CSCI 270: Introduction to Algorithms and the Theory of Computing

Jiapeng Zhang

Fall 2025

Welcome to CSCI 270: Introduction to Algorithms and the Theory of Computing. Here's some course information:

- Here's the `course website` and `piazza`.
- Lectures are held on Mondays and Wednesdays in LVL 17.
- Homeworks are not mandatory, but they're strongly recommended.
- We'll be using *Algorithm Design*, by Kleinberg and Tardos.
- These notes were taken by Julian and have **not been carefully proofread**. They're sure to contain some typos and omissions, due to Julian.

Contents

§1 Monday, August 25

Some quick housekeeping: both homework and attendance are not required, but they are strongly recommended. For more detailed information, see the course webpage or piazza.

§1.1 What is an algorithm?

Now onto the real stuff: What is an **algorithm**? Here's what Wikipedia says:

> *In mathematics and computer science, an algorithm is a finite sequence of mathematically rigorous instructions, typically used to solve a class of specific problems or to perform a computation.*

That's a pretty good definition but a mouthful. Here's the rough definition that we'll use.

Definition 1.1 — An **algorithm** is a precise sequence of instructions for solving a computational problem. It receives a particular instance of the computational problem as input, and outputs the solution to that instance of the problem.

When designing an algorithm, there are two things we're primarily concerned with:

1. *Correctness:* Is the algorithm producing the desired output?
2. *Efficiency:* How expensive is the algorithm in terms of time, memory, etc.?

Let's warm up with an example.

Example 1.2 (Triangular numbers)

Consider the following computational problem. One receives as input an integer n, and the desired output is the value of $1 + 2 + \ldots + n$. What algorithm should we use to solve this problem? One option is to outright compute $1 + 2 + \ldots + n$. That's certainly a correct algorithm, but it is going to take $n - 1$ many addition operations.

On the other hand, if you remember the formula for triangular numbers, then you know that another, better algorithm is to return $n(n+1)/2$. This requires a total of only 3 operations! (One addition, one multiplication, and one division.) To prove the correctness of this faster algorithm, we would need to formally prove that $n(n+1)/2 = 1 + \ldots + n$ for all natural numbers $n \in \mathbb{N}$. This can be done using **induction**, for instance.

Let's do another example.

Example 1.3 (Sorting)

Here's another computational problem: One receives as input an integer n and a list of numbers $[a_1, \ldots a_n]$, and the desired output is the same list of numbers in sorted order, i.e., $[a_{\sigma(1)}, \ldots, a_{\sigma(n)}]$ for some permutation $\sigma\colon [n] \to [n]$.[a] How should we solve this?

One naive option is to iterate through the list and find the smallest element, then iterate through it and find the second-smallest element, and so on. Once you're done, you know exactly where each element should be placed in the sorted version of the list. How long will this take? Well, it will require n many linear scans over the entire list, each of which takes n units of time. (I.e., 1 unit of time to look at each entry.) So the overall runtime is about $n \cdot n = n^2$.

A better solution is the **quicksort** algorithm, which proceeds as follows: select a "pivot" element from the list, partition the list into two lists according to whether each element is larger or smaller than the pivot, and recur this procedure on the newly created lists. The "base case" is when one of the lists has size 0 or 1, in which case it's already trivially sorted. It's not hard to see that this algorithm is correct, from this recursive structure. (Exercise: prove it!) What is its runtime? It depends on how the pivot is selected at each step. Let's assume for now that the pivot is magically equal to the median of the array at each step. In this case, the runtime T of the algorithm will be described by the following recurrence relation,

$$T(n) = \underbrace{n}_{\text{Divide by pivot}} + \overbrace{2 \cdot T(n/2)}^{\text{Recursive calls}} .$$

We're not going to prove it right now, but this recurrence relation yields a runtime of $O(n \log(n))$, which is a great improvement on our previous sorting algorithm! (Note that this assumes our algorithm can "magically" compute medians of arrays, though.)

[a] Here we abbreviate $[n] := \{1, \ldots, n\}$.

Hopefully these examples give you a bit of a flavor for algorithmic thinking. One of the key takeaways is that there are plenty of interesting and sophisticated ways of saving time on computational tasks!

Some of the topics we'll cover in class include:

- Divide and conquer,
- Dynamic programming,
- Greedy algorithms,
- Max-Flow/Min-Cut,
- Reductions,
- NP-hardness,
- Computability theory.

See you next time!

§2 Wednesday, August 27

§2.1 Divide and Conquer

We'll be discussing our first general algorithmic principle today: **divide and conquer**. We'll try to convince you that it's a simple but very powerful tool for solving algorithmic problems! At a high level, divide and conquer has three steps (not just two!):

1. **Divide the problem into subproblems:** This can take the form of dividing your array into subarrays, dividing your graph into subgraphs, etc.

2. **Conquer each subproblem separately:** Call your divide-and-conquer approach on each of the subproblems!

  - Crucially, this recursion must conclude with a *base case*, i.e., a subproblem that is so small it can be solved directly, without further subdivision.

3. **Combine the solutions of the subproblems:** Extract the solution to the original problem from the solutions to each subproblem.
   - Key point: In order for divide and conquer to be a fruitful approach, it must be the case that solutions to the subproblems can "easily" yield the solution to the original problem.

§2.1.1 MergeSort

Let's make this concrete with a divide-and-conquer-based sorting algorithm: **MergeSort**. It has the following pseudocode.

```
def MergeSort(A):
    if len(A) == 1:
        return A
    r = len(A) // 2
    left_sorted = MergeSort(A[:r])
    right_sorted = MergeSort(A[r:])
    return Merge(left_sorted, right_sorted)
```

So, the rough idea is to split the array down the middle, sort each subarray separately using a recursive call, and then merge them (using `Merge`). But how is `Merge` implemented? How can we merge two sorted lists into a single sorted list? (Exercise: think about it yourself!)

It turns out we can do it relatively easily, as follows.[1]

```
def Merge(A, B):
    l, r = len(A), len(B)
    C = [None] * (l + r) # Initialize empty array
    i, j = 0, 0
    for k in range(l + r):
        if (i < l) and (A[i] <= B[j]):
            C[k] = A[i]
            i += 1
        else:
            C[k] = B[j]
            j += 1
    return C
```

The key idea is simple: if `A` and `B` are both sorted arrays, then the smallest element between each of them is either `A[0]` or `B[0]`. We can directly compare these elements,

[1] Strictly speaking, the below implementation is not exactly correct. There's a bit more logic needed for when A or B's entries have been fully popped off. (But hopefully you get the idea.)

place whichever one is smaller as the first entry of `C`, and continue this process until we're done. Let's try to formalize this a bit.

Theorem 2.1

MergeSort(A) returns a sorted array with the same elements as A.

Proof sketch. By induction on n, the length of A. When $n = 1$, the claim is immediate. Assuming correctness of MergeSort for arrays of length $\leq k$, then an array of length $k + 1$ will correctly have its subarrays sorted by the recursive call to MergeSort (by the inductive hypothesis), and will then be correctly merged by Merge (by correctness of Merge). □

What about the runtime? On the one hand, `MergeSort` will make $\log(n)$ many layers of recursive calls, where n denotes the length of A. This is because each layer of recursion halves the size of the array, and the recursion ends when the array has length 1. On the other hand, the runtime of each layer of `Merge` calls is linear in n, because it makes 1 or 2 comparisons before adding each element to C, and the elements in the different `Merge` calls are disjoint. This means the overall runtime of `MergeSort` is $O(n \log(n))$, so MergeSort runs in **loglinear** time.

One way to formalize this is to build a recurrence relation for the runtime of MergeSort, which takes the form[2]

$$T(n) = 2 \cdot T(n/2) + n.$$

Note that $2 \cdot T(n/2)$ refers to each of the recursive calls of MergeSort, while n denotes the cost of merging. We're not going to prove it, but it turns out that there's a **Master Theorem** for deducing runtimes from recurrence relations.

Theorem 2.2 (Master Theorem)

Let $a \geq 1$, $b \geq 1$, and let $T(n)$ follow the recurrence relation $T(1) = \Theta(1)$ and $T(n) = a \cdot T(n/b) + f(n)$.

- If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = O(n^{\log_b a} \cdot \log(n))$.
- If there exists an $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = O(n^{\log_b a})$.

§2.1.2 Binary search

Let's briefly recall **binary search**. The underlying problem is that one receives as input a sorted array A and a value b, and the desired output is whether $b \in A$ or not. If A were *not* sorted, then we would have no option but to check every entry of A; this is sometimes referred to as *linear search* or *exhaustive search*.

When A is sorted, however, we can do better. We first compare the middle entry of A, i.e., $A[n/2]$ with b. If they are equal, we're done. If $A[n/2] < b$ then we know that all entries in $A[: n/2]$ are also less than b, owing to the fact that A is sorted. So we can remove half of the entries in A from contention. By the same token, if $A[n/2] > b$, then we can remove the top half of A from contention. In either case, we restrict focus to the surviving half of A and repeat our process, i.e., by checking either $A[n/4]$ or $A[3n/4]$. This results in logarithmic time search for sorted arrays, rather than linear time for unsorted arrays!

[2] As always, we let $T(n)$ denote the worst-case runtime of the algorithm on inputs of size n.

§2.1.3 String matching

Here's the problem: You receive a (very long) string S, after which you can do some preprocessing to generate a data structure. You then receive many strings A, and for each you must determine whether A is a substring of S. For those A which are a substring of S, you must also determine whether they occur.

Remark 2.3. This problem (and various versions of it) has real applications in genomics!

So how should we solve this problem? One approach is to enumerate all the substrings of S, sort them, and then perform binary search with each candidate A. That will make the computation for each candidate A very fast (logarithmic time), but the preprocessing will take quadratic time (because there are n^2 many substrings of S), which is pretty slow.

A better approach — though not obvious! — is to enumerate all the **suffixes** of S, sort them, and then search for each candidate A among these sorted suffixes. Here's the key point: if A appears as a substring of S, then it will be a **prefix** of one of these suffixes, which will guide our binary search. We'll talk more about this next time, see you then!

§3 Wednesday, September 3

§3.1 String matching II

Last time we talked about the string matching problem: You receive a (long) string S, you are permitted to perform preprocessing on it, and then you receive many candidate strings A, for which you must determine whether A is a substring of S.

We briefly began discussing an efficient solution for this problem last night, which referenced prefixes and suffixes. Let's go ahead and formalize those concepts now.

Definition 3.1 — A string A is a **prefix** of a string S if there exists a string U such that $S = AU$. Similarly, A is a **suffix** of S if there exists U such that $S = UA$.[a]

[a] UA and AU denote concatenation of strings, of course.

Example 3.2

Let S be an arbitrary string of length n. How many prefixes does it have? Answer: $n + 1$, stretching from the empty prefix to the entire string S.

Let's also formalize what a substring is.

Definition 3.3 — A string A is a **substring** of S if there exist strings L, R such that $S = LAR$.

Our algorithm from last time made use of the following lemma, which we will now prove.

Lemma 3.4

For any string S, a string A is a substring of S if and only if A is a prefix of a suffix of S.

Proof. First suppose that A is a substring of S. Then $S = LAR$ for some strings L and R. Then A is a prefix of AR, which is a suffix of S. Conversely, suppose A is a prefix of S', a suffix of S. Then $S = LS'$ for some string L, by the definition of suffix, and $S' = AR$ for some string R, by the definition of prefix. Combining the previous equalities, we have that $S = LS' = LAR$, implying A is a substring of S. □

Now recall our previous idea for solving the string matching problem:

1. Enumerate all suffixes of S.
2. Sort them.
3. For each candidate string A, perform binary search to determine whether there exists a suffix S' of S for which A is a prefix.

That sounds like a reasonable plan, but it's a bit imprecise. How exactly will we sort the suffixes of S? And how will we perform binary search to determine whether a candidate A is a prefix of a suffix? Let's spell out those details now.

Definition 3.5 — Let $A = \{a_1, \dots, a_k\}$ be an alphabet equipped with a total ordering. (I.e., $a_i < a_j$ or $a_i > a_j$ for each $i \neq j$.) The **dictionary order** (or *lexicographic order*) over finite strings with characters in A is as follows, for distinct strings S and T:

- If S is a prefix of T, then $S < T$. (And vice versa if T is a prefix of S.)
- Otherwise, let i be the first entry in which S and T disagree, i.e., $S[i] \neq T[i]$. Then $S < T$ if $S[i] < T$, and $S > T$ otherwise.

So, after fixing some arbitrary ordering of our underlying alphabet, we'll use the dictionary order to sort the suffixes of S. After sorting all these suffixes, how do we use binary search to check for prefixes? The key idea is that the dictionary order is based upon looking at prefixes of strings, so it has the right structure to look for prefixes.

More formally, consider a version of binary sort which, given a candidate string A and all sorted suffixes of S, searches for the *least* suffix S' which is $\geq A$ (under the dictionary order). If A appears as a substring of S, then A must be a prefix of S'. (Can you see why?) Conversely, if A is not a prefix of this S', then it does not appear as a substring anywhere in S.

So what is the runtime of this algorithm? Letting $n = |S|$, we need $O(n \log n)$ many comparisons to sort the suffixes of S, and $O(\log n)$ many comparisons to binary search a given candidate A. But how long does a given comparison take? Up to $\Theta(n)$ time in the worst case, if $\Omega(n)$ entries must be checked to compare the two strings in question. (I.e., if they're both long strings and they share a fairly long prefix.) This leads to an overall runtime of $O(n^2 \log n)$, which is not great.

But that's the *worst-case* runtime of our algorithm, which can be overly pessimistic. For an application like genome analysis, it often turns out to be the case that a comparison can be completed within a constant number of operations, about 50. If we were permitted to use this assumption, the runtime of our algorithm would lower to $O(50 \cdot n \log n) = O(n \log n)$, which is much better! This general idea is important to keep in mind: worst-case analysis is often too pessimistic, and algorithms can perform considerably better on real-world data than on worst-case data.

§3.2 Improving Binary Search

Now let's switch gears and talk about how to improve the binary search algorithm. In particular, how can we improve binary search if we have extra memory, i.e., RAM?

It turns out that we can leverage this extra memory using a simple lookup table. To this end, let Σ be our underlying alphabet, so that all strings S we consider are strings over Σ, i.e., $S \in \Sigma^*$.[3] We will use the added memory by computing the following lookup tables L_1 and L_2 in advance, for the largest k that our memory allows.

> **Definition 3.6** (Lookup Table) — For fixed $k \in \mathbb{N}$, and for each $B \in \Sigma^k$, let $L_1[B]$ be the smallest i such that $(R_i)_{1,\ldots,k} = B$, and let $L_2[B]$ be the largest j such that $(R_j)_{1,\ldots,k} = B$.

The intuition here is simple: for each prefix B, we're storing the "starting point" and "ending point" for binary searching B in the array, i.e., the smallest and largest entries with B as a prefix.

We're running out of time to discuss the consequences of the lookup table (as well as another strategy for improving binary search), so we'll pick this up next time and put some of it on the homework!

§4 Monday, September 8

Let's continue practicing the divide and conquer approach.

§4.1 Polynomial Multiplication

Problem 4.1 (Polynomial Multiplication)**.** You receive as input two degree n polynomials, $f(x) = \sum_{i=0}^{n} a_i x^i$ and $g(x) = \sum_{i=0}^{n} b_i x^i$, and you must compute $f(x) \cdot g(x)$.

How should we solve this? The straightforward, essentially brute-force approach is to simply multiply out all terms. So we would multiply $g(x) = \sum_{i=0}^{n} b_i x^i$ by a_0, then by $a_1 x$, then by $a_2 x^2$, etc. At the end, we would add up all the resulting polynomials we have (i.e., $n+1$ of them), which is just a matter of combining coefficients.

What is the running time of this simple algorithm? It's $\Theta(n^2)$, because each multiplication between a monomial and a degree n polynomial takes $\Theta(n)$ time — for the $n+1$ multiplications between the monomial and each term in the polynomial — and we will be performing this multiplication $n+1$ many times. The final step, in which we add together the resulting $n+1$ polynomials, also takes $\Theta(n^2)$ time, meaning the overall runtime is $\Theta(n^2)$.

So our algorithm is pretty slow. Can we do better, using a divide and conquer approach? Here's one simple observation: we can choose to represent the polynomials $f(x) = \sum_{i=0}^{n} a_i x^i$ and $g(x) = \sum_{i=0}^{n} b_i x^i$ as arrays

$$(a_n, a_{n-1}, \ldots, a_0), \qquad (b_n, b_{n-1}, \ldots, b_0).$$

We can then slice those arrays down the middle, which amounts to decomposing f (and similarly g) into the polynomials

$$f_0(x) = \sum_{i=0}^{n/2} a_i\, x^i, \qquad f_1(x) = \sum_{i=n/2+1}^{n} a_i\, x^i = x^{n/2+1} \sum_{i=n/2+1}^{n} a_i\, x^{i-n/2-1}.$$

[3] When Σ is an alphabet (i.e., a finite set), it is common to use Σ^* to denote the set of all finite strings over Σ. That is, $\Sigma^* = \bigcup_{i\in\mathbb{N}} \Sigma^i$.

Can you see how this leads to a divide and conquer algorithm for polynomial multiplication? (Hint: FOIL.) Here's the general idea:

```
def Multiplication(A, B):
    # Assume len(A) == len(B)
    n = len(A)
    r = n / 2

    # Divide
    f1 = A[:r]
    f0 = A[r:]
    g0 = B[:r]
    g1 = B[r:]

    # Conquer
    h0 = Multiplication(f0, g0)
    h1 = Multiplication(g0, g1)
    h2 = Multiplication(f1, g0)
    h3 = Multiplication(f1, g1)

    # Merge
    ans = x ** (2 * (r + 1)) * h3 + x ** (r + 1) * (h2 + h1) + h0
    return ans
```

In short, we're writing $f(x) = f_0(x) + x^{r+1} \cdot f_1(x)$ and likewise for $g(x)$. Then we're observing that

$$\begin{aligned} f(x) \cdot g(x) &= \big(f_0(x) + x^{r+1} \cdot f_1(x)\big) \cdot \big(g_0(x) + x^{r+1} \cdot g_1(x)\big) \\ &= x^{2(r+1)}(f_1 \cdot g_1) + x^{r+1}(f_1 \cdot g_0 + f_0 \cdot g_1) + f_0 \cdot g_0, \end{aligned}$$

which agrees with our `return` statement in the definition of `Multiplication()`. So we have a divide and conquer algorithm, which we now see is correct. What is its runtime?

Our algorithm's worst-case runtime has the recurrence

$$T(n) = 4T(n/2) + O(n),$$

which by the master theorem give us a guarantee of only $T(n) = O(n^2)$. That doesn't improve upon our runtime from earlier!

However, we can lower the number of recursive calls that we make to our solution (i.e., the coefficient on $T(n/2)$ in the recurrence) by leveraging the follow key observation:

$$f_0 \cdot g_1 + f_1 \cdot g_0 = (f_0 + f_1)(g_0 + g_1) - f_0 \cdot g_0 - f_1 \cdot g_1.$$

Why is this useful? Because it tells us that we can compute the left hand side without actually multiplying $f_0 \cdot g_1$ or $f_1 \cdot g_0$! Instead, we can leverage 3 other multiplications we have already performed. In total, this will reduce our number of recursive calls from 4 to 3 – which will make a big difference on the algorithm's runtime!

The new algorithm is as follows.

```python
def Multiplication(A, B):
    n = len(A) # Assume len(A) == len(B)
    r = n / 2

    f1 = A[:r]
    f0 = A[r:]
    g0 = B[:r]
    g1 = B[r:]

    h0 = Multiplication(f0, g0)
    h1 = Multiplication(f0 + f1, g0 + g1)
    h2 = Multiplication(f1, g1)

    ans = x ** (2 * (r + 1)) * h2 + x ** (r + 1) * (h1 - h2 - h0) + h0
    return ans
```

The correctness of this new algorithm follows from our "key observation" and correctness of the previous algorithm. Furthermore, the runtime of our new algorithm is governed by the following recurrence relation

$$T(n) = 3T(n/2) + O(n).$$

The master theorem then yields an upper bound of

$$T(n) = O(n^{1.585}),$$

where $1.585 \approx \log_2(3)$.

Remark 4.2. This is an example of a divide and conquer algorithm where the "merge" step (i.e., combining the smaller solutions into a large solution) is very clever! In general, there can be very tricky and non-obvious ideas for designing fast algorithms. Don't feel bad if they sometimes seem like magic – these algorithms can take years of hard work to discover!

§5 Wednesday, September 10

§5.1 Closest Pair of Points

Let's work on a new problem today!

Problem 5.1 (Closest Pairs)**.** Given a list of points $[x_1, \dots, x_n] \subseteq \mathbb{R}^2$, with each point represented as $x_i = (a_i, b_i) \in \mathbb{R}^2$, return the closest pair of points (x_i, x_j) as measured by Euclidean distance $d(x_i, x_j)$.

How should we solve this problem? The brute force solution is to simply compute the distances between all pairs of points and keep track of the minimum distance we've seen so far. That will take $\Theta(n^2)$ time. Can we do better, perhaps using a divide and conquer strategy?

Let's build intuition using the case in which the points are in $\mathbb{R}\,(= \mathbb{R}^1)$, rather than $\mathbb{R}^2$. How could we solve the closest pairs problem in $\mathbb{R}$ using a divide and conquer approach? Here's an idea:

1. Sort all points by their only coordinate, in time $O(n \log n)$.

2. Divide the points into the largest half of points, B, and the smallest half of points, A. (I.e., those which are larger than the median and those which are smaller.)

3. Recursively find the closest pair of points in each of A and B.

4. Let $a_{\max}$ be the largest point in A and $b_{\min}$ be the smallest point in B.

5. Return the closest pair among the following 3 candidates: the closest pair in A, the closest pair in B, and $(a_{\max}, b_{\min})$.

Why is this algorithm correct? Because if (u, v) is an arbitrary pair of points in our array, then either u and v are both in A, u and v are both in B, or one is in A and one is in B. Our recursive calls handle the first two cases, and our explicit comparison between $a_{\max}$ and $b_{\min}$ handles the final case. (Can you see why?)

What is the runtime of this algorithm? It is governed by the recurrence relation

$$T(n) = 2T(n/2) + O(1),$$

which leads to a runtime of $T(n) = O(n)$.[4] (Strictly speaking, $O(n \log n)$ due to our original sorting step, which we only have to perform once.)

Now how can we solve the more general case, over $\mathbb{R}^2$? The tricky part here is implementing the "merge" step. If we split our points according to a vertical line, say $y = c$, how can we find the closest pair of points between the left and right sides? Well, let A consist of those points that are to the left of our vertical line and B those points to the right. From the divide and conquer approach, we can assume that we have solutions for each of A and B.

Let's suppose that B yielded a smaller solution than A, and denote by δ the solution we have discovered for B, i.e., the distance between the closest pair of points in B. We would now like to determine whether there is a pair $(a, b) \in A \times B$ such that $d(a, b) < \delta$. If such a pair exists, then certainly $a[0] \geq c - \delta$, otherwise a is automatically at least δ far from any point in B. (Because it is δ far from the border separating A and B.) Likewise, it must be that $b[0] \leq c + \delta$. So we can restrict focus to those points whose x-coordinate is in $[c - \delta, c + \delta]$.

Now let's fix a particular candidate point $a \in A$ with $a[0] \geq y - c$. We'd like to determine if it can be matched to a $b \in B$ with $d(a, b) < 0$. Then, by identical reasoning as above, we can restrict focus to those points $b \in B$ with $b[1] \in [a[1] - \delta, a[1] + \delta]$. Geometrically, we're saying that if a does have a partner b, then the y-coordinate of b must be within distance δ of the y-coordinate of x.

Now come two important observations:

1. The number of points $b \in B$ with $b[0] \in [c, c + \delta]$ and $b[1] \in [a[1] - \delta, a[1] + \delta]$ is constant. This is owing to the fact that the corresponding region of space can be covered using finitely many square tiles of side length $\delta/2$. (In particular, 8 many such tiles.) If any tile were to contain 2 points in B, then these points would be within distance $< \delta$ of each other, producing contradiction with the fact that the closest pair of points in B has distance δ.

[4] This is because the recursion tree is a binary tree of depth $\log_2(n)$. There are $O(n)$ many leaves, each of which contribute a constant amount of computation from the base case, and $O(n)$ many non-leaves, each of which contribute a constant amount of work from each "merge" operation (i.e., the $O(1)$ in the recurrence relation).

2. Once we've restricted focus to points $b \in B$ with $b[0] \in [c, c+\delta]$, we can further narrow down to those points with $b[1] \in [a[1]-\delta, a[1]+\delta]$ in $\log(n)$ time, by first sorting all points in B according to their y-coordinate and then using binary search.

As a result, our final algorithm will be able to perform "merge" in $O(n \log n)$ time as follows.

1. Sort all points $b \in B$ with $b[0] \in [c, c+\delta]$ according to their y-coordinate, in $O(n \log n)$ time.
2. For each point $a \in A$ with $a[0] \in [c, c-\delta]$, first restrict focus to those points $b \in B$ with $b[1] \in [a[1] \pm \delta]$, and then compute the distances between a and each such point.
   - The former step takes time $O(\log n)$ for each fixed candidate $a \in A$, by using binary search, and the latter step takes constant time, as there are only a constant number of points in b. Thus the overall runtime in $O(n \log n)$.
3. Return the minimum of δ and the smallest distance encountered in Step (2.).

So we can indeed perform "merge" in loglinear time![5] This yields a recurrence of

$$T(n) = 2T(n/2) + O(n \log n),$$

which leads to an upper bound of $T(n) = O(n \log^2 n)$. Much better than $\Theta(n^2)$!

§5.2 Summarizing Divide and Conquer

Let's recap the general blueprint for divide and conquer algorithms:

1. Partition the problem into several subproblems. (Divide)
2. Recursively solve the subproblems. (Conquer)
3. Merge the solutions of the subproblems to extract the solution to the original problem. Making this efficient usually requires a smart way for reusing the computation from the subproblems!

Proving correctness of divide and conquer algorithms usually relies upon an inductive argument, where the only nontrivial part is proving correctness of the "merge" step. Proving runtime properties usually follows from establishing a recurrence relation and appealing to the Master Theorem.

Quick reminder: the first midterm is on September 22, in about two weeks, and will use problems from the first two homework problems. Divide and conquer will definitely be an important concept on the exam!

§6 Monday, September 15

We're going to start a new topic today: dynamic programming!

[5] Also note that we can find the boundary $y = c$ yielding equal-sized sets A and B in time $O(n \log n)$ by, for instance, sorting all points according to their x-coordinate and indexing into the median. This will in fact be a one-time cost, as the sets A and B yielded by this procedure are prefixes/suffixes of our sorted array and thus already sorted.

§6.1 Dynamic Programming

§6.1.1 Fibonacci Numbers

Let's begin with an example: Imagine that we'd like to compute the Fibonacci numbers, which are defined by $F[0] = 0$, $F[1] = 1$, and $F[i] = F[i-1] + F[i-2]$ for $i \geq 2$. How should we compute them? Here's a natural starting point.

```
def Fib(n):
    if n in [0, 1]:
        return n
    return Fib(n - 1) + Fib(n - 2)
```

Is this a good implementation? No, we're wasting lots of computation! We're computing `Fib(n - 2)` on the final line of the program, but `Fib(n - 1)` will also lead to another computation of `Fib(n - 2)` under the hood. This means we're unnecessarily computing `Fib(n - 2)` twice! And the issue is more severe for smaller inputs; it's not too hard to see that the recursive structure of this algorithm will lead to an exponentially large computation graph. That means an exponential runtime of $2^{O(n)}$ – pretty terrible!

How can we do better? One way is by using an iterative approach that prevents us from repeatedly computing the same quantity.

```
def Fib(n):
    F[0] = 0
    F[1] = 1
    for i in range(2, n + 1):
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

The runtime of our second implementation is simply $O(n)$, as we're performing constant work inside of a loop which repeats $O(n)$ times. That's a great improvement over $2^{O(n)}$!

§6.1.2 Weighted Interval Scheduling

Let's try a harder problem now.

Problem 6.1 (Weighted Interval Scheduling)**.** Receive as input n tasks, described by a starting time s_i, finish time f_i, and weight w_i for each task i. Output a set S of non-overlapping tasks maximizing the total weight $\sum_{i \in S} w_i$.

How should we solve this? One idea is to use a **greedy algorithm**, where we accept the first task that lands on our desk while we are unoccupied. But it's not too hard to see that this algorithm is not correct. In particular, imagine an input consisting of two overlapping tasks, the first of which has low weight and the second of which has high weight; a greedy algorithm will select the wrong one.

A better solution is to keep track of the tradeoffs between different tasks. Here's an idea.

```
def soln(S):
    A[0] = 0
    for t = 1 to tmax:
        A[t] = A[t - 1]
        for all i with f[i] = t:
            A[t] = max(A[t], w[i] + A[s[i] - 1])
    return A[tmax]
```

The intuition is that `A[t]` stores the optimal strategy for completing tasks before time t. Given this value, `A[t + 1]` will be equal to the maximum of `A[t]` and `w[i] + A[s[i] - 1]` across all tasks i with `f[i] = t + 1`. This corresponds to the choice of either completing a task that finishes at the current time `t + 1`, or not completing such a task.

§7 Wednesday, September 17

Quick reminder: next Monday we have our first midterm in class! You will complete the problems on paper in class, and upload photos of your solutions on Gradescope afterwards (before 3:30 p.m.).

§7.1 Dynamic Programming II

Let's continue our discussion of dynamic programming, this time with a new problem.

Problem 7.1 (Coin Exchange)**.** Receive as input a collection of n coins with values $p_1, \dots, p_n$, and a number t. Return whether or not the value t can be expressed using a combination of coins with values in $\{p_i\}_{i \le n}$. Each coin can be used at most once.

How should we solve this problem? If we were permitted to use coins more than once, then the following lemma would be very useful.

Lemma 7.2

Let $\{p_i\}_{i \le n}$ be a collection of coin values, and let $t \ge 1$ be a value. Then t can be expressed using a combination of coins with values in $\{p_i\}_{i \le n}$ (allowing repetition) if and only if the same is true for $t - p_i$ for some $i \le n$.

Proof. The reverse direction is immediate: If $t - p_i$ can be expressed using a collection of coins C, then t can be expressed using $C \sqcup \{p_i\}$. Conversely, suppose that t can be expressed using a collection of coins C. As $t \ge 1$, it must be that C is nonempty. Let p_i be a coin in C. Then $t - p_i$ can be expressed by the collection $C \setminus \{p_i\}$. □

In particular, Lemma 7.2 gives rise to the following solution, if we *are* permitted to reuse coins.

```
def soln(C, t):
    A = [False] * t
    A[0] = True
    for i in range(1, t + 1):
        for coin in C:
            if i - coin >=0 and A[i - coin]:
                A[i] = True
    return A[t]
```

Here's the intuition: we march from the value 1 to the value t, and for each one we check whether $t - p_i$ can be expressed for some coin value p_i. By Lemma 7.2, this algorithm is correct and will ultimately populate `A[t]` with the truth value of whether t can be expressed using such coins.

Unfortunately, however, we are *not* permitted to use a given coin multiple times. In order to keep track of this, let's add more power to our array `A`. Above, `A` is a 1-dimensional array whose ith entry tracks whether the value i can be expressed using coins a nonnegative number of times. Let us now define `A` to be a 2-dimensional array whose (i, j)th entry tracks whether the value i can be expressed using only the first j coins, where each coin is used at most once.

Populating the first row of `A` will be pretty easy; we simply need to check whether any coins have value exactly 1. More interestingly, how can we populate an arbitrary entry (i, j) of `A`, assuming that "earlier" entries of `A` have already been populated? (I.e., entries (i', j') with $i' < i$ or $i' = i$ and $j' < j$.)

Here's the key observation: $A[i, j] = 1$ if and only if $A[i, j - 1] = 1$ or $A[j - p_i, i - 1] = 1$ for some p_i. (Can you see why?) The resulting pseudocode is as follows.

```
def soln(C, t):
    n = len(C)
    A = [[False] * n for _ in range(t + 1)]

    # 0 is always expressible
    for j in range(n):
        A[0, j] = True

    # Populate table using key observation/recurrence
    for j in range(n):
        for i in range(1, t + 1):
            if i >= C[j]:
                A[i, j] = A[i, j] or A[i, j - 1] or A[i - C[j], j - 1]
    return A[t, n]
```

And that's it! To recap:

1. We defined an array `A` which stores solutions to subproblems.

2. We derived a recurrence that allows us to populate the entries of `A` using "earlier" entries of `A`.

3. We iterated through `A`, populating all its entries and recovering the solution to the original problem.

Index