COMP2054-ADE
Algorithms Data Structures & Efficiency

ADE Lec01
Analysis of Algorithms

Lecturer: Andrew Parkes

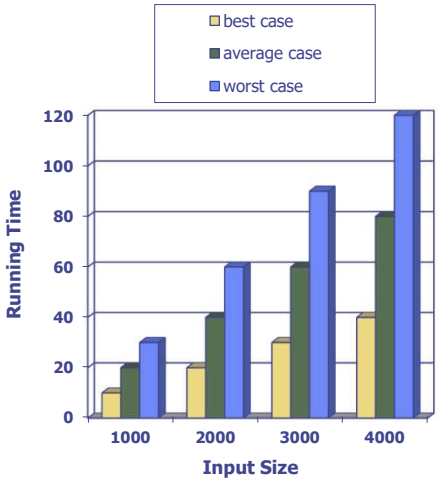
Email: andrew.parkes@nottingham.ac.uk

http://www.cs.nott.ac.uk/~pszajp/

Hello and welcome to the start of the contents of the ADE – Algorithms Data structures and Efficiency portion of the ACE module.
These slides give part a on the lecture 1 topic of "Analysis of Algorithms."

Running Time: "finite" but how big?

- Consider "batch algorithms": transform input data into output data – as opposed to "interactive"
- The running time of an algorithm typically grows with the input size.
- Even at given size the runtime is usually not fixed.
  - So have "best", "average" and "worst" cases.
  - A typical example →
- We (usually) focus on the worst case running time at given size
  - Useful, and easier to analyse
  - Average case time is often difficult to determine.



TAG: graph shows running time plotted against the "Input size". The run times increase with the size.
At each input size there are 3 different running times. "Best case" is always smaller than "average case" which is always smaller than "worst case".

In this module we focus on running times, though similar methods will also apply to analysing the space used by algorithms.
So consider a standard batch algorithm, meaning that we give it a set of data, press "go" and then we wait until it finishes and time how long that takes.
We expect larger problems take longer.

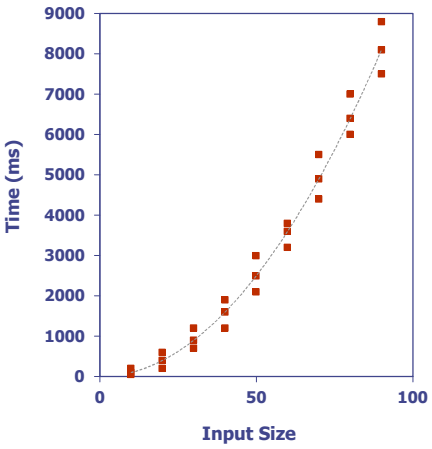
But even if we fix on a given input size, then usually the actual runtime can vary and depend on details of the input.
We'll see this later in algorithms for sorting, that the runtime might well depend on whether the input is already "nearly sorted".
So from doing such runs we get best and worst times and also the average.

A typical example is given.
We will usually focus on the worst case, as it is very useful and also because analysing the average case is usually much more difficult.

Experimental Studies

General Pattern:

- Write a program implementing the algorithm
- Run the program with inputs of "varying size and composition"
- Use a system method to get an (in)accurate measure of the actual running time
- Plot the results
  - Example is shown →
- Interpret & analyse. E.g. is it
  - A "power law", n^k, for some k
  - An "exponential", b^n, for some b



Doing an experimental study we hence do the obvious system.

Implement the algorithm – trying to do so in a reasonably effective fashion.

Then just run it with a variety of input sizes and different kinds of input data.

Then collect the runtimes and plot the answers.
Suppose we get a graph as shown, then we have to decide what we are going to say about it.

TAG: Graph shows a plot of Time in "ms" for milliseconds on y axis, against input size on the y axis. There is a variety of data points, with multiple data points for each input size, and showing an upward trend that curves upwards. It looks roughly like a parabola.

How would you communicate to someone else what the results say about how the runtime changes with the input size.
Well you could just send them the table of data; but that would not be very informative.

A reasonable question might be how you expect the runtime to grow with the input size.
Maybe it is a power law, n^k, maybe it is an exponential, b^n, or maybe something else.

Note that a common mistake is to refer to n to the k as being exponential, presumably on the grounds that “there is an exponent”,

but this is incorrect usage and it is a power law. For an exponential the size n needs to be in the exponent.

If we look at the data then at size 40 the worst case is about 2000, and doubling the size to 80 makes the worst case grow to about 7000.
If it were n squared, then doubling the size would make it 4 times slower and so be about 8000, if it were a linear then it would be about
4000. So maybe it is “something in between linear and square.”
But “something in between” is horribly vague and unscientific.
So the aim of the first set of lectures is to motivate and give the standard way of describing such things in a well defined and meaningful fashion.
Which is specifically the so-called “Big-Oh” notation, and also an associated “Big Oh family”.

You have probably seen big-Oh before, but the aim is to do it in more detail, and go through associated proofs.

Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult or time-consuming
- **Results may not be indicative of the running time on other inputs not included in the experiment.**
  - **Maybe we miss the "real worst case"**
- In order to compare two algorithms directly, the same hardware and software environments must be used

Experiments of course have their limitations.
We have to implement the algorithm and this might be time consuming.
More importantly, we have to select inputs and then this might bias results – maybe the worst case only happens on relatively few inputs and we do not happen to pick them,
Also, if we are using raw runtimes then to compare algorithms it might be best to use the same hardware and software,

Limitations of Theory

- It is necessary to implement the theory, which may be difficult or time-consuming
- Results may not be indicative of the typical running time on inputs encountered in real world.

So can be useful to be able to use both experiment and theory.

Conversely if we use just theory then it can also be difficult.

In this case it can be particularly difficult to account for the inputs seen in the real-world.

There might be patterns in the data that lead to the runtimes being quite different to the worst case.

I work in optimisation – genetic algorithms, etc. – and it is quite common that the typical behaviour is a lot better than simple theory would suggest.

Hence, it is useful to have skills in both.

Aside: Theory vs. Experiment

Standard science:
"Never believe a theory until it has been `confirmed' by an experiment"

Partially joking:

"Never believe an experiment until it has been confirmed by a theory"

- **Attributed to Sir Arthur Eddington**
  https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3597502/

Just for a quick side comment.

Part of the science method is that we need experiments to support any theory.

There are lots of nice theories that just do not seem to match reality.

A counter example to this is that experiments can also need confirmation. Usually experiments are a lot more messy and difficult than realised – they are not like text books – and some assumptions and approximations can creep in.

Hence, they also can need some confirmation.

Basically, for complex systems a mix of theory and experiment can be useful.

Of course, for simple cases, such as most of the cases in this module the theory will be enough.

Theoretical Analysis

- **AIM: Characterise running time as a function of the input size, *n*.**
- Uses a "high-level" description of the algorithm instead of an implementation
  - Takes into account all possible inputs
  - Allows us to evaluate the speed of an algorithm independently of the hardware/software/language environment

So let's move to the theory.
The aim is just to characterise the running times as a function of the input size, which we usually call n.

We tend to use a high-level description of algorithms instead of a raw implementation.

We need to be able take account of all possible inputs, and also to do so in a way that is not tied to a particular hardware or software or programming language.

Pseudocode (recap)

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

```
Algorithm arrayMax(A, n)
  Input array A of n integers
  Output maximum element of A

  currentMax ← A[0]
  for i ← 1 to n – 1 do
    if A[i] > currentMax then
      currentMax ← A[i]
  return currentMax
```


Hence, rather naturally we use some version of pseudocode.

The exact syntax does not matter, but is usually something that is procedural, and usually with similar semantics to C, even if the syntax can be quite different.

It is important to note that is procedural, for example, in that access to array is possible.
If doing in a functional language it would be using “impure” methods.

Pseudocode Details (recap)

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm ***method*** (***arg*** [, ***arg***…])

  Input …

  Output …
- Method call

  var.method (***arg*** [, ***arg***…])
- Return value

  return ***expression***
- Expressions

  ← Assignment (like = in Java)

  = Equality testing (like == in Java)

  n^2 Superscripts and other mathematical formatting allowed

This slide just gives an indication of the syntax we tend to use for pseudocode. It is designed to make the code readable.

E.g. not forcing unneeded braces,

And avoiding confusions between assignment and equality checking.

Also allowing to use mathematical format to help the easy reading.

Primitive Operations

- Basic computations performed by an algorithm
  - Identifiable in pseudocode
  - Largely independent from the programming language
  - Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the "RAM model" (next slide)
  - Tend to be close to "Assembly language"
  - No "hidden expenses"

- Examples:
  - Assigning a value to a variable
  - Indexing into an array
  - Comparing two numbers
  - Adding/subtracting/ multiplying/dividing two numbers
  - Calling a method
  - Returning from a method

In order to do the analysis we will break down algorithm into sequences of primitive operations and then count them.
What is a primitive operation?
These are just basic computations performed by an algorithm.
On a real computer the actual operations will be operations with logic gates, but these are too fine-grained and too far from the algorithm.

Hence the point of picking primitive operations is that counting them will be useful, but they are close enough to the algorithms to be identifiable.

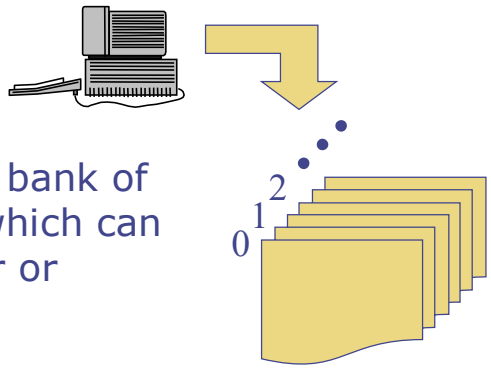
This gives something at roughly the level of assembly code.
So typically we might count operations on the right as being primitive.
Note these are just examples, and there is no unique definition.

But typically we would count an assignment as a primitive operation.

Indexing into an array might also be considered primitive.

Similarly, arithmetic operations might be considered as primitive.
On a CPU a floating point division is actually a very complex algorithm in itself, but in the assembly language it is a single instruction and so we will count it as primitive.

The Random Access Machine (RAM) Model



- A **CPU**
- A potentially-unbounded bank of **memory** cells, each of which can hold an arbitrary number or character

- Memory cells are numbered and accessing any cell in memory takes unit time (some fixed time).
  - (Note that RAM can stand for both "Random Access Machine" and "Random Access Memory," Which is an unfortunate, but standard, over-loading of terminology.)

TAG: Figure shows icon of a computer linked to bank of memory cells – arranged like multiple sheets of paper – and labelled 0, 1, 2, dot dot dot

One way to think of a computer, and hide many details, is also to regard it as a "Random Access Machine"

Which is just a processing unit, a CPU, acting on a large array.

See http://en.wikipedia.org/wiki/Random-access_machine vs http://en.wikipedia.org/wiki/Random-access_memory

The key assumption here is that accessing any element of the memory takes a given fixed amount of time.
This is an abstraction as it ignores the true costs of accessing memory – I expect that you will learn about these in the Operating Systems and Concurrency module.

But for many purposes it is sufficient to just use this simplification, with one caveat on the next slide:

Limitations of RAM model

- "… can hold an arbitrary number …" ?
  - Can we really expect to store "938566359286151800351666177735777777177177374717717471571777776136566161816161616" in one cell on a real computer?
- Here, we ignore such "bignum" issues. Instead:
- "all numbers are of equal size, as they all fit in a single register of the CPU"
- 64bits (signed int) allows up to 9,223,372,036,854,775,807
  - Exercise (offline): compare to: nanoseconds since big bang; national debt.
- Note: on real machines (usually) computing
  1 + 1 takes as long as 381513 + 243542
  Hence, we typically ignore the sizes of numbers in the arithmetic operations.

Can we really expect cells in the memory to hold an arbitrary number?
If the number is very big, then it will not fit in a single cell.
This is a "big numbers" issue.

We assume that numbers are small enough to fit in one cell.

Also, the physical hardware is such that adding two small numbers takes as long as adding two large numbers.
So when counting primitive operations we do not need to account for the sizes, or number of digits in a number.

This is quite different than if we were doing arithmetic ourselves by hand.

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm *arrayMax*(*A*, *n*)	# operations
currentMax ← *A*[0]	?
for *i* ← 1 **to** *n* − 1 **do**	?
if *A*[*i*] > *currentMax* **then**	?
currentMax ← *A*[*i*]	?
return *currentMax*	?
Total	EXERCISE "Try it!"

So let's finally move to an actual simple algorithm and start to do some counting. The algorithm we look at here is just a simple one that takes a non-empty array and returns the maximum element in the array.

The exercise I want you to try it to consider the algorithm and then fill in some counting of the numbers of primitive operations that will be taken, and starting by at least doing it for each statement or line of the code.

In case it is not clear, the intention here is to fill is some "count" that covers all the times that a line of code might be processed.

If reading or watching the video, then stop or pause now, and try it yourself !!

Counting Primitive Operations (partial) – "Offline -> Pause"

- **Worst case** number of primitive operations executed as a function of the input size, n

Algorithm *arrayMax*(*A*, *n*)	# operations
currentMax ← *A*[0]	?
for *i* ← 1 **to** *n* − 1 **do**	?
if *A*[*i*] > *currentMax* **then**	?
currentMax ← *A*[*i*]	2 (*n* − 1)
	?
return *currentMax*	?
Total	??

Here is a partial start.

We are going to do the worst case, that is we look for the largest possible number that can occur

The only line that might or might not be run is the assignment to the current Max.

So in the counting we will suppose that it might occur for every element of the array.

This happens in the array happens to be sorted in ascending order, then every entry it larger than the previous entry and so will be invoked.

For an array of size n, then remember the first element was already used, so the line is run n-1 times in the worst case.
For the counting operations of each time it is run, then we might say

Count one for the lookup of A[i]
Count one for the assignment.

So at this point I might pause the recording, please try to fill in the other entries and so arrive at some function of n for the total number of primitive operations.

Counting Primitive Operations (partial)

- Worst case number of primitive operations executed as a function of the input size, n

Algorithm *arrayMax*(*A*, *n*)	# operations
currentMax ← *A*[0]	?
for *i* ← 1 **to** *n* − 1 **do**	1 // for *i* ← 1
if *A*[*i*] > *currentMax* **then**	2 (*n* − 1)
currentMax ← *A*[*i*]	??
return *currentMax*	*?*
Total	??

The "(n-1)" arises from the number of times that the body of the for loop is executed.

What about the first line.
Maybe we can count it as 2 by just saying
1 primitive op for the "get A[0]"
1 primitive op for the "assign to currentMax"

But what else is going on in the loop?

There is the loop counter i. Operations on it are not free.
We have to increment it – there is a "hidden i++"

We have to continually test it as to whether it has become too large, and then we need to exit the loop.

We'll have to do these for each of the entries – which will a factor of n-1 (or so)

And then decide how many ops to assign this.

We might then reasonably come up with an answer:

Counting Primitive Operations (all)

- Worst case number of primitive operations executed as a function of the input size, n

Algorithm *arrayMax*(*A*, *n*)	# operations
currentMax ← *A*[0]	2
for *i* ← 1 **to** *n* – 1 **do**	1
if *A*[*i*] > *currentMax* **then**	2 (*n* – 1)
currentMax ← *A*[*i*]	2 (*n* – 1) (worst case)
{ increment counter: *i*++ }	2 (*n* – 1) ("hidden")
{ test counter: *i* ≤ (n-1) }	2 (*n* – 1) ("hidden")
return *currentMax*	1
Total	8*n* – 4

Notice the "hidden" increment and test of counter !!
Adding up all the contributions then we get 8n-4,

But at this point usually people are saying that they counted differently.

This is often fine.

The system for such counting is not going to possible to specify exactly.

Counting is "underspecified"

- Consider "$c \leftarrow A[i]$" then 'full' process can be
  - get A = pointer to start of array A, and store into a register
  - get i, and store into a register
  - compute A+i = pointer to location of A[i], and store back into a register
  - get value of "*(A+i)" (from RAM) and store value it into a register
  - copy the value into the location of c in the RAM
- might not want to count all this, e.g. just count
  - 'plus' of "A+i"
  - the assignment

How we count is somewhat under-specified, or "vague".

Many reasonable ways of counting exist and can give different answers.

Consider just the case of doing a assignment to a variable from an element of array.

Think back to your first year about what is actually going on.

So in this example from assigning a local variable a value from an array, there is quite a long process that goes on.
For full gory detail think of how it might be done in assembly code.

Firstly, "A" is a pointer, it's value is stored somewhere and so we may need to get it from RAM and copy it to a register.
Also, have to do the same for i, though i might or might not be stored in a register already.
Have to then add increase A by i – which is a call to some arithmetic operation.
The resulting pointer will be stored back somewhere, then we have to dereference it, and so store it somewhere

Finally, we need to store that value back to the location where c is stored.

It would (almost always) be excessive to count all these details.
Hence, we might decide to just count the plus operation and the assignment.

Maybe doing all the full counting would suggest 5 operations, but the simpler would be 2.

But they are both small constants.

If you are trying to predict the actual runtime then maybe the most expensive operation would be the ‘plus’

But even this would be challenging, as the CPU will do doing operations in parallel, or speculatively when possible (see “branch prediction from year 1)

Counting is "underspecified"

- There can be multiple right answers – if you get '2' and I count '4' then it does not mean you are wrong!
- Note: If I think an answer is '4' then '2' is probably also acceptable – but "2 n" probably will not be.
- It is most important to be able to
  - know what is happening in the underlying process
    - be able to link to C and assembly level notions
  - be able to use this to give a reasonably consistent justification of your answers

Hence, we cannot expect to arrive at a notion of "the one right answer".

However, in the case above then doing something where a small constant like 4 or 2 changed to 2n will probably be wrong.

The point here is to be able to take some code and be able to envisage what is happening in the underlying process.

Note that this is something that is relevant to the efficiency and runtime of an algorithm,

In answering questions about this kind of topic want to be able to see that nothing important about the underlying process is being missed.

And that you can give a reasonably consistent scheme.

Will not be providing one scheme that is "apply without thinking" as the point is that you should be able to do so yourself.

Note: Correctness vs. Efficiency

- Primitive operation counting is relevant to Efficiency – but not (directly) for Correctness
- For correctness, do not about runtime of the alg.
  - Do care about the time to find a proof if doing an automated search for proofs.
  - The verification seems to be quick in lean: e.g.

    #eval 2035713999 + 350135299
  - Very quick (<1sec) – not using succ internally (?)
- Note that we did not prove the algorithm correct
  - Would need to do arrays/lists in Lean first
    - Just for thought: And then what?
  - Can do efficiency of incorrect algorithms ☺

Note that this is something that is relevant to the efficiency and runtime of an algorithm, and not really something that appears in the question of whether or not the algorithm is itself correct.

We claimed the algorithm gave the maximum of an array.

If can express arrays, or lists, in Lean then there is a possibility to prove that it is correct.
For thought: If you had arrays or lists in Lean, then what might this look like?
But that is Thorsten's side.

In terms of efficiency, one can happily evaluate efficiency or run times of even incorrect algorithms ☺.

Estimating Running Time

- Algorithm ***arrayMax*** executes $8n - 4$ primitive operations in the worst case. Define:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of ***arrayMax***. Then
$$a(8n - 4) \leq T(n) \leq b(8n - 4)$$
- Hence, $T(n)$ is bounded 'above and below' by two linear functions
- Usually said as "*arrayMax* runs in linear time"

But let's get back to runtimes.
We did some counting and came up with 8n-4 primitive operations.
Not all primitive operations take the same actual runtime.
Hence we cannot directly get the runtime.
But there will be some slower, and some faster operations.
Suppose that the fastest ones take a, and the slowest b.

We end up with the actual worst case runtime being sandwiched between two linear functions.

Of course we just say usually say that the algorithm runs in linear time, but am trying to encourage to look in more detail, of what a statement means.

Remarks

- Do not get too obsessed with the fine details of counting of primitive operations
- The details of the counting and timing would probably depend
  - the compiler, and require inspection of the assembly code
  - the CPU architecture, pipelining, cache misses, etc, etc

A note is to seek a general understanding, and not to expect a definitive answer for the answer.

If marking answers it is not “I’m thinking of a number” and any other answers are incorrect !!
Reasonable answers would be accepted, and also, as common in real-life, defining reasonable is not possible to do exactly.

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- **The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm *arrayMax***

Instead the details will affect the coefficients and details, but changing the machine will not change the linear nature.

Of course, this is "obvious" in this case, but writing efficient implementations can benefit from knowing what is going on "under the hood" - idiom for "look at the car engine" – know what is going on as opposed to just using the engine (CPU) without understanding some basics of what it does.

So let's do a slightly more complex case:

Exercise: (exam-style question)

Given the following code fragment:

```
m ← 0
while (n ≥ 2)
    n ← n/2
    m++
return m
```

Give an analysis of its runtime

What might an exam style question look like?
Here is some code.

For the moment let's not worry about what it computes, but just focus on doing an analysis of the run-time.

If fact, today we just do a slightly simpler case for specified inputs that make the analysis easier, and capture the essence.

Exercise: what is T(n) of alg-lec1?

Algorithm: *alg-lec1*
Input: positive integer n, which is a power of 2.
I.e. there exists k such that $2^k = n$

```
m ← 0
while (n ≥ 2)
    n ← n/2
    m++
return m
```


We will just consider the input being a power of two.

Then now just do the same counting of primitive operations.
So let's follow the same procedure and annotate with counts

Exercise: what is T(n) of alg-lec1? (cont)

Algorithm: *alg-lec1*
Input: positive integer n, which is a power of 2
Output: integer m such that $2^m = n$

m ← 0	1
while (n ≥ 2)	?
n ← n/2	?
m++	?
return m	1

Here is just a start.

I put 1 for the first line.

People might complain that in previous computations, I counted two operations, but this time the 'zero' is likely to be provided by the compiler and does not need to be looked up, and so might be less work.
It might be done automatically or implicitly and might well not take any runtime.
But as before we do not worry too much.

What about the loops?
As before we can split into cost per time going around, and the number of times we go around.
So first we can do the "per pass"

Exercise

Algorithm: *alg-lec1*
Input: positive integer n, which is a power of 2
Output: integer m such that $2^m = n$

m ← 0	1
while (n ≥ 2)	? per pass
n ← n/2	? per pass
m++	? per pass
return m	1

(Pause and try/think)

The main line that controls the flow in the loop is the third line that divides n by 2 on each pass.

What might we count for that?

Internal Steps:

n ← ***n/2*** ?

1. read ***n*** from memory (RAM) and store in a register r1 (very fast piece of memory on the CPU)
2. read ***2*** from memory and store in a register r2
3. send registers r1 r2 through arithmetic division and store result in a register r3
4. write r3 back to ***n***

CPU steps needed is 4, does not depend on ***n***

As before let's try to break it down into the steps that will mean something "on the silicon".
We need to get n, and it may just be stored in the RAM. As in assembly we will need to load it into a register.
2 might just also be stored in the program executable, and so also needs to be copied to a register.
We then send the contents of the two registers through integer division, and the answer is stored back to a register.

Which then store back to the n – overwriting the old value.
In this way we get 4 steps.

Note that 4 is just a small constant and does not depend on n.
This is an effect of the way that CPUs work, and the assumption that everything works with small numbers.

If the 'n' was a big integer using some package then the answer could be different.

Might we have different answers?

Internal Steps: different compiler

n ← ***n/2*** ?

1. read ***n*** from memory (RAM) and store in a register r1 (very fast piece of memory on the CPU)
2. send registers r1 through a right shift of the bits and store result in a register r3
   e.g. compute 13/2=6 by 1101 → 110
3. write r3 back to ***n***

CPU steps needed is 3, different but still does not depend on ***n***

Suppose we have a different compiler, or we turn on the optimisation.
Then what can happen is that the compiler switches the "integer divide" to instead be a right shift.
This can be built into the silicon, and so would be one primitive operation.

Now the count would change.

Exercise (cont)

Algorithm: *alg-lec1*
Input: positive integer n, which is a power of 2
Output: integer m such that $2^m = n$

m ← 0	1
while (n ≥ 2)	3 per pass
n ← n/2	3 per pass
m++	3 per pass
return m	1

So lets continue and fill in some reasonable numbers for the counts per pass.
As before we do not worry too much about the exact values of trying to fill in.
The big question is how we determine the number of passes.

Thought Exercise (offline)

- Based on your knowledge of assembly, and machine architectures try to estimate the number of CPU cycles that might actually be used.
- "Divide" or "shift"? Which is faster?
- **Point: try to eventually build a mental model that is an "internal interpreter" so as to know how a program will run**
- **Such "internal interpreters" are vital for understanding programming (IMHO)**

The reason for going into such detail is to encourage you to have an "internal interpreter" that can see underneath the code and have a picture of what is needed.

For this I am particularly motivated, but an incident many years ago, in which a new student was trying to reproduce in Java some code that had been written in a more procedural language.

The first attempt was about 50 thousand times slower than the original.

It turned out that this happened because of not understanding what Java actually did internally, and so did something in an inner loop that added a lot of extra unneeded overhead.

Besides the relevance to the analysis in this module, it is a skill that might help you in the future in making programs that are fast.

Of course, to be used when appropriate – for rarely-used programs that run just a millisecond the efficiency might not matter,

For complex simulations (such as aerodynamics) that run on large computers for days, then the efficiency matters a lot.

How many passes through the loop of alg-lec1?

Hint: If ever stuck:

- Try simple concrete examples
- Start from "ridiculously simple" and work up to harder examples

(Real mathematicians often work this way; but then hide it when writing up ☹)

- Do a "trace of the program" by hand:

So now we have to decide how many times through the loop.

Many will already know the answer, but I want to do the details, and give an approach that should help with harder cases.

We will do this by just working up from the simplest possible examples.

How many passes through loop?

- Focus on the relevant portions:

  while (n ≥ 2)

  n ← n/2

- Simplest example?
- Exercise: What is smallest positive integer that is a power of two?
- Answer: 1 as $2^0 = 1$
- (If confused, or if you answered "2", then consider revising your maths about exponents and logarithms)

So lets just focus on the parts that affect the number of times running through the loops.

What is the smallest integer that is a power of two.

If you would have answered 2, then you might need to revise exponents ☺.

How many passes through loop? (cont)

while (n ≥ 2) { n ← n/2 ; }

- Case: n = 1 = 2^0 passes = 0
- Case: n = 2 = 2^1
  - n=2, then n=1 ; passes=1
- Case: n = 4 = 2^2
  - n=4, then n=2, then n=1 ; passes = 2
- Case: n = 8 = 2^3
  - n=8, 4 , 2, then n=1 ; passes = 3

Hence just working upwards.
We can do n = 1 and have no passes.

Working upwards we can find the numbers explicitly.

Then is a matter of spotting the pattern.

In this case spotting the pattern is really easy.

How many passes through loop? (cont)

- Case: $n = 2^m$
  - $n=2^m, 2^{m-1}, 2^{m-2}, \ldots, 2$
  - m passes through loop
- but note, n is the input not m, so want to write answer in terms of n. Use
  - $m = \log_2(n)$
- Result: passes through loop = $\log_2(n)$

If we have that n = 2^m, then we have m passes through the loop.

At this point it is tempting to use m and express the counts in terms of m.
However, m is not the input.
We need the answer in terms of n, not m.
Hence we just invert the relation.
Revise your logs if needed!!

Exercise (cont)

Algorithm: *alg-lec1*
Input: positive integer n, which is a power of 2
Output: integer m such that $2^m = n$

m ← 0	1
while (n ≥ 2)	3 ($\log_2(n)+1$)
n ← n/2	3 $\log_2(n)$
m++	3 $\log_2(n)$
return m	1
all together:	9 $\log_2(n)$ + 5

(the "+1" on line 2, is because the test is done even if it fails)

Putting all this together, we can get something like this.
There can be extra parts of "+1" because of tests being done when the loop fails.
As before the details of the small numbers are not terribly important.

We get a linear sum of a log and a constant, and with small constants.

Remarks

- Each pass through the loop the size of n is halved
  - the "$\log_2(n)$" is typical of such "halving on each iteration"
- **This concept also appears in sorting and searching; hence you MUST make sure you fully understand this example**
  - **The ADE half of the module will probably be incomprehensible otherwise**

Looking at this case more abstractly, we have a loop that "halves the size of n" on each pass.

The result is then a complexity that involves the log of n.

This is actually the core idea in many efficient algorithms.
We will see many algorithms with a log, and this idea is the underlying reason for the logs.

Summary

Goal:

- Build foundations for time-analysis of programs

Skills needed:

- Count primitive operations
- Counting of operations with
  - Loops
  - (Recursion)

This just repeats a summary of what we have done – just in case you are losing track.
Did not actually do recursion here – but will see it later in the ADE half.

Removing details

- According to precisely how we count steps we might get many different answers, e.g. something like
  - 5 $\log_2(n)$ + 2
  - 9 $\log_2(n)$ + 5, etc
- Also this counts "steps"
  - the translation to runtime depends on the compiler, hardware, etc
- **Need a way to suppress such details**

But looking again abstractly we see that the primitive counting is too difficult to do precisely, and that there is not "One unique answer"

Do not take the "5 $\log_2(n)$ + 2" etc. literally, they are just examples of the kinds of differences that might be encountered.

Hence, any system for describing and reasoning about efficiency needs to be insensitive to such details.

That is to basically hide them and don't care.

Next Lecture

"Suppressing the details"

A motivation and introduction to big-Oh

Oh… ohh!!

Hence in next lecture we start with Big-Oh and in particular how the systems are the natural answer to how to do the suppression of details.

Exercise: (Advanced, Offline/Take-home)

Given the following code fragment:

```
m ← 0
while (n ≥ 2)
    n ← sqrt(n)
    m++
return m
```

Give an analysis of its runtime.

For the counting, assume that the square root "sqrt" is a primitive operation (though in reality it is many operations).

Finally, if you think that the previous was too easy then as a challenge try doing the same kind of analysis for the case in which the "divide by two" is replaced by taking a square root.

So do the same code and follow the same system and try to express the complexity.

This is a good case of finding out the difference between "remembering an answer" such as "it's a log" and being able to derive an answer ☺

Hint: The intention is that n is an integer, and that try to keep everything in terms of integers.
So in the same way we originally pick nice inputs "2^m", try to pick nice inputs here as well,

e.g. so that the sqrt is only ever taken of a square number.

Hint: Try some example.

Hint: Consider running the code backwards – i.e. repeatedly squaring, e.g. 2 to 4 to 16 to 256 … and try to write the pattern using formulas.