# Notes for CIS 3200: Introduction to Algorithms
# Dynamic Programming

Erik Waingarten

## Contents

## 1 Dynamic Programming

This seems to be a topic that students find very mystifying, so these notes and the next few lectures are meant to de-mystify and give you an idea of how to begin thinking about dynamic programming as an algorithms paradigm. Up to now, we've been focusing on computational tasks like sorting a list or finding the median of a number of points, but we will now take a very broad view of computational problems and define the large class of *optimization problems*.

Roughly speaking, an optimization problem is usually phrased according to the following setup:

> **Optimization Problem**: We consider a fixed function $f$ which measures the quality (or cost) of a particular "solution", and the goal is to find the best solution, i.e., that which has the highest quality (or lowest cost) among a set of admissible solutions.

The above formulation has two main things which have remained unspecified. There is a notion of a function which determines the quality or cost, and the set of "allowable" solutions—then the goal is usually to maximize or minimize the function over the possible set of solutions. Some concrete examples of optimization problems include:

- Find the "best restaurant" within five miles.
- How many items can I buy which fit within my budget.
- How many and where should I place hospitals within a city so that each person has a hospital nearby?
- What is the best way to travel from the University of Pennsylvania to the Liberty Bell by car?

In all these cases, we want to maximize or minimize a function $f$ as we vary a set of solutions within an allowable set. We will call this function the *objective* function, and by "allowable solutions," we will mean that there is a set of *constraints* and we consider only solutions which satisfy the constraints. Then, the problem wants to maximize or minimize the function while satisfying a set of constraints. Mathematically, one can introduce the following notation:

- We consider a set of possible objects, or *solutions* which we denote by a set $\mathcal{S}$. Note that this set may be finite or infinite and represents all possible solutions that we will vary over. Here, it will be important from the algorithms perspective, that $\mathcal{S}$ is too large to iterate over.
- We specify a function $f\colon \mathcal{S} \to \mathbb{R}$ which measures, for each solution $s \in \mathcal{S}$ an associated value.
- We also specify a set of constraints $\mathcal{K} \subset \mathcal{S}$ which represent the allowable solutions. Similarly to above, it is usually the case that, either $\mathcal{K}$ is too large to iterate over, or there are few, but it is computationally difficult to iterate over these elements.

Given these definitions, the solution space $\mathcal{S}$, the objective function $f$, and the constraints $\mathcal{K}$, we want to solve

$$\text{min or max of } \{f(s) : s \in \mathcal{K} \subset \mathcal{S}\}.$$

In this class, we will be mostly focusing on *discrete algorithms*, which is to say that the solution space $\mathcal{S}$ will often be a discrete set. However, it is often useful to imagine $\mathcal{S}$ as being a continuous space (maybe something like $\mathbb{R}^d$), and each point in $s$ in the solution space $\mathcal{S}$ has a corresponding function value—so if we think of "graphing" the function $f$, we can imagine a manifold over the space. The goal is to find the minimum or maximum function value subject to a constraint set $\mathcal{K}$ on the solution space. Without giving more information, the algorithmic task of finding a minimum or maximum within this solution space means that we need effective ways to navigate the space.

In what follows, we will specify dynamic programming as a technique for solving these type of optimization problems. It will provide us with a framework, or guide to ask the right questions in order to devise an algorithm. The key is to remember our high level algorithmic paradigm:

> **Algorithmic Paradigm**: How do we take an algorithmic problem, appropriately break it up into "simpler" problems, such that once we solve each we can combine the solutions to our general problem.

## 1.1 An Example: Weighted Activity Selection Problem

Think of the following situation: you are the owner of a concert hall which can rent out the space to various performers which want to use the space. Each performer wants to book the concert hall and submits a time window on which they want to reserve the space for—this means that we consider a collection of $n$ performers, which we label $1, \ldots, n$, and each submits a request for a rental in which they include:

- Time window for renting, specified by the start time $s_i$ and end time $t_i$,
- A price that they are willing to pay for the desired time $p_i$.

Your task is to pick which of the performers you will rent out the space to, where the goal is to maximize the amount of money you will receive. However, there is a constraint, that you cannot rent out the concert hall to more than one performer at any specified time. Formalizing the above in the language of optimization problems, we have

- **Solution Space**: The set of solutions is given by the possible subsets of $[n]$ that we may choose to rent the space to, so $\mathcal{S}$ is the collection of all subsets of $[n]$. Notice that this is a discrete set, but it is very large—containing $2^n$ elements. Thus, we do not want to naively iterate through all possible sets.
- **Objective Function**: The objective function $f\colon \mathcal{S} \to \mathbb{R}$ maps a subset $S \in \mathcal{S}$, which is a subset $S \subset [n]$ to the total amount of money that we receive from rents, i.e.,
$$f(S) = \sum_{i \in S} p_i.$$
- **Constraints**: The constraint is that we can only rent out the space to one performer at any particular time, which means that we have consider the set of admissible solutions $\mathcal{K}$

comprising of

$$\mathcal{K} = \left\{ S \subset [n] : \forall \ell \in \mathbb{R}, \sum_{i \in S} \mathbf{1}\{\ell \in [s_i, t_i]\} \leq 1 \right\}.$$

The goal of computing the maximum amount of money that we can collect is the same as that of solving

$$\max_{S \in \mathcal{K}} f(S).$$

So we will describe dynamic programming as a systematic way to understand our problem. As we will see, it is oftentimes the case that effectively finding the minimum or maximum in complicated solution landscapes is very algorithmically difficult *unless* the specific problem that you are studying has a very special structure to it. This special structure allows you, as an algorithm designer, to effectively explore this solution space without necessarily enumerating over possible solutions—each speed-up of your algorithm corresponds to a more efficient way to search through this complicated solution space.

Therefore, even before we begin to think about efficient algorithms, we should start by looking for a "top-level guide" which highlights why our particular problem has a special structure and points to how we will explore the solution space. This guide should take the form of a guess about a systematic way to processes the input which would lead to provable guarantees on the corresponding algorithm. I want to emphasize that coming up with these top-level guides is really more of an art than a science, and students will get better at this with practice.

> **Top-Level Guide**: Guiding intuition which highlights a special structure to the problem which makes it computationally tractable. This guide allows you to systematically explore the solution space to find the minimizing or maximizing value.

In the case of our specific problem, a good "top-level guide" goes something along the following lines:

> In order to determine which performers to rent the space to, it may be helpful to consider the performers in some chronological order. If I rent to a performer, I must remove the set of other performers whose rent time intersects. If I do not rent, then I must not consider that performer anymore.

Following the above guide would correspond to an algorithm which first sorts the performers (perhaps according to the start time). Thus, from now on, we assume we've sorted the performers in order for $s_1 \leq s_2 \leq s_3 \leq \cdots \leq s_n$. Letting $\Omega_0 = [n]$, the goal is to systematically search through the possible subsets of $\Omega_0$ (which encode renting or not renting) to decide which gives the largest sum of money. The goal is to eventually guarantee that we've found a solution which is higher than everything in the solution space $\mathcal{K}$, and this involves asking a simple question as we process the elements. Given the fact that we will process elements in a sorted order, we have the following two cases to consider, when we need to decide, "do we rent to performer 1"?

- **If Rent**: If we decide to rent out to performer 1, then I receive a payment of $p_1$, but we must not be able to rent to any of the performers whose start time $s_i$ is before the end time $t_1$.

Thus, we have that the maximum amount of payment we can obtain among *sets of performers which include* 1 is the same as the maximum payment over set of performers we can obtain among those whose start time is after $t_1$. In particular, if $i^*(1)$ is the first performer whose start time $s_{i^*(1)}$ is after $t_1$, then I should update

$$\Omega_1 \leftarrow [i^*(1), n] \qquad \text{and ask} \qquad \max_{S' \subset \Omega_1} f(S').$$

This means that if I knew this maximum of the *sub-problem* which considers intervals in the range $[i^*(1), n]$, then the amount of money we obtain from renting out to performer 1 becomes $p_1$ plus this maximum.

- **If Not Rent**: If we decide not to rent out to performer 1, then I don't receive payment, but I also don't need to rule out any other performers. In particular, the maximum payment we can obtain when we don't rent out to performer 1 becomes the maximum payment over the set of performers $[2, n]$. If we knew the maximum payment we can obtain among performers $[2, n]$, that becomes the maximum among those which do not include 1.

In particular, we have the following scenario. In order to determine whether or not we should include performer 1, we need to (i) find the $i^*(1)$ which takes $O(\log n)$ time if we sorted $t_1, \dots, t_n$, (ii) solve the *same sub-problem* on the performers after and including $i^*(1)$, and (iii) solve the *same sub-problem* on the performers coming after and including 2—then, we combine these by letting

$$\max_{S \subset [n]} f(S) = \max \left\{ p_1 + \max_{S' \subset [i^*(1), n]} f(S'), \max_{S'' \subset [2, n]} f(S'') \right\}.$$

**Divide-and-Conquer?** The above expression leads to a natural "divide-and-conquer" algorithm, so let's analyze the performance of that solution (even though it will be too expensive). The algorithm would proceed in the following way, where we assume that the input is a sorted list of intervals and price pairs $\{(p_1, s_1, t_1), (p_2, s_2, t_2), \dots, (p_n, s_n, t_n)\}$:

1. **Base Case**: When there is only one performer, include it and return the price it pays.
2. **Divide**: Consider the two sub-problems corresponding to including or not including the first performer (that which has its start time first). One sub-problem contains $n - 1$ many performers (in the case we do not include it), as this considers the $n-1$ other performers. The other sub-problem includes the first performer, and it may be the case that some $k$ performers are ruled out, so we solve the problem of the remaining $n - k - 1$ performers.
3. **Combine**: We now take both sub-problems and we add $p_1$ to the second and pick the maximum.

Suppose we analyzed this algorithm, we'd write the recurrence relation $T(n)$ to be the maximum running time of the algorithm with $n$ performers, and we'd obtain, that after we sort the elements we have

$$\begin{aligned} T(n) &\le \underbrace{O(1)}_{\text{finding performers to remove, optimistically}} + \underbrace{T(n-1)}_{\text{first sub-problem}} + \underbrace{T(n-1)}_{\text{second sub-problem}} \\ &\le O(1) + 2 \cdot T(n-1). \end{aligned}$$

The claim is that this recurrence relation becomes $O(2^n)$, which is way too expensive. One must look closely at the execution of the algorithm above, and one notices that the *same exact recursive call is being solved multiple times*. Why should we have to re-compute something if we've already computed it—when we call a recursive call which has already been executed, if we had written the answer down, we could simply check what the answer is without having to recompute it. Dynamic programming is exactly this idea: we want to do divide-and-conquer, but we'll make sure that we never re-compute the exact same sub-problem by storing its solution. What we need to do is set the proper notation in order to formalize this notion of "not recomputing the same sub-problem."

**Dynamic Programming: Finding the Right Notation for Sub-Problems.** In this case, the claim is that all of the sub-problems that we consider correspond to intervals between a starting performer $i$ and all performers coming after $i$. Thus, we may introduce the notation:

$$G(i) = \text{ maximum total rent achievable from renting to performers among } i, i+1, \dots n.$$

and what we showed was that

$$G(1) = \max\{p_1 + G(i^*(1)), G(2)\}.$$

In the algorithm, we will store a table which keeps the values of $G$ for every $1, \dots, n$. Whenever we compute a value of $G(i)$, we store that value in the table, and before we need to compute any particular value of $G$, we first check in the table. There are at most $n$ positions in the table, and each recursive call is solved at most once.

This leads to the following algorithm:

- Initialize an array $G[1..n]$ which is initialized to $-1$ (where we note that $-1$ indicates that the sub-problem $G(i)$ has not been solved, since it cannot be negative).
- Use the divide-and-conquer algorithm above, but before calling each recursive call, first check if $G[i]$ is already set. If it is set, use it instead of recursing. If it is not set, then recurse to set it and update the array $G$.

The proof of correctness essentially follows from the proof of correctness of the divide-and-conquer algorithm, since we are exactly simulating the divide-and-conquer algorithm. The question here is, how much did we gain in the running time by storing the solutions to the various sub-problems.

**Claim 1.1.** *The running time of the algorithm is at most the number of sub-problems times the time it takes to solve a sub-problem assuming we have solved the sub-problems that it calls.*

There are $O(n)$ sub-problems, and it takes $O(\log n)$ to perform the binary search and find $i^*(j)$ for any particular $j$, and this means the algorithm takes $O(n \log n)$. Performing the sorting on the start times also takes $O(n \log n)$, so that we take total time $O(n \log n)$. So we've gone from $O(2^n)$ to $O(n \log n)$—a massive speedup.

## 1.2 An Example: Station Placement Problem

We receive as input the locations of towns along a rail-line $t_1 \leq t_2 \leq \cdots \leq t_n$, and the railway company wants to determine: how should they place $k$ stations in order to serve all of these towns? In order to place this in the optimization problem language we let:

- **Solution Space:** The solution space is all ways of placing $k$ stations along the railway, so a solution would correspond to a tuple $(s_1, \ldots, s_k) \in \mathbb{R}^k$ which corresponds to the $k$ stations that we may place. Notice here that it does not make sense to "brute force" over all solutions, as the set of solutions here is infinite.

- **Objective Function:** We must now specify an objective function to measure the quality of how good a tuple of stations $(s_1, \ldots, s_k)$ is at serving the collection of towns $t_1 \leq t_2 \leq \cdots \leq t_n$. There is some flexibility in defining the objective function here, but a reasonable objective function would say that a placement is good if every town needs to travel at most some quantity to get to its nearest station. In order to encode this, we may think of the objective function as being a "cost" which measures

$$f(s_1, \ldots, s_k) = \max_{i \in [n]} \min_{j \in [k]} |t_i - s_j|,$$

  which measures the maximum distance that any town $t_i$ must travel to get to its nearest station $s_j$.

In this case, we have no constraints, and the goal is to now pick a set of locations $(s_1, \ldots, s_k)$ which *minimizes* the cost $f(s_1, \ldots, s_k)$, for a fixed set of towns $t_1 \leq \cdots \leq t_n$. We again must come up with a top-level guide which serves to guide our search through the best possible placement of $s_1, \ldots, s_k$. Again, we are looking for a top-level guide which identifies some structure in the solutions which we must search through in order to find the placement.

> **Top-Level Guide**: What can I say about the types of optimal solutions which is special about this problem? Presumably, a constrained set of optimal solutions allows me to design a exploration mechanism which only considers the types of optimal solutions that I am looking for.

For this problem, it makes sense to consider how each station will serve the various towns so it makes sense to consider the following top-level guide:

> An optimal placing of stations to serve towns will have the property that each station serves an interval of towns. Thus, if I consider placing the stations from "left-to-right," I need to consider how many towns does the station need to serve? At the very beginning, how many towns should the first station serve, given that it will serve an interval between towns $t_1$ and town $t_i$?

So we must now analyze the question of, for placing the first station $s_1$, will it serve all towns between $t_1$ and $t_i$? There are then $n$ cases, corresponding to each of the towns that may be the last town that station $s_1$ serves:

- For each $i$, If $t_i$ is the last station that $s_1$ serves, then the cost incurred from station $s_1$ becomes $|t_1 - t_i|/2$, as the optimal place to serve stations $t_1$ through $t_i$ becomes to set $s_1$ at the middle, and pay this amount. Then, we must choose how to set stations $2, \ldots, k$ (a total of $k-1$ stations) which serve stations $t_{i+1}, \ldots, t_n$.

At this point, it makes sense to define our sub-problem notation, by letting $G$ be the function which will store the minimum cost placement. In this case, $G$ takes two arguments, the index $i \in [n]$ which encodes the first town considered (analyzing it from left to right), and the number of stations that we are allowed to place. Then, letting

$$G(i, \ell) = \text{ minimum cost of serving towns } t_i, \ldots, t_n \text{ with } \ell \text{ stations.}$$

and therefore, we have

$$G(i, \ell) = \min_{i' > i} \left\{ \max \left\{ \frac{|t_i - t_{i'}|}{2}, G(i'+1, \ell - 1) \right\} \right\},$$

because, if we let a station cover towns $t_i, \ldots, t_{i'}$, then those towns pay $|t_i - t_{i'}|/2$ to travel to that station—either this because the maximum that those stations need to pay, or the maximum lies within the towns served by other stations. Note that we need a base case for our recurrence, and this is naturally set to

$$G(i, \ell) = \begin{cases} 0 & i \geq n+1 \\ \infty & i \in [n] \text{ and } \ell = 0 \end{cases}$$

**Running Time.** Note that we now have $n \cdot k$ many sub-problems, and answering each sub-problem assuming we have solve the recursive sub-problems is $O(n)$, which means that our total running time of the "memoized" divide-and-conquer solution would become $O(n^2 k)$.

## 2 Summary: Top-Level Guides

We considered the *station placement* problem, which sought to find the location of $k$ stations along a line in such a way so as to minimize the maximum total distance that each town had to travel in order to get to its nearest station. The property, or the "top-level guide" was asking the question of how many towns the right-most station will serve. We will now consider, what makes for a good "top-level guide"? Presumably, the approach to solving a dynamic programming problem would say:

- We establish a top-level guide which asks a single (simple) question which we will search for the answer.
- We will need to devote some computational time in order to follow our top-level guide, and our hope is that:
  - For all the possible options that we need to consider, the resulting "sub-problem" would be a problem which is similar (or hopefully exactly the same) as the original problem we are trying to solve. This is in line with the "divide-and-conquer" mentality for algorithm design.

- It should be the case that, whenever we devote computational effort, we should always make "algorithmic progress." The notion of progress is vague and inherently problem specific—it can be, both, exploring the portion of the solution space which may eventually lead to an optimal solution, or exploring a portion of the solution space which will help us "rule out" many of solutions as potentially optimal.

The above descriptions are inherently somewhat vague, and that is what makes dynamic programming oftentimes challenging to learn. There is no precise formula for solving a particular problem—the hope is that with enough practice and time, there are certain design patterns which you will see as you encounter problems. There is an important aspect, which played an important role in both problems, is to define the proper *notation* in order to capture which sub-problems arise.

> **Importance of Notation**: In dynamic programming, it is important that we always define the notation which captures the sub-problems that we will solve. This is exactly the quantities that we need to store in order to not have to "re-do" work.

For Weighted Activity Selection, our sub-problems were parameterized by $i \in [n]$ and $G(i)$ was the maximum total rent achievable for performers $i, i+1, \dots, n$. For Station Placement, there were two parameters, $i \in [n]$ and $\ell \in [k]$, such that $G(i, \ell)$ was the maximum among the towns $i, i+1, \dots, n$ need to travel to get to the closest of $\ell$ stations. With this in mind, we will do another dynamic programming problem.

## 3 Optimal Static Binary Trees

We want to maintain a static set of $n$ keys which will never change, and we want to maintain a binary tree which stores these keys. The question is how best to store the binary tree if we know the probability that any particular key will be accessed. In particular, there are $n$ values $p_1, \dots, p_n \geq 0$ where $p_i$ denotes the probability that key $i$ is accessed. Thus, we have

$$\sum_{i=1}^{n} p_i,$$

and this is meant to capture some prior information which we know, since keys where the values of $p_i$ are high will appear more often that other keys. There is a motivation from the textbook where, for example, we want to store a set of words (each key corresponds to a word) and we want to store these words in a binary tree such that we can access them—however, we know that some words are much more common that others, so we can assign a probability to each word. Now, the goal is to minimize the *expected* time complexity of accessing a particular key *if we choose the keys according to the probability* $p_1, \dots, p_n$. More specifically, each key will be stored in a binary tree, and to each of the stored keys in a tree $T$, we can consider the depth $d_i$ that the specific key $i$ is stored in $T$—then, the search time would be $d_i$, since this is how far down we need to go down the tree to find the key. The goal is to find the tree $T$ which minimizes the expected time of an access according to distribution $p_1, \dots, p_n$.

- **Solution Space**: The solution space consists of all possible binary trees $T$ whose nodes are $[n]$ and a left-to-right order traversal of the tree $T$ goes in order $1, \dots, n$. This ordering is important, since we will use the tree $T$ to search, and this means that if $i$ is a root, the left-subtree nodes must always be less than $i$ and the right-subtree nodes must always be greater than $i$. The goal will be to output a tree $T$, and note that each such tree defines a sequence of depths $d_1, \dots, d_n$ as the depth of the nodes $1, \dots, n$ in the tree $T$. In this case, there are no constraints.
- **Objective Function**: Here, we want to minimize the expected depth, so that for any tree $T$ over $[n]$ which defines depths $d_1, \dots, d_n$, we write

  $$f(T) = \sum_{i=1}^{n} p_i \cdot d_i,$$

  which measures the expected depth of a key sampled from the distribution $p_1, \dots, p_n$.

The question is, what are we going to use about the problem which allows us to break it down. Recall the basic elements of a top-level guide: (i) we want to have a disciplined way to search through all possible binary trees which respect the ordering on $[n]$, (ii) we want to ensure that if we are going to search through part of the space, that the subsequent sub-problems are smaller versions of the original problem.

> **Top-Level Guide**: A binary tree which proceeds via an order of the nodes must select one of its elements as its root node, and once we consider a root node as $i$, we must consider the elements which come between $1, \dots, i-1$ in the left-subtree, the elements which come between $i+1, \dots, n$ as its right-subtree, and then the cost incurred when we combine.

Importantly, the sub-problems on the left- and the right-subtree are quite similar problems (and those which we may solve recursively). If we consider breaking down the problem by considering the root node as $i$, then the right-subtree will consider the nodes $i+1, \dots, n$; but then, if we pick $j$ as the root of the right-subtree, then the left-subtree of $j$ becomes $i+1, \dots, j-1$. In particular, the subproblems which will arise will consist of sub-intervals of the interval $1, \dots, n$. So a suggestion corresponds to considering

$$G(i, j) = \text{ cost of optimal binary tree on the nodes } i, \dots, j,$$

and our starting point becomes $G(1, n)$. As we've already said, we will want to say

$$G(1, n) = \min_{i \in [n]} \left\{ p_i + L_i + R_i \right\},$$

were we let $L_i$ and $R_i$ denote the left- and right-subtree if we were to choose the root at $i$. Now, the costs of the left- and the right-subtree will be

$$L_i = G(1, i-1) + \sum_{j=1}^{i-1} p_j \qquad \text{and} \qquad R_i = G(i+1, n) + \sum_{j=i+1}^{n} p_j,$$

because we are doing building the optimal tree on the intervals $1, \ldots, i-1$ and then the interval on $i+1, \ldots, n$, and then we have made these subtrees one level deeper. For each $j \in 1, \ldots, i-1$ or $i+1, \ldots, n$ the "probability" $p_j$ pays an extra level, since now it must go through the root $i$ in order to arrive at $p_j$. This results in the additional summation of $p_j$. Putting things together, we have

$$\begin{aligned} G(1,n) &= \min_{i \in [n]} \left\{ p_i + G(1, i-1) + \sum_{j=1}^{i-1} p_j + G(i+1, n) + \sum_{j=i+1}^{n} p_j \right\} \\ &= \min_{i \in [n]} \left\{ G(1, i-1) + G(i+1, n) + \sum_{i=1}^{n} p_i \right\}. \end{aligned}$$

This gives us the expression for $G(1,n)$, but the general formulation would be

$$G(\ell, k) = \min_{i \in \{\ell, \ldots, k\}} \left\{ G(\ell, i-1) + G(i+1, k) + \sum_{j=\ell}^{k} p_j \right\}.$$

Note that one needs to handle the base cases appropriately:

- A binary tree with a single element $i$ picks it at its root and has cost $p_i$.
- A binary tree covering an interval without any elements (for example, the interval $[i, i-1]$ or $[i+1, i]$) has cost 0.

**Running Time.** Here, there are $O(n^2)$ sub-problems (one for each interval $[i, j]$ with $i < j$, and solving each sub-problem requires one to go through all elements in the interval and check how to break it up. This gives us time $O(n)$ for combining sub-problems. This means that the total time that we take is $O(n^3)$.

## 4 The Edit Distance

We will now study what is perhaps the most famous dynamic programming problem. This is a problem which frequently comes up when we are seeking to compare sequences. Consider the case where there are two documents $x$ and $y$, where each document may be encoded by an array of characters (potentially of different lengths). We then want to compute what is the optimal way to change from the document $x$ to document $y$ by inserting, deleting, and substituting characters. Formally, the problem is described as follows:

- There is a fixed alphabet $\Sigma$ which is a finite set, and then the strings $x, y$ are $n$- and $m$-length strings in $\Sigma$. So $x \in \Sigma^n$ and $y \in \Sigma^m$. The goal is to find the best possible way to "edit" the string $x$ into the string $y$.
- The allowable possible edits are the following, were each operation "costs" one unit and the goal will be to find the minimum number of total edits which go from $x$ to $y$. The allowed edits are:

  - For any string $z \in \Sigma^\ell$ (for some $\ell$), and any $i \in [\ell]$, we can "delete" the character $z_i$. Once we do this, we go from $z \to z'$ where $z' \in \Sigma^{\ell-1}$ and contains the characters $z_1, \dots, z_{i-1}, z_{i+1}, \dots, z_\ell$; which we re-index in $z'$.
  - Similarly, we may start with a string $z \in \Sigma^\ell$ and specify an index $i \in \{0, \dots, \ell\}$ and a character $a \in \Sigma$, and we can "insert" the character $a$ between positions $i$ and $i+1$. We then obtain another string in $z' \in \Sigma^{\ell+1}$ where the new $i+1$-th character is now $a$, and all subsequent characters are shifted.
  - Finally, we may choose an index $i$ and substitute a character for another character.

The goal is to compute: which is the minimum number of edits which are needed in order to go from one string $x$ to another string $y$.

**Alignments.** There is one abstraction which will make the edit distance easier to reason about, and this is the notion of finding an alignment. In particular, an alignment between $x$ and $y$ consists of a sequence of tuples $(a_1, b_1), \dots, (a_\ell, b_\ell)$ where $\ell \leq n + m$ with the following properties:

- For all tuples $i \in [\ell]$, we have that $a_i$ is a character in $x$ or $*$, and $b_I$ is a character in $y$ or $*$.
- If we read the first element of each tuple in order, then ignoring $*$'s, we see the string $x$, and if we read the second element of each tuple in order, then ignoring $*$'s, we see the string $y$.

Then, we have the following claim:

$$\mathrm{ed}(x, y) = \min_{\substack{(a_1,b_1),\dots,(a_\ell,b_\ell) \\ \text{alignment of } x, y}} \sum_{i=1}^{\ell} \mathbf{1}\{a_i \neq b_i\}.$$

So the edit distance is trying to find the alignment between the two strings which has the minimum disagreement. How would one prove this fact? Well, first one shows that given a sequence of edits from $x$ to $y$, one can construct an alignment of length which is equal to the number of edits, and secondly, one can go from an alignment to a sequence of edits. Roughly speaking, if there is a tuple whose end-points are different, this would correspond to a substitution, if there is a tuple $(*, b_i)$, this corresponds to an insertion of $b_i$ to go form $x$ to $y$, and $(a_i, *)$ a deletion of $a_i$ in $x$ to go from $x$ to $y$.

So, what should we be asking for a top-level guide for the edit distance? Here, recall that we want to ensure our sub-problems are smaller versions of the original problem, and there are hopefully not too many options for the possible answers (since we are exploring all of them).

**Top-Level Guide**: What is the first tuple in the alignment?

We have three potential possibilities:

- We either have $(x_1, y_1)$, $(x_1, *)$ and $(*, y_1)$.
- Each of these have cost 1, and if we take a particular option, we must then consider the alignment of the remaining sub-strings.

  - In the case of $(x_1, y_1)$, the remaining sub-problem consists of the best alignment between $x_2, \dots, x_n$ and $y_2, \dots, y_m$.
  - In the case of $(x_1, *)$, the remaining sub-problem is now the best alignment of $x_2, \dots, x_n$ with $y_1, \dots, y_n$.
  - Finally, we also have $(*, y_1)$ gives rise to the best alignment of $x_1, \dots, x_n$ with $y_2, \dots, y_n$.

Notice that the remaining sub-problems always consider alignments among strings which are suffixes of $x$ and $y$. In particular, in order to define the right notation, we ought to consider

$$G(i,j) = \text{edit distance (i.e., best alignment cost) of substrings } x_i, \dots, x_n \text{ and } y_j, \dots, y_m,$$

where it is useful to consider the case $i = n+1$ or $j = m+1$ as considering the empty sub-string of $x$ or $y$. Therefore, we can write the recurrence:

$$G(i,j) = \min \left\{ \begin{array}{c} \mathbf{1}\{x_i \neq y_i\} + G(i+1, j+1), \\ 1 + G(i, j+1), \\ 1 + G(i+1, j) \end{array} \right\}.$$

and the base cases become:

- $G(i,j) = 0$ if $i = n+1$ and $j = m+1$.
- $G(i,j) = m - j$ if $i = n+1$ and $j < m$.
- $G(i,j) = n - i$ if $j = m+1$ and $i < n$.

**Runtime.** There are $nm$ sub-problems, and each sub-problem takes $O(1)$ time to solve. This gives a running time of $O(nm)$ to compute the edit distance between two strings.

**Bottom-Up vs Top-Down Approach.** So far, we've approached these dynamic programming problems in a very "top-down" fashion. We asked a high level questions which was aimed to chip away at the problem and explore the space of possible solutions. Indeed, the type of algorithm we were executing was very much in line with "divide-and-conquer" algorithms by working recursively. It is sometimes the case that recursion is undesirable in practice because there are large constant factors involved in storing the program state. In these cases, there may be some benefit to algorithms which are "bottom-up." It is important to note that in the "bottom-up" approach, we can use the same recurrence and figure out which order to compute them so as to build our solution bottom-up. In this case, we know that we want to solve for the entries of a two-dimensional array, where the number of rows is $n+1$ and the number of columns is $m+1$:

- The right-most column is $n, n-1, \dots, 1, 0$.
- The bottom-most row is $m, m-1, \dots, 1, 0$.
- Finally, in order to compute $G(i,j)$, we need to compute $G(i+1, j+1)$, $G(i, j+1)$, and $G(i+1, j)$. In particular, if we traverse this backwards, we can always compute these without recursion.

## 5 Optimal Sub-structure

Some of you have already asked, or have been asking during the office hours how to prove the correctness of dynamic programming problems. In general, there is a trend to proving that *any* algorithm for an optimization problem is correct which proceeds in the following way:

- We are interested in proving correctness for an algorithm that, say, maximizes the value of $f(x)$ as $x$ varies in a solution space $\mathcal{S}$ with a set of constraints.
- In order to do this, we've devised a dynamic programming recurrence relation which iterates through a set of possible solutions and outputs some value after performing part of the search. The proof of the correctness of the algorithm then consists of two claims:
  - **Considering Optimal**. First, if we consider the optimal solution $x^*$ which maximizes the quantity, then the algorithm (at some point) considered searching through the solution $x^*$. This means that there were a sequence of actions that the algorithm performed that led to this particular solution. This means that, even though we do not know what $x^*$ is, we can assume it in our analysis and argue that our search eventually considered the options. An example to keep in mind is that of the optimal binary tree. The proof of this usually will proceed by induction (which is how we often analyze recurrence relations). This means that the quantity that you output has cost which is at least as large as $f(x^*)$ (since we kept the maximum and at some moment considered $x^*$.
  - **Output Corresponds to Something**. Whatever cost you do end up outputting, it is always the case that it does correspond to a true solution. This is sometimes easier to establish, by remembering the choices that led to specific expressions, since the overall cost that we output will usually be that of the object we output. Thus, since we are outputting something in the solution space satisfying the constraints, the value $f(x)$ can only be at most $f(x^*)$. Putting both together gives us the desired claim.

There is an approach which shortcuts this argument by arguing about a property called "optimal sub-structure." Which means the following:

> **Optimal Sub-structure**: The optimal solution is made up of combining optimal solutions.

Consider the case of the optimal binary search trees. If we can prove that the optimal binary search tree considers a choice of root and places the optimal binary search tree to the left and to the right, then we end up with an optimal binary search tree. This shows up when we consider the proof of the first claim: the true optimal binary tree $x^*$ has some node $i^*$ at the root, and our algorithm will consider a case where $i^*$ is the root. Now, we consider that case, and we can say that, by induction, we find the optimal binary search tree on $[1, i^* - 1]$ and $[i^* + 1, n]$ on the right—however, we technically need to say that the combination of the optimal binary search trees on the left and the right is the optimal way to combine them.

## 5.1 Two Problems

Consider the following two computational problems, one of which will have "optimal sub-structure," and one which will not have optimal sub-structure. In both problems, we receive as input a graph $G$ where the vertices are labeled by $[n]$ and there are a set of edges.

- **Shortest Path Problem**: We receive as input two indices $i$ and $j$, and the goal is to find the shortest path between $i$ and $j$, where the length of the path is the number of edges in the path.
- **Longest Path Problem**: We receive as input two indices $i$ and $j$, and the goal is to find the longest simple path between $i$ and $j$, where "simple" means that it goes through each vertex at most once.

The thing to notice is that, if we break up the problem by finding the paths which go through a vertex $\ell$, then the shortest path problem does have optimal sub-structure, because we have the following claim.

**Claim 5.1.** *The shortest path from $i$ to $j$ which goes through vertex $\ell$ consists of the shortest path between $i$ to $\ell$, and union with the shortest path from $\ell$ to $j$.*

However, the longest path problem does not exhibit this same "optimal sub-structure." The longest path between $i$ to $j$ may not be the union of the longest path from $i$ to $\ell$ and then the longest path from $\ell$ to $j$—indeed, the longest path from $i$ to $\ell$ may share a vertex with the path from $\ell$ to $j$, so taking the union wouldn't even make sense. Thus, we do not believe that there exists a dynamic programming solution for the longest path problem, and we don't even believe there is any *polynomial-time* algorithm.