CS2521: Algorithm Analysis

N. Oren
n.oren@abdn.ac.uk

University of Aberdeen

Recap

- Algorithms describe a correct solution to a problem.
- We've begun examining how correctness can be checked for.
- Algorithms must also be efficient.
- How do we measure efficiency?

Algorithm Efficiency

- The efficiency of an algorithm is measured in terms of the resources it uses.
  - Time
  - Memory (space)
  - Network
- We want our analysis to be machine independent.
- We (usually) assume a single processor computer with unlimited memory.
- We call this model the RAM model.

The RAM model of Computation

- Each simple operation, e.g. (+,*,/,=,if, etc) takes one time step.
- Memory access takes one time step.
- Loops and subroutines are not simple operations. They are composed of their constituent operations.

The RAM model of Computation

- We measure run time by counting up the number of steps an algorithm takes on a problem instance.
- If we assume a fixed number of steps per second, we can trivially translate to clock time.
- Is the RAM model realistic?
  - multiplication takes longer than addition.
  - Complier optimisation (e.g. loop unrolling) can speed things up.
  - Memory access time changes depending on whether data is in cache, main memory or a page on the hard drive.
- All three assumptions are violated.
- So why do we still use this model?

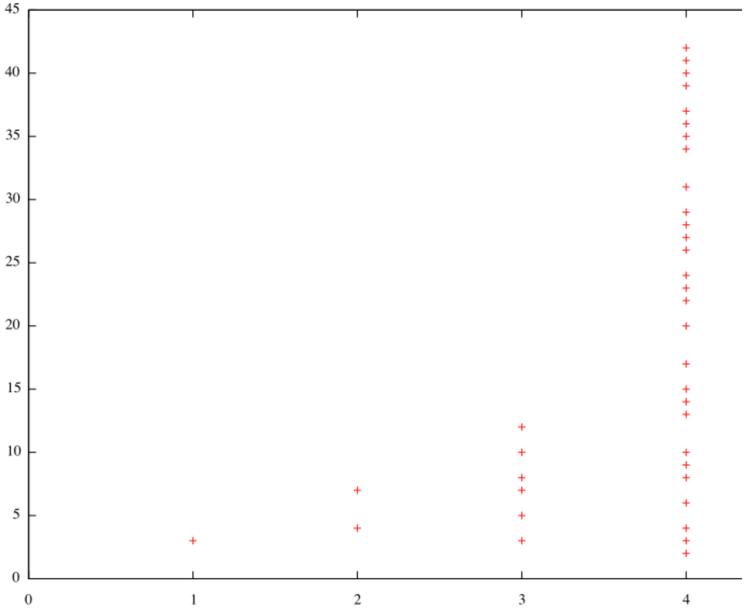
The RAM model of Computation

- We measure run time by counting up the number of steps an algorithm takes on a problem instance.
- If we assume a fixed number of steps per second, we can trivially translate to clock time.
- Is the RAM model realistic?
  - multiplication takes longer than addition.
  - Complier optimisation (e.g. loop unrolling) can speed things up.
  - Memory access time changes depending on whether data is in cache, main memory or a page on the hard drive.
- All three assumptions are violated.
- So why do we still use this model? The model works well at its level of abstraction (c.f. flat earth model).
- It is difficult to find a problem where the RAM model gives substantially different results (e.g. days vs. seconds) than the real world

The Good, Bad and Ugly

- We can measure how many steps an algorithm takes on a specific input instance.
- So what makes an algorithm good, or bad?
- We need to evaluate how it works over <u>all possible</u> instances.
- For the sorting problem, the set of possible input instances is
  - All possible arrangements of n keys
  - For all possible values of n

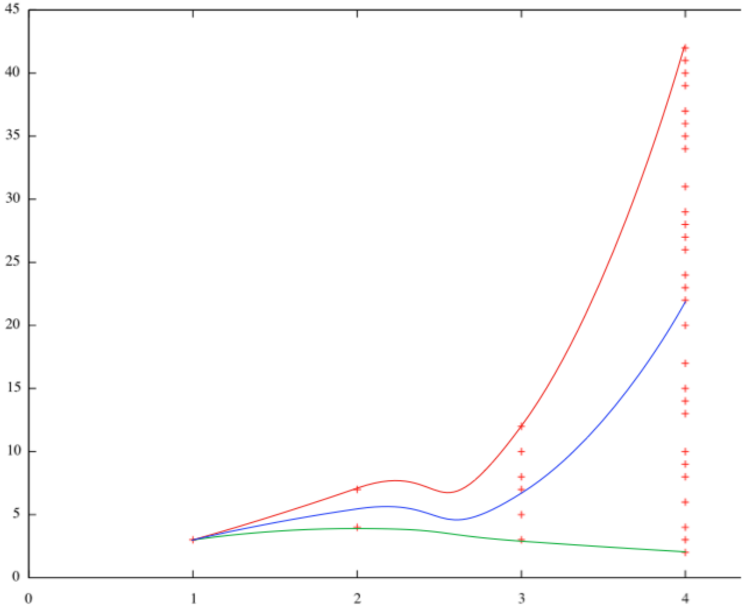
Complexity



Complexity

- We deal with integer sizes.
- We can (roughly) identify 3 cases
  - Best case complexity: the function defined by the minimum number of steps taken in any instance of size n.
  - Worst case complexity: the function defined by the maximum number of steps in any instance of size n.
  - Average case complexity: the function defined by the average number of steps over all instances of size n.

Complexity



Complexity

- In many situations, the worst case is the one that's important.
- The average case is often hard to compute (and define).
- What's the best, worst, average case when playing poker machines?
- Complexities can be represented as a numerical function of time (t) versus problem size (n). E.g. $t = n^2 - 4n + 3$

Example

Require: P, a set of n points

```
function NearestNeighbour(P)
    pick and visit an initial point p_0                          ▷ 1
    i=0                                                          ▷ 1
    while there are still unvisited points do
        i=i+1                                                    ▷ 1
        Let p_i be the closest unvisited point to p_{i-1}        ▷ 5*(|P|-i)
        Visit p_i                                                ▷ 1
    end while
    Visit p_0                                                    ▷ 1
end function
```

Example

Require: P, a set of n points

function NearestNeighbour(P)
  pick and visit an initial point p_0 $\triangleright$ 1
  i=0 $\triangleright$ 1
  while there are still unvisited points **do**
    i=i+1 $\triangleright$ 1
    Let p_i be the closest unvisited point to p_{i-1} $\triangleright$ 5*(|P|-i)
    Visit p_i $\triangleright$ 1
  end while
  Visit p_0 $\triangleright$ 1
end function

- Checking distance: $(x - x')^2 + (y - y')^2 = 5$ operations
- Complexity=$3 + \sum_{i=0}^{|P|} 2 + 5(|P| - i) = 3 + 2|P| + 5\sum_{i=0}^{|P|} i = 3 + 2|P| + 5|P|(5|P| - 1)/2 = 25/2|P|^2 - 1/2|P| + 3$
- What if we preprocess distances?

Complexity

- Exact functions are often difficult to work with.
  - They often have bumps (e.g. work slightly better if size is a power of 2).
  - Require too much detail to specify, and might depend on low level implementation/optimisation details.
- We care about the "big picture". E.g. what can we say about $T(n) = 54n^2 + 433n + 1043log_2(n) + 27$?
- We use O-notation to speak about time complexity. This notation ignores low level details.

O-notation

- O-notation ignores the differences between multiplicative constants.

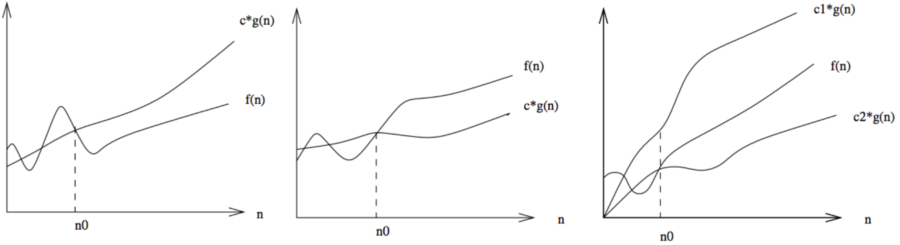
$$24n^3 \qquad 5n^3 \qquad 0.7n^3$$

- Such multiplicative differences are often implementation dependant (e.g. Java vs C++).
- We have 3 primary definitions:
  - $f(n) \in O(g(n))$: $c \cdot g(n)$ is a upper bound on $f(n)$. I.e. there is some constant $c > 0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$ where n_0 is a constant.
  - $f(n) \in \Omega(g(n))$: $c \cdot g(n)$ is a lower bound on $f(n)$. I.e. there is some constant $c > 0$ s.t. $f(n) \geq cg(n)$ for all $n \geq n_0$.
  - $f(n) \in \Theta(g(n))$: $c_1 g(n)$ provides an upper bound on $f(n)$ and $c_2 g(n)$ places a lower bound on $f(n)$ for all $n \geq n_0$.

Notation Abuse

- We write f(n)=O(g(n)) to denote that $f(n)$ is in the set $O(g(n))$
- Similarly for Θ and Ω
- This is an abuse of notation, but is commonly accepted.

O-notation



$$O(g(n)), \Omega(g(n)), \Theta(g(n))$$

O-notation

- So what does O-notation actually tell us?

O-notation

- So what does O-notation actually tell us?
- It allows us to do a rough comparison of equality when comparing algorithms.
- Algorithms with the same complexity are (roughly) equivalent to each other; if we can use an algorithm with complexity c to one one input set, then another algorithm with identical complexity would be approximately as good (for large input sets).
- Note: we only care about large input sets (i.e. when $n > n_0$)
- We also assume that $g(n)$ is non-negative for sufficiently large n.

Examples

- $3n^2 - 100n + 6 = O(n^2)$, we set $c = 3$ and $3n^2 > 3n^2 - 100n + 6 = O(n^2)$
- $3n^2 - 100n + 6 = O(n^3)$, set $c = 1$ and $n_0 = 4$
- $3n^2 - 100n + 6 \neq O(n)$ as $cn < 3n^2$ when $n > c$

Examples

- $3n^2 - 100n + 6 = \Omega(n^2)$, we set $c = 2$ and $2n^2 < 3n^2 - 100n + 6$
- $3n^2 - 100n + 6 \neq \Omega(n^3)$
- $3n^2 - 100n + 6 = \Omega(n)$ as for any c $cn < 3n^2 - 100n + 6$ when $n > 100c$

Example

- $3n^2 - 100n + 6 = \Theta(n^2)$ as both O and Ω apply
- $3n^2 - 100n + 6 \neq \Theta(n^3)$ as only o applies
- $3n^2 - 100n + 6 \neq \Theta(n)$ as only Ω applies.

Example

Is $2^{n+1} = \Theta(2^n)$?
Solve by going back to definitions.

- Is $2^{n+1} = O(2^n)$? It is if and only if (iff) there is a c such that for all sufficiently large n, $f(n) \leq cg(n)$. Now $2^{n+1} = 2 \cdot 2^n \leq c \cdot 2^n$ for any $c \geq 2$.
- Is $2^{n+1} = \Omega(2^n)$? It is iff there is a $c > 0$ s.t. $f(n) \geq c \cdot g(n)$. This is satisfied for any $0 < c \leq 2$.
- Together, this means that $2^{n+1} = \Theta(2^n)$.

Example

- Is $(x+y)^2 = O(x^2+y^2)$?

$$(x+y)^2 = x^2 + y^2 + 2xy$$

- If $x > y$ then

$$2xy < 2x^2 < 2(x^2+y^2)$$

- Similarly for $y > x$
- So middle term is always smaller or equal to $2(x^2+y^2)$
- So $(x+y)^2 \leq 3(x^2+y^2)$
- So result holds.

Why do we care?

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Dominance

- O-notation groups functions into classes. E.g. $f(n) = 0.34n$ and $g(n) = 1234n$ are in the same class, $\Theta(n)$.
- If two functions are in different classes, they will differ in notation, either $f(n) = O(g(n))$ or $g(n) = O(f(n))$ but not both.
- A faster growing function dominates a slower growing one. g dominates f when $f(n) = O(g(n))$ and $f(n) \neq \Theta(g(n))$. We can write this as $g >> f$.
- Formally, $f(n)$ dominates $g(n)$ if $\lim_{n\to\infty} g(n)/f(n) = 0$
- Does $2n^2$ dominate n^2?

Some common classes

- Constant functions: $\Theta(1)$ e.g. printing out a string, adding or comparing two numbers. No dependence on n.
- Logarithmic functions: $\Theta(\log(n))$. Binary search. Grows slowly, but still grows.
- Linear functions: $\Theta(n)$. scanning an array to find largest/smallest item, or compute average.
- Superlinear functions: $\Theta(n \log(n))$. Arises in various sorting algorithms. Grow faster than linear.
- Quadratic functions: $\Theta(n^2)$. Typically occurs when examining most or all pairs of items given N elements. Common in sorting.
- Cubic functions: $\Theta(n^3)$ occurs when examining all triples of items.

Some common classes

- Exponential functions: $\Theta(c^n)$ for some constant $c > 1$. Typicaly occurs when enumerating all subsets. These grow very quickly.
- Factorial functions: $\Theta(n!)$ occurs when generating all permutations of n items.
- Lots of other classes exist (e.g. $\log \log n$).
- Dominance:

$$n! >> 2^n >> n^3 >> n^2 >> nlogn >> n >> logn >> 1$$

Working with O

- The sum of two functions is governed by the dominant one. $(n^3 + n^2 = O(n^3))$
  - $O(f(n)) + O(g(n)) \rightarrow O(max(f(n), g(n)))$
  - $\Omega(f(n)) + \Omega(g(n)) \rightarrow \Omega(max(f(n), g(n)))$
  - $\Theta(f(n)) + \Theta(g(n)) \rightarrow \Theta(max(f(n), g(n)))$
- Multiplying by a constant does not have any effect, e.g. $O(cf(n)) = O(f(n))$
- Multiplying by functions requires more care
  - $O(f(n)) \times O(g(n)) \rightarrow O(f(n) \times g(n))$
  - $O(n \times n)$ dominates $O(n)$
- **Homework** (discussed in practical): Show that O-notation relationships are transitive.

Back to Algorithms

```
function SelectionSort(s)
    for all i=0 to n-1 do
        min=i
        for all j=i+1 to n-1 do
            if s[j]<s[min] then
                min=j
            end if
            swap s[i],s[min]
        end for
    end for
end function
```

- The outer loop repeats n times, the inner less than n times, so $O(n^2)$
- The outer loop repeats n times, the inner loop repeats n-i-1 times.

$$\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} n - i - 1$$

- We're adding up $(n-1) + (n-2) + \ldots + 2 + 1$
- This is approximately $n(n-1)/2$, i.e. $\Omega(n^2)$ (and $\Theta(n^{2)}))$.

Analysing the Insertion Sort

Require: a sequence $L = a_1, \ldots, a_n$

function insertion sort(L)
    for all $i = 1$ to length(L) **do**
        $j = i$
        while $j > 0$ and $a_j < a_{j-1}$ **do**
            swap a_j and a_{j-1}
            $j = j - 1$
        end while
    end for
    return L
end function

- To compute complexity, look at how many times the inner while loop iterates.
- Difficulty: two possible termination conditions.
- In the worst case, forget about early termination, so loop always goes around i times.
- Looking at outer loop, we have $i = 1, \ldots n$
- So $1 + 2 + \ldots n = \sum_{i=1}^{n} i = n(n+1)/2$
- So in the worst case $O(n^2)$

Example 2: String Matching

```
function matchString(s,t)
    ls=length(s)
    lt=length(t)
    for all i=0 to (ls-lt) do
        j=0
        while j<lt & s[i+j]=t[j] do
            j++
            if j=lt return i
        end while
    end for
    return -1
end function
```

- Assume string of length n, target length m.
- Outer for loop repeats $n - m$ times, inner loop maximum m times + 2 additional instructions. length(s)=m, length(t)=n.
- $m + 2 = \Theta(m) \rightarrow O(n + m + (n - m)m) \rightarrow O(n + m + nm - m^2)$
- Now $n \geq m$, so $n + m \leq 2n = \Theta(n)$ so $O(n + nm - m^2)$
- $n \leq nm$ so $n + nm = \Theta(nm)$ so $O(nm - m^2)$.
- We're seek upper bound, so drop m^2 term (as $n \geq m \rightarrow mn \geq m^2$).
- So $O(nm) = O(n^2)$ in worst case.

Example 3

- Finding a number in a sorted array with n elements.
- Brute force approach: $O(n)$. What is the average case complexity? Why?
- Smarter approach: binary search. **Homework (discussed in practical):** prove correctness of binary search.

```
function binarySearch(array, value)
    midpoint = ⌊|array|/2⌋                              ▷ ⌊...⌋ means round down
    if |array| = 1 and value != array[0] then           ▷ |array| means size of array
        return false
    else if value == array[midpoint] then
        return true
    else if array[midpoint] < value then
        return binarySearch(array[midpoint+1..|array|-1],value)
    else
        return binarySearch(array[0..midpoint-1],value)
    end if
end function
```

Example 3

```
1:  function binarySearch(array, value)
2:      midpoint = ⌊|array|/2⌋                          ▷ ⌊...⌋ means round down
3:      if |array| = 1 and value! = array[0] then      ▷ |array| means size of array
4:          return false
5:      else if value == array[midpoint] then
6:          return true
7:      else if array[midpoint] < value then
8:          return binarySearch(array[midpoint+1..|array|],value)
9:      else
10:         return binarySearch(array[0..midpoint-1],value)
11:     end if
12: end function
```

- Each time line 8 or 10 is invoked, it is run on an array half the size of the original.
- Given an array of length $n = 2^x$, binarySearch will be run x times.
- $log_2(n) = x$, so binary search is of $O(log(n))$ complexity.

Divide and Conquer

- Binary search is an example of a divide and conquer algorithm. We
  - Divide the problem into sub-problems.
  - Conquer each subproblem (in the above example, recursively).
  - Combine the subproblem solutions into a solution for the overall problem.

Divide and Conquer II

- Computing a^n is $O(?)$

Divide and Conquer II

- Computing a^n is $O(n)$ as $a \times a \ldots \times a$ requires $n-1$ multiplications.
- But note that $a^n = (a^{n/2})^2$ (if n is even) or $a^n = a(a^{(n-1)/2})^2$ (if odd).
- We can use this to build a recursive algorithm:

```
function power(a,n)
    if n = 1 return a
    x = power(a, ⌊n/2⌋)
    if n is even return x^2
    else return a × x^2
end function
```

- $O(logn)$. In cryptography, where $n \sim 2^{4096}$ this is a big deal, 12 operations vs 4095, you need 1/350th the number of computers in your server farm.

A bit more about logarithms

- $b^x = y$ is the same as $x = log_b y$
  - b is the base. If $b = 2$ we're dealing with the binary logarithm.
  - If $b = e$ it's the natural logarithm.
  - $log_a(xy) = log_a(x) + log_a(y)$
  - $log_a b = \frac{log_c b}{log_c a}$
- This last result indicates that the base of the logarithm isn't important when computing complexity, as it just shifts things by a constant amount.
- The growth rate of the logarithm of any polynomial function is $O(\log n)$ as $\log n^b = b \log n$