Algorithms

Table of Contents

1 What are they?

An algorithm is a way of doing something. It is a method, or recipe, or set of instructions to follow to solve some problem.

For example, suppose we have two pieces of data, x and y, and we need to exchange their values. As a test case:

before: x=3 and y=7

after: x=7 and y=3

This needs to work with *any* values. Whatever x and y are before, after they must have the values changed over.

How to do this? What algorithm to use?

We might say:

x ← y // this means copy the value of y to x

y ← x

Does this work? We can check using a *trace table*. We work this out on paper, keeping track of the values of the variables at each step, like this..

	x	y	
Before	3	7	start values
after x ← y	7	7	the 7 was copied to x
after y ← x	7	7	the 7 was copied back to y
At the end	7	7	

So, this does not work. When we copied 7 to x, we lost the initial value of x.

One way which does work is to use an extra storage space, which we will call 'temp', because it is only for temporary use:

temp ← x

x ← y

y ← temp

We check this with a trace table:

	x	y	temp	
Before	3	7	??	It makes no difference what temp is
after temp ← x	3	7	3	the 3 was copied to temp
after x ← y	7	7	3	the 7 was copied to x
after y ← temp	7	3	3	temp (start value of x) copied to y
At the end	7	3		Values exchanged

So this algorithm will exchange two data values.

Algorithm steps

So an algorithm is a sequence of steps, each step being an 'instruction'. These instructions become instructions to the computer in the program, so we can only have very simple steps - only the following are allowed:

1. Moving a value from one location to another, or a constant. So for example x ← y or x ← 3
2. Doing arithmetic. For example x ← 2*3-5 (the * means multiply)
3. Decisions based on simple logic . For example if x > 10 then x ← 2*3-5
4. Loops. For example do 10 times x ← x+1
5. Input and output (I/O) .

The input might come from the keyboard, or from a file of data. The output might be displayed on a screen, or on a printer, or written out to a file. We often ignore I/O in an algorithm. We take it that we already have the data the algorithm operates on. And we do not always output the results. We might, for example, use the results in another algorithm.

We cannot use instructions which require intelligence, like 'solve $x^2+3x-4=0$'. If we say something like 'test if x is a prime number' that in fact is using another algorithm, and we should give that algorithm, to say *how* to test if a number is prime.

The arithmetic must be basic, like add subtract multiply and divide. Actions like find the sine of an angle are algorithms in themselves.

We can only have a finite number of instructions, and the algorithm must end after a finite number of steps, with the correct result.

Pseudo-code

We write down algorithms in *pseudo-code*. 'pseudo' is pronounced 'sudo' - the p is silent. It means 'like, but not real'. So pseudo-code is like real program code, but not actual program code. We use pseudo-code because

1. It is easier to read and understand than actual code
2. It is less strict than code
3. It can easily be converted to any programming language

Algorithms and programs

Computers cannot run pseudo-code algorithms. They can only run actual programs, written in some programming language.

Algorithms should be *language-agnostic*. That means, they should not depend on any particular programming language. They will work in any language.

Data structures

The algorithm to process some data depends on what *data structure* it is in - a list, a stack, a tree or wherever. As as result, algorithms and data structures are usually studied together. In this text, we focus on search and sort algorithms, in a simple data structure, an array.

We look at other data structures and their algorithms in another text.

Test

1. Explain what an algorithm is.

2. List the advantages of using pseudo-code

3. Suppose we have 3 variables, x y and z. We want to cycle their values. So the value of x goes to y. The old value of y goes to z. z goes to x.

Make an algorithm to do this. Write it in pseudo-code

Use a trace table to show it works.

2 Time and Space Complexity

We often have a choice of using different algorithms. How to decide which is better? Remember all algorithms must 'work' - that is, end with the desired result. Otherwise they do not count as an algorithm.

An algorithm is better if

1. it is faster
2. it uses less memory

For a small number of data items, any algorithm is OK. It is only for a very large amount of data does speed and memory use matter.

The time complexity of an algorithm is how many steps it takes to finish, depending on the number of data items - normally called n. So we need to know how many steps, for *very large n*. If n is small, and an algorithm takes 10 or 20 microseconds, we just do not care. If n is a million, and it takes 2 hours, we do care. The step count for very large n is called the *asymptotic behaviour*.

Obviously any algorithm will be faster on a faster computer. So we ignore the actual time, and just count the steps. That way we can compare algorithms on different computers.

As an example - suppose an algorithm takes 3n+20 steps to execute, with n data items.

If n =100, this is 320. If n=1000, this is 3020. If n=1000000, it is 3000020. For large n, the +20 is negligible. So we ignore it.

So the actual time taken is 3nt, if each step takes t seconds. But on a faster processor t is less - whatever the algorithm. So the *stepcount* is 3n. If we double n, 3n doubles. If we triple it, it triples. So it is proportional to n. We say this is O[n]. This is *'big-O' notation*. O[n] is linear time.

$O[n^2]$ is slower. For 100 times the data, it takes 10000 times longer. This is *quadratic time*. Often this is too slow.

Some algorithms do not depend on how much data there is. For example, accessing an array element does not depend on how big the array is. This is 'constant time', O[1].

Some fast algorithms work by halving the data. So if we have 64 pieces of data, it halves them (32) and again (16) and 8,4,2,1. That is 6 halvings -so 6 steps. This is $O[\log_2 n]$, because log2 n is how many times we can halve n.

To handle 2000 items - that is just 1 step more than 1000 items.

The *space complexity* is how much memory it uses. Some algorithms just move data around in the data structure. These work 'in place' and use no extra memory beyond what the data itself needs. Other algorithms move the data into a different structure - these use twice the memory of the initial data structure.

Usually we can trade off speed and memory, so that an algorithm which uses more memory is faster.

Test

1. Suppose the time complexity of an algorithm is O[n^2]. What does the 'n' mean?

2. If the time complexity of an algorithm is O[$\log_2 n$], and 2000 items need 100 steps - how many steps would 8000 items take?

3 Search algorithms

In a search algorithm, we have a key field, and we want to find the matching record in some data structure - to fetch the value fields in that record. For example we might have a student id and want to find their phone number.

The search result might be that the key is not present in the data.

Linear search

Linear means 'in a line'.

This applies to lists of data. We simply start at the beginning of the list, and test elements in turn until we find it, or we reach the end without finding it.

In pseudo-code:

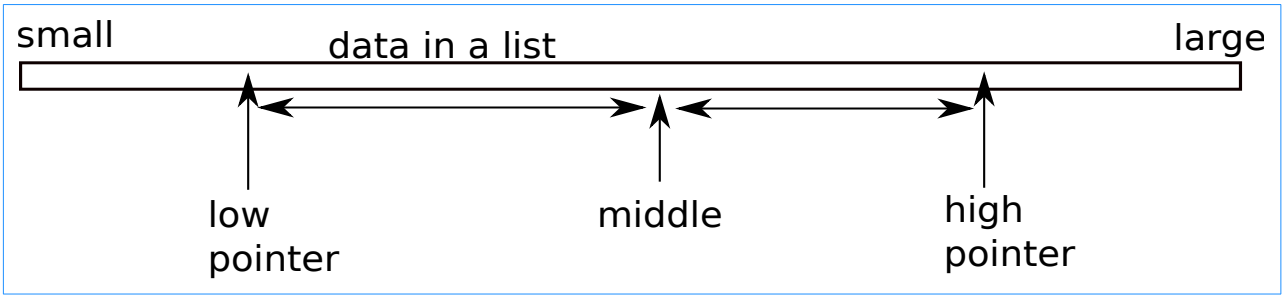
```
set index = 0
loop start
  if data at index == target
    end loop - target found
  index = index + 1
  if index passed end of list
    end loop -target not present
loop end
```

In other words, we start at the beginning of the list, and check items in turn, until we find it, or hit the end without finding it.

What is the *time complexity* of a linear search? Suppose the list is n=1000 items long. We might be lucky and find it at the first item - so just 1 step. Or unlucky, and find it at the last item, so 1000 steps. So it will be between 1 and 1000. On average, with random data, we will find it half way through, so n/2 steps. So the step count is proportional to n. Twice the amount of data means twice the step count. So this is *O[n]*. We would like a faster algorithm than this.

Binary search of a list

This is only for data which an *ordered* list:



We have two pointers to select out a range of the list. These pointers can be just indexes into an array.

We have a loop:

> We work out the middle, between high and low, and check the data there.
>
> If it is too big, the target must be in the lower half. We change high to middle and carry on, using the lower half.
>
> If it is too small, the target is in the upper half. Change low to middle and continue
>
> If the data at middle == the target it is found : finish

We initialise low to be the bottom of the list, and high to be the top.

What is the time complexity of a binary search? It is how often we can divide the list by 2. So it is $O[log_2\ n]$

This is very fast. If we are searching 1 million items - then we can search 2 million instead with just one more step.

But the data must be *ordered*.

Test

1. Write a program, in any language, which does teh following. Test each step before doing the next:

1. Fills an array with 10 random integers
2. Prints them out
3. Assigns a value to search for, at random
4. Does a linear search for the target, outputting the location if found, and a suitable message if it is not present.
5. Adapt the program to count, and output, the number of comparisons made
6. Adapt the program so it works not just for 10 integers, but for a value n, which can be altered.
7. Run it for n=10, 20, 30,..100 integers. Draw a graph of n against the number of comparisons.

2. Write and test a program which

1. Fills an array with integers, as follows: the first element should be zero, and every other element is a random amount up to 4 greater than the previous element.

2. Prints out the array.
3. Assigns a target, and does a binary search on the array to find it.

4 Bubble sort

To sort here means to put into order - increasing or decreasing.

For example 4 3 6 2 5 is not sorted. If we sort that data, we get 2 3 4 5 6 .

We have the data in a list - maybe an array.

We go through the list, comparing each number with the next, and exchanging them if they are in the wrong order. For example

4 3 6 2 5	swap
3 4 6 2 5	leave
3 4 6 2 5	swap
3 4 2 6 5	swap
3 4 2 5 6	finished this scan

After this one number (here 6) will be moved to its correct position. But others (like 2) are in the wrong position. One scan gets one number correct - so to get all n correct, do it n times:

n times:
  go through the list and swap each with the next if wrong order.

It is called a bubble sort because on each scan a number 'bubbles' up to the end.

Pseudo-code to sort an array d of n items, with indexes 0 to n-1, into increasing order, is

```
repeat n times:
  for i=0 to n-2 // scan through the array
    if d[i] > d[i+1] // compare each item with the next
      temp=d[i]  // swap them
      d[i]=d[i+1]
      d[i+1]=temp
  next i
```

What is the time complexity? Each scan requires n comparisons. In fact n-1, but for large n, this is about the same as n. We do n scans. So there is a total on n^2, and this is $O[n^2]$

This is a basic bubble sort. We can make two improvements

1. Each scan puts the largest item at the end, in its final place. This means the next scan can be one step shorter. Successive scans can get shorter and shorter.

2. We might get lucky, and have the array sorted before we end. This would be true immediately if teh data was already sorted. So, we

have a flag, tracking if we have made a swap. If we do a scan and no swap is needed, we stop.

```
for end=n-2 down to 0
  swapFlag = false
  for i=0 to end // scan through the array
    if d[i] > d[i+1] // compare each item with the next
      temp=d[i]  // swap them
      d[i]=d[i+1]
      d[i+1]=temp
      swapFlag = true
    if swapFlag == false end
  next i
next end
```

To start with, we might have 3 kinds of data:

1. In random order
2. In reverse order
3. In already sorted order

For cases 1 and 2, the time complexity is O[n^2]. For case 3, for the improvement as here, we just do one scan, so it is O[n]

Test

1. On paper, do a bubble sort of 7 2 8 3 4

2. Write and test a program (any language) which

    1. Fills an array with 10 random integers
    2. Prints them out
    3. Sorts the array with a bubble sort
    4. Prints them out again

5 Insertion sort

An insertion sort is a general type of sort. The idea to take each data element in turn, and insert them into something in the correct place, so they will be in order.

One version is to do this 'in place'. So we have just one array, and insert items at the front. So, we

1. Find the smallest element. Put it at position 0, swapping it with what was at 0

2. Find the smallest, from 1 to the end. Swap it with position 1

3. Find the smallest, from 2 to the end. Swap it with position 2

4. And so on

How do we find the smallest? We use the idea of the smallest so far. We also need to track where it is. For integers in an array d with index 0 to n-1:

```
smallestSoFar=d[0]
where =0
for i=1 to n-1
  if d[i]<smallestSoFar
    smallestSoFar=d[i]
    where=i
```

After this, the smallest is smallestSoFar, and it is at position 'where'

We can use this to do an insertion sort on the array d

```
for start=0 to n-2 // repeat scan, beginning at start
  smallestSoFar = d[start] // initialise
  where=start
  for i=start+1 to n-1 // look at all the rest
    if d[i]<smallestSoFar
      smallestSoFar=d[i] // found a smaller one
      where = i
  temp=d[start] // exchange value at start
  d[start]=smallestSoFar // with smallest one
  d[where] = temp
```

The first scan covers n items. The last is just 2. The average scan length is n/2. We do it n times. So this is $n^2/2$ steps, so this is $O[n^2]$

Test

Write an insertion sort program.

6 Merge sort

To start with, suppose we have a function that merges two sorted lists into a single sorted list. So for example with input

list 1 : 2,4,7,8,9,12,15

list 2 : 3,5,6,10,13

the output is: 2,3,4,5,6,7,8,9,10,13,15

How would this work? We start comparing the heads of the 2 lists - 2 and 3. 2 is less, so put it to the output and remove it from list1. Then compare 4 and 3. 3 is less so output it and remove from list2. Now compare 4 and 5. 4 is less so output.. and so on. Continue until one input list is empty, and just move the rest of the other list to output.

Then, we use this function as follows. The input is an unsorted list - maybe

9,2,5,8,1,3,7,4,3,3,8

Treat this as a sequence of lists, 1 element long, and merge them pairwise.

9	2	5	8	1	3	7	4	3	3	8
pair 1 - merged below		pair 2		pair 3						
2	9	5	8	1	3	4	7	3	3	8

Then we take pairs 2 elements long, and merge them:

2	9	5	8	1	3	4	7	3	3	8
pair 1				pair 2				pair 3		
2	5	8	9	1	3	4	7	3	3	8

Then pairs 4 elements long. In fact, one pair with 4 elements. The other pair has length 3 and length 0

2	5	8	9	1	3	4	7	3	3	8
pair 1								pair 2		
1	2	3	4	5	7	8	9	3	3	8

(pair 4 has zero length). Then pairs 8 elements long. In fact we have one length 8, merging with one length 3

1	2	3	4	5	7	8	9	3	3	2
pair 1										
1	2	2	3	3	3	4	5	7	8	9

One scan through takes n steps, through the whole list. How many scans? The sublist lengths were 1,2,4,8... Going backwards, this halves the list each time, and the number of scans is how often we can divide n by 2 - which is log2 n.

So the total steps is $O[\ n \log_2 n\ ]$. This is much faster than a bubble sort.

Test

Write a program which does a merge sort on an array of random integers