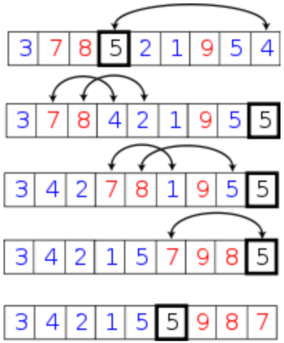
CI583: Data Structures and Operating Systems



What are you studying, and why?

Data structures and algorithms, with a focus in the latter part of the module on their application in operating systems.

Data structures and algorithms **are** programming!

They provide the fundamental *problem-solving tools* for the computer scientist and software engineer.

What are you studying, and why?

Choosing the right **representation** of a problem (the data structure) and **strategy** for attacking the problem (the algorithm) is the bulk of the work of finding a solution.

On this module, you will learn to recognise and understand these standard tools, enabling you to make effective use of them in their own work.

What are you studying, and why?

It’s tempting to think of this as a ”solved problem”:

When I want to use a stack|hashtable|tree, I use the one from my standard library. What kind of idiot writes their own?

This isn’t hard to rebut.

What are you studying, and why?

- there are many open problems in this subject, as we'll see, such as finding new algorithms for parallel computing,
- the programmer who really understands these concepts can make best use of them, even if that means picking the right data structure from the standard library and implementing the right well-known algorithm.

In fact, whatever kind of programming you do, you need to *invent* your own algorithms and *reason* about their performance.

What will you learn?

By the end of the module you should be able to:

- Assess how the choice of data structures and algorithm design methods impacts the *performance* of programs.
- Choose the *appropriate* data structure and algorithm design method for a specified application.
- Write programs using *object-oriented design principles*.

What will you learn?

By the end of the module you should be able to:

- Solve problems using data structures such as linked lists, stacks, queues, hash tables, binary trees, heaps and binary search trees.
- Solve problems using algorithm design methods such as the greedy method, divide and conquer, dynamic programming and backtracking.

How will you learn it?

Contact time is organised as follows:

- Weekly lectures, which provide the theoretical basis for the various topics. These will be available as videos/slides on studentcentral, and I will be online (on MS Teams) to discuss the material during the timetabled slot.
- Weekly lab sessions in which you will be able to experiment with and implement some of the ideas covered in lectures. Once the assignment has been given out, the labs can also be used as a clinic for that. Labs will be face-to-face (ie on campus).

How will you learn it?

Equally important is the **independent study** we expect you to do – this should come to no less than the standard minimum of **4 hours a week**, though you will need to put more time in at various points, and how you organise it is up to you.

Hint: If you do not put in this independent study time, you will probably struggle to keep up with the material.

How is it assessed?

100% on the programming assignment and reflective report, handed out roughly one third of the way through the module.

This is a programming problem, in which you will implement some of the data structures and algorithms described in the lectures.

You will also submit a reflective report on the complexity of the algorithms involved and of your solution.

There is no exam on this module.

Module overview

We will look at differing approaches to *representing data*.

These include *linked lists*, *arrays*, *stacks*, *queues*, *trees*, including *binary trees*, and *search trees*.

We will look at the pros and cons of each, and how to implement them.

Module overview

We will look at a variety of approaches to *searching, sorting and selecting data*.

In the process of doing this we will consider algorithmic strategies such as *the greedy method*, *divide-and-conquer*, *dynamic programming*, and *backtracking*.

We will also see how we can use *chance* to design elegant algorithms.

We will look at ways of *analysing the performance* of algorithms using simple mathematical methods.

Resources

Representative books and a web resource:

- Goodrich and Tamassia, Data Structures and Algorithms in Java (4th edition), John Wiley \& Sons.
- Cormen et al., Introduction to Algorithms (3rd edition), MIT Press.
- Part of an online course from Stanford University: https://www.coursera.org/course/algo

For the mathematically-inclined completist:
http://www-cs-faculty.stanford.edu/~uno/taocp.html

Resources

It is always helpful to be able to visualise new data structures. When you encounter a new one you should have a play with it – https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

Demo

Introduction

An *algorithm* is simply a *finite* list of precise instructions designed to accomplish a particular task.

We will sometimes present implementations of a given algorithm using Java, and sometimes using *pseudo-code*.

Pseudo-code

Pseudo-code gives the logic and control flow of a program.

It is not intended to be in any particular language, but hopefully you could easily translate it into any that you know.

```
-- Find the largest natural number that divides both a and b
-- without leaving a remainder.
function gcd (a, b)
  while b != 0
    if a > b
      a <- a-b
    else
      b <- b-a
    endIf
  endWhile
  return a
end
```

Introduction

A **data structure** is an object that collects related *data* and *behaviour*, such as a Java class.

An **abstract** data structure (ADT) defines data and behaviour that is common to a number of **concrete** data structures.

Abstract and concrete data structures

```
abstract class Shape {
  public abstract double area();
}
class Square extends Shape {
  double width;
  public double area() {
    return width*width;
  }
}
class Circle extends Shape {
  double radius;
  public double area() {
    return PI * (radius * radius);
  }
}
```

Simple collections: array

Probably the simplest data structure that represents a collection of values is the *array*.

An array is a collection with a fixed size (determined when the array is created).

In typed languages (such as Java) each element in the array has the same type.

Simple collections: array

We access elements in the array using an *index*, a number that identifies the position in the array.

We start counting at zero, so valid indices are between 0 and one less than the length of the array.

Simple collections: array

From a low-level point of view, we can think of an array as a convenient way to access a series of memory locations.

From a higher-level, we might think of the array as a series of ”letter boxes” or ”pigeon holes”.

Simple collections: array

Given an array, `a`, with 10 indices, we access the *ith* element as `a[i]`.

The first element is `a[0]` and if `i>9`, we get a **runtime error**.

Accessing an element has a fixed cost and is very efficient – getting the value of the 10th element has the same cost as getting the value of the 1st.

Simple collections: linked list

An equally simple data structure is the *linked list*.

A linked list is a collection with no fixed size. In typed languages, all elements must have the same type.

When we create a new list, it is empty. Then we can *cons* (add, insert) elements to the *head* of the list.

The head is the first (most recently consed) element of the list. Everything after that is called the *head*.

Searching

Suppose we have an array containing unsorted data and we need to find a particular element, x.

23	7	5	31	9	18	4	32	6

Our only option is to examine each element in the array, `y`, and check whether `x=y`.

As simple as it is, this is our *algorithm*, called *sequential search*.

How many steps will this take for an array of length n?

Searching

To see how many steps it will take to search our array (of length n) for an element, x, there are several cases we need to consider.

23	7	5	31	9	18	4	32	6

- The *best* case.
- The *worst* case.
- The *average* case.

Searching

In the best case scenario, x is the first element in the array.

This will take us one step for any value of n.

In the worst case scenario, x is the last element in the array, or is not found. This will take n steps.

Searching

The average case is harder to reason about.

It is sometimes important to consider, but we normally categorise algorithms by the *lower and upper bounds* of their performance.

Often the lower bound (best case) is not that revealing, because we can't rely on getting lucky!

Searching

What if we are able to guarantee that the array will be **sorted** before we start the search?

4	5	6	7	9	18	23	31	32

Then we can come up with better algorithms to do the searching.

In particular, as soon as we get to an element greater than the one we're looking for, we can give up.

Searching

```
-- return the position of x in the array, a, or -1
-- if x is not in a
function search(x, a)
  i <- 0
  while i < length(a)
    if a[i] = x
      return i
    elif a[i] > x
      return -1
    endif
    i <- i+1
  endwhile
end
```

Searching

4	5	6	7	9	18	23	31	32

What are best and worst cases for the new algorithm?

Searching

4	5	6	7	9	18	23	31	32

Unchanged!

However, the *average case* will be improved.

Searching

4	5	6	7	9	18	23	31	32

Let’s try again.

What if we start in the *middle* of the array?

Then either we find x straight away, or the element we’re looking at is greater than or less than x.

Searching

In either case, now we only need to consider half of the array.

At one step, we have halved the size of the problem. We can then apply the same step repeatedly.

This is called *binary search*.

Binary search

Searching the list when x=5

4	5	6	7	9	18	23	31	32

Step 1

- Pick the middle element (`n/2`), and call it `y`.
- `y` is greater than x, so ignore everything to the *right* of `y` and search again.

Binary search

Searching the list when x=5

4	5	6	7	9	18	23	31	32

Step 2

- Pick the new middle element and call it y.
- Again, y is greater than x, so ignore everything to the *right* of y and search again.

Binary search

Searching the list when x=5

4	5	6	7	9	18	23	31	32

Step 3

- Pick the new middle element and call it `y`.
- This time, `y = x` and we are done.

Binary search

Binary search **halves** the size of the problem at each step.

It performs incredibly well:

searching a list of one million items won’t take more than twenty steps.

Binary search

Steps required to find an element in an ordered array of length n.

n	**Steps**
10	4
100	7
1000	10
10,000	14
100,000	17
1,000,000	20
10,000,000	24
100,000,000	27
1000,000,000	30

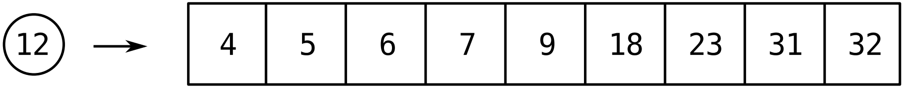
You can check this by repeatedly halving n until it is too small to divide further.

Binary search

```
-- Find the position of x in the array a
-- or -1 if x is not found
function bsearch (x, a)
  start <- 0
  end   <- length(a)
  while start <= end
    middle = (start + end) / 2
    if a[middle] < x
      start = middle + 1
    elif a[middle] = x
      return middle
    else -- must be a[middle] > x
      end = middle - 1
    endif
  endwhile
  return -1
end
```

Binary search

So why don't we make all our arrays sorted? Consider the cost of inserting an element:



This is now an expensive operation that may require relocating many elements.

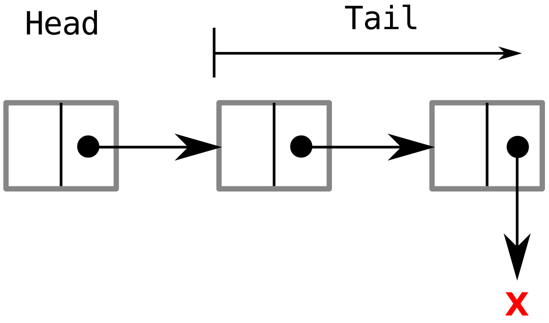
The same goes for deletion.

We will look into this sort of trade-off in detail during the module.

The linked list

More versatile, but equally simple as the array, the *linked list* has many uses and variations.

Each element in the list contains a value (the data item) and a link to the next item in the list.



The linked list

We call the first element in the list the *head*, everything else the *tail*, and the last element links to nothing.

We call the operation of sticking a new element on the front of the list *cons*.

Getting access to the head and consing are cheap operations with a fixed cost.

Unlike the array, accessing the `nth` element takes n steps.

The linked list

One of the ways of using a linked list in Java is to use the `ArrayList` class.

Or we could write our own. A class for nodes in the list:

```
private class Node {
  int data
  Node next

  public Node(int data) {
    this.data = data;
    next = null;
  }
}
```

The linked list

A class for the list itself:

```
public class LinkedList {
  Node head;

  public LinkedList(int data) {
    head = new Node(data);
  }

  public void cons (int data) {
    Node n = new Node(data);
    Node last = head;
    while (last.next != null) last = last.next;
    last.next = n;
  }

}
```

After the break

Next time we will introduce some simple mathematical notation for describing the *time* and *space* costs of an algorithm, called its **complexity**.

We will see that we can categorise all algorithms into classes which have the same complexity.

We will use our new notation to discuss the complexity of some of the operations we have been discussing so far.

Complexity

Recall the two algorithms for searching that we discussed last time – *sequential search* and *binary search*.

These algorithms perform very differently, especially for large inputs.

Complexity

In order to understand the algorithms (and thus the programs) we create, we need to understand two things:

- how much *time* they take to run for a given input, and
- how much *memory* they will consume whilst they're running.

Complexity

We’re not much interested in the *actual* time an algorithm will take because this will vary with the hardware used.

So, we measure the number of *steps* the algorithm will take for a given size of input, and how this increases with the size of the input.

We call the measure of the steps taken relative to the size of the input the **time complexity** (or just complexity) of the algorithm.

The measure of the memory consumed is called the **space complexity**.

Mathematical background

Usually, the **time** complexity is the most important measure and when we refer to the complexity of an algorithm without specifying which type, it’s the time complexity we mean.

There are a few simple mathematical ideas we need in order to describe complexity.

Floor and Ceiling

If n is a number then we say the *floor* of n, written $\lfloor n \rfloor$, is the largest integer that is less than or equal to n.

Similarly, we say that the *ceiling* of n, written $\lceil n \rceil$ is the smallest integer that is greater than or equal to n.

Floor and ceiling

For positive numbers, this is just rounding up and down.

So, $\lfloor 2.5 \rfloor$ is 2 and $\lceil 2.5 \rceil$ is 3.

We use this most often when talking about the complexity of an algorithm that depends on dividing the input in some way.

Floor and ceiling

So, if we need to compare the elements of a list of length n *pairwise* (compare elements 1 and 2, then elements 3 and 4, etc.), the number of steps required is $\lfloor n/2 \rfloor$.

If n=10 then we need $\lfloor 10/2 \rfloor = 5$ steps.

If n=11 then we need $\lfloor 11/2 \rfloor$ steps, which is also equal to 5.

Factorial

The *factorial* of a number, n, written `n!`, is the product of the numbers between 1 and n.

So, 4! = 1 x 2 x 3 x 4 = 24

You can see that factorials will get very big very quickly.

Logarithms

Logarithms play a very important role in the analysis of complexity.

You can think of them as the *dual* of raising a number to some power.

The *logarithm*, base y, of a number x is the power of y that will produce x.

Or,

$$\log_y x = z \Leftrightarrow y^z = x.$$

Logarithms

So, $\log_{10} 45$ is (roughly) 1.6532 because $10^{1.6532}$ is (roughly) 45.

The base of a logarithm can be any number, but we will normally use 2 or 10.

We use `log` as a shorthand for $\log_{10}$ and `lg` as a shorthand for $\log_2$.

Logarithms

Logarithms are strictly increasing functions, so if x>y then $\log x > \log y$. Other useful things to know:

- $\log_b 1 = 0$ (because $b^0 = 1$).
- $\log_b b = 1$ (because $b^1 = b$).
- $\log_b(x \times y) = \log_b x + \log_b y$.
- $\log_b x^y = y \times \log_b x$.
- $\log_a x = (log_b x)/(\log_b a)$.

We can use these identities to simplify equations, change the base of logs, etc.

We will also use simple ideas from probability and summations like $\sum_{n=1}^{10} n^2$.

Calculating complexity

Say we have an algorithm, `A`, that takes a list of numbers, `l`, of length n. `A` works in two stages:

- Do something once for element of l (e.g. double the number), then
- compare every element of l to every other element in the list.

We can see that the first stage will take n steps and the second n^2 steps.

Calculating complexity

We can describe the complexity of `A` with a function, `f`:

$$f(n) = kn + jn^2,$$

where `k` is the constant cost of doubling a number and `j` is the constant cost of comparing two numbers.

Calculating complexity

Disregarding the constants for a moment, when n=5, `f(n)` works out as `5+25` steps.

Here, n and n^2 are ”fairly similar” values.

When `n=100`, `A` takes 100+10,000 steps.

Now, n is starting to become much less significant than n^2.

Calculating complexity

As n increases further still, we can effectively forget about that part of `A` that takes n steps as the part that takes n^2 will *dominate*.

So, we say that the complexity of `A` is determined by the *largest* term, n^2, and we forget about the smaller terms.

Calculating complexity

A similar reasoning applies to the constants `k` and `j`.

In the n^2 stage of `A`, numbers are compared to each other, an operation which has a fixed cost, `j`.

So the complexity is really determined by jn^2 but since `j` never varies, we ignore it for the sake of clarity.

For any other algorithm with the same largest term, n^2, we say it has the same *order of complexity* as `A`, even though the details (e.g. constants and smaller terms) may differ widely.

Calculating complexity

A second algorithm, `B`, might have a more expensive operation performed n^2 times, governed by a different constant `j'`, and other smaller terms:

$$g(n) = k'(\frac{3n}{2}) + j'n^2.$$

Nevertheless, we say that `A` and `B` have the same order.

This might seem a very approximate measure, and it is, but it tells us what will happen as the size of the input to `A` and `B` grows.

Calculating complexity

As well as working out the order of algorithms, we can categorise them as follows:

- Those that grow *at least as fast* as some function `f`. This category is called *Big-Omega*, written $\Omega(f)$.
- Those that grow *at most as fast* as some function `f`. This is the most useful category, called *Big-O*, and written $O(f)$.
- Those that grow *at the same rate* as some function `f`. This category is called *Big-Theta*, written $\Theta(f)$.

We are not usually very interested in $\Omega(f)$. $\Theta(f)$ is sometimes of interest, but most of the time we are concerned with $O(f)$.

Calculating complexity

As an easy example, consider the Binary Search algorithm from the previous lecture.

At each step in the algorithm, the size of the problem is halved, until we are done.

Let n be the length of the input to the Binary Search algorithm. Furthermore, let $n = 2^k - 1$ for some `k`.

After the first pass of the loop, there are $2^{k-1} - 1$ elements in the first half of the list, 1 in the middle, and $2^{k-1} - 1$ in the second half.

After each pass of the loop, the power of 2 decreases by 1.

Calculating complexity

In the worst case, we continue until n=1, which is also when `k` is 1, since $2^1 - 1 = 1$.

This means there are at most `k` passes when $n = 2^k - 1$.

Solving this equation gives us

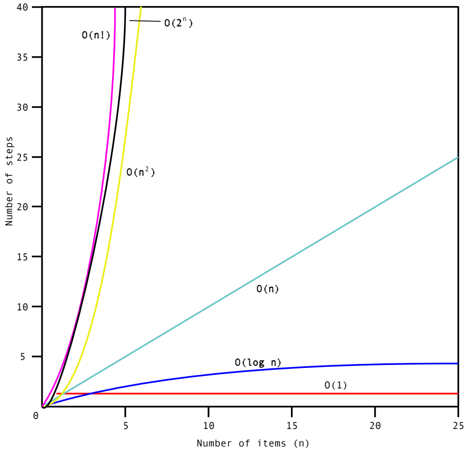
$$k = \lg(n + 1).$$

So Binary Search has a worst case complexity of $O(\lg n + 1)$, or just $O(\lg n)$.

A logarithm of one base can be converted to another by multiplying by a constant factor, so we just say $O(\log n)$.

Growth rates of algorithms

The complexity of most algorithms we come across are governed by some commonly occurring functions (this graph is an approximation):



Growth rates: Constant time. O(1)

Constant time means that *no matter how large the input is, the time taken doesn't change*.

Some examples of O(1) operations:

- Determining if a number is even or odd.
- Accessing an element in an array.
- Using a constant-size lookup table or hash table.

Growth rates: Logarithmic time, O(log n)

An algorithm which *cuts the problem in half each time* is logarithmic.

(Other patterns of computation end up being logarithmic, but this is a simple rule of thumb for spotting one.)

An O(log n) operation will take longer as n increases, but once n gets fairly large the number of steps required will increase quite slowly.

Growth rates: Linear time, O(n)

Linear time means that *for every element, a constant number of operations is carried out*, such as comparing each element to a known value.

The larger the input, the longer a O(n) operation takes.

Every time you double n, the operation will take twice as long.

An example of a linear time operation is finding an item in an unsorted list using sequential search.

Growth rates: Loglinear time, O(n log n)

O(n log n) means that /structure{an O(log n) operation is carried out for each item in the input}.

Several of the most efficient sort algorithms are in this order.

Examples of loglinear operations are quicksort (in the average and best case), heapsort and merge sort.

Growth rates: Quadratic time: $O(n^2)$

Quadratic time often means that for every element, you do something with every other element, such as comparing them.

The time taken for a quadratic operation increases drastically with the input size.

Examples of $O(n^2)$ operations are quicksort (in the worst case) and bubble sort.

Growth rates: Exponential time: $O(2^n)$

Exponential time means that the number of steps required will rise by a power of two with each additional element in the input data set.

This is a figure that gets very big very fast!

Exponential operations are normally impractical for any reasonably large n, although of course many problems may require an exponential algorithm.

An example of an $O(2^n)$ operation is the famous *travelling salesman* problem with a solution that uses *dynamic programming*.

We will study this problem in later weeks.

Growth rates: Factorial time: $O(n!)$

A factorial time solution involves doing something for all possible permutations of the n elements.

An operation with this complexity is impractical for all but small values of n.

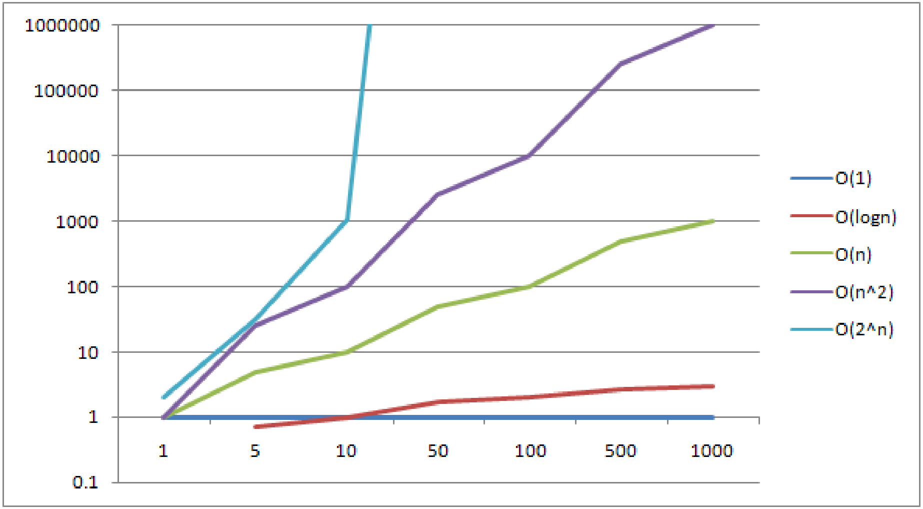
An example of an $O(n!)$ operation is the travelling salesman problem using brute force, where every combination of paths will be examined.

Growth rates of algorithms}

Another way of visualising the growth rates of the frequently encountered orders:

O(f(n))	n=10	n=100
O(1)	1	1
O(log n)	3	7
O(n)	10	100
O(n log n)	30	700
$O(n^2)$	100	1000
$O(2^n)$	1024	2^{100}
O(n!)	3,628,800	$100!$

Growth rates of algorithms



Next week's lecture

Next week: Simple sorting methods (*bubble sort*, *selection sort* and *insertion sort*), their implementation and performance.