Algoritmi notevoli

Algoritmi

- Ricerca
  (verificare la presenza di un valore in un array)
  - Ricerca sequenziale (array non ordinato)
  - Ricerca sequenziale (array ordinato)
  - Ricerca binaria (array ordinato)
- Ordinamento
  (ordinare i valori all'interno di un array in modo crescente o decrescente)
  - Stupid Sort
  - Selection Sort
  - Bubble Sort

Algoritmi di ricerca

- In generale un algoritmo di ricerca si pone come obiettivo quelli di trovare un elemento avente determinate caratteristiche all'interno di un insieme di elementi.
- Nel nostro caso definiremo algoritmi che verificano la presenza di un valore in un array.
- L'array può essere non ordinato o ordinato (ipotizziamo l'ordinamento crescente)

Ricerca sequenziale

- La ricerca sequenziale (o completa) consiste nella scansione sequenziale degli elementi dal primo all'ultimo e si interrompe quando il valore cercato è stato trovato, oppure quando si è sicuri che il valore non può essere presente.
- Ha il vantaggio di poter essere applicata anche a dati non ordinati.
- Negli esempi tratteremo la ricerca del valore **x** in un array float di nome **v** con **n** elementi con funzioni che restituiscono l'**indice** dell'elemento dell'array con valore **x** o **-1** in caso di valore non trovato.

Ricerca sequenziale C++

```
int ricercaSequenziale(float v[], int n, float x)
{
     int i=0;

     while (i<n && x!=v[i])
          i++;

     if (i==n)
          return -1;
     return i;
 }
```

Ricerca sequenziale in array ordinato

```
int ricercaSequenzialeOrd(float v[], int n, float x)
{
     int i=0;

     while (i<n && x<v[i])
          i++;

     if (i<n && v[i]==x)
          return i;
     return -1;
 }
```

Ricerca binaria (o logaritmica)

- L'algoritmo è simile al metodo usato per trovare una parola sul dizionario: sapendo che il vocabolario è ordinato alfabeticamente, l'idea è quella di iniziare la ricerca non dal primo elemento, ma da quello centrale, cioè a metà del dizionario. A questo punto il valore ricercato viene confrontato con il valore dell'elemento preso in esame:
- se corrisponde, la ricerca termina indicando che l'elemento è stato trovato;
- se è inferiore, la ricerca viene ripetuta sugli elementi precedenti (ovvero sulla prima metà del dizionario), scartando quelli successivi;
- se invece è superiore, la ricerca viene ripetuta sugli elementi successivi (ovvero sulla seconda metà del dizionario), scartando quelli precedenti;
- se tutti gli elementi sono stati scartati, la ricerca termina indicando che il valore non è stato trovato.

Ricerca binaria in C++

```
int ricercaBinaria(float v[], int n, float x)
{
   int primo,ultimo,medio;
   primo = 0;
   ultimo = n-1;
   while(primo<=ultimo)    // non tutti gli elementi sono stati scartati
   {
       medio = (primo+ultimo)/2;
       if(v[medio]==x)
           return medio; // valore x trovato alla posizione medio
       if(v[m]<x)
           primo = medio+1;          // scarto la prima metà
       else
           ultimo = medio-1;         // scarto la seconda metà
   }
   // se il programma arriva qui l'elemento non e' stato trovato
   // e sono stati scartati tutti gli elementi
   return -1;
}
```

Ricerca Binaria ricorsiva

- L'algoritmo si presta ad una definizione ricorsiva.
- Ad ogni chiamata della funzione si verifica se l'elemento ricercato si trova al centro dell'intervallo e in tal caso la funzione termina con successo, in caso contrario si modifica l'intervallo di ricerca e si effettua una nuova chiamata della funzione.
- Nel caso in cui l'intervallo di ricerca sia nullo si termina la ricorsione con insuccesso.

Implementazione ricorsiva

```
int ricercaRicorsiva(float v[], int inizio, int fine, float x)
{
   if(inizio>fine)     // terminazione con insuccesso
      return -1;
   int medio=(inizio+fine)/2;
   if (v[m]==x)        // terminazione con successo
      return medio;
   if (v[m]>x)
      return ricercaRicorsiva(v,inizio,medio-1,x);
   else
      return ricercaRicorsiva(v,medio+1,fine,x);
}
```

Confronto fra gli algoritmi

- In genere l'efficienza si misura in base al numero di confronti effettuati che dipende da n (lunghezza dell'array).
- Si individuano il caso migliore e peggiore ma in generale interessa il caso medio.

Algoritmo	Caso migliore	Caso peggiore	Caso medio	Caso medio con n = 1000
Ricerca Sequenziale	1	n	n / 2	500
Ricerca Binaria	1	lg_2 n	lg_2 n	10

Algoritmi di ordinamento

- Stupid Sort
  - particolarmente **inefficiente**, come si può intuire dal nome. Consiste nel mischiare in qualche modo gli elementi dell'array poi controllare se è ordinato e, se non lo è, ricominciare da capo.
- Selection Sort
  - consiste in più scansioni dell'array: al termine della prima il primo elemento conterrà il valore minore, poi si proseguirà ordinando la parte successiva dell'array.
- Bubble Sort
  - consiste nella scansione dell'array elemento per elemento, scambiando i valori dei due elementi consecutivi, quando il primo è maggiore del secondo.

Scambio di elementi

- Tutti gli algoritmi di ordinamento si basano sullo scambio degli elementi dell'array.
- In tutti gli esempi faremo riferimento nelle funzioni a un generico array di float ***v*** di lunghezza ***n***.
- Per lo scambio del valore di due elementi useremo la funzione:

```
void scambia(float &e1,float &e2)
{
    float app;  // appoggio
    app = e1;
    e1 = e2;
    e2 = app;
}
```

Stupid Sort

- L'algoritmo è probabilistico.
- La ragione per cui l'algoritmo arriva quasi sicuramente a una conclusione è spiegato dal teorema della scimmia instancabile: ad ogni tentativo c'è una probabilità di ottenere l'ordinamento giusto, quindi dato un numero illimitato di tentativi, infine dovrebbe avere successo.
- Il Bozo Sort è una variante ancora meno efficiente. Consiste nel controllare se l'array è ordinato e, se non lo è, prendere due elementi casualmente e scambiarli (indipendentemente dal fatto che lo scambio aiuti l'ordinamento o meno).

Bozo Sort C++

```
void bozoSort(float v[], int n){
  while(!ordinato(v,n)
     mescola(v,n);
}

bool ordinato(float v[],int n){
   for(int i=0;i<n-1;i++)
       if(v[i]>v[i+1])
           return false;
   return true;
}

void mescola(float v[], int n){
  int i1,i2   // indici casuali
  i1=(rand() % n);
  i2=(rand() % n);
  scambia(v[i1], v[i2]);
}
```

Selection Sort

- L'algoritmo ricerca l'elemento minore della regione del vettore da ordinare e lo sposta all'inizio della regione stessa.
- Ad ogni scansione viene spostato un elemento del vettore nella posizione corretta.
- L'ordinamento ha termine quando la regione considerata è costituita da un solo elemento.

Un esempio

Array di partenza	1	23	4	-56	65	21	32	15	0	-3
Scansione 1	-56	23	4	1	65	21	32	15	0	-3
Scansione 2	-56	-3	4	1	65	21	32	15	0	23
Scansione 3	-56	-3	0	1	65	21	32	15	4	23
Scansione 4	-56	-3	0	1	65	21	32	15	4	23
Scansione 5	-56	-3	0	1	4	21	32	15	65	23
Scansione 6	-56	-3	0	1	4	15	32	21	65	23
Scansione 7	-56	-3	0	1	4	15	21	32	65	23
Scansione 8	-56	-3	0	1	4	15	21	23	65	32
Scansione 9	-56	-3	0	1	4	15	21	23	32	65

Selection Sort C++ (1)

```
void selectionSort(float v[],int n) {
    for (int s = 0; s < n - 1; s++)
    {
      // n-1 scansioni (n è la dimensione dell'array)
      // la posizione dell'elemento minore è inizialmente
      // la prima della regione da analizzare
      int posizMin = s;
      for (int i = s + 1; i < n; i++)
      {
        // ricerca la posizione dell'elemento minore
        // fra quelli presenti nella regione
        if (v[i] < v[posizMin])
          posizMin = i;
      }
      scambia(v[s],v[posizMin]);
    }
}
```

Selection Sort C++ (2)

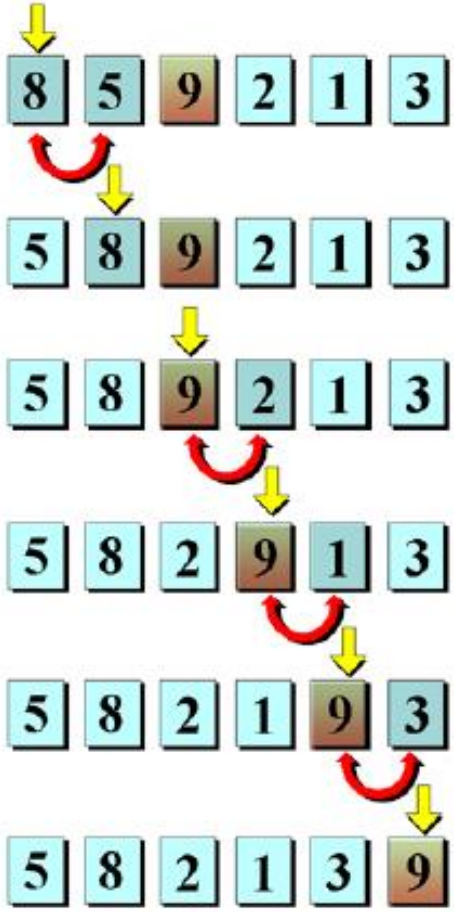
```
void selectionSort(float v[],int n) {
for (int s = 0; s < n - 1; s++)
  {
    // n-1 scansioni (n è la dimensione dell'array)
    for (int i = s + 1; i < n; i++)
    {
      // scambio di posizione fra il primo elemento
      // della sequenza e un elemento con valore minore
      if (v[i] < v[s])
      {
        scambia(v[i],v[s]);
      }
    } // fine ciclo interno

  }
}
```

Bubble Sort

- Consiste nella scansione dell'array elemento per elemento, scambiando i valori dei due elementi consecutivi, quando il primo è maggiore del secondo.
- Al termine della scansione, in genere l'array non risulta ordinato e si deve procedere a una nuova scansione e alla conseguente serie di eventuali scambi tra i valori di due elementi consecutivi.
- Sicuramente l'array risulta ordinato quando si sono effettuate n – 1 scansioni, se n sono gli elementi dell'array.
- E' detto bubblesort (ordinamento a bolle) per analogia con le bolle d'aria nell'acqua che, essendo leggere, tendono a spostarsi verso l'alto.

Un esempio



Bubble Sort C++ (1)

```
void bubbleSort(float v[], int n){
  for(int s=0;s<n-1;s++)
    for(int i=0;i<n-1;i++)
       if (v[i]>v[i+1])
          scambia(v[i],v[i+1];
}
```

Bubble Sort C++ (2)

- E' possibile migliorare l'efficienza dell'algoritmo controllando se sono stati effettuati spostamenti, in caso negativo l'array risulta già ordinato

```
void bubbleSort(float v[], int n){
  bool spostamento;
  int s=0;
  do
  {
     spostamento = false;
     for(int i=0;i<n-1;i++)
   if (v[i]>v[i+1])
       {
          spostamento = true;
     scambia(v[i],v[i+1];
       }
     s++;
  } while (spostamento && s<n-1);
}
```