

CMSC 351: Introduction

Justin Wyss-Gallifent

May 25, 2023

1	Introduction	2
2	Learning to Write Algorithms	2
3	Course Goals	2

1 Introduction

This course is a brief introduction to algorithms. It covers the basics of algorithm design, many common algorithms related to sorting and searching of lists, arrays and graphs, and ends with a brief discussion of the abstraction of time complexity.

2 Learning to Write Algorithms

I'm thinking of the course as an algorithms analogy to MATH310 which is an introduction to proofs. In fact I think of writing algorithms much like I think of writing proofs. It's hard to give a simple and straightforward explanation of "how to write an algorithm to do X" much the same way as it is to give a simple and straightforward explanation of "how to prove X". Rather the best approach is to learn some basic techniques while studying existing algorithms (or proofs) and initially practicing modifications of those algorithms (or proofs).

3 Course Goals

In broad strokes the goals of the course include:

1. Understand the algorithms.
2. Apply the algorithms to various input data.
3. Calculate various operation counts.
4. Calculate pseudocode time complexity in reasonable situations.
5. Make simple modifications to these algorithms in various ways.
6. Points 1,2,3,4 for simple modifications of these algorithms.
7. Understand common algorithmic approaches (linear, divide-and-conquer, etc.)
8. Understand and modify the nuances of some proofs related to algorithms.
9. Apply general recursion approaches - trees, the Master Theorem, constructive induction.
10. Understand how algorithms can be applied in real-world situations.
11. Understand new algorithms whose fundamental underpinnings are similar to familiar ones.
12. Understand the notions of Turing machines and of P and NP.

CMSC 351: Coin Changing

Justin Wyss-Gallifent

May 17, 2024

1	Introduction to Coin Changing	2
2	The Greedy Method (a Minimization Attempt)	2
3	An Algorithm for Minimization	3
3.1	A Dynamic Programming Idea	3
3.2	An Algorithm	5
3.3	Time Complexity	5
4	Counting the Ways	5
4.1	Introduction	5
4.2	A Dynamic Programming Approach	6
4.3	An Algorithmic Idea	7
4.4	An Actual Algorithm	7
4.5	Time Complexity	10

1 Introduction to Coin Changing

Suppose all you have are 1-cent, 5-cent and 10-cent coins but you have infinitely many of each. For any given (cent) total n we wish to obtain n cents out of our coins. Consider the following associated problems:

- (a) How can we do this if we don't care how many coins we use?
- (b) How can we do this if we wish to use the minimum number of coins?
- (c) How many ways can we do this if we don't care about using the minimum number of coins?

In this case our coins can be thought of in a list $C = [1, 5, 10]$ and even though we'll change that we'll always assume that we have a 1-cent coin. This guarantees that it's possible to obtain any amount.

Note 1.0.1. The reason this is known as the *coin changing problem* is that the original premise is that the total n is the amount of change being given for a purchase and the question was about how this can be done.

Note 1.0.2. The question can be asked even without a 1-cent coin but it gets more challenging. For example if $C = [3, 7]$ it's not clear at all which totals can be even be made.

2 The Greedy Method (a Minimization Attempt)

An intuitive approach (which doesn't always work, as we'll see) to using the minimum number of coins is to be *greedy*. Since we wish to use the minimum number of coins it seems sensible to use as many of the large coins as possible, followed by the next largest, and so on.

Example 2.1. Suppose $C = [1, 5, 10]$ and we wish to obtain $n = 27$ cents. We first grab two 10-cent coins (the most we can have) followed by one 5-cent coin (the most we can have) followed by two 1-cent coins. We have thus used 5 coins.

This is in fact optimal - it is the minimal number of coins, but this may not be obvious.

Example 2.2. Suppose $C = [1, 10, 25]$ and we wish to obtain $n = 30$ cents. We first grab one 25-cent coins (the most we can have), we can't grab any 10-cent coins, so we finish by grabbing five 1-cent coins. We have thus used 6 coins.

This solution is not optimal however since we could have grabbed three 10-cent coins instead.

3 An Algorithm for Minimization

3.1 A Dynamic Programming Idea

We've seen that our greedy approach is not guaranteed to give us an optimal (smallest) number of coins. However we know that there must be some optimal solution so we can ask - how might we go about finding it?

Before developing an algorithm let's make an observation. For now let's stick with $C = [1, 5, 10]$.

Let's suppose A is a function such that $A[x]$ equals the minimum number of coins necessary to obtain x cents. So for example it's easy to see that:

$$\begin{aligned}A[0] &= 0 \text{ (No coins.)} \\A[1] &= 1 \text{ (One 1-cent coin.)} \\A[2] &= 2 \text{ (Two 1-cent coins.)} \\A[3] &= 3 \text{ (Three 1-cent coins.)} \\A[4] &= 4 \text{ (Four 1-cent coins.)} \\A[5] &= 1 \text{ (One 5-cent coin.)} \\A[6] &= 2 \text{ (One 5-cent coin and one 1-cent coin.)}\end{aligned}$$

Suppose we know $A[0], \dots, A[x-1]$ for some x . Is there a sneaky way to obtain $A[x]$?

The answer is fairly easy! In the $C = [1, 5, 10]$ case there are three possibilities:

1. We could first select a 1-cent coin, then obtain $x-1$, then combine. We can do this with $A[x-1] + 1$ coins.
2. We could first select a 5-cent coin, then obtain $x-5$, then combine. We can do this with $A[x-5] + 1$ coins. Note that this is only a possibility if $x \geq 5$ because if $x < 5$ then $x-5 < 0$ which we can't do.
3. We could first select a 10-cent coin, then obtain $x-10$, then combine. We can do this with $A[x-10] + 1$ coins. Note that this is only a possibility if $x \geq 10$ because if $x < 10$ then $x-10 < 0$ which we can't do.

So what we'll do is assign $A[x]$ to be the minimum of these three (or rather the minimum of those that make sense).

Proof. It's fairly easy to see that this is optimal and here's the basic proof for the $C = [1, 5, 10]$ case which generalizes easily:

Assume by way of contradiction that:

$$A[x] < \min \{A[x-1] + 1, A[x-5] + 1, A[x-10] + 1\}$$

Suppose the coin combination used to obtain the actual optimal solution for x involves a c -cent coin where $c \in \{1, 5, 10\}$ (it has to involve at least one of

these). Then $x - c$ cents may be obtained by removing a c -cent coin from this optimal solution for x which implies that $A[x - c] \leq A[x] - 1$.

However the assumption tells us that $A[x] < A[x - c] + 1$ and so we have:

$$A[x] < A[x - c] + 1 \leq (A[x] - 1) + 1 = A[x]$$

This is a contradiction. \mathcal{QED}

Thus we have a way of obtaining $A[x]$ as long as we know all previous values. To jump-start this process observe that $A[0] = 0$ because it takes 0 coins to obtain 0 cents.

Let's see how this works for $A[0]$ through $A[20]$ with a table. We won't do all the values but rather point out some critical ones. We start with:

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$A[x]$	0																				

For $A[1]$ we look at $1 + A[1 - 1] = 1 + A[0] = 1$, $1 + A[1 - 5] = 1 + A[-4] = BAD$, $1 + A[1 - 10] = 1 + A[-9] = BAD$ so we only have one value and so $A[1] = 1$.

For $A[2]$ we look at $1 + A[2 - 1] = 1 + A[1] = 2$, $1 + A[2 - 5] = 1 + A[-3] = BAD$, $1 + A[2 - 10] = 1 + A[-8] = BAD$ so we only have one value and so $A[2] = 2$.

If we do $A[3], A[4]$ (try them!) we have:

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$A[x]$	0	1	2	3	4																

For $A[5]$ we look at $1 + A[5 - 1] = 1 + A[4] = 5$, $1 + A[5 - 5] = 1 + A[0] = 1$, and $1 + A[5 - 10] = 1 + A[-5] = BAD$ so we have two values and take the minimum and so $A[5] = 1$.

For $A[6]$ we look at $1 + A[6 - 1] = 1 + A[5] = 2$, $1 + A[6 - 5] = 1 + A[1] = 2$, and $1 + A[6 - 10] = 1 + A[-4] = BAD$ so we have two values and take the minimum and so $A[6] = 2$.

For $A[7]$ we look at $1 + A[7 - 1] = 1 + A[6] = 3$, $1 + A[7 - 5] = 1 + A[2] = 3$, and $1 + A[7 - 10] = 1 + A[-3] = BAD$ so we have two values and take the minimum and so $A[7] = 3$.

If we do $A[8], A[9]$ (try them!) we have:

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$A[x]$	0	1	2	3	4	1	2	4	4	5											

For $A[10]$ we look at $1 + A[10 - 1] = 1 + A[9] = 6$, $1 + A[10 - 5] = 1 + A[5] = 2$, and $1 + A[10 - 10] = 1 + A[0] = 1$ so we have two values and take the minimum and so $A[10] = 1$.

If we continue this process to $A[20]$ we find:

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$A[x]$	0	1	2	3	4	1	2	4	4	5	1	2	3	4	5	3	4	5	6	7	2

3.2 An Algorithm

The preceding idea can then give us a nice algorithm. Suppose C is a list of coin denominations, so for example $C = [1, 5, 10]$ in the example we've been working through. This algorithm will find $A[x]$ for $x = 1, \dots, n$ for some maximum n :

```
A = empty list which can grow as needed
A[0] = 0
C = list of coin denominations
for x = 1 to n:
    howmanycoins = infinity
    for each coin in C:
        if x - coin >= 0:
            howmanycoins = min(howmanycoins, 1+A[x - coin])
        end if
    end for
    A[x] = howmanycoins
end for
```

Take a minute to digest how this works. For each n up to and including $n = x$ we iterate through the coins looking at each $x - \text{coin}$ and for those that make sense we obtain the minimum of all the $1 + A[x - \text{coin}]$ values.

3.3 Time Complexity

While we haven't (yet) discussed time complexity in detail in this course there are two things we can observe:

- There are two loops - the outer loop iterates n times and the inner loop iterates $\text{length}(C)$ times and so the innermost body iterates $n\text{length}(C)$ times.
- This is an approach known as *dynamic programming* which typically means that we use earlier solutions to efficiently calculate later ones. In this case we saw that to calculate $A[x]$ we use inputs smaller than x . Once we know the values up to $A[x - 1]$, finding $A[x]$ is quick.

4 Counting the Ways

4.1 Introduction

Now let's consider the problem of how many ways we can obtain n cents. We'll go back to our initial currencies of 1, 5, and 10.

For specific values of n we can brute-force this. For example for $n = 10$ we can do:

- Ten 1-cent coins.

- Two 5-cent coins.
- One 10-cent coin.
- One 5-cent coin and five 1-cent coins.

Thus we have a total of 4 ways.

Suppose we wished to write an algorithm which would give the answer for any given n . How might we go about this? Moreover what if we had a different set of coins rather than 1,5,10?

4.2 A Dynamic Programming Approach

Before creating a general approach (which will lead to an algorithm) consider the following observation:

Suppose we have two coin denominations 2 and 5. If $n = 12$ then it's easy to see that there is 1 way to obtain $n = 12$ using only 2-cent coins. How about if we also allow 5-cent coins? Observe that we have a disjoint sum:

$$\begin{aligned} \# \text{ ways to get } 12 \text{ using 2 and/or 5} &= \# \text{ ways to get } 12 \text{ using 2} \\ &\quad + \# \text{ ways to get } 12 \text{ using 2 and at least one 5} \end{aligned}$$

We know the first summand is 1. For the second summand once we choose to use a single 5-cent coin we have 7 cents left to obtain and we can do that however we wish. Thus:

$$\begin{aligned} \# \text{ ways to get } 12 \text{ using 2 and/or 5} &= \# \text{ ways to get } 12 \text{ using 2} \\ &\quad + \# \text{ ways to get } 7 \text{ using 2 and/or 5} \end{aligned}$$

Take a moment to see what we have observed here. As a general rule for some n and two denominations c_1 and c_2 we have:

$$\begin{aligned} \# \text{ ways to get } n \text{ using } c_1 \text{ and/or } c_2 &= \# \text{ ways to get } n \text{ using } c_1 \\ &\quad + \# \text{ ways to get } n - c_2 \text{ using } c_1 \text{ and/or } c_2 \end{aligned}$$

This generalizes even further with a set of coin denominations $[c_1, \dots, c_r]$:

$$\begin{aligned} \# \text{ ways to get } n \text{ using } c_1, \dots, c_r &= \# \text{ ways to get } n \text{ using } c_1, \dots, c_{r-1} \\ &\quad + \# \text{ ways to get } n - c_r \text{ using } c_1, \dots, c_r \end{aligned}$$

Before proceeding note that the above is only true if $n \geq c_r$ since otherwise we can't use a c_r denomination coin at all. Thus more accurately:

- If $n \geq c_r$ then:

$$\begin{aligned} \# \text{ ways to get } n \text{ using } c_1, \dots, c_r &= \# \text{ ways to get } n \text{ using } c_1, \dots, c_{r-1} \\ &\quad + \# \text{ ways to get } n - c_r \text{ using } c_1, \dots, c_r \end{aligned}$$

- Otherwise we simply have:

$$\# \text{ ways to get } n \text{ using } c_1, \dots, c_r = \# \text{ ways to get } n \text{ using } c_1, \dots, c_{r-1}$$

4.3 An Algorithmic Idea

The above observation leads to the following. Suppose we have coin denominations $[c_1, \dots, c_r]$ and wanted to know the number of ways to obtain n cents using any combinations of these denominations.

Suppose we have an array A indexed 0 through n and we have some $k < n$ such that:

- $A[0], \dots, A[k]$ contain the number of ways to obtain 0 through k cents using any denominations from c_1 through c_r .
- $A[k+1], \dots, A[n]$ contain the number of ways to obtain 0 through k cents using any denominations from c_1 through c_{r-1} .

Suppose we wish to update $A[k+1]$ so that it contains the number of ways to obtain $k+1$ using any denominations from c_1 through c_r .

From the above rule we immediately see that:

- If $n \geq c_r$ then we update it as follows:

$$A[k+1] := A[k+1] + A[k+1 - c_r]$$

- Otherwise we leave it alone.

4.4 An Actual Algorithm

Our algorithmic approach will emerge from this idea. We will start with such an array and pre-load it with the number of ways to achieve each of $0, 1, \dots, n$ using no coins. Then we will update it with the number of ways to do so using just c_1 , then we will update it with the number of ways to do so using c_1 and/or c_2 , and so on, until we are done.

On to the algorithm!

Suppose we have coin denominations $C = [c_1, \dots, c_r]$ and we wish to find the number of ways to obtain n cents using any denomination from c_1 through c_r .

We first assign an array A indexed 0 through n as follows:

$$A = [1, 0, 0, \dots, 0]$$

In this array, $A[i]$ tells us the number of ways to obtain 0 through n cents using no coins at all.

We then iterate over the denominations and update accordingly. Here is the pseudocode:

```
function coincount(C,n):
    A = [1,0,0,...,0]
    for c in C:
        for i = 1 to n:
            if i >= c:
                A[i] = A[i] + A[i-c]
            end if
        end for
    end for
end function
```

Example 4.1. Let us walk through this with $n=10$ and $C = [1, 5, 10]$.

We assign:

$$A = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$

Observe that A now contains the number of ways to obtain 0 through 10 using no coins.

We assign $c = 1$ and pass through $i = 1, 2, \dots, 10$ yielding:

$$A = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]$$

Observe that A now contains the number of ways to obtain 0 through 10 using just 1-cent coins.

We assign $c = 5$ and pass through $i = 1, 2, \dots, 10$ yielding:

$$A = [1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3]$$

Observe that A now contains the number of ways to obtain 0 through 10 using 1- and 5-cent coins.

We assign $c = 10$ and pass through $i = 1, 2, \dots, 10$ yielding:

$$A = [1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 4]$$

Observe that A now contains the number of ways to obtain 0 through 10 using 1-, 5-, and 10-cent coins.

Example 4.2. Let us walk through this with $n=15$ and $C = [2, 5, 7]$.

We assign:

$$A = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$

Observe that A now contains the number of ways to obtain 0 through 15 using no coins.

We assign $c = 2$ and pass through $i = 1, 2, \dots, 15$ yielding:

$$A = [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0]$$

Observe that A now contains the number of ways to obtain 0 through 15 using just 2-cent coins.

We assign $c = 5$ and pass through $i = 1, 2, \dots, 15$ yielding:

$$A = [1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]$$

Observe that A now contains the number of ways to obtain 0 through 15 using 2- and 5-cent coins.

We assign $c = 7$ and pass through $i = 1, 2, \dots, 15$ yielding:

$$A = [1, 0, 1, 0, 1, 1, 1, 2, 1, 2, 1, 2, 2, 2, 3, 2]$$

Observe that A now contains the number of ways to obtain 0 through 15 using 2-, 5-, and 7-cent coins.

4.5 Time Complexity

We have not formally started talking about time complexity but for now it is absolutely worth simply mentioning that the algorithm involves two nested loops such that the innermost body iterates $\text{length}(C)(n + 1)$ time.

Thus intuitively the time required increases linearly as a function of either n or the number of denominations available.

CMSC 351: Big Notation

Justin Wyss-Gallifent

May 25, 2023

1	Inspiration	2
2	The Bigs	2
2.1	Big-O Notation	2
2.2	Big-Omega and Big-Theta Notations	4
2.3	All Together	5
3	A Limit Theorem	7
4	Common Functions	9
5	Intuition	9
6	Additional Facts	10
6.1	Use of n vs x	10
6.2	Cautious Comparisons	10
7	Thoughts, Problems, Ideas	11

1 Inspiration

Suppose two algorithms do exactly the same thing to lists of length n . We find out that the time they take in seconds is as follows. Note that these are just made up!

n	$A_1(n)$	$A_2(n)$
10	6	1
20	12	6
30	18	17
40	24	25
50	28	40
60	30	63
70	38	82
80	45	109
90	50	140
100	59	190

Observe that Algorithm 2 is better (faster) up until about $n = 40$, and then Algorithm 1 is better.

But can we formalize this more, both the comparison and the values themselves?

It turns out that the values satisfy:

$$0.4n \leq A_1(n) \leq 0.6n$$

and:

$$0.01n^2 \leq A_2(n) \leq 0.02n^2$$

Although we don't have an exact knowledge about other values we do certainly have a more rigorous way not only of comparing the two algorithms but of understanding each algorithm independently.

For example we can say that if $n = 150$ then Algorithm 2 will take at most $0.02(150)^2 = 450$ seconds. In this case we have an upper bound which is a multiple of n^2 .

Our goal is to formalize these notions.

2 The Bigs

2.1 Big-O Notation

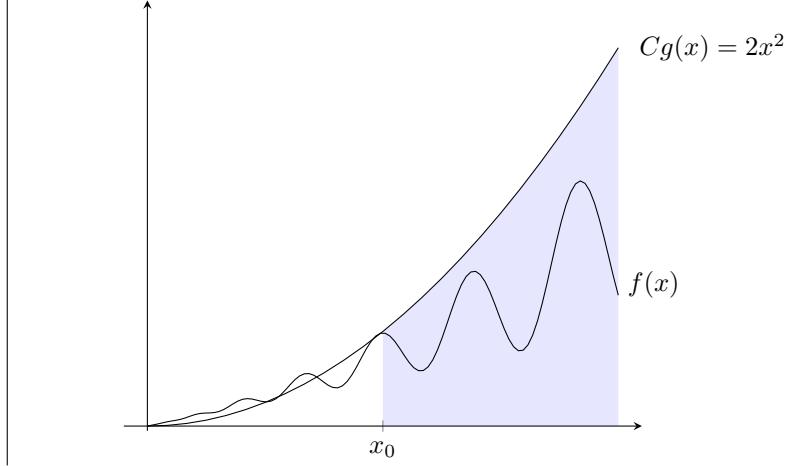
Recall the definition:

Definition 2.1.1. We say that:

$$f(x) = \mathcal{O}(g(x)) \text{ if } \exists x_0, C > 0 \text{ such that } \forall x \geq x_0, f(x) \leq Cg(x)$$

We think of this as stating that *eventually* $f(x)$ is smaller than some constant multiple of $g(x)$.

Example 2.1. For example, here $f(x) = \mathcal{O}(x^2)$ with $C = 2$ and x_0 as shown:



Note 2.1.1. There's frequently (but not always) a trade-off in that if C is large then x_0 might be smaller, or vice-versa. In light of this note that "eventually" could mean for a very large x_0 .

Example 2.2. It's true that $42000x \lg x = \mathcal{O}(x^2)$ with $C = 10$ because eventually $42000x \lg x \leq 10x^2$. However "eventually" in this case means $x_0 \approx 67367$. In other words this is the smallest x_0 such that if $x \geq x_0$ then $42000x \lg x \leq 10x^2$.

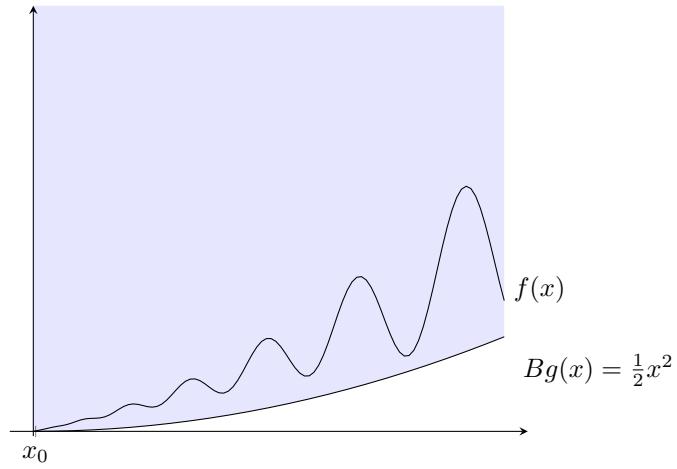
2.2 Big-Omega and Big-Theta Notations

We can extend upon this with:

Definition 2.2.1. We have:

$$f(x) = \Omega(g(x)) \text{ if } \exists x_0, B > 0 \text{ such that } \forall x \geq x_0, f(x) \geq Bg(x)$$

Example 2.3. For example, here $f(x) = \Omega(x^2)$ with $B = \frac{1}{2}$ and x_0 as shown:

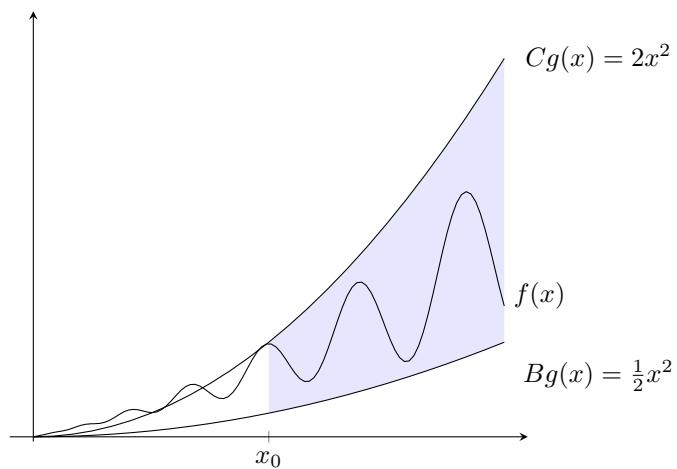


and with:

Definition 2.2.2. We have:

$$f(x) = \Theta(g(x)) \text{ if } \exists x_0, B > 0, C > 0 \text{ such that } \forall x \geq x_0, Bg(x) \leq f(x) \leq Cg(x)$$

Example 2.4. For example, here $f(x) = \Theta(x^2)$ with $B = \frac{1}{2}$ and $C = 2$ and x_0 as shown:



2.3 All Together

The basic idea is that \mathcal{O} provides an upper bound for $f(x)$, Ω provides a lower bound and Θ provides a tight bound. Therefore $f(x) = \Theta(g(x))$ if and only if $f(x) = \mathcal{O}(g(x))$ and $f(x) = \Omega(g(x))$.

Moreover observe that $\Theta \Rightarrow \mathcal{O}$ and $\Theta \Rightarrow \Omega$ but the converses are false.

Example 2.5. We show: $3x \lg x + 17 = \mathcal{O}(x^2)$

Consider the expression:

$$3x \lg x + 17$$

Note two things:

- If $x > 0$ then $\lg x < x$.
- If $x \geq \sqrt{17} = 4.1231\dots$ then $x^2 \geq 17$.

Thus if $x \geq 5$ both of these are true and we have:

$$3x \lg x + 17 \leq 3x(x) + x^2 = 4x^2$$

Thus $x_0 = 5$ and $C = 4$ works.

Note: It's not necessary to pick an integer value of x_0 here. I just did it because it's pretty. Using $x_0 = \sqrt{17}$ would have been fine too.

Example 2.6. We show: $\frac{100}{x^2} + x^2 \lg x = \mathcal{O}(x^3)$

Consider the expression:

$$\frac{100}{x^2} + x^2 \lg x$$

Note two things:

- If $x > 0$ then $\lg x < x$.
- If $x \geq 10$ then $x^2 \geq 100$ and then $\frac{100}{x^2} \leq 1 < x < x^3$.

Thus if $x \geq 10$ both of these are true and we have:

$$\frac{100}{x^2} + x^2 \lg x = \mathcal{O}(x^3) \leq x^3 + x^3 = 2x^3$$

Thus $x_0 = 10$ and $C = 2$ works.

Example 2.7. We show: $0.001x \lg x + 0.0001x - 42 = \Omega(x)$

Consider the expression:

$$0.001x \lg x - 42$$

Note that if $x \geq 2$ then $\lg x \geq 1$ and then:

$$0.001x \lg x - 42 \geq 0.001x - 42$$

This is a line with slope 0.001 and any line with smaller slope will eventually be below it. For example the line $0.0001x$ is below it when:

$$\begin{aligned} 0.001x - 42 &\geq 0.0001x \\ 0.0009x &\geq 42 \\ x &\geq \frac{42}{0.0009} = 46666.66... \end{aligned}$$

Thus if we have $x \geq 46666.66...$ then:

$$0.001x \lg x - 42 \geq 0.001x - 42 \geq 0.0001x$$

Thus $x_0 = 46667$ and $B = 0.0001$ works.

Example 2.8. We show: $10x \lg x + x^2 = \Theta(x^2)$

Consider the expression:

$$10x \lg x + x^2$$

Observe that for all $x \geq 1$ we have $\lg x > 0$ and hence:

$$10x \lg x + x^2 \geq x^2$$

And we have:

$$10x \lg x + x^2 \leq 10x(x) + x^2 = 11x^2$$

thus $x_0 = 1$, $B = 1$ and $C = 11$ works.

For simple polynomials there's very little work to show Θ .

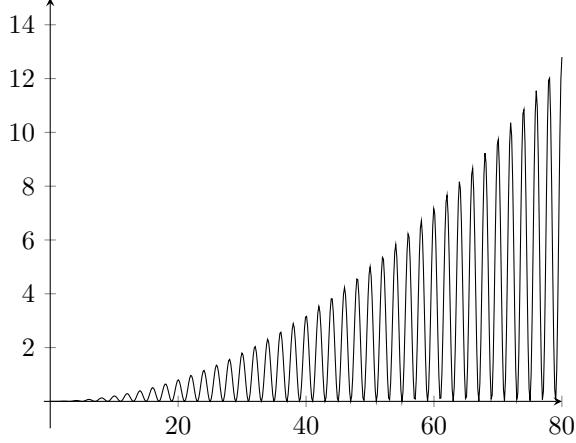
Example 2.9. Observe that $3x^2 = \Theta(x^2)$ because $x_0 = 0$ and $B = C = 3$ works.

Example 2.10. Consider $f(x) = 2x^2 - x$. Note that $2x^2 - x \leq 2x^2$ and

$2x^2 - x \geq 2x^2 - x^2 = 1x^2$ for $x \geq 1$ so that $x_0 = 1, B = 1, C = 2$ works for $2x^2 - x = \Theta(x^2)$.

Example 2.11. Consider $f(x) = 0.001x^2(1 + \cos(x\pi))$.

The graph of this function is:



The local maxima occur at $x = 0, 2, 4, 6, 8, \dots$ and the local minima occur at $x = 1, 3, 5, 7, 9, \dots$

Note that $0.001x^2(1 + \cos(x\pi)) \leq 0.001x^2(1 + 1) = 0.002x^2$ for $x \geq 0$ so that $f(x) = \mathcal{O}(x^2)$. However in addition note that when $x \in \mathbb{Z}$ is odd that $0.001x^2(1 + \cos(x\pi)) = 0.001x^2(1 - 1) = 0$ so that there is no $B > 0$ such that for large enough x we have $f(x) \geq Bx^2$. Consequently $f(x) \neq \Omega(x^2)$ and thus $f(x) \neq \Theta(x^2)$.

You might ask if there is any $g(x)$ such that $f(x) = \Theta(g(x))$ and the short answer is - yes, of course, because $f(x) = \Theta(f(x))$ but this is generally unsatisfactory. We are looking for *useful* $g(x)$ which help us understand $f(x)$. Saying essentially that $f(x)$ grows at the same rate as itself doesn't help much!

3 A Limit Theorem

There are a few alternative ways of proving \mathcal{O} , Ω and Θ . Here is one. Note that the following are unidirectional implications!

Theorem 3.0.1. Provided $\lim_{n \rightarrow \infty} f(n)$ and $\lim_{n \rightarrow \infty} g(n)$ exist (they may be ∞) then we have the following:

- (a) If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \neq 0, \infty$ then $f(n) = \Theta(g(n))$.

Note: Here we also have $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$ as well.

- (b) If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \neq \infty$ then $f(n) = \mathcal{O}(g(n))$.

(c) If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \neq 0$ then $f(n) = \Omega(g(n))$.

Proof. Here's a proof of (b). Suppose we have:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L \neq \infty$$

By the definition of the limit this means:

$$\forall \epsilon > 0, \exists n_0 \text{ st } n \geq n_0 \implies L - \epsilon < \frac{f(n)}{g(n)} < L + \epsilon$$

Specifically, if $\epsilon = 1$ if we take only the right inequality this tell us that:

$$\exists n_0 \text{ st } n \geq n_0 \implies \frac{f(n)}{g(n)} < L + 1$$

When $<$ is true, so is \leq so this means that when $n \geq n_0$ we have:

$$f(n) < (L + 1)g(n)$$

This is exactly the definition of \mathcal{O} using n_0 and $C = L + 1$. \mathcal{QED}

Example 3.1. Observe that:

$$\lim_{n \rightarrow \infty} \frac{n \ln n}{n^2} = \lim_{n \rightarrow \infty} \frac{\ln n}{n} = \lim_{n \rightarrow \infty} \frac{1/n}{1} = 0$$

Thus $n \ln n = \mathcal{O}(n^2)$.

The following example is far easier to prove using this theorem than from the definition of \mathcal{O} :

Example 3.2. We have $50n^{100} = \mathcal{O}(3^n)$.

Observe that 100 applications of L'hôpital's Rule yields:

$$\lim_{n \rightarrow \infty} \frac{50n^{100}}{3^n} = \lim_{n \rightarrow \infty} \frac{(100)(99)\dots(1)(50)}{(\ln 3)^{100} 3^n} = 0$$

The result follows.

4 Common Functions

In all of this you might wonder why we're always comparing functions to things like n^2 or $n \lg n$. We typically wouldn't say, for example, that $f(n) = \Theta(n^2 + 3n + 1)$.

The reason for this is that computer scientists have settled on a collection of “simple” functions, functions which are easy to understand and compare, and big-notation almost always uses these functions.

Here are a list of some of them, in order of increasing size:

$$1, \lg n, n, n \lg n, n^2, n^2 \lg n, n^3, \dots$$

To say these are “increasing size” means, formally, that any of these is \mathcal{O} of anything to the right, for example $n = \mathcal{O}(n \lg n)$ and $n^2 = \mathcal{O}(n^3)$ and so on.

There's a pattern there, that $n^k = \mathcal{O}(n^k \lg n)$ and $n^k \lg n = \mathcal{O}(n^{k+1})$, which is easy to prove.

In addition we have, for every positive integers k and $b \geq 2$:

$$n^k = \mathcal{O}(b^n)$$

These can be proved with the Limit Theorem.

Lastly, all of the above are $\mathcal{O}(n!)$, which is about the biggest one we ever encounter in this class.

5 Intuition

It's good to have some intuition here, and of course the following can be proved rigorously on a case-by-case basis, and you should try.

In essence the “largest term” always wins in a Θ sense. So for example if we have:

$$f(n) = n^2 - n \lg n + n + 1$$

The “largest term” is the n^2 so that wins and we can say:

$$n^2 - n \lg n + n + 1 = \Theta(n^2)$$

Likewise, for example:

$$n^2 \lg n + n \lg n - 100 = \Theta(n^2 \lg n)$$

6 Additional Facts

6.1 Use of n vs x

These statements about function of x are often phrased using the variable n instead. Typically this is done when n can only take on positive integers.

In this case it can still be helpful to draw the functions as if n could be any real number, otherwise we're left drawing a bunch of dots. In some cases though, like $f(n) = n!$, it's not entirely clear how we would sketch this for $n \notin \mathbb{Z}$.

Otherwise the calculations are basically identical, noting that the cutoff value n_0 must be a positive integer.

6.2 Cautious Comparisons

This notation brings a certain ordering to functions. Observe for example that $1000000 + n \lg n = \mathcal{O}(n^2)$ because *eventually* $1000000 + n \lg n \leq Cn^2$ for some $C > 0$. Thus we intuitively think of n^2 as “larger than” $1000000 + n \lg n$. However we have to make sure we understand that we really mean that a constant multiple of n^2 is eventually larger than $1000000 + n \lg n$.

Example 6.1. For example *eventually* $1000000 + n \lg n \leq 17n^2$ but *eventually* here means for $n \geq n_0 = 243$.

We should also note that it's common to believe that if one function $g(x)$ has a larger derivative than another function $f(x)$ that eventually $f(x) \leq g(x)$. This is false.

7 Thoughts, Problems, Ideas

1. It's tempting to think that if $f(x)$ and $g(x)$ are both positive functions defined on $[0, \infty)$ with positive derivatives and if $f'(x) > g'(x)$ for all x then eventually $f(x)$ will be above $g(x)$. Show that this isn't true. Give explicit functions and sketches of those functions.
2. Find the value x_0 (approximately) which justifies $1234 + 5678x \lg x = \mathcal{O}(x^2)$ with $C = 42$. Use any technology you like but explain your process.
3. Find the value x_0 (approximately) which justifies $4758x + 789x^2 \lg x = \mathcal{O}(x^3)$ with $C = 17$. Use any technology you like but explain your process.
4. Find the value x_0 (approximately) which justifies $0.00357x^{2.01} \lg x = \Omega(x^2)$ with $C = 100$. Use any technology you like but explain your process.
5. Show from the definition that $5x^2 + 10x \lg x + \lg x = \mathcal{O}(x^2)$.
6. Show from the definition that:

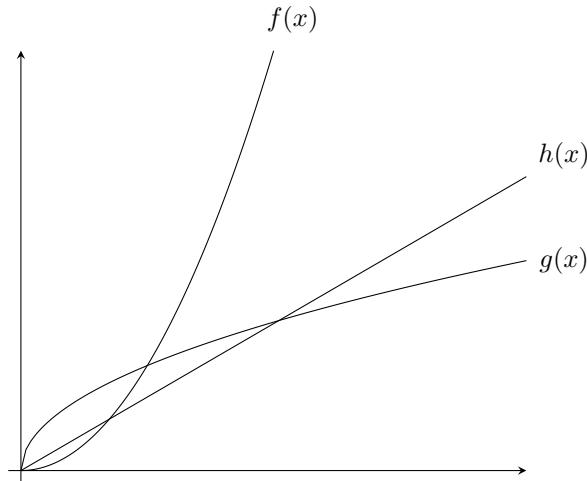
$$\sum_{i=0}^{n-1} \left[2 + \sum_{j=i}^{n-1} 3 \right] = \mathcal{O}(n^2)$$

7. Show from the definition that $(475632)2^n = \mathcal{O}(5^n)$.
8. Show from the definition that $x + x \log x = \Theta(x \lg x)$.
9. Show from the definition that:

$$\sum_{i=0}^{n-1} \left[1 + i + \frac{1}{i+1} \right] = \Theta(n^2)$$

10. Show from the definition that $x^3 + 5x + \ln x + 100 = \Omega(x^2)$.
 11. Show from the definition that:
- $$\sum_{i=0}^n [i^2 + 3i] = \Omega(n^3)$$
12. Show from the definition that $5^n \neq \mathcal{O}(2^n)$.
 13. Show that $5000 + 6000n^{1500} = \mathcal{O}(3^n)$.
 14. Show that $5^n = \Omega(n^{1000})$
 15. Show that $(0.001)5^n = \Omega(857n^{999})$
 16. Show from the definition that $\log_2 n = \Theta(\log_5 n)$ and $\log_5 n = \Theta(\log_2 n)$.
 17. Generalize the above problem. In other words prove that $\Theta(\log_b x) = \Theta(\log_c x)$ for any two bases $b, c > 1$.
 18. In the previous question why do we need $b, c > 1$?

19. Give an example of two functions $f(x)$ and $g(x)$ which are not constant multiples of one another and which satisfy $f(x) = \mathcal{O}(g(x))$ and $g(x) = \mathcal{O}(f(x))$. Justify from the definitions.
20. Give an example of two functions $f(x)$ and $g(x)$ which are not constant multiples of one another and which satisfy $f(x) = \Omega(g(x))$ and $g(x) = \Omega(f(x))$. Justify from the definitions.
21. If $f(n) = \mathcal{O}(g(n))$ with C_0 and n_0 and $g(n) = \mathcal{O}(h(n))$ with C_1 and n_1 which constants would prove that $f(n) = \mathcal{O}(h(n))$?
22. The functions $f(x) = \log_b x$ for $b > 1$ and $g(x) = x^c$ for $0 < c < 1$ have similar shapes for increasing x . however $f(x) = \mathcal{O}(g(x))$ always. Prove this.
 Note: This underlies the important fact that roots always grow faster than logarithms.
23. Consider the following three functions:



- (a) Write down as many possibilities as you can which satisfy $\square = \mathcal{O}(\diamond)$ where $\square, \diamond \in \{f(x), g(x), h(x)\}$.
- (b) Write down as many possibilities as you can which satisfy $\square = \Omega(\diamond)$ where $\square, \diamond \in \{f(x), g(x), h(x)\}$.
- (c) Write down as many possibilities as you can which satisfy $\square = \Theta(\diamond)$ where $\square, \diamond \in \{f(x), g(x), h(x)\}$.

CMSC 351: Rigorous Time Summary

Justin Wyss-Gallifent

September 7, 2023

1	Introduction	2
2	Time Complexity	2
3	Assignments Take Time	2
4	For Loops Take Time	3
5	While Loops Take Time	4
6	Conditionals Take Time	5
7	Combining	5
8	Best and Worst-Case	6
9	Summary on Time Consideration	6

1 Introduction

Exactly how to measure the time requirements of an algorithm is often a source of confusion so here's a brief but hopefully comprehensive explanation.

2 Time Complexity

When we discuss time complexity of code in the simplest case the code will depend upon some n which could be the length of a list, the number of times a loop iterates, etc. Our goal is to imagine that we could write down a function $T(n)$ which tells us how much time the code takes for any given n and then find a simple $f(n)$ so that $T(n) = \Theta(f(n))$ when possible, and \mathcal{O} or Ω perhaps when that's all we need or all we can get.

3 Assignments Take Time

Assignments take time.

Formally speaking each assignment takes its own time:

```
a = 0   c1
b = 0   c2
c = 0   c3
```

The total time is $c_1 + c_2 + c_3 = \Theta(1)$.

Of course we could argue that each assignment takes the same amount of time. Whether this is actually true or not might be system/hardware/implementation dependent but for time complexity it doesn't matter.

```
a = 0   c1
b = 0   c1
c = 0   c1
```

The total time is $3c_1 = \Theta(1)$.

Of course in light of that we could just lump it all together and suggest that all three lines together take some constant time:

```
a = 0
b = 0
c = 0
```

$$\left. \right\} c_1$$

The total time is $c_1 = \Theta(1)$.

And even more informally we might just say that the actual constant doesn't matter so we'll call it time 1:

```
a = 0
b = 0
c = 0
```

$$\left. \right\} 1$$

The total time is $1 = \Theta(1)$.

Note 3.0.1. We're not saying some are right and some are wrong, it's all a question of what we're measuring and how much detail we desire.

4 For Loops Take Time

Formally speaking everything takes time, including the maintenance associated to loops. Consider this pseudocode, assuming n is given:

```
sum = 0           c1
for i = 1 to n   c2 a total of n times (this is the maintenance line)
    // Comment  0 a total of n times
end
```

We have an initial time of c_1 and then technically speaking the **for** loop does n assignments at some time c_2 . Note that this is not the body of the **for** statement but rather this is the overhead consisting of the maintenance of the loop; assigning i , updating, and so on.

The body of the loop is a comment which takes 0 time.

The total time is $c_1 + c_2n + (0)n = \Theta(n)$.

Now then, suppose we add something inside the loop:

```
sum = 0           c1
for i = 1 to n   c2 a total of n times
    sum = sum + i c3 a total of n times
end
```

Now we have time cost $c_1 + c_2n + c_3n = \Theta(n)$.

However here is when computer scientists get understandably sloppy. Since the maintenance is just adding constant time c_2 to the body, and since constant time doesn't alter time complexity, generally we will ignore the contribution of the maintenance line.

```
sum = 0           c1
for i = 1 to n   Iterates n times
    sum = sum + i c3 a total of n times
end
```

The total time is $c_1 + c_3n = \Theta(n)$.

Of course this is no longer entirely accurate if we're assuming that the maintenance line takes time but when we're analyzing time complexity it's okay. If we're doing explicit time totals and if we wish to factor in that maintenance line time then it's not and we need to keep the c_2 in there.

In addition since the **for** loop will take some time the assignment before it, contributing constant time, could even be ignored:

```

sum = 0           Meh, contributes constant time overall
for i = 1 to n   Iterates  $n$  times
    sum = sum + i   $c_3$  a total of  $n$  times
end

```

The total time is $c_3n = \Theta(n)$.

Note 4.0.1. We're not saying some are right and some are wrong, it's all a question of what we're measuring and how much detail we desire.

5 While Loops Take Time

The same is true for `while` loops. Consider this pseudocode:

```

sum = 1            $c_1$ 
i = 1             $c_2$ 
while i <= n     $c_3$  a total of  $n$  times (this is the maintenance line)
    i = i + 1     $c_4$  a total of  $n$  times
end

```

The total time is $c_1 + c_2 + c_3n + c_4n = \Theta(n)$.

Again we can be a bit more sloppy, letting the `while` maintenance fold into the body of the loop and assuming all assignments take the same time:

```

sum = 1            $c_1$ 
i = 1             $c_1$ 
while i <= n    Iterates  $n$  times
    i = i + 1     $c_1$  a total of  $n$  times
end

```

The total time is $c_1 + c_1 + n(c_1) = \Theta(n)$.

Or even more sloppy:

```

sum = 1           } $c_1$ 
i = 1
while i <= n    Iterates  $n$  times
    i = i + 1     $c_1$  a total of  $n$  times
end

```

The total time is $c_1 + n(c_1) = \Theta(n)$.

Or even more:

```

sum = 1           Meh
i = 1            Meh
while i <= n    Iterates  $n$  times
    i = i + 1     $c_1$  a total of  $n$  times
end

```

The total time is $n(c_1) = \Theta(n)$.

Note 5.0.1. We're not saying some are right and some are wrong, it's all a question of what we're measuring and how much detail we desire.

6 Conditionals Take Time

The comparisons in a conditional also formally take time. Consider this pseudocode, where **a** and **b** are assumed to be given.

```
if a<b           c1
    print('hi')   c2
end
```

Formally speaking the time this requires is:

- If **a**<**b** passes then the time is $c_1 + c_2$.
- If **a**<**b** fails then the total time is c_1 .

Often then we'll just say the whole thing is constant time:

```
if a<b           c1
    print('hi')   c2
end
```

However this doesn't mean we can just ignore the comparison all the time, it depends in delicate situations on the body of the conditional.

```
if a<b           c1
    UNKNOWN     Without understanding this, can't get rid of c1
end
```

In the next section we'll see what happens when we start nesting things.

Note 6.0.1. We're not saying some are right and some are wrong, it's all a question of what we're measuring and how much detail we desire.

7 Combining

Combining is when we need to be careful and attentive especially when we want to be sloppy and especially with regards to conditionals which contain a body operating at non-constant time.

Consider this pseudocode:

```
if a<b           c1
    sum = 0         c2
    for i = 1 to n   c3 a total of n times
        sum = sum + i  c4 a total of n times
    end
end
```

Formally the conditional check is c_1 time no matter what and the assignment is c_2 . But then observe:

- If `a < b` passes then the total time is $c_1 + c_2 + c_3n + c_4n = \Theta(n)$.
- If `a < b` fails then the total time is $c_1 = \Theta(1)$.

In this case we can't blindly ignore the c_1 because if we did, then if `a < b` fails, then we'd be suggesting that this pseudocode takes 0 time.

Of course if this conditional had more stuff before (or after) it:

```

print('hi')           c5
if a < b            c1
    sum = 0          c2
    for i = 1 to n   c3 a total of n times
        sum = sum + i c4 a total of n times
    end
end

```

The `print` statement is contributing constant time and so removing the c_1 does not result in time 0 when `a < b` fails and therefore does not affect time complexity.

8 Best and Worst-Case

In the previous example:

```

print('hi')           c5
if a < b            c1
    sum = 0          c2
    for i = 1 to n   c3 a total of n times
        sum = sum + i c4 a total of n times
    end
end

```

Many resources (all over the internet) would simply say that this is $\mathcal{O}(n)$ but the truth is a bit more nuanced. The reality is:

- In a best-case scenario `a < b` fails and the time is $c_5 + c_1$ which is in fact $\Theta(1)$. It's also $\mathcal{O}(1)$ and $\Omega(1)$. Of course it is also $\mathcal{O}(n)$ but this is being pretty liberal.
- In a worst-case scenario `a < b` passes and the time is $c_5 + c_1 + c_2 + c_3n + c_4n$ which is in fact $\Theta(n)$. It's also $\mathcal{O}(n)$ and $\Omega(n)$.
- If anyone says casually that this is $\mathcal{O}(n)$ what they mean is that in the worst case it is $\mathcal{O}(n)$.

9 Summary on Time Consideration

So how far can we actually simplify if we're interested only in time complexity? In other words, what do we need to keep?

- With loops, provided that the body is guaranteed to take nonzero time we can ignore the maintenance line.

Example 9.1. In this example:

```
for i = 1 to n^2    c1 iterates  $n^2$  times
    print(i)        c2 iterates  $n^2$  times
end
```

We can ignore the c_1 :

```
for i = 1 to n^2
    print(i)        c2 iterates  $n^2$  times
end
```

The (best, worst, and average) time complexity is $\Theta(n^2)$:

- Any line that takes constant time can be ignored provided there is other adjacent code which takes nonzero time.

Example 9.2. In this example:

```
stuff = 1          c1
for i = 1 to n^2  c2 iterates  $n^2$  times
    print(i)        c3 iterates  $n^2$  times
end
```

We can ignore the c_2 as before and in addition the c_1 :

```
stuff = 1
for i = 1 to n^2
    print(i)        c3 iterates  $n^2$  times
end
```

The (best, worst, and average) time complexity is $\Theta(n^2)$:

- In a worst-case scenario a conditional is assumed to be true and the entire conditional can be replaced by the time that the body takes.

Example 9.3. In this example:

```
if a+b<c
    for i = 1 to n      c2 iterates  $n$  times
        print('spicy')  c2 iterates  $n$  times
    end
end
```

For worst-case time complexity we can ignore the c_2 as noted earlier and in addition we can ignore the c_1 :

```
if a+b<c
    for i = 1 to n
        print('spicy')  c2 iterates  $n$  times
    end
end
```

| The worst-case time complexity is $\Theta(n)$.

- In a best-case scenario if we can guarantee that there is an input for which the conditional fails then the entire conditional is constant time for the conditional check so we can't remove the conditional check carelessly.

Example 9.4. In this example:

```
if a+b<c           c1
    for i = 1 to n   c2 iterates n times
        print('spicy') c3 iterates n times
    end
end
```

For best-case time complexity if we know for sure that there can be a , b , and c in some input for which $a+b < c$ fails then we cannot ignore the c_1 but we can ignore everythign else:

```
if a+b<c           c1
    for i = 1 to n
        print('spicy')
    end
end
```

The best-case time complexity is $\Theta(1)$.

However if another command takes care of that, then we can:

Example 9.5. In this example:

```
print('Hi')          c4
if a+b<c           c1
    for i = 1 to n   c2 iterates n times
        print('spicy') c3 iterates n times
    end
end
```

For best-case time complexity we can ignore the c_2 and either the c_1 or the c_4 :

```
print('Hi')          c4
if a+b<c
    for i = 1 to n
        print('spicy')
    end
end
```

The best-case time complexity is $\Theta(1)$.

- In an average-case scenario it's much more delicate.

Example 9.6. Consider this example for average-case time complexity:

```

if a+b<c           c1
    for i = 1 to n   c2 iterates n times
        print('spicy')  c3 iterates n times
    end
end

```

we can ignore the c_2 as noted earlier, thus we can see this as:

```

if a+b<c           c1
    for i = 1 to n
        print('spicy')  c3 iterates n times
    end
end

```

Suppose the input is such that half the time $a+b < c$ passes and half the time it fails. When it fails the total time is c_1 and when it passes the total time is $c_1 + c_3n$ so the average total time is $c_1 + \frac{1}{2}c_3n$ which is $\Theta(n)$.

CMSC 351: Maximum Contiguous Sum

Justin Wyss-Gallifent

February 5, 2024

1	Introduction	2
1.1	Why it's Interesting	2
2	Brute Force Method	3
2.1	Introduction	3
2.2	Pseudocode with Time Complexity Notes	3
3	Divide-and-Conquer	5
3.1	Introduction	5
3.2	Pseudocode with Time Complexity Notes	6
3.3	Straddling Sum Example	8
4	Kadane's Algorithm	9
4.1	Introduction	9
4.2	Mathematics	9
4.3	Pseudocode with Time Complexity Notes	10
4.4	Example Walk-Through	10
5	Comment	10
6	Thoughts, Problems, Ideas	11
7	Python Code with Output	13
7.1	Brute Force	13
7.2	Naïve Method	15
7.3	Divide-and-Conquer	17
7.4	Kadane's Algorithm	19

1 Introduction

Given a list of integers find the contiguous sublist with maximum sum and return that maximum sum.

Example 1.1. For example in the list given here:

[-9, 3, 1, 1, 4, -2, -8]

There are many contiguous sublists each with a sum, for example:

- [-9,3,1] has sum -5
- [1,1,4,-2] has sum 4
- ...and so on...

Out of all of these the maximum sum is 9 arising from the sublist [3,1,1,4].

Two important notes:

- The contiguous sublist must be nonempty, meaning we can't take a sublist of length 0.
- The maximum contiguous sum may be negative in the case that the list contains only negative numbers.

1.1 Why it's Interesting

Beyond any real-world applications solving this problem is a great approach to many of the various approaches we'll be seeing through the course, and so doing it early is like putting out a little taste of a few things.

One real-world application would be if a list A contains the daily increase and decrease information for a stock and we wished to find the set of days over which the stock experienced the largest net growth. In the introductory example if each entry contains the profit (or loss) corresponding to a day of the week then we can say that the period of maximum profit was four days long and yielded a profit of 9.

2 Brute Force Method

2.1 Introduction

We can do this via brute force. We simply examine all sums starting at all indices and take the maximum.

2.2 Pseudocode with Time Complexity Notes

Here is our pseudocode with detailed time assignments. As we have commented on before we really don't need all these but we're keeping them here for practice:

```
\\" PRE: A is a list of length n.  
max = A[0]                                c1  
for i = 0 to n-1                            n times  
    sum = 0                                  c1  
    for j = i to n-1                        n - 1 - i + 1 times  
        sum = sum + A[j]                    c1  
        if sum > max  
            max = sum  
        end  
    end  
end  
\\" POST: max is the maximum sum.
```

The total time complexity can then be calculated as a nested sum:

$$\begin{aligned}
T(n) &= c_1 + \sum_{i=0}^{n-1} \left[c_1 + \sum_{j=i}^{n-1} (c_1 + c_2) \right] \\
&= c_1 + \sum_{i=0}^{n-1} \left[c_1 + (c_1 + c_2) \sum_{j=i}^{n-1} 1 \right] \\
&= c_1 + \sum_{i=0}^{n-1} [c_1 + (c_1 + c_2)(n - 1 - i + 1)] \\
&= c_1 + \sum_{i=0}^{n-1} [c_1 + (c_1 + c_2)n] - \sum_{i=0}^{n-1} (c_1 + c_2)i \\
&= c_1 + [c_1 + (c_1 + c_2)n] \sum_{i=0}^{n-1} 1 - (c_1 + c_2) \sum_{i=0}^{n-1} i \\
&= c_1 + [c_1 + (c_1 + c_2)n] (n) - (c_1 + c_2) \left(\frac{(n-1)n}{2} \right) \\
&= c_1 + c_1 n + (c_1 + c_2)n^2 - \frac{1}{2}(c_1 + c_2)n^2 + \frac{1}{2}(c_1 + c_2)n \\
&= \frac{1}{2}(c_1 + c_2)n^2 + \frac{1}{2}(3c_1 + c_2)n + c_1 \\
&= \Theta(n^2)
\end{aligned}$$

This is perhaps the default go-to method because it's very easy to understand. However $\Theta(n^2)$ time is less than ideal. Of course for small n it's fine, and therefore arguably useful, however.

Using a similar argument to extreme brute force we have best- and average-case time complexity also $\Theta(n^2)$.

3 Divide-and-Conquer

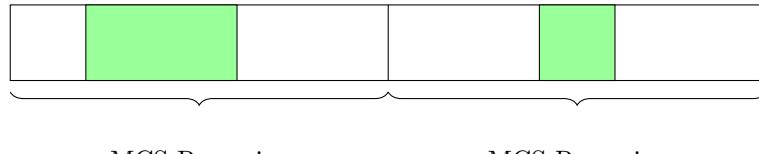
3.1 Introduction

Interestingly we can apply a divide-and-conquer approach using a recursive algorithm. Basically we split the list in half and then do three things: We find the MCS on the left side (via a recursive call), on the right side (via a recursive call), and the one that straddles the middle (not a recursive call). The bottom of the recursion occurs when the list is length 1.

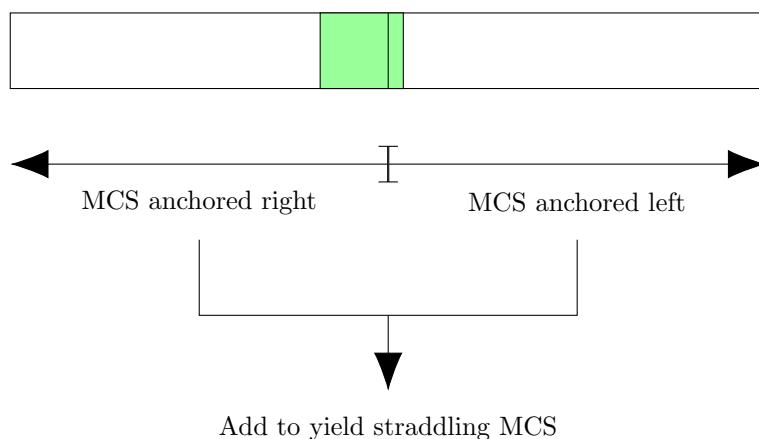
The MCS straddling the middle is calculated by setting $C = \lfloor (L + R)/2 \rfloor$ where L and R are the left and right indices of the (sub)list being managed and then splitting the list in half, the left half being $A[0], \dots, A[C]$ and the right half being $A[C + 1], \dots, A[n - 1]$ then taking the MCS which includes both $A[C]$ and $A[C + 1]$ (and perhaps more to the left and right).

Graphically:

- The two recursive MCS calls look like:



- And the straddling MCS looks like:



We then take the maximum of these three to get the overall maximum.

3.2 Pseudocode with Time Complexity Notes

For the time complexity we'll drop all the constant times that we safely can. We keep the c for the `return` because it's the only thing in the conditional body and the c in the iteration bodies. While it's true that these might really take different amounts of time because they are both constant this does not affect the overall time complexity.

```

function mcs(A,L,R)
    if L == R
        return(A[L])                                c
    else
        C = (L+R) // 2
        Lmax = mcs(A,L,C)
        Rmax = mcs(A,C+1,R)
        \\ Calculate the straddle max
        Lhmax = -infinity
        Lhsum = 0
        for i = C downto L                         C - L + 1 times
            Lhsum = Lhsum + A[i]
            if Lhsum > Lhmax
                Lhmax = Lhsum
            end
        end
        Rhmax = -infinity
        Rhsum = 0
        for i = C+1 to R                          R - (C + 1) + 1 = R - C times
            Rhsum = Rhsum + A[i]
            if Rhsum > Rhmax:
                Rhmax = Rhsum
            end
        end
        Smax = Lhmax + Rhmax
        \\ Return the overall max
        return(max([Lmax,Rmax,Smax]))
    end
end
result = mcs(A,0,len(A)-1)

```

In what follows we're glossing over various floor and ceiling functions which occur during the division process. This glossing over is fairly common in algorithm analysis and in general has no impact on the result in any meaningful complexity way.

Observe that the body of the `else` statement executes $C - L + 1 + R - C = R - L + 1$ times and the body of the `if` statement executes $1 = R - L + 1$ times too. Thus we can safely say that the `if` statement takes a total time of $(R - L + 1)c$ which is equal to c multiplied by the length of the sublist being processed.

At recursion depth 0 (the initial call) there is $2^0 = 1$ list of length n so the time required is cn .

At recursion depth 1 there are $2^1 = 2$ lists of length $n/2^1$ so the time required is $2(n/2^1)c = cn$.

At recursion depth 2 there are $2^2 = 4$ lists of length $n/2^2$ so the time required is $4(n/2^2)c = cn$.

This pattern continues such that at every recursion depth the time required is nc so the critical question is - how deep does the recursion go?

Well, this continues until the recursion length of the lists are 1. If k is the maximum recursion depth then this occurs when $n/2^k = 1$ or $k = \lg n$. Thus we have recursion depths 0 through $\lg 2$ and the total time is:

$$\underbrace{cn + cn + \dots + cn}_{1 + \lg n \text{ of these}} = c(1 + n \lg n) = \Theta(n \lg n)$$

Alternately this may be approached using the Master Theorem (we'll see later). If each call requires time $T(n)$ satisfying:

$$T(n) = 2T(n/2) + \Theta(n)$$

where the $\mathcal{O}(n)$ arises when calculating the maximum contiguous sublist crosses the middle. The Master Theorem then tell us that $T(n) = \mathcal{O}(n \log n)$.

3.3 Straddling Sum Example

To clarify the straddling sum consider the list $[4, -2, 5, 1, 2, 3, -1]$ with $l = 0$ and $r = 6$. We have $c = (6 + 0)/2 = 3$ and so we calculate the left-hand max anchored at index $c = 3$:

4	-2	5	1	2	3	-1	
			1				Lhsum = 1
			5	1			Lhsum = 6
			-2	5	1		Lhsum = 4
		4	-2	5	1		Lhsum = 8

Thus we have $\text{Lhmax} = 8$.

And the right-hand max anchored at index $c + 1 = 5$:

4	-2	5	1	2	3	-1	
				2			Rhsum = 2
				2	3		Rhsum = 5
				2	3	-1	Rhsum = 4

Thus we have $\text{Rhmax} = 5$.

In total then $\text{smax} = 8 + 5 = 13$, the total max straddling the center.

Note 3.3.1. This is the first case we've seen where it's evident why divide-and-conquer confers an advantage with regards to time complexity.

It's extremely common that divide-and-conquer leads to the introduction of a logarithmic factor in the time complexity and that this logarithmic factor replaces something larger.

4 Kadane's Algorithm

4.1 Introduction

Kadane's Algorithm is a sneaky way of solving the problem in $\mathcal{O}(n)$. The basic premise is based on the idea of dynamic programming.

In dynamic programming the idea is to use previous knowledge as much as possible when iterating through a process.

4.2 Mathematics

Theorem 4.2.1. Define M_i as the maximum contiguous sum ending at and including index i for $0 \leq i \leq n - 1$. Then we have $M_0 = A[0]$ and $M_i = \max(M_{i-1} + A[i], A[i])$.

Proof. It's clear that $M_0 = A[0]$ since $A[0]$ is the only contiguous sum ending at index 0.

Consider M_i for some $1 \leq i \leq n - 1$.

Denote by C_i the set of contiguous sums ending at index i and denote by C_{i-1} the set of contiguous sums ending at index $i - 1$.

Then we have:

$$\begin{aligned} C_i &= \{x + A[i] \mid x \in C_{i-1}\} \cup \{A[i]\} \\ M_i &= \max(C_i) = \max(\{x + A[i] \mid x \in C_{i-1}\} \cup \{A[i]\}) \\ &= \max(\{x + A[i] \mid x \in C_{i-1}\}, A[i]) \\ &= \max(M_{i-1} + A[i], A[i]) \end{aligned}$$

\mathcal{QED}

What this means is that we can progress through the list, calculating the maximum contiguous sum ending at and including index $i = 1, \dots, n - 1$ using the previous maximum contiguous sum ending at and including index $i - 1$.

More programatically: We start by setting `mcs=A[0]`. Well then go through the list from `i=0, ..., n-1`. At each step we take the maximum of `mcs` and `mcs+A[i]`. We compare this with an ongoing overall maximum and keep track of the largest of these.

4.3 Pseudocode with Time Complexity Notes

Here is the pseudocode:

```
\\" PRE: A is a list of length n.  
maxoverall = A[0]  
maxendingati = A[0]  
for i = 1 to n-1  
    maxendingati = max(maxendingati+A[i],A[i])  
    maxoverall = max(maxoverall,maxendingati)  
end  
\\" POST: maxoverall is the maximum contiguous sum.
```

Note that the use of the `max` commands can be replaced by conditionals. We're just being compact here.

The single iteration of the loop makes it clear that the time complexity is $\Theta(n)$.

Note 4.3.1. Worth noting is that it's not uncommon to have algorithms in which there is a $\mathcal{O}(n)$ solution which requires some rather ingenuous consideration of the solution.

4.4 Example Walk-Through

Consider the list:

$$[2, 3, -4, 5, 1, 2, -8, 3]$$

I'll use M to indicate the maximum overall contiguous sum and M_i to indicate the maximum contiguous sum ending at and including index i .

We initiate $M = 2$ and $M_i = 2$.

Then we iterate:

- \Rightarrow When $i = 1$ we get $M_i = \max(2 + 3, 3) = 5$ and $M = \max(2, 5) = 5$.
- \Rightarrow When $i = 2$ we get $M_i = \max(5 - 4, -4) = 1$ and $M = \max(5, 1) = 5$.
- \Rightarrow When $i = 3$ we get $M_i = \max(1 + 5, 5) = 6$ and $M = \max(5, 6) = 6$.
- \Rightarrow When $i = 4$ we get $M_i = \max(6 + 1, 1) = 7$ and $M = \max(6, 7) = 7$.
- \Rightarrow When $i = 5$ we get $M_i = \max(7 + 2, 2) = 9$ and $M = \max(7, 9) = 9$.
- \Rightarrow When $i = 6$ we get $M_i = \max(9 - 8, -8) = 1$ and $M = \max(9, 1) = 9$.
- \Rightarrow When $i = 7$ we get $M_i = \max(1 + 3, 3) = 4$ and $M = \max(9, 4) = 9$.

Thus the maximum contiguous sum is 9.

5 Comment

It's worth noting that even though Kadane's Algorithm is fastest it is arguably not the most clear.

6 Thoughts, Problems, Ideas

- Fill in the details of the extreme brute force time complexity calculation for the case where all the constants are 1:

$$T(n) = 1 + \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \left[1 + \left[\sum_{k=i}^j 1 \right] + 1 \right] = (\text{fill in!}) = \Theta(n^3)$$

- Fill in the details of the naïve method time complexity calculation for the case where all the constants are 1:

$$T(n) = 1 + \sum_{i=0}^{n-1} \left[1 + \sum_{j=i}^{n-1} (1+1) \right] = (\text{fill in!}) = \Theta(n^2)$$

- Modify the pseudocode for each of the four variations so that an additional variable `maxlen` is passed and so that the sublist whose sum is returned may not have a length longer than this.
- In the divide-and-conquer algorithm we divide the list in half in the line `c=(l+r)//2`. What effect does it have if this line is replaced by, for example, `c=(l+r)//3`? Explain
- Modify the pseudocode for Kadane's Algorithm so that it finds the maximum nonnegative sum. If every element in the list is negative return `false` instead of the sum.
- Modify the pseudocode for Kadane's Algorithm so that it returns the length of the sublist which provides the maximum contiguous sum.
- What is a useful loop invariant for Kadane's Algorithm? Prove the requirement of the Loop Invariant Theorem for this loop invariant.
- Modify the pseudocode for the naïve method so that it finds the maximum sum of a rectangular subarray within an $n \times m$ array. Assuming that all constant time complexities take time 1 calculate the \mathcal{O} time complexity of your pseudocode.
- Using the same approach as divide-and-conquer write an algorithm which returns the length of the longest string of 1s in list of 0s and 1s.
- Write a clear explanation of how the divide-and-conquer approach could be modified to find the maximum sum of a rectangular subarray within an $n \times m$ array. Pseudocode isn't necessary.
- What are your thoughts on modifying Kadane's Algorithm to find to find the maximum sum of a rectangular subarray within an $n \times m$ array?
- In Kadane's Algorithm consider the line:

```
maxoverall=max(maxoverall,maxendingati)
```

In a worst-case scenario what is the maximum number of times this will change the value of `maxoverall`? Explain.

13. Suppose your list A has the property that the maximum contiguous sum should only examine sublists of length 5 or less. Modify each of the algorithms in order to take this into account.
14. Modify each algorithm so that instead of finding the maximum contiguous sum a particular value is passed and the algorithm returns True if there is a contiguous sum equal to that value and False if not.
15. Modify the previous problems algorithm so that it returns the number of contiguous sums which equal that value.

7 Python Code with Output

7.1 Brute Force

Code:

```
import random
A = []
for i in range(0,7):
    A.append(random.randint(-10,10))
n = len(A)

print(A)

max = A[0]
for i in range(0,n):
    for j in range(i,n):
        sum = 0
        for c in range(i,j+1):
            sum = sum + A[c]
        if sum > max:
            max = sum
    print('Max of ' + str(A[i:j+1]) + ': ' + str(max))

print('Overall max: ' + str(max))
```

Ouput:

```
[-9, 3, 1, 1, 4, -2, -8]
Max of [-9]: -9
Max of [-9, 3]: -6
Max of [-9, 3, 1]: -5
Max of [-9, 3, 1, 1]: -4
Max of [-9, 3, 1, 1, 4]: 0
Max of [-9, 3, 1, 1, 4, -2]: 0
Max of [-9, 3, 1, 1, 4, -2, -8]: 0
Max of [3]: 3
Max of [3, 1]: 4
Max of [3, 1, 1]: 5
Max of [3, 1, 1, 4]: 9
Max of [3, 1, 1, 4, -2]: 9
Max of [3, 1, 1, 4, -2, -8]: 9
Max of [1]: 9
Max of [1, 1]: 9
Max of [1, 1, 4]: 9
Max of [1, 1, 4, -2]: 9
Max of [1, 1, 4, -2, -8]: 9
Max of [1]: 9
Max of [1, 4]: 9
Max of [1, 4, -2]: 9
Max of [1, 4, -2, -8]: 9
Max of [4]: 9
Max of [4, -2]: 9
Max of [4, -2, -8]: 9
Max of [-2]: 9
Max of [-2, -8]: 9
Max of [-8]: 9
Overall max: 9
```

7.2 Naïve Method

Code:

```
import random
A = []
for i in range(0,7):
    A.append(random.randint(-10,10))
n = len(A)
print(A)

max = A[0]
for i in range(0,n):
    sum = 0
    for j in range(i,n):
        sum = sum + A[j]
        if sum > max:
            max = sum
    print('Max of ' + str(A[i:j+1]) + ': ' + str(max))

print('Overall max: ' + str(max))
```

Output:

```
[-2, -1, 6, -1, 6, 3, 2]
Max of [-2]: -2
Max of [-2, -1]: -2
Max of [-2, -1, 6]: 3
Max of [-2, -1, 6, -1]: 3
Max of [-2, -1, 6, -1, 6]: 8
Max of [-2, -1, 6, -1, 6, 3]: 11
Max of [-2, -1, 6, -1, 6, 3, 2]: 13
Max of [-1]: 13
Max of [-1, 6]: 13
Max of [-1, 6, -1]: 13
Max of [-1, 6, -1, 6]: 13
Max of [-1, 6, -1, 6, 3]: 13
Max of [-1, 6, -1, 6, 3, 2]: 15
Max of [6]: 15
Max of [6, -1]: 15
Max of [6, -1, 6]: 15
Max of [6, -1, 6, 3]: 15
Max of [6, -1, 6, 3, 2]: 16
Max of [-1]: 16
Max of [-1, 6]: 16
Max of [-1, 6, 3]: 16
Max of [-1, 6, 3, 2]: 16
Max of [6]: 16
Max of [6, 3]: 16
Max of [6, 3, 2]: 16
Max of [3]: 16
Max of [3, 2]: 16
Max of [2]: 16
Overall max: 16
```

7.3 Divide-and-Conquer

Code:

```
import random

A = []
for i in range(0,10):
    A.append(random.randint(-10,10))
n = len(A)
print(A)

def mcs(A,l,r,indentlevel):
    print(indentlevel*'. ' + 'Finding mcs in A=' + str(A[l:r+1]))
    if l == r:
        print(indentlevel*'. ' + 'It is ' + str(A[l]))
        return(A[l])
    else:
        c = (l+r) // 2
        lhmax = A[c]
        lhsum = 0
        for i in range(c,l-1,-1):
            lhsum = lhsum + A[i]
            if lhsum > lhmax:
                lhmax = lhsum
        rhmax = A[c+1]
        rhsum = 0
        for i in range(c+1,r+1):
            rhsum = rhsum + A[i]
            if rhsum > rhmax:
                rhmax = rhsum
        cmax = lhmax + rhmax
        print(indentlevel*'. ' + 'Straddle max is ' + str(cmax))
        lmax = mcs(A,l,c,indentlevel+2)
        rmax = mcs(A,c+1,r,indentlevel+2)
        omax = max([lmax,rmax,cmax])
        print(indentlevel*'. ' + 'It is ' + str(omax))
        return(omax)

print(mcs(A,0,len(A)-1,0))
```

Output:

```
[-1, 2, 9, -3, 5, -8, 10, -6, 1, 0]
Finding mcs in A=[-1, 2, 9, -3, 5, -8, 10, -6, 1, 0]
..Finding mcs in A=[-1, 2, 9, -3, 5]
....Finding mcs in A=[-1, 2, 9]
.....Finding mcs in A=[-1, 2]
.....Finding mcs in A=[-1]
.....It is -1
.....Finding mcs in A=[2]
.....It is 2
.....It is 2
.....Finding mcs in A=[9]
.....It is 9
.....It is 11
....Finding mcs in A=[-3, 5]
.....Finding mcs in A=[-3]
.....It is -3
.....Finding mcs in A=[5]
.....It is 5
....It is 5
..It is 13
..Finding mcs in A=[-8, 10, -6, 1, 0]
....Finding mcs in A=[-8, 10, -6]
.....Finding mcs in A=[-8, 10]
.....Finding mcs in A=[-8]
.....It is -8
.....Finding mcs in A=[10]
.....It is 10
.....It is 10
.....Finding mcs in A=[-6]
.....It is -6
....It is 10
....Finding mcs in A=[1, 0]
.....Finding mcs in A=[1]
.....It is 1
.....Finding mcs in A=[0]
.....It is 0
....It is 1
..It is 10
It is 15
15
```

7.4 Kadane's Algorithm

Code:

```
import random

A = []
for i in range(0,20):
    A.append(random.randint(-10,10))

n = len(A)
print(A)

maxoverall = A[0]
maxendingati = A[0]
for i in range(1,n):
    maxendingati = max(maxendingati+A[i],A[i])
    maxoverall = max(maxoverall,maxendingati)
    print('Best sum ending with index i=' + str(i) + ' is ' + str(maxoverall))

print(maxoverall)
```

Output:

```
[4, -1, 0, 4, -7, 2, -3, -2, 3, 1, 7, 3, -5, 2, -3, 2, 9, -5, 3, -2]
Best sum ending with index i=1 is 4
Best sum ending with index i=2 is 4
Best sum ending with index i=3 is 7
Best sum ending with index i=4 is 7
Best sum ending with index i=5 is 7
Best sum ending with index i=6 is 7
Best sum ending with index i=7 is 7
Best sum ending with index i=8 is 7
Best sum ending with index i=9 is 7
Best sum ending with index i=10 is 11
Best sum ending with index i=11 is 14
Best sum ending with index i=12 is 14
Best sum ending with index i=13 is 14
Best sum ending with index i=14 is 14
Best sum ending with index i=15 is 14
Best sum ending with index i=16 is 19
Best sum ending with index i=17 is 19
Best sum ending with index i=18 is 19
Best sum ending with index i=19 is 19
19
```

CMSC 351: BubbleSort (Basic)

Justin Wyss-Gallifent

June 12, 2024

1	What it Does	2
2	How it Works	2
3	Visualization	3
4	Pseudocode and Time Complexity	4
5	Auxiliary Space	5
6	Stability	5
7	In-Place	5
8	Notes	5
9	Bubble Sort (Variation)	6
10	Closing Notes	7
11	Thoughts, Problems, Ideas	8
12	Python Test (Basic Version) with Output	10

1 What it Does

Sorts a list of elements such as integers or real numbers.

2 How it Works

We pass through the list from left to right swapping elements which are out of order. If we pass through once, the final entry is in the right place because the largest element will have been swapped repeatedly until it is at the end. Thus the second pass through can stop before the final entry. If we pass through twice, the final two entries are in the right place and so the third pass through can stop before the second-to-last entry. We continue onwards. Thus we pass through $n - 1$ times total since once the final $n - 1$ entries are in order, all are.

3 Visualization

Consider the following list:

8	4	3	4	1
---	---	---	---	---

Here is the result of the first pass through. We indicate in red pairs the swaps.

8	4	3	4	1
4	8	3	4	1
4	3	8	4	1
4	3	4	8	1
4	3	4	1	8

Notice after the first pass through the one rightmost element is correct.

Second pass through:

4	3	4	1	8
3	4	4	1	8
3	4	1	4	8

Notice after the second pass through the two rightmost elements are correct.

Third pass through:

3	4	1	4	8
3	1	4	4	8

Notice after the third pass through the three rightmost elements are in order.

The fourth pass through:

3	1	4	4	8
1	3	4	4	8

Notice after the fourth pass through the four rightmost elements are in order but since there are only five elements, they are all in order.

4 Pseudocode and Time Complexity

Here is the pseudocode with time assignments:

```

PRE: A is a list of length n.
for i = 0 to n-2
    for j = 0 to n-i-2
        if A[j] > A[j+1]
            swap A[j] and A[j+1]
        end
    end
POST: A is sorted.

```

This is a terrible implementation. The time required is always the same in the best-, worst-, and average-cases:

$$\begin{aligned}
T(n) &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-i-2} c \\
&= \sum_{i=0}^{n-2} (n - i - 2 + 1)c \\
&= \sum_{i=0}^{n-2} c(n - 1) - c \sum_{i=0}^{n-2} i \\
&= (n - 2 + 1)c(n - 1) - c \left(\frac{(n - 2)(n - 1)}{2} \right) \\
&= c(n - 1)^2 - \frac{1}{2}c(n - 2)(n - 1) \\
&= \frac{1}{2}cn^2 - \frac{1}{2}cn \\
&= \Theta(n^2)
\end{aligned}$$

A quick note here that if we were interested in every little detail of time then we would need to label separate time constants for both the conditional check and for the conditional body and then best-, worst-, and average-cases would have different total times depending upon the frequency with which the body of the conditional executed.

However in all cases the conditional is still taking constant time, even if those constant times are different, and hence the time complexity does not change.

5 Auxiliary Space

Our BubbleSort (Basic) pseudocode uses $\mathcal{O}(1)$ auxiliary space - two indices and possibly, depending on the implementation, a swap variable. This is of course pretty much ideal.

6 Stability

BubbleSort (Basic) is stable. This means that the order of identical entries is preserved. In the opening example if we think of the two 4s as different, perhaps the one on the left is red and one on the right is blue, then the red one is still on the left at the end. The reason for this is that equal elements are never swapped, meaning that the order of equal elements is always preserved.

7 In-Place

BubbleSort (Basic) is in-place. This means that the list is sorted by moving elements within the list, rather than creating a new list.

8 Notes

After k iterations the last k elements are correctly placed and sorted. If BubbleSort were running on a very long list and it was forced to stop early this means that some amount of the end of the list would be sorted but the beginning would not. This is the opposite of SelectionSort, as we'll see.

9 Bubble Sort (Variation)

We can improve the code somewhat because if any give pass through causes no swaps then the list is in order and we can exit. Here is the adjusted pseudocode with time assignments:

```

PRE: A is a list of length n.
stillgoing = True
for i = 0 to n-1
    didaswap = False
    for j = 0 to n-i-2
        if A[j] > A[j+1]
            swap A[i] and A[i+1]
            didaswap = True
        end
    end
    if didaswap == False
        break
    end
end
POST: A is sorted.

```

c_1
 c_3
 $n - i - 1$ times
 $\left. \begin{array}{l} c_2 \\ \end{array} \right\}$
 c_4

In this case we have:

- Worst-Case: If the list is in reverse order then each i iteration results in $n - i - 1$ comparisons and swaps. To see this observe that if we start with $[5, 4, 3, 2, 1]$ then the $i=0$ pass has $n - i - 1 = 5 - 0 - 1 = 4$ comparisons and swaps and results in $[4, 3, 2, 1, 5]$. The $i=1$ pass has $n - i - 1 = 5 - 1 - 1 = 3$ comparisons and swaps and results in $[3, 2, 1, 4, 5]$, and so on. We therefore use all of $i=0, \dots, n-1$ iterations (no `break`) and the time is then:

$$T(n) = c_1 + \sum_{i=0}^{n-1} \left[c_3 + \sum_{j=0}^{n-i-2} c_2 \right] = \dots = \Theta(n^2)$$

- Best-Case: If the list is already sorted then we have one pass through for $i=0$ with $n - i - 1$ iterations and since no swaps occur we exit. The time is then:

$$T(n) = c_1 + c_3 + (n - i - 1)c_2 + c_4 = \dots = \Theta(n)$$

- Average-Case: We need to clarify what an “average” list looks like. It turns out that in an average list we end up swapping about half the time. (this takes some combinatorics) and the result is still $\Theta(n^2)$.

This variation also uses $\mathcal{O}(1)$ auxiliary space, is stable, and is in-place.

10 Closing Notes

While BubbleSort is fairly useless in practice since it is slow and inefficient it is nonetheless useful for basic algorithmic understanding and some of the ideas therein can be used in other algorithms.

11 Thoughts, Problems, Ideas

1. Fill in the calculation:

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-i-2} c_1 = \dots = \Theta(n^2)$$

At the end, provide a rigorous proof from the definition of Θ that the result is $\Theta(n^2)$.

2. Fill in the calculation:

$$T(n) = c_1 + \sum_{i=0}^{n-1} \left[c_3 + \sum_{j=0}^{n-i-2} c_2 \right] = \dots = \Theta(n^2)$$

At the end, provide a rigorous proof from the definition of Θ that the result is $\Theta(n^2)$.

3. Suppose BubbleSort (Basic) is used to sort the list [34, 2, 17, 5, 10]. Show the list after each swap and count the number of swaps.
4. Modify the BubbleSort (Basic) pseudocode to produce an algorithm which moves all 0s (if any) to the right and otherwise preserves order.
5. BubbleSort (Basic) is stable. Explain in your own words what that means. What single-character change in the pseudocode would make it not stable?
6. How many comparisons occur in BubbleSort (Basic)?
7. In BubbleSort (Basic) how many swaps occur in a worst-case scenario? How about a best-case scenario?
8. In BubbleSort (Basic) think about how many swaps might occur in an average-case scenario. What if the list was mostly sorted? This is a qualitative rather than a quantitative question.
9. Each integer greater than 2 has a unique prime factorization. For example $243936 = 2^5 \cdot 3^2 \cdot 7^1 \cdot 11^2$. Suppose you want to write an algorithm which sorts the powers using BubbleSort and modifies the number accordingly, so we'd have, for example:

$$243936 = 2^5 \cdot 3^2 \cdot 7^1 \cdot 11^2 \implies 2^1 \cdot 3^2 \cdot 7^2 \cdot 11^5 = 142046982$$

You wish to use BubbleSort on the exponents but you are only given the integer n and you cannot construct or use a mutable list. You do have access to the commands:

- `ppow(n,p)` which gives the exponent corresponding to the power of p in n , so `ppow(243936,3)` returns 2.
- `parr(n)` which returns the list of primes which occur in the prime factorization of n , so `parr(243936)` returns [2,3,7,11]. You can index it with `parr(243935)[0]` yielding 2, for example.

Write the associated pseudocode.

10. If the algorithm in the previous problem is used to define a function `primesort(n)` (so for example `primesort(243936)=142046982`) explain why the output is never smaller than the input. For which `n` is it identical?
11. Modify BubbleSort so that after k iterations the first k elements are correct.
12. Modify BubbleSort so that after $2k$ iterations the first k and last k elements are correct. You can assume that the list has an even number of elements. Hint: Alternate!
13. Here is a re-labeling of BubbleSort (Basic) time requirements in which the time of the conditional and its body assignment have been separated:

```
PRE: A is a list of length n.  
for i = 0 to n-1                                n times  
    for j = 0 to n-i-2                          n - i - 2 - 0 + 1 = n - i - 1 times  
        if A[j] > A[j+1]                      c1  
            swap A[j] and A[j+1]                c2  
        end  
    end  
POST: A is sorted.
```

- (a) If $c_1 = 3$ and $c_2 = 10$, both in seconds, how long will it take to sort the list `[4,5,1,2,7,7]`?
- (b) Suppose the inputs to this implementation are such that on average the inequality is true with probability $0 \leq p \leq 1$. Calculate the resulting average time requirements exactly. When you have your result suppose n were some fixed value. Explain how the time required changes as p does. Is it linear or something else?
14. Provide a rigorous mathematical proof that the first iteration through BubbleSort places the largest element in the final position.
15. Suppose you hate `break` statements. Rewrite the BubbleSort (Variation) using a `while` loop and no `break`.

12 Python Test (Basic Version) with Output

Code:

```
import random
A = []
for i in range(0,10):
    A.append(random.randint(0,100))
n = len(A)

print('Start: '+str(A))
for i in range(0,n):
    for j in range(0,n-i-1):
        if A[j] > A[j+1]:
            temp = A[j]
            A[j] = A[j+1]
            A[j+1] = temp
    print('Iterate: '+str(A))
print(A)
```

Output:

```
Start: [74, 85, 2, 13, 82, 36, 29, 66, 31, 12]
Iterate: [74, 2, 13, 82, 36, 29, 66, 31, 12, 85]
Iterate: [2, 13, 74, 36, 29, 66, 31, 12, 82, 85]
Iterate: [2, 13, 36, 29, 66, 31, 12, 74, 82, 85]
Iterate: [2, 13, 29, 36, 31, 12, 66, 74, 82, 85]
Iterate: [2, 13, 29, 31, 12, 36, 66, 74, 82, 85]
Iterate: [2, 13, 29, 12, 31, 36, 66, 74, 82, 85]
Iterate: [2, 13, 12, 29, 31, 36, 66, 74, 82, 85]
Iterate: [2, 12, 13, 29, 31, 36, 66, 74, 82, 85]
Iterate: [2, 12, 13, 29, 31, 36, 66, 74, 82, 85]
Iterate: [2, 12, 13, 29, 31, 36, 66, 74, 82, 85]
[2, 12, 13, 29, 31, 36, 66, 74, 82, 85]
```

CMSC 351: SelectionSort

Justin Wyss-Gallifent

February 3, 2022

1	What it Does	2
2	How it Works	2
3	Visualization	2
4	Pseudocode with Time Complexity	3
5	Auxiliary Space	3
6	Stability	3
7	In-Place	4
8	Notes	4
9	Thoughts, Problems, Ideas	5
10	Python Test with Output	6

1 What it Does

Sorts a list of elements on which there is a total order. Think of integers or real numbers.

2 How it Works

We identify the smallest element in $A[0, \dots, n-1]$ and swap it with the element at index 0. At this point $A[0]$ is (trivially) sorted and we can ignore it. We then identify the smallest element in $A[1, \dots, n-1]$ and swap it with the element at index 1. At this point $A[0,1]$ is sorted and we can ignore it. We then identify the smallest element in $A[2, \dots, n-1]$ and swap it with the element at index 2. At this point $A[0, \dots, 2]$ is sorted and we can ignore it. We continue until $A[0, \dots, n-2]$ is sorted, at which point the entire list is sorted.

3 Visualization

Consider the following list:

4	3	1	8	3
---	---	---	---	---

We first find the smallest element in $A[0, \dots, 4]$:

4	3	1	8	3
---	---	---	---	---

We swap it with index 0:

1	3	4	8	3
---	---	---	---	---

We then find the smallest element in $A[1, \dots, 4]$:

1	3	4	8	3
---	---	---	---	---

We swap it with index 1, which does not change the result.

We then find the smallest element in $A[2, \dots, 4]$:

1	3	4	8	3
---	---	---	---	---

We swap it with index 2:

1	3	3	8	4
---	---	---	---	---

We then find the smallest element in $A[3, \dots, 4]$:

1	3	3	8	4
---	---	---	---	---

We swap it with index 3:

1	3	3	4	8
---	---	---	---	---

Then we are done.

4 Pseudocode with Time Complexity

Here is the pseudocode with time assignments:

```
\\" PRE: A is a list of length n.
for i = 0 to n-2
    minindex = i
    for j = i+1 to n-1
        if A[j] < A[minindex]
            minindex = j
        end
    end
    swap A[i] with A[minindex]
end
\" POST: A is sorted.
```

$n - 1$ times
 c_1
 $n - 1 - (i + 1) + 1 = n - i - 1$ times.
 $\left. \begin{array}{l} \\ \\ \end{array} \right\} c_2$
 c_3

The time required is always the same in the best-, worst-, and average-cases:

$$\begin{aligned}
T(n) &= \sum_{i=0}^{n-2} \left[c_1 + \left[\sum_{j=i+1}^{n-1} c_2 \right] + c_3 \right] \\
&= \sum_{i=0}^{n-2} [c_1 + (n - 1 - (i + 1) + 1)c_2 + c_3] \\
&= \sum_{i=0}^{n-2} [c_1 + (n - 1 - i)c_2 + c_3] \\
&= \sum_{i=0}^{n-2} [c_1 + (n - 1)c_2 + c_3] - c_2 \sum_{i=0}^{n-2} i \\
&= (n - 2 + 1)[c_1 + (n - 1)c_2 + c_3] - c_2 \left(\frac{(n - 2)(n - 1)}{2} \right) \\
&= \frac{1}{2}c_2n^2 + \left(c_1 - \frac{1}{2}c_2 + c_3 \right) n - (c_1 + c_3) \\
&= \Theta(n^2)
\end{aligned}$$

The same quick note from BubbleSort applies here with regards to labeling of the conditional.

5 Auxiliary Space

The auxiliary space is $\mathcal{O}(1)$ which includes two indices, `minindex` and potentially a swap variable.

6 Stability

SelectionSort is not stable. The reason for this is when SelectionSort finds a smallest element at index `minindex` and swaps it with the element at index `i`,

the element at index i could be moved to the right of an equal element that is sitting between them.

7 In-Place

SelectionSort is in-place.

8 Notes

After k iterations the first k elements are correctly placed and sorted. If SelectionSort were running on a very long list and it was forced to stop early this means that some amount of the beginning of the list would be sorted but the end would not. This is the opposite of BubbleSort, as we saw.

9 Thoughts, Problems, Ideas

1. Fill in the summation details of the time calculations. In other words, get an exact answer before the \mathcal{O} step.
2. Modify SelectionSort so that after k iterations the last k elements are correct.
3. Modify SelectionSort so that after $2k$ iterations the first k and last k elements are correct. Hint: Alternate!
4. Suppose the algorithm does this instead: It first scans for any element smaller than $A[0]$ and when it finds one it immediately exchanges it with $A[0]$. Then it tries again, repeatedly until it fails to find one. It then repeats the process with $A[1]$, $A[2]$, and so on. Write the pseudocode for this and find its worst-case \mathcal{O} time complexity.
5. The basic premise of SelectionSort is that the **for** loop is selecting (hence the name) the index of the next smallest element at each iteration, then that smallest element is swapped into its proper position. That locating step which takes $\mathcal{O}(n)$. Suppose that step is replaced by a single function **indexofsmallest(a,b)** which finds the index of the smallest element between indices i and j . Suppose this new function takes time $S(n)$.

```
for i = 0 to n-2           n - 1 times
    minindex = indexofsmallest(i,n-1)  S(n)
    swapindices(i,minindex)          c1
end
```

What would need to be true about the time requirement $S(n)$ to make the time complexity faster than the standard SelectionSort?

6. Modify the pseudocode so that it finds both the **minindex** and **maxindex** during each pass and swaps those into their appropriate positions. This is the *Cocktail Sort*.
7. In a worst-case scenario how many assignments does the pseudocode make?
8. Suppose a list A has an odd number of elements, so $n = 2k + 1$. Modify **SelectionSort** to create a new function **median** which finds the median of the list. How many comparisons will be necessary to find the median of the list $\{5, 3, 0, 16, 8, 3, 7\}$?
9. Rewrite your **median** function from the previous question so that it will work if A has either an even or an odd number of elements. You should not break the code into even/odd cases and you don't even need to check if n is even or odd but you can use the ceiling command **ceil** as well as simple arithmetic. It is enough!

10 Python Test with Output

Code

```
import random
A = []
for i in range(0,10):
    A.append(random.randint(0,100))
n = len(A)
print(A)

for i in range(0,n-1):
    minindex = i
    for j in range(i+1,n):
        if A[j] < A[minindex]:
            minindex = j
    print('In A['+str(i)+','+str(n-1)+'] the index of the
          minimum is '+str(minindex))
    temp = A[i]
    A[i] = A[minindex]
    A[minindex] = temp
    print('Swap indices ' + str(minindex) + ' and ' + str(i))
    )
print('Now: ' + str(A))

print(A)
```

Output:

```
[37, 89, 47, 59, 87, 23, 43, 98, 68, 30]
In A[0,9] the index of the minimum is 5
Swap indices 5 and 0
Now: [23, 89, 47, 59, 87, 37, 43, 98, 68, 30]
In A[1,9] the index of the minimum is 9
Swap indices 9 and 1
Now: [23, 30, 47, 59, 87, 37, 43, 98, 68, 89]
In A[2,9] the index of the minimum is 5
Swap indices 5 and 2
Now: [23, 30, 37, 59, 87, 47, 43, 98, 68, 89]
In A[3,9] the index of the minimum is 6
Swap indices 6 and 3
Now: [23, 30, 37, 43, 87, 47, 59, 98, 68, 89]
In A[4,9] the index of the minimum is 5
Swap indices 5 and 4
Now: [23, 30, 37, 43, 47, 87, 59, 98, 68, 89]
In A[5,9] the index of the minimum is 6
Swap indices 6 and 5
Now: [23, 30, 37, 43, 47, 59, 87, 98, 68, 89]
In A[6,9] the index of the minimum is 8
Swap indices 8 and 6
Now: [23, 30, 37, 43, 47, 59, 68, 98, 87, 89]
In A[7,9] the index of the minimum is 8
Swap indices 8 and 7
Now: [23, 30, 37, 43, 47, 59, 68, 87, 98, 89]
In A[8,9] the index of the minimum is 9
Swap indices 9 and 8
Now: [23, 30, 37, 43, 47, 59, 68, 87, 89, 98]
[23, 30, 37, 43, 47, 59, 68, 87, 89, 98]
```

CMSC 351: InsertionSort

Justin Wyss-Gallifent

July 9, 2024

1	What it Does	2
2	How it Works:	2
3	Visualization	2
4	Outline of Pseudocode	3
5	Pseudocode	3
6	Time Complexity Analysis	5
7	Auxiliary Space	8
8	Stability	8
9	In-Place	8
10	Notes	8
11	Thoughts, Problems, Ideas	9
12	Python Test	11

1 What it Does

Sorts a list of integers or real numbers.

2 How it Works:

We pass through the list from left to right. If we encounter an entry which is smaller than some of its predecessors then we need to move it as far left as is appropriate. We shift as many elements to the right as needed to make room for it and then insert it (hence the name) in the proper position.

3 Visualization

Consider the following list:

4	3	1	8	3
---	---	---	---	---

We don't bother to look at $A[0]==4$ because nothing to the left could be larger, because there's nothing to the left of it.

Thus we first look at $A[1]==3$ and note that there is one larger element to the left:

4	3	1	8	3
---	---	---	---	---

We shift it to the right and insert the 3 where it belongs:

3	4	1	8	3
---	---	---	---	---

We then look at $A[2]==1$ and note that there are two larger elements to the left:

3	4	1	8	3
---	---	---	---	---

We shift those to the right and insert the 1 where it belongs:

1	3	4	8	3
---	---	---	---	---

We next look at $A[3]==8$ and note that there are no larger elements to the left:

1	3	4	8	3
---	---	---	---	---

We do nothing. Again officially the pseudocode will actually shift nothing and then assign $A[3]=8$, which is a bit wasteful but there we go.

We then look at $A[4]==3$ and note that there are two larger elements to the left:

1	3	4	8	3
---	---	---	---	---

We shift those to the right and insert the 3 where it belongs:

1	3	3	4	8
---	---	---	---	---

Then we are done.

4 Outline of Pseudocode

To help see how the pseudocode work, here is the general outline. Pseudo-pseudocode, if you wish!

```
for i = 1 to n-1
    key = A[i], the value to potentially be moved left.
    j = i - 1, the first index to the left.
        shift A[j] to the right while key < A[j] and j >= 0
        j = j - 1
            this while loop is shifting things right
            to make room for key in the right place
        once this ends either key >= A[j] and should be left alone,
        meaning key should go into index j+1,
        or j = -1, meaning key should go into index 0 = j+1
        A[j+1] = key, so key is now in the right place
    end
once this ends, everything is right
```

5 Pseudocode

Here is the pseudocode with time assignments. Here I've gone old-school and tagged the `while` loop condition check as it clarifies a bit.

```
\PRE: A is a list of length n.
for i = 1 to n-1                      n - 1 iterations
    key = A[i]
    j = i-1
    while j >= 0 and key < A[j]      {c1} each time there's a check
        A[j+1] = A[j]
        j = j - 1
    end
    A[j+1] = key                     c4
end
\POST: A is sorted.
```

Here `key` holds the value at index `i` needs to be stored as we check to the left of it and potentially shift any larger left element to the right. Once we have moved those to the right we insert `key` into its appropriate position.

Notice that the `while` loop checks not just whether the element to the left is larger but also whether that searching process falls off the front of the list. It stops in either case, of course. In the case where it falls off the front of the list we have `j == -1` and we need to insert `key` at index `0 == j+1`. In the case where it encounters an element at index `j` which is smaller or equal than `key` and which does not need to be moved, it must insert `key` to the right of it at index `j+1`.

This is why the line after the `while` loop is `A[j+1] = key.`

6 Time Complexity Analysis

The time complexity analysis for InsertionSort is very different than BubbleSort and SelectionSort. This is due to the `while` loop which results in an unknown number of iterations. For a particular input of length n this loop could iterate never or perhaps a number of times linearly dependent on n . We thus have to separate cases.

1. Best-Case:

In a best-case scenario the `while` loop fails every time, which will happen iff the listed is sorted. In such a case we have:

$$T(n) = (n - 1)(c_1 + c_2 + c_4) = \Theta(n)$$

2. Worst-Case:

In a worst-case scenario the `while` loop passes as many times as possible, which will happen iff the list is in reverse order.

More specifically the `while` loops will pass for $j = i - 1, i - 2, \dots, 0$ (a total of i times) and fail at $j = -1$.

In such a case we will have:

$$T(n) = \sum_{i=1}^n [c_1 + (i)(c_2 + c_3) + c_2 + c_4] = \dots = \Theta(n^2)$$

3. Average Case:

First we'll observe that the running time can be written in terms of the number of *inversions* which exist in the original list. An *inversion* is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$.

Example 6.1. In our original list:

4	3	1	8	3
---	---	---	---	---

In this list there are 5 inversions: $(0, 1)$ (because $A[0] > A[1]$), $(0, 2)$ (because $A[0] > A[2]$), $(0, 5)$ (because $A[0] > A[5]$), $(1, 2)$ (because $A[1] > A[2]$), and $(3, 4)$ (because $A[3] > A[4]$).

If we consider the pseudocode carefully we see that each time an element is shifted (to the right) in preparation for an insertion that we are essentially correcting/removing an inversion.

Example 6.2. In our original list of $[4, 3, 1, 8, 3]$ we have five inversions and if we check back we see we shifted five elements.

Moreover since each shift corresponds to the `while` loop passing exactly once this means that the number of times the `while` loop passes is equal to the number of inversions in the list.

In light of this consider the time requirements again:

- For each of $i=1, \dots, n-1$ we have $c_1 + c_4$ before and after the `while` loop.
- For each inversion the `while` loop criteria will pass exactly once, requiring time $c_2 + c_3$.
- For each of $i=1, \dots, n-1$ the `while` loop criteria will fail exactly once, either after shifts have occurred or at the start if no shifts were necessary, requiring time c_2 .

Thus if there are I inversions then the total time requirement will be as follows, where C+P means “`while` loop checks and passes” and C+F means “`while` loop checks and fails”.

$$T(n, I) = \underbrace{(n-1)(c_1 + c_4)}_{\text{Pre+Post-While}} + \underbrace{I(c_2 + c_3)}_{\text{C+P}} + \underbrace{(n-1)c_2}_{\text{C+F}}$$

For the sake of simplicity let's assume no duplicates in what follows.

In an arbitrary list if we look at all possible there are $C(n, 2)$ pairs of elements and therefore $C(n, 2)$ possible inversions. For example in a list of length $n = 5$ there are $C(5, 2) = 10$ possible inversions. If the list is sorted there are 0 and if the list is in reverse order there are 10.

If we look at all possible lists, it turns out that on average we will have

$I = I(n) = C(n, 2)/2$ inversions and so the average the time requirement will be:

$$T(n, I(n)) = T\left(n, \frac{C(n, 2)}{2}\right) = \dots = \Theta(n^2)$$

7 Auxiliary Space

The auxiliary space is $\mathcal{O}(1)$.

8 Stability

InsertionSort is stable. This is because InsertionSort shifts elements to the left while they are smaller than those elements. Consequently an element will not be shifted to the left of an equal element and so the order of equal elements will be preserved.

9 In-Place

InsertionSort is in-place.

10 Notes

After k iterations the first k elements are sorted but not necessarily correctly placed. If InsertionSort were running on a very long list and it was forced to stop early this means that some amount of the beginning of the list would be sorted but in the wrong position and the end would not be sorted at all. This is different from both BubbleSort and SelectionSort.

Example 10.1. Consider the list:

[7,8,9,4,5,6,1,2,3]

The first six iterations operate on the elements 7, 8, 9, 4, 5, and 6. The first three don't move but they still count as iterations as far as the pseudocode is concerned.

The result is then:

[4,5,6,7,8,9,1,2,3]

Notice that the first six elements are sorted relative to one another but they are certainly not in their correct overall locations. It will take three more iterations to move the 1,2,3 for this to be the case.

11 Thoughts, Problems, Ideas

1. Show the step-by-step operation of InsertionSort on the list $\{5, 10, 3, 0, -6, 9\}$.
2. Assuming all constant times equal 1, fill in the details of the worst-case time calculation:

$$T(n) = \sum_{i=0}^{n-1} c_1 + i(c_2 + c_3) + c_2 + c_4 = \dots = \Theta(n^2)$$

3. Assuming all constant times equal 1, fill in the details of the average-case time calculation:

$$T(n, I(n)) = T\left(n, \frac{C(n, 2)}{2}\right) = \dots = \Theta(n^2)$$

4. Modify InsertionSort so the list is sorted in decreasing order.
5. The basic premise of InsertionSort is that for each index i we effectively shift everything to the left of this index which smaller than the value at this index to the right by 1 and then insert (hence the name) the value in the empty left location. Suppose you had a function `rshift(A,a,b)` which shifted the subarray $A[a, \dots, b]$ to the right by 1. Use this to develop a pseudocode version of InsertionSort. For each i this version should scan left until it finds where to put $A[i]$, then shift everything necessary to the right, and then insert. If `rshift` operates with constant time complexity what is the worst-case \mathcal{O} time complexity of your new pseudocode?
6. Another way to think of InsertionSort is that for each index i we want to know where to the left to insert it. Note that for each i the sublist to the left is always sorted as the algorithm progresses. Call this target index k . The series of swaps effectively shifts $A[k, \dots, i-1]$ to the right and inserts the original $A[i]$ to index k . Suppose you had a function `locateindex(A,i)` which would find the correct index in $A[0, \dots, i-1]$ to insert $A[i]$. Use this to develop a pseudocode version of InsertionSort. You'll still need to do the swapping manually.
7. Suppose you had both `rshift` and `locateindex` from the previous two problems. Now write the new pseudocode. If the time complexity of `locateindex` is $L(n)$ and the time complexity of `rshift` is $R(n)$ what would need to be true of these in order for the resulting search to be \mathcal{O} -better than InsertionSort?
8. Arguably the pseudocode is a bit confusing because the search index j is actually always one less than the target position. Tweak the pseudocode to change this.
9. We saw that in the average case the pseudocode has time requirement:

$$T(n, I) = n(c_1 + c_4) + I(c_2 + c_3) + n(c_2)$$

where I is the number of inversions in the input list. The reason that the average case turns out to be $\mathcal{O}(n^2)$ is that on average the number of inversions satisfies $I(n) = \mathcal{O}(n^2)$. We'll say a list of length n is *logarithmically sorted* (I made that term up) if the number of inversions is at most $\lceil (\lg n/n)C(n, 2) \rceil$. For example a logarithmically sorted list of length 10 will have at most $\lceil (\lg 10/10)C(10, 2) \rceil = 12$ inversions. Calculate the \mathcal{O} average time complexity of the pseudocode if it only encounters such lists.

12 Python Test

Code:

```
import random
A = []
for i in range(0,10):
    A.append(random.randint(0,100))
n = len(A)
print(A)
for i in range(0,n):
    key = A[i]
    print('Checking ' + str(key) + ' at index ' + str(i))
    j = i-1
    howmanytoleft = 0
    while j>=0 and key<A[j]:
        howmanytoleft = howmanytoleft + 1
        A[j+1] = A[j]
        j = j - 1
    if howmanytoleft == 0:
        print('There are 0 larger elements to the left. Leave alone.')
    else:
        A[j+1] = key
        print('There are ' + str(howmanytoleft) + ' larger elements to the left')
        print('Shift and insert: ' + str(A))
print(A)
```

Output:

```
[31, 24, 77, 79, 53, 20, 20, 75, 46, 0]
Checking 31 at index 0
There are 0 larger elements to the left. Leave alone.
Checking 24 at index 1
There are 1 larger elements to the left.
Shift and insert: [24, 31, 77, 79, 53, 20, 20, 75, 46, 0]
Checking 77 at index 2
There are 0 larger elements to the left. Leave alone.
Checking 79 at index 3
There are 0 larger elements to the left. Leave alone.
Checking 53 at index 4
There are 2 larger elements to the left.
Shift and insert: [24, 31, 53, 77, 79, 20, 20, 75, 46, 0]
Checking 20 at index 5
There are 5 larger elements to the left.
Shift and insert: [20, 24, 31, 53, 77, 79, 20, 75, 46, 0]
Checking 20 at index 6
There are 5 larger elements to the left.
Shift and insert: [20, 20, 24, 31, 53, 77, 79, 75, 46, 0]
Checking 75 at index 7
There are 2 larger elements to the left.
Shift and insert: [20, 20, 24, 31, 53, 75, 77, 79, 46, 0]
Checking 46 at index 8
There are 4 larger elements to the left.
Shift and insert: [20, 20, 24, 31, 46, 53, 75, 77, 79, 0]
Checking 0 at index 9
There are 9 larger elements to the left.
Shift and insert: [0, 20, 20, 24, 31, 46, 53, 75, 77, 79]
[0, 20, 20, 24, 31, 46, 53, 75, 77, 79]
```

CMSC 351: Binary Search

Justin Wyss-Gallifent

September 25, 2023

1	What it Does	2
2	How it Works	2
3	Pseudocode	3
4	Time Complexity Analysis	3
5	Thoughts, Problems, Ideas	7
6	Python Test	9

1 What it Does

Given a sorted list of elements and a target element, finds the index of the target element or returns failure if the target element does not exist.

2 How it Works

The algorithm first looks at the middle of the list. If the element is not there then it knows by comparison if the element is on the left or the right of that middle element and so it concentrates its search to half the list and repeats. It keeps repeating this process either until it finds the element or the sublist it is looking at shrinks to length 1 and the element is not found.

Example 2.1. Consider the list with 20 elements. We wish to find the number 17. We look at the entire list:

$$A = [0, 0, 4, 4, 6, 7, 8, 9, 9, 10, 12, 13, 13, 14, 14, 14, 17, 18, 19, 19, 19]$$

We reference the start and end by indices so we have $L = 0$ and $R = 19$. We find the center $C = \lfloor (19 + 0)/2 \rfloor = 9$ and find $A[9] = 10$. This is too small so 17 must be to the right.

We check the sublist by reassigning $L = 9 + 1 = 10$ and leaving $R = 19$:

$$[0, 0, 4, 4, 6, 7, 8, 9, 9, 10, \underline{12}, 13, 13, 14, 14, 14, 17, 18, 19, 19, 19]$$

We find the center $C = \lfloor (19 + 10)/2 \rfloor = 14$ and find $A[14] = 14$. This is too small so 17 must be to the right.

We check the sublist by reassigning $L = 14 + 1 = 15$ and leaving $R = 19$:

$$[0, 0, 4, 4, 6, 7, 8, 9, 9, 10, 12, 13, 13, 14, 14, \underline{17}, 18, 19, 19, 19]$$

We find the center $C = \lfloor (19 + 15)/2 \rfloor = 17$ and find $A[17] = 19$. This is too large so 17 must be to the left.

We check the sublist by leaving $L = 15$ and reassigning $R = 17 - 1 = 16$:

$$[0, 0, 4, 4, 6, 7, 8, 9, 9, 10, 12, 13, 13, 14, 14, \underline{17}, 18, 19, 19, 19]$$

We find the center $C = \lfloor (16 + 15)/2 \rfloor = 15$ and find $A[15] = 17$. This is exactly right so we return 15.

3 Pseudocode

In the following pseudocode we assign time values not to get a precise time measurement but only to look at the complexity. This is why we have not separated the time values for conditional checks versus bodies.

```
\\" PRE: A is a sorted list of length n.
\" PRE: TARGET is a target element.
function binarysearch(A,TARGET)
    L = 0
    R = n-1
    while L <= R
        C = floor((L+R)/2)
        if A[C] == TARGET
            return C
        elif TARGET < A[C]
            R = C-1
        elif TARGET > A[C]
            L = C+1
        end
    end while
    return FAIL
end
\" POST: Value returned is either the index or FAIL.
```

Note 3.0.1. A note to make sure that this does as intended in the FAIL case:

Assign $L=0$ and $R=n-1$. Take the middle index $C=\text{floor}((L+R)/2)$. and examine $A(C)$. If $A(C)==\text{TARGET}$ then we return C . Otherwise if $\text{TARGET}<A(C)$ then TARGET is to the left of C so we set $R=C-1$ and start again. On the other hand if $\text{TARGET}>A(C)$ then TARGET is to the right of C so we set $L=C+1$ and start again.

This process proceeds until either when we find TARGET or when $L=R$ and TARGET does not exist. In this latter case what happens is that we assign $C=\text{floor}((L+R)/2)=L=R$ and then since $A(C) \neq \text{TARGET}$ we end up with either $L = C + 1 = R + 1$ or $R = C - 1 = L - 1$ and in both cases $R < L$.

4 Time Complexity Analysis

1. Best-Case:

If the target is immediately located at $C=\text{floor}((L+R)/2)$ at the start of the first iteration then the total time requirement is:

$$T(n) = c_1 + c_2 = \Theta(1)$$

2. Worst Case:

The worst-case scenario happens if the TARGET is never found.

Consider the length of the list after each iteration. Initially it is length n .

After the first iteration the sublist length is $\frac{n}{2}$. Technically if n is odd then the length is this value ± 0.5 but this doesn't affect what follows.

After the second iteration the sublist length is $\frac{n}{4}$.

This continues such that after k iterations the sublist length is $\frac{n}{2^k}$.

In a worst case scenario the `while` loop iterates until `L==R` and then at the end of that iteration `R<L` and it fails. We have `L==R` when the sublist length is 1:

$$\begin{aligned}\frac{n}{2^k} &= 1 \\ 2^k &= n \\ k &= \lg n\end{aligned}$$

However noting that it has one more iteration when `L==R` we can then conclude that the loop iterates $1 + \lg n$ times and then the condition fails.

It follows that the total time requirement is:

$$T(n) = c_1 + c_2(1 + \lg n) + c_3 = \Theta(\lg n)$$

3. Average Case:

An average case can be defined by examining all possible positions of the TARGET within the list and taking the average time requirement assuming all possible positions are equally likely.

The calculation will be easier if we look at the case where $n = 2^N - 1$ for $N \in \mathbb{Z}^+$. Let's analyze the number of iterations of the while loop required for each element in the list.

When $n = 2^1 - 1 = 1$ there is only one element and it is found after one iteration. We could put this in a really boring table:

# elements	# iterations
1	1

When $n = 2^2 - 1 = 3$ there is one element which is found after one iteration and now let's stop to make an observation. The remaining elements will be in a left or right sublist and each of these sublists has length 1. It follows that effectively, one iteration later, we have two copies of the $N = 1$ case, meaning we will have two copies of that case with one more iteration each:

# elements	# iterations
1	1
$2(1)=2$	$1+1 = 2$

When $n = 2^3 - 1 = 7$ there is one element which is found after one iteration and then the remaining elements will be in a left or right sublist each of which is a copy of the $N = 2$ case so we'll have two copies of that case with one more iteration each:

# elements	# iterations
1	1
$2(1)=2$	$1+1=2$
$2(2)=4$	$2+1=3$

And again for $n = 2^4 - 1 = 15$:

# elements	# iterations
1	1
$2(1)=2$	$1+1 = 2$
$2(2)=4$	$2+1 = 3$
$2(4)=8$	$3+1 = 4$

In general when $n = 2^N - 1$ we have the following:

# elements	# iterations
1	1
2	2
4	3
8	4
\vdots	\vdots
2^{N-1}	N

Now then, the probability of getting an element requiring some number of iterations equals the number of such elements divided by n , and i iterations takes $c_1 + ic_2$ time, so we can write the following table:

Probability	Time
$\frac{1}{n}$	$c_1 + 1c_2$
$\frac{2}{n}$	$c_1 + 2c_2$
$\frac{4}{n}$	$c_1 + 3c_2$
$\frac{8}{n}$	$c_1 + 4c_2$
\vdots	\vdots
$\frac{2^{N-1}}{n}$	$c_1 + Nc_2$

The expected time is then:

$$\begin{aligned}
\sum_{i=1}^N (\text{Probability})(\text{Time}) &= \sum_{i=1}^N \frac{2^{i-1}}{n} [c_1 + ic_2] \\
&= \frac{1}{n} \sum_{i=1}^N 2^{i-1} [c_1 + ic_2] \\
&= \frac{1}{n} \left[c_1 \sum_{i=1}^N 2^{i-1} + c_2 \sum_{i=1}^N i2^{i-1} \right] \\
&= \dots = \Theta(\lg n)
\end{aligned}$$

5 Thoughts, Problems, Ideas

1. Show the steps of binary search when looking for the value 17 in the list $\{-3, 4, 7, 17, 20, 30, 40, 51, 105, 760\}$. At each step give the value of C and how the comparisons update L and R . Inside the `while` loop how many comparisons are made?
2. How does the particular pseudocode in the notes behave if the target exists at multiple indices?
3. Adjust the pseudocode so that if the target exists at multiple indices the function returns the first occurrence. Find the \mathcal{O} worst-case time complexity of this pseudocode.
4. Adjust the pseudocode so that if the target exists at multiple indices the function returns the last occurrence. Find the \mathcal{O} worst-case time complexity of this pseudocode.
5. Assuming the list values are distinct, adjust the pseudocode so that if the target does not exist the function returns the smallest value larger than the target.
6. Explain how your answer to the previous question can be used to modify InsertSort. Pseudocode is not necessary, a good explanation will suffice.
7. In the pseudocode implementation in the notes the worst-case total time requirement is $T_B(n) = c_1 + c_2(1 + \lg n)$. A straightforward linear search is linear, something like $T_L(n) = c_3 + c_4n$. If $c_1 = 2$, $c_2 = 10$ (it's a big compound statement), $c_3 = 1$ and $c_4 = 2$, Plot these functions together on an axis. for which n will each be faster? This question is intended to be computer-assisted.
8. Suppose the sorted list A is infinitely long. In other words think of A as an increasing function defined for all integers $n \geq 0$. Here some ideas for extending binary search:
 - Start with $L=0$ (of course) and R set at some arbitrary positive integer.
 - If we have ever found some C with $\text{TARGET} < A[C]$ then we can basically just do binary search on $A[0, \dots, C]$.
 - If we have never found some C with $\text{TARGET} < A[C]$ then each time we encounter $\text{TARGET} > A[C]$ we double R and continue.
 - (a) Show how modifying the algorithm this way and using $R=4$ and $\text{TARGET}=15$ will work with the list:
$$A = [0, 4, 5, 10, 11, 12, 15, 16, 18, 20, 30, 31, 50, 100, 117, 118, 119, 200, 203, \dots]$$
 - (b) Write the pseudocode for this algorithm.

9. Suppose that due to some noise in the data exactly one of the comparisons will register as incorrect.
 - (a) Give a specific example to illustrate that Binary Search could completely fail.
 - (b) Give a specific example to illustrate that Binary Search might still work.
10. Suppose the two-dimensional array A of size $n \times m$ which is indexed as $A[0, \dots, n-1][0, \dots, m-1]$ contains integers with the property that every entry is greater than or equal to the entry directly above it and is greater than or equal to the entry directly to the left of it.

An example of such an array is:

$$\begin{bmatrix} 2 & 7 & 8 & 10 & 14 \\ 3 & 8 & 9 & 11 & 15 \\ 6 & 10 & 12 & 12 & 20 \\ 7 & 11 & 13 & 13 & 21 \\ 8 & 11 & 15 & 20 & 22 \end{bmatrix}$$

Write the pseudocode for a function `binarysearch2d` which locates a target entry in the array and returns the location. Your algorithm should use a 2D version of binary search.

Hint: In the above example the middle entry is 12. If the target is less than 12 where could it be? If the target is more than 12 where could it be?

11. Binary Search only works on a sorted list. Suppose we have an unsorted list A and we wish to check if an element is in the list. We could sort it using one of our three sorts and then check if Binary Search returns FAIL or not. From a time perspective why is this a bad choice?

6 Python Test

Code:

```
import random
import math
def binarysearch(A,TARGET):
    L = 0
    R = len(A)-1
    while L <= R:
        print('Checking: '+str(A[L:R+1]))
        C = math.floor((L+R)/2)
        if A[C] == TARGET:
            print('Checking: [ ' + str(A[C]) + '] ')
            return(C)
        elif TARGET < A[C]:
            R = C-1
        elif TARGET > A[C]:
            L = C+1
    return(False)
A = []
for i in range(0,20):
    A.append(random.randint(0,50))
A.sort()
print(A)
TARGET = 10
result = binarysearch(A,TARGET)
if result == False:
    print('Not Found')
else:
    print('Found at index ' + str(result))
```

Output:

```
[1, 2, 7, 10, 12, 14, 15, 16, 17, 17, 22, 23, 28, 30, 37, 43, 45, 46, 49,
Checking: [1, 2, 7, 10, 12, 14, 15, 16, 17, 17, 22, 23, 28, 30, 37, 43, 45
Checking: [1, 2, 7, 10, 12, 14, 15, 16, 17]
Checking: [1, 2, 7, 10]
Checking: [7, 10]
Checking: [10]
Checking: [10]
Found at index 3
```

And:

```
[3, 3, 3, 5, 5, 6, 7, 8, 9, 15, 20, 22, 26, 28, 28, 29, 30, 32, 39, 49]
Checking: [3, 3, 3, 5, 5, 6, 7, 8, 9, 15, 20, 22, 26, 28, 28, 29, 30, 32,
Checking: [3, 3, 3, 5, 5, 6, 7, 8, 9]
Checking: [6, 7, 8, 9]
Checking: [8, 9]
Checking: [9]
Not Found
```

Fun fact related to `A.sort()`: Python's default `sort` method uses Timsort. Timsort is a merge sort and insertion sort hybrid which works well on real-world data in which there are often *runs* of sorted sublists within the list. Timsort essentially collects and merges those runs. Timsort is $\mathcal{O}(n \lg n)$.

CMSC 351: Recurrence Relations

Justin Wyss-Gallifent

September 28, 2023

1	Introduction	2
2	Solving Using Digging Down	3
3	Solving Using Trees	3
4	Thoughts, Problems, Ideas	10

1 Introduction

Suppose we are analyzing the time complexity $T(n)$ for an algorithm and suppose that we cannot find an explicit closed formula for $T(n)$ but instead we find a recurrence relation which $T(n)$ satisfies.

A recurrence relation for $T(n)$ tells us how to calculate $T(n)$ for various n in a recursive manner.

Example 1.1. Suppose we have:

$$T(n) = 3T(\lfloor n/5 \rfloor) + 2n + 1 \text{ and } T(1) = 4$$

Then for example here are some easy ones:

$$\begin{aligned} T(1) &= 4 \\ T(5) &= 3T(1) + 2(5) + 1 = 23 \\ T(25) &= 3T(4) + 2(25) + 1 = 120 \end{aligned}$$

A slightly more annoying one is:

$$T(7) = 3T(\lfloor 7/5 \rfloor) + 2(7) + 1 = 3T(1) + 15 = 27$$

An even more annoying one would be $T(2)$ because we're not given enough information. in reality we would need to be given $T(2)$, $T(3)$, and $T(4)$ as well. Luckily since we're only really usually really concerned with large n values we can get away with minimal base cases. ■

Formally a recurrence relation like this should have a floor or ceiling inside the recursive T but in practice this typically dropped for purposes of not getting too tangled up in the calculations.

For example we might just write:

$$T(n) = 3T(n/5) + 2n + 1 \text{ and } T(1) = 4$$

When we do this every specific calculation that follows becomes an approximation when the division yields a non-integer but these approximations are good enough and don't affect time complexity.

There are two goals we might have with a recurrence relation:

1. Find a closed expression for the function, meaning a $T(n) = \dots$ which isn't self-referential, or with summations, etc.
2. Find Θ for $T(n)$ if a closed expression is difficult.

2 Solving Using Digging Down

One way to obtain a closed expression is to dig into the recurrence relation. Consider the example:

$$T(n) = 2T(n/2) + \frac{1}{2}n \text{ with } T(1) = 7$$

We engage in a process of substitution:

$$\begin{aligned} T(n) &= 2T(n/2) + \frac{1}{2}n \\ &= 2 \left[2T(n/4) + \frac{1}{2}(n/2) \right] + \frac{1}{2}n \\ &= 4T(n/4) + n \\ &= 4 \left[2T(n/8) + \frac{1}{2}(n/4) \right] + n \\ &= 8T(n/8) + \frac{3}{2}n \\ &= 8 \left[2T(n/16) + \frac{1}{2}(n/8) \right] + \frac{3}{2}n \\ &= 16T(n/16) + 2n \\ &= \dots \end{aligned}$$

But how and when does it end?

Well, the general expression for the above is, for $k = 1, 2, 3, \dots$:

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + \frac{k}{2}n$$

This ends when $n/2^k = 1$ since then we get $T(1) = 7$. This is when $k = \lg n$. At that instant the expression becomes:

$$\begin{aligned} T(n) &= 2^{\lg n} T(1) + \frac{1}{2}n \lg n \\ &= 7n + \frac{1}{2}n \lg n \end{aligned}$$

We also see $T(n) = \Theta(n \lg n)$ at this point.

3 Solving Using Trees

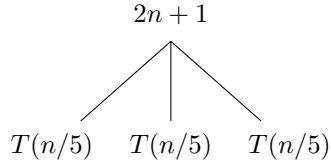
We can also use a recursively generated tree to find an explicit formula for $T(n)$. Such an approach can be messy or not, depending on the recurrence relation

and on the n -values we're analyzing. So as not to go off the deep end, let's consider the example again:

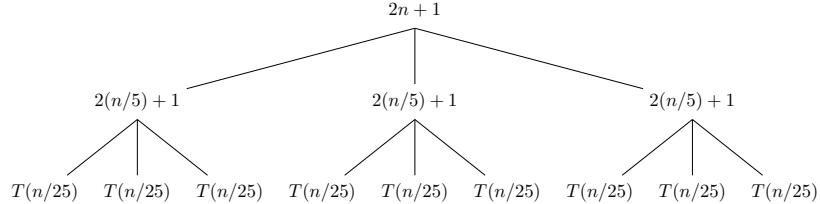
Example 3.1. Suppose $T(n) = 3T(n/5) + (2n + 1)$ with $T(1) = 4$. Let's examine $T(n)$ where $n = 5^k$ for some k . To understand why there's a tree involved we can view the total time done as a very small tree, first with one node:

$$T(n)$$

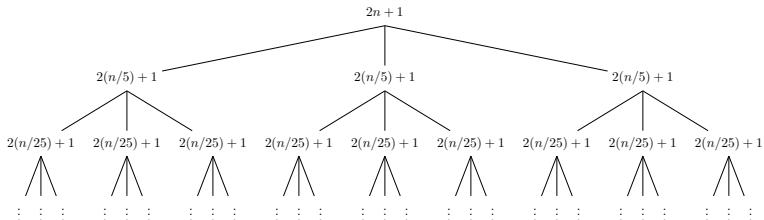
We can expand this according to the recurrence relation, the tree still showing the total time:



Of course each of these leaves is its own problem, the tree still showing the total time.



We could keep going...



But when does it stop?

The tree stops growing when we reach $T(1)$ in the nodes because $T(1) = 4$ and no more recursion happens.

If level $i = 0$ is the top level then as the tree grows we see that level i has entries $T(n/5^i)$ so the leaf level will happen when we reach $T(1)$ which is when $n/5^i = 1$ or $i = \log_5 n = k$. Thus the k^{th} level is the leaf level.

Thus in total there are $k + 1$ levels, $0, 1, \dots, k$, and in those levels:

Level	Count	Time	Level Time
$i = 0$	1	$2n + 1$	$1(2n + 1) = 3^0 (2(\frac{n}{5^0}) + 1)$
$i = 1$	3	$2(\frac{n}{5}) + 1$	$3(2(\frac{n}{5}) + 1) = 3^1 (2(\frac{n}{5^1}) + 1)$
$i = 2$	9	$2(\frac{n}{25}) + 1$	$9(2(\frac{n}{25}) + 1) = 3^2 (2(\frac{n}{5^2}) + 1)$
\vdots	\vdots	\vdots	\vdots
$i = k - 1$	3^{k-1}	$2(\frac{n}{5^{k-1}}) + 1$	$3^{k-1} (2(\frac{n}{5^{k-1}}) + 1)$
$i = k$	3^k	4	$3^k (4)$

Thus the total time is the sum of the levels:

$$\begin{aligned}
T(n) &= 4(3^k) + \sum_{i=0}^{k-1} 3^i \left(2\left(\frac{n}{5^i}\right) + 1 \right) \\
&= 4(3^k) + 2n \sum_{i=0}^{k-1} \left(\frac{3}{5}\right)^i + \sum_{i=0}^{k-1} 3^i \\
&= 4(3^k) + 2n \left(\frac{1 - (\frac{3}{5})^k}{1 - \frac{3}{5}} \right) + \left(\frac{3^k - 1}{3 - 1} \right) \\
&= 4(3^k) + 2n \left(\frac{5}{2} \right) \left(1 - \left(\frac{3}{5} \right)^k \right) + \frac{1}{2} (3^k - 1) \\
&= 4(3^{\log_5 n}) + 5n - 5n \left(\frac{3}{5} \right)^{\log_5 n} + \frac{1}{2} (3^{\log_5 n}) - \frac{1}{2} \\
&= \frac{9}{2}(3^{\log_5 n}) + 5n - 5n \left(\frac{3^{\log_5 n}}{5^{\log_5 n}} \right) - \frac{1}{2} \\
&= \frac{9}{2}(3^{\log_5 n}) + 5n - 5(3^{\log_5 n}) - \frac{1}{2} \\
&= -\frac{1}{2}(3^{\log_5 n}) + 5n - \frac{1}{2} \\
&= 5n - \frac{1}{2}(3^{\log_5 n} + 1)
\end{aligned}$$

Note that results from this equation will agree with the calculations we did earlier. For example:

$$T(5) = 5(5) - \frac{1}{2}(3^{\log_5 5} + 1) = 25 - \frac{1}{2}(3 + 1) = 23$$

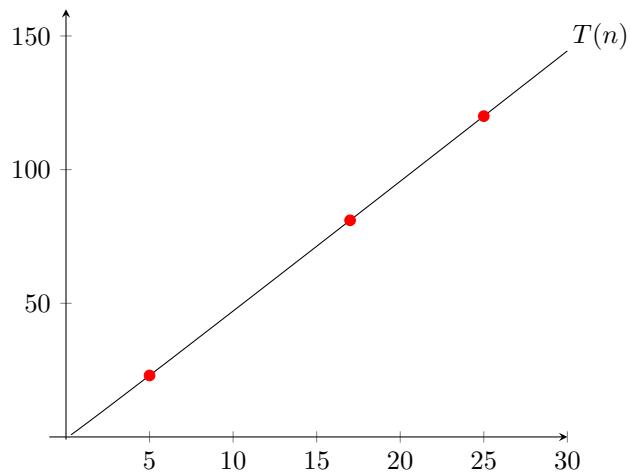
And:

$$T(25) = 5(25) - \frac{1}{2}(3^{\log_5 25} + 1) = 125 - \frac{1}{2}(9 + 1) = 120$$

While this formula is exact only for $n = 5^k$ it gives us a good approximation in other cases:

$$T(17) = 5(17) - \frac{1}{2}(3^{\log_5 17} + 1) \approx 81.04$$

Here is a graph of $T(n)$ as well as these points:



As far as time complexity observe that:

$$T(n) = 5n - \frac{1}{2}(3^{\log_5 n} + 1) < 5n$$

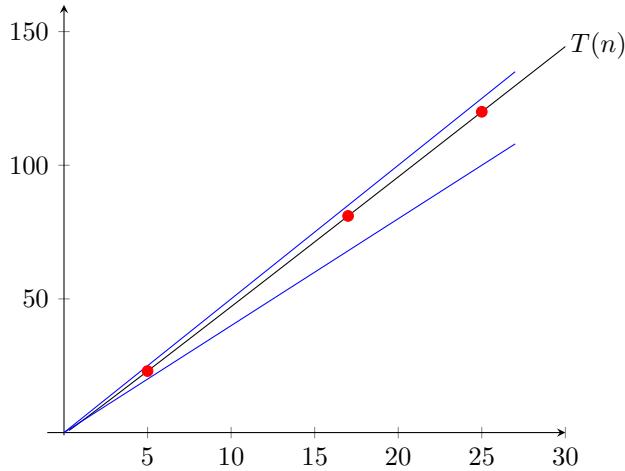
Thus $T(n) = \mathcal{O}(n)$.

And observe that since $\log_5 n < \log_3 n$ we have $3^{\log_5 n} < 3^{\log_3 n} = n$ so that for $n \geq 1$ we have:

$$T(n) = 5n - \frac{1}{2}(3^{\log_5 n} + 1) > 5n - \frac{1}{2}(n + 1) = \frac{9}{2}n - \frac{1}{2} \geq 4n$$

Thus $T(n) = \Omega(n)$ and together we have $T(n) = \Theta(n)$.

This corresponds to the picture in the sense that it certainly appears that $Bn \leq T(n) \leq Cn$ for some $B, C > 0$ and for sufficiently large n . In fact here is the same plot but with $4n$ and $5n$ plotted as well:



■

By popular demand here is a second tree example but more streamlined.

Example 3.2. Consider the recurrence relation given here:

$$T(n) = 2T(n/2) + \sqrt{n} \text{ with } T(1) = 3$$

For the sake of simplicity assume $n = 2^k$ for some k . During the growth of the tree the nodes in level i are $T(n/2^i)$ before being replaced by $\sqrt{n/2^i}$ as the tree grows down. The leaf level occurs when $n/2^i = 1$ or $i = \lg n = k$.

Thus in total there are $k + 1$ levels, $0, 1, \dots, k + 1$ and in those levels:

Level	Count	Time	Total Time
$i = 0$	1	\sqrt{n}	$1\sqrt{n} = \sqrt{1n}$
$i = 1$	2	$\sqrt{n/2}$	$2\sqrt{n/2} = \sqrt{2n}$
$i = 1$	4	$\sqrt{n/4}$	$4\sqrt{n/4} = \sqrt{4n}$
\vdots	\vdots	\vdots	\vdots
$i = k - 1$	2^{k-1}	$\sqrt{n/2^{k-1}}$	$2^{k-1}\sqrt{n/2^{k-1}} = \sqrt{2^{k-1}n}$
$i = k$	2^k	3	$2^k (3)$

Thus the total time is:

$$\begin{aligned}
 T(n) &= 3(2^k) + \sum_{i=0}^{k-1} \sqrt{2^i n} \\
 &= 3n + \sqrt{n} \sum_{i=0}^{k-1} (2^i)^{1/2} \\
 &= 3n + \sqrt{n} \sum_{i=0}^{k-1} (2^{1/2})^i \\
 &= 3n + \sqrt{n} \left(\frac{(2^{1/2})^k - 1}{2^{1/2} - 1} \right) \\
 &= 3n + \sqrt{n} \left(\frac{(2^k)^{1/2} - 1}{2^{1/2} - 1} \right) \\
 &= 3n + \sqrt{n} \left(\frac{\sqrt{n} - 1}{2^{1/2} - 1} \right) \\
 &= 3n + \frac{1}{\sqrt{2} - 1} (n - \sqrt{n}) \\
 &= 3n + (\sqrt{2} + 1)(n - \sqrt{n})
 \end{aligned}$$

This checks with the recurrence relation since for example the recurrence relation gives us $T(2) = 2T(1) + \sqrt{2} = 6 + \sqrt{2}$ and this formula gives us

$T(2) = 3(2) + (\sqrt{2} + 1)(2 - \sqrt{2}) = 6 + \sqrt{2}$ and for example the recurrence relation gives us $T(4) = 2T(2) + \sqrt{4} = 2(6 + \sqrt{2}) + 2 = 14 + 2\sqrt{2}$ and this formula gives us $T(4) = 3(4) + (\sqrt{2} + 1)(4 - \sqrt{4}) = 14 + 2\sqrt{2}$.

■

4 Thoughts, Problems, Ideas

1. For the example $T(n) = 3T(n/5) + (2n + 1)$ with $T(1) = 4$ show that calculating $T(125)$ directly (using the recurrence relation) and using the formula developed in the notes yields the same results.
2. For the example $T(n) = 2T(n/5) + (2n + 3)$ with $T(1) = 4$ draw the complete tree for $n = 125$ and fill in the values. What is the total time?
3. Suppose $T(n) = 2T(n/5) + (2n + 1)$ and $T(1) = 2$. Emulate the example in the notes in the following sense:
 - (a) Calculate $T(n)$ for a few values which are nice powers (of what?)
 - (b) Draw a generic version of the associated tree.
 - (c) Calculate the number of levels in the tree.
 - (d) Calculate the number of entries in each level.
 - (e) Separately for each non-leaf level calculate the total time in each entry and then add these to get the total time each non-leaf level.
 - (f) Calculate the total time in the leaf-level.
 - (g) Write down a sum for the total time in the tree.
 - (h) Simplify this sum to get the total time $T(n)$.
 - (i) Use this $T(n)$ to check your values from (a).
 - (j) Calculate the $\mathcal{O}(n)$ time complexity from $T(n)$.
 - (k) Calculate the time complexity from the Master Theorem.
 - (l) Rejoice at the beauty of equality.
4. Repeat the previous problem with $T(n) = 2T(n/4) + \sqrt{n}$ and $T(1) = 4$.
5. Repeat the previous problem with $T(n) = 3T(n/3) + 2$ and $T(1) = 7$.

CMSC 351: The Master Theorem

Justin Wyss-Gallifent

March 11, 2024

1	The Theorem (Straightforward Version)	2
2	Application and Examples	2
3	Motivation Behind the Theorem	6
3.1	Intuition Without the $f(n)$	6
3.2	Proof Without the $f(n)$	7
3.3	Intuition with the $f(n)$	7
4	Even More Rudimentary Intuition	8
5	Thoughts, Problems, Ideas	9

1 The Theorem (Straightforward Version)

Of course we would rather not do this sort of calculation every time so we might ask if there are reliable formulas which emerge in specific situations and the answer is yes, and these are encapsulated in the Master Theorem:

Theorem 1.0.1. Suppose $T(n)$ satisfies the recurrence relation:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

for positive integers $a \geq 1$ and $b > 1$ and where $\frac{n}{b}$ can mean either $\lfloor \frac{n}{b} \rfloor$ or $\lceil \frac{n}{b} \rceil$, it doesn't matter which. Then we have:

1. If $f(n) = \mathcal{O}(n^c)$ and $\log_b a > c$ then $T(n) = \Theta(n^{\log_b a})$.
 2. If $f(n) = \Theta(n^c)$ and $\log_b a = c$ then $T(n) = \Theta(n^{\log_b a} \lg n)$.
 - 2f. (Fancy Version) If $f(n) = \Theta(n^c \lg^k n)$ and $\log_b a = c$ then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
 3. If $f(n) = \Omega(n^c)$ and $\log_b a < c$ then $T(n) = \Theta(f(n))$.
- Note: For this case, $f(n)$ must also satisfy a *regularity condition* which states that there is some $C < 1$ and n_0 such that $af(n/b) \leq Cf(n)$ for all $n \geq n_0$. This regularity condition is almost always true and we will not worry about it.

Proof. Formal proof omitted. See the intuition section later, if you're interested.
 \mathcal{QED}

2 Application and Examples

When applying the Master Theorem it can be helpful to successively try the cases until we find one that works. It's often easiest to check Case 2 and 2f first as we'll see.

Example 2.1. Consider:

$$T(n) = 4T(n/2) + n^2 + \lg n$$

Observe that $f(n) = n^2 + \lg n$ and $\log_b a = \log_2 4 = 2$ and then:

- $f(n) = \Theta(n^2)$ with $c = 2$:

Observe $\log_2 4 = 2 = c$ so Case 2 applies and so:

$$T(n) = \Theta(n^{\log_2 4} \lg n) = \Theta(n^2 \lg n)$$

Example 2.2. Consider:

$$T(n) = 2T(n/8) + n$$

Observe that $f(n) = n$ and $\log_b a = \log_8 2 = 1/3$ and then:

- $f(n) = \Theta(n)$ with $c = 1$:
Observe $\log_8 2 = 1/3 \neq 1 = c$ so Case 2/2f do not apply.
- $f(n) = \mathcal{O}(n)$ with $c = 1$:
Observe $\log_8 2 = 1/3 > 1 = c$ so Case 1 does not apply.
- $f(n) = \Omega(n)$ with $c = 1$:
Observe $\log_8 2 = 1/3 < 1 = c$ so Case 3 applies and so:

$$T(n) = \Theta(f(n)) = \Theta(n)$$

Example 2.3. Consider:

$$T(n) = 3T(n/3) + \sqrt{n} + 1$$

Observe that $f(n) = \sqrt{n} + 1$ and $\log_b a = \log_3 3 = 1$ and then:

- $f(n) = \Theta(n^{1/2})$ with $c = 1/2$:
Observe $\log_3 3 = 1 \neq 1/2 = c$ so Case 2/2f do not apply.
- $f(n) = \mathcal{O}(n^{1/2})$ with $c = 1/2$:
Observe $\log_3 3 = 1 > 1/2 = c$ so Case 1 applies and so:

$$T(n) = \Theta(n^{\log_3 3}) = \Theta(n)$$

Example 2.4. Consider:

$$T(n) = 3T(n/4) + n \lg n + n$$

Observe that $f(n) = n \lg n + n$ and $\log_b a = \log_4 3 \approx 0.*$ and then:

- $f(n) = \Theta(n \lg n)$ with $c = 1$:
Observe $\log_4 3 \approx 0.* \neq 1 = c$ so Case 2/2f do not apply.
- $f(n) = \mathcal{O}(n^2)$ with $c = 2$:
Observe $\log_4 3 \approx 0.* > 2 = c$ so Case 1 does not apply.
- $f(n) = \Omega(n)$ with $c = 1$:
Observe $\log_4 3 \approx 0.* < 1 = c$ so Case 3 applies and so:

$$T(n) = \Theta(f(n)) = \Theta(n \lg n)$$

Example 2.5. Consider:

$$T(n) = 10T(n/2) + n^2 \lg n + n^2 + 1$$

Observe that $f(n) = n^2 \lg n + n^2 + 1$ and $\log_b a = \log_2 10 \approx 3.*$ and then:

- $f(n) = \Theta(n^2 \lg n)$ with $c = 2$:
Observe $\log_2 10 \approx 3.* \neq 2 = c$ so Case 2/2f do not apply.
- $f(n) = \mathcal{O}(n^3)$ with $c = 3$:
Observe $\log_2 10 \approx 3.* > 3 = c$ so Case 1 applies and so:

$$T(n) = \Theta(n^{\log_2 10})$$

Example 2.6. Consider:

$$T(n) = 9T(n/3) + n^2 \lg n + \lg n$$

Observe that $f(n) = n^2 \lg n + \lg n$ and $\log_b a = \log_3 9 = 2$ and then:

- $f(n) = \Theta(n^2 \lg n)$ with $c = 2$:
Observe $\log_3 9 = 2 = c$ so Case 2f applies with $k = 1$ and so:

$$T(n) = \Theta(n^{\log_3 9} \lg^2 n) = \Theta(n^2 \lg^2 n)$$

Here is a slightly trickier example:

Example 2.7. Consider:

$$T(n) = 16T(n/2) + n^3 \lg n$$

Observe that $f(n) = n^3 \lg n$ and $\log_b a = \log_2 16 = 4$. Now observe:

- $f(n) = \Theta(n^3 \lg n)$ with $c = 2$ and $k = 1$ so it looks like Case 2f with $c = 3$ and $k = 1$ but $\log_b a = 4 \neq 3 = c$ so it's not Case 2f.
- $f(n) = \mathcal{O}(n^4)$ with $c = 4$ but $\log_b a = 4 > 4 = c$ so it's not Case 1. Or it is?
- $f(n) = \mathcal{O}(n^{3.5})$ (as can be proven with the limit theorem) with $c = 3.5$ and $\log_b a = 4 > 3.5$ so it is in fact Case 1 when treated carefully and so:

$$T(n) = \Theta(n^{\log_2 16}) = \Theta(n^4)$$

Here are some examples where it does not apply:

Example 2.8. Suppose $T(n) = 2T(n/4) + 3T(n/2) + n$.

The Master Theorem does not apply because it has the wrong form. Note that there is another method which often applies called the Akra-Bazzi method. It applies to recurrence formulas of the following form under certain

conditions:

$$T(n) = f(n) + \sum_{i=1}^k a_i T(b_i n + h_i(n))$$

We will not cover it.

Example 2.9. Suppose $T(n) = 2T(n/4) + f(n)$ and all you know is that $f(n) = \mathcal{O}(n^2)$.

The fact that $f(n) = \mathcal{O}(n^2)$ implies that we could only use Case 1 and insists that $c = 2$. However $\log_b a = \log_4 2 = 0.5 \not> 2 = c$ and so Case 1 does not apply.

Example 2.10. Suppose $T(n) = 8T(n/4) + f(n)$ and all you know is that $f(n) = \Omega(n)$.

The fact that $f(n) = \Omega(n)$ implies that we could only use Case 3 and insists that $c = 1$. However $\log_b a = \log_4 8 = \frac{3}{2} \not< 1 = c$ and so Case 3 does not apply.

3 Motivation Behind the Theorem

3.1 Intuition Without the $f(n)$

If $f(n) = 0$ then $f(n) = \mathcal{O}(1) = \mathcal{O}(n^0)$ and $0 < \log_b a$ as long as $a > 1$ (which it is) and so all what follows here lies in the first case of the Master Theorem.

Consider a divide-and-conquer algorithm which breaks a problem of size n into a subproblems each of size n/b . In such a case we would have:

$$T(n) = aT(n/b)$$

Now observe:

- It seems reasonable that if $a = b$ then we have no overall gain because the number of new problems equals the reducing ratio (for example two problems of half the size doesn't help) but we can actually say more.

If we assume a reasonable $T(1) = \alpha$ for some constant α then this is essentially saying, for example, that $T(2) = 2T(2/2) = 2(1) = 2\alpha$, $T(4) = 2T(4/2) = 2(2) = 4\alpha$, $T(8) = 2T(8/2) = 2(4) = 8\alpha$, and so on, and in general it seems reasonable that $T(n) = n\alpha = \Theta(n)$.

This also seems reasonable with any $a = b$ (not just 2), that we still get $T(n) = \Theta(n)$.

This arises in the first case of the Master Theorem because if $a = b$ then $\log_b a = 1$ and then $T(n) = \Theta(n^{\log_b a}) = \Theta(n^1)$.

- On the other hand if $b > a$ then we have an overall decrease in time, for example if $T(n) = 2T(n/3)$ then the subproblems are $1/3$ the size and there are only two, that's good, better than $\Theta(n)$!

This arises in the first case of the Master Theorem because if $b > a$ then $\log_b a < 1$ and then $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\text{less than } 1})$.

- And on the other hand if $b < a$ then we have an overall gain in time, for example if $T(n) = 3T(n/2)$ then the subproblems are $1/2$ the size but there are three, that's bad, worse than $\Theta(n)$!

This arises in the first case of the Master Theorem because if $b < a$ then $\log_b a > 1$ and then $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\text{more than } 1})$.

3.2 Proof Without the $f(n)$

If $f(n) = 0$ we can in fact solve the recurrence relation easily using the digging-down approach. If we accept a base case of $T(1)$ then we have:

$$\begin{aligned} T(n) &= aT(n/b) \\ &= a^2T(n/b^2) \\ &= a^3T(n/b^3) \\ &\vdots \quad \vdots \end{aligned}$$

For any $k \geq 1$ we have $T(n) = a^k T(n/b^k)$ which ends when $n/b^k = 1$ which is $k = \log_b n$.

Thus we get:

$$\begin{aligned} T(n) &= a^k T(1) \\ &= a^{\log_b n} T(1) \\ &= a^{(\log_a n / \log_a b)} T(1) \\ &= (a^{\log_a n})^{1/\log_a b} T(1) \\ &= n^{1/\log_a b} T(1) \\ &= n^{\log_b a} T(1) \\ &= \Theta(n^{\log_b a}) \end{aligned}$$

3.3 Intuition with the $f(n)$

Now that we've intuitively and formally accepted that:

$$T(n) = aT(n/b) \implies T(n) = \Theta(n^{\log_b a})$$

Now let's suppose there is some additional time requirement $f(n)$ for a problem of size n .

- If this new time requirement is at most (meaning \mathcal{O}) polynomially smaller than the recursive part then the recursive part is the dominating factor. This is represented in the theorem by the line:

If $f(n) = \Theta(n^c)$ for $c < \log_b a$ then $T(n) = \Theta(n^{\log_b a})$.

- If this new time requirement is the same (meaning Θ) polynomially as the recursive part then they combine and a logarithmic factor is introduced (this is not obvious). This is represented in the theorem by the line:

If $f(n) = \Theta(n^c)$ for $c = \log_b a$ then $T(n) = \Theta(n^{\log_b a} \lg n)$.

- If this new time requirement is at least (meaning Ω) polynomially larger than the recursive part then this new time requirement is the dominating factor. This is represented in the theorem by the line:

If $f(n) = \Theta(n^c)$ for $c > \log_b a$ then $T(n) = \Theta(f(n))$.

Along with the regularity condition.

4 Even More Rudimentary Intuition

To think even more crudely, but not inaccurately, when you see the recurrence relation:

$$T(n) = aT(n/b) + f(n)$$

Think of the $aT(n/b)$ as the tree structure and $f(n)$ as the node weights. Basically the Master Theorem is saying:

1. If the tree structure is greater than the node weights then the node weights don't matter and the tree structure wins - winter.
2. If they balance out nicely then they combine - spring.
3. If the tree structure is less than the node weights then the tree structure doesn't matter and the node weights win - summer.

5 Thoughts, Problems, Ideas

1. Suppose $T(n) = 5T(n/5) + f(n)$.
 - (a) Apply the Master Theorem with $f(n) = \sqrt{n}$.
 - (b) Apply the Master Theorem with $f(n) = n + \sqrt{n}$.
 - (c) Apply the Master Theorem with $f(n) = 3n + \sqrt{n^3}$.
 - (d) Apply the Master Theorem with $f(n) = n \lg n$.
2. Suppose $T(n) = 4T(n/8) + f(n)$.
 - (a) Apply the Master Theorem with $f(n) = \sqrt{n}$.
 - (b) Apply the Master Theorem with $f(n) = n^{2/3} + \lg n$.
 - (c) Apply the Master Theorem with $f(n) = n$.

3. Suppose $T(n) = 4T(n/2) + f(n)$.
 - (a) Apply the Master Theorem with $f(n) = n + \sqrt{n}$.
 - (b) Apply the Master Theorem with $f(n) = n^2 + n + 1$.
 - (c) Apply the Master Theorem with $f(n) = n^3 \lg n$.
4. Binary Search has $T(n) = T(n/2) + \Theta(1)$. What is $T(n)$?
5. Merge Sort has $T(n) = 2T(n/2) + \Theta(n)$. What is $T(n)$?
6. The Max-Heapify routine in Heap Sort has $T(n) \leq T(2n/3) + \Theta(1)$. What is $T(n)$?
7. An optimal sorted matrix search has $T(n) = 2T(n/2) + \Theta(n)$. What is $T(n)$?
8. A divide and conquer algorithm which splits a list of length n into two equally sized lists, makes recursive calls to both and in addition uses constant time will have $T(n) = 2T(n) + C$. What is $T(n)$?

CMSC 351: MergeSort

Justin Wyss-Gallifent

October 4, 2023

1	What it Does	2
2	How it Works	2
3	Pseudocode:	3
4	Pseudocode Time Complexity:	4
5	Auxiliary Space	5
	5.1 Auxiliary Space without the Master Theorem 1	5
	5.2 Auxiliary Space without the Master Theorem 2	6
	5.3 Auxiliary Space with the Master Theorem	6
	5.4 Auxiliary Space Commentary	7
6	Stability	7
7	In-Place	7
8	Notes	7
9	Thoughts, Problems, Ideas	8
10	Python Test	9

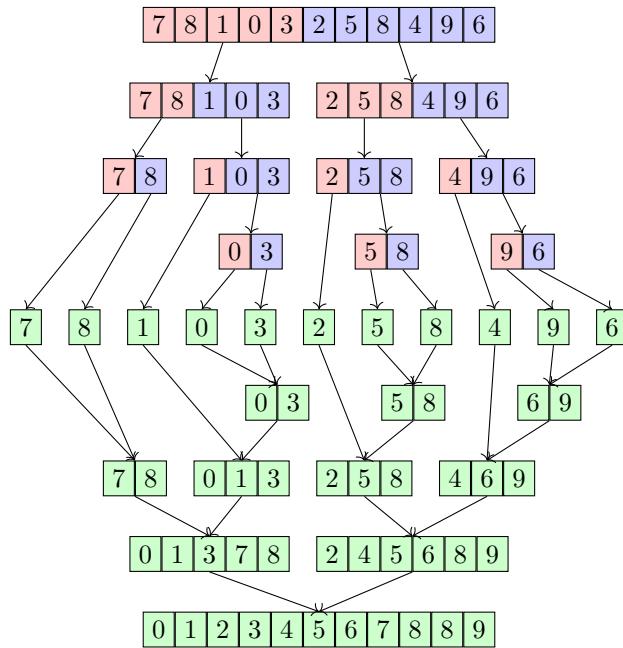
1 What it Does

Sorts a list of elements on which there is a total order. Think of integers or real numbers.

2 How it Works

Merge sort is a divide-and-conquer algorithm whereby the list is subdivided repeatedly in half. Each half is then divided in half again and so on until each sublist has size 1 and is obviously sorted. Pairs of sublists are then merged to preserve the sort.

Here is a visual representation. The red/blue divisions are illustrating how each (sub)list is divided in half. Any green element or group of elements are sorted. All the action above the center line is the recursive deconstruction while all the action below the center line is the re-merging of the sublists.



3 Pseudocode:

Nothing is assumed to be global here.

```
\\" PRE: A is a list of integers.
function mergesort(A)
    if len(A) > 1
        m = len(A) // 2
        L = A[0, ..., m-1]
        R = A[m, ..., len(A)-1]
        L = mergesort(L)
        R = mergesort(R)
        \\" Merge L and R back on top of A.
        Lind = 0
        Rind = 0
        Aind = 0
        while Lind < len(L) and Rind < len(R)
            if L[Lind] <= R[Rind]:
                A[Aind] = L[Lind]
                Lind ++
                Aind ++
            else:
                A[Aind] = R[Rind]
                Rind ++
                Aind ++
        end
        while Lind < len(L)
            A[Aind] = L[Lind]
            Lind ++
            Aind ++
        end
        while Rind < len(R)
            A[Aind] = R[Rind]
            Rind ++
            Aind ++
        end
    end
    return(A)
end
\\" POST: A is sorted.
```

A comment on the merging of L and R: we initialize indices for each of these and, while there are elements left in both, copy the smaller one off the corresponding list and overwrite it onto A. Once one has no elements remaining we simply copy all the elements in the other, one-by-one, and overwrite them onto A.

4 Pseudocode Time Complexity:

Observe that other than the two recursive calls to `mergesort` there are constant time calculations and three `while` loops.

However observe that the three `while` loops together result in a total of n iterations because together they just merge L and R back together and since L and R together form A, the claim follows.

Together then, other than the two recursive calls, $\Theta(n)$ time is required. It follows that the time complexity on an input of size n therefore satisfies the recurrence relation:

$$T(n) = 2T(n/2) + f(n) \text{ with } T(1) \text{ constant and } f(n) = \Theta(n)$$

This recurrence relation can be solved either with a recurrence tree or with the Master Theorem, resulting in $T(n) = \Theta(n \lg n)$.

Note that this is best, worst, and average-case. This is because MergeSort breaks down the list and puts it back together no matter what, even if the list is sorted at the start. Moreover the process of sorting the recursive parts during the reconstruction process is no quicker whether the parts are sorted or not.

5 Auxiliary Space

5.1 Auxiliary Space without the Master Theorem 1

The auxiliary space is a little confusing so let's step through it carefully. Here is a really simple pseudocode simplification of `mergesort`.

```
\\" PRE: A is a list of integers.  
function mergesort(A)  
    if len(A) > 1  
        split A to L and R  
        L = mergesort(L)  
        R = mergesort(R)  
        A = merge L and R  
    end  
    return(A)  
end  
\\" POST: A is sorted.
```

Consider a list of length $n = 16$. Our first call to Mergesort (recursion level 0) requires auxiliary space of 16 to do `split A to L and R`. In other words this is simply to allocate L and R .

The call `L = mergesort(L)` will subsequently require 8 more but this call ends and that memory is released before the call `R = mergesort(R)`. Consequently recursion level 1 only requires 8 more in total. Think of these 8 as being used first to manage L and then to manage R .

Now then, recursion level 1 makes four calls down to recursion level 2 but again these are done in series (first half of L then second half of L then first half of R then second half of R) so at that level only 4 more are needed in total. Think of these 4 as being used four times at that level,

This continues until we have lists of length 1 which require no space as they are not split, just returned. Consequently in the $n = 16$ case the total auxiliary memory required is:

$$16 + 8 + 4 + 2$$

Similarly simplicity suppose we have a list of length $n = 2^k$. Then the total auxiliary memory required is:

$$\begin{aligned}
n + \frac{n}{2} + \frac{n}{4} + \dots + \frac{n}{2^{k-1}} &= n \left(\left(\frac{1}{2}\right)^0 + \left(\frac{1}{2}\right)^1 + \left(\frac{1}{2}\right)^2 + \dots + \left(\frac{1}{2}\right)^{k-1} \right) \\
&= n \left(\frac{\left(\frac{1}{2}\right)^k - 1}{\frac{1}{2} - 1} \right) \\
&= 2n \left(1 - \left(\frac{1}{2}\right)^k \right) \\
&= 2n \left(1 - \frac{1}{n} \right) \\
&= \Theta(n)
\end{aligned}$$

5.2 Auxiliary Space without the Master Theorem 2

If that explanation is confusing, think from smaller lists to larger.

A list of length 2 splits to 1 + 1, requiring 2.

A list of length 4 splits to 2 + 2, requiring 4. It then runs Mergesort on the first half, requiring an additional 2, but that call is done and over and the memory released before it runs Mergesort on the second half, meaning the 2 for the second half is not additional. We thus have a total of 4 + 2.

A list of length 8 splits to 4 + 4, requiring 8. It then runs Mergesort on the first half, requiring an additional 4 + 2, but that call is done and over and the memory released before it runs Mergesort on the second half, meaning the 4 + 2 for the second half is not additional. We thus have a total of 8 + 4 + 2.

In general a list of length 2^k splits to $2^{k-1} + 2^{k-1}$, requiring 2^k . It then runs Mergesort on the first half, requiring an additional $2^{k-1} + \dots + 4 + 2$, but that call is done and over and the memory released before it runs Mergesort on the second half, meaning the $2^{k-1} + \dots + 4 + 2$ for the second half is not additional. We thus have a total of:

$$2^k + 2^{k-1} + \dots + 4 + 2$$

If $n = 2^k$ this is exactly the same sum as the previous subsection calculation.

5.3 Auxiliary Space with the Master Theorem

The Master Theorem can be applied to the auxiliary space of MergeSort if we are particularly careful. We might be tempted to think that the auxiliary space $S(n)$ satisfies the recurrence relation $S(n) = 2S(n/2) + f(n)$ where $f(n) = \Theta(n)$ but this is false. The reason for this is what we saw above, that the two adjacent recursive calls to MergeSort are called in series and the auxiliary space for the first call is released before the second call. The recurrence relation above

suggests that we are adding the auxiliary space of the two adjacent calls, much as we would add the time requirement, and this is not true, rather we should only include it once.

Thus instead the recurrence relation satisfied by the auxiliary space is $S(n) = S(n/2) + f(n)$ with $f(n) = \Theta(n)$.

Since $\Theta(n) = \Theta(n^1) = \Theta(n^c)$ with $c = 1$ and since $\log_b a = \log_2 1 = 0 < 1 = c$ the third case of the Master Theorem is satisfied and $S(n) = \Theta(n^1) = \Theta(n)$.

5.4 Auxiliary Space Commentary

It is reasonable to ask: Instead of creating new lists, applying Merge Sort to those, then remerging them on top of A , why don't we just apply Merge Sort directly to the left and right sublists of A , keeping them in-place, and then merge them back on top of A ? The same idea could be rephrased in terms of using references to A (thereby not creating new lists).

This does not work. The issue can be illustrated with the following example. Suppose $A = [5, 8, 6, 7, 4, 3, 2, 1]$. If we split this into $[5, 8, 6, 7]$ and $[4, 3, 2, 1]$ (using the same A or just use references to these sublists) and if we sort these we get $[5, 6, 7, 8]$ and $[1, 2, 3, 4]$.

At this point we have $A = [5, 6, 7, 8, 1, 2, 3, 4]$. If we now try to merge the left and right halves and if we do this on top of A , the first thing we do is pick out the 1 (it's the smallest first element in the two halves) but then we plop it down on top of $A[0]$ so now $A = [1, 6, 7, 8, 1, 2, 3, 4]$. Now we have lost the 5.

The practical upshot is that we are forced to put the left and right halves in new lists so that we can merge them from those new lists back on top of A .

6 Stability

Our MergeSort pseudocode is stable.

7 In-Place

Our MergeSort pseudocode is not in-place.

8 Notes

MergeSort is not iterative in any sense which lends itself to an easy analysis of what any intermediate steps look like.

9 Thoughts, Problems, Ideas

1. Diagram the action of MergeSort on the list: 5,0,7,10,3,8,10,4,1.
2. Suppose that in the recurrence relation:

$$T(n) = 2T(n/2) + f(n) \text{ with } T(1) \text{ constant and } f(n) = \Theta(n)$$

we had $f(n) = 5n + 2$ and $T(1) = 1$. Use a recurrence tree to calculate the time requirement and show that the result is still $\Theta(n \lg n)$.

3. Suppose for the sake of argument that Merge Sort took $T_M(n) = 7n \lg n + 10n$ and another sort you had available called **supersort(A)** took $T_S(n) = \frac{2}{3}n^2 + n$. For which n is **supersort** actually faster and how could you combine the two algorithms to produce a best-of-both-worlds result?
4. Rewrite the pseudocode of Merge Sort so that it does a three-way split instead of a two-way split.
5. Explain why MergeSort is stable.

10 Python Test

Code:

```
import random
def mergesort(A, indent):
    print(indent * '_' + 'Mergesort:' + str(A))
    if len(A) > 1:
        m = len(A) // 2
        L = A[:m]
        R = A[m:]
        mergesort(L, indent+2)
        mergesort(R, indent+2)
        i = 0
        j = 0
        k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                A[k] = L[i]
                i += 1
            else:
                A[k] = R[j]
                j += 1
            k += 1
        while i < len(L):
            A[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            A[k] = R[j]
            j += 1
            k += 1
        print(indent * '_' + 'Merge:' + str(L) + ' and ' + str(R))
        print(indent * '_' + 'Result: ' + str(A))
A = []
for i in range(0,11):
    A.append(random.randint(0,100))
print(A)
mergesort(A,0)
print(A)
```

Output:

```
[93, 95, 44, 67, 11, 89, 10, 71, 11, 88, 50]
Mergesort:[93, 95, 44, 67, 11, 89, 10, 71, 11, 88, 50]
--Mergesort:[93, 95, 44, 67, 11]
----Mergesort:[93, 95]
-----Mergesort:[93]
-----Mergesort:[95]
----Merge:[93] and [95]
----Result: [93, 95]
----Mergesort:[44, 67, 11]
-----Mergesort:[44]
-----Mergesort:[67, 11]
-----Mergesort:[67]
-----Mergesort:[11]
----Merge:[67] and [11]
----Result: [11, 67]
----Merge:[44] and [11, 67]
----Result: [11, 44, 67]
--Merge:[93, 95] and [11, 44, 67]
--Result: [11, 44, 67, 93, 95]
--Mergesort:[89, 10, 71, 11, 88, 50]
----Mergesort:[89, 10, 71]
-----Mergesort:[89]
-----Mergesort:[10, 71]
-----Mergesort:[10]
-----Mergesort:[71]
----Merge:[10] and [71]
----Result: [10, 71]
----Merge:[89] and [10, 71]
----Result: [10, 71, 89]
----Mergesort:[11, 88, 50]
-----Mergesort:[11]
-----Mergesort:[88, 50]
-----Mergesort:[88]
-----Mergesort:[50]
----Merge:[88] and [50]
----Result: [50, 88]
----Merge:[11] and [50, 88]
----Result: [11, 50, 88]
--Merge:[10, 71, 89] and [11, 50, 88]
--Result: [10, 11, 50, 71, 88, 89]
Merge:[11, 44, 67, 93, 95] and [10, 11, 50, 71, 88, 89]
Result: [10, 11, 11, 44, 50, 67, 71, 88, 89, 93, 95]
[10, 11, 11, 44, 50, 67, 71, 88, 89, 93, 95]
```

CMSC 351: HeapSort

Justin Wyss-Gallifent

March 7, 2024

1	Complete Binary Trees	2
1.1	Definition	2
1.2	Index Notes	2
1.3	Level Notes	3
2	Max Heaps	3
2.1	Definition	3
3	Converting to a Max Heap	4
3.1	Max Heapify - Fixing a Node	4
3.2	Convert to Max Heap - Fixing a Tree	6
4	Heapsort	8
4.1	Algorithm	8
4.2	Heapsort Worst-Case Time Complexity	13
4.3	Heapsort Best-Case Time Complexity	14
4.4	Heapsort Auxiliary Space	14
4.5	Heapsort Stability	14
4.6	Heapsort In-Place	14
4.7	Heapsort Usage Note	14
5	Pseudocode for Everything	15
5.1	Pseudocode for Maxheapify	15
5.2	Pseudocode for Converttomaxheap	15
5.3	Pseudocode for Heapsort	16
6	Thoughts, Problems, Ideas	16
7	Python Test	18

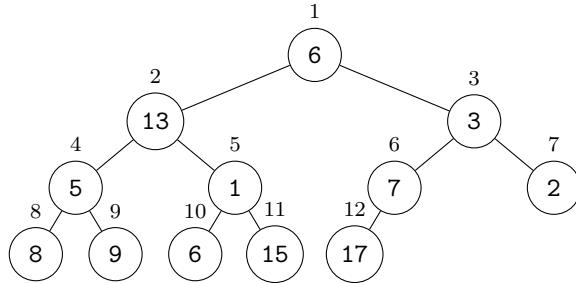
1 Complete Binary Trees

1.1 Definition

Definition 1.1.1. A *complete binary tree* is a binary tree in which all levels are completely filled, except possibly for the bottom level, and the bottom level has all entries as far left as possible.

Typically complete binary trees are 1-indexed where the indices are assigned left-to-right from the top down. In this way a complete binary tree can be represented by a simple list.

Example 1.1. Here is an example of a complete binary tree.



This may be represented by the 1-indexed list:

$$A = [6, 13, 3, 5, 1, 7, 2, 8, 9, 6, 15, 17]$$

1.2 Index Notes

Observations about indices that we'll find useful:

- If a node has index i then its left and right children (if it has them) have indices $2i$ and $2i + 1$ respectively.

| **Example 1.2.** In the above example the node with index 5 has children with indices 10 and 11.

- If a node has index i then its parent has index $\lfloor i/2 \rfloor$.

| **Example 1.3.** In the above example the node with index 7 has parent with index $\lfloor 7/2 \rfloor = \lfloor 3.5 \rfloor = 3$.

- As a special case of the above, if there are n nodes total then the largest node with children is the node with index $\lfloor n/2 \rfloor$.

| **Example 1.4.** In the above example there are $n = 12$ nodes and the largest one with children is the node with index $\lfloor 12/2 \rfloor = 6$.

- If a node with index i has children than all nodes with smaller indices also have children.

Example 1.5. In the above example the node with index $i = 4$ has children and then so do the nodes within indices 1, 2, and 3.

- Combining the previous two items tell us that if there are n nodes total then the nodes with indices 1, 2, ..., $\lfloor n/2 \rfloor$ are the ones that have children.

Example 1.6. In the above example the nodes with indices 1, 2, ..., 6 are the ones that have children.

1.3 Level Notes

- The leftmost node at level k (with level $k = 0$ being the level of the root) is the node with index 2^k .

Example 1.7. In the above example the leftmost node at level 3 is the node with index $2^{3-1} = 4$.

- A node with index i is located in level $\lfloor \lg i \rfloor$.

Example 1.8. In the above example the node with index 6 is located in level $\lfloor \lg(6) \rfloor = \lfloor 2.58 \rfloor = 2$.

- As a special case of the above, if there are n nodes total then the maximum level (the leaf level) equals $\lfloor \lg n \rfloor$.

Example 1.9. In the above example there are $n = 12$ nodes and $\lfloor \lg(12) \rfloor = \lfloor 3.58 \rfloor = 5$ levels.

- The number of levels between a node with index i and the leaf layer, inclusive, is then $\lfloor \lg n \rfloor - \lfloor \lg i \rfloor + 1$.

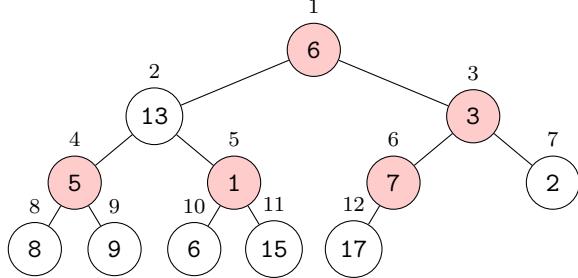
Example 1.10. In the above example the number of levels between the node with index 3 and the leaf level, inclusive, is $\lfloor \lg 12 \rfloor - \lfloor \lg 3 \rfloor + 1 = 3$.

2 Max Heaps

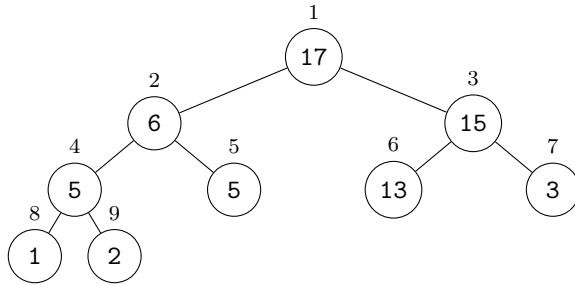
2.1 Definition

Definition 2.1.1. A *max heap* (we'll omit the word binary since all our trees will be binary) is a complete binary tree in which each node's key is greater than or equal to that node's children's key if that node has children. In other words keys non-strictly decrease (equality is acceptable) as we go down the branches.

Example 2.1. The example above is not a max heap. The nodes marked in red below violate the requirement because they have keys which are less than at least one of their children's keys:



Example 2.2. The following is a max heap, however:



3 Converting to a Max Heap

Given a complete binary tree, it's possible to rearrange the nodes so as to obtain a max heap. To do this we'll need two processes.

3.1 Max Heapify - Fixing a Node

The `maxheapify` function atrociously named because its functionality isn't reflected well in its name. A better name would be `swapkeydown`.

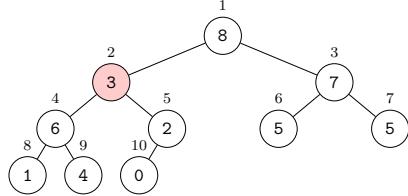
We call the `maxheapify` function on a node whose children are each already roots of their own max heaps. It swaps the key at index i down the tree as far as necessary (if at all) to ensure that the subtree rooted at index i is also a max heap.

It does this by asking "Is my key smaller than either of my children's keys?" If not, then we're done. If so, then the key is swapped with the largest child key and then `maxheapify` calls itself again on that child.

Note 3.1.1. We really want to emphasize that it is `maxiheapify` which calls itself again, recursively. We do not need to make these recursive call manually; We only manually make the first call!

Example 3.1. For example consider the red node (node with index 2) in the following tree. Note that the subtrees rooted at its children are max

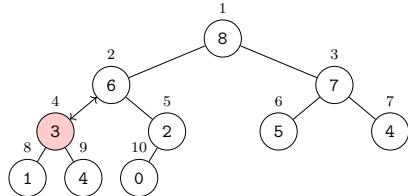
heaps:



We can float this problematic key down by repeatedly following the branch to the largest key. Here is the process.

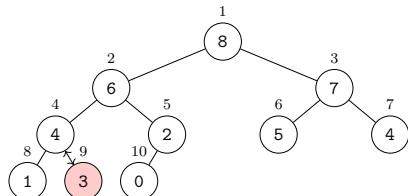
We call `maxheapify(A, 2)`.

We observe that the key with index 2 is smaller than the key at index 4 and so we interchange the keys with indices 2 and 4:



Our previous call to `maxheapify` then recursively calls `maxheapify(A, 4)`.

We observe that the key with index 4 is smaller than the key at index 9 and so we interchange the keys with indices 4 and 9:



Note 3.1.2. This process stops either when the key is larger than the keys of both its children or we reach the bottom of the tree.

Observe that because the smaller key moves down along a path which results in larger keys floating up, and because we never float a key up above a higher key, not only do the subtrees rooted at the child nodes remain max heaps but the subtree rooted at the node with index i becomes a max heap.

What is the time complexity of this? If i is the index of the node we're fixing then this node is in level $\lfloor \lg i \rfloor$.

In the best case we compare with the children and there's no issue, so this is $\Theta(1)$.

In the worst case we need to check all the way to the bottom level which is

$\lfloor n/2 \rfloor$ and so this is $\lfloor \lg n \rfloor - \lfloor \lg i \rfloor + 1$ levels. In order to get this to depend just on n we note:

$$\lfloor \lg n \rfloor - \lfloor \lg i \rfloor + 1 \leq 1 + \lg n$$

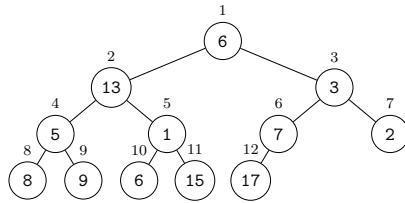
Thus in the worst case this is $\mathcal{O}(\lg n)$.

3.2 Convert to Max Heap - Fixing a Tree

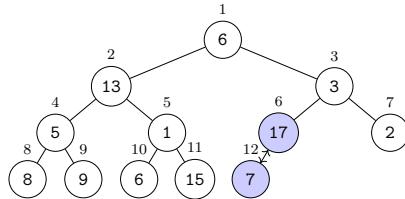
Given a complete binary tree we can convert it to a max heap by running `maxheapify` on all the nodes that have children. From our index arguments before we know this is $1, 2, \dots, \lfloor n/2 \rfloor$. We go through these in reverse order so that when we fix a node we are assured that the subtree rooted at that node is already fixed.

The process above is performed by the `converttomaxheap` function. Thankfully the name of this function is exactly what it does!

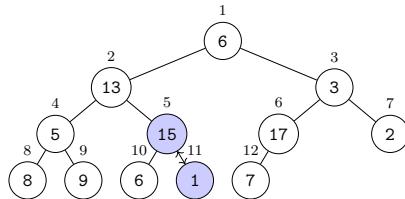
Example 3.2. Here is the process as applied to our original tree:



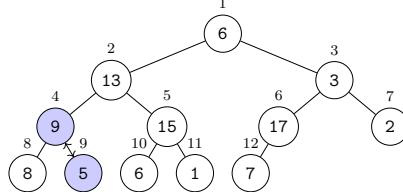
Here we have `maxindex(A)=12` and so `floor(maxindex(A)/2)==6` and so we start with the node with index 6 (the last node with children). Running `maxheapify(A, 6)` interchanges keys at indices along the chain $6 \leftrightarrow 12$ only:



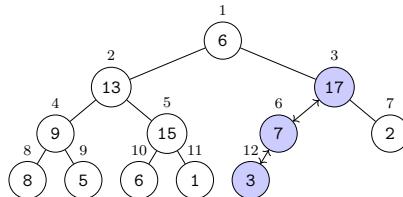
Running `maxheapify(A, 5)` interchanges keys at indices along the chain $5 \leftrightarrow 11$ only:



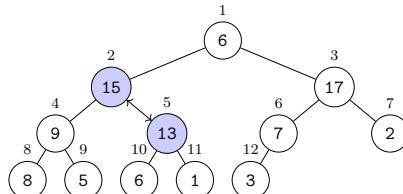
Running `maxheapify(A, 4)` interchanges keys at indices along the chain $4 \leftrightarrow 9$ only:



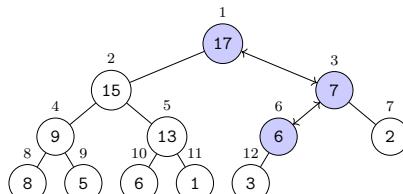
Running `maxheapify(A, 3)` interchanges keys at indices along the chain $3 \leftrightarrow 6 \leftrightarrow 12$ only:



Running `maxheapify(A, 2)` interchanges keys at indices along the chain $2 \leftrightarrow 5$ only:



Running `maxheapify(A, 1)` interchanges keys at indices along the chain $1 \leftrightarrow 3 \leftrightarrow 6$ only:



We can see that the result is now a max heap. The formal proof of this follows from the fact that running `maxheapify` on any particular node preserves the max-heap property of the two child subtrees and induces the max-heap property on the full subtree.

What is the time complexity of this? Consider we're running the process on $\lfloor n/2 \rfloor$ nodes.

In the best case we are running a $\Theta(1)$ process $\lfloor n/2 \rfloor$ times. Since we know that $n/2 - 1 \leq \lfloor n/2 \rfloor \leq n/2$ we know this is $\Theta(n)$.

In the worst case we are running a $\mathcal{O}(\lg n)$ process $\lfloor n/2 \rfloor$ times. Since we know that $\lfloor n/2 \rfloor \leq n/2$ we know this is $\mathcal{O}(n \lg n)$.

4 Heapsort

4.1 Algorithm

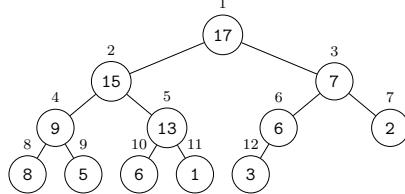
A max binary heap is structured such that extracting the keys in a sorted manner is very easy. There are several ways to do this, all are based on the observation that the largest key is at the root node so that key needs to be last in our sorted list. What we'll do is exchange it with the key in the final node in the tree and then ignore it from here on out, cutting it off from the tree structure.

Now then, the children of the new root node are still max heaps but the new root node (index 1) will almost certainly violate the max heap property so we fix this by running `maxheapify` again on the node with index 1 to fix the remaining tree back to a max heap.

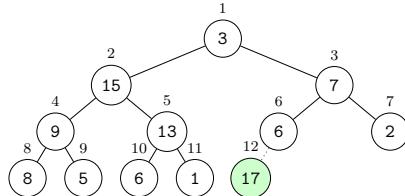
We then repeat the process on the new tree and keep repeating until we're done.

Example 4.1. Here is the process on our heap from earlier:

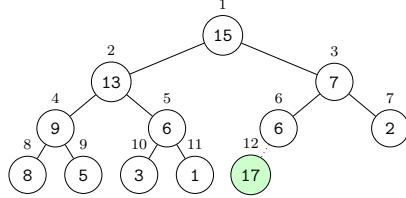
We start with:



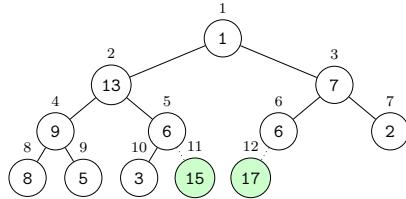
We interchange the keys at the nodes with indices 1 and 12 and cut the node with index 12 off from the tree:



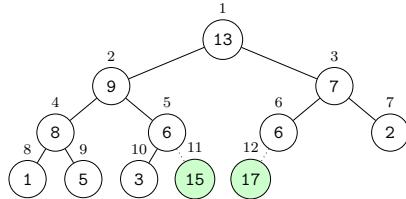
We then run `maxheapify` but only on the subtree:



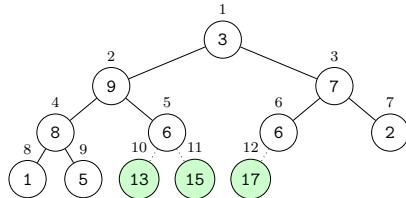
We interchange the keys at the nodes with indices 1 and 11 and cut the node with index 11 off from the tree:



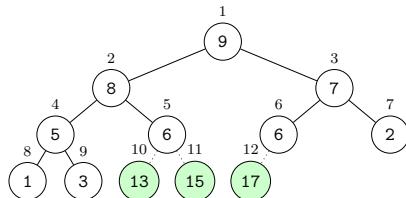
We then run `maxheapify` but only on the subtree:



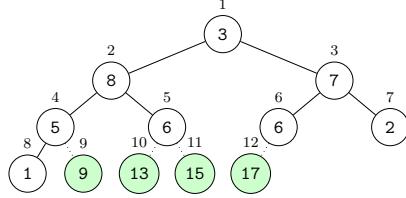
We interchange the keys at the nodes with indices 1 and 10 and cut the node with index 10 off from the tree:



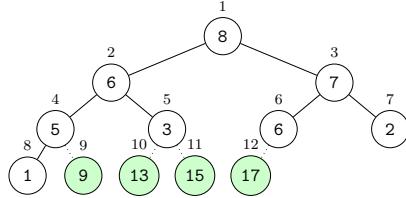
We then run `maxheapify` but only on the subtree:



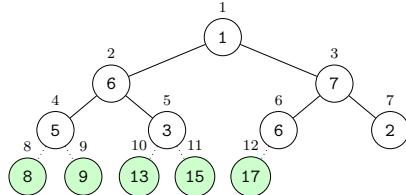
We interchange the keys at the nodes with indices 1 and 9 and cut the node with index 9 off from the tree:



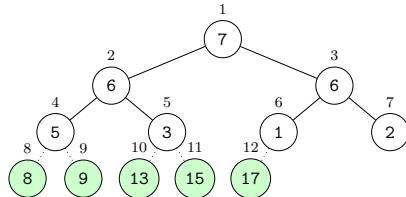
We then run `maxheapify` but only on the subtree:



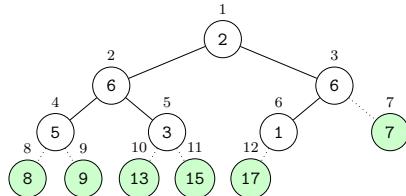
We interchange the keys at the nodes with indices 1 and 8 and cut node with index 8 off from the tree:



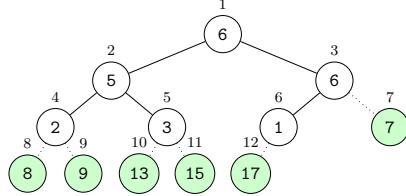
We then run `maxheapify` but only on the subtree:



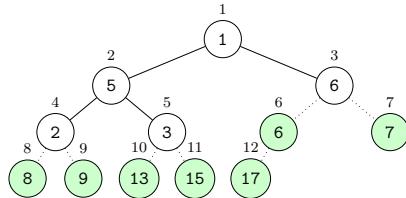
We interchange the keys at the nodes with indices 1 and 7 and cut the node with index 7 off from the tree:



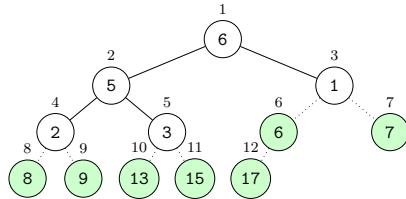
We then run `maxheapify` but only on the subtree:



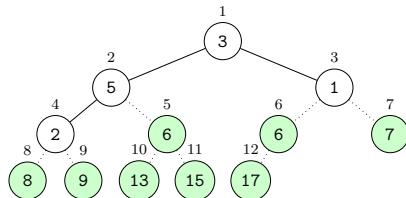
We interchange the keys at the nodes with indices 1 and 6 and cut node with index 6 off from the tree:



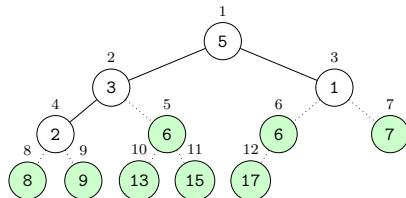
We then run `maxheapify` but only on the subtree:



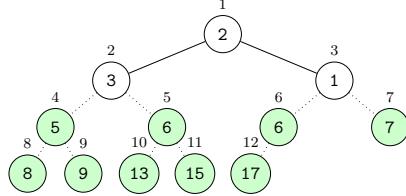
We interchange the keys at the nodes with indices 1 and 5 and cut the node with index 5 off from the tree:



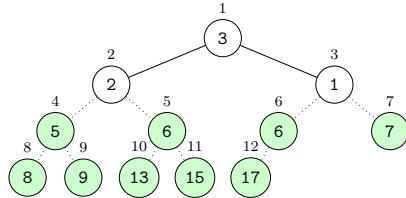
We then run `maxheapify` but only on the subtree:



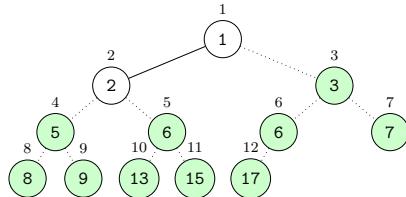
We interchange the keys at the nodes with indices 1 and 4 and cut the node with index 4 off from the tree:



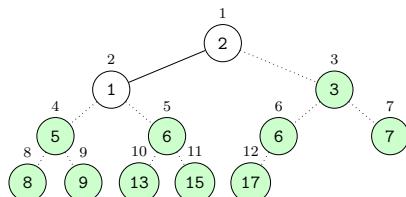
We then run `maxheapify` but only on the subtree:



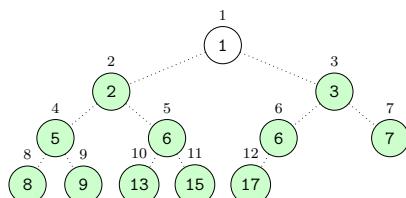
We interchange the keys at the nodes with indices 1 and 3 and cut the node with index 3 off from the tree:



We then run `maxheapify` but only on the subtree:



We interchange the keys at the nodes with indices 1 and 2 and cut the node with index 2 off from the tree:



At this point we're done and we simply extract the keys by the indices of the nodes:

1, 2, 3, 5, 6, 6, 7, 8, 9, 13, 15, 17

4.2 Heapsort Worst-Case Time Complexity

Consider that in the worst case we go through the following process:

- (a) We convert to maxheap. We've seen that this is worst-case $\mathcal{O}(n \lg n)$.
- (b) For $i = n, n-1, \dots, 2$ we swap node i with node 1 (this is $\Theta(1)$), we cut node i off the tree (this is $\Theta(1)$), then we run `maxheapify` on node 1. Running `maxheapify` is worse-case $\mathcal{O}(\lg n)$ but this is for a tree with n nodes. At this point our tree has $i-1$ nodes so it's $\mathcal{O}(\lg(i-1))$.

The total time required is then:

$$\begin{aligned}
\mathcal{O}(n \lg n) + \sum_{i=2}^n [\Theta(1) + \Theta(1) + \mathcal{O}(\lg(i-1))] &\leq \mathcal{O}(n \lg n) + \sum_{i=2}^n \mathcal{O}(\lg(i-1)) \\
&\leq \mathcal{O}(n \lg n) + \sum_{i=2}^n \mathcal{O}(\lg n) \\
&\leq \mathcal{O}(n \lg n) + (n-1)\mathcal{O}(\lg n)
\end{aligned}$$

Thus the time complexity is $\mathcal{O}(n \lg n)$.

4.3 Heapsort Best-Case Time Complexity

A few notes related to best-case time complexity:

1. If we start with a heap of distinct elements which is already a max heap then `converttamaxheap` will be $\Theta(n)$ (scan but no swaps). However when we start the swap-cut-maxheapify process we will be swapping smaller nodes with the root node and then `maxheapify` on the root node will be $\mathcal{O}(\lg(i - 1))$ again each time for a total of:

$$\mathcal{O}(n) + \sum_{i=2}^n [\Theta(1) + \Theta(1) + \mathcal{O}(\lg(i - 1))] = \mathcal{O}(n \lg n)$$

2. If we start with a heap of identical elements then `converttamaxheap` will be $\Theta(n)$ (scan but no swaps). In addition the swap-cut-maxheapify process will be swapping identical nodes with the root note and then `maxheapify` on the root node will be $\Theta(1)$ each time for a total of:

$$\mathcal{O}(n) + \sum_{i=2}^n [\Theta(1) + \Theta(1) + \Theta(1)] = \mathcal{O}(n)$$

4.4 Heapsort Auxiliary Space

HeapSort uses $\mathcal{O}(1)$ auxiliary space.

4.5 Heapsort Stability

HeapSort is unstable.

4.6 Heapsort In-Place

HeapSort is in-place.

4.7 Heapsort Usage Note

HeapSort itself is rarely used as a general sorting algorithm because something like QuickSort is better. However max heaps are used frequently for such things as priority queues and scheduling. The reason for this is that the process of insertion and deletion is $\Theta(\lg n)$ on a max heap versus $\Theta(n)$ on a list and so max heaps are useful whenever these processes are critical.

5 Pseudocode for Everything

Here is the pseudocode for the various functions.

5.1 Pseudocode for Maxheapify

Here it is assumed that A is a 1-indexed list representing a heap and n is the length of A , meaning the number of nodes/keys in the tree. The conditionals `leftnode <= n` and `rightnode <= n` simply check for the existence of children of node i before checking the keys residing there.

Here is the pseudocode:

```
// Maxheapify node i on the tree A with n nodes.
function maxheapify(A,i,n)
    leftnode = 2*i
    rightnode = 2*i+1
    largestnode = i
    if leftnode <= n and A[leftnode] > A[largestnode]
        largestnode = leftnode
    end
    if rightnode <= n and A[rightnode] > A[largestnode]
        largestnode = rightnode
    end
    if largestnode != i
        swap(A[i],A[largestnode])
        maxheapify(A,largestnode,n)
    end
end
```

5.2 Pseudocode for Converttomaxheap

Here is the pseudocode.

```
// Run maxheapify on a tree represented
// by the 1-indexed list A with n nodes.
function converttomaxheap(A,n)
    for i = floor(n/2) down to 1
        maxheapify(A,i,n)
    end
end
```

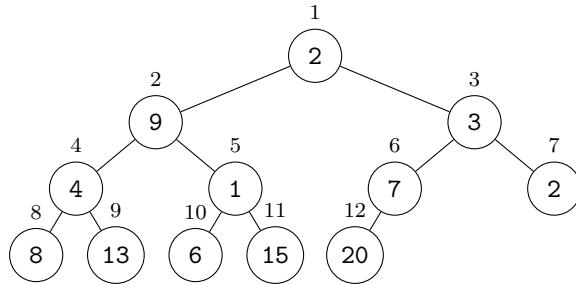
5.3 Pseudocode for Heapsort

Here is the pseudocode:

```
function heapsort(A,n)
    converttomaxheap(A,n)
    for i = n down to 2
        swap(A[1],A[i])
        maxheapify(A,1,i-1)
    end
end
```

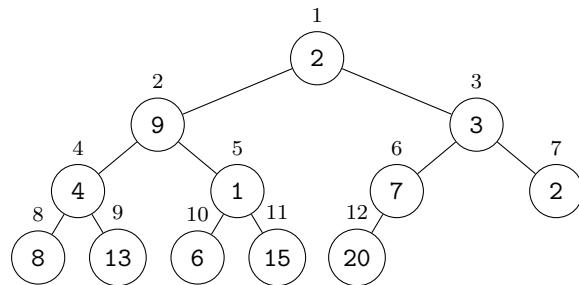
6 Thoughts, Problems, Ideas

1. Consider the following complete binary tree.



- (a) Which nodes violate the max heap property?
(b) Show the results of applying `converttomaxheap`. You do not need to show each step of each `maxheapify` but show the tree after each iteration of `maxheapify` executes.
2. Comparison of running times:
 - If $A = \{1, 2, 3, 4, 5, 6, 7\}$ is treated as a complete binary tree. If `maxheapify` takes 1 second to interchange the keys at two nodes how long will it take to run `heapsort`? Assume everything else takes zero time.
 - If $A = \{7, 6, 5, 4, 3, 2, 1\}$ is treated as an array and if it takes 1 second to swap two entries how long will standard BubbleSort take to sort the array? Assume everything else takes zero time.
3. Prove that $\lfloor \lfloor x/2 \rfloor /2 \rfloor = \lfloor x/4 \rfloor$.
4. The standard way to add an element to a max heap is to add it at the end (the $n + 1$ position) and then run `maxheapify` on all the required nodes. As a function of n , which nodes is this? What is the time complexity of this process?

5. Suppose node i is removed from a max heap. We can't just remove it because we will no longer have a tree. Instead the standard approach is to swap it with the ending node, delete the ending node, and then run `maxheapify` to clean up node i . On which nodes will this be necessary and under which conditions? What is the time complexity of this process?
6. Qualitatively speaking why might InsertSort be faster than HeapSort for smaller lists?
7. Consider the following complete binary tree:



Suppose you forget to `converttomaxheap` in your `heapsort` function. What will the result be? Would you consider the result sorted, unsorted, or something in between?

8. Given an array A indexed at 1, describe a process by which we could determine whether or not the array represents a max heap. Write the pseudocode for an algorithm which does this. What is the time complexity of this process?
9. Describe how you could find the k^{th} largest element in a max heap. Write the pseudocode for an algorithm which does this. What is the time complexity of this process?
10. Modify the various algorithms for the min-heap case.
11. Modify the various algorithms assuming the heap is indexed starting at 0 rather than 1.
12. Provide a formal mathematical proof of the following:

Suppose T is a complete binary tree with the property that the subtrees of the root node are themselves max heaps. Prove that running `maxheapify` on the root node results in a max heap overall.

7 Python Test

Code:

```
# In order to work with the Python array as tree nodes starting at 1,
# We create a list A[0,...,n] and ignore the 0th entry.
import random
import math
A = []
for i in range(0,10):
    A.append(random.randint(0,100))
heapsize = len(A)-1;
nodecount = len(A)-1
def maxheapify(i):
    leftnode = 2*i
    rightnode = 2*i+1
    largestnode = i
    if leftnode <= heapsize and A[leftnode] > A[largestnode]:
        largestnode = leftnode
    if rightnode <= heapsize and A[rightnode] > A[largestnode]:
        largestnode = rightnode
    if largestnode != i:
        temp = A[i]
        A[i] = A[largestnode]
        A[largestnode] = temp
        maxheapify(largestnode)
def converttomaxheap():
    for i in range(math.floor(heapsize/2),0,-1):
        maxheapify(i)
def heapsort():
    global heapsize
    converttomaxheap()
    print('After converttomaxheap:')
    print(A[1:])
    for i in range(nodecount,1,-1):
        temp = A[1]
        A[1] = A[i]
        A[i] = temp
        print('After switch:')
        print(A[1:])
        heapsize = heapsize - 1
        maxheapify(1)
        print('After maxheapify:')
        print(A[1:])
    print(A[1:])
heapsort()
```

```
print(A[1:])
```

Output:

```
[74, 60, 82, 5, 8, 7, 90, 31, 50]
After converttamaxheap:
[90, 60, 82, 50, 8, 7, 74, 31, 5]
After switch:
[5, 60, 82, 50, 8, 7, 74, 31, 90]
After maxheapify:
[82, 60, 74, 50, 8, 7, 5, 31, 90]
After switch:
[31, 60, 74, 50, 8, 7, 5, 82, 90]
After maxheapify:
[74, 60, 31, 50, 8, 7, 5, 82, 90]
After switch:
[5, 60, 31, 50, 8, 7, 74, 82, 90]
After maxheapify:
[60, 50, 31, 5, 8, 7, 74, 82, 90]
After switch:
[7, 50, 31, 5, 8, 60, 74, 82, 90]
After maxheapify:
[50, 8, 31, 5, 7, 60, 74, 82, 90]
After switch:
[7, 8, 31, 5, 50, 60, 74, 82, 90]
After maxheapify:
[31, 8, 7, 5, 50, 60, 74, 82, 90]
After switch:
[5, 8, 7, 31, 50, 60, 74, 82, 90]
After maxheapify:
[8, 5, 7, 31, 50, 60, 74, 82, 90]
After switch:
[7, 5, 8, 31, 50, 60, 74, 82, 90]
After maxheapify:
[7, 5, 8, 31, 50, 60, 74, 82, 90]
After switch:
[5, 7, 8, 31, 50, 60, 74, 82, 90]
After maxheapify:
[5, 7, 8, 31, 50, 60, 74, 82, 90]
[5, 7, 8, 31, 50, 60, 74, 82, 90]
```

CMSC 351: QuickSort

Justin Wyss-Gallifent

March 6, 2024

1	What it Does	2
2	Overview	2
3	Partitioning Overview	3
4	Pseudocode	4
5	Pivot Key Choice	9
6	Pseudocode Time Complexity	9
7	Auxiliary Space	11
8	Stability	11
9	In-Place	11
10	Notes	11
11	Thoughts, Problems, Ideas	12
12	Python Test - Pivot on Final Element	14
13	Python Test - Pivot on Random Element	16

1 What it Does

Sorts a list of elements on which there is a total order. Think of integers or real numbers.

2 Overview

QuickSort works by first choosing an element in the list called the pivot key (at some initial pivot index) and then rearranging the list (including probably moving the the pivot key) so that every element smaller than pivot key is to the left of it and every element larger than the pivot key is to the right of it. This is called the partitioning process.

We then apply QuickSort recursively to the left and right sublists.

When QuickSort is applied to a single element it does nothing, since a single element is always sorted.

The choice of pivot key is nuanced. For now we will consistently choose the final key in the list. If another key is chosen it is simply first exchanged with the final key in the list before proceeding.

3 Partitioning Overview

The encoding of the partitioning process can seem a bit convoluted so it's worth summarizing what is effectively happening in the following way which clarifies that it does what we claim it does.

Pick the leftmost element which is greater than the pivot key and swap it with the first subsequent element which is less than or equal to the pivot key. Repeating until there are no subsequent elements left. The final swap will be with the actual pivot key and the result will be that all elements to the left of the pivot key will be less than or equal to the pivot key and all elements to the right of the pivot key will be greater than the pivot key.

Example 3.1. Consider the list A with the pivot key $p = 3$.

index	0	1	2	3	4
A	5	2	4	1	3

The leftmost element greater than $p = 3$ is $A[0] = 5$. The first subsequent element less than or equal to $p = 3$ is $A[1] = 2$ so we swap those two:

index	0	1	2	3	4
A	2	5	4	1	3

The leftmost element greater than $p = 3$ is $A[1] = 5$. The first subsequent element less than or equal to $p = 3$ is $A[3] = 1$ so we swap those two:

index	0	1	2	3	4
A	2	1	4	5	3

The leftmost element greater than $p = 3$ is $A[2] = 4$. The first subsequent element less than or equal to $p = 3$ is $A[4] = 3$ so we swap those two:

index	0	1	2	3	4
A	2	1	3	5	4

The leftmost element greater than $p = 3$ is $A[3] = 5$. There are no subsequent elements less than or equal to $p = 3$. Thus we are done.

Observe that the pivot keys is such that all elements to the left are less than or equal to it and all elements to the right are greater than it.

4 Pseudocode

The actual algorithmic implementation is a bit more nuanced and has a small quirk.

```
\\" PRE: A is a list of length n.  
\\" Note that A is considered global here.  
function quicksort(A,L,R)  
    if L < R then  
        resultingpivotindex = partition(A,L,R)  
        quicksort(A,L,resultingpivotindex-1)  
        quicksort(A,resultingpivotindex+1,R)  
    end  
end  
function partition(A,L,R)  
    \\" To use a different pivotkey  
    \\" swap it with A[R] here.  
    pivotkey = A[R]  
    t = L  
    for i = L to R-1  
        if A[i] <= pivotkey  
            A[t] <-> A[i]  
            t = t + 1  
        end  
    end  
    A[t] <-> A[R]  
    return t  
end  
quicksort(A,0,n-1)  
\\" POST: A is sorted.
```

Loosely speaking t keeps track of the leftmost key larger than the pivot key and i hunts down the subsequent key less than or equal to the pivot key. We say “loosely” because if the list starts with keys less than or equal to the pivot key then the algorithm will swap them with themselves for a while.

Example 4.1. Let's trace the implementation on the following list with $n = 7$ elements. Note that the initial call to `quicksort` is the call `quicksort(A, 0, 7-1)` which then calls `partition(A, 0, 6)` which runs `for i = 0 to 5`. We have $L=0$ and $R=6$.

index	0	1	2	3	4	5	6
A	2	1	6	1	8	4	5
							p=5

The pivot key is $p=5$ at index 6.

We set $t=0$ and iterate from $i=0$:

index	0	1	2	3	4	5	6
A	2	1	6	1	8	4	5
							p=5
	t, i						

We see $A[i]=A[0]=2 \leq 5$ is true and swap $A[i] \leftrightarrow A[t]$ which effectively swaps $A[0] \leftrightarrow A[0]$. We increase t so now $t=1$. We now have $i=1$:

index	0	1	2	3	4	5	6
A	2	1	6	1	8	4	5
							p=5
	t, i						

We see $A[i]=A[1]=1 \leq 5$ is true and swap $A[i] \leftrightarrow A[t]$ which effectively swaps $A[1] \leftrightarrow A[1]$. We increase t so now $t=2$. We now have $i=2$:

index	0	1	2	3	4	5	6
A	2	1	6	1	8	4	5
							p=5
	t, i						

We see $A[i]=A[2]=6 \leq 5$ is false and we do nothing. We now have $i=3$:

index	0	1	2	3	4	5	6
A	2	1	6	1	8	4	5
							p=5
	t, i						

Notice that finally t indicates the location of the leftmost element greater than the pivot key. This is when things actually start to happen.

We see $A[i]=A[3]=1 \leq 5$ is true and swap $A[i] \leftrightarrow A[t]$ which effectively swaps $A[3] \leftrightarrow A[2]$. We increase t so now $t=3$. We now have $i=4$:

index	0	1	2	3	4	5	6
A	2	1	1	6	8	4	5
							p=5
	t, i						

We see $A[i] = A[4] = 6 \leq 5$ is false and we do nothing. We now have $i=5$:

index	0	1	2	3	4	5	6
A	2	1	1	6	8	4	5
	t			i	p=5		

We see $A[i] = A[5] = 4 \leq 5$ is true and swap $A[i] \leftrightarrow A[t]$ which effectively swaps $A[5] \leftrightarrow A[3]$. We increase t so now $t=4$. There is no i since the loop has ended:

index	0	1	2	3	4	5	6
A	2	1	1	4	8	6	5
	t				p=5		

The loop has ended and we do a final $A[t] \leftrightarrow A[R]$ which effectively swaps $A[4] \leftrightarrow A[6]$, putting the pivot key in location t so that finally we are done this partition process:

index	0	1	2	3	4	5	6
A	2	1	1	4	5	6	8
	t						

Now we are done.

Example 4.2. Let's trace the implementation on the following list with $n = 12$ elements. Note that the initial call to `quicksort` is the call `quicksort(A, 0, 12-1)` which then calls `partition(A, 0, 11)` which runs `for i = 0 to 10`. We have $L=0$ and $R=11$.

For example consider this list:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	10	4	3	1	7	4	3	5	6	2	11	5
												p=5

Set $t=0$ and $i=0$:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	10	4	3	1	7	4	3	5	6	2	11	5
	t, i											p=5

No swap, iterate $i=1$:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	10	4	3	1	7	4	3	5	6	2	11	5
	t	i										p=5

Swap, iterate $t=1$ and iterate $i=2$:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	4	10	3	1	7	4	3	5	6	2	11	5
	t	i										p=5

Swap, iterate $t=2$ and iterate $i=3$:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	4	3	10	1	7	4	3	5	6	2	11	5
	t	i										p=5

Swap, iterate $t=3$ and iterate $i=4$:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	4	3	1	10	7	4	3	5	6	2	11	5
	t	i										p=5

No swap, iterate $i=5$:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	4	3	1	10	7	4	3	5	6	2	11	5
	t	i										p=5

Swap, iterate $t=4$ and iterate $i=6$:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	4	3	1	4	7	10	3	5	6	2	11	5
	t	i										p=5

Swap, iterate $t=5$ and iterate $i=7$:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	4	3	1	4	3	10	7	5	6	2	11	5
	t	i										p=5

Swap, iterate $t=6$ and iterate $i=8$:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	4	3	1	4	3	5	7	10	6	2	11	5
	t	i										p=5

No swap, iterate $i=9$:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	4	3	1	4	3	5	7	10	6	2	11	5
	t		i		p=5							

Swap, iterate t=7 and iterate i=10:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	4	3	1	4	3	5	2	10	6	7	11	5
	t		i		p=5							

Loop done. Swap A[t]<->A[R]:

index	0	1	2	3	4	5	6	7	8	9	10	11
A	4	3	1	4	3	5	2	5	6	7	11	10

5 Pivot Key Choice

For simplicity we choose the last key of the sublist as the pivot but this is perhaps less than idea. For example if the list is already sorted then choosing the last element as the pivot key results in the algorithm taking as long as possible and not actually changing anything - try it on some data and see!

Intuitively if the input is close to sorted then choosing the last element (or in fact the first element) as the pivot will result in very slow running time.

In order to prevent this there are other ways we can choose the initial pivot key. One way is simply randomly. This adds very little time to the process since typically random number generation is $\Theta(1)$. Whether it helps or not is something we'll have to see.

It might be tempting to choose the initial pivot index so that the pivot key is the median, since that results in a nice balanced partition, but this is in and of itself challenging as we'll see later.

6 Pseudocode Time Complexity

Let $T(n)$ be the time complexity of a call to QuickSort.

A call to QuickSort on a list of length n invokes both a partitioning call and two subsequent recursive calls to QuickSort. If the `resultingpivotindex` returns index k then one of them to a sublist of length k and one of them to a sublist of length $n - k - 1$

Those recursive calls take time $T(k)$ and $T(n - k - 1)$ respectively.

The partition call on a list of length n does some constant time c_1 work and also iterates over $n - 1$ elements, say it takes time c_2 for each iteration, thus in total it takes $c_1 + c_2(n - 1)$.

Hence we have the recurrence relation:

$$T(n) = T(k) + T(n - k - 1) + c_2n + (c_1 - c_2)$$

1. **Worst Case:** The worst-case occurs when the `resultingpivotindex` is the first or last element in the sublist.

This results in the sublist being only one element smaller than the list itself and the other sublist being length zero. Without loss of generality if $k = 0$ in the above relation we have

$$T(n) = T(n - 1) + c_2n + (c_1 - c_2)$$

We cannot solve this with the Master Theorem but observing that $T(1) = c_3$ for some constant c_3 we can derive a pattern:

$$\begin{aligned}
 T(1) &= c_3 \\
 T(2) &= T(1) + c_2(2) + (c_1 - c_2) = c_1 + c_2 + c_3 \\
 T(3) &= T(2) + c_2(3) + (c_1 - c_2) = (c_1 + c_2 + c_3) + 3c_2 + (c_1 - c_2) = 2c_1 + 3c_2 + c_3 \\
 T(4) &= T(3) + c_2(4) + (c_1 - c_2) = (2c_1 + 3c_2 + c_3) + 4c_2 + (c_1 - c_2) = 3c_1 + 6c_2 + c_3 \\
 T(5) &= T(4) + c_2(5) + (c_1 - c_2) = (3c_1 + 6c_2 + c_3) + 5c_2 + (c_1 - c_2) = 4c_1 + 10c_2 + c_3 \\
 T(6) &= T(5) + c_2(6) + (c_1 - c_2) = (4c_1 + 10c_2 + c_3) + 6c_2 + (c_1 - c_2) = 5c_1 + 15c_2 + c_3 \\
 &\vdots \quad = \quad \vdots \\
 T(n) &= (n-1)c_1 + \frac{n(n+1)}{2}c_2 + c_3
 \end{aligned}$$

This results in $T(n) = \Theta(n^2)$.

2. **Best-Case:** The best-case occurs when the `resultingpivotindex` is in the middle of the sublist.

This results in the sublists being of equal size. then in the above relation we have

$$T(n) = 2T(n/2) + c_2n + (c_1 - c_2)$$

which results in $T(n) = \Theta(n \lg n)$ by the Master Theorem.

3. **Average-Case:**

As per earlier discussions we need to understand what “average” means.

QuickSort is interesting in that the underlying partition method has time complexity $\theta(n)$ which does not depend upon the order of the elements. This is because $n - 1$ comparisons are made in every case, and then some other constant stuff.

As a consequence the only thing that influences the time complexity of QuickSort is the resulting pivot position after each partition which dictates the sizes of the sublists on which the recursive calls are made.

Thus one perfectly reasonable way to understand the average case is to not worry about the lists themselves but instead to look at all possible pivot positions and average the time complexity of all of them. In this case we get the following recurrence relation:

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \frac{1}{n} \sum_{k=0}^{n-1} [T(k) + T(n - k - 1) + \Theta(n)] & \text{otherwise} \end{cases}$$

This is not easy to solve but essentially the idea is to show that this is $\mathcal{O}(n \lg n)$. This is done by constructive induction. I am omitting the proof for now.

7 Auxiliary Space

Quicksort uses $\Theta(1)$ auxiliary space.

8 Stability

QuickSort is not stable. This is because the final swap $A[t] \leftrightarrow A[R]$ could place the pivot key to the left of an equal key.

9 In-Place

QuickSort is in-place.

10 Notes

A few notes:

1. Ideally (mathematically) at each step the median of the keys would be used as the pivot element. The exercises ask you to think about why. There is a method we'll see later called the Median of Medians which can find the median in $\mathcal{O}(n)$ time. However it's still slow, relatively speaking, and thus...
2. In practice choosing an element randomly is the usual approach, even though this the actual implementation is slightly slower. The exercises ask you to think about why.
3. After k iterations we can draw some conclusions about which elements are correctly placed. The exercises discuss this further.

11 Thoughts, Problems, Ideas

1. Show the steps of the first partition of QuickSort on the list [10, 6, 7, 2, 4, 3]. Use the final index as the initial pivot index.
2. Show the full steps of QuickSort on the list [10, 6, 7, 2, 4, 3]. Use the final index as the initial pivot index.
3. Consider QuickSort used on the list [11, 6, 5, 45, 23, 7, 2]. Which initial pivot index and pivot key should be used as the first pivot to ensure that the result of the first partition is equally balanced? Show the result of doing this first **partition**.
4. Suppose the **resultingpivotindex** somehow ended up consistently one third of the way through the list. What would be the correponding recurrence relation? Can the Master Theorem be used to solve this? Can you say anything about the time complexity?
5. Consider the theoretical case where the **resultingpivotindex** consistently ends up 1/2 of the way through the list and the theoretical case where the **resultingpivotindex** consistently ends up 1/4 of the way through the list. Essentially the corresponding recurrence relations would be something like:

$$T_{1/2}(n) = T(\lfloor n/2 \rfloor) + T(\lfloor n/2 \rfloor) + \Theta(n)$$
$$T_{1/4}(n) = T(\lfloor n/4 \rfloor) + T(\lfloor 3n/4 \rfloor) + \Theta(n)$$

Suppose in addition that:

- The $\Theta(n)$ term is actually $5n + 2$.
 - You know that $T(0) = 0$ and $T(1) = 2$.
- (a) Find each time complexity for keys $n = 0, 10, 20, \dots, 100$. (You are welcome to do this in recursive code and if you do, include your code. It's not hard - each is six simple lines of Python, for example.)
 - (b) Plot the corresponding data and connect with smooth lines.
 - (c) Which of these does your data suggest has a better Θ time complexity? Explain.
6. Give a specific example which illustrates the fact that QuickSort is not stable. Illustrate where in the process the stability breaks down. You don't need to show the entire implementation, just enough to justify.

7. Consider these two approaches to pivot key selection:
- Obtain index of the median, use as initial pivot index.
 - Choose random index, use as initial pivot index.
- In practice both of these are average case $\Theta(n \lg n)$.
- (a) Mathematically, using the median is better. Why?
 - (b) In practice, using a random element is the standard approach. Why not the median?
8. Suppose a list has $n = 2^k - 1$ elements for some k and suppose that somehow, magically, the index of the median is chosen at every stage for the initial pivot index. In this case we can explicitly calculate the number of calls to `quicksort`, the length of each list it is called on, and the number of subsequent calls to `partition`. Remember that `quicksort` only calls `partition` in the case $l < r$ so when $l == r$ (list of length 1) `quicksort` exits.
- Consider the case where $n = 2^4 - 1 = 15$. Observe there will be:
- 1 initial call to `quicksort` with a list of length 15, resulting in 1 call to `partition` and then ...
 - a total of 2 calls to `quicksort` with lists of length 7, resulting in a total of 2 calls to `partition` and then ...
 - a total of 4 calls to `quicksort` with lists of length ???, resulting in a total of ??? calls to `partition` and then ...
 - and so on, until it ends.
- (a) Complete the counting argument above - finish the third bullet point and then the remaining bullet points until the argument ends. There should be only four bullet points total.
 - (b) What would the counting argument be for $n = 2^k - 1$ for an arbitrary k ? It's not hard, just generalize the pattern in (a).
 - (c) Suppose that each call to `partition` on a list of length i takes time $c_1 i$. Ignoring all other time requirements (the call to `partition` is the important one) write down and evaluate the sum which gives the total time requirement of the algorithm. Does the time complexity of this result correspond to the best-case analysis?
9. Modify the QuickSort pseudocode so that it chooses the first element as the pivot key.
 10. Modify the QuickSort pseudocode so that it randomly chooses a pivot.
 11. Modify the QuickSort pseudocode so that it sorts the list in decreasing order.

12 Python Test - Pivot on Final Element

Code:

```
import random
A = []
for i in range(0,15):
    A.append(random.randint(0,100))

A = [5,8,3,4,10,7]

n = len(A)
print(A)
def quicksort(l,r,indent):
    if l<r:
        resultingpivotindex = partition(l,r,indent+2)
        quicksort(l,resultingpivotindex-1,indent+2)
        quicksort(resultingpivotindex+1,r,indent+2)
        print(indent*'_' + 'Recombine: '+str(A[l:r+1]))

def partition(l,r,indent):
    print(indent*'_' + 'Subarray: ' + str(A[l:r+1]))
    # To use a different pivotvalue
    # swap it with A[r] here.
    pivotvalue = A[r]
    t = l
    for i in range(l,r):
        if A[i] <= pivotvalue:
            temp = A[t]
            A[t] = A[i]
            A[i] = temp
            t = t + 1
            print(A)
    temp = A[t]
    A[t] = A[r]
    A[r] = temp
    print(indent*'_' + 'Pivot around final element.')
    print(indent*'_' + 'Result: ' + str(A[l:r+1]))
    return(t)

quicksort(0,n-1,0)
print(A)
```

Output:

```
[50, 4, 24, 90, 92, 84, 33, 27, 81, 13, 2, 44, 62, 28, 80]
--Subarray: [50, 4, 24, 90, 92, 84, 33, 27, 81, 13, 2, 44, 62, 28, 80]
--Pivot around final element.
--Result: [50, 4, 24, 33, 27, 13, 2, 44, 62, 28, 80, 92, 81, 84, 90]
----Subarray: [50, 4, 24, 33, 27, 13, 2, 44, 62, 28]
----Pivot around final element.
----Result: [4, 24, 27, 13, 2, 28, 50, 44, 62, 33]
-----Subarray: [4, 24, 27, 13, 2]
-----Pivot around final element.
-----Result: [2, 24, 27, 13, 4]
-----Subarray: [24, 27, 13, 4]
-----Pivot around final element.
-----Result: [4, 27, 13, 24]
-----Subarray: [27, 13, 24]
-----Pivot around final element.
-----Result: [13, 24, 27]
-----Recombine: [13, 24, 27]
-----Recombine: [4, 13, 24, 27]
---Recombine: [2, 4, 13, 24, 27]
---Subarray: [50, 44, 62, 33]
---Pivot around final element.
---Result: [33, 44, 62, 50]
---Subarray: [44, 62, 50]
---Pivot around final element.
---Result: [44, 50, 62]
---Recombine: [44, 50, 62]
---Recombine: [33, 44, 50, 62]
--Recombine: [2, 4, 13, 24, 27, 28, 33, 44, 50, 62]
---Subarray: [92, 81, 84, 90]
---Pivot around final element.
---Result: [81, 84, 90, 92]
---Subarray: [81, 84]
---Pivot around final element.
---Result: [81, 84]
---Recombine: [81, 84]
--Recombine: [81, 84, 90, 92]
Recombine: [2, 4, 13, 24, 27, 28, 33, 44, 50, 62, 80, 81, 84, 90, 92]
[2, 4, 13, 24, 27, 28, 33, 44, 50, 62, 80, 81, 84, 90, 92]
```

13 Python Test - Pivot on Random Element

Code:

```
import random
A = []
for i in range(0,13):
    A.append(random.randint(0,100))
n = len(A)
print(A)
def quicksort(l,r,indent):
    if l<r:
        pivotindex = partition(l,r,indent+2)
        quicksort(l,pivotindex-1,indent+2)
        quicksort(pivotindex+1,r,indent+2)
        print(indent*'_' + 'Recombine: ' + str(A[l:r+1]))
def partition(l,r,indent):
    print(indent*'_' + 'Subarray: ' + str(A[l:r+1]))
    p = random.randint(l,r)
    temp = A[p]
    A[p] = A[r]
    A[r] = temp
    pivot = A[r]
    t = l
    for i in range(l,r):
        if A[i] <= pivot:
            temp = A[t]
            A[t] = A[i]
            A[i] = temp
            t = t + 1
    temp = A[t]
    A[t] = A[r]
    A[r] = temp
    print(indent*'_' + 'Pivot around index ' + str(p-1))
    print(indent*'_' + 'Result: ' + str(A[l:r+1]))
    return(t)
quicksort(0,n-1,0)
print(A)
```

Output:

```
[92, 55, 6, 43, 30, 43, 37, 66, 63, 24, 92, 3, 79]
--Subarray: [92, 55, 6, 43, 30, 43, 37, 66, 63, 24, 92, 3, 79]
--Pivot around index 2
--Result: [3, 6, 79, 43, 30, 43, 37, 66, 63, 24, 92, 92, 55]
----Subarray: [79, 43, 30, 43, 37, 66, 63, 24, 92, 92, 55]
----Pivot around index 3
----Result: [43, 30, 37, 24, 43, 66, 63, 55, 92, 92, 79]
-----Subarray: [43, 30, 37, 24]
-----Pivot around index 3
-----Result: [24, 30, 37, 43]
-----Subarray: [30, 37, 43]
-----Pivot around index 2
-----Result: [30, 37, 43]
-----Subarray: [30, 37]
-----Pivot around index 1
-----Result: [30, 37]
-----Recombine: [30, 37]
-----Recombine: [30, 37, 43]
---Recombine: [24, 30, 37, 43]
----Subarray: [66, 63, 55, 92, 92, 79]
----Pivot around index 1
----Result: [55, 63, 66, 92, 92, 79]
----Subarray: [66, 92, 92, 79]
----Pivot around index 0
----Result: [66, 92, 92, 79]
----Subarray: [92, 92, 79]
----Pivot around index 1
----Result: [92, 79, 92]
----Subarray: [92, 79]
----Pivot around index 0
----Result: [79, 92]
----Recombine: [79, 92]
----Recombine: [79, 92, 92]
----Recombine: [66, 79, 92, 92]
---Recombine: [55, 63, 66, 79, 92, 92]
__Recombine: [24, 30, 37, 43, 43, 55, 63, 66, 79, 92, 92]
Recombine: [3, 6, 24, 30, 37, 43, 43, 55, 63, 66, 79, 92, 92]
[3, 6, 24, 30, 37, 43, 43, 55, 63, 66, 79, 92, 92]
```

CMSC 351: Limitations on Comparisons

Justin Wyss-Gallifent

October 16, 2023

1	Introduction to Decision Trees	2
2	Decisions	2
3	Decision Tree for An Algorithm	4
4	Analysis of Comparison-Based Sorting	6
5	Thoughts, Problems, Ideas	8

1 Introduction to Decision Trees

Note that the fastest sorting algorithms we have seen having a worst-case time complexity $T(n) = \mathcal{O}(n \lg n)$. We might ask if there is some sorting algorithm which has a smaller worst-case time complexity.

Before answering this question let's take a second to observe that all the sorting algorithms we've looked at so far (BubbleSort, SelectSort, InsertSort, HeapSort, MergeSort, and QuickSort) are all comparison-based. What this means is that they all work by comparing pairs of numbers repeatedly in various ways. This seems like an obvious necessity because we're trying to sort and sorting is based on some sort of comparison.

Before we check out some non-comparison based sorting algorithms let's take an abstract look at these sorting algorithms. What we mean by that is - let's look at them as a whole, rather than individually.

2 Decisions

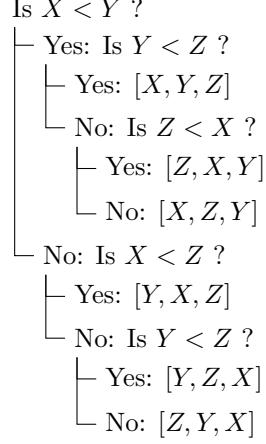
Every comparison-based sort algorithm involves a number of comparisons as it manages the data. Most of those comparisons lead to decisions such as "If $X > Y$ do this, otherwise do that". There always a minimum number of decisions that the algorithm must make. Consider these examples of lists and sorting them.

Example 2.1. Suppose a list is $[X, Y]$ and it needs to be sorted. We'll implement ImmortalSort. Only one decision needs to be made:

```
Is X < Y ?  
  └ Yes: [X, Y]  
  └ No: [Y, X]
```

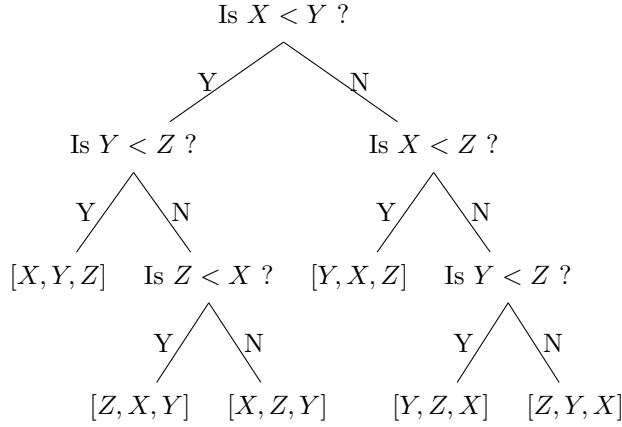
The answer to this question is sufficient to sort the data and this question is absolutely necessary.

Example 2.2. Suppose a list is $[X, Y, Z]$ and it needs to be sorted. We'll implement ImmortalSort. Several decisions need to be made. Let's think them out in a nested manner:



Observe that at different junctions we need to think differently. For example if $X < Y$ and $Y < Z$ then we're done at $[X, Y, Z]$ but if $X < Y$ and $Y \not< Z$ then we could have either $[Z, X, Y]$ or $[X, Z, Y]$ and we need another decision.

Observe that this series of decisions forms a full binary tree (every non-leaf node has exactly two children):



This series of decisions is not the only way we could sort the data. We could have started with a different initial question, for example.

3 Decision Tree for An Algorithm

Each and every comparison-based sorting algorithm must make a series of decisions as it sorts the data and they might be different from method to method.

The decision tree for a comparison-based sort algorithm is then the full binary tree which displays all the decision branches which arise from the algorithm's comparisons.

Consider BubbleSort. This implementation is specifically written so that all comparisons are \leq .

```
for i = 0 to n-1
    for j = 0 to n-i-2
        if A[j] <= A[j+1]
            nothing
        else
            swap A[j] and A[j+1]
        end
    end
end
```

Let's see how this works on a list of length 3. Before proceeding observe that with a list of length 3, BubbleSort will check...

- Is $A[0] < A[1]$? Swap if false.
- Is $A[1] < A[2]$? Swap if false.
- Is $A[0] < A[1]$ again? Swap if false.

This implementation of BubbleSort may make useless comparisons and we should not count those as decisions. We'll note as we go through.

Assume the original list is $A = [X, Y, Z]$ unsorted. There are six possibilities:

- Starting with $A = [X, Y, Z]$, if $X < Y < Z$:

Alg Comparison	List Comparison	Result	A is now
$A[0] < A[1]$	$X < Y$	True	$[X, Y, Z]$
$A[1] < A[2]$	$Y < Z$	True	$[X, Y, Z]$
$A[0] < A[1]$	$X < Y$	Previously True	$[X, Y, Z]$

- Starting with $A = [X, Y, Z]$, if $X < Z < Y$:

Alg Comparison	List Comparison	Result	A is now
$A[0] < A[1]$	$X < Y$	True	$[X, Y, Z]$
$A[1] < A[2]$	$Y < Z$	False so Swap	$[X, Z, Y]$
$A[0] < A[1]$	$X < Z$	True	$[X, Z, Y]$

- Starting with $A = [X, Y, Z]$, if $Z < X < Y$:

Alg Comparison	List Comparison	Result	A is now
$A[0] < A[1]$	$X < Y$	True	$[X, Y, Z]$
$A[1] < A[2]$	$Y < Z$	False so Swap	$[X, Z, Y]$
$A[0] < A[1]$	$X < Z$	False so Swap	$[Z, X, Y]$

- Starting with $A = [X, Y, Z]$, if $Y < X < Z$:

Alg Comparison	List Comparison	Result	A is now
$A[0] < A[1]$	$X < Y$	False so Swap	$[Y, X, Z]$
$A[1] < A[2]$	$X < Z$	True	$[Y, X, Z]$
$A[0] < A[1]$	$Y < X$	Reflexively True	$[Y, X, Z]$

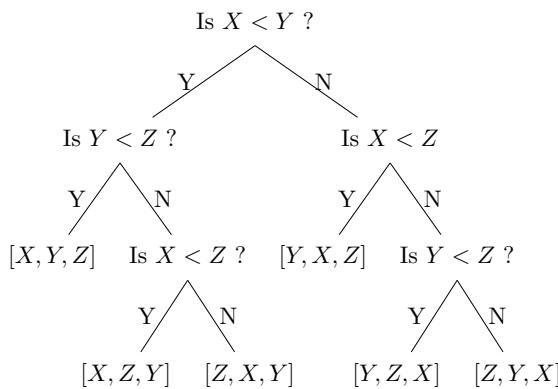
- Starting with $A = [X, Y, Z]$, if $Y < Z < X$:

Alg Comparison	List Comparison	Result	A is now
$A[0] < A[1]$	$X < Y$	False so Swap	$[Y, X, Z]$
$A[1] < A[2]$	$X < Z$	False so Swap	$[Y, Z, X]$
$A[0] < A[1]$	$Y < Z$	True	$[Y, Z, X]$

- Starting with $A = [X, Y, Z]$, if $Z < Y < X$:

Alg Comparison	List Comparison	Result	A is now
$A[0] < A[1]$	$X < Y$	False so Swap	$[Y, X, Z]$
$A[1] < A[2]$	$X < Z$	False so Swap	$[Y, Z, X]$
$A[0] < A[1]$	$Y < Z$	False so Swap	$[Z, Y, X]$

Here is the corresponding tree:



4 Analysis of Comparison-Based Sorting

We have the following interesting theorem. We really only need the Ω part here but the whole thing is interesting:

Theorem 4.0.1. We have $\Omega(\lg(n!)) = \Omega(n \lg n)$ and $\mathcal{O}(\lg(n!)) = \mathcal{O}(n \lg(n))$ and consequently $\Theta(\lg(n!)) = \Theta(n \lg n!)$.

Proof. We do this in steps:

- (a) Suppose that $f(n) = \Omega(\lg(n!))$. We claim that $f(n) = \Omega(n \lg n)$.

Observe that:

$$\begin{aligned}\lg(n!) &= \lg(n) + \lg(n-1) + \dots + \lg(2) + \lg(1) \\ &= [\lg(n-0) + \lg(n-1) + \dots + \lg(n - \lfloor n/2 \rfloor)] + [\text{the rest}] \\ &\geq \lg(n-0) + \lg(n-1) + \dots + \lg(n - \lfloor n/2 \rfloor) \\ &\geq \underbrace{\lg(n/2) + \lg(n/2) + \dots + \lg(n/2)}_{1 + \lfloor n/2 \rfloor \text{ terms}} \\ &\geq (1 + \lfloor n/2 \rfloor)(\lg n - \lg 2) \\ &\geq \frac{1}{2}n(\lg n - 1) \\ &\geq \frac{1}{4}n \lg n \quad \text{If } n \geq 4\end{aligned}$$

Consequently if $f(n) \geq B \lg(n!)$ then $f(n) \geq \frac{1}{4}Bn \lg n$ and the result follows.

- (b) Suppose that $f(n) = \Omega(n \lg n)$. We claim that $f(n) = \Omega(n!)$.

Observe that:

$$\lg(n!) = \lg(n) + \lg(n-1) + \dots + \lg(2) + \lg(1) \leq \lg n + \lg n + \dots + \lg n + \lg n = n \lg n$$

Consequently if $f(n) \geq Bn \lg n$ then $f(n) \geq B \lg(n!)$ and the result follows.

- (c) Suppose that $f(n) = \mathcal{O}(\lg(n!))$. We claim that $f(n) = \mathcal{O}(n \lg n)$.

Using the calculation from (b) we see that if $f(n) \leq C \lg(n!)$ then $f(n) \leq Cn \lg n$ and the result follows.

- (d) Suppose that $f(n) = \mathcal{O}(n \lg n)$. We claim that $f(n) = \mathcal{O}(n!)$.

Using the calculation from (a) we see that if $f(n) \leq Cn \lg n$ then $f(n) \leq 4 \lg(n!)$ and the result follows.

\mathcal{QED}

Let's look now and how this relates to decision trees. First we have:

Theorem 4.0.2. Any comparison sort requires, in the worst-case, $\Omega(n \lg n)$ (comparison-based) decisions.

Proof. A comparison-based algorithm needs to decide between $n!$ possible permutations of the list and each permutation will be a leaf in the decision tree so the number of leaves must be $n!$.

In general if h is the height of a binary tree then the number of leaves is less than or equal to 2^h (which would be a perfect binary tree) and so we must have:

$$\# \text{ Leaves} = n! \leq 2^h$$

And so:

$$h \geq \lg(n!)$$

Let d be the number of decisions necessary in the worst case. In a worst-case scenario we follow the tree as far down as possible, thus $d = h$ and so:

$$d \geq \lg(n!)$$

Thus by the lemma $d = \Omega(n \lg n)$.

\mathcal{QED}

Corollary 4.0.1. Any comparison sort has worst-case time complexity $\Omega(n \lg n)$.

Proof. Each (comparison-based) decision takes constant time and the result follows immediately. \mathcal{QED}

Consider what this is saying more specifically. We know that our various (comparison-based) sorting algorithms have best- and worst-cases. Often these are the same, like Bubble Sort ($\Theta(n^2)$) and Merge Sort ($\Theta(n \lg n)$), and sometimes they are not, like Insertion Sort (best-case $\Theta(n)$ and worst-case $\Theta(n^2)$) and Quick Sort (best-case $\Theta(n \lg n)$ and worst-case $\Theta(n^2)$).

When we think of worst-case we think of “one or more (annoying) lists”. So for example when we say Merge Sort is worst-case $\Theta(n \lg n)$ we are observing that there are some n_0, B such that for each $n \geq n_0$ there are one or more (annoying) lists which take time at least $Bn \lg n$.

What this theorem is saying is that we can never do better in the worst-case, meaning that if our sorting algorithm is comparison based then there are some n_0, B such that for each $n \geq n_0$ there are one or more (annoying) lists which will take time at least $Bn \lg n$.

5 Thoughts, Problems, Ideas

1. Consider the following modification of BubbleSort:

```
for i = 0 to n-1
    for j = n-2 down to i
        if A[j] <= A[j+1]
            nothing
        else
            swap A[j] and A[j+1]
        end
    end
end
```

Write down the decision tree for this algorithm as applied to the set $\{X, Y, Z\}$.

2. Consider the following pseudocode for InsertSort:

```
for i = 0 to n-1
    key = A[i]
    j = i-1
    while j >= 0 and key < A[j]
        A[j+1] = A[j]
        j = j - 1
    end
    A[j+1] = key
end
```

Write down the decision tree for this algorithm as applied to the set $\{X, Y, Z\}$.

3. Consider the following pseudocode for SelectionSort:

```
for i = 0 to n-2
    minindex = i
    for j = i+1 to n-1
        if A[j] < A[minindex]
            minindex = j
    end
    A[i] <-> A[minindex]
end
```

Write down the decision tree for this algorithm as applied to the set $\{X, Y, Z\}$.

CMSC 351: CountingSort

Justin Wyss-Gallifent

March 27, 2024

1	What it Does	2
2	How it Works	2
3	Pseudocode and Time Complexity	4
4	Auxiliary Space	5
5	Stability	5
6	In-Place	5
7	Usage Note	5
8	Thoughts, Problems, Ideas	6
9	Python Test	7

1 What it Does

Sorts a list of (preferably small) nonnegative integers for which we know the maximum. This can be tweaked to do a bit more.

2 How it Works

Counting sort works by counting the number of each integer and creating a new list with that information.

For example consider this list:

A	[3		10		4		3		4		5		4		3		5		6		1		3		0]
---	---	---	--	----	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	--	---	---

Knowing that the maximum value is 10 we create an list `POS` with indices from 0 to 10 such that `POS[k]` contains the count of `k` in `A`. The name `POS` we're using may seem a bit weird but it will make sense soon.

index	0	1	2	3	4	5	6	7	8	9	10												
POS	[1		1		0		4		3		2		1		0		0		0		1]
# of	0s	1s	2s	3s	4s	5s	6s	7s	8s	9s	10s												

Next we modify `POS` above by replace it with its cumulative version, meaning we successively add each entry to the next.

index	0	1	2	3	4	5	6	7	8	9	10										
POS	[1		2		2		6		9		11		12		12		12		13]

For a moment observe that this needs to give us the sorted version of `A` which looks like this:

index	0	1	2	3	4	5	6	7	8	9	10	11	12												
Sorted A	[0		1		3		3		3		4		4		4		5		5		6		10]

Observe that each `POS[i]==j` indicates that in the sorted version of the list if there are any occurrences of `i` then they end at position `j`, meaning index `j-1`.

For example `POS[3]==6` which means that in the sorted list the 3s end at position 6, meaing index 5, and `POS[4]==9` which means that in the sorted list the 4s end at position 9, meaning index 8.

Then we construct a new list `ANEW` by going through `A` in reverse order, taking each element, finding the position (and then index) to place it using `POS`, coping it over, then decreasing that entry in `POS` in preparation in case there's another value to come. So think about `A` as the input list, `POS` as the processing list, and `ANEW` as the output list.

We go in reverse order to make the sorting stable. Observe that duplicate values get filled into `ANEW` from right-to-left and so we should reverse through `A` to match that.

You may have wondered why we don't just use the first `POS` list to build a new list by simply going through `POS` and writing that many of each value. The reason is that we are typically thinking of the number in our list as keys which are associated with lots of data. This means we are not simply sorting numbers but rather we are moving around troves of information associated to those numbers. Consequently our creation of `ANEW` should be treated as though it is not just writing individual numbers but copying over that associated information.

Here are the first two steps:

`A[12]==0` (the value) and `POS[0]==1` (the position, one more than the index, to put it) so assign `ANEW[1-1]=0...`

<code>A</code>	3	10	4	3	4	5	4	3	5	6	1	3	0
----------------	---	----	---	---	---	---	---	---	---	---	---	---	---

index	0	1	2	3	4	5	6	7	8	9	10	
<code>POS</code>	1	2	2	6	9	11	12	12	12	12	13	

index	0	1	2	3	4	5	6	7	8	9	10	11	12
<code>ANEW</code>	0												

...and update `POS[0]=0` (ready for the next 0, which there isn't one).

`A[11]==3` (the value) and `POS[3]==6` (the position, one more than the index, to put it) so assign `ANEW[6-1]=3...`

<code>A</code>	3	10	4	3	4	5	4	3	5	6	1	3	0
----------------	---	----	---	---	---	---	---	---	---	---	---	---	---

index	0	1	2	3	4	5	6	7	8	9	10	
<code>POS</code>	1	2	2	6	9	11	12	12	12	12	13	

index	0	1	2	3	4	5	6	7	8	9	10	11	12
<code>ANEW</code>	0					3							

...and update `POS[3]=5` (ready for the next 3, which there is one).

And so on until we have filled up `ANEW`. Typically we then copy `ANEW` back to `A`.

The reason for going through `A` in reverse order is that it results in CountingSort being stable.

3 Pseudocode and Time Complexity

Here is the pseudocode:

```
\\" PRE: A is a list of length n with minimum 0 and maximum k
POS = list of zeros of length k+1
ANEW = list of zeros of length n
for i = 0 to n-1
    POS[A[i]] = POS[A[i]] + 1
end
for i = 1 to k
    POS[i] = POS[i] + POS[i-1]
end
for i = n-1 down to 0
    ANEW[POS[A[i]]-1] = A[i]
    POS[A[i]] = POS[A[i]] - 1
end
for i = 0 to n-1
    A[i] = ANEW[i]
end
\\" POST: A is sorted.
```

Note 3.0.1. If A is given but k is not we can simply add a simple loop to calculate it first since it's the maximum of the list. This process is $\Theta(n)$.

Best-, Worst-, and Average-Cases:

In most languages to allocate an empty list of length n it takes $\Theta(n)$ time, so our initial assignments of POS and $ANEW$ take $\Theta(k)$ and $\Theta(n)$ respectively. Note that this is the first time that something other than n is showing in our time complexity.

We then iterate the four `for` loops which take time $\Theta(n)$, $\Theta(k)$, $\Theta(n)$, and $\Theta(n)$ respectively.

All together this is then $\Theta(n + k)$. This is the same for best-case, worst-case, and average-case.

If this is confusing, think of $n + k$ as a single variable, so what this means is that there are $B, C > 0$ and some index m_0 such that:

$$\text{If } n + k \geq m_0 \text{ then } B(n + k) \leq T(n, k) \leq C(n + k)$$

Note 3.0.2. If k is a fixed constant and we let n vary then the time complexity is $\Theta(n)$.

Note 3.0.3. If k is not fixed but we can guarantee that $k \leq n$ then the time complexity is $\Theta(n)$.

4 Auxiliary Space

CountingSort uses $\Theta(n + k)$ auxiliary space since it is required to create the POS list as well as the ANEW list.

5 Stability

Our particular pseudocode implementation of CountingSort is stable.

6 In-Place

CountingSort is not in-place.

7 Usage Note

CountingSort can be modified to manage other types of data. See the exercises for examples. It is often used as an auxiliary method inside other methods like RadixSort.

8 Thoughts, Problems, Ideas

1. Suppose $k_1, k_2 \in \mathbb{Z}$ with $k_1 < k_2$. Suppose every x in your list is an integer satisfying $k_1 \leq x \leq k_2$. Modify CountingSort so that it can manage this list. What would the Θ time complexity be?
2. Suppose $k_1, k_2 \in \mathbb{R}$ with $k_1 < k_2$ and both having at most one digit after the decimal point. Suppose every x in your list is a real number satisfying $k_1 \leq x \leq k_2$ with x having at most one digit after the decimal point. Modify CountingSort so that it can manage this list. What would the Θ time complexity be?
3. Suppose $d \in \mathbb{Z}$ with $d \geq 0$, $k_1, k_2 \in \mathbb{R}$ with $k_1 < k_2$ and both having at most d digits after the decimal point. Suppose every x in your list is a real number satisfying $k_1 \leq x \leq k_2$ with x having at most d digits after the decimal point. Modify CountingSort so that it can manage this list. What would the Θ time complexity be?

9 Python Test

Code:

```
import random
A = []
k = 5
n = 10
for i in range(0,n):
    A.append(random.randint(0,k))
print(A)
ANEW = [0] * n
LOC = [0] * (k+1)
for i in range(0,n):
    LOC[A[i]] = LOC[A[i]] + 1
print('Count array: '+str(LOC))
for i in range(1,k+1):
    LOC[i] = LOC[i] + LOC[i-1]
print('Cumulative count array: '+str(LOC))
for i in range(n-1,-1,-1):
    ANEW[LOC[A[i]]-1] = A[i]
    LOC[A[i]] = LOC[A[i]] - 1
    print('Positioning A['+str(i)+'] in position '+str(LOC[A[i]]-1))
    print('Result: '+str(ANEW))
for i in range(0,n):
    A[i] = ANEW[i]
print(A)
```

Output:

```
[3, 2, 0, 5, 2, 5, 0, 1, 2, 1]
Count list: [2, 2, 3, 1, 0, 2]
Cumulative count list: [2, 4, 7, 8, 8, 10]
Positioning A[9] in position 2
Result: [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
Positioning A[8] in position 5
Result: [0, 0, 0, 1, 0, 0, 2, 0, 0, 0]
Positioning A[7] in position 1
Result: [0, 0, 1, 1, 0, 0, 2, 0, 0, 0]
Positioning A[6] in position 0
Result: [0, 0, 1, 1, 0, 0, 2, 0, 0, 0]
Positioning A[5] in position 8
Result: [0, 0, 1, 1, 0, 0, 2, 0, 0, 5]
Positioning A[4] in position 4
Result: [0, 0, 1, 1, 0, 2, 2, 0, 0, 5]
Positioning A[3] in position 7
Result: [0, 0, 1, 1, 0, 2, 2, 0, 5, 5]
Positioning A[2] in position -1
Result: [0, 0, 1, 1, 0, 2, 2, 0, 5, 5]
Positioning A[1] in position 3
Result: [0, 0, 1, 1, 2, 2, 2, 0, 5, 5]
Positioning A[0] in position 6
Result: [0, 0, 1, 1, 2, 2, 2, 3, 5, 5]
[0, 0, 1, 1, 2, 2, 2, 3, 5, 5]
```

CMSC 351: RadixSort

Justin Wyss-Gallifent

October 25, 2023

1	What it Does	2
2	How it Works	2
3	Details	3
3.1	Some Variables	3
3.2	Using Counting Sort for the Stable Sort	3
3.3	Using Another Sort of Sort for the Stable Sort	4
4	Thoughts, Problems, Ideas	5
5	Python Test	6

1 What it Does

Consider the following three situations:

- We are sorting a list of n decimal numbers all between 000 and 999 inclusive.
- We are sorting a list of n binary numbers all between 00000000 and 11111111 inclusive.
- We are sorting a list of n 5-digit strings, each string is made up of the letters A through Z inclusive.

In each of these situations we are sorting a collection of items where each item is a string of either numbers or letters. Radix sort often works especially well for such situations.

2 How it Works

Radix Sort works as follows:

1. Stable sort the list by the least significant digit.
2. Stable sort the list by the next most significant digit.
3. Etc.

Let's see how this works with an example.

Example 2.1. For example consider the following list. Observe that we have prepended 0s as necessary to equalize the lengths.

170,045,075,090,802,024,002,066

First we identify each right-most (least significant) digit:

170, 045, 075, 090, 802, 024, 002, 066

We sort the list according to that digit, preserving order within that group:

170, 090, 802, 002, 024, 045, 075, 066

Then we identify the next digit, moving leftwards:

170, 090, 802, 002, 024, 045, 075, 066

We sort the list according to that digit, preserving order within that group:

802, 002, 024, 045, 066, 170, 075, 090

Then we identify the next digit, moving leftwards:

802, 002, 024, 045, 066, 170, 075, 090

We sort the list according to that digit, preserving order within that group:

002, 024, 045, 066, 075, 090, 170, 802,

| Now we are done.

3 Details

3.1 Some Variables

Our list has n items in it. Let d be the number of digits (or characters) in each item in the list and let b be the base, or number of different characters available.

| **Example 3.1.** If we're sorting n decimal numbers between 000 and 999 inclusive then $d = 3$ and $b = 10$.

| **Example 3.2.** If we're sorting n binary numbers between 00000000 and 11111111 inclusive then $d = 8$ and $b = 2$.

| **Example 3.3.** If we're sorting n strings between AAAA and ZZZZZ inclusive then $d = 5$ and $b = 26$. Here technically we're saying $A = 0$ and $Z = 25$.

3.2 Using Counting Sort for the Stable Sort

We commented early that we "Stable sort the list..." but we didn't say how so let's visit that now.

In essentially all cases the base b is independent of n and so we are effectively doing d separate stable sorts and for each stable sort we are sorting by a single digit or character in each item. Essentially that means for each stable sort we are treating each item as if it were just that digit or character with the other digits or characters just going along for the ride.

If b is the base then this means each stable sort is sorting n items where each item is essentially a number between 0 and $b - 1$. Assuming b is independent of n then we might as well use Counting Sort to do the stable sort part.

If this is the case then each counting sort is $\Theta(n + (b - 1))$ and we are doing d of those so the time complexity is $\Theta(d(n + (b - 1)))$. Again assuming b is independent of n then this is $\Theta(dn)$.

Of course if d is independent of n then this becomes $\Theta(n)$.

| **Example 3.4.** If we're sorting n decimal numbers between 000 and 999 inclusive then $d = 3$ and $b = 10$ and the time complexity is $\Theta(3(n + (10 - 1))) = \Theta(n)$.

| **Example 3.5.** If we're sorting n binary numbers between 00000000 and 11111111 inclusive then $d = 8$ and $b = 2$ and the time complexity is $\Theta(8(n + (2 - 1))) = \Theta(n)$.

| **Example 3.6.** If we're sorting n strings between AAAA and ZZZZZ inclusive then $d = 5$ and $b = 26$. Here technically we're saying $A = 0$ and $Z = 25$ and the time complexity is $\Theta(5(n + (26 - 1))) = \Theta(n)$.

It might be tempting to conclude that we always get $\Theta(n)$ and this is true when both b and d are independent of n , but certainly we can fabricate ideas where this is not true.

It's possible that the number of digits d might change with n .

Example 3.7. If we're sorting n numbers between 0 and $2^n - 1$ represented in binary then $d = n$ and $b = 2$ and the time complexity is $\Theta(n(n + (2-1))) = \Theta(n^2)$.

It's less likely that the base may change with n .

As far as auxiliary space, if we're using Counting Sort then that underlying counting sort is $\Theta(n + (b-1))$. We repeat this d times but this is in series so the memory of each sorting step is freed up before the next one and so the auxiliary space is $\Theta(n + (b-1))$.

As far as stability, Radix Sort is stable.

As far as in-place, if we're using Counting Sort then the underlying sort itself is not in-place and so Radix Sort will not be.

3.3 Using Another Sort of Sort for the Stable Sort

Because b is essentially always fixed it makes sense to use Counting Sort for the underlying stable sort but this is not really mandatory. We could use any other stable sorting method such as Bubble Sort, Merge Sort, etc.

The main issue with this of course is that the underlying sort will not be best-case $\Theta(n)$ as counting sort is and this would affect Radix Sort.

Example 3.8. If we're sorting n decimal numbers between 000 and 999 inclusive then $d = 3$ and $b = 10$. If we use Merge Sort as the underlying sort then each Merge Sort pass is $\Theta(n \lg n)$ and so the time complexity of Radix Sort would be $\Theta(d(n \lg n))$.

4 Thoughts, Problems, Ideas

1. Suppose M is a fixed positive integer and b is a fixed base. Suppose we have a list of length n which contains integers between 0 and M inclusive. Explain why, if RadixSort+CountingSort is used to sort the list, the time complexity is the same.
2. Suppose $b = 2$ is the fixed base. Suppose that we have a list of length n which contains integers between 0 and n inclusive. Show that the RadixSort+CountingSort time complexity will be $\Theta(n \lg n)$.
Hint: How many digits are needed to represent the integers 0 through n inclusive in base 2? Does the fixed base matter?
3. Suppose that we have a list of length n which contains integers between 0 and $n - 1$ inclusive. Show that the RadixSort+CountingSort time complexity can in fact be made to be $\Theta(n)$. Hint: What can we make b equal to?
4. Explain how RadixSort+CountingSort can sort n integers between 0 and $n^2 - 1$ inclusive in $\Theta(n)$ time with two iterations of CountingSort.
5. Explain how RadixSort+CountingSort can sort n integers between 0 and $n^3 - 1$ inclusive in $\Theta(n)$ time with three iterations of CountingSort.
6. Explain how RadixSort+CountingSort can sort n integers between 0 and $n^n - 1$ inclusive in $\Theta(n^2)$ time with n iterations of CountingSort.
7. Suppose that M is a fixed positive integer and suppose we have a list of length n which contains integers between 0 and M inclusive. Show that the RadixSort+CountingSort time complexity will be $\Theta\left(\frac{n \lg M}{\lg n}\right)$.
8. Radix sort is the method used for alphabetizing whereby the underlying sort simply sorts by character in some way. Don't worry about how this underlying sort works. Show how RadixSort works on the list of words:

ANTE,ANTI,AMEX,BITE,BARK,INTO,INIT,
LARK,PARK,PACK,RITE,UNTO,UNIT,ZOOT

You only need to show the result of each RadixSort loop iteration.

9. Suppose a list A contains integers between 0 and 999 inclusive. For indices i, j we wish to compare $A[i]$ and $A[j]$ and return the minimum but we are not permitted to use any sort of comparison. Explain how we could use RadixSort to do this. What would the Θ time complexity be? Explain.

5 Python Test

We assume arrays are global and we use Counting Sort for the underlying sort. The Radix here is 10.

```
import random
# This unstable version of counting sort
# sorts by the digit passed to it.
# digit = 1,10,100,etc.
def countingsort(A,digit):
    n = len(A)
    ANEW = [0] * n
    C = [0] * 10
    for i in range(0,n):
        sdigit = int((A[i]/digit)%10)
        C[sdigit] = C[sdigit] + 1
    for i in range(1,10):
        C[i] = C[i] + C[i-1]
    for i in range(n-1,-1,-1):
        sdigit = int((A[i]/digit)%10)
        ANEW[C[sdigit]-1] = A[i]
        C[sdigit] = C[sdigit] - 1
    for i in range(0,n):
        A[i] = ANEW[i]
# The radixsort function sorts by increasing digit.
def radixsort(A):
    maxval = max(A)
    d = 1
    while d < maxval:
        print('Sorting by radix: '+str(d))
        countingsort(A,d)
        print('Result: '+str(A))
        d = 10*d
A = []
for i in range(0,15):
    A.append(random.randint(0,1000))
n = len(A)
print(A)
radixsort(A)
print(A)
```

Output:

```
[91, 546, 43, 88, 954, 731, 975, 285, 737, 519, 830, 686, 744, 637, 322]
Sorting by radix: 1
Result: [830, 91, 731, 322, 43, 954, 744, 975, 285, 546, 686, 737, 637, 88,
Sorting by radix: 10
Result: [519, 322, 830, 731, 737, 637, 43, 744, 546, 954, 975, 285, 686, 88,
Sorting by radix: 100
Result: [43, 88, 91, 285, 322, 519, 546, 637, 686, 731, 737, 744, 830, 954,
[43, 88, 91, 285, 322, 519, 546, 637, 686, 731, 737, 744, 830, 954, 975]
```

CMSC 351: Integer Multiplication

Justin Wyss-Gallifent

March 29, 2022

1	Introduction	2
2	Schoolbook Multiplication	2
	2.1 Method	2
	2.2 Time Complexity	2
3	A Sneaky Approach	3
	3.1 Two 2-Digit Numbers	3
	3.2 Two 4-Digit Numbers	4
	3.3 Generalized	4
4	Karatsuba Method Theory	5
5	Karatsuba Method Implementation	6
	5.1 Recursion Note 1	6
	5.2 Recursion Note 2	6
	5.3 Splitting Note	6
6	Tree Diagrams	7
7	Pseudocode	8
8	Thoughts, Problems, Ideas	9
9	Python Code and Output	11

1 Introduction

Suppose we have two n -digit numbers and wish to multiply them. What is the worst-case time complexity of this operation?

2 Schoolbook Multiplication

2.1 Method

The first and most obvious way to multiply two numbers is the way we learn in school. Here is an example. The carry digits are not shown:

$$\begin{array}{r} 1 \quad 0 \quad 2 \\ 2 \quad 5 \quad 7 \\ \hline 7 \quad 1 \quad 4 \\ 5 \quad 1 \quad 0 \\ 2 \quad 0 \quad 4 \\ \hline 2 \quad 6 \quad 2 \quad 1 \quad 4 \end{array}$$

2.2 Time Complexity

What is the time complexity of this algorithm? Well row i in the intermediate calculation equals digit b_i multiplied by each digit in A , (with a possible carry addition) so row i requires n operations. Since there are n rows there are n^2 digit multiplications (each with a possible carry addition). Without even worrying about the additions we're at $\Theta(n^2)$.

Could we do any better?

3 A Sneaky Approach

3.1 Two 2-Digit Numbers

Let $A = a_1a_0$ and $B = b_1b_0$ be the base-10 digit representations of two 2-digit numbers. In reality then $A = 10a_1 + a_0$ and $B = 10b_1 + b_0$ and the product is:

$$AB = (10a_1 + a_0)(10b_1 + b_0) = 100a_1b_1 + 10(a_1b_0 + a_0b_1) + a_0b_0$$

For now let's ignore the 100 and the 10 and consider that we have four multiplications. Can we do better?

Observe that:

$$a_1b_0 + a_0b_1 = (a_1 + a_0)(b_1 + b_0) - a_0b_0 - a_1b_1$$

So consequently the middle expression can actually be calculated using two multiplications that we've already done as well as one new one. In summary:

$$AB = 100a_1b_1 + 10[(a_1 + a_0)(b_1 + b_0) - a_0b_0 - a_1b_1] + a_0b_0$$

Of course you may observe that the newly required product $(a_1 + a_0)(b_1 + b_0)$ may itself be the product of two 2-digit numbers but for now let's just be satisfied that they're certainly smaller than the original two 2-digit numbers.

In addition in order to guarantee that we have actually obtained the digits of the product AB we will actually need to perform the multiplications by 100 and 10 and the resulting additions.

However multiplication by a 100 can be done using two decimal shifts (insert two zeros) and multiplication by 10 can be done using one decimal shift (insert one zero) and shifting by n digits is $\Theta(n)$ (insert n zeros).

Thus in total to calculate all of the required digits precisely we have:

- A total of 3 multiplications, a_0b_0 , a_1b_1 , and $(a_1 + a_0)(b_1 + b_0)$, two of which have half as many digits as the original two numbers and one is a significantly easier product.
- A total of 6 additions/subtractions of numbers with at most 4 digits.
- A total of $2 + 1 = 3$ decimal shifts.

3.2 Two 4-Digit Numbers

Suppose we wanted to calculate a product such as $(1234)(5678)$. Ordinarily this would take 16 single-digit multiplications.

Instead let's write two 4-digit numbers as $A = A_1A_0$ and $B = B_1B_0$ where the A_i and B_i are pairs of digits. In reality then $A = 100A_1 + A_0$ and $B = 100B_1 + B_0$ and the product is:

$$AB = (100A_1 + A_0)(100B_1 + B_0) = 10000A_1B_1 + 100(A_1B_0 + A_0B_1) + A_0B_0$$

Again observe that:

$$A_1B_0 + A_0B_1 = (A_1 + A_0)(B_1 + B_0) - A_0B_0 - A_1B_1$$

So once again the middle term can be calculated using two multiplications that we've already done as well as one new one, and again the required new product is certainly simpler than the original one. In summary again:

$$AB = 10000A_1B_1 + 100[(A_1 + A_0)(B_1 + B_0) - A_0B_0 - A_1B_1] + A_0B_0$$

To calculate this out in order to obtain the digits we have:

- A total of 3 multiplications, A_0B_0 , A_1B_1 , and $(A_1 + A_0)(B_1 + B_0)$, two of which have half as many digits as the original two numbers and one is a significantly easier product.
- A total of 6 additions/subtractions of numbers with at most 8 digits.
- A total of $4 + 2 = 6$ decimal shifts.

Importantly note that the three multiplications can essentially be done by applying the method for two 2-digit numbers.

3.3 Generalized

This approach will then extend to two 8-digit numbers, two 16-digit numbers, and so on. In general if $A = A_1A_0$ and $B = B_1B_0$ where the A_i and B_i are two n -digit numbers where (for simplicity) we'll say n is even then we can write:

$$AB = 10^n(A_1B_1) + 10^{n/2}[(A_1 + A_0)(B_1 + B_0) - A_0B_0 - A_1B_1] + A_0B_0$$

Then we can reduce finding the digits of AB to:

- A total of 3 multiplications, A_0B_0 , A_1B_1 , and $(A_1 + A_0)(B_1 + B_0)$, two of which have half as many digits as the original two numbers and one is a significantly easier product.
- A total of 6 additions/subtractions of numbers with at most $2n$ digits.
- A total of $n + n/2$ decimal shifts.

4 Karatsuba Method Theory

This leads to the following generalized observation. Suppose A and B are both n digit numbers. Calculation of the digits of the multiplication AB can be done using three multiplications involving numbers with essentially half as many digits and then $\Theta(n)$ worth of addition and shifts.

Thus if $T(n)$ is the time complexity for multiplying A and B then $T(n)$ satisfies:

$$T(n) = 3T(n/2) + \Theta(n)$$

The Master Theorem with $a = 3$, $b = 2$ and $c = 1$ tells us that since $1 < \log_2 3$ that:

$$T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{\lg 3}) \approx \Theta(n^{1.5849625})$$

This is significantly faster than $\Theta(n^2)$, especially for large n .

For smaller n of course it depends upon the actual specifics of the time requirements.

Note 4.0.1. We're playing fast and loose with powers of 2, half-sizes and so on, but this is just to keep the explanation tidy and avoid fiddly cases and floor and ceiling functions. The result still holds with those details added, it's just far harder to understand.

Note 4.0.2. As we'll see in the actual Python implementation for small numbers the savings (in single-digit multiplications) are practically nonexistent. For large numbers they're very noticeable.

5 Karatsuba Method Implementation

5.1 Recursion Note 1

The implementation is actually sneakier than we might suspect. Recall the original case where $A = a_1a_0$ and $b = b_1b_0$ we noticed that the product $(a_1 + a_0)(b_1 + b_0)$ could potentially be a product of two 2-digit numbers. When applying the Karatsuba method we use recursion even here if necessary to ensure the reduction of everything to single-digit products.

So for example if $A = 76$ and $B = 48$ then we have:

$$(76)(48) = 100(7)(4) + 10[(7+6)(4+8) - (7)(4) - (6)(8)] + (6)(8)$$

Note the central term $(13)(12)$. We apply recursion again:

$$(13)(12) = 100(1)(1) + 10[(1+3)(1+2) - (1)(1) - (3)(2)] + (3)(2)$$

Now the central term involves the multiplication of two 1-digit numbers.

5.2 Recursion Note 2

In the case where one of A and B has one digit then even if the other has more than one, at this point it's certainly $\Theta(n)$ to simply multiply. Consequently the actual implementation of Karatsuba's algorithm uses this as the base case in the recursion so we've done this in the pseudocode too.

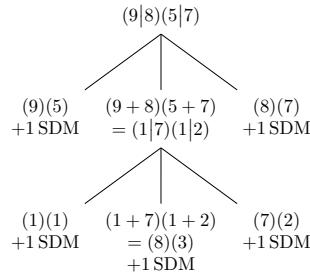
5.3 Splitting Note

Since it's entirely possible that A and B have differing numbers of digits we split based upon the shortest one in order to guarantee that both can, in fact, be split. Additionally we split from the right side (the units digit) to ensure that the decimal shifting (multiplication by powers of 10) works appropriately.

6 Tree Diagrams

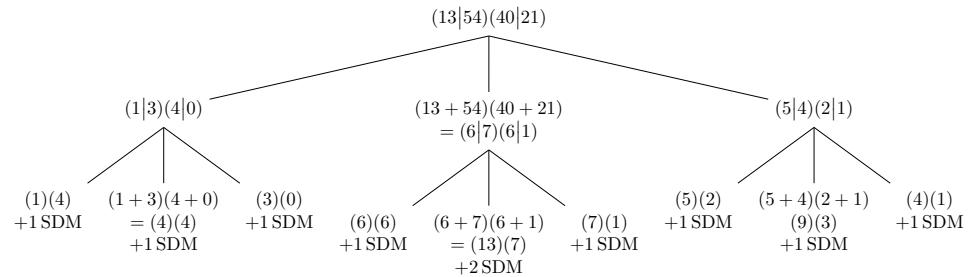
A tree diagram can succinctly show how Karatsuba's Algorithm plays out in terms of the breaking down to single-digit multiplications.

Example 6.1. Consider the product $(98)(57)$. This tree breaks down all the products until one of the numbers is single-digit:



From here we can see that there are 5 single-digit multiplications. This example is actually slower than schoolbook multiplication, which would require $(2)(2) = 4$ single-digit multiplications. ■

Example 6.2. Consider the product $(1354)(4021)$. This tree breaks down all the products until one of the numbers is single-digit:



From here we can see that there are 10 single-digit multiplications. This example is faster than schoolbook multiplication, which would require $(4)(4) = 16$ single-digit multiplications. ■

7 Pseudocode

The pseudocode can be a bit confusing because it needs to manage two numbers with differing numbers of digits, it needs to manage finding a split location and split from the right, and it needs to manage the powers of 10 correctly.

```
\ A,B are the list representations of numbers.
function karatsuba(A,B)
    if either A or B is single-digit
        return(A*B)
    else
        sp = floor((minimum number of digits in A,B)/2)
        A1,A0 = split A, sp digits from the right
        B1,B0 = split B, sp digits from the right
        k1 = karatsuba(A1,B1)
        k2 = karatsuba(A1+A0,B1+B0)
        k3 = karatsuba(A0,B0)
        // The powers of 10 should be thought of as shifts.
        r = 10^(2*sp)*k1 + 10^(sp)*(k2-k3-k1) + k3
        return(r)
    end
end
```

8 Thoughts, Problems, Ideas

1. Draw a tree diagram for Karatsuba's Algorithm applied to (12)(34) and use it to count the number of SDMs.
2. Draw a tree diagram for Karatsuba's Algorithm applied to (98)(76) and use it to count the number of SDMs.
3. Draw a tree diagram for Karatsuba's Algorithm applied to (1201)(2231) and use it to count the number of SDMs.
4. Draw a tree diagram for Karatsuba's Algorithm applied to (345)(12231) and use it to count the number of SDMs. This is a little trickier than the previous few because of the different integer lengths. Trust the pseudocode!
5. Let $A = a_2a_1a_0$ and $B = b_2b_1b_0$ be the base-10 digit representations of two 3-digit numbers. Formally then $A = 100a_2 + 10a_1 + a_0$ and $B = 100b_2 + 10b_1 + b_0$.
 - (a) Evaluate the product AB and collect the result into the form:
$$10000(T1) + 1000(T2) + 100(T3) + 10(T4) + (T5)$$
We'll say that the $T1, \dots$ are the terms. How many different products arise in all of the terms together?

- (b) Rewrite the terms $T2$ and $T4$ in such a way as to reduce the number of total multiplications which are necessary to evaluate the product down to 7.
- (c) Explain (not even pseudocode) how this might lead to a recursive algorithm for multiplication much like the Karatsuba Algorithm.
- (d) Write down the recurrence relation for this algorithm and solve it using the Master Theorem. Is it faster or slower than Karatsuba?

6. We observed that if n is even and $A = A_1A_0$ and $B = B_1B_0$ are two n -digit numbers that the special product $(A_1 + A_0)(B_1 + B_0)$ might not be a product of two $n/2$ -digit numbers, thereby making the recurrence relation $T(n) = 3T(n/2) + \Theta(n)$ feel a little iffy. We glossed over this but let's patch it a bit.
- (a) Show that in the $n = 2$ case that even if both $a_1 + a_0$ and $b_1 + b_0$ are not single-digit numbers that the product $(a_1 + a_0)(b_1 + b_0)$ can be calculated using one single-digit multiplication and $\Theta(n)$ additions and decimal shifts. [10 pts]
- (b) Generalize this argument in the following sense: Show that even if both $A_1 + A_0$ and $B_1 + B_0$ are not $n/2$ -digit numbers that the product $(A_1 + A_0)(B_1 + B_0)$ can be calculated using one $n/2$ -digit multiplication and $\Theta(n)$ additions and decimal shifts, thereby rendering the recurrence relation accurate. [10 pts]

9 Python Code and Output

Code:

```
import random
import math

numdig = 4
AA = random.randint(10***(numdig-1),10***(numdig)-1)
BB = random.randint(10***(numdig-1),10***(numdig)-1)
mcounter = 0

AA = 3451
BB = 45862

def karatsuba(A,B,indent):
    global mcounter
    #print(indent*' ' + 'Pair = ' + str(A) + ' ' + str(B))
    if A<10 or B<10:
        mcounter = mcounter + (len(str(A))*len(str(B)))
        print(indent*' ' + 'Base Product = ' + str(A*B))
        return(A*B)
    else:
        AS = [int(i) for i in str(A)]
        BS = [int(i) for i in str(B)]
        sl = min(len(AS),len(BS)) // 2
        A1 = int(''.join(map(str,AS[0:len(AS)-sl])))
        A0 = int(''.join(map(str,AS[len(AS)-sl:])))
        B1 = int(''.join(map(str,BS[0:len(BS)-sl])))
        B0 = int(''.join(map(str,BS[len(BS)-sl:])))
        print(indent*' ' + 'Recurse to (A1,B1) = ('+str(A1)+','+str(B1)+')')
        k1 = karatsuba(A1,B1,indent+2)
        print(indent*' ' + 'Recurse to (A1+A0,B1+B0) = ('+str(A1)+'+'+str(A0)+','+str(B1)+'+'+str(B0)+')')
        k2 = karatsuba(A1+A0,B1+B0,indent+2)
        print(indent*' ' + 'Recurse to (A0,B0) = ('+str(A0)+','+str(B0)+')')
        k3 = karatsuba(A0,B0,indent+2)
        k = (10***(2*sl))*k1 + (10***(sl))*(k2-k3-k1) + k3
        print(indent*' ' + 'Product = ' + str(10***(2*sl)) + '*' + str(k1) + '+' + str(k2) + '+' + str(k3))
        return(k)

print('Start (A,B) = ('+str(AA)+','+str(BB)+')')
p = karatsuba(AA,BB,2)

print('n = ' + str(numdig))
print('Result: ' + str(p))
print('Number of SDM: ' + str(mcounter))
print('Note that '+str(numdig)+'^lg(3) = ' + str(numdig**math.log(3,2)))
print('Direct Python Calculation: ' + str(AA*BB))
print('Would Take SDM: ' + str(numdig**2))
```

Output with two 2-digit numbers. Observe this takes exactly the same number of SDM as schoolbook multiplication and a bit less than twice $n^{\lg 3}$:

```
Start (A,B) = (95,96)
Recurse to (A1,B1) = (9,9)
    Base Product = 81
Recurse to (A1+A0,B1+B) = (9+5,9+6) = (14,15)
    Recurse to (A1,B1) = (1,1)
        Base Product = 1
    Recurse to (A1+A0,B1+B) = (1+4,1+5) = (5,6)
        Base Product = 30
    Recurse to (A0,B0) = (4,5)
        Base Product = 20
    Product = 100*1+10*(30-20-1)+20 = 210
Recurse to (A0,B0) = (5,6)
    Base Product = 30
Product = 100*81+10*(210-30-81)+30 = 9120
n = 2
Result: 9120
Number of SDM: 5
Note that  $2^{\lg(3)} = 3.0000000000000004$ 
Direct Python Calculation: 9120
Would Take SDM: 4
```

Output with two 3-digit numbers. Observe this takes exactly the same number of SDM as schoolbook multiplication and again a bit less than twice $n^{\lg 3}$:

```
Start (A,B) = (840,240)
Recurse to (A1,B1) = (84,24)
    Recurse to (A1,B1) = (8,2)
        Base Product = 16
    Recurse to (A1+A0,B1+B) = (8+4,2+4) = (12,6)
        Base Product = 72
    Recurse to (A0,B0) = (4,4)
        Base Product = 16
    Product = 100*16+10*(72-16-16)+16 = 2016
Recurse to (A1+A0,B1+B) = (84+0,24+0) = (84,24)
    Recurse to (A1,B1) = (8,2)
        Base Product = 16
    Recurse to (A1+A0,B1+B) = (8+4,2+4) = (12,6)
        Base Product = 72
    Recurse to (A0,B0) = (4,4)
        Base Product = 16
    Product = 100*16+10*(72-16-16)+16 = 2016
Recurse to (A0,B0) = (0,0)
    Base Product = 0
```

```
Product = 100*2016+10*(2016-0-2016)+0 = 201600
n = 3
Result: 201600
Number of SDM: 9
Note that  $3^{\lg(3)}$  = 5.704522494691118
Direct Python Calculation: 201600
Would Take SDM: 9
```

Output with two 4-digit numbers. Observe this takes fewer SDM as schoolbook multiplication and again a bit less than twice $n^{\lg 3}$:

```
Start (A,B) = (7087,2600)
Recurse to (A1,B1) = (70,26)
    Recurse to (A1,B1) = (7,2)
        Base Product = 14
    Recurse to (A1+A0,B1+B) = (7+0,2+6) = (7,8)
        Base Product = 56
    Recurse to (A0,B0) = (0,6)
        Base Product = 0
    Product = 100*14+10*(56-0-14)+0 = 1820
Recurse to (A1+A0,B1+B) = (70+87,26+0) = (157,26)
    Recurse to (A1,B1) = (15,2)
        Base Product = 30
    Recurse to (A1+A0,B1+B) = (15+7,2+6) = (22,8)
        Base Product = 176
    Recurse to (A0,B0) = (7,6)
        Base Product = 42
    Product = 100*30+10*(176-42-30)+42 = 4082
Recurse to (A0,B0) = (87,0)
    Base Product = 0
    Product = 10000*1820+100*(4082-0-1820)+0 = 18426200
n = 4
Result: 18426200
Number of SDM: 10
Note that  $4^{\lg(3)} = 9.000000000000002$ 
Direct Python Calculation: 18426200
Would Take SDM: 16
```

Summary with two 10-digit numbers. Observe this takes fewer SDM as school-book multiplicaion and again a bit less than twice $n^{\lg 3}$:

```
Pair = 8035207000 9075773597
...
n = 10
Result: 72925719537029579000
Number of SDM: 64
Note that  $10^{\lg(3)} = 38.45585757936911$ 
Direct Python Calculation: 72925719537029579000
Would Take SDM: 100
```

Summary with two 20-digit numbers. Observe this takes fewer SDM as school-book multiplicaion and again a bit less than twice $n^{\lg 3}$:

```
Pair = 49521157366056646229 93687401978021091533
...
n = 20
Result: 4639508576570589185099645217795808279057
Number of SDM: 222
Note that  $20^{\lg(3)} = 115.36757273810733$ 
Direct Python Calculation: 4639508576570589185099645217795808279057
Would Take SDM: 400
```

Very brief summary with two 100-digit numbers (numbers not listed):

```
n = 100
Result: ...
Number of SDM: 2721
Note that  $100^{\lg(3)} = 1478.8529821647205$ 
Direct Python Calculation: ...
Would Take SDM: 10000
```

Very brief summary with two 1000-digit numbers (numbers not listed):

```
n = 1000
Result: ...
Number of SDM: 98486
Note that  $1000^{\lg(3)} = 56870.559662951775$ 
Direct Python Calculation: ...
Would Take SDM: 1000000
```

In each case observe that that the number of SDM is a bit less than twice $n^{\lg 3}$ which aligns with our assertion that Karatsuba is $\Theta(n^{\lg 3})$.

CMSC 351: Graphs

Justin Wyss-Gallifent

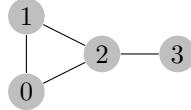
April 7, 2024

1	Introduction	2
2	Summary of Definitions	2
3	Definitions	3
4	Counting Notation	4
5	Walks, Trails, Paths, Cycles, Trees	4
6	Adjacency Matrices	5
7	Adjacency Lists	6
8	Considerations	7
	8.1 Storage	7
	8.2 Access	7
	8.3 Math	7
9	The Degree and Other Matrices for a Graph	8
10	Thoughts, Problems, Ideas	9

1 Introduction

Graphs are essentially diagrams (or data, at an abstract level) which represent networks and in general connectivity between objects.

For example the following diagram could represent the connectivity between four computers:



Here we see that computers 0,1,2 are connected to one another and computer 3 is only connected to computer 2.

2 Summary of Definitions

This is just here to help organize. We define:

1. Graph, edge, vertex (node).
2. Adjacent edges.
3. Degree.
4. Loop.
5. Multiple edge(s).
6. Simple.
7. Weighted and unweighted.
8. Directed and undirected.
9. Walk, trail, path.
10. Connected graph.
11. Cycle.
12. Tree.
13. Adjacency matrix for an unweighted undirected simple graph.
14. Adjacency matrix for a weighted directed simple graph.
15. Adjacency list for an unweighted undirected simple graph.
16. Degree matrix for an unweighted undirected simple graph.
17. Laplacian matrix for an unweighted undirected simple graph.
18. These final five definitions are not comprehensive nor completely uniform.

3 Definitions

We have the following formal definitions:

Definition 3.0.1. A *graph* consists of *vertices* or *nodes* connected by *edges*.

| **Example 3.1.** The diagram above is a graph consisting of four vertices and four edges.

Definition 3.0.2. Two vertices are *adjacent* if they are connected by an edge.

Definition 3.0.3. The *degree* of a vertex is the number of edge connections incident at that vertex.

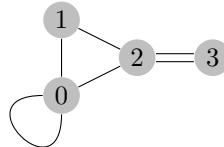
| **Example 3.2.** In the above graph vertices 0,1 have degree 2, vertex 2 has degree 3 and vertex 3 has degree 1.

Note 3.0.1. If the graph has loops and multiple edges (see below) then these must be accounted for.

Definition 3.0.4. A *loop* is an edge which joins a vertex to itself.

Definition 3.0.5. Two vertices may be joined by more than one edge, in which case we say there are *multiple edges* between the vertices.

| **Example 3.3.** This graph has a loop (vertex 0 to itself) and multiple edges (vertex 2 to vertex 3):



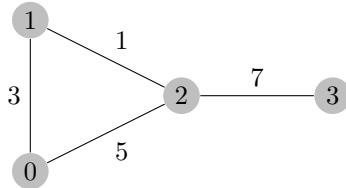
Definition 3.0.6. A graph is *simple* if it has no loops and no multiple edges.

| **Example 3.4.** The first graph is simple.

Definition 3.0.7. A graph is *weighted* if each edge has a numerical weight associated to it.

Most, if not all, of the graphs we will look at will be weighted simple graphs where each weight yields some sort of cost of connection. A simple example of a road network between cities in which each weight is the distance.

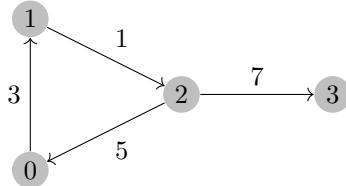
| **Example 3.5.** Here is a weighted simple graph:



Definition 3.0.8. A graph is *directed* if the edges have directions, usually indicated by arrows.

Working with directed graphs is much more challenging than working with undirected graphs.

Example 3.6. Here is a weighted simple directed graph:



4 Counting Notation

It's common to use V for the number of vertices and E for the number of edges. Sometimes V and E are also used for the sets of vertices and edges but I'll try to avoid that here.

5 Walks, Trails, Paths, Cycles, Trees

Definition 5.0.1. A *walk* of length k from vertex u to vertex v is a sequence of k edges which start at vertex u and end at vertex v .

Intuitively a walk is simply a way of getting from vertex u to vertex v . It's permissible to repeat edges and vertices.

Definition 5.0.2. A *trail* is a walk in which all edges are distinct. Note that vertices may be repeated.

Note 5.0.1. Walks and trails of length 0 are permitted. Specifically there is a walk/trail of length 0 from each vertex to itself.

Definition 5.0.3. A *path* is a walk in which all vertices are distinct. Note that if vertices are distinct then so are edges.

Note 5.0.2. We can't have a path of length 0 because we can't repeat vertices.

We will primarily focus on paths because these make the most sense when we're discussing optimization. For example if you wanted the shortest distance between two cities you would certainly not repeat any of the intermediate cities

or roads!

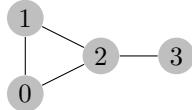
Definition 5.0.4. A *cycle* is a trail in which all vertices are distinct except we insist that the starting and ending vertices are the same.

Note 5.0.3. In a graph in which loops are allowed a loop is a cycle consisting of the point and the edge.

Definition 5.0.5. A graph is *connected* if for any two vertices there is at least one path joining those vertices.

Definition 5.0.6. Formally a *tree* is an undirected simple connected graph with the property that every two vertices are connected by exactly one path.

Example 5.1. In our original graph:



Here are some observations:

- Formally we have $VS = \{0, 1, 2, 3\}$ and $ES = \{(0, 1), (0, 2), (1, 2), (2, 3)\}$.
- The sequence $\langle 3, 2, 0, 2, 1 \rangle$ is a walk of length 4 from vertex 3 to vertex 1. It is not a trail since the edge $(0, 2)$ is repeated and so automatically it is not a path either.
- The sequence $\langle 2, 0, 1, 2, 3 \rangle$ is a walk of length 4 from vertex 2 to vertex 3. It is also a trail since no edges are repeated. It is not a path though since vertex 2 is repeated.
- The sequence $\langle 3, 2, 1, 0 \rangle$ is a walk of length 3 from vertex 3 to vertex 0. It is also a trail and a path.
- This graph is connected.

6 Adjacency Matrices

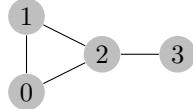
A graph can be represented by an adjacency matrix. For a graph G with V vertices the adjacency matrix A will be $V \times V$ as explained below for various types of graphs.

Note 6.0.1. When we work with matrices in mathematics it's traditional to index from 1 upwards but in computer science it's traditional to index from 0 upwards. I'm going to stick with the computer science approach here so when we discuss an $m \times n$ matrix we really mean an $m \times n$ array A indexed as $AM[i][j]$ with $0 \leq i \leq m - 1$ and $0 \leq j \leq n - 1$. This may seem a little weird from a mathematics perspective but it makes the coding significantly easier.

1. If the graph is simple, unweighted and undirected then:

$$am_{ij} = \begin{cases} 1 & \text{if vertices } i \text{ and } j \text{ are connected by an edge} \\ 0 & \text{if not} \end{cases}$$

Example 6.1. The graph from the beginning:



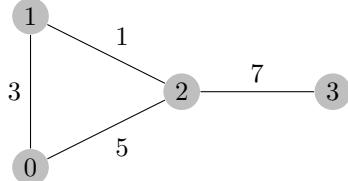
has adjacency matrix:

$$AM = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

2. If the graph is simple and weighted, but not directed, then:

$$am_{ij} = \begin{cases} w & \text{if vertices } i \text{ and } j \text{ are connected by an edge with weight } w \\ 0 & \text{if not} \end{cases}$$

Example 6.2. The following graph:



has adjacency matrix:

$$AM = \begin{bmatrix} 0 & 3 & 5 & 0 \\ 3 & 0 & 1 & 0 \\ 5 & 1 & 0 & 7 \\ 0 & 0 & 7 & 0 \end{bmatrix}$$

Note 6.0.2. This is not comprehensive.

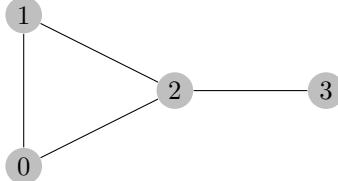
7 Adjacency Lists

While using the adjacency matrix AM is convenient it is not always the best option. Another option is to store an adjacency list.

Definition 7.0.1. The *adjacency list* of a graph G is the list AL such that $AL[i]$ contains a list of vertices adjacent to i .

Note 7.0.1. An adjacency list is a list of lists.

Example 7.1. The following graph:



has adjacency list:

$$AL = [[1, 2], [0, 2], [0, 1, 3], [2]]$$

Note 7.0.2. We can generalize the notion of an adjacency list to a weighted graph by storing $(j, w) \in AL[i]$ if there is an edge from i to j with weight w .

8 Considerations

8.1 Storage

Observe the following:

- The adjacency matrix AM will always require V^2 values.
- The adjacency list AL will vary in accordance with the number of edges.

8.2 Access

Observe the following. If we are focusing on a particular vertex x :

- The adjacency matrix AM will require a scan of V iterations (either row x or column x) to figure out which other vertices x is connected to.
- The adjacency list AL allows us to immediately access a list of the other vertices x is connected to.

8.3 Math

Observe the following. If we are focusing on a particular vertex x :

- The adjacency matrix AM is a convenient structure which we can work with mathematically.
- The adjacency list isn't.

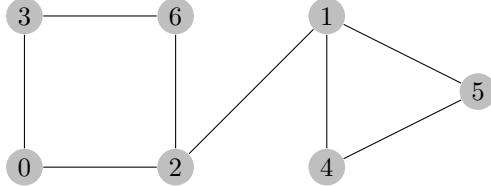
9 The Degree and Other Matrices for a Graph

Definition 9.0.1. For a graph G with n vertices the degree matrix for G is the $V \times V$ diagonal matrix D such that d_{ii} equals the degree of vertex i .

There are many other matrices associated to a graph. One of the most useful is the Laplacian Matrix $L = D - AM$ which is used to study the connectivity of a graph.

10 Thoughts, Problems, Ideas

- Provide the adjacency matrix and adjacency list for the graph shown here:



- Suppose a graph G has adjacency list:

$$AL = [[7], [2, 4], [1, 4], [6, 7], [1, 2, 6, 7], [], [3, 4], [0, 3, 5]]$$

Draw a picture of G and write down its adjacency matrix.

- For a graph G with n vertices write the pseudocode for an algorithm which takes the adjacency matrix AM for G and produces the corresponding adjacency list AL . What is the $\Theta(n)$ time complexity of this? Assume it takes $\Theta(1)$ time to allocate any fixed size list and array full of zeros.
- For a graph G with n vertices write the pseudocode for an algorithm which will takes the adjacency list AL for G and produce the corresponding adjacency matrix AM . What is the Θ time complexity of this? Assume it takes $\Theta(1)$ time to allocate any fixed size list and array full of zeros.
- Given the adjacency matrix AM for a graph with n vertices and a list V of k vertices, write the pseudocode for an algorithm which would determine whether V specifies a walk, returning either **TRUE** or **FALSE**. What is the best- and worst-case Θ time complexity for this?
- Given the adjacency matrix AM for a graph with n vertices and a list V of k vertices, write the pseudocode for an algorithm which would determine whether V specifies a trail, returning either **TRUE** or **FALSE**. What is the best- and worst-case Θ time complexity for this?
- Given the adjacency matrix AM for a graph with n vertices and a list V of k vertices, write the pseudocode for an algorithm which would determine whether V specifies a path, returning either **TRUE** or **FALSE**. What is the best- and worst-case Θ time complexity for this?
- Given the adjacency list AL for a graph with n vertices and a list V of k vertices, write the pseudocode for an algorithm which would determine whether V specifies a walk, returning either **TRUE** or **FALSE**. What is the best- and worst-case Θ time complexity for this?
- Given the adjacency list AL for a graph with n vertices and a list V of k vertices, write the pseudocode for an algorithm which would determine whether V specifies a trail, returning either **TRUE** or **FALSE**. What is the best- and worst-case Θ time complexity for this?
- Given the adjacency list AL for a graph with n vertices and a list V of k vertices, write the pseudocode for an algorithm which would determine whether V specifies a path, returning either **TRUE** or **FALSE**. What is the best- and worst-case Θ time complexity for this?

- Given the adjacency list AL for a graph with n vertices and a list V of

k vertices, write the pseudocode for an algorithm which would determine whether V specifies a path, returning either **TRUE** or **FALSE**. What is the best- and worst-case Θ time complexity for this?

CMSC 351: Shortest Path Algorithm

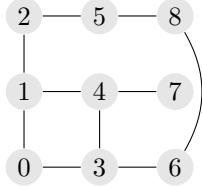
Justin Wyss-Gallifent

July 30, 2024

1	Introduction	2
2	Algorithm Outline	2
3	Detailed Example	2
4	Pseudocode	5
5	Pseudocode Time Complexity	6
6	Pseudocode Auxiliary Space	7
7	Removing the Target	7
8	Thoughts, Problems, Ideas	8
9	Python Test and Output	9

1 Introduction

Consider the following graph:



Suppose we wish to find the shortest path between two vertices s and t , say $s = 0$ and $t = 7$. How can we go about this?

2 Algorithm Outline

Intuitively the idea is this: First assign all vertices with a (tentative distance) value of ∞ and with no predecessor. Technically we could assign them `NULL` instead but ∞ allows us to think of it as a distance that will be computed as the algorithm progresses.

Then we assign the starting vertex s as having distance 0 from itself and we put s on a queue of vertices which need to be dealt with.

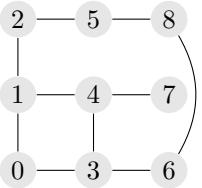
Next we repeat the following until t gets a distance assigned:

- Dequeue a vertex x , for each ∞ vertex y that x is connected to, assign y with x 's distance plus 1 and additionally label it as having x as a predecessor, then put y on the queue.

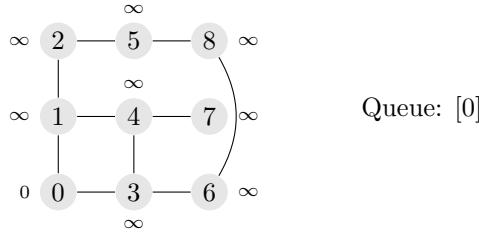
Once we finally assign t a distance we can use the predecessor labels to find the shortest path back to the origin.

3 Detailed Example

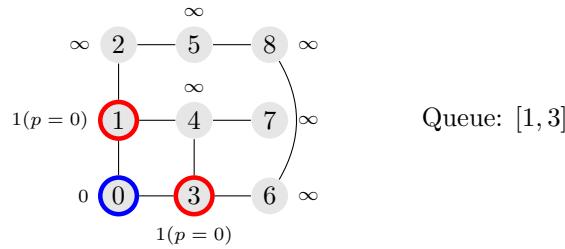
Example 3.1. Let's see how this works on a really easy graph. In the following suppose we wish to find the shortest path path from vertex $s = 0$ to vertex $t = 7$:



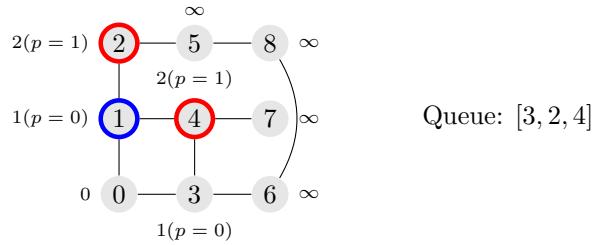
The first thing we do is assign all the vertices with a distance of ∞ except for $s = 0$ itself which we assign a distance of 0. We also initialize the queue.



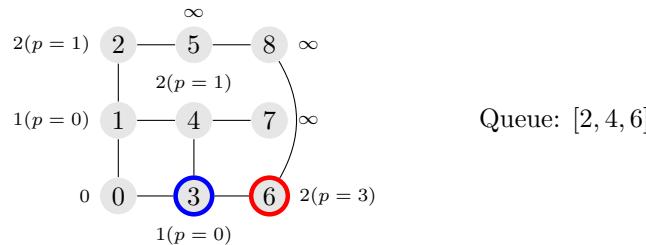
We then dequeue 0 and we go over all the ∞ vertices adjacent to 0 (those are 1,3) and we assign those distance $0 + 1 = 1$ and label them as having predecessor 0. We note that vertex 7 is not amongst them. We also enqueue them.



We then dequeue 1 and we go over all the ∞ vertices adjacent to 1 (those are 2,4) and we assign those distance $1 + 1 = 2$ and label them as having predecessor 1. We note that 7 is not amongst them. We also enqueue them.

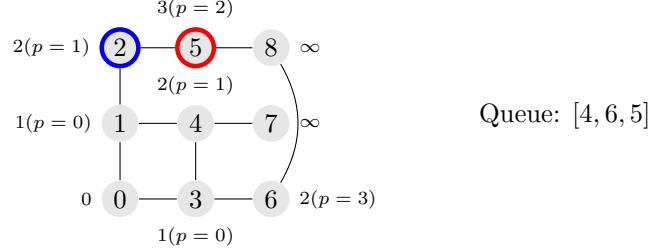


We then dequeue 3 and we go over all the ∞ vertices adjacent to 3 (that is 6) and we assign that distance $1 + 1 = 2$ and label it as having predecessor 3. We note that 7 is not amongst them. We also enqueue it.

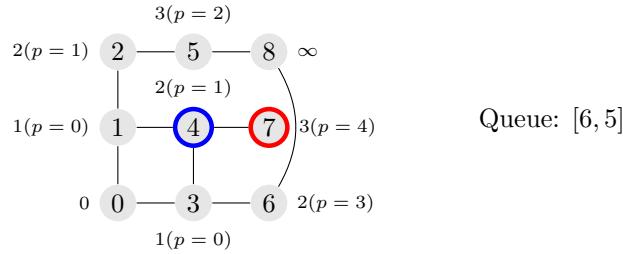


We then dequeue 2 and we go over all the ∞ vertices adjacent to 2 (that is

5) and we assign that distance $2 + 1 = 2$ and label it as having predecessor 2. We note that 7 is not amongst them. We also enqueue it.



We then dequeue 4 and we go over all the ∞ vertices adjacent to 4 (that is 7) and we assign that distance $2 + 1 = 3$ and label it as having predecessor 4. We note that 7 is amongst them so we return it and are done. The pseudocode does not enqueue it onto the queue here because the `enqueue` happens after the `return` conditional.



Since we've located vertex 7 we can backtrack using the predecessors. The predecessors go 4, 1, 0 so the path is $\langle 0, 1, 4, 7 \rangle$ and has length 3.

Note that the predecessors can be stored as a list as:

$$P = [\text{NULL}, 0, 1, 0, 1, 2, 3, 4, \text{NULL}]$$

We extract the path by observing that:

$$P[7] = 4, P[4] = 1, P[1] = 0$$

4 Pseudocode

Here is the pseudocode. The distance and predecessor data is stored in lists (if each vertex is an object these can instead just be properties of the object). The list of vertices to deal with next is stored in a queue.

```
function shortestpath(G,s,t)
    dist = distance array of size V full of inf
    pred = predecessor array of size V full of NULL
    Q = empty queue
    dist[s] = 0
    Q.enqueue(s)
    while Q is nonempty
        x = Q.dequeue
        for each infinity vertex y adjacent to x
            dist[y] = dist[x] + 1
            pred[y] = x
            if y == t
                return(pred)
            end
            Q.enqueue(y)
        end
    end
end
```

Note that some pseudocode you'll see for this include an array which keeps track of whether each vertex has been visited or not. This isn't necessary since this can be determined based upon whether or not a vertex has a finite distance assigned.

5 Pseudocode Time Complexity

The best-case is easy, if s and t are connected by an edge and if t is the first vertex visited from s then assuming that it takes $\Theta(V)$ to initialize `dist` and `pred` then the total time is $\Theta(V)$

The worst-case is not as obvious.

- There is $\Theta(V)$ time required inside the function but excluding the while loop.
- The while loop could iterate as many $V - 1$ times. and the body of the while loop, excluding the for loop, takes constant time. This then a total of $\Theta(V)$.
- Over the course of the entire algorithm the body of the for loop iterates twice for each edge, taking constant time for each, so that's $\Theta(2E) = \Theta(E)$.

Thus we can say for certain that in the worst-case:

$$T(V, E) = \Theta(V + E)$$

In addition note that for an undirected simple graph we have $E \leq C(V, 2) = \frac{1}{2}V(V - 1)$ since at most each pair of vertices may be connected by an edge.

Thus we could also say that $T(V) = \mathcal{O}(V^2)$.

Again, just to really reiterate, this is assuming that we have the graph stored as an adjacency list.

6 Pseudocode Auxiliary Space

The auxiliary space is $\Theta(V)$ to store `dist` and `pred`, each of which have length V , to store `Q`, which has length less than V , and to store a couple of extra variables.

7 Removing the Target

A classic modification of this is to remove the target t and allow the code to simply run until the queue is empty and return the `pred` list afterwards. Here is the pseudocode in brief, with no comments:

```
function shortestpath(G,s)
    dist = distance array of size n full of inf
    pred = predecessor array of size n full of null
    Q = empty queue
    dist[s] = 0
    Q.enqueue(s)
    while Q is nonempty
        x = Q.dequeue
        for each vertex y connected to x
            if y has a label of infinity
                dist[y] = dist[x] + 1
                pred[y] = x
                Q.enqueue(y)
        end
    end
    return(pred)
end
```

The `pred` list will then return a list of predecessors which implicitly yields the shortest paths from s to each and every vertex in the graph.

8 Thoughts, Problems, Ideas

1. Is it more advantageous to have a graph G represented by an adjacency matrix or an adjacency list in order to implement this pseudocode algorithm in actual code? Explain.
2. Modify the pseudocode so that it returns the length of the shortest path from s to t .
3. This algorithm may be modified to find a shortest path tree by not targeting a specific vertex but rather proceeding until all vertices have been accounted for. In this case the predecessor list which is returned can be used to reconstruct the entire tree. Give the pseudocode for this algorithm.
4. Suppose the graph were weighted instead of unweighted. If the line

```
dist[y] = dist[x] + 1
```

were replaced by

```
dist[y] = dist[x] + edgeweight(x to y)
```

would this effectively find the shortest total weighted distance from s to t ? If so, explain why. If not, explain why with a graph.

5. Modify the shortest path algorithm so that it finds the shortest path from s to itself (a cycle) and returns FALSE if no such cycle exists.

9 Python Test and Output

The following code is applied to the graph above. This follows the model of the pseudocode and in addition creates and returns a list of the vertices in the order in which they were visited.

```

def shortestpath(AL,n,u,up):
    dist = [float('inf')] * n
    pred = [None] * n
    Q = []
    dist[u] = 0
    print('Dist: '+str(dist))
    Q.append(u)
    print('Queue: '+str(Q))
    while len(Q) != 0:
        x = Q.pop(0)
        print('Pop '+str(x)+ ' and process vertices '+str(AL[x]))
        for y in AL[x]:
            if dist[y] == float('inf'):
                print('__Process: '+str(y))
                dist[y] = dist[x] + 1
                print('__Dist: '+str(dist))
                pred[y] = x
                if y == up:
                    return(dist,pred)
                Q.append(y)
                print('__Queue: '+str(Q))
            else:
                print('__Already done: '+str(y))

AL = [
    [1, 3],
    [0, 2, 4],
    [1, 5],
    [0, 4, 6],
    [1, 3, 7],
    [2, 8],
    [3, 8],
    [4],
    [5, 6]
]
n = 9

u = 0
up = 7
[dist,pred] = shortestpath(AL,n,u,up)

path = []
x = up
while x != None:
    path.append(x)
    x = pred[x]
path.reverse()
print('Path: '+str(path))
print('Length: '+str(len(path)-1))

```

Output:

```
Dist: [0, inf, inf, inf, inf, inf, inf, inf, inf]
Queue: [0]
Dequeue 0 and process vertices [1, 3]
__Process: 1
__Dist: [0, 1, inf, inf, inf, inf, inf, inf, inf]
__Queue: [1]
__Process: 3
__Dist: [0, 1, inf, 1, inf, inf, inf, inf, inf]
__Queue: [1, 3]
Dequeue 1 and process vertices [0, 2, 4]
__Already done: 0
__Process: 2
__Dist: [0, 1, 2, 1, inf, inf, inf, inf, inf]
__Queue: [3, 2]
__Process: 4
__Dist: [0, 1, 2, 1, 2, inf, inf, inf, inf]
__Queue: [3, 2, 4]
Dequeue 3 and process vertices [0, 4, 6]
__Already done: 0
__Already done: 4
__Process: 6
__Dist: [0, 1, 2, 1, 2, inf, 2, inf, inf]
__Queue: [2, 4, 6]
Dequeue 2 and process vertices [1, 5]
__Already done: 1
__Process: 5
__Dist: [0, 1, 2, 1, 2, 3, 2, inf, inf]
__Queue: [4, 6, 5]
Dequeue 4 and process vertices [1, 3, 7]
__Already done: 1
__Already done: 3
__Process: 7
__Dist: [0, 1, 2, 1, 2, 3, 2, 3, inf]
Path: [0, 1, 4, 7]
Length: 3
```

CMSC 351: Breadth-First Traverse

Justin Wyss-Gallifent

November 6, 2023

1	Introduction	2
2	Intuition	2
3	Algorithm	3
4	Working Through an Example	3
5	Pseudocode	7
6	Pseudocode Time Complexity	8
7	Modifying to Search	8
8	Thoughts, Problems, Ideas	9
9	Python Test and Output	10

1 Introduction

Suppose we are given a graph G and a starting vertex s . Suppose we wish to simply traverse the graph in some way looking for a particular value node. We're not interested in minimizing distance or cost or any such thing, we're just interested in the traverse.

2 Intuition

One classic way to go about this is a breadth-first traverse. The idea is that starting with s we check all vertices connected to s first, and then all vertices connected to those, and so on. In this sense we're covering the graph in “layers of increasing distance from s ”. This is the idea of a breadth-first traverse.

Note 2.0.1. The shortest path algorithm basically follows a breadth-first approach.

Note 2.0.2. Breadth-first traversing is more useful if there's a target and we suspect that the target is close to the starting vertex.

Note 2.0.3. Breadth-first traversing is more useful for things like web-crawling when we might want to find all of the closer vertices first and the algorithm may truncate early.

Note 2.0.4. Breadth-first traversing is more useful when we are trying to explore a strongly connected part of a graph, the idea being that we want to explore close to home before venturing too far away.

3 Algorithm

The algorithm for breadth-first traverse starting at a vertex s proceeds as follows:

We first set up:

- A queue $Q = [s]$.
- A boolean list $VISITED$ of length V which indicates whether a vertex has been visited or not and fill it full of F , or 0, except $VISITED[s] = T$.
- A list $VORDER = [s]$ which will contain the vertices in the order we visit them.

We then repeat the following steps until the queue is empty:

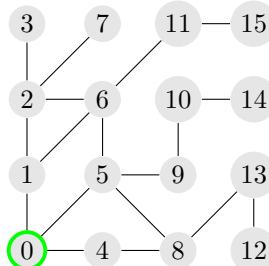
1. $x = Q.dequeue$
2. Find all vertices adjacent to x which have not been visited. For each, put them on Q and update $VISITED$ and $VORDER$.

By convention and consistency when we “find all connected and unvisited vertices” we’ll do it in increasing numerical order.

4 Working Through an Example

In this example we won’t show every single iteration. Instead we’ll only show those as noted.

Example 4.1. Consider the following graph.



Suppose we wish to traverse the graph starting at the node $s = 0$.

Thus for our above example we start with:

$$Q = [0]$$

$$VISITED = [T, F, F]$$

$$VORDER = [0]$$

Iterate! We dequeue $x = 0$. We find all connected and undiscovered vertices

$\{1, 4, 5\}$ those get put onto Q and we update $VISITED$ and $VORDER$:

$Q = [1, 4, 5]$

$VISITED = [T, T, F, F, T, T, F, F, F, F, F, F, F, F, F]$:

$VORDER = [0, 1, 4, 5]$

Iterate! We dequeue $x = 1$, we find all connected and undiscovered vertices $\{2, 6\}$ so those get put onto Q and we update $VISITED$ and $VORDER$:

$Q = [4, 5, 2, 6]$

$VISITED = [T, T, T, F, T, T, T, F, F, F, F, F, F, F, F]$:

$VORDER = [0, 1, 4, 5, 2, 6]$

Iterate! We dequeue $x = 4$, we find all connected and undiscovered vertices $\{8\}$ so those get put onto Q and we update $VISITED$ and $VORDER$:

$Q = [5, 2, 6, 8]$

$VISITED = [T, T, T, F, T, T, T, F, T, F, F, F, F, F, F]$:

$VORDER = [0, 1, 4, 5, 2, 6, 8]$

Iterate! We dequeue $x = 5$, we find all connected and undiscovered vertices $\{9\}$ so those get put onto Q and we update $VISITED$ and $VORDER$:

$Q = [2, 6, 8, 9]$

$VISITED = [T, T, T, F, T, T, T, F, T, T, F, F, F, F, F]$:

$VORDER = [0, 1, 4, 5, 2, 6, 8, 9]$

Iterate! We dequeue $u = 2$, it gives us undiscovered vertices $\{3, 7\}$ so those get put onto Q and we update $VISITED$ and $VORDER$:

$Q = [6, 8, 9, 3, 7]$

$VISITED = [T, T, T, T, T, T, T, T, F, F, F, F, F, F]$:

$VORDER = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7]$

Iterate! We dequeue $u = 6$, we find all connected and undiscovered vertices $\{11\}$ so those get put onto Q and we update $VISITED$ and $VORDER$:

$Q = [8, 9, 3, 7, 11]$

$VISITED = [T, T, T, T, T, T, T, T, T, F, T, F, F, F, F]$:

$VORDER = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11]$

Iterate! We dequeue $u = 8$, we find all connected and undiscovered vertices $\{13\}$ so those get put onto Q and we update $VISITED$ and $VORDER$:

$Q = [9, 3, 7, 11, 13]$

$VISITED = [T, T, T, T, T, T, T, T, T, F, T, F, T, F, F]$:

$VORDER = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11]$

Iterate! We dequeue $u = 9$, we find all connected and undiscovered vertices $\{10\}$ so those get put onto Q and we update $VISITED$ and $VORDER$:

$Q = [3, 7, 11, 13, 10]$

$VISITED = [T, T, T, T, T, T, T, T, T, F, T, F, T, F, F]$:

$VORDER = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 9]$

Iterate! We dequeue $u = 3$, we find all connected and undiscovered vertices $\{\}$.

$$Q = [7, 11, 13, 10]$$

$$VISITED = [T, T, T, T, T, T, T, T, T, T, F, T, F, F]:$$

$$VORDER = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 9]$$

Iterate! We dequeue $u = 7$, we find all connected and undiscovered vertices $\{\}$.

$$Q = [11, 13, 10]$$

$$VISITED = [T, T, T, T, T, T, T, T, T, T, F, T, F, F]:$$

$$VORDER = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 9]$$

Iterate! We dequeue $u = 11$, we find all connected and undiscovered vertices $\{15\}$ so those get put onto Q and we update $VISITED$ and $VORDER$:

$$Q = [13, 10, 15]$$

$$VISITED = [T, T, T, T, T, T, T, T, T, T, F, T, F, T]:$$

$$VORDER = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 9, 15]$$

Iterate! We dequeue $u = 13$, we find all connected and undiscovered vertices $\{12\}$ so those get put onto Q and we update $VISITED$ and $VORDER$:

$$Q = [10, 15, 12]$$

$$VISITED = [T, T, F, T]:$$

$$VORDER = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 9, 15, 12]$$

Iterate! We dequeue $u = 10$, we find all connected and undiscovered vertices $\{14\}$ so those get put onto Q and we update $VISITED$ and $VORDER$:

$$Q = [15, 12, 14]$$

$$VISITED = [T, T, T]:$$

$$VORDER = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 9, 15, 12, 14]$$

Iterate! We dequeue $u = 15$, we find all connected and undiscovered vertices $\{\}$.

$$Q = [12, 14]$$

$$VISITED = [T, T, T]:$$

$$VORDER = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 9, 15, 12, 14]$$

Iterate! We dequeue $u = 12$, we find all connected and undiscovered vertices $\{\}$.

$$Q = [14]$$

$$VISITED = [T, T, T]:$$

$$VORDER = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 9, 15, 12, 14]$$

Iterate! We dequeue $u = 14$, we find all connected and undiscovered vertices $\{\}$.

$$Q = []$$

$$VISITED = [T, T, T]:$$

$VORDER = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 9, 15, 12, 14]$

Since Q is empty we're done.

We return $VORDER = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 13, 10, 15, 12, 14]$.

5 Pseudocode

Here is the pseudocode:

```
function bft(G,s)
    QUEUE = [s]
    VISITED = list of FALSE of length V
    VISITED[s] = TRUE
    VORDER = [s]
    while QUEUE is not empty
        x = QUEUE.dequeue
        for all y adjacent to x
            if VISITED[y] == FALSE
                QUEUE.enqueue(y)
                VISITED[y] = TRUE
                VORDER.append(y)
        end
    end
    return(VORDER)
end
```

6 Pseudocode Time Complexity

Suppose V is the number of vertices and E is the number of edges.

- The initialization takes $\mathcal{O}(V)$. This could in fact take $\Theta(1)$ depending on the architecture but the choice has no effect on the result.
- Each vertex gets enqueued and dequeued exactly once so this is $V \Theta(1)$ each for a total of $\Theta(V)$ for just the `dequeue`, not the `for` loop.
- The body of the `for` loop iterates $2E$ times over the course of the entire algorithm, once for each vertex at each end of the edge. The body takes constant time $\Theta(1)$. so overall this is $\Theta(2E) = \Theta(E)$.

The time complexity is therefore $\mathcal{O}(V) + \Theta(V) + \Theta(E) = \mathcal{O}(V + E)$. If initialization is actually $\Theta(1)$ then this becomes $\Theta(V + E)$.

Note 6.0.1. Note that our pseudocode and analysis assumes we have direct access to a vertex's edges using something like an adjacency list. If we use an adjacency matrix then the inner loop becomes $\Theta(V)$ and the entire pseudocode becomes $\Theta(V^2)$.

Note 6.0.2. There are breadth-first traverses which run in $\mathcal{O}(E \lg V)$ but they require a radically different pseudocode based upon a heap structure instead of a simple list S . This heap structure is what leads to the $\lg V$ factor.

7 Modifying to Search

Breadth-first traverse can be tweaked if there is a target node in mind. How would you tweak the pseudocode to exit as soon as the target was found and how would that change the time complexity?

8 Thoughts, Problems, Ideas

1. Suppose node i is a structure with properties $i.height$, $i.weight$ and $i.volume$. Modify the pseudocode to return the weight of the first node encountered whose weight is more than 100. You may assume such a node exists.
2. Same as above but no such assumption. Return $NULL$ if no such node is found.
3. When s is dequeued all of the vertices connected to s will be newly discovered. This is not true for any other vertex x because the algorithm-parent of x will already be discovered. Under what circumstances, for every other vertex x , would the algorithm-parent be the only previously discovered vertex?
4. Let q_i be the length of Q after the i th iteration of the `while` loop. We'll say $q_0 = 1$ to be comprehensive since $Q = [s]$ when no iterations have completed. So in the example in the notes $q_1 = 3$, $q_3 = 4$, and so on. Of course there is some k such that $q_k = 0$ as this is when the algorithm ends. Moreover in the example q_i initially (nonstrictly) increases and then (nonstrictly) decreases. Must the q_i always follow this pattern? Explain.
5. Building off the previous problem what is the maximum that k might be? How about the minimum? What would a graph look like (qualitatively) if its k -value were somewhere in the middle? Explain using specific examples of graphs.
6. Describe the impact if the graph were given with the adjacency matrix rather than the adjacency list. How would that impact the pseudocode and the time complexity?
7. Modify the pseudocode so that we may pass a `maxdepth` and so that the algorithm will go no further than that depth from the starting vertex.

9 Python Test and Output

The following code is applied to the graph above. This follows the model of the pseudocode and in addition creates and returns a list of the vertices in the order in which they were discovered.

Code:

```
def bfs(EL,n,s):
    Q = [s]
    D = [False] * n
    D[s] = True
    V = [s]
    print('Q = ' + str(Q))
    print('V = ' + str(V))
    #print('D = ' + str(D).replace('True','T').replace('False','F'))
    while len(Q) != 0:
        x = Q.pop(0)
        for y in EL[x]:
            if not D[y]:
                D[y] = True
                V.append(y)
                Q.append(y)
        print('Q = ' + str(Q))
        print('V = ' + str(V))
        #print('D = ' + str(D).replace('True','T').replace('False','F'))
    return(V)
EL = [
    [1, 4, 5],
    [0, 2, 6],
    [1, 3, 6, 7],
    [2],
    [0, 8],
    [0, 6, 8, 9],
    [1, 2, 5, 11],
    [2],
    [4, 5, 13],
    [5, 10],
    [9, 14],
    [6, 15],
    [13],
    [8, 12],
    [10],
    [11]
]
n = 16
```

```
s = 0
visited = bfs(EL,n,s)
print('Discovered = ' + str(visited))
```

Output:

```
Q = [0]
V = [0]
Q = [1, 4, 5]
V = [0, 1, 4, 5]
Q = [4, 5, 2, 6]
V = [0, 1, 4, 5, 2, 6]
Q = [5, 2, 6, 8]
V = [0, 1, 4, 5, 2, 6, 8]
Q = [2, 6, 8, 9]
V = [0, 1, 4, 5, 2, 6, 8, 9]
Q = [6, 8, 9, 3, 7]
V = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7]
Q = [8, 9, 3, 7, 11]
V = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11]
Q = [9, 3, 7, 11, 13]
V = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 13]
Q = [3, 7, 11, 13, 10]
V = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 13, 10]
Q = [7, 11, 13, 10]
V = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 13, 10]
Q = [11, 13, 10]
V = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 13, 10]
Q = [13, 10, 15]
V = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 13, 10, 15]
Q = [10, 15, 12]
V = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 13, 10, 15, 12]
Q = [15, 12, 14]
V = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 13, 10, 15, 12, 14]
Q = [12, 14]
V = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 13, 10, 15, 12, 14]
Q = [14]
V = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 13, 10, 15, 12, 14]
Q = []
V = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 13, 10, 15, 12, 14]
Discovered = [0, 1, 4, 5, 2, 6, 8, 9, 3, 7, 11, 13, 10, 15, 12, 14]
```

CMSC 351: Depth-First Traverse

Justin Wyss-Gallifent

July 28, 2024

1	Introduction:	2
2	Intuition	2
3	Visualization	2
4	Algorithm Implementation	2
5	Recursive Implementation	3
5.1	Pseudocode	3
5.2	Pseudocode Time Complexity	4
6	Stack Implementation	5
6.1	Pseudocode	5
6.2	Pseudocode Time Complexity	7
7	Stack/Doubly-Linked-List Implementation	7
7.1	Introduction	7
7.2	Pseudocode	9
7.3	Pseudocode Time Complexity	11

1 Introduction:

Suppose we are given a graph G and a starting node s . Suppose we wish to simply traverse the graph in some way looking for a particular value associated with a node. We're not interested in minimizing distance or cost or any such thing, we're just interested in the traversal process.

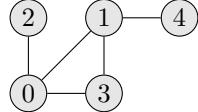
2 Intuition

One classic way to go about this is a depth-first traverse. The intuitive idea is that starting with a starting node s we follow one path as far as possible before backtracking. When we backtrack we only do so as little as possible until we can go deeper again.

Observe that this description does not lead to a unique traversal because there may be multiple paths that we can follow from a given vertex.

3 Visualization

Before writing down some explicit pseudocode let's look at an easy graph and look at how the above intuition might pan out. Consider this graph:



Suppose we start at the vertex $s = 0$. We have three edges we can follow, let's suppose we follow the edge to the vertex 3 first. From 3 we can only go to 1 (we can't go back to 0 since we've visited it already) and then to 4 (we can't go back to 0 or 3). At that point we have gone as deep as we can along that branch so we go back to the most recent branch for which there are other paths available. We have to go back to 0 and from there we go to 2. Thus our depth-first traverse follows the vertices in order 0, 3, 1, 4, 2.

4 Algorithm Implementation

There are several classic approaches to constructing an algorithm for depth-first search to which we will add one more for reasons we shall make clear:

- Using recursion.
- Using a stack.
- Using a stack which is modified to be a doubly-linked-list.

5 Recursive Implementation

5.1 Pseudocode

The pseudocode for the recursive implementation is as follows:

```
// These are global.
VORDER = []
VISITED = list of length V full of FALSE
function dft(G,x):
    VORDER.append(x)
    VISITED[x] = TRUE
    for all adjacent nodes y (in some order)
        if VISITED[y] == FALSE
            dft(G,y)
        end if
    end for
end function
dft(G,s)
```

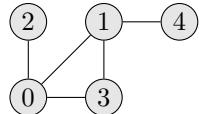
In this pseudocode the list `VORDER` will record the order in which we visit the vertices while the list `VISITED` will indicate whether or not a vertex has been visited.

The line:

```
for all adjacent nodes y (in some order)
```

does not suggest which order we should follow the adjacent nodes in. In what follows we'll follow them in decreasing order.

Example 5.1. Let's return to our example from earlier:



We'll start our traversal at the vertex $x = 0$. Before the function is called we have:

index	0	1	2	3	4
VORDER					
VISITED	F	F	F	F	F

Our first call is `dft(G,0)` and before the `for` loop we then have:

index	0	1	2	3	4
VORDER	0				
VISITED	T	F	F	F	F

The **for** loop cycles through the vertices 3,2,1 (call this depth 1) and the first call is `dft(G,3)` which yields, before its own **for** loop:

index	0	1	2	3	4
VORDER	0	3			
VISITED	T	F	F	T	F

From `dft(G,3)` the **for** loop cycles through the vertices 1,0 (call this depth 2) and the first call is `dft(G,1)` which yields, before its own **for** loop:

index	0	1	2	3	4
VORDER	0	3	1		
VISITED	T	T	F	T	F

From `dft(G,1)` the **for** loop cycles through the vertices 4,0 (call this depth 3) and the first call is `dft(G,4)` which yields, before its own **for** loop:

index	0	1	2	3	4
VORDER	0	3	1	4	
VISITED	T	T	F	T	T

From `dft(G,4)` the **for** loop cycles through the vertices 1 (call this depth 4) but since that vertex has been visited, `dft` is not called on it again and we are sent back to depth 3 and our loop is on vertex 0 but since that vertex has been visited, `dft` is not called on it again and we are sent back to depth 2 and our loop is on vertex 0 but since that vertex has been visited, `dft` is not called on it again and we are sent back to depth 1 and our loop is on vertex 2 so we call `dft(G,2)` which yields, before its own **for** loop:

index	0	1	2	3	4
VORDER	0	3	1	4	2
VISITED	T	T	T	T	T

From `dft(G,2)` the **for** loop cycles through the vertices 0 but since that vertex has been visited, `dft` is not called on it again and we are sent back to depth 1 and our loop is on vertex 1 but since that vertex has been visited, `dft` is not called on it again.

Then we are done. Observe that the order in which we visited the nodes is 0, 3, 1, 4, 2.

5.2 Pseudocode Time Complexity

Suppose V is the number of nodes and E is the number of edges. What follows is exactly the same as breadth-first traverse so if that made sense you can possibly skip this.

- The initialization takes $\mathcal{O}(V)$. This could in fact take $\Theta(1)$ depending on the architecture but the choice has no effect on the result.
- Each node gets processed once so this is $V\Theta(1)$ each for a total of $\Theta(V)$.

- Since each edge is attached to two nodes the `for` loop will iterate a total of $2E$ times over the course of the entire algorithm. This gives a total of $\Theta(2E) = \Theta(E)$.

The time complexity is therefore $\mathcal{O}(V) + \Theta(V) + \Theta(E) = \mathcal{O}(V + E)$. If initialization is actually $\Theta(1)$ then this becomes $\Theta(V + E)$.

Note 5.2.1. Note that our pseudocode and analysis assumes we have direct access to a node's edges using something like an adjacency list. If we use an adjacency matrix then the inner loop becomes $\Theta(V)$ and the entire pseudocode becomes $\Theta(V^2)$.

6 Stack Implementation

6.1 Pseudocode

The pseudocode for the stack implementation is as follows:

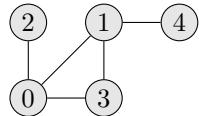
```
VORDER = []
VISITED = list of length V full of FALSE
STACK = [s]
while STACK is not empty:
    x = STACK.pop()
    if VISITED[x] == FALSE:
        VISITED[x] = TRUE
        VORDER.append(x)
    end if
    for all nodes y adjacent to x:
        if VISITED[y] == FALSE:
            STACK.append(y)
        end if
    end for
end while
```

The line:

for all nodes y adjacent to x

does not suggest which order we should follow the adjacent nodes in. In what follows we'll follow them in increasing order.

Example 6.1. Let's return to our example from earlier:



We'll start our traversal at the vertex $x = 0$. We start with the following

before the `while` loop:

	S = [0]				
index	0	1	2	3	4
VORDER					
VISITED	F	F	F	F	F

Iterate! We pop $x = 0$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertices 1, 2, 3 and push them all onto the stack:

	S = [1,2,3]				
index	0	1	2	3	4
VORDER	0				
VISITED	T	F	F	F	F

Iterate! We pop $x = 3$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertices 0, 1 and push 1 (but not 0) onto the stack:

	S = [1,2,1]				
index	0	1	2	3	4
VORDER	0	3			
VISITED	T	F	F	T	F

Iterate! We pop $x = 1$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertices 0, 3, 4 and push 4 (but not 0, 3) onto the stack:

	S = [1,2,4]				
index	0	1	2	3	4
VORDER	0	3	1		
VISITED	T	T	F	T	F

Iterate! We pop $x = 4$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertex 1 but nothing is pushed onto the stack:

	S = [1,2]				
index	0	1	2	3	4
VORDER	0	3	1	4	
VISITED	T	T	F	T	T

Iterate! We pop $x = 2$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertex 0 but nothing is pushed onto the stack:

$$S = [1]$$

index	0	1	2	3	4
VORDER	0	3	1	4	2
VISITED	T	T	T	T	T

Iterate! We pop $x = 1$ off the stack. Since it's visited we do nothing with it. We then iterate over the vertices 0, 3, 4 but nothing is pushed onto the stack:

S = []					
index	0	1	2	3	4
VORDER	0	3	1	4	2
VISITED	T	T	T	T	T

Then we are done. Observe that the order in which we visited the nodes is 0, 3, 1, 4, 2.

6.2 Pseudocode Time Complexity

It's not uncommon for various online sources to give essentially this pseudocode and state that it has time complexity $\Theta(V + E)$ but this is false as can be easily demonstrated.

Consider a graph with V vertices which is *complete*, meaning each vertex is connected to every other vertex. We start by pushing the starting vertex onto the stack.

When we pop this vertex there is 1 visited vertex and the **for** loop will push all of the remaining $V - 1$ (unvisited) vertices onto the stack.

When we pop the next vertex there are 2 visited vertices and the **for** loop will push $V - 2$ (unvisited) vertices onto the stack, all of which will be repeats.

This will repeat through the entire process and all together the number of vertices which get pushed onto the stack will be:

$$1 + (V - 1) + (V - 2) + \dots + 2 + 1 + 0 = 1 + \frac{(V - 1)V}{2} = \Theta(V^2)$$

Since the **while** loop iterates once for each vertex on the stack it will iterate $\Theta(V^2)$ times.

The **for** loop will (as with the argument for the recursive implementation) iterate $2E$ times for a final time complexity of $\Theta(V^2 + E)$.

7 Stack/Doubly-Linked-List Implementation

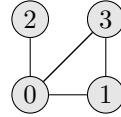
7.1 Introduction

It is possible to modify the stack version to bring it down to $\Theta(V + E)$. The trick is to find a way to ensure that an item is only ever popped off the stack

once, but we can't be sloppy about it. In theory there are two ways to go about this. When we are about to push something onto the stack, other than checking if it has been visited:

- (a) We check whether it has been or still is on the stack and if so, we don't push it.
- (b) We check whether it has been on the stack, we don't push it, and if it is on the stack, we remove the earlier occurrence.

While (a) seems easier to do it does not work, as is easily demonstrated by this graph:



Examine approach (a). Let's start at 0, so $S = [0]$. We then pop $x = 0$ (marking it as visited) and suppose we push 3, 2, 1 onto the stack in that order, so $S = [3, 2, 1]$. We then pop $x = 1$ (marking it as visited) and don't push anything, so $S = [3, 2]$. We then pop $x = 2$ (marking it as visited) and don't push anything, so $S = [3]$. We then pop $x = 3$ (marking it as visited), and don't push anything, so $S = []$, and then we are done. However we have now visited the vertices in the order 0, 1, 2, 3 which is not a BFT since after visiting 1 we should go to 3.

On the other hand examine approach (b). Let's start at 0, so $S = [0]$. We then pop $x = 0$ (marking it as visited) and suppose we push 3, 2, 1 onto the stack in that order, so $S = [3, 2, 1]$. We then pop $x = 1$ (marking it as visited) and we push 3, replacing the earlier occurrence, so $S = [2, 3]$. We then pop $x = 3$ (marking it as visited) and we push nothing, so $S = [2]$. We then pop $x = 2$ (marking it as visited) and don't push anything, so $S = []$, and then we are done. Now we have visited the vertices in the order 0, 1, 3, 2.

One way to accomplish (b) is to turn the stack into a doubly-linked-list and to keep an array of pointers, one for each vertex, which point to the vertex's location on the stack and are NULL by default. When a vertex is pushed onto the stack we check if its pointer is NULL and if not then we remove the previous vertex from within the stack (this is $\Theta(1)$ for a doubly-linked-list) and then push it on the end of the stack and update the pointer.

7.2 Pseudocode

The pseudocode for this new implementation is as follows:

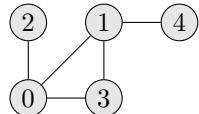
```
VORDER = []
STACK = [s] (functions as doubly-linked-list)
VISITED = list of length V full of FALSE
SP = list of length V full of NULL pointers
SP[x] = points to x in STACK
while STACK is not empty:
    x = STACK.pop()
    if VISITED[x] == FALSE:
        VISITED[x] = TRUE
        VORDER.append(x)
    end if
    for all nodes y adjacent to x:
        if VISITED[y] == FALSE:
            if SP[y] != NULL:
                delete y from STACK
            end if
            STACK.append(y)
            SP[y] = point to y in STACK
        end if
    end for
end while
```

The line:

```
for all adjacent nodes y (in some order)
```

does not suggest which order we should follow the adjacent nodes in. In what follows we'll follow them in increasing order.

Example 7.1. Let's return to our example from earlier:



Illustrating the pointers is a bit of a pain so we will avoid doing so. Instead the critical difference between this implementation and the stack implementation is that whenever a vertex is pushed onto the stack we check if it is already on the stack and if so we delete its earlier occurrence. In what follows this is the only significant difference.

We'll start our traversal at the vertex $x = 0$. We start with the following before the `while` loop:

S = [0]					
index	0	1	2	3	4
VORDER					
VISITED	F	F	F	F	F

Iterate! We pop $x = 0$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertices 1, 2, 3 and push them all onto the stack:

S = [1,2,3]					
index	0	1	2	3	4
VORDER	0				
VISITED	T	F	F	F	F

Iterate! We pop $x = 3$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertices 0, 1 and push 1 (but not 0) onto the stack which deletes the earlier 1:

S = [2,1]					
index	0	1	2	3	4
VORDER	0	3			
VISITED	T	F	F	T	F

Iterate! We pop $x = 1$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertices 0, 3, 4 and push 4 (but not 0, 3) onto the stack:

S = [2,4]					
index	0	1	2	3	4
VORDER	0	3	1		
VISITED	T	T	F	T	F

Iterate! We pop $x = 4$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertex 1 but nothing is pushed onto the stack:

S = [2]					
index	0	1	2	3	4
VORDER	0	3	1	4	
VISITED	T	T	F	T	T

Iterate! We pop $x = 2$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertex 0 but nothing is pushed onto the stack:

$$S = []$$

index	0	1	2	3	4
VORDER	0	3	1	4	2
VISITED	T	T	T	T	T

Then we are done. Observe that the order in which we visited the nodes is 0, 3, 1, 4, 2.

7.3 Pseudocode Time Complexity

With this modification each vertex is popped exactly once and so the `while` loop iterates V times. The `for` loop iterates $2E$ times and we then we have a total time complexity of $\Theta(V + E)$ once more.

CMSC 351: Dijkstra's Algorithm

Justin Wyss-Gallifent

November 8, 2023

1	Introduction	2
2	Algorithm	2
3	Working Through an Example	3
4	Rudimentary Pseudocode	9
5	Rudimentary Pseudocode Time Complexity	9
6	Elegant Pseudocode	10
7	Elegant Pseudocode Time Complexity	11
8	Mathematical Proof that it Works	12
9	Thoughts, Problems, Ideas	15
10	Python Test and Output	16

1 Introduction

Dijkstra's Algorithm is essentially an extension of the shortest path algorithm in which the graph is weighted. In this case instead of looking for a shortest path we are looking for a path of minimal weight.

What we'll actually do is better, we'll find a shortest weight tree which is a tree that is a subgraph of the graph such that, treating the starting vertex as the root, explicitly shows us how to get to every other vertex with minimal weight.

2 Algorithm

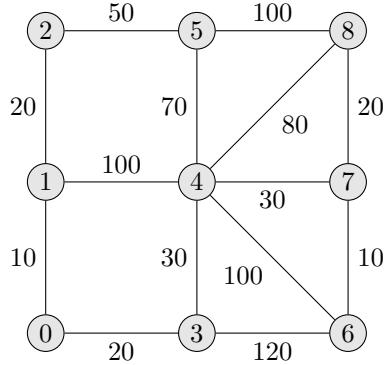
Starting with a weighted, undirected simple graph and a starting vertex s , Dijkstra's Algorithm proceeds as follows. Here the graph has V vertices and $w(x, y)$ is the weight of the edge from x to y .

- (a) Create a set $S = \{\}$.
- (b) Create a distance array d of length V consisting of all ∞ except set $d[s] = 0$.
- (c) Create a predecessor array p of length V consisting of all *NULL*.
- (d) Pick a vertex x with minimal distance which is not already in S . Add it to S . For all vertices y adjacent to x , if $d[x] + w(x, y) < d[y]$ then assign $d[y] = d[x] + w(x, y)$ and set $p[y] = x$.
- (e) Repeat step (d) until S contains all vertices, meaning we can go no further.

Note 2.0.1. In step (d) there may be more than one option for x . In such a case we may pick any of them.

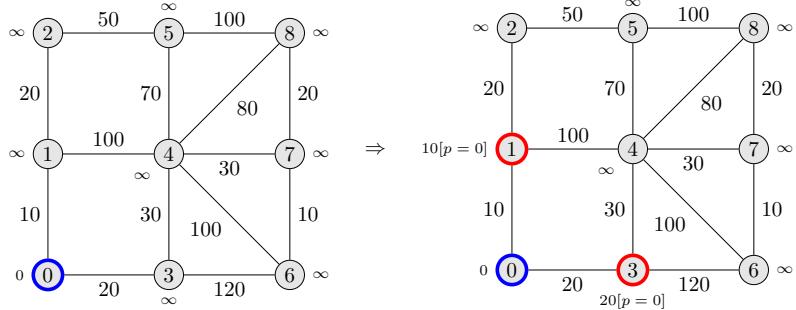
3 Working Through an Example

Example 3.1. Consider the following graph:



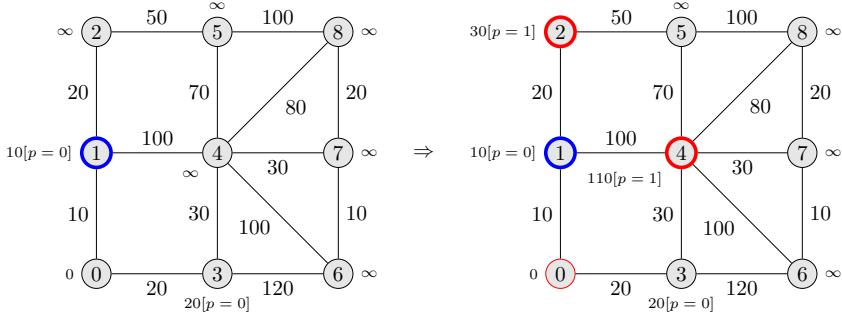
Suppose we choose 0 to be our starting vertex. We set S and P as instructed.

Iterate! We look at the graph and choose the vertex with minimum assigned distance which is not already in $S = \{\}$. This is of course currently vertex 0 with distance 0. so we look at all vertices connected to vertex 0 and assign their distances as vertex 0's distance plus the edge weight. We only do this if this value is smaller than their current weight but since their current weight is ∞ then of course we replace it. We also set those vertices' predecessor to be 0.



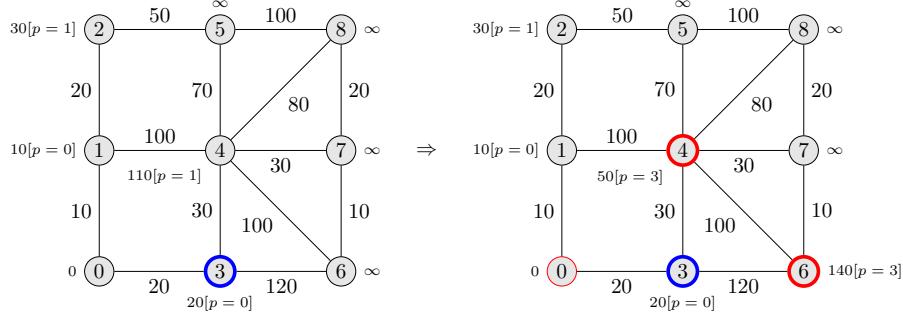
We put this vertex in S , so $S = \{0\}$

Iterate! We look at the graph and choose the vertex with minimum assigned distance which is not already in $S = \{0\}$. This is of course vertex 1 with distance 10. Then we look at all vertices connected to vertex 1 and assign their distances as vertex 1's distance plus the edge weight. We only do this if this value is smaller than their current weight and if so we also set those vertices' predecessor to be 1.



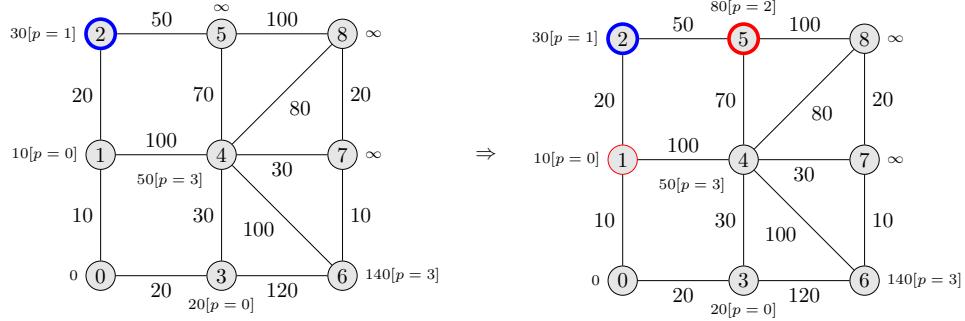
We put this vertex in S , so $S = \{0, 1\}$.

Iterate! We look at the graph and choose the vertex with minimum assigned distance which is not already in $S = \{0, 1\}$. This is of course vertex 3 with distance 20. Then we look at all vertices connected to vertex 3 and assign their distances as vertex 3's distance plus the edge weight. We only do this if this value is smaller than their current weight and if so we also set those vertices' predecessor to be 3. Note that vertex 4 gets its weight reassigned because it was 110 but from vertex 3 it's $20 + 30 = 50 < 110$.



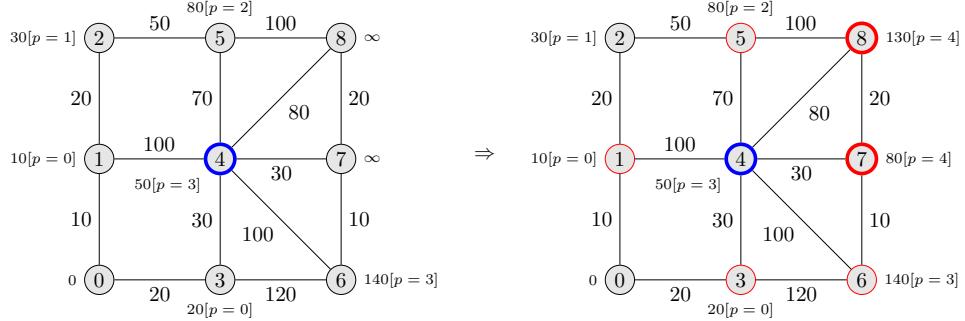
We put this vertex in S , so $S = \{0, 1, 3\}$.

Iterate! We look at the graph and choose the vertex with minimum assigned distance which is not already in $S = \{0, 1, 3\}$. This is of course vertex 2 with distance 30. Then we look at all vertices connected to vertex 2 and assign their distances as vertex 2's distance plus the edge weight. We only do this if this value is smaller than their current weight and if so we also set those vertices' predecessor to be 2.



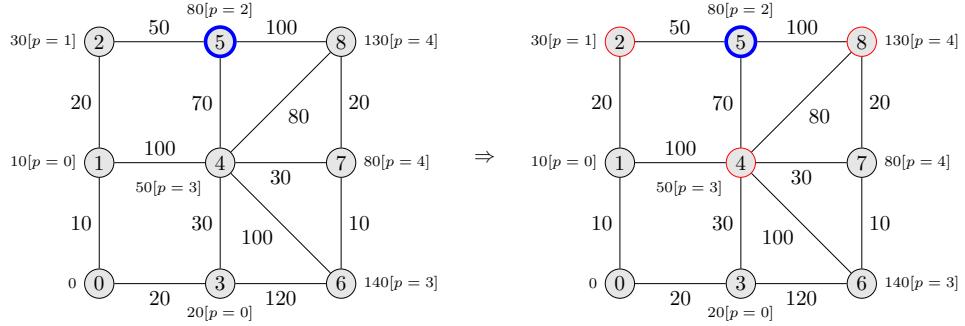
We put this vertex in S , so $S = \{0, 1, 3, 2\}$.

Iterate! we look at the graph and choose the vertex with minimum assigned distance which is not already in $S = \{0, 1, 3, 2\}$. This is of course vertex 4 with distance 50 Then we look at all vertices connected to vertex 4 and assign their distances as vertex 4's distance plus the edge weight. We only do this if this value is smaller than their current weight and if so we also set those vertices' predecessor to be 4.



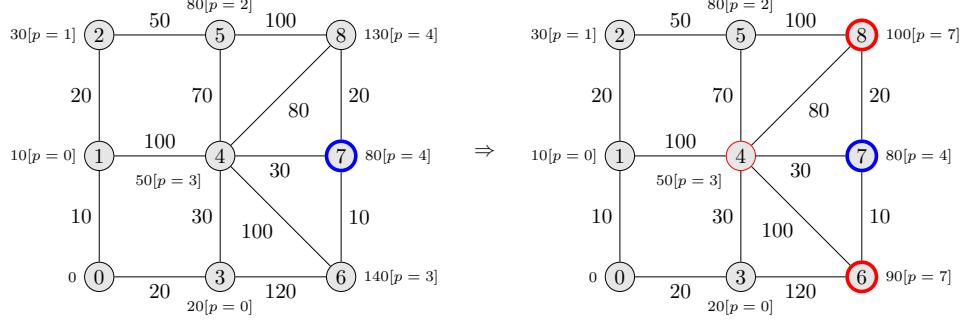
We put this vertex in S , so $S = \{0, 1, 3, 2, 4\}$

Iterate! We look at the graph and choose the vertex with minimum assigned distance which is not already in $S = \{0, 1, 3, 2, 4\}$. Both vertices 5 and 7 work so we can pick either. Let's choose vertex 5 with distance 80. Then we look at all vertices connected to vertex 5 and assign their distances as vertex 5's distance plus the edge weight. We only do this if this value is smaller than their current weight and if so we also set those vertices' predecessor to be 5. Here there are no changes.



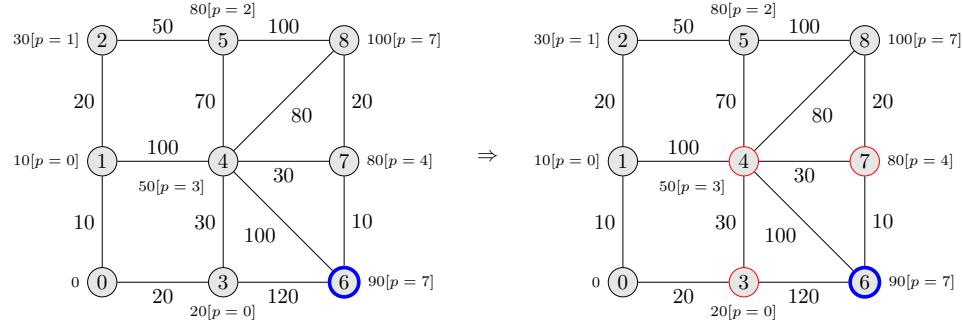
We put this vertex in S , so $S = \{0, 1, 3, 2, 4, 5\}$.

Iterate! We look at the graph and choose the vertex with minimum assigned distance which is not already in $S = \{0, 1, 3, 2, 4, 5\}$. This is vertex 7 with distance 80. Then we look at all vertices connected to vertex 7 and assign their distances as vertex 7's distance plus the edge weight. We only do this if this value is smaller than their current weight and if so we also set those vertices' predecessor to be 7. Here there are no changes.



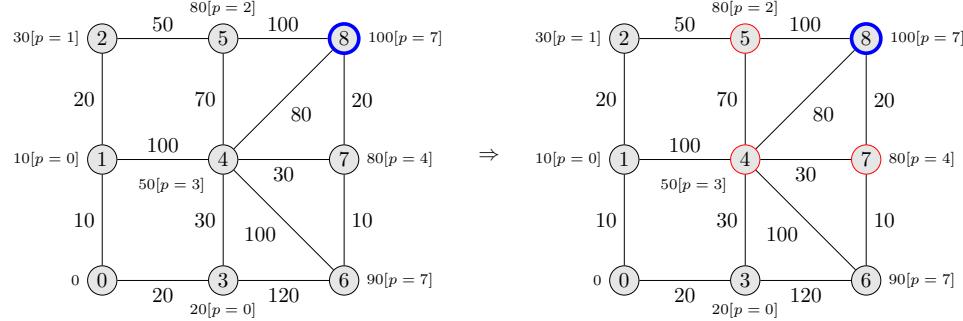
We put this vertex in S , so $S = \{0, 1, 3, 2, 4, 5, 7\}$.

Iterate! We look at the graph and choose the vertex with minimum assigned distance which is not already in $S = \{0, 1, 3, 2, 4, 5, 7\}$. This is vertex 6 with distance 90. Then we look at all vertices connected to vertex 6 and assign their distances as vertex 6's distance plus the edge weight. We only do this if this value is smaller than their current weight and if so we also set those vertices' predecessor to be 6. Here there are no changes.



We put this vertex in S , so $S = \{0, 1, 3, 2, 4, 5, 7, 6\}$.

Iterate! We look at the graph and choose the vertex with minimum assigned distance which is not already in $S = \{0, 1, 3, 2, 4, 5, 7, 6\}$. This is vertex 8 with distance 100. Then we look at all vertices connected to vertex 8 and assign their distances as vertex 8's distance plus the edge weight. We only do this if this value is smaller than their current weight and if so we also set those vertices' predecessor to be 8. Here there are no changes.



We put this vertex in S , so $S = \{0, 1, 3, 2, 4, 5, 7, 6, 8\}$

Now S contains every vertex and we are done.

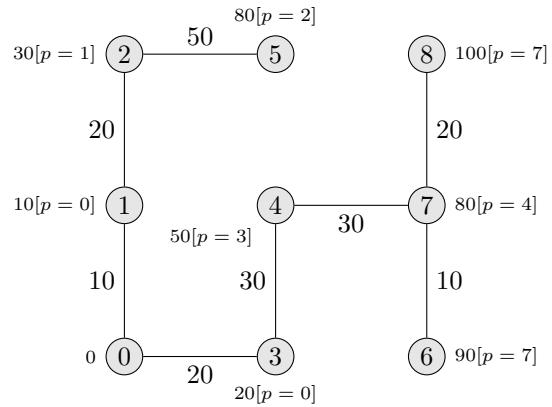
Our array of predecessors is $P = [\text{NULL}, 0, 1, 0, 3, 2, 7, 4, 7]$ and this tells us how to construct our tree in the sense that the predecessor of each vertex is its parent and this tells us the edges. For example $P[0] = \text{NULL}$ because there is no predecessor of 0 as it is the root vertex, $P[1] = 0$ and so we need to have the edge $(1, 0)$, and so on.

In other words this array gives us the set of edges:

$$(1, 0), (2, 1), (3, 0), (4, 3), (5, 2), (6, 7), (7, 4), (8, 7)$$

In the graph we keep only those edges we get the following. The labels show the minimum weight path back to vertex 0 and the brackets show the

predecessors still.



4 Rudimentary Pseudocode

Here is the pseudocode for a very rudimentary implementation. This code returns an array `pred` with the property that `pred[v]` gives the predecessor of the vertex `v` in the minimal weight tree.

```
\\" PRE: G is a graph with V vertices.
\\" PRE: s is the starting vertex.
def dijkstra(G,start):
    dist = [inf,...,inf] of length V.
    pred = [NULL,...,NULL] of length V.
    S = []
    dist[start] = 0
    while length(S) != V
        x = vertex in G-S with smallest distance
        for each vertex y connected to x
            if dist[x] + (Weight of Edge x,y) < dist[y]:
                dist[y] = dist[x] + (Weight of Edge x,y)
                pred[y] = x
        end
        append x to S
    end
    return(pred)
end
```

5 Rudimentary Pseudocode Time Complexity

The argument for time complexity is similar to but not exactly the same as that for the Shortest Path Algorithm. Assuming we have stored the graph as an adjacency list:

- There is $\Theta(V)$ time required inside the function but excluding the while loop.
- The while loop iterates V times and the body of the while loop, excluding the for loop, takes $\Theta(V)$ time. This is due to the process of finding the vertex in $G - S$ with smallest distance. You should consider how this might be done. This then a total of $\Theta(V^2)$.
- Over the course of the entire algorithm the body of the for loop iterates twice for each edge, taking constant time for each, so that's $\Theta(2E) = \Theta(E)$.

Thus we can say for certain that in the worst-case:

$$T(V, E) = \Theta(V^2 + V + E) = \Theta(V^2 + E)$$

Since for a connected graph we have $V < 2E$ and $E \leq V^2 - V$ this is also $\mathcal{O}(E^2)$ and $\mathcal{O}(V^2)$.

6 Elegant Pseudocode

Here is the pseudocode for a more standard implementation. Instead of using simple lists we will manage the vertices using two structures kept in alignment:

- A min-heap **MH** such that each node contains a vertex number and the corresponding distance from **s**. The distance is the value used for min-heapedness. This is initialized with root node **MH[1]** storing vertex **s** with distance 0 and all other nodes the other vertices all with distance **INF**.
- A management structure **MS** indexed by vertex which contains the corresponding distance from **s**, the vertex's predecessor, the heap location of the vertex, and a flag indicating whether the vertex is (still) in the heap. This is initialized with **MS[s]** storing distance 0 with heap location 0 and predecessor **NULL** and all other **MS[i]** storing distance **INF** with corresponding heap location and predecessor **NULL**.

```
\\" PRE: G is a graph with n vertices.
\" PRE: s is the starting vertex.
def dijkstra(G,s):
    initialize MH
    initialize MS
    while MH is not empty
        Extract root node (u,udist) from MH
        Update MS to indicate u no longer in heap
        for every edge attached to u
            v = vertex at the other end
            (vdist,vinheap,vpred) = MS[v]
            if vinheap and udist + weight(u,v) < vdist
                update MH,MS with new distance, pred
            endif
        endfor
    endwhile
    return(MS)
end
```

} (A)

} (B)

} (C)

7 Elegant Pseudocode Time Complexity

It's tempting to say that since the `while` loop iterates V times and the `for` loop iterates at most E times that there are V iterations of (B) and EV iterations of (C).

However we can be a bit more careful here. All together the `for` loop will follow each edge exactly twice. This is because an edge from u_i to u_j is followed once for u_i , directly after u_i is removed, and once for u_j , directly after u_j is removed. It is then never visited again. Thus in total (C) will iterate $2E$ times. This update of `MH` and `MS` is $\mathcal{O}(\lg V)$ for a total of $\mathcal{O}(2E \lg V)$.

In addition (B) will iterate V times. Extraction and update of `MH` and `MS` is $\mathcal{O}(\lg V)$ for a total of $\mathcal{O}(V \lg V)$.

Along with (A), which we presume is $\mathcal{O}(1)$, we have a total time complexity of:

$$\mathcal{O}(2E \lg V + V \lg V + 1) = \mathcal{O}((2E + V) \lg V) = \mathcal{O}(E \lg V)$$

Note 7.0.1. Arguably since we're allocating structures in (A) one might suggest that they're both $\mathcal{O}(V)$ since each has V items. This does not change the final time complexity calculation.

8 Mathematical Proof that it Works

It may (or may not) make intuitive sense that Dijkstra's Algorithm does what it is claimed to do, but a proof is fairly straightforward.

The key point to understand is that while, in general, assigning a distance to a vertex is not final since that distance may be updated later, when a vertex is added to S the distance assigned to that vertex is in fact final and will not be updated later. It's this latter point we need to prove.

By $d(x)$ we denote the distance as assigned to vertex x by the algorithm. By $c(x_1, \dots, x_j)$ we mean the total cost along the path $\langle x_1, \dots, x_j \rangle$.

Note 8.0.1. This can also be rephrased in terms of a loop invariant and proven using the Loop Invariant Theorem. In that case the loop invariant is the statement about S at each iteration and the proof of the maintenance step is essentially what follows.

Theorem 8.0.1. We claim that whenever we put some x in S that the minimal cost path from s to x is the assigned $d(x)$.

Proof. Suppose not and that at some point we have an S and we are about to add (but have not yet added) some very first $x \notin S$ for which $d(x)$ is not equal to the minimal cost path from s to x .

We know that there must be some some minimal cost path from s to x and since it starts at $s \in S$ and ends at $x \notin S$ we write:

$$\langle s = v_1, v_2, \dots, v_i, v_{i+1}, \dots, v_k = x \rangle$$

Where v_i, v_{i+1} is the first pair along the path with $v_i \in S$ and $v_{i+1} \notin S$. Note that possibly $v_{i+1} = x$ here.

That path must be cheaper than $d(x)$ because $d(x)$ is not minimal:

$$c(s = v_1, \dots, v_i, v_{i+1}, \dots, v_k = x) < d(x) \quad (\text{I})$$

Since $v_i \in S$ we know two things about it. First, since v_i was added earlier, so $d(v_i)$ is minimal:

$$d(v_i) \leq c(v_1, \dots, v_i) \quad (\text{II})$$

Second, that when the algorithm processed v_i all adjacent nodes had their d -values set or updated, so:

$$d(v_{i+1}) \leq d(v_i) + c(v_i, v_{i+1}) \quad (\text{III})$$

From here putting (III) and (II) together:

$$d(v_{i+1}) \leq d(v_i) + c(v_i, v_{i+1}) \leq c(v_1, \dots, v_i) + c(v_i, v_{i+1}) = c(v_1, \dots, v_{i+1}) \quad (\text{IV})$$

In addition if $x = v_{i+1}$ then $d(x) = d(v_{i+1})$ and if $x \neq v_{i+1}$ then the algorithm chose x over v_{i+1} (when selecting what to put in S next) and so $d(x) \leq d(v_{i+1})$. Either way we have:

$$d(x) \leq d(v_{i+1}) \quad (\text{V})$$

Now putting (V) and (IV) and (I) together:

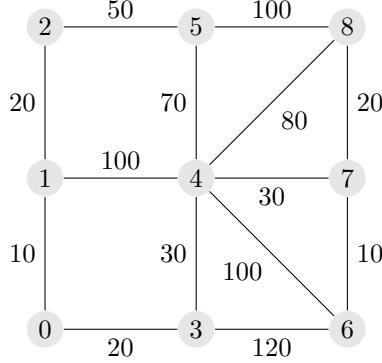
$$d(x) \leq d(v_{i+1}) \leq c(v_1, \dots, v_{i+1}) \leq c(s = v_1, \dots, v_i, v_{i+1}, \dots, v_k = x) < d(x)$$

This is a contradiction and we are done.

\mathcal{QED}

9 Thoughts, Problems, Ideas

1. For the graph given in the notes and replicated here, derive the minimal weight tree starting at vertex 7:



2. If the algorithm simply needs to find the minimum weight path from s to a specific vertex t at which point can it stop and why? Modify the pseudocode accordingly.
3. Dijkstra's Algorithm produces a minimal weight tree rooted at a specific vertex s . Even though this tree is still a tree when we consider some other vertex $s' \neq s$ as the root this tree need not be a minimal weight tree rooted at s' . Show by a four-vertex example that this is the case. Justify your example.
4. A *spanning tree* for a graph G is a subgraph of G which contains all vertices of G and is also a tree. A *minimal spanning tree* is a spanning tree that has smallest possible weight. Show by a 3-vertex example that Dijkstra's Algorithm does not necessarily produce a minimal spanning tree. Justify your example.

10 Python Test and Output

The following code is applied to the graph above. This follows the model of the pseudocode and in addition creates and returns a list of the vertices in the order in which they were visited.

Code:

```
# Return the minimum value and index in dist but not in S.
def min(G,dist,S):
    n = len(G)
    mdist = float('inf')
    # Find a minimum value.
    for v in range(n):
        if v not in S:
            if dist[v] < mdist:
                mdist = dist[v]
    # Find the first index corresponding to that value.
    mi = None
    for v in range(n):
        if (v not in S) and (dist[v] == mdist):
            mvertex = v
            break

    return(mvertex)

def dijkstra(G,u):
    n = len(G)
    dist = [float('inf')] * n
    pred = [None] * n
    S = []
    dist[u] = 0
    print('S: ' + str(S))
    print('dist: ' + str(dist))
    while len(S) != n:
        print('')
        x = min(G,dist,S)
        print('Vertex not in S with minimum distance: ' + str(x))
        S.append(x)
        print('S = ' + str(S))
        for y in range(n):
            if G[x][y] != 0:
                print('Vertex: ' + str(y) + ': ',end='')
                if dist[x] + G[x][y] < dist[y]:
                    print('Update from '+str(dist[y]),end=' ')
                    print(' to ' + str(dist[x]+G[x][y]))
```

```

        dist[y] = dist[x] + G[x][y]
        pred[y] = x
    else:
        print('Do not update from '+str(dist[y]), end=' ')
        print(' to ' + str(dist[x]+G[x][y]))
    print('dist = ' + str(dist))
return(pred)

G = [[ 0, 10, 0, 20, 0, 0, 0, 0, 0],
      [ 10, 0, 20, 0, 100, 0, 0, 0, 0],
      [ 0, 20, 0, 0, 0, 60, 0, 0, 0],
      [ 20, 0, 0, 0, 30, 0, 120, 0, 0],
      [ 0, 100, 0, 30, 0, 70, 100, 30, 80],
      [ 0, 0, 60, 0, 70, 0, 0, 0, 100],
      [ 0, 0, 0, 120, 100, 0, 0, 10, 0],
      [ 0, 0, 0, 0, 30, 0, 10, 0, 20],
      [ 0, 0, 0, 0, 80, 100, 0, 20, 0]]
u = 0
pred = dijkstra(G,u)
print(pred)

```

Output:

```
S: []
dist: [0, inf, inf, inf, inf, inf, inf, inf, inf]

Vertex not in S with minimum distance: 0
S = [0]
Vertex: 1: Update from inf to 10
Vertex: 3: Update from inf to 20
dist = [0, 10, inf, 20, inf, inf, inf, inf, inf]

Vertex not in S with minimum distance: 1
S = [0, 1]
Vertex: 0: Do not update from 0 to 20
Vertex: 2: Update from inf to 30
Vertex: 4: Update from inf to 110
dist = [0, 10, 30, 20, 110, inf, inf, inf, inf]

Vertex not in S with minimum distance: 3
S = [0, 1, 3]
Vertex: 0: Do not update from 0 to 40
Vertex: 4: Update from 110 to 50
Vertex: 6: Update from inf to 140
dist = [0, 10, 30, 20, 50, inf, 140, inf, inf]

Vertex not in S with minimum distance: 2
S = [0, 1, 3, 2]
Vertex: 1: Do not update from 10 to 50
Vertex: 5: Update from inf to 90
dist = [0, 10, 30, 20, 50, 90, 140, inf, inf]

Vertex not in S with minimum distance: 4
S = [0, 1, 3, 2, 4]
Vertex: 1: Do not update from 10 to 150
Vertex: 3: Do not update from 20 to 80
Vertex: 5: Do not update from 90 to 120
Vertex: 6: Do not update from 140 to 150
Vertex: 7: Update from inf to 80
Vertex: 8: Update from inf to 130
dist = [0, 10, 30, 20, 50, 90, 140, 80, 130]

Vertex not in S with minimum distance: 7
S = [0, 1, 3, 2, 4, 7]
Vertex: 4: Do not update from 50 to 110
Vertex: 6: Update from 140 to 90
Vertex: 8: Update from 130 to 100
```

```
dist = [0, 10, 30, 20, 50, 90, 90, 80, 100]

Vertex not in S with minimum distance: 5
S = [0, 1, 3, 2, 4, 7, 5]
Vertex: 2: Do not update from 30 to 150
Vertex: 4: Do not update from 50 to 160
Vertex: 8: Do not update from 100 to 190
dist = [0, 10, 30, 20, 50, 90, 90, 80, 100]

Vertex not in S with minimum distance: 6
S = [0, 1, 3, 2, 4, 7, 5, 6]
Vertex: 3: Do not update from 20 to 210
Vertex: 4: Do not update from 50 to 190
Vertex: 7: Do not update from 80 to 100
dist = [0, 10, 30, 20, 50, 90, 90, 80, 100]

Vertex not in S with minimum distance: 8
S = [0, 1, 3, 2, 4, 7, 5, 6, 8]
Vertex: 4: Do not update from 50 to 180
Vertex: 5: Do not update from 90 to 200
Vertex: 7: Do not update from 80 to 120
dist = [0, 10, 30, 20, 50, 90, 90, 80, 100]
[None, 0, 1, 0, 3, 2, 7, 4, 7]
```

CMSC 351: Floyd's Algorithm

Justin Wyss-Gallifent

November 10, 2023

1	Introduction	2
2	A Dynamic Example	2
3	Path Storage and Reconstruction	4
4	Algorithm	5
5	Time Complexity	5

1 Introduction

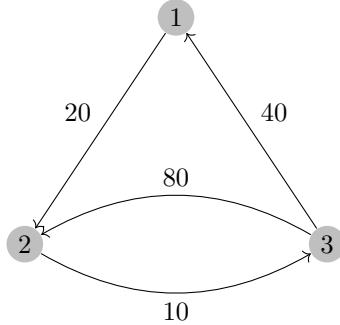
Suppose we have a directed, weighted, simple and connected graph in which both positive and negative (and zero) weights are allowed and we wish to find a shortest path between any two vertices.

The word “path” in this context is a bit different from our definition in that it is allowing repeated edges and vertices. Consequently because we permitting negative weights we could possibly get paths with arbitrarily low (negative) distances if the graph had a cycle with total negative weight since we could just follow the cycle arbitrarily many times.

Since this adds a layer of confusion to the situation we will insist that the graph has no cycles with negative total weight. By doing this the term “shortest path” aligns with our use of the word “path”.

2 A Dynamic Example

Consider the following directed, weighted, simple and connected graph:



Here is the adjacency matrix for that graph which we'll denote by d (rather than the typical A). One way to think about the adjacency matrix is that $d[i, j]$ equals the length of a shortest path from i to j not permitting any intermediate vertices.

$$d = \begin{bmatrix} 0 & 20 & \infty \\ \infty & 0 & 10 \\ 40 & 80 & 0 \end{bmatrix}$$

Suppose now for each i, j we also allow 1 to be an intermediate vertex. Observe that the length of a shortest path from 3 to 2 shrinks because we may use 1 as an intermediate vertex. More formally we observe that:

$$d[3, 1] + d[1, 2] < \text{previous } d[3, 2]$$

Note that none of the other shortest paths change because 1 doesn't help any-

thing else. Let's update the matrix accordingly:

$$d[3, 2] = d[3, 1] + d[1, 2]$$

Now for all entries $d[i, j]$ equals the length of a shortest path from i to j permitting intermediate vertex 1. We call this - Pass By 1:

$$d = \begin{bmatrix} 0 & 20 & \infty \\ \infty & 0 & 10 \\ 40 & 60 & 0 \end{bmatrix}$$

Suppose now for each i, j we also allow 1, 2 to be intermediate vertices. Observe that the length of a shortest path from 1 to 3 shrinks because we may use 2 as an intermediate vertex. More formally we observe that:

$$d[1, 2] + d[2, 3] < \text{previous } d[1, 3]$$

Note that none of the other shortest paths change because 2 doesn't help anything else. Let's update the matrix accordingly:

$$d[1, 3] = d[1, 2] + d[2, 3]$$

Now for all entries $d[i, j]$ equals the length of a shortest path from i to j permitting intermediate vertices 1, 2. We call this - Pass By (1 and) 2:

$$d = \begin{bmatrix} 0 & 20 & 30 \\ \infty & 0 & 10 \\ 40 & 60 & 0 \end{bmatrix}$$

Suppose now for each i, j we also allow 1, 2, 3 to be intermediate vertices. Observe that the length of a shortest path from 2 to 1 shrinks because we may use 3 as an intermediate vertex. More formally we observe that:

$$d[2, 3] + d[3, 1] < \text{previous } d[2, 1]$$

Note that none of the other shortest paths change because 3 doesn't help anything else. Let's update the matrix accordingly:

$$d[2, 1] = d[2, 3] + d[3, 1]$$

Now for all entries $d[i, j]$ equals the length of a shortest path from i to j permitting intermediate vertices 1, 2, 3. We call this - Pass By (1 and 2 and) 3:

$$d = \begin{bmatrix} 0 & 20 & 30 \\ 50 & 0 & 10 \\ 40 & 60 & 0 \end{bmatrix}$$

But now since we're allowing all vertices as intermediate vertices what we've actually obtained is a matrix of shortest paths between all pairs of vertices.

3 Path Storage and Reconstruction

Before we turn this into an algorithm note that the matrix doesn't actually give us the paths, just the lengths of those paths. To obtain the paths let's go back to the start and take another matrix along for the ride.

We define the matrix p which we initialize by setting $p[u, v] = u$ for each edge u to v and by setting $p[v, v] = v$ for each vertex v . All other entries are NULL. Observe that in general $p[u, v]$ is the predecessor of v in a shortest path from u to v not permitting any intermediate vertices:

$$p = \begin{bmatrix} 1 & 1 & \text{NULL} \\ \text{NULL} & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}$$

During our first update of d above we saw that it was quicker to get from 3 to 2 via 1. The value of $p[3, 2]$ needs to be the predecessor of 2 on the path from 3 to 2 but since we go via 1 we should reassign this to be $p[1, 2]$:

$$p[3, 2] = p[1, 2]$$

Now:

$$p = \begin{bmatrix} 1 & 1 & \text{NULL} \\ \text{NULL} & 2 & 2 \\ 3 & 1 & 3 \end{bmatrix}$$

Likewise during our second update of d we update:

$$p[1, 3] = p[2, 3]$$

Now:

$$p = \begin{bmatrix} 1 & 1 & 2 \\ \text{NULL} & 2 & 2 \\ 3 & 1 & 3 \end{bmatrix}$$

And during our third update of d we also update:

$$p[2, 1] = d[3, 1]$$

Now:

$$p = \begin{bmatrix} 1 & 1 & 2 \\ 3 & 2 & 2 \\ 3 & 1 & 3 \end{bmatrix}$$

Now that we are done let's look at how to reconstruct a shortest path from p . Consider a shortest path from 3 to 2. Start by initializing $\text{path} = [2]$. Then

observe that $p[3, 2] = 1$ which means 1 is the predecessor of 2 along a shortest path from 3 to 2 and so we *prepend* 1 and now $path = [1, 2]$. Then observe that $p[3, 1] = 3$ which means 3 is the predecessor of 1 along a shortest path from 3 to 1 and so we prepend 2 and now $path = [3, 1, 2]$.

4 Algorithm

Each of these updates only uses data from the previous d and so it forms an algorithm nicely.

```

d = adjacency matrix for a graph
p = n x n array all null
for each edge (u,v):
    p[u,v] = u
end for
for each vertex v:
    p[v,v] = v
end for
for k = 1 to n:
    for i = 1 to n:
        for j = 1 to n:
            if d[i,k] + d[k,j] < d[i,j]:
                d[i,j] = d[i,k] + d[k,j]
                p[i,j] = p[k,j]
            end if
        end for
    end for
end for

```

The first two **for** loops simply initialize p . Each iteration of k adds another vertex to the list of permissible intermediate vertices and updates the matrices d and p correspondingly.

5 Time Complexity

Starting with an adjacency matrix the first **for** loop is $\Theta(V^2)$ (scanning all entries) and the second **for** loop is $\Theta(V)$. The remaining code is $\Theta(V^3)$ and so overall we have $\Theta(V^3)$.

While this may seem slow this can actually be faster than Dijkstra, for example, if the graph has many many edges.

CMSC 351: Spanning Trees

Justin Wyss-Gallifent

April 24, 2024

1	Introduction	2
2	Prim's Algorithm	3
	2.1 The Algorithm	3
	2.2 Mathematics for Prim	7
	2.3 Pseudocode and Time Complexity	8
3	Kruskal's Algorithm	10
	3.1 The Algorithm	10
	3.2 Mathematics for Kruskal's Algorithm	14
	3.3 Graph Data, Pseudocode, Time Complexity	16
4	Prim v Kruskal	17
5	Thoughts, Problems, Ideas	18

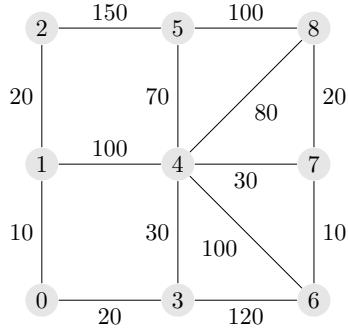
1 Introduction

We've discussed shortest paths and Dijkstra's Algorithm, which finds the minimal cost tree from a given starting vertex to all other vertices, but these involve choosing an initial vertex.

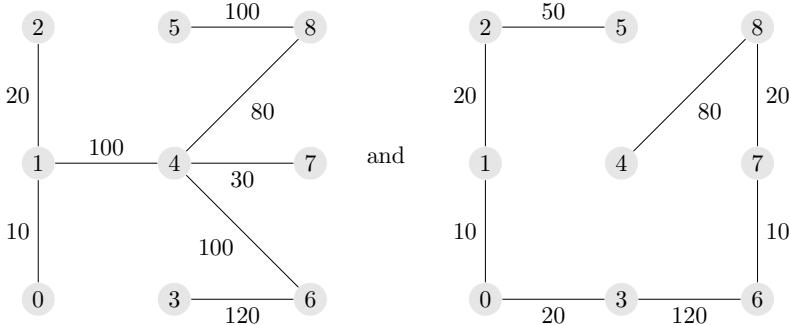
Suppose now we have a weighted graph G and we wish to find a subgraph of G which is not only a tree but which spans the original graph (includes all vertices), and also has minimal cost, where "minimal" means taken over all possible trees.

Consider the following graph:

Example 1.1. Consider this weighted graph:



There are many ways to get a spanning tree. Here are two:



Observe that the spanning tree on the left has a total cost of $20 + 10 + 100 + 100 + 80 + 30 + 100 + 120 = 560$ while the spanning tree on the right has a total cost of $150 + 20 + 10 + 20 + 120 + 10 + 20 + 80 = 420$.

Clearly the one on the right costs less, but could we have done better?

Before proceeding further:

Theorem 1.0.1. Every tree with n vertices has $n - 1$ edges.

Proof. By structural induction. A single vertex is a tree with $n = 1$ vertices and $n - 1 = 0$ edges. A new tree is created by adding an edge to an existing vertex and a new vertex at the other end. This contributes 1 to the vertex count and 1 to the edge count. \mathcal{QED}

2 Prim's Algorithm

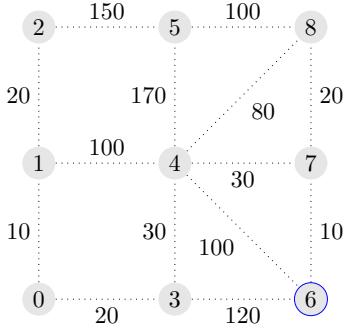
2.1 The Algorithm

Prim's Algorithm works by growing a tree $T \subseteq G$. In what follows denote by $v(H)$ the vertex set of a graph H and by $e(H)$ the edge set of a graph H :

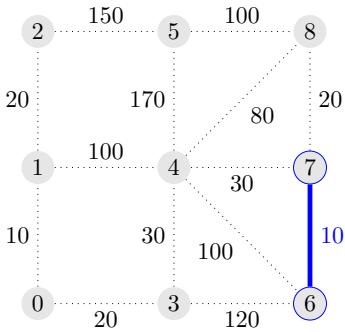
1. Define T to be a graph consisting of one vertex in G . That is, $v(T) = x$ for any $x \in v(G)$ and $e(T) = \emptyset$.
2. While $v(T) \neq v(G)$ pick a minimal weight edge $e(x, y)$ with $x \in v(T)$ and $y \in v(G) - v(T)$ and add the edge and the vertex to T , so $v(T) = v(T) + y$ and $e(T) = e(T) + \text{edge}(x, y)$.

Note 2.1.1. Note that the condition that the minimum cost edge adds a new vertex to the tree is equivalent to insisting that the minimum cost edge does not form a cycle.

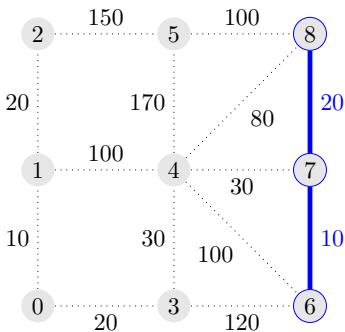
Example 2.1. Consider the example from earlier. Here we have excluded all edges by dashing them. We'll start with an arbitrary choice of vertex 6.



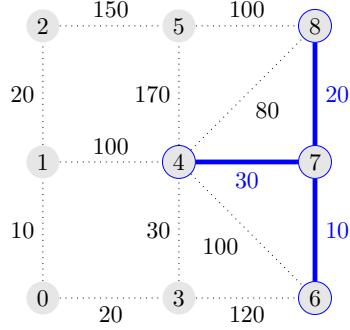
We now choose a minimum weight edge going between a vertex in $\{6\}$ and a vertex in $\{0, 1, 2, 3, 4, 5, 7, 8\}$. We choose $\text{edge}(6, 7)$ which picks up vertex 7:



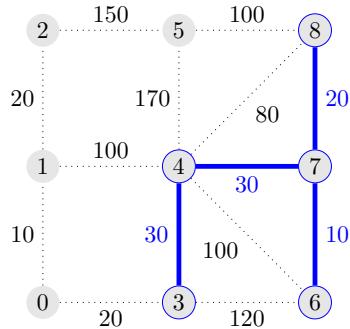
We now choose a minimum weight edge going between a vertex in $\{6, 7\}$ and a vertex in $\{0, 1, 2, 3, 4, 5, 8\}$. We choose $\text{edge}(7, 8)$ which picks up vertex 8:



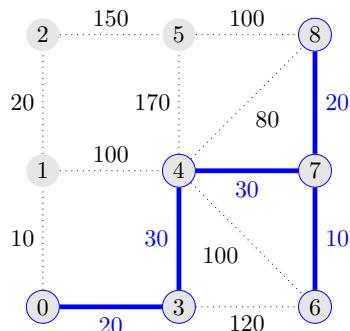
We now choose a minimum weight edge going between a vertex in $\{6, 7, 8\}$ and a vertex in $\{0, 1, 2, 3, 4, 5\}$. We choose $\text{edge}(4, 7)$ which picks up vertex 4:



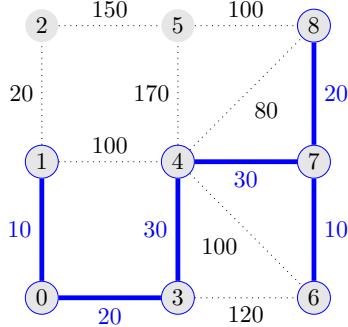
We now choose a minimum weight edge going between a vertex in $\{4, 6, 7, 8\}$ and a vertex in $\{0, 1, 2, 3, 5\}$. We choose $\text{edge}(3, 4)$ which picks up vertex 3:



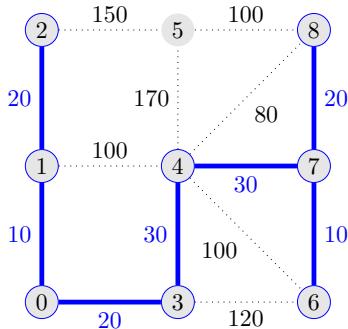
We now choose a minimum weight edge going between a vertex in $\{3, 4, 6, 7, 8\}$ and a vertex in $\{0, 1, 2, 5\}$. We choose $\text{edge}(0, 3)$ which picks up vertex 0:



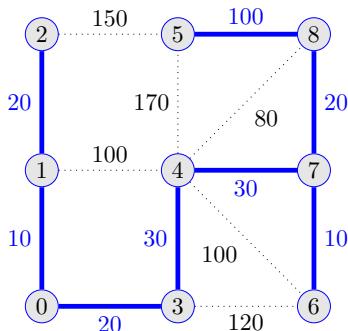
We now choose a minimum weight edge going between a vertex in $\{0, 3, 4, 6, 7, 8\}$ and a vertex in $\{1, 2, 5\}$. We choose $\text{edge}(0, 1)$ which picks up vertex 1:



We now choose a minimum weight edge going between a vertex in $\{0, 1, 3, 4, 6, 7, 8\}$ and a vertex in $\{2, 5\}$. We choose $\text{edge}(1, 2)$ which picks up vertex 2:



We now choose a minimum weight edge going between a vertex in $\{0, 1, 2, 3, 4, 6, 7, 8\}$ and a vertex in $\{5\}$. We choose $\text{edge}(5, 8)$ which picks up vertex 8:



We have now picked up all vertices and we are done. We now have a minimum cost spanning tree. The total cost is 240.

2.2 Mathematics for Prim

Theorem 2.2.1. Prim's Algorithm creates a minimal spanning tree.

Proof. Let G be a weighted and connected graph on which we have run Prim's Algorithm resulting in the tree which contains the edges $\{e_1, \dots, e_{n-1}\}$.

Since the empty set $\{\}$ may be extended (somehow) to a minimal spanning tree but the set $\{e_1, \dots, e_{n-1}\}$ may not (it's already a tree and adding any edge will add a cycle and ruin this), there is some maximum i such that $S = \{e_1, \dots, e_{i-1}\}$ may be extended (somehow) to a minimal spanning tree but $\{e_1, \dots, e_i\}$ may not be extended (somehow) to a minimal spanning tree. Note that S may be empty here!

Define W to be the set of nodes selected by Prim's Algorithm just before e_i is added. If we put $e_i = \text{edge}(x, y)$ this means that x is in W but y is not.

Define U to be any minimal spanning tree containing S . Since there is certainly a path P from x to y in U , and since x is in W but y is not in W , if we follow this path we eventually encounter some node x' which is still in W followed by a node y' which is not in W .

Define the set:

$$T = U - \{\text{edge}(x', y')\} + \{\text{edge}(x, y)\}$$

Observe that the deletion disconnects the tree U because it removes the only connection from x' to y' and then the addition reconnects it because x' and y' are now connected by following P from x' to x , then going to y , then following P from y to y' .

Consider now:

- If $c(x, y) < c(x', y')$ then the total cost of T is less than that of U which contradicts the fact that U is a minimal spanning tree.
- If $c(x, y) > c(x', y')$ then Prim's Algorithm would have chosen (x', y') (which it could have done since $x' \in U$) instead of (x, y) but it didn't.
- If $c(x, y) = c(x', y')$ then T is also a minimal spanning tree. However note that T contains the edges in S and edge $e_i = \text{edge}(x, y)$, this tells us that $\{e_1, e_2, \dots, e_{i-1}, e_i\}$ can be extended to a minimal spanning tree (that being T) which contradicts our assumption.

In all cases we have contradictions and we are done.

\mathcal{QED}

2.3 Pseudocode and Time Complexity

Loosely speaking the pseudocode is easy:

```
\\" PRE: G is a graph
T = graph consisting of an arbitrary vertex in G
while vertices in T != vertices in G
    select (u,v) = min cost edge (u,v) with u in T and v in G-T
    T = T + (u,v)
end
\" POST: T is a minimal spanning tree
```

The devil is in the details, however, specifically in the critical line:

```
select (u,v) = min cost edge (u,v) with u in T and v in G-T
```

Here are some options:

- This can be easily done in $\Theta(V^3)$:

If G is stored as an adjacency matrix AM we create a boolean list INT of length V storing whether each vertex is in T .

Then to accomplish our critical line we scan every entry in AM and for each we use INT to check if one vertex is in T and one is not, and we pick the one with minimum cost and include it in T .

The scanning is $\Theta(V^2)$ and the checks are $\Theta(1)$ and since the while loop runs V times, this entire process is $\Theta(V^3)$.

- There is a more refined approach with an adjacency matrix which is $\Theta(V^2)$:

If G is stored as an adjacency matrix then in addition to INT above we create a distance list $DIST$ of length V full of keys each equal to \inf except we set $DIST[s] = 0$ and we create a predecessor list $PRED$ of length V .

To jump-start the process we first include s in T and go through all vertices adjacent to s and for each we label their $DIST$ to be the edge weight to s and for each we label their $PRED$ to be s .

Then we iterate. To accomplish our critical line we scan pick the vertex x with minimum $DIST$ and which is not already in T and we include it in T along with the edge stored in $PRED$. Then for each vertex y adjacent to x and not in T we update $DIST$ to be the minimum of its current value and its distance to x . If we update $DIST$ the we also update $PRED$ to update the predecessor of x .

The scanning is $\Theta(V)$ and the “for each vertex” is $\Theta(V)$ but these are done in series so together they are $\Theta(V + V) = \Theta(V)$ and since the while loop runs V times, this entire process is $\Theta(V^2)$.

- Using various other data structures this can be brought down further. For

example with a min heap to store the edge weights and an adjacency list we can obtain $\Theta(E \lg V)$ and using a Fibonacci heap (which is a collection of min heaps) to store the edges weights and adjacency list we can obtain $\Theta(E + V \lg V)$.

3 Kruskal's Algorithm

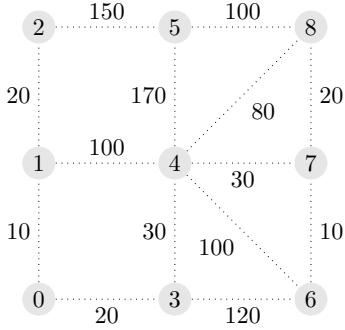
3.1 The Algorithm

Kruskal's Algorithm works as follows:

1. For now, exclude all edges.
2. Pick an excluded edge of minimum cost which does not form a cycle when included. Include it.
3. Repeat step 2 until we span the original graph.

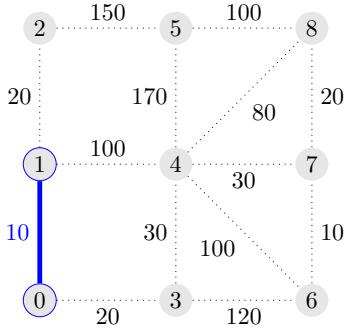
Note 3.1.1. We will eventually span the original graph because the graph spans itself. Moreover we can do this without adding a cycle because cycles are not necessary to achieve spanning.

Example 3.1. Consider the example from earlier. Here we have excluded all edges by dashing them:

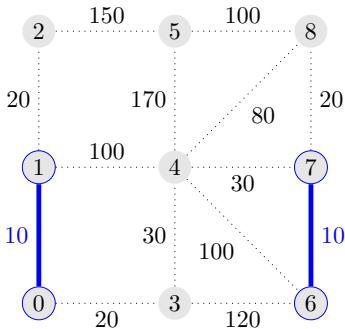


Since there are 9 vertices we'll need 8 edges so this will be an 8-step procedure.

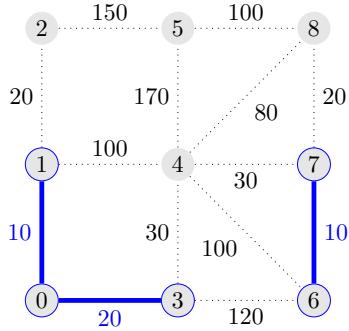
We add edge $0 - 1$ which does not create a cycle:



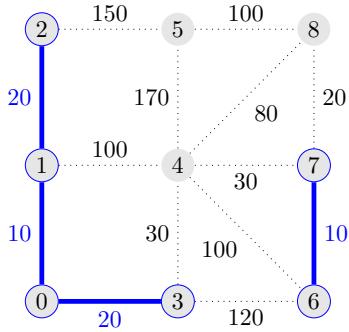
We add edge $6 - 7$ which does not create a cycle:



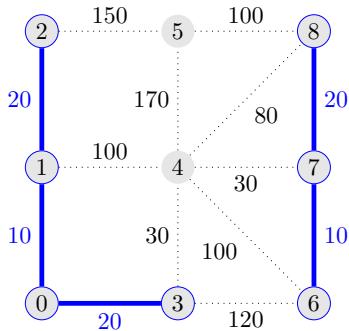
We add edge $0 - 3$ which does not create a cycle:



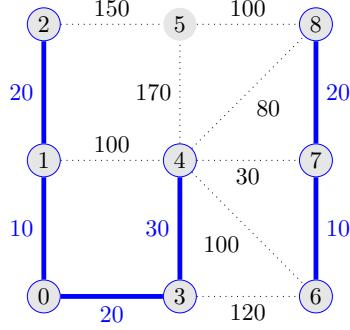
We add edge $1 - 2$ which does not create a cycle:



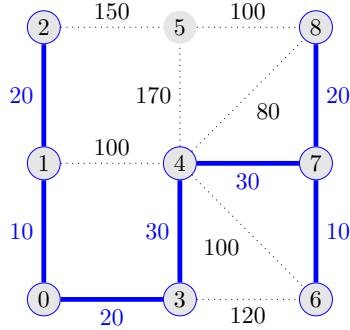
We add edge $7 - 8$ which does not create a cycle:



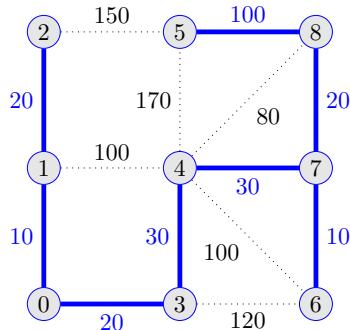
We add edge $3 - 4$ which does not create a cycle:



We add edge $4 - 7$ which does not create a cycle:



The next minimum cost edge is $4 - 8$ but adding it creates the cycle $4 - 8 - 7$ so we skip over it. and instead we add edge $5 - 8$ which does not create a cycle:



We now have a minimum cost spanning tree. The total cost is 240.

3.2 Mathematics for Kruskal's Algorithm

Theorem 3.2.1. Kruskal's Algorithm creates a MST.

Proof. The fact that we actually obtain a spanning tree follows from the fact that we proceed until we span the graph and from the fact that we do not add an edge if it would create a cycle, therefore we get a tree.

We now claim that at each iteration of the algorithm that if S is the set of edges we have included then S is contained in a MST.

If this is true then when we have obtained a spanning tree at the end this spanning tree is contained in a MST but since the only spanning tree it is contained in is itself, it must be that MST.

We proceed by structural induction, meaning we show that at the start S is contained in a MST and that if at any iteration this is true then it is still true after Kruskal adds another edge.

The statement is obviously true at the start because no edges are included and hence any MST will work.

Suppose the statement is true at some S and let $S \subseteq T$ where T is the MST. Kruskal's Algorithm adds some edge, $S + \{e\}$. We need to prove that there is a MST containing $S + \{e\}$.

Consider two cases:

- $e \in T$:

Then $S + \{e\} \subset T$ and so T is a MST for $S + \{e\}$.

- $e \notin T$:

Since T is a tree and we have added an edge, $T + \{e\}$ contains a cycle.

Since $S + \{e\}$ does not contain a cycle (as this is how Kruskal works) there is some edge $f \in \text{cycle} \subseteq T + \{e\}$ with $f \notin S + \{e\}$. Observe now that $S + \{e\} \subseteq T + \{e\} - \{f\}$. We claim that $T + \{e\} - \{f\}$ is a MST for $S + \{e\}$.

Since $e, f \notin S$ and Kruskal chose e and not f we have $w(e) \leq w(f)$. This tells us $w(T + \{e\} - \{f\}) \leq w(T)$.

Observe that $T + \{e\} - \{f\}$ is also a spanning tree since we took the spanning tree T , added an edge to create a cycle and then removed an edge within that cycle, getting a tree back. Since T is minimal we also have $w(T + \{e\} - \{f\}) \geq w(T)$.

Together this tells us that $w(T + \{e\} - \{f\}) = w(T)$ and so $T + \{e\} - \{f\}$ is a MST containing $S + \{e\}$ as desired.

\mathcal{QED}

3.3 Graph Data, Pseudocode, Time Complexity

Loosely speaking the pseudocode is easy:

```
\ \ PRE: G is a graph
T = empty graph
while T is not a spanning tree:
    find (u,v) = minimum cost edge in G-T
    such that T+(u,v) does not form a cycle
    T = T + (u,v)
end
end
\ \ POST: T is a minimal spanning tree
```

Of course as before the devil is in the details. Keeping track of the edges in $G-T$ is straightforward and selecting a minimum cost edge is not hard (this could be done with a simple linear search or with a binary heap) the significant challenge here is determining when $T + (u, v)$ does not contain a cycle.

- Fairly simple data structures can be used to achieve this and such an implementation can achieve a time complexity of $\mathcal{O}(E \lg E)$. This is equivalent to $\mathcal{O}(E \lg V)$ because $E \lg E = \mathcal{O}(E \lg V^2) = \mathcal{O}(2E \lg V) = \mathcal{O}(E \lg V)$ and $E \lg V = \mathcal{O}(E \lg(2E)) = \mathcal{O}(E \lg E)$.
- A more technical approach here is to use a *disjoint-set data structure* which is a data structure that handles sets well, and quickly, and allows such calculations with low time complexity. This is beyond the scope of this course, however such an implementation of Kruskal's Algorithm can be shown to run in $\mathcal{O}(E\alpha(V))$ time, where α is the inverse of the single-valued Ackermann function. Without diving too deeply into details this function α is “almost constant”.

4 Prim v Kruskal

There are a few considerations that come into play when choosing between Prim and Kruskal.

1. If Prim ends early then we still have a tree.
2. Kruskal's selection process allows us to be much more flexible if we wish to interact and make specific choices of edges. This is because we can choose any minimal weight edge which does not form a cycle, whereas in Prim we are restricted to growing the tree.
3. Prim allows us to select a starting vertex which gives us a different sense of control, especially when combined with the first point.
4. Prim tends to run faster in dense graphs (lots of edges). This is because even though there are more edges, Prim requires us to grow a tree which restricts the edges we can choose from. Especially earlier on in the algorithm Prim will have far fewer choices even in a very dense graph.
5. Kruskal tends to run faster in sparse graphs (few edges). This is because it's fairly quick to choose from a sparse set of edges.
6. Kruskal requires us to do cycle detection and this in and of itself can be challenging, or at least obscure.

5 Thoughts, Problems, Ideas

1. In Prim's Algorithm we need to check if our minimum-cost edge joins a vertex in T to a vertex in $G - T$. This is equivalent to determining if the edge creates a cycle. However in Kruskal's Algorithm these are not equivalent. Explain
2. In the Graphs chapter exercises you need to write the pseudocode to detect if a graph contains a cycle. Integrate that pseudocode into Kruskal's Algorithm and discuss the time complexity.
3. (a) If G has 5 nodes and 5 edges with weights $\{1, 2, 3, 4, 5\}$ and you construct a Kruskal minimal weight spanning tree, what are the minimum and maximum total possible weights? Explain and draw example graphs. Do not do this by attempting to list all graphs!
(b) Repeat the previous question for 6 nodes and 6 edges with weights $\{1, 2, 3, 4, 5, 6\}$.
4. ...more to come...

CMSC 351: Minimax

Justin Wyss-Gallifent

November 29, 2023

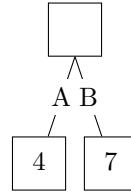
1	Introduction	2
2	Minimax Algorithm	3
	2.1 Inspiration	3
	2.2 Algorithm	4
	2.3 Time Complexity	5
3	Practicalities	5
4	Examples	5

1 Introduction

Consider the following abstraction of a game:

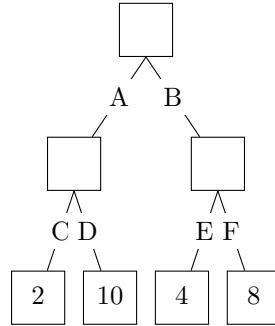
Max and Min are playing a game in which the entire state of the game can be represented by a single game value. Max's objective is to obtain the largest game value possible while Min's objective is to obtain the smallest game value possible.

Suppose it is Max's move and there are two moves available, *A* and *B*. Each leads to a specific game value:



Clearly Max would choose move *B*, resulting in a game value of 7.

Suppose on the other hand it is Max's move but Max doesn't know the game values which result. Instead, after Max's move it is Min's move. However suppose Max can see two moves ahead as follows:



What should Max do?

Clearly Max sees that there is a 10 available on the left so Max might choose move *A* but if Max does so then Min will choose move *C* and the game will have a value of 2. On the other hand if Max chooses move *B* then Min will choose move *E* and the game will have a value of 4.

It follows that Max should choose move *B* to obtain the highest possible value (given Min's response) of 4.

2 Minimax Algorithm

2.1 Inspiration

Now let's un-abstract it a bit.

Imagine a game with two players, Max and Min. We have a function which assigns a value to each position in the game.

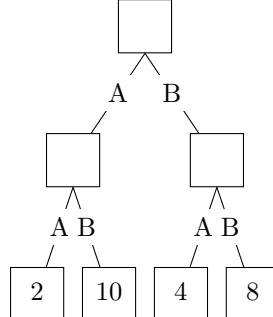
Definition 2.1.1. A function which assigns a value to each position in the game is called a *heuristic function*. We'll denote this by h .

Max's goal is to achieve the maximum possible value while Min's goal is to achieve the minimum possible value.

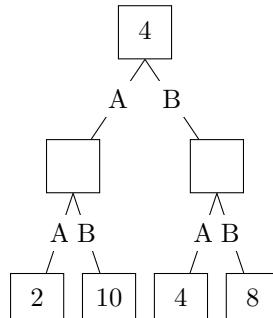
The underlying idea is that Max may look some number of moves ahead (maybe 1, maybe 100), calculate the corresponding heuristic values, and will make a choice of current move based upon that.

The Minimax Algorithm is a simple algorithm which gives the best possible move, assuming it is Max's turn. If there are multiple branches with the same value then Max may choose either branch.

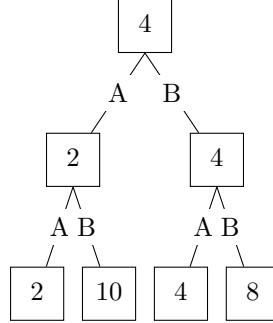
Let us revisit the opening example:



Assume the leaf values are the values of the function two moves ahead. We saw in the opening that Max ought to choose move B to eventually achieve $h = 4$. We could represent this by putting an 4 in the root:



In addition we could look at the middle layer and consider those are Min's moves. On the left (after Max chooses move *A*) clearly Min would choose move *A* to eventually achieve $h = 2$, and on the right (after Max chooses move *B*) Min would also choose move *A* to eventually achieve $h = 4$. We could fill those values in too:



We now make the simple observation that we can fill the tree in from the bottom up using the following algorithm:

- If we are filling in Min's row we should use the smaller (the minimum) of the child values.
- If we are filling in Max's row we should use the larger (the maximum) of the child values.

Note 2.1.1. It is important to note that the leaves of the tree are the only nodes which we fill with the values of the heuristic function based on the states of the game.

When we are filling in the nodes above these are minimax values, not heuristic function values!

So now onto the algorithm in pseudocode.

2.2 Algorithm

Let's assume we have a tree consisting of a set of nodes. The leaf nodes are assumed to represent the ends of the game and so for a leaf node *n* the value *n.value* is pre-assigned. All other nodes have *n.value* undefined to start with.

The function `minimax` is called with the first argument being the root of the tree and the second argument being 0. Recursive calls will have *depth* being even when it is Max's turn and odd when it is Min's turn.

```

function minimax(node nd, depth):
    if nd.value is not undefined:
        return(nd.value)
    else:
        if depth is even:
  
```

```

        return( max(minimax(ndc,depth+1) for all children ndc of nd) )
else:
    return( min(minimax(ndc,depth+1) for all children ndc of nd) )
end if
end if
end function

```

2.3 Time Complexity

The algorithm makes no assumptions about how many children each node has, the depth of the tree, or whether the tree is complete. Suppose however that the depth of the tree is d and the tree is complete with $b \geq 2$ children per node (this is the branch factor). The function itself runs at constant time for a given node so we need to count the number of nodes in the tree.

Assuming the constant time for one node is 1 then this yields a simple geometric sum for the time complexity:

$$1 + b + b^2 + \dots + b^d = \frac{b^{d+1} - 1}{b - 1} = \left(\frac{1}{b - 1} \right) (b^{d+1} - 1) \leq b^{d+1} - 1$$

This yields a time complexity of $\mathcal{O}(b^d)$.

3 Practicalities

In real-world applications it is generally not realistic to construct a tree which goes all the way to the end of the game due to the sheer number of possible moves available to each player and the number of moves the game lasts. In such cases we proceed as follows:

1. Decide how many moves we can realistically look ahead.
2. Expand the tree to that depth and to each leaf node (which will correspond to a game arrangement) we assign a value.
3. Apply the Minimax Algorithm to that tree.

4 Examples

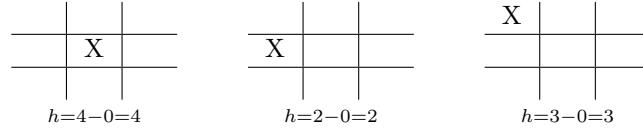
Example 4.1. Consider tic-tac-toe where Max is playing X and goes first while Min is playing O and goes second.

While this is a simple game to play, drawing a tree diagram is not trivial. Assuming Max starts, there are 9 moves available, followed by 8 for Min, and so on. Of course we can reduce this by symmetries since technically speaking Max has only three opening moves - Center, Edge, Corner.

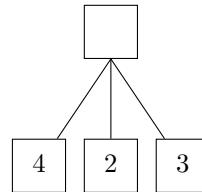
One reasonable heuristic h would be the number of open winning lines (OWLs) for Max minus the number of OWLs for Min. An OWL is a line which contains at least one of the player's marks and none of the opponent's.

At the start of the game we have $h = 0 - 0 = 0$. and it is Max's move.

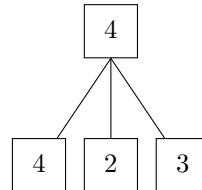
Let's look just one move ahead. If we look at Max's three opening moves (up to symmetry) we can calculate the heuristic function h :



Let's put the values of the heuristic function after one move into the leaves of a tree:

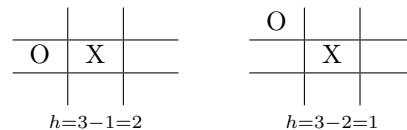


If we then fill it in according to minimax:

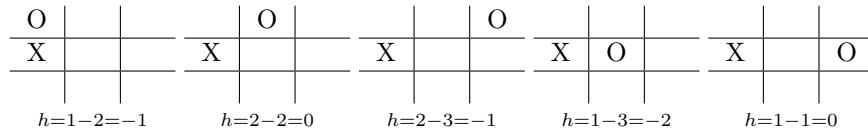


We see that it is in Max's best interest to play the middle.

Let's now look at two moves ahead. If Max plays the center then up to symmetry Min has two possible moves:



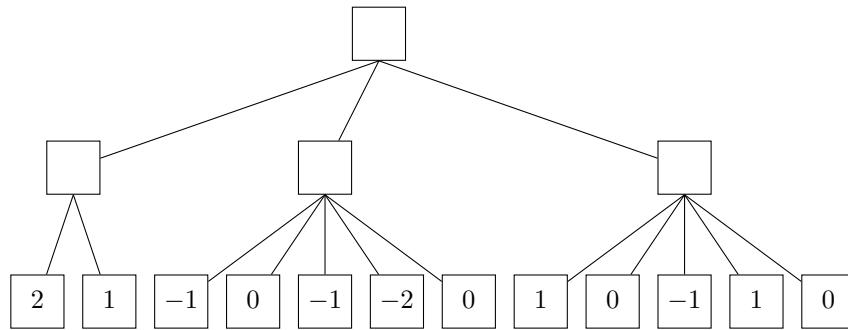
If Max plays the edge then up to symmetry Min has five possible moves:



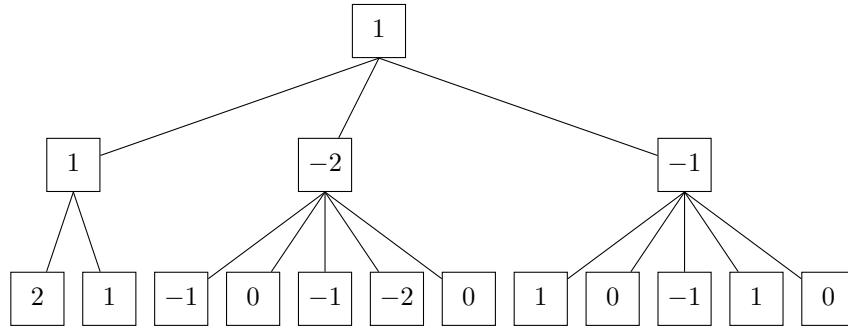
If Max plays the corner then up to symmetry Min has five possible moves:

X O	X O	X O	X O	X O
$h=2-1=1$	$h=2-2=0$	$h=2-3=-1$	$h=3-2=1$	$h=2-2=0$

Let's put the values of the heuristic function after two moves into the leaves of a tree:



If we then fill it in according to minimax:



We see that it is still in Max's best interest to play the middle.

CMSC 351: Huffman Encoding

Justin Wyss-Gallifent

May 10, 2024

1	Introduction	2
1.1	An Introductory Example	2
1.2	Prefix Codes and Fixed-Length Codes	2
2	Huffman Encoding	3
2.1	Introduction	3
2.2	Algorithm	3
2.3	(Non)Uniqueness	5
2.4	Greediness	5
3	Time Complexity	6
3.1	Introduction	6
3.2	Tree Building	6
3.3	Code Extraction	6
4	Minimality Proof	7

1 Introduction

1.1 An Introductory Example

Suppose we wish to encode the character string HELLOOOO (with lots of 0s!) by assigning a binary string (called a *code word*) to each character.

There are four characters so one idea would be to assign E=00, H=01, L=10, O=11. Then we would have:

$$\text{HELLOOOO} = 0100101011111111$$

We might wonder, however, if we can do this with fewer bits. What if we tried E=0, H=1, L=00, O=11? Then we would have:

$$\text{HELLOOOO} = 10000011111111$$

This is 12 bits instead of 16.

However there is a problem which is that 10000011111111 could be other character strings such as HEEEEHHHHHHHH.

But perhaps there is some other way that will work.

First off let's recognize what went wrong with our attempt above. One problem is that the string 0 (for E) is the prefix for 00 (which is L) so when we encounter 00 we don't know whether it should represent EE or L.

1.2 Prefix Codes and Fixed-Length Codes

First let's get some definitions down.

Definition 1.2.1. What we are trying to develop here is a *code* and the binary strings mentioned above are called *code words*.

Definition 1.2.2. A *prefix code* is a code which has the property that no code word is the prefix of another code word.

Example 1.1. Our initial code E=00, H=01, L=10, O=11 is a prefix code but our second attempt E=0, H=1, L=00, O=11 is not a prefix code.

Definition 1.2.3. A *fixed-length code* is a code in which each code has exactly the same length.

Example 1.2. Our initial code is a fixed-length code but our second attempt is not a fixed-length code.

It is easy to see that any fixed-length code is a prefix code but not every prefix code is a fixed-length code.

Okay, so are there any prefix codes which are not fixed-length codes?

Consider the code H=000, 0=1, L=01, and E=001. This is a prefix code but not a fixed-length code.

Moreover using this code we have HELLO0000 = 00000101011111 which requires 14 bits, fewer than the 16 required for our fixed-length code.

2 Huffman Encoding

2.1 Introduction

Before proceeding let's reflect upon our last example. You may protest and suggest that while HELLO0000 uses fewer bits with our new code than with our fixed-length code, this will not be true of all character strings using this new code. For example the single character string H uses 3 bits instead of 2.

Thus you might ask - are there any prefix codes which are not fixed-length codes and which will result in every possible character string requiring no more bits than it would have using a fixed-length code? The answer to this is no, unfortunately.

The key point for us, though is to answer the following more specific question:

Given a specific character string, can we construct a prefix code which has the property that this code will result in our character string using the minimum number of bits possible amongst all codes?

The answer to this is yes!

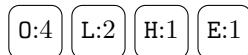
2.2 Algorithm

It makes sense that when encoding a character string we should identify the characters which occur most frequently and the code should assign those characters a code word with as few bits as possible.

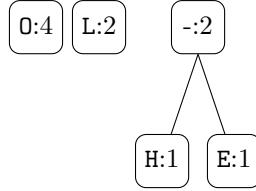
In light of that we first sort our characters by count. Let's do this with HELLO0000. Note that H and E have the same count and could be in either order.

Character	Count
O	4
L	2
H	1
E	1

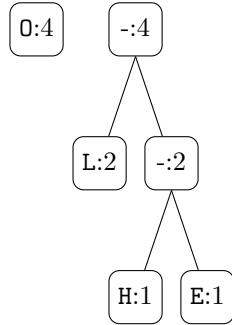
We start by creating a binary tree for each letter. Each will be simply a root node and each has a value assigned to it which is the character count.



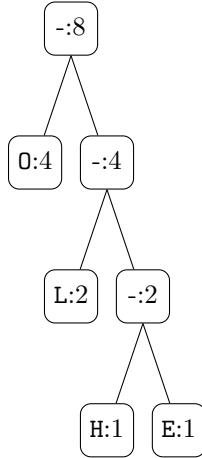
Next we pick the two binary trees with the lowest total count and combine them under a new parent root node. We assign that root (and the entire tree) a count equal to the sum of the counts of the two binary trees:



We then repeat, noting that the two binary tries with the lowest total count are the one containing the *L* and the one we just built:

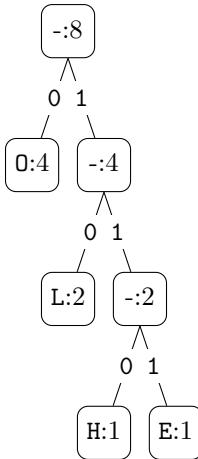


Then one last time:



Observe that the most frequent characters are closer to the top of the tree because they were picked up later on in the process.

We then label the branches with 0 on the left and 1 on the right:



We then read the encoding from the tree in that the binary string leading to each character will be the code word for that character. Here is the table including the counts:

Character	Count	Code Word
O	4	0
L	2	10
H	1	110
E	1	111

Then we have the final result requiring 14 bits:

HELLO = 11011110100000

2.3 (Non)Uniqueness

Observe that the encoding is not unique for several reasons. There may be trees (even single characters) with equal counts and there may be more than one way to pick two trees with minimum total count. Moreover when we combine two trees to form a new tree it doesn't really matter which goes left and which goes right, meaning which gets assigned a branch value of 0 and which gets assigned a branch value of 1.

2.4 Greediness

Observe that the Huffman algorithm is a greedy algorithm in that at each iteration it makes the best possible choice without knowing what will happen in future iterations.

Even though the algorithm is greedy, which does not in general guarantee an optimal result, we shall see that it does turn out nicely here.

3 Time Complexity

3.1 Introduction

There are two processes we are interested in here. First is the time complexity of building the tree and second is the time complexity of extracting the codes.

3.2 Tree Building

The most efficient way to construct the final binary tree is as follows:

```
identify each of the n starting binary trees with
their count and use this as a value to construct
a min heap H where each node is a count-tree pair.
for i = 1 to n-1:
    extract the binary tree t1 with the smallest count from H
    rebuild H
    extract the binary tree t2 with the smallest count from H
    rebuild H
    combine t1 and t2 to form the tree t
    insert t into H
end for
```

It takes time $\mathcal{O}(n \lg n)$ to construct H (this can be done in $\mathcal{O}(n)$ but the math is less obvious and it does not affect our final result). The loop iterates $n - 1$ times and for each iteration the extraction and rebuilding takes time $\mathcal{O}(\lg n)$ (not Θ since the heap is getting smaller).

Consequently the overall time complexity is:

$$\mathcal{O}(n \lg n) + (n - 1)\mathcal{O}(\lg n) = \mathcal{O}(n \lg n)$$

3.3 Code Extraction

Before proceeding, a small theorem:

Theorem 3.3.1. Suppose a binary tree has the property that each node has either 0 or 2 children - called a *full binary tree*. If N is the number of nodes and L is the number of leaves then we have $N = 2L - 1$.

Proof. An outline of the proof is as follows, a more rigorous proof would use structural induction. such a binary tree can be constructed by starting with a single node and then progressively picking a leaf and adding two children to it, then repeating. Each time we pick a leaf and add two children we increase N by 2 and increase L by 1, thus we have N following the pattern 1, 3, 5, 7, 9, ... and L following the pattern 1, 2, 3, 4, 5, ... and we see $N = 2L - 1$. \mathcal{QED}

Our approach to code extraction will be to construct an object which contains the characters and their codes. We will do this in one pass through the tree: meaning we will visit each node exactly once.

The pseudocode is very simple:

```

function extractcodes(node nd, binary string s):
    if nd is a leaf:
        object[leaf character] = s
    else:
        extractcodes(nd.leftchild, s+'0')
        extractcodes(nd.rightchild, s+'1')
    end if
end function
global object = {}
extractcodes(root, '')

```

Whenever `extractcodes` encounters a leaf node the current `binary string s` will be the associated binary string along the path to that node. Consequently the string will be assigned to the `leaf character` stored in the leaf.

Suppose we have n characters in our binary tree. Since each of those n characters corresponds to a leaf, by our theorem earlier there are $2n - 1$ nodes in the tree and hence our pseudocode has time complexity $\Theta(2n - 1) = \Theta(n)$.

4 Minimality Proof

Note 4.0.1. In all of the above we have used the count of the characters rather than the frequency. This is because the count is always an integer whereas the frequency can be an awkward fraction. In what follows, however, we will switch to frequency for easier calculation. Note that this does not affect anything in the algorithm since higher count corresponds to higher frequency.

Intuition suggests that the prefix code constructed this way be efficient, meaning it should use very few bits. This is because characters which occur infrequently end up in trees which get picked repeatedly by the algorithm and consequently end up lower in the final tree, thereby requiring more bits, whereas characters which occur frequently end up higher in the tree, requiring fewer bits.

More formally though we have the following:

Theorem 4.0.1. A prefix tree constructed via the Huffman algorithm yields a minimal prefix code for the original character string. That is, using the prefix code resulting from the prefix tree grown by the Huffman algorithm, the total number of bits used to encode the character string is as small as possible.

We break the proof up into a series of steps:

(a) **Notation:**

In what follows we assume we have a given string. Let A be the set of characters in the string. For each $x \in A$ let $f(x)$ be the frequency of x in the string.

(b) **Definition:**

Suppose T is a binary tree such that each $x \in A$ corresponds to a leaf in T . For each $x \in A$ let $d(x, T)$ be the depth of x in the tree T . Define:

$$s(T) = \sum_{x \in A} f(x)d(x, T)$$

Observe that minimizing s is equivalent to minimizing the total number of bits used.

(c) **Swap Lemma:**

Suppose T is a binary tree such that each $x \in A$ corresponds to a leaf in T . Let $y, z \in A$ and let T' be the tree obtained by swapping y and z . Then:

$$s(T') - s(T) = (f(y) - f(z))(d(z, T) - d(y, T))$$

Proof:

We have the following, noting that because of the swap we have $d(y, T') = d(z, T)$ and $d(z, T') = d(y, T)$:

$$\begin{aligned} s(T') - s(T) &= \sum_{x \in A} f(x)d(x, T') - \sum_{x \in A} f(x)d(x, T) \\ &= [f(y)d(y, T') + f(z)d(z, T')] - [f(y)d(y, T) + f(z)d(z, T)] \\ &= [f(y)d(z, T) + f(z)d(y, T)] - [f(y)d(y, T) + f(z)d(z, T)] \\ &= f(y)(d(z, T) - d(y, T)) + f(z)(d(y, T) - d(z, T)) \\ &= f(y)(d(z, T) - d(y, T)) - f(z)(d(z, T) - d(y, T)) \\ &= (f(y) - f(z))(d(z, T) - d(y, T)) \end{aligned}$$

(d) **Optimal Lemma:**

There exists an optimal tree (minimal $s(T)$) such that two characters with the lowest frequencies are siblings and are at the deepest level of the tree.

Proof:

There are only finitely many possible trees so clearly there is one, call it T , with minimal $s(T)$. Choose y to be a character with lowest frequency and maximum depth and z to be chosen similarly after y . Proceed as follows:

First, if y is an only child then create T' by removing y and assigning y to the parent. Then $s(T') < s(T)$, a contradiction, so we can assume y has a sibling.

Second, if y is not at the deepest level of the tree then there exists a node w with $d(w, T) > d(y, T)$. Create T' by swapping w and y . Then by the Swap Lemma we have:

$$s(T') = s(T) + (f(w) - f(y))(d(y, T) - d(w, T))$$

Since $f(w) > f(y)$ (since y and z have lowest frequencies and y is, of those, deepest) and $d(w, T) > d(y, T)$ we have $s(T') < s(T)$, a contradiction, so we can assume y is at the deepest level of the tree.

Third, if z is the sibling of y we are done, otherwise assume $w \neq z$ is the sibling of y . Create T' by swapping w with z . Then by the Swap Lemma we have:

$$s(T') = s(T) + (f(w) - f(z))(d(z, T) - d(w, T))$$

Since $f(w) \geq f(z)$ (since y and z have lowest frequencies) and $d(z, T) \leq d(y, T)$ (assumed) and $d(y, T) = d(w, T)$ (because they are siblings) we have $s(T') \leq s(T)$.

However since T is optimal we cannot have $<$ and hence $S(T') = S(T)$ and so T' is also optimal so we replace T with T' .

(e) **Proof of Theorem:**

The proof is by induction on n , the number of different characters in A .

Clearly the theorem is true for $n = 1$ so let us assume it is true for $n = k$ and we will show it is true for $n = k + 1$.

Let A be an alphabet with $k + 1$ letters and frequency function f . Let H_A be the prefix tree resulting from applying the Huffman algorithm to A and f . Let y, z be the two characters chosen first by the algorithm.

We know by the Optimal Lemma that there is an optimal tree OPT_A such that y, z are deepest siblings. Note that the Lemma doesn't guarantee exactly that but it guarantees that two characters with minimal frequencies are deepest siblings so we can just swap those two with y, z without changing $s(OPT_A)$.

We claim that $s(H_A) = s(OPT_A)$.

Define the alphabet $B = A - \{y, z\} \cup \{\alpha\}$ where α is some new character and with $f(\alpha) = f(y) + f(z)$.

Define OPT_B to be the tree obtained by removing $\{y, z\}$ from OPT_A and

assigning α to their parent. Note that:

$$\begin{aligned}
s(OPT_A) &= \sum_{x \in A} f(x)d(x, A) \\
&= f(y)d(y, A) + f(z)d(z, A) + \sum_{x \in A - \{y, z\}} f(x)d(x, A) \\
&= f(y)d(y, A) + f(z)d(y, A) + \sum_{x \in A - \{y, z\}} f(x)d(x, A) \\
&= f(\alpha)d(y, A) + \sum_{x \in A - \{y, z\}} f(x)d(x, A) \\
&= f(\alpha)(d(\alpha, A) + 1) + \sum_{x \in A - \{y, z\}} f(x)d(x, B) \\
&= f(\alpha) + f(\alpha)d(\alpha, A) + \sum_{x \in A - \{y, z\}} f(x)d(x, B) \\
&= f(\alpha) + \sum_{x \in B} f(x)d(x, B) \\
&= f(y) + f(z) + \sum_{x \in B} f(x)d(x, B) \\
&= f(y) + f(z) + s(OPT_B)
\end{aligned}$$

Define H_B to be the tree obtained by removing $\{y, z\}$ from H_A and assigning α to their parent. Note that $s(H_A) = s(H_B) + f(y) + f(z)$ by a similar calculation as the above.

In addition consider that when Huffman is applied to A and f we chose y and z first and combined them to form a new tree with frequency $f(y) + f(z)$ and then continue with Huffman. The remaining process is algorithmically equivalent to starting with the alphabet B and $f(\alpha) = f(y) + f(z)$ and hence H_B can be thought of as the result of applying the Huffman algorithm to B and f , and hence since B has k characters we have $s(H_B) = s(OPT_B)$ by the inductive assumption.

Then we have:

$$s(H_A) = S(H_B) + f(y) + f(z) \leq s(OPT_B) + f(y) + f(z) = s(OPT_A)$$

Since OPT_A is optimal for A we then must have $s(H_A) = s(OPT_A)$ and so the tree created by the Huffman algorithm is optimal.

CMSC 351: P, NP, Etc. Part 1

Justin Wyss-Gallifent

August 15, 2022

1	Introduction	2
2	Informalities	2
3	Decision Problems	3
3.1	What is a Decision Problem?	3
3.2	Rephrasing as Decision Problems	3
3.3	Proofs and Witnesses	4
4	Turing Machines	5
4.1	Basics	5
4.2	Deterministic	5
4.3	Non-Deterministic	5
5	Thoughts, Problems, Ideas	7

1 Introduction

Here we're going to abstract our knowledge of time complexity to look at classifying problems. In order to do this we need to introduce the concepts of decision problems, Turing machines, etc.

2 Informalities

A great example to consider is the generalization of Sudoku. Starting with an $n^2 \times n^2$ grid with some squares filled with values between 1 and n^2 , is it possible to complete the grid with numbers between 1 and n^2 so that each row and each column contains exactly one of each and such that each of the n^2 subsquares of size $n \times n$ contains exactly one of each?

Example 2.1. Classic Sudoku is $3^2 \times 3^2 = 9 \times 9$, the numbers are between 1 and $3^2 = 9$, and there are $3^2 = 9$ subsquares of size 3×3 . ■

Example 2.2. Simple Sudoku is $2^2 \times 2^2 = 4 \times 4$, the numbers are between 1 and $2^2 = 4$, and there are $2^2 = 4$ subsquares of size 2×2 . ■

Example 2.3. More complicated is $4^2 \times 4^2 = 16 \times 16$, the numbers are between 1 and $4^2 = 16$, and there are $4^2 = 16$ subsquares of size 4×4 . ■

If a Sudoku board is partially filled in we can ask whether it can be completely filled in.

If a suggested solution is given it is easy to check if the solution is valid and this check can be performed in an amount of time which grows at a polynomial rate with the size of the board. Can you write an algorithm for this?

However actually coming up with a solution seems to be much harder. It appears that (although nobody has proven this) that the amount of time grows at an exponential rate with the size of the board and that nothing is really any better than actually trying all combinations.

Thus we might informally suggest that checking a given solution is easier than finding a solution.

3 Decision Problems

3.1 What is a Decision Problem?

Definition 3.1.1. A *decision problem* is a problem which takes an input, formally an *instance*, and produces either YES or NO.

We'll often abstractly write Q for a decision problem and I for an instance and then $Q(I) = \text{YES}$ or $Q(I) = \text{NO}$ depending on whether the result is YES or NO for that instance.

Example 3.1. Q: Given two real numbers x and y , is the sum greater than 100?

I: $x = 50, y = 80$ yields $Q(I) = \text{YES}$
I: $x = 10, y = 10$ yields $Q(I) = \text{NO}$

■

Example 3.2. Q: Given a natural number n , does n have nontrivial factors?

I: $n = 10$ yields $Q(I) = \text{YES}$
I: $n = 5$ yields $Q(I) = \text{NO}$

■

Example 3.3. Q: Given a list of integers A , is it sorted?

I: $A = [1, 2, 3]$ yields $Q(I) = \text{YES}$
I: $A = [2, 1, 3]$ yields $Q(I) = \text{NO}$

■

Example 3.4. Q: Given a graph G , does it contain a cycle?

■

Example 3.5. Q: Given a set A is there a subset which adds to 0?

I: $A = \{1, 4, -3, 5, -1\}$ yields $Q(I) = \text{YES}$
I: $A = \{1, 4, 5, 3, -2\}$ yields $Q(I) = \text{NO}$

■

3.2 Rephrasing as Decision Problems

Many types of problems which are not decision problems can be rephrased as decision problems. The rephrased problem will not be exactly the same but will carry over the same notion in a computationally equivalent sense.

Example 3.6. Non-decision problem: Find a solution to a partially filled Sudoku board.

This can be rephrased as:

Q: Given a partially filled Sudoku board, does a solution exist?

Observe that deciding if a solution exists essentially seems to be as difficult as finding one.

■

Example 3.7. Non-decision problem: For a graph G , find the shortest path between two vertices s and t .

This can be rephrased as:

Q: Given two vertices s and t and a length k , is there a path of length less than or equal to k between s and t ?

Observe that deciding if there is a path of length less than or equal to k essentially seems to be as difficult as finding one. ■

3.3 Proofs and Witnesses

Definition 3.3.1. Given a decision problem and an instance if the answer is YES then a *proof*, or *witness*, or *certificate* is essentially proof that the answer is yes.

Informally we'll use the term *invalid witness* to refer to something which doesn't work.

Example 3.8. Q: Given a set, is there a subset which adds to 0?

I: $\{1, 4, -3, 5, -1\}$ has $\{4, -3, -1\}$ as a witness. ■

Note 3.3.1. If we have access to a valid witness for a decision problem and instance then we can say that the answer is YES. However if we don't have access to a witness or if we have an invalid witness then we cannot say that the answer is no.

Example 3.9. For the decision problem: Given a set, is there a subset which adds to 0?

For the instance $\{1, 4, -3, 5, -1\}$ if we present $\{1, 4\}$ then this is an invalid witness and we know nothing. ■

4 Turing Machines

4.1 Basics

Definition 4.1.1. A *Turing machine* consists of:

- An infinitely long (in both directions) tape divided into cells. Each cell is either blank or contains one of a finite set of symbols.
- A read-write head which points to one cell at a time. It can read the cell or write to it, or it can move one cell left or right.
- A current state taken from a finite set of possible states and which is initialized at the start.
- A finite table of instructions.

A Turing machine is given a starting state and then it proceeds to run according to its state, its set of rules, and the tape. It may then stop operating at some ending state.

While this may seem particularly simple in reality every single algorithm that we generally think of and work with can be modeled by a particular Turing machine.

4.2 Deterministic

Definition 4.2.1. A *deterministic Turing machine* (DTM) is a Turing machine which has a single course of action for any combination of its internal state state and the input/memory. In such a case one starting state will result in one ending state.

Note 4.2.1. All of the thinking we've been doing in this class is based around the idea of a DTM.

4.3 Non-Deterministic

Definition 4.3.1. A *non-deterministic Turing machine* (NTM) is a Turing machine which can have more than one course of action for any combination of its internal state and the input/memory. In such a case one starting state can (but doesn't have to) result in many ending states.

Note 4.3.1. Understand that NTM should be treated as a thought experiment used to study computing rather than a specific machine on which algorithms are actually run.

Example 4.1. If we are trying to solve the decision problem as to whether there is a subset of $\{-1, -2, 3, 4\}$ then a DTM needs to check all possible subsets one by one whereas a NTM can try them simultaneously.

This means it could add all subsets simultaneously, or it could compare all subsets against a different set simultaneously, or whatever. The key point is that these operations happen simultaneously. ■

Example 4.2. If a DTM is doing a breadth-first search on a graph then at a given vertex it must check each adjacent vertex one by one in some order. An NTM can check them all simultaneously and the result is many ending states, one corresponding to each branch. ■

Example 4.3. If a DTM is trying to factor a number by brute force then it must try every factor. An NTM can try them all simultaneously and the result is many ending states, one corresponding to each result. ■

Time complexity can of course then change drastically when we are thinking about DTMs v NTMs.

Example 4.4. If a problem of size n involves a star graph with $n + 1$ vertices (one central vertex connected to n other vertices, none of which are connected to each other) and each of the n outer vertices require a $\Theta(1)$ calculation then a DTM renders the problem $\Theta(1 + n) = \Theta(n)$ as it has to check the n vertices one by one but a NTM renders the problem $\Theta(1 + 1) = \Theta(1)$ as it can check them simultaneously by branching. ■

A NDM can obviously emulate a DTM because it can be instructed to simply follow one course of action. A DTM can to some degree emulate a NDM but only in a breadth-first sense. It cannot follow all possibilities to all depths simultaneously.

5 Thoughts, Problems, Ideas

1. Prove that if an algorithm has time complexity $T(n) = n!$ then the algorithm does not run in polynomial time.
2. Consider this decision problem:

YES \vee NO: Given a set S with n elements, is there a subset which adds to zero?

For each of the following inputs, first answer YES or NO. If the answer is YES, give an example of a valid witness and an invalid witness.

- (a) $S = \{5, 1, 8, -2, 10, -2, 100, 7, -2\}$
- (b) $S = \{-10, 20, 1, 2, 3, 6, 80, -11, 4\}$
- (c) $S = \{-2, -1, 4\}$

3. Consider this decision problem:

YES \vee NO: Does the integer $n \geq 2$ have a factor which is not 1 or itself?

For each of the following inputs, first answer YES or NO. If the answer is YES, give an example of a valid witness and an invalid witness.

- (a) $n = 100$
- (b) $n = 97$
- (c) $n = 51$

4. Consider this decision problem:

YES \vee NO: Does the graph with n vertices labeled 0 through $n - 1$ and represented by a given adjacency list have a cycle?

For each of the following inputs, first answer YES or NO. If the answer is YES, give an example of a valid witness and an invalid witness.

- (a) $[[1, 3], [0, 2], [1, 3], [0, 2]]$
- (b) $[[1, 2, 3], [0], [0], [0]]$
- (c) $[[1, 2], [0, 3, 4], [0], [1], [1]]$

CMSC 351: P, NP, Etc. Part 2

Justin Wyss-Gallifent

December 2, 2023

1	Polynomial Time	2
2	P	3
3	NP	6
	3.1 Definition	6
	3.2 An Algorithm for a Verifier which Proves NP	8
4	P v NP	9
5	Problem Reduction and Equivalence	10
	5.1 Introduction	10
	5.2 Using Explicit Sets	10
	5.3 Decision Problems and Algorithms	12
	5.4 Stepping Away from Decision Problems	14
	5.5 Some Facts about Polynomial Reducibility	14
6	Thoughts, Problems, Ideas	15

1 Polynomial Time

A reminder:

Definition 1.0.1. An algorithm runs in *polynomial time* if $T(n) = \mathcal{O}(n^k)$ for some $k \in \mathbb{N}$ where n is the input size.

Example 1.1. MergeSort has $T(n) = \Theta(n \lg n) = \mathcal{O}(n^2)$ and hence MergeSort is polynomial time. ■

Example 1.2. Generating a list of all permutations of $\{1, 2, \dots, n\}$ has $T(n) = \Theta(n!)$ which is not polynomial time. Why not? Can you prove this? ■

Generally speaking we think of polynomial time as “fast” but depending on the coefficients, in reality polynomial time can be incredibly slow.

2 P

Definition 2.0.1. The set P is the set of all decision problems Q such that $Q \in P$ when there is a DTM polynomial-time algorithm (polynomial in the size of the instance) such that:

- If $Q(I) = YES$ then the algorithm outputs YES .
- If $Q(I) = NO$ then the algorithm outputs NO .

Example 2.1. Q: Given a list A , is A sorted?

We can write a polynomial-time algorithm which takes an instance (a list) A and checks whether each element is greater than or equal to the previous element and returns YES if they all are and NO otherwise.

Thus this decision problem is P . ■

Example 2.2. Q: Given x, y and d , is $d = \gcd(x, y)$?

We can write a polynomial-time algorithm which takes an instance (a triple x, y, d), calculates the gcd of x and y and compares it to d . If equal it returns YES and otherwise NO .

Thus this decision problem is P . ■

Example 2.3. Q: Given two lists A and B of the same length, do they contain the same values?

We can write a polynomial-time algorithm which takes an instance (two lists A and B) sorts them and then compares them element by element. If all elements are equal, return YES , otherwise return NO .

Thus this decision problem is P . ■

Example 2.4. Q: Given a graph G on V vertices and two specific vertices s and t and a distance d , is there a path of length less than or equal to d from s to t ?

We can write a polynomial-time algorithm which takes an instance (the data G, V, s, t, d), runs the shortest path algorithm using s as the starting vertex and checks if the distance from s to t is less than d and replies YES or NO accordingly.

Thus this decision problem is P . ■

Example 2.5. Q: Given a partially filled Sudoku board, is there a solution?

There is no known polynomial-time algorithm which takes an instance (a partially filled Sudoku board) and replies *YES* or *NO* accordingly.

Thus it is unknown if this decision problem is P . It is suspected that the answer is no. ■

Example 2.6. Q: Given a set A is there a subset which adds to 0?

There is no known polynomial-time algorithm which takes an instance (a set A) and replies *YES* or *NO* accordingly.

Thus it is unknown if this decision problem is P . It is suspected that the answer is no. ■

Example 2.7. Q: Given a graph G , does G contain a Hamiltonian cycle? (This is a cycle which contains each vertex exactly once.)

There is no known polynomial-time algorithm which takes any instance (a graph G) and replies *YES* or *NO* accordingly.

Thus it is unknown if this decision problem is P . It is suspected that the answer is no. ■

Example 2.8. Q: Given a program which may or may halt, does it halt?

It has been proven that there is no polynomial-time algorithm which takes any instance (a program) and replies *YES* or *NO* accordingly.

Thus this decision problem is not in P . ■

In many cases if a decision version of an optimization problem is P then the optimization problem itself can be solved in polynomial time.

Example 2.9. Given a weighted connected graph G and vertices s, t consider the optimization problem of finding the length of the shortest path from s to t .

A decision version of this is:

Q: Given a weighted connected graph T , vertices s, t , and a value k is there a path from s to t of length less than or equal to k ?

Suppose we have a polynomial-time algorithm `pathexists(G,s,t,k)` which answers the decision version in polynomial time. In other words within polynomial time it returns *YES* if there is a path from s to t of length less than or equal to k and *NO* if not.

Since the graph is connected we know there is a path from s to t and the length of this path could at most be the total sum of all the edge weights. So what we can do is:

```
function findshortestpathlength(G,s,t,k)
    max = sum of edge weights in G
    shortest = max
    for i = max down to 0
        if pathexists(G,s,t,i) then
            shortest = i
        end
    end
    return(shortest)
end
```

This function will return the length of the shortest path and will do so in polynomial time on a DTM.

Thus this optimization process can be solved in polynomial time. ■

3 NP

3.1 Definition

Before talking about NP , here is P again so we can compare:

Definition 3.1.1. The set P is the set of all decision problems Q such that $Q \in P$ when there is a DTM polynomial-time algorithm (polynomial in the size of the instance) such that:

- If $Q(I) = YES$ then the algorithm outputs YES .
- If $Q(I) = NO$ then the algorithm outputs NO .

And now NP :

Definition 3.1.2. The set NP is the set of all decision problems Q such that $Q \in NP$ when there is a NTM polynomial-time algorithm (polynomial in the size of the instance) such that:

- If $Q(I) = YES$ then the algorithm outputs YES .
- If $Q(I) = NO$ then the algorithm outputs NO .

Now then, let's stop to point out that this was the original definition of NP but there's an equivalent and more modern definition of NP which is based upon verification of solutions. Here is the simple version:

Definition 3.1.3. The set NP is the set of all decision problems Q such that $Q \in NP$ when there is a DTM polynomial-time algorithm V called a *verifier* that verifies an instance and potential witness in polynomial time (polynomial in the size of the instance and witness). This means for an instance I and potential witness x that:

- If $V(I, x) = YES$ then there is a witness, although it may not necessarily be x .
- If $V(I, x) = NO$ then x is not a witness.

Note 3.1.1. The idea is that V can use x but doesn't have to. The key is that it must run in polynomial time.

Note 3.1.2. A small warning here that neither bullet point is an iff. This means that if there is a witness we don't necessarily get YES from V and if x is not a witness we don't necessarily get NO from V .

This warning is clarified by the following two examples:

Example 3.1. Q: Given a set of n integers is there a subset which adds to 0?

The verifier $V(I, x)$ takes the set and a particular subset. It sums the values in x and sees if the result is 0 and returns YES or NO accordingly. The following calls yield the following results:

We find $V(\{3, 4, 2, -7\}, \{3, 4, -7\}) = YES$ and we can conclude there is a witness.

We find $V(\{3, 4, 2, -7\}, \{3, 4\}) = NO$ and we can conclude that $\{3, 4\}$ is not a witness. This does not mean there isn't a witness. In this case there is. ■

Example 3.2. Q: Given a list of n integers are any of them greater than 100?

The verifier $V(I, x)$ takes the list and a particular element. It checks the list and sees if there is a value greater than 100 and return YES or NO accordingly. It ignores the particular element. The following calls yield the following results:

We find $V(\{1, 5, 103, 10\}, 103) = YES$ and we can conclude there is a witness.

We find $V(\{1, 5, 103, 10\}, 10) = YES$ and we can conclude there is a witness.

We find $V(\{1, 5, 3, 10\}, 10) = NO$ and we can conclude that 10 is not a witness. This does not mean there isn't a witness. In this case there isn't. ■

Here are some other examples.

Example 3.3. Q: Given a set S of integers can we partition S into two subsets A and B whose sums are equal?

We can write a polynomial-time verifier which takes an instance (a set S) and a potential witness (two subsets A and B), sums the elements in A and B separately, checks if they are equal and replies YES or NO accordingly. This will satisfy the definition.

Thus this decision problem is NP .

Example 3.4. Q: Given a partially-filled sudoku board is there a solution?

We can write a polynomial-time verifier which takes an instance (a partially-filled sudoku board) and a potential witness (a potential solution) and checks if the solution works, replying YES or NO accordingly. This will satisfy the definition.

Thus this decision problem is NP . ■

Some other problems which can be seen to be NP in a similar way:

Example 3.5. Q: Given a set of n integers, is there a subset which adds to 0? ■

| **Example 3.6.** Q: Given a graph G does it contain a cycle? ■

Note 3.1.3. I read somewhere once that some people believe that we should just use VP to mean verifiable in polynomial time on a DTM instead of using NP . That way the machine is always a DTM.

Or even better, SP (for solvable in polynomial time) and VP (verifiable in polynomial time). But there we go.

3.2 An Algorithm for a Verifier which Proves NP

This is an informal presentation of how we might write a verifier which proves a decision problem Q is in NP . The following algorithm will need to run in polynomial time as a function of the sizes of I and x :

```
function V(I,x)
    if we can decide Q(I) without looking at x
        decide Q(I)
        return YES or NO accordingly
    else
        if x is a valid witness?
            return YES
        end if
    end if
    return NO
```

Now pretend that all you know is the input and output of the verifier. Observe that:

- If the verifier returns YES then all you can conclude is that there is a valid witness. You don't know if x is valid because you don't know if the algorithm decided Q or checked x .
- If the verifier returns NO then all you know is that x is not a valid witness. You don't know if there is a valid witness

This algorithm satisfies the definition of a polynomial-time verifier and proves that $Q \in NP$.

4 P v NP

Note 4.0.1. First, observe that $P \subseteq NP$. To see this, note that if a decision problem is in P then if we are given an instance and a witness we can just ignore the witness, run the polynomial-time decider and say *YES* or *NO* accordingly.

Definition 4.0.1. The *P v NP Problem* asks whether $P = NP$ or not. In other words is it the case that when we can verify any potential witness in polynomial time that we can also solve the decision problem in polynomial time?

This is perhaps the greatest unsolved problem in computer science. There is overwhelming evidence that $P \neq NP$ in the sense that there are many important problems for which potential witnesses can be verified in polynomial time but no polynomial-time solution has been found. However note that this does not mean that such solutions don't exist.

5 Problem Reduction and Equivalence

5.1 Introduction

This last section looks at what it might mean if solving one decision problem might be “as easy as” solving another. Before diving into this concept we’ll first look at a similar idea with sets.

5.2 Using Explicit Sets

First of all observe that a set A can be thought of as a decision problem Q if we treat the elements in A as instances and say that $Q(I) = YES$ if $I \in A$ and $Q(I) = NO$ if $I \notin A$.

| **Example 5.1.** If $A = \{2, 5, 10\}$ then $Q(2) = YES$ and $Q(3) = NO$. ■

| **Example 5.2.** Let A be the set of integer multiple of 3. Then $Q(6) = YES$ and $Q(5) = NO$. ■

| **Example 5.3.** Let A be the set of partially filled sudoku boards which are solvable. Clearly there are some partially filled boards x for which $Q(x) = TRUE$ and some for which $Q(x) = FALSE$. ■

Definition 5.2.1. Given sets A and B we say that A is *polynomially reducible* to B if there is some function p which can be computed in polynomial time and such that $x \in A$ iff $p(x) \in B$.

Note 5.2.1. The idea here is that making a decision about whether $x \in A$ or not can be, in polynomial time, altered to a question about whether $p(x) \in B$ or not.

| **Example 5.4.** Let A be the set of integer multiples of 2 and let B be the set of integer multiples of 3.

To prove that A is polynomially reducible to B we define p by $p(x) = 3x/2$. Integer multiplication is a polynomial-time procedure.

Observe that:

\implies : If $x \in A$ then $x = 2k$ for some $k \in \mathbb{Z}$ and then $p(x) = 3x/2 = 3(2k)/2 = 3k$ so $p(x) \in B$.

\impliedby : If $p(x) \in B$ then $p(x) = 3k$ for some $k \in \mathbb{Z}$ and then $3x/2 = 3k$ so $x = 2k$ so $x \in A$. ■

Example 5.5. Define the sets A and B as follows:

$$A = \{s \mid s \text{ is a string}\}$$
$$B = \{s \mid s \text{ is a string ending with "Z"}\}$$

To prove that A is polynomially reducible to B we define p by $p(s) = s + "Z"$ where $+$ is string concatenation. String concatenation is a polynomial-time procedure.

Observe that:

\Rightarrow : If $s \in A$ then $p(s) = s + "Z"$ ends with "Z" so $p(s) \in B$.

\Leftarrow : If $p(s) \in B$ then $p(s)$ ends with "Z". Thus $s + "Z"$ ends with "Z" and so s is a string so $s \in A$.

■

5.3 Decision Problems and Algorithms

Now let's work this up to some algorithms. First, here's the definition:

Definition 5.3.1. For decision problems Q_1 and Q_2 We say that Q_1 is *polynomially reducible* to Q_2 if there is some algorithm p which runs in polynomial time which converts instances for Q_1 to instances for Q_2 such that $Q_1(I) = YES$ iff $Q_2(p(I)) = YES$.

If this is the case then we'll write:

$$Q_1 \leq_P Q_2$$

Note 5.3.1. It's not important how long Q_2 takes and sometimes Q_2 is called an *oracle* instead to try to impart the idea that it just knows with no work at all.

Example 5.6. Suppose a function `ORACLE(n)` decides something and returns YES or NO. Consider the following algorithm for `QUESTION(n)`:

```
function QUESTION(n)
    for i = 1 to n
        if ORACLE(i)
            return(YES)
    end
    end
    return(NO)
end
```

Observe that `QUESTION` is polynomially reducible to `ORACLE`. We would thus write $QUESTION \leq_P ORACLE$.

Note that there might be other algorithms that do whatever `QUESTION` does and they may do it faster, but we don't know. What we do know, however, is that this algorithm for `QUESTION` reduces the problem to `ORACLE` in polynomial time so it's essentially "no harder than" `ORACLE(n)`. ■ ■

Here is another example:

Example 5.7. Consider these two decision problems:

ISHAMILTON(G): Given an undirected graph on n vertices, is there a Hamiltonian cycle in the graph?

ORACLE(G): Given a directed graph on n vertices, is there a Hamiltonian cycle in the graph?

Given a undirected graph G the adjacency matrix for G also represents the

adjacency matrix for G' where G' is obtained from G by replacing each (undirected) edge with two edges, one in each direction. This takes no time. We can then apply $\text{ORACLE}(G)$ to find a Hamiltonian cycle in G' which is also a Hamiltonian Cycle in G .

It follows that $\text{ISHAMILTON} \leq_P \text{ORACLE}$.

Now then, it is widely believed that $\text{ISHAMILTON} \notin P$ and so if this is true, then $\text{ORACLE} \notin P$ also. ■

And one more:

Example 5.8. Consider these two decision problems:

$\text{ISZEROSUBSET}(S)$: Given a set S of n integers is there a subset which adds to 0?

$\text{ORACLE}(S, x)$: Given a set S of n integers and one integer x in the set, is there a subset of $S - \{x\}$ which adds to $-x$?

To see this suppose we have a set S of integers. We iterate through each element x of S and for each we ask if $\text{ORACLE}(S, x) == \text{YES}$. If it is true for at least one x then we return YES for $\text{ISZEROSUBSET}(S)$ and otherwise we return NO .

It follows that $\text{ISZEROSUBSET} \leq_P \text{ORACLE}$. ■

5.4 Stepping Away from Decision Problems

This same idea of polynomial reduction can then apply to problems which are not decision problems.

Example 5.9. Consider the non-decision problem:

SOLVE(B): Given a partially filled $n^2 \times n^2$ sudoku board B which has a solution, find it.

And the decision problem:

ORACLE(B, x, y, v): Given a partially filled $n^2 \times n^2$ sudoku board B , is there a solution with the value v placed into the empty space with coordinates (x, y) ?

Observe that if we are given a partially filled sudoku board we can iterate through the empty spaces and for each one we can test values 1 through n^2 using *ORACLE*. We know there's a solution so for each empty space we will find a value which works so we add this to our solution. At the end we have solved it. Note that there are at fewer than $(n^2)(n^2) = n^4$ empty spaces and we have to test at most n^2 values in each space so this is $\mathcal{O}(n^6)$.

Thus we can calculate *SOLVE* in polynomial time as a function of *ORACLE*.

5.5 Some Facts about Polynomial Reducibility

Now we can formalize some facts for decision problems Q_1 and Q_2 :

- If $Q_1 \leq_P Q_2$ and $Q_2 \in P$ then $Q_1 \in P$.
- If $Q_1 \leq_P Q_2$ and $Q_1 \notin P$ then $Q_2 \notin P$.

For the mathematicians here, a nice way of thinking about this is to imagine three functions $q_1(x)$, $q_2(x)$, and $p(x)$. Suppose we know for sure that $p(x)$ is a polynomial and we know that $q_1(x) = p(q_2(x))$.

Then we can say:

- q_1 is a polynomial in terms of q_1 .
- If $q_2(x)$ is a polynomial then $q_1(x)$ is a polynomial.
- If $q_1(x)$ is not a polynomial then $q_2(x)$ is not a polynomial.

Note that if $p(x)$ is not a polynomial we can say nothing.

6 Thoughts, Problems, Ideas

1. Explain how you know that the following decision problems are in P . You don't need to provide pseudocode, a basic explanation will suffice.
 - (a) $Y \vee N$: Given a list with n elements, is it unsorted?
 - (b) $Y \vee N$: Given the adjacency matrix for a graph with n vertices, is there one vertex which is connected to all the others?
 - (c) $Y \vee N$: Given base-10 list representations of two n digit numbers A and B , is $AB \geq 5 \cdot 10^{2n-2}$?
For example is $84 \cdot 23 \geq 58 \cdot 10^2 = 500$?
 - (d) $Y \vee N$: Given a list with n elements, is the maximum to the left of the minimum?
2. For each of the following you are given a problem $PROB$ and an associated decision problem DEC . For each, write pseudocode to show that if $DEC \in P$ then $PROB$ can be solved in polynomial time.
Note: Don't worry about whether or not it's true in the real world that $DEC \in P$, just assume it is and base your pseudocode on it.
 - (a) Given a simple connected unweighted graph with n vertices.
 $PROB$: Find length of the longest path.
 DEC : For any given k , is there a path of length k ?
 - (b) Given a list A of n integers, a subset $S \subset A$, and a target t .
 $PROB$: Assuming there is a subset of A containing S which sums to t , find it.
 DEC : Is there a subset of A containing S which sums to t ?
 - (c) Given an integer $n \geq 2$.
 $PROB$: Find the smallest prime factor of n .
 $PROB$: For any given k , is k prime?
3. Explain why reverse-sorting a list is polynomially reducible to sorting a list.
4. Suppose a garage contains n motorcycles each of which has 1, 2 or 4 cylinders. Explain why fixing all cylinders on all motorcycles is polynomially reducible to fixing one cylinder.
5. Suppose G is a simple graph with n vertices. Explain why counting the edges in the graph is polynomially reducible to calculating the degree of a vertex.

CMSC 351: Review

Justin Wyss-Gallifent

January 28, 2021

1	Proofs	2
1.1	Weak Induction	2
1.2	Strong Induction	2
1.3	Constructive Induction	2
1.4	Structural Induction	5
2	Combinatorics	6
2.1	Permutations and Combinations	6
2.2	Probability and Expected Value	6
3	Calculus Thread	7
3.1	Sequences and Sums	7
3.2	L'hôpital's Rule	9
3.3	Manipulation of Logarithms	10
3.4	Differentiation	11
3.5	Integration	11
3.6	Integral Bounds for Sums	12
4	Thoughts, Problems, Ideas	14

1 Proofs

1.1 Weak Induction

To prove $\forall n \geq n_0 P(n)$ we first prove $P(n_0)$ (this is the base case) and then we prove $\forall k \geq n_0 P(k) \rightarrow P(k + 1)$ (this is the inductive step). The assumption of $P(k)$ in the inductive step is the inductive hypothesis.

1.2 Strong Induction

The inductive hypothesis becomes $\forall k \geq n_0, P(n_0) \wedge P(n_0 + 1) \wedge \dots \wedge P(k) \rightarrow P(k + 1)$. We often need more than one base case. The quantify we need can be determined by examining how far back we go in the inductive step. If the inductive step refers back to $P(j)$ for $j < k$ then we must have $j \geq n_0$.

1.3 Constructive Induction

Useful when we have an idea (a guess) about a formula but we need to figure out some constants. We verify our guess while simultaneously finding the constants.

Example 1.1. Suppose we suspect that:

$$\sum_{k=1}^n k = an^2 + bn$$

To use constructive induction we first note that if this is going to be true for the base case then we need:

$$1 = \sum_{k=1}^1 k = a(1)^2 + b(1)$$

Thus we know $a + b = 1$ and so $b = 1 - a$. Thus we can now suspect that:

$$\sum_{k=1}^n k = an^2 + (1 - a)n$$

Then we note that if this is going to be true for the inductive step then we need:

$$\text{If } \sum_{k=1}^n k = an^2 + (1 - a)n \text{ then } \sum_{k=1}^{n+1} k = a(n+1)^2 + (1 - a)(n+1).$$

Well observe that:

$$\sum_{k=1}^{n+1} k = \left[\sum_{k=1}^n k \right] + (n+1) = an^2 + (1 - a)n + (n+1)$$

Thus we need:

$$\begin{aligned}
an^2 + (1-a)n + (n+1) &= a(n+1)^2 + (1-a)(n+1) \\
an^2 + n - an + n + 1 &= an^2 + 2an + a + n - an + 1 - a \\
an^2 + (2-a)n + 1 &= an^2 + (a+1)n + 1 \\
(2-a)n &= (a+1)n \\
2-a &= a+1 \\
2a &= 1 \\
a &= \frac{1}{2}
\end{aligned}$$

Thus we have:

$$\sum_{k=1}^n k = \frac{1}{2}n^2 + \frac{1}{2}n$$

This is our familiar:

$$\sum_{k=1}^n k = \frac{n(n+1)}{2}$$

There are two issues that often get questioned and while they're not issues we'll really worry about it's certainly worth addressing them.

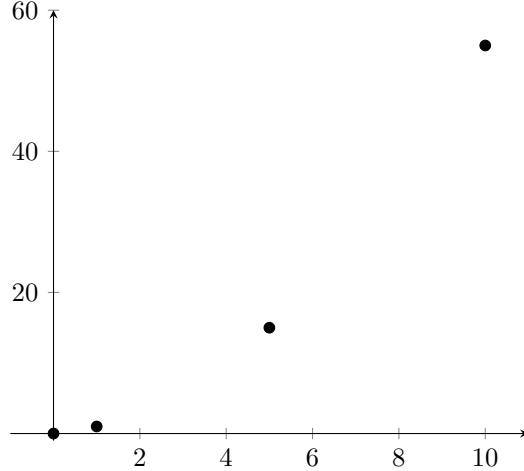
First, where does our guess come from? This is part art and part science and depends highly on the situation. Let's consider our example sum but let's start at 0:

$$\sum_{k=0}^n k$$

Suppose we do some tests:

$$\begin{aligned}
\sum_{k=0}^0 k &= 0 \\
\sum_{k=0}^1 k &= 1 \\
\sum_{k=0}^5 k &= 15 \\
\sum_{k=0}^{10} k &= 55
\end{aligned}$$

If we plot these we perhaps guess that the result is quadratic with a y -intercept of 0:



Because of this we might guess that:

$$\sum_{k=0}^n k = an^2 + bn$$

Second, what happens if we make a wrong guess? Let's see. Suppose we suspect it's cubic instead:

$$\sum_{k=0}^n k = an^3 + bn$$

The base case $n = 0$ tells us $0 = 0 + 0$ or nothing.

The case $n = 1$ tells us that $1 = a + b$ so $b = 1 - a$ as before. So far so good.
Thus we can now suspect that:

$$\sum_{k=1}^n k = an^3 + (1 - a)n$$

Then we note that if this is going to be true for the inductive step then we need:

$$\text{If } \sum_{k=1}^n k = 3n^2 + (1 - a)n \text{ then } \sum_{k=1}^{n+1} k = a(n+1)^3 + (1 - a)(n+1).$$

Well observe that:

$$\sum_{k=1}^{n+1} k = \left[\sum_{k=1}^n k \right] + (n+1) = an^3 + (1 - a)n + (n+1)$$

Thus we need:

$$\begin{aligned} an^3 + (1-a)n + (n+1) &= a(n+1)^3 + (1-a)(n+1) \\ an^3 + (2-a)n + 1 &= a(n^3 + 3n^2 + 3n + 1) + (n - an + 1 - a) \end{aligned}$$

This needs to be true for all $n \geq 0$ which basically means they need to be equal as polynomials with n being the variable. But we have a problem because there is an n^2 on the right but not on the left. Thus we have chosen...poorly.

1.4 Structural Induction

Used when we are trying to prove some property about all items in a set where the set is defined recursively. We prove that the property holds for the base items and then we prove that the recursive addition of new items preserves the property.

Example 1.2. We define a binary tree as following:

- A single node is a binary tree.
- If B_1 and B_2 are binary trees then a single node as parent to B_1 and B_2 is also a binary tree.

Let's show that the number of nodes in a binary tree $N(T)$ satisfies $N(T) \leq 2^{H(T)+1} - 1$ where $H(t)$ is the height.

Base Case: A binary tree T consisting of a single node has $N(t) = 1$ and $H(T) = 0$ and hence satisfies $1 \leq 2^{0+1} - 1$.

Inductive step: Suppose T is constructed by taking a single node as parent to B_1 and B_2 and suppose we have:

$$N(B_1) \leq 2^{H(B_1)+1} - 1 \text{ and } N(B_2) \leq 2^{H(B_2)+1} - 1$$

We claim that:

$$N(T) \leq 2^{H(T)+1} - 1$$

To see this, note that $N(T) = 1 + N(B_1) + N(B_2)$ and $H(T) = 1 + \max\{H(B_1), H(B_2)\}$. From here note that:

$$\begin{aligned} N(T) &= 1 + N(B_1) + N(B_2) \\ &\leq 1 + 2^{H(B_1)+1} - 1 + 2^{H(B_2)+1} - 1 \\ &\leq 2^{\max\{H(B_1), H(B_2)\}+1} + 2^{\max\{H(B_1), H(B_2)\}+1} - 1 \\ &\leq 2 \left(2^{\max\{H(B_1), H(B_2)\}+1} \right) - 1 \\ &\leq 2^{\max\{H(B_1), H(B_2)\}+1+1} - 1 \\ &\leq 2^{H(T)+1} - 1 \end{aligned}$$

This is as desired.

2 Combinatorics

2.1 Permutations and Combinations

Basic formulas:

$$\begin{aligned} n \text{ objects, permute } k &= \frac{n!}{(n-k)!} \\ n \text{ objects, choose } k &= \frac{n!}{k!(n-k)!} \\ n \text{ categories, permute } k &= n^k \end{aligned}$$

2.2 Probability and Expected Value

Suppose X is a random variable which takes on numerical outcomes x_1, \dots, x_n with respective probabilities p_1, \dots, p_n then the expected value of X is:

$$E(X) = p_1x_1 + \dots + p_nx_n$$

Example 2.1. Suppose an algorithm sorts the values in a list and returns the alternating sum/difference of the result. For example if you give it $[5, 8, 4, 1]$ it first sorts to get $[1, 4, 5, 8]$ and then returns $1 - 4 + 5 - 8 = -6$.

If the possible inputs to the algorithm are $[5, 8, 4, 1]$, $[10, 20, 0]$, $[2, 1]$ and $[0, 5, 2, -3]$ all equally likely, what is the expected outcome?

Well there are four outcomes:

$$\begin{aligned} [5, 8, 4, 1] &\Rightarrow [1, 4, 5, 8] \Rightarrow 1 - 4 + 5 - 8 = -6 \\ [10, 20, 0] &\Rightarrow [0, 10, 20] \Rightarrow 0 - 10 + 20 = 10 \\ [2, 1] &\Rightarrow [1, 2] \Rightarrow 1 - 2 = -1 \\ [0, 5, 2, -3] &\Rightarrow [-3, 0, 2, 5] \Rightarrow -3 - 0 + 2 - 5 = -6 \end{aligned}$$

Since all are equally likely they have probabilities 0.25 each and so the expected value is:

$$0.25(-6) + 0.25(10) + 0.25(-1) + 0.25(-6) = \dots$$

3 Calculus Thread

3.1 Sequences and Sums

Some basic sums:

$$\begin{aligned}\sum_{i=1}^n 1 &= n \\ \sum_{i=1}^n i &= \frac{n(n+1)}{2} \\ \sum_{i=1}^n i^2 &= \frac{n(n+1)(2n+1)}{6} \\ \sum_{i=0}^n r^i &= \frac{r^{n+1} - 1}{r - 1} \\ \sum_{i=0}^n 2^i &= 2^{n+1} - 1 \\ \sum_{i=1}^n i2^i &= (n-1)2^{n+1} + 2\end{aligned}$$

Note: Most of these should be familiar. the only one that might not be is the final one.

Proof. We have:

$$\begin{aligned}\sum_{i=1}^n i2^i &= 2 \left[\sum_{i=1}^n i2^i \right] - \left[\sum_{i=1}^n i2^i \right] \\ &= \left[\sum_{i=1}^n i2^{i+1} \right] - \left[\sum_{i=1}^n i2^i \right] \\ &= [1 \cdot 2^2 + 2 \cdot 2^3 + \dots + (n-1)2^n + n2^{n+1}] \\ &\quad - [1 \cdot 2^1 + 2 \cdot 2^2 + \dots + (n-1)2^{n-1} + n2^n] \\ &= n2^{n+1} - 2^n - 2^{n-1} - \dots - 2^2 - 2^1 \\ &= n2^{n+1} - (2^n + 2^{n-1} + \dots + 2^1) \\ &= n2^{n+1} - (2^{n+1} - 2) \\ &= (n-1)2^{n+1} + 2\end{aligned}$$

\mathcal{QED}

These can be used in various ways:

Example 3.1. Consider the sum:

$$\sum_{i=2}^n (2i + 2^{-i} + i^2)$$

We split it up:

$$\sum_{i=2}^n (2i + 2^{-i} + i^2) = \sum_{i=2}^n 2i + \sum_{i=2}^n 2^{-i} + \sum_{i=2}^n i^2$$

Separately these are:

$$\begin{aligned} \sum_{i=2}^n 2i &= 2 \sum_{i=2}^n i = 2 \left[\left[\sum_{i=1}^n i \right] - 1 \right] = 2 \left[\frac{n(n+1)}{2} - 1 \right] \\ \sum_{i=2}^n 2^{-i} &= \sum_{i=2}^n \left(\frac{1}{2} \right)^i = \left[\sum_{i=0}^n \left(\frac{1}{2} \right)^i \right] - 1 - \frac{1}{2} = \left[\frac{\left(\frac{1}{2} \right)^{n+1} - 1}{\frac{1}{2} - 1} \right] - \frac{1}{2} \\ \sum_{i=2}^n i^2 &= \left[\sum_{i=1}^n i^2 \right] - 1 = \left[\frac{n(n+1)(2n+1)}{6} \right] - 1 \end{aligned}$$

The result is then the sum of these:

$$2 \left[\frac{n(n+1)}{2} - 1 \right] + \left[\frac{\left(\frac{1}{2} \right)^{n+1} - 1}{\frac{1}{2} - 1} \right] - \frac{1}{2} + \left[\frac{n(n+1)(2n+1)}{6} \right] - 1$$

3.2 L'hôpital's Rule

It will be useful to remember two versions of L'hôpital's Rule:

Theorem 3.2.1. Suppose we are attempting to evaluate:

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)}$$

- If $\lim_{x \rightarrow \infty} f(x) = \lim_{x \rightarrow \infty} g(x) = 0$ then:

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)}$$

- If $\lim_{x \rightarrow \infty} f(x) = \lim_{x \rightarrow \infty} g(x) = \infty$ then:

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)}$$

Note 3.2.1. The theorem is valid for sequences as well as functions, we just treat f and g as continuous functions of n .

Example 3.2. We have:

$$\lim_{n \rightarrow \infty} \frac{n}{5n+1} = \lim_{n \rightarrow \infty} \frac{1}{5} = \frac{1}{5}$$

Oftentimes we'll need to use it repeatedly.

Example 3.3. We use it five times in a row here:

$$\lim_{n \rightarrow \infty} \frac{2^n}{n^5} = \lim_{n \rightarrow \infty} \frac{(\ln 2)^5 2^n}{(5)(4)(3)(2)(1)} = \infty$$

3.3 Manipulation of Logarithms

$$\log_b a = \frac{\log_c a}{\log_c b} \text{ (Change of Base)}$$

$$\log_b(xy) = \log_b x + \log_b y$$

$$\log_b \left(\frac{x}{y} \right) = \log_b x - \log_b y$$

$$\log_b (x^p) = p \log_b x$$

Note 3.3.1. We use the Change of Base all the time when we play fast and loose with big notation and logarithms. For example:

$$\log n = \frac{1}{\lg 10} \lg n = \Theta(\lg n) \text{ and } \lg n = \frac{1}{\log 10} \log n = \Theta(\log n)$$

Which gives an example of why logarithms don't matter for big notation.

3.4 Differentiation

Some basic rules:

$$\begin{aligned}\frac{d}{dx}x^r &= rx^{r-1} \text{ for } r \neq 0 \\ \frac{d}{dx}\ln x &= \frac{1}{x} \\ \frac{d}{dx}a^x &= (\ln a)a^x \\ \frac{d}{dx}f(x)g(x) &= f'(x)g(x) + f(x)g'(x)\end{aligned}$$

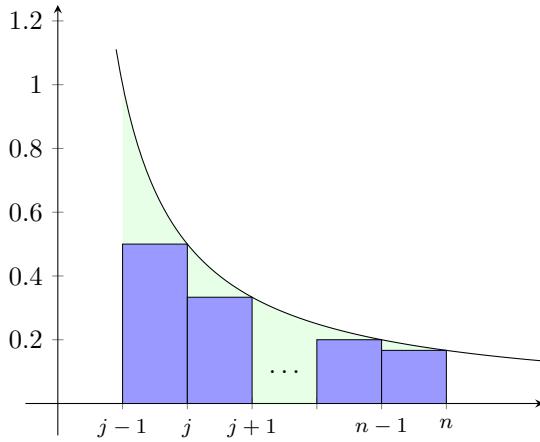
3.5 Integration

Some basic rules:

$$\begin{aligned}\int x^r &= \frac{1}{r+1}x^{r+1} + C \quad \text{for } r \neq -1 \\ \int x^{-1} &= \ln x + C \\ \int u \, dv &= uv - \int v \, du \text{ (Integration by Parts)}\end{aligned}$$

3.6 Integral Bounds for Sums

Consider the following picture of a decreasing function $f(x)$ and some rectangles below it:



It's clear from this picture that the sum of the areas of the rectangles is smaller than the area under the curve from $j - 1$ to n . If we define $a_i = f(i)$ then the sum of the areas of the rectangles (all have width 1) is:

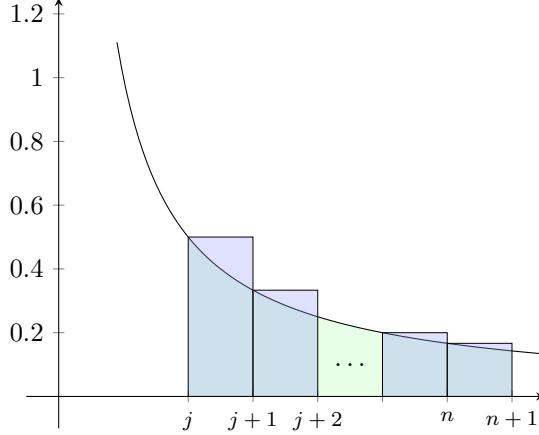
$$(1)f(j) + (1)f(j+1) + \dots + (1)f(n) = a_j + a_{j+1} + \dots + a_n$$

which is clearly smaller than the area under the curve between $x = j - 1$ and $x = n$.

Thus we have:

$$\sum_{i=j}^n a_i \leq \int_{j-1}^n f(x) dx$$

Likewise consider this picture with the rectangles all shifted to the right and the same function.



The sum of the areas of the rectangles (all have width 1) doesn't change, it still is:

$$(1)f(j) + (1)f(j+1) + \dots + (1)f(n) = a_j + a_{j+1} + \dots + a_n$$

which is clearly greater than the area under the curve between $x = j$ and $x = n + 1$.

Thus we have:

$$\int_{i=j}^{n+1} f(x) dx \leq \sum_{i=j}^n a_i$$

Together we have:

Theorem 3.6.1. As a general rule if a_i (and its corresponding $f(x)$ having $f(i) = a_i$) are decreasing then:

$$\int_j^{n+1} f(x) dx \leq \sum_{i=j}^n a_i \leq \int_{j-1}^n f(x) dx$$

Example 3.4. Suppose we want to get integral-related bounds for:

$$\sum_{i=3}^{100} \frac{1}{i}$$

Observe that we have:

$$\int_3^{101} \frac{1}{x} dx \leq \sum_{i=3}^{100} \frac{1}{i} \leq \int_2^{100} \frac{1}{x} dx$$

Calculating the left and right sides yields:

$$3.5166 \approx \ln(101) - \ln(3) \leq \sum_{i=2}^{100} \frac{1}{i} \leq \ln(100) - \ln(2) \approx 3.9120$$

Just for reference note that:

$$\sum_{i=3}^{100} \frac{1}{i} = 3.68737751\dots$$

Warning: Note that the sum is from $i = j$ to $i = n$ but one of the integrals goes from $j - 1$ and the other goes to $n + 1$. It's entirely possible that the function $f(x)$ is undefined at either $j - 1$ or $n + 1$ or both in which case we need to tweak a bit.

Example 3.5. Suppose we wanted an upper bound for:

$$\sum_{i=1}^{20} \frac{1}{i^2}$$

It might be tempting to simply use the right-hand inequality:

$$\sum_{i=1}^{20} \frac{1}{i^2} \leq \int_0^{20} \frac{1}{x^2} dx$$

but the function is undefined at $x = 0$. While the integral may still be (via improper integrals) it may not be and in any case is more work than we need. The approach is to simply separate out the first term of the sequence first:

$$\sum_{i=1}^{20} \frac{1}{i^2} = \frac{1}{1^2} + \sum_{i=2}^{20} \frac{1}{i^2} \leq 1 + \int_1^{20} \frac{1}{x^2} dx$$

Theorem 3.6.2. If a_n (and its corresponding $f(x)$) are increasing then what would the inequalities look like?

4 Thoughts, Problems, Ideas

1. Prove using weak induction that:

$$\sum_{i=0}^n 2^i = 2^{n+1} - 1$$

2. Verify and find the corresponding constants using structural induction for:

$$\sum_{i=0}^n 3^i = \alpha 3^n + \beta$$

3. Define a set of graphs S by: An single node is in S and if $G \in S$ then the result of adding an edge and a new node to a node already in S is in S . Prove that $V = E + 1$.
4. How many possible comparisons of the form $x < y$ are there if $x, y \in \{a, b, c, d\}$?
5. How many possible comparisons of the form $x - y < z$ are there if $x, y, z \in \{a, b, c, d, e, f, g\}$?
6. How many possible comparisons of the form $x + y < z$ are there if $x, y, z \in \{a, b, c, d, e, f, g\}$ if we assume $x + y$ and $y + x$ are equivalent?
7. Suppose an algorithm adds any set of numbers given to it. If the input to the algorithm could be one of three sets, either $\{1, 2, 3\}$, $\{4, 5, 1\}$, and $\{0, 2, 3, 10\}$, with probabilities 0.5, 0.3 and 0.2 respectively, what is the expected value of the output of the algorithm?
8. Suppose algorithm A can accept any nonnegative integer and finds the square root of its input. If the probabilities of the input being $0, 1, 2, 3, \dots$ are $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \dots$ respectively what can you say about the expected value of the output of the algorithm?
9. Calculate the sum:

$$\sum_{i=2}^{2n} (3i - 1)^2$$

10. Calculate the sum:

$$\sum_{i=3}^{100} (0.2)^{5i}$$

11. Suppose b is an unknown base and you know $\log_b 3 = \alpha$ and $\log_b 4 = \beta$. Calculate each of:

$$\log_b 36, \lg 3, \log_8 \frac{81}{4}$$

12. Calculate the derivative:

$$\frac{d}{dx} x^2 + 3x e^{2x}$$

13. Calculate the integral:

$$\int_0^1 xe^x dx$$

14. Calculate the integral:

$$\int_2^4 \ln x dx$$

Hint: Use IBP with $u = \ln x$ and $dv = 1 dx$.

15. Calculate the integral:

$$\int_2^4 x \ln x dx$$

16. Find integral-related bounds for:

$$\sum_{i=1}^{20} \frac{1}{i}$$

17. Find integral-related bounds for:

$$\sum_{i=1}^{20} \frac{1}{i+3}$$

18. Find integral-related bounds for:

$$\sum_{i=1}^{100} \frac{1}{i^3}$$

19. Find integral-related bounds for:

$$\sum_{i=1}^{10} \frac{1}{\sqrt{i}}$$

20. Calculate and write down the inequality pair that corresponds to the theorem for increasing functions. Show work, including pictures as necessary.

21. Find integral-related bounds for:

$$\sum_{i=1}^5 i + \sqrt{i}$$

22. Find integral-related bounds for:

$$\sum_{i=1}^5 \ln(i)$$

CMSC 351: Algorithm Time Complexity

Justin Wyss-Gallifent

July 22, 2023

1	Introduction	2
2	Problems, Algorithms, Time Complexity	2
3	Complications	3
3.1	Storage Assumptions	3
3.2	Unclear Algorithm Descriptions	5
3.3	What Goes Inside?	5

1 Introduction

It's very common to hear statements such as "Bubble sort is $\mathcal{O}(n^2)$ " but what does this mean exactly?

2 Problems, Algorithms, Time Complexity

Typically in computer science we have a problem we wish to solve (or a calculation we wish to perform) and we create an algorithm to do so.

Problem \rightarrow Algorithm \rightarrow (Pseudo)Code

That algorithm then has an implementation in (pseudo)code and that implementation has a time complexity.

Definition 2.0.1. Technically speaking an *algorithm* is a finite sequence of instructions, rigorously given, which solves a problem or performs a computation.

Most of the time the instructions in the algorithm are clear enough that it is obvious what the corresponding (pseudo)code would look like and so the time complexity becomes clear.

Example 2.1. Consider the problem of determining if a list is sorted. This description of the problem is not an algorithm. An algorithm might be something like:

Algorithm 1: Check if every element is no larger than the next one and return True if this is the case and False otherwise.

We could implement Algorithm 1 as follows:

```
function isSorted1(A):
    n = length(A)
    for i = 0 to n-2:
        if A[i] > A[i+1]:
            return False
        end if
    end for
    return True
end function
```

We can easily see that this pseudocode has time complexity $\Theta(n)$ and so we say that Algorithm 1 has time complexity $\Theta(n)$ where n is the length of the list.

Of course this is not the only algorithm which determines if a list is sorted.

Algorithm 2: Check if every element is no larger than all the following elements and return True if this is the case and False otherwise.

We could implement Algorithm 2 as follows:

```
function isSorted2(A):
    n = length(A)
    for i = 0 to n-2:
        for j = i+1 to n-1:
            if A[i] > A[j]:
                return False
            end if
        end for
    end for
    return True
end function
```

We can easily see that this pseudocode has time complexity $\Theta(n^2)$ and so we say that Algorithm 2 has time complexity $\Theta(n^2)$ where n is the length of the list.

What is critical then is that when we make a statement such as:

Determining if a list is sorted is $\Theta(n)$.

What we really mean is:

The fastest possible known algorithm for determining if a list is sorted has an implementation which has time complexity $\Theta(n)$.

In the above example it would be Algorithm 1.

Of course you might protest - perhaps there is a faster algorithm. And perhaps you're right, and if you found such an algorithm you could then disagree with the above statement and say something like "No, determining if a list is sorted is actually $\Theta(\lg n)!$ " and you could present your algorithm for doing this.

Note 2.0.1. In reality this is partly the reason why we might avoid using Θ and stick with \mathcal{O} . If we have shown a problem is $\mathcal{O}(n)$ using a $\mathcal{O}(n)$ algorithm then even if an updated algorithm is $\mathcal{O}(\lg n)$, meaning the problem is $\mathcal{O}(\lg n)$, it is still $\mathcal{O}(n)$ as well, whereas this is not the case if we've used Θ .

If this is a bit confusing don't forget that if $f(n) = \mathcal{O}(\lg n)$ then $f(n) = \mathcal{O}(n)$ as well but if $f(n) = \mathcal{O}(n)$ then we cannot tell if $f(n) = \mathcal{O}(\lg n)$.

3 Complications

There are several points to consider, however:

3.1 Storage Assumptions

In our list example above we are assuming that our list is stored in some "nice" way such that we can access list elements in constant time. There are of course

other ways to store a list of numbers such as in a linked list, in a stack, in a queue, and so on, and if we were to alter our assumption of how the list is stored this may or may not change the result.

Example 3.1. Suppose a stack A represents a list of numbers and we wish to determine if the list is sorted. Assume when we pop from an empty stack we get NULL.

An algorithm might be something like:

Algorithm: Pop elements off the stack and check if each element is greater than or equal to the next one popped. Return True if this is the case and False otherwise.

We could implement this as follows:

```
function issorted(A):
    x = A.pop
    if x == NULL:
        return True
    y = A.pop
    if y != NULL:
        if x < y:
            return False
        x = y
        y = A.pop
    end if
    return True
end function
```

We can easily see that if n is the number of elements on the stack then this pseudocode has time complexity $\Theta(n)$ and so we say that our algorithm has time complexity $\Theta(n)$.

While the storage may vary one critical item to note is that the way we are storing the data must be agnostic to the problem. What this means, essentially, is that we can't choose to store the data in a way that makes the problem obvious.

Example 3.2. Suppose we said that we will store a list in the entries $A[1], A[2], \dots$ and require that $A[0]$ contains 0 if the list is unsorted and 1 if the list is sorted. Then our algorithm will be:

Algorithm: Check if $A[0]$ is 0 or 1.

With pseudocode:

```
function issorted(A)
    if A[0] == 1:
```

```
    return True
end if
return False
end function
```

Now we claim that determining if a list is sorted is $\Theta(1)$.

Clearly we're being dishonest here in the sense that we're storing our list in a way which is not agnostic to the problem we're trying to solve, and this is a no-no.

3.2 Unclear Algorithm Descriptions

It's not uncommon for the description of an algorithm to be unclear in some specifics.

Example 3.3. Kruskal's algorithm (for graph) involves detecting if adding an edge to a graph forms a cycle. In the description and in the pseudocode we will often simply see some fragment like the following:

```
if adding this edge forms a cycle:
blah blah
```

However it is not entirely clear how we would know if adding an edge would form a cycle or not. In such a case we should be more precise when discussing time complexity.

3.3 What Goes Inside?

When we are working with an object such as a list it's fairly clear that when we say that some algorithm is $\Theta(n)$ the n is referring to the number of elements in the list. However if the data we are working with is something different then we have to be more specific.

Some examples that we will see as we progress will be things like:

- When working with $n \times n$ arrays then it makes sense to use n inside.
- When working with $n \times m$ arrays then it makes sense to use both n and m inside.
- When working with graphs with V vertices and E edges then it makes sense to use both V and E inside.

CMSC 351: Algorithm Design

Justin Wyss-Gallifent

February 23, 2021

1	Introduction	2
2	General Steps	2
3	Algorithm Design	4
4	Thoughts, Problems, Ideas	6

1 Introduction

This is not, per se, a course in algorithm design. In theory that is meant to be part of CMSC451. However it's worth introducing some elements of algorithm design which we can keep in mind and revisit as we discuss the algorithms in this course.

2 General Steps

Algorithm design can be broken down a variety of ways but here is one possible way that we can think about how we might design an algorithm to solve a certain problem.

1. Problem statement: Come up with a rigorous statement of what the problem is.

Typically the first step is fairly straightforward but often edge cases can complicate things. If the problem involves a set of numbers we might ask how we should respond if the set is empty. If the problem involves returning the index of an entry in a database we might ask how we should respond if there are multiple results.

2. Model development: How can we represent the problem using a model which lends itself to analysis?

The second step often takes some mathematical knowledge. Is our data best represented in a list? An array? A linked list? A heap? How? Note that this is less a computer science question than a mathematics one.

3. Algorithm specification: What should the algorithm do, explicitly and rigorously? It's at this stage where we nail down how the real-world problem manifests in the algorithm.

This step is typically a mash-up of the first two since it involves rephrasing the first step in terms of the model we've chosen.

4. Algorithm design: Now we figure out how to actually design the algorithm.

Typically this culminates in the pseudocode and to some degree is merged with the next step in the sense that checking correctness can be considered a part of design. Together these two steps are often the most challenging.

5. Checking correctness: Once we've designed an algorithm how can we check if it's correct?

If it's not then we need to go back and fix our algorithm.

6. Algorithm analysis: Typical analysis include running time in big- \mathcal{O} notation as well as memory usage.

It's at this point where we might decide that our algorithm is not sufficient. Perhaps the time complexity is not good enough or the memory

usage is too high. This could mean we need to go back to the algorithm specification state or even re-think the development of the model.

7. Algorithm implementation: Now we get to actually implement the algorithm in code.

This is to some degree the “computer programming” stage rather than the “computer science” stage.

8. Program testing: Once we have code then we can do real-world tests not just to see the algorithm in action but to see if we’ve overlooked anything.

Example 2.1. Suppose we wish to write an algorithm to find the maximum of a set of numbers. Here’s how we might go through the stages.

1. Problem statement: We have a set of numbers and we wish to find the maximum. These numbers could be positive or negative and could be integers. For now we’ll avoid real numbers and we’ll assume the set is nonempty.
2. Model development: We’ll store the numbers in an array because this lends itself well to computation. Thus our model is an array of numbers.
3. Algorithm specification: The algorithm needs to somehow examine all the numbers and figure out the largest.
4. Algorithm design: One idea - we’ll go through the numbers one by one, keeping track of and updating a variable which will contain the largest so far. To jump-start the process we’ll assign the largest so far to be the first one. Here’s the pseudocode:

```
\\" PRE: The array A contains all the numbers.  
max = A[0]  
for i = 1 to len(A)-1  
    if A[i] > max  
        max = A[i]  
    end  
end  
\\" POST: The variable max contains the maximum.
```

5. Checking correctness: We could introduce a loop invariant such as:

$LI(i)$: When i iterations have completed, max contains the maximum of $A[0, \dots, i]$.

We would then check this loop invariant.

6. Algorithm analysis: If there are n numbers then the length of the array is n and the loop iterates $n - 1$ times. Before the loop we have an assignment which takes c_1 time and inside the loop we have a check and possible assignment which takes c_2 time. Thus we have total time requirement $c_1 + c_2(n - 1) = \mathcal{O}(n)$.
7. Algorithm implementation: In Python something like:

```
max = A[0]
for i in range(1, len(A)):
    if A[i] > max:
        max = A[i]
```

8. Program testing: We could, for example, stress-test by giving the algorithm thousands of sets of numbers to make sure it never has any crashes and to spot-check some of the output. If we have other reliable ways of finding the maximum we could compare our output against that output.

3 Algorithm Design

The algorithm design stage is very often the most challenging. Some classic notions that arise are as follows. This is not at all comprehensive and these are not mutually exclusive. To help digest the ideas we'll work through a real-world example.

1. Linear: Can we simply step through the data item by item?

Example 3.1. You have a bunch of unlabeled boxes in your attic. You're looking for a bottle-opener so you have to check every box in turn.

2. Divide and Conquer: This is recursive and related to decrease and conquer. Here we divide into smaller problems and tackle each of those and combine the results. Examples: Merge sort, quick sort.

Example 3.2. You have two friends who are willing to help. Each of you takes one third of the boxes and looks through them one by one.

3. Decrease and Conquer: This is recursive and related to divide and conquer. Here we reduce the problem to a single smaller version of itself. Examples: Finding factorials, binary search.

Example 3.3. You were smart enough to use red boxes for kitchen items so you can focus just on those.

4. Greedy Algorithms: Greedy algorithms make the best choice at each stage but may not make the best overall choice. Typically this approach to a problem is useful when best local solutions are good enough or when they do, in fact, lead to best global solutions.

Example 3.4. Open a box and find something that'll open a bottle. It might not be a bottle-opener but that's fine.

5. Brute Force: This almost always works but the time required can be prohibitive.

Example 3.5. Look at every item carefully. Is it a bottle-opener?

6. Dynamic Programming: Remembering the past and applying it to the future.

Example 3.6. Maybe at the start of all this you don't know what a bottle-opener is. After you try opening a bottle with whatever you find you start developing a better idea what a bottle-opener looks like and it makes subsequent attempts easier.

7. Randomized Algorithm: There may be values which need to be assigned as part of the algorithm. It's not uncommon to make random choices at this stage, if only because non-random choices are often biased in some way.

Example 3.7. Just grab whatever, at random, out of any box. Why not? Could be a bottle-opener.

4 Thoughts, Problems, Ideas

1. Each of the following algorithm descriptions is lacking at least one aspect of rigor or clarification. Identify.
 - (a) An algorithm which finds the index of the maximum number in a list of numbers.
 - (b) An algorithm which sorts a list of numbers.
 - (c) An algorithm which rotates a square array by 90° .
 - (d) An algorithm that takes three positive integers a, b, n and finds the first n digits of the decimal expansion of a/b , returning them in a list.
2. Which of the following models could work for the problem given. Model choices include:
 - An n -dimensional array (specify n)
 - An undirected unweighted graph
 - A directed weighted graph
 - An undirected weighted graph
 - A directed weighted graph
 - some combination thereof if it seems reasonable

Justify.

- (a) An algorithm which finds the shortest route in a highway network.
- (b) An algorithm which finds the longest period of daily increase in TSLA stock.
- (c) An algorithm which finds patterns in temperature data throughout a city.
- (d) An algorithm which finds weaknesses in a small computer network.
3. Suppose the only storage mechanism you have is three stacks S , T , and U , so no arrays, lists, etc. For a stack S you can $S.push(x)$ and $x=S.pop$, and $S.pop$ will return `NULL` if a stack is empty. Suppose you have a list of distinct numbers contained in a stack S and you wish to sort them into U . Explain how you could do this.
4. Repeat the previous question but now you have only two stacks S and T and you want the numbers sorted back into S .

5. This algorithm should find the spread (max-min) of a list but something is wrong. What?

```
\\" PRE: A[0,...,n-1] is a list.
spread = -inf
for i = 0 to n-2
    for j = i+1 to n-1
        if A[i] - A[j] > spread
            spread = A[i] - A[j]
    end
end
\" POST: spread found.
```

6. Suppose an algorithm needs to take a value n and return the list:

$$\{0!, 1!, 2!, 3!, \dots, (n-1)!\}$$

However you don't have access to a factorial function so you need to calculate those manually. How is this problem related to dynamic programming?

- 7. Suppose $f(n)$ is defined for all $0 \leq n \leq 1000000$. Suppose an algorithm needs to find a local maximum of $f(n)$. In other words it needs to find an n_0 such that $f(n_0 - 1) < f(n_0)$ and $f(n_0 + 1) < f(n_0)$. How could a greedy approach aid in this?
- 8. Suppose in the previous question "local maximum" is replaced with "maximum". How could some degree of randomization be introduced in order to get a reasonable (although perhaps not perfect) result?
- 9. In what way could your local post office be considered as doing a divide-and-conquer approach?

10. Often conditionals involving too many conjunctions and/or negations can be confusing. In such cases some rewriting can simplify the situation. Rewrite each of the following as a simple conditional with an expression of the form `if X COMPARISON Y` where `comparison` is one of \geq , \leq , $>$, $<$, $==$ or $!=$ and where `X` and `Y` are basic arithmetic expressions. The first one has been done for you.

- (a) if A and B have opposite signs
Solution: if $A*B < 0$
- (b) if A and B have the same sign
- (c) if at least one of A,B is zero
- (d) if neither A nor B is zero
- (e) if both A and B are zero
- (f) if at least one of A and B is nonzero
- (g) if $(A+B>0 \text{ and } C-D>0)$ or $(A+B<0 \text{ and } C-D<0)$
- (h) if $(A+B>0 \text{ and } C-D<0)$ or $(A+B<0 \text{ and } C-D>0)$

11. Often access to data is restricted by architecture. For example when you pull data from a database via an API you can only retrieve it in chunks and for purposes of storage and bandwidth the size of those chunks might be restricted. As such you may need to plan your algorithm around this.
- For each of the following assume you have a list A whose length n is a multiple of 5. You cannot access the entries in A directly. Instead you have a separate working memory $R[0, \dots, 4]$ which you can access directly. You also have the commands $\text{in}(i)$ which copies $A[i, \dots, i+4]$ to $R[0, \dots, 4]$ and $\text{out}(i)$ which copies $R[0, \dots, 4]$ to $A[i, \dots, i+4]$. This is called a *block transfer*. In general you want to minimize the number of block transfers you make and you must be careful since neither command called on an index i may spill over the end of A .

Using this restriction as well as $\Theta(1)$ auxiliary space, write pseudocode which does each of the following. The first is done for you.

- (a) Prints each element in the list.

Solution: One possible solution:

```
blockcount = n/5
for i = 0 to blockcount-1
    in(5*i)
    for j = 0 to 4
        print(R[j])
    end
end
```

- (b) Finds the minimum of the list.

- (c) Counts the number of occurrences of a specified value.

- (d) Passes through the list once from left to right and increases each value by 1.

- (e) Replaces all occurrences of a specified value by another specified value.

- (f) Passes through the list once from left to right and interchanges any adjacent entries which are out of order.

- (g) Reverses the list. Do this first with the assumption that n is a multiple of 10.

12. Repeat question 11 under the slightly different constraints:

- The functions in and out take a second value k between 1 and 5 inclusive so that $\text{in}(i, k)$ copies $A[i, \dots, i+k-1]$ into $R[0, \dots, k-1]$ and $\text{out}(i, k)$ does the reverse.
- The list length is not necessarily a multiple of 5.

13. Repeat question 11 under the slightly different constraint:

- The list length is not necessarily a multiple of 5 but is at least 5.

CMSC 351: Algorithm Analysis

Justin Wyss-Gallifent

January 17, 2022

1	Introduction	2
2	Counting Particular Operations	3
3	Time Measurement and Complexity of an Algorithm	6
3.1	Introduction	6
3.2	Examples	8
3.3	Focusing on Big-O and on Worst-Case	10
3.4	Smaller Time Complexity is Not Always Better:	10
3.5	Common O and General Facts	11
4	Space Requirements	11
4.1	Space Complexity	11
4.2	Auxiliary Space	11
4.3	Quirky Ideas	13
5	Thoughts, Problems, Ideas	15

1 Introduction

In this course we can put our goals broadly into two categories:

- Primary Goal: To measure things which are reliant upon the code structure and not upon things like hardware specifics, data management, and so on.
- Secondary Goal: To measure things which are related to things like hardware specifics, data management, and so on.

We will focus mostly on our primary goals and we will look at our secondary goals only under ideal and very specific conditions.

2 Counting Particular Operations

Some particular operations we might count would be the following. This is by no means comprehensive and in fact these are just simple examples to get us thinking.

- Total number of assignments in an algorithm.
- Total number of comparisons in an algorithm.
- Total number of accesses to certain types of memory by an algorithm.

In a more complicated real-world scenario we might be counting database inserts, database record-locking queries, front-end calls to a back-end REST API, updates to the display, and so on.

Personal note: I can't count (pun intended) the number of times I've had a client complain about slow software which can be traced back to a particular operation being repeated far more frequently than the software was designed to sensibly manage or which invoked calls for data on a scale not planned for.

When we use pseudocode we'll need to be very clear about making it code-like enough to accurately measure these things.

We'll usually focus on three measurements and we'll look at each of them in terms of how they change as the input size n changes.

1. **Worst-Case Analysis:** Worst-case analysis might be thought of better as "maximum-case analysis". Here the idea is to see how large a given quantity might be. For example if we're counting assignments in a block of code a worst-case analysis would be asking for the maximum number of assignments that might occur.
2. **Best-Case Analysis:** Worst-case analysis might be thought of better as "minimum-case analysis". Here the idea is to see how small a given quantity might be. For example if we're counting assignments in a block of code a best-case analysis would be asking for the minimum number of assignments that might occur.
3. **Average-Case Analysis:** This is a tricky issue because trying to figure out what an "average-case" looks like involves having an understanding of what all inputs look like and what the probability of each is. This can be situation-dependent. We'll see this more in manageable examples.

In all cases it would be ideal to calculate $\Theta(n)$ to really constrain each case but if that is not possible then we will aim for $\mathcal{O}(n)$ since this will provide an upper bound on the measurement.

Example 2.1. For example consider this simple block of code which sorts a pair of distinct values:

```
\\" PRE: A is an array of two distinct numbers A[0] and A[1].
if A[0] > A[1]
    temp = A[0]
    A[0] = A[1]
    A[1] = temp
end
\" POST: A is sorted.
```

Let's look at some counts:

- If we're analyzing assignments we can view the code as:

```
if A[0] > A[1]
    temp = A[0]  <=
    A[0] = A[1]  <=
    A[1] = temp  <=
end
```

In a worst-case scenario $A[0] > A[1]$ is true and three assignments are made.

In a best-case scenario $A[0] < A[1]$ is false and no assignments are made.

In an average-case scenario we must have an idea what the input looks like. Two distinct numbers, great, but are they random? Maybe the input is more likely to have a larger first number, or maybe a larger second number? A simple average-case analysis can be thought of as suggesting that on average half the time the first number is smaller and half the time the first number is larger. Thus half the time we have three assignments and half the time we have none. This gives us an average count of:

$$0.5(3) + 0.5(0) = 1.5$$

- If we're analyzing comparisons we can view the code as:

```
if A[0] > A[1]  <=
    temp = A[0]
    A[0] = A[1]
    A[1] = temp
end
```

Here there is always one comparison so that's the worst-case, the best-case and the average-case.

Example 2.2. For example consider this simple block of code which does a single pass through a list of length n and swaps adjacent values if they are out of order:

```
\\" PRE: A is an list of length n indexed as A[0,...,n-1].
for i = 0 to n-2
    if A[i] > A[i+1]
        swap = A[i]
        A[i] = A[i+1]
        A[i+1] = swap
    end
end
\" POST: A is sorted.
```

Let's look at some counts:

- If we're analyzing assignments we can view the code as:

```
for i = 0 to n-2
    if A[i] > A[i+1]
        swap = A[i]      ==
        A[i] = A[i+1]    ==
        A[i+1] = swap   ==
    end
end
```

In a worst-case scenario the conditional will pass $n-1$ times and $3(n-1) = \Theta(n)$ assignments are made.

In a best-case scenario the conditional will fail every time and 0 assignments are made.

In an average-case scenario we must have an idea what the input looks like. Perhaps half the time the conditional will pass. This is $(n-1)/2$ times out of $n-1$. Then $3(n-1)/2 = \Theta(n)$ assignments are made.

- If we're analyzing comparisons we can view the code as:

```
for i = 0 to n-2
    if A[i] > A[i+1]      ==
        swap = A[i]
        A[i] = A[i+1]
        A[i+1] = swap
    end
end
```

Here there are always $n-1 = \Theta(n)$ comparisons so that result applies for all of the worst-, best- and average-cases.

3 Time Measurement and Complexity of an Algorithm

3.1 Introduction

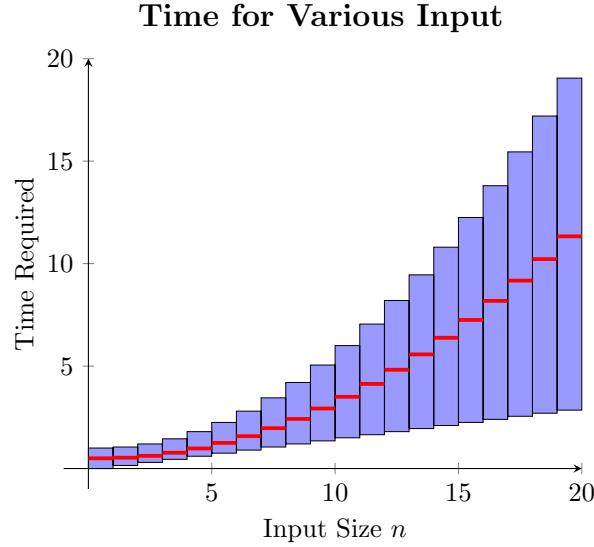
Typically we are less interested in getting an explicit formula for the time complexity (which is good, since the calculations can be messy) but rather in how fast the time complexity grows as n does. In other words we are interested in statements involving \mathcal{O} , Θ and Ω .

One key to understanding what's going on is that for each input size n there are many possible inputs, and each input has a time requirement associated to it. For each n there will be some input(s) which take the minimum time, some input(s) which take the maximum time, and then an average time taken over all inputs with their respective probabilities.

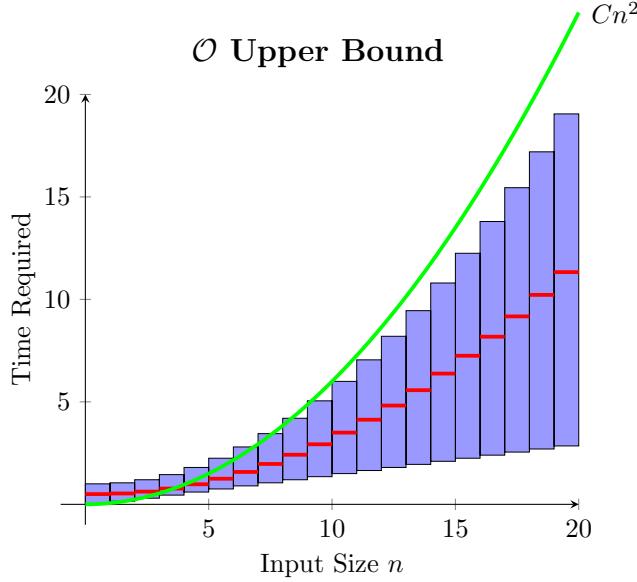
We can think of it something like this made-up example:

Example 3.1. In this picture each blue rectangle indicates the range of times required for each particular n . The top of each rectangle is each n 's worst-case, the bottom of each rectangle is each n 's best-case, the red line in each rectangle is each n 's average case.

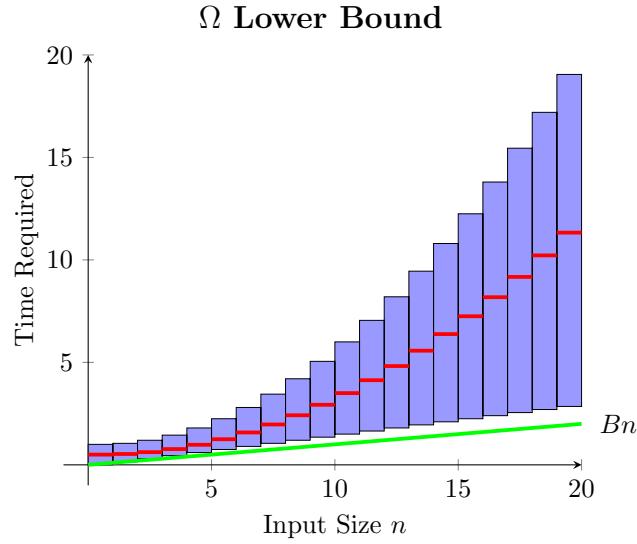
Observe that the average is not necessarily the middle because the distribution of time requirements within each n is not necessarily uniform:



Finding $T(n) = \mathcal{O}(g(n))$ for the worst-case $T(n)$ means finding an eventual bound above the worst-case and hence above everything, such as the following. For example if the green line is $g(n) = Cn^2$ for some $C > 0$ then we can say that the worst-case, best-case and average case are all $\mathcal{O}(n^2)$, meaning it would never get worse than that as $n \rightarrow \infty$:



Finding other bounds can be helpful, too. Finding $T(n) = \Omega(g(n))$ for the best-case $T(n)$ means finding an eventual bound below the best-case and hence below everything, such as the following. For example if the green line is $g(n) = Bn$ for some $B > 0$ then we can say that the worst-case, best-case and average case are all $\Omega(n)$, meaning that it would never get better than that as $n \rightarrow \infty$.



Other bounds might be less helpful. For example if we find $T(n) = \Omega(g(n))$ for the worst-case $T(n)$ then all we have is an eventual bound below the worst-case.

This only tells us that the worst-case would never get better than this but this is not particularly helpful for a worst-case situation, as it could get a lot worse!

3.2 Examples

Here are a couple of thorough examples.

Example 3.2. Consider the pseudocode code here with time requirements given. It finds the maximum in a list:

<pre>max = A[0] for i = 1 to n-1 if A[i] > max max = A[i] end end</pre>	c_1 c_2 iterates $n - 1$ times c_3 iterates $n - 1$ times c_1 iterates when the conditional passes
--	---

In a worst-case scenario the conditional passes for each iteration of the loop and the total time requirement is:

$$T(n) = c_1 + (n - 1)(c_2 + c_3 + c_1) = \Theta(n)$$

In a best-case scenario the conditional fails for each iteration of the loop and the total time requirement is:

$$T(n) = c_1 + (n - 1)(c_2 + c_3) = \Theta(n)$$

In an average-case scenario we must have an idea what the input looks like. Perhaps half the time the conditional will pass and half the time it will fail. Then the total time requirement is:

$$T(n) = c_1 + \frac{n - 1}{2}(c_2 + c_3 + c_1) + \frac{n - 1}{2}(c_2 + c_3) = \Theta(n)$$

Note that the time complexity is identical every time but the exact time is different for each.

Example 3.3. Consider the pseudocode code here with time requirements given. It goes through a list once and swaps adjacent elements until it finds two that are not adjacent and then it stops.

```

for i = 0 to n-2       $c_1$  iterates  $n - 1$  times
    if A[i] > A[i+1]    $c_2$  iterates  $n - 1$  times
        swap = A[i]        $c_3$  iterates when the conditional passes
        A[i] = A[i+1]      $c_3$  iterates when the conditional passes
        A[i+1] = swap      $c_3$  iterates when the conditional passes
    else
        break              $c_4$  iterates when the conditional fails
    end
end
```

In a worst-case scenario every iteration has the condition true which then results in a swap and the break never happens. The total time requirement is:

$$T(n) = (n - 1)(c_2 + 3c_3) = \Theta(n)$$

In a best-case scenario the first iteration has the condition false and the break occurs immediately. The total time requirement is:

$$T(n) = c_1 + C_2 + c_4 = \Theta(1)$$

In an average-case scenario we must have an idea what the input looks like. Perhaps on average the conditional fails halfway through. Then the total time requirement is:

$$T(n) = \left(\frac{n-1}{2}\right)(c_1 + c_2 + 3c_3) + (c_1 + c_2 + c_4) = \Theta(n)$$

Note that the time complexity is not identical every time.

3.3 Focusing on Big-O and on Worst-Case

While finding Θ is ideal, if we can't do this then we'll look for \mathcal{O} as this gives an "it can't be worse than this" result.

While finding all of worst-, best-, and average-cases are ideal, if we can't do all three then we'll look at the worst-case.

3.4 Smaller Time Complexity is Not Always Better:

When it comes to time complexity, smaller functions are preferable, because it means that as the input size n increases the time requirement doesn't increase as much.

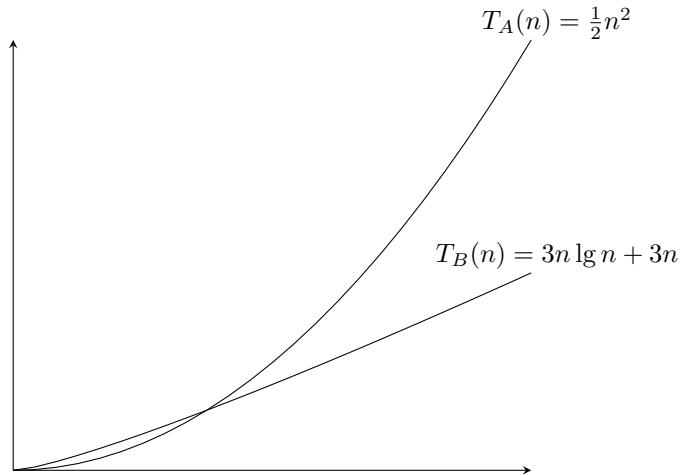
However for small values of n it's entirely possible that this isn't the case.

For this reason it's not necessarily good enough to simply know that an algorithm has a certain \mathcal{O} runtime.

Example 3.4. Suppose you have two algorithms A and B and all you know is that $T_A(n) = \mathcal{O}(n^2)$ and $T_B(n) = \mathcal{O}(n \lg n)$. You might automatically think that B is better.

However suppose algorithm A has explicit time complexity $T_A(n) = \frac{1}{2}n^2$ while algorithm B has time complexity $T_B(n) = 3n \lg n + 3n$.

Here is a plot of both, treating n as a real number:



It seems that for n -values before a certain threshold that $T_A(n) < T_B(n)$ and consequently algorithm A is preferable. If we actually test these values for specific values of n we find that $T_A(n) < T_B(n)$ for $n = 0, 1, \dots, 37$ and $T_A(n) > T_B(n)$ for $n = 38, 39, \dots$. It follows that if our input size is $n = 37$ or smaller then algorithm A is preferable.

When it comes to algorithm design we can imagine that A and B do the same thing but perhaps B is recursive, calling itself on smaller and smaller inputs. We could then suggest that when B is handed an input of size 37 or smaller it might actually invoke A in order to do a quicker job.

3.5 Common O and General Facts

The following arise most frequently:

- $\Theta(1)$: Constant time. The best.
- $\Theta(\lg n)$: Generally the best next largest. Binary search works at this speed. Most things don't.
- $\Theta(n)$: Not bad. Standard linear operations tend to be like this.
- $\Theta(n \lg n)$: Common for things involving trees, recursion, and so on. The best sorting algorithms are in this category.
- $\Theta(n^2), \Theta(n^3), \dots$: Not bad when the input size is small but can be useless for large input size. Sloppy algorithms on arrays tend to land here.
- $\Theta(2^n), \Theta(3^n), \dots$: Pretty useless except for very small input size.
- $\Theta(n!)$, $\Theta(n^n)$: Ditto.

4 Space Requirements

There are several ways in which we can look at the space that an algorithm requires.

4.1 Space Complexity

Definition 4.1.1. One is *space complexity* which measures how much memory is used for everything, including the data, extra memory requirements, and so on. We will not examine this.

4.2 Auxiliary Space

Definition 4.2.1. Second is *auxiliary space* which measures how much space is required in addition to of the input data. The input data itself is not taken into account.

Example 4.1. The following code swaps two variables `a` and `b`:

```
\\" PRE: a and b are variables.  
temp = a  
a = b  
b = temp  
\\" POST: The values are switched.
```

This code uses a single external value `temp`. If we're looking at auxiliary space then we would say it requires just one value and hence the auxiliary space is $\mathcal{O}(1)$.

Suppose we wanted to reverse a list with an even number of elements. Here is one approach:

```
\\" PRE: A is a list.
B = []
for i = len(A)-1 down to 0
    B.append(A[i])
end
for i = 0 to len(A)-1
    A[i] = B[i]
end
\" POST: A is reversed.
```

This code creates a new array B by reading the elements of A in reverse order and growing B accordingly. Due to the creation of this new list, if A has n elements then our algorithm creates a brand new list also requiring n elements. In addition we have the loop variable i and therefore uses $n+1 = \mathcal{O}(n)$ auxiliary space.

We can do better, though:

```
\\" PRE: A is a list.
for i = 0 to len(A)/2-1
    temp = A[i]
    A[i] = A[len(A)-1-i]
    A[len(A)-1-i] = temp
end
\" POST: A is reversed.
```

In this case we essentially swap elements - the first with the last, the second with the second-to-last, and so on. We require one additional auxiliary variable for the swapping, $temp$, and one for the loop i , and so this algorithm uses $1 + 1 = \mathcal{O}(1)$ auxiliary space.

Definition 4.2.2. This algorithm is said to reverse the list *in-place*.

It will often be the case that one algorithm may be slower but in-place while another may be faster and not in-place. In a case like this we may need to decide (using some external reasoning) which algorithm is “better” for our circumstances.

4.3 Quirky Ideas

Sometimes it can be educational (even if not useful) to attempt to write algorithms which are restricted to small amounts of auxiliary space or other quirks with working memory.

Example 4.2. Consider the following BubbleSort pseudocode:

```
\\" PRE: A is a list of length n.
stillgoing = True
while stillgoing
    stillgoing = False
    for i = 0 to n-2
        if A[i] > A[i+1]
            temp = A[i]
            A[i] = A[i+1]
            A[i+1] = temp
            stillgoing = True
        end
    end
end
\\" POST: A is sorted.
```

This requires three auxiliary variables - `stillgoing`, `i` and `temp`. Can we modify this code to remove the required `temp`? This variable arises to do the swap, so can we swap with no third variable? Interestingly yes.

Observe that if x and y are given and if we do the three steps in order:

$$\begin{aligned}x &= x + y \\y &= x - y \\x &= x - y\end{aligned}$$

The result will be that the variables switched. For example if $x = 10$ and $y = 42$ then:

$$\begin{aligned}x &= x + y = 10 + 42 = 52 \\y &= x - y = 52 - 10 = 42 \\x &= x - y = 52 - 42 = 10\end{aligned}$$

Thus the pseudocode can be modified according with the conditional becoming:

```
...
if A[i] > A[i+1]
    A[i] = A[i] + A[i+1]
    A[i+1] = A[i] - A[i+1]
    A[i] = A[i] - A[i+1]
    stillgoing = True
end
...
```

These sorts of exercises are useful because:

- They force us to consider what space requirements actually are.
- They force us to consider how wasteful we often are.
- If tight memory situations do arise we don't automatically give up.

5 Thoughts, Problems, Ideas

1. Consider the following pseudocode:

```
\\" PRE: A[0,...,n-1] is a list of integers.  
positiveproduct = 1  
for i = 0 to n-1  
    if A[i] > 0  
        positiveproduct = positiveproduct * A[i]  
    end  
end  
\\" POST: positiveproduct contains the product of the  
\\"         positive entries.
```

Assume for the average cases that the conditional is true half the time.

- In each of the worst-, best-, and average-cases how many conditional checks will take place?
- In each of the worst-, best-, and average-cases how many assignments will take place?
- If time values are assigned as follows:
 - Assignments take time c_1 .
 - Loop maintenance takes time c_2 .
 - Conditional checks take time c_3 .
 - Multiplication takes time c_4 .

In each of the worst-, best-, and average-cases how much time will be required? Calculate and give the time complexity of each.

2. Consider the following pseudocode:

```
\\"PRE: A[0,...,n-1] is a list of integers.  
sum = 0  
for i = 0 to n-1  
    if A[i] < 0  
        break  
    else  
        sum = sum + A[i]  
    end  
end  
\\" POST: sum is the sum up until the first negative entry.
```

Assume for the average cases that the conditional is true half the time.

- In each of the worst-, best-, and average-cases how many conditional checks will take place?

- (b) In each of the worst-, best-, and average-cases how many assignments will take place?
- (c) In each of the worst-, best-, and average-cases how many **break** statements will take place?
- (d) If time values are assigned as follows:
 - Assignments take time c_1 .
 - Loop maintenance takes time c_2 .
 - Conditional checks take time c_3 .
 - Addition takes time c_4 .
 - Break statements takes time c_5 .

In each of the worst-, best-, and average-cases how much time will be required? Calculate and give the time complexity of each.

3. Consider the following pseudocode:

```
\\"PRE: A[0,...,n-1] is a list of positive integers
\\      and M is a positive integer.
sum = 0
for i = 0 to n-1
    sum = sum + A[i]
    if sum > M
        break
    end
end
\\ POST: sum is the sum up until it is more than M.
```

Assume for the average cases that the average value in each array entry is $2M/n$.

- (a) In each of the worst-, best-, and average-cases how many conditional checks will take place?
- (b) In each of the worst-, best-, and average-cases how many assignments will take place?
- (c) If time values are assigned as follows:
 - Assignments take time c_1 .
 - Loop maintenance takes time c_2 .
 - Conditional checks take time c_3 .
 - Addition takes time c_4 .
 - Break statements takes time c_5 .

In each of the worst-, best-, and average-cases how much time will be required? Calculate and give the time complexity of each.

4. Suppose Algorithm A has running time $T_A(n) = 0.01n$ while Algorithm B has running time $T_B(n) = 12 \lg n$. For which n is algorithm A better and for which n is algorithm B better?
5. Suppose Algorithm A has running time $T_A(n) = 100 + 100n^{3/4} \lg n$ while Algorithm B has running time $T_B(n) = n^2$. Find the smallest n_0 such that for all $n \geq n_0$ we know that Algorithm B is faster.
6. Consider the following pseudocode:

```
\\" PRE: A is a list of length n
\" with each entry between 1 and n inclusive.
B = [1]
for i = 0 to n-2
    if A[i] > A[i+1]
        B.append[1]
    end
end
\" POST: What a mystery!
```

What are the best- and worst-case auxiliary space requirements?

7. Consider the following pseudocode:

```
\\" PRE: A is a list of length n
\" with each entry between 1 and n inclusive.
B = []
for i = 0 to n-1
    for j = 1 to A[i]
        B.append(A[i])
    end
end
\" POST: What a mystery!
```

What are the best- and worst-case auxiliary space requirements?

8. Suppose you have a list A of length n and a command `swap(i, j)` that swaps $A[i]$ and $A[j]$. Specifics of pseudocode aside what is the maximum number of swaps that could possibly be necessary to put the list in order?
9. Modify the array-reversing pseudocode so that it will work with lists containing an odd number of elements as well, while still only requiring $\mathcal{O}(1)$ auxiliary space.

10. Consider the following pseudocode:

```
\\" PRE: A is a list of length n.  
count = 0  
for i = 0 to n-1  
    for j = 0 to n-1  
        if i < j and A[i] > A[j]  
            count = count + 1  
        end  
        if i > j and A[i] < A[j]  
            count = count + 1  
        end  
    end  
end  
\\" POST: What a mystery!
```

Suppose you know that:

- Each of the four basic arithmetic operations takes time 3.
 - Each comparison takes time 2.
 - Each assignment (including each iteration assignment in a loop) takes time 1.
- (a) Find the value of `count` for the inputs `A=[1,2,3,4,5]`, `A=[5,4,3,2,1]` and `[1,4,3,5,2]`.
- (b) In general what value does `count` give?
- (c) Replace both `if` statements by a single `if` statement with a few simple arithmetic operations and a single comparison which achieves the same goal.
- (d) Which out of the original pseudocode or your modified pseudocode take would take the least time in a worst-case scenario?

CMSC 351: Integer Addition

Justin Wyss-Gallifent

April 4, 2021

1	Introduction	2
2	Schoolbook Addition	2
3	Pseudocode	2
4	Pseudocode Time Complexity	2
5	Improvements	3
6	Thoughts, Problems, Ideas	4
7	Python Test and Output	5

1 Introduction

Suppose we have two n -digit numbers and wish to add them. What is the worst-case time complexity of this operation?

2 Schoolbook Addition

The first and most obvious way to add two numbers is the way we learn in school. We add digit-by-digit and carry if necessary:

$$\begin{array}{r} & 1 & 1 & 1 \\ 3 & 6 & 7 & 2 & 8 \\ 6 & 5 & 9 & 1 & 6 \\ \hline 1 & 0 & 2 & 6 & 4 & 4 \end{array}$$

3 Pseudocode

If we store each number digit-by-digit in arrays A and B then the pseudocode for adding them and putting the result in c is as follows. To make things a little simpler we are storing the 1's digit in A[0], the 10's digit in A[1] and so on, so we print the lists backwards.

```
\\" PRE: A and B are lists of length n containing
\\"      the digits of two numbers.
\\ PRE: C is an empty list of length n+1.
C = list of 0s of length n+1
carry = 0
for i in range(0,n):
    C[i] = A[i] + B[i] + carry
    if C[i] > 9
        carry = the 10s digit of C[i]
        C[i] = the 1s digit of C[i]
    else
        carry = 0
    end
end
C[n] = carry
\\ POST: C contains the digit-by-digit result of adding A and B.
```

4 Pseudocode Time Complexity

What is the time complexity of this algorithm? Well it does constant-time operations before the loop, n constant-time operations for the loop, and constant-time operations after the loop, so worst-case, best-case, and average-case are all $\Theta(n)$.

5 Improvments

Could we do any better?

For numbers $a_n...a_1$ and $b_n...b_1$ we wish to find $c_{n+1}c_n...c_1$ (we go to c_{n+1} because there may be an additional digit). To find c_1 we absolutely have to calculate $a_1 + b_1$ since there is no other way to find that digit since we certainly can't figure it out from the remaining a_i and b_i .

Likewise to calculate c_2 we'll potentially need a carry digit from $a_1 + b_1$ but again we absolutely have to calculate $a_2 + b_2$. This pattern continues and in general we have no choice but to do the individual digit additions. Thus there are n required operations for a time complexity of $\Theta(n)$.

6 Thoughts, Problems, Ideas

1. Assume **A** and **B** are binary strings of length n and rewrite the pseduode, removing addition and comparison and instead using logical operators **and** (only once) and **xor** (only once).
2. The addition pseudocode can be rewritten to eliminate **carry** and instead store the carry pre-emptively in **C**. Do so.
3. Two's Complement: For a given binary number **B** the Two's Complement of the number is obtained by negating all the bits and adding 1. For example the two's complement of $B=01101$ is $\text{not}(B)+1=10010+1=10011$. For a number **B** with N bits if we add **B** and its two's complement we always get 2^N , for example $B+\text{not}(B)+1=01101+10011=100000$. Consequently for $A \geq B$ we have $A+\text{not}(B)+1=A+(2^N)-B=2^N+(A-B)$ and so we can calculate $A-B$ by instead calculating $A+\text{not}(B)+1$ and ignoring the resulting leftmost digit. For example:

$$\begin{aligned}1011101 - 0110111 &= 1011101 + \text{not}(0110111) + 1 \\&= 1011101 + 1001000 + 1 \\&= 10100110\end{aligned}$$

Write the pseudocode for this. Just for extra fun and excitement:

- Do not use a carry bit.
- Do not use any conditionals.
- Use only one loop.

You can just assume the additional resulting bit will be ignored.

7 Python Test and Output

Code:

```
import random
A = []
B = []
for i in range(0,7):
    A.append(random.randint(0,9))
    B.append(random.randint(0,9))
n = len(A)

print('' + str(A[::-1]))
print('' + str(B[::-1]))

C = [0] * (n+1)
carry = 0
for i in range(0,n):
    C[i] = A[i] + B[i] + carry
    if carry == 0:
        print(str(A[i])+'+'+str(B[i])+'='+str(C[i]))
    else:
        print(str(A[i])+'+'+str(B[i])+'+'+str(carry)+'='+str
              (C[i]))
    if C[i] > 9:
        carry = C[i] // 10
        C[i] = C[i] % 10
        print('Carry the '+str(carry))
    else:
        carry = 0
C[n] = carry

print(C[::-1])
```

Output:

```
[7, 2, 8, 9, 9, 6, 2]
[2, 6, 8, 3, 4, 3, 0]
2+0=2
6+3=9
9+4=13
Carry the 1
9+3+1=13
Carry the 1
8+8+1=17
Carry the 1
2+6+1=9
7+2=9
[0, 9, 9, 7, 3, 3, 9, 2]
```

CMSC 351: *k*th Order

Justin Wyss-Gallifent

November 3, 2022

1	Introduction:	2
2	Naïve Brute-Force Method	2
2.1	Approach	2
2.2	Time Complexity	2
3	Thoughts on Other Possibilities	4
3.1	Sorting First	4
3.2	Partially Sorting	4
3.3	Using a Min Heap	4
4	QuickSelect - Decrease and Conquer	5
4.1	Approach	5
4.2	Pseudocode	6
4.3	Time Complexity - The Beginning	7
4.4	Pivot Choice - Median of Medians	8
4.5	Return to Time Complexity	13
5	QuickSort-Related Note	15
6	Thoughts, Problems, Ideas:	16
7	Python Test and Output - Naïve	17
8	Python Test and Output - Median of Medians	18

1 Introduction:

Given an unsorted list of n distinct numbers suppose we wish to find the k th smallest of those. Here both n and k can vary. How might we go about this? This question arises when finding the minimum, the maximum, and, what is valuable to us, the median.

Definition 1.0.1. The k th order element in a list is the k th smallest entry. So the 1st order entry is the minimum, for example.

In what follows we shall assume that we have a list of distinct numbers.

2 Naïve Brute-Force Method

2.1 Approach

One brute-force method to finding the k th order entry might be to first find the smallest, then the next smallest, and so on, until we reach the k th smallest. This is not hard to implement, the pseudocode follows.

The general idea is that we find the minimum and assign it to `kth`. We then iterate by repeatedly: Go through `A` and find the smallest which is less than the maximum but larger than `kth`. This would be the next smallest. We do this $k-1$ times since `kth` was already the 1st order statistic.

```
def kthorder(A, k):
    n = len(A)
    mini = minimum element in A
    maxi = maximum element in A
    kth = mini
    for i = 1 to k-1
        nextsmallest = maxi
        for j = 0 to n-1:
            if (A[j] < nextsmallest) and (A[j] > kth)
                nextsmallest = A[j]
        end
    end
    kth = nextsmallest
    return(kth)
end
```

2.2 Time Complexity

Finding the maximum and minimum is $\Theta(n)$. The inner loop is $\Theta(n)$ and iterates $k - 1$ times, and so the entire algorithm is $\Theta(n + (k - 1)n) = \Theta(kn)$. This isn't bad and for a specific pre-determined k it's just $\Theta(n)$ but we'd like

something with a better time complexity for all possible n and k . For example when finding the median k depends upon n . Could we possibly do better?

3 Thoughts on Other Possibilities

There are a number of other approaches including:

3.1 Sorting First

We could simply sort the entire list and then select the $(k - 1)$ st entry (to adjust for the index). We can use something like MergeSort or HeapSort which has $\mathcal{O}(n \lg n)$ time complexity. This is nice in the sense that our time complexity does not depend on k .

3.2 Partially Sorting

We can do okay even with something like partial reverse BubbleSort. Each pass of reverse Bubble sort will fix one more starting element and so if we do k passes the time will essentially be:

$$n + (n - 1) + (n - 2) + \dots + (n - k - 1) = \Theta(kn)$$

Not really better than a brute force approach.

3.3 Using a Min Heap

Analogous to building a max heap we could build a min heap. When we built a max heap we said it was worse-case $\mathcal{O}(n \lg n)$ time complexity but this is not asymptotically tight and in fact it is worse-case $\mathcal{O}(n)$ (we have not shown this) and the same is true for a min heap. Since extracting an element and fixing the heap takes worst-case $\mathcal{O}(\lg n)$ time and we would do this k times the time complexity would be worse-case $\mathcal{O}(n + k \lg n)$.

These are all pretty good and interesting but can we do better overall? Perhaps something that's $\mathcal{O}(n)$ no matter what k is?

4 QuickSelect - Decrease and Conquer

4.1 Approach

Its not at all clear how decrease-and-conquer might help here but it's worth recalling the initial step of QuickSort. What we did was we chose a pivot value and then performed a partition on the list which resulted in the pivot value being re-located such that all smaller values were to the left and all larger values to the right. It follows that if the pivot value ends up at index p then the pivot value, now $A[p]$, is the $(p + 1)$ st order entry.

If we want the $(p + 1)$ st order entry then we simply return $A[p]$ and we're done.

If we want order less than $p + 1$ we know that the value we are seeking is smaller and hence to the left of index p so we repeat the process (choose pivot value and partition) on $A[0, \dots, p - 1]$ whereas if we want order greater than $p + 1$ we know that the value we are seeking is larger and hence to the right of index p so we repeat the process (choose pivot value and partition) on $A[p + 1, \dots, n - 1]$.

What this then leads to is exactly a decrease-and-conquer approach which reduces the size of the list each time and results in the desired value being constrained in a smaller and smaller list.

Since the element of desired order certainly exists in the list if the target rank is not found before the list reaches length 1 then it must be found at that point.

The partitioning process used in `quicksort` is $\Theta(n)$ which is great so the critical issue here is how much we can shorten our sublist at each stage. If we can shorten it quickly then the decrease-and-conquer approach may result in a time complexity better than $\Theta(kn)$.

Shortening the sublist means making good pivot value choices at each stage. It's not clear yet how we will do this but for now we can at least write down the pseudocode and fill in the pivot value choice details later.

4.2 Pseudocode

Here is the pseudocode. The `partition` function is stolen from the Quick-Sort pseudocode so we've omitted the details. Note that there is a critical line `pivotvalue = choosepivotindex(l,r)` which needs details. At least for right now, don't worry about how this is done, the critical thing is that after `partition` is called all the elements to the left of the pivot value are less than the pivot value and all the elements to the right of the pivot value are more than the pivot value.

```
def selectkth(A,L,R,k)
    pivotindex = choosepivotindex(L,R)
    A[index] <-> A[R]
    pivotindex = partition(L,R)
    if k-1 < pivotindex
        return(selectkth(L,pivotindex-1,k))
    else if k-1 > pivotindex
        return(selectkth(pivotindex+1,R,k))
    else
        return(A[pivotindex])
    end
end

def partition(A,L,R)
    same as quicksort
end

def choosepivotindex(A,L,R)
    somehow pick an index in A[L,...,R]
    return this index
end
```

4.3 Time Complexity - The Beginning

Suppose a call to `selectkth` takes time $T(n)$. This will then involve a call to `choosepivotindex`, a call to `partition`, and a further call to `selectkth`. Observe:

- Let's say the call to `choosepivotindex` takes time $CPI(n)$.
- The call to `partition` takes time $\Theta(n)$.
- The new call to `selectkth` will be on a sublist of length $f(n)$ for some function of n hence will take time $T(f(n))$.

Thus we have:

$$T(n) = CPI(n) + \Theta(n) + T(f(n))$$

If the pivot value selection is not engineered well in `choosepivotindex` then the pivot value could end up at one end or the other during each recursive call to `selectkth` and the list would decrease in length by 1 each time. In this case we would have $f(n) = n - 1$ and then:

$$T(n) = CPI(n) + \Theta(n) + T(n - 1)$$

Even if $CPI(n) = 0$ we still end up with quadratic time complexity (you can prove this!) so this is no better than brute force.

To fix this we need to engineer `choosepivotindex` well enough such that it reduces the sublist by something significant (not just 1) each time and such that it is fast enough that the first sum is small.

4.4 Pivot Choice - Median of Medians

So now we need to figure out how to write the `choosepivotindex` function so that it shrinks the list length by a good amount. The method we'll use is the Median of Medians.

The Median of Medians is a fast recursive method for finding a value close to the median. There are a few ways this can be implemented but we'll take the standard approach of creating a method which results in mutual recursion.

Once we've established the median of medians function we will integrate it into the pseudocode loosely as follows:

```
def selectkth(A,L,R,k)
    mom, index = mom(A[L:R])
    A[momindex] <-> A[R]
    pivotindex = partition(L,R)
    if k-1 < pivotindex
        return(selectkth(L,pivotindex-1,k))
    else if k-1 > pivotindex
        return(selectkth(pivotindex+1,R,k))
    else
        return(A[pivotindex])
    end
end

def partition(A,L,R)
    same as quicksort
end

def mom(A)
    find mom
    return mom, index
end
```

Now on to how the Median of Medians method actually will work.

Broadly speaking the method works recursively as follows for a list containing n elements:

1. Divide the list into $\lfloor n/5 \rfloor$ sublists of length 5 and perhaps one group with the remaining elements and sort those sublists.
2. Select the median of each sublist. For the final sublist if it has two or four elements select the lower median.
3. Apply `selectkth` recursively on the smaller list of those values in order to find the median (or lower median) of that new list.

Note 4.4.1. You might wonder why we're using sublists of length 5 and we will discuss that later. Just to avoid confusion, though, we'll use MOM-5 to denote the Median of Medians using sublists of length 5.

Note that the call to `selectkth` inside this method itself will make calls to `partition` and on to `choosepivotindex`, hence the mutual recursion. These successive calls will be on smaller lists, of course.

Example 4.1. Let's demonstrate MOM-5 on the following list:

10, 8, 5, 12, 11, 15, 21, 99, 7, 6, 70, 17, 3, 35, 71, 1, 2, 30, 36, 31, 32, 33, 60, 29, 28, 34, 40, 41, 80

First we divide this list into columns and sort each column:

5	6	3	1	28	34
8	7	17	2	29	40
10	15	35	30	32	41
11	21	70	31	33	80
12	99	71	36	60	

The list formed by the medians (and lower median) of the columns is 10, 15, 35, 30, 32, 40 and calling `selectkth` specifically to get the (lower) median will return 30.

Of course this call to `selectkth` itself involves partitions and deeper calls to the median of median function.

■

Now that we see how MOM-5 will work, here is the pseudocode again with the mutual recursion visible:

```
def selectkth(A,L,R,k)
    mom, index = mom(A[L:R])
    A[momindex] <-> A[R]
    pivotindex = partition(L,R)
    if k-1 < pivotindex
        return(selectkth(L,pivotindex-1,k))
    else if k-1 > pivotindex
        return(selectkth(pivotindex+1,R,k))
    else
        return(A[pivotindex])
    end
end

def partition(A,L,R)
    same as quicksort
end

def mom(A,L,R)
    divide A into sublists and sort those
    mlist = new list of (lower?) medians of sublists
    s = apply selectkth to get the (lower?) median of mlist
    return index
end
```

General Argument, Nice Case:

We claim that the value obtained by MOM-5 does a good job of subdividing the list via **partition**. To see this let's figure out exactly what property it has.

To keep the calculation simple assume we have a list of n elements where n is an odd multiple of 5 in order to make the argument more straightforward. If this is not the case we need to tweak the approach slightly but the same basic outcome is guaranteed.

Consider this median of medians. Since it's the median of a set of $n/5$ elements it's greater than or equal to $\lceil (n/5)/2 \rceil = \lceil n/10 \rceil$ of them (because the median of N elements is greater than or equal to $\lceil N/2 \rceil$ of them by definition). However those themselves are greater than or equal to three others each, since they're each medians of 5. Thus our median of medians is greater than or equal to $3 \lceil \frac{n}{10} \rceil \geq \frac{3}{10}n$ elements in the original list.

In a symmetric fashion we can see that our median of medians is less than or equal to $\frac{3}{10}n$ elements in the original list.

Thus if we choose either of the sublists consisting of elements less than the median of medians or the elements greater than the median of medians and denote its length by $f(n)$ then we know:

$$\frac{3}{10}n \leq f(n) \leq \frac{7}{10}n$$

Example 4.2. Consider this list of 35 elements. Each column has been ordered and the median of medians has been shaded:

5	6	3	9	28	1	50
8	7	17	13	29	2	51
10	15	35	16	32	30	57
11	21	70	22	33	31	58
12	99	71	37	60	36	73

Now let's shade the medians of the other columns which are less than or equal to 30 and within those columns let's shade the values which are less than or equal to their respective medians:

5	6	3	9	28	1	50
8	7	17	13	29	2	51
10	15	35	16	32	30	57
11	21	70	22	33	31	58
12	99	71	37	60	36	73

We can see that our median of medians, 30, is guaranteed to be greater than or equal to 3 entries in each of $4 = \lceil \frac{\# \text{Columns}}{2} \rceil = \lceil \frac{n}{2} \rceil$ columns. So it is greater than or equal to $3 \lceil n/10 \rceil$ entries in the entire list.

Similarly here is the picture showing the values which are guaranteed to be greater than or equal to our median of medians:

5	6	3	9	28	1	50
8	7	17	13	29	2	51
10	15	35	16	32	30	57
11	21	70	22	33	31	58
12	99	71	37	60	36	73

■

If the number of elements is not an odd multiple of 5 then we need to tweak the argument slightly because it may not be true for smaller n but turns out to be true for large enough n , which is sufficient for time complexity arguments. I have opted out of fiddling with the details because I think keeping it simple helps understand what's going on.

We'll explore some of these quirks in the exercises.

4.5 Return to Time Complexity

Recalling that we designated $f(n)$ to be the function returning the size of the sublist after `partition` is called, we now know that for a list of length n if we choose the pivot value according to the median of medians approach we have:

$$\frac{3}{10}n \leq f(n) \leq \frac{7}{10}n$$

So now suppose that a call to `selectkth` takes time $T(n)$. This call will then involve a call to `mom`, a call to `partition`, and a call to `selectkth`. Observe:

- The call to `mom` involves the division into sublists, the small sorts, the new list construction (length $n/5$) and a further call to `selectkth`. For a list of length n this takes time $\Theta(n) + T(0.2n)$
- The call to `partition` takes time $\Theta(n)$.
- The new call to `selectkth` will be on a sublist of length at most $\frac{7}{10}n$ hence will take time $T(0.7n)$.

Thus we now have a worst-case recurrence inequality involving $T(n)$:

$$T(n) \leq \Theta(n) + T(0.2n) + \Theta(n) + T(0.7n)$$

We can rewrite this as:

$$T(n) \leq T(0.7n) + T(0.2n) + f(n) \quad \text{with } f(n) = \Theta(n)$$

We'd like to use this to figure out the time complexity of $T(n)$. Because of the inequality we'll only be able to get a \mathcal{O} time complexity on the worst-case. Note that it is certainly possible to use the lower bound of $\frac{3}{10}n$ to obtain a Ω time complexity on the worst-case but we won't do that here.

Theorem 4.5.1. The recurrence relation:

$$T(n) \leq T(0.7n) + T(0.2n) + f(n) \quad \text{with } f(n) = \Theta(n)$$

satisfies $T(n) = \mathcal{O}(n)$.

Proof. The proof is by strong induction.

Two things to note first:

- Since $f(n) = \Theta(n)$, we know there is some C_0 and n_0 such that if $n \geq n_0$ that $f(n) \leq C_0 n$.
- For each n with $n \leq 5n_0$ the time $T(n)$ is a fixed constant. Define:

$$M = \max \left\{ \frac{T(n)}{n} \mid 1 \leq n \leq 5n_0 \right\}$$

Then for all $n \leq 5n_0$ we have $T(n)/n \leq M$ and so $T(n) \leq Mn$.

We claim that for all $n \geq n_0$ we have $T(n) \leq C_1 n$ where $C_1 = \max \{10C_0, M\}$.

Base Cases: The base cases cover all of $n = n_0, \dots, 5n_0$. These are clear because for $n \leq 5n_0$ we know $T(n) \leq Mn \leq C_1 n$.

Inductive Step: The inductive step applies to $n = 5n_0$ onwards. For $n \geq 5n_0$ we assume that $T(k) \leq C_1 k$ for all $n_0 \leq k < n$ and we'll prove that $T(n) \leq C_1 n$.

Observe that $n_0 = 0.2(5n_0) \leq 0.2n < n$ so the induction hypothesis applies to $0.2n$ and so $T(0.2n) \leq C_1(0.2n)$. Likewise $T(0.7n) \leq C_1(0.7n)$.

From here we get:

$$\begin{aligned} T(n) &\leq T(0.7n) + T(0.2n) + f(n) \\ &\leq C_1(0.7n) + C_1(0.2n) + C_0 n \\ &\leq 0.9C_1 n + C_0 n \\ &\leq 0.9C_1 n + 0.1C_1 n \\ &\leq C_1 n \end{aligned}$$

This finishes the induction step.

\mathcal{QED}

5 QuickSort-Related Note

Recall that the time complexity of QuickSort was challenging because the best-case is achieved by somehow choosing the pivot value such that the two sublists are of equal size. However this involves choosing the median at each step and this is a challenging addition to the code. In response the typical approach is to select the pivot value randomly which gives rise to $\mathcal{O}(n \lg n)$ average-case time complexity.

Another choice is to use our new median-of-medians approach to select the pivot value. In this case each time the list gets split we can guarantee that one side is at least $3/10$ the size of the original list and the other side is at most $7/10$ the size of the original list.

The worst-case then arises when we get exactly the $30/70$ split each time. The resulting recurrence relation is then:

$$T(n) = T\left(\frac{3}{10}n\right) + T\left(\frac{7}{10}n\right) + \Theta(n)$$

This cannot be solved by the Master Theorem but the Akra-Bazzi method yields a solution of $\Theta(n \lg n)$.

As theoretically interesting as this is, for what it's worth the time overhead hidden in the repeated application of the median-of-medians approach makes this approach not useful in practice.

6 Thoughts, Problems, Ideas:

1. If we used a max heap instead of a min heap to find the k th order statistic and assuming we could build the max heap in $\mathcal{O}(n)$ time, what would the \mathcal{O} time complexity of extracting the k th smallest element be?

2. Assume k is given. Modify BubbleSort so that after i iterations the first i elements are correctly positioned and so that it stops when exactly the first k elements are correctly positioned.

3. Given the list of 45 distinct elements:

3, 43, 29, 41, 32, 59, 85, 7, 60, 4, 31, 11, 20, 8, 80,
66, 22, 50, 16, 90, 26, 79, 2, 96, 6, 54, 81, 93, 53, 99,
61, 36, 62, 14, 40, 17, 30, 95, 34, 74, 5, 98, 64, 72, 87

(a) Separate the list into columns containing five elements each and sort each column.

(b) Identify the median of medians.

(c) Draw a table identifying the values guaranteed to be less than or equal to the MOM

(d) Draw a table identifying the values guaranteed to be greater than or equal to the MOM

4. In determining the median of medians of a list containing n distinct elements suppose n is an even multiple of 5. Consequently the median of medians will not actually be an element in the list so instead we choose the “lower median” which is the largest value below the median. Show that the MOM is still greater than or equal to $\frac{3}{10}n$ elements in the original list but the less than or equal to bound is slightly better.

5. In determining the median of medians of a list containing n distinct elements suppose n is an even multiple of 5 plus 1. We group the elements into an odd number of groups of five elements and a single group of one element and proceed as before. Show that the MOM is still greater than or equal to $\frac{3}{10}n$ elements.

6. In the “General Argument, Nice Case” for the median of medians we assumed n to be an odd multiple of 5 and obtained $\frac{3}{10}n \leq f(n) \leq \frac{7}{10}n$. Calculate the bounds by modifying the argument for the case in we choose an odd multiple of 7 instead.

7 Python Test and Output - Naïve

Code:

```
import random

def kthorder(A,k):
    n = len(A)
    mini = A[0]
    maxi = A[0]
    for i in range(0,n):
        if A[i] < mini:
            mini = A[i]
        if A[i] > maxi:
            maxi = A[i]
    kth = mini
    print('Minimum: ' + str(kth))
    for i in range(1,k):
        nextsmallest = maxi
        for i in range(0,n):
            if (A[i] < nextsmallest) and (A[i] > kth):
                nextsmallest = A[i]
        kth = nextsmallest
        print('Next Smallest: ' + str(kth))
    return(kth)

A = []
while len(A) < 20:
    r = random.randint(0,100)
    if r not in A:
        A.append(r)

print(A)

print('Final Answer: ' + str(kthorder(A,5)))
```

Output:

```
[67, 84, 27, 56, 82, 57, 15, 13, 66, 41, 30, 97, 52, 99, 87, 47, 45, 20, 53,
Minimum: 13
Next Smallest: 15
Next Smallest: 20
Next Smallest: 27
Next Smallest: 30
Final Answer: 30
```

8 Python Test and Output - Median of Medians

Code Comment:

The process of repeatedly partitioning a given list alters that list and this is reflected in the fact that the functions take the array argument by reference (the default in Python).

Observe however that the Median of Median function creates a new list (the list of medians of groups of mostly five) on which the process is recursively called. When creating that new list, `mom` needs to alter the working list by sorting in groups, hence it makes a copy using `AA = A[:]`, and then works on `AA`.

Consequently it may be helpful to keep in mind that there are several arrays being managed, each of them by reference.

Code Part 1:

```
import random
import math

def selectkth(A,l,r,kwish):
    # Find the pseudomedian.
    pmed = mom(A[l:r+1])
    # Find the index of the pseudomedian
    pmedi = 0
    while A[pmedi] != pmed:
        pmedi = pmedi + 1
    # Swap that entry with the final entry.
    A[r],A[pmedi] = A[pmedi],A[r]
    # Partition on the final entry.
    pivotindex = partition(A,l,r)
    if kwish < pivotindex+1:
        return(selectkth(A,l,pivotindex-1,kwish))
    elif kwish > pivotindex+1:
        return(selectkth(A,pivotindex+1,r,kwish))
    else:
        return(A[pivotindex])

def partition(A,l,r):
    pivot = A[r]
    t = l
    for i in range(l,r):
        if A[i] <= pivot:
            A[t],A[i] = A[i],A[t]
            t = t + 1
    temp = A[t]
    A[t] = A[r]
    A[r] = temp
```

```
return t
```

Code Part 2:

```
def mom(A):
    # Make a copy because we're going to mess it up doing our grouped sorting
    AA = A[:]
    n = len(AA)
    mlist = []
    for i in range(0,int(math.ceil(float(n)/5))):
        Li = 5*i
        Ri = Li + 5
        if Ri > n-1:
            Ri = n-1
        AA[Li:Ri] = sorted(AA[Li:Ri])
        mlist.append(AA[Li+(Ri-Li-1)//2])
    if len(mlist)==1:
        return mlist[0]
    s = selectkth(mlist,0,len(mlist)-1,(len(mlist)+1)//2)
    return(s)

LIST = []
while len(LIST) < 20:
    r = random.randint(0,100)
    if r not in LIST:
        LIST.append(r)

print('Array: '+str(LIST))
kwish = random.randint(1,len(LIST))
kth = selectkth(LIST,0,len(LIST)-1,kwish)
print("I'm looking for rank: " + str(kwish))
print('It is: ' + str(kth))
print('Here is the Python sorted array for checking:')
LIST.sort()
print(LIST)
if LIST[kwish-1] == kth:
    print('Success!')
```

Output:

```
Array: [97, 22, 52, 92, 34, 54, 2, 30, 3, 43, 13, 86, 28, 16, 33, 51, 1, 0,
I'm looking for rank: 12
It is: 34
Here is the Python sorted array for checking:
[0, 1, 2, 3, 13, 16, 22, 28, 30, 32, 33, 34, 41, 43, 51, 52, 54, 86, 92, 97]
Success!
```

Output:

```
Array: [58, 77, 27, 41, 92, 19, 44, 7, 65, 31, 85, 84, 57, 94, 81, 71, 20, 8
I'm looking for rank: 3
It is: 19
Here is the Python sorted array for checking:
[5, 7, 19, 20, 27, 31, 41, 44, 55, 57, 58, 65, 71, 77, 81, 82, 84, 85, 92, 9
Success!
```

Output:

```
Array: [48, 11, 63, 59, 21, 41, 72, 43, 61, 74, 34, 9, 47, 100, 73, 38, 28,
I'm looking for rank: 18
It is: 87
Here is the Python sorted array for checking:
[9, 11, 21, 28, 34, 38, 41, 42, 43, 47, 48, 59, 61, 63, 72, 73, 74, 87, 89,
Success!
```

CMSC 351: Depth-First Traverse

Justin Wyss-Gallifent

July 28, 2024

1	Introduction:	2
2	Intuition	2
3	Visualization	2
4	Algorithm Implementation	2
5	Recursive Implementation	3
5.1	Pseudocode	3
5.2	Pseudocode Time Complexity	4
6	Stack Implementation	5
6.1	Pseudocode	5
6.2	Pseudocode Time Complexity	7
7	Stack/Doubly-Linked-List Implementation	7
7.1	Introduction	7
7.2	Pseudocode	9
7.3	Pseudocode Time Complexity	11

1 Introduction:

Suppose we are given a graph G and a starting node s . Suppose we wish to simply traverse the graph in some way looking for a particular value associated with a node. We're not interested in minimizing distance or cost or any such thing, we're just interested in the traversal process.

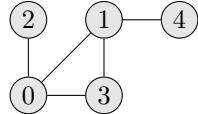
2 Intuition

One classic way to go about this is a depth-first traverse. The intuitive idea is that starting with a starting node s we follow one path as far as possible before backtracking. When we backtrack we only do so as little as possible until we can go deeper again.

Observe that this description does not lead to a unique traversal because there may be multiple paths that we can follow from a given vertex.

3 Visualization

Before writing down some explicit pseudocode let's look at an easy graph and look at how the above intuition might pan out. Consider this graph:



Suppose we start at the vertex $s = 0$. We have three edges we can follow, let's suppose we follow the edge to the vertex 3 first. From 3 we can only go to 1 (we can't go back to 0 since we've visited it already) and then to 4 (we can't go back to 0 or 3). At that point we have gone as deep as we can along that branch so we go back to the most recent branch for which there are other paths available. We have to go back to 0 and from there we go to 2. Thus our depth-first traverse follows the vertices in order 0, 3, 1, 4, 2.

4 Algorithm Implementation

There are several classic approaches to constructing an algorithm for depth-first search to which we will add one more for reasons we shall make clear:

- Using recursion.
- Using a stack.
- Using a stack which is modified to be a doubly-linked-list.

5 Recursive Implementation

5.1 Pseudocode

The pseudocode for the recursive implementation is as follows:

```
// These are global.
VORDER = []
VISITED = list of length V full of FALSE
function dft(G,x):
    VORDER.append(x)
    VISITED[x] = TRUE
    for all adjacent nodes y (in some order)
        if VISITED[y] == FALSE
            dft(G,y)
        end if
    end for
end function
dft(G,s)
```

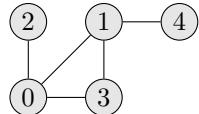
In this pseudocode the list `VORDER` will record the order in which we visit the vertices while the list `VISITED` will indicate whether or not a vertex has been visited.

The line:

```
for all adjacent nodes y (in some order)
```

does not suggest which order we should follow the adjacent nodes in. In what follows we'll follow them in decreasing order.

Example 5.1. Let's return to our example from earlier:



We'll start our traversal at the vertex $x = 0$. Before the function is called we have:

index	0	1	2	3	4
VORDER					
VISITED	F	F	F	F	F

Our first call is `dft(G,0)` and before the `for` loop we then have:

index	0	1	2	3	4
VORDER	0				
VISITED	T	F	F	F	F

The **for** loop cycles through the vertices 3,2,1 (call this depth 1) and the first call is `dft(G,3)` which yields, before its own **for** loop:

index	0	1	2	3	4
VORDER	0	3			
VISITED	T	F	F	T	F

From `dft(G,3)` the **for** loop cycles through the vertices 1,0 (call this depth 2) and the first call is `dft(G,1)` which yields, before its own **for** loop:

index	0	1	2	3	4
VORDER	0	3	1		
VISITED	T	T	F	T	F

From `dft(G,1)` the **for** loop cycles through the vertices 4,0 (call this depth 3) and the first call is `dft(G,4)` which yields, before its own **for** loop:

index	0	1	2	3	4
VORDER	0	3	1	4	
VISITED	T	T	F	T	T

From `dft(G,4)` the **for** loop cycles through the vertices 1 (call this depth 4) but since that vertex has been visited, `dft` is not called on it again and we are sent back to depth 3 and our loop is on vertex 0 but since that vertex has been visited, `dft` is not called on it again and we are sent back to depth 2 and our loop is on vertex 0 but since that vertex has been visited, `dft` is not called on it again and we are sent back to depth 1 and our loop is on vertex 2 so we call `dft(G,2)` which yields, before its own **for** loop:

index	0	1	2	3	4
VORDER	0	3	1	4	2
VISITED	T	T	T	T	T

From `dft(G,2)` the **for** loop cycles through the vertices 0 but since that vertex has been visited, `dft` is not called on it again and we are sent back to depth 1 and our loop is on vertex 1 but since that vertex has been visited, `dft` is not called on it again.

Then we are done. Observe that the order in which we visited the nodes is 0, 3, 1, 4, 2.

5.2 Pseudocode Time Complexity

Suppose V is the number of nodes and E is the number of edges. What follows is exactly the same as breadth-first traverse so if that made sense you can possibly skip this.

- The initialization takes $\mathcal{O}(V)$. This could in fact take $\Theta(1)$ depending on the architecture but the choice has no effect on the result.
- Each node gets processed once so this is $V\Theta(1)$ each for a total of $\Theta(V)$.

- Since each edge is attached to two nodes the `for` loop will iterate a total of $2E$ times over the course of the entire algorithm. This gives a total of $\Theta(2E) = \Theta(E)$.

The time complexity is therefore $\mathcal{O}(V) + \Theta(V) + \Theta(E) = \mathcal{O}(V + E)$. If initialization is actually $\Theta(1)$ then this becomes $\Theta(V + E)$.

Note 5.2.1. Note that our pseudocode and analysis assumes we have direct access to a node's edges using something like an adjacency list. If we use an adjacency matrix then the inner loop becomes $\Theta(V)$ and the entire pseudocode becomes $\Theta(V^2)$.

6 Stack Implementation

6.1 Pseudocode

The pseudocode for the stack implementation is as follows:

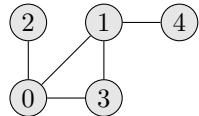
```
VORDER = []
VISITED = list of length V full of FALSE
STACK = [s]
while STACK is not empty:
    x = STACK.pop()
    if VISITED[x] == FALSE:
        VISITED[x] = TRUE
        VORDER.append(x)
    end if
    for all nodes y adjacent to x:
        if VISITED[y] == FALSE:
            STACK.append(y)
        end if
    end for
end while
```

The line:

for all nodes y adjacent to x

does not suggest which order we should follow the adjacent nodes in. In what follows we'll follow them in increasing order.

Example 6.1. Let's return to our example from earlier:



We'll start our traversal at the vertex $x = 0$. We start with the following

before the `while` loop:

	S = [0]				
index	0	1	2	3	4
VORDER					
VISITED	F	F	F	F	F

Iterate! We pop $x = 0$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertices 1, 2, 3 and push them all onto the stack:

	S = [1,2,3]				
index	0	1	2	3	4
VORDER	0				
VISITED	T	F	F	F	F

Iterate! We pop $x = 3$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertices 0, 1 and push 1 (but not 0) onto the stack:

	S = [1,2,1]				
index	0	1	2	3	4
VORDER	0	3			
VISITED	T	F	F	T	F

Iterate! We pop $x = 1$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertices 0, 3, 4 and push 4 (but not 0, 3) onto the stack:

	S = [1,2,4]				
index	0	1	2	3	4
VORDER	0	3	1		
VISITED	T	T	F	T	F

Iterate! We pop $x = 4$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertex 1 but nothing is pushed onto the stack:

	S = [1,2]				
index	0	1	2	3	4
VORDER	0	3	1	4	
VISITED	T	T	F	T	T

Iterate! We pop $x = 2$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertex 0 but nothing is pushed onto the stack:

$$S = [1]$$

index	0	1	2	3	4
VORDER	0	3	1	4	2
VISITED	T	T	T	T	T

Iterate! We pop $x = 1$ off the stack. Since it's visited we do nothing with it. We then iterate over the vertices 0, 3, 4 but nothing is pushed onto the stack:

S = []					
index	0	1	2	3	4
VORDER	0	3	1	4	2
VISITED	T	T	T	T	T

Then we are done. Observe that the order in which we visited the nodes is 0, 3, 1, 4, 2.

6.2 Pseudocode Time Complexity

It's not uncommon for various online sources to give essentially this pseudocode and state that it has time complexity $\Theta(V + E)$ but this is false as can be easily demonstrated.

Consider a graph with V vertices which is *complete*, meaning each vertex is connected to every other vertex. We start by pushing the starting vertex onto the stack.

When we pop this vertex there is 1 visited vertex and the **for** loop will push all of the remaining $V - 1$ (unvisited) vertices onto the stack.

When we pop the next vertex there are 2 visited vertices and the **for** loop will push $V - 2$ (unvisited) vertices onto the stack, all of which will be repeats.

This will repeat through the entire process and all together the number of vertices which get pushed onto the stack will be:

$$1 + (V - 1) + (V - 2) + \dots + 2 + 1 + 0 = 1 + \frac{(V - 1)V}{2} = \Theta(V^2)$$

Since the **while** loop iterates once for each vertex on the stack it will iterate $\Theta(V^2)$ times.

The **for** loop will (as with the argument for the recursive implementation) iterate $2E$ times for a final time complexity of $\Theta(V^2 + E)$.

7 Stack/Doubly-Linked-List Implementation

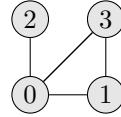
7.1 Introduction

It is possible to modify the stack version to bring it down to $\Theta(V + E)$. The trick is to find a way to ensure that an item is only ever popped off the stack

once, but we can't be sloppy about it. In theory there are two ways to go about this. When we are about to push something onto the stack, other than checking if it has been visited:

- (a) We check whether it has been or still is on the stack and if so, we don't push it.
- (b) We check whether it has been on the stack, we don't push it, and if it is on the stack, we remove the earlier occurrence.

While (a) seems easier to do it does not work, as is easily demonstrated by this graph:



Examine approach (a). Let's start at 0, so $S = [0]$. We then pop $x = 0$ (marking it as visited) and suppose we push 3, 2, 1 onto the stack in that order, so $S = [3, 2, 1]$. We then pop $x = 1$ (marking it as visited) and don't push anything, so $S = [3, 2]$. We then pop $x = 2$ (marking it as visited) and don't push anything, so $S = [3]$. We then pop $x = 3$ (marking it as visited), and don't push anything, so $S = []$, and then we are done. However we have now visited the vertices in the order 0, 1, 2, 3 which is not a BFT since after visiting 1 we should go to 3.

On the other hand examine approach (b). Let's start at 0, so $S = [0]$. We then pop $x = 0$ (marking it as visited) and suppose we push 3, 2, 1 onto the stack in that order, so $S = [3, 2, 1]$. We then pop $x = 1$ (marking it as visited) and we push 3, replacing the earlier occurrence, so $S = [2, 3]$. We then pop $x = 3$ (marking it as visited) and we push nothing, so $S = [2]$. We then pop $x = 2$ (marking it as visited) and don't push anything, so $S = []$, and then we are done. Now we have visited the vertices in the order 0, 1, 3, 2.

One way to accomplish (b) is to turn the stack into a doubly-linked-list and to keep an array of pointers, one for each vertex, which point to the vertex's location on the stack and are NULL by default. When a vertex is pushed onto the stack we check if its pointer is NULL and if not then we remove the previous vertex from within the stack (this is $\Theta(1)$ for a doubly-linked-list) and then push it on the end of the stack and update the pointer.

7.2 Pseudocode

The pseudocode for this new implementation is as follows:

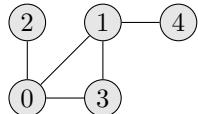
```
VORDER = []
STACK = [s] (functions as doubly-linked-list)
VISITED = list of length V full of FALSE
SP = list of length V full of NULL pointers
SP[x] = points to x in STACK
while STACK is not empty:
    x = STACK.pop()
    if VISITED[x] == FALSE:
        VISITED[x] = TRUE
        VORDER.append(x)
    end if
    for all nodes y adjacent to x:
        if VISITED[y] == FALSE:
            if SP[y] != NULL:
                delete y from STACK
            end if
            STACK.append(y)
            SP[y] = point to y in STACK
        end if
    end for
end while
```

The line:

```
for all adjacent nodes y (in some order)
```

does not suggest which order we should follow the adjacent nodes in. In what follows we'll follow them in increasing order.

Example 7.1. Let's return to our example from earlier:



Illustrating the pointers is a bit of a pain so we will avoid doing so. Instead the critical difference between this implementation and the stack implementation is that whenever a vertex is pushed onto the stack we check if it is already on the stack and if so we delete its earlier occurrence. In what follows this is the only significant difference.

We'll start our traversal at the vertex $x = 0$. We start with the following before the `while` loop:

S = [0]					
index	0	1	2	3	4
VORDER					
VISITED	F	F	F	F	F

Iterate! We pop $x = 0$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertices 1, 2, 3 and push them all onto the stack:

S = [1,2,3]					
index	0	1	2	3	4
VORDER	0				
VISITED	T	F	F	F	F

Iterate! We pop $x = 3$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertices 0, 1 and push 1 (but not 0) onto the stack which deletes the earlier 1:

S = [2,1]					
index	0	1	2	3	4
VORDER	0	3			
VISITED	T	F	F	T	F

Iterate! We pop $x = 1$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertices 0, 3, 4 and push 4 (but not 0, 3) onto the stack:

S = [2,4]					
index	0	1	2	3	4
VORDER	0	3	1		
VISITED	T	T	F	T	F

Iterate! We pop $x = 4$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertex 1 but nothing is pushed onto the stack:

S = [2]					
index	0	1	2	3	4
VORDER	0	3	1	4	
VISITED	T	T	F	T	T

Iterate! We pop $x = 2$ off the stack. Since it's not visited we mark it as such and append it to the visiting order. We then iterate over the vertex 0 but nothing is pushed onto the stack:

$$S = []$$

index	0	1	2	3	4
VORDER	0	3	1	4	2
VISITED	T	T	T	T	T

Then we are done. Observe that the order in which we visited the nodes is 0, 3, 1, 4, 2.

7.3 Pseudocode Time Complexity

With this modification each vertex is popped exactly once and so the `while` loop iterates V times. The `for` loop iterates $2E$ times and we then we have a total time complexity of $\Theta(V + E)$ once more.

CMSC 250: Program Verification

1. The Big Idea

We'd like to verify that an algorithm does what it should. What this essentially means is that before the algorithm we have a certain number of pre-conditions that are set, then the algorithm executes, and then we have some post-conditions which we require to be satisfied.

Defn: We say that the algorithm is correct if, when various pre-conditions are set and the algorithm is executed, the post-conditions are met.

2. A Pseudo-Example.

Suppose we wish to write an algorithm which takes a positive real n and finds another real m with $m^2 = n$. The basic structure would then look like this:

```
\\" Precondition: n is any positive real.  
MYSTERIOUS ALGORITHM: Calculates m somehow.  
\\" Postcondition: m^2 = n
```

Here are some examples of correct and incorrect algorithms:

Correct:

```
\\" Precondition: n is any positive real.  
Assign m = sqrt(n)  
\\" Postcondition: m^2 = n
```

Correct:

```
\\" Precondition: n is any positive real.  
Assign m = -sqrt(n)  
\\" Postcondition: m^2 = n
```

Incorrect (because the post-condition will not be met for any positive real):

```
\\" Precondition: n is a positive integer.  
Assign m = n  
\\" Postcondition: m^2 = n
```

3. Fact: This is Hard!

It's not easy to verify whether a given algorithm is correct, especially when the algorithm is long and complicated. Testing can be performed whereby lots of input is provided and output is tested and this can provide some assurance but it is by no means a proof.

Here we'll focus specifically on the correctness of loops.

4. Correct Loops

We can make the above definition more specific when it comes to a loop.

Defn: We say that a loop is correct (with regards to its pre- and post-conditions) if, when various pre-conditions are met and the loop terminates after a finite numbers of steps, the post-conditions are met.

5. Loop Invariants

Focusing on loops allows us to introduce the concept of a loop invariant. A loop invariant is a predicate (either T or F) whose status we can monitor through the loop and which can help us determine if a loop is correct.

Defn: A loop invariant is a predicate which satisfies the following:

- For each iteration, if it is true at the start of the iteration then it is true when the iteration has finished.

In and of itself this is not particularly useful but when we attach two other conditions:

- It is true before the first iteration of the loop.
- If the loop terminates after finitely many steps, then if the loop invariant is true then the post-conditions are satisfied.

Then the loop will be correct.

A loop invariant can be denoted $LI(n)$ where n denotes the iteration count of the loop. With this notation $LI(0)$ represents the loop invariant before the first iteration, $LI(n)$ represents the loop invariant at the start of the n^{th} iteration, $LI(n+1)$ represents the loop invariant at the end of the n^{th} iteration (or at the start of the $(n+1)^{\text{st}}$ iteration), and so on. When the loop ends the loop invariant will be represented by $LI(N)$ where N is the number of iterations which the loop took to complete.

The correctness of the loop as it relates to the loop invariant is then summarized in the:

Loop Invariant Theorem: Let a while loop with guard predicate G be given, along with pre- and post-condition predicates PRE and $POST$. Also let a predicate $LI(n)$ be given. Then, if the following four conditions are met, the loop will be correct:

- Basis Property:** If PRE is true then $LI(0)$ is true (in other words that LI is true before the first iteration of the loop).
- Inductive Property:** For all $k \geq 0$ if the guard G is true and if $LI(k)$ is true, then $LI(k+1)$ is true. Note that it can be helpful when doing this to introduce new variables to clarify when variables change within the loop. We'll see this in our examples.
- Termination:** After a finite number of steps we have $\sim G$, meaning the loop terminates.
- Correctness of Post-Condition:** If $LI(N)$ then $POST$, where N is the least number of steps after which the loop terminates (that is, G is false).

Proof: Suppose PRE is true. By the Basis Property then $LI(0)$ is true. The Basis Property coupled with the Inductive Property then tells us that $LI(n)$ is true for all $n \geq 0$. By Termination we know the loop terminates and so $\sim G$. Let N be the last number of steps after which it terminates then we know that $LI(N)$ is true. Then by the Correctness of Post-Condition we have $POST$.

\mathcal{QED}

In brief (this is a summary of the above) there are four things we have to prove:

- Basis:** If PRE then $LI(0)$.
- Inductive:** If G and $LI(k)$ then $LI(k+1)$.
- Termination:** Eventually $\sim G$.
- Correctness of Post:** If $LI(N)$ then $POST$, where N is the least number of steps after which $\sim G$.

6. **Example:** Finding x^p for any $x \in \mathbb{R}^*$ and any $p \in \mathbb{Z}^+$.

We have a nonzero real number x and a positive integer p and we wish to calculate x^n . Here is the pseudocode:

```
\\" PRE: x is a nonzero real number
\" PRE: p is a positive integer
\" PRE: i = 1
\" PRE: result=1
while i <= p
    result=result*x
    i=i+1
end
\" POST: result = x^p
```

We'll introduce the list invariant;

$LI(n): n = i-1 \text{ and } result = x^{(i-1)}$

Now then let's observe the requirements of the theorem:

Base: Check that PRE implies $LI(0)$.

Let's assume PRE, so then we have: x a nonzero real number, p a positive integer and $i = 1$.

We claim $LI(0)$, in other words, we claim that $0 = i-1$ and $result = x^0$

Well $i = 1$ and so $0 = 1-1$ and $result = 1 = x^0$ as desired.

Induction: Check that for all $k \geq 0$ if the guard G is true and if $LI(k)$ is true, then $LI(k+1)$ is true.

Suppose the guard is true (so $i \leq n$) and $LI(k)$ is true. To have $LI(k)$ true means:

$k = i-1 \text{ and } result = x^{(i-1)}$

At the end of the iteration we have the new value of i , which we'll denote i' and a new $result$, which we'll denote $result'$, and we need to check that $LI(k+1)$ is true, meaning:

$k+1 = i'-1 \text{ and } result' = x^{(i'-1)}$

First, observe that $i' = i+1$ and so $k = i-1 \Rightarrow k+1 = i \Rightarrow k+1 = i'-1$.

Second, observe that $result = x^{(i-1)}$ and $result' = result * x$ tell us that at the end of the iteration we have $result' = x^{(i-1)} * x = x^i = x^{(i'-1)}$.

Termination: Check that after a finite number of steps we have $\sim G$.

The loop starts at $i = 1$, performs $i = i+1$, and terminates at $i = p+1$ so the loop will terminate after p iterations.

Post Check: Check that if N is the least number of steps after which $\sim G$ and if $LI(N)$, then POST.

Since the loop terminates after p iterations we have $N = p$. Suppose that $LI(N) = LI(n)$ is true, which means:

$N = i-1 \text{ and } result = x^{(i-1)}$

Then observe $result = x^{(i-1)} = x^N = x^p$, which is POST.

\mathcal{QED}

7. **Example:** Finding the maximum of a list.

We have a list A for which we wish to find the minimum. Here is the pseudocode:

```
\\" PRE: A = list of real numbers.
\\ PRE: len = length of A
\\ PRE: max = A[0]
\\ PRE: i = 1
while i < len
    if A[i] > max
        max = A[i]
    end
    i=i+1
end
\\ POST: max = maximum of A[0..len-1].
```

We'll introduce the list invariant;

$LI(n) : n = i-1 \text{ and } \max = \text{maximum value in } A[0..i-1]$

Now then let's observe the requirements of the theorem:

Base: Check that PRE implies $LI(0)$.

Let's assume PRE, so then we have: A a list of numbers, $\max = A[0]$ and $i = 1$.

We claim $LI(0)$, in other words, we claim that $0 = i-1$ and $\max = \text{maximum value in } A[0..i-1]$.

Well $i = 1$ and so $0 = 1-1$ and $\text{maximum value in } A[0..0] = A[0] = \max$ as desired.

Induction: Check that for all $k \geq 0$ if the guard G is true and if $LI(k)$ is true, then $LI(k+1)$ is true.

Suppose the guard is true (so $i < n$) and $LI(k)$ is true. To have $LI(k)$ true means:

$k = i-1 \text{ and } \max = \text{maximum value in } A[0..i-1]$.

At the end of the iteration we have the new value of i , which we'll denote i' and a new \max , which we'll denote \max' , and we need to check that $LI(k+1)$ is true, meaning:

$k+1 = i'-1 \text{ and } \max' = \text{maximum value in } A[0..i'-1]$

First, observe that $i' = i+1$ and so $k = i-1 \Rightarrow k+1 = i \Rightarrow k+1 = i'-1$.

Second, since at the start of the loop $\max = \text{maximum value in } A[0..i-1]$. The code examines $A[i]$ and if $A[i] > \max$ then \max is updated to this new value, otherwise it is left alone. Consequently at the end of the loop we have $\max' = \text{maximum value in } A[0..i] = A[0..i'-1]$ as desired.

Termination: Check that after a finite number of steps we have $\sim G$.

Since the loop starts at $i = 1$, performs $i = i+1$, and terminates at $i = \text{len}$ the loop will terminate after $\text{len}-1$ iterations.

Post Check: Check that if N is the least number of steps after which $\sim G$ and if $LI(N)$, then POST.

Since the loop terminates after $\text{len}-1$ iterations we have $N = \text{len}-1$. Suppose that $LI(N)$ is true, which means:

$N = i-1 \text{ and } \max = \text{maximum value in } A[0..i-1]$

Then observe $\max = \text{maximum value in } A[0..i-1] = A[0..N] = A[\text{len}-1]$, which is POST.

8. **Example:** Performing an insert sort on a list.

We have a list A and we wish to perform an insert sort. We're focusing on the outer `while` loop.

```
\\" PRE: A = a list
\" PRE: len = the length
\" PRE: i = 1
while i < len
    nextval = A[i]
    j = i-1
    while j >= 0 and nextval < A[j]
        A[j+1] = A[j]
        j = j-1
    end
    A[j+1] = nextval
    i = i + 1
end
\" POST: A is a sorted list
```

We'll introduce the list invariant;

$$LI(n) : n = i-1 \text{ and } A[0..i-1] \text{ is sorted}$$

Now then let's observe the requirements of the theorem:

Base: Check that PRE implies $LI(0)$.

Let's assume PRE, so then we have: A is a list and $i = 1$.

We claim $LI(0)$, in other words, we claim that $0 = i-1$ and $A[0..i-1]$ is ordered.

Well $i = 1$ and so $0 = 1-1$ and $A[0..i-1] = A[0]$ which is a single element and hence is ordered.

Induction: Check that for all $k \geq 0$ if the guard G is true and if $LI(k)$ is true, then $LI(k+1)$ is true.

Suppose the guard is true (so $i < n$) and $LI(k)$ is true. To have $LI(k)$ true means:

$$k = i-1 \text{ and } A[0..i-1] \text{ is sorted}$$

At the end of the iteration we have the new value of i , which we'll denote i' , and an updated list which we'll denote A' , and we need to check that $LI(k+1)$ is true, meaning:

$$k+1 = i'-1 \text{ and } A'[0..i'-1] \text{ is sorted}$$

First, observe that $i' = i+1$ and so $k = i-1 \Rightarrow k+1 = i \Rightarrow k+1 = i'-1$.

Second, at the start of the loop $A[0..i-1]$ is sorted. The code assigns `nextval = A[i]`, this is the value it needs to insert into the correct position in $A[0..i]$. The code starts at index $j=i-1$ and for each j if `nextval < A[j]` then `nextval` needs to be inserted somewhere before index j and so $A[j]$ is moved to the right by assigning `A[j+1] = A[j]`. Eventually either `nextval >= A[j]` which means `nextval` should go to the right of index j or else $j == -1$ which will happen if the code reaches the start of the list without finding a value smaller than `nextval`. Either way the inner while loop ends and `nextval` is inserted using `A[j+1] = nextval`. We now have, for our updated list A' , that $A'[0..i] = A'[0..i'-1]$ is sorted as desired.

Termination: Check that after a finite number of steps we have $\sim G$.

Since the loop starts at $i = 1$, performs $i = i+1$, and terminates at $i = \text{len}$ the loop will terminate after $\text{len}-1$ iterations.

Post Check: Check that if N is the least number of steps after which $\sim G$ and if $LI(N)$, then POST.

Since the loop terminates after $\text{len}-1$ iterations we have $N = \text{len}-1$. Suppose that $LI(N)$ is true.

$$N = i-1 \text{ and } A[0..i-1] \text{ is sorted}$$

Then observe $A[0..i-1] = A[0..N] = A[0..\text{len}-1]$ is sorted, which is POST.