ARRAY IN C#

Lezione 3 Ricerca

Table of contents

- Searching RICERCA
- Sorting ORDINAMENTO
- Merge FUSIONE

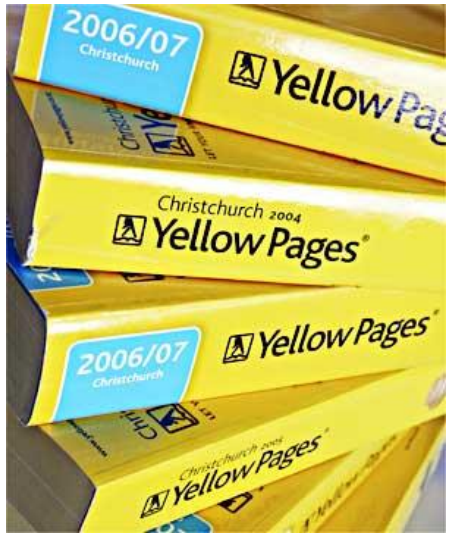
Sorting and Searching

- Fundamental problems in computer science and programming
- Sorting is done to make searching easier
- Multiple different algorithms to solve the same problem
  - How do we know which algorithm is "better"?
- Look at searching first
- Examples will use arrays of ints to illustrate algorithms

LINEAR SEARCH

Searching

- Given a list of data find the location of a particular value or report that value is not present
- linear search
  - intuitive approach
  - start at first item
  - is it the one I am looking for?
  - if not go to next item
  - repeat until found or all items checked
- If items not sorted or unsortable this approach is necessary

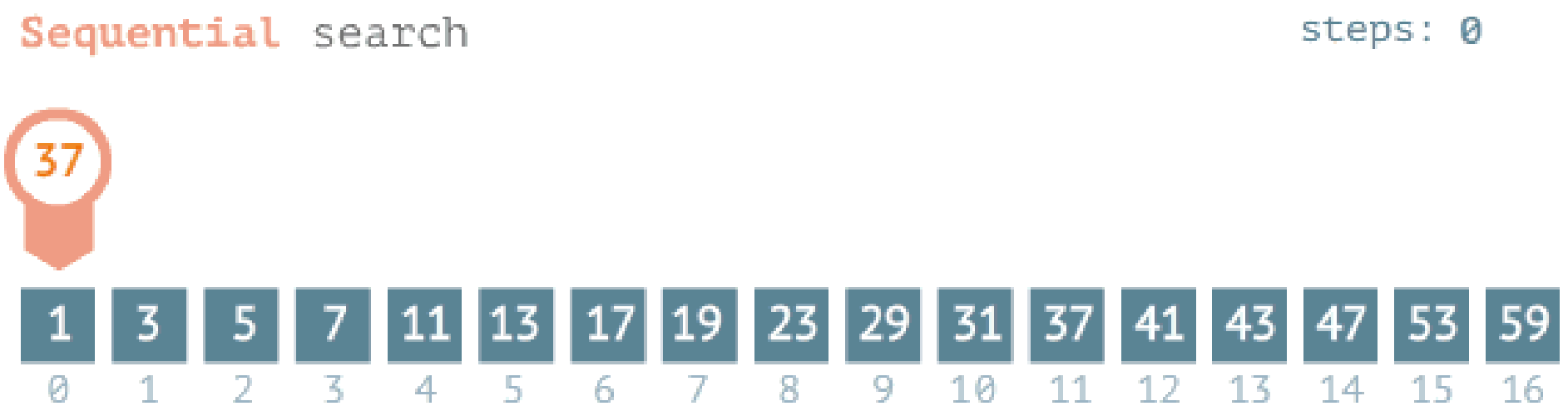


Ricerca Lineare ↔ Linear Search

- Nel caso in cui il vettore non sia ordinato, è necessario utilizzare la **ricerca lineare** (detta anche **sequenziale)**. Poiché non è possibile stabilire a priori in quale parte dell'array potrebbe trovarsi l'elemento da ricercare, è necessario analizzare gli elementi dell'array in sequenza e confrontarli con l'elemento **(chiave)** da ricercare per capire se fa appartiene o no al vettore.

Ricerca Lineare ↔ Linear Search

- Funzionamento della **ricerca lineare** con una **chiave** di valore **37**



Ricerca Lineare ↔ Linear Search

- L’algoritmo è molto semplice: finché non si raggiunge la fine della sequenza, si verifica se l’elemento corrente è uguale alla *chiave*; in caso affermativo si restituisce l’*indice* di tale elemento interrompendo l’operazione di ricerca. Altrimenti si procede fino alla fine della sequenza e, se la *chiave* non è stata trovata, si restituisce il valore *-1*.

Linear Search 1

```
public static int RicercaSeq1(int[] vet,int x)
{
    int pos = -1;
    int i = 0;
    bool trovato = false;
    do
    {
        if (vet[i] == x)
            trovato = true;
        else
            i++;
    } while (!trovato && i < vet.Length);
    if (trovato)
        pos = i;
    return pos;
}
```

```
static void Main(string[] args)
{
    int[] vet = { 2,9,8,7,6,13,11,4};
    int num = 6;
    Console.WriteLine(RicercaSeq1(vet, num));
}
```

Linear Search 2

```
public static int RicercaSeq2(int[] vet, int x)
{
    int pos = -1;
    bool trovato = false;
    for (int i = 0; i < vet.Length && !trovato; i++)
        if (vet[i] == x)
        {
            trovato = true;
            pos = i;
        }
    return pos;
}
```

```
static void Main(string[] args)
{
    int[] vet = { 2,9,8,7,6,13,11,4};
    int num = 6;
    Console.WriteLine(RicercaSeq2(vet, num));
}
```

Linear Search in ordered array

```
public static int RicercaSeqOrd(int[] vet,int x)
{
    int pos = -1;
    int i = 0;
    while (i < vet.Length && x > vet[i])
       i++;

    if (i < vet.Length && vet[i] == x)
        pos = i;
    return pos;
}
```

```
static void Main(string[] args)
{
    int[] vet = { 2,3,4,6,7,8,11,13};// { 2,9,8,7,6,13,11,4};
    int num = 6;
    Console.WriteLine(RicercaSeq1(vet, num));
}
```

ALGORITHM ANALYSIS (BIG O)

Complexity

- In examining algorithm efficiency we must understand the idea of complexity
  - Space complexity
  - Time Complexity

Space Complexity

- When memory was expensive and machines didn't have much we focused on making programs as space efficient as possible and developed schemes to make memory appear larger than it really was (virtual memory and memory paging schemes)
- Although not as important today space complexity is still important in the field of embedded computing (hand held computer based equipment like cell phones, palm devices, etc)

Time Complexity

- Is the algorithm “fast enough” for my needs?
- How much longer will the algorithm take if I increase the amount of data it must process?
- Given a set of algorithms that accomplish the same thing, which is the right one to choose ?

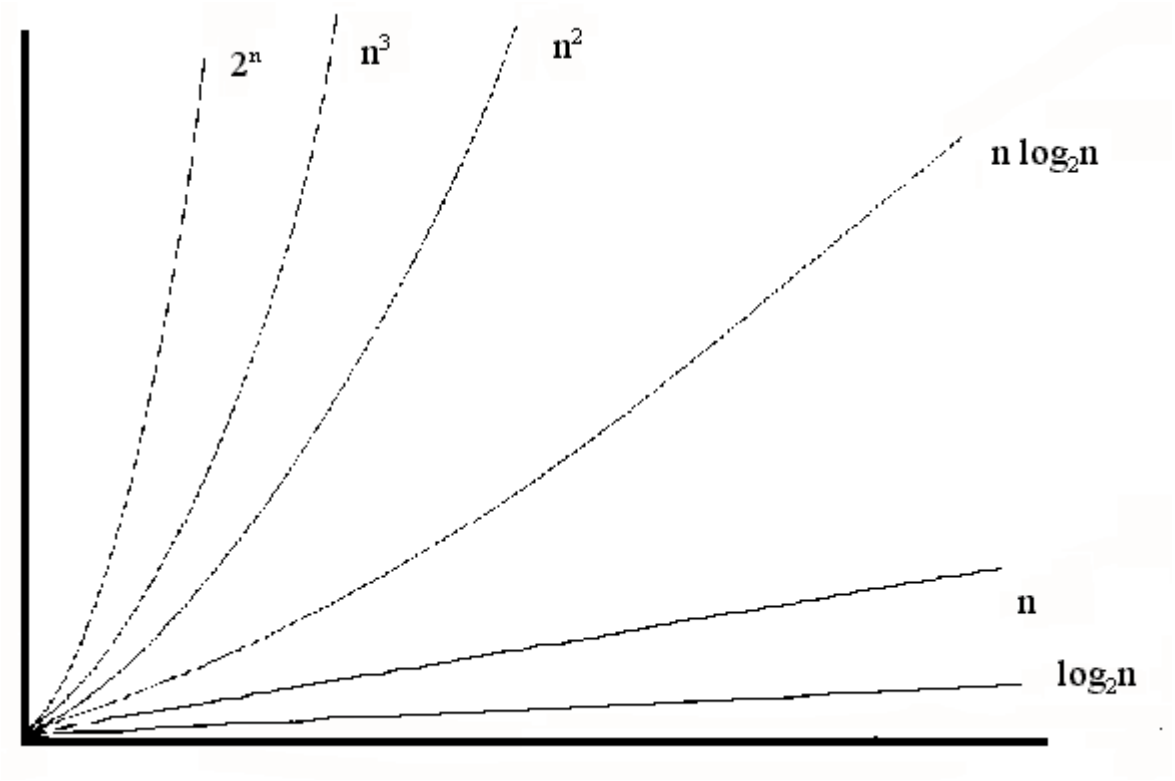
Cases to examine

- Best case
  - if the algorithm is executed, the fewest number of instructions are executed
- Average case
  - executing the algorithm produces path lengths that will on average be the same
- Worst case
  - executing the algorithm produces path lengths that are always a maximum

What is Big O => Order

- Big O is the rate at which performance of an algorithm degrades as a function of the amount of data it is asked to handle
- For example: O(n) indicates that performance degrades at a linear rate; $O(n^2)$ indicates the rate of degradation follows a quadratic path.

Common growth rates



Serial Search Analysis

- What are the worst and average case running times for linear search?
- What is the best case running time for linear search?
- We must determine the O-notation for the number of operations required in search.
- Number of operations depends on *n*, the number of entries in the array.

Worst Case Time for Linear Search

- For an array of *n* elements, the worst case time for serial search requires *n* array accesses: O(*n*).
- Consider cases where we must loop over all *n* records:
  - desired element appears in the last position of the array
  - desired element does not appear in the array at all

Average Case for Linear Search

Assumptions:

1. All keys are equally likely in a search
2. We always search for a key that is in the array

Example:

- We have an array of 10 element.
- If search for the first element, then it requires 1 array access; if the second, then 2 array accesses. *etc.*

The average of all these searches is:

(1+2+3+4+5+6+7+8+9+10)/10 = 5.5

(1 + 10)/2

Best case 1- worst case 10

Average Case Time for Linear Search

Generalize for array size *n.*

Expression for average-case running time:

$(1+2+\ldots+n)/n = n(n+1)/2n = (n+1)/2$

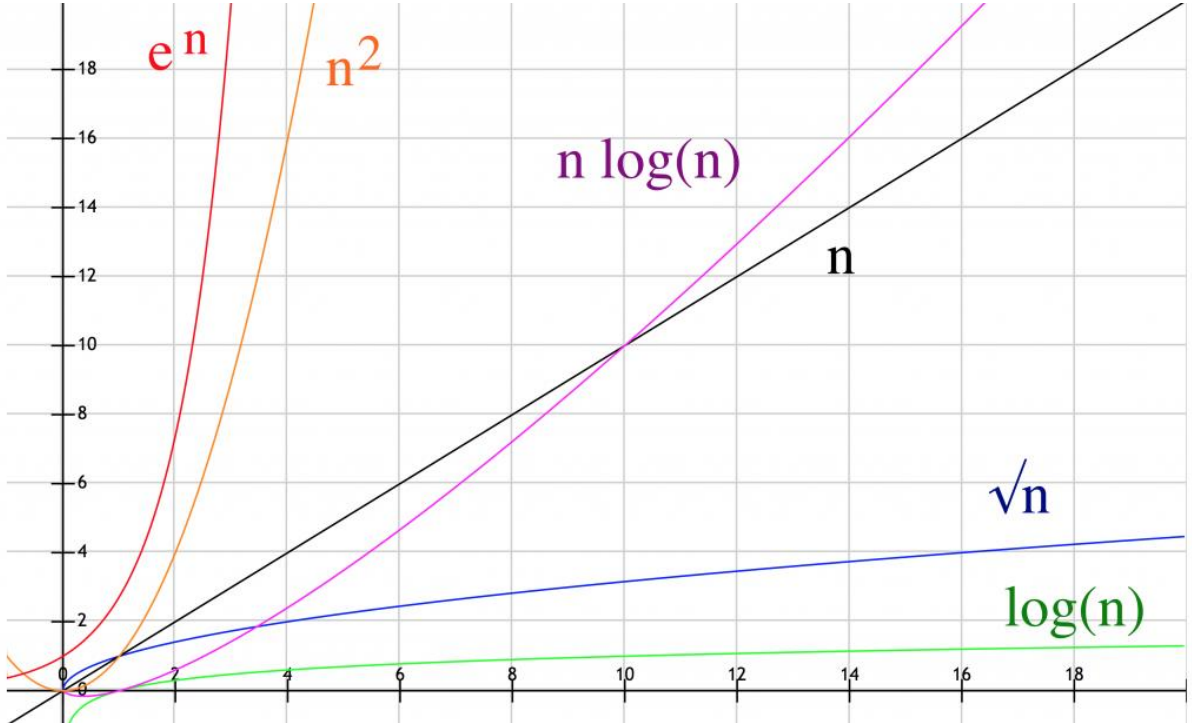
Therefore, average case time complexity for linear search is O(n).

Average Case Time for Linear Search in ordered array?

- Find best, worst and average case

COMPLESSITÀ COMPUTAZIONALE

Definizione



- La **complessità computazionale** identifica le risorse minime necessarie (in termini di tempo di esecuzione e quantità di memoria utilizzata) per la risoluzione di un problema in funzione di un determinato tipo di *input*.

Spazio di esecuzione

- Indica lo spazio di memoria occupato da un algoritmo.
- Ad esempio, se si vuole creare un *array* di *n* elementi, l'algoritmo richiederà uno spazio pari a **O(n)**.
- La creazione di una *matrice* di dimensione *n x n* richiederà uno spazio pari a **$O(N^2)$.**
- L'algoritmo di *ricerca lineare* in un array di dimensione *n*, richiederà uno spazio pari a **O(1)**, poiché non utilizza *strutture dati* **supplementari** per la sua esecuzione.

Tempo di esecuzione

- **O (big O – O grande):** indica **indica il caso peggiore del tempo di esecuzione di un algoritmo**.
- Ad esempio, l'*iterazione* di tutti gli elementi di un array di dimensione pari ad *N*, avrà un tempo di esecuzione pari a **O(N)**, poiché si ha la necessità di passare tutti gli *N* elementi.

Complessità computazionale ricerca lineare

- **Caso peggiore:** l'elemento ricercato è l'ultimo dell'array o non è presente;
- **Caso medio:** se ogni elemento è equamente distribuito, ciascun elemento ha la stessa probabilità di essere trovato (**O(N/2) —> O(N)**);
- **Caso migliore:** l'elemento ricercato è il primo dell'array;

Caso peggiore	**O(N)**
Caso medio	**O(N)**
Caso migliore	**O(1)**

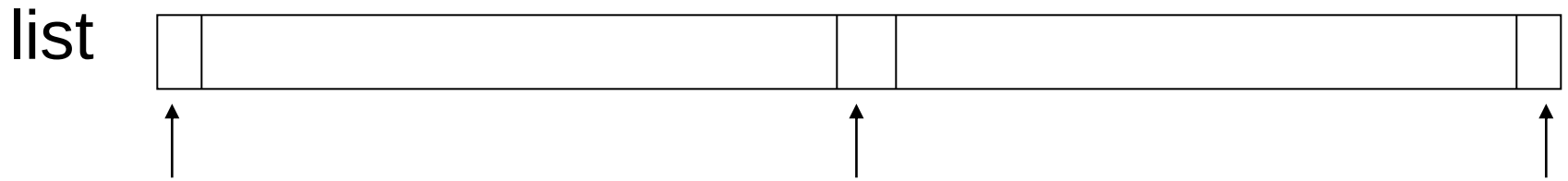
QUESTIONS

BINARY SEARCH

Searching in a Sorted List

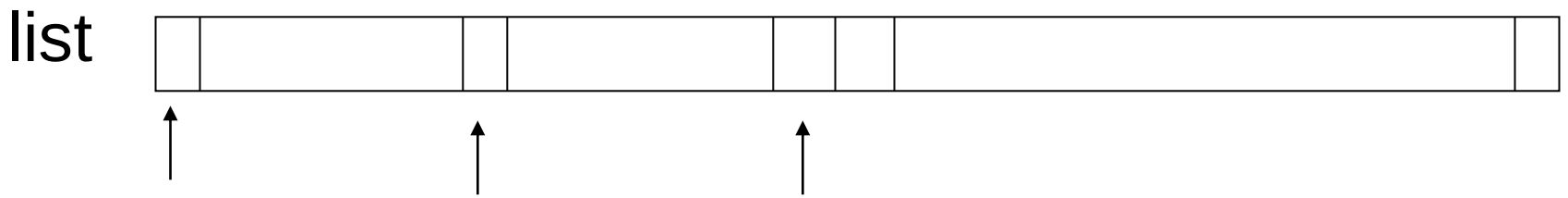
- If the items are sorted then we can *divide and conquer*
- dividing your work in half with each step
  - generally a good thing
- The Binary Search on List in Ascending order
  - Start at middle of list
  - is that the item?
  - If not is it less than or greater than the item?
    - less than, move to second half of list
    - greater than, move to first half of list
  - repeat until found or sub list size = 0

Binary Search



low item middle item high item

Is middle item what we are looking for? If not is it more or less than the target item? (Assume lower)



low item middle item high item

and so forth…

Ricerca Binaria ↔ Binary Search

- Nel caso in cui la sequenza di elementi è ordinata, è possibile utilizzare la **ricerca binaria**.
- Il processo, che continua finché non si trova l'elemento da ricercare o finché non si raggiunge la fine della sequenza è il seguente:
  - Si confronta la *chiave* con l'elemento centrale della sequenza;
  - Se la *chiave* è uguale all'elemento centrale, la ricerca termina;
  - Se la *chiave* è maggiore dell'elemento centrale, si prosegue la ricerca nella sottosequenza di destra;
  - Se invece la *chiave* è minore dell'elemento centrale, si prosegue la ricerca nella sottosequenza di sinistra;

Ricerca Binaria ↔ Binary Search

- Funzionamento della **ricerca lineare** con una **chiave** di valore **37**

Binary Search

```
static int RicercaBin(int[] v,int x)
{
    int pos=-1;
    int inf = 0, sup = v.Length - 1, centro;
    bool trovato = false;
    do{
        centro = (inf + sup) / 2;
        if (v[centro] < x)
            inf = centro + 1;
        else if (v[centro] > x)
            sup = centro - 1;
        else
            trovato = true;
    }while((!trovato )&&(inf<=sup));
    if (trovato)
        pos = centro;
    return pos;
}
```

```
static void Main(string[] args)
int[] vet = { 2, 4, 6, 7, 8, 9, 11, 13 };
int num = 6;
Console.WriteLine(RicercaBin(vet, num));
```

Complessità computazionale ricerca binaria

- **Caso peggiore:** l'elemento ricercato non è presente nell'array. Poiché ad ogni iterazione la dimensione degli elementi analizzati viene dimezzata, verranno effettuati $log_2 N$ confronti;
- **Caso medio:** verranno effettuati circa la metà dei confronti del caso peggiore, quindi $\frac{\log_2 N}{2}$
- **Caso migliore:** l'elemento ricercato si trova nella posizione centrale dell'array;

Caso peggiore	**O(logN)**
Caso medio	**O(logN)**
Caso migliore	**O(1)**

Compare the searching algorithms

Algorithm	Best case	Worst case	Average case	Example Avg if N=1000
Linear search				
Linear search In ordered array				
Binary search				

QUESTIONS

INTERPOLATION SEARCH

Ricerca interpolata

- Migliora la ricerca binaria, restringe ripetutamente la sequenza sulla quale effettuare la ricerca, ma il punto di divisione non è più quello centrale, bensì quello calcolato mediante l'*interpolazione* che permette di determinare la posizione nella quale è più probabile trovare l'elemento.

$$\text{posizione} = \left[inf + \frac{(sup - inf) * (chiave - vet[inf])}{vet[sup] - vet[inf]} \right]$$

```
public static int RicercaInterpolata(int[] arr, int chiave)
{

    int inf = 0, sup = (arr.Length - 1);

    while (inf <= sup && chiave >= arr[inf] && chiave <= arr[sup])
    {
        int pos = inf + (((sup - inf) / (arr[sup] - arr[inf])) * (chiave - arr[inf]));

        if (arr[pos] == chiave)
            return pos;
        if (arr[pos] < chiave)
            inf = pos + 1;
        else
            sup = pos - 1;
    }
    return -1;
}
```

QUESTIONS