

Searching for an element:

There's an important theory in computer science about searching an element.

Given a value x , find out if x is in an array and its corresponding position.

Obviously, you can check the element inside one by one.

This is called **sequential search** (順序搜尋) or **linear search** (線性搜尋).

Namely, it's an $O(n)$ algorithm, which is very slow when many data are needed to found.

Coding: For $k:=1$ to n do if $x = a[k]$ then break; if $a[k]=x$ then writeln('found');

The other more commonly used searching is called **Binary Search**.

It makes use of **random access** and the properties of a sorted array.

First of all, you need to know what **random access** and **sequential access** are.

Random access means you can access an element at will without any limitation.

For example, you can access any element of an array by giving an index.

Hard disk, CD-ROM, floppy also use random access for data storage.

Sequential access allows one accessing data with only two operations either the next or the previous.

Like a tape, you must use forward and rewind or play sequentially to listen what it stored.

Forward and play are in sense the next while rewind is an analogy of previous.

Magnetic tape is the most common for data storage using sequential access.

Binary Search (二分搜尋):

It's a searching performed on a sorted array.

We assume the array is sorted in ascending order.

Left

Middle

Right

Pick up the middle value and compare it with the target x , if they are equal, it's found.

Otherwise, if x is smaller, then we proceed to search the Left side only, else we searching the Right side only. Repeat this process until no more elements are there.

After each comparison, we can drop half of elements.

Therefore, **Binary Search** is far better than sequential search.

For N elements, we need at most $(\lfloor \log_2 N \rfloor + 1)$ comparisons, where $\lfloor x \rfloor$ denotes the floor function of x . (the trunc(x) in Pascal)

Consequently, it's an $O(\lg N)$ algorithm.

As seen from the algorithm, we must use random access to get the value of the middle one. So random access is necessary condition of using binary search.

Coding:

Common Declaration: *To prevent overflow, use $mid := left + (right - left) \div 2$ instead.*

```
const n=10;
  left:integer=1;
  right:integer=n;
  mid:integer=(1+n) div 2;
type arrayA=array[1..n] of integer;  {The type is necessary for passing an array to a function}
var  a:arrayA;
     temp,x:integer;
     found:boolean;
```

Recursive Thinking:

If the middle value is not our target, then we have to find the smaller part.

→ Searching the smaller part is equivalent to searching the original one.

We can break down the problem like this.
Get the recursive formula:
Result of the whole part is the result of the smaller.

Iterative Thinking:

While there are still elements in the search range of an array, check the middle one. And modify the search range.

Recursion version:

```
function b_search(var a:arrayA;var x:integer;left,right:integer):boolean;
begin
mid:=(left+right) div 2;
temp:=a[mid];
if left<=right then
if temp=x then b_search:=true
else if temp<x then b_search:=b_search(a,x,mid+1,right)
else b_search:=b_search(a,x,left,mid-1)
else b_search:=false;
end;
```

While-do version:

```
function b_search(var a:arrayA;var x:integer;left ,right:integer):boolean;
begin
found:=false;
while (left<=right) and not found do
begin
mid:=(left+right) div 2;
temp:=a[mid];
if temp=x then found:=true
else if temp<x then left:=mid+1
else right:=mid-1;
end;
b_search:=found;
end;
```