HSSP Algorithms Course Notes

Satya Holla

July 2023

These notes are meant to be read in order, and are intended for high school students with a strong technical background, but relatively little experience in algorithms or discrete math. Sections with an asterisk are optional.

Contents

1 Preliminaries: 07/09/2023 **3**
1.1 Logistics . . . 3
1.2 Introduction . . . 3
1.3 Problems and algorithms . . . 3
1.4 Efficiency and asymptotic notation . . . 4
1.4.1 Asymptotic notation . . . 5
1.5 Examples of algorithms . . . 5
1.6 Model of computation . . . 5
1.7 Example . . . 6
1.8 Exercises . . . 7

2 Searching: 07/16/2023 **9**
2.1 Motivation and outline . . . 9
2.2 Searching . . . 9
2.2.1 *Lists and arrays . . . 10
2.2.2 Linear search . . . 10
2.2.3 Binary search . . . 11
2.3 Exercises . . . 13

3 Review and sorting: 07/23/2023 **14**
3.1 Review, introducing anonymous feedback form . . . 14
3.2 Sorting . . . 14
3.2.1 Insertion sort . . . 14
3.3 Exercises . . . 16

4 Divide-and-conquer, mergesort: 07/30/2023 **17**
4.1 Divide-and-conquer . . . 17
4.2 Mergesort . . . 17
4.3 *Lower bounds: the comparison model . . . 19
4.4 Exercises . . . 21

5 Trees and graphs: 08/06/2023 **22**
5.1 Motivation . . . 22
5.2 Warm-up: linked lists . . . 22
5.3 Trees . . . 23
5.4 Graphs . . . 25
5.5 Exercises . . . 27

6 Breadth-first search, hashing **28**
6.1 Shortest paths: BFS 28
6.2 Karatsuba algorithm for multiplication 30
6.3 Exercises 32

7 Important notation **33**
7.1 Logical notation 33
7.2 Sets 34
7.3 Asymptotic notation 35

8 Resources **36**

1 Preliminaries: 07/09/2023

1.1 Logistics

- Instructor: Satya Holla (he/him)
- Class time: Sunday 2:00PM - 3:00PM
- Email `C15664s1-teachers@esp.mit.edu` with any questions about the class.

1.2 Introduction

The purpose of this class is to give students an appreciation for algorithmic thinking, as well as the knowledge of the basic ideas behind a variety of algorithms. By the end of the class, students will:

- have a working model of computation to frame algorithmic thinking
- understand the use of asymptotic notation in analyzing algorithm efficiency
- understand basic search and sort algorithms, as well as their uses
- know how basic data structures work
- understand the concept of a graph, as well as basic graph algorithms

Another major purpose of the class is to be **fun** - this should not feel like taking a course for a grade. That being said, you shouldn't treat this class like watching a YouTube video for fun. Try to **interact** with the material as much as you can. Although we will never ask for solutions, problems may be given after lectures, and will be found in these notes. Team problem solving sessions may take place during the Zoom portion of the class. In both cases, your attitude should be to try your best without worrying too much about actually getting the answers. Remember: the material is supposed to be challenging, so don't be discouraged if you are confused. Feel free to email the address mentioned above if you feel stuck on anything.

1.3 Problems and algorithms

We can now begin by providing precise definitions of some basic concepts. Keep in mind that what we present in this section are not universally accepted terminology or definitions, but capture the essence of what we are trying to do.

Definition 1. *A* problem *is a function which maps each input to a set of (correct) outputs.*

This is a pretty abstract definition which can be more easily understood by an example. Consider the problem P_{sort} of sorting an input list of numbers from least to greatest. Then, we can write out one value of the function:

$$P_{\text{sort}}([3, 2, 4, 1]) = \{[1, 2, 3, 4]\}$$

In this example, the problem *input* is the list $[3, 2, 4, 1]$, and the set of correct *outputs* corresponding to this input is the set consisting of one element, which is $[1, 2, 3, 4]$. We will cover sorting in greater depth later in the course.

However, in general the set of outputs may have more than one element. For example, consider the problem P_{invsqr} of outputting a value which, when squared, equals the input. Then, one value of P_{invsqr} is given by:

$$P_{\text{invsqr}}(4) = \{-2, 2\}$$

The next point of interest is the set of inputs which is the domain of a problem. In general, problems can have unrestricted domains, but for the purpose of algorithm design, we care about problems whose domain is:

- sufficiently general

- includes inputs of unbounded size

For example, a problem we don't care about much in algorithm development is the problem of sorting lists of 100 numbers (in which the domain is all lists of size 100). Instead, we care about the problem of sorting *any* list, whose size is unspecified (and is usually denoted by some parameter, like n). The reason for wanting inputs of unbounded size is subtle, and will be properly explained in Section 1.4: basically, we care about how well algorithms perform in the limiting case of the input size growing very large. Before we can explain this, however, we need to have a good definition of an algorithm:

Definition 2. *An* algorithm $\mathcal{A}$ *is a procedure which maps any valid input to a single*[1] *output. We say that* $\mathcal{A}(i)$ *is the output of algorithm* $\mathcal{A}$ *on input* i*.*

This definition is a bit more fuzzy than the one earlier. For one, what is a "procedure"? Intuitively, a procedure is a set of commands which, when applied to the input, will yield the desired output. These commands can include loop statements (e.g., while, for), conditional statements (if), and generally end with a return statement, which returns the desired output. This raises the question, what constitutes a command? The answer to this is that it really just depends on context. When communicating the details of a complicated algorithm, we might simply say "sort this list" as a step in the algorithm, without detailing each step involved in the sort. This is generally fine, since the intended audience will likely know how to sort numbers. However, if we used this primitive in designing a sorting algorithm itself, it would be confusing and nonsensical.

As for the other potentially confusing term "valid input", descriptions of algorithms generally specify the set of inputs which they consider valid. For example, for a (number) sorting algorithm, a list of fruits might not be a valid input. We finish with a definition that relates the concept of algorithm and problem.

Definition 3. *An algorithm* $\mathcal{A}$ solves *a problem* P *if its output is always correct. That is,* $\mathcal{A}(i) \in P(i)$ *for any input* i*.*

1.4 Efficiency and asymptotic notation

It is generally quite easy to design *correct* algorithms to problems. Suppose, for example, that we wanted to design an algorithm that outputted the square root of the input. Then, one strategy would be to pick a random number x smaller than the input and compute x^2. We return x if x^2 equals the input, and otherwise pick another random number and continue. Such an algorithm technically solves the problem - whenever it outputs a value, that value is the desired square root. However, it doesn't work very efficiently. By picking random numbers, you essentially have no chance of finding the correct square root. What we want are *efficient* algorithms, and for the purpose of this class, this means *fast* algorithms.

How should we characterize speed, though? One solution is to run the algorithm on an input and time it, and compare this time to other algorithms. This is a reasonable idea, but there are some issues to smooth out. Suppose we tried this for the problem of sorting lists.

1. How do we deal with the fact that different computers have different speeds? If I run a sorting algorithm on a 22-core supercomputer, and run a different algorithm on my calculator, it would be hard to compare them.

    - Solution: Instead of actually measuring the time (which in general is pretty noisy), let's instead count the number of "basic instructions" (more on this in Section 1.6) used by the algorithm. That way, we don't have to worry about which machine it is being run on. This being said, we will still use the term "runtime" to refer to this measure, even if it isn't strictly a time.

2. How should we decide which input size to time it on? Should we test it on inputs of size $n = 100$, or $n = 1000$? You can imagine that some algorithms will be better on smaller inputs, but worse on larger inputs, than others.

[1] This is not completely true - to be precise, such algorithms which, given an input, always produce the same output, are called *deterministic*. For now, ignore this distinction.

- Solution: Instead of trying it for one input, we can instead find the "runtime" (i.e., number of operations) $T(n)$ as a function of the input size n.

3. Even once you choose an input size (say $n = 100$), you still have to choose an actual list of size 100. There are a variety of choices for this (e.g., one choice is a list which is already sorted, and another is a list in reverse order), and depending on the choice, certain algorithms can be better or worse.

    - Solution: Generally, we care about *worst-case* performance. That is, we pick the input which makes the algorithm take the longest time. The intuition behind this is that we don't really know what the input distribution will be - we can't really consider it to be random most of the time, so it's better to be pessimistic.

1.4.1 Asymptotic notation

Great! Now, we have a pretty decent idea of how to compare algorithms. However, algorithm designers use one extra thing to make their lives easier: **asymptotic notation**. The two main principles behind asymptotic notation are:

- We don't care about low order terms. If the runtime is $T(n) = n^3 + n + \log n + 45$, we say that $T(n) = O(n^3)$, which means that, ignoring low order terms, the runtime is at most n^3.
- We don't care about constant factors. If the runtime was instead $T(n) = 35n^3 + n + \log n + 45$, we can *still* say that the runtime is $O(n^3)$.

Formally, for two functions $f(n)$ and $g(n)$, we say $f(n)$ is in $O(g(n))$ (also denoted $f(n) \in O(g(n))$), if

$$f(n) < c \cdot g(n)$$

for all $n > N$, for some constants c and N.

1.5 Examples of algorithms

Algorithms are ubiquitous in everything we do in computing, and even sometimes in other regimes. Here is a short list of some cool applications of algorithms, some of which will be covered in the course.

- Karatsuba multiplication algorithm: `https://en.wikipedia.org/wiki/Karatsuba_algorithm`
- Shortest paths: `https://en.wikipedia.org/wiki/Shortest_path_problem`
- Algorithms for cryptography: there are many of these, but one example is `https://en.wikipedia.org/wiki/RSA_(cryptosystem)`

1.6 Model of computation

The previous discussion of efficiency (see the first point raised in Section 1.4) assumed that we can count "basic instructions", to come up with a runtime function $T(n)$. However, to do this, we need to know what basic instructions are allowed, and for that we need a model. The model we use for this class is the **Word RAM model**. The details aren't too important, but the main takeaways are:

- basic arithmetic operations like adding, multiplication, subtraction, and division can be done in $O(1)$ time (in other words, they all take a constant amount of time)
- you can read or write to a given memory location in $O(1)$ time

This second bullet is an operation called indirection is important: it's what allows you to store lists as numbers in contiguous memory, and be able to access any element of the list in $O(1)$ time, given the index of the element (for example, given a list [1,2,3,4,...,n] in memory, you can easily access the 1st, 2nd, 200th, or generally kth element in $O(1)$ time.

1.7 Example

The following example shows the power of asymptotic analysis in determining efficiency. Two sorting algorithms are presented, insertion sort and quicksort[2]. They are both run on the same input, for lists of size 1, 10, 100, 1,000, 10,000, and 100,000. The resulting runtimes are shown below. Here is the program, written in Python:

```
import random
import time

def insertion_sort(L):
    n = len(L)
    for first_unsorted_idx in range(1, n):
        first_unsorted_val = L[first_unsorted_idx]
        i = first_unsorted_idx - 1
        while i >= 0 and first_unsorted_val < L[i]:
            L[i + 1] = L[i]
            i -= 1
        L[i+1] = first_unsorted_val
    return

def quick_sort(L, start=0, end=None):
    n = len(L)
    if end is None: end = n
    if end - start <= 1: return
    pivot_idx = random.randrange(start, end)
    pivot = L[pivot_idx]
    left = start
    right = end - 1
    while left < right:
        while left < n and L[left] <= pivot:
            left += 1
        while right > -1 and L[right] > pivot:
            right -= 1
        if left < right:
            L[left], L[right] = L[right], L[left]
    quick_sort(L, start, left)
    quick_sort(L, left, end)

if __name__ == '__main__':
    for size in (1,10,100,1000,10000,100000):
        example = [random.uniform(0, 1) for i in range(size)]
        test = (example, example.copy())
        t1 = time.time()
        insertion_sort(test[0])
        t2 = time.time()
        quick_sort(test[1])
        t3 = time.time()
        assert test[0] == test[1]
        print(f'size: {size}')
        print(f'\tinsertion sort ran in {t2 - t1}')
        print(f'\tquicksort ran in {t3 - t2}')
        print(f'speedup using quicksort: {(t2 - t1)/(t3 - t2)}\n')
```

[2]My quicksort algorithm only works when the values of the list are all distinct - it is more annoying to write otherwise, but isn't fundamentally different.

The output is below:

```
size: 1
        insertion sort ran in 2.1457672119140625e-06
        quicksort ran in 1.6689300537109375e-06
speedup using quicksort: 1.2857142857142858

size: 10
        insertion sort ran in 9.059906005859375e-06
        quicksort ran in 3.409385681152344e-05
speedup using quicksort: 0.26573426573426573

size: 100
        insertion sort ran in 0.0004200935363769531
        quicksort ran in 0.0004181861877441406
speedup using quicksort: 1.0045610034207526

size: 1000
        insertion sort ran in 0.04390883445739746
        quicksort ran in 0.004621028900146484
speedup using quicksort: 9.501960581983283

size: 10000
        insertion sort ran in 3.8098928928375244
        quicksort ran in 0.04733395576477051
speedup using quicksort: 80.48963648360726

size: 100000
        insertion sort ran in 385.8964948654175
        quicksort ran in 0.506472110748291
speedup using quicksort: 761.930393946612
```

Don't worry about the details of this analysis, but insertion sort is an $O(n^2)$ algorithm, whereas quicksort is an $O(n \log n)$ algorithm. It is clear that quicksort becomes much much better than insertion sort for large values of n, which is in line with the fact that $n \log n$ is asymptotically smaller than n^2. In fact, we can see that the improvement ratio grows by almost 10, each time we multiply the list size by 10, which matches what we would expect.

1.8 Exercises

The following exercises might help in gaining familiarity with asymptotic notation. The exercises make use of the following new asymptotic notation symbol, Θ, defined so that

$$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \text{ and } g(n) \in O(f(n))$$

(Read the symbol $\iff$ as "if and only if" - it's also commonly written as "iff"). In other words, Θ corresponds to some constant-factor-ignoring notion of equality[3]. For example, $5n^3 + n \in \Theta(n^3)$, but $5n^2 \notin \Theta(n^3)$.

Now, for the following pairs of functions f and g, answer the following:

A Is $f \in O(g)$?

B Is $g \in O(f)$?

C Is $f \in \Theta(g)$? (note that if this is true, then A and B are also true)

D Are f and g incomparable? (i.e., none of the above)

[3]Formally, we say that Θ is an *equivalence relation*. See the textbook 6.042 OCW in Section 8: Resources

When unspecified, take $\log n$ to mean $\log_2 n$. These exercises should prove quite quite challenging, and you may have to do your own research to answer some of these.

1. $f(n) = \log_3 n$, $g(n) = \log_2(n)$
2. $f(n) = 3n^3$, $g(n) = 4n^3$
3. $f(n) = 15n^2 + 2/n$, $g(n) = 15n \log n + n^{1.5}$
4. $f(n) = 15n^2 + 2/n$, $g(n) = 15n \log^5 n + n^{1.5}$ *Hint*[4]
5. $f(n) = 2^n$, $g(n) = 3^n$
6. $f(n) = |n \sin(n)|$, $g(n) = n$ (for integer n)
7. $f(n) = n^{\log n}$, $g(n) = (\log n)^n$
8. $f(n) = n^{\log n}$, $g(n) = 2^n$
9. $f(n) = \log(n!)$, $g(n) = n \log n$ *Hint*[5]
10. $f(n) = \log(n^2!)$, $g(n) = n^2 \log n$
11. $f(n) = \log n$, $g(n) = 4 \log(n^3)$

I will try to release solutions to each set of exercises after the next class, so try your best to work on them! Once again, they are completely optional.

[4]The notation $\log^5 n$ just means the same thing as $(\log n)^5$

[5]The notation $n!$ refers to the factorial function. See the textbook 6.042 OCW in Section 8: Resources

2 Searching: 07/16/2023

2.1 Motivation and outline

Searching and sorting are, for good reason, the standard first two topics of most algorithms courses. They are two of the most common problems in computing, and subroutines involving searching and sorting appear in many of the algorithms you will see later on. Furthermore, they serve as excellent examples of the power of algorithms involving recursion (although the binary search shown is phrased iteratively). In the following sections, we start with the problem of searching, then sorting, and describe algorithms for each. We conclude with an optional (marked with an asterisk) subsection on lower bounds for sorting.

Note: due to time constraints, only searching was covered in lecture, so I will move the sorting material to Section 3.

2.2 Searching

The searching problem is the following:

$$P_{\text{search}}(L, x) = \begin{cases} \{i \mid L[i] = x\} & \text{if } x \in L \\ \{\text{NULL}\} & \text{else} \end{cases}$$

Since this is a bit abstract, let's try to decipher it. The problem has two inputs, a list L and a value x. The set of correct outputs is the set of all *indices* i such that the i-th value in L is equal to x. If x does not appear in the list L, there is only one correct output, the special symbol NULL. Let's look at a few examples:

$$P_{\text{search}}([3, 5, 2, 7, 8], 2) = \{2\}$$
$$P_{\text{search}}([3, 5, 2, 7, 5], 5) = \{1, 4\}$$
$$P_{\text{search}}([3, 5, 2, 7, 8], 3) = \{0\}$$
$$P_{\text{search}}([3, 5, 2, 7, 8], 1) = \{\text{NULL}\}$$

You might have noticed something a bit strange - in the examples, all of the numbers seem to be one value smaller than they should be! This is because, for the purposes of this class, we will be using **zero-indexing**[6], which means the first index is considered to be zero, not one. That's why, in the first example above, we say that the index of 2 in the list $[3, 5, 2, 7, 8]$ is 2, not 3.

It might not be immediately clear why this would be a useful problem to know how to solve. Consider the following scenario:

Scenario 1. *Suppose you are a software engineer at social media company BaceFook. Currently, all of your users' data consists of two fields: a username and name (not the best social media app, but oh well). You have three users, and you want to store their data in a list like so:*

$$U = [\{\textit{user: hsolla, name: Hatya}\}, \{\textit{user: lsolla, name: Ladhana.}\}, \{\textit{user: zmuck, name: Zark}\}]$$

Your desired functionality is: anyone should be able to search up a user's username and find their name.

How should you code your desired functionality? You can't directly look up the user's name if all you know is their username, given the structure of the list. The solution here is to use a search algorithm. Suppose you are trying to look up the name of the user "lsolla". First, you find the index, in U, of the user "lsolla" (in this case, the index is 1). Next, you look up the user $U[1]$ (this can be done in $O(1)$ operations), and finally return the name field of this user (in this case, "Ladhana"). This use case is quite common in managing databases, and is part of the reason why searching is so important.

This raises an interesting point, though - how are you able to look up $U[1]$ in $O(1)$ operations? We return to the Word RAM model (see Section 1.6) for an explanation.

[6]The reason I chose to use zero-indexing is because most programming languages use it, since it makes more sense in the framework of computer architecture (more about that in Section 2.2.1).

2.2.1 *Lists and arrays

This rather short section brings together some vaguely discussed ideas from before, to explain the following fact:

Fact 1. *In the Word RAM model, for any (properly-implemented) list A and index i, if $A[i]$ exists, then it can be read from or written to in $O(1)$ time.*

If you are willing to believe this fact, you can go ahead and skip this section - it is kind of subtle and not very relevant for the rest of the course. The key lies in the distinction between *lists* and *arrays*.

Definition 4. *A* list[7] *is an ordered collection of elements. Lists of size n support the following operations:*

- *init: Initializes all values of the list $L(i)$ to* NULL*, for $i = 0$ to $i = n - 1$. This operation is used only once, to create a list.*
- *get(i): an operation which returns the value $L(i)$, for list L, if $0 \leq i \leq n - 1$.*
- *set(i, x): an operation which sets $L(i)$ to the value x (denoted $L(i) \leftarrow i$). Future "gets" of the index i now return x. Like get, this operation is only defined when $0 \leq i \leq n - 1$.*

This definition is pretty abstract, for good reason. Lists are an example of what is called an **interface**. Interfaces are a powerful concept in programming - they describe some structure by the operations the structure should support. In this case, the structure of a list is defined by the three operations "init", "get" and "set" - these operations entirely define what a list is. Nowhere in the definition of list does it describe how you should actually *store* or *implement* a list. That is on purpose - as a programmer, you can choose how you want to implement the interface. One option is to store pairs of the form $(i, L(i))$. Another option is to use an array.

Definition 5. *An* array *is a data structure which supports the list interface. In an array, all of the data is stored contiguously in blocks (called* words*), where each block stores one element, and the blocks store the elements* in order *(that is, the i-th block in the contiguous chunk stores the i-th element of the list). The following list operations are implemented as follows on list L:*

- *init: A chunk of contiguous memory of size n blocks is allocated, and in $O(n)$ time, each element is set to* NULL*. The starting memory location of the chunk of memory is kept in the variable* memstart*.*
- *get: To read the i-th element's value, we calculate the memory address of $L[i]$:*

$$\text{addr} = \text{memstart} + i * b$$

  Then, we retrieve the b bits of memory starting at addr*. This all takes $O(1)$ time in the Word RAM model.*
- *set: Set can be implemented using the same idea as get. First, we calculate the memory address we want to write to, then we write to the address in $O(1)$ time.*

A depiction of an array is shown in Figure 1.

For the remainder of the course, we will assume that we can index into lists in $O(1)$ time, and retrieve and change values accordingly.

2.2.2 Linear search

The problem of searching has a very obvious solution. You can work your way from the start to the end of the list, and check each element to see whether it equals the desired value. If you find the desired value, return the index, and otherwise, return NULL. The pseudocode for this algorithm is below.

[7] This is specifically a *static* list, but the distinction is not important as of now.

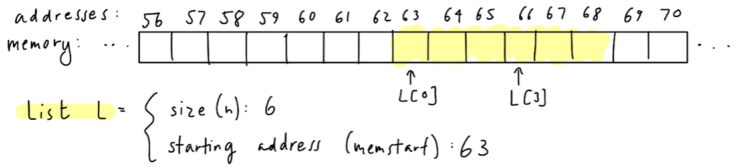


Figure 1: An array used to implement list L

Algorithm 1 Linear Search

Input: list L of size n, value x
Output: output $\in P_{\text{search}}(L, x)$ - here, i either satisfies $L[i] = x$, or $i = \text{NULL}$, and $x \notin L$

```
i ← 0
while i < n do
    if L[i] = x then
        return i
    end if
end while
return NULL
```

The linear search algorithm is quite simple, and runs in $O(n)$ time, which is optimal (since the entire input list L has to be read at least once, to see if x is in it). Notice that in the algorithm, we look through all $L[0], L[1], ..., L[n-1]$, but we don't examine $L[n]$. This is because L is a list of size n, and we are zero-indexing, so $L[n-1]$ is actually the last element.

There isn't much more to say about linear search - next, we analyze a much more interesting algorithm, binary search.

2.2.3 Binary search

Linear search is technically an (asymptotically) optimal search algorithm, as discussed before, but it doesn't really capture the idea behind Scenario 1 well. In reality, when people focus on the searching problem, they will try to **preprocess** the list by sorting it in ascending order, so that searching becomes much easier. To be precise, we define what this means:

Definition 6. *List L is sorted in ascending order if, for all $0 \leq i < j < n$, $L[i] \leq L[j]$.*

A classic example is dictionaries, which are alphabetically sorted. When you search for a word in the dictionary, you *don't* use linear search. Instead, if you're looking for the word "binary", you might flip to a random page (say page 113), observe the word "epsilon" as the first word on the page, then continue your search on the first 112 pages, since dictionaries are in alphabetical order, and "e" comes after "b". This very intuitive concept defines the **binary search algorithm**, whose pseudocode is given below as Algorithm 2.

Let's try to prove the correctness of binary search. Taking a look at the pseudocode, the algorithm starts off by creating a "left" and "right" pointer. The key idea behind the correctness of the algorithm is the maintenance of the following property, which we will call our **invariant**:

Property 1 (loop invariant)**.** *If element* x *is in list* L*, then it must be in the sublist*

$$[L[\text{left}], L[\text{left}+1], ..., L[\text{right}-1]]$$

We need to show that this invariant is true, which we will do in two steps, which formally make up an inductive argument (although we won't go into that):

1. Show that it is true initially
2. Show that it stays true each time we go through an iteration of the **while** loop

Algorithm 2 Binary Search

Input: sorted (in ascending order) list L of size n, value x
Output: output $\in P_{\text{search}}(L, x)$

```
1: left ← 0
2: right ← n
3: while right > left do
4:     mid ← ⌊(left + right)/2⌋
5:     if L[mid] = x then
6:         return mid
7:     else if L[mid] > x then
8:         right ← mid
9:     else
10:        left ← mid + 1
11:    end if
12: end while
13: return NULL
```

Clearly, the invariant starts off true, since "left" starts off initialized to 0, and "right" starts off initialized to n, so $[L[\text{left}], L[\text{left} + 1], ..., L[\text{right - 1}]]$ is just the entire list L. At any point during the execution of the algorithm, let's call the sublist defined above the "active sublist", since the rest of the list doesn't matter for our search. Let's see how the invariant *stays* true by examining the **while** loop on line 3 more closely.

The **while** loop corresponds to the act of (roughly) splitting the list in two, comparing the middle element with x, and possibly continuing the search on a smaller sublist. To see why this works, first, notice from the definition that "mid" will always be in the active sublist (if you're confused, notice that "mid" is always at least as large as "left"). This value of "mid" is recomputed on line 4 each iteration of the loop. Next, let's consider the three cases of the comparison step:

- Case 1 (line 5): $L[\text{mid}] = x$ - clearly, we can just return "mid" and be done in this case.
- Case 2 (line 7): $L[\text{mid}] > x$ - in this case, since the list is sorted, we know that the elements $L[\text{mid}], L[\text{mid} + 1], ..., L[\text{right} - 1]$ of the active sublist are *all* too large. Therefore, we can ignore all of them, and make the new active sublist

  $$L[\text{left}], L[\text{left} + 1], ..., L[\text{mid} - 1]$$

  which is accomplished by setting right ← mid.
- Case 3 (line 9): $L[\text{mid}] < x$ - this is similar to Case 2, except now, we want to make the new active sublist

  $$L[\text{mid} + 1], L[\text{mid} + 2], ..., L[\text{right} - 1]$$

  which is accomplished by setting left ← mid + 1

In all of the cases, the invariant is maintained. To prove the correctness of our algorithm, we only need one more step, which is to show the correctness of the stopping condition of the **while** loop. This is made much simpler with the loop invariant. Our stopping condition (line 3) is if right $\leq$ left, in which case we return NULL. This is completely in line with our invariant - if right $\leq$ left, then our active sublist is empty! Remember, the active sublist is:

$$[L[\text{left}], L[\text{left} + 1], ..., L[\text{right} - 1]]$$

It's a little hard to see what happens if right $\leq$ left using the notation above - let's instead use the equivalent notation that the active sublist is the list

$$[L[i] \mid \text{left} \leq i < \text{right}]$$

(read the vertical bar $\mid$ as "such that"), from which it becomes clear that the size of the active sublist is (right $-$ left), so if right $\leq$ left, then the active sublist is empty. In this case, our invariant

tells us that x is *not in the list* L (since, if it were, it would be in the active sublist, but the active sublist is empty). To recap, the algorithm has two **return** statements, both of which are guaranteed to return the correct value:

- The first **return** statement is on line 6, which clearly will only return a correct value
- The second **return** statement is on line 13, and as discussed, will only return NULL if the active sublist is empty, which by the invariant we proved means that x is not in L.

Of course, correctness is not everything. All we have shown is that, when the algorithm returns a value, it is a correct output for the problem. But it's also important to show that the algorithm will terminate, and terminate quickly - after all, creating an infinite loop gives you an algorithm which is always technically correct, but isn't useful. To prove termination, note the following property:

Property 2 (runtime)**.** *Each iteration of the* ***while*** *loop, the value* $(right - left)$ *will decrease by at least half. In other words, the new value will be at most half the old value. As a result,* $(right - left)$ *also decreases by at least* 1 *each iteration, since fractional values of* $(right - left)$ *are impossible. The proof of this property is left as an exercise.*

Why is this useful? Note that the stopping condition of the **while** loop is when right $\leq$ left, or in other words, when right $-$ left ≤ 0. So if (right $-$ left) decreases by at least half each iteration of the **while** loop, eventually it will become ≤ 0! This shows that the loop is only executed a finite number of times - specifically, it shows that the loop only executes at most $\log n$ times. This leads to the following theorem:

Theorem 1. *Binary search runs in* $O(\log n)$ *time on a sorted list of size* n*.*

Proof. A (slightly informal) proof of Theorem 1 is that, from Property 2, the **while** loop executes at most $O(\log n)$ times. Each iteration of the while loop involves a finite number of $O(1)$ runtime operations, so each loop runs in $O(1)$ time. Therefore, the total runtime is $O(1 \cdot \log n + 1)$, where the final 1 comes from the final return statement. But this is just $O(\log n)$. □

Most of the time, we will not analyze algorithms as formally or in-depth as in this section, but it is good to see such analyses at least a few times. The key takeaway from this section is the use of properties like Properties 1 and 2 to prove claims about correctness and runtime. Such properties can generally be proven by **mathematical induction** - more about this can be found in Chapter 5 of the 6.042 OCW textbook, linked in Section 8: Resources.

2.3 Exercises

This exercises give some practice with the material, especially binary search. The final two exercises are especially challenging.

1. Prove Proposition 2. *Hint: prove it for each of the three cases, similar to the proof for Proposition 1 - you may want to prove separately for when* $(right - left)$ *is even or odd.*
2. Suppose you have an unsorted list of size n, and you want to search for a value in the list k times. For what values of k is it better to sort the list before performing the searches? Assume you can sort the list in $O(n \log n)$ time, and give your answer in asymptotic notation.
3. Suppose you have a list of values $L = [d, d+1, ..., d+s-1]$, for some known d. A malicious virus on your computer somehow erases some subset of the values from the list, leaving you with a new list L'. Describe an $O(\log n)$ algorithm to find the first missing element (keep in mind that this may be the first element of L). For example, if your original list was $[8, 9, 10, 11, 12, 13]$, the virus could modify it to be $[8, 9, 11, 13]$, in which case you should output 10. *Hint: think about how to make the problem similar to binary search*
4. Suppose you have two sorted lists A and B, one of size n and another of size m. Devise an algorithm to find the median of the concatenation of A and B, in $\Theta(m+n)$ time. (For example, if $A = [1, 2, 2, 3, 4, 5]$ and $B = [2, 5, 6, 7, 7]$, we want to find the median of the set of numbers $A + B = [1, 2, 2, 3, 4, 5, 2, 5, 6, 7, 7]$. Sorting $A + B$ gives us $[1, 2, 2, 2, 3, 4, 5, 5, 6, 7, 7]$, and the median is 4.) *Hint: think about how to make the problem similar to binary search*

3 Review and sorting: 07/23/2023

3.1 Review, introducing anonymous feedback form

Use this anonymous feedback form: `https://forms.gle/okivERNkJXhaRbm16` to ask questions or give feedback about the class. A large part of today's class was focused on reviewing binary search, from last lecture.

3.2 Sorting

After learning about binary search, one question you may have is: how do we get our lists to be sorted in ascending order? We mentioned earlier that it often makes sense to preprocess a list by sorting, so that future searches are "cheap" (i.e., fast), but we have not yet actually talked much about the sorting problem. Let us define it here:

$$P_{\text{sort}}(L) = \{L^*\}, \text{ where } L^* \text{ is sorted in ascending order, and contains the same elements as } L$$

To be precise, L^* must not only contain the same elements, but each element in L should appear the *same number of times* in L^*. Some examples are given below:

$$P_{\text{sort}}([4,7,3,6]) = \{[3,4,6,7]\}$$

$$P_{\text{sort}}([]) = \{[]\}$$

$$P_{\text{sort}}([4]) = \{[4]\}$$

$$P_{\text{sort}}([4,7,7,3,6,6,6]) = \{[3,4,6,6,6,7,7]\}$$

Notice that the empty list, and any list with only one element, is sorted by default. Let's look at some sorting algorithms.

3.2.1 Insertion sort

Here is the pseudocode for insertion sort, which is essentially the same code as in the example in Section 1.7.

Algorithm 3 Insertion Sort

Input: list L of size n with comparable elements
Output: output $\in P_{\text{sort}}(L)$

```
1: for firstUnsortedIndex from 1 to n − 1 do   ▷ this for loop includes the 1, but not the n − 1
2:     firstUnsortedValue ← L[firstUnsortedIndex]
3:     placeInSortedList ← firstUnsortedIndex − 1
4:     while placeInSortedList ≥ 0 and firstUnsortedValue < L[placeInSortedList] do
5:         L[placeInSortedList + 1] ← L[placeInSortedList]
6:         placeInSortedList ← placeInSortedList − 1
7:     end while
8:     L[placeInSortedList + 1] ← firstUnsortedValue
9: end for
10: return L
```

Rather than giving an in-depth analysis of the correctness of insertion sort, I will instead give a brief overview of the algorithm. The algorithm works on the following principle: we keep a sublist of sorted elements, and each iteration of the outer **for** loop, a new element of our original list is added to the sorted sublist. Initially, the sorted sublist only contains one element (the element at index 0). Going through an iteration of the **for** loop, we first find the index of the first element *not* part of the sorted sublist, then add that element to the sorted sublist by finding the correct spot for it. This index is the value "firstUnsortedIndex" in the pseudocode. To add the element at index "firstUnsortedIndex" to our sorted sublist, we search for the correct spot, which is what happens in the **while** loop on line 4 - we search from the end of the sorted sublist down to the beginning,

and stop once we find an element smaller (or equal) to our "firstUnsortedValue" (which is just the element at index "firstUnsortedIndex". A formal proof of correctness is left as an exercise.

Aside from correctness, we should know the runtime of insertion sort, in big Oh (or big Theta) notation.

Theorem 2. *Insertion sort runs in $\Theta(n^2)$ time on a list of size n.*

Proof. Remember: to analyze the runtime, we count the number of operations done in the worst case. Let's start with the inner **while** loop. How many instructions does this **while** loop take? Each iteration of the **while** loop, we do a constant number of operations (we change the value of L[placeInSortedList], and we change the value of placeInSortedList, each of which is an $\Theta(1)$ operation). Therefore, to count the number of operations in the entire **while** loop, we just have to count the number of times the while loop runs! But, if we remember, this **while** loop takes the first "unsorted" element, and finds where to insert it in the sorted sublist. In the worst case, we will have to search all the way from the end of the sorted sublist to the beginning, and place the first unsorted element in the beginning - this will take Θ(firstUnsortedIndex) operations. Now that we have this part of the puzzle, we can look at the outer **for** loop. Each run of this **for** loop involves some constant time operations (lines 2, 3, and 8) and one run of the **while** loop (which, as discussed, takes Θ(firstUnsortedIndex) time). We run the **for** loop $n-1$ times. Therefore, the runtime of the entire algorithm is:

$$\sum_{\text{firstUnsortedIndex}=1}^{n-1} \Theta(1) + \Theta(\text{firstUnsortedIndex})^{8}$$

To simplify this, let's use i instead of "firstUnsortedIndex", and let's also ignore the $\Theta(1)$ term (notice that "firstUnsortedIndex" will always be at least $\Theta(1)$), so our summation is now:

$$\sum_{i=1}^{n-1} \Theta(i) = \Theta\left(\sum_{i=1}^{n-1} i\right)$$

where in this step, we use the fact that we are ignoring constants with big Theta notation. Let's prove one direction of this. Since we are ignoring constants, we know that we are taking the sum of many $\Theta(i)$ terms, which are all less (asymptotically) than $i \cdot c_i$ for some c_i, for $i = 1, 2, ..., n-1$. If we take $C = \max c_i$, we also have that each $\Theta(i)$ term is less (asymptotically) than $i \cdot C$, so we can move the constant C outside of the summation. In other words:

$$\sum_{i=1}^{n-1} \Theta(i) \leq_{\text{asymptotically}} \sum_{i=1}^{n-1} i \cdot c_i \leq \sum_{i=1}^{n-1} i \cdot C = C \cdot \sum_{i=1}^{n-1} i \in O\left(\sum_{i=1}^{n-1} i\right)$$

We similarly can show a lower bound using the same technique. Next, we have to evaluate the sum

$$\sum_{i=1}^{n-1} i$$

We can solve it exactly, since this sum is a pretty classic sum with a nice closed form, but I'll leave that as an exercise. Instead, I will use this as an opportunity to show a clever technique for attacking these types of problems, when we only care about the asymptotic result. First, notice that

$$\sum_{i=1}^{n-1} i \leq \sum_{i=1}^{n} n = n^2$$

Next, notice that

$$\sum_{i=1}^{n-1} i \geq \sum_{i=n/2}^{n-1} \frac{n}{2} = \frac{n}{2}\left(\frac{n}{2} - 1\right) = \frac{n^2}{4} - \frac{n}{2}$$

Putting these two together, we have that our runtime is $\Theta(n^2)$, since both $\frac{n^2}{4} - \frac{n}{2}$ and n^2 are $\Theta(n^2)$. □

[8] This isn't technically very correct notation, but it's common enough that I will use it anyway. In reality, you don't really add big Thetas, but instead add the functions in the relevant complexity classes - the math works out the same way, though.

3.3 Exercises

1. Selection sort is another sorting algorithm that works as follows: start by finding the smallest element of the list, and place it at the beginning (index 0). Next, find the second smallest element, and place it at index 1. Continue this process until the list is sorted.

   Write pseudocode like that in the notes for selection sort, and analyze its runtime.

2. Prove that insertion sort is correct. *Hint: Prove the following loop invariant.*

   Property 3 (Loop invariant for insertion sort)**.** *The "sorted sublist"*

   $$[L[0], L[1]..., L[\text{firstUnsortedIndex} - 1]]$$

   is sorted at the start of each ***for*** *loop, and contains the same elements as it did at the start of the algorithm.*

3. Solve exactly for the sum

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

   in a nice closed form (i.e., without summations). *Bonus: find a way to generate the closed form for*

   $$\sum_{i=1}^{n} i^k$$

   for any integer k. Hint: there are many ways to do this - one way is to use telescoping series, and another way is to prove that the answer will be a polynomial of some degree.

4 Divide-and-conquer, mergesort: 07/30/2023

4.1 Divide-and-conquer

Divide-and-conquer is a widespread algorithm design paradigm which is very useful in solving a variety of problems, including sorting, matrix multiplication, convex hull, etc. In this section, we explore mergesort, which is usually the first divide-and-conquer algorithm taught in algorithms courses. Before that, we need to explain what the paradigm is.

Definition 7. *A **divide-and-conquer** algorithm $\mathcal{A}$ consist of the following three steps:*

1. *The input to the algorithm is split into K subproblems (for some constant K)*
2. *Each of the K subproblems is recursively solved (using the same algorithm $\mathcal{A}$)*
3. *The solutions to the subproblems are somehow aggregated to form the solution to the original problem*

The outline for a divide-and-conquer algorithm given above mentions solving the subproblems "recursively". Recursion is a difficult concept to understand at first - when designing recursive algorithms, something that might help is to start by assuming you have a working algorithm that solves the problem for any input that is *smaller* than the current input. Then, you can use your own algorithm as a subroutine. For example, suppose you are designing an algorithm to generate the n-th Fibonacci number (see Algorithm 4 below[9]). For now, forget lines 1-5 of the algorithm, and focus on line 6. On line 6, to compute the n-th Fibonacci number, we return the sum of the $(n-1)$-th and $(n-2)$-th Fibonacci number. To compute the $(n-1)$-th and $(n-2)$-th Fibonacci numbers, *we are using the same function "Fibonacci" which the pseudocode describes.* However, since we are using it on "smaller" input (in this case, $n-1$ and $n-2$ are smaller than n), we don't run into any issues. Note that recursion doesn't generally work unless each recursive call is done on smaller input.

Now, let's take a look at lines 1-5. Whenever designing recursive algorithms, it's important that the recursion doesn't continue forever - otherwise, the algorithm will never terminate. In other words, when the input gets "small enough", the algorithm should just compute the solution normally. In the case of the Fibonacci algorithm, if the input is $n = 1$ or $n = 2$, then the algorithm will just return the correct value. The inputs $n = 1$ and $n = 2$ are called the **base cases**, and are solved without recursion. Since recursion is really more of a topic for a programming class, we will end our discussion with this, and move on to designing divide-and-conquer algorithms

Algorithm 4 Fibonacci

Input: A positive integer n
Output: The n-th Fibonacci number

```
1: if n = 1 then                                          ▷ base case
2:     return 0
3: else if n = 2 then                                     ▷ second base case
4:     return 1
5: else
6:     return Fibonacci(n − 1) + Fibonacci(n − 2)
7: end if
```

4.2 Mergesort

Now, we can move on to mergesort, a classic example of a divide-and-conquer algorithm. The idea behind mergesort is quite simple. Given an unsorted array L, the first step is to divide L into two (roughly) equal halves, called L_{left} and L_{right}. Next, we recursively sort L_{left} and L_{right}, again

[9]Note that this algorithm is actually quite bad for generating the Fibonacci sequence - in fact, there is a closed form for the Fibonacci numbers, which is much faster. Also, the given algorithm is not a divide-and-conquer algorithm.

using mergesort. Finally, we can *merge* the now-sorted L_{left} and L_{right} to form our desired result. The pseudocode is shown as Algorithm 5 below. Notice the base case on line 1: if the size of the array we are sorting is ever smaller than 2, we can just immediately return them, as such arrays are already sorted by default (as discussed previously).

Algorithm 5 Mergesort

Input: list L of size n with comparable elements
Output: output $\in P_{\text{sort}}(L)$
1: **if** $n \leq 1$ **then**
2: **return** L
3: **end if**
4: mid $\leftarrow \lfloor n/2 \rfloor$
5: $L_{\text{left}} \leftarrow \text{Mergesort}(L[0 : \text{mid}])$ ▷ the notation $L[0 : \text{mid}]$ means $[L[0], L[1], ..., L[\text{mid} - 1]]$
6: $L_{\text{right}} \leftarrow \text{Mergesort}(L[\text{mid} : n])$
7: $L_{\text{sorted}} \leftarrow \text{Merge}(L_{\text{left}}, L_{\text{right}})$
8: **return** L_{sorted}

As described, the algorithm seems to work, but we haven't really explained the procedure used to merge L_{left} and L_{right}. Before presenting the pseudocode for "Merge", I will explain it from a high level, which will be sufficient to analyze the runtime.

The "Merge" procedure takes as input two sorted lists A and B, and outputs a sorted list containing the combined elements of A and B. To start, we create a new array C to store the elements, and keep a pointer C_ptr to the start of that array (remember: a pointer is a common tool in algorithm design to keep track of a position within an array. We also create two pointers A_ptr and B_ptr to keep track of positions within arrays A and b, and initialize both to the start of their respective arrays. Next, we pretty much do the intuitive thing you'd expect - each round, we compare the elements in the locations of A_ptr and B_ptr, and we place whichever element is smaller in the location of C_ptr (in other words, we place the element into the output). We move C_ptr up by one spot, as well as the pointer of whichever array had the smaller element. If we ever reach the end of an array, we no longer use elements from that array, and we finish once we reach the ends of both arrays. Based on the rules we have for incrementing pointers, we will go through all elements exactly once, and each round, we will always place the smallest of the remaining elements of A and B into C (so C will be sorted). The pseudocode is presented below for completeness (with "output" instead of C).

Now, we should start thinking about the runtime of Mergesort. Before that, we need to know how quickly Merge runs.

Theorem 3. *"Merge" runs in $\Theta(m + n)$ time, to sort a list of size n with a list of size m.*

Proof. By analyzing the pseudocode in Algorithm 6, we see that the main body of code involves a **while** loop in lines 7-15. Before the **while** loop are many $O(1)$ operations (we haven't shown why "append" runs in $O(1)$ time, but you can take it for granted that it does[10]). Within the **while** loop (in lines 8-13) are also only $O(1)$ operations. Therefore, the runtime can be measured by measuring how many times the **while** loop runs. Since each of the iterations of the loop corresponds to one element being added to the output, the loop must run $n + m$ times, so Merge runs in $\Theta(n + m)$ time. □

Now, we can finally move to the bigger picture: analyzing the efficiency of Mergesort.

Theorem 4. *"Mergesort" runs in $\Theta(n \log n)$ time on a list of size n.*

Proof. In class, I showed a diagram involving a recursion tree to prove Theorem 4. In the notes, I will show a different method (mainly due to the difficulty of adding diagrams to LaTeX): writing and solving a recurrence. Let's call the (worst-case) runtime of Mergesort $T_{\text{merge}}(n)$ - that is, as a function of the size n. By examining the pseudocode in Algorithm 5, we can write the following recurrence:

$$T_{\text{merge}}(n) = 2T_{\text{merge}}\left(\frac{n}{2}\right) + \Theta(n)$$

[10]To be precise, for arrays, "append" runs in $O(1)$ *amortized* time, but the distinction does not matter now.

Algorithm 6 Merge

Input: sorted list A of size n and sorted list B of size m
Output: output is a sorted list, and for all x, if x appears a times in A and b times in B, x will appear exactly $a + b$ times in the output ▷ this is a formal notion of combining A and B

```
output ← new Array(n + m)          ▷ initializing a new list to store the merged elements
A.append(∞)
B.append(∞)                        ▷ the infinities are just to make the pseudocode nicer
A_ptr ← 0
B_ptr ← 0
output_ptr ← 0
while A_ptr < n or B_ptr < m do
    if A[A_ptr] ≤ B[B_ptr] then
        output[output_ptr] ← A[A_ptr]
        A_ptr ← A_ptr + 1
    else
        output[output_ptr] ← B[B_ptr]
        B_ptr ← B_ptr + 1
    end if
end while
return output
```

What this means is, the amount of time needed to run Mergesort on an input of size n is the sum of the amount of time to recursively call Mergesort on L_{left} and L_{right} (so 2 times $T_{\text{merge}}\left(\frac{n}{2}\right)$, since L_{left} and L_{right} each have $n/2$ elements), and the amount of time to call Merge (which runs in $O(n)$ - we are merging L_{left} and L_{right}, which are each $\Theta(n)$ in size). This leads to the recurrence seen above. You might be wondering about the fact that $n/2$ is not an integer unless n is even, but it doesn't really affect anything, so it's best not to worry about it. (If that answer doesn't satisfy you, you can start by analyzing Mergesort for when n is a power of 2, for which $n/2$ is well-defined. After this, notice that Mergesort doesn't take less time for a larger input, so for any n, if we round up to the nearest power of 2, we at most double n, and $2n \log(2n) \in \Theta(n \log n)$ anyway, so our concerns are put to rest.)

There is a well-known theorem, called the Master Theorem, which allows us to solve recurrences of this form. If you're curious, you can probably find many proofs online. This particular recurrence yields the solution:

$$T_{\text{merge}}(n) = \Theta(n \log n)$$

which you can (kind of) verify by trying to substitute $n \log n$ into the recurrence:

$$n \log n == n(\log n - 1) + n = 2 \cdot \left(\frac{n \log(n/2)}{2}\right) + n$$

where I removed the Θ on both $n \log n$ and n so that the math makes more sense. □

The Master Theorem is quite interesting, and I recommend reading more about it, but it's slightly out of the scope of the class, so our analysis ends here.

4.3 *Lower bounds: the comparison model

An important area of algorithmic research involves finding lower bounds for the runtime of certain problems (i.e., finding out when certain problems *must* take a certain amount of time to solve). Lower bounds are very useful - they let us know when certain problems are impossible, and when we have found optimal algorithms. For many problems, a **trivial lower bound** can be applied - an example of this is shown in Theorem 5.

Theorem 5. *Any deterministic algorithm which solves the problem P_{search} on a list of size n takes $\Omega(n)$[11] time*[12].

Proof. Take any deterministic algorithm $\mathcal{A}$ which solves P_{search}. We can construct an instance list L which forces $\mathcal{A}$ to take $\Omega(n)$ time. We do this by *adversarially* constructing L. Suppose we run $\mathcal{A}$ on a list L with undetermined values (we will slowly construct L as the algorithm progresses), and where the value to be searched for is 1. We know that $\mathcal{A}$ will read some values during the execution of the algorithm. Each time $\mathcal{A}$ reads an index i, we set $L[i] = 0$, unless every index except for i has already been read, in which case we set $L[i] = 1$. Suppose $\mathcal{A}$ returns index j. There are a few cases to consider

1. It read $L[j]$ as 1 - in this case, $\mathcal{A}$ took at least n reads on L, so its runtime was $\Omega(n)$
2. It read $L[j]$ as 0 - in this case, we can choose arbitrary values for any element which $\mathcal{A}$ did not read in the course of execution, and this gives us a list L which makes $\mathcal{A}$ fail correctness.
3. It never read the value $L[j]$ - in this case, setting $L[j] = 0$, and again choosing arbitrary values for all values (except j) which were not read by $\mathcal{A}$, will give us a list L in which $\mathcal{A}$ fails correctness.

Therefore, any correct deterministic algorithm will take $\Omega(n)$ to search a list, in the worst case. □

The idea of an adversary constructing the worst input after seeing each decision you make is a powerful one in proving lower bounds, and fits well with the idea of worst-case analysis. The lower bound presented above is called a trivial lower bound because it is the same size as the input to the algorithm, and in many cases, it is clear that algorithms must read their entire input to solve a problem. I gave a somewhat formal proof above, but thinking about the search problem should make it clear why we need $\Omega(n)$ time to solve it.

Next, I will present an argument for a non-trivial lower bound for the sorting problem. Non-trivial lower bounds are often hard to find unless we restrict the model of computation to some extent, and to that end, we need to understand the **comparison model**. In the comparison model, algorithms can only interact with the values to be sorted by comparing two of them at a time, and learning the result of the comparison. After comparing two entries in the list $L[i]$ and $L[j]$, the algorithm decides on the next two entries to compare, and continues doing this until it has enough information to completely decide the sorted order of the list.

Theorem 6. *In the comparison model, any deterministic algorithm which solves P_{sort} takes worst-case $\Omega(n \log n)$ time.*

Proof. For this proof, it might be easier to imagine that the values being sorted are not numbers. Instead, imagine that your friend has n blocks in a row, each a different shape, and your friend has decided some ordering of them, which you know nothing about. Your goal (as an algorithm) is to ask questions of the form, "Is block i smaller than block j", and after asking some number of questions, move the blocks into sorted order. This is essentially the motivating principle behind the comparison model. I will show that you need at least $\Omega(n \log n)$ questions to find the sorted order.

First, we will assume that no two blocks are equal - one will always be larger than the other. Now, consider what your algorithm does in the comparison model. At the start, any of the $n!$ (if you don't know what this means, look up "factorial function") different permutations of the blocks could be the correct sorted order. Any time a question is asked, the answer will either be "yes" or "no". Depending on the answer, some of the permutations will no longer be valid. For example, if you know that a dinosaur block is "larger" than a cat block, any permutation which has the cat block after the dinosaur block cannot possibly be the correct sorted order.

Now, the best we can hope to do is to reduce the number of valid permutations by half each question. This is because, before asking a question, we already know which subset of permutations will be valid if we receive "yes" as an answer, and which subset will be valid if we receive "no" as

[11] If you don't remember what Ω is, check out Section 7.3

[12] Note that this is also true of randomized algorithms, but the runtime definitions are a little trickier to phrase, so I will restrict attention to deterministic algorithms

an answer. In the worst case, we will receive the answer ("yes" or "no") which leads to the *larger* subset being valid. (If it helps, we can think of this as another example of adversarially generating the correct sorted order.)

Now that we have established the above fact, we can prove that we need at least $\Omega(n \log n)$ questions. We start with $n!$ permutations, and each question, we reduce the number of permutations we have to search by a factor of $1/2$. Therefore, the number of questions we need is

$$\log_2(n!) = \log_2(n) + \log_2(n-1) + \cdots + \log_2(1)$$

where we use the definition of log, as well as some of its properties. We finish it off with a useful asymptotic bounding technique you should try to remember:

$$\log_2(n)+\log_2(n-1)+\cdots+\log_2(1) \geq \log_2(n)+\log_2(n-1)+\cdots+\log_2\left(\frac{n}{2}\right) \geq \frac{n}{2}\log_2\left(\frac{n}{2}\right) \in \Omega(n \log n)$$

Stare at these for a little while, and hopefully they will make sense. We wrap up the proof by noticing that if we need $\Omega(n \log n)$ comparisons, we must also need at least that many operations. □

4.4 Exercises

1. Prove the correctness of the Mergesort algorithm, using strong induction (you might need to look up what this is). You may assume that the Merge subroutine works correctly.

2. Prove the correctness of the Merge subroutine, using the ideas of loop invariants discussed previously.

3. Suppose you have a list of 2-D points $L = [(x_i, y_i)]$, and you want to know if any of the points are within a distance of 1 from each other. Find an $O(n \log n)$ algorithm that achieves this. (This problem is adapted from a previous year of MIT's 6.046 course.) *Hint 1: start by sorting the list, possibly by both x and y value. Hint 2: Use divide-and-conquer - the divide step should involve dividing the set of points into two equal halves, such that each half is relatively close together (one way of doing this is to simply choose the left half of the points and the right half of the points). Hint 3: Think about how you want to merge your solutions - you only need to worry about points near the halfway x coordinate being too close to each other.*

4. See `https://leetcode.com/problems/the-skyline-problem/` - don't worry about actually coding the solution (although that would also probably be a good learning experience), just design the algorithmic solution. *Hint: think about how to merge two skylines - it can be done similarly to mergesort*

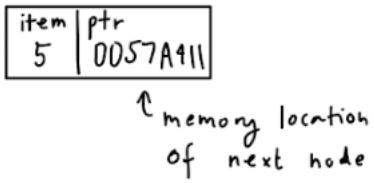


Figure 2: A node of a linked list

5 Trees and graphs: 08/06/2023

5.1 Motivation

One major design choice programmers often face is the choice of how to represent their data. So far, we have shown lists as a method of describing some sequence or set of data, but often there is more structure to the data that is lost when we represent it as a list. Concretely, we usually want the property that, if we know where a piece of data D is, we can find other data which is "related" to D quickly, for some definition of "related". This is evident even in our discussion of lists. When we sort lists, we gain the property that, given the location of any value x, we can quickly find the first element larger than x, just by moving down the list from x.

We want to be able to do similar things with other, more complicated relationships in data, and for that purpose we introduce two new **data structures**, graphs and trees.

5.2 Warm-up: linked lists

Before talking about either graphs or trees, we should start with a more basic example of a data structure.

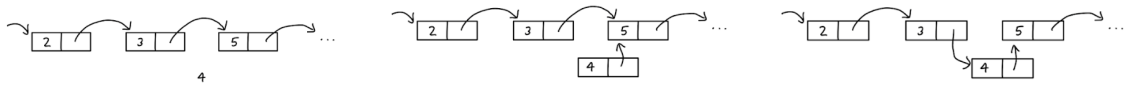
Definition 8. *A* linked list *is a data structure which supports the list interface, consisting of a chain of nodes connected by pointers.*

That definition might not have made much sense to you all, so I will explain each of the parts (we briefly covered data structures and interfaces in Section 2.2.1, so brush up on that if you need to). Basically, a linked list is some way of representing some data, and it allows you to store an ordered collection of elements (i.e., a list). The method used to store the data is different from that of an array, though - remember, an array used contiguous blocks of memory to store data. Linked lists, on the other hand, store each element of an ordered collection in a node, a structure consisting of two memory blocks we will call "item" and "ptr". This is schematically depicted in Figure 2.

The "item" cell of a node holds some arbitrary values (e.g., an element of a list for every node), and the "ptr" cell holds *the memory location of the next node in the sequence* (we call this next node the *successor*). This In this way, if we know the location of the first node (called the *head*), we can jump from each node to the next, and eventually reach the last node. The "ptr" of the last node in a linked list often stores the value NULL, to signify that there are no more elements.

The question remains: why would we ever want to use linked lists? Unlike arrays, we *cannot* access any element of a linked list in $O(1)$ time, if we know its index. However, linked lists allow us to very quickly insert and delete elements. In particular, if we want to add an element v immediately after node N, and we happen to know the location of node N, we can do so in $O(1)$ time. We do this in three steps:

1. Create a new node N', with "item" equal to v
2. Set the "ptr" value of N' to be the same as the pointer value of N
3. Set the "ptr" value of N to point to N'



(a) We start with a pointer to the node with item "3"

(b) Create a new node whose pointer points to "3"'s successor

(c) Set"3"'s new successor to the node with "4"

Figure 3: Inserting the element "4" after the element "3" in a linked list

This is depicted in Figure 3, where arrows are used in place of memory addresses. Most of the time, we will only know the memory location of the head of the linked list, which leads to $O(1)$ time inserts to the beginning of the linked list (see Algorithm 7). The important distinction to make is that we cannot, in general, insert into an array in $O(1)$ time. To insert into the beginning of an array, for example, we need to shift all of the elements in the array forward by one memory cell, before adding the element to the start of the array, which takes $O(n)$ time.

Algorithm 7 Insert at start

Input: a linked list L with head at memory address ℓ, value to insert v
Output: a linked list formed by adding value v to the start of L
1: newNode $\leftarrow$ **new** Node(v, ℓ)
2: **return** newNode

5.3 Trees

Now, we can move on to a slightly more complicated data structure, a tree. Rather than giving a formal graph-theoretic definition of trees (this will be provided later in Section 5.4), I will provide a definition of the tree *data structure*, which describes how we represent trees when programming.

Definition 9. *A* tree *is a data structure consisting of a root node, with the following fields:*

- *An "item" memory cell holding arbitrary values*
- *A list of the memory locations of* children*, which are roots of smaller trees*

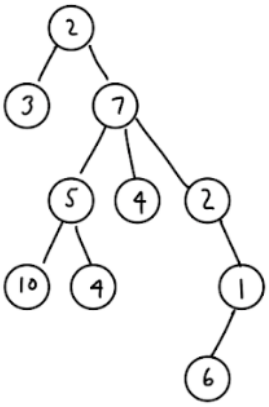
A diagram representing a tree is shown in Figure 4a. A tree can be seen as a generalization of linked lists - if you take the "children" list to always hold one child, it becomes the same as the "ptr" field discussed before. Trees have a vast number of applications, and it would be impossible to fully cover more than a few of them in the time left, so I will instead give an overview of one application, the binary search tree.

Definition 10. *A* binary search tree *is a tree with the following additional structural properties:*

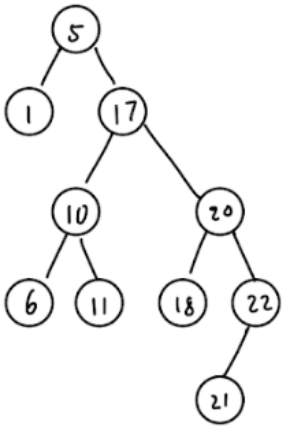
- *Each node in a binary tree, rather than having a list of children, has at most two children. Specifically, each node has a designated "left" and "right" child (which may or may not exist), and both children are themselves roots of binary search trees.*
- *Given a binary search tree with root node value r, all nodes in the left subtree are less than or equal to r, and all nodes in the right subtree are greater than or equal to r.*

The definition given above of binary search trees (henceforth, BSTs) describes them recursively as data structures which are used to *store a sorted list of elements*, similarly to a sorted array (see Figure 4b). We can efficiently search for elements in a BST, just like we can in an array. Let's say we have a BST with root value r, and we want to search for the element x. There are three cases:

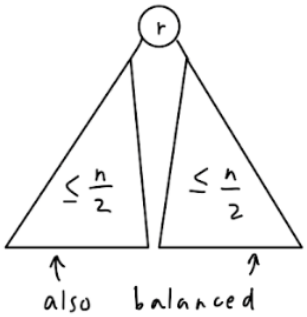
1. $x = r$ - in this case, we have successfully found x.
2. $x < r$ - in this case, we recursively search the left subtree attached to r.
3. $x > r$ - in this case, we recursively search the right subtree attached to r.



(a) A tree whose root is the node labelled 2. The children of the node with item 2 are the nodes with items 3 and 7.



(b) A binary search tree whose root is labelled 2. The left child of the root is the node 3, and the right child of the root is the node 7.



(c) A schematic "ideal" balanced binary tree whose root is labelled r. Both the left and right subtree have fewer than $n/2$ nodes, and are themselves balanced binary trees.

Figure 4: Various diagrams of trees

There are still two questions left unanswered by this.

- How efficient is the search process above? That is, how many levels will we have to go down the tree before we reach a node with no children (called a leaf)?
- Why would we want to use binary search trees instead of sorted arrays?

To answer the first question, we must first understand the concept of a *balanced* binary tree. The formal definition for balanced binary trees is a bit complicated, but intuitively, a balanced binary tree is one in which the left and right subtrees of the root have roughly the same number of nodes, and they are also both balanced binary trees themselves. To get a feeling for this, let's call an "ideal" balanced binary tree a binary tree in which the left and right subtree have at most $n/2$ nodes (see Figure 4c). Imagine searching through such an ideal balanced BST. Each time we go down one level and search through a subtree, we reduce the number of nodes we are searching through by a factor of 2. Therefore, we go down at most $\log_2 n$ levels, which means our search algorithm is asymptotically just as efficient as binary search!

In reality, balanced BSTs don't need to have the exact property of subtree sizes we described above. Instead, the relevant property we need to enforce is that the *depth* of the tree, or the length of the longest root-to-leaf path, is $O(\log n)$, which will ensure that searches are efficient. Certain implementations mentioned later guarantee this property.

We still haven't given a good reason why BSTs (even balanced ones) are in any way more useful than sorted arrays, which was our second question. To answer this, remember the analogous question we had about linked lists vs. arrays. We found that linked lists had the useful property of being able to quickly insert or delete nodes. This is also the case for balanced BSTs, which we state as the following theorem:

Theorem 7. *The following procedures can be implemented on a balanced binary search tree in* $O(\log n)$ *time:*

- *searching for an element*
- *inserting a new element, deleting an element, or modifying the value of an element*
- *finding the successor (next largest value) or predecessor (next smallest value) of any element*

I will not prove Theorem 7, but there are many implementations of balanced binary trees which make this possible, including AVL trees, red-black trees, and splay trees (which are honestly one of the coolest data structures I've ever seen). Read up on these if you're interested (although their analyses are usually complicated - the simplest here is probably AVL trees). The nice thing to note is that, in these implementations, inserting, deleting, and modifying will still *preserve* the binary search tree order and the balanced tree property.

5.4 Graphs

We begin with an abstract definition of graphs:

Definition 11. *A* graph $G = (V, E)$ *consists of a set of vertices* V *and a set of edges* E*, where each edge* $e \in E$ *is a (unordered) pair of two vertices* $\{u, v\}$*. Vertices are also called* nodes*.*

To explain this a little, graphs are just structures involving nodes connected together in some way. Nodes are pretty abstract and can be anything - some examples are given below:

- A graph can represent Facebook friend relationships. Each node is a Facebook user, and two nodes share an edge if the corresponding users are friends.
- A graph can represent how close locations are to each other. Each node may be a city, and two nodes share an edge if there is a highway directly connecting them.

For simplicity, in any graph with n nodes, we will henceforth label our nodes with the integers from 0 to $n - 1$. Now, with the framework of graphs in mind, we can more formally explain what a tree is:

Definition 12. *A* tree $T = (V_T, E_T)$ *is a connected graph without cycles. That is, for any two nodes* $u, v \in V_T$*, there is a path of edges in* E_T *which connects* u *and* v*, but there is no set of edges in* E_T *which forms a loop.*

Technically, Definition 11 is the definition of an unweighted, undirected graph. Variations on graphs exist (see Figure 5), which merit their own definitions:

Definition 13. *A* directed graph *is a graph in which the set of edges* E *is a subset of* $V \times V$*.*[13] *In other words, each edge* e *is an* ordered *pair* (u, v) *of two nodes, so the edge* (u, v) *may exist in the graph without the edge* (v, u) *existing.*

Directed edges can capture more structure than undirected edges - for example, we can represent Instagram follower relationships with directed edges, but not undirected edges, since you can follow someone without them following you back.

Definition 14. *A* weighted graph *is a graph* $G = (V, E, w)$*, where* $w : E \to \mathbb{R}$ *is a weight function, which assigns a weight* $w(e)$ *to each edge.*

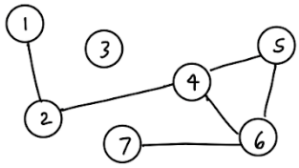
Weighted graphs also capture structure not present in unweighted graphs - for example, the earlier graph of cities connected by highways can have a weight associated with each highway, which describes the highway's length.

Now that we know what a graph is, we need to know how to represent graphs in data structures. For this, I will focus on representing weighted, directed graphs, since all the other graphs we have mentioned are special cases of this. The simplest representation of a graph is as an adjacency matrix, defined below:

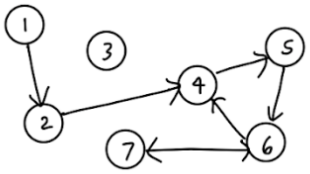
Definition 15. *The* adjacency matrix A_G *of an* n*-node graph* G *is an* $n \times n$ *matrix* A *such that the* (i, j) *entry of* A_G *is a* 1 *if and only if there is an edge pointing from node* i *to node* j *in* G*.*

Along with adjacency matrices, we can store the weight information of edges in a *weight matrix* W_G, which similarly stores the weight of the $i \to j$ edge (if it exists) in its (i, j) entry. An example of an adjacency matrix for the graph shown in Figure 5b is shown below, where I use 1-indexing for ease of understanding:

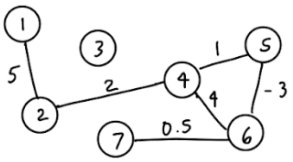
[13] See Section 7.2 for an explanation of what the $\times$ symbol means



(a) An unweighted, undirected graph. Nodes are represented as circles, and edges as straight lines connected them. Here, the vertex set is $\{1, 2, 3, 4, 5, 6, 7\}$.



(b) A directed, unweighted graph. Now, edges are represented as arrows from one node *to* another node. Note that two nodes may both point to each other (as is the case for 6 and 7)



(c) A weighted, undirected graph. Weights are written near edges, and may be any real number.

Figure 5: Various diagrams of graphs

$$A_G = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Adjacency and weight matrices are very nice in some ways - with them, you can easily (in $O(1)$ time) check to see if two nodes are connected, and find the weight of the edge connecting them if they are. However, they sometimes perform poorly when a graph does not have many edges. The size of an adjacency/weight matrix is n^2, and when the number of edges in the graph is $o(n^2)$ (basically, much fewer than n^2 - see Section 7.3 for further explanation), an n^2 size representation of the graph is too bulky. Imagine a graph with 10^{10} nodes (this is not unimaginable - 10^{10} is the estimated world population by 2057), but where each node only has at most 20 edges connected to it (for example, if the nodes are people, the edges represent links to immediate family members). Then, if we wanted to iterate through all of the nodes immediately connected to a given node, it would take around 10^{10} time with an adjacency matrix, when there are only 20 values to iterate through! Since many graphs in practice have fewer than $\Theta(n^2)$ edges, a popular graph representation is one with adjacency lists.

Definition 16. *An* adjacency list $N_G[v]$ *of a node v in graph G is a list (which may be implemented as an array, a linked list, etc.) of all neighbors of v (i.e., nodes w such that $v \to w$ is an edge).*

Using adjacency lists solves the problem that adjacency matrices presented. We can store an array of size n, where each entry i holds the adjacency list of node i. Furthermore, just like with adjacency matrices, we can easily add weight information to adjacency lists. To conclude, matrix representations are usually better when the graph is *dense* (i.e., has $\Theta(n^2)$ edges), and adjacency list representations are usually better when the graph is *sparse* (i.e., has $o(n^2)$ edges).

We wrap up the material for today with a few graph problems which are algorithmically of interest to us:

- Given a graph G and two nodes u and v, what is the length of the shortest path from u to v?
- Given a graph G, does G contain a cycle (i.e., a sequence of nodes $\{v_1, v_2, ...v_k\}$ such that all v_i are distinct, there is always an edge from v_i to v_{i+1}, and there is an edge from v_k to v_1)?
- Given a graph G, is G connected (i.e., is there always a path between any two nodes)?

5.5 Exercises

1. A *FIFO queue* is an interface which supports the following operations:

   - push: inserts an element into the queue
   - pop: removes and returns the *oldest* element still in the queue.

   Describe how to implement a FIFO queue, such that each operation takes $O(1)$ time. *Hint: use a linked list.*

2. One way to more formally describe balanced trees is with *depth-balancing* (aka height-balancing). A node is depth-balanced if the depths of its right and left subtrees differ by at most 1. A tree is depth-balanced if all of its nodes are depth-balanced. Prove that a depth-balanced tree has $O(\log n)$ depth. *Hint: show any depth-balanced tree with depth d has at least $2^{d/2}$ nodes.*

3. Read `https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/resources/mit6_006s20_lec7/` to see how AVL trees work

4. Describe an algorithm to iterate through all of the elements of a BFS tree in $O(n)$ time. *Hint: use a recursive algorithm starting at the root.*

5. Describe an algorithm to generate an adjacency matrix from adjacency lists in $O(n^2)$ time on a graph with n nodes. Do the same in the opposite direction, still in $O(n^2)$ time.

6 Breadth-first search, hashing

6.1 Shortest paths: BFS

Now that we have some idea of what graphs are, we can tackle one of the most fundamental problems in graph processing: finding shortest paths. For today's purposes, we only consider **unweighted undirected** graphs (although making the graphs directed doesn't actually change much). In this setting, shortest paths can be solved in $O(n+m)$ time on a graph G with n nodes and m edges, with an algorithm called **breadth-first search** (or **BFS**).

Before we get ahead of ourselves, we need to define our problem and desired solution more carefully. We start with some definitions.

Definition 17. *A* path *P from node v_0 to node v_k in a graph $G = (V, E)$ is a sequence of nodes $P = (v_0, v_1, v_2, ..., v_k)$ such that for all $i = 0, 1, ..., k-1$, there is an edge from v_i to v_{i+1} in E. If G is unweighted, then the* length *of a path is the number of edges traversed in the path (so the length of P above is k). A* shortest path *from node u to v is a (not necessarily unique) path of minimum length. The* distance *between u and v in G is the length of the shortest path from u to v.*

One example which demostrates our problem is finding **Erdös numbers**. Paul Erdös was a famous mathematician whose prolific publications led to the Erdös number's use as a (albeit, not very good) measure of research success. The Erdös number is defined as follows:

- Paul Erdös has an Erdös number of 0.
- Anyone who jointly published a paper with Paul Erdös has an Erdös number of 1.
- Anyone left who jointly published a paper with someone with an Erdös number of 1, has an Erdös number of 2, and so on.

In other words, if we create a graph in which nodes are mathematicians, and edges connect two mathematicians if they ever jointly worked on a paper, then the Erdös number of a mathematician m is the length of the shortest path, or distance, between m and Paul Erdös on the graph. We now define the problem.

Definition 18. *The* single-source shortest paths *problem takes the following inputs:*

- *a graph G, represented as adjacency lists*
- *a node s, called the source*

A correct output to the problem is a list "distances", where for every node v, distances[v] holds the distance between s and v.

The output to the problem might be a bit confusing. Storing distances might make sense for finding Erdös numbers, but if the problem is called "shortest paths", shouldn't the output be a list which stores the shortest path from s to any other node? The following two theorems shed some light on this subtlety.

Theorem 8. *Given an undirected connected graph $G = (V, E_G)$, and a source node s, there exists a tree $T = (V, E_T)$ rooted at s with the following two properties:*

1. *Any edge in the tree is an edge in the graph (i.e., $E_T \subseteq E_G$).*
2. *Given any node v, the unique path from v up the tree to the root s is a shortest path.*

Before continuing, let's take a moment to analyze what would happen if we tried to store shortest paths as a list of paths from s to every node, as discussed earlier. There are a total of $n-1$ paths we need to store (one for each node other than s), and each path contains at most n vertices. This means that we might need up to $\Theta(n^2)$ space (see Exercise 1) to store the shortest paths like this. Furthermore, any algorithm that generates a solution represented like this needs $\Omega(n^2)$ time. What Theorem 8 suggests is that we don't need to store this much information - it suffices to store a *tree* which has all of the shortest path information.

Recall from 5.3 one way to represent a tree - for each node, we have a (possibly empty) list of the children of that node. Instead of doing this, for today, we will store trees slightly differently. Since every node has a *unique parent* in a tree (except for the root, which has no parent), storing the parent of every node suffices to store the tree. Note that this representation of a tree requires $\Theta(n)$ space, which is much better than the $\Theta(n^2)$ space from before!

Now that we understand the space savings that Theorem 8 is communicating, let's try to understand what a shortest paths tree is. Essentially, if we have a shortest path tree T as described, and we are given any node v, we can generate a shortest path from v to s by following the parent pointer from v all the way up to the root (which is s).

There is still one problem with Theorem 8 - the theorem specifies that with a tree, we can find a shortest path *from* any node *to* s, but our inital problem was to find shortest paths *from* s *to* any node. However, in undirected graphs, paths are reversible, so either one suffices, and we will use them interchangeably (see Exercise 2 to see how to get around this for directed graphs). Finally, we can prove the theorem.

Proof. Let $D[P]$ be the length of any path P. For every node v in the graph, let the parent of v be the first node on a shortest path from v to s. The tree formed this way clearly satisfies property (1) of Theorem 8. All we need to show now is that the path from any node to the root in this tree is a shortest path. We show this by induction on the depth of nodes (recall that the depth of a node in a tree is the number of edges on its path to the root).

First, consider the nodes of depth 0. There is only one such node, which is the root s. Trivially, the shortest path from s to s in the tree is the same as in G, since the path is just (s), a single-node path.

Next, for any node u, let the path formed by following parent pointers from u up to the root s in our tree be $P_T(u)$, and let the distance of any path P be $D[P]$. Consider some node v. We know from construction that there is some shortest path P^* from v to s which looks like $(v, w, a_1, a_2, ..., s)$, where w is the parent of v in the tree. Now, since w is the parent of v, it must have depth $d-1$ in the tree, so we know that the path $P_T(w)$ is a shortest path from w to s. Finally, we see that:

$$\begin{aligned} D[P_T(v)] &= D[(v,w)] + D[P_T(w)] && (1)\\ &\leq D[(v,w)] + D[(w, a_1, a_2, ..., s)] && (2)\\ &= D[P^*] && (3) \end{aligned}$$

where steps (1) and (3) use the additive property of distances[14]., and step (2) uses the fact that $P_T(w)$ is a shortest path. □

Our next theorem shows that finding distances is, in some sense, equivalent to finding a shortest path tree.

Theorem 9. *In undirected unweighted connected graph G with n nodes and m edges, given a list D of distances from s to every node, we can calculate a shortest paths tree in $O(n+m)$ time.*

Proof. The following algorithm accomplishes the task in $O(n+m)$ time, assuming the graph is given as adjacency lists.

For every node $v \neq s$, find the neighbor of v with the smallest distance to s, and set this neighbor as the parent of v. Since any shortest path from v to s must go through at least one of the neighbors, choosing the neighbor with the smallest distance clearly is the best local decision. □

Theorem 9 tells us that computing distances, at least for the case of unweighted undirected graphs, is sufficient (although with small changes, the same is true for weighted, directed graphs). Now, we have the intuition required to describe breadth-first search (or BFS), shown as Algorithm 8 below.

The idea behind BFS is very simple. We start with a single node s, and mark its distance as 0. We then explore all of the neighbors of s, and mark all of their distances as 1. We proceed

[14]In an unweighted graph, the length of (v, w) is just one, but in weighted graphs, this is not necessarily true, so I left the statement as general as possible.

to iterate through all nodes of distance 1, iterate through each of their neighbors, and mark any node we haven't seen yet as having distance 2. This process continues until we no longer have any nodes of a certain distance. The pseudocode describes this process more exactly.

Algorithm 8 BFS

Input: graph G with n nodes numbered 0 to $n-1$, starting node s
Output: BFS distances from s to each node

```
1:  distances ← new Array(n, ∞)          ▷ This notation means the array is initialized to all ∞
2:  thisLevel ← new LinkedList([s])                ▷ We initialize to a single node with item s
3:  distances[s] ← 0
4:  levelNumber ← 0
5:  while thisLevel is not empty do
6:      levelNumber ← levelNumber + 1
7:      newLevel ← new LinkedList()
8:      for u in thisLevel do
9:          for v in u.neighbors() do
10:             if distances[v] = ∞ then      ▷ If it equals ∞, the node has not been visited before
11:                 newLevel.add(v)
12:                 distances[v] ← levelNumber
13:             end if
14:         end for
15:     end for
16:     thisLevel ← newLevel
17: end while
18: return distances
```

I won't bother proving correctness, since much of the intuition from before can be applied to this, but I will prove runtime, since it introduces certain interesting ideas.

Theorem 10. *BFS runs in $O(n+m)$ on a graph with n nodes and m edges.*

Proof. Consider each of the operations before the **while** loop on line 5. Initializing the "distances" array takes $O(n)$ time, and all of the rest of the operations are $O(1)$. Similarly, all of the operations within each of the loops takes $O(1)$ time. It only remains to prove that the loops execute a small number of times. In the past, when we have seen nested loops, we argued that inner loops execute at most some number of times for every execution of the outer loop, and we multiplied the two values to get the total number of executions of the inner loop. Here, we use a different strategy: we analyze each loop separately from the rest.

First, consider the **while** loop on line 5. This loop executes once for every possible "levelNumber" (which is just a distance). Since the maximum distance of any node from s is $n-1$, the **while** loop executes at most $O(n)$ times.

Next, consider the **for** loop on line 8. This **for** loop is run at most one time on each node in the graph, and so it executes $O(n)$ times.

Finally, consider the innermost **for** loop on line 9. This loop executes at most once for each node u and neighbor v, so it executes at most twice for each edge $\{u, v\}$. Therefore, it runs $O(m)$ times.

Putting all of these together, the algorithm runs in $O(n+m)$ time. □

6.2 Karatsuba algorithm for multiplication

Since we're now at the end of the class, I wanted to make the last topic a cool algorithm you probably haven't seen before. In elementary school, you learned an algorithm for multiplying two

numbers. In base 10, given two numbers[15], say $abcd$ and $efgh$, you compute:

$$\begin{aligned} abcd \times efgh &= (1000a + 100b + 10c + d)(1000e + 100f + 10g + h) \\ &= 1000000ae + 100000(af + be) + 10000(ag + bf + ce) \\ &+ 1000(ah + bg + cf + de) + 100(bh + cg + df) + 10(ch + dg) + dh \end{aligned}$$

In other words, you memorize each of the 1-digit by 1-digit multiplications, and by expanding each of the numbers, you can compute the product by computing 16 of these 1-digit by 1-digit multiplications, and adding appropriately. In general, if you want to multiply two n-digit by n-digit numbers, you need to compute n^2 1-digit by 1-digit multiplications to solve the problem, so this algorithm runs in $\Theta(n^2)$ time.

The previous paragraph should make you a little confused. Earlier, in section 1.6, I said that multiplication can be done in $O(1)$ time, so why can't we just multiply two n-digit numbers like that? The truth is that the previous $O(1)$ time multiplication was a bit of a simplification. In reality, computers hold numbers in **registers** with some fixed number of bits (usually 32 or 64). For most purposes, we won't need to multiply numbers larger than 10^{32} for our algorithms, so we can ignore the size of the registers. However, for the problem of n-digit by n-digit multiplication, we obviously can no longer assume this. Even if we are able to multiply any two 32-digit numbers in $O(1)$ time, this is essentially the same as multiplying single-digit numbers in base 10^{32} instead of base 10. n-digit numbers in base 10 become $n/32$-digit numbers in base 10^{32}, so being able to multiply 32-bit numbers quickly only gives you an improvement of $n^2 \to (n/32)^2$ in number of elementary operations.

Another complication which you may have noticed is that the previous analysis says nothing of the number of additions necessary to multiply the two numbers. It turns out that this is kind of irrelevant. Addition of two n-bit numbers can be done easily in $O(n)$ time (look up addition circuits for more information), and although the proof of this is a bit complicated, the additions in our algorithms end up not contributing to the runtime. Therefore, for the purposes of simplicity, we will analyze our multiplication algorithm complexity as the number of 1-digit by 1-digit multiplications required.

In 1960, Anatoly Karatsuba discovered an algorithm that requires $\Theta(n^{\log_2 3})$ such multiplications. Consider first naïvely trying to apply divide-and-conquer to multiply two numbers X and Y, where

$$X = A \cdot 10^{n/2} + B$$
$$Y = C \cdot 10^{n/2} + D$$

and A, B, C, D are all $n/2$-digit numbers. (For example, if $X = 123456$ and $Y = 789012$, then A would equal 123, $B = 456$, $C = 789$, $D = 012$.) If we apply divide and conquer, we see that

$$X \cdot Y = A \cdot C \cdot 10^n + (A \cdot D + B \cdot C) \cdot 10^{n/2} + B \cdot D$$

So we have to calculate $A \cdot C$, $A \cdot D$, $B \cdot C$, and $B \cdot D$, for a total of 4 $n/2$-digit by $n/2$-digit multiplications. Let $T(n)$ be the number of 1-digit by 1-digit multiplications for this algorithm to multiply two n-digit by n-digit numbers. Then,

$$T(n) = 4 \cdot T(n/2)$$

is the recurrence we need to solve. If we plug in $T(n) = n^2$, we see that it solves the recurrence, so the runtime with this naïve divide-and-conquer is $\Theta(n^2)$

Let's take a closer look at the process described above, and define the following:

$$Z_0 = B \cdot D$$
$$Z_1 = A \cdot D + B \cdot C$$
$$Z_2 = A \cdot C$$

so that our final answer is:

$$X \cdot Y = Z_2 \cdot 10^n + Z_1 \cdot 10^{n/2} + Z_0$$

[15]For the rest of today, I will assume that computers, like us, operate in base 10 (instead of base 2), for simplicity.

The key fact that lets the Karatsuba algorithm perform divide-and-conquer more efficiently is that we don't actually need to calculate both $A \cdot D$ and $B \cdot C$. We don't care what either of their values are - instead, we care about the value of the sum Z_1. But with some nice algebraic manipulation, we can see that

$$Z_1 = (A + B) \cdot (C + D) - Z_0 - Z_2$$

Amazingly, this means that we only need to compute *three* $n/2$-digit by $n/2$-digit multiplications to calculate Z_0, Z_1, and Z_2. To clarify, the three multiplications are

$$A \cdot C,\ B \cdot D, \text{ and } (A + B) \cdot (C + D)$$

after which Z_0, Z_1, and Z_2 can be calculated with just some subtractions. Let's now analyze the complexity of the Karatsuba algorithm.

Theorem 11. *The Karatsuba algorithm requires* $\Theta(n^{\log_2 3})$ 1*-digit by* 1*-digit multiplications.*

Proof. Based on the fact that we only need three $n/2$-digit by $n/2$-digit multiplications, our recurrence becomes:

$$T^*(n) = 3T^*(n/2)$$

where T^* is the complexity of the Karatsuba algorithm. Just like with mergesort, I won't show how I solved the recurrence here, but by substituting

$$T^*(n) = 3^{\log_2 n}$$

into the equation, we see that it is a solution:

$$3^{\log_2 n} = 3 \cdot 3^{\log_2 (n/2)} = 3 \cdot 3^{\log_2 n - 1} = 3^{\log_2 n}$$

Therefore,

$$T^*(n) = 3^{\log_2 n} = 3^{\log_3 n \cdot \log_2 3} = \left(3^{\log_3 n}\right)^{\log_2 3} = n^{\log_2 3} = n^{1.58\ldots}$$

is the solution to the recurrence (where we use some simple log properties to simplify). Compare this to our earlier $\Theta(n^2)$ complexity of our elementary school algorithm - this is much faster! □

6.3 Exercises

1. Earlier, we said that for the single-source shortest path problem, if we tried to store a shortest path from s to v as a separate list for each v, it would take $\Theta(n^2)$ space. In reality, we only showed that $O(n^2)$ was an upper bound. To show that it is also a lower bound, find an example of a graph G and source node s for which the sum of the lengths of all shortest paths to s is $\Omega(n^2)$.

2. Show how we can form a shortest path tree for directed graphs. *Hint: one option is just to represent trees as in Section 5.3. Another option is to first reverse all edges in the directed graph, then find the shortest path tree as described in this new graph. In this shortest path tree, any path from a node v to the source s corresponds to a path* from s to v *in the original graph.*

3. Suppose you and your friends are driving cars, and you have an undirected graph where edges are roads and nodes are stores. Your friends want to buy watermelons to eat together later, so you want to find which friend is closest to a watermelon store. You know which nodes your friends are currently at, as well as which nodes are watermelon stores. Come up with an $O(n + m)$ algorithm to find the shortest path from *any* friend to *any* watermelon store. *Hint: use BFS. Modify your original graph by adding two nodes, one with edges to all of your friends' nodes, and another with edges to all of the watermelon stores. Then, compute the distance between these two nodes.*

4. Read `https://en.wikipedia.org/wiki/Strassen_algorithm` to see another cool divide and conquer algorithm, which is pretty similar to Karatsuba's algorithm.

7 Important notation

Here, we review some important notation used in the class.

7.1 Logical notation

I won't give very formal definitions for logic, but this section might help you understand some of the ideas in the class.

- A **proposition** is a logical statement, which can be either true or false
- A **logical operation** connects one or more propositions to form new propositions

For example, the statement "It is raining outside" is a proposition. We can represent propositions as symbols. For propositions P and Q, we define the following logical operations:

- $\neg P$ (read: not P) is true if P is false, and false if P is true
- $P \vee Q$ (read: P or Q) is true if P is true or Q is true, and is false if both P and Q are false
- $P \wedge Q$ (read: P and Q) is true if both P and Q are true, and is false otherwise
- $P \implies Q$ (read: P implies Q) is true if P is false, or Q is true (or both) - this is probably the most confusing one, but thinking about specific examples might be helpful
- $P \impliedby Q$ (read: P is implied by Q) is the same proposition as $Q \implies P$
- $P \iff Q$ (read: P if and only if Q) is true if both P and Q are true, or if both P and Q are false - it is also the same proposition as $(P \implies Q) \wedge (Q \implies P)$

So far, what we have covered is referred to as **propositional logic**. Next, we move on to **predicate logic**. Before continuing this section, read the intro of Section 7.2. **Predicates** are similar to propositions - they are essentially propositions which are functions of one or more input variables. For example, the predicate "x is a prime number" may or may not be true, depending on the value of x. We can represent predicates in a similar notation as functions - so as a running example, let's use $P(x)$ to refer to the predicate above concerning primes. Propositions can be considered predicates with no input.

Predicates "lose variables" if we fill in the value of the input (so $P(3)$ is now a [true] proposition with zero input variables). But often, we want to make statements about the relationship between predicates and the elements of a set, not just one specific input like 3. For example, we might want to say something like "All numbers of the form $2^{2^n} + 1$ for natural number n are prime" (this is not true - look up Fermat primes - but we might still want to say it). We can do this (and things like this) with the help of two symbols called **quantifiers**, which relate the truth of predicates to inputs from an entire set:

- The symbol $\exists$ is called the "existential quantifier", and can be read as "there exists" - $\exists x \in S,\ A(x)$ is true if and only if there is at least one value of x in the set S for which $A(x)$ is true.
- The symbol $\forall$ is called the "universal quantifier", and can be read as "for all" - $\forall x \in S,\ A(x)$ is true if and only if, for every value of x in the set S, $A(x)$ is true.

Here are some examples which illustrate the use of the predicates:

- To say "All numbers of the form $2^{2^n} + 1$ for $n = 0, 1, 2, ...$ are prime" in logical notation, we can write:
$$\forall n \in \mathbb{N},\ P(2^{2^n} + 1)$$
which is read as "For all n in the set of natural numbers, $2^{2^n} + 1$ is a prime number".
- To say "There is a prime even number", we write:
$$\exists n \in 2\mathbb{Z},\ P(n)$$
which can be read "There exists a number n in the set of even integers such that n is also prime"

7.2 Sets

A **set** is a mathematical object which contains other entities called **elements**. The elements of a set can be pretty much anything: numbers, letters, geometric objects like circles, or even other sets. Sets serve as a sort of mathematical building block, and actually are used in the very formal definitions of many things you may have taken for granted. For example, a very formal definition of real numbers defines each real number as a(n) (ordered) set of sets of rational numbers, which themselves are (ordered) sets of natural numbers, which are often defined as sets themselves. It's fine if you didn't catch all of that - the takeaway is that sets are very important. For sets A and B, and some symbol x, the following notation is used:

- $x \in A$ or $A \ni x$ (read: x is in A) mean that the x is an element A
- $A \subseteq B$ or $B \supseteq A$ (read: A is a subset of B) mean that every element of A is also an element of B - that is,
$$\forall y,\ y \in A \implies y \in B$$
- $A = B$ (read: A equals B) means that $A \subseteq B$ and $B \subseteq A$
- $A \subset B$ or $B \supset A$ (read: A is a proper subset of B) mean that A is a subset of B, and A is *not* equal to B
- $A \cup B$ (read: the union of A and B) is the set of all elements appearing in either A or B, or both. That is:
$$x \in (A \cup B) \iff (x \in A) \vee (x \in B)$$
- [16]$A \cap B$ (read: the intersection of A and B) is the set of all elements in both A and B. That is:
$$x \in (A \cap B) \iff (x \in A) \wedge (x \in B)$$
- $A \times B$ (read: the Cartesian product of A and B) is the set of ordered pairs (a, b) of elements $a \in A$ and $b \in B$. That is,
$$(a, b) \in A \times B \iff a \in A \wedge b \in B$$

Sets can be classified in different ways:

- finite vs. infinite
- ordered vs. unordered

Unordered sets can be represented comma-separated elements in curly braces. The following sets are commonly used:

- $\mathbb{N}$: the set of natural numbers $\{1, 2, 3, ...\}$
  - We use $\mathbb{N}_0$ to refer to the natural numbers, with 0 as an added element (so $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$)
- $\mathbb{Z}$: the set of integers $\{..., -2, -1, 0, 1, 2, ...\}$
- $\mathbb{Q}$: the set of rational numbers - numbers of the form $\frac{p}{q}$, where $p, q \in \mathbb{Z}$

Set-builder notation is a common way of describing subsets of a larger set, as shown in the example below:

$$E = \left\{ x \in \mathbb{N} \;\middle|\; \frac{x}{2} \in \mathbb{N} \right\}$$

In this example, the set E is defined. You can read the set-builder notation as "the set of all x in $\mathbb{N}$ such that $\frac{x}{2}$ is in $\mathbb{N}$". This is just the set of even positive numbers.

[16]Note that the symbols for union and intersection look a lot like the symbols for AND and OR in Section 7.1 - this is for good reason, as

$$\forall x, x \in A \cup B \iff x \in A \vee x \in B$$

The similar statement for $\wedge$ and $\cap$ is also true.

7.3 Asymptotic notation

See Section 1.4 for a discussion of the ideas behind asympototic notation. The rest of this section compiles some definitions.

Consider functions f and g, each functions of a single variable.

- We say $f \in O(g)$ if and only if there exist some constants c and N such that

$$\text{for all } n > N, f(n) < c \cdot g(n)$$

  An alternative characterization (if you know some analysis) is:

$$f \in O(g) \iff \limsup_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty$$

- We say $f \in o(g)$ if and only if

$$\forall c > 0,\ \exists N \in \mathbb{N} \text{ such that } n > N \implies f(n) < c \cdot g(n)$$

  Here, I used some of the logic notation as practice. What this is essentially saying is that g is "bigger" than f - even if you pick a really small constant like $c = 0.000001$, eventually (for large enough n) $c \cdot g(n)$ will still always be larger than $f(n)$. The alternative analysis characterization is:

$$f \in o(g) \iff \limsup_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

- We say that $f \in \Omega(g)$ if and only if $g \in O(f)$.
- We say that $f \in \omega(g)$ if and only if $g \in o(f)$.
- We say that $f \in \Theta(g)$ if and only if $f \in O(g)$ and $f \in \Omega(g)$.
- Finally, we say that $f \sim g$ if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

8 Resources

Linked are some other resources you may find helpful:

- 6.006 OCW: A lot of material in this class overlaps with a course taught at MIT, whose materials can be found here: `https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/`
- 6.042 OCW: I sent this in an email earlier, but here is a textbook for a different MIT course (which is actually a prerequisite to the one linked above): `https://ocw.mit.edu/courses/6-042j-mathematics-for-computer-science-spring-2015/resources/mit6_042js15_textbook/`
- CLRS: The (probably) most well-known textbook for algorithms is CLRS: Introduction to Algorithms, but I've found it to be hard to read if you're just starting out. It might be useful for reference, though.