CITS2200 Data Structures and Algorithms

Topic 7

Performance Analysis 2: Asymptotic Analysis

- Choosing abstract performance measures
  - worst case, expected case, amortized case
- Asymptotic growth rates
  - Why use them? Comparison in the limit. "Big O"
- Analysis of recursive programs

Reading: Lambert and Osborne, Sections 4.2–4.3.

1. Educational Aims

The aims of this topic are:

1. to develop a mathematical competency in describing and understanding algorithm performance, and
2. to begin to develop an intuitive feel for these mathematical properties.

It is essential for a programmer to be able to understand the capabilities and limitations of different data structures. Asymptotic analysis provides the foundation for this understanding (even though you would not expect to do such analysis on a regular basis).

2. Worst Case, Expected Case, Amortized Case

Abstract measures of time and space will still depend on actual input data.

eg Exhaustive sequential search

```
public int eSearch(...) {
  ...
  i = 0;
  while (a[i] != goal && i < n)  i++;
  if (i == n)  return -1;                  // goal not found
  else  return i;
}
```

Abstract time

- goal is first element in array — a units
- goal is last element in array — $a + bn$ units

for some constants a and b.

Different growth rates — second measure increases with n.

What measure do we use? A number of alternatives. . .

2.1 Worst Case Analysis

Choose data which have the largest time/space requirements.

In the case of `esearch`, the worst case complexity is $a + bn$

Advantages

- relatively simple
- gives an upper bound, or *guarantee*, of behaviour — when your client runs it it might perform better, but you can be sure it won't perform any worse

Disadvantages

- worst case could be unrepresentative — might be unduly pessimistic
  - knock on effect — client processes may perform below their capabilities
  - you might not get anyone to buy it!

Since we want behaviour guarantees, we will *usually consider worst case analysis* in this unit.

(Note there is also 'best case' analysis, as used by second-hand car sales persons and stock brokers.)

2.2 Expected Case Analysis

Ask what happens in the average, or "expected" case.

For `eSearch`, $a + \frac{b}{2}n$, assuming a uniform distribution over the input.

Advantages

- more 'realistic' indicator of what will happen in any given execution
- reduces effects of spurious/non-typical/outlier examples

For example, Tony Hoare's `Quicksort` algorithm is generally the fastest sorting algorithm in practice, despite it's worst case complexity being significantly higher than other algorithms.

Disadvantages

- only possible if we know (or can accurately guess) probability distribution over examples (with respect to size)
- more difficult to calculate
- often does not provide significantly more information than worst case when we look at growth rates
- may also be misleading. . .

2.3 Amortized Case Analysis

Amortized analysis is a variety of worst case analysis, but rather than looking at the cost of doing the operation once, it examines the cost of repeating the operation in a sequence.

That is, we determine the worst case complexity $T(n)$ of performing a sequence of n operations, and report the amortized complexity as $T(n)/n$.

An alternative view is the accounting method: determine the individual cost of each operation, including both its execution time and its influence on the running time of future operations. The analogy: imagine that when you perform fast operations you deposit some "time" into a savings account that you can use when you run a slower operation.

Reading: Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, Chapter 17.

2.4 Amortized Analysis for a Multi-delete Stack

A multi-delete stack is the stack ADT with an additional operation:

1. *mPop(i)*: delete the top i elements from the stack

Assuming a linked representation, the obvious way to execute `mPop(i)` is to perform pop i times.

If each *pop* takes *b* time units, *mPop(i)* will take approximately ib time units — linear in i!

Worst case is nb time units for stack of size n.

But...

Before you can delete i elements, need to (somewhere along the way. . .) individually insert i elements, which takes i operations and hence ic time for some constant c.

Total for those $i+1$ operations is $i(c+b)$. The time for i operations is approximately linear in i. The *average* time for each operation

$$\frac{i}{i+1}(c+b)$$

is approximately constant — independent of i.

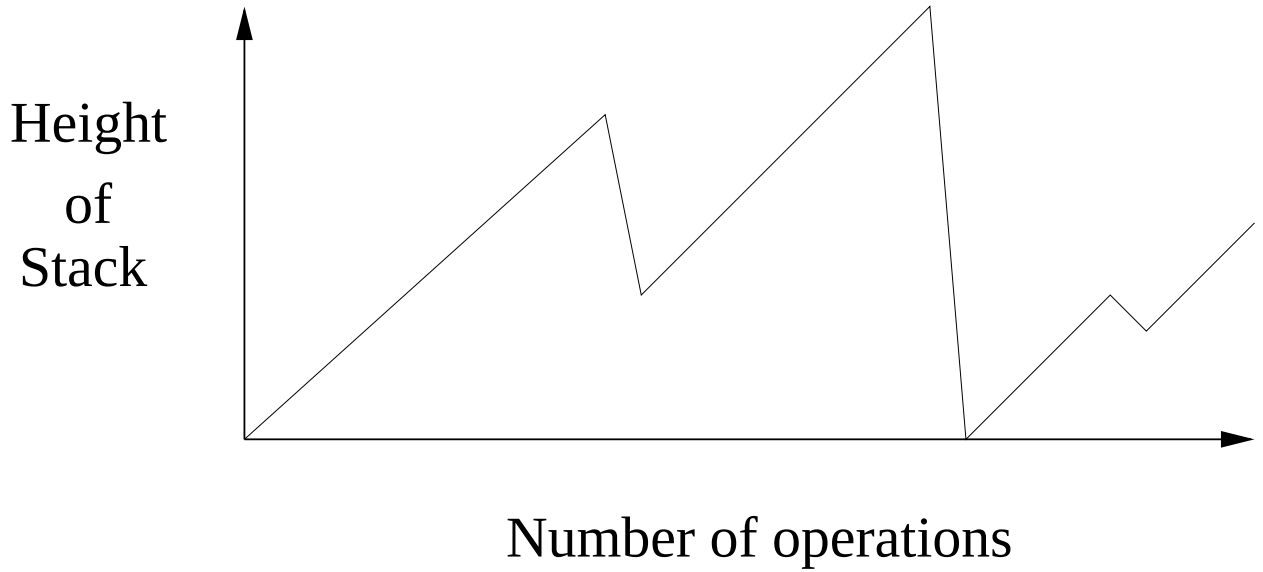
More accurate for larger i, which is also where its more important!

$$\left(\lim_{i \to \infty} \frac{i}{i+1}(c+b) = c+b \right)$$

This is called an *amortized analysis*. The cost of an expensive operation is amortized over the cheaper ones which *must* accompany it.

The Accounting Method for the Multi-delete Stack

Every time `push` is called we take a constant time (say a) to perform the operation, but we also put a constant amount of time (say b) in our "time-bank". When it comes time to perform multi-pop $mPop(i)$, if there are i items to delete, we must have at least ib time units in the bank.



Where Amortized Analysis Makes a Difference

In the block implementations of the data structures we have seen so far, we simply throw an exception when we try to add to a full structure.

Several implementations (e.g. `Java.util.ArrayList`) do not throw an exception in this case, but rather create an array *twice* the size, copy all the elements in the old array across to the new array, and then add the new element to the new array.

This is an expensive operation, but it can be shown that the *amortized* cost of the *add* operation is constant.

3. Asymptotic Growth Rates

We have talked about comparing data structure implementations — using either an empirical or analytical approach.

Focus on analytical:

- independent of run-time environment
- improves understanding of the data structures

We said we would be interested in comparisons in terms of *rates of growth*.

Theoretical analysis also permits a deeper comparison which the other methods don't — *comparison with the performance barrier inherent in problems...*

Wish to be able to make statements like:

> Searching for a given element in a block of n distinct elements using only equality testing takes n comparisons in the worst case.
>
> Searching for a given element in an ordered list takes at least $\log n$ comparisons in the worst case.

These are *lower bounds* (on the *worst case*) — they tell us that we are never going to do any better *no matter what algorithm we choose*.

Again they reflect growth rates (linear, logarithmic)

In this section, we formalise the ideas of analytical comparison and growth rates.

3.1 Why Asymptopia

We would like to have a *simple description* of behaviour for use in comparison.

- Evaluation may be misleading.
  Consider the functions $t_1 = 0.002m^2$, $t_2 = 0.2m$, $t_3 = 2\log m$.

  Evaluating at $m = 5$ gives $t_1 < t_2 < t_3$. This could be misleading — for "serious" values of m the picture is the opposite way around.

  Want a description of behaviour over the full range.

- Want a *closed form*.

  eg. $\frac{n(n+1)}{2}$ not $n + (n-1) + \cdots + 2 + 1$

  Some functions don't have closed forms, or they are difficult to find — want a closed form approximation

- Want simplicity.

  Difficult to see what $2^{n-\frac{1}{n}} \log n^2 + \frac{3}{2} n^{2-n}$ does. We want to abstract away from the smaller perturbations...

What simple function does it behave *like*?

Solution

Investigate what simple function the more complex one *tends to* or *asymptotically approaches* as the argument approaches infinity, ie *in the limit*.

Choosing large arguments has the effect of making less important terms fade away compared with important ones.

eg. What if we want to approximate $n^4 + n^2$ by n^4 ?

How much error?

n	n^4	n^2	$\frac{n^2}{n^4 + n^2}$
1	1	1	50%
2	16	4	20%
5	625	25	3.8%
10	10 000	100	1%
20	160 000	400	0.25%
50	6 250 000	2 500	0.04%

3.2 Comparison "in the Limit"

How well does one function approximate another?

Compare growth rates. Two basic comparisons...

1.
$$\frac{f(n)}{g(n)} \to 0 \quad \text{as} \quad n \to \infty$$

$\Rightarrow$ $f(n)$ grows more slowly than $g(n)$.

2.

$$\frac{f(n)}{g(n)} \to 1 \quad \text{as} \quad n \to \infty$$

$\Rightarrow$ $f(n)$ is *asymptotic* to $g(n)$.

In fact we won't even be this picky — we'll just be concerned whether the ratio approaches a constant $c > 0$.

$$\frac{f(n)}{g(n)} \to c \quad \text{as} \quad n \to \infty$$

This really highlights the distinction between different orders of growth — we don't care if the constant is 0.00000000001 !

3.3 'Big O' Notation

In order to talk about comparative growth rates more succinctly we use the *'big O'* notation...

Definition

$f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer $n_0 \geq 1$ such that, for all $n \geq n_0$,

$$f(n) \leq cg(n).$$

— f "grows" no faster than g, for sufficiently large n

— growth rate of f is *bounded from above* by g

Example:

Show (prove) that n^2 is $O(n^3)$.

Proof

We need to show that for some $c > 0$ and $n_0 \geq 1$,

$$n^2 \leq cn^3$$

for all $n \geq n_0$. This is equivalent to

$$1 \leq cn$$

for all $n \geq n_0$.

Choosing $c = n_0 = 1$ satisfies this inequality. □

Exercise:

Show that $5n$ is $O(3n)$.

Exercise:

Show that 143 is $O(1)$.

Exercise:

Show that for any constants a and b, an^3 is $O(bn^3)$.

Example:

Prove that n^3 is not $O(n^2)$.

Proof (by contradiction)

Assume that n^3 is $O(n^2)$. Then there exists *some* $c > 0$ and $n_0 \geq 1$ such that

$$n^3 \leq cn^2$$

for all $n \geq n_0$.

Now for *any* integer $m > 1$ we have $mn_0 > n_0$, and hence

$$(mn_0)^3 \leq c(mn_0)^2.$$

Re-arranging gives

$$\begin{aligned} m^3 n_0^3 &\leq c m^2 n_0^2 \\ m n_0 &\leq c \\ m &\leq \frac{c}{n_0} \end{aligned}$$

This is contradicted by any choice of m such that $m > \frac{c}{n_0}$. Thus the initial assumption is incorrect, and n^3 is *not* $O(n^2)$. $\square$

From these examples we can start to see that big O analysis focuses on *dominating terms*.

For example a polynomial

$$a_d n^d + a_{d-1} n^{d-1} + \cdots + a_2 n^2 + a_1 n + a_0$$

— $O(n^d)$

— is $O(n^m)$ for any $m > d$

— is not $O(n^l)$ for any $l < d$.

Here $a_d n^d$ is the dominating term, with *degree* d.

For non-polynomials identifying dominating terms may be more difficult.

Most common in CS

- polynomials — $1, n, n^2, n^3, \ldots$
- exponentials — $2^n, \ldots$
- logarithmic — $\log n, \ldots$

and combinations of these.

3.4 'Big Ω' Notation

Big O bounds from above. For example, if our algorithm operates in time $O(n^2)$ we know it grows *no worse* than n^2. But it might be a lot better!

We also want to talk about lower bounds — eg

> No search algorithm (among n distinct objects) using only equality testing can have (worst case time) growth rate better than linear in n.

We use *big* Ω.

Definition

$f(n)$ is $\Omega(g(n))$ if there are a constant $c > 0$ and an integer $n_0 \geq 1$ such that, for all $n \geq n_0$,

$$f(n) \geq cg(n).$$

— f grows no slower than g, for sufficiently large n

— growth rate of f is *bounded from below* by g

Note $f(n)$ is $\Omega(g(n))$ if and only if $g(n)$ is $O(f(n))$.

4. Analysis of Recursive Programs

Previously we've talked about:

- The power of recursive programs.
- The unavoidability of recursive programs (they go hand in hand with recursive data structures).
- The potentially high computational costs of recursive programs.

They are also the most difficult programs we will need to analyse.

It may not be too difficult to express the time or space behaviour recursively, in what we call a *recurrence relation* or *recurrence equation*, but general methods for solving these are beyond the scope of this unit.

However some can be solved by common sense!

Example:

What is the time complexity of the recursive addition program from Topic ???

```
public static int increment(int i) {return i + 1;}

public static int decrement(int i) {return i - 1;}

public static int add(int x, int y) {
  if (y == 0) return x;
  else return add(increment(x), decrement(y));
}
```

- `if`, `else`, `==`, `return`, etc — constant time
- `increment(x)`, `decrement(y)` — constant time
- `add(increment(x), decrement(y))`? — depends on size of `y`

Recursive call is same again, except `y` is decremented. Therefore, we know the time for `add(...,y)` in terms of the time for

`add(...,decrement(y))`.

More generally, we know the time for size n input in terms of the time for size $n-1$...

$$\begin{aligned} T(0) &= a \\ T(n) &= b + T(n-1), \quad n > 1 \end{aligned}$$

This is called a *recurrence relation*.

We would like to obtain a *closed form* — $T(n)$ in terms of n.

If we list the terms, its easy to pick up a pattern...

$$\begin{aligned} T(0) &= a \\ T(1) &= a + b \\ T(2) &= a + 2b \\ T(3) &= a + 3b \\ T(4) &= a + 4b \\ T(5) &= a + 5b \\ &\vdots \end{aligned}$$

From observing the list we can see that

$$T(n) = bn + a$$

Example:

```
public static int multiply(int x, int y) {
  if (y == 0) return 0;
  else  return add(x, multiply(x, decrement(y)));
}
```

- `if`, `else`, `==`, `return`, etc — constant time
- `decrement(y)` — constant time
- `add` — linear in size of 2nd argument
- `multiply` — ?

We use:

a	const for `add` terminating case
b	const for `add` recursive case
a'	const for `multiply` terminating case
b'	const for `multiply` recursive case
x	for the size of `x`
y	for the size of `y`
$T_{add}(y)$	time for `add` with 2nd argument y
$T(x, y)$	time for multiply with arguments x and y

Tabulate times for increasing y...

$$
\begin{aligned}
T(x,0) &= a' \\
T(x,1) &= b' + T(x,0) + T_{add}(0) = b' + a' + a \\
T(x,2) &= b' + T(x,1) + T_{add}(x) = 2b' + a' + xb + 2a \\
T(x,3) &= b' + T(x,2) + T_{add}(2x) = 3b' + a' + (xb + 2xb) + 3a \\
T(x,4) &= b' + T(x,3) + T_{add}(3x) = 4b' + a' + (xb + 2xb + 3xb) + 4a \\
&\vdots
\end{aligned}
$$

Can see a pattern of the form

$$T(x,y) = yb' + a' + [1 + 2 + 3 + \cdots + (y-1)]xb + ya$$

We would like a closed form for the term $[1 + 2 + 3 + \cdots + (y - 1)]xb$.

Notice that, for example

$$1 + 2 + 3 + 4 = (1 + 4) + (2 + 3) = \frac{4}{2}.5$$

$$1 + 2 + 3 + 4 + 5 = (1 + 5) + (2 + 4) + 3 = \frac{5}{2}.6$$

In general,

$$1 + 2 + \cdots + (y - 1) = (\frac{y - 1}{2}).y = \frac{1}{2}y^2 - \frac{1}{2}y$$

(Prove inductively!)

Overall we get an equation of the form

$$a'' + b''y + c''xy + d''xy^2$$

for some constants a'', b'', c'', d''.

Dominant term is xy^2:

— linear in x (hold y constant)

— quadratic in y (hold x constant)

There are a number of well established results for different types of problems. We will draw upon these as necessary.

5. Summary

Choosing performance measures

- worst case — simple, guarantees upper bounds
- expected case — averages behaviour, need to know probability distribution
- amortized case — may 'distribute' time for expensive operation over those which *must* accompany it

Asymptotic growth rates

- compare algorithms
- compare with inherent performance barriers
- provide simple closed form approximations
- big O — upper bounds on growth
- big Ω — lower bounds on growth

Analysis of recursive programs

- express as recurrence relation
- look for pattern to find closed form
- can then do asymptotic analysis