This document was created by Nicholas Bertollo on Friday 16th May, 2025 and was last updated Tuesday 9th September, 2025.

This document was made by taking some Ed posts, cleaning them up, adding some context, and presenting them. This means that some of the content of this document doesn't make complete sense in terms of ordering and whatnot, however generally speaking I've fixed them up to make sense. This being said, I respond to a lot of Ed posts that only a handful of people see, and therefore this is partially to justify that time.

Let me know if something is wrong in the document, since it was taken from Ed posts, this document is in varying degrees of quality.

Note that this is useless for bringing into the exam with you as it's much more holistic then what is required of the exam and not content based whatsoever, I would note, however, that I've linked the diagram LaTeXcode in the associated Ed post that was presented in.

1 Structure

The general structure of an algorithms problem is:

1. Create the algorithm.
2. Prove the algorithm is correct.
3. Suggest and prove a time complexity.

These three sections are completely distinct. You shouldn't talk about time complexity or why your algorithm is correct while you describe your algorithm, nor should you talk about your time complexity in your proof. The general idea is that a software developer who has taken this course should be able to understand and implement your algorithm from your english description. They should also understand your time complexity analysis. A mathematician/computer scientist should be able to understand all three sections and be convinced enough that it correct and at that time complexity.

1.1 Algorithm

This is the hard bit most of the time and I cannot give you a way to solve the problem or design all possible algorithms, however here is a statement I wrote about the difficulties in this course relating to the difference between content and problem solving.

1.1.1 Problem Solving

In terms of problem solving in general there are many things which I feel students do not consider when they have a problem in front of them. For example, explore through examples, break down into base assumptions, draw pictures (this is a huge one), and trying to break your problem into a smaller problem, are ideas which are usually universal.

Sometimes I think students think it's a matter of:

1. Understand the problem
2. Solve the problem
3. Profit

However this is nowhere near the case.

There are ideas/skills that relate to algorithms directly. The large one I try and consider in my tutorials is being able to explain yourself. I sometimes pressure students to be able to explain an algorithm to me on the fly using words, because this is both what you do in assignments and also it's also extremely useful to be able to argue. Or even argue a proof, where a proof is defined as using logic to justify the decisions you made when designing your algorithm and why these decisions have the required outcome.

I cannot tell you precisely what you need to do to be able to be able to prepare for the exam. However I know if you consider doing the following

1. Go to tutorials and get advice from your tutors, they will be able to actively help you manage these ideas.
2. Ask questions here on Ed
3. Resist looking at the answers and feel free to spend a long time on a question.
4. Ask friends for hints
5. Work with others on the problem.
6. Go to another tutorial instead of your original, different tutors teach differently
7. Actively think about problem solving techniques that have worked before to solve the problem
   (a) Explore the problem by doing examples
   (b) Explore the problem by drawing pictures
   (c) Consider the resources your problem has
   (d) Consider asking others for help but not for the answer
   (e) Learn to be able to describe your ideas concisely
   (f) Simplify the problem into a easier problem, solve that, and use what you learn to solve the original problem.
   (g) Break the problem down into simplifying steps.
   (h) Consider the precise definition (axioms) of the things you're working with
   (i) Consider theorems and ideas you've seen in class that relate
   (j) Consider the pros and cons of certain data structures and when they should be used.

(k) Consider similar problems and see whether tricks from before work again now

(l) Ask why certain restrictions on the problem are there

(m) Ask what the time complexity implies about the way the problem should be solved

(n) Learn to ask unambiguous questions about your ideas (this is unbelievably important)

(o) Learn to formalise observations you made into simple unambiguous statements

(p) Learn to formalise your ideas in the form of first-order logic.

(q) Think about other problems that "reduces" to the original problem, i.e. that is equivalent to the other problem under some fast transformation algorithm.

(r) Actively think about how you learn best! This is difficult and even I struggle.

then you will be fine. Especially as you learned a majority of these ideas implicitly during mathematics and other subjects that required creative problem solving.

Possibly the point of this post is just for you to recognise that problem solving is a skill and you'll get better at it if you continue working, I feel like people forget this and that people can solve problems better than you because they're better than you, which simply isn't the case.

1.2 Proof

We've learned two major proofing techniques in this course, invariance proofs and exchange argument. We have also developed skills in getting better at logic and induction.

This being said a lot of students still don't have a good grasp on what a proof is. Here is a statement I wrote to a student confused on why we use proofing techniques.

1.2.1 Proof vs Proofing techniques

A general theme is that we don't care tooooooo much whether you use one proofing technique or another. A proof is correct irrespective of whether you used direct proof or an induction proof, or a proof by contradiction. Therefore we're really looking for whether logic flows because at an abstract level, a proof is a sequence of implications stemming from true statements.

We use proofing techniques to be able to help us prove things, and also to allow the proof to be rigorous, i.e. without being ambigious.

I will make a small point that induction should be a clear well-reasoned english explanation, just because you're formalising doesn't mean you need to be super clunky in the way you prove something. Take Kruskals's proof (below) for example, it's maybe not the best proof, I did it quickly, but it reads like an english sentence even though it has notation, mathematics concepts, and whatnot.

1.2.2 Invariance

I will note that in a sense an invariance argument isn't anything knew, an invariance is a statement which stays correct over a given action. Here is a couple of examples

1. In assignment 1, you have to prove the invariant "The elements to the left of the current element is a decreasing sequence."
2. In assignment 2, you had to prove the invariant that after you had swapped two elements in the chain created that the structure is still a chain.
3. For a min-heap we have to keep in the invariant of the "min-heap property," you can see that after we add or remove an element we have to prove that the min-heap property is still true.
4. For an AVL tree, we have to keep two invariants, 1. the BST property, and 2. the property that the depth of the children's subtrees are no more then 1 away from eachother.

Invariants are all over the place and we've been proving them the whole time!

To prove an invariance you just have to show that the statement is true at the beginning, the statement is maintained over an action, and that once the loop has terminated this statement has given useful information. You usually do this via an induction.

1.3 Time Complexity

The time complexity of an algorithm is a function $T(n)$ where n is a number associated with the input to the function[1] $T(n)$ outputs the number of constant time operations that the algorithm performs on an input of size n *in the worst case*! We artificially **define** what a constant time operation is in this course, it is not obvious what a constant time operation is without these definitions. [2]

In computer science we consider three type of time complexity.

1. Worst Case time complexity (used all over this course)
2. Expected case time complexity (used for hashmaps and quicksort)
3. Best case time complexity (Not considered and largely useless)

I will not go further in defining best case or expected case time complexity as we have **NEVER** considered them in this course (other then briefly for hash maps, but the point stands). Worst case time complexity is just assuming that the input to the function we're for a given input n is such that it maximises the amount of steps the algorithm performs.

Note that I have not talked about asymptotic analysis, this is because they're completely distinct concepts. Big Oh and time complexity are different, we just use Big Oh to describe time complexity.

[1]To be rigorous you consider the size of the input, however you consider.this in later courses.

[2]Consider multiplication we define it as a constant time operation, but obviously multiplying two number should depend on the size of the numbers.

1.4 Asymptotic Analysis

For a function $f(n)$, we can upper bound this function upto a multiple by using Big Oh notation, we're going to define that $f(n) \in O(g(n))$ for some function $g(n)$ if and only if there exists some numbers C and N such that

$$f(n) \leq C \cdot g(n)$$

for all $n \geq N$.

The idea of this is that for sufficiently large numbers, $f(n)$ is less then $g(n)$ upto a multiple. Now we can also define a lower bound $f(n) \in \Omega(h(n))$ for some function $h(n)$, however this is perfectly equivalent to the definition above but $\leq$ replaced with $\geq$.

NOTE this isn't **BEST CASE ANALYSIS**! The best case time complexity of an algorithm and the lower bound of the worst case are completely separate ideas. A lower bound implies we already have the worst case function $T(n)$ and we're finding a function smaller then it whereas best case implies that we're performing the algorithm on an instance that minimises the number of constant time operations. For example, in a search algorithm the best case time complexity is almost always $\Theta(1)$, however the lower bound of the worst case time complexity is usually $\Omega(n)$ if there is a input space of size n.

I will make another point that best case, in almost all instances, is useless, whereas lower bound of worst case is useful because it allows us to know when to stop looking for upper bounds.

1.5 Questions!

1. We find the maximum of an array A by iterating over A and keeping a variable $c := -\infty$ which we increase to $A[i]$ if $A[i] > c$. Prove the invariant "After the ith iteration of the loop, c is the maximum of the first i elements of A".

2. Consider an algorithm which checks whether a number n (represented in binary) is prime by iterating i from 2 to $\sqrt{n}$ and checking whether n modulo i is 0. Show the correctness of this algorithm and justify the time complexity based on the amount of bits that it takes to store n.

3. Design an algorithm which, given a graph $G = (V, E)$, n weight functions $w_0, w_1, w_2, \cdots, w_{n-1} : E \to \mathbb{R}_{>0}$ such that $w_i(e) \leq w_{i+1}(e)$ for each $e \in E$ and $i \in \mathbb{Z}_{n-1}$, two distinct nodes $s, t \in V$, and a length L, finds a weight function index i such that the distance between s and t is precisely L using weight function w_i. Prove correctness and find and justify the time complexity.

4. Prove that $n \in \Theta(n-1)$.

5. For a connected simple graph $G = (V, E)$, prove $\log(|E|) \in \Theta(\log(|V|))$.

6. A graph $G = (V, E)$ is bipartite if we can partition V into the sets A and B such that $E \subseteq A \times B$. First show that if G is a tree, then G is bipartite, then show that a graph G is bipartite if and only if it has only even cycles. Where a cycle is considered even if the cycle contains an even amount of edges.

2 Graph Algorithms

In this course we consider many graph algorithms, however I more want to highlight that the first couple weeks of the course can be explained through the following graph.

$$\begin{tikzcd}
& \text{Undirected Graph} \arrow[d, "\text{No self-loops, No parallel edges}"] \\
\text{Connected Graph} \arrow[d] & \text{Simple Graph} \arrow[l, "\text{Exists a path between any two nodes}"'] \arrow[d, "\text{No Cycles}"] \\
\text{Tree} \arrow[dr, "\text{Has a root}"] & \text{Forest} \arrow[l] \\
\text{Complete Tree} \arrow[d] & \text{Rooted Tree} \arrow[l, "\text{Each layer is as filled as possible}"'] \arrow[d, "\leq 2 \text{ Children per node}"] \\
\bullet \arrow[d, "\text{Heap Property}"] & \text{Binary Tree} \arrow[l] \arrow[d, "\text{BST Property}"] \\
\text{Heap} & \text{BST} \arrow[d, "\text{Balancing Constraint}"] \\
& \text{AVL}
\end{tikzcd}$$

There is a disclaimer that this graph has both mathematical and ADT's both considered "the same" in some sense. However I hope this can be overlooked for the sake of pedagogy. Note that I don't talk about graphs that much because they are significantly easier then the topics below.

2.1 Questions!

1. Consider a directed graph $G = (V, E)$ with positive weights and a subset of vertices $S \subseteq V$. Find the shortest cycle that include a node of S.
2. Create and justify an operation Heapify that creates a heap from an array A in $O(|A|)$ time. **Note:** Trivial algorithm, difficult to justify time complexity.

3 Greedy & Exchange Argument

3.1 Exchange

What is exchange argument and why do we use exchange argument to prove the correctness of greedy? Since greedy is defined as taking a locally optimal choice, we have to prove that this choice is correct. And therefore we want to prove that another choice is unoptimal, or equivalent, to the greedy solution.

We do this in one of two ways.

1. We show that if we made a choice different to our greedy algorithm, then this would necessarily move further away from the solution. E.g. with Kruskal's proof we show that if we replace our choice with some optimal choice then this makes it less optimal.

2. We show that if we have an instance of some optimal solution, then we only get a better solution by getting closer to greedy solution, refer to problem 6 of the week 9 tutorial sheet.

Here is an example by proving Kruskal's algorithm. Kruskal's is greedy because it's making the choice to constantly take the lowest weight edge.

3.1.1 Kruskal's Proof

Question Solve the MST problem. Kruskal's orders edge weights in increasing order and at each step, greedily takes the smallest weight edge and sees if it forms a cycle, if it does then we don't add it to our MST, if it doesn't then we do. Prove via exchange argument that this algorithm correctly finds the MST.

Solution Prove Kruskal's Algorithm. We prove Kruskal's by exchange argument.

For our exchange argument we supposed we have the output of Kruskals on a graph $G = (V, E)$, call this $KRUS$ (we assume it's a spanning tree, proof by obvious), and we have our optimal solution which is an MST of G, $OPT \subseteq E$. If they are the same, then we are done. So we assume they're different and consider $x \in OPT \setminus KRUS$.

Because $x \in OPT$ this implies it's apart of the MST. We therefore consider the unique cycle created by adding x to $KRUS$, and consider another edge of this cycle y such that $y \in KRUS \setminus OPT$.

We remove y from $KRUS \cup \{x\}$, and therefore we have a spanning tree (this technically has to be proved but it's easy to see). This is the EXCHANGE that give it it's name, we're exchanging y with x, making it "closer" to OPT in some sense, so we now have $KRUS^* = KRUS \setminus \{y\} \cup \{x\}$.

Now we consider the weights of y and x, since $KRUS$ adds in increasing order, we know that y was added to $KRUS$ before adding x was attempted, this implies that $w(y) \leq w(x)$ and so the sum of the weights of $KRUS^* = KRUS \setminus \{y\} \cup \{x\}$ is greater than or equal to the sum of the weights $KRUS$ by swapping with an element of OPT.

We can continue doing this to make $KRUS^* = OPT$, exchanging edges until they're equal and this will mean $w(OPT) = w(KRUS^*) \geq w(KRUS)$. Therefore either these swaps were with equally weighted edges $w(x) = w(y)$, in which case $w(OPT) = w(KRUS)$, and so $KRUS$ will be an MST, or $w(y) < w(x)$ at any point so $w(OPT) = w(KRUS^*) > w(KRUS)$ and so we reach a contradiction, as OPT is an MST.

And therefore $KRUS$ is a MST and so Kruskal's is correct.

3.2 Questions!

Here are some greedy algorithm / exchange problems that I've come across before. Note that we want the algorithm, proof, and time complexity.

1. Given the coin amounts 1c, 2c, 4c, 8c, 16c. Design a greedy algorithm which, for any amount of cents n, will output the minimum number of coins to make n. Design an algorithm and prove via exchange argument.

2. We define an independent set $I \subseteq V$ over a graph $G = (V, E)$ as a set such that for each edge, only one of it's nodes is in I. For a rooted tree T, find the maximal independent set in $O(n)$ time. **Hint:** Consider whether the leaves are independent.

3. Given a vector space V, a finite subset $S \subseteq V$, and a weight function $w : S \to \mathbb{R}_{\geq 0}$. Design a greedy algorithm which finds a linearly independent subset $I \subseteq S$ such that
$$\sum_{v \in I} w(v)$$
is maximised. You might realise this is similar to the proof of Kruskal's algorithm.

4. Prove Kruskals, Prim's, and Dijkstra's.

5. Given a rational number a/b, design a greedy algorithm which will find a list of natural numbers X such that
$$a/b = \sum_{x \in X} 1/x$$
Note: We are not necessarily looking for an optimal solution in $|X|$

4 Divide & Conquer

4.1 Induction & Recursion

This is a small side note that when considering a divide and conquer algorithm, we do not consider "recursing all the way down". This is important as if you think about all the recursions, then the idea of the divide and conquer becomes increasingly difficult to justify.

But how do you do this without thinking of "recursing all the way down"? Within divide and conquer we always use induction to prove that if the algorithm is correct for our subproblem, then it is correct for our current instance. With divide and conquer we use this intuition to be able to create the algorithm itself. By assuming that our subproblems output the correct output, and then making our algorithm output the correct output given this. This means that our induction and our algorithm are basically the same and should come together as a pair. This is why divide and conquer is interesting for the time complexity step, because the algorithm and proof come in a pair.

4.2 Unrolling

Sometimes I think that people get confused what they're really doing. Like what is the aim of unrolling?

When we unroll we aim to solve for the function T such that T has the property that:

$$T(n) = \begin{cases} \underbrace{a}_{\text{Number of instances}} \cdot T(\underbrace{\frac{n}{b}}_{\text{Size of each instance}}) + \underbrace{f(n)}_{\text{Combine step}} & \text{if } n > 1 \\ c & \text{otherwise} \end{cases}$$

This is the general case for constants a, b, c and function f. To unroll this is to simply use the definition many times until you have $T(1)$, which we know is $O(1)$.

For the sake of simplicity let's assume that $n = b^k$ for some k. This simply makes it easier because when we divide n by b k-times, we know we will get 1.

Now by the definition we've given let's unroll $T(n) = aT(\frac{n}{b}) + f(n)$ once by substituting the formula for T into the right hand side.

This gets us $T(n) = a(aT(\frac{n}{b^2} + f(\frac{n}{b}))) + f(n)$. We can simplify this to:

$$T(n) = a^2T(\frac{n}{b^2}) + af(\frac{n}{b}) + f(n)$$

We can now do this again if we like:

$$T(n) = a^3T\left(\frac{n}{b^3}\right) + a^2f\left(\frac{n}{b^2}\right) + af\left(\frac{n}{b}\right) + f(n)$$

Now we can recognise a pattern. I will skip a few steps that are highlighted more in the example, however the fully solved form of the recurrence is:

$$\begin{aligned}T(n) &= a^{\log_b(n)} \cdot c + \sum_{k=0}^{\log_b(n)-1} a^k \cdot f\left(\frac{n}{b^k}\right)\\ &= n^{\log_b(a)} \cdot c + \sum_{k=0}^{\log_b(n)-1} a^k \cdot f\left(\frac{n}{b^k}\right)\end{aligned}$$

This is how we prove the master theorem in general. Technically you could memorise this and plug everything in and then simplify.

Question Prove Master Theorem. Prove Master Theorem. This contains a *lot* of algebraic manipulation, however isn't overly difficult.

4.2.1 Example!

We aim to solve for the function T that solves:

$$T(n) = \begin{cases} 3T(\frac{n}{4}) + \sqrt{n} & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

Now unrolling once $T(n) = 9T(\frac{n}{4^2}) + 3 \cdot \sqrt{\frac{n}{4}} + \sqrt{n}$. Twice $T(n) = 3^3 \cdot T(\frac{n}{4^3}) + 9 \cdot \sqrt{\frac{n}{4^2}} + 3 \cdot \sqrt{\frac{n}{4}} + \sqrt{n}$. Now if we do this until k times until $\frac{n}{4^k} = 1$, this will becomes $n = 4^k \to k = \log_4(n)$. So we know we have to unroll $k = \log_4(n)$ times. And therefore we know that at the end it will look like this:

$$\begin{aligned}T(n) &= 3^k \cdot T\left(\frac{n}{4^k}\right) + 3^{k-1} \cdot \sqrt{\frac{n}{4^{k-1}}} + 3^{k-2} \cdot \sqrt{\frac{n}{4^{k-2}}} + \cdots + \sqrt{n}\\ &= 3^{\log_4(n)} T(1) + 3^{\log_4(n)-1} \cdot \sqrt{\frac{n}{4^{\log_4(n)-1}}} + \cdots + \sqrt{n}\\ &= n^{\log_4(3)} \cdot 1 + \sqrt{n} \cdot \left(\frac{3^{\log_4(n)-1}}{2^{\log_4(n)-1}} + \cdots + 1\right)\end{aligned}$$

However the term on the right is a geometric series with $\log_4(n)$ terms with ratio $\frac{3}{2}$.

$$\begin{aligned}&= n^{\log_4(3)} + \sqrt{n} \cdot \sum_{k=0}^{\log_4(n)-1} \left(\frac{3}{2}\right)^k\\ &= n^{\log_4(3)} + \sqrt{n} \cdot \left(\frac{1 \cdot ((\frac{3}{2})^{\log_4(n)} - 1)}{\frac{3}{2} - 1}\right)\\ &= n^{\log_4(3)} + 2 \cdot \sqrt{n} \cdot \left(n^{\log_4(\frac{3}{2})} - 1\right)\\ &= n^{\log_4(3)} + 2 \cdot n^{\log_4(3)} - 2 \cdot \sqrt{n}\\ &= 3 \cdot n^{\log_4(3)} - 2 \cdot \sqrt{n}\\ &\in \Theta(n^{\log_4(3)})\end{aligned}$$

This is the first case of the master theorem. I used some non-obvious facts. Like $a^{\log_b(c)}$ is commutative, so $a^{\log_b(c)} = c^{\log_b(a)}$ but this is trivial to prove.

4.3 Gaining information from Time Complexity

With divide and conquer you can gain a lot of information from the time complexity. Consider the following statement, "You cannot search $O(n)$ elements in less then linear time". This is an obvious statement but it breaks divide and conquer algorithms into two schools. The first school is such that we recurse into all sub instances and the other is that we do not. These latter is mostly in the form of a search problem.

If you're not recursing into all sub-instances, this implies you are not using all information of the problem. Therefore if you are searching for an element with a certain property, you have to prove that that element is within the sub instance you recurse into. This implies that your proof necessarily becomes an existence proof. A good example is problem 4 of week 11 tutorial, there are similar problems below.

Now even if you recurse into all sub-instances you can still gain information from the time complexity. We will examine this via an example.

Consider problem 6 of the week 10 tutorial. This problem has an $O(n)$ time complexity and since it's divide and conquer in this course, the time complexity is in the form $T(n) = a \cdot T(n/2) + f(n)$. This implies that the recurrence relation is in the form $T(n) = T(n/2) + O(n)$ or $T(n) = 2 \cdot T(n/2) + O(\log(n))$. Therefore we either "throw away" half the elements each time or only have an extremely limited $O(\log(n))$ amount of time to figure out the majority of a given instance. I believe the former is more likely, and therefore since you're throwing away sub-instances, this implies you necessarily have to do an existence proof, and therefore have to split the array in a way to guarantee the majority is in the sub-instance. We have not solved the problem, however simply with a time complexity we've gained a direction to aim to head towards.

4.4 Questions!

1. Given a complete binary tree T find a lucky node. A node u is considered lucky if it's parent and children nodes are of larger or equal value. Design a divide & conquer algorithm to solve this problem in $O(\log(n))$ time, where $n = |V|$.

2. As an extension of problem 4 of week 11 tutorial, we aim to find a local minimum of a $n \times n$ grid in $O(n)$ time. We define a local minimum as all adjacent tiles being larger or equal to the given tile. **Hint:** Solve problem 4 of week 11 tutorial and solve this problem naively first before starting this problem.

3. We extend this problem even further and consider the k dimensional grid $\mathbb{Z}_n^k$ with side length n. A tile $y \in \mathbb{Z}_n^k$ is adjacent to a tile $x \in \mathbb{Z}_n^k$ if $|x_i - y_i| = 1$ for exactly one $i \in \mathbb{Z}_n$, where x_i refers to the ith component of x. Design an algorithm which finds a local minimum in $O(k^2 \cdot n)$ time.

4. Given a set of n points in $\mathbb{R}^2$, find the two points which are closest together in $O(n \log^2(n))$ time. Try and find an $O(n \log(n))$ solution if possible.

5. Count the number of ways you can validly express n pairs of brackets.