Programmazione I

Il Linguaggio C

Ordinamento

Daniel Riccio

Università di Napoli, Federico II

29 novembre 2021

Sommario

- Argomenti
  - Ordinamento
  - Selection Sort
  - Insertion Sort
  - Bubble Sort

Esercizi

Dato il vettore di interi V[]={34, 12, 65, 21, 89, 3}, si mostrino le diverse configurazioni intermedie del vettore prodotte dall'insertion sort:

t_0: V[] = {34, 12, 65, 21, 89, 3}
t_1: V[] = {12, 34, 65, 21, 89, 3}
t_2: V[] = {12, 34, 65, 21, 89, 3}
t_3: V[] = {12, 21, 34, 65, 89, 3}
t_4: V[] = {12, 21, 34, 65, 89, 3}
t_5: V[] = {3, 12, 21, 34, 65, 89}

Esercizi

Dato il vettore di interi V[]={34, 12, 65, 21, 89, 3}, si mostrino le diverse configurazioni intermedie del vettore prodotte dal selection sort:

t_0: V[] = {34, 12, 65, 21, 89, 3}
t_1: V[] = {3, 12, 65, 21, 89, 34}
t_2: V[] = {3, 12, 65, 21, 89, 34}
t_3: V[] = {3, 12, 21, 65, 89, 34}
t_4: V[] = {3, 12, 21, 34, 89, 65}
t_5: V[] = {3, 12, 21, 34, 65, 89}

Esercizi

Dato il vettore di interi V[]={34, 12, 65, 21, 89, 3}, si mostrino le diverse configurazioni intermedie del vettore prodotte dal bubble sort:

t_0: V[] = { 34, 12, 65, 21, 89, 3 }
t_1: V[] = { 12, 34, 65, 21, 89, 3 }
t_2: V[] = { 12, 34, 65, 21, 89, 3 }
t_3: V[] = { 12, 34, 21, 65, 89, 3 }
t_4: V[] = { 12, 34, 21, 65, 89, 3 }
t_5: V[] = { 12, 34, 21, 65, 3, 89 }
t_6: V[] = { 12, 34, 21, 65, 3, 89 }
t_7: V[] = { 12, 21, 34, 65, 3, 89 }
t_8: V[] = { 12, 21, 34, 65, 3, 89 }
t_9: V[] = { 12, 21, 34, 3, 65, 89 }
t_{10}: V[] = { 12, 21, 34, 3, 65, 89 }
t_{11}: V[] = { 12, 21, 34, 3, 65, 89 }
t_{12}: V[] = { 12, 21, 3, 34, 65, 89 }
t_{13}: V[] = { 12, 21, 3, 34, 65, 89 }
t_{14}: V[] = { 12, 3, 21, 34, 65, 89 }
t_{15}: V[] = { 3, 12, 21, 34, 65, 89 }

Problemi di ordinamento

Supponiamo di avere una enorme mole di dati rappresentati da interi (es. codici identificativi di persone). Inoltre, si accede ai dati solo in lettura (i.e. non si aggiungono o cancellano valori) mediante operazioni di ricerca.

Supponiamo di effettuare **M** ricerche su **N** valori

Ricerca Lineare

Numero di operazioni: O(**M** × **N**)

Cerchiamo una soluzione
che richieda al più O(**N** + **M** × log(**N**))

M, **N**	**M** × **N**	**N** + **M** × log(**N**)
M=10, **N**=16	160	56
M=100, **N**=256	25 600	1 056
M=1000 **N**=4096	4 096 000	16 096
M=10000 **N**=65536	655 360 000	225 536

Ordinamento più ricerca binaria

Numero di operazioni:

- Ordinamento O(**N**)
- Ricerca binaria O(**M** × log(**N**)) → **M** ricerche binarie ciascuna di costo log(**N**)

Counting sort

Counting Sort, si basa sull'ipotesi che ognuno degli **n** elementi in input sia un intero nell'intervallo da 1 a **k** ove k è un numero non troppo grande.

Si determina, per ogni elemento **x** in input, il numero di elementi minori di **x**. Questa informazione può essere usata per porre l'elemento **x**, direttamente nella sua esatta posizione nell'array di output. Per es., se vi sono 10 elementi minori di **x**, **x** va messo in undicesima posizione nell'array di output.

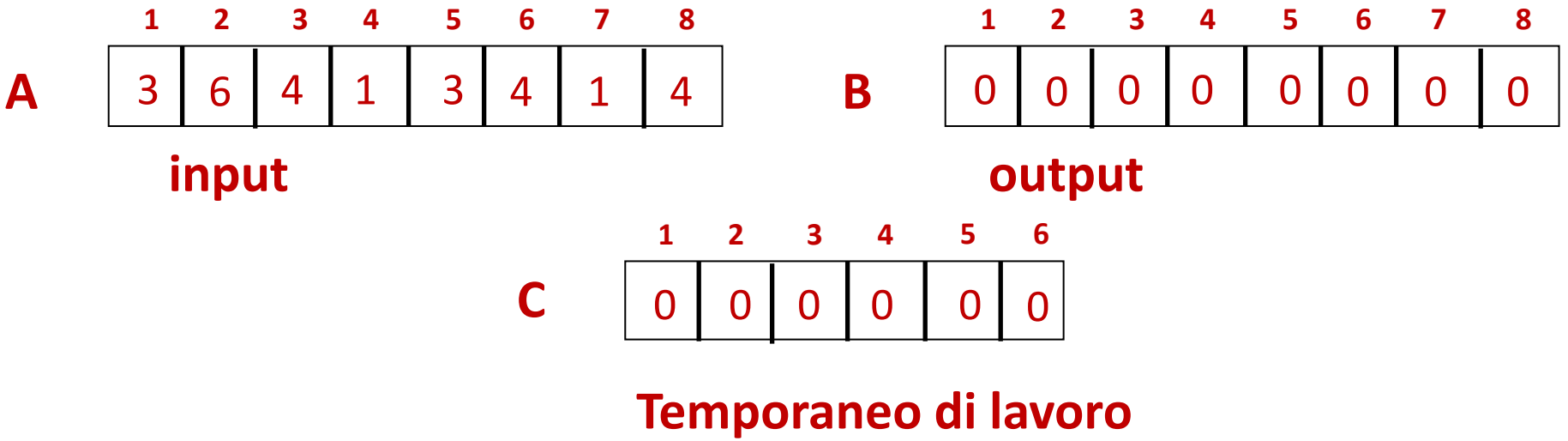
L'algoritmo deve gestire situazioni in cui alcuni elementi abbiano lo stesso valore, infatti non si vuole metterli tutti nella stessa posizione nell'array di output.

Counting sort

- È veloce perché fa delle ipotesi sull'input, infatti, assume che l'input consista di numeri interi in un piccolo intervallo.
- Stabile
- Risorse occorrenti: 3 array
  - array **A**: array di dimensione **n**, contenente gli **n** numeri da ordinare:
  - array **B**: array di dimensione **n**, contenente la sequenza degli **n** numeri ordinata
  - Array **C**: array di dimensione **k**, temporaneo di lavoro, dove **k** è il massimo numero trovato in **A**

Counting sort

Esecuzione di Counting Sort su un array di input A[8], dove ogni elemento di A é un intero positivo non più grande di k=6.



Si dimensiona l'array C, con il massimo numero "k" in A

Counting sort

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	4	4

input

	1	2	3	4	5	6
C	0	0	1	0	0	0

	1	2	3	4	5	6
C	0	0	1	0	0	1

	1	2	3	4	5	6
C	0	0	1	1	0	1

	1	2	3	4	5	6
C	1	0	1	1	0	1

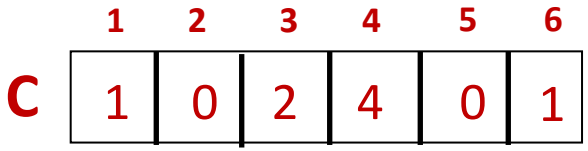
	1	2	3	4	5	6
C	1	0	2	1	0	1

	1	2	3	4	5	6
C	1	0	2	2	0	1

	1	2	3	4	5	6
C	1	0	2	3	0	1

	1	2	3	4	5	6
C	1	0	2	4	0	1

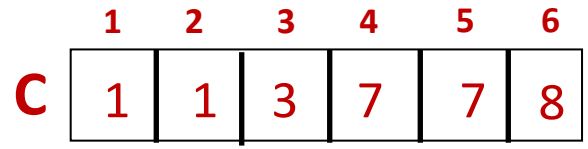
Counting sort



prima

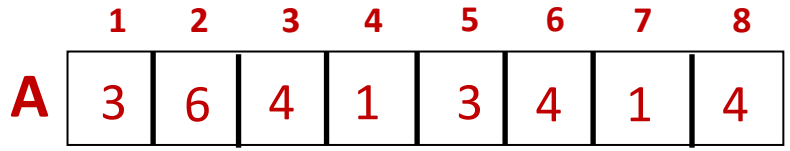
Si determina per ciascun i = 1,2,…,k, quanti elementi dell'input sono minori o uguali di i. Questo viene fatto determinando la somma cumulata degli elementi dell'array C[ ].

```
For x = 2 to 6
   C[i] = C[i] + C[i-1]
```

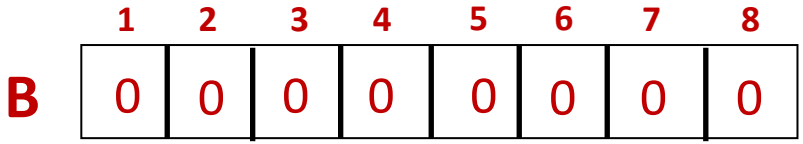


dopo

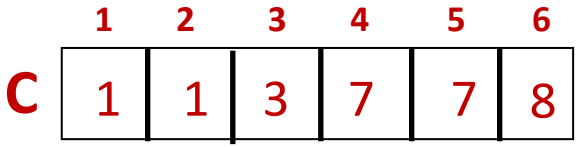
Counting sort



input

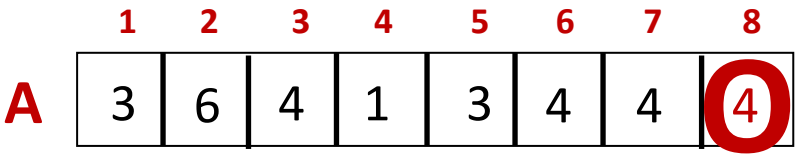


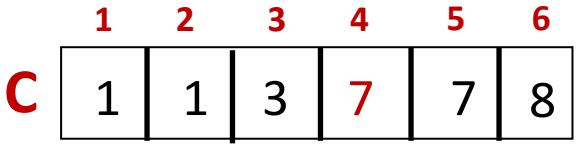
output

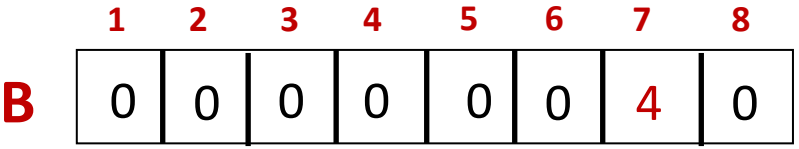


Counting sort

Iterazione 1







Partendo dall'ultima posizione di A, si prende il numero contenuto in A [8] = 4.

Guardiamo cosa è contenuto in C [4] = 7.

Sistemiamo il 4 in B [7]

Counting sort

Iterazione 2

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	4	4

	1	2	3	4	5	6
C	1	1	3	7	7	8

? Qualcosa non funziona, così facendo perdiamo un dato

	1	2	3	4	5	6	7	8
B	0	0	0	0	0	0	4	0

Dobbiamo gestire il caso in cui gli elementi contenuti in A, non siano tutti distinti.

Counting sort

Iterazione 2

Per questo, decrementiamo C[4], in modo che alla prossima iterazione, nel caso di elementi non distinti, il numero si inserisca nell'array di output, nella sua corretta posizione.

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	4	4

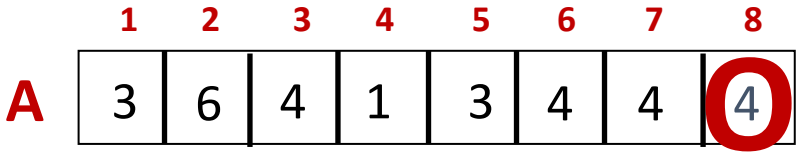
	1	2	3	4	5	6	7	8
B	0	0	0	0	0	0	4	0

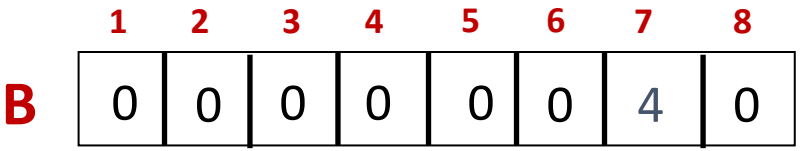
		1	2	3	4	5	6
Prima	**C**	1	1	3	7	7	8

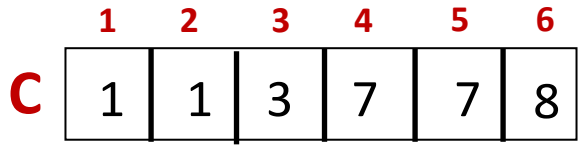
		1	2	3	4	5	6
Dopo	**C**	1	1	3	6	7	8

Counting sort

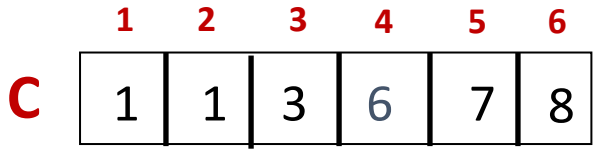
Iterazione 1







Dopo



Partendo dall'ultima posizione di A, si prende il numero contenuto in A [8] = 4

Guardiamo cosa è contenuto in C [4] = 7

Sistemiamo il 4 in B [7]

Decrementiamo C [4]

Counting sort

Iterazione 2

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	4	4

	1	2	3	4	5	6	7	8
B	0	0	0	0	0	4	4	0

	1	2	3	4	5	6
C	1	1	3	6	7	8

Dopo

	1	2	3	4	5	6
C	1	1	3	5	7	8

Penultima posizione di A, si prende il numero contenuto in A [7] = 4

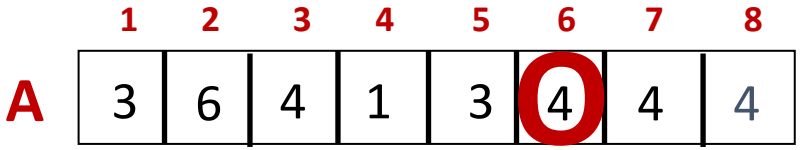
Guardiamo cosa è contenuto in C [4] = 6

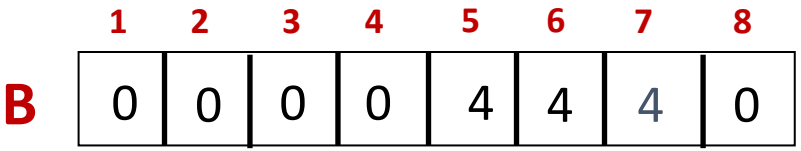
Sistemiamo il 4 in B [6]

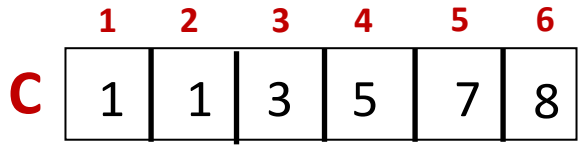
Decrementiamo C [4]

Counting sort

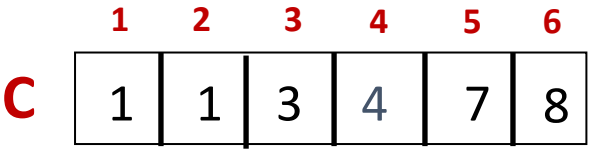
Iterazione 3







Dopo



Terzultima posizione di A, si prende il numero contenuto in A [6] = 4

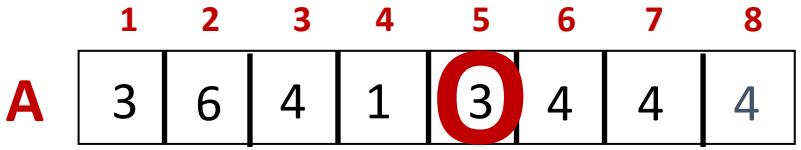
Guardiamo cosa è contenuto in C [4] = 5

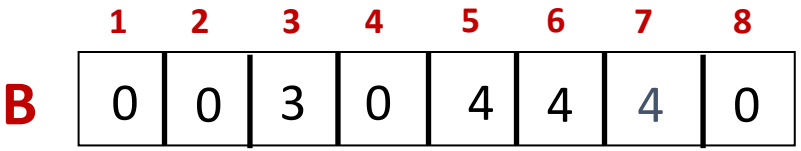
Sistemiamo il 4 in B [5]

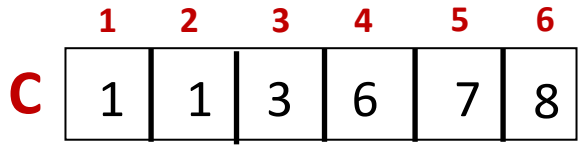
Decrementiamo C [4]

Counting sort

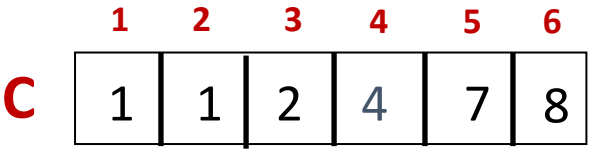
Iterazione 4







Dopo



Quartultima posizione di A, si prende il numero contenuto in A [5] = 3

Guardiamo cosa è contenuto in C [3] = 3

Sistemiamo il 3 in B [3]

Decrementiamo C [3]

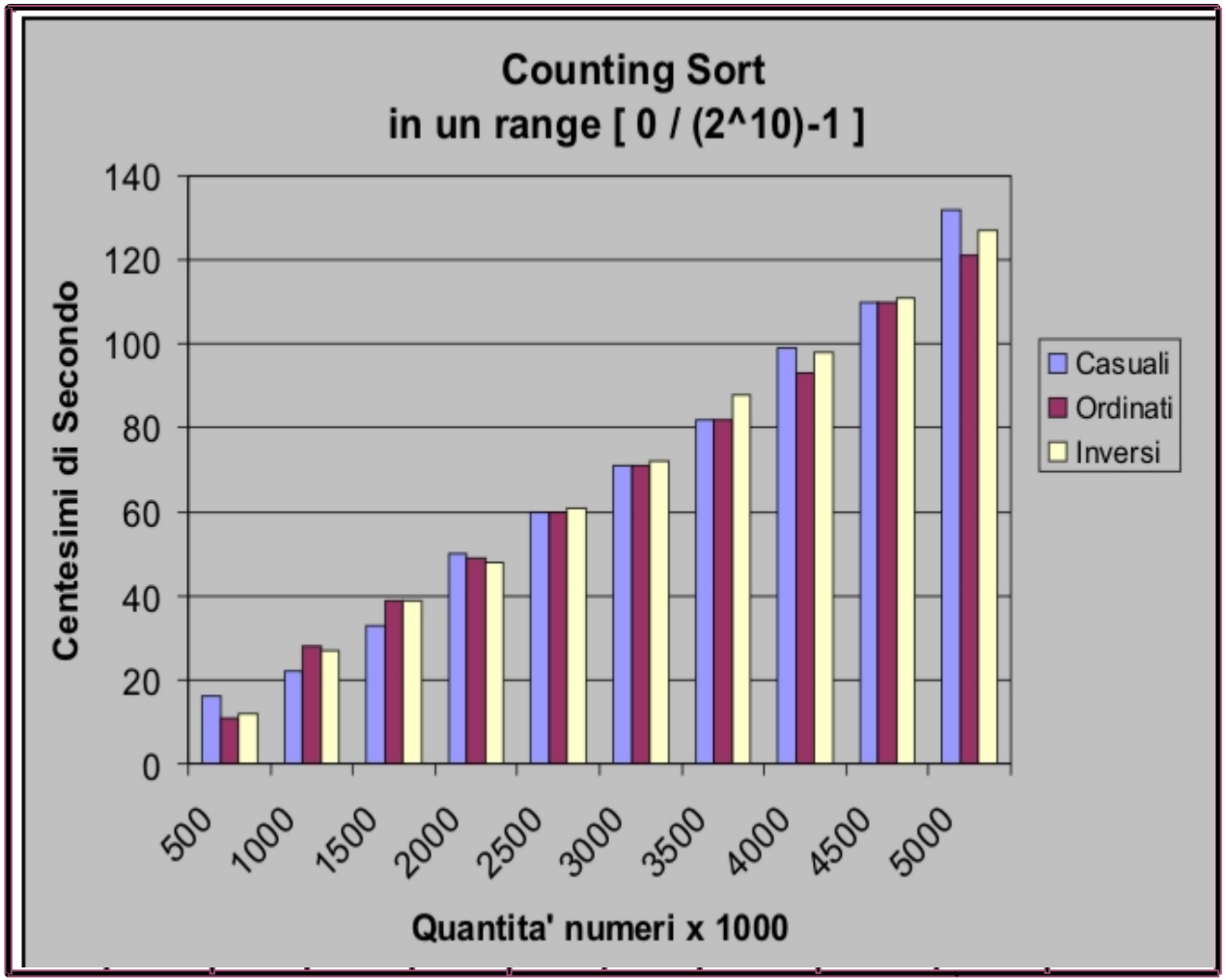
Counting sort – indipendenza dalla distribuzione

Ordine dei numeri

CASUALE

CRESCENTE

DECRESCENTE



time

L'ordine dei numeri non influisce sulle prestazioni dell'algoritmo.

Counting sort – indipendenza dalla distribuzione

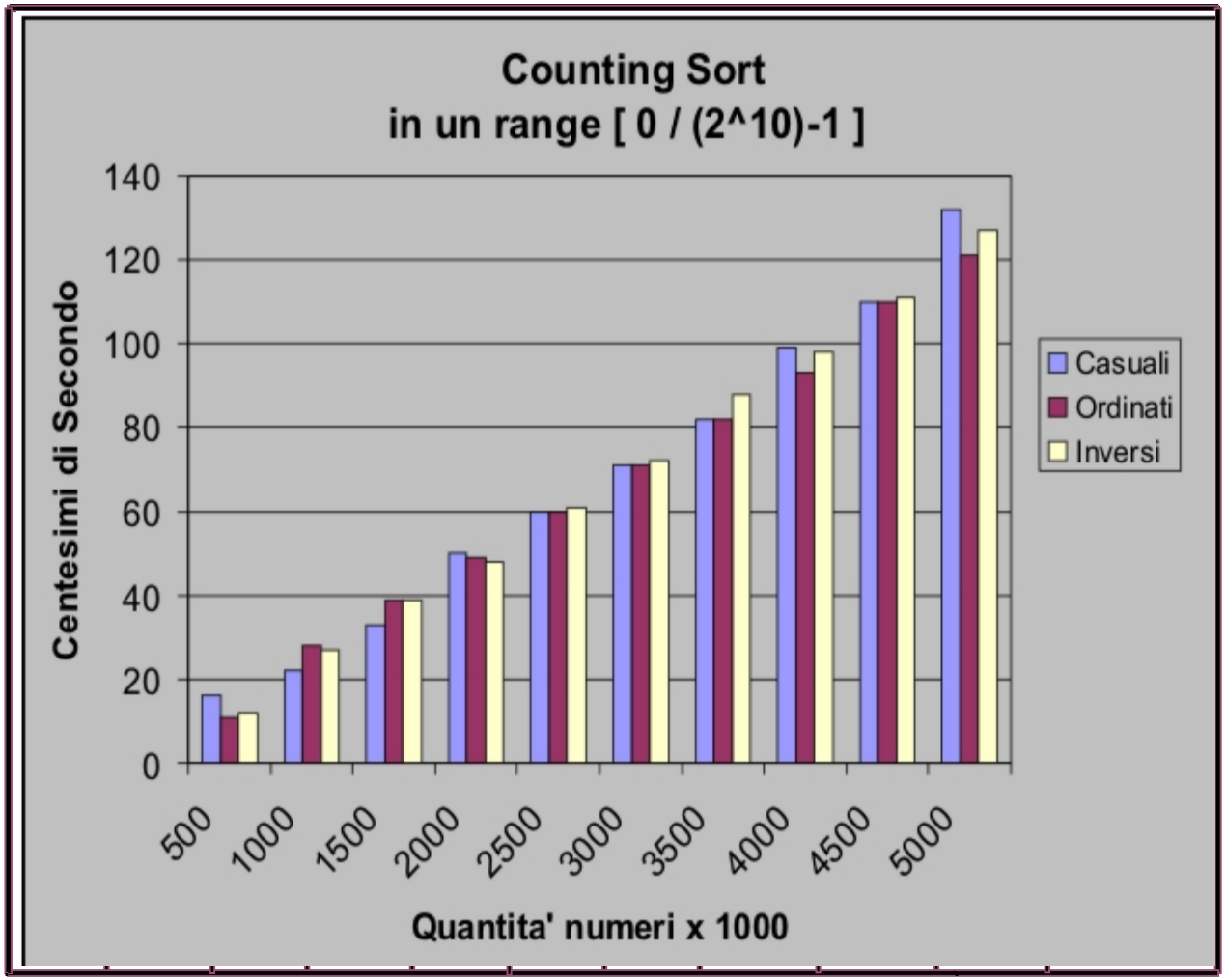
Ordine dei numeri

CASUALE

CRESCENTE

DECRESCENTE

Il tempo cresce in modo lineare, all'aumentare della dimensione n dei dati in ingresso.



time

Counting sort

```c
#include <stdio.h>
#include <stdlib.h>

void CountingSort(int A[], int n);

int V[] = {34, 12, 65, 21, 89, 3};

int main(int argc, char *argv[])
{
   int i=0;
   int n=6;

   printf("Vettore non ordinato: ");
   for(i=0; i<n; i++)
      printf("%d ", V[i]);

   CountingSort(V, n);

   printf("\nVettore ordinato: ");
   for(i=0; i<n; i++)
      printf("%d ", V[i]);

   return 0;
}
```

Counting sort

```
void CountingSort(int A[], int n)
{
   int i=0;
   int *C=NULL;
   int *B=NULL;
   int max=0;

   for(i=0; i<n; i++)
      if(A[i]>max)
         max = A[i];

   printf("Il massimo è: %d", max);

   B = (int *)calloc(n, sizeof(int));
   if(!B){
      printf("memoria insufficiente\n");
      return;
   }

   C = (int *)calloc(max+1, sizeof(int));
   if(!C){
      free(B);
      printf("memoria insufficiente\n");
      return;
   }

   // riempiamo il vettore C
   for(i=0; i<n; i++)
      ++C[A[i]];

   // accumuliamo le posizioni in C
   for(i=1; i<=max; i++)
      C[i] = C[i] + C[i-1];

   // ordiniamo costruendo B
   for(i=n-1; i>=0; i--){
      B[C[A[i]]-1] = A[i];
      C[A[i]] = C[A[i]]-1;
   }

   for(i=0; i<n; i++)
      A[i] = B[i];

      free(B);
      free(C);
   }
```

Esercizio

Scrivere una versione generalizzata di **BubbleSort** ed applicarlo per ordinare un vettore di double

Generalizzazione di BubbleSort

Supponiamo di voler utilizzare la stessa funzione di ordinamento (es. **BubbleSort**), senza doverla riscrivere completamente per diversi tipi di dato.

Problemi:

1) Bisogna specificare il tipo dei parametri nella firma della funzione;
2) Bisogna effettuare i confronti all'interno della funzione;
3) Bisogna scambiare gli elementi nel vettore.

Generalizzazione di BubbleSort

Bisogna specificare il tipo dei parametri nella firma della funzione

```
void BubbleSort(int A[], int n);
void BubbleSort(double A[], int n);
```

Soluzione:

Definiamo un nuovo tipo di dato **item** e utilizziamo questo tipo per l'implementazione della funzione

```
typedef double item;

void BubbleSort(item A[], int n);
```

Generalizzazione di BubbleSort

1) Bisogna effettuare i confronti all'interno della funzione

```
if(V[j] > V[j+1])
    Swap(&V[j],&V[j+1]);
```

Soluzione:
Implementiamo il confronto con una funzione separata e la passiamo come parametro alla funzione **BubbleSort** (puntatore a funzione)

```
int confronta(item a, item b){

   if(a > b)
      return 1;
   else
      return 0;
}

void BubbleSort(item A[], int n, int (*fpc)(item , item ));
```

```
int (*fpc)(item , item );
```

Generalizzazione di BubbleSort

1) Bisogna scambiare gli elementi nel vettore

```
Swap(&V[j],&V[j+1]);
```

Soluzione:

Implementiamo lo scambio con una funzione separata e la passiamo come parametro alla funzione **BubbleSort** (puntatore a funzione)

```
void Swap(item *a, item *b)
{
   item temp;

   temp = *a;
   *a = *b;
   *b = temp;
}
```

```
void (*fps)(item *, item *);
```

```
void BubbleSort(item V[], int n, int (*fpc)(item , item ),
                void (*fps)(item *, item *))
```

Generalizzazione di BubbleSort

```
void BubbleSort(int V[], int n) {
   int i,j,q;

   for(i=1;i<n;i++) {

      for(j=0;j<n-i;j++) {
         if(V[j]>V[j+1]) {
            Swap(&V[j],&V[j+1]);
         }
      }
   }
}
```

```
void BubbleSort(item V[], int n, int (*fpc)(item , item ),
                void (*fps)(item *, item *)) {
   int i,j,q;

   for(i=1;i<n;i++) {

      for(j=0;j<n-i;j++) {
         if(fpc(V[j],V[j+1])) {
            fps(&V[j],&V[j+1]);
         }
      }
   }
}
```

Esempio con record

Supponiamo di voler utilizzare funzione **BubbleSort** per ordinare record di tipo **Persona**

```
#include <stdio.h>
#include <string.h>

typedef struct {
    char nome[256];
    char cognome[256];
    int eta;
} Persona;

typedef Persona item;
```

Esempio con record

Prototipi delle funzioni:

```
void BubbleSort(item A[], int n, int (*fp)(item , item ),
                void (*fps)(item *, item *));

int confronta(item a, item b);
void Swap(item *a, item *b);
void StampaItem(item p);


item V[4]={
   {"Carlo  ", "Rossi  ", 25},
   {"Luisa  ", "Bianchi", 28},
   {"Gennaro", "Conte  ", 40},
   {"Anna   ", "De Caro", 21}
};
```

Esempio con record

```
void StampaItem(item p)
{
   printf("\nNome: %s", p.nome);
   printf("\nCognome: %s", p.cognome);
   printf("\nEta: %d", p.eta);
   printf("\n\n");
}
```

```
int confronta(item a, item b){

   if(a.eta > b.eta)
      return 1;
   else
      return 0;
}
```

```
void Swap(item *a, item *b)
{
   item temp;

   strcpy(temp.nome, a->nome);
   strcpy(temp.cognome, a->cognome);
   temp.eta = a->eta;

   strcpy(a->nome, b->nome);
   strcpy(a->cognome, b->cognome);
   a->eta = b->eta;

   strcpy(b->nome, temp.nome);
   strcpy(b->cognome, temp.cognome);
   b->eta = temp.eta;
}
```

Esercizio

Scrivere una versione generalizzata di **SelectionSort** ed applicarlo per ordinare un vettore di stringhe

Challenge n. 1

Si scriva una funzione in C, che presi in input un array di stringhe **Testo** ed il numero di stringhe da ordinare (un intero **n**), produca in **Testo** un elenco di stringhe in ordine lessicografico crescente.

Input	**Output**
Testo	Testo
“uno”	“cinque”
“due”	“due”
“tre”	“quattro”
“quattro”	“tre”
“cinque”	“uno”

Iscrizione e Sottomissione

La partecipazione alla challenge in oggetto richiede una fase di registrazione, in cui il candidato invia una email al docente (daniel.riccio@unina.it) con le seguenti informazioni:

Cognome, Nome, Matricola.

Ciascun partecipante deve scrivere il codice della sola funzione ***OrdinaStringhe***, il cui prototipo deve necessariamente essere il seguente:

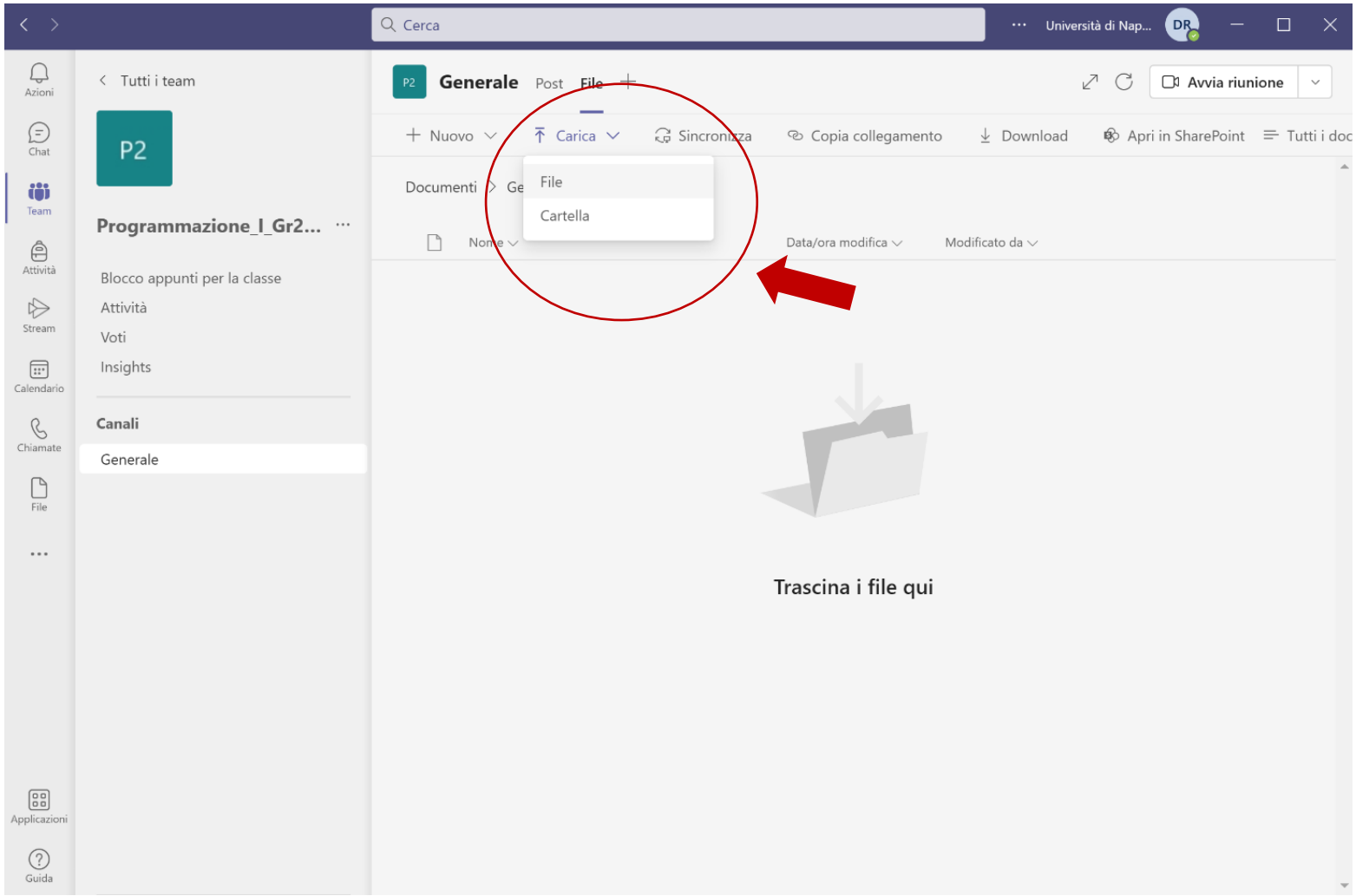
```
void OrdinaStringhe(unsigned char Testo[N][CHR_L], int n);
```

La funzione prende in input un array di **N** stringhe di lunghezza **CHR_L**, dove **N** e **CHR_L** sono due costanti definite mediante la direttiva al preprocessore **#define**, e un intero **n**, che indica il numero di stringhe effettivamente presente nell'array **Testo**.

Il codice deve essere inserito in un file con il nome: Cognome_Nome_Matricola.txt.

Il file deve essere caricato nella cartella **Challenge_n_1** della sezione **File** del canale **Generale** del gruppo Teams **Programmazione_I_Gr2**, come mostrato in figura.

Iscrizione e Sottomissione



Testing e Risultati

Il codice viene eseguito 100 volte sullo stesso input. Ciascuna esecuzione è cronometrata e la media dei tempi di esecuzione è calcolata solo sulle ultime 50 esecuzioni, per garantire che il tempo non sia influenzato dallo stato della macchina o dalle latenze del sistema su cui il programma viene eseguito.

Il codice seguente può essere utilizzato dai candidati per misurare il tempo di una singola esecuzione:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

#define N 5
#define MAX_ITER 100
#define CHR_L 20

unsigned char Testo_in[N][CHR_L]={“uno”, “due”, “tre”,
“quattro”, “cinque”};

void OrdinaStringhe(unsigned char Testo[N][CHR_L], int n);

int main()
{
    int i=0, j=0;
    int check = 1;
    clock_t start, end;
    double cpu_time_used, avg_time_used;

    start = clock();
    OrdinaStringhe(Testo, 5);
    end = clock();
    cpu_time_used = ((double) (end - start)) /
CLOCKS_PER_SEC;
    printf("Tempo impiegato: %f sec\n",
cpu_time_used);

    // Qui verrà effettuata la verifica che l’elenco
sia ordinato correttamente
    //    -> check =1,
    // oppure no
    //    -> check = 0;

    if(check==0)
        printf("Correctness Test not passed\n");
    else
        printf("Correctness Test passed with Average
Time: %f\n", …);

    return 0;
}

void OrdinaStringhe(unsigned char Testo[N][CHR_L], int
n){
   …
}
```

Testing e Risultati

Periodicamente verrà aggiornato un file Classifica_Temporanea.pdf, in cui sarà riportato l'elenco delle soluzioni, con i relativi tempi di esecuzione:

Classifica Temporanea		
Posizione	**Matricola**	**Tempo medio di esecuzione**
1	N86/…	0.27 ms
2	N86/…	0.30 ms