CS3230 - Design and Analysis of Algorithms

Sherisse Tan Jing Wen

May 15, 2024

Contents

Chapter 1

Introduction

1.1 Designing Algorithms

When designing / analysing Algorithms, we are concerned with both the Space and Time complexity of the Algorithm in question. (This module will mainly focus on Time Complexity)

Generally, when designing any Algorithm, we can follow the steps below:

1. Understand the Problem
2. Design a method to solve the Problem
3. Convert the method into psuedocode
4. Choose the data structure(s) you need to implement the Algorithm
5. Prove the correctness of the Algorithm (Chapter 4)
6. Analyse the Algorithm's Space and Time Complexity (Chapter 2)

Specifically, when we are interested in solving an Optimization Problem, we can follow the following steps to identify what algorithm we can use (See Chapter 3 for more information on the Types of Algorithms covered):

1. Identify the Optimal Substructure
2. Formulate the Recursive Equation
3. Do the Substructures overlap?
    - **Yes**: Dynamic Programming (Section 3.2)
4. Is there a Greedy Choice?
    - **Yes**: Greedy Algorithm (Section 3.3)

Definition 1.1.1 – Optimal Substructure The Global Optimal Solution to the problem can be found from the Optimal Solutions to 1 or more subproblems

Definition 1.1.2 – Greedy Choice The Global Optimal Solution to the problem can be obtained from greedily selecting the Local Optimal Solution to each subproblem

1.2 Recurrence Relations

Recurrence Relations occur in recursive algorithms, where a function can call itself in each function call. As such, they are typically made up of two parts:

1. The cost of each individual function call
2. The number of recursive calls made in each individual function call

Recurrence Relations typically take the form: $T(n) = aT(n/b) + f(n)$, although they can also take the form $T(n) = aT(n - x) + f(n)$, where $f(n)$ refers to the cost of each individual function call, a refers to the number of recursive calls made, and (n/b) or $(n - x)$ refers to the size of each recursive call.

Chapter 2

Analysis

Typically, running the same algorithm on different machines, and sometimes even running it multiple times, on the same machine, will result in different real-world time taken. This can be due to a variety of factors, such as the processing power of the machine, what other processes are being run, etc.

As such, instead of taking the time taken to run the algorithm in a real-world context, we instead look at the number of operations required for the algorithm to complete, usually in the worst-case, but sometimes in the best or average case, or even the expected case.

2.1 Asymptotic Analysis

However, the equation for the number of operations required can often be complex, with multiple terms and constant factors. As such, we take advantage of the following observation: "Larger terms will always grow larger faster than smaller terms". This means that when analysing the Time Complexity of an arbitrary algorithm, we can reduce the equation of the number of operations required into just the largest term, without any constant factors.

For example, the equation $n^2 + 2n$ can be reduced to just $O(n^2)$.

Asymptotic Analysis refers to this process by which we can analyse the Time Complexity of an Algorithm, where we are concerned with the rate of growth of the total number of operations required, as the input size grows larger, and more concretely, as it approaches infinity.

2.2 Notations

There are 5 main Asymptotic Notations for the purposes of Algorithm Analysis.

Let $f(n)$ be the function of the number of operations the algorithm runs given an input n, and $g(n)$ be the function of the Time Complexity of the algorithm,

1. **Big-O,** $O()$: The Upper-Bound of $f(n)$
   $O(g) = f : \exists$ constants $c > 0, n_0 > 0$ $s.t.$ $\forall n >= n_0, 0 <= f(n) <= cg(n)$
2. **Big-Omega,** $\Omega()$: The Lower-Bound of $f(n)$
   $\Omega(g) = f : \exists$ constants $c > 0, n_0 > 0$ $s.t.$ $\forall n >= n_0, 0 <= cg(n) <= f(n)$
3. **Big-Theta,** $\Theta()$: The Tight-Bound of $f(n)$
   $\Theta(g) = f : \exists$ constants $c_1, c_2 > 0, n_0 > 0$ $s.t.$ $\forall n >= n_0, 0 <= c_1g(n) <= f(n) <= c_2g(n)$
4. **Little-O,** $o()$: The Strict Upper-Bound of $f(n)$
   $o(g) = f : \exists$ constants $c > 0, n_0 > 0$ $s.t.$ $\forall n >= n_0, 0 <= f(n) <= cg(n)$
5. **Little-Omega,** $\omega()$: The Strict Lower-Bound of $f(n)$
   $\omega(g) = f : \exists$ constants $c > 0, n_0 > 0$ $s.t.$ $\forall n >= n_0, 0 <= cg(n) <= f(n)$

Note that the following relations between the various Asymptotic Notations are True:

1. **Reflexivity** (O, Ω, Θ)
2. **Transivity** (for All)
3. **Symmetry** (Θ)
4. **Complementarity** (Between O and Ω, and o and ω)

2.3 Working with Recurrence Relations

As Recursive algorithms call themselves multiple times, it is not as simple as reducing the equation to the largest term as we want to see the total number of operations throughout all recursive calls.

There are 3 main methods to analyse the Time Complexity of Recurrence Relations:

- Master Theorem
- Telescoping (and Recursion Tree)
- Subsitution Method

2.3.1 Master Theorem

The Master Theorem is the simplest method, and thus, the preferred method, to analyse Recurrence Relations. However, take note that in order to apply the Master Theorem, certain conditions must apply.

Let $a >= 1, b > 1$ and f be asymptotically positive, that is, $f(n)$ is positive for all, sufficiently large, n.

The cases of the Master Theorem are as follows:

1. If $\exists$ a constant $\epsilon > 0,\ s.t.\ f(n) = O(n^{\log_b a-\epsilon}),$ then $T(n) = \Theta(n^{\log_b a})$
2. If $\exists$ a constant $k\ s.t.\ f(n) = \Theta(n^{\log_b a}\log^k n),$ then:
   (a) If $k >= 0,$ then $T(n) = \Theta(n^{\log_b a}\log^{k+1} n)$
   (b) If $k = -1,$ then $T(n) = \Theta(n^{\log_b a}\log\log n)$
   (c) If $k < -1,$ then $T(n) = \Theta(n^{\log_b a})$
3. If $\exists$ a constant $\epsilon > 0\ s.t.\ f(n) = \Omega(n^{\log_b a+\epsilon}),$ and if $f(n)$ additionally satisfies the Regularity Condition: $af(n/b) <= Cf(n)$ for some constant $C > 0$ and all sufficiently large $n,$ then $T(n) = \Theta(f(n))$

2.3.2 Telescoping

As its name implies, this method makes use of the Telescoping Series, which is defined by:

$$\text{For any sequence, } a_0, a_1, \ldots, a_n, \sum_{k=0}^{n-1}(a_k - a_{k+1}) = a_0 - a_n$$

Example 1: $T(n) = 2T(n/2) + n$

$$T(n) = 2T(n/2) + n \tag{2.1}$$
$$T(n)/n = T(n/2)/n/2 + 1 \tag{2.2}$$
$$T(n)/n = T(1)/1 + (\log n)(1) \tag{2.3}$$
$$\therefore T(n) = O(n\log n) \tag{2.4}$$

We get $\log n$ at Step 2.3 by performing the following calculation based on the recursion tree derived:

$$n/2^k = 1 \tag{2.5}$$
$$n = 2^k \tag{2.6}$$
$$k = \log n \tag{2.7}$$

Example 2: $T(n) = T(n-1) + C$

$$\begin{aligned}
T(n) &= T(n-1) + C \\
T(n-1) &= T(n-2) + C \\
&... \\
T(2) &= T(1) + C \\
T(1) &= C
\end{aligned}$$

$$\begin{aligned}
T(n) &= T(n-1) + C \\
&= T(n-2) + 2C \\
&... \\
\\
\therefore T(n) &= T(n-k) + kC \\
\therefore T(n) &= O(n)
\end{aligned}$$

Example 3: $T(n) = 2T(n-1) + C$

$$\begin{aligned}
T(n) &= 2T(n-1) + C \\
T(n-1) &= 2T(n-2) + C \\
&... \\
T(2) &= 2T(1) + C \\
T(1) &= C
\end{aligned}$$

$$\begin{aligned}
T(n) &= 2T(n-1) + C \\
&= 4T(n-2) + 2C \\
&... \\
\\
\therefore T(n) &= 2^k T(n-k) + kC \\
\therefore T(n) &= 2^k + k \\
&= (2^n)
\end{aligned}$$

2.3.3 Subsitution

Generally, the Subsitution Method is also a Trial-and-Error approach. It involves guessing the bound that we are looking for, then proving that it satisfies the definition of the Bounds as defined in Section 2.2

However, using this method, there is the possibility of finding a bound which is not tight enough. For example, we guessed that an algorithm is $O(n^3)$ and successfully proved it. However, a tighter bound for the algorithm could be $O(n^2)$.

As such, this method is not advised and usually requires intuition and experience to be able to accurately guess a correct bound, and a tight-enough one.

2.4 Amortized Analysis

There is another method, called Amortized Analysis, which also analyses the Time Complexity of an Algorithm, but by analysing the average performance over multiple operations.

Generally, we can achieve a lower Amortized Time Complexity by spreading out the costs of the operation. This is typically applicable in the following scenarios (not limited to):

- 1 or more Inexpensive Operations, 1 or more Expensive Operations
- The Operation is Expensive in the worst case, but Inexpensive in other cases

2.4.1 Aggregate Method

The Aggregate Method is the most straightforward method for Amortized Analysis, however, it is typically harder to do than the other methods.

The intuition for this method invites us to consider the following: Given any sequence of Q operations, what is the worst possible running time (in average) of a specific type operation over these Q operation?

2.4.2 Accounting Method

The Accounting Method is also known as the Banker's Method. This is because the method can be likened to that of a Bank in operation.

This Method involves charging the i^{th} operation a fictitious Amortized Cost, $c(i)$. This Amortized Cost is then consumed to perform the operation in question, and any amount not immediately consumed is stored in the 'bank' for use by any subsequent operations.

The idea here is that we can impose an additional charge on the Inexpensive Operation(s), with the extra cost stored in the bank to be used to help offset the cost of the more Expensive Operation(s).

Note that the Bank Balance **must not go negative**, that is, we must ensure that $\forall n, \sum_{i=1}^{n} t(i) \leq \sum_{i=1}^{n} c(i)$, where n is the number of operations performed in total.

From the above, we can also conclude that the total Amortized Cost provides an Upper-Bound on the total true costs. Therefore, if this is True, we can see that the Amortized Analysis considers all the Operations and their costs.

Example 1: *Dynamic Table Insertion*

A general Dynamic Table is as follows: Whenever the table 'overflows', "grow" it by creating a new table of double the current table size. (Note that there are a large variety of different functions to define the table size, but this is the generic one covered in lecture)

Prove that Insertion in the Dynamic Table is Amortized $\Theta(1)$

First, notice that the main expensive cost comes from having to move the elements to the new table. Therefore, we have to find an Amortized Cost for the Insertion Operation that negates / reduces the impact of having to move the elements to the new table.

Next, note that we only move the elements from a Table$_k$ to a Table$_{k+1}$ when we have inserted an additional k elements. Then, we need to move $2k$ elements in total to the next table. Therefore, we can see that for each k element we insert into the table, we need to ensure that we have an additional 2 in the Bank for the moving operation.

Therefore, we should charge $1 + 2 = 3$ for each Insertion Operation, which is $\Theta(1)$

2.4.3 Potential Method

The Potential Method is named as such as it can be likened to the Potential in Physics Energy. In the Potential Method, we first investigate what function will decrease after performing the expensive operation in the Algorithm / Data Structure.

Specifically, given the following notations:

- ϕ: Potential Function associated with the Algorithm / Data Structure
- $\phi(i)$: The Potential Value at the end of the i^{th} operation

The Potential Function ϕ must satisfy the following:

- $\phi(0) = 0$
- $\forall i > 0, \phi(i) \geq 0$
- Amortized Cost of the i^{th} operation = Actual Cost of the i^{th} operation + $\Delta\phi_i$
- Amortized Cost of n operations *geq* the Actual Cost of n operations

Generally, if we want to show that the Actual Cost of n operations is $O(g(n))$, it suffices to show that the Amortized Cost of n operations is $O(g(n))$, where the Amortized Cost, $C(i)$, is given by the following equation: $C(i) = T(i) + \Delta\phi_i$

To select an appropriate Potential Function, ϕ, it is advisable to select ϕ such that for the Expensive operations, $\Delta\phi_i$ is Negative to such an extent that it can nullify or reduce the effect of the Actual Cost.

Example 1: *Dynamic Table Deletion*

Similar to that of a general Dynamic Table, however, this time, we want to consider the scenario where we are deleting elements from the table. How the Dynamic Table works is that, if after removing an element, the table which can store $2n$ elements has n elements left, we "shrink" the table by moving all the elements into a smaller table of half the size. (Note that there are a large variety of different functions to define the table size, but this is the generic one covered in lecture)

Prove that Deletion (`pop()`) in the Dynamic Table is Amortized $\Theta(1)$

For the Potential Method, we have to find a function for something which decreases after the expensive operation. Here, similar to the Insertion example, the expensive operation is the moving of elements to the smaller table.

The Potential Function, $\phi(i)$ is thus: The number of `NULL` elements in the table. Notice that when we simply remove an element, if there is no moving of elements, the number of `NULL` elements in the table increases by 1, up till $\frac{n}{2}$. Then, when we move, the number of `NULL` elements becomes 0.

	Actual cost	Difference	Amortized Cost
No resizing	1	1	2
Resizing	N+1	- (N - 1)	2

Chapter 3

Types of Algorithms

This section introduces the various Types of Algorithms that exist, and are taught in this module. For more information on proving the correctness of Algorithms, refer to Chapter **??**

3.1 Divide-and-Conquer

Divide-and-Conquer is a type of Recursive Algorithm, whereby we can split the entire problem into multiple sub-problems, which can each be solved / approached in a similar manner. Broken down, the steps for a typical Divide-and-Conquer algorithm is as follows:

1. Divide the problem (instance) into subproblems
2. Conquer the subproblems by solving them recursively
3. Combine the solutions of the subproblems

This requires a slightly different interpretation of the normal Recurrence Relation Equation: $T(n) = aT(n/b) + f(n)$, as $f(n)$ now also includes the cost of combining the result from solving each sub-problem.

Divide-and-Conquer Algorithms can offer a smaller worst-case run-time if applied properly, by reducing the number of expensive operations required. For example, Stressen's Matrix Multiplication Algorithm runs in $O(n^{\log_2 7})$ time, as compared to the naive approach which runs in $O(n^3)$ time.

Another popular Divide-and-Conquer Algorithm is to find an element in a **sorted** list in $O(\log n)$ time. Additionally, `Merge Sort` is also a Divide-and-Conquer Algorithm

3.2 Dynamic Programming

However, Recursive Algorithms can cause redundant computations, especially when larger problems reduce to the same sub-problems. This then causes multiple recomputations of the sub-problems, which can exponentially increase the run-time of the Algorithm.

In this scenario, we can use Dynamic Programming, which is essentially a normal Recursive Algorithm implementation, but with Memoization. Memoizing the result of the sub-problems allows us to avoid recomputations, as we can simply retrieve the already computed results. Typically, the results are saved to a `HashMap` for easy and quick retrieval and storage in $O(1)$ time.

Typically, Dynamic Programming Algorithms are utilised when the following are present:

1. Optimal Substructure
2. Overlapping Sub-Problems (Otherwise, a normal Recursive Algorithm would achieve the same result)

3.2.1 Top-Down vs. Bottom-Up

Recursion with Memoization is also referred to as the 'Top-Down' Approach in Dynamic Programming. In this approach, we start from the General Case and recurse down to the Base Case, which is where the large number of recomputations occur. This is implemented recursively.

The other method, the 'Bottom-Up' approach, instead goes from the Base Case to the General Case, building up from the sub-problem to solve the general problem. This is implemented iteratively, and typically, no memoization is required for a single run.

Note that typically the 'Bottom-Up' approach is actually minutely faster than the 'Top-Down' approach due to the required recursive function calls for the latter. However, this difference is usually small and does not have a major impact in most cases.

Example 1: *Fibonacci*

Top-Down:

```
int FIBO(int n) {
  if (n <= 2) return 1;
  if (memo[n] != -1) return memo[n]; // memo[1..n] is initialised to -1

  memo[n] = FIBO(n-1) + FIBO(n-2);
  return memo[n];
}
```

Bottom-Up:

```
memo[1] = 1;
memo[2] = 1;

for (int i = 3; i <= n; i++) {
  memo[i] = memo[i-1] + memo[i-2];
}

return memo[n];
```

3.3 Greedy

Greedy Algorithms beat Dynamic Programming and Divide-and-Conquer Algorithms when they work properly. The technique is to recast the problem so that only one subproblem needs to be solved at each step.

Greedy Algorithms, similar to Dynamic Programming Algorithms, thus also require showing Optimal Substructure of the problem, as well as proving the Greedy Choice Property.

Generally, when designing Greedy Algorithms, the steps are as follows:

1. Cast the problem where we have to make a choice which leaves just 1 subproblem to solve
2. Prove that there is always an Optimal Solution to the original problem (Global Optimal) that makes the Greedy Choice, so the Greedy Choice is safe
3. Use Optimal Substructure to show that we can combine an Optimal Solution to the subproblem with the Greedy Choice to get the Optimal Solution to the original problem

3.4 Randomised Algorithms

Randomised Algorithms, as its name suggests, is a type of Algorithm which is not fully Deterministic, that is, the same Algorithm can run on the same Device but may produce different results. As compared to Deterministic Algorithms, Randomised Algorithms are a function of the input n, and the random bit(s) chosen.

Randomised Algorithms are interesting in that they present a new method to potentially further optimise Algorithms, provided we can accept a level of variation in our program. For example, the Deterministic QuickSort Algorithm runs in $O(n^2)$ in the worst case, but the Randomised QuickSort runs in expected $O(n \log n)$ time.

There are 2 different types of Randomised Algorithms, both of which sacrifice something to achieve a better expected running time. Exactly what is sacrified, depends on the type of Randomised Algorithm in question.

3.4.1 Las Vegas Randomised Algorithm

Las Vegas Randomised Algorithms sacrifice the Worse-case Running Time of the Algorithm, while maintaining the guarantee of Correctness. It thus produces a better **Expected** Running Time of the Algorithm.

As such, in order to prove a Las Vegas Randomised Algorithm works 'correctly', it usually involves proving that it has a 'good' expected running-time which is bounded by some time function.

3.4.2 Monte Carlo Randomised Algorithm

On the other hand, Monte Carlo Randomised Algorithms sacrifice the Correctness of the Algorithm, while maintaining the guarantee of a 'good' run-time.

As such, in order to prove a Monte Carlo Randomised Algorithm works 'correctly', it usually involves proving that the algorithm has a small, and thus acceptable, probability of error.

An example of a popular, introductory Monte Carlo Algorithm is Frievald's Algorithm for proving the correctness of Matrix Multiplication.

Chapter 4

Proving Correctness

Proving the Correctness of an Algorithm is an important step in designing and analysing algorithms. Intuitively, if we cannot prove that it works, the Algorithm cannot be considered a reliable solution to the problem.

Depending on the type of algorithm, there are different methods to prove its correctness. However, at its core, we can prove the correctness of algorithms by proving certain statements or properties about said algorithm, which when put together, prove the algorithm.

Note that this Chapter goes through how to prove the correctness of our Algorithms. For information on the different Types of Algorithms, see Chapter 3.

4.1 Loop Invariants

Loop Invariants are mainly used to help prove Iterative Algorithms.

Example 1: *Insertion Sort*

For reference, the psuedocode of Insertion Sort is as follows:

Algorithm 1 Insertion Sort Algorithm

```
**Input:** $A[1 \dots n]$
**for** $i = 2$ **to** $n$ **do**
    $cur\_elem \leftarrow A[i]$
    $j \leftarrow i - 1$
    **while** $j > 0$ **and** $A[j] > cur\_elem$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    **end while**
    $A[j+1] \leftarrow cur\_elem$
**end for**
```

Notice that there are two loops in Insertion Sort. Thus, we require 2 Loop Invariants: 1 for the Outer `For` Loop, and 1 for the Inner `While` Loop. To come up with the properties for the Loop Invariants required, let's look deeper into the code.

First, generally, we can see that Insertion Sort works by iterating through each element in the array (except the first element), and shifts them forwards to the left by 1 as long as they are smaller than the elements before it.

Thus, at the start of each Outer `For` Loop, we can see that:

1. `A[1..i-1] are sorted values of B[1..i-1]`
2. `A[i..n] is equal to B[i..n]`

and, at the end of the Outer `For` Loop, we can see that:

1. `A[1..i] are sorted values of B[1..i]`
2. `A[i+1..n] is equal to B[i+1..n]`

Next, we notice that the Inner `While` Loop performs the task of shifting all elements over by 1 to the right as long as the current element we are looking at is smaller than them. This gives us the clue to come up with our Loop Invariant for the Inner `While` Loop.

At the start of the Loop:

1. `A[1..j] equals C[1..j], A[i+1..n] equals C[i+1..n], cur_elem is C[i] = B[i]`
2. `A[j+2..n] equals C[j+1..i-1], and each of these values are > C[i]`

At the end of the Loop, we can also conclude that the same properties apply, just that `j = j - 1`.

4.2 Mathematical Induction

Mathematical Induction is typically used to prove the correctness of Recursive Algorithms, including subsets such as Dynamic Programming, Divide-and-Conquer, etc. which use Recursion.

Example 1: *Binary Search*

Refer to the following psuedocode for Binary Search:

Algorithm 2 Binary Search Algorithm

```
function BINARY-SEARCH($A$, $lb$, $ub$, $x$)
    if $lb > ub$ then
        return false
    else
        $mid \leftarrow \lfloor (lb + ub)/2 \rfloor$
        if $x == A[mid]$ then
            return true
        else if $x > A[mid]$ then
            return BINARY-SEARCH($A$, $mid + 1$, $ub$, $x$)
        else
            return BINARY-SEARCH($A$, $lb$, $mid - 1$, $x$)
        end if
    end if
end function
```

The proof for Binary Search thus proceeds as follows:

1. Based on the length of the search, $n = ub - lb + 1$
2. Base Case: If $n \leq 0$, that is $ub < lb$, then the algorithm works correctly as it returns `False`
3. Induction Step: If $n > 0$, assume that the algorithm works correctly for all values of $ub - lb + 1 < n$
4. If $x =$ `A[mid]`, then the algorithm works correctly
5. WLOG, If $x >$ `A[mid]`, then x is in the array iff it is in `A[mid + 1..ub]` as `A` is sorted. Thus, by Induction, the answer must be in `Binary-Search(A, mid + 1, ub, x)`
6. Thus, the answers returned are correct and the algorithm works correctly

4.3 Cut-and-Paste Argument

The Cut-and-Paste argument is typically used to show Optimal Substructure, but can also be used to show Greedy Choice Property.

A typical Cut-and-Paste Argument generally runs as follows:

1. Suppose that our solution derived from our algorithm is optimal. but is made up of suboptimal subproblem solutions. (Therefore, not an Optimal Substructure)
2. Then, we must be able to 'cut' out the suboptimal solutions from our solution, and 'paste' the optimal solutions into our solution.
3. However, this results in a more optimal solution than our original derived solution
4. This contradicts our initial assumption that the solution derived from our algorithm is optimal.

Example 1: Ding Dong Ditch (PA3) Problem Optimal Substructure Proof

Suppose in the optimal solution, a friend j rings the doorbell of house i. We are now left with the subproblem of ringing $B_j - 1$ other houses. Assume Optimal Substructure, then this subproblem can also be optimally computed.

Let the optimal value of this subproblem be x. If this subproblem is not optimal, that is, there is a lower sum of anger values possible, y, where $y < x$, then we can put A_i back to get $A_i + y < A_i + x$, i.e., better than optimal. This is a Contradiction of our original assumption where ringing the doorbell of house i is optimal.

Thus, Optimal Substructure is established.

4.4 Exchange Argument

The correctness of the Greedy Choice Property can be done by using the Exchange Argument. Generally, this involves swapping the value(s) of **an** Optimal Solution, and showing that by following the Greedy Choice, we can still get **an** Optimal Solution, that is, showing that the optimality of the solution does not change.

Typically, an Exchange Arguments runs as follows:

1. Define the solutions, that is the solution from the Greedy Choice, and another arbitrary Optimal Solution
2. Compare the two solutions and find the difference in them
3. **Exchange** the elements in question to argue that not taking the Greedy Choice would not result in a more optimal solution

Example 1: Ding Dong Ditch (PA3) Problem Greedy Choice Proof

Greedy Choice: Friend j rings the doorbeel of the houses in increasing order of anger values.

Suppose in the optimal solution, friend j does not ring the doorbell of the house with the smallest anger value, $A[0]$, and instead rings the doorbell of another arbitrary house with anger value $A[i]$. However, we know that $A[i] \geq A[0]$. Thus, this would result in a larger (or equal) overall anger value than if friend j rang the doorbell of the house with the smallest anger value, $A[0]$.

Therefore, the correctness of the Greedy Choice is established.

Overall Example 1: Design of Greedy Algorithm with Optimal Substructure and Greedy Choice proofs

Given n events, where each event takes 1 unit of time and provides a profit of g_i dollars ($g_i > 0$) if the event starts at, or before time t_i, design an algorithm that finds a schedule to maximise the profit earned. Note that only 1 event can be scheduled at any time interval.

First, we design the algorithm as follows:

1. Sort the events by time t in decreasing order.
2. Then, starting from time $t = n$, at each time t we are considering, further sort all events for which their $t_i \leq t$ in decreasing order of Profit.
3. Greedily take the event e_i for which the Profit is the largest.
4. Repeat until time $t = 0$

This is clearly $O(n \log n)$.

The Optimal Substructure Proof is as follows:

Let P be the original problem of scheduling events $1, \ldots, n$ with an Optimal Solution A. Given that we schedule event 1 first, we are clearly left with subproblem P' of scheduling events $2, \ldots, n$. Let S' be the optimal solution to P', and we see that `profit(S)` $=$ `profit(S')` $+g_i$. Hence, the optimal solution for P includes the optimal solution to P', and there is Optimal Substructure.

The Greedy Choice Proof is as follows:

The greedy choice in our solution is to choose the valid events with the highest profit at each time t.

Now, suppose there exists another more optimal solution, S, which does not take the greedy choice. Then, there must exist at least 1 other event in S, let this be e_j, which does not exist in our solution but has a higher profit than at least one of the events taken in our solution. However, this contradicts our assumption that our solution was created by taking the greedy choice as otherwise, we would have taken this event, e_j, as well.

4.5 Expected Values

This is generally related to Randomised Algorithms, where we want to see what is the expected value of something happening. Typically, we can make use of Indicator Variables for this, where we let an arbitrary Random Variable, X, be either 0 or 1 depending on whether something happens.

Example 1: *Coupon Collector Problem*

Suppose there are n students in a Tutorial Class and the TA wishes to make sure that all students answer at least 1 question. However, the TA asks each question by picking a student at random. What is the expected number of questions that the TA has to ask before they achieve their goal?

First, notice the keyword '**expected**' in the question. This gives some hints that we can use the Expectation of a Random Variable to solve the question. Now, note the following:

$$\begin{aligned}
P_1 &= 1 \\
P_2 &= \frac{n-1}{n} \\
P_3 &= \frac{n-2}{n} \\
&... \\
P_n &= \frac{1}{n}
\end{aligned}$$

Therefore, we can conclude that:

$$\begin{aligned}
E[X=1] &= 1 \\
E[X=2] &= \frac{n}{n-1} \\
E[X=3] &= \frac{n}{n-2} \\
&... \\
E[X] &= n
\end{aligned}$$

Thus, by the Linearity of Expectation,

$$\begin{aligned}
E[X=n] &= 1 + \frac{n}{n-1} + \frac{n}{n-2} + \cdots + n \\
&= n(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + 1) \\
&= n(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}) \\
&= n(\lg n) \\
&= n \lg n
\end{aligned}$$

Example 2: *Advanced Coupon Collector Problem*

Now, suppose that the TA wants to select q computing students of the m total computing students in the Class, of which there are n students total, that is, $q < m < n$. The TA selects students randomly as in the basic Coupon Collector Problem. What is the expected complexity of this algorithm, assuming the selection can be done in $O(1)$ time?

First, note that since the selection is $O(1)$ time, the expected complexity of the overall problem can thus be reduced to the expected number of times the TA needs to select a student until they get their q computing students.

The probability of selecting a new k^{th} student after $k-1$ computing students is given by the general formula: $P_k = \dfrac{m-k+1}{n}$. Therefore, if we let X_k be the number of selections, $E[X_k] = \dfrac{1}{P_k} = \dfrac{n}{m-k+1}$.

By the Linearity of Expectation,

$$\begin{aligned}
E[X=q] &= \frac{n}{m} + \frac{n}{m-1} + \frac{n}{m-2} + \cdots + \frac{n}{m-q+1} \\
&= n(\sum_{i=m-q+1}^{m} \frac{1}{i}) \\
&= n(\sum_{i=1}^{m} \frac{1}{i} - \sum_{i=1}^{m-q} \frac{1}{i}) \\
&= n(\Theta(\log m) - \Theta(\log(m-q))) \\
&= \Theta(n \log(\frac{m}{n-q}))
\end{aligned}$$

Chapter 5

NP-Completeness

5.1 Polynomial Time

When we talk about Polynomial Time Algorithms, we refer to Algorithms for which the runtime is Polynomial in the length of the problem instance.

Specifically,

- Instance is a Numerical Value: the length of the input is (typically) taken as the binary encoding of the numeric value
- Instance is an Array / Data Structure: the size of the Data Structure

5.2 Psuedo-Polynomial Time

A Psuedo-Polynomial Time Algorithm is one that:

- Runs in polynomial time in the numeric value of the input, BUT
- Does not run in polynomial time in the length of the input

5.3 Reductions

Generally, Reductions between computational problems help give a way to compare the **hardness** of the two problems in question.

We say that A ***reduces to*** B only if:

Given two problems, A and B, A can be solved as follows:

1. Convert an instance α of A to an instance β of B
2. Solve β and obtain the solution for B given β
3. Based on this solution, obtain the solution of α

NOTE: When performing Reductions, be careful about the direction

5.3.1 Polynomial-Time Reduction

We say that there is a Polynomial-Time Reduction, or a $p(n)$-Time Reduction, from A to B when the following is satisfied:

Given any arbitrary instance α of A of size n,

- An instance β of B can be constructed in $p(n)$ time
- A solution to A for the input α can be recovered from the solution of B for the input β in $p(n)$ time

This also gives rise to the **Running Time Composition Claim**

Definition 5.3.1 – Running Time Composition If

- There is a $p(n)$-Time Reduction from Problem A to Problem B, and
- There exists a $T(n)$-Time Algorithm to solve Problem B on instances of size n

then, there is a $(T(p(n)) + 2(p(n))) = (T(p(n)) + O(p(n)))$ Time Algorithm to solve Problem A on instances of size n.

The Running Time Composition can easily be seen by taking the Algorithm to solve Problem A, as the Conversion of α to β, solving Problem B, then converting the solution back to solve α.

Poly-Reducibility

Definition 5.3.2 – Poly-Reducibility We say that A is Poly-Reducible to B, that is, $A \leq_p B$, if there is a $p(n)$-Time Reduction from A to B for some Polynomial Function $p(n) = O(n^c)$ for some constant c.

Some results of Poly-Reducibility (Given $A \leq_p B$):

- If B has an Polynomial Time Algorithm, then so does A (Follows directly from Running Time Composition)
- If B is "easily solvable" (can be solved in poly-time), then so is A

5.3.2 Karp Reduction

Karp Reduction is the most common method of Reductions (and is the only one taught in this course).

It states that:

Given 2 Decision Problems A and B, a Polynomial-Time Reduction from A to B, denoted $A \leq_p B$, is a transformation from instances α of A to instances β of B such that:

1. α is a YES-Instance of $A \Leftrightarrow \beta$ is a YES-Instance of B
2. The Transformation takes Polynomial Time in the size of α

Decision vs. Optimization Problems

Each problem can be categorised into 2 categories:

1. Decision Problem: The solution to the problem is either 'Yes' (a YES-Instance), or 'No' (a NO-Instance)
2. Optimisation Problem: The solution to the problem requires an exact value

Definition 5.3.3 – Decision Problems A Decision Problem is a Function that maps an instance space I to the Solution Set {YES, NO}

Specifically, any Decision Problem is reducible to its Optimization Problem version, that is, given the solution of the Optimization Problem, we can easily see the solution of the Decision Problem. Thus, we can say that the Decision Problem is no harder than the Optimization Problem.

5.4 NP

Definition 5.4.1 – P Class The set of all **decision** problems which have efficient (poly-time) algorithm solutions

Definition 5.4.2 – NP Class The set of all **decision** problems which have an efficient certifier. Short for: "Non-Deterministic Polynomial Time".

Note that P $\subseteq$ NP. The question then is if P $=$ NP?

Definition 5.4.3 – NP-Hard Class If a Problem X is not known to be in NP, then we say that X is NP-Hard

5.4.1 NP-Complete

Definition 5.4.4 – NP-Complete Class A Problem X in the NP Class is NP-Complete if $\forall A \in$ NP, $A \leq_p X$

It is important to note that all NP-Complete Problems are also considered NP, therefore, all NP-Complete Problems are also poly-reducible to each other.

To prove that a Problem X is NP-Complete, we have to:

1. Show that X is in NP (Find an efficient verifier)
2. Show that X is in NP-Hard (Poly-reducible to another NP-Complete Problem Y)

Chapter 6

Useful Formulas

6.1 Mathematics

Exponentials

1. $a^{-1} = \frac{1}{a}$
2. $(a^m)^n = a^{mn} \leftarrow$ Can be extended to if $n < 0$
3. $a^m a^n = a^{m+n}$
4. $e^x \geq 1 + x$

Logarithms

1. $\log 1 = 0$
2. $\log 0 =$ `undefined`
3. $\log^k n = (\log n)^k$
4. $\log \lg n = \log(\log n)$
5. $\log_c ab = \log_c a + \log_c b$
6. $\log a^n = n \log a$
7. $\log_b a = \frac{log_c a}{log_c b}$
8. $\log_b(\frac{1}{a}) = -\log_b a$
9. $a^{\log_b c} = c^{\log_b a}$
10. $b^{\log_b a} = a$

L'Hopital's Rule:

$$\lim_{x \to c} \frac{f(x)}{g(x)} = \lim_{x \to c} \frac{f'(x)}{g'(x)}$$

Binomial Theorem

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$$

$$\tbinom{n}{k} = {}^nC_k = \frac{n!}{k!(n-k)!}$$

Stirling's Approximation

$$n! = \sqrt{2\pi n}(\frac{n}{e})^k(1 + \Theta(\frac{1}{n}))$$

$$\log(n!) = \Theta(n \lg n)$$

Summations

Arithmetic Progression (AP):

$$\begin{aligned}\sum_{k=1}^{n} k &= 1 + 2 + \cdots + n \\ &= \frac{1}{2}n(n+1) \\ &= \Theta(n^2)\end{aligned}$$

Geometric Progression (GP):

$$\begin{aligned}\sum_{k=0}^{n} x^k &= 1 + x + \cdots + x^n \\ &= \frac{x^{n+1}-1}{x-1}\end{aligned}$$

$$\begin{aligned}\sum_{k=0}^{\infty} x^k &= 1 + x + \cdots + x^n \\ &= \frac{1}{1-x}\end{aligned}$$

Harmonic Series:

$$\begin{aligned}H_n &= 1 + \frac{1/2}{+} \cdots + \frac{1}{n} \\ &= \sum_{k=1}^{n} \frac{1}{k} \\ &= \ln n\end{aligned}$$

6.2 Probability

Terminologies

1. Sample Space S: A set whose elements are called 'elementary events'
2. Event: A subset of the Sample Space S

Probability Distribution

A Probability Distribution $Pr[]$ is a Mapping from Events to Real Numbers satisfying the following properties:

- $Pr[A] \geq 0$ for any Event A
- $Pr[S] = 1$
- Mutually exclusive events

For any 2 events A and B,

$$Pr[A \cup B] = \begin{cases} Pr[A] + Pr[B], & \text{if A and B are mutually exclusive} \\ Pr[A] + Pr[B] - Pr[A \cap B], & \text{Otherwise} \end{cases}$$

Inclusion-Exclusion Principle

$|A \cup B| = |A| + |B| - |A \cap B|$

Conditional Probability

The Conditional Probability of an event A given another event B is: $Pr[A|B] = \dfrac{Pr[A \cup B]}{Pr[B]}$

Bayes Theorem can also be used to calculate Conditional Probability. It says that: $Pr[A|B] = \dfrac{Pr[A]Pr[B|A]}{Pr[B]}$

Random Variables

A Random Variable X is a Function that maps the Sample Space S to Real Numbers.

The Probability Density of X is given by: $f(x) = Pr[X = x]$

For example, The probability that the maximum of the value rolled by 2 dice is 3 is:

$Pr[X = 3] = \dfrac{5}{36}$

Note: To get $\dfrac{5}{36}$, just count all the possibilities.

Expectation

The Expectation, or mean, of a Random Variable X is given by: $E[X] = \sum_x x \cdot Pr[X = x]$

For example, suppose X is the outcome of a fair 6-sided die. Then the expected value of the die is:

$E[X] = 1(Pr[X = 1]) + 2(Pr[X = 2]) + 3(Pr[X = 3]) + 4(Pr[X = 4]) + 5(Pr[X = 5]) + 6(Pr[X = 6]) = 3.5$

Linearity of Expectation

The Linearity of Expectation says that: For any two events, X and Y, and any constant a,

$$E[X + Y] = E[X] + E[Y]$$
$$E[aX] = aE[X]$$

Additionally, if X and Y are independent, then $E[XY] = E[X]E[Y]$

Bernoulli Trial

A Bernoulli Trial has a probability p of success, and a probability $q = 1 - p$ of failure. That is, there is only 2 possible outcomes.

Suppose we want to find the number of trials needed to obtain a success for the first time given a sequence of independent Bernoulli Trials, each with the same probability p of success.

$$Pr[X = k] = q^{k-1}p$$
$$E[X] = \frac{1}{p}$$

Let X be the number of successes in n Bernoulli Trials, then:

$$Pr[X = k] = \sum_{k=0}^{n} \binom{n}{k} q^{n-k} p^k$$
$$E[X] = np$$