Imperial College London

Discrete Mathematics: Year 2

Big-O and Complexity

Xin Wang

November 25, 2020

Abstract

Algorithms are at the heart of computing which is a significant area in EIE. The design and analysis of algorithms are essential skills to an engineer and there several established design and analysis strategies.

Understanding "Time Complexity" is important in order to write fast code. Time Complexity is the computational complexity that describes the amount of time it takes to run an algorithms.

Contents

1 Introduction

Algorithm: A sequence of computational steps that takes a value as **input** and produces one/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 requires such as memory, communication bandwidth, computer hardware and, most importantly, time.

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 primitive operations "steps" 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.

The design techniques to be covered are:

- Divide and conquer
- Dynamic programming
- Greedy algorithms

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

- Breadth first
- 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

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:

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

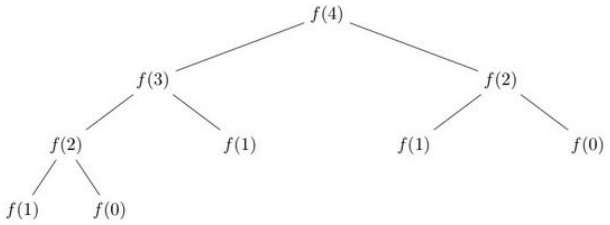


Figure 1: Recursion tree of Fibonacci series

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

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.

```
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]
```

2.3 Iterative algorithm

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.

```
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
```

2.4 Binet's formula

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

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 a 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.

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.

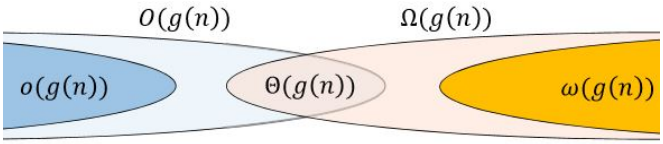


Figure 2: Areas of coverage of the different domains of notations

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:

- 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 $0.1n^2$

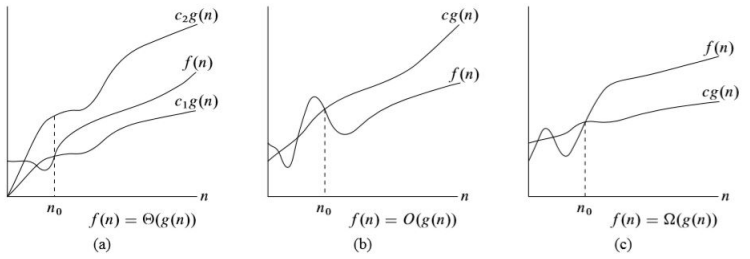


Figure 3: Graphical examples of Θ-notation, O-notation and Ω-notation respectively.

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, 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 intuition about the rate of growth of function because it gives a tight bound unlike big-O and Ω which gives the upper and lower bounds respectively.

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

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

- List $g(n)$:

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

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

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

- 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:

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

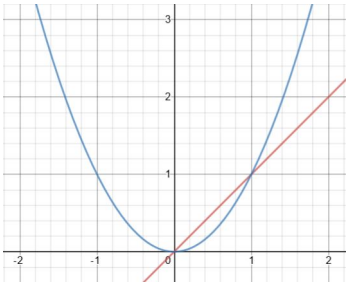
or

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

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

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



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

- List $g(n)$: $g(n) = n$
- Express as $\frac{f(n)}{g(n)}$:

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

- 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 there is only an **asymptotic lower bound**. 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** $cg(n)$.

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

$$0 \leq cg(n) \leq f(n)$$

where: $n \geq n_0$

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

3.1.4 Little-o o notation

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 (Refer to Figure 2).

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

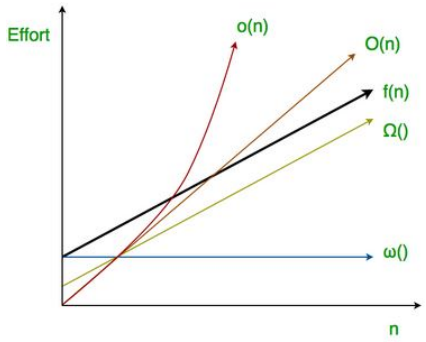


Figure 4: A graphical representation of the relationships of the four notations to $f(n)$

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

$$0 \leq f(n) < cg(n)$$

where: $n \geq n_0$

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

- Bound $0 \leq f(n) \leq cg(n)$ holds for **some** constant $c > 0$
- Bound $0 \leq f(n) < cg(n)$ holds for **all** constant $c > 0$

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

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

3.1.5 Little-omega ω notation

The asymptotic low bound provided by Ω-notation may/may not be **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:

$$0 \leq cg(n) < f(n)$$

where: $n \geq n_0$

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

- Bound $0 \leq f(n) \leq cg(n)$ holds for **some** constant $c > 0$
- Bound $0 \leq f(n) < cg(n)$ holds for **all** constant $c > 0$

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

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

3.2 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.

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 4: $O(log(n))$

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.

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.

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:

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

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

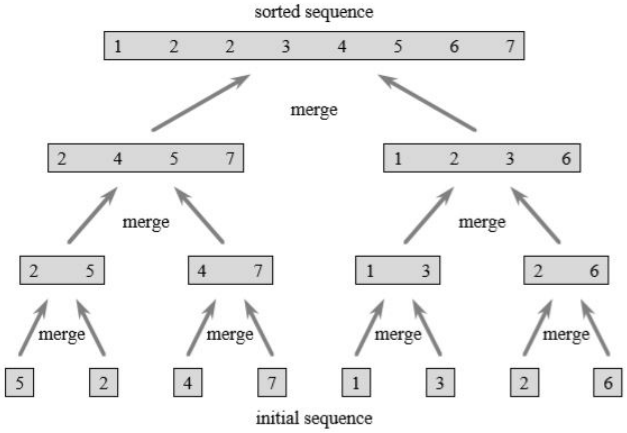


Figure 5

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. $T(n) = O(n^d log[n]) = O(log[n])$

5 Dynamic Programming

Dynamic Programming applies when sub-problems **overlap** and solves the common sub-problems just once and saves it in a table, avoiding recomputing it. The "programming" refers to tabular programming and has nothing to do with computer programming.

Dynamic Programming usually applies to **optimization problems** - where problems have many possible solutions e.g. Google Maps to find the shortest path between sources. Each solution has a value and the goal is to find a solution with optimal (min or max) value. Solutions are called *an optimal solution to the problem* instead of *the optimal solution to the problem* as there may be several solutions.

There are two key attributes that a problem must have in order to be able to apply dynamic programming to it:

- **Optimal substructure**: An optimal solution to a problem contains optimal solutions to subproblems.

  Optimal solutions to whole problem = combination of optimal solutions to sub-problems.

- **Overlapping subproblems**: A recursive solution contains a "small" number of distinct subproblems repeated many times.

There are four steps to developing a dynamic-programming algorithm:

1. **Characterize** the structure of an optimal solution
2. **Recursively** define the value of an optimal solution
3. **Compute** value of an optimal solution (Usually bottom-up fashion)
4. **Construct** an optimal solution from computed information

There are similarities and differences between DP and DAC:

- Similarities: Both partition a problem into smaller subproblems and build the solution of larger problems from solutions of smaller problems
- Difference:
  - DAC work top-down i.e. know exact smaller problems that need to be solved to solve larger problem.
  - DP usually work bottom-up. Solve all smaller size problems and build larger problem solutions from them.

5.1 Applying Dynamic Programming

Mentioned earlier, a naive recursive solution is inefficient since it solves the same sub-problems repeatedly. It would be much more efficient to solve each problem only **once** and save the solution in memory. Dynamic programming uses additional memory to save computation time - an example of **time-memory trade-off**.

> **Memoisation** - A common optimization technique used to primarily speed up computer programs by storing results of computations and returning the cached results when same inputs occur again.

There are two common ways to implement a dynamic-programming approach: **top-down with memoization method** and **bottom-up method**. Both method are $O(n^2)$ but bottom-up method is more efficient.

5.1.1 Data strutures

Types of data structures used are important in the overall efficiency of the algorithm. For example, using array will require arguments to be integers only but often arguments are unknown. A common solution is to use a **hash table** where a data structure T maps to values $T[k]$. The performance is slightly worse than array but advantages are many i.e. insertion/lookup/deletion are $O(1)$ typically and $O(n)$ worse-case.

> **Optimal substructure**: If an optimal solution can be constructed from optimal solutions of its sub-problems. Indicates the usefulness of the dynamic programming and greedy algorithms for a problem.

Data structures will be covered in depth in another chapter.

5.2 Fibonacci numbers and dynamic programming

The mathematical equation describing it is:

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

Solving the problem using dynamic programming, there are two approaches in Dynamic Programming: Memoization (Top-Down) and Tabulation (Bottom-Up).

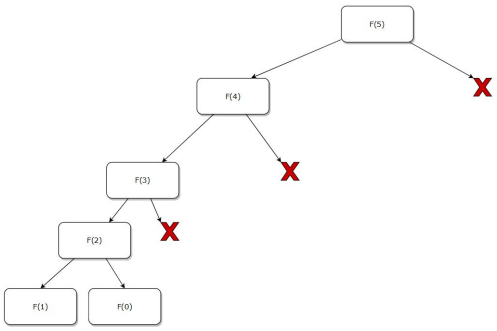
The time complexity of the recursive solution is exponential – $O(\Phi^N)$ to be exact. This is due to solving the same subproblems multiple times.

For the top-down approach, each subproblem is solved one time. Since each subproblem takes a constant amount of time to solve, this gives us a time complexity of $O(N)$. However, since we need to keep an array of size $N + 1$ to save our intermediate results, the space complexity for this algorithm is also $O(N)$.

In the bottom-up approach, we also solve each subproblem only once. So the time complexity of the algorithm is also $O(N)$. Since we only use two variables to track our intermediate results, our space complexity is constant, $O(1)$.

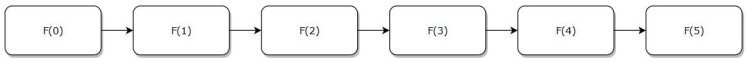
5.2.1 Memoization (Top-Down)

The idea here is similar to the recursive approach, but the difference is that the solutions to subproblems encountered is saved. This way, if the same subproblem is encountered more than once, the saved solution can be used instead of having to recalculate it i.e. compute each subproblem exactly one time.



5.2.2 Tabulation (Bottom-Up)

In the bottom-up dynamic programming approach, reorganize the order in which the subproblems is solved i.e. compute $F(0)$, then $F(1)$, then $F(2)$, and so on.



This will allow us to compute the solution to each problem only once, and we'll only need to save two intermediate results at a time. **This will allow us to use less memory space in our code.**

5.3 Case study: The Rod Cutting Problem

This case study concerns deciding where to cut a long steel rod into smaller rods for maximum profit.

Background: For the inches of the smaller rods to cut into i, the price p_i varies. Determine the maximum revenue r_n obtainable by cutting the rod up and selling the pieces.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Figure 6: Prices for each corresponding inch sold

For example, consider the case where the long rod of 4 inches $n = 4$ is cut into 4 smaller rods, there are various ways to cut it as shown:

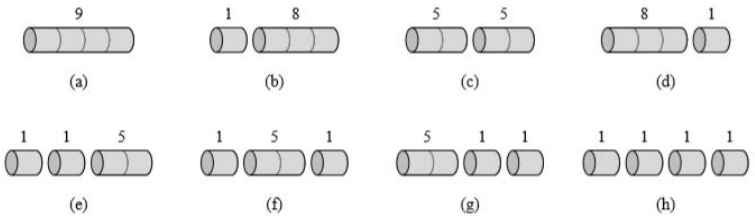


Figure 7: Various ways of cutting the rod with the prices shown. Option (c) is clearly the most optimal.

Denoting the decomposition into pieces using additive notation: The optimal solution cutting rods into k pieces for some $1 \leq k \leq n$:

$$n = i_1 + i_2 + ... + i_k$$

The rods of length $i_1, i_2, \ldots, i_k$ provides sum of corresponding revenue:

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$$

Framing the values r_n for $n \geq 1$ in terms of optimal revenue:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, ..., r_{n-1} + r_1)$$

where the first argument p_n corresponds to no cuts at all.

Since it is unknown at the beginning which value of i optimizes revenue, all possible values of i have to be considered and the one that maximizes revenue chosen.

5.3.1 Recursive top-down implementation

```
CUT-ROD(p,n):
    if n = 0:
        return 0
    q = -infty

    for i = 1 to n:
        q = max(q.p[i] + CUT-ROD(p,n-1))
    return q
```

Points to note:

- The procedure takes, as input, an array of $p[1...n]$ of prices and an integer n.
- Returns the max revenue possible for a rod of length n.
- If rod is length $n = 0$ then no revenue is possible.
- Else, initialize the for loop that computes:

$$q = \max_{1 \leq i \leq n}(p_1 + \text{CUT-ROD}(p, n-1))$$

This program is inefficient as the program calls itself recursively with the same parameter values - solving the same sub-problems repeatedly.

A recursion tree can be created to visualise the number of calls made to CUT-ROD where each node gives the size n or corresponding sub-problem.

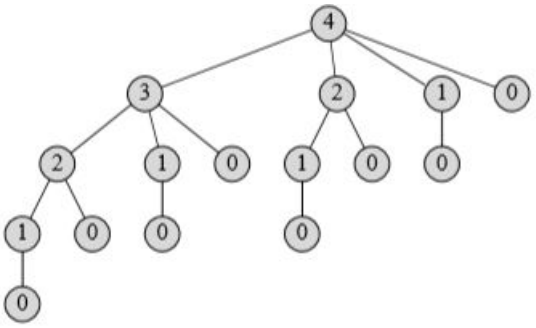


Figure 8: Graphical representaion of the number of calls made to CUT-ROD.

Note the exponential behaviour as n increases.

5.3.2 Using dynamic programming

As mentioned previous with the two methods: **bottom-up method** and **top-down method**, both have the same running time usually with exceptions. The botton-up approach does have better constant factors since it has less overheads for procedure calls.

Top-down with memoization method:

```
MEMOIZED-CUT-ROD(p,n):
    let r[0...n] be a new array

    for i = 0 to n:
        r[i] = -infty

    MEMOIZED-CUT-ROAD-AUX(p,n,r)

MEMOIZED-CUT-ROD-AUX(p,n,r):
    if r[n] >= 0:
        return r[n]
    if n == 0:
        q=0
    else q = -infty:
        for i = 1 to n:
            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p,
               n-i, r))
    r[n] = q

    return q
```

Points to note:

- Main procedure MEMOIZED-CUT-ROD initializes a new auxillary array $r[0...n]$ with value $-\infty$ (denotes unknown - **known values are always positive**).
- Calls helper routine MEMOIZED-CUT-ROD-AUX
- Help routine checks to see where desired value is known already (positive or not), if so then return the value.
- If not, compute the desired value, store it and return it.

Bottom-up method: Natural ordering of sub-problems: smallest sub-problem solved first.

```
r[0] = 0; // Arrayr[0...n] stores the computed
            // optimal values
for j = 1 to n do // Consider problems in increasing
                    // order of size
    q = -infty;
    for i = 1 to j do // To solve a problem of size j,
                        // consider all decompositions
                        // into i and j-i
        q = max(q, p[i] + r[j-i]);
    end
    r[j] = q;
end
return r[n];
```

Points to note:

- New array created to store the values.
- $r[0]$ initialized to 0 since rod length 0 earns no revenue.
- Solve each sub-problem of size j for $j = 1, 2, \ldots, n$ in order of increasing size.
- Direct references to array used instead of making recursive calls to solve the problem.
- Return value is the optimal value r_n

Bottom-up method with reconstruction:

```
// Arrays[0...n] stores the optimal size of the first
   piece to cut off
r[0] = 0; // Array r[0...n] stores the computed
          // optimal values
for j = 1 to n do
    q=-infty;
    for i = 1 to j do // Solve problem of size j
        if q < p[i] + r[j-i] then
            q = p[i] + r[j-i];
            s[j] = i; // Store the size of the
                      // first piece
        end
    end
    r[j] = q;
end
while n > 0 do // Print sizes of pieces
    Prints[n];
    n=n-s[n];
end
```

6 Greedy algorithms

Optimization algorithms usually have a sequence of steps with a set of choices at each step. Using dynamic programming for many problems is sometimes an overkill. A simpler, more efficient algorithm will work better sometimes i.e. a $O(2^n)$ problem using dynamic programming is $O(n^2)$ but with greedy algorithms is $O(n)$.

> **Greedy algorithms**: Any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage.

Greedy algorithms always make a best possible choice in the hopes that each local optimal solution will lead to a global optimal solution. Greedy algorithms are not always possible but are faster when applied. The differences between Greedy Algorithms and Dynamic Programming are summarised below:

Dynamic Programming:

- Optimization problems
- Series of choices
- Considers all choices to find which is the best

Greedy Algorithm:

- Optimization problems
- Series of choices
- Makes the best choice without looking at alternatives

6.1 Case study: Activity-selection problem

The problem of scheduling several competing activities that require exclusive use of a common resource i.e. classroom that can serve only one activity at a time.

The problem is summarized into the following points:

- Let $S = 1, 2, ..., n$ be the set of activities that compete for the common resource.
- Each activity i has a starting time s_i and finish time f_i with $s_i \leq f_i$.
- No activities can share the same resource at any time - Activities i and j are **compatible**.
- Problem: Select largest set of **mutually compatible activities**.

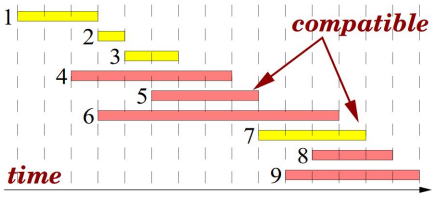


Figure 9: Activity 5 and 7 showing compatibility

The set of data is then sorted in increasing order of smallest to largest finishing time. After sorting, it is easy to identify the subsets of mutually compatible activities.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Figure 10: Table of activities with its start and finishing times

The goal is to develop a recursive greedy algorithm. There are several steps involved that result in the algorithm.

Optimal substructure of the activity selection problem:

6.2 Elements of the greedy strategy

The general process to develop a greedy algorithm:

1. Identify optimal substructure.
2. Case problem as a greedy algorithm with the greedy choice property.
3. Write a simple iterative algorithm.

In order to know when a greedy algorithm will solve a particular optimization problem, the greedy-choice property and optimal substructure is needed. If these two properties can be proved, it is possible to develop a greedy algorithm.

6.2.1 Greedy-choice property

> **Greedy-choice property**: There exists an optimal solution that is consistent with the greedy choice made in the first step of the algorithm.

This aspect is where greedy algorithms differ from dynamic programming. In dynamic programming, the choice made at each step depends on solutions of the sub-problems (thus bottom-up manner). In greedy algorithms, a choice is made depending on the choices made so far, not any future choices. This means that greedy choice algorithms solves problems in the **top-down manner**. Proof is required to show that the greedy choice is always part of the the optimal solution.

6.2.2 Optimal substructure

Problem shows optimal substructure if an optimal solution to the problem contains within it the optimal solutions to sub-problems. This property is required by dynamic programming and greedy algorithms.