# Strutture Dati e Algortimi

## Un algoritmo di ordinamento ottimo

Luca Di Gaspero, Università degli Studi di Udine

# Riepilogo degli algoritmi visti

| Algoritmo | Idea | Caso Pessimo | Caso Ottimo |
|---|---|---|---|
| **Genera e verifica** | Genero tutte le permutazioni e verifico quale di esse corrisponde al vettore ordinato | $O(n!)$ | $O(n)$ |
| **Selection sort** | Cerco (seleziono) l'elemento minimo fra quelli rimasti da ordinare e lo scambio con l'elemento corrente | $O(n^2)$ | $O(n^2)$ |
| **Insertion sort** | Cerco di inserire l'elemento corrente fra quelli precedenti, già ordinati | $O(n^2)$ | $O(n)$ |
| **Bubble sort** | Effettuo più passaggi facendo *affiorare* gli elementi più grandi, finché non sono necessari più scambi | $O(n^2)$ | $O(n)$ |

$\Omega(n \log n)$ è un limite inferiore alla complessità di qualunque algoritmo di ordinamento basato sui confronti

# Una variazione di selection sort

Lo schema di selection sort può essere opportunamente modificato per funzionare anche con l'elemento massimo, anziché quello minimo

```
void SelectionSortMax(int a[], int n) {
  int indice_massimo, i;
  for (i = n - 1; i > 0; i--) {
    indice_massimo = CercaMassimoFinoA(a, n, i);
    Scambia(&a[i], &a[indice_massimo]);
  }
}

int CercaMassimoFinoA(int a[], int n, int i) {
  int j, m = 0;
  for (j = 1; j <= i; j++)
    if (a[m] < a[j])
      m = j;
  return m;
}
```

La complessità è sempre $O(n^2)$

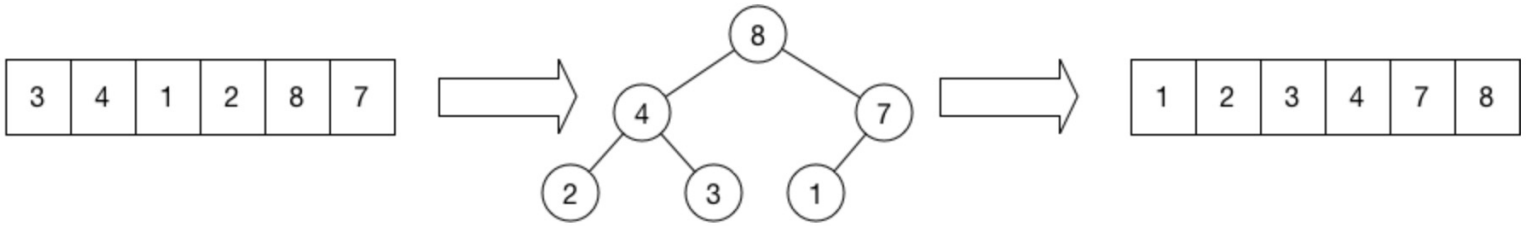
In particolare, il contributo all complessità totale della funzione di ricerca del massimo $t(n, i)$ è lineare in $i$ ($t(n, i) = O(i)$)

È possibile diminuire tale complessità?

- esiste una struttura dati che consente di estrarre l'elemento massimo in modo più efficiente?

# Heapsort: un algoritmo di ordinamento ottimo

**Idea**: uso lo stesso schema del **selection sort** ma sfruttando una struttura dati più efficiente per ottenere il massimo di un insieme di valori: usando uno **heap tree** (cfr. coda di priorità)




Schema concettuale:

1. costruisco uno heap tree con i valori del vettore aggiungendone gli elementi uno alla volta (è come se facessi `Enqueue` su una coda di priorità in cui il valore di priorità coincide con il dato dell'array)
2. finché ci sono elementi estraggo il massimo dallo heap tree e lo inserisco nella sua posizione definitiva nell'array ordinato ( `Dequeue` )

# Heapsort: versione concettuale

```
HeapSort(v, n):
  /* In questa versione, usiamo concettualmente tre strutture distinte:
      1. l'array originale v,
      2. una coda di priorità pq
      3. un array destionazione r
  */
  pq = CreaCodaPriorita();
  r = CreaArray(n);
  for (i = 0; i < n; i++)
    Enqueue(pq, v[i]);
  for (i = n - 1; i >= 0; i--) {
    r[i] = First(pq);
    Dequeue(pq);
  }
  /* r è il vettore ordinato */
```

# Heapsort: complessità

- $n$ operazioni di `Enqueue` nell'albero, ciascuna delle quali ha costo $O(\log i) \subseteq O(\log n)$, dunque questa parte costa $n \cdot O(\log n) = O(n \log n)$
- $n$ operazioni di `Dequeue` dall'albero, ciascuna delle quali ha costo $O(\log i)$, pertanto anche questa parte costa $O(n \log n)$
- La complessità complessiva dell'algoritmo è dunque $O(n \log n)$, ossia Heap sort è un algoritmo di ordinamento **ottimo**

# Heapsort: implementazione *in-place*

```
void heap_sort(float a[], int n) {
  int i;
  for (i = 0; i < n; i++) {
    // di fatto è equivalente ad effettuare un Enqueue()
    _hs_riorganizza_heap_enqueue(a, i + 1, i);
  }
  i = n - 1;
  while (i > 0) {
    // e le due istruzioni seguenti sono equivalenti ad effettuare un Dequeue()
    _scambia(&a[0], &a[i]);
    i = i - 1;
    _hs_riorganizza_heap_dequeue(a, i + 1);
  }
}
```

# Heapsort: implementazione *in-place*

```
void _hs_riorganizza_heap_enqueue(float a[], int n, int i) {
  /* Enqueue */
  while (i > 0 && a[i] > a[_hs_padre(i)]) {
    _scambia(&a[i], &a[_hs_padre(i)]);
    i = _hs_padre(i);
  }
}

void _hs_riorganizza_heap_dequeue(float a[], int n) {
  int i = 0;
  /* Dequeue */
  while (_hs_sinistro(i) < n && i != _hs_migliore_padre_figli(a, n, i)) {
    int figlio_migliore = _hs_migliore_padre_figli(a, n, i);
    _scambia(&a[i], &a[figlio_migliore]);
    i = figlio_migliore;
  }
}
```

# Heapsort: implementazione *in-place*

```
int _hs_migliore_padre_figli(float a[], int n, int i) {
  int k = i;
  if (_hs_sinistro(i) < n && a[k] < a[_hs_sinistro(i)])
    k = _hs_sinistro(i);
  if (_hs_destro(i) < n && a[k] < a[_hs_destro(i)])
    k = _hs_destro(i);
  return k;
}
```

# Heapsort: implementazione *in-place*

```
int _hs_padre(int i) {
  return (i - 1) / 2;
}
int _hs_sinistro(int i) {
  return 2 * i + 1;
}
int _hs_destro(int i) {
  return 2 * i + 2;
}
```