Lecture 23:
Measuring and Analysing Algorithm Complexity

Aims:

- To look at various ways of comparing algorithms;
- To look at the idea of expressing the running time of an algorithm as a function of input size;
- To look at the idea of defining the worst-, best- and average-case running times.

23.1. Choosing an Algorithm

- For every problem there is a multitude of algorithms that solve the problem. So you have a choice of algorithms to code up as programs.
- If a program is likely to be used only *once* on a *small* amount of data, then you should select the algorithm that is easiest to implement. Code it up correctly, run it and move on to something else.
- But if the program will be used many times and has a lifetime that makes maintenance likely, then other factors come into play including readability, extensibility, portability, reusability, ease of use and efficiency. It is efficiency that we be looking at in this part of the module.
- The efficiency of an algorithm depends on its use of resources, such as:
  - the time it takes the algorithm to execute;
  - the memory it uses for its variables;
  - the network traffic it generates;
  - the number of disk accesses it makes;
  - etc.

  We are going to focus pretty much exclusively on time.
- Note, however, that trade-offs are often necessary. An algorithm may save time by using more space; or it may reduce disk accesses by using more main memory. Efficiency often also conflicts with other criteria such as readability, extensibility, etc.
- It is often said that there is no need to worry about selecting an efficient algorithm or improving the efficiency of an algorithm. According to this viewpoint, an algorithm that is impractically inefficient today will become at least adequately efficient on

the faster hardware that we anticipate will be available in a few years' time. This viewpoint ignores at least two facts:

  - Increases in hardware speed are outstripped by increases in our ambitions. For example, even allowing for good advances in hardware speed, we will need supremely efficient algorithms if we are to advance the state-of-the-art in realistic gaming systems.
  - As we will see, some algorithms are so unreasonably inefficient that no reasonable increases in hardware speed will ever make the algorithms practical

  So it is worth worrying about and we'll begin to look at how to do it.

- Suppose you have two or more algorithms that are designed to solve the same problem. There are two main ways of comparing their running times:

  Benchmarking: code them up as programs, execute both programs on some typical inputs and measure their running times;

  Analysis: reason about the algorithms to develop formulae that are predictive of their running times.

- Let's briefly discuss benchmarking first. This has several disadvantages:

  - If we are choosing between two (or more) algorithms, we would prefer to *predict* their running times, compare the predictions and code up only one of the algorithms. But, benchmarking requires us to code up both algorithms as programs, before we can carry out any comparisons.
  - As we discussed in lecture 2, for most problems there are an infinite number of problem instances. Benchmarking measures performance on only a finite subset of the instances. This raises questions about the representativeness of the chosen subset. (We had similar concerns when we discussed program *testing*.)

  - Any running times that we obtain will depend on the programming language, compiler, hardware, etc.
  - Measuring execution time is fraught with difficulties. If you are running on a multi-user machine, then elapsed time is not the correct measure. You must measure CPU time. But for certain programming languages, additional housekeeping activities might run at unpredictable points and you may not wish to include these in your measurements. For example, if you use Java, the Java garbage collector will run at various points to reclaim the memory space used by unneeded objects. You may want to exclude from your measurements any time spent garbage collecting. Another problem is that the timing commands provided by many programming languages do not report times to a sufficient level of precision. For the very simplest operations, the timing commands might report a time of zero. It then becomes necessary to find the time it takes to execute 10 (or whatever) copies of the command and then divide by 10.

- We are not going to discuss benchmarking any further; we will focus on analysis. You shouldn't assume that benchmarking is useless. It is, in fact, very important in practice.

  As you will discover, algorithm *analysis* tends to present us with very rough-and-ready formulae. Benchmarking can be important for finer-grained discriminations. For example, suppose we have three algorithms, A, B and C. Analysis may predict that B has by far the highest running time, enabling us to discard further consideration of algorithm B. Analysis may also predict that A has a higher running time than C. But the predictions are rough-and-ready, so if the predictions are in some sense close, then we may choose to code up both A and C as programs and benchmark them.

23.2. Analysis of Running Times

23.2.1. Dependence on Input Size

- The running time of an algorithm will be the sum of
  - a fixed part that is independent of the parameters;
  - a variable part that will depend on the parameters.
- For example, the running time of the following algorithm...

Parameters: A positive integer, n.

Returns: The sum of the integers from 1 to n.

```
{   sum := 0;
    for i := 1 upto n
    {   sum := sum + i;
    }
    return sum;
}
```

is

- the time taken to initialise *sum* and to return it, which will be the same no matter what n is, plus
- the time we spend executing the loop body, which will depend on the value of n.

Suppose the first and last commands take 4 microseconds to execute in total, and suppose the assignment in the loop body takes 2 microseconds to execute, then we can define the running time in microseconds of this algorithm for input n, $t(n)$:

$$t(n) =_{\text{def}} 4 + 2n$$

(I've simplified! I ignored the time taken to initialise i, to test i each time round the loop and to increment i each time round the loop.) The point is the run time is expressed as a function of the input size.

- In the previous problem, there was an infinite number of instances. But each one was of a different size. In general, some of the inputs may be the same size.

  Consider this example:

Problem 23.1.
Parameters: *A finite-length list, L, of positive integers.*
Returns: *The sum of the integers in the list.*

Many of the problem instances are of the same size:

Size:	0	1	2	3	...
	[]	[1] [2] [3] ⋮	[1, 1] [1, 2] [1, 3] ⋮ [2, 1] [2, 2] ⋮	[1, 1, 1] [1, 1, 2] [1, 1, 3] ⋮ [1, 2, 1] [1, 2, 2] ⋮	

Even so, we can still express the running time as a function of the input size.

Parameters: A finite-length list, L, of positive integers.

Returns: The sum of the integers in the list.

```
{   sum := 0;
    for each x in L
    {   sum := sum + x;
    }
    return sum
}
```

Making similar assumptions (and simplifications), the running time of this algorithm, $t(n)$, might also be $4 + 2n$, where n is the length of L. But you can see more clearly now what we mean by this. We mean that for any problem instance of size n (and, as we have seen, there may be more than one instance of any given size), the time taken will be $4 + 2n$ microseconds.

23.2.2. Worst, Best and Average Cases

- But we've now got a problem. Some algorithms will have different running times even for two problem instances of the same size.
- For example, consider sorting lists of integers into ascending order. Here are some problem instances all of size 3:

Size:	3
	$[11, 13, 15]$ $[11, 15, 13]$ $[13, 11, 15]$ $[13, 15, 11]$ $[15, 11, 13]$ $[15, 13, 11]$

Although these problem instances are all of the same size, it's likely that a sorting algorithm will do less work for those instances that are already more nearly-sorted into ascending order.

But, in that case, how can we come up with a function $t(n)$ that expresses the running time in terms of *just* the input size, n?

- Here's another example that makes the same point.

Algorithm: LINEARSEARCH$(x, a, lower, upper)$

Parameters: x is an integer; $a[lower \ldots upper]$ is an array of integers; $0 < lower \leq upper$.

Returns: The position of the first occurrence of x in a if found, otherwise **fail**.

```
{   for i := lower upto upper
    {   if a[i] = x
        {   return i;
        }
    }
    return fail;
}
```

Although this has more than one parameter, in terms of problem size the only one that matters is the length of the array, as this may vary from instance to instance. The length is $upper - lower + 1$, but let's call it n.

Here are four problem instances. They are all the same size: $n = 10$. But the running time for the first will be lower than for the second, which will be lower than for the third, which will be lower than for the fourth.

x	a (length 10)
11	1 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 2 0
15	1 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 2 0
20	1 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 2 0
21	1 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 2 0

Again, these inputs are all the same size but their running times differ, so how can we come up with a function $t(n)$ that expresses the running time in terms of just the input size, n?

- When an algorithm's running time depends on the actual input and not just the input size, we have several options:

  Worst-Case: Define $t(n)$ to be the *worst-case* running time among all inputs of size n. In other words, make the least favourable assumptions.

  The nice thing about reporting a worst-case running time is that it is a guarantee. The algorithm will not take longer than the worst-case time. The algorithm could run in much less time for some inputs, but it never takes longer than the worst-case time, no matter what the input is.

  Best-Case: Define $t(n)$ to be the *best-case* running time among all inputs of size n. In other words, make the most favourable assumptions.

  There are many people who believe that this is a bogus measure; it does not offer a reliable way to compare algorithms. An algorithm that has an excellent best-case running time may be slow in general but fast on just one or a few inputs.

Average-Case: Define $t(n)$ to be the *average-case* running time over all inputs of size n. In other words, take an average over all the possibilities.

This is quite an attractive option, but it is often much harder to compute than worst- or best-case running times. At its simplest, you are required to assume that all instances of a given size are equally likely. This may not be true in practice. Alternatively, you have to define a probability distribution that describes how likely each instance is and use this to weight the average.

- We will focus on worst-case running times.

Class Exercise

- Give formulae for the
  - worst-case running time
  - best-case running time; and
  - average-case running time

  of the LinearSearch algorithm as functions of n, the length of the array. Assume that each operation takes 1 microsecond. (Again, to keep things simple, you can ignore the time it takes to initialise i, to test whether $i > upper$ each time round the loop and to increment i each time round the loop.)

  What assumption did you make when computing the average-case running time? Discuss the reasonableness of this assumption.

23.3. Time Complexity

- To develop our formulae, we have had to make assumptions about how long different operations take. E.g. in the previous exercise I told you to assume that testing $a[i]$ took 1 microsecond. This is obviously problematic. Where do the numbers come from?
- We could write a program that contains some simple commands (initialising a variable, comparing the contents of a variable with another, etc.) and some timing commands. Then we could find out how long each operation takes and use these numbers in our formulae. The weakness of this is that it is machine-dependent, programming language-dependent, compiler-dependent, etc.
- This is a point at which it is worth remembering what our goals are. We want to be able to compare algorithms. We want to do this by analysing the algorithms to obtain rough-and-ready predictions of their running times. This will tell us, for example, which one or ones are worth coding up as programs for benchmarking or, if the predicted running times are very poor it will tell us that we should instead put more effort into finding a better algorithm to begin with. The analysis should be something we can do quite quickly. If it takes too long (e.g. because it is too complicated), then we might just as well have coded up all the algorithms and benchmarked them. Furthermore, the predicted running times should be robust enough to hold true across different hardware, programming languages and compilers.
- In the light of these goals, here is what we do.
  - We *count* 'primitive' operations, instead of timing them.
  - We might have quite a high-level view of what constitutes a 'primitive' operation.

  - Consider the following, for example:

    $mid := (lo + hi) \operatorname{div} 2;$

    For the most part it is acceptable to count this as one operation, no matter how complicated the right-hand expression is! Only very rarely will you want to count it as 2 or 3 operations.
  - We may not even count *all* operations. We might just count *major* operations: those that we think are most significant for a given problem. For example, if the problem is to search for a value in a list, we might only count comparisons of the value and items in the list. Or, if we are looking at certain sorting algorithms, we might only count comparisons or swaps.
  - With these simplifications, we say we are computing the *time complexity*, rather than the *running time*. We continue to focus on worst-cases, so we say we are computing the *worst-case time complexity of an algorithm*.

- This simplification helps us obtain cheap-to-compute rough-and-ready formulae. Of course, it does mean that when two algorithms have even fairly similar time complexities, we have to be more wary of drawing firm conclusions about which is the more efficient, and we may need to using benchmarking as a follow-up more often.

- So, unless otherwise stated, let's count the following:

  Assignment: One for each assignment command, no matter how complicated the expression

  One-armed conditional: One for the test, possibly plus the cost of the branch, making the most pessimistic assumptions about whether the branch is executed

Two-armed conditional: One for the test, plus the cost of one of the branches, making the most pessimistic assumptions about which branch is executed

Unbounded iteration: One for the test plus the cost of the body, multiplied by the number of iterations, making the most pessimistic assumptions about the number of iterations

Bounded iteration: One for the test plus the cost of the body plus one for incrementing, multiplied by the number of iterations, making the most pessimistic assumptions about the number of iterations, plus one for initialisation

Other: One for return commands. And we'll just exclude procedures and anything else from our examples to keep the material a bit simpler.

Class Exercise

- Give a formula for $t(n)$, in terms of n, that defines the worst-case time complexity of the following, counting only assignments to *maxSoFar*.

Algorithm: ARRAYMAX(a)

Parameters: An array, $a[1 \ldots n]$.

Returns: The largest element in a.

```
{   maxSoFar := a[1];
    for i := 2 upto n
    {   if a[i] > maxSoFar
        {   maxSoFar := a[i];
        }
    }
    return maxSoFar;
}
```

23.4. Summary

- We've made numerous simplifications (and there are more to come). We obtain a formula in terms of the input size which predicts:

Algorithm	—	our predictions concern algorithms or programs
Worst-Case	—	where predictions would vary for inputs of the same size, we offer guarantees by being pessimistic
Time	—	our predictions concern execution time
Complexity	—	but we count the ('major') 'primitive' operations, rather than measuring time on some architecture

Acknowledgements

I have drawn ideas from [AU92], [GT02], [Har92] and [HSR96].

Clip Art (of head with bomb) licensed from the Clip Art Gallery on DiscoverySchool.com.

References

[AU92] A. V. Aho and J. D. Ullman. *Foundations of Computer Science*. W.H. Freeman, 1992.

[GT02] M. T. Goodrich and R. Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. Wiley, 2002.

[Har92] D. Harel. *Algorithmics: The Spirit of Computing*. Addison-Wesley, 2nd edition, 1992.

[HSR96] E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms/C++*. W.H. Freeman, 1996.