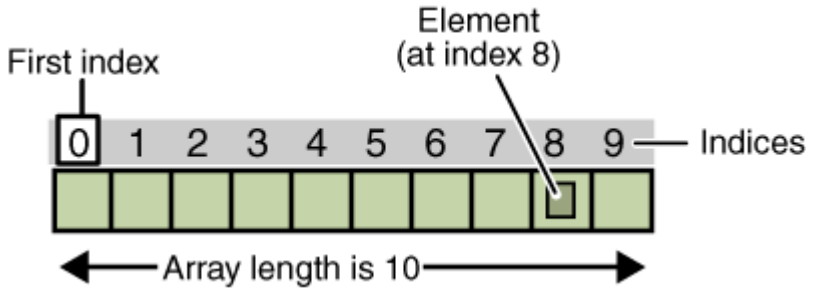
UNIVERSITÀ
DI PARMA

algoritmi su array
ricerca e ordinamento

Alberto Ferrari

- struttura ***statica omogenea***
  - non in tutti i linguaggi … *array dinamici*
- ***accesso diretto*** a ogni elemento attraverso l'***indice***
  - complessità dell'accesso ***O(1)***



algoritmo di visita

- percorrere ***una e una sola volta*** tutti gli ***elementi***

```
void visita_array(int a[], int n) {
  for (int i = 0; i < n; i++)
      elabora(a[i]);
}
```

- se *elabora* ha complessità ***x*** passi la complessità dell'algoritmo risulta:
  `1+n(1+x+1)+1 = (x+2)n+2 =` ***O(n)***

- stabilire se un ***valore*** è ***presente*** all'interno dell'array restituendo l'***indice*** dell'elemento o ***-1*** se non presente
- ricerca ***sequenziale*** (*lineare*): algoritmo per trovare un elemento in un insieme ***non ordinato***
  - si effettua la scansione dell'array ***sequenzialmente***
- ricerca ***binaria*** (*dicotomica*): algoritmo per trovare un elemento in un insieme ***ordinato***
  - si inizia la ricerca dall'***elemento centrale***, si confronta questo elemento con quello cercato:
    - se ***corrisponde***, la ricerca termina con successo
    - se è ***superiore***, la ricerca viene ripetuta sugli elementi ***precedenti***
    - se è ***inferiore***, la ricerca viene ripetuta sugli elementi ***successivi***

ricerca sequenziale

```
int linearSearch(int array[],
                 int size, int val) {
  for (int i=0;i<size;++i)
      if (array[i]==val)
          return i;
  return -1;
}
```

- ***complessità*** computazionale
  - caso ***peggiore O(n)***
  - caso ***ottimo O(1)***
  - caso ***medio O(n/2)***

ricerca binaria

```
int binarySearch(int array[], int size,
                 int val) {
    int first, last, medium;
    first = 0;
    last = size - 1;
    while(first <= last) {
        medium = (first + last) / 2;
        if(array[medium] == val)
            return medium; // value found
        if(array[medium] < val)
            first = medium + 1;
        else
            last = medium - 1;
    }
    return -1; // not found
}
```

- ***complessità*** computazionale
  - caso ***peggiore O(log2n)***
  - caso ***ottimo O(1)***
  - caso ***medio O(log2n)***

array – algoritmi di

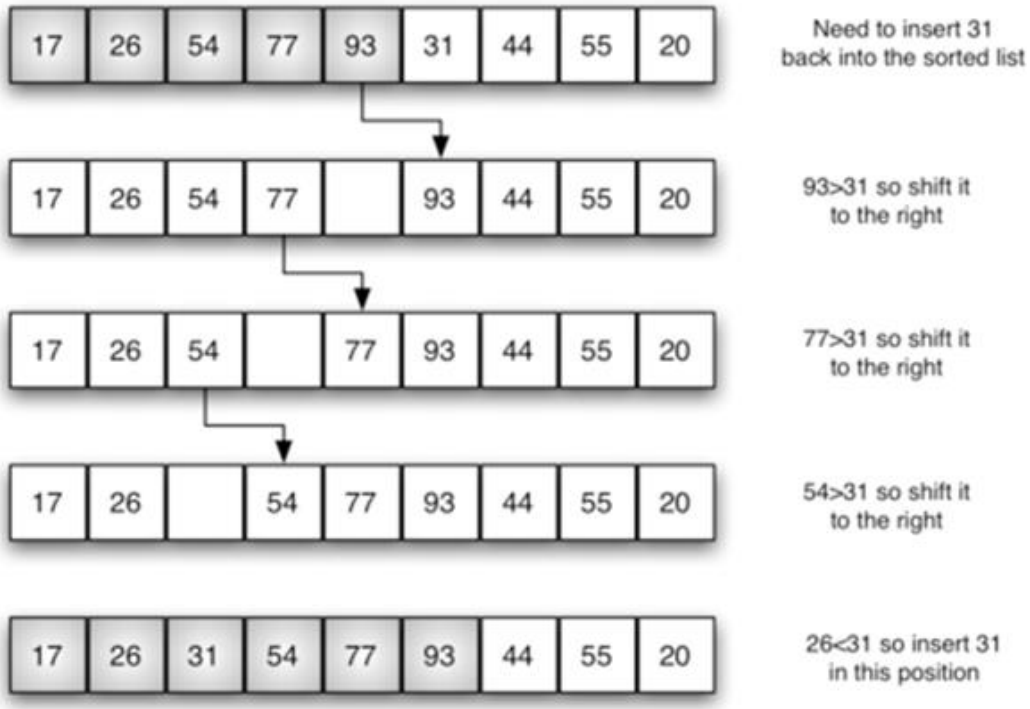
ordinamento

- l'***ordinamento*** degli elementi di un array avviene considerando il valore della ***chiave primaria***
- negli esempi le chiavi sono interi e la ***relazione d'ordine totale è <=***
- caratteristiche degli algoritmi di ordinamento:
  - ***efficienza*** (complessità computazionale)
  - ***stabilità***: l'algoritmo è stabile se ***non altera l'ordine*** relativo di elementi dell'array aventi la stessa chiave primaria
  - ***sul posto***: l'algoritmo opera sul posto se la ***dimensione*** delle ***strutture ausiliarie*** di cui necessita è ***indipendente dal numero di elementi*** dell'array da ordinare

https://www.toptal.com/developers/sorting-algorithms/

insertion sort

- al generico passo ***i*** l'array è considerato ***diviso*** in
  - una sequenza di ***destinazione*** ***a[0] ... a[i - 1] già ordinata***
  - una sequenza di ***origine*** ***a[i] ... a[n - 1] ancora da ordinare***
- l'obiettivo è di ***inserire*** il valore contenuto in ***a[i]*** al ***posto giusto*** nella sequenza di destinazione facendolo scivolare a ritroso, in modo da ***ridurre*** la sequenza di origine di un elemento



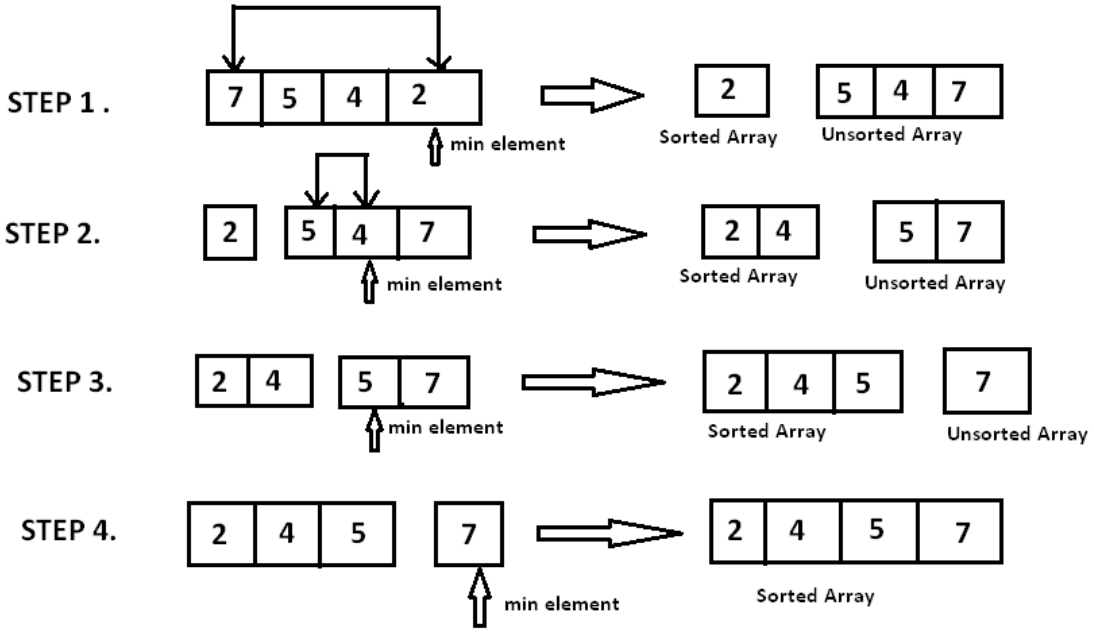
insertion sort

```
void insertsort(int array[], int size) {
  int i, j, app;
  for (i=1; i<size; i++) {
    app = array[i];
    j = i-1;
    while (j>=0 && array[j]>app) {
      array[j+1] = array[j];
      j--;
    }
    array[j+1] = app;
  }
  return;
}
```

- ***sul posto***
- ***stabile***
- ***complessità $O(n^2)$***
- efficiente su array già ***parzialmente ordinati***

selection sort

- al generico passo ***i*** vede l'array diviso in:
  - una sequenza di ***destinazione*** ***a[0] ... a[i - 1]*** già ***ordinata***
  - una sequenza di ***origine*** ***a[i] ... a[n - 1]*** da ordinare
- l'***obiettivo*** è ***scambiare*** il valore ***minimo*** della seconda sequenza ***con*** il valore contenuto in ***a[i]*** in modo da ridurre la sequenza di origine di un elemento



selection sort

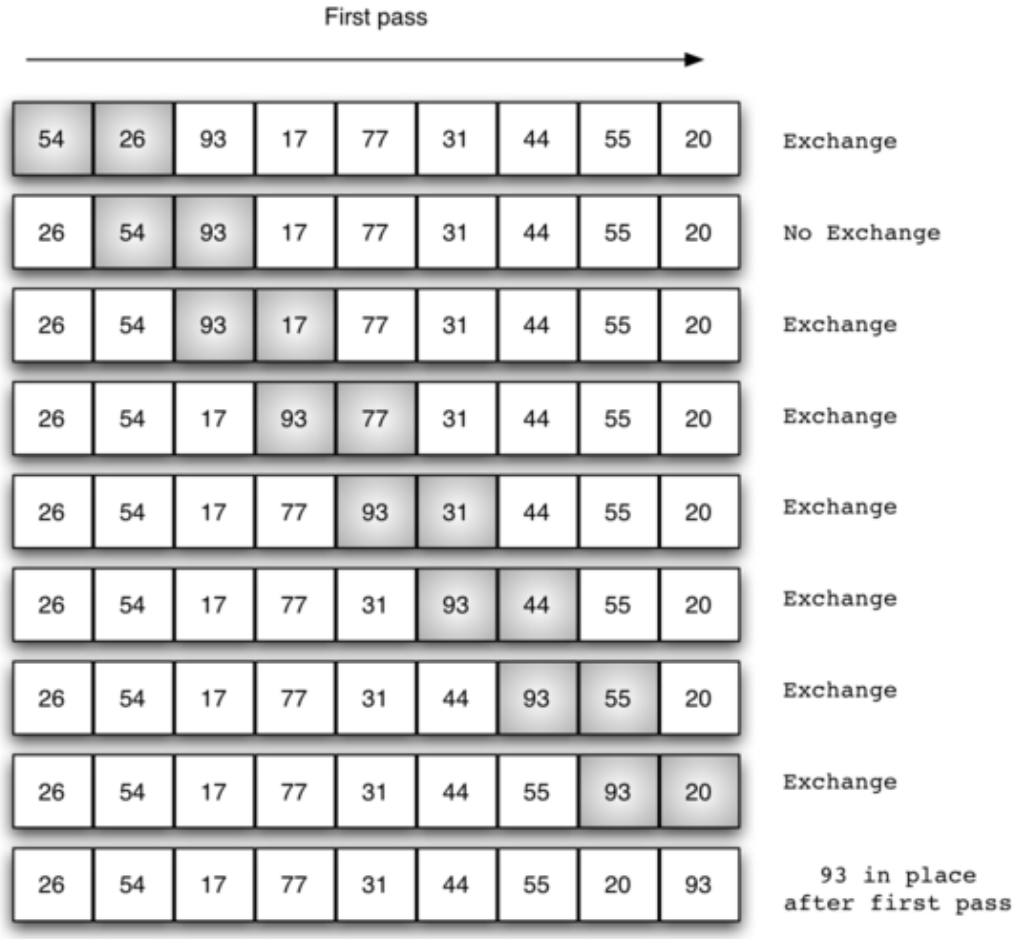
```
void selectsort(int array[],int size){
  int i,j,min;
  for (i=0; i<size; i++) {
    min = i;
    for (j=i+1; j<size; j++)
        if (array[min]> array[j])
            min = j;
    if (min != i)
        swap(array[i],array[min]);
    }
}
```

- *sul posto*
- *non stabile*
- *complessità $O(n^2)$*

```
void swap(int &x, int &y) {
    int temp = x;
    x = y;
    y = temp;
}
```

bubblesort

- al generico passo i vede l'array diviso in:
  - una sequenza di ***destinazione***
    a[0] ... a[i - 1] già ***ordinata***
  - una sequenza di ***origine***
    a[i] ... a[n - 1] da ordinare
- l'obiettivo è di far emergere il valore ***minimo*** della sequenza di origine confrontando e ***scambiando sistematicamente*** i valori di elementi adiacenti a partire dalla fine dell'array, in modo da ridurre la sequenza di origine di un elemento
  - sul posto
  - stabile
  - complessità $O(n^2)$
  - efficiente per array parzialmente ordinati (*versione ottimizzata*)



bubble sort

- versione semplificata

```
void bubble_sort(int array[], int size) {
 int i,last;
 for (last = size - 1; last > 0; last-- ){
        for (i=0; i<last; i++){
            if (array[i]>array[i+1])
              swap(array[i], array[i+1]);
        }
    }
}
```

- se ***non*** avvengono ***scambi*** l'array risulta ***ordinato***

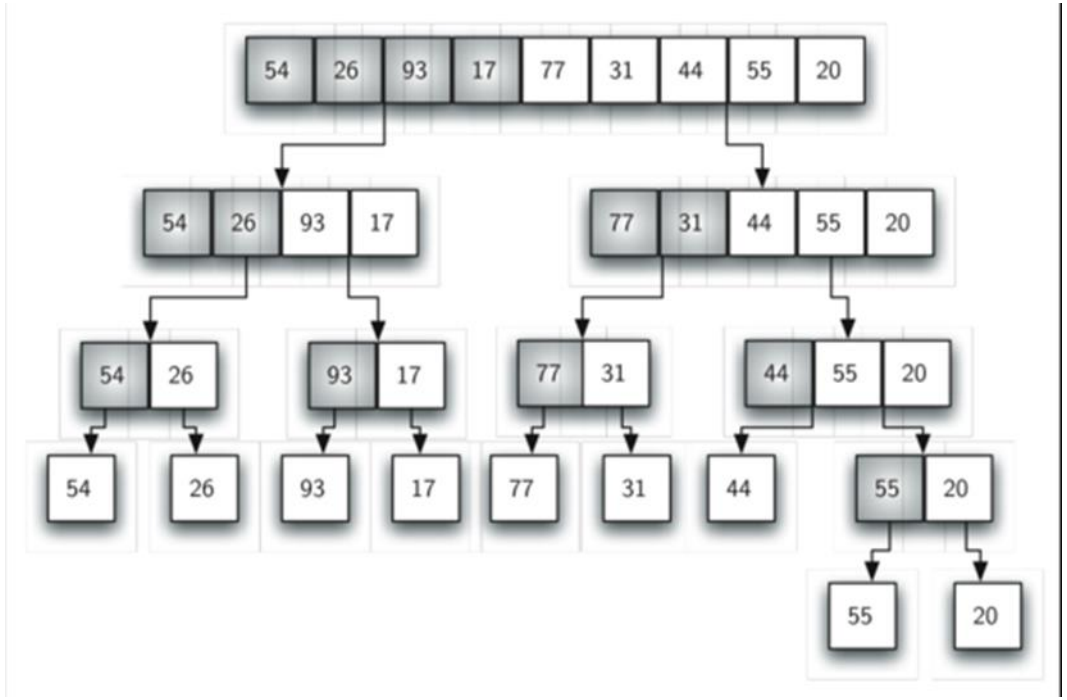
```
void bubble_sort(int array[], int size) {
 int i,last;
 bool swapped;
 for (last = size - 1; last > 0; last-- ){
        swapped = false;
        for (i=0; i<last; i++){
            if (array[i]>array[i+1])
              swap = true;
              swap(array[i], array[i+1]);
        }
       if (!swapped)
          return;
    }
}
```

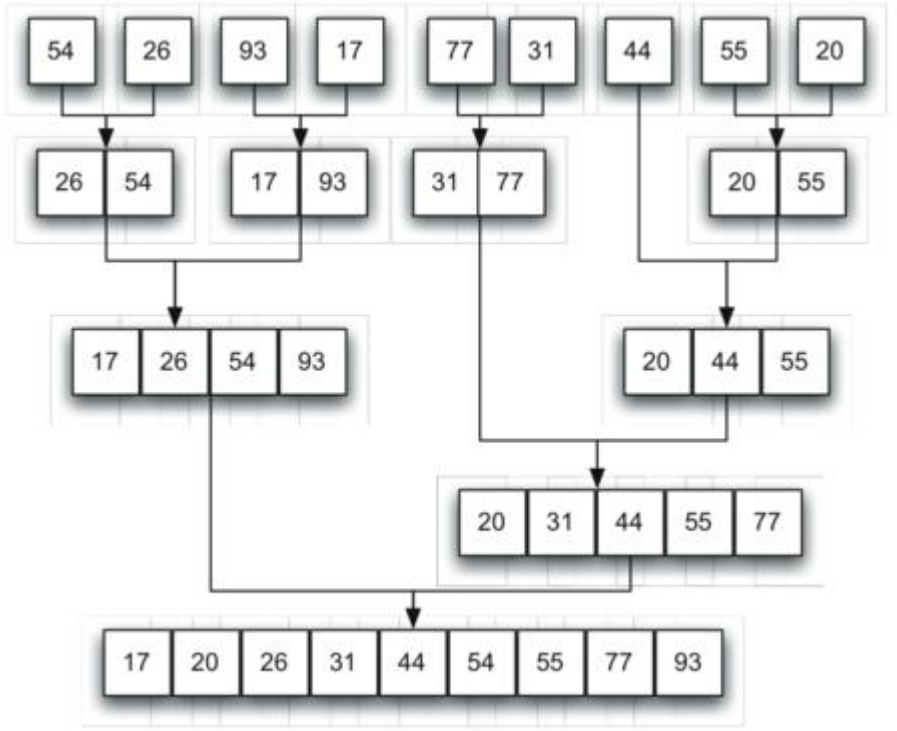
merge sort

- algoritmo ***ricorsivo***
- sfrutta la tecnica del ***divide et impera***
  - suddivisione del problema in ***sottoproblemi*** della stessa natura di ***dimensione*** via via ***più piccola***
- ***non*** opera ***sul posto***: *nella fusione usa un array di appoggio il cui numero di elementi è proporzionale al numero di elementi dell'array da ordinare*
- ***stabile***
- complessità $\boldsymbol{O(n \log(n))}$

merge sort

- se la sequenza da ordinare ha ***lunghezza 0*** o ***1***, è ***ordinata***
- altrimenti la sequenza viene divisa (***divide***) in due ***metà*** *(se numero dispari di elementi la prima ha un elemento in più della seconda)*
- ogni ***sottosequenza*** viene ***ordinata***, applicando ***ricorsivamente*** l'algoritmo (***impera***)
- le due ***sottosequenze ordinate*** vengono ***fuse*** (***combina***)
- si estrae ripetutamente il ***minimo*** delle due sottosequenze e lo si pone nella ***sequenza in uscita***, che risulterà **ordinata**


54 26 93 17 77 31 44 55 20
54 26 93 17
77 31 44 55 20
54 26
93 17
77 31
44 55 20
54
26
93
17
77
31
44
55 20
55
20


54 26 93 17 77 31 44 55 20
26 54
17 93
31 77
20 55
17 26 54 93
20 44 55
20 31 44 55 77
17 20 26 31 44 54 55 77 93

merge sort

```
void merge_sort(int array[],
                int left, int right) {
  int center; // middle index
  if(left<right) {
     center = (left+right)/2;
     //sort first half
     merge_sort(array, left, center);
     //sort second half
     merge_sort(array, center+1, right);
     //merge sorted arrays
     merge(array, left, center, right); }
}
```

```
void merge(int array[], int left,
           int center, int right){
  int i = left;     //index first array
  int j = center+1; //index second array
  int k = 0;   //index new temporary array
  int temp[DIM_ARRAY];  //temporary array
  while ((i<=center) && (j<=right)) {
    if (array[i] <= array[j]) {
        temp[k] = array[i]; i++; }
    else { temp[k] = array[j]; j++; }
    k++;
  }
  while (i<=center) {
    temp[k] = array[i]; i++; k++; }
  while (j<=right) {
    temp[k] = array[j]; j++; k++; }
  for (k=left; k<=right; k++){
    array[k] = temp[k-left]; }
}
```