34. Algoritmi preimpostati: ricerca di un elemento in un array di N elementi.
Quanti passi richiede in media il cercare un elemento in un array di N elementi?

Con un array generico: circa N/2 se l'elemento è presente, circa N se non è presente.
Per indicare la complessità di un algoritmo si usa la seguente notazione *O(X)*, dove *X* è il numero medio di passi richiesti dall'algoritmo. In questo caso è *O(N)* (un numero proporzionale ad N). Ad esempio se N = 1.000.000, il numero di passi sarà <= 1.000.000.

Si chiama algoritmo di ricerca lineare:

```
int search(const double a[], int n, double target)
{
  bool found;
  int location = 0;

  for (found=false; !found && location < n; location++) {
        if(a[location] == target) {
        found = true;
        }
}

return (found ? location-1 : -1);          //ritorna la cella dell'array dell'elemento cercato oppure
}                                          -1 se l'elemento non è presente
```

Con un array ordinato: ≤ log2(N) se l'elemento è presente, log2(N) se non è presente. La complessità è *O(log2(N))* (un numero proporzionale al logaritmo in base 2 di N). Ad esempio con N = 1.000.000 → meno di 20 passi.

Si chiama algoritmo di ricerca binaria:

```
int search(const double a[], int n, double target)
{
  bool found=false;
  int location, left = 0, right = n-1;

  while (!found && (left <= right)) {
    location = (left + right)/2;

    if (a[location] == target)
      found = true;
    else if (a[location] < target)
      left = location + 1;
    else
      right = location – 1;

  }

  return (found ? location : -1);
}
```

Versione ricorsiva:

```
int search(const double a[], int n, double target) {
  return search1(a,0,n-1,target);
}

int search1(const double a[], int left, int right, double target)
{
  int res,pivot = (left + right)/2;
  if (left > right)
    res = -1;
  else if (target == a[pivot])
    res = pivot;
  else if (target < a[pivot])
    res = search1(a,left,pivot-1,target);
  else // (target > a[pivot])
    res = search1(a,pivot+1,right,target);
  return res;
}
```

34.1 Metodi di ordinamento
Ci sono moltissimi metodi di ordinamento di un array. I più diffusi sono: selection sort, insertion sort, bubble sort, quick sort.

N.B.! All'esame non sarà richiesta la conoscenza di tutti gli algoritmi di ordinamento. Indi per cui studiati il più semplice, ovvero il *Bubble Sort*, nelle sue due varianti: iterativa e ricorsiva.

34.1.1 Ordinamento per selezione: Selection Sort.
Cerco elemento più piccolo dell'array e lo scambio con il primo elemento dell'array.
Cerco il secondo elemento più piccolo dell'array e lo scambio con il secondo elemento dell'array. Proseguo in questo modo fintanto che l'array non è ordinato.

```
void selectionsort(int A[], int N) {
  for(int i = 0; i < N - 1; i++ ){
    int min = i;
    for(int j = i+1; j < N; j++)
      if (A[j] < A[min]) min = j;
    swap(A[i],A[min]);
  }
}
```

Selection sort
Relativamente semplice: O(N^2) passi in media. Ad esempio N = 1.000.000 → ≤ 1.000.000.000.000 = 10^12 passi (!)

34.1.2 Ordinamento per inserzione: Insertion Sort.
È il metodo usato dai giocatori di carte per ordinare in mano le carte. Considero un elemento per volta e lo inserisco al proprio posto tra quelli già considerati (mantenendo questi ultimi ordinati). L'elemento considerato viene inserito nel posto rimasto vacante in seguito allo spostamento di un posto a destra degli elementi più grandi.

```
void insertionsort(int A[], int N) {
  for(int i = N-1; i > 0; i--)
    if (A[i] < A[i-1]) {
      swap(A[i],A[i-1]);
    }

  for(int i = 2; i <= N-1; i++) {
    int j = i;
    int v = A[i];
    while(v < A[j-1]) {
      A[j] = A[j-1]; j--;
    }
    A[j] = v;
  }
}
```

Insertion sort
Relativamente semplice: O(N^2) passi in media. Ad esempio N = 1.000.000 → ≤ 1.000.000.000.000 = 10^12 passi (!)

34.1.3 Ordinamento a bolle: Bubble Sort.

Si basa su scambi di elementi adiacenti se necessari, fino a quando non è più richiesto alcuno scambio e l'array risulta ordinato.

VERSIONE ITERATIVA

```
void bubblesort (int v[],int n)
{
  int i,k;
  for (k=n-1;k>0;k--)
    for (i=0;i<k;i++)
      if (v[i] > v[i+1])
        swap(v[i],v[i+1]);
}
```

VERSIONE RICORSIVA

```
void ordina(int v[], int d{
        ordinarec(v,d,d-1,0);
}

void ordinarec(int v[], int d, int i, int j){

  if(i>0){
    if(j<i){
      if(v[j]>v[j+1]){
          swap(v,j);
      }
      ordinarec(v,d,i,++j);
    }
    ordinarec(v,d,--i,0);
  }

}
```

Bubble sort

Relativamente semplice: O(N^2) passi in media. Ad esempio N = 1.000.000 → ≤ 1.000.000.000.000 = 10^12 passi (!)

34.1.4 Ordinamento Quicksort.

È un algoritmo di ordinamento del tipo divide et impera.

Si basa su un processo di partizionamento dell'array in modo che le seguenti tre condizioni siano verificate:

- per qualche valore di i, l'elemento A[i] si trova al posto giusto.
- tutti gli elementi A[0],...,A[i-1] sono minori od uguali ad A[i].
- tutti gli elementi A[i+1],...,A[N-1] sono maggiori od uguali ad A[i].

L'array è ordinato partizionando ed applicando ricorsivamente il metodo ai sotto array.

Scegliamo arbitrariamente un elemento (e.g. A[r]), che chiameremo pivot (o elemento di partizionamento).

Scandiamo l'array dall'estremità sinistra fino a quando non troviamo un elemento A[i] <= A[r].

Scandiamo l'array dall'estremità destra fino a che non troviamo un elemento A[j] >= A[r].

Scambiamo A[i] e A[j], e iteriamo.

Procedendo in questo modo si arriva ad una situazione in cui tutti gli elementi a sinistra di i sono minori di A[r], mentre quelli a destra di j sono maggiori di A[r].

Versione iterativa

```
int partition(int A[],int l, int r) {
int i = l-1, j = r, v = A[r];

while (true) {
        while (A[++i] < v);
        while (v < A[--j])
            if (j == l) break;
        if (i >= j) break;
        swap(A[i],A[j]);
}
swap(A[i],A[r]);
return(i);
}
```

Versione ricorsiva

```
void quicksort (int A[], int N) {
quicksort_aux(A, 0, N-1);
}

void quicksort_aux (int A[], int l, int r) {
if (r <= l) return;
int i = partition(A, l, r);
quicksort_aux(A, l, i-1);
quicksort_aux(A, i+1, r);
}
```

Quicksort

Complesso: O(N*log2(N)) passi in media. Ad esempio N = 1.000.000 → ≤ 20.000.000 = 2*10^7 passi (!)

34.1.5 Ordinamento Shell Sort

La lentezza dell'algoritmo Insertion Sort risiede nel fatto che le operazioni di scambio avvengono solo tra elementi contigui. Esempio: se l'elemento più piccolo è in fondo all'array, occorrono N scambi per posizionarlo al posto giusto.

Per migliorare questo algoritmo è stato pensato l'algoritmo Shell Sort. L'idea è quella di organizzare l'array in modo che esso soddisfi la proprietà per cui gli elementi aventi tra loro distanza h costituiscono una sequenza ordinata, indipendentemente dall'elemento di partenza. Se si applica l'algoritmo con una sequenza di h che termina con 1, si ottiene un file ordinato.

```
void ShellSort(int A[], int l, int r) {
int h;

for(h = 1; h <= (r-1)/9; h = 3*h+1);
for( ; h > 0; h /= 3)
    for(int i = l+h; i <= r; i++) {
        int j = i;
        int v = A[i];
        while((j >= l + h) && (v < A[j-h])) {
            A[j] = a[j-h];
             j = j - h;
```

```
            }
            A[j] = v;
        }
}
```

Shell Sort

Complesso: O(N^3/2) passi in media. Ad esempio N = 1.000.000 → ≤ 1.000.000.000 = 10^9 passi (!) Usando sequenze particolari di h si possono ottenere prestazioni diverse (ad esempio O(N*(log2(N))^2).

34.2 Altre operazione su array ordinati

Fusione ordinata di due array ordinati (merging)

Ad esempio: con *a[N1]={1,3,4,8}* e *b[N2]={2,3,5,6}*, facendo *merge(a,b)* si ottiene *c[]={1,2,3,3,4,5,6,8}*.
Complessità: *O(N1+N2).*

L'array *c* che contiene i due array fusi insieme, deve essere delle dimensioni opportune.

```
void mergeArray(const double a[],int n1,const double b[],int n2,double c[])
{
   for (int i=0,j=0,k=0;k<n1+n2;k++) {
   if (j==n2 || (i<n1 && a[i]<b[j])) {
     c[k]=a[i];
     i++;
   }
   else   {
     c[k]=b[j];
     j++;
   }
  }
}
```

Versione ricorsiva (supponendo che la dimensione degli array sia la stessa)

```
void merge(int a[], int b[], int c[], int d){
  merge(a,b,c,d,0,0,0);
}

void merge(int a[], int b[], int c[], int d1, int d2, int i, int j, int k){

  if(k<d1+d2){
    if(j==d2|| ( (i<d1) && (a[i] < b[j]) ){
      c[k]=a[i];
      merge(a,b,c,d,++i,j,++k);
    } else {
      c[k]=b[j];
      merge(a,b,c,d,i,++j,++k);
    }
  }

}
```

Inserimento di un elemento in un array ordinato
Consiste sostanzialmente nel fare il merging tra un array di un elemento e il nostro array (oppure si può semplicemente inserire un elemento e riordinare l'array).

All'array *c* fondi l'array *a* con l'array *b* che ha il numero inserito dall'utente.

Oppure, per ogni elemento da inserire: si trova la posizione che l'elemento occupa, si scalano tutti gli altri elementi avanti di uno, si inserisce l'elemento nel suo posto (se l'elemento è minore

dell'elemento in posizione *i*, si scalano tutti gli elementi dal finale a quello in *i*, estremi inclusi, di uno verso destra, e poi si inserisce l'elemento nella posizione *i-1*). Servono due cicli annidati, uno per vedere dove inserire l'elemento e uno per scalare gli elementi avanti di una posizione.

Estrazione di un elemento da un array ordinato

Per estrarre un elemento da un array ordinato posso usare l'algoritmo di ricerca binaria ed estrarre l'indice in cui si trova l'elemento cercato.

Per estrarne tutte le occorrenze posso usare l'algoritmo di ricerca lineare, scorrere tutto l'array e salvare gli indici in cui appare il mio elemento in un array e poi stamparlo.

Eliminazione di un elemento

Per eliminare un elemento da un array ordinato basta scorrere l'array e trovare l'indice che quell'elemento occupa (si può usare la ricerca binaria) e poi non appena si ha trovato l'indice dell'elemento da riordinare spostare tutti gli elementi dell'array indietro di una posizione: al posto dell'elemento *k* ci va l'elemento *k+1* e così via. Alla fine ricordarsi di decrementare di uno la dimensione dell'array.

Versione iterativa

```
for(int i=0;i<DIM;i++){
   if(a[i]==x){
     for(int c=i;c<DIM-1;c++){
          a[c]=a[c+ù1];
     }
     DIM--; break;      //oppure usare una condizione booleana per eliminare solo un elemento (così elimina tutte le occorrenze di x)
   }
 }
```

Versione ricorsiva

Elimina tutte le occorrenze.

```
void elimina(int a[], int x, int& DIM){
  elimina(a,x,DIM,0,0);
}

void elimina(int a[], int x, int& DIM, int i, int c){

  if(i<DIM){
    if(a[i]==x){
      spostael(a,x,DIM,i);
      DIM--;
      i=i-1;
    }
    i=i+1;
    c=i;
    elimina(a,x,DIM,i,c);
  }

}
```

```
void spostael(int a[], int x, int DIM, int c){

  if(c<DIM-1){
    a[c]=a[c+1];
    c=c+1;
    spostael(a,x,DIM,c);
  }

}
```

Inserimento di un elemento in un array ordinato

dimV è la dimensione virtuale dell'array (ordinato).

```
 for(int k=0;k<dimV;k++){
   if(a[k]>=val){
     for(int c=dimV;c>=k;c--){
          a[c+1]=a[c];
     }
     a[k]=val;
     dimV++;
     break;       //al posto del break si può usare una condizione booleana
   }
 }

if(a[dimV-1]<val){     //caso in cui val sia maggiore di tutti!
  a[dimV]=val;
  dimV++;
}
```