# Esercitazione 1

Introduzione agli algoritmi di ordinamento

# Algoritmi di ordinamento

- Algoritmi utilizzati per elencare gli elementi di un insieme secondo una sequenza stabilita da una relazione d'ordine, in modo che ogni elemento sia minore (o maggiore) di quello che lo segue
- Si considereranno **array di interi** di lunghezza **n** da ordinare con diversi algoritmi
- Consideriamo array di interi inizializzati con **valori casuali**

# Generazione di numeri (pseudo)casuali

- Funzione di libreria **rand()**
  - Inclusa nella libreria **stdlib.h**
  - Genera un numero intero (pseudo)casuale compreso tra 0 e **RAND_MAX**, costante che dipende dall'implementazione
    - Es: $(2^{15}-1)$ o $(2^{31}-1)$
- ***Numeri pseudocasuali:*** fissato il primo valore della sequenza (***seme***), è fissata tutta la sequenza di valori che saranno generati nelle successive invocazioni della funzione **rand()**

# Generazione di numeri (pseudo)casuali

- I numeri sono generati da un ***algoritmo deterministico*** in modo da avere le stesse ***proprietà statistiche*** di una sequenza di numeri generata da un ***processo casuale***
- Esempio banale ($x_0$ è il seme)

$$X_{i+1} = ( a * x_i + c) ( MOD\ m )$$

a > o moltiplicatore

c >= 0 incremento

m > 0 modulo

# Inizializzazione seme

- **Per ottenere sequenze diverse, occorre cambiare il valore del seme**
- Per cambiare il valore del seme, si utilizza la funzione **srand(n)**, dove **n** è il nuovo valore (**unsigned int**) che si vuol dare al seme – ***come sceglierlo?***

***Per ottenere sequenze 'quasi' completamente casuali, si può tentare di dare al seme un valore casuale***

# Inizializzazione seme

- **Possibile inizializzazione**

  Sfruttare il valore di ritorno della funzione **time()**

  - Ritorna il numero di secondi trascorsi dal 1 gennaio, 1970, GMT
  - Includere header **time.h**
- La funzione **time()** prende in ingresso un puntatore (time_t *), che poniamo pari a 0
- Invocazione: **srand(time(0))**

# Esercizio numeri casuali

Programma *array_casuali.cc*

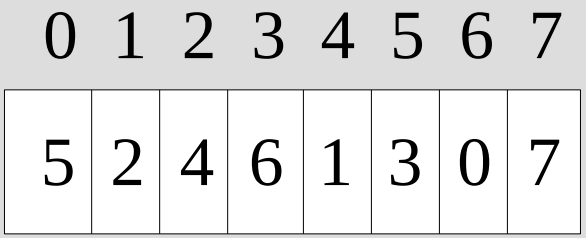
- Crea un array di interi di lunghezza decisa dall'utente a tempo di esecuzione del programma
- Inizializza l'array con valori (pseudo) casuali
- Stampa l'array su **stdout**

# Algoritmo insertion sort

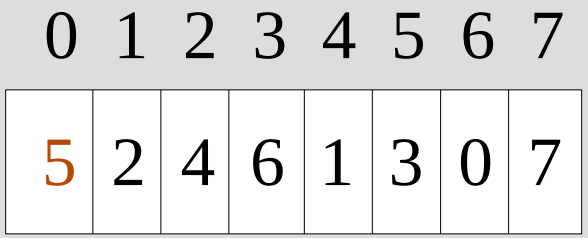
- Algoritmo ***efficiente per un piccolo numero di elementi***
- **Concetto di base**:
  - Si scandisce l'array di elementi
  - Ogni elemento viene inserito al posto giusto tra quelli precedenti (già considerati e ordinati)
  - Gli elementi vengono riordinati sul posto
- Supponiamo di ordinare in senso non descrescente

# Insertion sort: funzionamento

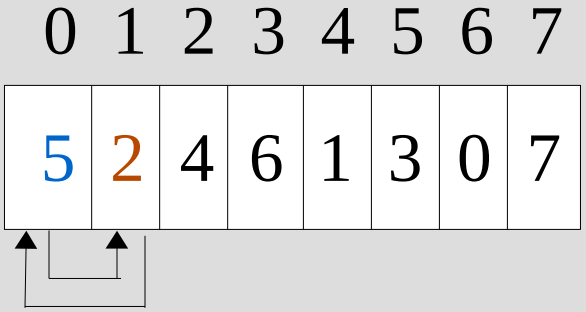
a)



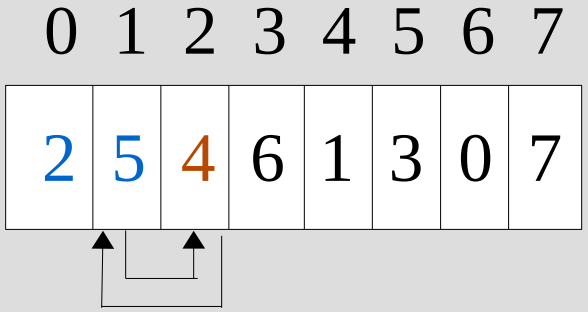

b)




c)

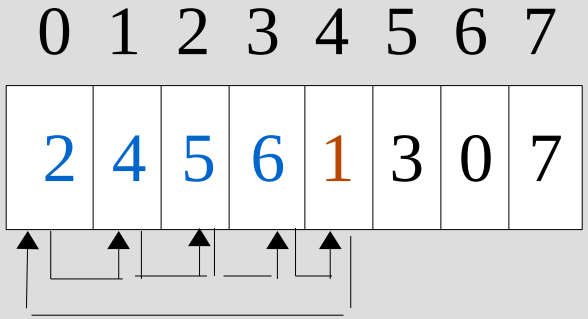



d)




e)


f)



# Insertion sort: funzionamento

- Il procedimento precedente si ripete fino ad avere l'array ordinato alla fine della scansione
- Ad ogni iterazione, l'elemento considerato viene confrontato con i valori precedenti, già ordinati
- Programma: *insertion_sort.cc*
  - Programma che ordina in senso non decrescente un array di n interi inizializzato con valori casuali

# Tempi di esecuzione

- Per paragonare diversi algoritmi, è Interessante misurare il ***costo dell'algoritmo*** in termini di ***tempi di esecuzione dell'ordinamento***
- Il paragone delle misurazioni di costo deve essere 'fair' per i diversi algoritmi
- ***Idea di base: partire dallo stesso vettore***
- **Ipotesi: partire dal caso peggiore**
  - **L'array da ordinare va inizializzato con valori decrescenti** (nel modo che preferite, purché a costo O(n))

# Come misurare i tempi di esecuzione

Uso della funzione ***clock()***:

*#include <time.h>*

- *clock_t clock( void );*
- ***clock()*** ritorna il numero di clock ticks trascorsi dall'inizio del programma, o -1 se tale informazione non è disponibile
- Per convertire il valore di ritorno in secondi, lo si divide per la costante **CLOCKS_PER_SEC**

# Come misurare i tempi di esecuzione

**Idea di base:**

- Invocare **clock()** prima dell'algoritmo di ordinamento
- Invocare **clock()** dopo l'algoritmo di ordinamento
- La differenza tra i due valori ritornati, divisa per il valore di **CLOCKS_PER_SEC dà il tempo di esecuzione dell'ordinamento**

Programma *misura_costo.cc*

# Misurare il costo di insertion sort

- Inseriamo in *insertion_sort.cc*
  - Le funzioni per la misurazione dei tempi di esecuzione
  - Il riempimento iniziale dell'array di numeri interi da ordinare con valori decrescenti
- Misuriamo il costo per n variabile
  - N = 5000 10000 20000 40000 80000 160000

# Misure

- Misuriamo i costi dell'insertion sort per diversi valori di **n** su una specifica architettura:

| n | Tempo |
|---|---|
| 5000 | 0.07 |
| 10000 | 0.26 |
| 20000 | 1.00 |
| 40000 | 3.97 |
| 80000 | 15.83 |
| 160000 | 63.50 |

# Costo computazionale di una istruzione

- Come ricavare il costo di una istruzione
- Costo per insertion sort
  O(*n*^2) => T(*n*) ≈ *c* * *n*^2
- Quindi, dato un *n_0* > 0 *sufficientemente grande*,
  *c* = T(*n_0*) / ( *n_0* ^ 2 )
- Per n=160000, c = 2.48e-009

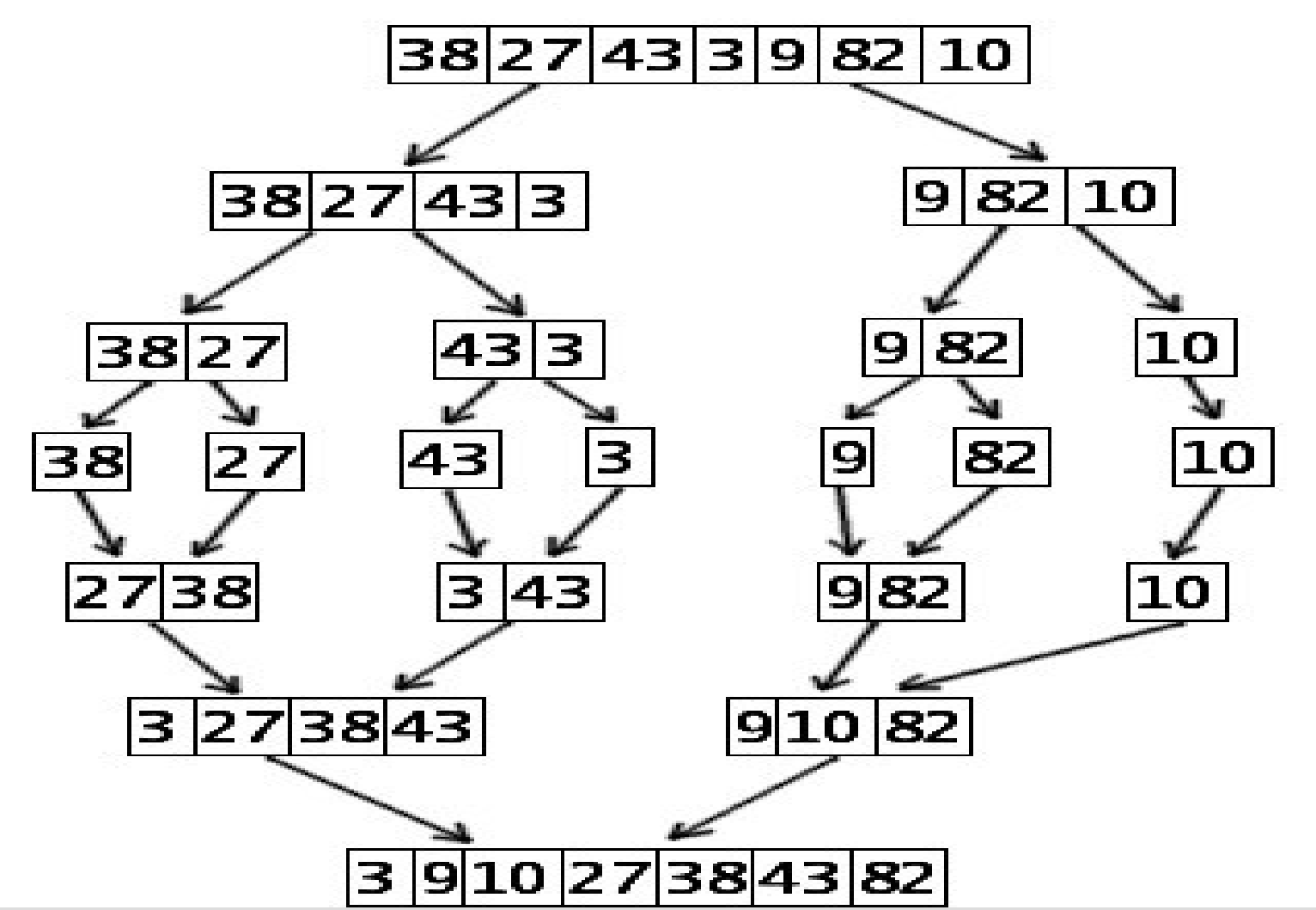
# Merge sort

- Approccio ***Divide et impera***
- ***Divide:*** la sequenza degli n elementi viene suddivisa in 2 sottosequenze di n/2 elementi ciascuna
- ***Impera:*** le 2 sottosequenze sono ordinate *in modo ricorsivo* utilizzando l'algoritmo merge sort
- ***Combina***: le 2 sottosequenze ordinate vengono fuse per generare un'unica sequenza ordinata

# Ricorsione in merge sort

- La ricorsione tocca il fondo (***condizione di terminazione***) quando la sequenza da ordinare ha lunghezza 1
  - ***Caso base*** in cui non vi è nulla da fare, ogni sequenza di lunghezza 1 è già ordinata
- ***Passo ricorsivo***: divide in 2 la sequenza e si richiama sulle 2 sottosequenze individuate

# 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

# Fusione

- Fusione di 2 sottosequenze ordinate in un'unica sequenza ordinata (passo *“**combina**”*)
- ***Operazione chiave del merge sort***
- Operazione da effettuare in **uscita dalla ricorsione** - man mano che la catena di invocazioni ricorsive innestate viene distrutta
  - Pensiamo dove posizionare questa fase nel codice ricorsivo...

# Fusione

- Per realizzarla usiamo una funzione ausiliaria

  ***fondi (int a1[], int n1, int a2[], int n2)***

  che prende in ingresso due sequenze ordinate **a1** e **a2** di lunghezza rispettiva **n1** e **n2**, e le fonde in un'unica sequenza ordinata che verrà memorizzata in **a1**

- ***NOTA: a1[] e a2[] possono essere visti come gli indirizzi di due elementi dello stesso array***

a1[]=&a1[0], n1=N/2 a2[]=&a1[N/2], n2=N-N/2

# Programma

- ***merge_sort.cc***
- Suggerimento: partire dall'implementazione della funzione

  ***fondi (int a1[], int n1, int a2[], int n2)***
- **ATT:** *a1 viene sovrascritto! Quindi può essere necessario un array ausiliario...*