CompSci 220

- Data Structures & Algorithms
- Slides written by AProf Gimel'farb & modified by Mike Barley

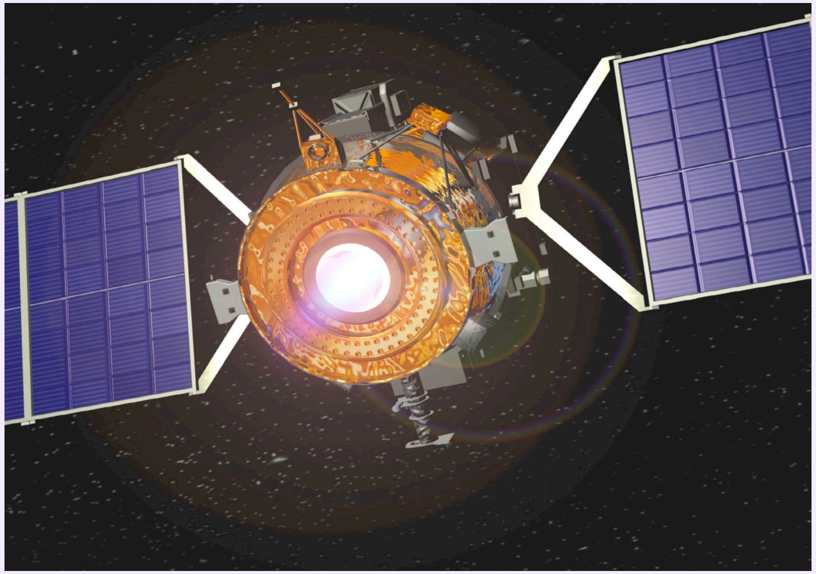
Contact Details

- Lecturer: Mike Barley
- Office Hours: By arrangement
- City Office: Room 394
- Tamaki Office: TBA
- Email: barley@cs.auckland.ac.nz
- Ph ext: x86133 (almost never in my office)

Overview to My Part

- This part is all about analysing how long an algorithm will “run”:
  - Intro to basic “tools”
  - Applying these tools to sorting algorithms
  - Applying these tools to searching algorithms

Who am I
and why am I teaching this?



- My area of expertise is Artificial Intelligence
- I have never taught this part before
- It's been a number of decades since I last looked at this area
- However, I have become increasingly interested in this area: *intelligent automatic software configuration*

Division of My Part

- Intro to tools: 5 lectures
- Intro to sorting: 3 1/2 lectures
- Intro to search: 2 1/2 lectures
- On the 12th lecture we rest :^)

The 5 “Tool” Lectures

- Terms & Definitions & Examples (today)
- Estimating Running Time (also today)
- Complexity Measures (Thursday)
- Computing Simple Time Complexities (Tuesday next)
- Computing Time Complexities of Recursion (also Tuesday next)

Overview of Today’s 1st Half

- Defining basic terms
- Bases for describing & comparing algorithms
- Working thru simple examples
- Exercises

Pattern for Today’s 1st Half’s Examples

- Problem description
- Naïve algorithm
- Brief analysis leading to insights about its complexity
- More Sophisticated algorithm arising from insight
- Brief statement about its complexity

Some Informal Definitions

- **algorithm** - a system of uniquely determined rules that specify successive steps in solving a problem
- **program** - a clearly specified series of computer instructions implementing the algorithm
- **elementary operation -** a computer instruction executed in a single time unit (computing step)
- **running** (computing) **time** of an algorithm **-** a number of its computing steps (elementary operations)

Efficiency of Algorithms: How to compare algorithms / programs

- by **domain of definition** – what inputs are legal?
- by **correctness** – is output correct for each legal input? (in fact, you need a **formal proof**!)
- by **basic resources** – ***maximum*** or ***average*** requirements:
  - **computing time**
  - **memory space**

Example 1: $s = \sum_{i=0}^{n-1} a[i]$

<u>Problem Statement:</u> given an array of n numbers sum them together.

Example 1: $s = \sum_{i=0}^{n-1} a[i]$

Naïve Algorithm:

Algorithm sum (**input**: array $a[n]$)

begin $s \leftarrow 0$

for $i \leftarrow 0$ **step** $i \leftarrow i + 1$ **until** $n - 1$ **do**

$s \leftarrow s + a[i]$ **end for**

return s

end

Example 1: $s = \sum_{i=0}^{n-1} a[i]$

<u>Brief Statement of Complexity:</u>

To sum elements of an array $a[n]$, elementary add operations are repeated n times $\Rightarrow$

Running time $T(n) = cn$ is **linear** in n

This is as good as it gets.

Example 2: GCD

- **Problem:** The **greatest common divisor**, k = GCD(n, m) is the greatest positive integer such that it divides both two positive integers m and n
- **Examples:** $GCD(2, 17) = 1$, $GCD(6, 9) = 3$, $GCD(12, 20) = 4$
- ***Naïve Algorithm:*** A ***"brute-force"*** **linear** solution: to exhaust all integers from the minimum of m and n, to the first one that divides both m and n

Working out an example

1. **9245 / 7515 = 0?**
2. **9245 / 7514 = 0 & 7515 / 7514 = 0?**

…

7511. 9245 / 5 = 0 & 7515 / 5 = 0?

- Is it practicable to use such an algorithm to find **GCD(9245, 7515)** or what about **GCD(3,787,776,332, 3,555,684776)?**

Naive GCD Analysis

- Let *m* > *n*, what do we learn when we divide *m* by *n*?
- If the reminder = 0, what does that tell us?
- If the remainder > 0, what does that tell us?

Euclid's Insight

- ***Euclid's analysis***: if k divides both m and n, then it divides their difference ($n - m$ if $n > m$):
- I.e., let n = c * k and m = d * k
  then n - m = (c - d) * k
  therefore GCD(n, m) = GCD(n-m, m).
- Therefore

$$\mathbf{GCD}(\boldsymbol{n}, \boldsymbol{m}) = \mathbf{GCD}(\boldsymbol{n}-\boldsymbol{m}, \boldsymbol{m})$$

Why??

Euclid's Insight

Since GCD(n,m) = GCD(n-m, m)

then GCD(n,m) = GCD(n-2m, m)

and k divides every difference

when the subtraction is repeated λ

times until $n - \lambda m < m$

Therefore $\mathbf{GCD}(\boldsymbol{n}, \boldsymbol{m}) = \mathbf{GCD}(\ \boldsymbol{n}\ \mathbf{mod}\ \boldsymbol{m}, \boldsymbol{m}\ )$

where $\boldsymbol{n}$ **mod m** is the ***remainder*** of division of n by m
(in Java/C: ***n%m***, e.g. 13%5 = 3)

If the remainder > 0, what does that tell us?

- It tells us a new smaller number that has the same GCD with *m* and with *n* as *m* and *n.*
- *How can we use this info to our advantage?*
- We don’t have to try every integer between the min of *m* and *n,* we need only try the remainders of the divisions.

Euclid's GCD Algorithm

More Sophisticated Algorithm:

```
GCD(input: int max, min)   // assume that max > min
begin  if min == 0
       then return max
       else return  GCD(min, max mod min)
       endif
end
```

Is it correct?

How would you prove it?

What is its running time?

How would you determine that?

Euclid's GCD $\approx c\log(n+m)$ time

GCD(9245,7515) = 5

9245 **mod** 7515 = 1730	7515 **mod** 1730 = 595
1730 **mod** 595 = 540	595 **mod** 540 = 55
540 **mod** 55 = 45	55 **mod** 45 = 10
45 **mod** 10 = 5	10 **mod** 5 = **0** $\Rightarrow$ **GCD=5**

8 steps vs **7511** steps of the brute-force algorithm!

Example 3: Sums of Subarrays

Problem Statement:

Given an array $(a[i]: i = 0, 1, \ldots, n - 1)$ of size n, compute $n - m + 1$ sums:

$$s[j] = \sum_{k=0}^{m-1} a[j+k];\ j = 0, \ldots, n - m$$

of all contiguous subarrays of size m

Sums of Subarrays

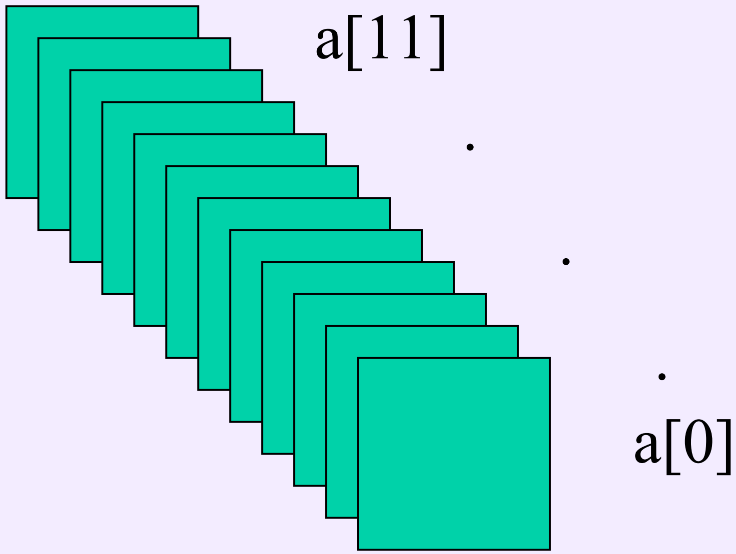
Worked example:

Let j = 3, m = 4

n = 12, then

s[3] = a[3] + a[4] +

a[5] + a[6]



Naïve Algorithm (2 *nested loops*)

Algorithm slowsum (**input:** array $a[2m]$)
  begin array $s[m + 1]$
    for $j \leftarrow 0$ **to** m **do**
      $s[j] \leftarrow 0$
      for $k \leftarrow 0$ **to** $m-1$ **do**
        $s[j] \leftarrow s[j] + a[k + j]$
      end for
    end for
    return s
  end

Sums of Subarrays

- **Complexity** : cm operations per subarray; in total: $cm(n - m + 1)$ operations
- Time is **linear** if m is fixed and **quadratic** if m is growing with n, such as $m = 0.5n$

$$\boxed{T(n) = c\frac{n}{2}\left(\frac{n}{2}+1\right) \cong c' \cdot n^2 = n^2 T(1)}$$

Getting Linear Computing Time

Quadratic time due to reiterated innermost computations:

$$s[j] = a[j] + \underline{a[j+1] + \ldots + a[j+m-1]}$$

$$s[j+1] = \qquad \underline{a[j+1] + \ldots + a[j+m-1]} + a[j+m]$$

How many times is **a[k]** added?

Linear time $T(n) = c(m + 2m) = 1.5cn$ after excluding reiterated computations:

$$s[j+1] = s[j] + a[j+m] - a[j]$$

More sophisticated algorithm

Algorithm fastsum (**input**: array $a[2m]$)

begin array $s[m+1]$

compute $s[0]$

compute s[j] for $j \leftarrow 1$ **to** m

return s

end

Linear time (2 *simple loops*)

```
Algorithm fastsum (input: array a[2m])
  begin array s[m + 1]
      s[0] ← 0
      for k ← 0 to m−1 do
          s[0] ← s[0] + a[k]
      end for
      for j ← 1 to m do
          s[j] ← s[j−1] + a[j + m − 1] − a[j − 1]
      end for
      return s
  end
```

Computing Time for T(1)=1μs

Array size	n	2,000	2,000,000
Size / number of subarrays	m / $m + 1$	1,000 / 1,001	1,000,000 / 1,000,001
Naïve (***quadratic***) algorithm	$T(n)$	2 *sec*	> 23 *days*
Efficient (***linear***) algorithm	$T(n)$	1.5 *msec*	1.5 *sec*

Exercises: Textbook, p.12

1.1.1: Quadratic algorithm with processing time $T(n)=cn^2$ spends $500\mu sec$ on 10 data items. What time will be spent on 1000 data items?

Solution: $T(10) = c\cdot 10^2 = 500 \rightarrow c = 500/100 = 5\ \ \mu sec/item$
$\rightarrow\ T(1000) = 5\cdot 1000^2 = 5\cdot 10^6$ μsec or $T(1000)= 5\ sec$

1.1.2: Algorithms **A** and **B** use $T_A(n) = c_A n\log_2 n$ and $T_B(n) = c_B n^2$ elementary operations for a problem of size n. Find the fastest algorithm for processing $n = 2^{20}$ data items if **A** and **B** spend 10 and 1 operations, respectively, to process $2^{10}=1024$ items.

Solution: $T_A(2^{10}) = 10 \rightarrow \boldsymbol{c_A} = 10/(10\cdot 2^{10}) = \mathbf{2^{-10}}$;
$T_B(2^{10}) = 1 \rightarrow \boldsymbol{c_B} = 1/2^{20} = \mathbf{2^{-20}}$
$\rightarrow \boldsymbol{T_A(2^{20})} = 2^{-10}\cdot 20 \cdot 2^{20} = \mathbf{20\cdot 2^{10}} \ll \boldsymbol{T_B(2^{20})} = 2^{-20} \cdot 2^{40} = \mathbf{2^{20}} \rightarrow$
Algorithm **A** is the fastest for $n = 2^{20}$

2nd Half: Estimating Running Time

The Heart of Algorithmic Complexity (AC)

- The Question that AC is normally to answer is: *Assume we know how long it takes for algorithm A to run for n "items", approximately how long will it take for 2n items?*
- Answering this type of question typically involves "counting" how many elementary operations occur per item.
- Unfortunately, we usually need more sophisticated counting techniques than using one's fingers.

Counting Elementary Ops

Algorithm slowsum (**input**: array $a[2m]$)
  begin array $s[m + 1]$
    for $j \leftarrow 0$ **to** m **do**
      $s[j] \leftarrow 0$
      for $k \leftarrow 0$ **to** $m-1$ **do**
        $s[j] \leftarrow s[j] + a[k + j]$
      end for
    end for
    return s
  end

Estimated Time to Sum Subarrays

- Ignore data initialisation
- "Brute-force" summing with two nested loops:

$$T(n) = m(m+1) = {}^{n}/_{2}\left({}^{n}/_{2} + 1\right)$$
$$= 0.25n^2 + 0.5n$$

- For a large n, $\mathrm{T}(n) \cong 0.25n^2$
  - e.g., if $n \geq 10$, the linear term $0.5n \leq 16.7\%$ of $\mathrm{T}(n)$
  - if $n \geq 500$, the linear term $0.5n \leq 0.4\%$ of $\mathrm{T}(n)$

Quadratic vs linear term

$T(n) = 0.25n^2 + 0.5n$				
n	$T(n)$	$0.25n^2$	$0.5n$	
10	30	25	5	16.7%
50	650	625	25	3.8%
100	2550	2500	50	2.0%
500	62750	62500	250	0.4%
1000	250500	250000	500	0.2%

Quadratic Time to Sum Subarrays: $T(n) = 0.25n^2 + 0.5n$

- Factor $c = 0.25$ is referred to as a "**constant of proportionality**"
- An actual value of the factor does not effect the behaviour of the algorithm for a large n:
  - Double value of $n \rightarrow$ 4-fold increase in $T(n)$:

$$T(2n) = 4\ T(n)$$

Running Time: Estimation Rules

- Running time is proportional to the **most significant term** in $T(n)$
- Once a problem size becomes large, the most significant term is that which has the largest power of n
- This term increases faster than other terms which reduce in significance

Running Time: Estimation Rules

- Constants of proportionality depend on the compiler, language, computer, etc.
  – It is useful to ignore the constants when analysing algorithms.
- Constants of proportionality are reduced by using faster hardware or minimising time spent on the “inner loop”
  – *But this would not effect behaviour of an algorithm for a large problem!*

Elementary Operations

- Basic arithmetic operations (**+** ; – ; $*$; / ; **%**)
- Basic relational operators (==, **!**=, >, <, >=, <=)
- Basic Boolean operations (AND,OR,NOT)
- Branch operations, **return**, …

Input for problem domains (meaning of n):

Sorting: n items

Graph / path: n vertices / edges

Image processing: n pixels

Text processing: string length

Estimating Running Time

- **Simplifying assumptions:**

  all elementary statements / expressions take the same amount of time to execute

  - e.g., simple arithmetic assignments
  - `return`

- Loops increase in time **linearly** as

$$k \cdot T_{\text{body of a loop}}$$

where k is number of times the loop is executed

Estimating Running Time

- Conditional / switch statements like **if** {*condition*} **then** {*const time* T_1} **else** {*const time* T_2} are more complicated (one has to account for branching frequencies: $T = f_{\text{true}} T_1 + (1 - f_{\text{true}}) T_2 \le \max\{T_1, T_2\}$
- **Function calls**:

$$T_{\text{function}} = \sum T_{\text{statements in function}}$$

- **Function composition**:

$$T(f(g(n))) = T(g(n)) + T(f(n))$$

Example 1.6: Textbook, p.13

Logarithmic time due to an exponential change $i = k, k^2, k^3, \ldots, k^m$ of the loop control in the range $1 \le i \le n$:

for $i = k$ **step** $i \leftarrow ik$ **until** n **do**
… {const # of elementary operations}
end for

m iterations such that $k^{m-1} < n \le k^m \Rightarrow$
$T(n) = c \lceil \log_k n \rceil$

Example 1.7: Textbook, p.13

<u>$n \log n$ running time</u> of the conditional nested loops:

m $\leftarrow 2$; **for** $j \leftarrow 1$ **to** n **do**
  if $(j = m)$ **then**
    $m \leftarrow 2m$
    for $i \leftarrow 1$ **to** *n* **do** …{const # of operations}
    end for
  end if
end for

The inner loop is executed k times for $j = 2, 4, \ldots, 2^k$;
$k < \log_2 n \le k + 1$; in total: $\boxed{T(n) = kn = n \lfloor \log_k n \rfloor}$

Exercise 1.2.1: Textbook, p.14

Conditional nested loops: linear or quadratic running time?

$m \leftarrow 1$; **for** $j \leftarrow 1$ **to** n **do**

  if $(j = m)$ **then** $m \leftarrow m(n-1)$

   for $i \leftarrow 1$ **to** n **do** …{const # of operations}

   end for

  end if

end for

The inner loop is executed only twice, for $j = 1$ and $j = n - 1$;
  in total: $T(n)=2n$ → **linear** running time