Lesson 11 – Complexity of Algorithms

Sommario

1. **Introduzione**
2. **Analisi esatta di complessità**
3. **Analisi asintotica**
4. **Notazione *O***
5. **Caso peggiore/caso migliore**

Cosa si analizza/valuta di un algoritmo

- Correttezza
  - Dimostrazione formale (matematica)
  - Ispezione informale

- Efficienza/Utilizzo delle risorse
  - **Tempo di esecuzione**
  - *Utilizzo della memoria*

- Semplicità
  - Facile da capire e da manutenere

Tempo di esecuzione

- Il tempo di esecuzione di un programma dipende da:
  - Hardware
  - Compilatore
  - **Input**

- Per poter confrontare l'efficienza di programmi diversi, è necessario prescindere (astrarre) dalle specificità di particolari computer e compilatori:
  - **Modello di Computazionale**

Modello computazionale

- Modello RAM (Random-Access Memory)
  - Memoria principale infinita
    - Ogni cella di memoria può contenere una quantità di dati finita.
    - Impiega lo stesso tempo per accedere ad ogni cella di memoria.
  - Singolo processore
  - Set delle istruzioni elementari
- Operazioni Elementari (OE) eseguibili in una unità di tempo (per semplicità)
  - dichiarazione di variabile
  - addizione/moltiplicazione/divisione/modulo/…
  - assegnamento
  - confronto
  - lettura/scrittura di un intero/carattere

Problema

- Input
  - Una lista $L = (a_1, a_2, \ldots, a_N)$ di N interi (positivi e negativi)

- Output
  - $$M = \max\left\{\left|\sum_{j=1}^{N}\left(a_j - a_i\right)\right| \mid i \in \{1, \ldots, N\}\right\}$$

- Esempio:
  - $L = (1, 2, -1)$ dove $N = 3$
  - $M = \mathbf{5}$ ottenuto quando $\boldsymbol{i = 3}$

Algoritmo 1

```
int computeM_1(int L[], int N)
{
  int max = 0;
  for(int i = 1; i <= N; i++)
  {
    int sum = 0;
    for(int j = 1; j <= N; j++)
      sum += L[j-1] - L[i-1];
    if(abs(sum) > max)
      max = abs(sum);
  }
  return max;
}
```

Algoritmo 1

```
int computeM_1(int L[], int N)
{
  int max = 0; // 2 OE
  for(int i = 1; i <= N; i++)
  {
    int sum = 0;
    for(int j = 1; j <= N; j++)
      sum += L[j-1] - L[i-1];
    if(abs(sum) > max)
      max = abs(sum);
  }
  return max;
}
```

Algoritmo 1

```
int computeM_1(int L[], int N)
{
  int max = 0;
  for(int i = 1; i <= N; i++) // 3(N+1) OE
  {
    int sum = 0;
    for(int j = 1; j <= N; j++)
      sum += L[j-1] - L[i-1];
    if(abs(sum) > max)
      max = abs(sum);
  }
  return max;
}
```

int i=1; // 2 OE

i<=N; // N+1 OE

i++; // 2N OE

Algoritmo 1

```
int computeM_1(int L[], int N)
{
  int max = 0;
  for(int i = 1; i <= N; i++)
  {
    int sum = 0; // 2N OE
    for(int j = 1; j <= N; j++)
      sum += L[j-1] - L[i-1];
    if(abs(sum) > max)
      max = abs(sum);
  }
  return max;
}
```

Algoritmo 1

```
int computeM_1(int L[], int N)
{
  int max = 0;
  for(int i = 1; i <= N; i++)
  {
    int sum = 0;
    for(int j = 1; j <= N; j++) // 3N(N+1) OE
      sum += L[j-1] - L[i-1];
    if(abs(sum) > max)
      max = abs(sum);
  }
  return max;
}
```

```
int j=1; // 2*N OE
j<=N; // (N+1)*N OE
j++; // 2N*N OE
```

Algoritmo 1

```
int computeM_1(int L[], int N)
{
  int max = 0;
  for(int i = 1; i <= N; i++)
  {
    int sum = 0;
    for(int j = 1; j <= N; j++)
      sum += L[j-1] - L[i-1]; // 5N * N OE
    if(abs(sum) > max)
      max = abs(sum);
  }
  return max;
}
```

Algoritmo 1

```
int computeM_1(int L[], int N)
{
  int max = 0;
  for(int i = 1; i <= N; i++)
  {
    int sum = 0;
    for(int j = 1; j <= N; j++)
      sum += L[j-1] - L[i-1];
    if(abs(sum) > max) // 2N OE
      max = abs(sum); // 2N OE (at most!)
  }
  return max;
}
```

Algoritmo 1

```
int computeM_1(int L[], int N)
{
  int max = 0;
  for(int i = 1; i <= N; i++)
  {
    int sum = 0;
    for(int j = 1; j <= N; j++)
      sum += L[j-1] - L[i-1];
    if(abs(sum) > max)
      max = abs(sum);
  }
  return max; // 1 OE
}
```

Algoritmo 1

```
int computeM_1(int L[], int N)
{
  int max = 0;
  for(int i = 1; i <= N; i++)
  {
    int sum = 0;
    for(int j = 1; j <= N; j++)
      sum += L[j-1] - L[i-1];
    if(abs(sum) > max)
      max = abs(sum);
  }
  return max;
} // TOTALE: 8N² + 12N + 6
```

Un algoritmo più “furbo”

- Input
  - Una lista $L = (a_1,\ a_2,\ldots,\ a_N)$ di N interi (positivi e negativi)

- Output
  - $M = \max\left\{\left|\sum_{j=1}^{N}\left(a_j - a_i\right)\right| \ \mid\ i \in \{1,\ldots,N\}\right\} =$

$$= \max\left\{\left|S - N \cdot a_i\right| \ \mid\ i \in \{1,\ldots,N\}\right\}$$

$$\text{dove} \quad S = \sum_{j=1}^{N}\left(a_j\right)$$

Algoritmo 2

```
int computeM_2(int L[], int N)
{
  int sumL = 0; // 2 OE
  for(int i = 1; i <= N; i++)
    sumL += L[i-1];
  int max = 0;
  for(int i = 1; i <= N; i++)
  {
    int temp = abs(sumL - N * L[i-1]);
    if(temp > max)
      max = temp;
  }
  return max;
}
```

Algoritmo 2

```
int computeM_2(int L[], int N)
{
  int sumL = 0;
  for(int i = 1; i <= N; i++) // 3(N+1) OE
    sumL += L[i-1];
  int max = 0;
  for(int i = 1; i <= N; i++)
  {
    int temp = abs(sumL - N * L[i-1]);
    if(temp > max)
      max = temp;
  }
  return max;
}
```

```
int i=1; // 2 OE
i<=N; // N+1 OE
i++; // 2N OE
```

Algoritmo 2

```
int computeM_2(int L[], int N)
{
  int sumL = 0;
  for(int i = 1; i <= N; i++)
    sumL += L[i-1]; // 3N OE
  int max = 0;
  for(int i = 1; i <= N; i++)
  {
    int temp = abs(sumL - N * L[i-1]);
    if(temp > max)
      max = temp;
  }
  return max;
}
```

Algoritmo 2

```
int computeM_2(int L[], int N)
{
  int sumL = 0;
  for(int i = 1; i <= N; i++)
    sumL += L[i-1];
  int max = 0; // 2 OE
  for(int i = 1; i <= N; i++)
  {
    int temp = abs(sumL - N * L[i-1]);
    if(temp > max)
      max = temp;
  }
  return max;
}
```

Algoritmo 2

```
int computeM_2(int L[], int N)
{
  int sumL = 0;
  for(int i = 1; i <= N; i++)
    sumL += L[i-1];
  int max = 0;
  for(int i = 1; i <= N; i++) // 3(N+1) OE
  {
    int temp = abs(sumL - N * L[i-1]);
    if(temp > max)
      max = temp;
  }
  return max;
}
```

```
int i=1; // 2 OE
i<=N; // N+1 OE
i++; // 2N OE
```

Algoritmo 2

```
int computeM_2(int L[], int N)
{
  int sumL = 0;
  for(int i = 1; i <= N; i++)
    sumL += L[i-1];
  int max = 0;
  for(int i = 1; i <= N; i++)
  {
    int temp = abs(sumL - N * L[i-1]); // 5N OE
    if(temp > max) // N OE
      max = temp; // N OE (at most)
  }
  return max;
}
```

Algoritmo 2

```
int computeM_2(int L[], int N)
{
  int sumL = 0;
  for(int i = 1; i <= N; i++)
    sumL += L[i-1];
  int max = 0;
  for(int i = 1; i <= N; i++)
  {
    int temp = abs(sumL - N * L[i-1]);
    if(temp > max)
      max = temp;
  }
  return max; // 1 OE
}
```

Algoritmo 2

```
int computeM_2(int L[], int N)
{
  int sumL = 0;
  for(int i = 1; i <= N; i++)
    sumL += L[i-1];
  int max = 0;
  for(int i = 1; i <= N; i++)
  {
    int temp = abs(sumL - N * L[i-1]);
    if(temp > max)
      max = temp;
  }
  return max;
} // TOTALE: 16N + 11 OE
```

Tempi di esecuzione

Algoritmo	Tempo di Esecuzione (n° OE)
computeM_1	$8N^2 + 12N + 6$
computeM_2	$16N + 11$

Algoritmo	N=1	N=2	N=4	N=8	N=16	N=32	N=64	N=128
computeM_1	26	62	182	614	2246	8582	33542	132614
computeM_2	27	43	75	139	267	523	1035	2059

OSSERVAZIONE: **La differenza diventa molto rilevante al crescere dell'Input: consideriamo l'Analisi Asintotica!**

Analisi asintotica: notazione *O*

- Con la **notazione** ***O*** indichiamo che una funzione f è limitata superiormente (o *dominata*) da un'altra funzione g.

- Più formalmente:

$$f(n) \in O(g(n)) \quad \Leftrightarrow \quad \lim_{n \to \infty} \left| \frac{f(n)}{g(n)} \right| < \infty$$

- Quindi
  - $f \in \mathcal{O}(1)$ significa che f è limitata
  - $f \in \mathcal{O}(n)$ significa che f non cresce più velocemente di una funzione lineare

- Il significato di $f \in \mathcal{O}(g)$ è simile a quello di $p \leq q$ tra numeri.

Esempi

$$\lim_{n\to+\infty}\frac{2^n}{n^2}=\infty \qquad \lim_{n\to+\infty}\frac{2n^2+n}{n^2}=2 \qquad \lim_{n\to+\infty}\frac{2n^2+n}{2^n}=0$$

$$2^n \notin O\left(n^2\right) \qquad 2n^2+n\in O\left(n^2\right) \qquad 2n^2+n\in O\left(2^n\right)$$

Complessità algoritmica

- La complessità di un algoritmo può essere espressa mediante notazione asintotica: “come varia asintoticamente il numero di Operazioni Elementari al variare di N ?”
  - $\mathcal{O}(1)$ *Costante*
  - $\mathcal{O}(\log_2 N)$ *Sub-lineare*
  - $\mathcal{O}(N)$ *Lineare*
  - $\mathcal{O}(N \log_2 N)$ *Approssimativamente lineare*
  - $\mathcal{O}(N^2)$ *Quadratica*
  - $\mathcal{O}(N^3)$ *Cubica*
  - $\mathcal{O}(k^N)$ *Esponenziale*

- Due algoritmi per risolvere il *problema dell’ordinamento*:
  - *Bubblesort*: $\mathcal{O}(N^2)$
  - *Mergesort*: $\mathcal{O}(N \log_2 N)$

Tempi di esecuzione …….

Algoritmo	Tempo di Esecuzione (n° OE)	Complessità
computeM_1	$8N^2 + 12N + 6$	$O(N^2)$
computeM_2	$16N + 11$	$O(N)$

Problema

- Input
  - Una lista $L = (a_1, a_2, \ldots, a_N)$ di N interi (positivi e negativi)

- Output
  - $$S = \max\left(\left\{ \sum_{k=i}^{j} a_k \mid 1 \le i \le j \le N \right\} \right)$$

- Esempio:
  - $L = (2, -4, 8, 3, -5, 4, 6, -7, 2)$ dove $N = 9$
  - $S = \mathbf{16}$ ottenuto quando $\boldsymbol{i = 3}$ e $\boldsymbol{j = 7}$

Algoritmo 1

```
int computeS_1 (int L[], int N)
{
 int maxSum = 0;
 for (int i=1; i<=N; i++)
 {
  for (int j=i; j<=N; j++)
  {
   int sum = 0;
   for (int k=i; k<=j; k++)
    sum += L[k-1];
   if (sum > maxSum)
    maxSum = sum;
  }
 }
 return maxSum;
}
```

Algoritmo 1

```
int computeS_1 (int L[], int N)
{
 int maxSum = 0; // 2 OE
 for (int i=1; i<=N; i++)
 {
  for (int j=i; j<=N; j++)
  {
   int sum = 0;
   for (int k=i; k<=j; k++)
    sum += L[k-1];
   if (sum > maxSum)
    maxSum = sum;
  }
 }
 return maxSum;
}
```

Algoritmo 1

```
int computeS_1 (int L[], int N)
{
 int maxSum = 0;
 for (int i=1; i<=N; i++) // 3(N+1) OE
 {
  for (int j=i; j<=N; j++)
  {
   int sum = 0;
   for (int k=i; k<=j; k++)
    sum += L[k-1];
   if (sum > maxSum)
    maxSum = sum;
  }
 }
 return maxSum;
}
```

```
int i=1; // 2 OE
i<=N; // N+1 OE
i++; // 2N OE
```

Algoritmo 1

```
int computeS_1 (int L[], int N)
{
 int maxSum = 0;
 for (int i=1; i<=N; i++)
 {
  for (int j=i; j<=N; j++) // 3N(N+3)/2 OE
  {
   int sum = 0;
   for (int k=i; k<=j; k++)
    sum += L[k-1];
   if (sum > maxSum)
    maxSum = sum;
  }
 }
 return maxSum;
}
```

```
int j=i; // 2N OE
j<=N; // N(N+1)/2 + N OE
j++; // 2N(N+1)/2 OE
```

$$N+(N-1)+(N-2)+\ldots+2+1=\sum_{i=1}^{N}(N-i+1)=\frac{N(N+1)}{2}$$

Algoritmo 1

```
int computeS_1 (int L[], int N)
{
 int maxSum = 0;
 for (int i=1; i<=N; i++)
 {
  for (int j=i; j<=N; j++)
  {
   int sum = 0; // 2N(N+1)/2 OE
   for (int k=i; k<=j; k++)
    sum += L[k-1];
   if (sum > maxSum) // N(N+1)/2 OE
    maxSum = sum; // N(N+1)/2 OE (at most!)
  }
 }
 return maxSum;
}
```

$$N+(N-1)+(N-2)+\ldots+2+1=\sum_{i=1}^{N}(N-i+1)=\frac{N(N+1)}{2}$$

Algoritmo 1

```
int computeS_1 (int L[], int N)
{
 int maxSum = 0;
 for (int i=1; i<=N; i++)
 {
  for (int j=i; j<=N; j++)
  {
   int sum = 0;
   for (int k=i; k<=j; k++)
    sum += L[k-1];
   if (sum > maxSum)
    maxSum = sum;
  }
 }
 return maxSum;
}
```

// $N(N^2+6N+5)/2$ OE

int k=i; // $2N(N+1)/2$ OE

k<=j; // $N(N^2+3N+2)/6$ + $N(N+1)/2$ OE

k++; // $2N(N^2+3N+2)/6$ OE

$$\sum_{i=1}^{N}\sum_{j=i}^{N}(j-i+1)=\frac{N\left(N^2+3N+2\right)}{6}$$

Algoritmo 1

```
int computeS_1 (int L[], int N)
{
 int maxSum = 0;
 for (int i=1; i<=N; i++)
 {
  for (int j=i; j<=N; j++)
  {
   int sum = 0;
   for (int k=i; k<=j; k++)
    sum += L[k-1]; // 3N(N²+3N+2)/6 OE
   if (sum > maxSum)
    maxSum = sum;
  }
 }
 return maxSum;
}
```

Algoritmo 1

```
int computeS_1 (int L[], int N)
{
 int maxSum = 0;
 for (int i=1; i<=N; i++)
 {
  for (int j=i; j<=N; j++)
  {
   int sum = 0;
   for (int k=i; k<=j; k++)
    sum += L[k-1];
   if (sum > maxSum)
    maxSum = sum;
  }
 }
 return maxSum; // 1 OE
}
```

Algoritmo 1

```
int computeS_1 (int L[], int N)
{
 int maxSum = 0;
 for (int i=1; i<=N; i++)
 {
  for (int j=i; j<=N; j++)
  {
   int sum = 0;
   for (int k=i; k<=j; k++)
    sum += L[k-1];
   if (sum > maxSum)
    maxSum = sum;
  }
 }
 return maxSum;
}//TOTALE: N³ + 8N² + 13N + 6  OE   ovvero O(N³)
```

Un'idea più "furba"

- Input
  - Una lista $L = (a_1, a_2, \ldots, a_N)$ di N interi (positivi e negativi)
- Output

$$S = \max\left(\left\{\sum_{k=i}^{j} a_k \mid 1 \le i \le j \le N\right\}\right)$$

1. Fissiamo l'indice i
2. Facciamo variare l'indice k da i a N
   - Sommiamo i primi k elementi incrementando la somma parziale calcolata al passo k - 1
   - Confrontiamo la nuova somma parziale con il massimo corrente

Algoritmo 2

```
int computeS_2 (int L[], int N)
{
 int maxSum = 0;
 for (int i=1; i<=N; i++)
 {
  int sum = 0;
  for (int k=i; k<=N; k++)
  {
   sum += L[k-1];
   if (sum > maxSum)
    maxSum = sum;
  }
 }
 return maxSum;
}
```

Algoritmo 2

```
int computeS_2 (int L[], int N)
{
 int maxSum = 0; // 2 OE
 for (int i=1; i<=N; i++)
 {
  int sum = 0;
  for (int k=i; k<=N; k++)
  {
   sum += L[k-1];
   if (sum > maxSum)
    maxSum = sum;
  }
 }
 return maxSum;
}
```

Algoritmo 2

```
int computeS_2 (int L[], int N)
{
 int maxSum = 0;
 for (int i=1; i<=N; i++) // 3(N+1) OE
 {
  int sum = 0;
  for (int k=i; k<=N; k++)
  {
   sum += L[k-1];
   if (sum > maxSum)
    maxSum = sum;
  }
 }
 return maxSum;
}
```

```
int i=1; // 2 OE
i<=N; // N+1 OE
i++; // 2N OE
```

Algoritmo 2

```
int computeS_2 (int L[], int N)
{
 int maxSum = 0;
 for (int i=1; i<=N; i++)
 {
  int sum = 0; // 2N OE
  for (int k=i; k<=N; k++)
  {
   sum += L[k-1];
   if (sum > maxSum)
    maxSum = sum;
  }
 }
 return maxSum;
}
```

Algoritmo 2

```
int computeS_2 (int L[], int N)
{
 int maxSum = 0;
 for (int i=1; i<=N; i++)
 {
  int sum = 0;
  for (int k=i; k<=N; k++) // 3N(N+3)/2 OE
  {
   sum += L[k-1];
   if (sum > maxSum)
    maxSum = sum;
  }
 }
 return maxSum;
}
```

```
int k=i; // 2N OE
k<=N; // N(N+1)/2 + N OE
k++; // 2N(N+1)/2 OE
```

$$N+(N-1)+(N-2)+\ldots+2+1=\sum_{i=1}^{N}(N-i+1)=\frac{N(N+1)}{2}$$

Algoritmo 2

```
int computeS_2 (int L[], int N)
{
 int maxSum = 0;
 for (int i=1; i<=N; i++)
 {
  int sum = 0;
  for (int k=i; k<=N; k++)
  {
   sum += L[k-1]; // 3N(N+3)/2 OE
   if (sum > maxSum) // N(N+3)/2 OE
    maxSum = sum; // N(N+3)/2 OE  (at most!)
  }
 }
 return maxSum;
}
```

$$N+(N-1)+(N-2)+\ldots+2+1=\sum_{i=1}^{N}(N-i+1)=\frac{N(N+1)}{2}$$

Algoritmo 2

```
int computeS_2 (int L[], int N)
{
 int maxSum = 0;
 for (int i=1; i<=N; i++)
 {
  int sum = 0;
  for (int k=i; k<=N; k++)
  {
   sum += L[k-1];
   if (sum > maxSum)
    maxSum = sum;
  }
 }
 return maxSum; // 1 OE
}
```

Algoritmo 2

```
int computeS_2 (int L[], int N)
{
 int maxSum = 0;
 for (int i=1; i<=N; i++)
 {
  int sum = 0;
  for (int k=i; k<=N; k++)
  {
   sum += L[k-1];
   if (sum > maxSum)
    maxSum = sum;
  }
 }
 return maxSum;
} //TOTALE: 4N² + 17N + 6  OE quindi O(N²)
```

Tempi di esecuzione

Algoritmo	Tempo di Esecuzione (n° OE)
computeS_1	$N^3 + 8N^2 + 13N + 6$
computeS_2	$4N^2 + 17N + 6$

Algoritmo	N=1	N=2	N=4	N=8	N=16	N=32	N=64	N=128
computeS_1	28	72	250	1134	6358	41382	295750	2229894
computeS_2	27	56	138	398	1302	4646	17478	67718

Tempi di esecuzione

Algoritmo	Tempo di Esecuzione (n° OE)	Complessità
computeS_1	$N^3 + 8N^2 + 13N + 6$	$O(N^3)$
computeS_2	$4N^2 + 17N + 6$	$O(N^2)$

Caso Migliore e Caso Peggiore

- Il numero di istruzioni da eseguire può variare significativamente, anche su input della stessa dimensione, a seconda della “forma” dell’input.

- Nel caso del *Bubble Sort*, cambia molto se il vettore in input è già ordinato oppure molto “disordinato” (ordinato al contrario).

➔ Analizziamo Caso Migliore e **Caso Peggiore**

Bubble sort algorithm

```
void bubbleSort(int L[], int N) {
  bool ordinato = false;
  int steps = 0;
  while(!ordinato) {
    ordinato = true;
    int k = N – steps;
    steps++;
    for(int i = 1; i < k ; i++)
      if(L[i-1] > L[i])
      {
        ordinato = false;
        int temp = L[i-1];
        L[i-1] = L[i];
        L[i] = temp;
      }
  }
}
```

Complessità nel “caso migliore”

- Il numero di passi dipende dalla forma dell’input
  - Caso migliore: vettore già ordinato

```
void bubbleSort(int L[], int N) {
  bool ordinato = false; // 2 OE
  int steps = 0; // 2 OE
  while(!ordinato) { // 2 OE (entriamo 1 sola volta)
    ordinato = true; // 2 OE
    int k = N – steps; // 3 OE
    steps++; // 2 OE
    for(int i = 1; i < k ; i++)
      if(L[i-1] > L[i]) {
        ordinato = false;
        int temp = L[i-1];
        L[i-1] = L[i];
        L[i] = temp;
      }
  }
}
```

Complessità nel “caso migliore”

- Il num di passi dipende dalla forma dell’input
  - Caso migliore: vettore già ordinato

```
void bubbleSort(int L[], int N) {
  bool ordinato = false;
  int steps = 0;
  while(!ordinato) {
    ordinato = true;
    int k = N – steps;
    steps++;
    for(int i = 1; i < k ; i++) // 3N OE
      if(L[i-1] > L[i]) {
        ordinato = false;
        int temp = L[i-1];
        L[i-1] = L[i];
        L[i] = temp;
      }
  }
}
```

```
int i=1; // 2 OE
i<k; // N OE
i++; // 2(N-1) OE
```

Complessità nel “caso migliore”

- Il num di passi dipende dalla forma dell’input
  - Caso migliore: vettore già ordinato

```
void bubbleSort(int L[], int N) {
  bool ordinato = false;
  int steps = 0;
  while(!ordinato) {
    ordinato = true;
    int k = N – steps;
    steps++;
    for(int i = 1; i < k ; i++)
      if(L[i-1] > L[i]) { // 2(N-1) OE
        ordinato = false; // nessuna OE
        int temp = L[i-1]; // nessuna OE
        L[i-1] = L[i]; // nessuna OE
        L[i] = temp; // nessuna OE
      }
  }
}
```

Complessità nel “caso migliore”

- Il num di passi dipende dalla forma dell’input
  - Caso migliore: vettore già ordinato

```
void bubbleSort(int L[], int N) {
  bool ordinato = false;
  int steps = 0;
  while(!ordinato) {
    ordinato = true;
    int k = N – steps;
    steps++;
    for(int i = 1; i < k ; i++)
      if(L[i-1] > L[i]) {
        ordinato = false;
        int temp = L[i-1];
        L[i-1] = L[i];
        L[i] = temp;
      }
  }
} // TOTALE: 5N + 11 OE    ovvero O(N)
```

Complessità nel “caso peggiore”

- Il num di passi dipende dalla forma dell’input
  - Caso peggiore: vettore ordinato al contrario

```
void bubbleSort(int L[], int N) {
  bool ordinato = false; // 2 OE
  int steps = 0; // 2 OE
  while(!ordinato) { // N OE (entriamo N-1 volte)
    ordinato = true; // 2(N-1) OE
    int k = N – steps; // 3(N-1) OE
    steps++; // 2(N-1) OE
    for(int i = 1; i < k ; i++)
      if(L[i-1] > L[i]) {
        ordinato = false;
        int temp = L[i-1];
        L[i-1] = L[i];
        L[i] = temp;
      }
  }
}
```

Complessità nel “caso peggiore”

- Il num di passi dipende dalla forma dell’input
  - Caso peggiore: vettore ordinato al contrario

```
void bubbleSort(int L[], int N) {
  bool ordinato = false;
  int steps = 0;
  while(!ordinato) {
    ordinato = true;
    int k = N – steps;
    steps++;
    for(int i = 1; i < k ; i++) // 3(N² + N - 2)/2 OE
      if(L[i-1] > L[i]) {
        ordinato = false;
        int temp = L[i-1];
        L[i-1] = L[i];
        L[i] = temp;
      }
  }
}
```

```
int i=1; // 2*(N-1) OE
i<k; // N(N-1)/2 + (N-1) OE
i++; // 2*N(N-1)/2 OE
```

$$(N-1)+(N-2)+\ldots+2+1=\sum_{i=1}^{N}(k-1)=\frac{N(N-1)}{2}$$

Complessità nel “caso peggiore”

- Il num di passi dipende dalla forma dell’input
  - Caso peggiore: vettore ordinato al contrario

```
void bubbleSort(int L[], int N) {
  bool ordinato = false;
  int steps = 0;
  while(!ordinato) {
    ordinato = true;
    int k = N – steps;
    steps++;
    for(int i = 1; i < k ; i++)
      if(L[i-1] > L[i]) { // 2N(N-1)/2 OE
        ordinato = false; // N(N-1)/2 OE
        int temp = L[i-1]; // 3N(N-1)/2 OE
        L[i-1] = L[i]; // 2N(N-1)/2 OE
        L[i] = temp; // N(N-1)/2 OE
      }
  }
}
```

$$(N-1)+(N-2)+\ldots+2+1=\sum_{i=1}^{N}(k-1)=\frac{N(N-1)}{2}$$

Complessità nel “caso peggiore”

- Il num di passi dipende dalla forma dell’input
  - Caso peggiore: vettore ordinato al contrario

```
void bubbleSort(int L[], int N) {
  bool ordinato = false;
  int steps = 0;
  while(!ordinato) {
    ordinato = true;
    int k = N – steps;
    steps++;
    for(int i = 1; i < k ; i++)
      if(L[i-1] > L[i]) {
        ordinato = false;
        int temp = L[i-1];
        L[i-1] = L[i];
        L[i] = temp;
      }
  }
} // TOTALE: 6N² + 5N – 6 OE   ovvero O(N²)
```

Conclusioni

- A meno di essere particolarmente “fortunati” … non ci imbatteremo quasi mai nel caso migliore!
- L’analisi del caso peggiore ci garantisce un tempo massimo entro cui il programma comunque termina
- Ciò motiva l’assunzione di considerare solo l’*analisi asintotica del caso peggiore*.

- Ordinamento
  - *Bubblesort*: $\mathcal{O}(\mathrm{N}^2)$
  - *Mergesort*: $\mathcal{O}(\mathrm{N}\ \log_2 \mathrm{N})$

- Tipicamente Bubblesort è usato per non più di 100 elementi.