1 Designing an Algorithm

Designing an algorithm is an art and something which this course will help you perfect. At the fundamental level, an algorithm is a set of instructions that manipulate an input to produce an output. For those of you with experience programming, you have often written a program to compute some function $f(x)$ only to find yourself riddled with (a) syntax errors and (b) algorithmic errors. In this class, we won't worry about the former, and instead focus on the latter. In this course, you will not be asked to construct any implementations of algorithms. Meaning we don't expect you to write any 'pseudocode' or code for the problems at hand. Instead, give an explanation of what the algorithm is intending to do, provide an argument (i.e. proof) as to why the algorithm is correct, and provide an analysis of the algorithm's performance (i.e. runtime).

A general problem in this course will ask you to *design* an algorithm X to solve a certain problem with a runtime Y[1]. Your solution should contain three parts:

1. An algorithm description.
2. A proof of correctness.
3. A statement of the complexity.

I strongly suggest that your solutions keep these three sections separate (see the examples). This will make it much easier for you to keep your thoughts organized (and the grader to understand what you are saying).

1.1 Algorithm Description

When specifying an algorithm, you have to provide the right amount of detail. I often express that this is similar to how you would right a lab report in a chemistry or physics lab today compared to what you would write in grade school. The level of precision is different because you are writing to a different audience. Identically, the audience to whom you are writing you should assume has a fair experience with algorithms and programming. If written correctly, your specification should provide the reader with an exercise in programming (i.e. actually implementing the algorithm in a programming language). You

[1] If no runtime is given, find the best runtime possible.

should be focusing on the exercise of designing the algorithm. In general, I suggest you follow these guidelines:

(a) You are writing for a *human* audience. Don't write C code, Java code, Python code, or any code for that matter. Write plain, technical English. Its highly recommended that you use LaTeX to write your solutions. The examples provided should give you a good idea of how to weave in the technical statements and English. For example, if you want to set m as the max of an array a of values, **don't** write a for loop iterating over a to find the maximizing element. Instead the following technical statement is sufficient.[2]

$$m \leftarrow \max_{x \in a}\{x\} \tag{1.1}$$

(b) Don't spend an inordinate time trying to find 'off-by-one' errors in your code. This doesn't really weigh in much on the design of the algorithm or its correctness and is more an exercise in programming. Notice in the example in (1.1), if written nicely, you won't even have to deal with indexing! Focus on making sure the algorithm is clear, not the implementation.

(c) On the other hand, you can't generalize too much. There should still be a step-by-step feel to the algorithm description. However, there are some simplifications you can make. If we have in class already considered an algorithm X that you want to use as a subroutine to then by all means, make a statement like 'apply X here' or 'modify X by doing $(\dots)$ and then apply here'. Please don't spend time writing out an algorithm that is already well known.

(d) If you are using a new data structure, explain how it works. Remember that data structures don't magically whisk away complexity. For example, one can find in $O(1)$ time the minimum element using a min heap but it takes $O(\log n)$ time to add an element. Don't forget these when you create your own data structures. However, if you are using a common data structure like a stack, you can take these complexities as given without proof. Make a statement like 'Let S be a stack' and say nothing more.

[2] It is my notational practice to set a variable using the $\leftarrow$ symbol. This avoids the confusing abusive notation of the $=$ symbol.

1.2 Proof of Correctness

A proof of correctness should explain how the nontrivial elements of your algorithm works. Your proof will often rely on the correctness of other algorithms it uses as subroutines. Don't go around reproving them. Assume their correctness as a lemma and use it to build a strong succinct proof.

In general you will be provided with two different types of problems: Decision Problems and Optimization Problems. You will see examples of these types of problems throughout the class, although you should be familiar with Decision Problems from CS 21.

Definition 1.1 (Decision Problem)**.** A decision problem is a function $f : \Sigma \to \{\text{TRUE}, \text{FALSE}\}$. We call Σ the domain of the problem. Given an input x, an algorithm solving the decision problem efficiently finds if $f(x)$ is TRUE or FALSE.[3]

Definition 1.2 (Optimization Problem)**.** An optimization problem is a function $f : \Sigma \to \mathbb{R}$ along with a subset $\Gamma \subseteq \Sigma$. The goal of the problem is to find the $x \in \Gamma$ such that for all $y \in \Gamma$, $f(x) \leq f(y)$. We often call Γ the *feasible region* of the problem.

Recognize that as stated, this is a minimization problem. Any maximization problem can be written as a minimization problem by considering the function $-f$. We we call x the $\arg\min$ of f and could efficiently write this problem as finding

$$x \leftarrow \arg\min_{y \in \Gamma} \{f(y)\} \tag{1.2}$$

When proving the correctness of a decision problem there are two parts. Colloquially these are called $yes \to yes$ and $no \to no$, although because of contrapositives its acceptable to prove $yes \to yes$ and $yes \leftarrow yes$. This means that you have to show that if your algorithm return TRUE on input x then indeed $f(x) =$ TRUE and if your algorithm returns FALSE then $f(x) =$ FALSE.

When proving the correctness of an optimization problem there are also two parts. First you have to show that the algorithm returns a feasible solution. This means that you return an $x \in \Gamma$. Second you have to show optimality. This means that there is no $y \neq x \in \Gamma$ such

[3] Often a decision problem f is phrased as follows: Given input (x, k) with $x \in \Sigma, k \in \mathbb{R}$ calculate if $g(x) \leq k$ for some function $g : \Sigma^* \to \mathbb{R}$.

that $f(y) < f(x)$. This is the tricky part and the majority of what this course focuses on.

Many of the problem in this class involve combinatorics. These proofs are easy if you understand them and tricky if you don't. To make things easier on yourself, I suggest that you break your proof down into lemmas that are easy to solve and finally put them together in a legible simple proof.

Furthermore, each algorithmic approach covered in this course (e.g. dynamic programming, greedy algorithms, divide and conquer, etc...) lends itself to a specific structure of proof. Consider which general class it fits into and then write a proof for the algorithm you've constructed.

1.3 Algorithm Complexity

This is the only section of your proof where you should mention runtimes. This is generally the easiest and shortest part of the solution. Explain where your complexity comes from. This can be rather simple such as: 'The outer loop goes through n iterations, and the inner loop goes through $O(n^2)$ iterations, since a substring of the input is specified by the start and end points. Each iteration of the inner loop takes constant time, so overall, the runtime is $O(n^3)$.' Don't bother mentioning steps that *obviously* don't contribute to the asymptotic runtime. However, be sure to include runtimes for all subroutines you use. For more information on calculating runtimes, read the next section.

1.4 Example Solution

The following is an example solution. I've riddled it with footnotes explaining why each statement is important. Note the problem, I have solved here is a dynamic programming problem. It might be better to read that chapter first so that you understand how the algorithm works before reading this.

Exercise 1.3 (Longest Increasing Subsequence)**.** Given an array of integers $x_1, \ldots, x_n$, find the *longest increasing subsequence* i.e. the longest sequence of indices $i_1 < i_2 < \cdots < i_k$ such that $x_{i_1} \leq x_{i_2} \leq \cdots \leq x_{i_k}$. Design an algorithm that runs in $O(n^2)$.

Algorithm 1.4 (Longest Increasing Subsequence)**.** This is a dynamic programming algo-

rithm.[4] We will construct tables ℓ and p where $\ell[j]$ is the length of the longest increasing subsequence that ends with x_j and $p[j]$ is the index of the penultimate element in the longest subsequence.[5]

1. For $j = 1$ to n:[6]
   (a) Initialize $\ell[j] \leftarrow 1$ and $p[j] \leftarrow$ NULL.
   (b) For every $k < j$ such that $x_k < x_j$: If $\ell[k] + 1 > \ell[j]$, then set $\ell[j] \leftarrow \ell[k] + 1$ and $p[j] \leftarrow k$.
2. Let j be the arg max of ℓ. Follow p backwards to construct the subsequence. That is, return the reverse of the sequence $j, p[j], p[p[j]], \ldots$ until some term has $p[j] =$ NULL.[7]

Proof of Correctness. First, we'll argue that the two arrays are filled correctly. Its trivial to see that the case of $j = 1$ is filled correctly. By induction on k, when $\ell[j]$ is updated, there is some increasing subsequence which ends at x_j and has length $\ell[j]$. This sequence is precisely the longest subsequence ending at x_k followed by x_j. The appropriate definition for $p[j]$ is immediate.[8] This update method is exhaustive as the longest increasing subsequence ending at x_j has a penultimate element at some x_k for $k < j$ and this case is considered by the inductive step.

By finding the arg max of ℓ, we find the length of the longest subsequence as the subsequence must necessarily end at some x_j. By the update rules stated above, for $k = p[j]$, we see that $\ell[k] = \ell[j] - 1$ and $x_k < x_j$. Therefore, a longest subsequence (not necessarily unique) is the solution to the subproblem k and x_j. The backtracking algorithm stated above, recursively finds the solution to the subproblem.[9] Reversing the subsequence pro-

[4] A sentence like this is a great way to start. It immediately tells the reader what type of algorithm to expect and can help you get some easy partial credit.

[5] We've told the reader all the initializations we want to make that aren't computationally trivial. Furthermore, we've explained what the ideal values of the tables we want to propagate are. This way when it comes to showing the correctness, we only have to assert that their tables are filled correctly.

[6] Its perfectly reasonable to use bullet points or numbered lists to organize your thinking. Just make sure you know that the result shouldn't be code.

[7] Resist the urge to write a while loop here.

[8] We've so far argued that the updating is occurring only if a sequence of that length exists. We now only need to show that all longest sequences are considered.

[9] Backtracking is as complicated as you make it to be. All one needs to do is argue that the solution to the backtracked problem will help build recursively the solution to the problem at hand.

duces it in the appropriate order.

Complexity. The outer loop runs n iterations and the inner loop runs at most n iterations, with each iteration taking constant time. Backtracking takes at most $O(n)$ time as the longest subsequence is at most length n. The total complexity is therefore: $O(n^2)$.[10]

[10] Don't bother writing out tedious arithmetic that both of us know how to do.