Complessità computazionale

Complessità computazionale

- Costo degli algoritmi in termini di risorse di calcolo
  - Tempo, spazio di memoria

- Esempio: dato un vettore v di n interi ordinati in maniera non decrescente verificare se un intero k è presente o meno in v

Ricerca sequenziale

```
int ricerca(int v[], int size, int k)
{ int i;
    for (i=0; i<size; i++)
      if (v[i] == k) return i;
    return -1;
}
```

… nessun vantaggio dal fatto che v è ordinato …

Ricerca lineare in un array ordinato

```
int  ricercaord(int a[], int n, int elem)
{
  int i = 0;
  int trovato = 0;

  while(i < n && !trovato)  // visita finalizzata
              if(a[i] >= elem)
                        trovato = 1;
          else i++;

   if(i == n)                    // raggiunto fine array senza trovare
    elem
              i = -1;
   else if(a[i] > elem)        // elem non può essere più trovato
                 i = -1;

   return i;
}
```

… vantaggio limitato, nel caso peggiore non cambia niente …

Ricerca binaria

```
int  ricercabin(int a[], int n, int elem)
{
   int h =0,     // estremo inferioredell'intervallo in cui ricercare
   int k = n-1; // estremo superiore dell'intervallo in cui ricercare
   int p;          // posizione dell'elemento se trovato
   int trovato = 0;     // inizialmente elemento non trovato

   while(h <= k && ! trovato) {
       p = (h + k) / 2;   // posizione centrale
       if (a[p] == elem)
              trovato = 1;    // permette di uscire dal ciclo
       else if(a[p] > elem)
               k = p-1;          // la ricerca continua nella prima metà
       else h = p+1; }   // la ricerca continua nella seconda metà

    return (trovato? p : -1);
}
```

… vantaggio sostanziale, anche nel caso peggiore…

Fattori per la valutazione del tempo

- La macchina usata
- La configurazione dei dati
- La dimensione dei dati
- Caso peggiore di configurazione dei dati
- Funzione della dimensione dell’input
- Comportamento asintotico

Macchina astratta

- Costo unitario delle istruzioni e delle condizioni atomiche
- Le strutture di controllo hanno un costo pari alla somma dei costi dell'esecuzione delle istruzioni interne, più la somma dei costi delle condizioni
- Le chiamate a funzioni hanno un costo pari al costo di tutte le sue istruzioni e condizioni; il passaggio dei parametri ha costo nullo
- Istruzioni e condizioni con chiamate a funzioni hanno costo pari alla somma del costo delle funzioni invocate più uno

Esempio

- Calcolare il costo per l’esempio precedente nel caso v[n] = {1, 3, 9, 17, 34, 95,96, 101} e k=9

Inizializzazioni (i=0)	1 +	
Confronti (i<size)		3 +
Confronti (v[i]==k)	3 +	
Instruzioni (i++)		2 +
Istruzioni (return i)		1 =
		10

Cosa cambia se k=10 ?

Caso peggiore

- Caso che a parità di dimensione produce il costo massimo
  - Se accettabile nel caso peggiore …
- Nel caso dell'esempio, k non presente

Inizializzazioni (i=0)	1 +	
Confronti (i<size)		n+1 +
Confronti (v[i]==k)		n +
Istruzioni (i++)		n +
Istruzioni (return -1)	1 =	

3n+3

Caso medio

- Equiprobabilità dell'input
  - La probabilità che k sia in posizione i (1<=i<=n) vale 1/n
  - Costo del caso in posizione i: 3i+1
  - Costo caso medio

$$\frac{1}{n}\sum_{i=1}^{n}(3i+1) = \frac{1}{n}\left(3\frac{n^2+n}{2}+n\right) =$$

$$= \frac{3n+5}{2}$$

Costo come funzione della dimensione dell’input

- Che cosa è la dimensione ?
  - Vettore … numero di elementi
  - Albero ? … numero dei nodi
  - Grafo ? … numero archi più numero nodi

- Esempio: calcolo del fattoriale, con tipo intero non limitato

Esempio

```
int fattoriale(int n)
{ int i = 1;
  int fatt = 1;
  while (i <=n) {
      fatt = fatt * i;
      i++; }
  return fatt;
}
```

Inizializzazioni (i=1) 1 +	
Inizializzazioni (fatt=1)	1 +
Confronti (i<=N)	n+1
+	
Istruzioni (fatt=fatt*i)	n +
Istruzioni (i++)	n +
Istruzioni (return fatt)	1 =

3n+4

Dimensione dell’input

- n, 3n+4
- Numero d di cifre decimali necessarie per rappresentare n in decimale … d $\log_{10}$ n
  … 3 10^d + 4 … esponenziale

  – Esempio: somma inefficiente

```
int somma (int n, int m) {
        while (m>0) { n=n+1; m=m-1;}
return n; }
```

… lineare nel valore di m, esponenziale nella sua dimensione …

Comportamento asintotico

- Nell’analizzare la complessità tempo di un algoritmo siamo interessati a come aumenta il tempo al crescere della taglia *n* dell’input.
- Siccome per valori “piccoli” di *n* il tempo richiesto è comunque poco, ci interessa soprattutto il comportamento per valori “grandi” di n (<u>il comportamento asintotico</u>)

Comportamento asintotico

- Comportamento al crescere della dimensione n dei dati all'infinito
  - Trascurare tutte le costanti moltiplicative ed additive e tutti i termini di ordine inferiore
  - Suddivisione di algoritmi in classi di complessità

$a\,n+b$	lineare
$a\,n^2 + bn + c$	quadratica
$a \log_g n + h$	logaritmica
a^n	esponenziale
n^n	esponenziale

Comportamento asintotico

- Abbiamo una scala di complessità:

$$\vdots$$

$$2^n$$

$$\vdots$$

$$n^3$$

$$n^2$$

$$n \log n$$

$$n$$

$$\log n$$

$$1$$

vogliamo inserire ogni algoritmo in questa scala

Comportamento asintotico

- Supponiamo di avere, per uno stesso problema, sette algoritmi diversi con diversa complessità. Supponiamo che un passo base venga eseguito in un microsecondo (10^{-6} sec).
- Tempi di esecuzione (in secondi) dei sette algoritmi per diversi valori di *n*.

	n=10	n=100	n=1000	$n=10^6$
$\sqrt{n}$	$3*10^{-6}$	10^{-5}	$3*10^{-5}$	10^{-3}
$n+5$	$15*10^{-6}$	10^{-4}	10^{-3}	1 sec
$2*n$	$2*10^{-5}$	$2*10^{-4}$	$2*10^{-3}$	2 sec
n^2	10^{-4}	10^{-2}	1 sec	10^6 (~12gg)
n^2+n	10^{-4}	10^{-2}	1 sec	10^6 (~12gg)
n^3	10^{-3}	1 sec	10^5 (~1g)	10^{12} (~300 secoli)
2^n	10^{-3}	~$4*10^{14}$ secoli	~$3*10^{287}$ secoli	~$3*10^{301016}$ secoli

Comportamento asintotico

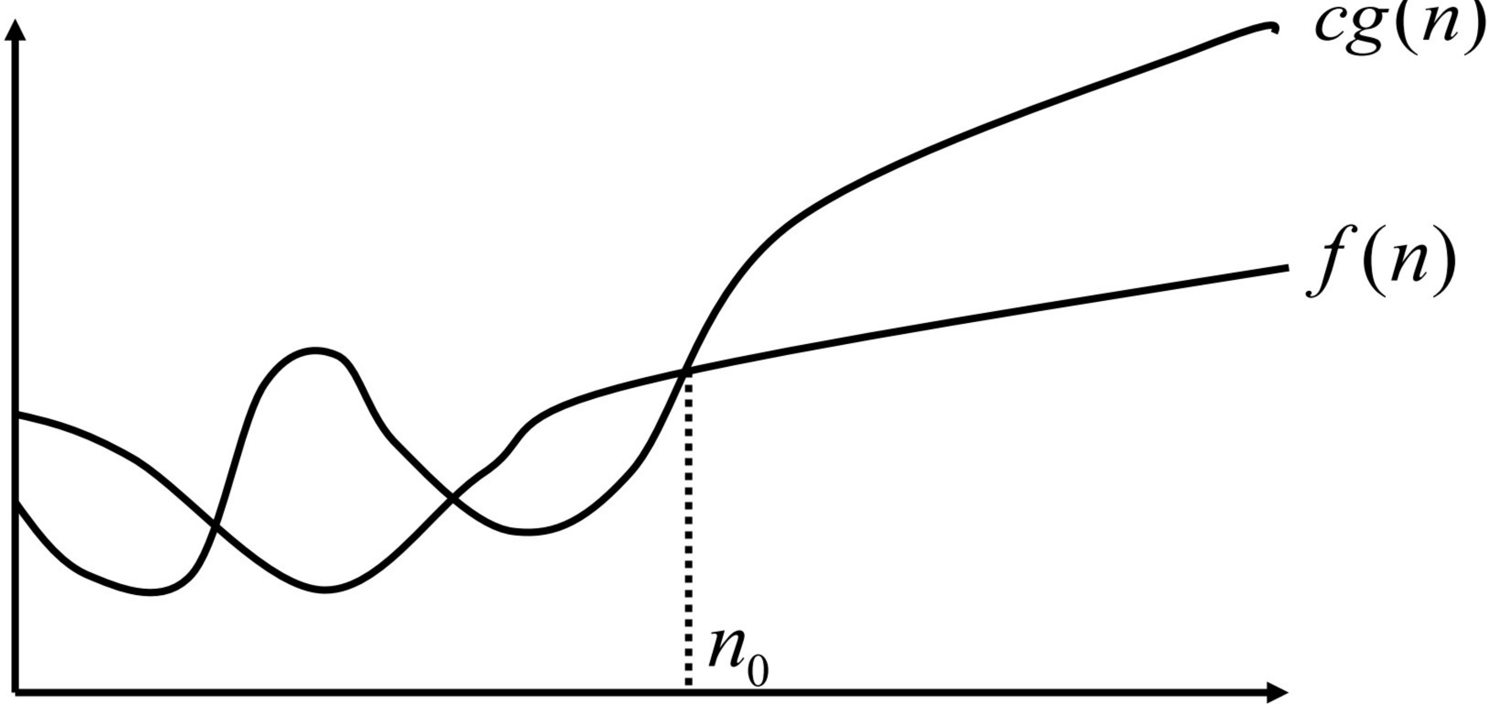
- Per piccole dimensioni dell'input, osserviamo che tutti gli algoritmi hanno tempi di risposta non significativamente differenti.
- L ' algoritmo di complessità esponenziale ha tempi di risposta ben diversi da quelli degli altri algoritmi (migliaia di miliardi di secoli contro secondi, ecc.)
- Per grandi dimensioni dell'input (n=10^6), i sette algoritmi si partizionano nettamente in cinque classi in base ai tempi di risposta:
- Algoritmo rad(n) frazioni di secondo
- Algoritmo n+5, 2*n *secondi*
- Algoritmo n^2, n^2+n *giorni*
- Algoritmo n^3 secoli
- Algoritmo 2^n miliardi di secoli

Notazione O ed □

- f e g funzioni dai naturali ai reali positivi
- f(n) è O di g(n), f(n)□O(g(n)), se esistono due costanti positive c ed n_o tali che se n >= n_o, f(n) <= c g(n)
  - Applicata alla funzione di complessità f(n),la notazione O ne limita superiormente la crescita e fornisce quindi una indicazione della bontà dell’algoritmo
- f(n) è Omega di g(n), f(n) □ □(g(n)), se esistono due costanti positive c ed n_o tali che se n >= n_o, c g(n) <= f(n)
  - La notazione □ limita inferiormente la complessità, indicando così che il comportamento dell’algoritmo non è migliore di un comportamento assegnato

Notazione asintotica O
(limite superiore asintotico)

$$O(g(n)) = \{ f(n) : \text{esistono } c > 0 \text{ ed } n_0 \text{ tali che } 0 \le f(n) \le cg(n) \text{ per ogni } n \ge n_0 \}$$



Esempi

$$\boxed{f(n) = 2n^2 + 5n + 5 = O(n^2)}$$

infatti $0 \le 2n^2 + 5n + 5 \le cn^2$

per $c = 4$ ed $n_0 = 5$

Vedremo che in generale per $a_2 > 0$

$$\boxed{f(n) = a_2 n^2 + a_1 n + a_0 = O(n^2)}$$

$$\boxed{f(n) = 2 + \sin n = O(1)}$$

infatti $0 \le 2 + \sin n \le c \cdot 1$

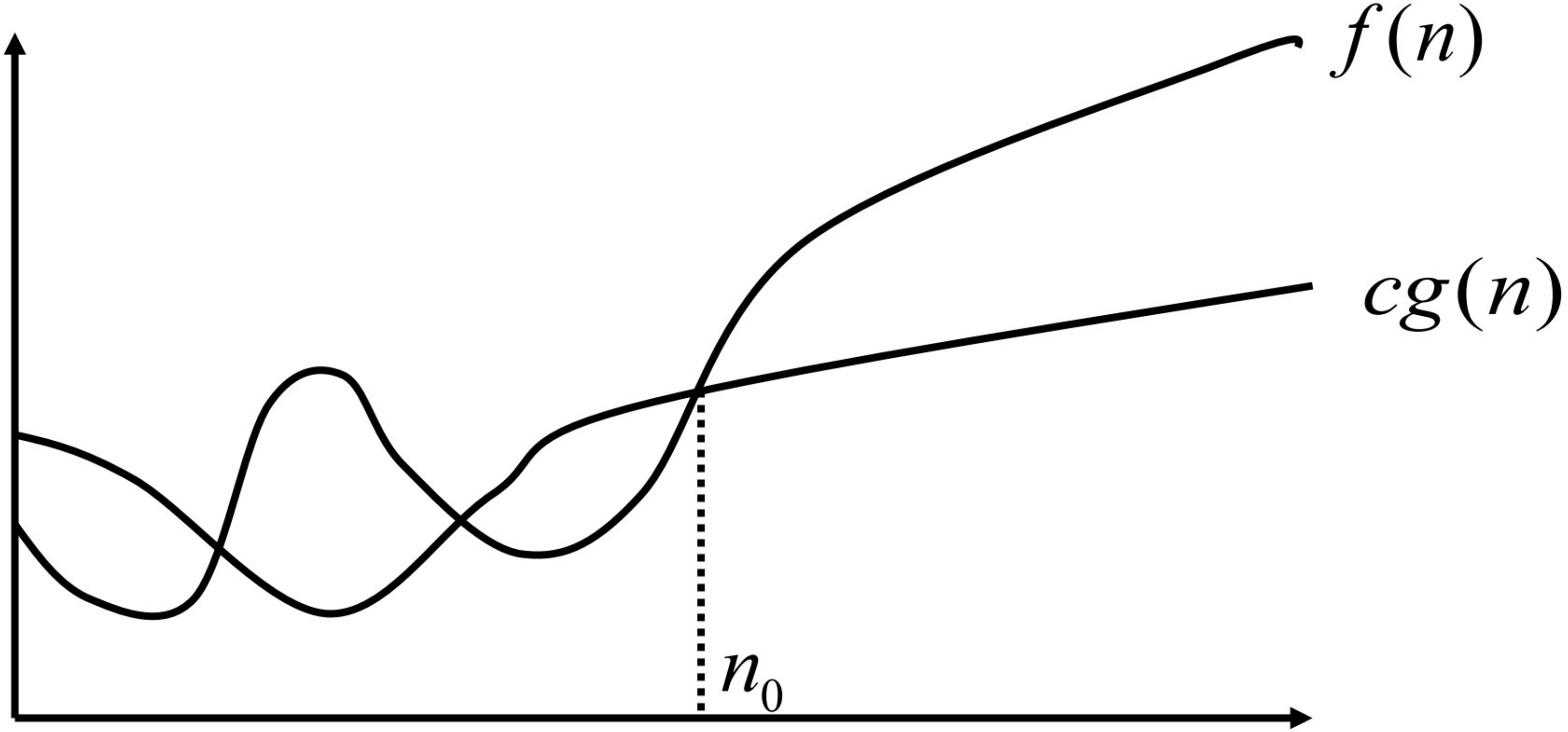
per $c = 3$ ed $n_0 = 1$

Limiti superiori O(g(n))

- Scoprire un algoritmo per la risoluzione di un problema equivale a stabilire un limite superiore di complessità
  - Il problema è risolubile entro i limiti di tempo stabiliti dalla funzione di complessità
- Suddivisione di algoritmi in classi di complessità
  - Algoritmi diversi per la risoluzione dello stesso problema possono avere diversa complessità e saranno confrontati sulla base della complessità asintotica nel caso medio o pessimo

Notazione asintotica Ω (limite inferiore asintotico)

$$\Omega(g(n)) = \{ f(n) : \text{esistono } c > 0 \text{ ed } n_0 \text{ tali che } f(n) \geq cg(n) \geq 0 \text{ per ogni } n \geq n_0 \}$$

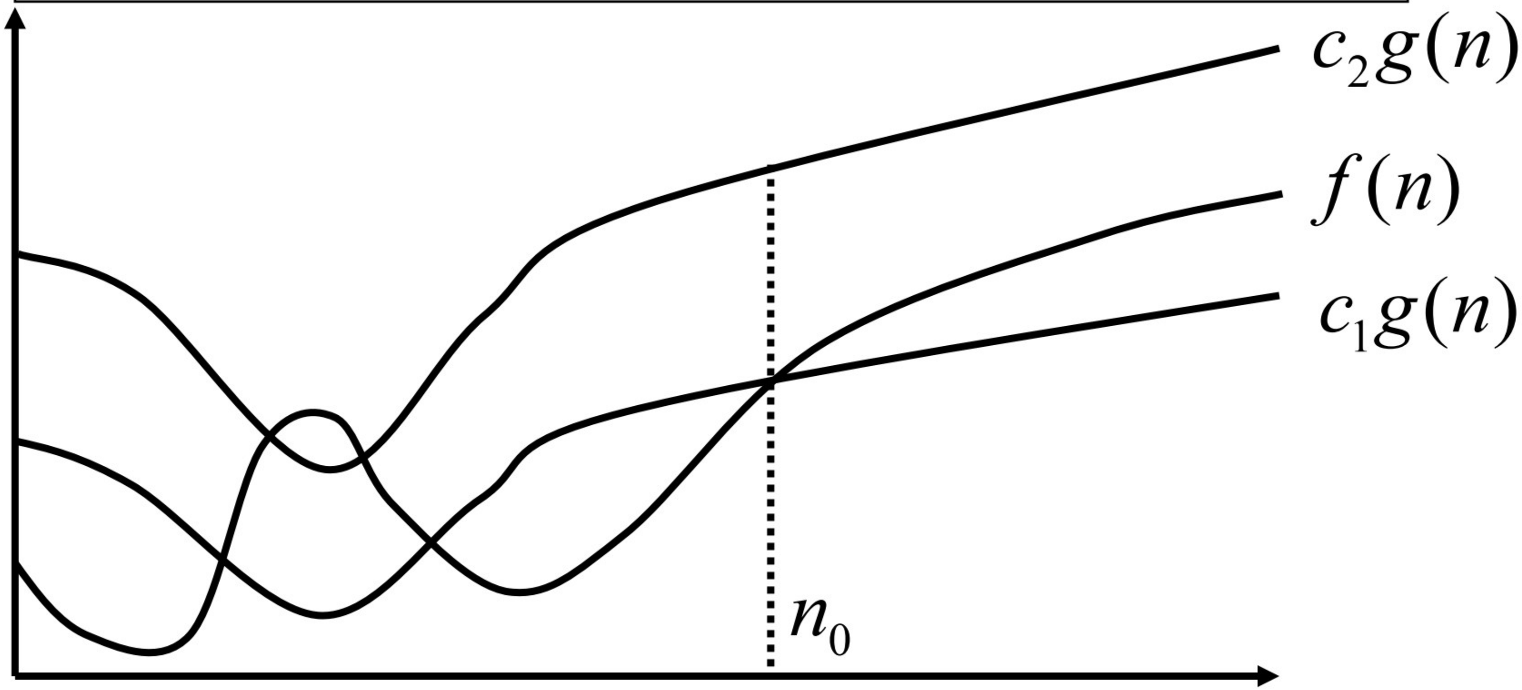


Limiti inferiori $\Omega(g(n))$

- La ricerca di limiti inferiori di complessità risponde alla domanda se non si possano determinare algoritmi più efficienti di quelli noti
- Quando la complessità di un algoritmo è pari al limite inferiore di complessità determinato per un problema, l’algoritmo si dice ottimo (in ordine di grandezza)
- Non esiste una teoria generale all’individuazione di limiti inferiori alla complessità dei problemi

Notazione asintotica Θ (limite asintotico stretto)

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n)) = \{ f(n) : \text{esistono } c_1, c_2 > 0 \text{ ed } n_0 \text{ tali che per ogni } n \geq n_0 \quad 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \}$$



Alcune tecniche per l'individuazione di limiti inferiori

- Dimensione n dei dati
  - Se nel caso pessimo occorre analizzare tutti i dati allora $\Omega(n)$ è un limite inferiore alla complessità del problema (esempio: ricerca di un elemento o del massimo in un array)
  - E' una tecnica banale, la maggior parte dei problemi hanno limiti inferiori più alti
- Eventi contabili
  - Se la ripetizione di un evento un dato numero di volte è essenziale per la risoluzione di un problema
    - Esempio: nella generazione delle permutazioni di n lettere l'evento è la generazione di una nuova permutazione che si ripete per tutte le permutazioni, ossia n! volte.
    - Nota: n!

Regole per la valutazione della complessità (1)

- Scomposizione
  - alg è la sequenza di alg1 ed alg2;
  - alg1 è O(g1(n)); alg2 è O(g2(n))
  - alg è O(max(g1(n),g2(n)))
- Esempio

```
i=0;
```

g1(n)

```
while(i<n) {
  Stampastelle(i);
  i=i+1;
}
```

g2(n)

```
for(i=0; i<2*n; i++)
  scanf("%d", &numero);
```

Regole per la valutazione della complessità (1)

- Blocchi annidati
  - alg è composto da due blocchi annidati;
  - Blocco esterno è O(g1(n)); blocco interno è O(g2(n))
  - alg è O(g1(n)*g2(n))
- Esempio

g1(n)

g2(n)

```
for(i=0; i<n; i++) {
  scanf("%d", &j);
  printf("%d", j*j);
  do {
    scanf("%d", &numero);
    j=j+1;
  } while (j<=n);
```

Esempio

- Prodotto di matrici

```
float A[N][M], B[M][P], C[N][P];
int i, j, k;
```

O(N) O(P) O(M)

```
for(i=0; i<N; i++)
  for(j=0; j<P; j++) {
    C[i][j]=0;
    for(k=0; k<M; k++)
      C[i][j]+=A[i][k] * B[k][j];
  }
```

- Complessità asintotica del programma: O(N*P*M).

Regole per la valutazione della complessità (2)

- Sottoprogrammi ripetuti
  - alg applica ripetutamente un certo insieme di istruzioni la cui complessità all'i-esima esecuzione vale $f_i(n)$; il numero di ripetizioni è g(n)
  - alg è $O(\sum_{i=1}^{g(n)} f_i(n))$
  - per $f_i(n)$ tutte uguali … O(g(n) f(n))

Regole per la valutazione della complessità (3)

- Operazione dominante
  - Sia f(n) il costo di esecuzione di un algoritmo alg; un’istruzione i è dominante se viene eseguita g(n) volte, con f(n) <= a g(n)

  - Se un algoritmo ha una operazione dominante allora è O(g(n))

Esempio

- Ricerca binaria

```
int ricerca(int v [], int size, int k)
{ int inf =0, sup = size-1;
   while (sup >=inf)
   {  int med = (sup + inf ) / 2;
      if (k==v[med])
                        return med;
      else if (k>v[med])
                        inf = med+1;
      else sup = med–1
   }
   return -1;
}
```

Esempio

- Istruzioni dominanti
  - sup >=inf
  - med = (sup + inf) / 2
- Sequenza di elementi utili da considerare: n, n/2, n/4, … $n/2^i$
- Terminazione: numero elementi pari ad 1, ossia $n/2^i=1$
- $i=\log_2 n$, ossia l’algoritmo è O(log n).

Ricorsione e valutazione della complessità

- Negli algoritmi ricorsivi la soluzione di un problema si ottiene applicando lo stesso algoritmo ad uno o più sottoproblemi
  - Valutazione della complessità basata sulla soluzione di relazioni di ricorrenza
    - La funzione di complessità f(n) è definita in termini di se stessa su una dimensione inferiore dei dati
- La complessità dipende anche dal lavoro di combinazione
  - preparazione delle chiamate ricorsive e combinazione dei risultati ottenuti

Relazioni di ricorrenza con lavoro costante (1)

- Il lavoro di combinazione è indipendente dalla dimensione dei dati
- Relazioni lineari di ordine h
  - $C(1) = c_1, \quad C(2) = c_2, \ldots C(h) = c_h$
  - $C(n) = a_1\, C(n\text{-}1) + a_2\, C(n\text{-}2) + \ldots a_h\, C(n\text{-}h) + b \quad$ per $n > h$
  - La soluzione è di ordine esponenziale con n
  - Esempio: Fibonacci (h = 2)
    - $C(0) = C(1) = c$
    - $C(n) = C(n\text{-}1) + C(n\text{-}2) + b$

```
unsigned fib(unsigned n) {
    if (n<2) return n;
   else return fib(n-1)+fib(n-2);
}
```

Relazioni di ricorrenza con lavoro costante (2)

- Relazioni lineari di ordine h con $a_h = 1$ e $a_j = 0$ per $1 \leq j \leq h-1$
  - $C(1) = c_1$, $C(2) = c_2$, ... $C(h) = c_h$
  - $C(n) = C(n-h) + b$ per $n > h$
  - La soluzione è di ordine lineare con n
  - Esempio: Fattoriale (h = 1)
    - $C(0) = c$
    - $C(n) = C(n-1) + b$

```
int fact(int n) {
   if (n==0) return 1;
   else return n*fact(n-1);
}
```

Relazioni di ricorrenza con lavoro costante (3)

- Relazioni con partizione dei dati
  - $C(1) = c$
  - $C(n) = a\,C(n/p) + b$ per $n > 1$ con $p>1$
  - La soluzione è di ordine
    - $\log n$ se $a = 1$
    - $n^{\log_p a}$ se $a > 1$
  - Esempio: Ricerca binaria (complessità logaritmica)
    - $C(1) = c$
    - $C(n) = C(n/2) + b$

Ricerca Binaria Ricorsiva

```
int ricercaBinaria(int valore, int vettore[], int primo, int
ultimo)
{
      if (primo > ultimo) return -1;
      int mid=(primo+ultimo)/2;
      if (valore==vettore[mid])
       return mid;
      if (valore<vettore[mid])
       return ricercaBinaria(valore,vettore,primo,mid-1);
   else
       return ricercaBinaria(valore,vettore,mid+1,ultimo);
}
```

Relazioni di ricorrenza con lavoro lineare (1)

- Il lavoro di combinazione è proporzionale alla dimensione dei dati
- Relazioni lineari di ordine h con $a_h = 1$ e $a_j = 0$ per $1 \leq j \leq h-1$
  - $C(1) = c_1$, $C(2) = c_2$, … $C(h) = c_h$
  - $C(n) = C(n-h) + b\,n + d$ per $n > h$
  - La soluzione è di ordine <u>quadratico in n</u>

Relazioni di ricorrenza con lavoro lineare (2)

- Relazioni con partizione dei dati
  - $C(1) = c$
  - $C(n) = a\ C(n/p) + b\ n + d$ per $n > 1$
  - La soluzione è di ordine
    - lineare se $a < p$
    - $n \log n$ se $a = p$
    - $n^{\log_p a}$ se $a > p$
  - Esempio: Mergesort (complessità $n \log n$)
    - $C(1) = c$
    - $C(n) = 2\ C(n/2) + M(n) + c$
    - La complessità dell'algoritmo di Merge $M(n)$ è $O(n)$

Mergesort

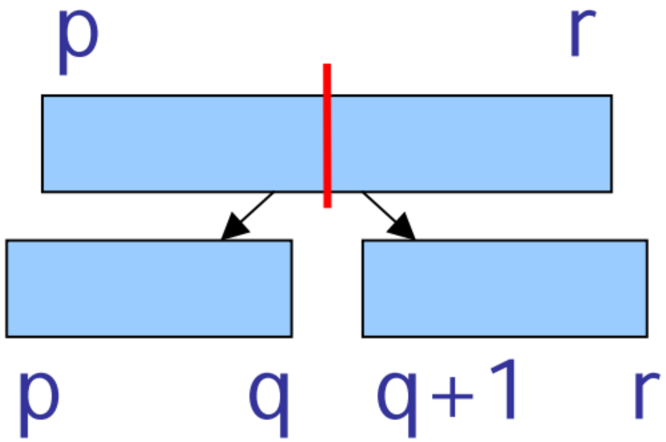
- Inventato da von Neumann nel 1945
- Esempio del paradigma algoritmico del divide et impera
- Richiede spazio ausiliario (O(N))
- E’ implementato come algoritmo standard nelle librerie di alcuni linguaggi (Perl, Java)
- E’ facile implementare una versione stabile

Mergesort

- Si divide il vettore dei dati in due parti ordinate separatamente, quindi si fondono le parti per ottenere un vettore ordinato globalmente
- il problema è la fusione

Mergesort

- Ricorsivo, “divide et impera”
- Stabile
- Divisione:
  - due sottovettori SX e DX rispetto al centro del vettore.

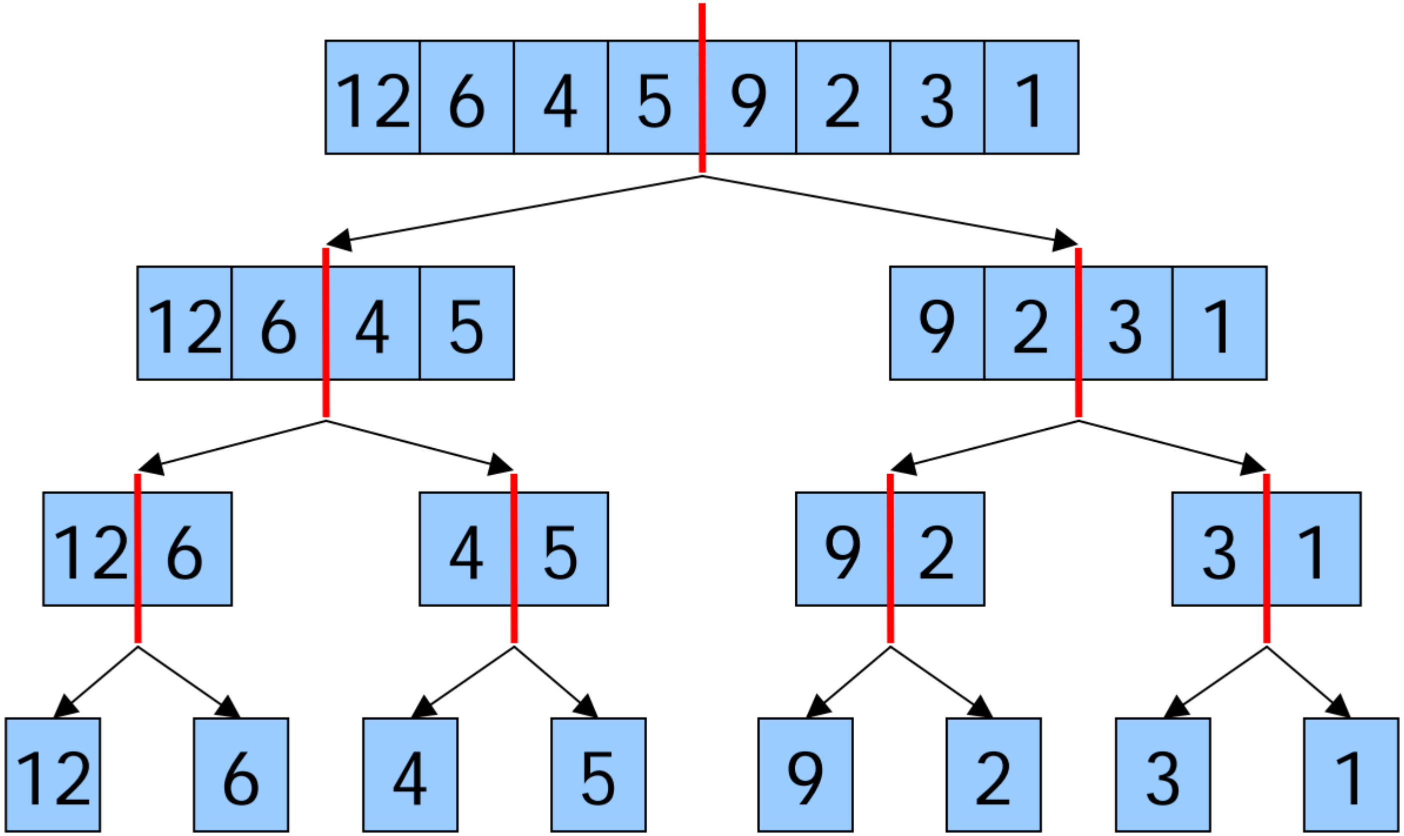


Mergesort

- Ricorsione
  - merge sort su sottovettore DX
  - merge sort su sottovettore SX
  - condizione di terminazione: con 1 (p=r) o 0 (p>r) elementi è ordinato
- Ricombinazione:
  - fondi i due sottovettori ordinati in un vettore ordinato

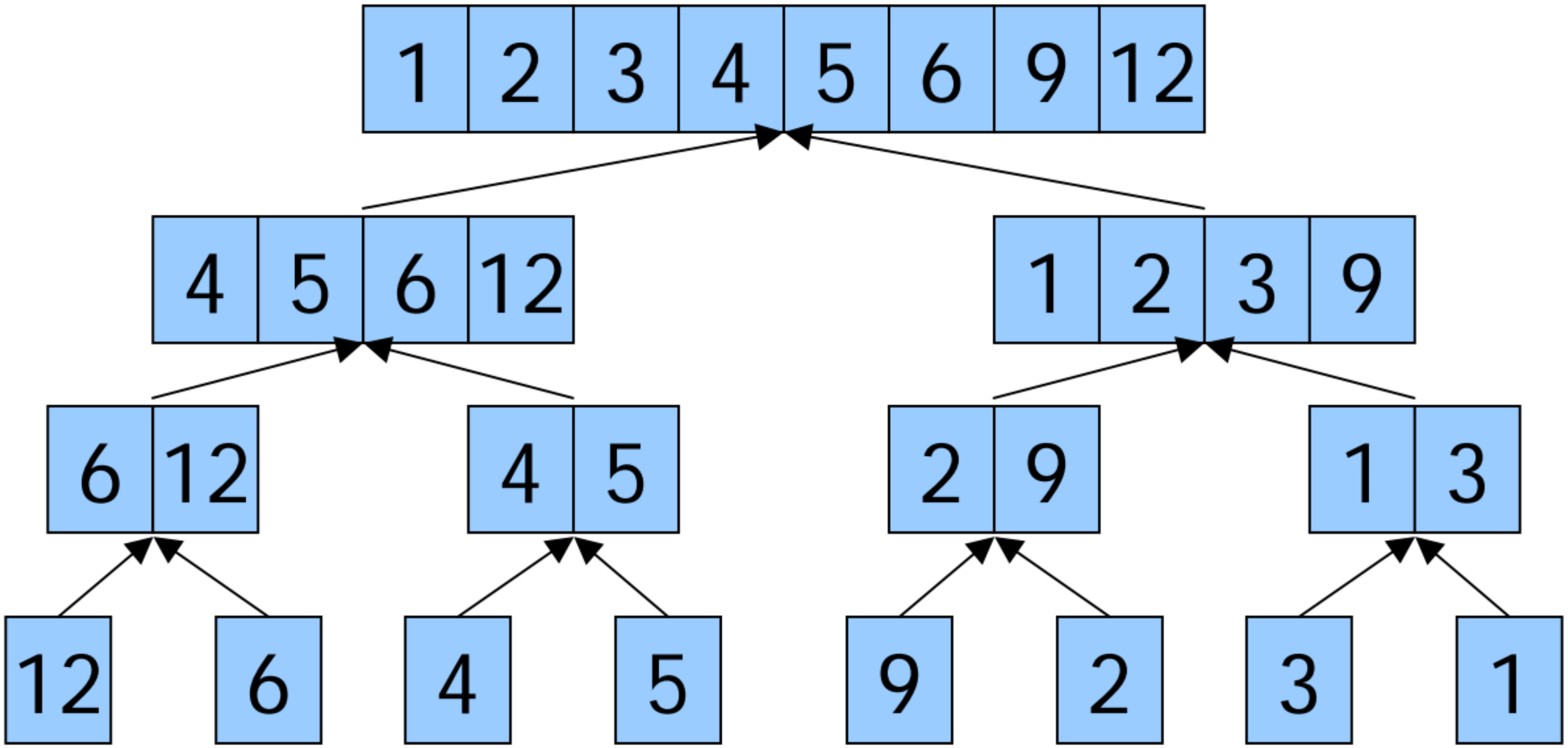
Esempio

- Divisione ricorsiva:



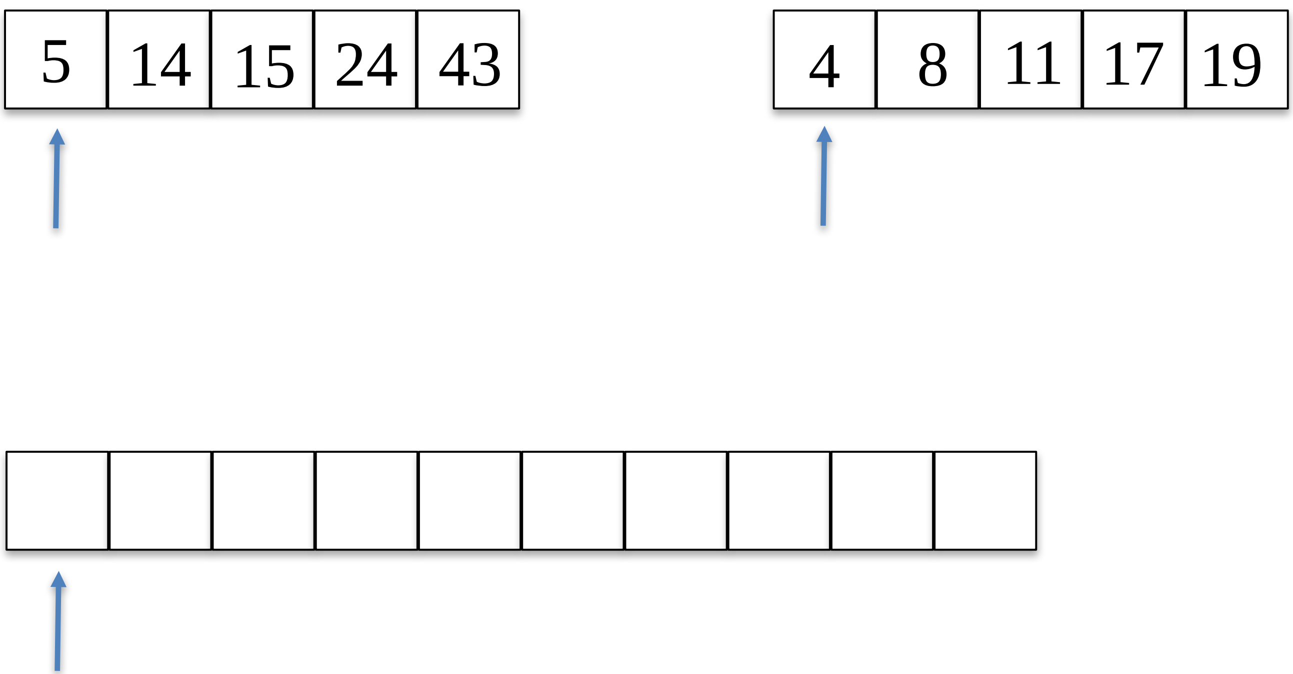
Esempio

- Ricombinazione:



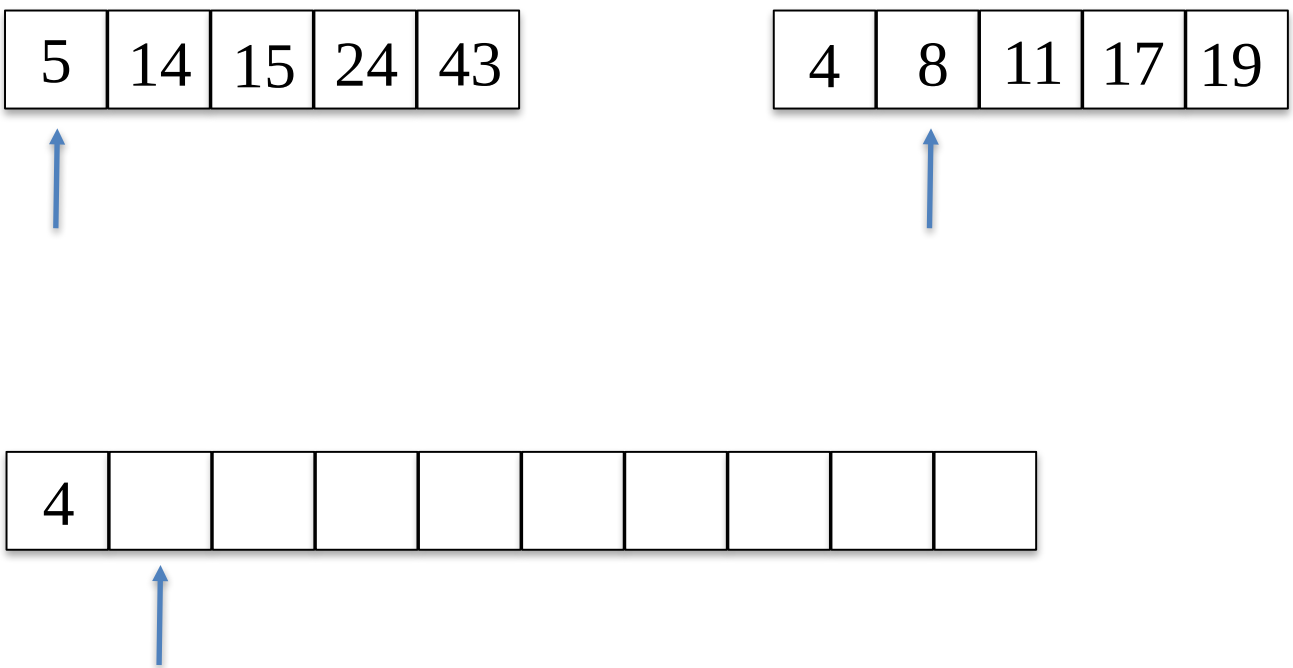
Esempio

- Fusione di due vettori ordinati:



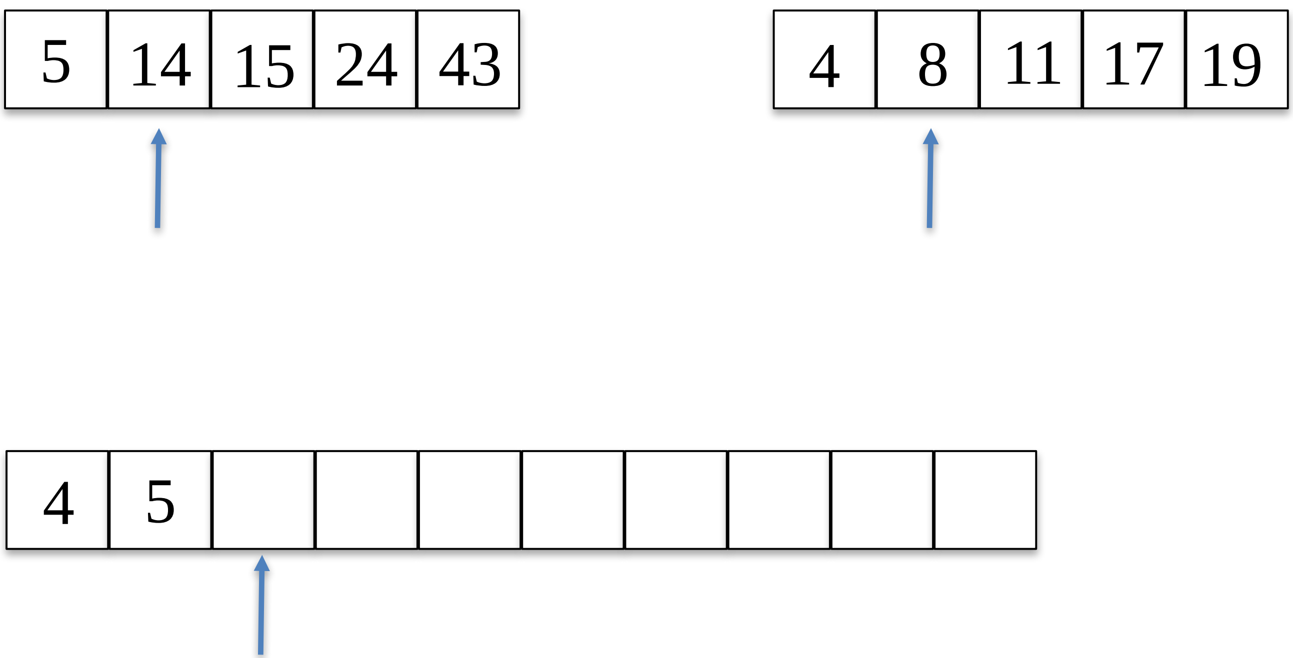
Esempio

- Fusione di due vettori ordinati:



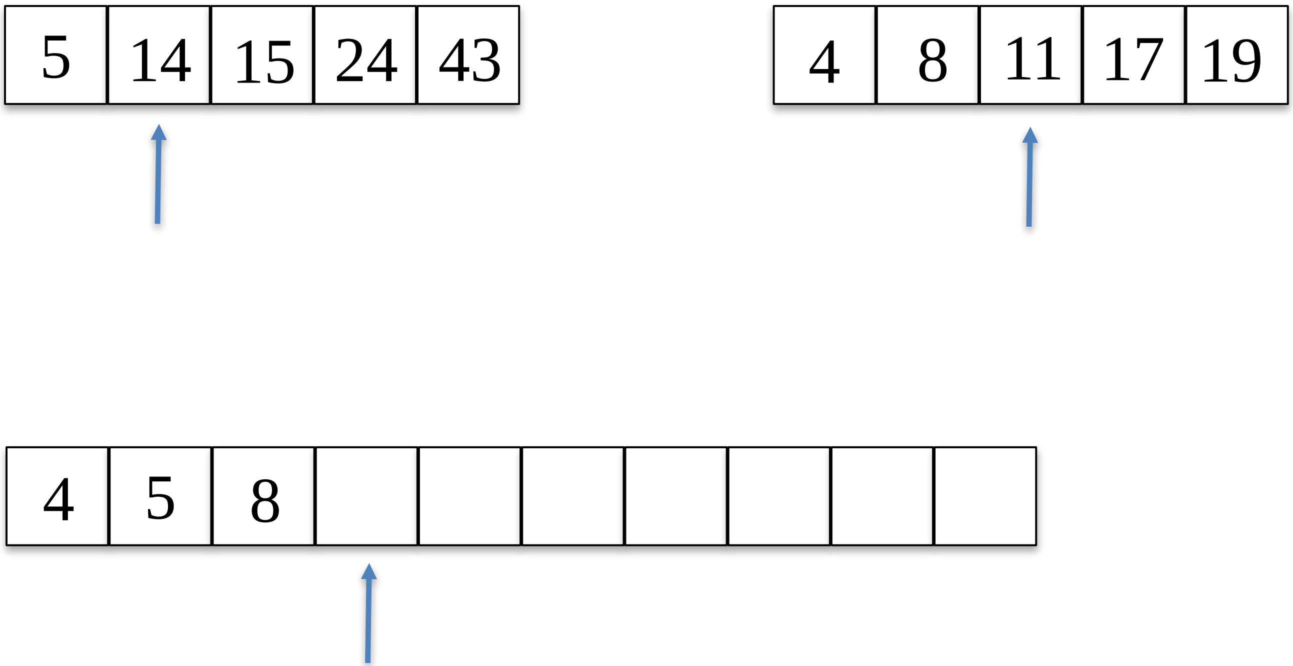
Esempio

- Fusione di due vettori ordinati:



Esempio

- Fusione di due vettori ordinati:



Esempio

- Fusione di due vettori ordinati:

5	14	15	24	43

4	8	11	17	19

4	5	8	11						

Esempio

- Fusione di due vettori ordinati:

5	14	15	24	43

4	8	11	17	19

4	5	8	11	14					

Esempio

- Fusione di due vettori ordinati:

5	14	15	24	43

4	8	11	17	19

4	5	8	11	14	15				

Esempio

- Fusione di due vettori ordinati:

5	14	15	24	43

4	8	11	17	19

4	5	8	11	14	15	17			

Esempio

- Fusione di due vettori ordinati:

5	14	15	24	43

4	8	11	17	19

il secondo vettore è terminato, non occorrono più confronti

4	5	8	11	14	15	17	19		

Esempio

- Fusione di due vettori ordinati:

5	14	15	24	43

4	8	11	17	19

il secondo vettore è terminato, non occorrono più confronti

4	5	8	11	14	15	17	19	24	

Esempio

- Fusione di due vettori ordinati:

5	14	15	24	43

4	8	11	17	19

i valori sono stati tutti inseriti, fine del procedimento

4	5	8	11	14	15	17	19	24	43

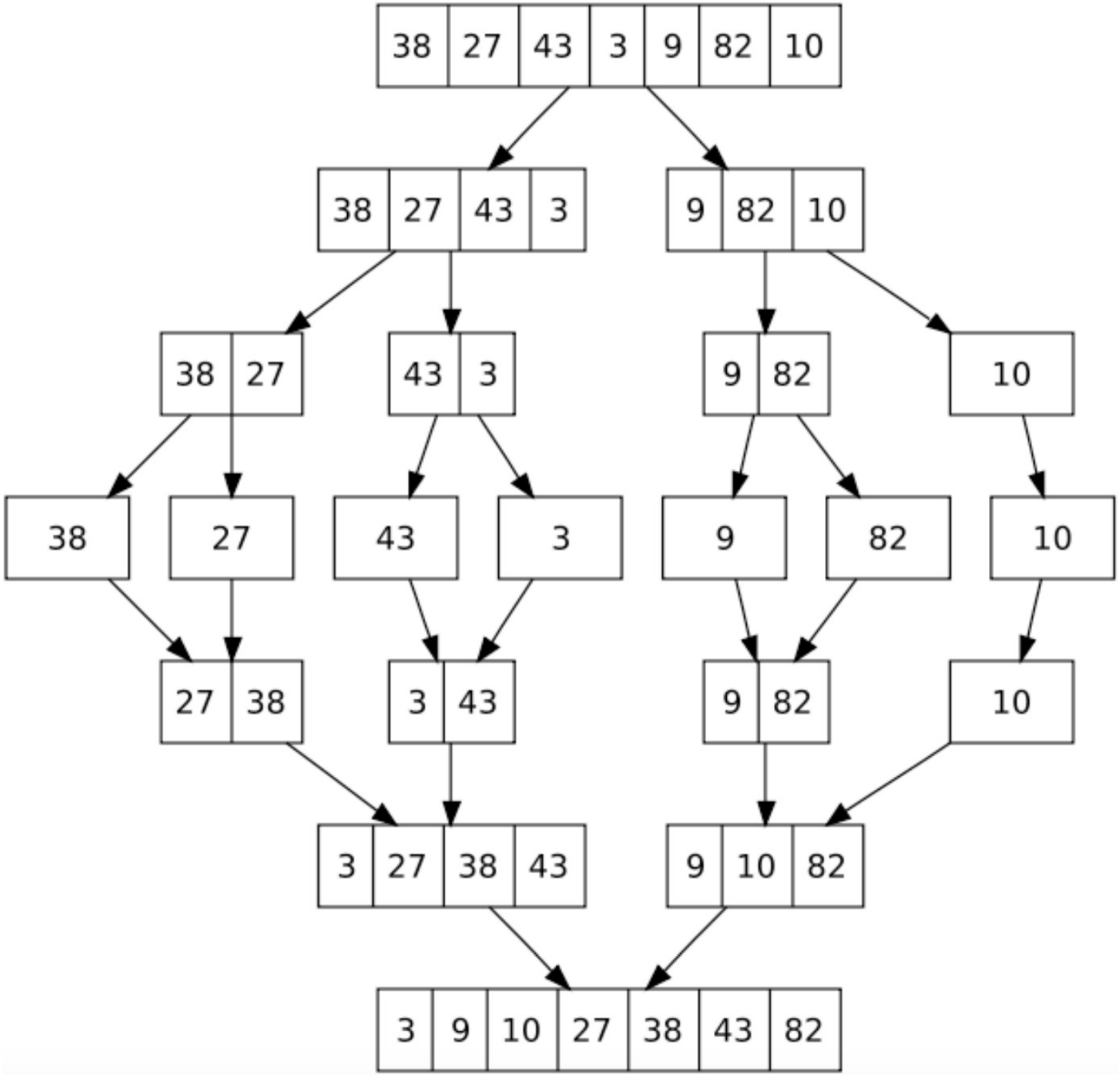
Implementazione

```
void MergeSort(int A[], int p, int r) {
     int q;
     if (p < r) {
          q = (p + r)/2;
          MergeSort(A, p, q);
          MergeSort(A, q+1, r);
          Merge(A, p, q, r);
     }
     return;
}
```

Implementazione

```
void Merge(int A[], int p, int q, int r){
     int B[MAX], i=p, j=q+1, k=p;
     while (i<=q && j<=r)
          if ( A[i] < A[j] )
               B[k++] = A[i++];
          else B[k++] = A[j++];
     while (i<=q) B[k++] = A[i++];
     while (j<=r) B[k++] = A[j++];
     for ( k=p; k<=r; k++ ) A[k] = B[k];
     return;
}
```

Esempio


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Mergesort: costo

- Dividi: calcola la metà di un vettore

$$D(n)=\Theta(1)$$

- Risolvi: risolve 2 sottoproblemi di dimensione n/2 ciascuno

$$2T(n/2)$$

- Terminazione: semplice test

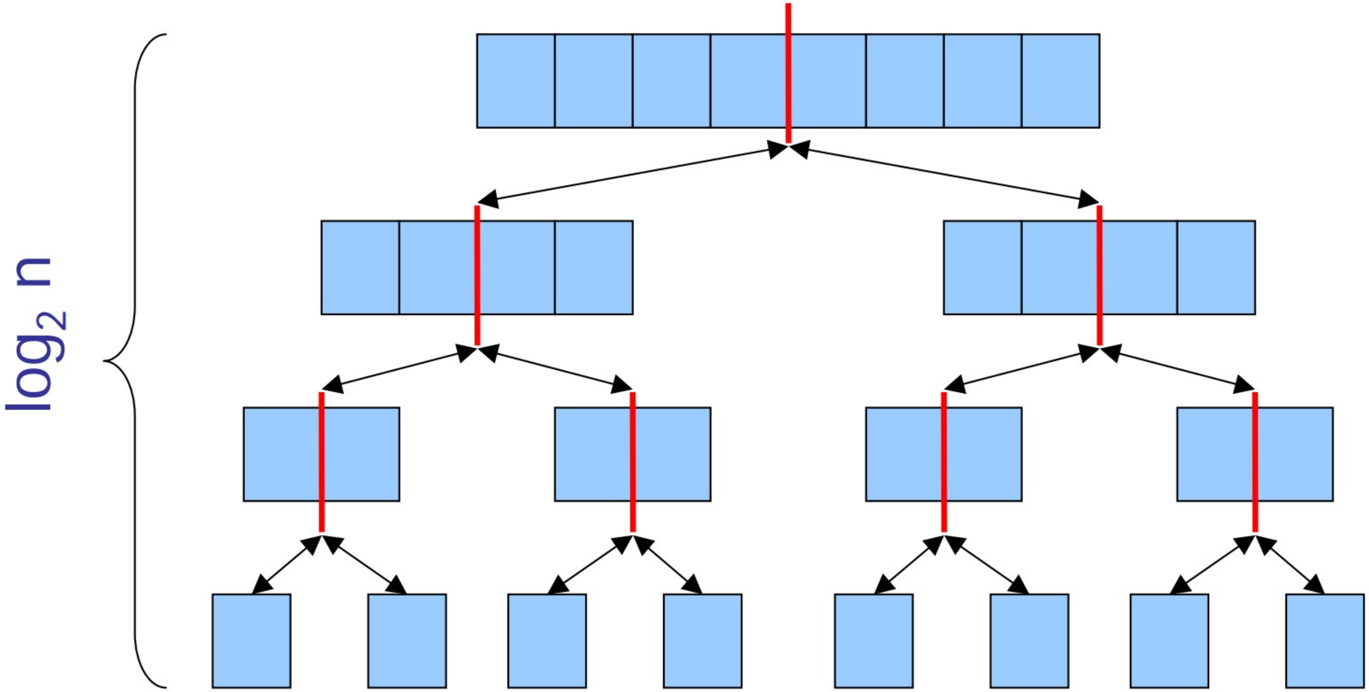
$$\Theta(1)$$

- Combina: basata su Merge

$$C(n) = \Theta(n)$$

Mergesort: costo

- Intuitivamente:



Livelli di ricorsione: $\log_2 n$

Operazioni per livello: n

Operazioni totali: $n \log_2 n$