8 Algoritmi

8.1 Divide et impera

È una tecnica di risoluzione di problemi che consiste in tre passi:

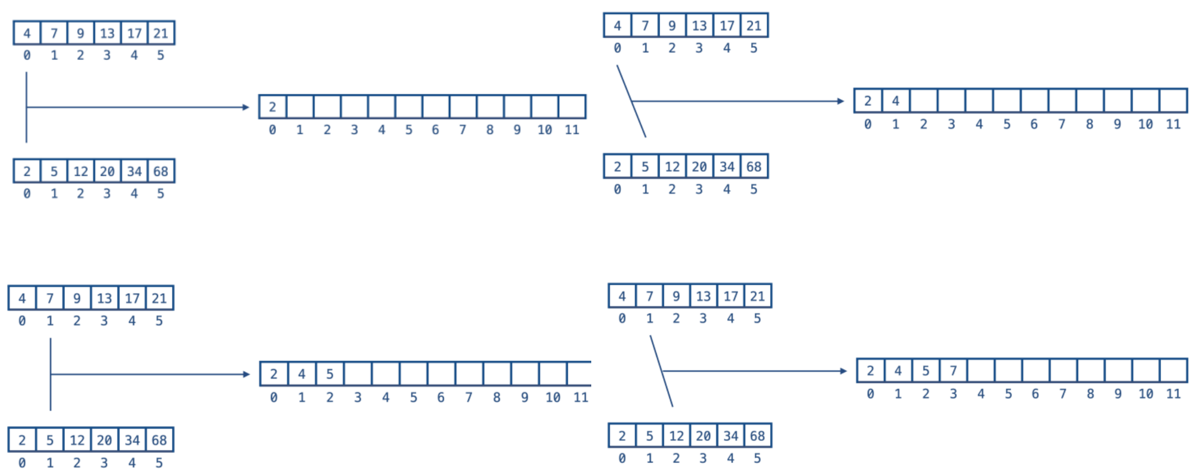
- **Dividere** il problema in 2 o più sotto problemi identici ma di dimensione ridotta rispetto a quello originale
- **Risolvere** i sotto problemi *ricorsivamente*
- **Combinare** le soluzioni dei sotto problemi per ottenere la soluzione del problema iniziale

8.2 Ordinamento

Definizione 8.1 (Algoritmo stabile)**.** *Un algoritmo di ordinamento si dice stabile quando preserva l'ordine iniziale tra due elementi con la stessa chiave.*

8.2.1 Merge sort

L'idea è di usare la tecnica precedentemente descritta del **Divide et Impera** e di spezzare l'array in due sotto-array di uguale dimensione, ordinarli e poi fonderli in uno unico.
La fusione verrà fatta confrontando i primi due elementi di ogni sotto-array, copiando il più piccolo nell'array finale, e proseguendo con il confronto del più grande con il successivo.



Esempio di implementazione:

```
void merge_sort(int a[], size_t dim, char order) {
  sort(a, 0, dim-1, order);
}

void sort(int a[], size_t inizio, size_t fine, char order) {
  if ((fine - inizio) >= 1) {
    // Passo ricorsivo
    size_t centro1 = (inizio + fine)/2;
    zie_t centro2 = centro1 + 1;

    sort(a, inizio, centro1, order);
    sort(a, centro2, fine, order);

    merge(a, inizio, centro1, centro2, fine, order);
  }
  // Il caso base non serve, un array di un elemento e' ordinato
}

void merge(int a[], size_t sin, size_t centro1, size_t centro2, size_t dx, char
    order) {
  size_t sin i = sin;
```

```c
    size_t dx_i = centro2;
    size_t fondi i = 0;
    int temp_a[dx - sin + 1];

    while (sin_i <= centro1 && dx_i <= dx) {
      switch (order) {
        case 'I':
          if (a[sin_i] <= a[dx_i]) {
            temp_a[fondi_i++] = a[sin_i++];
          } else {
            temp_a[fondi_i++] = a[dx_i++];
          }
          break;
        default:
          if (a[sin_i] <= a[dx_i]) {
            temp_a[fondi_i++] = a[dx_i++];
          } else {
            temp_a[fondi_i++] = a[sin_i++];
          }
          break;
      }
    }

    // Se esaurisco il sotto-array sinistro
    if (sin_i == centro2) {
      while (dx_i <= dx) {
        temp_a[fondi_i++] = a[dx_i++];
      }
    } else {
      // Se esaurisco quello destro
      while (sin_i <= centro1) {
        temp_a[fondi_i++] = a[sin_i++];
      }
    }

    // Copio l'array temporaneo in quello originale
    for (size_t i = sin; i <= dx; i++) {
      a[i] = temp_a[i-sin];
    }
  }
```

Listing 11: Algoritmo merge sort

8.2.2 Insertion sort

Proprietà: al termine del passo j-esimo dell'algoritmo l'elemento j-esimo viene in inserito al posto giusto e i primi $j+1$ elementi sono ordinati.

```
insertionSort(A) =
var j:Int = 0;
var i:Int = 0;     Θ(1)
var k:int = 0;
for (j=1; j<n; j++) {    n − 1 volte
  k = A[j];
  i = j-1;     Θ(1)  n − 1 volte
  while(i >= 0 && A[i]>k) {
    A[i+1] = A[i];       Θ(1)  sum_{j=1}^{n-1} (t_j − 1) volte
    i=i-1;
  }
  A[i+1] = k;    Θ(1)  n − 1 volte
}
```

Listing 12: Algoritmo insertion sort

Complessità:

$$\sum_{j=1}^{n-1} t_j$$

0	1	2	3	4	5	j	i	k	while
5	2	4	6	1	3	0	0	0	no
5	2	4	6	1	3	1	0	2	si
5	5	4	6	1	3	1	-1	2	no
2	5	4	6	1	3	1	-1	2	no
2	5	4	6	1	3	2	1	4	si
2	5	5	6	1	3	2	0	4	no
2	4	5	6	1	3	2	0	4	no
2	4	5	6	1	3	3	2	6	no
2	4	5	6	1	3	3	2	6	no

Table 2: Esempio di esecuzione

- Caso pessimo: l'array è ordinato decrescente e quindi ogni volta devo scalare l'elemento fino alla prima posizione. Abbiamo che $t_j = j$ e $\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$, quindi $O(n^2)$
- Caso migliore: l'array è ordinato crescente e quindi per ogni iterazione non entro nel while perché la condizione è falsa. Abbiamo $t_j = 1$ e $\sum_{j=1}^{n-1} j = n-1$, quindi $O(n)$
- Caso medio: come il caso pessimo $O(n^2)$

Correttezza:

- dimostro l'**invariante di ciclo** per assicurarmi che la mia proprietà venga mantenuta durante tutta l'esecuzione. Lo faccio tramite *induzione*:
  - Caso base: per $j = 1$
  - Hp induttiva: per $j = n'$
  - Passo induttivo: dimostro che vale anche per $j = n' + 1$
- verifico la **terminazione**: il *for* è eseguito esattamente $n-1$ volte e il *while* al più $j-1$ volte, quindi tutte le iterazioni sono finite e l'algoritmo termina.

Memoria impiegata: ordina in loco quindi non usa memoria aggiuntiva.

8.2.3 Selection sort

Proprietà: al termine del passo j-esimo dell'algoritmo i primi $j+1$ elementi di A sono ordinati e contengono i $j+1$ elementi più piccoli di A.

```
insertionSort(A) =
var j:Int = 0;
var i:Int = 0;     Θ(1)
var min:int = 0;
for (i=0; i<n-1; i++) {   n − 1 volte
  min = i;    Θ(1) n − 1 volte
  for(j=i+1; j<n; j++) {
    if A[j] < A[min] {min = j};      Θ(1) Σ_{j=1}^{n−1} (t_j − 1) volte
  }
  swap(A[i],A[min]);    Θ(1) n − 1 volte
}
```

Listing 13: Algoritmo selection sort

Complessità

$$\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} \in O(n^2)$$

0	1	2	3	4	5	j	i	min
5	2	4	6	1	3	0	0	0
1	2	4	6	5	3	1	0	4
1	2	4	6	5	3	2	1	1
1	2	3	6	5	4	3	2	5
1	2	3	4	5	6	4	3	3
1	2	3	4	5	6	5	4	4

Table 3: Esempio di esecuzione

- Caso pessimo: $O(n^2)$
- Caso migliore: $O(n^2)$
- Caso medio: $O(n^2)$

Correttezza:

- dimostro l'**invariante di ciclo** per assicurarmi che la mia proprietà venga mantenuta durante tutta l'esecuzione. Sempre tramite induzione.
- verifico la **terminazione** in maniera analoga all'insertion sort.

Memoria impiegata: ordina in loco quindi non usa memoria aggiuntiva.

8.2.4 Bubble sort

Questo algoritmo scorre l'array e, a coppie, ordina gli elementi facendo più passate. Il nome *bubble* deriva dal fatto che ad ogni passata i numeri più grandi (o piccoli) si spostano verso la fine dell'array come le bolle d'aria salgono a galla.

```c
void bubble_sort (int a[], size_t dim, char order) {
  int temp;
  for (unsigned int passate = 0; passate < dim; passate++) {
    for (size_t i=0; i < (dim - 1); i++) {
      switch (order) {
        case 'I':
          if (a[i] > a[i+1]) {
            temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;
          }
          break;
        default:
          if (a[i] < a[i+1]) {
            temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;
             break;
          }
      }
    }
  }
}
```

Listing 14: Algoritmo bubble sort

Complessità
Il primo *for* esegue n cicli e quello interno ne esegue $n-1$. Di conseguenza la complessità è $n \cdot (n-1)$, ovvero $\mathbf{n^2}$.

8.3 Linear sort

Gli algoritmi di ordinamento di questo tipo sfruttano il fatto che l'array da ordinare abbia determinate proprietà.

Esempio 8.1. Dato un array A di n interi compresi tra 1 e k:

$$\forall 0 < j \leq n . A[j] \in [1, \ldots, k]$$

```
linearSort(A:[Int], B:[Int], k:Int) -> Void {
  // Inizializzo un array che tiene conto dei numeri da 1 a k
  for (var i:Int = 1; i<=k; i++) C[i] = 0;  Θ(k)

  var j:Int = 1;
  // Conto quante volte compare ogni numero nell'array originale
  for (j=1; j<=n; j++) C[A[j]] += 1;  Θ(n)

  j=1;
  var z:Int = 1;
  // Dispongo ogni numero nell'array finale in ordine sapendo quante volte compare
  for (z=1; z <= k; z++) {  Θ(k)
    for (var v:Int = 0; v < C[z]; v++) {  Θ(n)
      B[j] = z;
      j++;
    }
  }
}
```

Listing 15: Algoritmo linear sort

Complessità
In questo caso la complessità è $\Theta(n + k)$ e si usa quando $k \in O(n)$.

8.3.1 Radix sort

Questo algoritmo funziona in maniera simile a come il cervello umano ordina gruppi di numeri: si ordinano (tramite un algoritmo di ordinamento **stabile** prima le cifre delle migliaia, poi quelle delle centinaia, quelle delle decine ed infine le unità. Notiamo però che il risultato NON è corretto.

1094	**9**86	10**9**4	12**5**	1120
986	**2**34	1**2**5	112**0**	234
234	**1**25	11**2**0	234	1094
125	1**0**94	2**3**4	98**6**	125
1120	1**1**20	9**8**6	109**4**	986

Per farlo funzionare dobbiamo ordinare le cifre partendo da quelle meno significative, quindi dalle unità.

109**4**	11**2**0	1**1**20	**1**094	125
98**6**	10**9**4	**1**25	**1**120	234
23**4**	2**3**4	**2**34	125	986
12**5**	1**2**5	**9**86	234	1094
112**0**	9**8**6	1**0**94	986	1120

9 Complessità

9.1 Notazione asintotica

Quando scriviamo un algoritmo, per calcolarne il costo, bisogna fare una serie di assunzioni sulla macchina astratta su cui lavoriamo:

- L'accesso alle celle di memoria avviene in tempo costante.
- Le operazioni elementari avvengono in tempo costante:
  - Operazioni aritmetiche e logiche della ALU