2 Binary search

Summarising the previous section: we gave the usual method of searching an array of n elements in time $O(n)$. The code assumed an array of integers, but it would work with data of any type.

(2.1) Binary search works with a *sorted* array of n elements and has runtime $O(\log n)$[1]

Of course the array elements must come from an ordered type. Give it a name: `SORTABLE`. Our binary search routine will use a *3-way* comparison because sometimes comparing items can be expensive.

```
int threeway ( SORTABLE x, SORTABLE y ) ...

if x < y, returns a negative value
if x == y, returns 0
if x > y, returns a positive value
```

Example: `strcmp ()` is a 3-way comparison.

(2.2) Binary search program.

```
#include <stdio.h>
typedef int SORTABLE;

int threeway (SORTABLE x, SORTABLE y)
{
  if ( x<y )
    return -1;
  else if ( x == y )
    return 0;
  else
    return 1;
}

int bs_find ( SORTABLE x, int n, SORTABLE a[] )
{
  int i = 0, j = n-1;
  int place = -1;
  while ( i <= j && place < 0 )
  {
    int m = (i+j)/2;
    int comp = threeway ( x, a[m] );
        // negative: x precedes a[m], positive: x follows a[m]
    if ( comp == 0 )
      place = m;
```

[1]The base of the logarithm doesn't matter, but here it will be $\log_2$.

```
    else if ( comp < 0 )
      j = m-1;
    else
      i = m+1;
  }
  return place;
}

main()
{
  int n = 7;
  SORTABLE a[7] = {-4, -2, 0, 1,2, 5, 7 };

  SORTABLE key[3] = {-3, -4, 3};

  int i;
  for (i=0; i<3; ++i)
  {
    int place = bs_find ( key[i], n, a );
    if ( place >= 0 )
      printf("%d is at position %d\n", key[i], place);
    else
      printf("%d is not stored\n", key[i]);
  }
}
---------------------------------
-3 is not stored
-4 is at position 0
3 is not stored
```

(2.3) *Correctness.* There is an invariant condition ensuring correctness:

> *if* x occurs in the array *then* it occurs between i and j, i.e., $x = a[r]$ for some r where $i \leq r \leq j$.

This is preserved. If $i \leq j$ and the range of possible indexes where x occurs is between i and j, then this range can be split as shown:

$$i \ldots m-1, \quad m, \quad m+1 \ldots j.$$

If $a[m] == x$ then the loop breaks and location m is returned.

Otherwise suppose that x does occur, so $x == a[r]$, say, with $i \leq r \leq j$.

The array is sorted, so if x precedes $a[m]$ then x precedes every item in $a[m \ldots j]$ and therefore $x \in a[i \ldots m-1]$ and j is correctly updated.

Similarly, if x follows $a[m]$ then i is correctly updated. Thus the condition: if x occurs in the array then it occurs between i and j, is preserved.

(2.4) *Runtime.* The *search range*, the range of positions where x can be found, is $i \ldots j$ and its size is $j - i + 1$. Roughly speaking, the range is halved at each step.

Now

$$m = \lfloor \frac{i+j}{2} \rfloor$$

(m is rounded down, though the code would work just as well if m were rounded up.)

The new search range has size

$$(m-1) - i + 1 \quad \text{or} \quad j - (m+1) + 1:$$
$$m - i \quad \text{or} j - m;$$
$$\frac{i+j-1}{2} \leq m \leq \frac{i+j}{2}.$$

If x precedes $a[m]$, the revised range has size $m - i$. Since $m \leq (i+j)/2$, the revised range has length at most

$$\frac{i+j}{2} - i = \frac{j-i}{2} \leq \frac{1}{2} \times (j - i + 1).$$

If x follows $a[m]$, the revised range has size $m - i$ and

$$m \geq \frac{i+j-1}{2}$$

so the revised range has length at most

$$\frac{j-i+1}{2} \leq \frac{1}{2}(j-i+1)$$

In all cases, if $R = j - i + 1$ and S is the size of the revised range,

$$S \leq R/2.$$

The initial range size is n. After r iterations, if there are r iterations, the revised range is at most

$$2^{-r} n$$

So if $2^{-r}n < 1$, the range is empty and the loop breaks. This will certainly happen if

$$n + 1 \leq 2^r \iff$$
$$r \geq \log_2(n+1) \iff$$
$$r \geq \lceil \log_2(n+1) \rceil$$

So the maximum number of iterations is at most $\lceil \log_2(n+1) \rceil$, so

(2.5) Corollary *Binary search has runtime* $O(\log n)$.