Sommario

- Strutture dati e algoritmi:
  - Analisi della complessità degli algoritmi
  - Ricerca (lineare e binaria)
  - Ordinamento (per selezione)

Strutture dati e algoritmi

- Come esempi concreti di applicazioni in C++ si useranno le strutture dati e gli algoritmi.
- Si presentano i principali metodi utilizzati per organizzare e rappresentare l'*informazione* (le strutture dati) al fine di ottenerne un'*elaborazione* efficiente (gli algoritmi).
- La prima struttura dati che si considera è l'array: una tabella di oggetti.
- Gli algoritmi possono essere implementati con funzioni.

Analisi della complessità degli algoritmi

- Uno stesso problema può essere risolto da algoritmi di diversa *efficienza*: tale efficienza diventa rilevante quando la quantità di dati da manipolare diventa "grande".
- La *complessità computazionale* o efficienza può essere valutata rispetto all'uso delle seguenti risorse: *spazio* e *tempo*. In generale il fattore tempo è quello critico.
- Il *tempo di esecuzione T* di un algoritmo viene espresso come una funzione della dimensione *n* dei dati in ingresso: *T(n)*.

Analisi della complessità degli algoritmi

- Tuttavia il tempo di esecuzione di un'applicazione dipende dal sistema e dal linguaggio di implementazione.
- Allora si usa il numero di *passi base* compiuti durante l'esecuzione dell'algoritmo: per es. istruzioni di assegnamento e di confronto.
- Poiché tale misura interessa solo per grandi quantità di dati, si fa riferimento all'ordine di grandezza: *complessità asintotica.*

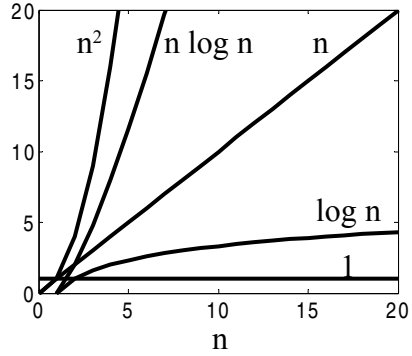
Analisi della complessità degli algoritmi

- Per indicare la *complessità asintotica* si usa la notazione *O-grande (Big-Oh):*
  - date due funzioni a valori positivi *f* e *g*, si dice che *f(n)* è *O(g(n))* se esistono due numeri positivi *c* ed *N* tali che *f(n)<=cg(n)* per qualsiasi *n>=N*.
- Per esempio: $f(n)=2n^2 + 3n +1=O(n^2)$

Analisi della complessità degli algoritmi

Alcune classi di *complessità asintotica:*

- O(1): costante, non dipende dalla dimensione dei dati
- O(log n): logaritmica
- O(n): lineare
- O(n log n): pseudolineare
- $O(n^2)$: quadratica



Analisi della complessità degli algoritmi

- Analizzare la *complessità degli algoritmi* è di *estrema importanza* nonostante le elevate prestazioni dei computer attuali. Algoritmi mal progettati non hanno applicazioni pratiche.
- Per esempio, su un computer che esegue un miliardo di operazioni al secondo un algoritmo pseudolineare impiega alcuni decimi di secondo a manipolare dieci milioni di dati, mentre un algoritmo quadratico impiega ventisette ore!

Esempi di complessità asintotica

- Un ciclo per calcolare la somma dei valori di un array:

```
const int n = ...
int a[n],i,j,sum;
for(i=0, sum=0; i<n ;i++)
      sum+=a[i];
```

- Vengono inizializzate due variabili, poi il ciclo itera n volte (dove `n` è la dimensione dell'array) eseguendo due assegnamenti, uno per aggiornare `sum` e l'altro per incrementare `i`. Quindi si ha:

  $T(n)= 2+ n (1+1) =2 +2 n = O(n)$

  che è una complessità asintotica lineare.

Esempi di complessità asintotica

- La complessità solitamente cresce quando si usano cicli annidati. Per esempio calcolare tutte le somme di tutti i sotto-array che iniziano dalla posizione 0:

```
for(i=0; i<n ;i++){
      for(j=1,sum=a[0]; j<=i; j++)
            sum+=a[j];
      cout<<"Somma sub "<<sum<<endl;}
```

- Il ciclo più esterno esegue tre assegnamenti (`i`, `j` e `sum`) e il ciclo interno, che esegue, a sua volta, *2 i* assegnamenti (`sum` e `j`) per ogni $i \in [1,...,n-1]$. Quindi si ha:

$$T(n) = 1 + 3n + \sum_{i=1}^{n-1} 2i = 1 + 3n + n(n-1) = O(n^2)$$

Esempi di complessità asintotica

- Anche in presenza di cicli annidati, non è detto che la complessità cresca. Per esempio eseguire la somma dei valori che si trovano nelle ultime cinque celle dei sotto-array che partono da 0:

```
for(i=4; i<n ;i++){
      for(j=i-3,sum=a[i-4]; j<=i;j++)
            sum+=a[j];
      cout<<"Somma sub "<<sum<<endl;}
```

- Il ciclo più interno esegue sempre lo stesso numero di assegnamenti per ogni iterazione del ciclo esterno, pertanto si ha:

$T(n) = 1 + (n-4)(1 + 2 + 4 \cdot 2) = O(n)$

Il problema della ricerca

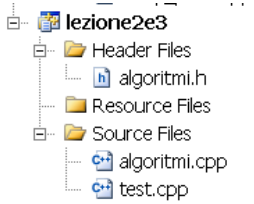
- In molte applicazioni pratiche è importante trovare un certo valore all'interno di una tabella: l'array è la struttura dati che può implementare una semplice tabella.

- *Dato un array e un oggetto (chiave), stabilire se l'oggetto è contenuto in un elemento dell'array, riportando in caso affermativo l'indice di tale elemento, altrimenti -1.*

- Consideriamo il caso generale di un *array non ordinato*. Algoritmo di ricerca lineare: gli elementi dell'array vengono considerati uno dopo l'altro finché o il valore cercato è trovato o la fine dell'array è raggiunta (complessità asintotica lineare O(n)).

Un progetto per il programma di ricerca

- Il `main()` e le funzioni per verificare se l'algoritmo è corretto sono inserite nel file *test.cpp*.
- La funzione per la ricerca è dichiarata in *algoritmi.h* e definita in *algoritmi.cpp*.



Ricerca Lineare

algoritmi.h

```
#ifndef    ALGORITMI_H
#define    ALGORITMI_H
  int SequentialSearch(int *data, int dim, int key);
#endif
```

algoritmi.cpp

```
#include "algoritmi.h"
int SequentialSearch(int *data, int dim, int key){
    for (int i = 0; i < dim; i++)
        if (data[i] == key)
            return i;
    return -1;
}
```


Ricerca Lineare

test.cpp

```
#include <iostream>
#include <cstdlib>
#include "algoritmi.h"
using namespace std;

void test1();

int main(){

   test1();

   return 0;
}
```

Una possibile uscita

Inserire dim, seed e key: 5 3 7196
48 7196 9294 9091 7031
Array[1] vale 7196

```
void test1(){
   int dim, seed;
   int *vt, val;
   cout<<"Inserire dim, seed e key: ";
   cin>>dim>>seed>>val;

   srand(seed);
   vt= new int[dim];
   for(int i=0;i<dim;i++){
        vt[i]=rand();
        cout<<vt[i]<<" ";
   }
   cout<<endl;
   int ind=SequentialSearch(vt,dim,val);
   if (ind>=0)
        cout<<"Array["<<ind<<"] vale
"<<val;
   else
        cout<<"Nell'array non c'e` "<<val;
   delete [] vt;
}
```


Ricerca binaria

Se *l'array è ordinato*, si può seguire una diversa *strategia* per cercare la chiave nell'elenco. Tale ricerca risulta più efficiente perche esegue meno confronti, complessità asintotica logaritmica *O(log n)*.

Algoritmo di ricerca binaria:
Per prima cosa, si confronta la chiave con l'elemento centrale dell'array, se c'è corrispondenza la ricerca è finita.
Altrimenti si decide di continuare la ricerca nella metà destra o sinistra rispetto l'elemento considerato (*l'array è ordinato*, quindi la chiave, se presente, è maggiore o minore dell'elemento considerato) e si utilizza di nuovo la stessa strategia.

Ricerca binaria

```
int BinarySearch(int *data, int dim, int key)
{
  int first=0, last=dim-1;
  while(first <= last)
  {
      int mid = (first + last) / 2;
      if (key==data[mid])
            return mid;
      else if (key<data[mid])
            last=mid - 1;
      else
            first=mid + 1;
  }
  return -1;
}
```

Divisione tra interi

Il problema dell'ordinamento

- *Dato un array i cui elementi contengono tutti una chiave di ordinamento e data una relazione d'ordine sul dominio delle chiavi, determinare una permutazione degli oggetti contenuti negli elementi dell'array tale che la nuova disposizione delle chiavi soddisfi la relazione d'ordine.*

- Vediamo un algoritmo di ordinamento intuitivo ma con una complessità asintotica quadratica $O(n^2)$. Esistono algoritmi più efficienti (per esempio, il quicksort ha una complessità asintotica $O(n \log n)$.

Ordinamento per selezione

- L'idea di base è trovare l'elemento con il valore *minore* e scambiarlo con l'elemento nella prima posizione. Quindi si cerca il valore minore fra gli elementi rimasti (escludendo la prima posizione) e lo si scambia con la seconda posizione. Si continua finché tutti gli elementi sono nella posizione corretta.

- Vediamo una possibile implementazione.

Ordinamento per selezione

```
void Swap(int *vt, int e1, int e2)
{
   int tmp = vt[e1]; vt[e1] = vt[e2]; vt[e2] = tmp;
}

void SelectionSort(int *data, int dim)
{
   int i, j, least;
   for (i = 0; i < dim - 1; i++)
   {
        for (j = i + 1, least = i; j < dim; j++)
        {
              if (data[j]<data[least])
                    least = j;
        }
        Swap(data, least, i);
   }
}
```

Il numero di iterazioni del ciclo interno dipendono da quello esterno

Esempio di ordinamento e ricerca

```
void test2(){
   const int dim=10;
   int vt[dim];
   for(int i=0;i<dim;i++){
       vt[i]=dim-i;
       cout<<vt[i]<<" ";
   }
   cout<<endl;

   SelectionSort(vt,dim);

   for(int i=0;i<dim;i++)
       cout<<vt[i]<<" ";
   cout<<endl;
}
```

Una possibile uscita

```
10 9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9 10
```

Esempio di ordinamento e ricerca

```
void test3(){
   const int dim=10;
   int vt[dim];
   for(int i=0;i<dim;i++){
       vt[i]=dim-i;
       cout<<vt[i]<<" ";
   }
   cout<<endl;
   SelectionSort(vt,dim);
   for(int i=0;i<dim;i++)
       cout<<vt[i]<<" ";
   cout<<endl;
   int val=5;
   cout<<BinarySearch(vt,dim,val)<<endl;
   val=15;
   cout<<BinarySearch(vt,dim,15)<<endl;
}
```

Una possibile uscita

```
10 9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9 10
4
-1
```