Imperial College London

Discrete Mathematics: Year 2

Big-O and Complexity

Xin Wang

November 15, 2020

Abstract

Discrete maths studies mathematical concepts using graph theory i.e. a set of points where each point is connected to a specific set of other points by edges. It is different from continuous as values can only take certain values e.g. set of numbers. This is a significant area of study as all computers operate using discrete maths.

Algorithms are at the heart of computing and is a significant area in EIE. The design and analysis of algorithms are essential skills to an engineer and there are several established Design and Analysis techniques. An algorithm is simply a finite set of well-defined instructions, written to solve a specific problem.

Contents

1 Introduction

Algorithm: A sequence of computational steps that takes a value as **input** and produces one or a set of values as **output**.

Instance (of a problem): Consists of the input (satisfying any program constraints) required to compute a solution to the problem.

Algorithms have many practical applications in fields such as biology and computing e.g. sorting algorithms which are common since any computational programs will often have sorting elements in it. The data produced by algorithms are then stored in data structures for review or further processing.

Data structures: A way to store and organize data in order to facilitate access and modifications. There are various types and each have their own advantages and disadvantages.

Algorithms are divided into two major sections: **Design** and **Analysis**.

1.1 Analysis of algorithms

Analysis (of algorithms): Predicting the resources that the algorithm require such as time, memory, communication bandwidth and computer hardware.

The main concept concerned in algorithm analysis is **complexity notation**. It is used to compute the time an algorithm takes and prove an algorithm works.

In general, the time taken grows with the size of the input thus it is common to describe the program running time as **a function of the size of its input**.

Running time (of an algorithm): The number of "steps" i.e. primitive operations executed.

There are two types of running time: **Worse case running time** and **Average case running time**.

Usually, the worst case running time is used because:

- It gives the upper bound on the running time for any input and it occurs often.
- The average case is the same function as the worst case.

1.2 Design of algorithms

> **Design** (of algorithms): A mathematical process to approach problem-solving.

Algorithms are fundamentally a series of steps to solve a given problem. There are numerous types of algorithm design techniques and each technique differs in complexity with certain advantages in different scenarios.

Given a problem to solve, the following questions often determine the technique used:

1. How do we design an algorithm that solves the problem?
2. How long will the algorithm take?
3. How much memory will the algorithm use?
4. Is it possible to do better?

The design techniques to be covered are:

1. Divide and conquer
2. Dynamic programming
3. Greedy algorithms

Over the course of this course, graph algorithms are covered including general purpose graph design methods:

1. Breadth first
2. Depth first

2 The Fibonacci Series

The Fibonacci series is used to show the importance of analysis and design of algorithms. The Fibonacci series is originally used to model the growth of rabbits e.g. $F(1) = 1$, $F(2) = 1$, $F(3) = 3$, $F(4) = 5$.

Fibonacci series $F(n)$ is defined by:

$$F(n) = F(n-1) + F(n-2)$$

There are several different ways to compute Fibonacci series, each with various efficiency.

2.1 Recursion

```
def F_recurse(n):
if n<=2:
    return 1
else:
    return F_recurse(n-1) + F_recurse(n-2)
```

This algorithm is very simple but very inefficient since this algorithm repeatedly has to compute the same values again and again i.e. $F(4)$ computes $F(3)$ and $F(2)$ in which $F(3)$ computes $F(2)$ and $F(1)$ again:

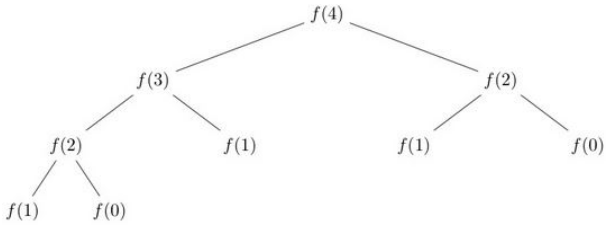


Figure 1: Recursion tree of Fibonacci series

In reality, calculating any value in the Fibonacci series only requires the previous two values e.g. $F(4)$ only requires $F(3)$ and $F(2)$. So a more efficient algorithm would store the last two values to calculate the next value in the Fibonacci series.

2.2 Pen and paper algorithm

```
def F_pnp(n):
f = [1]*(n+1) # Create the needed space
f[1] = f[2] = 1 # Two initial values are 1

for i in range(3, n+1):
    f[i] = f[i-1] + f[i-2]
return f[n]
```

As mentioned previous, **only the two most recent values** are used to calculate the next value in the Fibonacci series. This results in higher high memory usage that increases exponentially to n.

2.3 Iterative algorithm

```
def F_iter(n):
a = b = 1 # 1st and 2nd value of series are both 1

for i in range (3,n+1): # Starting from 3rd
    a,b = b, a+b # Two most recent values
return b
```

Only the two most recent values are used, only a fixed amount of RAM is used for any n. This is the most efficient code for calculating the Fibonacci sequence.

2.4 Binet's formula

$$F(n) = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}} \quad \text{where } \phi = \frac{1+\sqrt{5}}{2}$$

With the general formula, time complexity becomes $\Theta(1)$ i.e. constant **but memory complexity is not** $\Theta(1)$. This is because for each larger n, higher precision would be required to represent the Fibonacci numbers.

2.5 Matrix algorithm

Theoretically, the matrix method is the best solution:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{bmatrix}$$

This is due to the special characteristic of matrix multiplication. For example M^8 would not actually need to multiply 8 times, instead 3 times would be enough:

$$M^8 = ((M^2)^2)^2$$

This method results in a time complexity of $\Theta(\log(n))$ and a memory complexity of $\Theta(1)$. But, in practice, it is not more efficient than the iterative method.

This is due to the memory usage affective performance. **Context is important**.

3 Growth of function

3.1 Asymptotic notation

Asymptotic notation are mathematical tools to represent the time complexity of algorithms for asymptotic analysis when the input tends towards a particular value or a limiting value.

Performance analysis is important since that it affects other critical aspects such as the user experience and program modularity. When studying two given algorithms, one might be more efficient for certain input ranges and not as efficient for other input ranges. Asymptotic analysis shows these characteristics by evaluating the performance of an algorithm **in terms of varying input size**.

> **Asymptotic analysis** (of an algorithm): Evaluates how much resources i.e. time or memory space is occupied by an algorithm in terms of input size and not the actual running time.

Asymptotic notations are defined **in terms of functions**, including algorithms and programs, whose domain is the set of natural numbers

$$\mathbb{N} = 0, 1, 2, \ldots$$

It is important to realise that asymptotic notation can be applied, not only to running time of a algorithm, but also to functions that characterise other aspects of algorithms such as memory usage. Therefore, it is critical to understand which aspect the asymptotic notation is applied to. Usually, asymptotic notations try to universally characterise running time no matter the type of input - a blanket statement that covers all inputs including worst case running times.

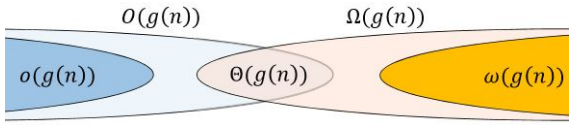


Figure 2: Areas of coverage of the different domains of notations

3.1.1 Theta (Θ) notation

Theta notation is **asymptotically tight bound**, it means the function lies in the upper and the lower bound. Graphically, it encloses the function from above and below. It is commonly used for analysing the average case complexity of an algorithm by using the worst-case time and best-case time.

For a given function $f(n)$, a function is considered equal to $\Theta(g(n))$ if there are **positive** constants c_1, c_2 and n_0 such that:

$$0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n) \text{ where: } n \geq n_0$$

In words, function $f(n)$ is considered to belong to the set $\Theta(g(n))$ if there are positively defined constants c_1 and c_2, such that from the point of value n_0 towards the right, the function $f(n)$ is "sandwiched" between $c_1\, g(n)$ and $c_2\, g(n)$ for a sufficiently large range n. In other words, $f(n)$ is **approximately proportional** to $g(n)$.

Alternatively, Θ-notation can be defined as:

$$0 < c_1 \leq \frac{f(n)}{g(n)} \leq c_2$$

A simple way to get the Θ-notation of an expression is to drop low order terms and ignore leading constants. The reason it is possible is because, remember, asymptotic complexity is not concerned with comparing performance of different algorithms. It is for understanding **how performance of individual algorithms scales with respect to the input size**.

In summary:

- Can ignore constant factors e.g. 3 in $3n^2$ and Theta notation is $\Theta(n^2)$
- Only note the asymptotic behavior i.e. $100n + 0.1n^2$ is $\Theta(n^2)$ since for large enough n the size of the equation will be dominated by $0.1n^2$

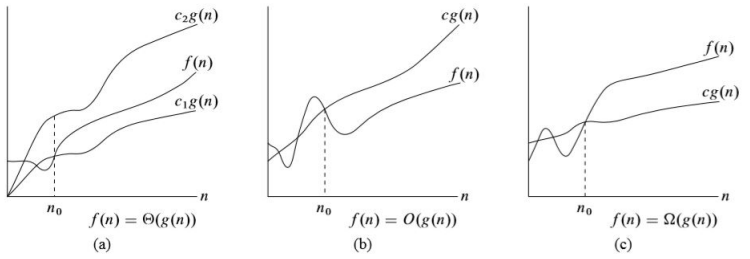


Figure 3: Graphical examples of Θ-notation, O-notation and Ω-notation respectively.

The Theta notation $\Theta(g(n))$ **is mathematically treated as a set** so it is correct to say $f(n)$ **is a member of** $\Theta(g(n))$ $[f(n) \in \Theta(g(n))]$ where $g(n)$ are functions. However, usually it is written $f(n) = \Theta(g(n))$ which has certain advantages but should be very carefully used.

Asymptotically tight bound: For all values of n at and to the right of n_0, the value of $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$. In other words, for all $n \geq n_0$, the function of $f(n)$ is equal to $g(n)$ to within a constant factor.

Asymptotically nonnegative: $f(n)$ is nonnegative whenever n is sufficiently large.

Tight upper bound: An upper bound where there are no smaller value that is a better upper bound.

Tight lower bound: A lower bound is the greatest lower bound, or an infimum, if no greater value is a better lower bound.

The definition of Θ-notation requires that every member of $f(n) \in \Theta(g(n))$ be **asymptotically nonnegative**. Among all the notations, Θ notation gives the best details about the rate of growth of function because it gives a **tight bound** unlike O and Ω notations which only gives the upper and lower bounds respectively. Due to the definition of Theta:

$$f(n) \textbf{ is } \Theta(g(n)) \textbf{ if and only if it is equal to } O(g(n)) \textbf{ and } \Omega(g(n))$$

Example 1: Proof $f(x) = n^2 - n = \Theta(n^2)$

Process:

1. List $g(n)$:

$$g(n) = n^2 \text{ and } f(n) = n^2 - n$$

2. Express as $\frac{f(n)}{g(n)}$:

$$\frac{f(n)}{g(n)} = \frac{n^2 - n}{n^2} = 1 - \frac{1}{n}$$

3. Try $n_0 = 2$:

$$0 < c_1 = \frac{1}{2} \leq 1 - \frac{1}{n} \leq 1 = c_2$$

3.1.2 Big-O (O) notation

The Big-O notation is used when only the **asymptotic upper bound** is concerned. The Big-O gives an upper bound on a function to within a constant factor - for all values n at and to the right of n_0, the function is **on or below** $c * g(n)$.

For a given function $f(n)$, a function is $O(g(n))$ if there are positive constants c_0 and n_0 such that:

$$0 \leq f(n) \leq c * g(n) \quad \text{where: } n \geq n_0$$

or

$$0 \leq \frac{f(n)}{g(n)} \leq c \quad \text{where: } n \geq n_0$$

Meaning: "Program takes at most $g(n)$ steps to run"

Like Θ-notation, $f(n) = O(g(n))$ indicates that a function $f(n)$ is a member of the set $O(g(n))$. **Note $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$ since Θ-notation is a stronger notion that O-notation**.

Example 1: $n = O(n^2)$

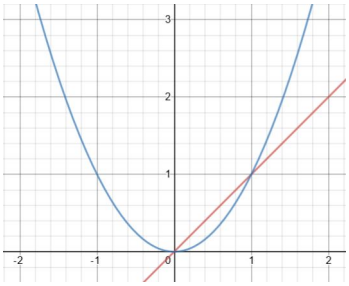


Figure 4: Graphical representation of O-notation: Red - n and Blue - n^2

Example 2: Given $f(n) = \begin{cases} 0 & n \text{ even} \\ n & n \text{ odd} \end{cases}$, prove $f(n) = O(g(n))$

Process:

1. List $g(n)$:

$$g(n) = n$$

2. Express as $\frac{f(n)}{g(n)}$:

$$\frac{f(n)}{g(n)} = \begin{cases} 0 & n \text{ even} \\ 1 & n \text{ odd} \end{cases}$$

3. State the bounds:

$$0 \leq \frac{f(n)}{g(n)} \leq 1 = c$$

3.1.3 Big-Omega (Ω) notation

The Big-Omega notation is used when the **asymptotic lower bound** is concerned. The Big-Omega gives a lower bound on a function to within a constant factor - for all values n at and to the right of n_0, the function is **on or above** $c * g(n)$.

For a given function $f(n)$, a function is $\Omega(g(n))$ if there are positive constants c_0 and n_0 such that:

$$0 < c * g(n) \leq f(n) \quad \text{where: } n \geq n_0$$

or

$$0 < c \leq \frac{f(n)}{g(n)} \quad \text{where: } n \geq n_0$$

Meaning: "Program takes at least $g(n)$ steps to run"

Note $f(n) = \Theta(g(n))$ implies $f(n) = \Omega(g(n))$ since Θ-notation is a stronger notion that Ω-notation.

Example 1: $2^n = \Omega(n)$

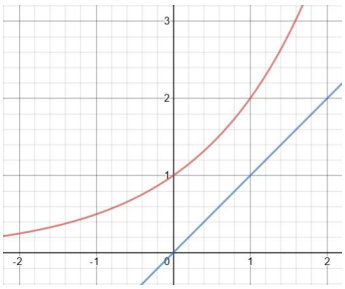


Figure 5: Graphical representation of Θ-notation: Red - 2^n and Blue - n

3.1.4 Little-o o notation

Little-o means **loose upper-bound** of $f(n)$ i.e. a rough estimate of the maximum order of growth whereas Big-O may be the actual order of growth. In other words, the asymptotic upper bound provided by O-notation may/may-not be **asymptotically tight** i.e. $2n^2 = O(n^2)$ is asymptotically tight but not $2n = O(n^2)$ since there will be some leeway. The little-o notation denotes a higher bound that is **not asymptotically tight**.

Asymptotically tight: A bound has the the tightest possible bounds, meaning it satisfies the conditions of Θ-notation where it requires the tightest bounds.

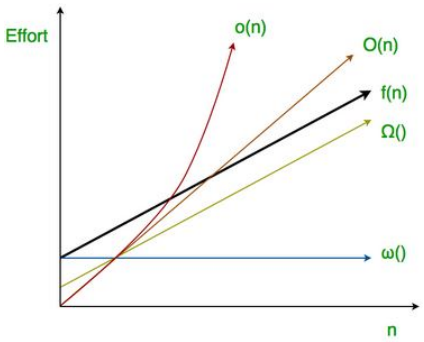


Figure 6: A graphical representation of the relationships of the four notations to $f(n)$

For a given function $f(n)$, a function is $o(g(n))$ if for any constant $c > 0$ there is a constant n_0 such that:

$$0 \leq f(n) < c * g(n) \quad \text{where: } n \geq n_0$$

or

$$0 \leq \frac{f(n)}{g(n)} < c \quad \text{where: } n \geq n_0$$

The main difference between Big-O and Little-O is: Little-O:

- Big-O: Bound $0 \leq f(n) \leq c * g(n)$ holds for **some** constants $c > 0$
- Little-O: Bound $0 \leq f(n) < c * g(n)$ holds for **all** constants $c > 0$

Due to the definition of the little-o notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity (Refer to Figure 4 above).

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Example 1: If $f(n) = n^2$ and $g(n) = n^3$ then check whether $f(n) = o(g(n))$ or not.

$$\begin{aligned}\lim_{n\to\infty} \frac{n^2}{n^3} &= \lim_{n\to\infty} \frac{1}{n} \\ &= \frac{1}{\infty} \\ &= 0\end{aligned}$$

The result is 0 thus it satisfies the equation mentioned above.

3.1.5 Little-omega ω notation

Just like the little-o and Big-O, little-omega means **loose lower-bound** of $f(n)$ i.e. a rough estimate of the minimum order of growth whereas Big-Omega may/may not be **asymptotically tight**. The little-omega notation denotes a lower bound that is **not asymptotically tight**.

For a given function $f(n)$, a function is $\omega(g(n))$ if for any positive constant $c > 0$ there is an n_0 such that:

$$0 \leq c * g(n) < f(n) \text{ where: } n \geq n_0$$

or

$$0 \leq c < \frac{f(n)}{g(n)} \text{ where: } n \geq n_0$$

The main difference between Big-Omega and Little-Omega is that:

- Big-Omega: Bound $0 < c * g(n) \leq f(n)$ holds for **some** constant $c > 0$
- Little-Omega: Bound $0 < c * g(n) < f(n)$ holds for **all** constant $c > 0$

Due to the definition of the little-omega notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

3.2 Complexity calculations

1. **Polynomials**: Take only the highest power:

$$O(A_P n^p + A_{p-1} n^{p-1} + \cdots + A_1 n + A_0) = O(n^p)$$

2. **Multiplication**:

   - Multiplying with constant k: $f(n) = (g(n))$

$$kf(n) = O(g(n))$$

   - Multiplying with another function $f_2(n) = O(g_2(n))$: $f_1(n) = O(g_1(n))$

$$f_1(n) * f_2(n) = O(g_1(n) * g_2(n))$$

3. **Addition**: Take only the largest term:

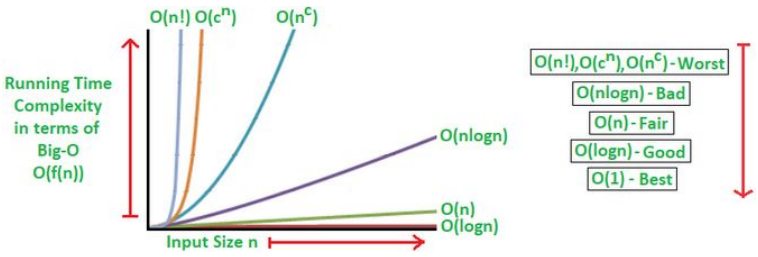
   Given $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ where $g_1(n) = O(g_2(n))$ i.e. $g_2(n)$ is larger than $g_1(n)$:

$$f_1(n) + f_2(n) = O(g_2(n))$$

3.3 Common Big-O Examples

Time complexity scale:

$$Constants < Logarithms < Polynomials < Exponentials$$



Example 1: $O(1)$

Time complexity is 1 - always just one operation. It is **constant**. $O(1)$ is the best possible time complexity as it means constant time operations.

$$f(n) = O(1) = O(g(n)) \text{ where: } g(n) = 1$$

where from the definition of Big-O: $c > 0$ and n_0 such that $n \geq n_0$

$$\frac{f(n)}{g(n)} \leq c \text{ where: } g(n) = 1$$
$$f(n) \leq c$$

Thus showing constant c is independent of $f(n)$

Common example: A program removing one letter from a list.

$$[a, b, c, d, e] \rightarrow [a, b, c, d]$$

Example 2: $O(n)$

Linear - runtime is *Order* n since number of operations required is **proportional** to n to be computed. The complexity of the operation does not change, runtime is solely dependent on the size of n.

Common example: A program that duplicates each letter in a list.

$$[a, b, c, d, e] \rightarrow [aa, bb, cc, dd, ee]$$

Example 3: $O(n^c)$ where c is a constant

Polynomial time complexity. Usually not acceptable measure - the number of operations required is n^c. n^2 is called quadratic time complexity e.g. a list with 3 variables will require 9 operations.

Common example: A program to add every member of list to each other.

$$[a, b, c, d, e] \rightarrow [abcde, bacde, cabde, dabce, eabcd]$$

Example 4: $O(c^n)$ where c is a constant

Exponential time complexity. The worse possible complexity.

Common example: Brute forcing a password by trying every possible combination.

Example 5: $O(log(n))$

Logarithmic time complexity - the opposite of exponential time complexity. As the size of n increases, the smaller proportion the number of operations will be required - it grows very slowly i.e. if n doubles the $\log(n)$ increases by a small constant factor. Can be thought of as **not much bigger than 1**.

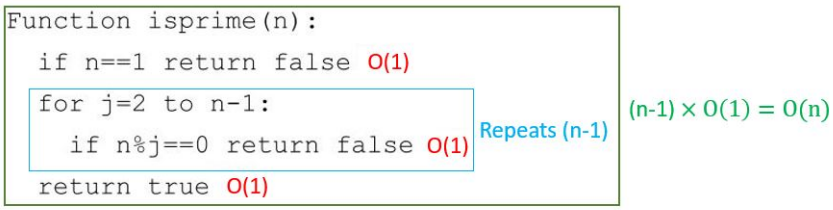
Common example: Looking people up in a phone book - not every person needs to be checked, able to use divide-and-conquer to narrow the search alphabetically.

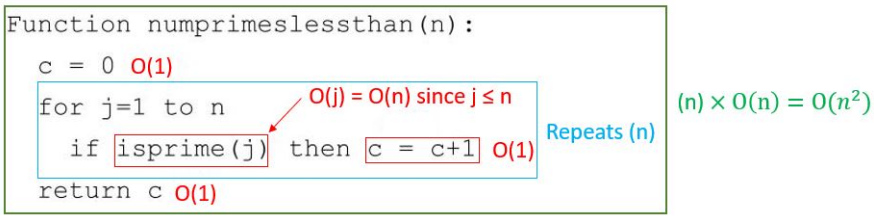
3.4 Example: `isprime` and `numprimelessthan`

Given the following program:

```
1  Function isprime(n):
2      if n==1 return false
3      for j=2 to n-1:
4          if n%j==0 return false
5      return true
6
7  Function numprimeslessthan(n):
8      c = 0
9      for j=1 to n:
10         if isprime(j) then c = c+1
11     return c
```

Define $f(n)$ = amount of time to run program with input n i.e. $f(n) = O(?)$





There the time required to run the program is $f(n) = O(n^2)$

The next step is to check if the program speed can be improved. The program is mathematically defined: If $f(n) = O(n^2)$ then

$$\sum_{j=1}^{n} f(j) = O(n^2)$$

There can be no theoretical faster way to calculate it:

Express:

$$\sum_{j} f(j) = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} = O(n^2)$$

4 Divide-and-Conquer

Divide-and-Conquer is a strategy for designing algorithms by solving a problem recursively - applying three steps at each level of recursion.

1. **Divide**: Divide problem into a number of sub-problems (smaller instances of the same problem).
2. **Conquer**: Solve sub-problems by solving recursively.
3. **Combine**: Combine solutions to the sub-problems into the solutions for the original problem.

> **Recursive case**: Sub-problems that are large enough to solve recursively.
>
> **Base case**: Sub-problems that are small enough that cannot be used with recursion anymore.

Sometimes dividing a problem into sub-problems can save time.

Example 1: Given an algorithm defined by $O(n^2)$ where $n = 128$ compare.

- Normal head on approach: $n = 128$ so will take $128^2 = \underline{16384}$ steps.
- Divided into two:
  - $n = 64$ so will take $64^2 = 4096$ steps for each half.
  - 8192 for each half. 128 steps to combine solutions.
  - Total steps required: $\underline{8320}$

4.1 Recurrence

> **Recurrence**: An equation/inequality that describes a function in terms of its value on smaller inputs.

Recurrences are closely connected with the Divide-and-Conquer paradigm since it is a natural way to characterize the running times of Divide-and-Conquer algorithms.

Following **Example 1**: The recurrence can be found by finding the gain for a general n.

- 0 divide: n^2
- 1 divide: $2\left(\frac{n}{2}\right)^2 + n = \frac{n^2}{2} + n$

- 2 divide: $4\left(\frac{n}{4}\right)^2 + n + 2\left(\frac{n}{2}\right) = \frac{n^2}{4} + 2n$

  ...

- General divide: $2^k\left(\frac{n}{2^k}\right)^2 + n + \cdots + 2^k\left(\frac{n}{2^k}\right) = \frac{n^2}{2^k} + nk$

Taking the max where $\frac{n}{2^k} = 1$ thus $k = log_2 n$: the recurrence is:

$$T(n) = O(n\ log_2 n)$$

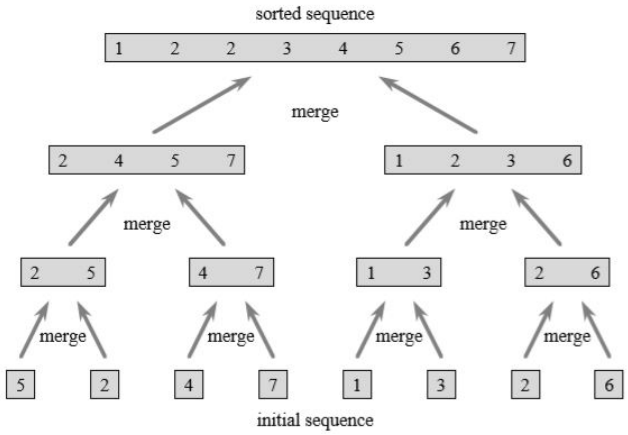


Figure 7

Recurrences can take many forms and some divide sub-problems into unequal sizes. Sub-problems are not constrained to being a constant fraction of the original problem. There are three methods for solving recurrences (obtaining the asymptotic O bounds of the solution):

- **Substitution method**: Guess a bound then use mathematical induction to prove the guess was correct.
- **Recursion-tree method**: Convert recurrence into tree whose nodes represent costs incurred at that level of recursion. Techniques for bounding summations are used to solve recurrence.
- **Master method**: Provide bounds in the form: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ where $a \geq 1$, $b > 1$ and $f(n)$ is a given function.

4.1.1 Master method

The master method provides a general "cookbook" approach to solving recurrences in the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

or

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

where $a \geq 1$, $b > 1$ and $d \geq 0$ are constants with $f(n)$ begin an asymptotically positive function.

The equation describes several points on the running time of an algorithm:

- A problem of size n divided into a sub-problems, each of size $\frac{n}{b}$ where a and b are positive constants.
- a sub-problems are solved recursively, each with time $T\left(\frac{n}{b}\right)$.
- Function $f(n)$ encompasses the cost of dividing the problem and combining the results of the sub-problems.

The Master Method is based on the **Master Theorem** which has three cases, each with its own conditions.

$$T(n) = \begin{cases} O(n^d) & \text{if } d > log_b a \\ O(n^d log[n]) & \text{if } d = log_b a \\ O(n^{log_b a}) & \text{if } d < log_b a \end{cases}$$

Example 1: Consider $T(n) = 9T\left(\frac{n}{3}\right) + n$.

1. Note $a = 9$, $b = 3$ and $f(n) = n$ $(d = 1)$.
2. Finding $log_b a$ which is 2, it meets Case 3.
3. $T(n) = O(n^{log_b a}) = O(n^2)$

Example 2: Consider $T(n) = T\left(\frac{2n}{3}\right) + 1$.

1. Note $a = 1$, $b = \frac{3}{2}$ and $f(n) = 1$ $(d = 0)$.
2. Finding $log_b a$ which is 0, it meets Case 2.
3. $T(n) = O(n^d log[n]) = O(log[n])$

4.2 Proof of correctness: Loop invariants

A loop invariant is a property of a program loop that is true before and after each iteration. It is a logical assertion, sometimes checked within the code by an assertion call. Knowing its invariants is essential in understanding the effect of a loop and why the loop is correct.

To prove a loop is correct, the following aspects of the loop variant must be shown:

- **Initialisation**: It is true prior to the first iteration of the loop
- **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination**: When the loop terminates, the invariant gives a useful property that helps show that the algorithm is correct i.e. the array is sorted.

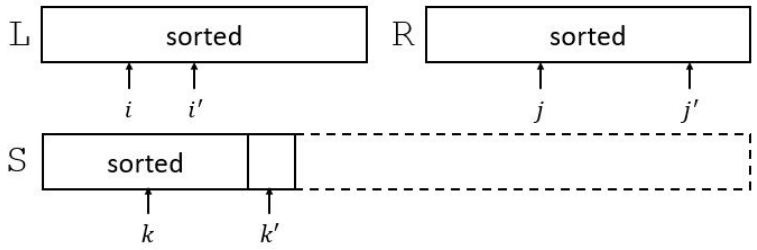
Note the similarity to **mathematical induction**, where to prove that a property holds, a **base case** and an **inductive step** is proven. Likewise, showing that the invariant holds before the first iteration corresponds to the base case, and showing that the invariant holds from iteration to iteration corresponds to the inductive step.

The third property **Termination** is the most important one, since the loop invariant is used to show the correctness. The termination property differs from mathematical induction, where the inductive step is applied infinitely but loop invariant has a point when the loop terminates.

The **merge-sort** is used as a working example. Recall how a merge-sort works, assume two sorted arrays, L and R and produce a sorted array that has all the elements in both L and R. With a pointer to the first element of each array, the first element of the final array will either be the **lowest one between**:

- The first element of L
- The first element of R

Increment the corresponding pointer so it points to the second element in that array instead of the first and the process repeats.




```
Merge(L, R):
    m = length(L) + length(R)
    S = empty array of size m
    i = 1; j = 1
    for k' = 1 to m:
        if L[i] <= R[j]:
            S[k'] = L[i]
            i = i + 1
        else: (L[i] > R[j])
            S[k'] = R[j]
            j = j + 1
    return S
```

Loop invariant: At the start of each iteration k of the for loop:

- The nonempty part of S contains the $k-1$ smallest elements of L and R, **in sorted order**. In other words, given the two S array indexes k_1 and k_2 that satisfies the condtion the k_1 is smaller than k_2 and smaller than the array size, the value at k_1 will be smaller or equal than the value at k_2.

$$\forall k_1, k_2 \text{ with } 0 \leq k_1 < k_2 < |S| : \ S[k_1] \leq S[k_2] \tag{1}$$

- $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied to S. In other words, all items in S-(k)-are **smaller than all remaining items** in arrays L-(i')-and R-(j').
  - Array L:

$$\forall k, i' \text{ with } 0 \leq k < |S|, \ i \leq i' < |L| : \ S[k] \leq S[i'] \tag{2}$$

  - Array R

$$\forall k, j' \text{ with } 0 \leq k < |S|, \ j \leq j' < |L| : \ S[k] \leq S[j'] \tag{3}$$

Proving the loop invariant:

- **Initialisation**: The loop invariant holds prior to the first iteration of the loop.
  - $i = j = 1$, and S is completely empty thus satisfying equation (1).
  - $L[1]$ is the smallest element of L thus satisfying equation (2).
  - $R[1]$ is the smallest element of R thus satisfying equation (3).
- **Maintenance**: Suppose without loss of generality that $L[i] \leq R[j]$ and $L[i]$ is the smallest element not yet copied to S.
  - The current nonempty part of S consists of the $k'-1$ smallest element.
  - After one loop, $L[i]$ is copied to S, the nonempty part (left) of S will consist of the k' smallest elements.
  - Incrementing k' (in the for loop update) and i basically reestablishes the loop invariant for the next iteration.
- **Termination**: By the loop invariant, the array S up to the point k' will contain the smallest elements of L and R meaning $S[k'] = \min\{L[i], R[j]\}$ in sorted order thus:

$$S[k'] \leq L[i] \text{ and } S[k'] \leq R[j]$$

4.3 Theoretically optimal

Sorting algorithms encountered so far all share an interesting property: the sorted order they determine is based only on comparisons between the input elements i.e. sorting algorithms comparison sorts.

It can be prove that any comparison sort must make $n \log(n)$ comparisons in the worst case to sort n elements, if there are no assumptions on the values of the elements. Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.

The process of sorting an array requires the use of a **decision tree**. This is used to prove the worst-case performance.

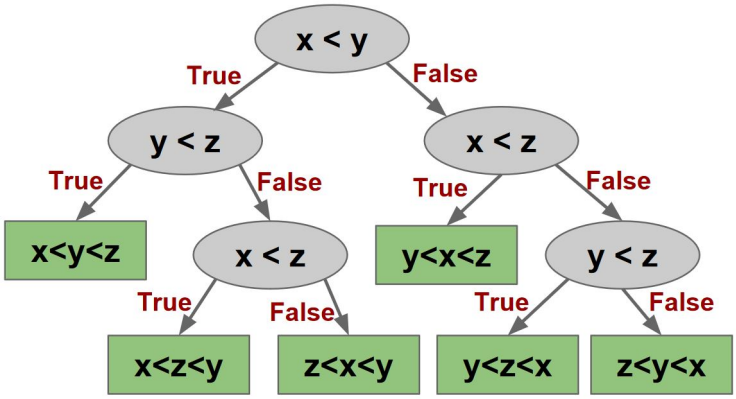


Figure 8: The decision tree for sorting $\{x, y, z\}$

Each leaf node corresponds to a possible sorted order of $\{x, y, z\}$, a decision tree needs to contain all possible orders i.e. $n!$ possible orders for n elements.

The second variable required is **height**. A binary tree with height h has at most 2^h leaves.

Thus:

$$L \geq n!$$
$$2^h \geq n!$$

Make h the subject:

$$h \geq \log(n!) \in \Omega(n \log(n))$$

When the lower bound and the upper bound is the same, a **theoretical optimal** is obtained.

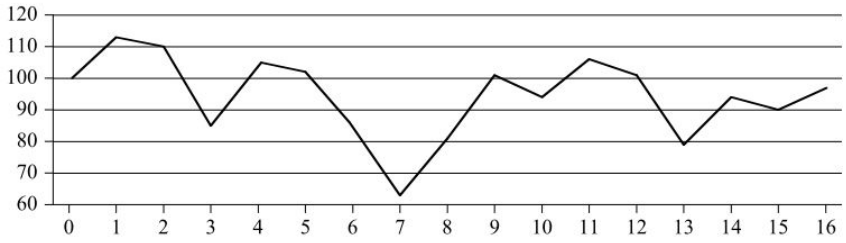
4.4 Maximum subarray problem

The maximum sum subarray problem is the task of finding a contiguous subarray with the largest sum, within a given one-dimensional array $A[1 \dots n]$ of numbers. In other words, the task is to find indices i and j with $1 \leq i \leq j \leq n$, such that the sum

$$\sum_{x=i}^{j} A[x]$$

is as large as possible. Each number in the input array A could be positive, negative, or zero.

The most famous of this kind of problem is the problem of trying to maximise stock-trading return i.e. buy low and sell high. The following shows the price of the stock over a 17-day period. One may buy the stock at any one time, starting after day 0.



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

The goal is to determine how to maximize profit i.e. find the highest and lowest prices, and then work left from the highest price to find the lowest prior price, work right from the lowest price to find the highest later price, and take the pair with the greater difference. Note that, it is not that one has to buy at the lowest price and sell at the highest price; one just has to make sure that the overall profit is maximum.

There are several methods to solving this kind of problem and each will be covered.

4.4.1 Brute-force

The brute-force solution: An iterative approach to try every possible pair of buy and sell dates in which the buy date precedes the sell date. In other words, given an array of numbers, find the largest consecutive subarray sum by calculating **every possible subarray combination**.

```
for (int i = 0; i < n; i++) // O(n)
{
    for (int j = i; j < n; j++) // O(n)
    {
        int sum = 0; // reset the sum to 0
        for (int k = i; k <= j; k++) // O(n)
        {
            sum += a[k]; // calculate the sum
        }
        if(sum>max){max=sum;} // O(1): replace the max
    }
}
```

Generating every possible combination of subarrays is $\binom{n}{2}$ for a given n, this corresponds to two for loops. There is a third for loop that calculates the sum. This means the complexity of this algorithm is $O(n^3)$

4.4.2 Brute-force v2

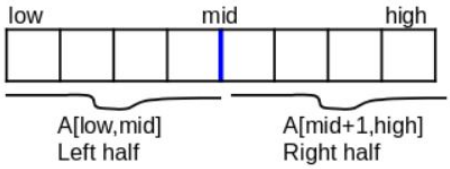
The slightly better brute-force solution: An iterative approach by starting at all positions and calculating running sums.

```
for(int i = 0; i < n; i++) // O(n)
{
    int sum = 0;
    for (int j = i; j < n; j++) // O(n)
    {
        sum += a[j]; // O(1)
        if (sum > max) {max = sum;} //O(1)
    }
}
```

The complexity of this algorithm is $O(n^2)$.

4.4.3 Divide and Conquer Method

Given an array, break down the array into 2 (or more) parts and solve them recursively.

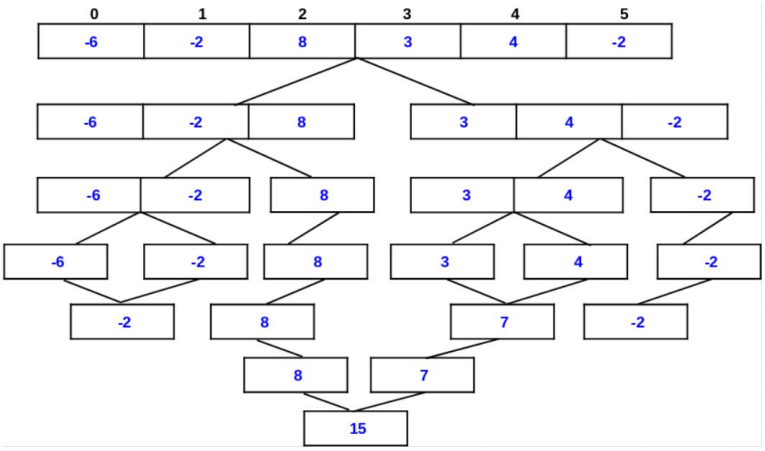


Three possible situations may arise: The maximum subarray can be in the:

1. Left half: `A[low,mid]`
2. Right half: `A[mid+1,high]`
3. Cross the middle of the array: `A[i,j]`

Thus the Divide-and-Conquer algorithm performs the following:

- Divide an array in two halves.
- Find maximum subarray sum in left half.
- Find maximum subarray sum in right half.
- Find maximum subarray sum which crosses the midpoint.
  - Start from midpoint then traverse toward the left side of an midpoint **till sum is increasing.**
  - Perform the same process on right side of an midpoint starting from $mid + 1$ index.



- Maximum of the three is the answer.

Pseudo-code:

```
def max_crossing_subarray(a, low, mid, high):
    left_sum = -float("inf")
    current_sum = 0
    for i in range(mid, low-1, -1):
        current_sum += a[i]
        if current_sum > left_sum:
            left_sum = current_sum
            max_left = i
    right_sum = -float("inf")
    current_sum = 0
    for j in range(mid+1, high+1):
        current_sum += a[j]
        if current_sum > right_sum:
            right_sum = current_sum
            max_right = j
    return (max_left, max_right, left_sum+right_sum)


def max_subarray(a, low, high):
    if low == high:  # base case
        return (low, high, a[low])
    else:
        mid = (low+high)/2
        left_low, left_high, left_sum = max_subarray(a
    , low, mid)
        right_low, right_high, right_sum =
    max_subarray(a, mid+1, high)
        cross_low, cross_high, cross_sum =
    max_crossing_subarray(a, low, mid, high)

        if left_sum >= right_sum and left_sum >=
    cross_sum:
            return (left_low, left_high, left_sum)
        elif right_sum >= left_sum and right_sum >=
    cross_sum:
            return (right_low, right_high, right_sum)
        else:
            return (cross_low, cross_high, cross_sum)
```

4.5 Multiplying large numbers

The naive method for multiplication of large binary numbers is:

Naïve algorithm: long multiplication

```
      1011    (=11 in decimal)
    * 1110    (=14 in decimal)
    ======
   0000000    (=1011*0)
   0010110    (=1011*1<<1)
   0101100    (=1011*1<<2)
 + 1011000    (=1011*1<<3)
 =========
  10011010    (=154 in decimal)
```

The complexity is $O(n^2)$ i.e. if there are $n = 4$ then there are 16 multiplication operations.

The divide-and-conquer method can be used.

4.5.1 Karatsuba's algorithm

The Karatsuba algorithm is a fast multiplication algorithm that uses a divide and conquer approach to multiply two numbers.

- Divide each number into two halves:

  X_L and X_R contain leftmost and rightmost $\frac{n}{2}$ bits of X

  $$X = X_L * 2^{\frac{n}{2}} + X_R$$

  Y_L and Y_R contain leftmost and rightmost $\frac{n}{2}$ bits of Y

  $$Y = Y_L * 2^{\frac{n}{2}} + Y_R$$

- Multiplication:

$$\begin{aligned} XY &= (X_L * 2^{\frac{n}{2}} + X_R)(Y_L * 2^{\frac{n}{2}} + Y_R) \\ &= [2^n X_L * Y_L] + \left[2^{\frac{n}{2}}(X_L Y_R + X_R Y_L)\right] + [X_R Y_R] \end{aligned}$$

There are four multiplications of size $\frac{n}{2}$, so the problem is divided into size n into four sub-problems of size $\frac{n}{2}$. But that doesn't help because solution of recurrence $T(n) = 4T(\frac{n}{2}) + O(n)$ is $O(n^2)$. The tricky part of this algorithm is to **change the middle two terms to some other form so that only one extra multiplication would be sufficient**.

$$X_L Y_R + X_R Y_L = (X_L + X_R)(Y_L + Y_R) - X_L Y_L - X_R Y_R$$

Thus the equation becomes:

$$XY = [2^n X_L * Y_L] + \left[2^{\frac{n}{2}}(X_L + X_R)(Y_L + Y_R) - X_L Y_L - X_R Y_R\right] + [X_R Y_R]$$

With above trick, the recurrence becomes $T(n) = 3T(\frac{n}{2}) + O(n)$ and solution of this recurrence is $O(n^{1.59})$.

4.6 Master method exceptions

The Master method doesn't always apply, for example:

$$T(n) = T(n-1) + O(1)$$

which has to be worked by hand to $O(n)$.