 $H(n) = 1 + \log_2 n$

In generale (considerando la potenza di 2 più vicina) $H(n) = 2 + \log_2 n$

Altezza dello stack: $H(n) = \Theta(\log n)$

Problema

Implementando direttamente `mergeSort` come è stato scritto, ad ogni chiamata su un array A di lunghezza > 1:

- si creano due nuovi array B e $C \Rightarrow$ spreco di spazio
- si copia in essi il contenuto di $A \Rightarrow$ spreco di tempo

Si può utilizzare una soluzione alternativa che si appoggia ad un unico array ausiliario X per il merge

Implementazione per indici

```
1  PROCEDURA mergeSort(Array A, intero i, intero f, Array X)
2      IF f-i > 1 THEN
3      |   m <- (i+f)/2
4      |   mergeSort(A, i, m, X)
5      |   mergeSort(A, m, f, X)
6      |   merge(A, i, m, f, X)
7
8  ALGORITMO mergeSort(Array A[0..n-1])
9      Sia X un array di lunghezza n
10     mergeSort(A, 0, n, X)
```

Il vettore X è necessario per fare il merge. Non viene dichiarato all'interno del `merge`, e quindi localmente, ma è definito globalmente all'interno dell'algoritmo `mergeSort` perché altrimenti ogni volta che verrebbe chiamato il `merge` si consumerebbe nuova memoria.

- Si utilizza un unico array creato inizialmente per tutti i merge

Spazio
Array X: $\Theta(n)$

Stack ricorsione: $\left.\begin{array}{l} H(n) = \Theta(\log n) \\ \text{dim. record attivazione} = \mathrm{O}(1) \end{array}\right\} \Theta(\log n)$

Spazio utilizzato: $\Theta(n) + \Theta(\log n) = \Theta(n)$

Numero confronti: $\Theta(n \log n)$. L'analisi sul numero di confronto fatta precedentemente rimane valida anche in questa versione.

Tempo: se il costo dei confronti è $\mathcal{O}(1) \rightarrow \Theta(n \log n)$

Tecnica *divide-et-impera*

Sia $\mathcal{P}$ un problema e $\mathcal{I}$ un'istanza di $\mathcal{P}$

Se $\mathcal{I}$ è piccola risolvi $\mathcal{P}$ direttamente su $\mathcal{I}$, altrimenti:

- dividi $\mathcal{I}$ in istanze di lunghezza minore della lunghezza di $\mathcal{I}$
- risolvi queste istanze separatamente
- ricava la soluzione di $\mathcal{I}$ combinando opportunamente le soluzioni ottenute

```
ALGORITMO risolviProblema (Istanza I) -> soluzione
    IF |I| <= C THEN // CASO BASE
    |   //se la lunghezza dell'istanza è già "abbastanza corta" (minore di una costante)
    |   risolvi P in I direttamente
    |   RETURN soluzione
    ELSE
    |   dividi I  in m istanze I_1, ..., I_m con |I_j| < |I| per j = 1...m
    |   sol_1 <- risolviProblema(I_1) // RICORSIONE
    |   sol_2 <- risolviProblema(I_2)
    |       ...
    |   sol_m <- risolviProblema(I_m)
    |
    |   RETURN combina(Sol1,Sol2...Solm)
```

$$T(I) = \begin{cases} \text{se } |I| < c & \text{costante} \\ \text{altrimenti} & T_{\text{dividi}}(I) + T(I_1) + T(I_2) + \ldots + T(I_m) + T_{\text{combina}}(sol_1, \ldots, sol_m) \end{cases}$$

Calcolo del minimo e del massimo in un vettore

Input: vettore A di $n > 0$ elementi
Output: il valore minimo e il valore massimo in A

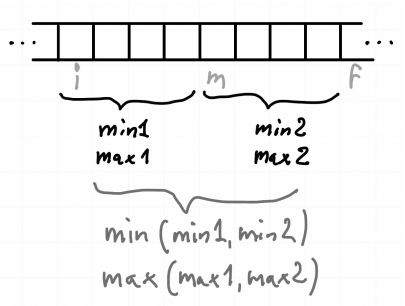
```
ALGORITMO minMax(Array A[0..n-1]) -> (elemento, elemento)
    min <- A[0]
    max <- A[0]
    FOR i <- 1 TO n-1 DO
    |   IF A[i] < min THEN min <- A[i]
    |   IF A[i] > max THEN max <- A[i]
    RETURN (min, max)
```

Si fanno $2n - 2$ confronti $= \Theta(n)$

Algoritmo ricorsivo

Utilizzando la tecnica *divide-et-impera*, è possibile suddividere il problema in due più semplici:

- per $n = 1, 2$ la risposta è immediata
- per $n > 2$ bisogna procedere ricorsivamente:



```
FUNZIONE minMax(Array A, indice i, indice f) -> (elemento, elemento)
    IF f-i = 1 THEN RETURN (A[i], A[i])
    ELSE IF f-1 = 2 THEN
    |   IF A[i] < A[i+1] THEN RETURN (A[i], A[i+1])
    |                    ELSE RETURN (A[i+1], A[i])
    ELSE
    |   m <- (i+f) / 2
    |   (min1, max1) <- minMax(A, i, m)
    |   (min2, max2) <- minMax(A, m, f)
    |
    |   IF min1 < min2 THEN min <- min1
    |                  ELSE min < min2
    |   IF max1 > max2 THEN max <- max1
    |                  ELSE max <- max2
    |   RETURN (min, max)

ALGORITMO minMax(Array A[0..n-1]) -> (elemento, elemento)
    RETURN minMax(A, o, n)
```

Sia $C(n)$ il numero di confronti per un array di lunghezza n

$$C(n) = \begin{cases} 0 & \text{se } n = 1 \\ 1 & \text{se } n = 2 \\ C(\lfloor \frac{n}{2} \rfloor) + C(\lceil \frac{n}{2} \rceil) + 2 & \text{altrimenti} \end{cases}$$

Per n potenza di 2:

$$\begin{aligned} C(n) &= 2C\left(\frac{n}{2}\right) + 2 = 2\left[2C(\frac{n}{2^2}) + 2\right] + 2 = 2^2 C\left(\frac{n}{2^2}\right) + 2^2 + 2 = 2^2\left[2C\left(\frac{n}{n^3}\right) + 2\right] + 2^2 + 2 = \\ &= 2^3 C\left(\frac{n}{2^3}\right) + 2^3 + 2^2 + 2 = \cdots = 2^k C\left(\frac{n}{2^k}\right) + \sum_{i=1}^{k} 2^i = 2^k C\left(\frac{n}{2^k}\right) + 2^{k+1} - 2 = \\ &= \frac{n}{2} C(2) + n - 2 = \frac{n}{2} + n - 2 = \frac{3}{2}n - 2 \end{aligned}$$

$\downarrow$

$\frac{n}{2^k} = 2 \text{ per } n = 2^{k+1}, \text{ cioè } k = \log_2 n - 1; \quad nC(2) = 1$

$C(n) = \frac{3}{2}n - 2$ confronti

Questo algoritmo risparmia quindi circa il 25% di confronti rispetto a quello precedente, con il contro di avere uno stack dovuto alla ricorsione.

Per n non potenza di 2:
Si scrive n come $n = n' + n''$ con n' la più grande potenza di $2 < n$. Dopodiché si divide l'array in una prima parte di lunghezza n' e in una seconda parte di lunghezza n''. Dividendo in questo modo si riesce a dimostrare per induzione che $C(n) = \lceil \frac{3}{2}n \rceil - 2$

Matrici

Somma di matrici n×n

Input: $A = [a_{ij}], B = [b_{ij}]$, matrici $n \times n$ di interi
Output: $C = [c_{ij}] = A + B$ somma delle matrici A e B

Definizione di somma di matrici: $c_{ij} = a_{ij} + b_{ij}$

In totale si fanno n^2 somme

Prodotto di matrici n × n

Input: $A = [a_{ij}], B = [b_{ij}]$, matrici $n \times n$ di interi
Output: $C = [c_{ij}] = A \cdot B$ prodotto delle matrici A e B

Definizione di prodotto di matrici: $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$

$$\left.\begin{array}{l} n \text{ prodotti} \\ n-1 \text{ somme} \end{array}\right\} \; 2n-1 \text{ operazioni per ciascun elemento } c_{ij}$$

Per tutta la matrice il numero totale delle operazioni è $n^2(2n-1) = 2n^3 - n^2 = \Theta(n^3)$

- upper bound $\Theta(n^3)$
- lower bound $\Omega(n^2)$ *(non si riesce ad arrivare ad n^2)*

Divide-et-impera immediato

$$\text{Se } A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad \text{e} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\text{con } A_{ij}, B_{ij} \text{ matrici } \frac{n}{2} \times \frac{n}{2}, \text{ allora}$$

$$C = A \cdot B = \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{bmatrix}$$

Utilizzando la tecnica *divide-et-impera*, si può pensare ad un algoritmo che per matrici 1×1 fa il prodotto degli unici elementi presenti, mentre per $n > 1$ si suddividono le matrici in 4 parti e si applica la formula sopra:

```
ALGORITMO moltiplica(Matrici n×n A, B) -> Matrice
    IF n = 1 THEN RETURN [a11 b11]
    ELSE
    |   decomponi A e B in matrici n/2×n/2
    |   calcola le seguenti matrici n/2×n/2
    |   |   c11 = a11 b11 + a12 b21
    |   |   c12 = a11 b12 + a12 b22
    |   |   c21 = a21 b11 + a22 b21
    |   |   c22 = a21 b12 + a22 b22
    |   RETURN C
```

Se $n = 1$ si fa una sola operazione: $c = [a_{11}b_{11}]$

Se $n > 1$ si dividono le matrici A e B e si calcolano ricorsivamente gli elementi della matrice C:

- 8 prodotti tra matrici $\frac{n}{2} \times \frac{n}{2} \to 8T\left(\frac{n}{2}\right)$
- 4 somme tra matrici $\frac{n}{2} \times \frac{n}{2} \to 4\left(\frac{n}{2}\right)^2 = n^2$

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 8T(\frac{n}{2}) + n^2 & \text{altrimenti} \end{cases} \longrightarrow \text{equazione di ricorrenza}$$

Risolvendo l'equazione di ricorrenza si ottiene $T(n) = \Theta(n^3)$, e quindi presenta lo stesso costo dell'algoritmo che moltiplica direttamente le righe per le colonne: non c'è quindi nessun guadagno effettivo.

Algoritmo di Strassen

Date $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ e $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$

Considero le matrici:

- $M_1 = (A_{21} + A_{22} - A_{11}) \cdot (B_{22} - B_{12} + B_{11})$
- $M_2 = A_{11} \cdot B_{11}$
- $M_3 = A_{12} \cdot B_{21}$
- $M_4 = (A_{11} - A_{21}) \cdot (B_{22} - B_{11})$
- $M_5 = (A_{21} + A_{22}) \cdot (B_{12} - B_{11})$
- $M_6 = (A_{12} - A_{21} + A_{11} - A_{22}) \cdot B_{22}$
- $M_7 = A_{22} \cdot (B_{11} + B_{22} - B_{12} - B_{21})$

Si può verificare che:

$$C = A \cdot B = \begin{bmatrix} M_2 + M_3 & M_1 + M_2 + M_5 + M_6 \\ M_1 + M_2 + M_4 - M_7 & M_1 + M_2 + M_4 + M_5 \end{bmatrix}$$

Utilizzando sempre la tecnica *divide-et-impera*, si può pensare ad un algoritmo che per matrici 1×1 fa il prodotto degli unici elementi presenti, mentre per $n > 1$ si suddividono le matrici in 4 parti e si applica la formula sopra:

```
ALGORITMO moltiplica(Matrici n×n A, B) -> Matrice
    IF n = 1 THEN RETURN [a11 b11]
    ELSE
    |   decomponi A e B in matrici n/2×n/2
    |   calcola le seguenti matrici n/2×n/2
    |   |   M1 = (a21 + a22 - a11)(b22 - b12 + b11)
    |   |   M2 = a11 b11
    |   |   M3 = a12 b21
    |   |   M4 = (a11 - a21)(b22 - b11)
    |   |   M5 = (a21 + a22)(b12 - b11)
    |   |   M6 = (a12 - a21 + a11 - a22) b22
    |   |   M7 = a22 (b11 + b22 - b12 - b21)
    |   e
    |   |   c11 = M2 + M3
    |   |   c12 = M1 + M2 + M5 + M6
    |   |   c21 = M1 + M2 + M4 - M7
    |   |   c22 = M1 + M2 + M4 + M5
    |   RETURN C
```

Se $n = 1$ si fa una sola operazione: $c = [a_{11} b_{11}]$

Se $n > 1$ si dividono le matrici A e B e si calcolano ricorsivamente gli elementi della matrice C:

- 7 prodotti tra matrici $\frac{n}{2} \times \frac{n}{2} \to 7T\left(\frac{n}{2}\right)$
- 24 somme tra matrici $\frac{n}{2} \times \frac{n}{2} \to 24\left(\frac{n}{2}\right)^2 = 6n^2$

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ 7T(\frac{n}{2}) + 6n^2 & \text{altrimenti} \end{cases} \quad \longrightarrow \text{equazione di ricorrenza}$$

$$T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$$

Quick sort

Nel caso semplice la soluzione è immediata, altrimenti si divide l'array in 2 parti, si ordinano le due parti ricorsivamente e si combinano le soluzioni.

Si procede quindi tramite la tecnica *dividi-et-impera*. La divisione in questo caso è complessa mentre la combinazione è semplice:

- **Dividi**: si sceglie un elemento chiamato pivot e si divide l'array in due sotto-array: una con elementi minori del pivot e una con elementi maggiori. Ricorsivamente, si ripete il processo sui sotto-array fino a quando non rimangono singoli elementi.
- **Impera**: gli elementi ottenuti dalla parte dividi vengono poi combinati per ottenere la lista ordinata.

```
ALGORITMO quickSort(Array A)
    IF lunghezza di A > 1 THEN
    |   //divide -> algoritmo di partizionamento
    |   scegli un elemento x di A //pivot
    |   B <- {y ∈ A | y < x}
    |   C <- {y ∈ A | y > x}
    |
    |   quickSort(B)
    |   quickSort(C)
    |   A <- concatenazione di B, x, C //impera
```

L'algoritmo di partizionamento è la parte più delicata e complessa del quick sort, e in linea di massima funziona nel seguente modo:

- Partiziona $A[i \cdots f-1]$ rispetto a un elemento di $A[i \ldots f-1]$ scelto come pivot, spostando gli elementi in modo che prima del pivot ci siano gli elementi minori o uguali, e dopo quelli maggiori
- Restituisce la posizione finale del pivot

Ci sono varie possibili implementazioni. Di seguito se ne propone una:

```
ALGORITMO partiziona(Array A, indice i, indice f) -> indice
    perno <- A[i]
    dx <- f
    sx <- i
    WHILE sx < dx DO
    |   DO dx <- dx - 1 WHILE A[dx] > perno
    |   DO sx <- sx + 1 WHILE A[sx] <= perno AND sx < dx
    |   IF sx < dx THEN
    |   |   scambia A[sx] con A[dx]
    scambia A[i] con A[dx]
    RETURN dx
```

Questo algoritmo considera come pivot il primo elemento della porzione. Dopodiché ripete in loop le seguenti azioni:

1. scansione da dx fino a un elemento $\leq$ del pivot
2. scansione da sx fino a un elemento $>$ del pivot
3. scambia i due elementi trovati

Questo ciclo termina quando le due scansioni si incontrano. Infine si scambia l'elemento all'indice dx con il pivot e si restituisce proprio dx.

```
ALGORITMO quickSort(Array A[0..n-1])
    quickSort(A, 0, n)

PROCEDURA quickSort(Array A, indice i, indice f)
    IF f-i > 1 THEN
    |   m <- partiziona(A, i, f)
    |   quickSort(A, i, m)
    |   quickSort(A, m+1, f)
```

Dato che con l'algoritmo di partizionamento il pivot è già nella sua posizione ordinata, l'algoritmo di `quickSort` può limitarsi a riordinare ricorsivamente le parti di destra e di sinistra: la ricombinazione non è quindi necessaria in quanto viene implicitamente già effettuata.

Caso base di Quick Sort e Merge Sort

Entrambi gli algoritmi verificano che le sottosequenze abbiano più di un elemento prima di procedere con la suddivisone e le chiamate ricorsive. Ciò è implementato in entrambi i casi con uno stesso controllo: `IF f-1 > 1`

Se questo non si verifica si ricade nel caso base in cui non c'è bisogno di continuare con la divisione in quanto le sottosequenze sono già ordinate per definizione. I casi base dei due algoritmi sono però leggermente diversi.

Merge sort: caso base $n = 1$

- Il merge sort divide l'array in due metà quasi uguali: se la lunghezza dell'array è pari, le due metà avranno la stessa dimensione, se dispari invece una metà avra un elemento in più.
  Questo procedimento garantisce che entrambe le sottosequenze risultanti abbiano almeno un elemento, evitando così sottosequenze di dimensione zero.

Quick sort: caso base $n = 0, 1$

- Nel quick sort le partizioni dipendono dal pivot scelto e dai valori degli elementi. In scenari peggiori l'array potrebbe essere partizionato in due sottoarray rispettivamente di $n - 1$ e 0 elementi.
  Nella pratica la condizone `f-i > 1` impedisce chiamate ricorsive su sottosequenze di dimensione 0, ma concettualmente bisogna ricordarsi che l'algoritmo di partizionamento può generare tali sottosequenze, che quindi è importante considerare sull'analisi delle prestazioni di questo algoritmo di ordinamento.

TL;DR: a differenza del merge sort, in cui era importante considerare solo il caso base $n = 1$, qui bisogna considerare anche $n = 0$ per i casi in cui l'array viene partizionato in due sottoarray rispettivamente da $n - 1$ e 0 elementi.

Analisi

Numero di confronti

Sia $C(n)$ il numero di confronti per array di lunghezza n. $C(n)$ varia in funzione del perno. Dato un perno in posizione finale k dopo il partizionamento, l'equazione di ricorrenza è:

$$C(n) = \begin{cases} 0 & \text{se } n \leq 1 \\ \underset{\uparrow \text{partiziona}}{C_{part}(n)} + \underset{\uparrow 1^a \text{ chiamata ric.}}{C(k)} + \underset{\uparrow 2^a \text{ chiamata ric.}}{C(n-k-1)} & \text{altrimenti} \end{cases}$$

Si osserva che nell'algoritmo di partizione ogni elemento dve essere confrontato con il perno, eccetto il perno stesso. Il perno viene confrontato con se stesso solo nel caso in cui la porzione sia già ordinata. In alcuni casi uno stesso elemento può essere confrontato sia dalla scansione da destra che da quella sinistra. Di conseguenza $C_{part}(n)$ può valere sia n che $n - 1$.

Caso peggiore

$$C_w(n) = \begin{cases} 0 & \text{se } n \leq 1 \\ n + \underset{\uparrow}{\max}\{\, C_w(k) + C_w(n-k-1) \mid k = 0\,..\,n{-}1 \,\} & \text{altrimenti} \end{cases}$$

$\uparrow$ max per $k = 0 \vee k = n-1$

Il numero massimo di confronti si ottiene quando la partizione è completamente sbilanciata, ovvero se il perno è l'elemento minore o l'elemento massimo, ovvero se $k = 0$ oppure $k = n-1$. Sostituendo uno dei due casi:

$$\begin{aligned} C_w(n) &= n + \mathbf{C_w(n-1)} = n + \mathbf{n-1} + \mathbf{C_w(n-2)} = \\ &= n + n - 1 + \mathbf{n-2} + \mathbf{C_w(n-3)} = \cdots = n + n - 1 + n - 2 + \cdots + 2 + C_w(1) = \\ &= \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1 = \Theta(n^2) \end{aligned}$$

$\uparrow$ $C_w(1) = 0$

Caso migliore

$$C_b(n) = \begin{cases} 0 & \text{se } n \leq 1 \\ n + \underset{\uparrow}{\min}\{\, C_b(k) + C_b(n-k-1) \mid k = 0\,..\,n{-}1 \,\} & \text{altrimenti} \end{cases}$$

$\uparrow$ min per una partizione bilanciata

Il numero minimo di confronti si ottiene quando il perno è in mezzo e di conseguenza la partizione è bilanciata. Si può quindi approssimare l'equazione con:

$$C_b(n) = n + 2C_b\left(\frac{n}{2}\right) \approx n \log_2 n = \Theta(n \log n)$$

Caso medio

$$C(n) = \begin{cases} 0 & \text{se n} \leq 1 \\ \dfrac{\displaystyle\sum_{k=0}^{n-1} [\, n + C(k) + C(n-k-1) \,]}{2} & \text{altrimenti} \end{cases}$$

Per calcolare il caso medio si fa la media considerando tutti i valori di k. Se $n > 1$ si ottiene:

$$C(n) = \frac{1}{n}\sum_{k=0}^{n-1} n + \frac{1}{n}\sum_{k=0}^{n-1} C(k) + \frac{1}{n}\sum_{k=0}^{n-1} C(n-k-1) = n + \frac{2}{n}\sum_{i=0}^{n-1} C(i) = n + \frac{2}{n}\sum_{i=1}^{n-1} C(i)$$

$\uparrow$ Caso base: $C(0) = C(1) = 0$

$$C(n) = \begin{cases} 0 & \text{se n} \leq 1 \\ n + \dfrac{2}{n}\displaystyle\sum_{i=1}^{n-1} C(i) & \text{altrimenti} \end{cases}$$

Si dimostra ora per induzione che $C(n) \leq 2n \ln n$ per $n \geq 1$:

- **Base**: $n = 1$
  $C(1) = 0; \quad 2 \ln 1 = 0$
- **Induzione:** si assume che la formula valga per $n - 1$

$$C(n) = n + \frac{2}{n}\sum_{i=1}^{n-1} C(i) \underset{\uparrow}{\leq} n + \frac{2}{n}\sum_{i=2}^{n-1} 2i \ln i = n + \frac{4}{n}\sum_{i=2}^{n-1} i \ln i \underset{\uparrow}{\leq} n + \frac{4}{n}\frac{n^2}{2}\ln n - \frac{n^2}{4} =$$

ipotesi induttiva — vedere il problema a fine pdf

$$= \cancel{n} + 2n \ln n - \cancel{n} = 2n \ln n$$

$C(n) \leq 2n \ln n \approx 1,39 n \log_2 n = \mathcal{O}(n \log n)$

Poiché il caso medio è molto vicino a quello migliore, capita molto spesso. Per ottenere il caso migliore si possono eseguire degli scambi casuali per mettere in disordine l'array (spesso accade che questi siano parzialmente ordinati).

Questo algoritmo **non è stabile** (esiste una versione stabile che però incrementa la costante nel caso medio)

Altezza della pila

Il record di attivazione del quick sort utilizza spazio costante (3 variabili), e di conseguenza l'uso di memoria aggiuntiva è proporzionale al numero di record di attivazione che posso trovare contemporaneamente sulla pila (ossia alla sua altezza massima).

Il caso migliore si ottiene per partizioni bilanciate, ovvero quando l'algoritmo viene chiamato lo stesso numero di volte per entrambe le parti. Questo caso è già stato studiato nel merge sort, e l'altezza è circa $\log_2 n$.

Il caso peggiore si ottiene per partizioni sbilanciate, ovvero quando il pivot è sempre il maggiore o il minore. Nel primo caso la prima chiamata ricorsiva cresce a sinistra fino ad n, mentre quella a destra rimane bassa, mentre nel secondo caso avviene il contrario.

Ottimizzazione con singola chiamata ricorsiva

```
1  ALGORITMO quickSort(Array A[0..n-1])
2      quickSort(A, 0, n)
3
4  PROCEDURA quickSort(Array A, indice i, indice f)
5      WHILE f-i > 1 DO
6      |   m <- partiziona(A, i, f)
7      |   IF m-i < f-m THEN
8      |   |   quickSort(A, i, m)
9      |   |   i <- m + 1
10     |   ELSE
11     |   |   quickSort(a, m+1, f)
12     |   |   f <- m
```

Si può ottimizzare il quick sort eliminando l'ultima chiamata ricorsiva, che è in coda, e facendo la chiamata ricorsiva sempre sulla parte più corta, lasciando poi alla prossima iterazione l'ordinamento della parte più grande. Così facendo, una chiamata ordina un array lungo al più la metà rispetto all'array ordinato dal chiamante, e quindi l'altezza della pila risulta essere $\log n$.

Spazio

Versione base, caso peggiore: $\left.\begin{array}{l}\text{altezza pila } \Theta(n)\\ \text{record di attivazione } \mathcal{O}(1)\end{array}\right\}$ spazio $\Theta(n)$

Versione migliorata: $\left.\begin{array}{l}\text{altezza pila } \Theta(\log n)\\ \text{record di attivazione } \mathcal{O}(1)\end{array}\right\}$ spazio $\Theta(\log n)$

Teorema fondamentale delle ricorrenze

Siano $m, a, b', b'', c \in \mathbb{R}^+$ con $a > 1$. L'equazione

$$F(n) = \begin{cases} 1 & \text{se } n = 1 \\ mF(\frac{n}{a}) + b''n^c & \text{se } n > 1 \end{cases}$$

soddisfa le seguenti relazioni:

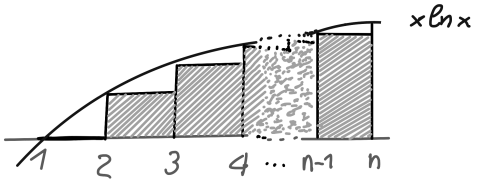
$$F(n) = \begin{cases} \Theta(n^c) & \text{se } m < a^c \\ \Theta(n^c \log n) & \text{se } m = a^c \\ \Theta(n^{\log_a m}) & \text{se } m > a^c \end{cases}$$

Il risultato può essere esteso anche al caso in cui per $n > 1$ l'equazione data sia della forma

$$F(n) = m_1 F\left(\left\lfloor \frac{n}{a} \right\rfloor\right) + m_2 F\left(\left\lceil \frac{n}{a} \right\rceil\right) + b''n^c \quad \text{con } m = m_1 + m_2$$

Problema

Trovare una limitazione superiore per $\sum_{i=2}^{n-1} i \ln i$



$$\sum_{i=2}^{n-1} i \ln i \leq \int_2^n x \ln x \, dx$$

Si utilizzi l'integrazione per parti considerando $g'(x) = x;\ f(x) = \ln x$

$$\int x \ln x \, dx = \frac{x^2}{2} \ln x - \int \frac{1}{x} \frac{x^2}{2} \, dx = \frac{x^2}{2} \ln x - \frac{x^2}{4} + c$$

$$\left[\frac{x^2}{2} \ln x - \frac{x^2}{4} \right]_2^n = \frac{n^2}{2} \ln n - \frac{n^2}{4} - 2 \ln 2 + 1 \leq \frac{n^2}{2} \ln n - \frac{n^2}{4}$$

$$\sum_{i=2}^{n-1} i \ln i \leq \frac{n^2}{2} \ln n - \frac{n^2}{4}$$

Si ricordi l'integrazione per parti: $\int f(x) g'(x) \, dx = f(x) \cdot g(x) - \int f'(x) g(x) \, dx$