COMP2230/COMP6230 Algorithms

Lecture 2

Table of Contents

Analysing Algorithms

We typically care about two types of computational complexity in relation to an algorithm.

- How long it takes to run.
- How much space it takes up.

Time Complexity

Definition (Time Complexity)

The *time complexity* of an algorithm is the computational complexity that describes the amount of time it takes to run an algorithm as a function of the size of the input to the algorithm. Time complexity is often estimated by the number of elementary operations performed by the algorithm, with the assumption that an elementary operation takes a fixed amount of time to perform.

Assumptions

Under the assumption that the elementary operations take a fixed amount of time to perform, then the number of operations and the amount of time taken differ by no more than a constant fact.

That is, if the amount of time taken by the algorithm is $f(n)$, and the number of elementary operations required by the algorithm is $g(n)$, then $f(n) = cg(n)$ for some constant c.

Space Complexity

Definition (Space Complexity)

The *space complexity* of an algorithm is the amount of memory space required to solve an instance of the computational problem, as a function of the size of the input to the algorithm.

Categorisation of Problems

There are many ways to categorise problems. There are generally four overarching categories.

- Unsolvable
- Intractable
- Problems of Unknown Complexity
- Feasible/Tractable Problems

Types of Time Complexity

Definition (Worst Case Time)

The *worst case time* is the maximum time needed to execute the algorithm, taken over all inputs of size n.

Definition (Average Case Time)

The *average case time* is the average time needed to execute the algorithm, taken over all inputs of size n.

Definition (Best Case Time)

The *best case time* is the minimum time needed to execute the algorithm, taken over all inputs of size n.

Barometer Instruction

Definition (Barometer Instruction)

The *barometer instruction* is the instruction that is executed the most number of times in an algorithm.

Example

```
/**
* Algorithm linearSearch
* Input: An int array, an int target
* Output: The first index the target is found, or -1 if it does not
     exist
*/
int linearSearch(int[] array, int target) {
    for(int i = 0; i < array.length; i++) {
        if(target == array[i]) { // target is found
            return i;
        }
    }
    return -1; // target not found
}
```

A good barometer instruction for this linear search function would be `target == array[i]`.

Example Continued

```
/**
* Algorithm linearSearch
* Input: An int array, an int target
* Output: The first index the target is found, or -1 if it does not
     exist
*/
int linearSearch(int[] array, int target) {
    for(int i = 0; i < array.length; i++) {
        if(target == array[i]) { // target is found
            return i;
        }
    }
    return -1; // target not found
}
```

The worst case occurs when the target is not in the array, or is the last element of the array. In this case there are n comparisons.

Example Continued

```
/**
* Algorithm linearSearch
* Input: An int array, an int target
* Output: The first index the target is found, or -1 if it does not
     exist
*/
int linearSearch(int[] array, int target) {
    for(int i = 0; i < array.length; i++) {
        if(target == array[i]) { // target is found
            return i;
        }
    }
    return -1; // target not found
}
```

The beset case occurs when the target is the first element of the array. In this case there is 1 comparisons.

Example Continued

The average case is more complex.

Is it the average over all possible inputs?

Is it the average over all inputs that have the target in the array?

Complexity Classes

We would like to categorise algorithms in such a way that we can compare the run time of two algorithms.

We wish to do this categorsation based on how fast the function, defined by how long the algorithm runs, grows.

Run Time Growth Example

Example

Assume we have an algorithm that has run time

$$t(n) = 60n^2 + 5n + 1.$$

n	$60n^2 + 5n + 1$	$60n^2$
10	6051	6000
100	600501	600000
1000	60005001	60000000
10000	60000050001	6000000000

Limits of Growth

$$\lim_{n \to \infty} \frac{60n^2 + 5n + 1}{60n^2} = 1$$

Asymptotically Equivalent

Definition (Asymptotically Equivalent)

Given functions $f(x)$ and $g(x)$, we define the binary relation $f(x) \sim g(x)$ as $x \to \infty$ if and only if

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = 1.$$

Asymptotic Upper Bound, $O(g(x))$

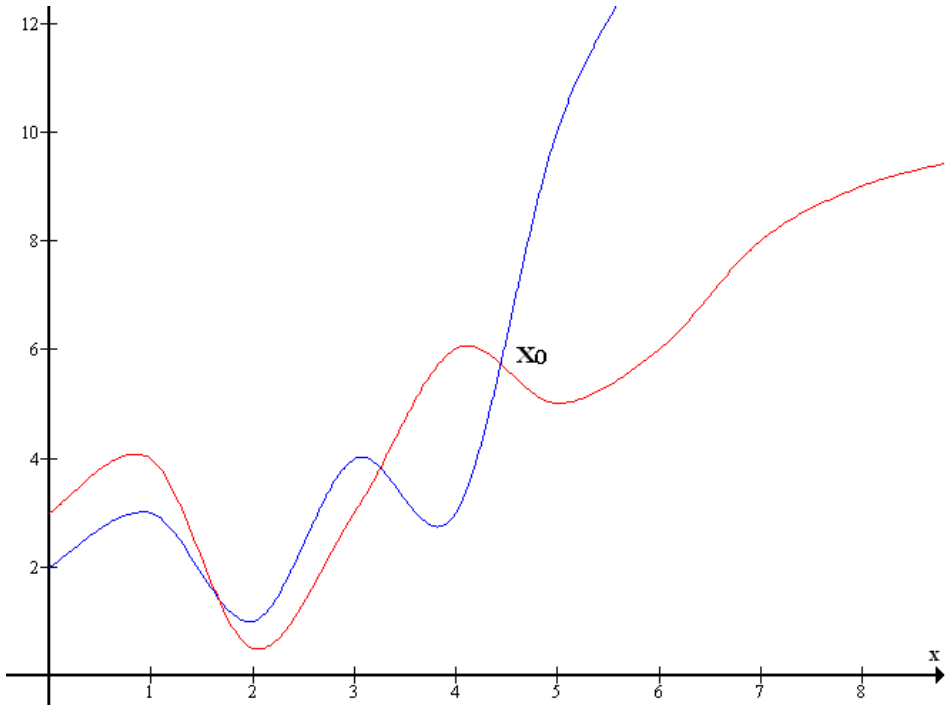
Definition (Asymptotic Upper Bound, $O(g(x))$)

Let $f(x)$ and $g(x)$ be eventually nonnegative functions on the positive integers. Then we write $f(x) = O(g(x))$ if there exists a positive real number M and a real number x_0 such that $|f(x)| \leq Mg(x)$ for all $x \geq x_0$.

Asymptotic Upper Bound, $O(g(x))$

Definition (Asymptotic Upper Bound, $O(g(x))$)

Let $f(x)$ and $g(x)$ be eventually nonnegative functions on the positive integers. Then we write $f(x) = O(g(x))$ if there exists a positive real number M and a real number x_0 such that $|f(x)| \leq Mg(x)$ for all $x \geq x_0$.



$O(g(x))$ Example

Example

Let's consider

$$t(n) = 60n^2 + 5n + 1.$$

We are after a function that is always bigger than it for all x larger than some x_0.

So breaking the function down a bit, we have the $5n$ component. $5n^2 \geq 5n$ for every $n \geq 1$.

And obviously $n^2 \geq 1$ for all $n \geq 1$.

Putting these bits together we get

$$60n^2 + 5n^2 + n^2 \geq 60n^2 + 5n + 1.$$

$O(g(x))$ Example

Example

So we can collect like terms to get

$$66n^2 \geq 60n^2 + 5n + 1.$$

We are trying to satisfy

$$|f(x)| \leq Mg(x)$$

so we can take our $M = 66$, $g(x) = x^2$, and we are done. We have $|60n^2 + 5n + 1| \leq 66n^2$. So $t(n) = O(n^2)$.

Asymptotic Lower Bound, $\Omega(g(x))$

Definition (Asymptotic Lower Bound, $\Omega(g(x))$)

Let $f(x)$ and $g(x)$ be eventually nonnegative functions on the positive integers. Then we write $f(x) = \Omega(g(x))$ if there exists a positive real number M and a real number x_0 such that $|f(x)| \geq Mg(x)$ for all $x \geq x_0$.

$\Omega(g(x))$ Example

Example

Taking the same function, $t(n) = 60n^2 + 5n + 1$ we will find a function such that $t(n) \geq Mg(x)$.

First, lets look at just $60n^2$. This is obviously smaller than $60n^2 + 5n$, because we have the same expression, and a bit more.
Similarly, $60n^2 + 5n \leq 60n^2 + 5n + 1$ because we've just added 1 to the right hand side.
By transitivity, we have that

$$60n^2 \leq 60n^2 + 5n + 1,$$

which is exactly what we need. So we have $|t(n)| \geq 60n^2$. So $t(n) = \Omega(n^2)$.

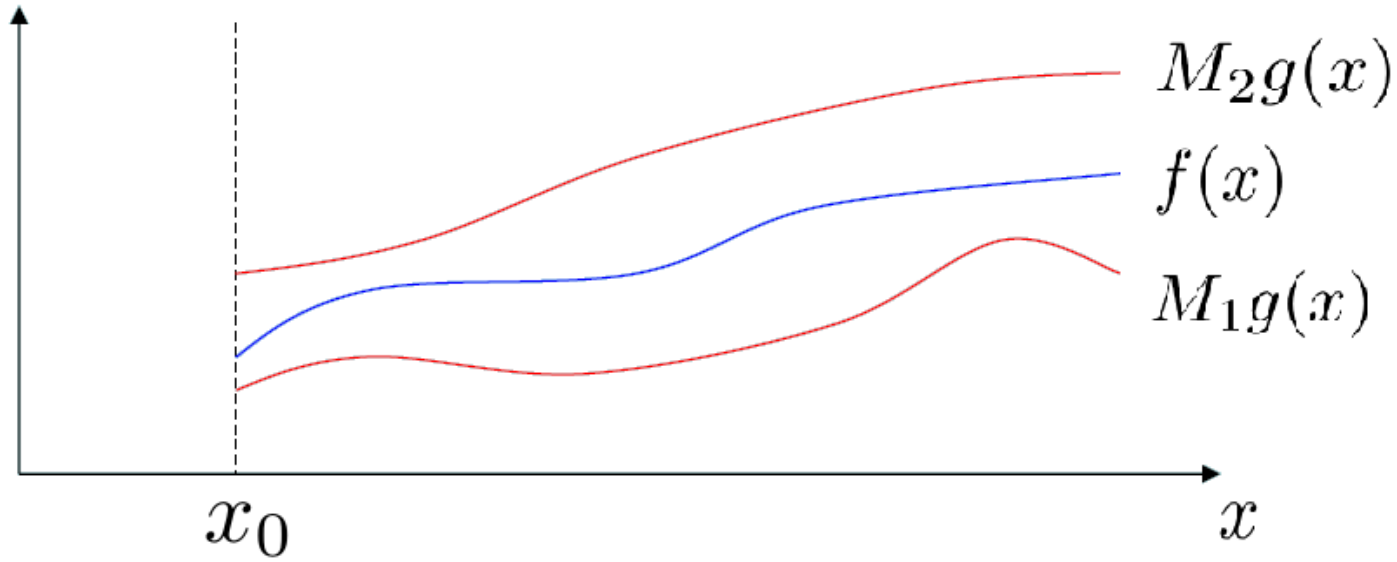
Asymptotic Tight Bound, $\Theta(g(x))$

Definition (Asymptotic Tight Bound, $\Theta(g(x))$)

Let $f(x)$ and $g(x)$ be eventually nonnegative functions on the positive integers. Then we write $f(x) = \Theta(x)$ if there exists positive real numbers M_1, M_2 and a real number x_0 such that $M_1 g(x) \leq f(x) \leq M_2 g(x)$ for all $x \geq x_0$.

Asymptotic Tight Bound, $\Theta(g(x))$

$\Theta(g(x))$ is the simplest, we only need to show that $t(n) = O(g(x))$ and $t(n) = \Omega(g(n))$ to show that $t(n) = \Theta(g(n))$.



$\Theta(g(x))$ Example

Example

Our function, $t(n) = 60n^2 + 5n + 1$, is $t(n) = O(n^2)$ and $t(n) = \Omega(n^2)$, so $t(n) = \Theta(n^2)$.

$O(g(x))$ Example 2

Example

$n = O(2^n)$ as 2^n is bigger than n for most values of n.

But there is no M we can find such that $n \geq M2^n$ for large n. As n increases, 2^n grows too quickly.

So $n \neq \Omega(2^n)$ and as a result $n \neq \Theta(2^n)$.

Properties of Big O

Theorem (Properties of Big O)

If $f(x) = O(g_1(x))$ *and* $h(x) = O(g_2(x))$

1. $f(x) + h(x) = O(\max\{g_1(x), g_2(x)\})$
2. $f(x)h(x) = O(g_1(x)g_2(x))$

and if $f(x) = O(g(x))$ *and* $g(x) = O(h(x))$ *then* $f(x) = O(h(x))$.

Classical Functions

Theorem (Classical Functions)

1. *Let* $p(x) = a_k x^k + a_{k-1} x^{k-1} + \cdots + a_1 x + a_0$ *be a nonnegative polynomial in* x *of degree* k. *Then* $p(x) = \Theta(n^k)$
2. $\log_b(x) = \Theta(\log_a(x))$
3. $\log(n!) = \Theta(n \log(n))$
4. $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log(n))$

Names of Common Growth Rates

$\Theta(g(x))$	Name
$\Theta(1)$	Constant
$\Theta(\log(\log(n)))$	Log log
$\Theta(\log(n))$	Log
$\Theta(n^c), 0 < c < 1$	Sublinear
$\Theta(n)$	Linear
$\Theta(n \log(n))$	n log n
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^k), k \geq 1$	Polynomial
$\Theta(c^n), c > 1$	Exponential
$\Theta(n!)$	Factorial

Feel for Growth Rates

n	n	$nlog(n)$	n^2	n^3	1.5^n	2^n	$n!$
10	$<$ 1sec	$<$ 1sec	$<$ 1sec	$<$ 1sec	$<$ 1sec	$<$ 1sec	4sec
30	$<$ 1sec	$<$ 1sec	$<$ 1sec	$<$ 1sec	$<$ 1sec	18min	10^{25}yrs
50	$<$ 1sec	$<$ 1sec	$<$ 1sec	$<$ 1sec	11min	36yrs	$> 10^{25}$yrs
100	$<$ 1sec	$<$ 1sec	$<$ 1sec	1sec	12892yrs	10^{17}yrs	$> 10^{25}$yrs
1000	$<$ 1sec	$<$ 1sec	1sec	18min	$> 10^{25}$yrs	$> 10^{25}$yrs	$> 10^{25}$yrs
10000	$<$ 1sec	$<$ 1sec	2min	12days	$> 10^{25}$yrs	$> 10^{25}$yrs	$> 10^{25}$yrs
100000	$<$ 1sec	2sec	3hrs	32yrs	$> 10^{25}$yrs	$> 10^{25}$yrs	$> 10^{25}$yrs
1000000	1sec	20sec	12days	31710yrs	$> 10^{25}$yrs	$> 10^{25}$yrs	$> 10^{25}$yrs

Basics of Recurrence Relations

Consider the following recurrence relation

$$c_n = n + c_{\lfloor \frac{n}{2} \rfloor}$$

with initial condition $c_1 = 0$.

Old Techniques

$$c_n = n + c_{\lfloor \frac{n}{2} \rfloor}$$

is not linear, or homogeneous.

We need to solve this by simply substituting in previous terms and hoping we get an equation we can work with.

Solving $c_n = n + c_{\lfloor \frac{n}{2} \rfloor}$

$$c_n = n + c_{\lfloor \frac{n}{2} \rfloor}$$

Let's consider just n of the form $n = 2^k$ so we don't have to worry about the division by 2. So we end up with

$$c_{2^k} = 2^k + c_{2^{k-1}}.$$

We can then expand out the $c_{2^{k-1}}$ term to get

$$c_{2^k} = 2^k + 2^{k-1} + c_{2^{k-2}}.$$

Continuing in this way gives...

Solving $c_n = n + c_{\lfloor \frac{n}{2} \rfloor}$

$$c_{2^k} = 2^k + 2^{k-1} + 2^{k-2} + \cdots + 2^1 + c_1.$$

Using our initial condition of $c_1 = 0$ gives

$$c_{2^k} = 2^k + 2^{k-1} + 2^{k-2} + \cdots + 2^1.$$

We can show with induction that $\sum_{i=0}^{m} i^k = 2^{m+1} - 1$, and we are missing the first term of that series, so we get $2^{k+1} - 2$. So we get

$$c_{2^k} = 2^{k+1} - 2.$$

Solving $c_n = n + c_{\lfloor \frac{n}{2} \rfloor}$

Now we have

$$c_{2^k} = 2^{k+1} - 2$$

which is in terms of k, and we can translate it back to be in terms of n by remembering that $n = 2^k$, to get

$$c_n = 2n - 2,$$

which, when using the table from earlier gives $c_n = \Theta(n)$.

Solving $c_n = n + c_{\lfloor \frac{n}{2} \rfloor}$

Keep in mind we have only shown this relationship holds for a particular n, those of the form $n = 2^k$. To get the rest we need to define *smoothness*.

Smooth Functions

Definition (Smooth Function)

A function f defined on positive integers is *smooth* if for any positive integer $a \geq 2$, there are positive constants C and N, depending on a, such that

$$f(an) \leq Cf(n)$$

and

$$f(n) \leq f(n+1)$$

for all $n \geq N$.

Smooth Functions

A more intuitive way to think about smooth functions is the following lemma.

Lemma

A function, f, is smooth if the following properties hold.

- *f is eventually non decreasing.*
- *For every integer $b \geq 2$, $f(bn) = O(f(n))$, that is, it's growth rate doesn't increase.*

Examples of Smooth Functions

Lemma (Smooth Functions)

Polynomial and logarithmic functions are smooth. Any function that is not $O(n^k)$ is not smooth.

Smoothing Lemma

Lemma (Smoothing Lemma)

If t is an eventually non decreasing function, f is a smooth function, and $t(n) = \Theta(f(n))$ for all n a power of b, then $t(n) = \Theta(f(n))$.

Solving $c_n = n + c_{\lfloor \frac{n}{2} \rfloor}$

We can finish solving $c_n = n + c_{\lfloor \frac{n}{2} \rfloor}$ now.

We have shown that $c_n = \Theta(n)$ when $n = 2^k$. So in this case $f(n) = n$, which is obviously non decreasing.

We also have that $f(an) = O(n)$ for any a we pick.

It's not too much of a stretch to take for granted that c_n is eventually non decreasing, and can formally be shown with induction.

So we can conclude that $c_n = \Theta(n)$ for all values of n, not just values of the form $n = 2^k$, by the smoothing lemma.

What does $c_n = n + c_{\lfloor \frac{n}{2} \rfloor}$ Represent?

$$c_n = n + c_{\lfloor \frac{n}{2} \rfloor}$$

On the right hand side, the n component represents some kind of overhead in splitting a problem into sub problems.

The $c_{\lfloor \frac{n}{2} \rfloor}$ component represents the time required to do a sub problem of size $\frac{n}{2}$.

Master Theorem 1

Theorem (Master Theorem/Main Recurrence Theorem 1)

If

$$T_n \leq aT_{\lfloor \frac{n}{b} \rfloor} + f(n)$$

or

$$T_n \leq aT_{\lceil \frac{n}{b} \rceil} + f(n)$$

and $f(n) = O\left(n^k\right)$ *then*

$$T_n = \begin{cases} O\left(n^k\right) & \text{if } a < b^k \\ O\left(n^k \log(n)\right) & \text{if } a = b^k \\ O\left(n^{\log_b(a)}\right) & \text{if } a > b^k \end{cases}$$

Master Theorem 2

Theorem (Master Theorem/Main Recurrence Theorem 2)

If

$$T_n \geq aT_{\lfloor \frac{n}{b} \rfloor} + f(n)$$

or

$$T_n \geq aT_{\lceil \frac{n}{b} \rceil} + f(n)$$

and $f(n) = \Omega\left(n^k\right)$ *then*

$$T_n = \begin{cases} \Omega\left(n^k\right) & \text{if } a < b^k \\ \Omega\left(n^k \log(n)\right) & \text{if } a = b^k \\ \Omega\left(n^{\log_b(a)}\right) & \text{if } a > b^k \end{cases}$$

Master Theorem 3

Theorem (Master Theorem/Main Recurrence Theorem 3)

If

$$T_n = aT_{\lfloor \frac{n}{b} \rfloor} + f(n)$$

or

$$T_n = aT_{\lceil \frac{n}{b} \rceil} + f(n)$$

and $f(n) = \Theta\left(n^k\right)$ *then*

$$T_n = \begin{cases} \Theta\left(n^k\right) & \text{if } a < b^k \\ \Theta\left(n^k \log(n)\right) & \text{if } a = b^k \\ \Theta\left(n^{\log_b(a)}\right) & \text{if } a > b^k \end{cases}$$

Example 1 of Master Theorem

Example

We have

$$c_n = n + c_{\lfloor \frac{n}{2} \rfloor}$$

and $c_1 = 0$. So in this case $a = 1$, $b = 2$, $f(n) = n$ so $f(n) = \Theta(n)$, and so $k = 1$. Then we have $1 < 2^1 \Rightarrow a < b^k$, so we are in the first case. This tells us that

$$c_n = \Theta(n^k) = \Theta(n),$$

which is what we concluded earlier.

Example 2 of Master Theorem

Example

Take

$$c_n = n + 2c_{\lceil \frac{n}{2} \rceil}$$

and $c_1 = 0$. We have $a = 2$, $b = 2$, $f(n) = n$, so $f(n) = \Theta(n)$, so $k = 1$. We then have $2 = 2^1 \Rightarrow a = b^k$, so $c_n = \Theta(n \log(n))$.

Example 3 of Master Theorem

Example

Take

$$c_n = n + 7c_{\lfloor \frac{n}{4} \rfloor}.$$

We have $a = 7$, $b = 4$, $f(n) = n$, so $f(n) = \Theta(n)$, so $k = 1$. We then have $7 > 4^1 \Rightarrow a > b^k$, so we are in the third case. So $c_n = \Theta\left(n^{\log_4(7)}\right)$.