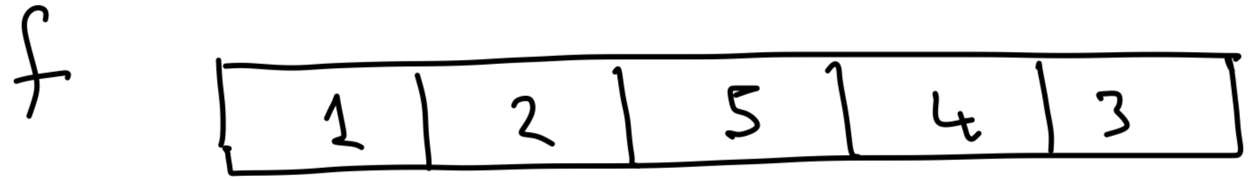
Lecture 7

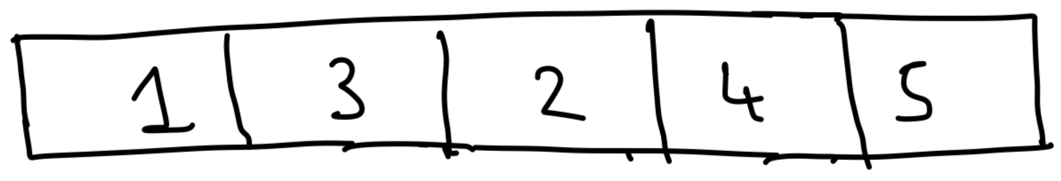
A Challenge

Suppose we have f[5] of int which contains single digit values.
We can read it like a single number.



12543

Suppose we decide to rearrange it so it reads like the next highest number that we could make using these digits.



13245

How would we do this?

Let us analyse the problem and try to learn about it. For now we ignore the fact that we are using an array and just study some numbers. Let us look at the number

9876543210

Notice that the digits are ordered in descending order from left to right. This is actually the **largest** number you could make using this collection of digits, so there is no next highest number that can be made.

Now let us look at another number.

9054387621

This isn’t the largest number that can be made so we must try to see how to make a higher one. Before we do so, I want to concentrate on the last 3 digits

621

Could we rearrange them to get a higher number than 621? No we couldn’t.

Look at the last 4 digits

7621

Could we rearrange them to get a higher number ?
No, once again we couldn’t.

Look at the last 5 digits

87621

Could we rearrange them to get a higher number? No, that is also not possible.

What each of these examples 621, 7621 and 87621 have in common
is that the digits in each number are ordered in descending order.

This is something important that we have discovered so we make a note of this.

Fact 1: When the digits of a number are arranged in descending order, from left to right, then that number is the highest (biggest) number that can be made using those digits.

Often, when you discover one interesting fact, there may be another one close by. We look once again at the number

621

and we ask, what is the smallest number we could make using these 3 digits? The answer would be

126

And what is the smallest number we could make using the last 4 digits

7621

It would be

1267

And of course if we were looking at the last 5 digits

87621

Then the smallest number we could make would be

12678

What these smallest numbers have in common is that the digits in each of the numbers are ordered in ascending order.
So we note this as another interesting fact.

Fact 2: When the digits of a number are arranged in ascending order, from left to right, then that number is the lowest (smallest) number that can be made using those digits.

Now, let us return again to our number

9054387621

Let us look this time at the last 6 digits

387621

This is not arranged in descending order. So this means it must be possible
to make a higher number. We split it into 2 sections to look closely at it

3   87621

The next highest numbers we can get using these digits must begin with what digit?

It must be bigger than 3

and it must be in the digits in the sequence 87621

The next digit bigger than 3 will be 6.

As 87621 is arranged in descending order we search from the right until we get the first digit that is bigger than 3.

Now we swap those digits and we get

687**3**21

But we are not finished yet. Look again at the final 5 digits on their own

    6   87321

They are in descending order so they are the **biggest** number possible using these digits. But we now want to start with the **smallest** number possible using these digits. So to get that we reverse the last 5 digits

    6   12378

So the next highest using these digits is

    612378

So returning to our original number, the next highest number we can get by rearranging

9054387621

is

9054612378

Algorithm.

Now, based on our analysis of the problem, we can start to design an algorithm to generate the next highest number. We assume the current number is stored in an integer array f[10]

Find the largest index i where f[i-1] < f[i] being careful to remember that the current number represented by f could be the largest and so such an index i would not exist.

Assuming we did find such an index, we now do this….

Find the largest index j in f where f[i-1] < f[j]

Swap the values in f[i-1] and f[j]

Reverse the final part of the array starting at index i.

Find the largest index i where f[i-1] < f[i] being careful to remember that the current number represented by f could be the largest and so such an index i would not exist.

Look for something which might not be there.....sounds like a Bounded Linear Search to me.

Find the largest index j in f where f[i-1] < f[j]

We know such a value must exist so that sounds like a Linear Search.

Suddenly this problem appears a lot easier than we thought it was.

Assignment 2

- Read in 10 single digit values into an array f[10].
- Rearrange the values in f to get the next smallest value if it exists.
- Print out the new number.
- Deadline Wednesday 10th April.

Binary Search

- This is another way to search.
- It requires some special conditions if we want to use it.
- It is very efficient when it can be used.

Suppose we have an array with values in it

int f[100]

We also have an int x which also contains a value

I am only going to tell you 2 facts about f

f[0] <= x

x < f[99]

I want you to write statements to find an index i in the array where

f[i] <= x && x < f[i+1]

Does such a place exist?

We know that f[0] <= x
, suppose x < f[1], then such a place does exist.

But suppose f[1] <= x, then what?
Well suppose x < f[2], then such a place exists.

But suppose f[2] <= x, then what?
We could check f[3], if f[3] <= x then such a place exists..

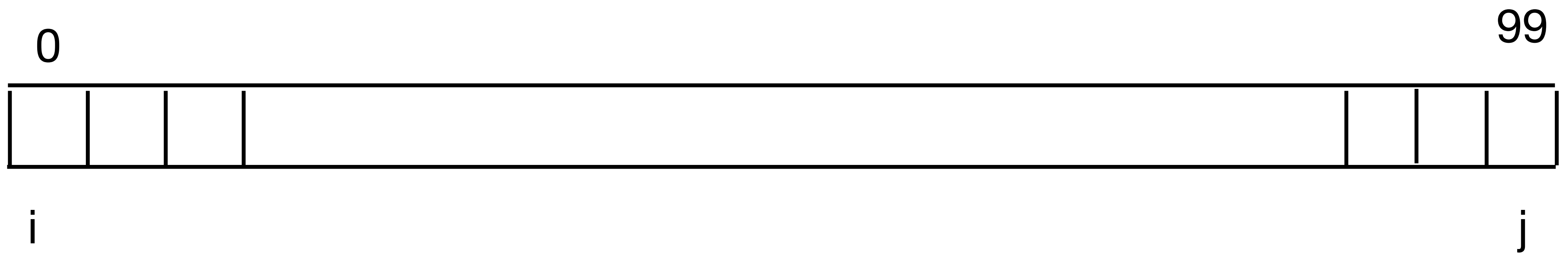
but .....suppose we keep looking and are not finding a place
i where f[i] <= x. Finally we would arrive at position f[98]

suppose f[98] <= x, then we already know x < f[99]

So this tells us that the sort of place we are looking for will exist.

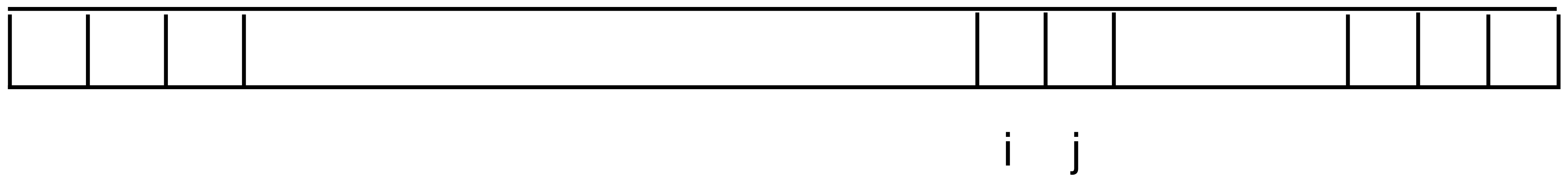
Of course, many such places may exist. We are just asked to find one
of them.

Now, let us consider what is true at the start



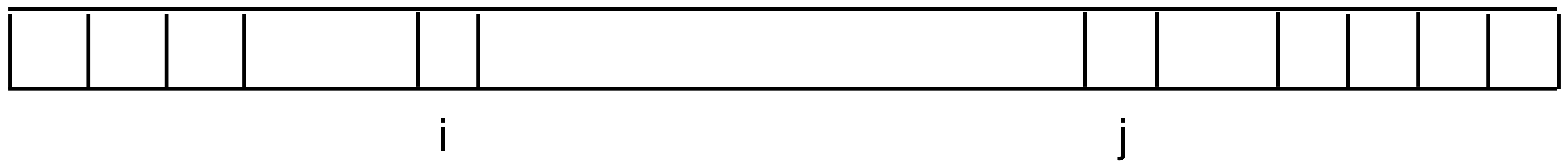
f[i] <= x && x < f[j]

The situation at the end when we have found
a pair of adjacent locations with the property we want



f[i] <= x && x < f[j]

At some stage after starting our search, but before finding a pair of locations side by side.



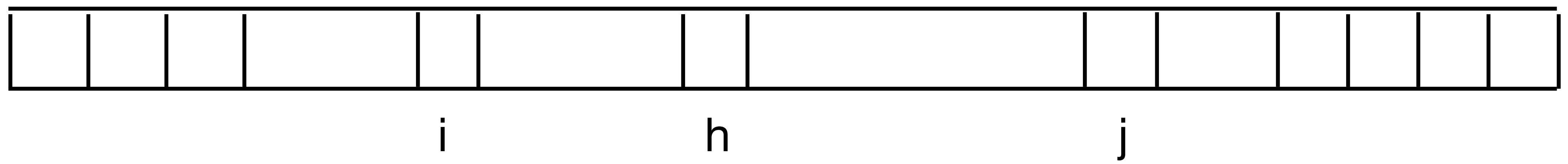
```
f[ i ] <= x  &&  x < f[ j ]
```

The next thing you would want to do is to try to move i and j closer together. But you can only do so if you keep the picture true. Let us see if we can do so.

So, we want to move i and j closer together. Suppose we consider some index h which is between i and j

What would allow us to move i up to h while still keeping the picture true?

What would allow us to move j down to h while still keeping the picture true?



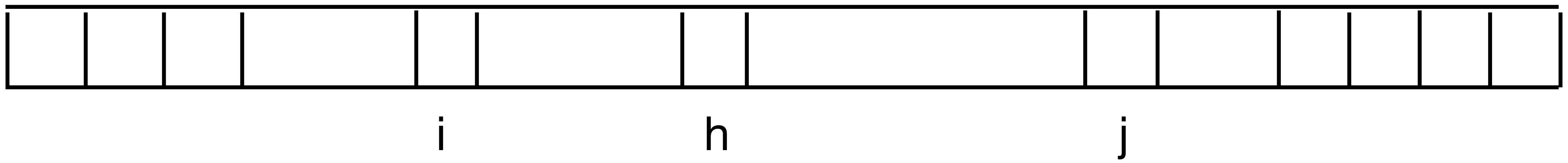
```
f[ i ] <= x  &&  x < f[ j ]
```

What would allow us to move i up to h while still keeping the picture true?

If f[h] <= x

What would allow us to move j down to h while still keeping the picture true?

If x < f[h]



f[i] <= x && x < f[j]

So we seem to have a program like this

```
   int i ;
   int j ;
   int h ;
   int x ;
   int f[100] ;

// Assume f[100] has values and x has a value and
// f[0] <= x and x < f[99]
```

```
   i = 0 ;
   j = 99 ;

// f[ i ] <= x && x < f[ j ]

   while ( j != i+1 )
   {
     // choose h between i and j
     if ( f[ h ] <= x)
       { i = h ; }
     else if ( x < f[ h ])
       { j = h ; }
   }

// f[ i ] <= x && x < f[ j ] && j = i+1
```

It just remains for us to “choose h between i and j”

I could choose h to be i+1

I could choose h to be j-1

But let us try to be as symmetric as possible and
choose h = (i+j)/2

This chooses a h as close to the middle between i and j
as we possibly can.

```
   int i, j, h, x ;
   int f[100] ;

// Assume f[100] has values and x has a value and
// **f[0] <= x and x < f[99]**

   i = 0 ;
   j = 99 ;

// **f[ i ] <= x && x < f[ j ]**

   while ( j != i+1 ) ;
   {
     h = (i+j)/2 ;
     if ( f[ h ] <= x)
       { i = h ; }
     else if ( x < f[ h ])
       { j = h ; }
   }

// **f[ i ] <= x && x < f[ j ] && j = i+1 so**
// **f[ i ] <= x  &&  x < f[ i+1 ]**
```

This gives a very efficient program as the amount of the array to be searched is halved each time we execute the loop.

If, in addition to this, we also know that the values in the array are ordered in ascending order, then there will only ever be one index i where

```
   f[ i ] <= x  &&  x < f[ i+1 ]
```

So, in this case if x is actually stored in f it will be at position i.

To try to write this using a for loop would not be very beautiful.