Introduction

This chapter lays the groundwork for the material covered in the course and sets out the expectations for the readers.

The Interview Room

You're in an interview room, and you've just completed white-boarding the interviewer's question. Inside, you feel ecstatic; It's your dream company and you nailed the question. However, just when you think the coast is clear, the interviewer follows up with a question on the *time* or *space* complexity of your solution. The blank drawn across your face gives away your ineptness at analyzing algorithm complexity in space or time. In this common interview situation, most computer science graduates are still able to muster up a reasonable answer. For candidates lacking the formal education, however, reasoning about algorithms in terms of the ***big Oh*** notation becomes a ***big uh oh!***.

The intended readership of this course includes folks who are graduates of boot-camps, career-switchers into programming jobs or just established

industry veterans who want a quick revision on the topic of complexity theory. The course uses layman's terms to describe the concepts commonly used in the industry while intentionally avoiding the intricate mathematical gymnastics often involved in analyzing algorithms so that the subject material is comprehensible for readers at all skill levels. At the same time, this course may not deliver great value to individuals who already have a firm grasp of algorithms and data-structures or to those that hold a degree in computer science.

Why does complexity matter?

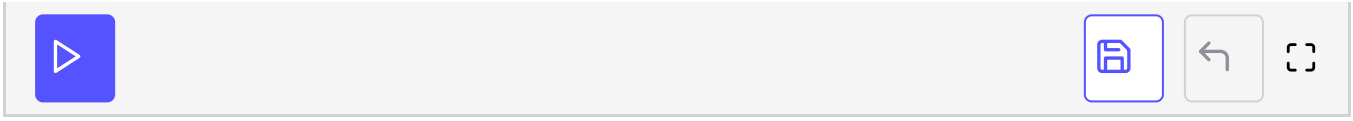
If you are working on a class assignment or a customer application with a few dozen users, you can take the liberty to ignore the space and time complexity of your solution. However, the moment you step into the professional world and write software with strict SLAs (service level agreements) or for millions of users, your choice of algorithm and data-structures starts to matter. A well designed and thought out software will scale elegantly and set itself apart from poorly written competitors.

Usually, the *101* example used to teach complexity is sorting of integers. Below are three example programs that sort integers using different sorting algorithms.

Bubble Sort

```
import java.util.Random;
class Demonstration {

    static int SIZE = 10000;
    static Random random = new Random(System.currentTimeMillis());
    static int[] input = new int[SIZE];

    public static void main( String args[] ) {
        createTestData();
        long start = System.currentTimeMillis();
        bubbleSort(input);
        long end = System.currentTimeMillis();
        System.out.println("Time taken = " + (end - start));
    }

    static void bubbleSort(int[] input) {
        for (int i = 0; i < SIZE; i++) {
            for (int j = 0; j < SIZE - 1; j++) {
                if (input[j] > input[j + 1]) {
                    int tmp = input[j];
                    input[j] = input[j + 1];
                    input[j + 1] = tmp;
                }
```

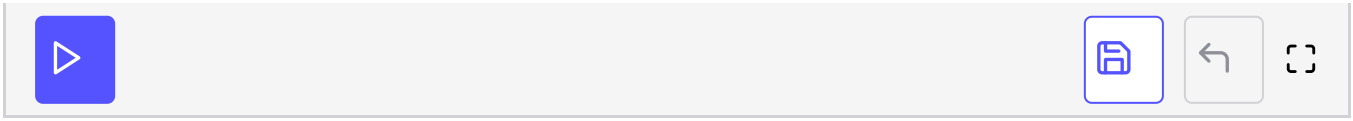
```
            }
        }
    }

    static void createTestData() {
        for (int i = 0; i < SIZE; i++) {
            input[i] = random.nextInt(10000);
        }
    }
}
```



Merge Sort

```
import java.util.Random;
class Demonstration {

    static int SIZE = 5000;
    static Random random = new Random(System.currentTimeMillis());
    static int[] input = new int[SIZE];
    static int[] scratch = new int[SIZE];

    public static void main( String args[] ) {
        createTestData();
        long start = System.currentTimeMillis();
        mergeSort(0, input.length - 1, input);
        long end = System.currentTimeMillis();
        System.out.println("Time taken = " + (end - start));
    }

    static void mergeSort(int start, int end, int[] input) {

        if (start == end) {
            return;
        }

        int mid = (start + end) / 2;

        // sort first half
        mergeSort(start, mid, input);

        // sort second half
        mergeSort(mid + 1, end, input);

        // merge the two sorted arrays
        int i = start;
        int j = mid + 1;
        int k;

        for (k = start; k <= end; k++) {
            scratch[k] = input[k];
        }

        k = start;
        while (k <= end) {

            if (i <= mid && j <= end) {
```

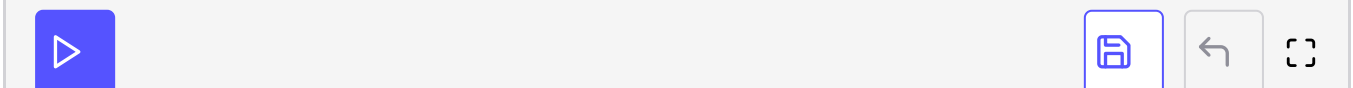
```
                input[k] = Math.min(scratch[i], scratch[j]);

                if (input[k] == scratch[i]) {
                    i++;
                } else {
                    j++;
                }
            } else if (i <= mid && j > end) {
                input[k] = scratch[i];
                i++;
            } else {
                input[k] = scratch[j];
                j++;
            }
            k++;
        }
    }

    static void createTestData() {
        for (int i = 0; i < SIZE; i++) {
            input[i] = random.nextInt(1000);
        }
    }
}
```



Merge Sort

Dual Pivot Quicksort

```
import java.util.Random;
import java.util.Arrays;

class Demonstration {

    static int SIZE = 5000;
    static Random random = new Random(System.currentTimeMillis());
    static int[] input = new int[SIZE];

    public static void main( String args[] ) {
        createTestData();
        long start = System.currentTimeMillis();
        Arrays.sort(input);
        long end = System.currentTimeMillis();
        System.out.println("Time taken = " + (end - start));
    }

    static void createTestData() {
        for (int i = 0; i < SIZE; i++) {
            input[i] = random.nextInt(10000);
        }
    }
}
```



Java's Dual Pivot Quicksort

The code implementations above run on datasets of 5000 elements. Below are some crude tests (run on a Macbook) for the three algorithms. All times are in milliseconds

Size	Bubble Sort	Merge Sort	Java's util.Arrays.sort
5000	90	9	4
10000	299	15	17
100000	24719	26	33
1000000	2474482	150	113

The results tabulated above should provide plenty of motivation for software developers to increase their understanding of algorithmic performance. Bubble sort progressively degrades in performance as the input size increases, whereas the other two algorithms perform roughly the same. We'll have more to say about their performances in the next chapter.

Time & Space

The astute reader would notice the use of an additional array in the merge sort code, named *scratch* array. This is the classic trade-off in the computer science realm. Throwing more space at a problem allows us to bring down the execution time, and vice versa. Generally, time and space have an inversely proportional relationship with each other; if space increases then execution time decreases, and if execution time increases then space decreases.

Need for Speed

This chapter discusses analogies from real life to draw a parallel between algorithmic complexity and comparisons we make in day to day situations.

The Race Track

Let's say you are a participant in your school's annual sports competition, which includes the 100-meter dash. You are a fast runner but not the fastest. You know Milkha is the fastest runner in the school and will probably finish first on the day of the race. The night before the competition you explain your predicament to your mother. She innocently asks, *"But how fast is Milkha?"* She hasn't seen you or Milkha run, and has no clue about your respective sprinting abilities. You reply back *"Faster than me."*. She quizzes back *"How much faster than you?"*. You quip back *"Faster than me, but definitely slower than Usain Bolt,"* someone your mother has seen run. By providing these comparisons to your mother, you are essentially setting a lower and upper bound on the running abilities of Milkha. Milkha will definitely run definitely faster than you, but will surely be slower than Usain Bolt.

You can also compare yourself directly with Usain Bolt and say you run slower than him, but then you are not ***bounding*** your skill tightly enough; you might as well say you run slower than the speed of light. That statement, though factually correct, isn't informative enough. The closer a reference point you can share with respect to your speed, the better your friends will be able to judge your chances of winning on the competition day. However, if you tell them you are even slower than Sogelau Tuvalu then surely no one will be betting money on you.

Note how you can provide both an upper bound and a lower bound. You run no faster than Usain Bolt and you run no slower than Sogelau. The tighter these bounds are, the better. If either of these bounds becomes loose, the less informative it is.

When algorithms or data-structures are analyzed for their time or space complexity, we are - in a sense - making statements similar to the sprinting competition scenario. An algorithm's performance or space requirement is bound by an upper and lower threshold. This threshold is defined in terms of mathematical functions that operate on the input size to the algorithm. Since all the algorithms that solve a given problem are analyzed in terms of the input size to them, it gives us the ability to uniformly compare them against each other and reason about the best solution.

Why not use a stop-watch?

We, as humans, are biased to measure performance using our stop-watches. A novice reader may suggest timing different algorithm runs just like cars on a racing track to determine the more performant one. However, in the case of algorithms, there are too many variations that can affect an algorithm's performance: the hardware used for the test, the characteristics of the input fed to the algorithms, the operating system or other programs concurrently running on the system, etc. All these factors can affect an algorithm's performance.

Here's a simplistic scenario: Suppose you are tasked with determining the performance of an array vs a linked-list for search/retrieval operation. Let's say you are looking for what is placed at the first index of both the data-structures. The time to retrieve the value at the first spot will come out to be the same on your stop-watch for both the data-structures, all else being equal. However, if you were looking for what was placed at the tail-end of each data-structure, the array data-structure would win hands down! In the case of linked-list you'll be walking down to the end of the chain to retrieve the value, whereas for the array it'll be a simple memory address retrieval operation; consequently, the choice of the data-structure is impacting the performance of our retrieval operation. If you based your stopwatch tests on retrieving an item that somehow gets placed at the beginning of each of the data-structures, the result wouldn't be informative enough.

This is further amplified in the case of sorting integers. The sorting experiment we ran in the previous section was similar to a stop-watch test, but it doesn't differentiate much between the performance of merge sort and dual-pivot quicksort algorithms. In order to truly understand the difference

dual-pivot quicksort algorithms. In order to truly understand the difference, we'll be required to undertake a mathematical analysis of both algorithms.

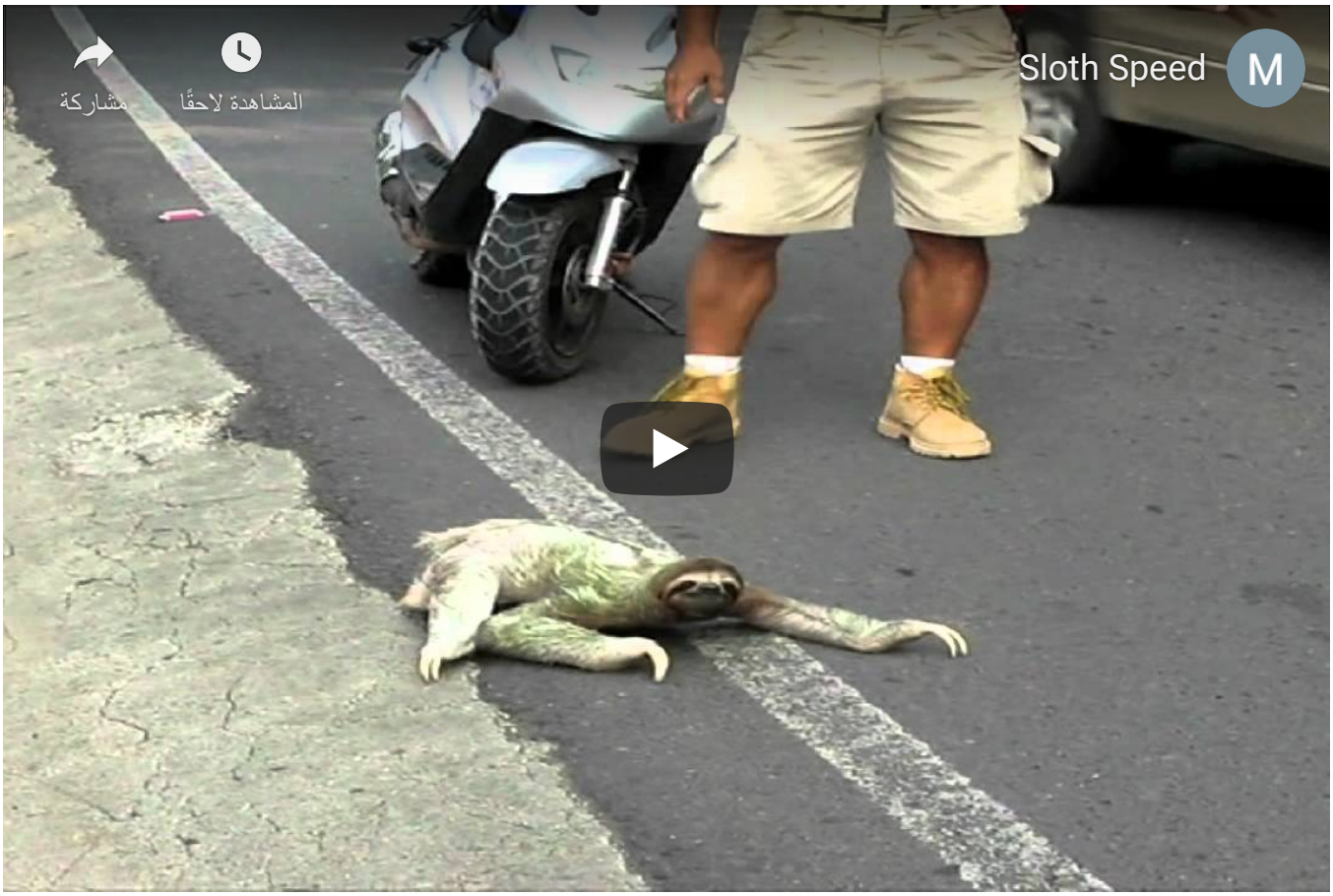
Theoretically, we can come up with inputs that'll make merge sort perform better than the quicksort algorithm but this may not be apparent from running simplistic *stop-watch* tests. Conversely, we can provide already sorted lists to different sorting algorithms and have them perform roughly the same, again clouding the true performance capabilities of each.

Uniform Comparison

We need to create a framework for analyzing algorithms and the performance of data-structures that is independent of the aforementioned variations, which can cloak the true performance of algorithms and data-structures. The basis of this framework is grounded in mathematics and tied to the input size.

Furthermore, algorithms can have their good days and bad days. On a good day, Usain Bolt broke the 100m sprint record; on a bad day, who knows he might even lose to you! But he'll definitely be faster than a sloth even on his worst day, and slower than a cheetah on his best day. Algorithms or data-structures have their good days called ***best case*** and their bad days called ***worst case***. Note the use of superlatives, as they can't really get better than their best case or worse than their worst case scenarios. Then there's the middle ground, called the **average case**, which is how an algorithm or a data-structure is expected to perform on most inputs.

In computer science and especially in interviews, we are concerned only about the worst case performance of an algorithm or a data-structure. If we have a handle on the worst case performance and consider it reasonable for the problem at hand - and unable to get worse - we can proceed in using that algorithm/data-structure. In fact, now we have a guarantee that it can't perform any worse than that. And in fact, it is likely to perform closer to its average case.


مشاركة
المشاهدة لاحقًا
Sloth Speed
M

Analyzing Algorithms

We start our journey into the world of complexity by analyzing the insertion sort algorithm.

Insertion Sort Analysis

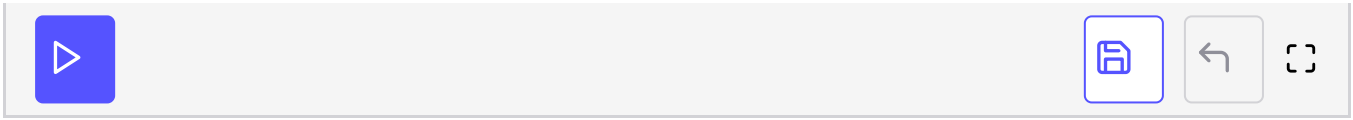
Let's start with the example of an array that contains 5 elements sorted in the worst possible order, i.e. descending when we want to sort the array in an ascending order. We'll use the insertion sort algorithm as the guinea pig for our analysis.

Array Sorted in Descending Order

We'll not delve into explaining the algorithm itself as our focus is on analyzing algorithm performance. The coursework assumes that readers have an understanding of the fundamental algorithms used in everyday computing tasks, or they can do a quick web-search to get up to speed on the algorithm under discussion. Let's proceed with the implementation of the algorithm first:

```
class Demonstration {

    static void insertionSort(int[] input) {
        for (int i = 0; i < input.length; i++) {
            int key = input[i];
            int j = i - 1;
            while (j >= 0 && input[j] > key) {
                if(input[j] > key){
                    int tmp = input[j];
                    input[j] = key;
                    input[j + 1] = tmp;
                    j--;
                }
            }
        }
    }

    static void printArray(int[] input) {
        System.out.println();
```

```
        for (int i = 0; i < input.length; i++)
            System.out.print(" " + input[i] + " ");
        System.out.println();
    }

    public static void main( String args[] ) {
        int[] input = new int[] {7, 6, 5, 4, 3, 2, 1};
        insertionSort(input);
        printArray(input);
    }
}
```



Below, we extract out the meat of the algorithm.

```
1.          for (int i = 0; i < input.length; i++) {
2.              int key = input[i];
3.              j = i - 1;
4.              while (j >= 0 && input[j] > key) {
5.                  if (input[j] > key) {
6.                      int tmp = input[j];
7.                      input[j] = key;
8.                      input[j + 1] = tmp;
9.                      j--;
10.                  }
11.             }
12.         }
```

Analyzing the program

For simplicity, let's assume that each code statement gets executed as a single machine instruction and we measure the total cost of running the program in terms of total number of instructions executed. This gives us a simple formula to measure the running time of our program expressed below:

Running Time** = **Instructions executed in For loop** + **Instructions executed in While loop

Analyzing the outer for loop

Let's start with the outer for loop and see how many instructions would get executed for each iteration of the outer loop. For now, forget about the inner while loop; we'll analyze it separately from the outer for loop.

Code Statement	Cost
1. for (int i = 0; i < input.length; i++) {	For the first iteration, the initialization statement is executed but not for any subsequent iteration. Similarly, the increment statement is executed for every iteration except the first one. The inequality check is made for every iteration. Note that when the loop exits, the increment and inequality check are executed again. Imagine a 3 element array then the instructions executed are 1st iteration : initialization and inequality 2nd iteration : increment and inequality 3rd iteration : increment and inequality 4th iteration : increment and inequality So we can conclude that we'll execute ***2 x (n + 1)*** instructions when an array contains ***n*** elements. Cost per iteration is 2 instructions and we have *n + 1* iterations.
2. int key = input[i];	executed once per iteration
3. int j = i - 1;	executed once per iteration
lines 4 - 11 skipping inner while loop	
12. }	
Total cost per iteration	**4 instructions**

Total cost for n iterations	**[2 x (n + 1) + 2n] instructions**

Analyzing the inner while loop

Code Statement	Cost
4. while (j >= 0 && input[j] > key) {	Loop starts with j = 0. The two conditions for the loop get tested once per iteration of the loop. However, there will be one additional execution of these conditions, when the loop exits. So if the loop runs for 3 iterations then the loop conditions get tested 4 times. Cost = 2 *per iteration* and 2 for the final exit of loop. Note for simplicity, we'll assume that both the conditions get tested even if the first is false i.e. we don't short-circuit.
5. if (input[j] > key) {	executed once per iteration of loop
6. int tmp = input[j];	executed once per iteration of loop
7. input[j] = key;	executed once per iteration of loop
8. input[j + 1] = tmp;	executed once per iteration of loop
9. j--;	executed once per iteration of loop
10. }	--
11. }	--
Total cost per iteration of	= 7 instructions

inner loop	= 7 instructions
Total cost for n iterations of inner loop	= **(n x 7)** + **2** instructions

Note that with the above analysis, we got a handle on the number of instructions that get executed ***per iteration*** of each loop given an input size of *n*. We can compute the instruction count for all iterations and sum them up to get the execution count for the entire run. In later sections, we'll compute a generalization that'll allow us to count the number of executions for the entire algorithm with a single formula.

It is also important to be cognizant that we are considering each line of code as a machine instruction, which isn't true. A line of code may result in several individual machine instructions; however, this shouldn't affect our analysis as long as we maintain the same yardstick when performing analysis across algorithms. Think of it as expressing the speed of an athlete in km/hour, miles/hour, centimeters/minutes etc. Whichever metric you choose for your representation, it will not affect the speed with which the athlete runs.

Analyzing Algorithms Part II

In this lesson, we'll dry run insertion sort and count the instructions executed on an array of length 5.

We'll now dry run our algorithm on the input array and trace the number of instructions executed in each iteration. We'll sum up the number of instructions across all the iterations to get a final count of instructions for the entire run. In the previous section, we worked out the instruction counts of the inner and outer loop to be those below:

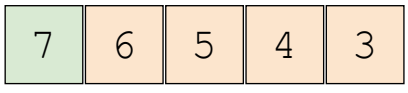
Number of instructions executed per iteration of outer for loop = 4
Number of instructions executed per iteration of inner while loop = 7

The code from the previous section is reproduced below:

```
1.          for (int i = 0; i < input.length; i++) {
2.              int key = input[i];
3.              j = i - 1;
4.              while (j >= 0 && input[j] > key) {
5.                  if (input[j] > key) {
6.                      int tmp = input[j];
7.                      input[j] = key;
8.                      input[j + 1] = tmp;
9.                      j--;
10.                  }
11.             }
12.         }
```

First Iteration of Outer Loop | when i = 0, key = 7

j = -1 inner loop doesn't execute



Total cost for 1st iteration = Outer Loop + Inner Loop
= 4 + [(iterations * 7) + 2]

= 4 + [(iterations * 7) + 2]

= 4 + [(0 * 7) + 2]
= 6 instructions

Second Iteration of Outer Loop | when i = 1, key = 6

j = 0

Total cost for 2nd iteration = Outer Loop + Inner Loop
= 4 + [(iterations * 7) + 2]
= 4 + [(1 * 7) + 2]
= 13 instructions

Third Iteration of Outer Loop | when i = 2, key = 5

j = 1

j = 0

Total cost for 3rd iteration = Outer Loop + Inner Loop
= 4 + [(iterations * 7) + 2]
= 4 + [(2 * 7) + 2]
= 20 instructions

Fourth Iteration of Outer Loop | when i = 3, key = 4

j = 2

j = 1

j = 0

Total cost for 1st iteration = Outer Loop + Inner Loop
= 4 + [(iterations * 7) + 2]
= 4 + [(3 * 7) + 2]
= 27 instructions

Fifth Iteration of Outer Loop | when i = 4, key = 3

j=3

j=2

j=1

j=0

Total cost for 1st iteration = Outer Loop + Inner Loop
= 4 + [(iterations * 7) + 2]
= 4 + [(4 * 7) + 2]

= 4 + [(4 * 7) + 2]
= 34 instructions

However, remember that to exit the loop the increment and the inequality statements will execute for one last time, so we need to add that cost too.
= 34 + 2
= 36 instructions

Iteration	Instructions Executed
1	6
2	13
3	20
4	27
5	36
Total Cost	6 + 13 + 20 + 27 + 36 = 102 instructions

One can observe that as the size of the array increases, the inner loop's iterations will correspondingly increase, and the running time of the program will increase.

Using Formulas

From the previous section, we can plug the formulas we produced to get the same result. There were 5 iterations of the outer for loop, so the instructions executed for the outer *for* loop are:

$$[2 * (n + 1) + 2n], \ where\ n\ is\ number\ of\ iterations$$

$$= [2 * (5 + 1) + 2 * 5]$$

$$= [12 + 10]$$

$$= 22\ instructions$$

The number of iterations for the inner loop varies for each iteration of the outer loop given by **(k x 7) + 2**

- No iteration on first run of outer loop.
  = (0 x 7) + 2 = 2
- 1 iteration in second run of outer loop:
  = (1 x 7) + 2 = 9
- 2 iterations on third run of outer loop:
  = (2 x 7) + 2 = 16
- 3 iterations on fourth run of outer loop:
  = (3 x 7) + 2= 23
- 4 iterations on fifth run of outer loop:
  = (4 x 7) + 2= 30
- Total instructions executed for the inner while loop are then: 2 + 9 + 16 + 23 + 30 = 80 instructions.

The number of instructions for the entire run comes out to be ***80*** + ***22*** = ***102 instructions***. Note how our formulas match the exact count we found from the dry run!

Analyzing Algorithms Part III

In this lesson, we will work out a generalization for the number of instructions executed for an array of length n

Generalizing Instruction Count

Let's try to generalize the running time for an array of size ***n***.

Code Statement	Cost
1. for (int i = 0; i < input.length; i++) {	executed **(2 x n)** + **2** times.
2. int key = input[i];	executed **n** times
3. int j = i - 1;	executed **n** times
4. while (j >= 0 && input[j] > key) {	We already calculated the cost of each iteration of the inner loop. We sum up the costs across all the iterations. Note that the inner loop will be executed ***n*** times and each execution will result in ***(iterations x 7)+2*** instructions being executed. And the number of iterations will successively increase from 0, 1, 2 all the way up to(n-1). See the dry run in the previous lesson to convince yourself of the validity of this result.
5. Inner loop statments	[(0 x 7) + 2] + [(1 x 7) + 2] + [(2 x 7) + 2] + ... [((n-1) x 7) + 2] = 2n + 7 * [0 + 1 + 2 + ... (n-1)]

11. } //inner loop ends 12. }	= 2n + 7 [$^{n(n-1)}/_{2}$]

If we use the above formulas and apply an array of size 5 to them, then the cost should match with what we calculated earlier.

$$Total\ Cost = Outer\ loop\ instructions + Inner\ loop\ instructions$$

$$= [2 * (n + 1) + 2n] + [2n + 7[\frac{n * (n - 1)}{2}]]$$

$$= [2 * (5 + 1) + 10] + [10 + 7[\frac{5 * (5 - 1)}{2}]]$$

$$= [12 + 10] + [10 + 7[\frac{5 * 4}{2}]]$$

$$= 22 + [10 + 7 * 10]$$

$$= 22 + 10 + 70$$

$$= 102\ instructions$$

Summation of Series

We glossed over how we converted the sum of the series 0 + 1 + 2 ...(n-1) to $^{n(n-1)}/_{2}$? However, even though we promised to steer clear of complicated mathematics as much as possible, this is one of the cardinal summations that you must know. Without a formal proof, remember that the sum of the first k natural numbers can be represented as:

$$1 + 2 + 3 + 4 \cdots k$$

$$= \frac{k * (k + 1)}{2}$$

If you add the first 5 natural numbers the sum is:

$$
\begin{aligned}
&= \frac{k * (k+1)}{2} \\
&= \frac{5 * (5+1)}{2} \\
&= \frac{30}{2} \\
&= 15
\end{aligned}
$$

However, our summation sums up to n-1 and includes a zero. We can drop the zero because adding zero to anything yields the same value. We know:

$$1 + 2 + 3 + 4 \cdots (k-2) + (k-1) + k = \frac{k * (k+1)}{2}$$

We subtract k on both sides of the equation to get:

$$
\begin{aligned}
1 + 2 + 3 + 4 \cdots (k-2) + (k-1) &= \frac{k * (k+1)}{2} \\
1 + 2 + 3 + 4 \cdots (k-2) + (k-1) &= \frac{k^2 + k - 2k}{2} \\
1 + 2 + 3 + 4 \cdots (k-2) + (k-1) &= \frac{k^2 - k}{2} \\
1 + 2 + 3 + 4 \cdots (k-2) + (k-1) &= \frac{k * (k-1)}{2}
\end{aligned}
$$

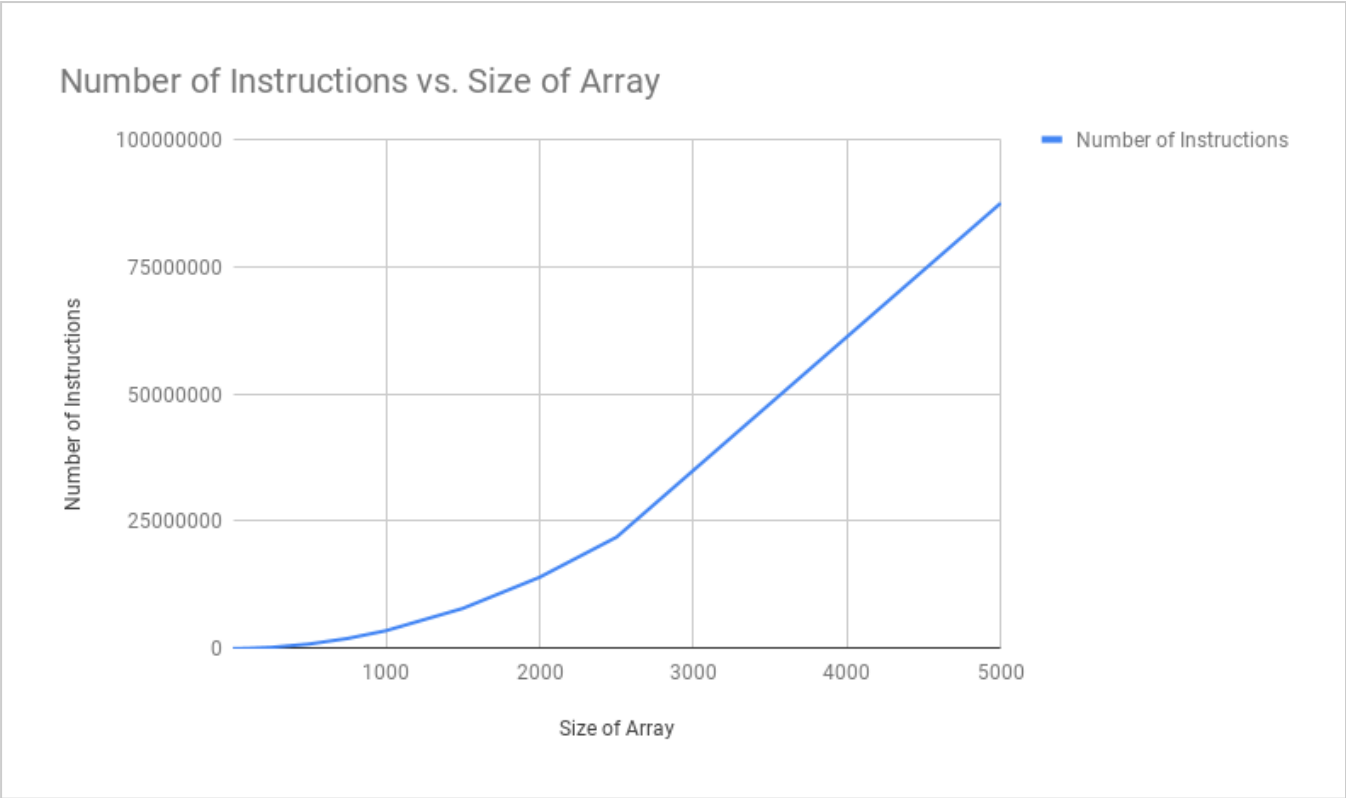
Coming across this summation is very common in algorithmic analysis and, without getting too technical, if you can identify this series, then you know how to apply a formula to sum it up.

Running Time as *n* Increases

As the size of the array increases, so does the number of instructions executed. Below is a table that shows the instructions executed for the worst

case as the size of the array increases.

Size of Array	Instructions Executed
5	102
10	377
25	2252
75	19877
100	35252
250	219377
500	876252
1000	3502502



Size of Array vs Number of Instructions Executed

The intention of doing the dry run and accounting for each line of code executed is to instill in the reader that the instruction count varies with the input size. Calculating down to this detail is often not necessary and with experience one can correctly recognize the complexity of various code snippets. As you'll learn in the next chapters, we reason about algorithm complexity in a ***ballpark*** sense, and calculating exact instruction counts is almost always unnecessary.

Problem Set 1

Practice problems to hone analysis skills.

The quiz questions below are related to the insertion sort algorithm discussed in the lesson.

1

Following is another implementation of insertion sort. If we feed an already sorted array to the following snippet will the algorithm execute a linear number of instructions? Insertion sort's best case running time is linear (think running a single loop) and not quadratic.

```
void sort(int[] input) {
    for (int i = 1; i < input.length; i++) {
        int key = input[i];
        for (int j = i - 1; j >= 0; j--) {
            if (input[j] > key) {
                int tmp = input[j];
                input[j] = key;
                input[j + 1] = tmp;
            }
        }
    }
}
```

2

We calculated the number of instructions required for the worst case of our algorithm. Can you determine the number of instructions executed for the best case when the array size is 5? The best case happens when the array is already sorted in ascending order. The code is reprinted below:

```
1.          for (int i = 0; i < input.length; i++) {
2.              int key = input[i];
3.              j = i - 1;
4.              while (j >= 0 && input[j] > key) {
5.                  if (input[j] > key) {
6.                      int tmp = input[j];
7.                      input[j] = key;
8.                      input[j + 1] = tmp;
9.                      j--;
10.                  }
11.             }
12.         }
```

3

Can you generalize and express the number of instructions executed for the best case as a function of the size of the array n?

Check Answers

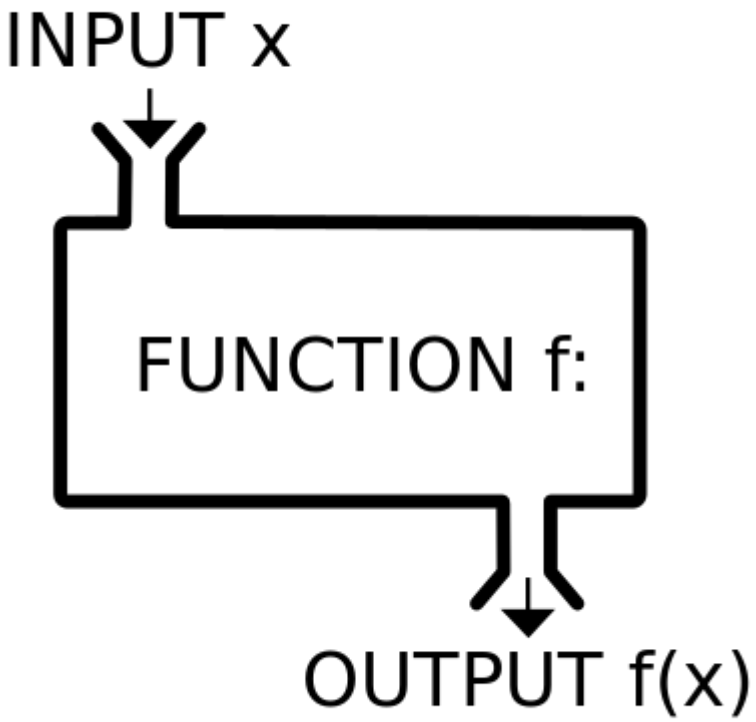
Functions and Their Growth

This lesson discusses the building blocks for analyzing algorithms.

The yardstick to measure the performance of algorithms is specified in terms of functions. For the mathematically uninitiated, we explain functions below.

Functions

Think of a function like a *machine* or a *blackbox* that takes inputs from one end and spits outputs from the other end.



Function is just a black-box that takes in an input and spits out a result denoted by f(x)

Let's consider the following function:

$$f(n) = n^2$$

This is the traditional format of expressing a function, with n as the input fed to the "machine". The output of the "machine" is expressed in terms of the input; in the above function, the output is $\boldsymbol{n^2}$ defined in terms of the input $\boldsymbol{n}$. The input n variable can take on different values and the output will vary accordingly. The below table captures the output values generated when **n** takes on different interger values.

$f(n)$	n^2
f (0)	0
f (1)	1
f (2)	4
f (3)	9
f (4)	16
f (5)	25

Comparing Functions

Now you know what a function is. Let's see how functions compare with each other. Consider the following two functions:

$$f(n) = n^2$$

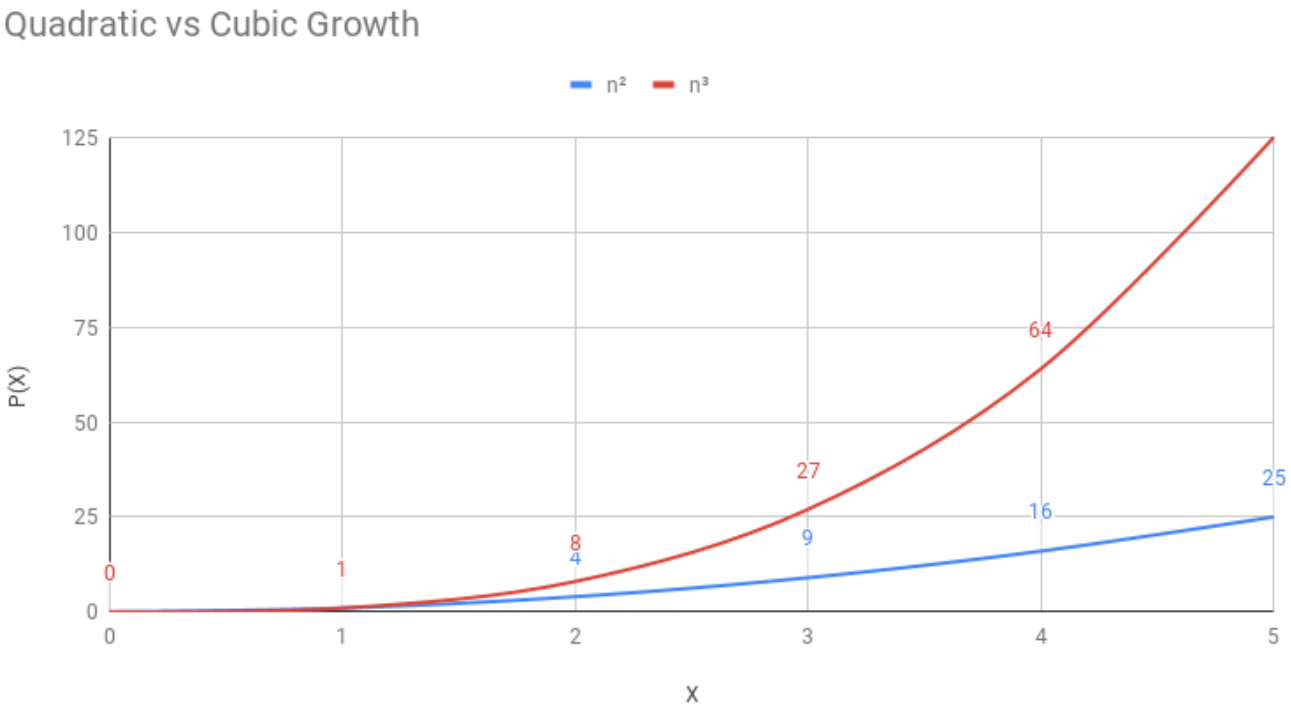
$$f(n) = n^3$$

If we restrict the values n can assume to only non-negative numbers, then we can determine which of the two functions will produce a numerically bigger output than the other. *The function which produces a bigger number will*

output than the other. ***The function which produces a bigger number will be said to grow faster than the other.***

$f(n)$	n^2	n^3
f (0)	0	0
f (1)	1	1
f (2)	4	8
f (3)	9	27
f (4)	16	64
f (5)	25	125

We can observe from the output that the function $f(n) = n^3$ grows faster than the other function $f(n) = n^2$ when n > 1.



Relating Back to Insertion Sort

In the previous section, we worked out an algebraic expression for the total number of instructions executed when sorting an input array of size n. That

expression is expressed as a function of n below:

$$f(n) = [2 * (n + 1) + 2n] + [2n + 7[\frac{n(n-1)}{2}]]$$

We have captured the time complexity of insertion sort as a function of the input size. In upcoming sections, we will now be able to wield developed mathematical tools to gain further insights.

Comparing functions may remind you of our race-track example at the beginning of the course. We are eventually headed in that direction, where we can model the performance of algorithms as functions, whose input n represents the size of the input to the algorithms. We will not be directly interested in the values the functions spit out. Rather, how they grow and the growth of functions is dependent on the values they produce.

Theta Notation

We formally introduce theta notation, which forms the basis of mathematical analysis of algorithms.

Example

Let's consider the following three functions:

$$f(n) = n^2 + 2$$

$$f(n) = 2n^2 - 1$$

$$f(n) = 2n^2 + 4$$

The table below shows how these functions grow as the value of n grows from 0 and onwards. Note that we aren't considering negative values for **n**, but we'll explain why shortly.

f (n)	$n^2 + 2$	$2n^2 - 1$	$2n^2 + 4$
f (0)	2	-1	4
f (1)	3	1	6
f (2)	6	7	12
f (3)	11	17	22
f (4)	18	31	36
f (5)	27	49	54

f (6)	38	71	76
f (7)	51	97	102
f (8)	66	127	132
f (9)	83	161	166
f (10)	102	199	204

We can observe from the output that the function ***f(n)*** = ***2n^2 - 1*** grows faster than the function ***f(n)*** = ***n^2 + 2***, but it grows slower than ***f(n)*** = ***2n^2 + 4*** once n becomes greater than or equal to 2. The astute reader would notice that the output of the function ***f(n)*** = ***2n^2*** is, in a sense, being **sandwiched** by the output of the other two functions when $n \geq 2$. All the yellow rows in the above table, show the values of the function ***f(n)*** = ***2n^2 - 1*** being sandwiched by the output of the other two functions.

Formal Definition

> Whenever we can bound the output of a function ***f(n)*** by the outputs of two other functions in the following form, we say that the function ***f(n)*** ***belongs to the set Θ(g(n))*** (pronounced as theta of g(n))
>
> $$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$
>
> $$where\ n > n_0\ and\ c_1\ and\ c_2\ are\ positive$$

Let's apply the formal definition to the example functions we just discussed above.

> $$n^2 + 2 < 2n^2 - 1 < 2n^2 + 4$$
>
> $$1(n^2 + 2) < 2n^2 - 1 < 2(n^2 + 2)$$
>
> $$where\ n \geq 2\ and\ c_1 = 1\ c_2 = 2\ and\ g(n) = n^2 + 2$$

We can choose positive constants so that our function $2n^2 - 1$ becomes sandwiched by the other two functions as soon as n becomes greater than 2. Hence we can say that the function $\mathbf{2n^2 - 1}$ is Θ of g(n) where g(n) equals $\mathbf{n^2 + 2}$.

Explanation

There are several points to ponder about the above relationship.

- Note that to satisfy the formal definition, none of the three functions can produce negative values after n has crossed a certain threshhold marked by n_0. In our example, f(n) produces a negative value when n=1. However, once n≥2 all subsequent values produced by all three functions are nonnegative. Consider the following function.

  $$f(n) = (-1)^n \, n^2$$

  The above function will not produce nonnegative values (i.e. 0 or positive numbers) for any threshold that n crosses, as it will oscillate between a negative and a positive number for odd and even values of n respectively. We require g(n) to be *asymptotically nonnegative*, a mathematically-fancy term meaning that as n grows and passes a certain value n_0, it should always produce nonnegative numbers as output. This restriction is partly because it makes it easier to reason about the complexity of algorithms, as input is either none (i.e. 0 or some positive number).

- Once our function $f\ (\ n\ )$ satisfies the inequality (i.e. we are able to sandwich it) we can claim that,

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

  Note the use of equality instead of membership. Even though Θ g (n) is a set and f (n) belongs to the set g (n), we take the liberty to use the

equality instead of membership when undertaking algorithmic analysis.

- We say that g(n) is a *asymtotically tight bound* for f(n), which is just a mathematical way of saying that **the growth of f(n), as n grows larger and larger, mirrors that of g(n). Or f(n) is equal to g(n) within a constant factor when the values of n are past the threshold n_0.**
- Note that in our example we used constants $c_1 = 1$ and $c_2 = 2$, but we could use any other set of constants that satisfy the inequality.

Quiz

Q Can you determine the Θ(g(n)) for the polynomial $5n^3 + n$?

Check Answers

Θ(1)

Let's say we have a function that always outputs the same constant value for any value of n.

$$f(n) = 7$$

The above function will always produce 7, no matter what the value of n is. Let's try to see if we can find a Θ(g(n)) for our constant function.

$$c_1 n^0 < f(n) < c_2 n^0$$

$$c_1 n^0 < 7 < c_2 n^0$$

$$c_1 < f(n) < c_2$$

We can now choose c_1= 2 or c_2= 8 and claim that our function *f(n)* belongs to the set $\Theta(n^0)$, which is equivalent to *Θ(1)* as we know n^0=1. This is considered slight abuse of the notation, as the expression *Θ(1)* doesn't indicate which variable is tending to infinity.

Θ(1) is used to denote either a constant or a constant function with respect to some variable.

Big O and Big Omega Notations

Discusses the Big O notation with examples

In the previous section, we defined Θ notation. In this section, we'll discuss big O and big Ω notations. The reader would notice the use of the adjective "big" with these notations and rightly suspect that there exist complementary small o and small ω notations, which we'll discuss in the next lesson.

Big O

Big O is expressed as *O(g(n))* and is pronounced as "big oh of g of n". We observed that Θ provides both an asymptotically upper and lower tight bound. Big O only provides an asymptotic upper bound which may not necessarily be a tight bound.

Big O is the most commonly talked-about notation when discussing algorithms in industry settings or in interviews. You won't see Θ or other notations being discussed, and for good reason too. Generally, if we are guaranteed that an algorithm will perform no worse than a certain threshold, and that threshold is acceptable for the problem at hand, then knowing its best-case performance or finding a very tight upper bound may not be productive.

Formal Definition

> Similar to Θ we define O(g(n)) as a set of functions and a function f(n) can be a member of this set if it satisfies the following conditions:
>
> $$0 \leq f(n) \leq cg(n)$$
>
> *where c is a positive constant and the inequality holds after n crosses a positive threshold* n_0

Explanation

- if $f(n) = O(g(n))$ then it also implies that $f(n) = O(g(n))$

- If f(n) = Θ (g(n)) then it also implies that f(n) = O (g(n))

- The notation means that the function f(n) is bounded by above to within a constant factor of g(n).
- Note that the inequality requires f(n) to be greater than 0 after n crosses n_0.
- The set O(g(n)) may also contain other functions also that satisfy the inequality. There could be several values of *c* that can be used to satisfy the inequality.

Insertion Sort

Previously, we came up with the following f(n) for the insertion sort algorithm

$$f(n) = [2 * (n + 1) + 2n] + [2n + 7[\frac{n(n-1)}{2}]]$$

Now we can attempt to find the O(g(n)) for this expression. As a general rule of thumb, when working with big O we can drop the lower order terms and concentrate only on the higher order terms. The term $^{n(n-1)}/_{2}$ is the highest order term, since it involves a square of n. The above expression is thus quadratic, and we can ignore all the constants and linear terms. Thus a tight bound on the expression would be $O(n^2)$. We can prove it below:

$$\frac{n(n-1)}{2} \leq cn^2$$

$$let\ c = 10\ and\ n_0\ = 10$$

You can always pick a different set of constants as long as it satisfies the inequality for all $n > n_0$.

As a general rule of thumb, for an expression consisting of polynomials the expression/function is *O(the highest polynomial degree in the expression)*.

Big Ω

The complementary notation for big O is the *big omega* notation. The big omega notation provides an asymptotic lower bound and is expressed as Ω(g(n)).

Formal Definition

> As before, Ω(g(n)) is a set of functions and any function f(n) that satisfies the below constraints belongs to this set and f(n) is said to be big omega of g(n).
>
> $$0 \leq g(n) \leq f(n)$$
>
> $where\ c\ is\ a\ positive\ constant\ and\ the\ above\ inequality\ holds\ afte r\ n\ crosses\ a\ positive\ threshold\ n_0$

Explanation

- Notice that similar to big O, the definition mandates we only consider positive values for f(n).
- Big Ω is not necessarily a tight lower bound.
- If f(n) Θ(g(n)) then it also implies f(n) Ω(g(n)).

Relation to Θ

One can see that if a function **f(n)** is **Θ(g(n))** then it follows that **f(n)** must be **O(g(n))** and **Ω(g(n))**. In the previous lesson we proved that ***f(n)*** = ***2n^2-1*** was Θ((n^2)). We can simply consider the right and left handsides of the following inequality in isolation to prove that f(n) is also O(g(n)) and Ω(g(n)).

> $$1(n^2 + 2) < 2n^2 - 1 < 2(n^2 + 2)$$

Quiz

1

You are told that a function f(n) is $O(n^3)$, does it also imply that the f(n) is $\Theta(n^3)$?

2

You are told that a function f(n) is O(n) and also $\Omega(n)$, does it also imply that the f(n) is $\Theta(n)$?

Check Answers

Small omega and Small o Notations

In this lesson, we discuss notations which imply loose bounds.

The small o and small ω are complementary notations to the big O and big Ω notations. For algorithm analysis, the most important notation is the big O. For the sake of completeness, we mention the small o and small ω notations too.

Small o

The small o is **not** an asymptotically tight upper bound. The formal definition is similar to big O, with one important difference. A function f(n) belongs to the set o(g(n)), if the following condition is satisfied.

> $$0 \leq f(n) < cg(n)$$
>
> $for\ any\ positive\ constant\ c,\ there\ exists\ some\ constant\ n_0\ which\ if\ n\ surpasses\ ,\ the\ above\ inequality\ holds$

Note that the inequality should not hold for *some* positive constant c, as is the case for big O; rather, it should hold for all positive constants.

Unlike big O, which may or may not be tight, the small o notation is necessarily not tight.

Small ω

Small ω is similarly **not** a tight lower bound. Big Ω, on the contrary, may or may not be a tight bound. A function f(n) belongs to the set ω(g(n)) if the following inequality holds:

> $$0 \leq g(n) < cf(n)$$

For **any** positive constant c, there exists some constant n_0 which if n surpasses, the above inequality holds. The above inequality should hold for **all constants**. Note that in the case of big omega, the inequality was required to hold for *some constant*.

Explanation

We can see that the small case notations are, in a sense, *relaxed* compared to their upper case notations.

- $2n^2$ is equal to $\omega(n)$ but $2n^2 \neq \omega(n^2)$. To understand the first claim consider the inequality:

  $$cn \leq 2n^2$$

  Let's pick a really big number for c=1000. To make the inequality hold I can set n_0=c=1000, and because of n being squared the right side will be bigger for every value of n greater than n_0.

  Now, to understand why $2n^2 \neq \omega(n^2)$, consider the inequality:

  $$cn^2 < 2n^2$$

  If I pick c = 3 then - no matter what value of n_0 I choose - I can never satisfy the above inequality.

- For small o, $2n^2$ is $o(n^3)$ but $2n^2 \neq o(2n^2)$

1 If f(n) is $\Theta(n^3)$, is f(n) also $o(n^3)$ and $\omega(n^3)$?

2

If f(n) is $O(n^3)$, is f(n) also $o(n^3)$?

Check Answers

Problem Set 2

Practice problems relating to analysis notations.

Question 1

Suppose your friend discovers a new algorithm and in his excitement tells you that his algorithm has a lower bound of $O(n^2)$. Can you explain why your friend's statement makes no sense?

Question 2

Does $O(2^{2n})$ equal $O(2^n)$?

Question 3

Give an example of an algorithm whose best case is equal to its worst case?

Question 4

Work out the time complexity for the algorithm given below:

```
void averager(int[] A) {

    float avg = 0.0f;
    int j, count;

    for (j = 0; j < A.length; j++) {
        avg += A[j];
    }

    avg = avg / A.length;

    count = j = 0;

    do {

        while (j < A.length && A[j] != avg) {
            j++;
        }
```

```
            if (j < A.length) {
                A[j++] = 0;
                count++;
            }
        } while (j < A.length);
    }
```

Question 5

Q

What is the complexity of the below snippet

```
for( int i=0; i<array.length; i++){
  for(int j=0; j<10000; j++)
  {
    // some useful work done here.
  }
}
```

COMPLETED 0%

1 of 1

Question 6

Consider the following snippet of code and determine its running time complexity?

```
    void complexMethod(int[] array) {
        int n = array.length;
```

```
    int runFor = Math.pow(-1, n) * Math.pow(n, 2);
    for (int i = 0; i < runFor; i++) {
        System.out.println("Find how complex I am ?")
    }
}
```

Question 7

Determine the time complexity for the following snippet of code

```
void complexMethod(int n, int m) {

   for (int j = 0; j < n; j++) {
      for (int i = 0; i < m % n; i++) {
          System.out.println("")
      }
   }
}
```

For non-java folks, m % n notation means m modulus n.

Question 8

Determine the time complexity for the following snippet of code

```
void someMethod(int n) {

   for (int j = 0; j < n; j++) {
      for (int i = 0; i < 3; i++) {
         for (int i = 0; i < n; i++) {
             System.out.println("I have 3 loops");
         }
      }
   }
}
```

Question 9

Determine the time complexity for the following snippet of code

```
void someMethod(int n, int m) {

   for (int i = 0; i < n; i++) {
```

```
        for (int i = 0; i < m; i++) {
            System.out.println("I have 2 loops");
        }
    }
}
```

Solution Set 2

Solutions to problem set 2.

Solution 1

Big O notation denotes an upper bound but is silent about the lower bound. Treat it like a cap on the worst that can happen. So when someone says that an algorithm has a lower bound of $O(n^2)$ that translates into saying the lower bound can at worst be quadratic in complexity. By definition, the lower bound is the best an algorithm can perform. Combine the two statements and your friend is saying the algorithm he found in the best case can perform in quadratic time or better. It can also perform in linear time or constant time, we just don't know. So your friend is effectively not telling you anything about the lower bound on the performance of his new found algorithm.

Solution 2

We can employ the Big O definition we learnt earlier to determine if $O(2^{2n})$ and $O(2^n)$ are equivalent. Intuitively, it would feel 2^{2n} will yield a greater number than 2^n for values of $n > 0$. According to Big O definition:

$$0 \leq f(n) \leq cg(n)$$

For $O(2^{2n})$ and $O(2^n)$ to be equivalent, we need to prove that the following inequalities would hold:

$$0 \leq 2^{2n} \leq c2^n$$

$$0 \leq 2^n \leq c2^{2n}$$

Let's consider the first inequality

$$0 \leq 2^{2n} \leq c2^n$$

Divide both sides by 2^n will give

$$0 \leq 2^n \leq c$$

It's easy to see that once we fix the constant *c*, we can vary the *n* to make the inequality false. Therefore, $O(2^{2n})$ and $O(2^n)$ are not equivalent.

Solution 3

Counting Sort and ***Radix Sort*** are two algorithms which have the same best, worst and average case complexities.

Solution 4

Below is the code-snippet for which we want to find the complexity

```
1.     void averager(int[] A) {
2.
3.         float avg = 0.0f;
4.         int j, count;
5.
6.         for (j = 0; j < A.length; j++) {
7.             avg += A[j];
8.         }
9.
10.        avg = avg / A.length;
11.
12.        count = j = 0;
13.
14.        do {
15.
16.            while (j < A.length && A[j] != avg) {
17.                j++;
18.            }
19.
20.            if (j < A.length) {
```

```
21.                 A[j++] = 0;
22.                 count++;

23.             }
24.         } while (j < A.length);
25.     }
```

The for loop from **lines 6-8** calculates the average of the contents of the array and is O(n) in complexity. The `do-while` loop from **lines 14-24** is tricky. A novice may wrongly conclude the complexity of the snippet to be $O(n^2)$ by glancing at the nested while loop within the outer do-while loop.

There are two possibilities, one the average calculated for the given input array also occurs in the array and second the average is a float and doesn't appear in the array.

If the average is a float then the nested while loop on **lines 16-17** will increment the value of the variable *j* to the size of the input array and the do-while condition will also become false. The complexity in this case will be O(n)

If the average does appear in the array, and say in the worst case the array consists of all the same numbers then the nested while loop of **lines 16-17** will not run and the if clause of **lines 20-23** will kick-in and increment j to the size of the array as the outer do-while loop iterates.

Hence the overall complexity of the snippet is O(n).

Solution 6

To the untrained eye, it may appear that since there's a single loop, the runtime complexity would likely be O(n), which is incorrect. If you look at the snippet again:

```
1.     void complexMethod(int[] array) {
2.         int n = array.length;
3.         int runFor = Math.pow(-1, n) * Math.pow(n, 2);
4.         for (int i = 0; i < runFor; i++) {
5.             System.out.println("Find how complex I am ?")
6.         }
7.     }
```

the worst case happens for even sizes of the input array. The loop doesn't run when the size of the array is an odd number. Next, the loop runs a quadratic

number of times the length of the array when the length is even. Ask yourself,

does a bigger array result in the loop running for a longer period of time? Yes, it does. The bigger the array size and provided it is an even number, the single loop runs as if it were a nesting of two loops. Therefore the complexity of this snippet of code is $O(n^2)$;

Solution 7

Let's examine the snippet again:

```
void complexMethod(int n, int m) {

   for (int j = 0; j < n; j++) {
      for (int i = 0; i < m % n; i++) {
          System.out.println("I am complex")
      }
   }
}
```

- Let's start with the case when n equals m. In that case, **m%n** will equal 0 and the inner loop will not run. So complexity will be **O(n)**.
- If **n** > **m** then m%n will equal **m**, so now the inner loop will run for m times, giving us a total complexity of **O(m*n)**.
- The last case is when **n** < **m** then m%n will equal values ranging from **1 to n-1**. So the inner loop in the worst case would run for **n-1** times. The complexity in the last case would then be O(n*(n-1)) which is **$O(n^2)$**.

Note that **O(m*n)** is a tighter bound for the second case, but since we are talking in big O terms, we can say for the second case, when **n** > **m**, then **O(m*n)** < **$O(n^2)$** so for the second case, we can say the complexity will be **$O(n^2)$**.

So in the worst case, the complexity would be **$O(n^2)$**.

Solution 8

It may seem that the complexity of the snippet is cubic since we have 3 loops. But the second loop runs for a constant number of times.

```
void someMethod(int n, int m) {

   for (int j = 0; j < n; j++) {
      for (int i = 0; i < 3; i++) {
         for (int i = 0; i < n; i++) {
             System.out.println("I have 3 loops");
         }
      }
   }
}
```

One way to think about the above snippet is to unroll the second loop. We can say the above code is equivalent to the following

```
void someMethod(int n) {

   for (int j = 0; j < n; j++) {
      for (int i = 0; i < n; i++) {
          System.out.println("I have 3 loops");
      }
      for (int i = 0; i < n; i++) {
          System.out.println("I have 3 loops");
      }
      for (int i = 0; i < n; i++) {
          System.out.println("I have 3 loops");
      }
   }
}
```

The output of both the snippets would be the same. From the unrolling, it is evident that the three inner loops contribute $n + n + n = 3n = O(n)$ and the outmost loop runs for n too therefore the overall complexity is $O(n^2)$.

Solution 9

```
void someMethod(int n, int m) {

   for (int j = 0; j < n; j++) {
      for (int i = 0; i < m; i++) {
          System.out.println("I have 2 loops");
      }
   }
```

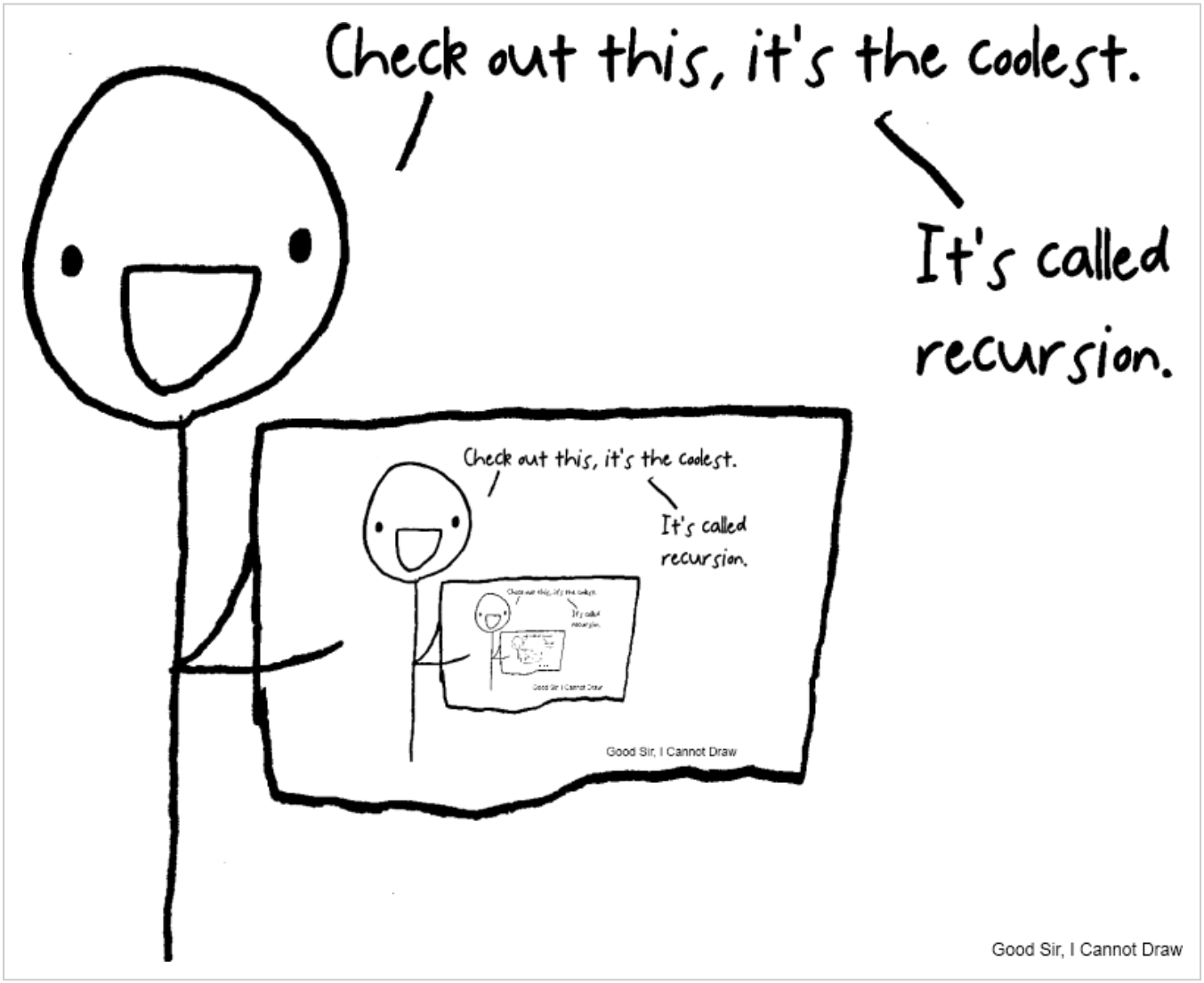
```
    }
}
```

The above snippet is a traditional example of nested loops. Whenever you get nested loops, each running for a variable number of times, the complexity of the entire snippet is the product of the variables controlling the repetition of each loop. The string message would be printed for a total of ***m*n*** times and thus the overall, complexity will be ***O(m*n)*** = ***O(mn)***.

Recurrence

This chapter introduces recursive algorithms and how they can be analyzed.

The word *recurrence* literally means **the fact of occurring again**.



Merge Sort

Merge sort is a typical text-book example of a recursive algorithm. The idea is very simple: We divide the array into two equal parts, sort them recursively, and then combine the two sorted arrays. The base case for recursion occurs when the size of the array reaches a single element. An array consisting of a single element is already sorted.

We'll determine the time complexity of merge sort by unrolling the recursive

calls made by the algorithm to itself, and examining the cost at each level of recursion. The total cost is the sum of individual costs at each level.

The running time for a recursive solution is expressed as a *recurrence equation* (an equation or inequality that describes a function in terms of its own value on smaller inputs). The running time for a recursive algorithm is the solution to the recurrence equation. The recurrence equation for recursive algorithms usually takes on the following form:

Running Time = Cost to divide into n subproblems + n * Cost to solve each of the n problems + Cost to merge all n problems

In the case of merge sort, we divide the given array into two arrays of equal size, i.e. we divide the original problem into sub-problems to be solved recursively.

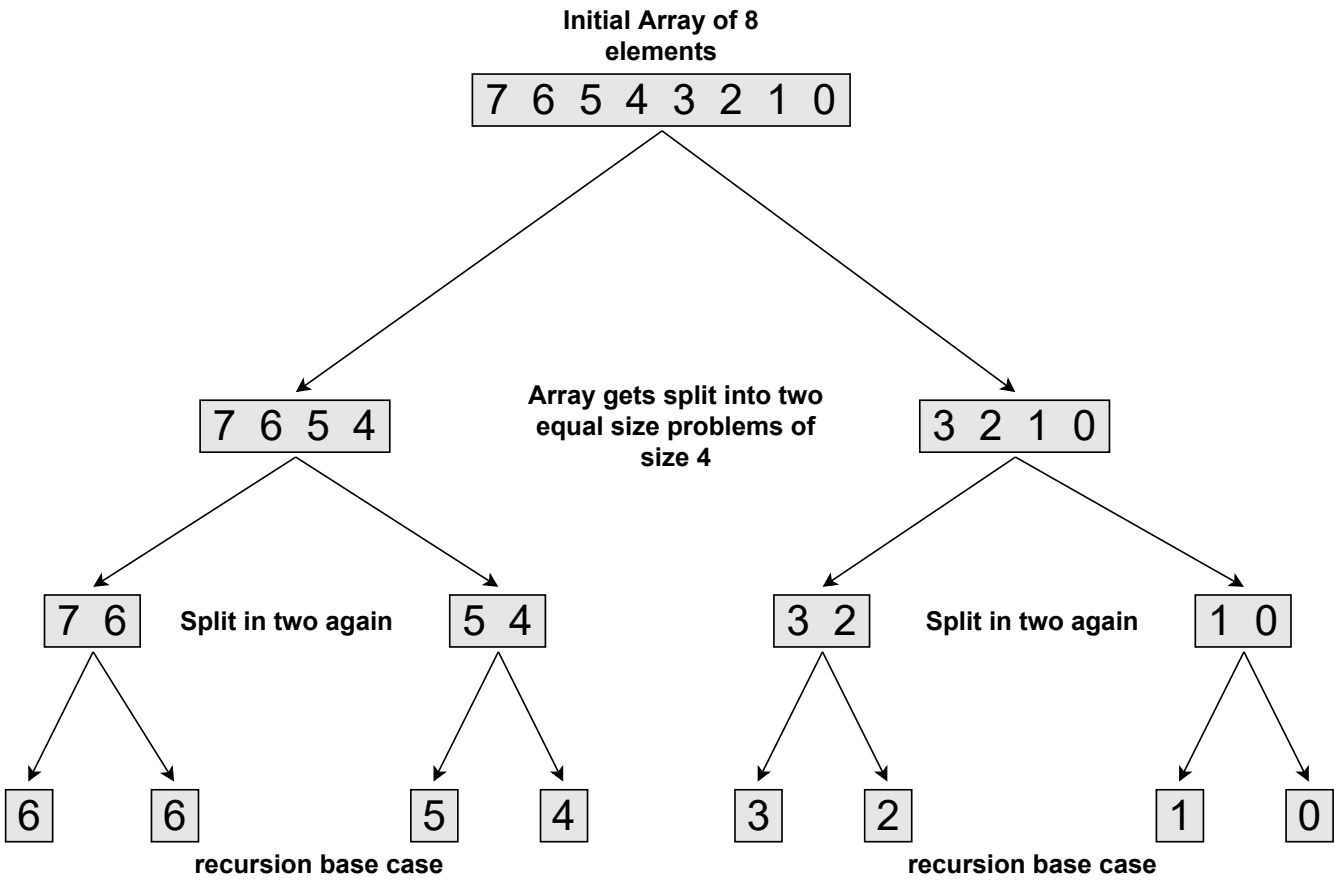
Following is the recurrence equation for merge sort.

Running Time = Cost to divide into 2 unsorted arrays + 2 * Cost to sort half the original array + Cost to merge 2 sorted arrays

> $$T(n) = Cost\ to\ divide\ into\ 2\ unsorted\ arrays + 2 * T(\frac{n}{2}) + Cost\ to\ merge\ 2\ sorted\ arrays when n > 1$$
>
> $$T(n) = O(1)\ \ when\ n = 1$$

Remember the *solution* to the recurrence equation will be the *running time* of the algorithm on an input of size n.

Merge Sort Recursion Tree

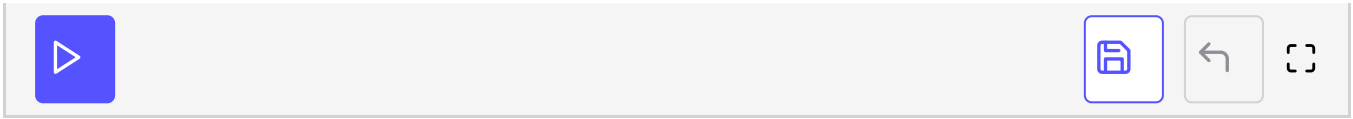


Working of Merge Sort

Merge Sort Implementation

```
class Demonstration {

  private static int[] scratch = new int[10];

  public static void main( String args[] ) {
    int[] input = new int[]{ 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
    printArray(input,"Before: ");
    mergeSort(0, input.length-1, input);
    printArray(input,"After:  ");

  }

  private static void mergeSort(int start, int end, int[] input) {

    if (start == end) {
      return;
    }

    int mid = (start + end) / 2;

    // sort first half
    mergeSort(start, mid, input);

    // sort second half
    mergeSort(mid + 1, end, input);

    // merge the two sorted arrays
```

```
    int i = start;
    int j = mid + 1;
    int k;


    for (k = start; k <= end; k++) {
      scratch[k] = input[k];
    }

    k = start;
    while (k <= end) {

      if (i <= mid && j <= end) {
        input[k] = Math.min(scratch[i], scratch[j]);

        if (input[k] == scratch[i]) {
          i++;
        } else {
          j++;
        }
      } else if (i <= mid && j > end) {
        input[k] = scratch[i];
        i++;
      } else {
        input[k] = scratch[j];
        j++;
      }
      k++;
    }
  }

  private static void printArray(int[] input, String msg) {
    System.out.println();
    System.out.print(msg + " ");
    for (int i = 0; i < input.length; i++)
      System.out.print(" " + input[i] + " ");
    System.out.println();
  }


}
```



Cost to divide

Line-19 is the cost to divide the original problem into two sub-problems, with each one of them as half the size of the original array. The mathematical operation takes place in constant time, and we can say that the cost to divide the problem into 2 subproblems is constant. Let's say it is denoted by ***d***.

Cost to Merge

Line-27 to 55 is the cost to merge two already-sorted arrays. Note the use of a scratch array here. Merge Sort can be implemented in-place but doing so isn't trivial. For now, we'll stick to using an additional array. You'll notice that the

trivial. For now, we'll stick to using an additional array. You'll notice that the merge part has two loops:

1- **for loop line 32 to 34**
2- **while loop line 37 to 55**

Note that both the loops iterate for ***end-start+1***, which is the size of the sub-problem the algorithm is currently working on. The cost of merging two arrays varies with the input size, as follows:

Size of each array = n/2, Cost to merge = 2 * $n/2$ = n
Size of each array = n/4, Cost to merge = 2 * $n/4$ = $n/2$
Size of each array = n/8, Cost to merge = 2 * $n/8$ = $n/4$

•

•

Size = 1, Cost = 1 - base case for recursion

Cost to solve 2 subproblems of half the size

The cost to solve each of the two sub-problems (of half the size of the original array) isn't so straightforward, as each of them is expressed as a recursive call to merge sort.

Let's start with the base case: when the problem size becomes a single element, the 1-element array is already sorted and the recursion bottoms out. The cost of solving the base case is constant time, for simplicity, let's assume it's one.

Recurrence Part II

This chapter continues the discussion on analyzing the complexity of recursive algorithms.

Determining Recursion Levels

Now let's take an example of how we get to the base case, starting with an array of size 8. We'll first divide it into half, creating two arrays of size 4. Next, we'll split each of the two arrays of size 4 into two arrays of size 2. Now we have four arrays of size 2 but we haven't hit the recursion's base case, so we'll split each of the four arrays of size 2 into a single array of size 1. When we finally hit the recursion's base case, we will have eight arrays of a single element.

Note, we need to find the number of levels or recursions it took to break the problem of size 8 into a problem of size 1. At each recursion level, we divide the problem by 2. So we may ask, *how many times do we need to divide the input size by 2 to get to a problem of size 1?*. For 8 we know we need to divide thrice by 2 to get to a problem size of 1. In other words, we can ask, *2 raised to what power would yield 8?*

$$2^x = 8$$

The above equation is in exponential form and can be expressed in *logarithmic* form as follows:

$$log_2(8) = x$$

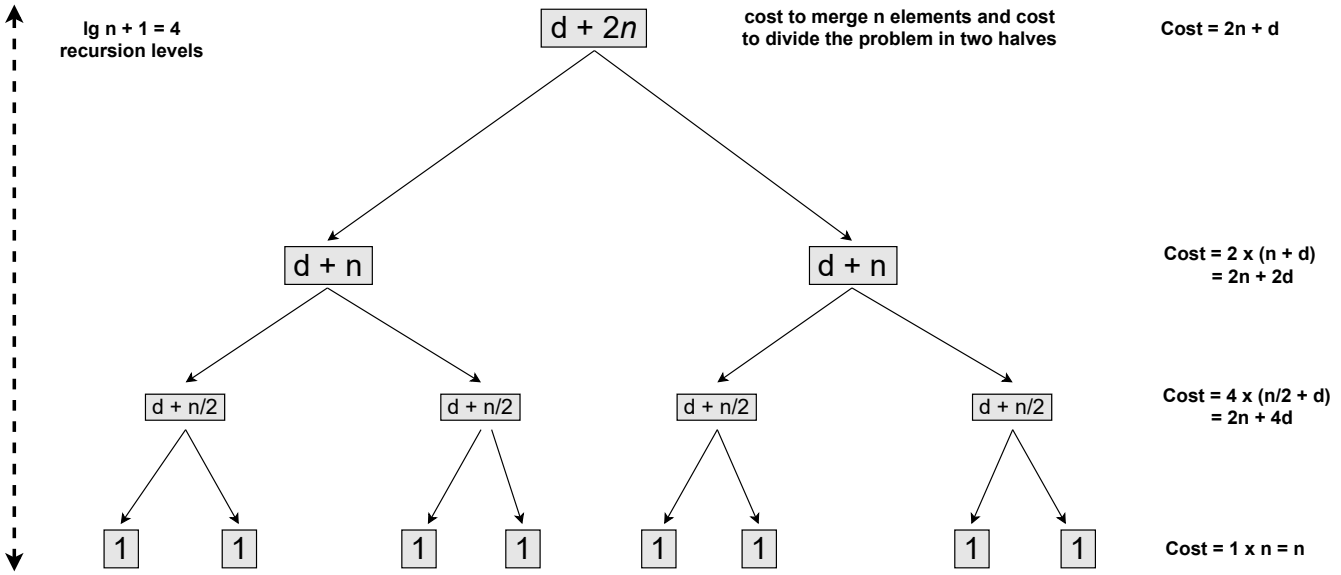
$$log_2(8) = 3$$

Note in our case, the number of recursion levels is $\mathbf{log_2 n + 1}$. We need to add 1 for the root level.

Knowing the number of recursion levels a given input results in, we can

determine the running time of the algorithm by multiplying the number of recursion levels with the cost incurred at each level.

The illustration below shows the recursion levels and the total cost at each level, where ***n*** is the input size and ***d*** is the cost to divide the problem into two subproblems.



Analyzing Cost of Merge Sort

At the base case, the cost to merge a single element array is constant. But we have 8 such base cases, one for each element of the array. If we use n as the initial input size we can see that the cost of the base case is n x O(1) = n. Now, as we return from the base case recursion, we can calculate the cost at the level immediately above the base case. Here, we'll be merging 2 arrays of 1 element in each of the four calls. Expressed in terms of the initial input size, we merge n/4 elements in each call for a total of 4 calls. The cost is thus 4 x n/4 = n. Similarly, at the root level above, we'll be merging n/2 elements in 2 calls for a total cost of 2 x n/2 = n. Note, the cost at each level is O(n) for merging the results from the below levels. If we know how many levels exist, we can simply multiply the cost with the levels to get the total cost for running Merge Sort.

Tying it back

So far we have determined that the runtime of Merge Sort is the solution to the following recurrence equation

> $$T(n) = Cost\ to\ divide\ into\ 2\ subproblems + 2 * T(\frac{n}{2}) + Cost\ to\ merge\ 2\ subproblems$$

e.g. 2 subproblems

$$T(n) = d + 2T(\frac{n}{2}) + 2n$$

Additionally, we have determined the cost at each recursion level to be **d + 2n**. Since the number of recursion levels is **log n + 1**, the solution to the recurrence equation is thus:

$$T(n) = (logn + 1) * (d + 2n)$$

$$T(n) = dlogn + d + 2nlogn + 2n$$

$$T(n) = 2nlogn + 2n + dlogn + d$$

Dropping the lower order terms, we are left with

$$T(n) = O(nlogn)$$

Note that we have made some simplifications.

- For one, the cost to divide the problem into two subproblems is not always *d*. For instance, at the second level it is *2d* and at the third level it is *4d* but we just used *d* in our analysis. The divide step would occur at the second last recursion level for a total of n/2 times. In that case, the total cost will be d * $^{n}/_{2}$ - also expressed as *dn* - which is still Θ(n) and doesn't change the big O for the algorithm.
- Our algorithm for merging the two sorted arrays is simplified for explanation-purposes, and can be optimized, but it still won't change the big O for algorithm.

Space Complexity

In the sample code provided, we used an extra scratch array equal to the size of the input. As pointed out earlier, Merge Sort can be implemented in-place. However, it is error-prone and much harder to implement. The space complexity is thus O(n) for our given implementation.

Binary Search - Recursive Implementation

This chapter details the reasoning behind binary search's time complexity

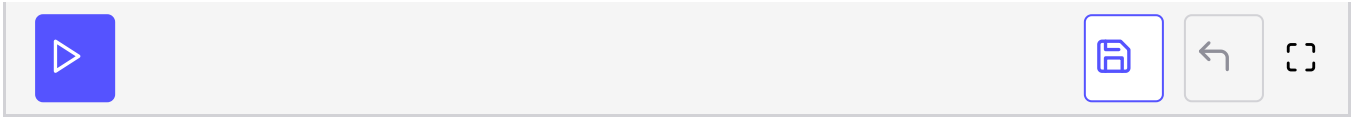
Binary search is a well-known search algorithm that works on an already-sorted input. The basic idea is to successively eliminate half of the remaining search space at each step till either the target is found or there's no more search space left.

Binary search is recursive in nature but can also be implemented iteratively. Below is the recursive implementation of binary search.

Binary Search Implementation

```
class Demonstration {
    public static void main( String args[] ) {
        int[] input = new int[] { 1, 3, 4, 6, 7, 101, 1009 };
        System.out.println(binarySearch(0, input.length - 1, 1009, input));
        System.out.println(binarySearch(0, input.length - 1, -1009, input));
        System.out.println(binarySearch(0, input.length - 1, 5, input));
        System.out.println(binarySearch(0, input.length - 1, 6, input));
    }

    /**
     * start and end are inclusive indices
     *
     * @param start
     * @param end
     * @param target
     * @param input
     * @return
     */
    public static boolean binarySearch(int start, int end, int target, int[] input) {

        if (start > end) {
            return false;
        }

        int mid = (start + end) / 2;

        if (input[mid] == target) {
            return true;
        } else if (input[mid] > target) {
            return binarySearch(start, mid - 1, target, input);
        } else {
            return binarySearch(mid + 1, end, target, input);
        }
```

```
        }
    }


}
```



Binary Search Recursive

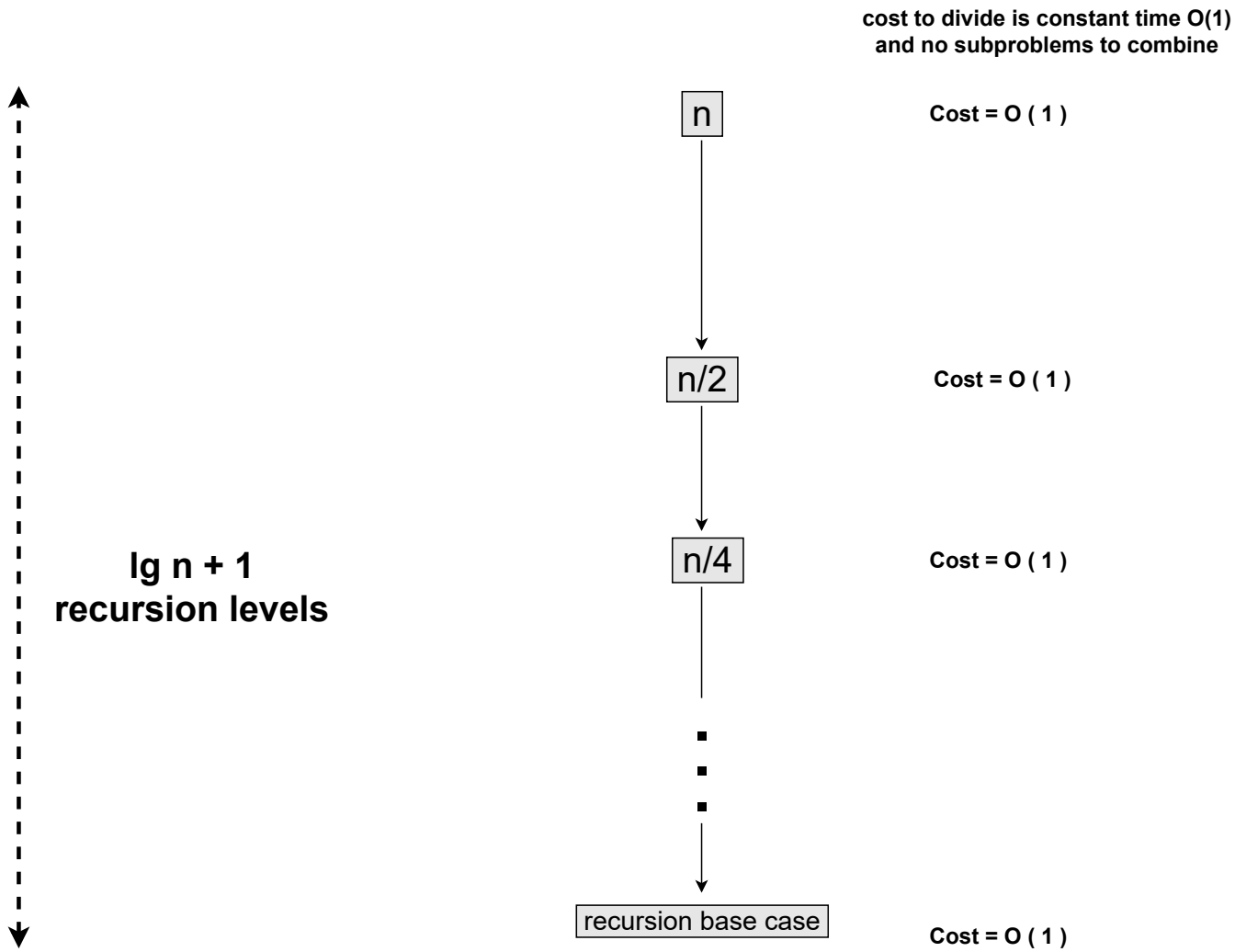
Recurrence Equation

The recurrence equation for binary search is given below:

> $$Running\ Time = Cost\ to\ divide\ into\ p\ subproblems + p * Cost\ to\ solve\ each\ of\ the\ p\ problems + Cost\ to\ merge\ all\ p\ problems$$
>
> $$T(n) = O(1) + 2 * T\frac{n}{2} + O(1)$$
>
> $$T(n) = T(\frac{n}{2})$$

The running time of binary search is the solution to the above recurrence. Note that we divide the original problem into 2 subproblems. Division is just an arithmetic operation that takes constant time and has no relation to the size of the input. We don't have to combine the solution of the subproblems, as we eliminate one half of the search space at each recursion.

The recursion tree appears below.



Analyzing Cost of Merge Sort

Note that the cost at each recursion level is constant, as all we are doing at each level is a bunch of arithmetic operations, and there are no subproblems to combine. The running time in the worst case occurs when we are unable to find the target value in the sorted array. The running time is then simply the number of recursion levels multiplied by the recursion cost at each level.

$$T(n) = Cost\ at\ each\ recursion\ level * total\ recursion\ levels$$

We already know that the cost at each recursion level is constant or O (1). We need to determine how many recursion levels exist in the worst case. This is akin to asking, "how many times do we divide the input array of size *n* by 2 to get 1?" or, "*2 raised to what power would yield n?*". Assume that the array size is a power of 2 for simplicity.

This brings us back to logarithms. As explained in the previous lesson, the

This brings us back to logarithms. As explained in the previous lesson, the number of recursion levels for an array of size n when successively halved will equal:

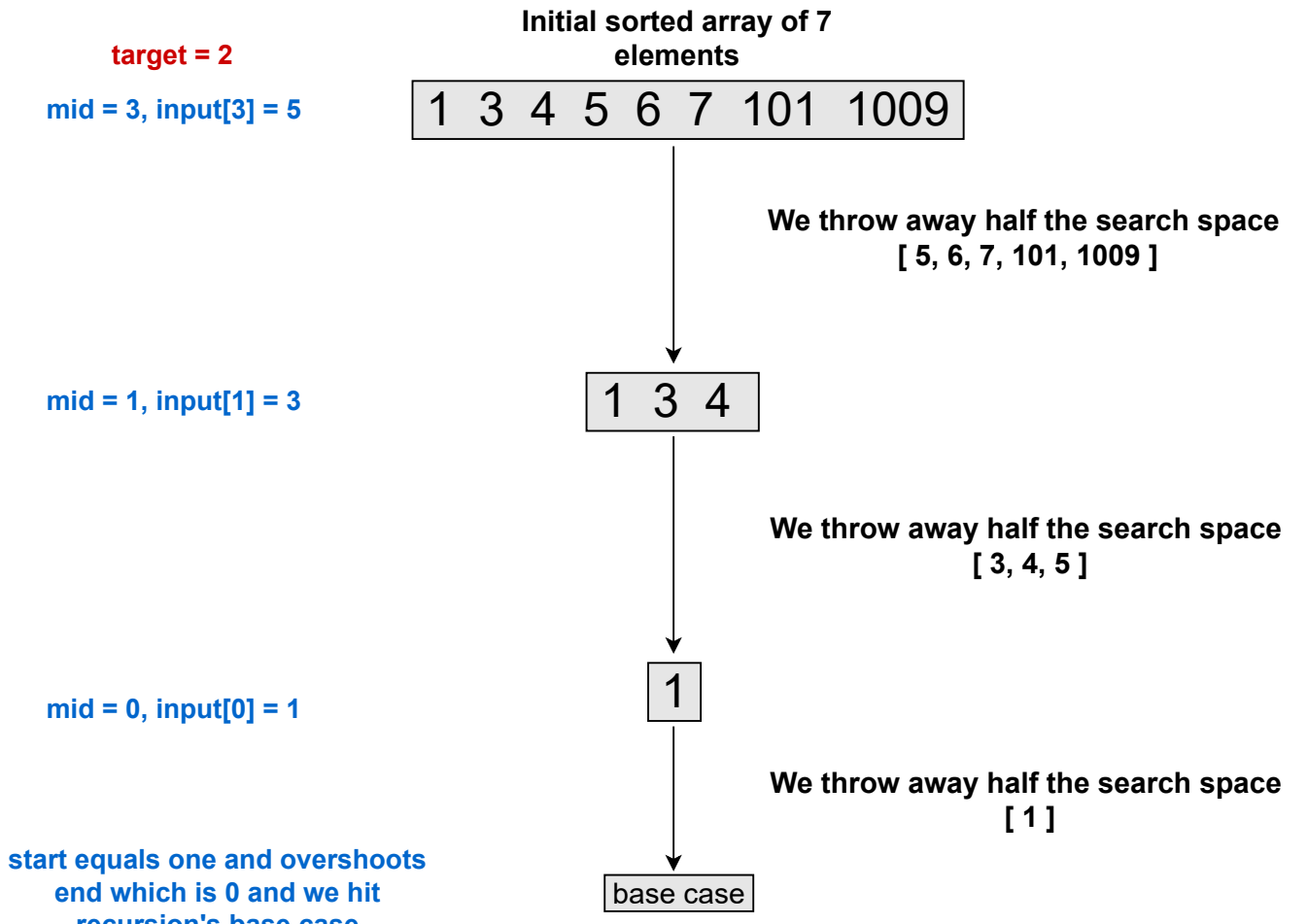
$$log_2 = lgn$$

Note that the way we have implemented our algorithm, we recurse one more level after all the search space is exhausted to detect that there's nothing left to search anymore. The total number of recursion steps would then be **lgn + 1**

The running time becomes:

$$T(n) = O(1) * (lgn + 1)$$

Below is a pictorial representation of how binary search would proceed using the given algorithm.



recursion's base case

Working of Binary Search

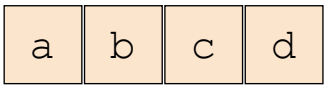
Recursion Levels' Nuances

The astute reader would notice that even though in the above diagram we claim to be throwing away half the search space, we are actually throwing a little more than that. We also discard the mid value that we just tested against. This strategy results in lgn+1 recursion steps. However, if we throw away exactly half of the search space and unnecessarily keep the mid-index in the search space for the next recursion, the number of recursion steps in the worst case can be lgn+2. That being said, the big O for the algorithm will be unaffected.

Permutations

This chapter shows how one can reason about recursive problems without extensive mathematical knowledge

Finding permutations of a given character or integer array is a common interview problem. Assume we have the following character array of size 4 with no repeating characters, and we need to find out all the permutations of size 4 for the below array.



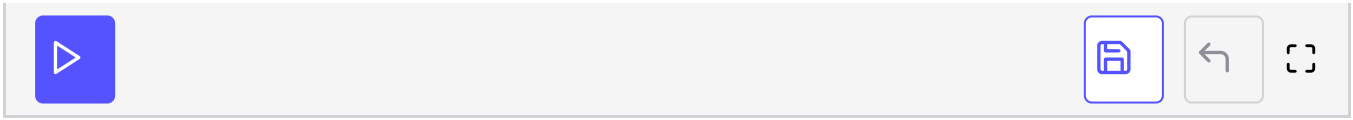
Character Array to Permute

The permutations will be generated using recursion and we'll give a solution that is easy to understand, even though it may not be the optimal one in terms of space usage. Our emphasis is on learning to reason about the complexity of algorithms. Take a minute to read through the code listing below before we attempt to figure out the time complexity.

Permutation Implementation

```
class Demonstration {
    public static void main( String args[] ) {
        char[] array = new char[] { 'a', 'b', 'c', 'd' };
        char[] perm = new char[array.length];
        boolean[] used = new boolean[256];
        permute(array, perm, 0, used);
    }

    /**
     * Note that we are using a boolean array to remember what
     * character we have already used and another array perm
     * which contains the actual permutation. We can actually
     * generate permutations without these additional arrays but
     * for now the desire is to make the code easier to understand.
     */
    static void permute(char[] array, char[] perm, int i, boolean[] used) {

        // base case
        if (i == perm.length) {
            System.out.println(perm);
```

```
            return;
        }

        for (int j = 0; j < array.length; j++) {

            if (!used[array[j] - 'a']) {
                perm[i] = array[j];
                used[array[j] - 'a'] = true;
                permute(array, perm, i + 1, used);
                used[array[j] - 'a'] = false;
            }
        }
    }
}
```



Permutations of a String

The number of permutations for a string of size n is n! (pronounced as n factorial). A factorial of a number is the product of all the numbers starting from 1 up to n. Below are a few examples:

0! = 1
3! = 3 x 2 x 1 = 6
5! = 5 x 4 x 3 x 2 x 1 = 120
10! = 10 x 9 x 8 .... 1 = 3628800

The number of ways to permute a string of size n is ***n!***. One way to rationalize is to think of n empty slots.

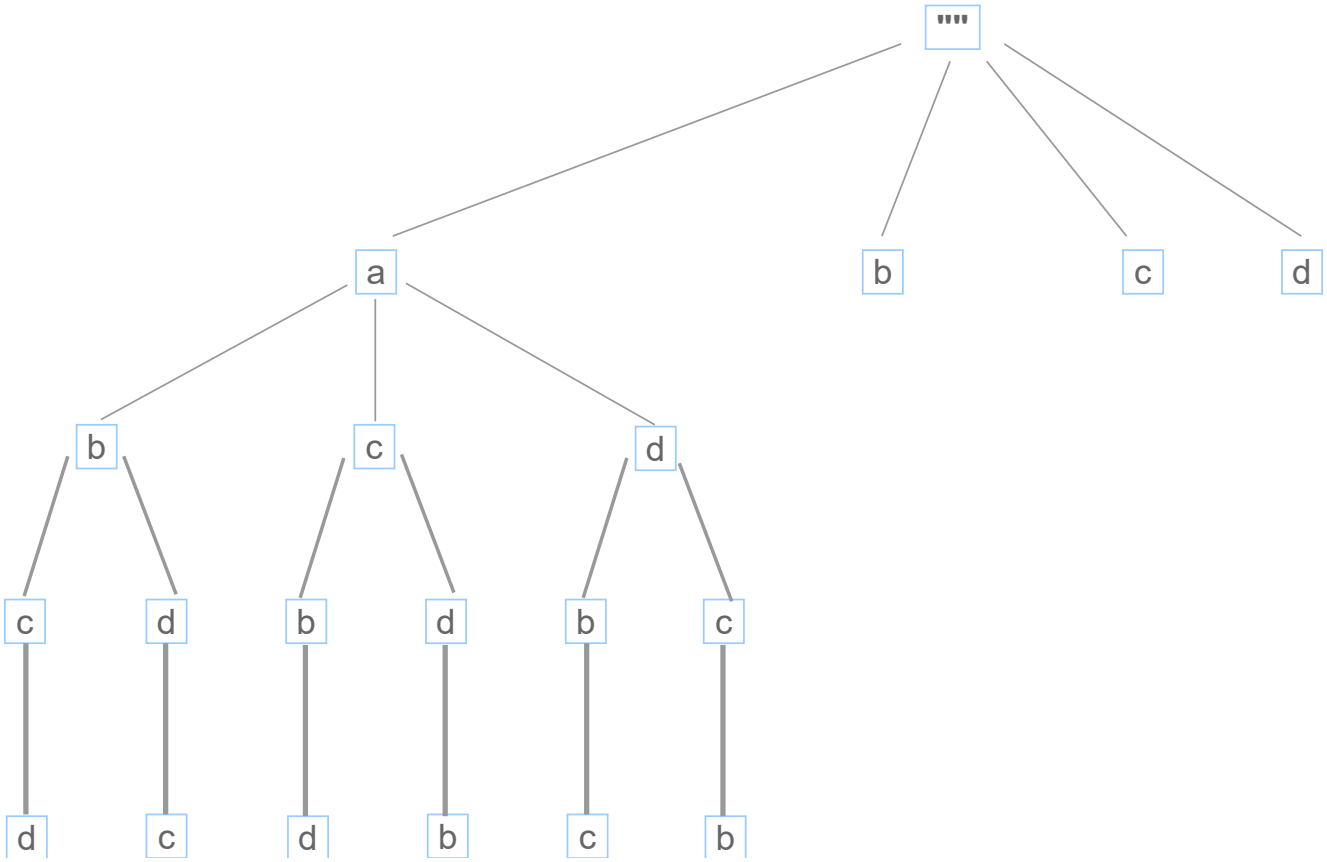
We can pick any of the n characters and put it in the first slot. Once the first slot if fixed, we can fill the second slot in any of the n-1 ways using the remaining n-1 characters of the string. In the same vein of reasoning, we can fill the third slot in n-2 ways, and so on and so forth. By the time we get to the nth slot, there will only be one element left and we will only have one way to fill it up. Thus the total number of ways we can fill up n empty slots (i.e. arrange n elements in a different order or by the number of permutations) is:

$$n * (n-1) * (n-2)...1$$

$$= n!$$

We can immediately see that the time complexity for generating all the permutations is also O(n!). Note that even though this is a recursive problem, it doesn't lend itself to the divide-and-conquer strategy; it can't be broken up into smaller subproblems that can then be combined. The solution is to find every possible permutation, and that would take n! time.

Having said that, the intent of this chapter is to reason about complexity using recursion trees, so let us draw out the recursion tree and see how one can deduce the complexity from the tree.



Recursion Tree

The above diagram shows the complete recursion tree when the first slot is filled with the character 'a'. The subtrees for b, c and d would be similar when chosen as first characters. Each node of the tree represents a recursive call to the permute function itself. The root of the tree is an imaginary empty string to ease the thought process. Once we analyze the complexity of the subtree rooted at "a", we can simply multiply it with n to get the final complexity. This stems from the fact that there will be *n* such subtrees rooted with each of the n characters in the string.

If we can find the total recursive calls made and the cost incurred in each

If we can find the total recursive calls made and the cost incurred in each invocation, we can multiply the two to get the total cost for the subtree rooted at "a".

Cost of each *permute* invocation

The permute method in the given code consists of the base case, where we detect that a permutation is complete and output it. The rest of the code caters to the regular case, where we made a recursive call but still have not formed a permutation. In the base case we are outputting the array, which is an O(n) operation behind the scenes since every character needs to be traversed in order to be output to the console.

In the regular case, note that the loop runs for the entire length of the array, though it may skip those characters that have already been used. The cost incurred is still O(n) as the loop runs the length of the array. In summary, we can say that the cost of each invocation of the *permute* method will be O(n).

Number of recursive calls

Once we fix "a" as the root, then there are only 3 choices left for the second slot. Therefore, the fan-out from the root element is 3. Once the second slot is filled, we are left with only 2 elements, and thus the fan-out from the nodes at the second level of the tree (rooted at a) is 2. On filling the third slot, there's only one element left and the fan-out is only one. We can generalize that the fan-out would decrease in the following order:

$$n$$

$$n-1$$

$$n-2$$

$$.$$

$$.$$

$$1$$

Note that the fan-out from our imaginary empty string root element is n, which is the number of ways we can fill up the first slot.

Now to determine the total number of invocations, which are represented by

the nodes in the recursion tree, we need to find the nodes at each level of the tree and sum them up. At any given level we'll multiply the fan-out for that level with the number of nodes that appear in the previous level.

Level	Fanout	Fanout * Nodes in Previous Level
1	n	n * 1
2	n - 1	(n-1) * n
3	n - 2	(n-2) * (n-1) * n
4	n - 3	(n-3) * (n-2) * (n-1) * n
.	.	.
.	.	.
nth	1	1 * 2 ... (n-3) * (n-2) * (n-1) * n

If we sum up the third column of the above table we'll get the total number of nodes or invocations for the permute method. However, the summation of all the terms in the third column will require mathematical gymnastics on our end. As software developers, we need to find a ballpark or asymptotic behavior rather than the exact expression representing the time complexity for a given algorithm. If you look carefully, at the nth level, the number of nodes is exactly n! and for each of the previous levels the number of nodes will necessarily be less than n!. There are a total of n levels, excluding the root level which we only introduced for instructional purposes. We can then say that the total number of nodes is capped by:

$$Total\ number\ of\ nodes = n * n!$$

This insight will help us determine the big O for the algorithm.

Tying it together

Now we know the total invocations of the permute method and the cost of each invocation. The running time of the algorithm is given as

$$T(n) = Recursive\ calls\ made * Cost\ of\ each\ call$$

$$T(n) = (n * n!) * n$$

$$T(n) = n^2 * n!$$

To further simplify, we can reason that n^2 is less than the product of (n+1)* (n+2)

$$T(n) = n^2 * n! < (n+2) * (n+1) * n!$$

$$T(n) = n^2 * n! < (n+2)!$$

$$T(n) = (n+2)!$$

Dropping constants

$$T(n) = O(n!)$$

1 If we tweaked the permutation problem to produce all the permutations of a string of size 1 to n, then what would be the complexity of the resulting algorithm?

2

What would be the time complexity of generating combinations of size 2 for a character array of size n?

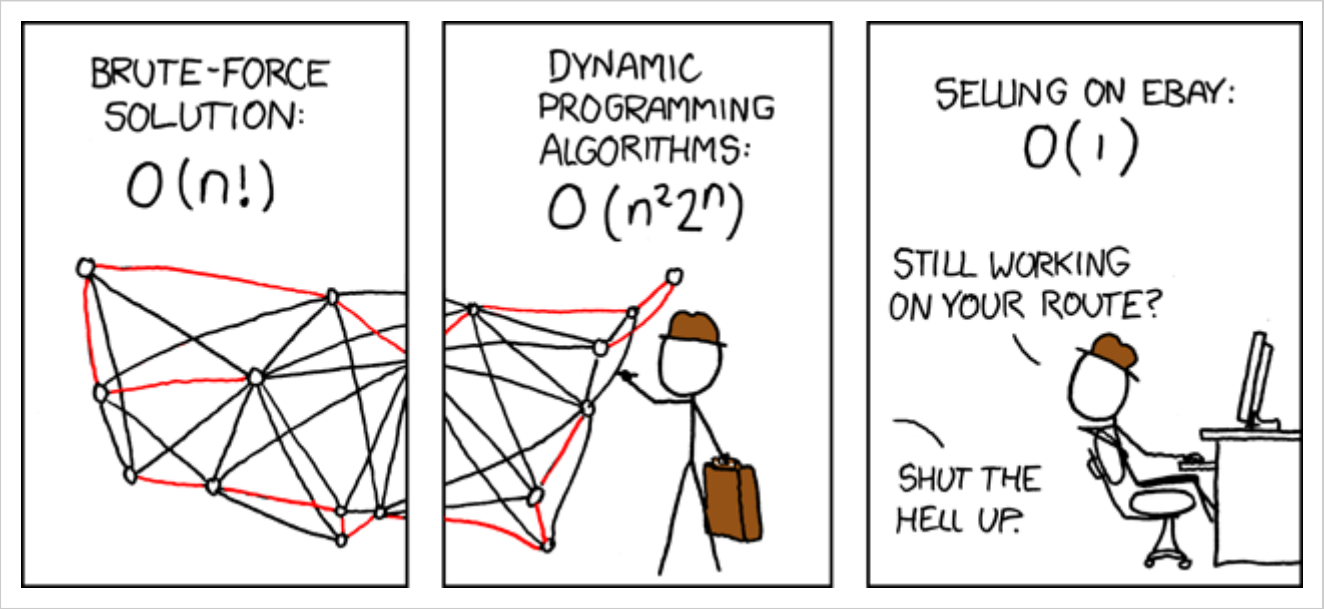
3

Without writing code, can you guess what would be the time complexity of generating combinations of all sizes for a character array of size n?

Check Answers

Dynamic Programming

This chapter works on a sample dynamic programming problem to show how complexity for this class of problems can be worked out.



Dynamic programming problems are similar to the divide-and-conquer problems with one important difference: The subproblems in divide-and-conquer are distinct and disjoint, whereas in the case of dynamic programming, the subproblems may overlap with one another. Also, dynamic programming problems can be solved in a bottom-up fashion instead of just a top-down approach.

Computing Fibonacci numbers is a textbook example of a dynamic programming problem. Furthermore, dynamic programming problems are usually optimization problems, and there may be several optimal solutions. However, the *value* of the optimal solution will be same.

The word "programming" in dynamic programming doesn't mean coding in the literal sense; in fact, it was used to mean a *tabular* solution.

Since dynamic programming problems involve subproblems that are overlapping or common, the answer to these subproblems can be stored in a table and reused when needed. This approach is called memoization and

avoids the need to recompute the answer to subproblems each time they are encountered.

The scope of this text is limited to teaching evaluation of time and space complexities for various algorithms categories. Therefore, we'll restrict ourselves to one simple DP problem example and slice and dice it various ways to understand how complexity can be worked out for DP class of algorithms.

String Sum Representation

You are provided a positive integer *n* and asked to construct all strings of 1s, 2s, and 3s that would sum up to n. For example,

if n = 3, then the following strings will sum up to n:

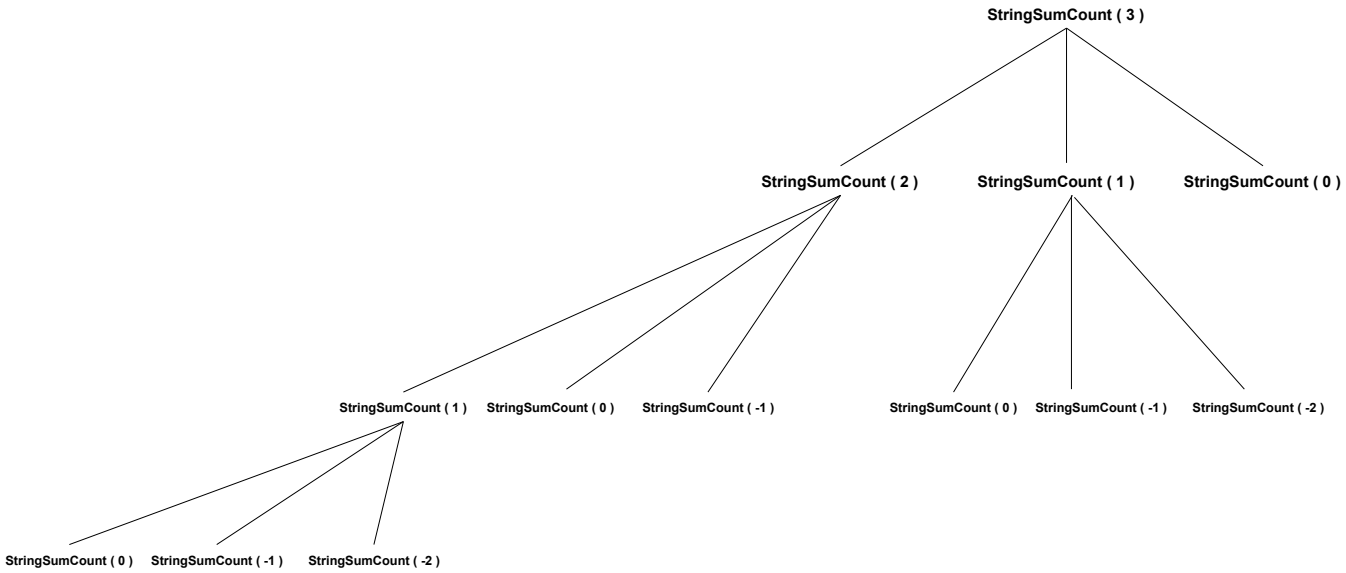
- 111
- 12
- 21
- 3

Solution

Think of 1s, 2s, and 3s as building blocks to construct n. Let's say we pick 1 as the first block. If we know how many strings can represent the integer *(n-1)*, we can deduce that the same number of strings can also represent n, as prefixing 1 to the strings making up the integer *n-1* would make up n. This leads to a recursive solution, where we ask ourselves how many strings can sum up to integers *n-1*, *n-2*, and *n-3*? We can prefix those strings with 1, 2, or 3 respectively to sum up to *n*.

We also need to define our base case. If we keep making recursive calls by subtracting 1, 2, and 3 from *n*, we'll eventually hit 0 or a negative number. A negative number implies that the string can't sum up to n and we return 0. However, if we hit 0, we know that the string we are currently considering neatly sums up to *n* and is one of the solutions. We return 1 to account for this solution.

Below is the recursion tree for the algorithm.



Recursion Tree

The recursive code for the problem appears below. From drawing out the recursion tree, one can observe that the same problems are being solved repeatedly. For instance the *StringSumCount(2)* is solved twice. As mentioned earlier, dynamic programming problems exhibit overlapping subproblems.

String Sum Implementation

```
class Demonstration {
    public static void main( String args[] ) {
        long start = System.currentTimeMillis();
        int result = StringSumCount(5);
        long end = System.currentTimeMillis();
        System.out.println("Number of Strings : " + result + "    Time taken = " + (end - sta
    }

    static int StringSumCount(int n) {

        if (n == 0) {
            return 1;
        }

        if (n < 1) {
            return 0;
        }

        int sum = 0;

        // Number of ways to represent with 1
        sum += StringSumCount(n - 1);

        // Number of ways to represent with 2
        sum += StringSumCount(n - 2);

        // Number of ways to represent with31
```

```
        sum += stringSumCount(n - 3);

        return sum;


    }
}
```

The above code is recursive and inefficient, but before we improve upon it we'll figure out the time complexity using the recursion tree method. Let's start with the tree height or the number of levels the recursion tree can possibly have. The longest string that can sum up to a given *n* will consist of exactly *n* 1s. The longest path that we'll traverse from the root to a leaf would consist of all 1s, which implies the number of levels would be n+1. The work done in each recursive call is constant since we are only executing statements that take constant time. However, a newbie mistake would be to think that the time complexity would be the product of (n+1)*O(1). Even though the work done in each recursive call is constant, the number of recursive calls are exponentially increasing at each level!

The right way to think about complexity in this scenario is to think about the number of nodes that'll be generated in the recursion tree. Each node represents a recursive call with a constant execution time; therefore, the complexity can be expressed as:

$$T(n) = Number\ of\ Nodes * O(1)$$

Calculating an upper bound on the number of nodes in the recursion tree is easy. We simply work out the number of nodes for a full tree with every parent having exactly three children. At level 0 there is a single root, at level 1 there are 3 nodes, at level 2 there are 9 nodes, so and so forth. At the *n+1*th level the number of nodes will be 3^n. The sum of all the nodes will then be,

$$Numberofnodes = 3^0 + 3^1 + 3^0 ... 3^n$$

At this point, it is clear that the algorithm we devised is exponential in complexity. Even if you don't know how to mathematically transform the above summation into a neat looking formula, you can conclude two facts: One, the number of nodes will be at least 3^n or $\Omega(3^n)$; Two, the number of nodes must be less than 3^{n+1} or $O(3^{n+1})$. The intuition for the second fact comes from experience with binary trees, where a tree with n levels has 2^n-1 nodes. We can extend the same concept and quickly work out a few verified examples to see that a n level 3-ary tree must have less than 3^{n+1} nodes. In summary, we can claim that the time complexity of our recursive algorithm will be $O(3^{n+1})$ or $O(3^n)$.

Top Down and Bottom Up Approaches

This chapter continues the discussion on the complexity of dynamic programming problems.

Solution

In the previous section, we discussed a recursive solution with exponential complexity. Now we'll look at two ways to solve the same problem, first using the DP top-down approach and then the DP bottom-up approach.

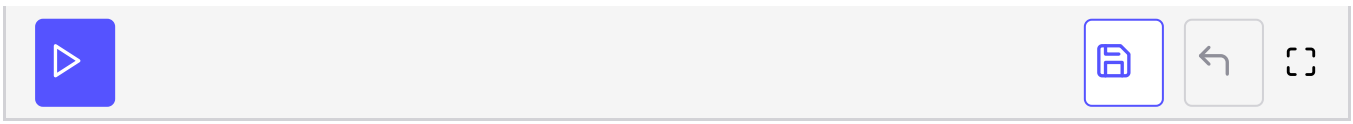
We saw in the recursive solution that subproblems are solved over and over again. We can store the solutions of the subproblems and avoid calculating them repeatedly. This is called *memoization*, where the algorithm remembers solutions to previously computed problems.

The only change required to make our recursive algorithm polynomial is to start storing the solutions to the subproblems; the necessary code appears below.

Top Down Implementation

```
class Demonstration {

    static int[] computed;

    public static void main( String args[] ) {
        long start = System.currentTimeMillis();

        int n = 50;
        computed = new int[n + 1];
        for (int i = 1; i < n + 1; i++)
            computed[i] = -1;

        System.out.println(StringSumCount(n));
        long end = System.currentTimeMillis();
        System.out.println("Time taken = " + (end - start));
    }

    static int StringSumCount(int n) {

        if (n == 0) {
            return 1;
        }
```

```
        if (n < 1) {
            return 0;
        }

        // If solution is already calculated
        if (computed[n] != -1)
            return computed[n];

        int sum = 0;

        // Number of ways to represent with 1
        sum += StringSumCount(n - 1);

        // Number of ways to represent with 2
        sum += StringSumCount(n - 2);

        // Number of ways to represent with31
        sum += StringSumCount(n - 3);

        computed[n] = sum;
        return sum;
    }
}
```



Top Down Approach

Note that we can easily compute the value for n=50, but it will timeout if we try the same value for the recursive solution. Now let's try to reason about the time complexity for our modified algorithm, which is recursive but no longer exponential. The key insight is to realize that we compute each subproblem once. If we can find the number of problems actually computed, we'll get a handle on the complexity. We know that we'll solve the maximum number of subproblems when we generate the longest string consisting of all 1s. The subproblems will include:

StringSumCount(0), StringSumCount(1), StringSumCount(2), ... StringSumCount(n)

Above is the complete list of problems that need to be solved, and it is easy to see that the number of problems will be exactly *n+1* (i.e from 0 to *n*). The complexity will be O(n+1) or O(n). Note how we determined the complexity without undertaking any extensive mathematical analysis!

Also, take stock that storing the solution to subproblems requires extra space. Since there are n+1 problems, we'll need to store n+1 solutions requiring O(n) space.

space.

Bottom Up Approach

We solved the problem using a recursive solution, then a top-down DP solution and finally we'll work out a DP bottom-up solution. In the bottom-up approach, we'll work on the subproblems. From the solution of the subproblems, we'll construct the solution to the next subproblem higher up in the hierarchy. Let H(n) define the number of strings that can sum up to n. The base case appears below:

H (0) = 0

Now we express the solution for H(1) as follows:

H (1) = if 1 - 1 >= 0 then H(1 - 1) +
if 1 - 2 >= 0 then H(1 - 2) +
if 1 - 3 >= 0 then H(1 - 3)

Or more generally,

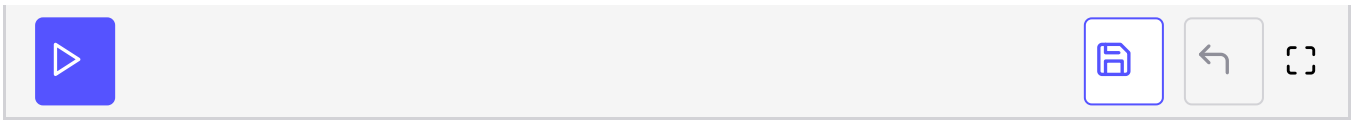
H (n) = if n - 1 >= 0 then H(n - 1) +
if n - 2 >= 0 then H(n - 2) +
if n - 3 >= 0 then H(n - 3)

The intuition for this recurrence is the same as before. We can tack on a 1 as the prefix to all the strings whose characters sum up to *n-1* to get strings that'll sum up to *n*. Similarly, we can prefix strings with a '2' or a '3'. The implementation can be simplified by initializing the solution array for n = 0, 1, 2, and 3. This helps us avoid putting the if conditions.

Bottom Up Implementation

```
class Demonstration {
    public static void main( String args[] ) {
        long start = System.currentTimeMillis();
        int n = 50;
        System.out.println(StringSumCount(n));
        long end = System.currentTimeMillis();
        System.out.println("Time taken = " + (end - start));
    }

    static int StringSumCount(int n) {

        int[] computed = new int[n + 1];
        computed[0] = 0;
        computed[1] = 1;
        computed[2] = 2;
```

```
        computed[3] = 4;

        for (int i = 4; i < n + 1; i++) {
            computed[i] = computed[i - 1] + computed[i - 2] + computed[i - 3];
        }

        return computed[n];
    }

}
```



Bottom-Up Iterative Solution

It is trivial to deduce that the complexity for the iterative bottom-up approach is O(n) which is the length for which the *for* loop runs for.

Comparision

To better appreciate the performance improvements we get by converting an exponential algorithm into a polynomial one, we ran all of the 3 algorithms on different values of n and measured the time it took for the algorithm to complete. This is by no means a definitive or comprehensive test, but it let us crudely measure the performance of the different algorithms. The times are in milliseconds.

n	Recursive	Top Down	Bottom Up
5	0	0	0
25	87	0	0
30	484	0	0
35	7971	0	0
40	182736	0	0

The recursive solution performs very poorly compared to the other two dynamic programming solutions. This should help you appreciate how

dynamic programming solutions. This should help you appreciate how

different implementations using different algorithm strategies can drastically affect running times.

Problem Set 3

Questions to understand recursive complexity analysis

Question 1

In the previous lesson, we implemented merge sort where we divided the array into two parts at each recursion step. Imagine you are asked to implement merge sort by dividing the problem into three parts instead of two. You can assume for simplicity that the input size will always be a multiple of 3. Use a priority queue to merge sub-arrays in the combine step.

a- Provide implementation for the 3-way division merge sort.

b- Provide a generalized expression for the number of recursion levels.

c- Work out the time complexity for the 3-way merge sort.

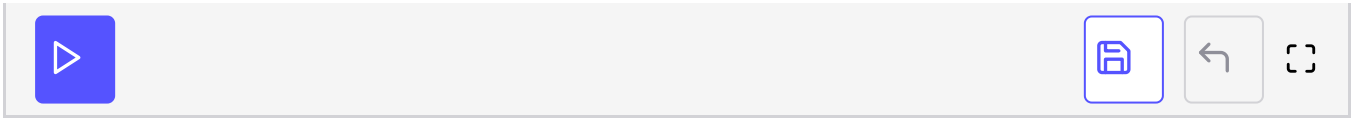
d- Will implementing merge sort by dividing the problem into a greater number of subproblems improve execution time?

Question 2

We implemented unoptimized code for generating permutations of a string. The solution used extra space and also ran the main loop for the entire length of the array on each invocation. Given the following optimized solution, can you work out the time complexity?

```
class Demonstration {
    public static void main( String args[] ) {
        char[] array = new char[] { 'a', 'b', 'c', 'd' };
        permutate(array, 0);
    }

    private static void swap(char[] str, int i, int j) {
        char temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }

    static void permutate(char[] str, int index) {
```

```
        // base case
        if (index == str.length) {
            System.out.println(str);
            return;
        }

        // regular case
        for (int i = index; i < str.length; i++) {
            swap(str, index, i);
            permutate(str, index + 1);
            swap(str, index, i);
        }
    }
}
```



Question 3

We are asked to implement a *n-dimensional* integer list. The constructor will take in the parameter *n* and the class will expose two methods `get(int n)` and `put(int n)`. If n equals 3, then we'll first create a list of 3 element, then initialize each element of first list with another list of 3 elements and finally initialize each element of the second list with a list of 3 elements. The get and put functions act on the elements of the innermost list.

As an example, with n=3, we'll get a total of 27 elements. The get and put methods should be able to specify index from 0 to 26.

a- Can you code the class?

b- What is the time complexity of initializing the list?

c- What is the time complexity of the get and put methods?

d- What is the space complexity?

Solution Set 3

Solutions to problem set 3

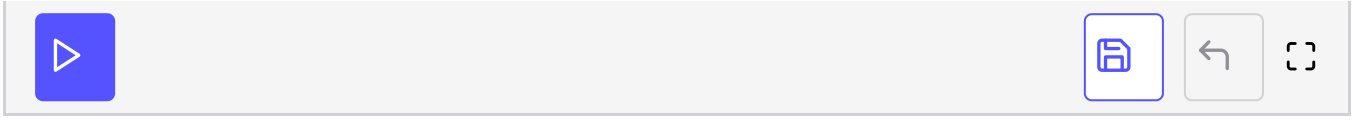
Solution 1

a-

We can implement merge sort by dividing the initial input array into 3 subproblems instead of 2. The only caution we need to exercise is to carefully pass the boundaries for the three parts in the subsequent recursive portions. Combining the three arrays is equivalent of solving the problem of merging n number of sorted arrays, using a priority queue. One way the algorithm can be implemented is shown below.

```
import java.util.Random;
import java.util.PriorityQueue;

class Demonstration {
    public static void main( String args[] ) {
        createTestData();
        long start = System.currentTimeMillis();
        mergeSort(0, input.length - 1, input);
        long end = System.currentTimeMillis();
        System.out.println("Time taken = " + (end - start));
        printArray(input);
    }

    private static int SIZE = 100;
    private static Random random = new Random(System.currentTimeMillis());
    static private int[] input = new int[SIZE];
    static PriorityQueue<Integer> q = new PriorityQueue<>(SIZE);

    private static void printArray(int[] input) {
        System.out.println();
        for (int i = 0; i < input.length; i++)
            System.out.print(" " + input[i] + " ");
        System.out.println();
    }

    private static void createTestData() {

        for (int i = 0; i < SIZE; i++) {
            input[i] = random.nextInt(10000);
        }
    }
```

```
    private static void mergeSort(int start, int end, int[] input) {

        if (start >= end) {

            return;
        } else if (start + 1 == end) {
            if (input[start] > input[end]) {
                int temp = input[start];
                input[start] = input[end];
                input[end] = temp;
            }

            return;
        }

        int oneThird = (end - start) / 3;

        // sort first half
        mergeSort(start, start + oneThird, input);

        // sort second half
        mergeSort(start + oneThird + 1, start + 1 + (2 * oneThird), input);

        // sort third half
        mergeSort(start + 2 + (2 * oneThird), end, input);

        // merge the three sorted arrays using a priority queue
        int k;

        for (k = start; k <= end; k++) {
            q.add(input[k]);
        }

        k = start;
        while (!q.isEmpty()) {
            input[k] = q.poll();
            k++;
        }
    }
}
```



Merge Sort with 3 Subproblems

b-

How many recursion levels will the be generated?

Say we are provided with an array of size n. The question we need to ask ourselves is how many times do we need to successively divide n by 3 to get to 1? If there are 9 elements, we'll divide once to get 3 and then one more time to get 1. This can be expressed as a logarithmic equation as follow:

$$log_3 n = x$$

$$log_3 n = x$$

The number of levels of the recursion tree will be 1 more than $\log_3 n$ so the correct answer would be $\log_3 n + 1$.

c-

The recurrence equation will be given as:

$$T(n) = Cost\ to\ divide\ into\ 3\ subproblems + 3 * T(\frac{n}{3}) + Cost\ to\ merge\ 3\ subproblems$$

We determined in the previous question the number of recursion levels for the 3-way merge sort will be $\boldsymbol{log_3 n+1}$. Next, we need to determine the cost to merge the three subproblems. Unlike in traditional merge sort, the 3-way merge sort uses a priority queue to create a min-heap before attempting a merge of the three subproblems. Insertion into a priority queue takes $\boldsymbol{lgn}$ time and since we insert all the n elements into the queue, the total time to create the heap comes out to be $\boldsymbol{nlgn}$. Finally, the cost to divide the problem into three subproblems is constant and can be represented by $\boldsymbol{d}$. Therefore, the time complexity will be:

$$T(n) = (log_3 n + 1) * (d + nlgn)$$

$$T(n) = dlog_3 n + d + nlgnlog_3 n + nlgn$$

$$T(n) = O(nlgnlog_3 n)$$

Since lgn > $\log_3 n$, we can simplify to:

$$T(n) = O(n(lgn)^2)$$

d-

Had we not used a priority queue while implementing the 3-way merge sort and instead written a merge of three arrays in O(n) then the complexity of the

3-way merge would have come out to be ***O(nlog$_3$n)***. We used priority queue to

simplify the code at the expense of efficiency. In professional settings, it is almost always preferable to write readable and maintainable code than to optimize for running time. It would appear that ***log$_3$n*** is less than ***lgn*** for large values of n, however 3-way merge may not end up being faster than 2-way merge sort for reasons discussed below.

Subdividing a problem into more parts may not necessarily improve runtime of the algorithm. In the case of merge sort if we divide the array into three parts we end up with lesser number of recursion levels since $\log_3 n$ is less than $\log_2 n$. However, the number of comparisons performed in the combine-step increases in case of 3-way merge sort and the running time will come out to be greater than that of traditional merge sort for large inputs.

Similar argument can be made about binary search. Dividing the search space into three parts may not necessarily speed up the running time for ternary search.

Solution 2

The optimized code for printing permutations uses a slightly different loop for the regular case as depicted below.

```
        // regular case
        for (int i = index; i < str.length; i++) {
            swap(str, index, i);
            permutate(str, index + 1);
            swap(str, index, i);
        }
```

In the unoptimized version, we ran the for loop for the entire length of the array. However, in the optimized version, we can see that the loop would run for:

$$n$$

$$n-1$$

$$n-2$$

$$.$$

.

1

The total number of invocations of the permute method would still remain unchanged, only the cost of each permute method would change. If you were to determine the exact expression representing the cost then it would come out to be smaller in value than the exact expression for the unoptimized version. However, the big O will remain unchanged as we already have seen that the series

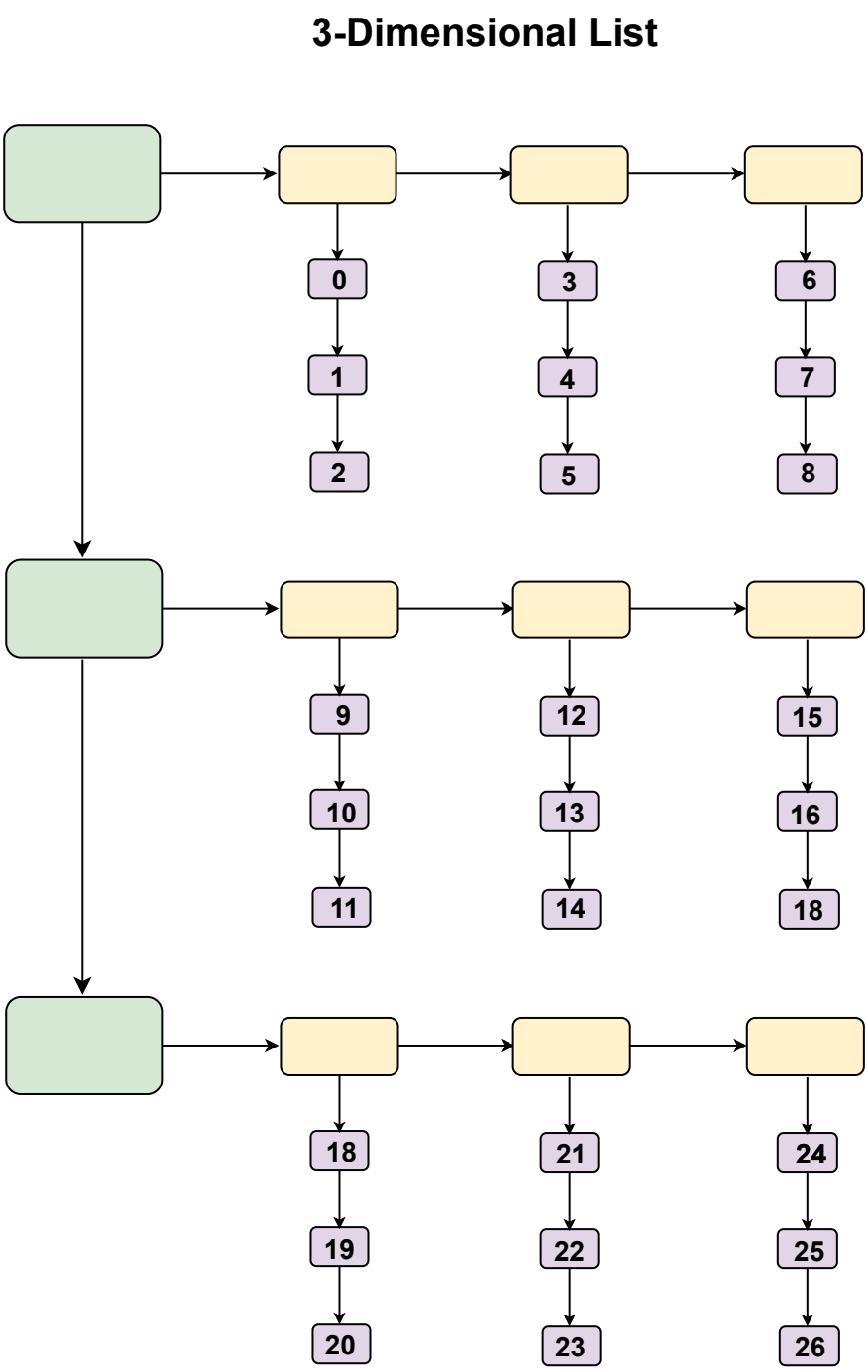
$$1 + 2 + 3...n - 2 + n - 1 + n$$

sums upto $^{n(n+1)}/_{2}$ which is again O(n).

Solution 3

a- The below picture will help you visualize the resulting data-structure when n=3. The problem can be easily solved recursively. From the diagram below, it should be clear that the total number of nodes in the leaf-lists would be:

$$Total\ Leaf\ Nodes = n^n = 3^3 = 27$$



We are essentially creating a tree of Linked Lists with the integers being stored at the leaf nodes. The root is a linked list of three objects. Each object at the root level points to another linked list of three objects. In turn, objects at the second level point to yet another linked list of three objects. The objects at the last level point to a linked list of a single element in which integers are stored.

You can see how we can recursively do the allocation given the value of *n*. The following code is one way to solve the problem. The tricky part to the solution is given an index, we need to find the appropriate node it points to. Say, if we are given index=25, then we'll divide it by *n* for *n* times and at each iteration

note the remainder.

index = 25 and n = 3

25 / 3 = 8 and remainder is 1
8 / 3 = 2 and remainder is 2
2 / 3 = 0 and remainder is 2

You can verify that to get to index 25, we'll start with the third linked list at the root level, then the third linked list at the second level and finally, the second list at the leaf level. Look at the `getIndices` method to see how we compute the right linked lists to go to.

```
import java.util.LinkedList;
import java.util.List;
import java.util.Stack;

class Demonstration {
    public static void main( String args[] ) throws Exception{
        // 1D list
        NDimensionalList list = new NDimensionalList(1);
        list.put(0, -10);
        System.out.println(list.get(0));
        System.out.println();

        // 2D list
        list = new NDimensionalList(2);
        list.put(0, -3);
        System.out.println(list.get(0));
        list.put(3, 5);
        System.out.println(list.get(3));
        System.out.println();

        // 3D list
        list = new NDimensionalList(3);
        list.put(25, -100);
        System.out.println(list.get(25));
        list.put(0, 21);
        System.out.println(list.get(0));
        list.put(26, 211);
        System.out.println(list.get(26));
        System.out.println();

        // 5D list
        list = new NDimensionalList(5);
        list.put(124, 313);
        System.out.println(list.get(124));
        System.out.println();

    }
}

class NDimensionalList {
```

```
int n;
int maxIndex;

List<Object> superList;

public NDimensionalList(int n) {
    if (n <= 0)
        throw new IllegalArgumentException();

    this.n = n;
    this.maxIndex = (int) Math.pow(n, n);
    allocate(n, superList);
}

// Recursive method that initializes the resulting N-Dimensional List
private void allocate(int rem, List<Object> list) {

    if (rem == -1)
        return;

    if (superList == null) {
        superList = list = new LinkedList<Object>();
        allocate(rem - 1, list);
    } else {
        for (int i = 0; i < n; i++) {
            List<Object> subList = new LinkedList<Object>();
            list.add(i, subList);
            allocate(rem - 1, subList);
        }
    }
}

// Calculates the index to the right node
private Stack<Integer> getIndices(int index) {
    Stack<Integer> stack = new Stack<Integer>();

    for (int i = 0; i < n; i++) {
        stack.push(index % n);
        index = index / n;
    }

    return stack;
}

@SuppressWarnings("unchecked")
public Integer get(int i) throws Exception {

    if (i < 0 || i >= maxIndex)
        throw new IndexOutOfBoundsException();

    Stack<Integer> stack = getIndices(i);
    List<Object> temp = superList;

    while (!stack.isEmpty()) {
        temp = (List<Object>) temp.get(stack.pop());
    }

    return (Integer) (temp.get(0));
}

@SuppressWarnings("unchecked")
public void put(int i, int num) {
```

```
        if (i < 0 || i >= maxIndex)
            throw new IndexOutOfBoundsException();

        Stack<Integer> stack = getIndices(i);
        List<Object> temp = superList;

        while (!stack.isEmpty()) {
            temp = (List<Object>) temp.get(stack.pop());
        }

        temp.add(0, num);
    }
}
```

d- Before we determine the complexity of initialization, let's find out the space complexity, which will also help us figure out the former. Looking at the diagram, you can count all the nodes and verify they sum up to:

$$Total\ Nodes = 3^1 + 3^2 + 3^3$$

$$Total\ Nodes = 3 + 9 + 27$$

$$Total\ Nodes = 39$$

The above is a geometric series and the sum of powers of any integer ***x*** can be represented as:

$$Sum = x^0 + x^1 + x^2 + \cdots + x^k$$

$$Sum = 1 + x^1 + x^2 + \cdots + x^k$$

$$Sum = \frac{x^{k+1} - 1}{x - 1}$$

We'll plug in the value of the n equal to 3 in the above formula and also subtract one since our sum doesn't include $3^0 = 1$.

$$Sum = \frac{3^{3+1} - 1}{}$$

$$Sum = \frac{3^4 - 1}{3 - 1}$$

$$Sum = \frac{81 - 1}{2}$$

$$Sum = \frac{80}{2}$$

$$Sum = 40$$

But we also need to subtract 1.

$$Total\ Nodes = 40 - 1 = 39$$

Therefore for an n-dimensional array the total number of nodes will be:

$$Total\ Nodes\ in\ NDimensional\ List = \frac{n^{n+1} - 1}{n - 1} - 1$$

We can rewrite the above equation as:

$$Total\ Nodes\ in\ NDimensional\ List = \frac{n^{n+1}}{n - 1} - \frac{1}{n - 1} - 1$$

For big O we can ignore the last term, which is a constant. Similarly, the second term will tend to zero as *n* becomes larger and larger

$$\frac{1}{n - 1} \to 0\ as\ n\ \to \infty$$

So both the second and the last terms will not affect big O as the input size *n* becomes larger and larger. For very large n, total nodes are roughly equal to:

$$\frac{n^{n+1}}{n - 1}$$

$$Total\ Nodes\ in\ NDimensional\ List \approx \frac{n^{n+1}}{n-1}$$

From the denominator, we can also ignore the constant 1 as for very large values of *n*, the values *n-1* and *n* are roughly equal.

$$n - 1 \approx n$$

We can now restate the sum of nodes as:

$$Total\ Nodes\ in\ NDimensional\ List \approx \frac{n^{n+1}}{n}$$

$$Total\ Nodes\ in\ NDimensional\ List \approx \frac{n^n * n}{n}$$

$$Total\ Nodes\ in\ NDimensional\ List \approx n^n$$

$$Total\ Nodes\ in\ NDimensional\ List\ is\ O(n^n)$$

There, we have worked out the space complexity to be ***O(n^n)***.

b- In order to determine the time to initialize the n-dimensional list, we need to realize that each node of the tree has to be visited and initialized. In fact, in our given solution, the leaf node is also a linked list of length one. We already know the big O for the total number of nodes in the list, therefore the big O for initialization would be proportional to the number of nodes that need to be visited which is **O(n^n)**.

c- To determine the complexity of the `get` and `put` method realize that both the methods call `getIndices` which runs a loop for *n*. Therefore to get the indices of the linked lists to traverse, we need to run a loop for **O(n)**. Next, once we have determined the indices, we need to traverse to the right leaf node.

Look at the diagram above and note that to get to the last leaf node with **index = 27**, we need to hop over all the 3 green nodes, then we hop over 3 yellow nodes and lastly we hop over 3 purple nodes for a total of **3 * 3 = 9 nodes**. So

can the complexity to reach a leaf node be **n * n= n^2 ?** If you think about it, it

makes sense. The n-dimensional list will have exaclty *n* levels. At each level, in the worst case, we'll need to hop over *n* nodes to get to the next linked list to traverse or hit a leaf node. Hence the complexity will indeed be **O(n^2)**.

But Don't forget the `getIndices` method ! The total complexity is:

> $$O(n) + O(n^2) = O(n^2)$$

Array

This lesson talks about the complexity of operations on an array.

In this section of the course, we'll consider various data-structures and understand how different operations on them can be analyzed for complexity. We'll also digress into general discussions around trade-offs of space and time, so as to cultivate the readers' thought process for reasoning about complexity.

We'll start with the most basic data-structure - *the Array*. Almost all languages allow users to create arrays. Insert or retrieval operations on an array are well-known to be constant or O(1) operations. But what makes these operations constant time operations?

Contiguous memory allocation is the key to an array's constant time retrieval/insert of an element given the index. Any language platform that you code in would request the underlying operating system to allocate a contiguous block of memory for an array. Suppose the following is a block of memory allocated by the operating system for an array of 10 elements in computer memory.

Block of memory allocated by the operating system.

Suppose the array we requested above is for integers. The next part to the puzzle is the size of the *type* for which we requested the array. The type refers to the data type the array holds, it could be integers, characters or even objects. In the case of Java, an integer is 4 bytes so let's go with that. Realize that if given the starting address of the array in the computer memory, it is trivial to compute the starting memory address of any element in the array given the index of the element.

As an example, say the starting address of the array for the below snippet of code is 25.

```
// initialize array of 10 elements.
// The variable myArray points to memory address 25
int[] myArray = new int[10]
```

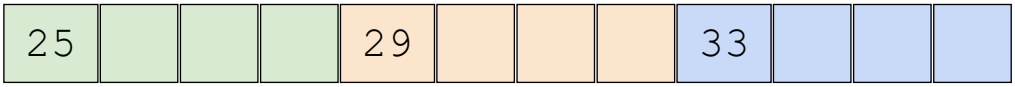
Arrays with 10 elements and start address of 25

The variable `myArray` literally points to **25** in the computer's memory. We also know each integer is 4 bytes in size. The first integer would lie at address **25**, the second integer would lie at **25** + **(1 * 4)** = **29**, the third integer would lie at **25** + **(2 * 4)** = **33**. To find out where the **nth** element of the array lies in the memory, we can use the following generalized formula:

$$Address\ of\ Nth\ Array\ Element = (Start\ Address\ of\ Array + (n * Size\ of\ Type))$$

For our array example the formula would become:

$$A[n] = Start\ Address\ of\ Array + (n * 4)$$



Memory representation of first 3 integers of the array

The final piece to the puzzle is to understand that only a mathematical calculation is required in order to access any element of the array. Ask yourself if the array size is 10, 100 or 1 million? Does the number of steps change for computing the formula to find the address of any element in the array? No, it doesn't. The size of the array has no bearing on how we retrieve

array? No, it doesn't. The size of the array has no bearing on how we retrieve the address for a given element. Once we compute the address, the OS and the underlying hardware will work in tandem to retrieve the value for the array element from the physical memory.

The mathematical calculation will always take the same amount of time whether we retrieve the address for the first element or the millionth element in the array. Hence retrieval or access operation on an array is constant time or O(1). Retrieving the last element or the middle element of the array requires the same number of steps to compute the element's address in memory.

Other Considerations

The astute reader would notice that an array forces us to block a chunk of memory. If most of our array is sparse then we end up wasting a lot of space. This is a well-known trade-off we make to gain time in exchange for space. Insert and retrieval become constant time operations, but then we may be wasting a lot of space.

Nuances in Scripting Languages

Folks with experience in Javascript or other languages which don't require declaring the size of the array at the time of initialization, may wonder if the same principles discussed above apply. Yes, they do. Behind the scenes, arrays have fixed sizes. If the user attempts to add items to an array which is completely filled, the platform will create a new array. The new array will have bigger capacity, and it copies elements from the previous array into the new array. This complexity is masked from the developer. These types of self-resizing arrays are called *dynamic arrays* and are discussed in later sections.

Pop Quiz

Q

In our analysis we reasoned that since the size of the type is fixed (integer was assumed to be 4 bytes), we can mathematically compute the starting address of any item in the array. Would the array access still be constant time if it were an array of types which could have varying sizes?

Assume you have an array of objects. These objects could have any number of fields including nested objects. Can we consider the access time to be constant?

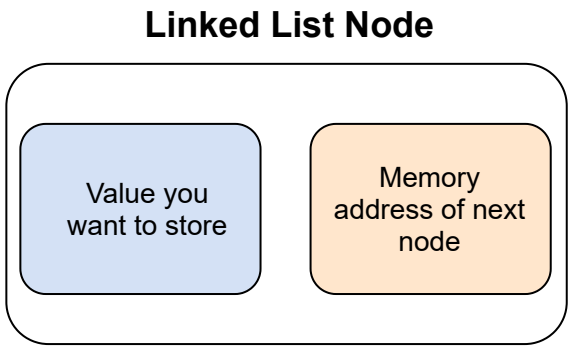
Check Answers

Linked List

This lesson discusses the complexity of operations on a linked list.

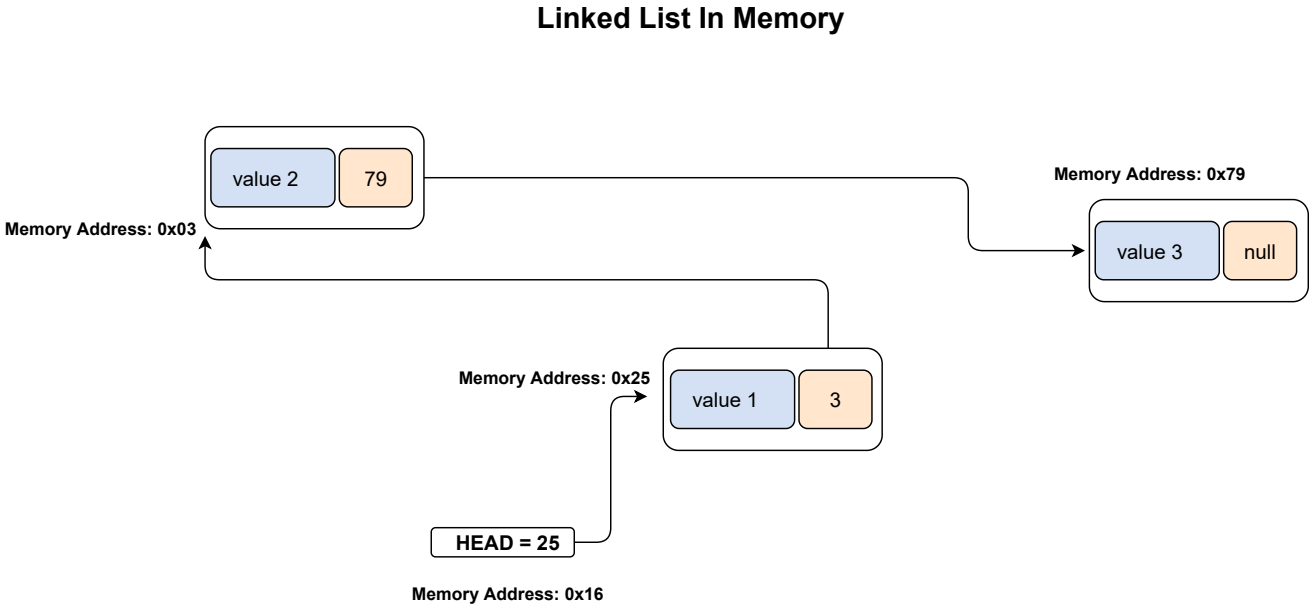
Linked list is the natural counterpart to the array data-structure. In linked list's case we trade more time for less space. A linked list is made up of nodes, and each node has the following information:

- value
- pointer to next node



You can think of a linked list as a chain. If you have a single value, you'll have a linked list of one node. If you require 100 values to store, your linked list will be 100 nodes long. Contrast it to an array, and whether you need to store one value or a 100, you'll need to declare an array of size 100 or the maximum values you intend to eventually store.

Linked list is able to save on memory and be slow in retrieval because it ***doesn't require contiguous allocation of memory***. Nodes of a linked list could be all over the place in a computer's memory. This placement requires us to always remember the first or head node of the linked list. Below is one possible representation of a linked list consisting of three nodes in memory. Note that `head` is a variable that contains the *address* of the first node in the linked list. If we lose the `head`, we lose the linked list. Unlike an array, you can see that the nodes of the linked list can be anywhere.



Given the above representation, it becomes very easy to reason about the various operations in a linked list.

- If we want to retrieve the ***nth*** element in the list, we'll need to walk the list from the head all the way to the ***nth*** element. In the worst case, we could be asked to retrieve the last element of the list. Therefore, in the worst case, the retrieval would need us to walk to the back of the linked list for a total of **n** steps, which would make the complexity of retrieval for a linked list O(n).
- Note that retrieving the first element or the head is still a O(1) or constant time operation.
- Insertion is an interesting operation. If you add at the start of the linked list, it is a constant time operation. If you desire to add an element at the ***nth*** position, where n could be the very end of the linked list, then the insert operation takes O(n) time.

Another way to think about these operations is to ask yourself this question: if the size of the linked list increases, does it affect how many steps are required to perform a given operation? If we insert at the head of the linked list, no matter how long our linked list is, the operation will take a constant number of steps. However, adding to the very end of the linked list requires us to first reach the last element, and only then are we be able to append a new element.

Therefore, the size of the linked list starts to matter and affects complexity.

Improving Runtimes

At times, runtimes of operations can be improved on data structures. For instance, if your application logic requires adding new elements at the tail of a linked list, then one can maintain a `tail` pointer (similar to the `head` pointer) that will always point to the end of the linked list. This makes insertion at the end of a linked list O(1) operation instead of O(n).

Space Complexity

The astute reader would notice that we are also storing an additional variable with each node of the linked list - a variable that points to the next node in the chain. So if we have a linked list of 1000 nodes, then we have one thousand of these `next` pointer variables taking up memory too! The space complexity would be:

> $$n + n * (pointer\ variable\ size)$$

Since the pointer variable size would be a constant number of bytes in any language, we'll be left with,

> $$n + n * (constant)$$
>
> $$n + n * c$$
>
> $$(c + 1)n$$
>
> $$O(n)$$

Pop Quiz

Q

What would the time complexity to find the predecessor of a node in a linked list?

Check Answers

Hash Table

This chapter discusses operations on hash tables.

A hash table or hash map is essentially a *key, value* pair. The simplest way to think about a hash table is to imagine a row of houses in a neighborhood and a watchman for the neighborhood. You only know the names of the people in the neighborhood, but not their houses. You ask the watchman, where Mr. Zuckerberg lives and he points to house number 5. Next, you ask him where Mr. Gates lives and he points to house number 2. A hash table is similar in the sense that we'll give the data-structure a ***key***, and it will map our key to a slot where we can then place our ***value***.

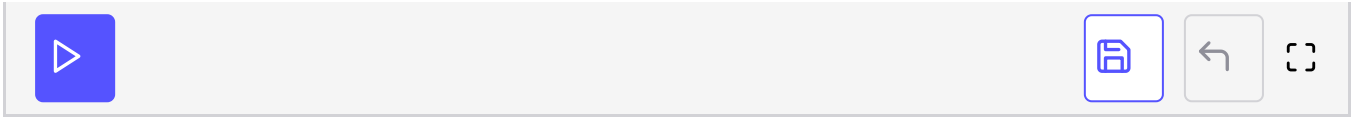
The watchman maintains the mapping in his head, and the hash table will use a ***hash function*** to compute the mapping. Just like the watchman, the hash function will always point to the same slot when given the same key.

Array as a Hash Table

Let's say we are given a string of lowercase English-alphabet characters and are asked to find the frequency of each character in the given string. We can use an array of integers to act as our hash table.

```
class HelloWorld {
    public static void main( String args[] ) {

      String givenStr = "aanalsnlisgohlsdfnkladpndkkfla";
      int[] hashTable = new int[256];

      for(int i=0;i<givenStr.length();i++){
        char currentChar = givenStr.charAt(i);
        int key = hashChar(currentChar);
        hashTable[key] = hashTable[key] + 1;
      }

      for(int i=0;i<hashTable.length;i++){
        if(hashTable[i] !=0){
         System.out.println((char)('a'+i) + " :  " + hashTable[i]);
        }
      }
    }
```

```
    static int hashChar(char currentChar) {
      return currentChar - 'a';

    }
}
```



The hash function in our example is very simple. It merely subtracts 97='a' from the integer equivalent of the current character to normalize the index to one of the 26 slots.

Operations on Hash Table

- Let's talk about insertion first. Ask yourself, if the hash table consists of a million or a dozen entries, does it affect the number of steps to compute the hash of the key and place the value in the resulting slot? No, it doesn't. The hash function - **if assumed to take constant time to compute the hash** - isn't affected by the size of the table. Once the slot is determined, placing the value in the slot should also not be dependent on the size of the rest of the table and is, therefore, a constant time operation. Note, the assumption in bold is crucial for the hash table to maintain constant time operations.

- The hard part was the insertion operation, if that takes constant time then retrieval should be a breeze. We simply compute the hash of the given key, go to the right slot and retrieve the value. All-in-all, a constant number of steps allows for constant time.

When Rubber Meets the Road

If only the previous paragraphs were true in reality. Unfortunately, things get complicated when two different keys have their hashes pointing to the same slot. It's the same as the watchman saying Bill Gates and Jennifer Gates(daughter) live in house number 2. A **collision** is said to take place when two or more keys hash to the same slot. There are different ways to handle collisions, and each way will affect the complexity of different operations.

Collisions

If we design a naive hash table in which collisions are simply handled by a

linked list, then insertion will still take constant time. Because, every time; a collision happens, we simply append to the head of the linked list maintained for the slot. However, imagine a really bad hash function, which hashes all keys to the same slot. In that scenario, you end up with a linked list masquerading as a hash table. Insertions are still O(1), but retrieval might mean going to the end of the linked list to retrieve a value. Therefore, the retrieval complexity becomes O(n) and defeats the purpose of using a hash table.

To mitigate the collision scenario, we can use a self-balancing binary search tree. If we are guaranteed by the hash function that a maximum of K keys can hash to the same slot, the retrieval in case of collision will take

$$log_2 K$$

time which is simply lgK. Note, we have sought an implicit guarantee from the hash function to work in constant time to come up with worst case retrieval complexity.

Hash Functions

There's a lot of science and mathematics behind creating good hash functions which are beyond the scope of this text. However, do remember that the hash function must take constant time to compute the hash of the given key. Also, collisions do happen and are expected. Your best bet for O(1), or constant time insert and retrieval operations is, to use hash tables provided by standard libraries in your language of choice.

Doubly Linked List

This section explores operations on a doubly linked list

We ended the section on linked list with a question on the complexity of an operation that seeks to find the parent of a given linked list node. The answer is O(n) since if you give me a linked list node, I still need to start from the head and traverse down the linked list matching each node with the one given to me until I find an exact match. At that point, I return the just previously seen node that I can keep track of using an additional variable. Worst case, it takes me to the end of the linked list for a total of n operations.

Doubly Linked List

Meet doubly linked list. It solves the above described problem by keeping a link to the previous node. Sure, we now have double the number of nodes, but the space complexity doesn't change.

> $$n + 2n * (pointer\ variable\ size)$$

Since the pointer variable size would be a constant number of bytes in any language, we'll be left with,

> $$n + 2n * (constant)$$
>
> $$n + n * 2c$$
>
> $$(2c + 1)n$$
>
> $$O(n)$$

However, by using the additional back pointer node, we are now able to

determine the predecessor for a node in constant time or O(1). The complexity for the rest of the operations remains the same as the linked list.

One may wonder what the benefit is of finding the predecessor in constant time. Usually, a doubly linked list is combined with a hash table to create a more advanced data structures called a least recently used cache or LRU cache.

Pop Quiz

Q

Given a doubly linked list node, what is the complexity of deleting the node?

Check Answers

Stacks and Queues

This lesson talks about operations on stacks and queues.

Stack

Stack is a well-known data-structure, which follows *first in, last out* paradigm. It offers `push` and `pop` operations. We'll examine the complexity of these operations when stack is implemented using an array or a linked list.

Stack using Linked List

A stack can be implemented using a linked list. New items are always appended and removed at the head of the list. We discussed in the linked list section that appending an item to the head of a linked list takes constant time. Therefore, both `push` and `pop` operations take constant or O(1) time.

However, note that when we use a linked list we are still using an extra pointer in each node of the list to keep track of the next node in the chain. That additional cost allows us to theoretically have a stack with an infinite capacity. We can keep on pushing onto the stack for as long as the computer memory allows. Instead of ***O(n)*** space, we'll be using ***O(2n)*** space which is still ***O(n)***.

Stack using an Array

Stack can also be implemented using an array. We can keep an integer pointer ***top*** to keep track of the top of the stack and increment it whenever a new push occurs. Poping elements is also a constant time operation because we can easily return ***array[top]*** followed by a decrement of the ***top***.

While using an array, we are saving the cost of having an additional pointer. At the same time, the size of the stack is limited to the initial size of the array. If the stack is completely filled then we'll need to create a bigger array, and copy over the stack to the new array, and discard the old one. A dynamic array may be used in this scenario to amortize the cost.

Another disadvantage of using the array is that it ties up memory if the stack is lightly used. Assume you created a stack thinking in the worst case it could have a thousand elements. The backing array for the stack will also be of size 1000. Say the stack is only filled with 100 elements for 90% of the time, then 90% of the memory isn't being used 90% of the time. This scenario wouldn't happen in case the stack was implemented using a linked list.

Queue

The queue is a type of data-structure that lets the first element enqueued also be the first element dequeued or *first in, first out*. Again, we can implement a queue using either an array or a linked list. It offers two methods: `enqueue` and `dequeue`.

Queue using Linked List

We can implement a queue using a linked list, where elements are always dequeued from the head and enqueued at the tail. We'll need two pointers - one head and one tail - to keep track of the front and back of the queue. Both the enqueue and dequeue operations take constant time. However, the price paid is in terms of the extra *next* pointer variable that will be part of each node of the list.

Queue using Array

Implementing a queue using an array is slightly tricky, since it requires appropriate manipulation of the front and the end pointers to keep track of the head and the tail of the queue respectively. However, both the operations take constant time since accessing any array element takes constant time.

The same tradeoffs exist that we discussed when implementing a stack using an array or a linked list. The queue can grow without bounds if implemented via a linked list, but not with an array. Using a linked list will incur an additional cost of a pointer node but will give flexibility in growing the queue without copying.

Summary

In crux, arrays should be used to implement queues or stacks when the maximum size of each is known beforehand. Also, it would help to know the expected utilization of the data-structure so that one is aware of how much memory will be tied up and may not be used.

Linked list should be used to back an array or stack when the maximum size isn't known beforehand. It'll also make sense to use a linked list when maximum size is known in advance but is only expected as a spike (i.e. the stack or the queue will not hold the maximum or near maximum number of elements for most of the time.)

Tree Structures

This lesson is a general discussion on reasoning about space and time complexity of tree structures

In this lesson we won't attempt to describe various tree data-structures and their associated operations. Rather, we'll do a general discussion on how to reason and think about space and time complexity when tree structures are involved.

Tree Traversals

When traversing a tree, does the size of the tree affect how much time it takes for you to go through every node of the tree? Yes, it does. A bigger tree will take more time to go through than a smaller tree. The following recursive algorithms for tree traversals all take O(n) time since we are required to visit each node.

- pre-order traversal
- in-order traversal
- post-order traversal
- depth first search. Note the above three algorithms are types of depth first traversals
- breadth first traversal/search

For the above algorithms, the more interesting question pertains to space complexity for the above algorithms. Since one can write these algorithms recursively, it's not immediately clear what the space complexity is. A novice may think it is constant when it is not. Look at the code below:

```
class TreeNode {
    TreeNode left;
```

```
    TreeNode right;
    int val;
}

void recursiveDepthFirstTraversal(TreeNode root) {
    if (root == null)
        return;

    recursiveDepthFirstTraversal(root.left);
    System.out.println(root.val);
    recursiveDepthFirstTraversal(root.right);
}
```

A naive look at the `recursiveDepthFirstTraversal` method may mislead you to think that there's no extra space being used. However, behind the scenes, the recursive calls are stored on a stack by the underlying language platform. When we implement the non-recursive version then we are better able to appreciate the space requirements. Take a look at the non-recursive version of in-order traversal below:

```
void depthFirstTraversal(TreeNode root) {

1.      Stack<TreeNode> stack = new Stack<>();
2.      TreeNode trav = root;
3.
4.      while (trav != null || !stack.isEmpty()) {
5.
6.          while (trav != null) {
7.              stack.push(trav);
8.              trav = trav.left;
9.          }
10.
11.         trav = stack.pop();
12.         System.out.println(trav.val);
13.
14.         trav = trav.right;
    }
}
```

Now it should be apparent that the algorithm relies on the use of a stack. It is either implicit in case of recursion or explicit if an iterative version is written out.

Height of Trees

In the traversal algorithm, a loose upper bound on the space complexity would be O(n) because the stack can be filled at most with every node of the tree. However, you might realize that the stack - at any time during the execution of the algorithm - could be ***filled at most with the maximum number of nodes that appear in the longest path from the root to a leaf node***. This observation allows us to put a tighter bound on the space requirements, as we know that the longest path from the root to the leaf would be the ***height of the tree***. Therefore, in the traversal algorithm above the space complexity will be:

> $$O(log_2 n)$$

Height of a binary tree is given by $\log_2 n$ where n is the total number of nodes in the tree. You can think of it as given *n*, how many times do you need to divide it by 2 to get to 1? The answer is $\log_2 n$.

Trees becoming Linked Lists

One should be wary of scenarios when trees turn into linked lists. In case of a binary search tree, if it consists of all the nodes with the same values or inserts are made in an ascending order of values, then the binary search tree would turn into a linked list. Suddenly the *O(lgn)* promise for search is broken and search is now linear or *O(n) operation.*.

Graph Traversals

If asked to do graph traversal, the complexity would be O(n) where n is the total number of nodes. Visiting each node once is the least number of nodes you can visit to cover the entire graph. For a graph with cycles, you'll need to remember the visited nodes so as not to enter into an infinite cycle. The space complexity would change now. Can you think of the space complexity in case of a graph with cycles?

In the worst case, the graph is just one circle, and unless you visit the node again, you won't know you were in a cycle. In that case, you'll end up remembering all the nodes of the graph, and the space complexity will

become O(n).

Binary Trees

A binary tree is a tree which has at most two children. Don't confuse a binary tree with a binary search tree. Searching in a binary tree will take O(n) time, but in a binary search tree it'll take O(lgn). However, an unbalanced binary search tree can still result in linear search.

Let's reason about the space complexity of representing a binary tree consisting of only integers. We can store the binary tree in an array or create a class like below to represent each node of the tree

```
class TreeNode {
    int val;
    TreeNode leftChild;
    TreeNode rightChild;
}
```

There’s no right answer here, and the right approach would depend on the scenario you are involved in. If it is known that the binary tree will be full, i.e. all nodes except the leaves will have two children, then it would make sense to use an array. It’ll save us the cost of the two pointer-nodes in the other approach. But if the binary tree is expected to grow then it may be better to use the second approach. Also, if the binary tree is very tall but is sparsely filled, then the array approach would waste a lot of space.

In the array approach we'll allocate space for the full binary tree, even if the number of nodes is less. Otherwise, in the second approach, the space complexity will be proportional to the size of the number of nodes. The extra pointer nodes add constant space per node so the big O space requirement will still be O(n).

Another aspect to pay attention to is that using an array to store a binary tree makes access to any node of the tree a constant time operation. However, if you choose to represent it using the TreeNode approach then accessing the leaf nodes would take time proportional to the height of the tree.

Problem Set 4

Question 1

Kim is new to programming and designs the following hash function for her hash table.

```
int computeHash(int key) {

    int hash = 0;
    for (int i = 1; i <= key; i++) {
        hash += i;
    }

    return hash;
}
```

a- What is the time complexity of the hash function?

b- Can we make the hash function take constant time?

c- What is the space requirements/complexity if Kim chooses to stick with her hash function and wants to ensure no collisions happen?

Solution Set 4

Solutions to problem set 4.

Solution 1

a- Kim is unfortunately using a bad hash function. Her function computes the sum of integers from **1** to **n** and hashes **n** to the slot numbered **sum**. To calculate the hash of a given *k*, the loop would run for k steps. The function thus has a complexity of O(k). Note, how a hash table using this hash function would fail to provide O(1) insert and retrieval operations.

b- The hash function is computing the sum of numbers from 1 to k. We already know that the summation of consecutive numbers from 1 to k can be represented by the following formula

$$sum = \frac{k(k+1)}{2}$$

The hash function can compute the above formula and return the hashed value of the input key. The above computation would take constant time when worked out.

c- If Kim wants to avoid collisions she will want to make sure that the biggest key can hash to a slot without collision. For example, if the biggest key is equal to 10, then it's hash would be:

$$\frac{10(10+1)/2}{2} = 55$$

If Kim uses an array as a hash table, then it must be big enough to accommodate the biggest key. The array size should be at least $k(k+1)/2$ elements long in length which is equivalent of saving the space

elements long in length which is equivalent of saying the space is

> $$O(k^2)$$

where k is value of the biggest key expected.

Priority Expiry Cache Problem

We'll walk through an actual phone screen question asked at Tesla in this lesson.

We all have heard of the often repeated Least Recently Used (LRU) Cache problem asked in interviews. The problem is trivial once you realize that a hash table and a doubly linked list can be used to solve it efficiently. We'll work on a variant of the LRU problem that was asked in a real-life Tesla phone screen. The intent of this problem is to exercise our minds to be able to compose various data-structures to achieve desired worst-case complexity for various operations. Without further ado, the problem is presented below:

The PriorityExpiryCache has the following methods that can be invoked:

- `get(String key)`
- `set(String key, String value, int priority, int expiry)`
- `evictItem(int currentTime)`

The rules by which the cache operates is are follows:

1. If an expired item is available. Remove it. If multiple items have the same expiry, removing any one suffices.
2. If condition #1 can't be satisfied, remove an item with the least priority.
3. If more than one item satisfies condition #2, remove the least recently used one.

And it goes without saying that we have to design our function implementations to be as fast as possible.

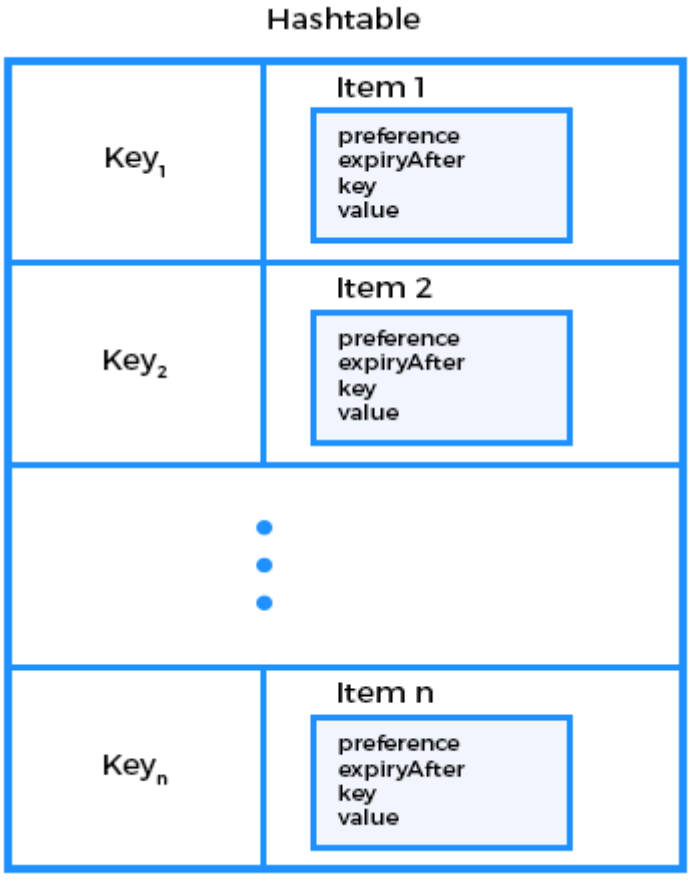
Let's start with the `get()` method. Given a key we need to return the

associated value. The fastest way we can implement a retrieval operation is to use a hash table. The insertion and retrieval times would both be O(1). Besides

the key and the value provided to us, we'll need to store the **expiry** time and the **preference**. So Let's create a class `Item` that holds all these three artifacts associated with the key.

```
class Item {
    int preference;
    int expireAfter;
    String key;
    String value;
}
```

We can now create a hash table using the `Item.key` and the value as the item object itself. It should look like as follows:

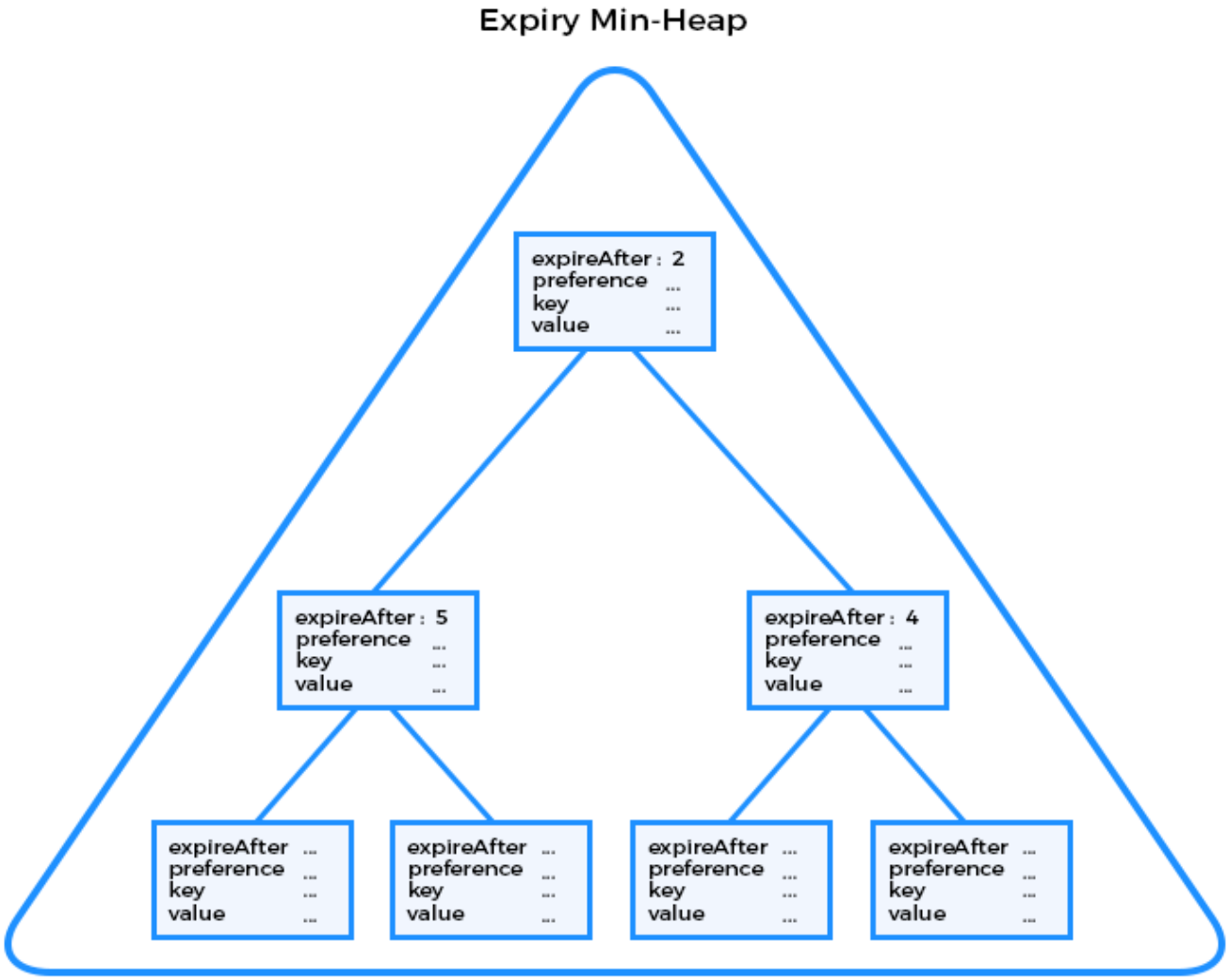


We can serve `get()` requests using the above hashtable in O(1). Whenever a get is invoked, we have to capture the notion of an item being most recently used. Hold onto that thought for now as we'll solve it once we create the other needed data-structures.

Let's start rule#1 which says that an expired item should be removed first. We need to find the item with the minimum expiry time. **Whenever you hear**

yourself looking for the maximum or the minimum among a set of values,

your likely choice of data-structure is a heap (called a priority queue in some language implementations) as it provides O(1) look-up (not removal) for minimum or maximum value. A max-heap can be used for looking up maximum values while a min-heap can be used for looking up minimum values. For our case, we can create a min-heap of the objects of class `Item`. The min-heap is predicated on the `expireAfter` variable, i.e. the min-heap is built based on the value of `expireAfter` variable for each object. Conceptually, the min-heap looks like as follows:

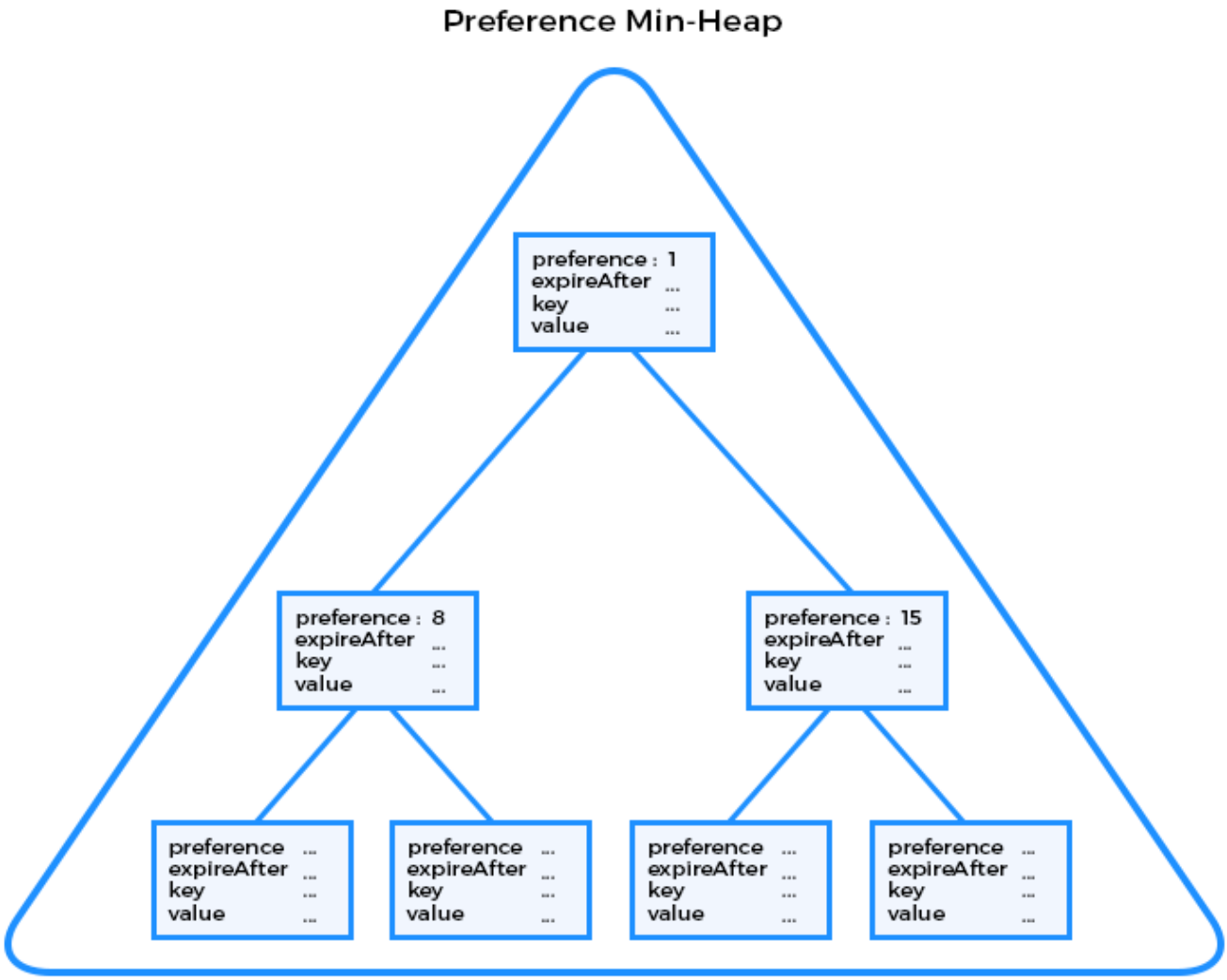


When `evictItem()` is invoked, we peek at the top of the min-heap and see if the item with the minimum expiry time is indeed expired. If so we pop the top of the min-heap and also remove it from our hashtable. Removing the minimum item from the min-heap is O(lgn) and removal from the hashtable is O(1). The total cost thus far is O(lgn) +

> $$O(lgn) + O(1) = O(lgn)$$

$$O(\log n) + O(1) = O(\log n)$$

Now consider, what happens if there are no expired items. In that case we need to find an item with the least preference. Again, that requirement hints of using a min-heap. The top of the heap will hold the item with the minimum preference. So now we have two min heaps. The second min heap is predicated on the `preference` variable. The min-heap based on preference looks like:

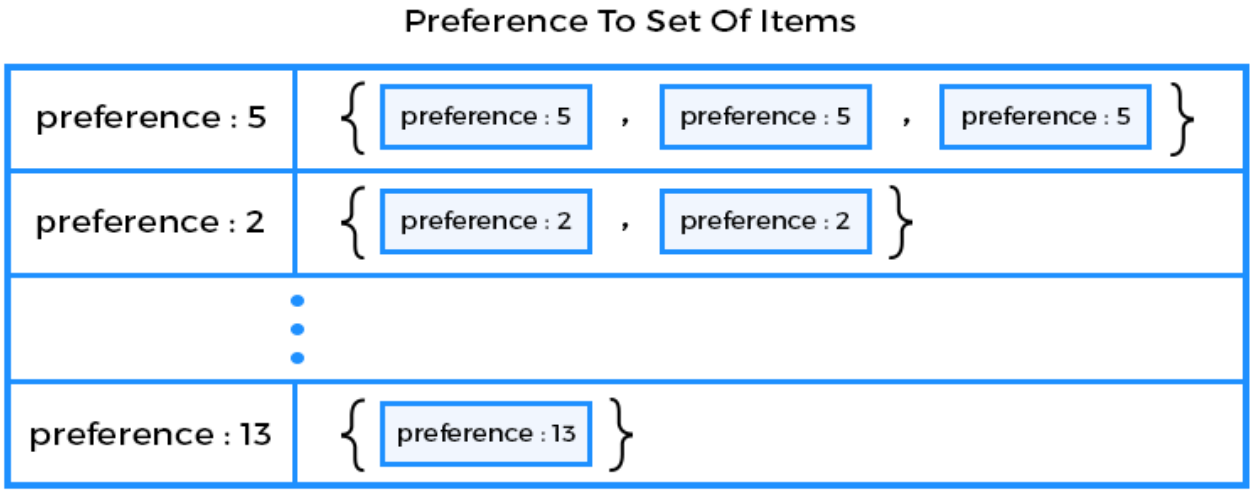


The next part is tricky, if more than two elements have the same preference, we need to remove the one which was least recently used. This implies that unlike our expiry min-heap, **we can't simply evict the item at the top of the preference min-heap, because it may have been used recently and there might be another item in the min-heap with the same preference but was accessed least recently among all the items with the same priority.**

The last sentence of the previous paragraph is very important. Looking at the top of the preference min-heap we need to get to all the elements with the least preference. Once we get to these elements we can remove the one that

least preference. Once we get to those elements we can remove the one that was least recently used. Ok, now think it through, once we get the least

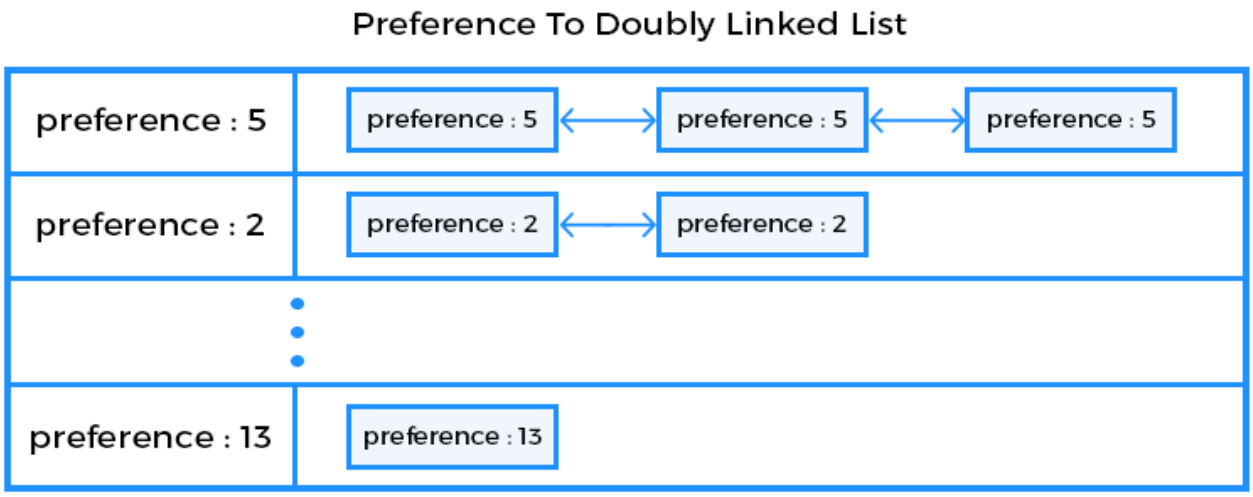
preference from the min-heap, how can we most efficiently get to all the elements with the same preference? Use a hashtable again! We can have a mapping of preference to the set of all elements with that preference. This should look like as follows:



However, keeping all the items with the same preference doesn't help us know which item was least recently used. We need to bring order to these items so that they are arranged in order of when they were last accessed. **Whenever you want to order entities, use a linked list. When you want a collection of entities without any order among them use a set.** So now, we have a mapping of preferences to a list. The key to the hashtable is the preference (an integer) and the value is a list of all the items with the same preference.

Now let's dig into how we can create the notion of least recently used when using a list. We can institute a convention that the head (front) of the list will always have the most recently used item. The implication is that whenever an item is accessed, we'll cut it out from its current position in the linked list and place it at the head of the list. The last item, thus, is always the least recently used item. However, we can't efficiently cut out an element from say the middle oof the list unless we know its predecessor (the one behind) and successor (the one after) nodes in O(1) time. We don't want to traverse the list whenever we want to cut out an item. This should ring a bell that we need a doubly-linked list, which provides O(1) access to both predecessor and successor nodes. Great! so now we have a hashtable of preferences to doubly-linked lists of items with the same preference, which should conceptually look

like as follows:



Now we have all the pieces in the puzzle together. We ended up with:

- An expiry-time min-heap (priority queue). The constituent elements of the min-heap can be the doubly-linked list nodes that wrap an item. This will ease manipulating the doubly-linked list when required, as we'll shortly see.
- A preference min-heap (priority queue). The elements making up the min-heap should be the nodes from the doubly-linked list that wrap an item but the min-heap is predicated on the preference value instead of the expiry time.
- A hash table with a mapping of key to doubly-linked list node that wraps the item with the key.
- A hash table with a mapping of preference to doubly-linked list of all the items with the same preference.

get ( )

Now we can reason about the `get()` method's complexity. When `get()` is invoked we get the key as a parameter. Using the key we can get the doubly-linked list node that contains the corresponding `Item` object in O(1) time. Next, in O(1) time we can get the wrapped item and return it to the caller. But just before returning, we must also detach the node from its current position in the list of all items with the same preference and place it at the start of the list, to indicate it is the most recently used item. This should be trivial because

we can get the preference of the item in O(1), use it to hash into the bucket

which contains the doubly-linked list of all the items with the same preference. We can remove the item from the doubly linked list in O(1) given the back and forward pointers. Once detached we can now attach it to the head of the doubly-linked list in O(1). The worst-case complexity of the `get()` method is thus:

> $$O(1)$$

evictItem ( )

Before discussing `set()` we'll discuss the `evictItem()` as it will be used by the `set()` method, when the cache is full.

We can peek (not remove) the expiry min-heap first to determine if the item with the minimum expiry time is expired and can be removed or not. Peeking or finding min in a min-heap is a O(1) operation. Let's say if the item with minimum expiry is indeed expired, then we can remove the expired item in O(logn) time. We'll also remove it from the preference min-heap which is another O(lgn) operation. We'll also need to delete it from the hashtable of key to wrapped items, which will be an O(1) operation. Next, we also need to remove it from the doubly-linked list of items with the same preference. Since we were returned the doubly-linked list node wrapping the item from the expiry min-heap, we can easily remove it from the doubly-linked list in O(1). The overall complexity will be:

> $$O(lgn) + O(lgn) + O(1) + O(1) = O(lgn)$$

Now, let's consider the case when there are no expired items. We'll peek the item with least preference from the preference min-heap in O(1) time. But note that this may or may not be the candidate for removal since it may have been more recently used than another item with the same preference. We'll use the preference of the wrapped item at the top of the min-heap to hash into the bucket containing the doubly-linked list of items with the same preference. Note this is an O(1) operation. Once we have the doubly-linked list

preference. Note this is an O(1) operation. Once we have the doubly-linked list we can simply remove the last item from the list as it is the least recently used

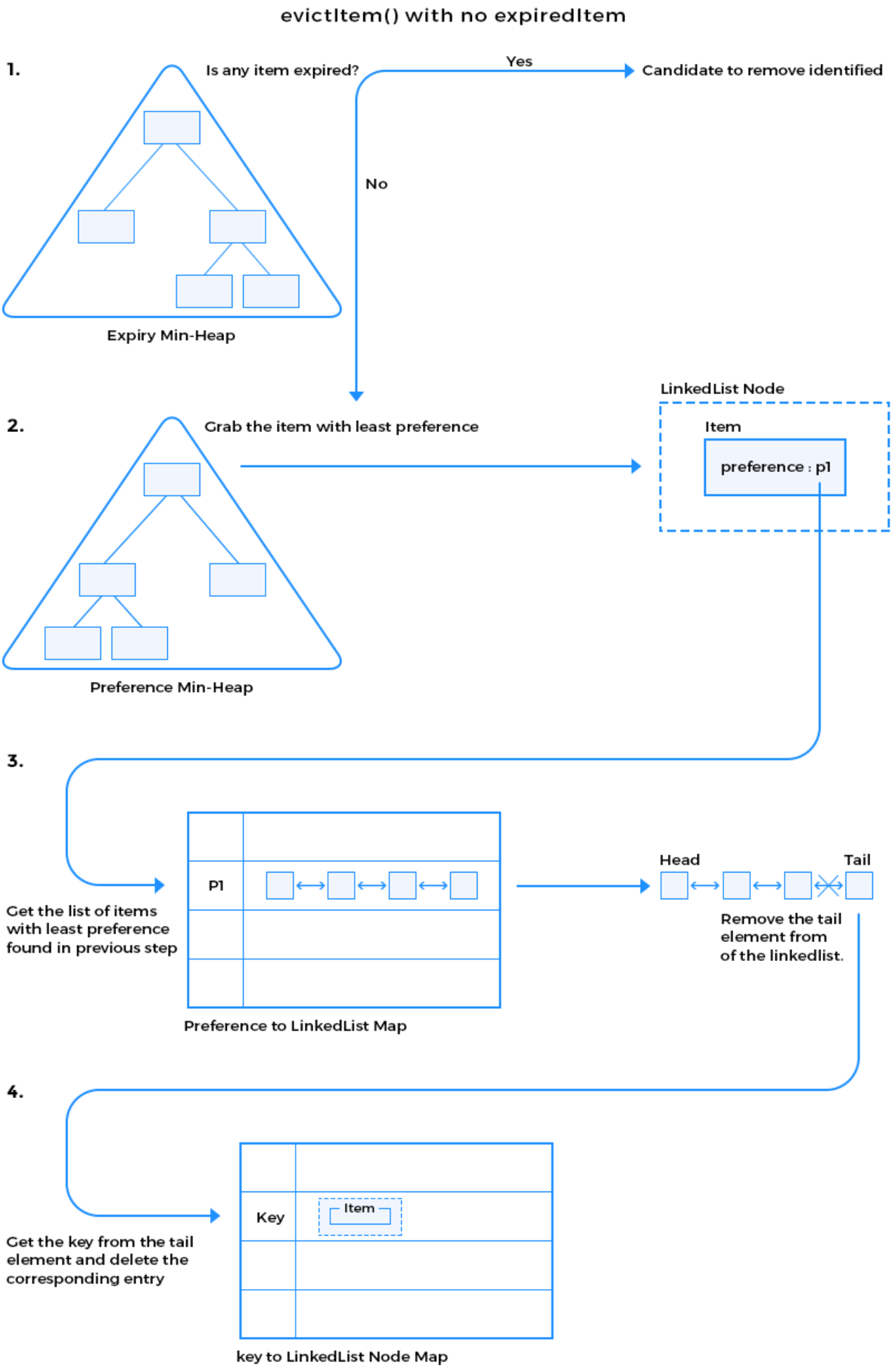
with the least preference. Remember to also remove this item from the expiry min-heap, preference min-heap, and the hash table of key to wrapped item. The runtime in this scenario is same as before:

> $$O(lgn) + O(lgn) + O(1) + O(1) = O(lgn)$$

The worst-case runtime complexity of the `evictItem()` method will be the worst-case complexity of the two scenarios, which is:

> $$O(lgn) \ or \ O(lgn) \ is \ still \ O(lgn)$$

A pictorial representation of the `evictItem()` logic when no expired item is found shown below:


evictItem() with no expiredItem
1.
Is any item expired?
Yes
Candidate to remove identified
No
Expiry Min-Heap
2.
Grab the item with least preference
LinkedList Node
Item
preference : p1
Preference Min-Heap
3.
Get the list of items with least preference found in previous step
P1
Preference to LinkedList Map
Head
Tail
Remove the tail element from of the linkedlist.
4.
Get the key from the tail element and delete the corresponding entry
Key
Item
key to LinkedList Node Map

set ()

set ()

Consider the case when the cache has capacity to accept a new item. The `set()` method will require us to add the new element to the two min-heaps, and each operation will take O(lgN). Inserting into the key-to-wrapped-item hashtable is a O(1) operation. For the preference-to-doubly-linked-list hashtable we'll simply add at the end of the list with the same preference, which would turn out to be another O(1) operation.

The case where the cache is already full we'll first execute an evict operation which will take us *O(lgn)* as we just determined and then execute the sequence of adding the new item to all of the four data-structures. all in all a O(lgN) + O(lgN) = O(lgN) operation.

$$O(lgN) + O(lgN) = O(lgN)$$

This problem is an excellent example to learn how we can compose various data-structures to construct even more complex data-structures with efficient operations.

The explanation above is language agnostic but a solution in Java is presented below.

main.java

PriorityExpiryCache.java

ListNode.java

Item.java

DoublyLinkedList.java

```
import java.util.*;

public class PriorityExpiryCache {

    int maxSize;
    int currSize;

    PriorityQueue<ListNode<Item>> pqByExpiryTime = new PriorityQueue<>((a, b) -> a.data.expi
    PriorityQueue<ListNode<Item>> pqByPreference = new PriorityQueue<>((a, b) -> a.data.pref
    HashMap<Integer, DoublyLinkedList<Item>> preferenceToList = new HashMap<>();
```

```
    HashMap<Integer, DoublyLinkedList<Item>> preferrenceToList = new HashMap<>();
    HashMap<String, ListNode<Item>> keyToItemNode = new HashMap<>();


    public PriorityExpiryCache(int maxSize) {
        this.maxSize = maxSize;
        this.currSize = 0;
        LinkedList<Item> l = new LinkedList<>();
    }

    public Set<String> getKeys() {
        return keyToItemNode.keySet();
    }


    /**
     * 1. Remove all expired items first
     * 2. If none are expired, evict the ones with lowest preference
     * 3. If there's a tie on items with least preference, evict the ones
     * which are least recently used.
     */
    public void evictItem(int currentTime) {

        if (currSize == 0) return;

        currSize--;

        // Check expired items first
        if (pqByExpiryTime.peek().data.expireAfter < currentTime) {

            ListNode<Item> node = pqByExpiryTime.poll();
            Item item = node.data;

            DoublyLinkedList<Item> dList = preferrenceToList.get(item.preference);
            dList.removeNode(node);

            // Remove from hashmap too
            if (dList.size() == 0) {
                preferrenceToList.remove(item.preference);
            }

            // Remove from hashmap
            keyToItemNode.remove(item.key);

            // Remove from preference queue too
            pqByPreference.remove(item.preference);

            return;
        }

        // Next check if preference items are to be removed
        int preference = pqByPreference.poll().data.preference;

        DoublyLinkedList<Item> dList = preferrenceToList.get(preference);

        // Remove the end
        ListNode<Item> leastRecentlyUsedWithLeastPreference = dList.removeLast();
        keyToItemNode.remove(leastRecentlyUsedWithLeastPreference.data.key);

        // Remove from the expiry queue
        pqByExpiryTime.remove(leastRecentlyUsedWithLeastPreference);
```

```
        if (dList.size() == 0) {
            // Remove the dList too
            preferrenceToList.remove(dList);
        }
    }

    /**
     * Get the value of the key if the key exists in the cache and isn't expired.
     */
    public Item getItem(String key) {

        if (keyToItemNode.containsKey(key)) {
            ListNode<Item> node = keyToItemNode.get(key);
            Item itemToReturn = node.data;

            DoublyLinkedList<Item> dList = preferrenceToList.get(itemToReturn.preference);

            dList.removeNode(node);
            dList.addFront(itemToReturn);

            return itemToReturn;
        }

        return null;
    }

    /**
     * update or insert the value of the key with a preference value and expire time.
     * Set should never allow more items than maxItems to be in the cache. When evicting
     * we need to evict the lowest preference item(s) which are least recently used.
     */
    public void setItem(Item item, int currentTime) {

        if (currSize == maxSize) {
            evictItem(currentTime);
        }

        // Get the linkedlist for the preference queue
        DoublyLinkedList<Item> dlist = null;
        if (preferrenceToList.containsKey(item.preference)) {
            dlist = preferrenceToList.get(item.preference);
        } else {
            dlist = new DoublyLinkedList<>();
            preferrenceToList.put(item.preference, dlist);
        }

        ListNode<Item> node = dlist.addFront(item);
        keyToItemNode.put(item.key, node);

        // Update the expiry time pqueue
        pqByExpiryTime.add(node);

        // Update the preference pqueue
        pqByPreference.add(node);

        currSize++;
    }
}
```

In the above example, we print the keys left in the cache after each `evictItem()` operation. As you can see the key **C** is retained in the cache till the end. We perform a `get()` on the **C** key. that makes it the most recently used item with the same preferences. The first `evictItem()` operation removes key **B** as it is expired. The second `evictItem()` operation removes key **D** as it has the least preference out of the remaining items. The third `evictItem()` removes **A**. Note that we could also have removed **E** here as both **A** and **E** are never accessed after being added. In our implementation we always add a new element to the front of the doubly-linked list, which causes **A** to be removed. Finally, on the fourth `evictItem()` call the key **E** is removed.

Cost Over Sequence of Operations

This chapter introduces the reader to aggregate analysis of algorithms.

Amortized Analysis

Readers who own a mortgage would probably be aware of amortization. The word ***amortize*** means to gradually write off the initial cost of an asset. When undertaking amortized analysis we average the cost of a sequence of operations on a data-structure, even though a single operation may be expensive. This is different than the *average* case because there's no probability involved and the average time is guaranteed.

There are three ways to conduct amortized analysis

- Aggregate Method
- Accounting Method
- Potential Method

We'll briefly describe each before showing detailed examples.

Aggregate Method

In aggregate method, we find an upper bound on the total cost of a sequence of ***n*** operations. Say the upper bound is given by ***T(n)***, then the amortized cost is expressed as:

$$Amortized\,Cost = \frac{T(n)}{n}$$

In the next section, we show the amortized cost of insertions in a dynamic array using aggregate analysis.

Accounting Method

In the accounting method, we assign costs to each individual operation. The trick is to charge certain operations more than they cost. That way, the extra credit can pay for operations that are charged less but actually cost more. Exercise caution though, as the credit can't fall below zero at any point in the sequence.

Potential Method

The potential method is similar to the accounting method. However, the credit is stored as a whole for the data-structure rather than for individual operations.

One can find data-structures offered in different languages where the costs of various operations on the data-structure are expressed as amortized costs. For instance, in the Java language the ArrayList is one such data-structure which is resizable and has constant amortized cost for insertion.

The intent of this chapter is to give a flavor and reasonable working knowledge of amortized analysis to the reader. Therefore we limit ourselves to examples of the aggregate and accounting methods.

Dynamic Array

This chapter discusses the time complexity of insert operations on a dynamic array.

What is it?

A dynamic array resizes itself automatically when it runs out of capacity and new elements need to be added to it. In Java, the ***ArrayList*** class is one such example. The official Java documentation states:

```
/**
* As elements are added to an ArrayList, its capacity grows automatically.
* The details of the growth policy are not specified beyond the fact that
* adding an element has constant amortized time cost.
*
*/
```

Note that a dynamic array only resizes itself to grow bigger, but not smaller. The claim is that the amortized cost of inserting n elements in a dynamic array is O(1) or constant time.

Implementation

We know that an array is a contiguous block of memory and can't be resized. Internally, a dynamic array implementation simply discards the previous allocation of memory and reallocates a bigger chunk. When the existing members are copied over to the new array and now new ones can also be added because of additional capacity.

One common strategy is to double the array size whenever it reaches full capacity. Below, we'll work out the complexity for this strategy.

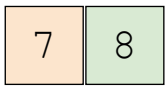
First Element

We'll trace how an array expands, starting with a size of 1 as new elements get added. The initial state of the array is shown below with one element.

Second Element Added

Now we add another element. Since the capacity is 1, we need to double it to 2 elements. We copy the previous element and insert the new one.

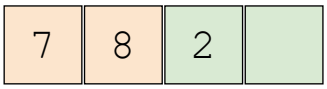
Cost = 1 copy + 1 new insert



Third Element Added

Since we don't have any more capacity, we'll need to double our array size again. Note that, at this point, we'll need to copy the previous two elements and add the new element.

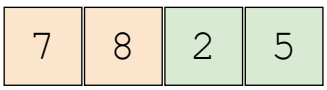
Cost = 2 copies + 1 new insert



Fourth Element Added

When we want to add the fourth element, there's one empty slot in our array where we can insert it without having to resize the array.

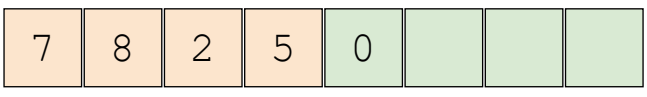
Cost = 1 new insert



Fifth Element Added

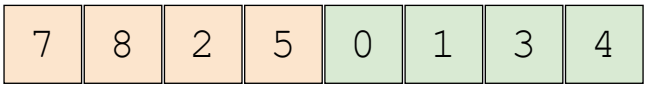
When we want to add the fifth element, there's no space in the array, so we need to double the array size from four to eight. We'll need to copy the four elements from the old array to the new array and insert the fifth element.

Cost: 4 copies + 1 new insert



Sixth, Seventh & Eight Element Added

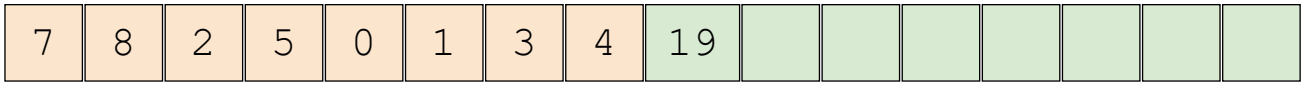
We have space for 3 more elements, so we can add the sixth, seventh and eighth element.
Cost for adding sixth, seventh and eighth element is just 3 new inserts.



Ninth Element Added

Adding the ninth element will cause us to double our array again, making the array the size of sixteen elements. Note that this time we'll need to copy the 8 elements from the older array to the new array, and also insert the ninth element.

Cost: 8 copies and 1 new insert



Adding Up the Costs

Let's take stock of our costs

Element #	Cost
1	1 insert
2	1 insert + 1 copy = 2
3	1 insert + 2 copies = 3
4	1 insert = 1
5	1 insert + 4 copies = 5
6	1 insert = 1

7	1 insert = 1
8	1 insert = 1
9	1 insert + 8 copies = 9 operations

You can probably see a pattern here. The highlighted rows incur additional copy operations. Coincidentally, they are also powers of 2 plus one.

$$2^n + 1$$

$$2^0 + 1 = 2,\ when\ n = 0$$

$$2^1 + 1 = 3,\ when\ n = 1$$

$$2^2 + 1 = 5,\ when\ n = 2$$

$$2^2 + 1 = 9,\ when\ n = 3$$

So now it becomes very easy for us to reason about the number of total operations that would take place when we add the ***(2^n+1)-th*** element to our dynamic array. It'll cause our array to double in size to ***(2^{n+1})*** slots.

For **$2^n + 1$** elements, the total operations would be

Total Operations = Total Inserts + Total Copies

Total Inserts

We know that we inserted **$2^n + 1$** elements.

Total Copies

In our example scenario:

- 1 copy when we doubled size from 1 to 2
- 2 copies when we doubled size from 2 to 4
- 4 copies when we doubled size from 4 to 8
- 8 copies when we doubled size from 8 to 16

Using the above information we can generalize that we will make that 2^n copies whenever we increase the size of the array from 2^n to 2^{n+1}. Now don't forget that our array was initially of size 0 and went through several doublings in size, so we need to sum up all the copy operations. When we added the 2^n + **1th** element, we would have performed the following copy operations starting all the way from the first doubling of the array.

$$1 + 2 + 4 ... 2^n$$

Total Cost

The total cost is just the total number of operations that we did to get the array to size 2^{n+1} when we added the (2^n+1)**-th** element.

$$Total\ Cost = Total\ Inserts + Total\ Copies$$

$$Total\ Cost = (2^n + 1) + (1 + 2 + 4 ... 2^n)$$

Rearranging the terms, we can write

$$Total\ Cost = (1) + (1 + 2 + 4 ... 2^{n-1} + 2^n)$$

The arithematic series $(1 + 2 + 4 ... 2^{n-1} + 2^n)$ is known to sum up to $2^{n+1} - 1$; for example:

$$1 + 2 = 3\ is\ equivalent\ to\ 2^{1+1} - 1 = 3$$

$$1 + 2 + 4 = 7\ is\ equivalent\ to\ 2^{2+1} - 1 = 7$$

$$1 + 2 + 4 + 8 = 15\ is\ equivalent\ to\ 2^{3+1} - 1 = 15$$

We can rewrite the total cost as:

$$Total\ Cost = 2^n + 1 + 2^{n+1} - 1$$

$$Total\ Cost = 2n + 2n + 1$$

$$Total\ Cost = 2^n + (2^n * 2)$$

$$Total\ Cost = 2^n(1 + 2)$$

$$Total\ Cost = 2^n * 3$$

Amortized Cost

To calculate the average or the amortized cost, all we need to do is to divide the total cost by the total number of elements added, similar to how we'll calculate average.

$$Amortized\ Cost = \frac{Total\ Cost}{Total\ Elements}$$

$$Amortized\ Cost = \frac{2^n * 3}{2^n + 1}$$

The equation $^{2^n}$ / $_{2^n + 1}$ would be roughly equal to 1 as n tends to infinity leaving the amoritzed cost as:

$$Amortized\ Cost = [\frac{2^n}{2^n + 1}] * 3$$

$$Amortized\ Cost = [1] * 3$$

$$Amortized\ Cost = 3$$

Hence we can claim that the amortized cost of insert operations is constant for a dynamic array.

Fancy Stack

In this chapter, we discuss the complexity of various operations of a stack which supports a multipop operation.

We are all aware of the push and pop operations offered by a stack data structure. Both push and pop are trivial and take O(1) or constant time. Now imagine the stack also offers a multipop operation, which allows us to pop k elements at once, assuming the stack holds atleast k elements. If we conduct n operations on the stack, what is the average time complexity of each of the n operations?

Naive Analysis

Consider a sequence of n pop, push, and multipop operations. The maximum number of elements that can accumulate in the stack is also n since the sequence can consist of all push operations. In the worst case, the multipop operation would take O(n) time to pop all the n elements of a stack. So, in general, the time complexity for execution of n operations in the worst case will be:

$$n * O(n) = O(n^2)$$

Since the n operations can consist of all multipop operations, each multipop takes O(n) time in the worst case. Though this analysis is correct, the bound isn't tight enough.

Aggregate Analysis

Intuitively, if at most n elements have been pushed onto the stack then it is possible to pop them back out only once. The multipop operation internally makes repeated calls to pop operation, but pop can only be called at most n times in a sequence of n operations. Thus the **total** cost of n operations can be n, at most. Note that we say there can be at most n pop operations. However,

since we start with an empty stack any legal sequence of n operations can consist of at most $n/2$ pop operations, as an equal number of push operations need to be executed to make the stack non-empty. Since each individual operation has cost O(1) whether it be push or pop, n operations will have cost O(n). The amortized cost for each operation then becomes:

$$AmortizedCost = AverageCost = \frac{O(n)}{n} = O(1)$$

Accounting Analysis

We'll use the accounting method to analyze the fancy stack. The actual costs for the three operations are:

Push : 1
Pop : 1
MultiPop : k - assuming stack has atleast k elements.

As a first step, we'll need to assign amortized costs to each stack operation. Let's choose the following amortized costs for the stack operations:

Push : 2
Pop : 0
MultiPop : 0

Now imagine that whenever we push an item on the stack, we pay 1 rupee as the cost for the push operation and we place an additional rupee as credit with the object we are pushing on the stack. When the time comes to pop the object, the cost will be paid by the extra rupee that we stored alongside the object when it was pushed onto the stack.

Now it's easy to reason that if we have a sequence of n pop, push, or multipop operations, the amortized cost will be **2*n = O(n)** in the worst case. The amortized cost is an upperbound on the actual cost, and we can thus declare that the cost of n operations will be O(n).

Changing Cost Assignment

A reader may wonder if the amortized cost assignments can be changed. For instance, will the following values work?

Push : 0
Pop : 2
MultiPop : 0

While undertaking accounting method of analysis, it is imperative that ***the total amortized cost for any sequence or subsequence of operations shouldn't be negative or less than the actual cost***. If the sequence only consists of push operations, then the total amortized cost will turn out to be less than the actual cost, thus we can't use the suggested assignments.

Q

If we introduced a multipush(k) operation for our fancy stack, how would that affect the cost of stack operations?

Check Answers

Problem Set 5

Questions to practice amortized analysis.

Question 1

A sequence of n operations are performed on a data-structure. The ith operation costs i if i is an exact power of k for some fixed integer $k \geq 2$ and 1 otherwise.

- Use aggregate analysis to determine amortized cost per operation.

Question 2

Redo the analysis of the dynamic array using accouting method.

Solution Set 5

Solutions to problem set 4

Solution 1

The question asks us to work out the amortized cost of each operation for a total of n operations. An operation costs i if it is a power of k and costs 1 if it isn't a power of k.

For simplicity let's assume $k = 2$ and the total number of operations is a power of k. Let the total number of operations be 8. The operations and their cost is shown below.

Operation Number	1	2	3	4	5	6	7	8
Cost	1	2	1	4	1	1	1	8

In the above table, operation# 2, 4 and 8 are powers of $k = 2$ and are shown in blue. We need to calculate the total cost of all the operations and then divide it by n to get the amortized cost per operation.

We'll to compute two costs to get the total cost of n operations. One operations which are a power of $k = 2$ and two which aren't a power of k.

Operations power of k

We'll ignore the first operations as a power of k even though $k^0=1$. The cost in our example is thus $2 + 4 + 8 = 14$. The first question we need to answer is that given n operations, how many of those operations will be powers of k. The

answer is k raised to what power would equal n? And as we have seen earlier,

it is $\boldsymbol{log_k n}$. Let's plug in the values for our example and see if the answer matches what we expect.

$$log_2 8 = 3$$

This is exactly what we see in our example, 2, 4 and 8 appear as powers of ***k*** = ***2***. Note, if we were counting the first operation as a power of 2 then we would have $\log_k n + 1$ powers of ***k*** occurring in ***n*** operations.

Next, we need to find out what these operations would sum to.You can deduce by inspection or if you are mathematically adept would already know that the sum of the first ***x*** powers of ***k*** is $\boldsymbol{k^{x+1} - 1}$. We can test it on our example.

$$2^{3+1} - 1 = 2^4 - 1 = 15 = 1 + 2 + 4 + 8$$

As you can see that this includes 1 as a power of ***k*** so we will need subtract an additional 1 for our example.

So now we can calculate the sum of all the operations in ***n*** operations that are powers of ***k***. There will be $\boldsymbol{log_k n}$ such operations and their sum is represented by

$$k^{(number\ of\ operations\ that\ are\ power\ of\ k+1)} - 1 - 1$$

$$= k^{(log_k n+1)} - 2$$

Operations not power of k

We have already calculated the number of operations that are power of ***k*** among the total ***n*** operations, which is $\boldsymbol{log_k n}$ so the number of operations not power of ***k*** are $\boldsymbol{n - log_k n}$.

Total Cost

The total cost is thus the sum of costs of operations that are power of k and those that are not power of k.

> $$Total\ Cost = Cost\ of\ operations\ power\ of\ k + Cost\ of\ operations\ not\ power\ of\ k$$
>
> $$Total\ Cost = (k^{(log_k n+1)} - 2) + 1 * (n - log_k n)$$
>
> $$Total\ Cost = ((k^{log_k n} * k) - 2) + (n - log_k n)$$
>
> $$Total\ Cost = ((n * k) - 2) + (n - log_k n)$$
>
> $$Total\ Cost = nk - 2 + n - log_k n$$

We can plug in values for ***n*** and ***k*** from our example to verify that our expression correctly computes the total cost.

> $$Total\ Cost = nk - 2 + n - log_k n$$
>
> $$Total\ Cost = 8 * 2 - 2 + 8 - log_2 8$$
>
> $$Total\ Cost = 16 - 2 + 8 - 3$$
>
> $$Total\ Cost = 19$$

Amortized Cost

To get the amortized cost we'll simply divide the total cost by the total number of operations which is ***n***.

> $$Amortized\ Cost = \frac{nk - 2 + n - log_k n}{n}$$
>
> $$Amortized\ Cost = \frac{nk}{n} + \frac{n}{n} - \frac{log_k n}{n} - \frac{2}{n}$$
>
> $$Amortized\ Cost = k + 1 - \frac{log_k n}{n} - \frac{2}{n}$$

As ***n*** becomes larger and larger $^2/_n$ becomes 0. Similarly as ***n*** increases the term $^{\log_k n}/_n$ also tends to zero because the denominators increases faster than

term $^{\log k}/_{n}$ also tends to zero because the denominators increases faster than

the numerator. Finally, we can ignore the constant term 1 and are left with only ***k***. Hence the amortized cost of each operation is thus ***O(k)***.

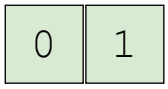
Solution 2

We need to work out the amortized cost of the dynamic array's insert operation using accounting method. We need to be mindful of the following costs

- Cost to insert the element the first time in the array
- Cost to copy the element whenever the array is doubled

The first cost is simple to account for. We can start with a cost of 1 to insert an element in the array. One may falsely argue that we can add an additional unit of cost to account for the copying on the doubling of the array for a total cost of 2. Let's see how that is false but gives us insight to reach the correct solution.

After adding the first element the array has a credit of 1 unit



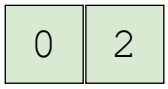
Adding the second element, causes a doubling and the credit of 1 unit is utilized to copy the first element

The problem is apparent now, on adding the third element the dynamic array would double and the first element has no credit to copy itself over to the new array. The key here is to realize that whenever a doubling happens, the elements added after the initial array has doubled need to bring in credit to copy all the elements that existed before the doubling.

Let's take an example, say the array is of size two but only a single element has been added. When we add the second element, we assign a cost of 3 units to the insert operation. One unit is paid for the insert, one unit is left to copy

the second element itself when the array doubles next time and the third unit

is there to cover for charge to copy the first element. Below is the pictorial representation.

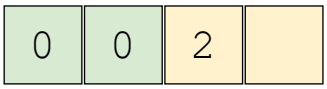


The second element holds a credit of two

Now when the third element gets inserted the array gets doubled.

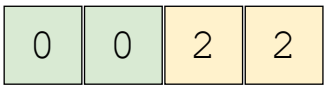
The credit is utilized to copy over the first two elements to the new array

Now, the third element is added with a credit of 3, one of which is utilized in inserting the third element itself, leaving us with 2 units of credit.



Third element with a credit of 2

Now when the fourth element gets added, it brings a credit of 2 units too.



Fourth element also comes in with a credit of 2

Now we have a total credit of four in the data-structure and four elements in all. Now when we double the array, we'll have enough credit to cover for the charge to copy the four elements to the new array.

Credit is consumed in copying the old half of the array when a doubling occurs

Now one can observe that when the fifth element is added, it'll bring with it

Now one can observe that when the fifth element is added, it'll bring with it the credit to copy the first element when a doubling is required. Similarly, the

sixth element brings credit for copying the second element so on and so forth. Essentially, the new half of the array brings with it the credit to copy the old half of the array when the next doubling is required.

Hence the amortized cost of insertion is 3 units or constant.

Quantifying Chance & Randomness

This chapter discusses pre-requisite concepts for undertaking probabilistic analysis.

Before we start this section, I want to take the opportunity to caution readers that it is highly unlikely for both this and the next section to appear in your coding interviews. These sections are for enthusiasts and folks eager to have more of the flavor of the complexity theory. If your intent is preparation for interviews, then the previous sections should be most helpful. That being said, you are more than welcome to continue reading as knowing more is never a disadvantage :)

Probability

This section is a very brief introduction of probability. It's just enough to get the reader up to speed on the knowledge required to understand randomized algorithms and probabilistic analysis. Probability is defined as ***the likelihood of something happening***. The mathematical study of probability describes the ways in which one can quantify the chances of an event taking place.

We start with the text book example of flipping a coin. When we flip a coin, it can land either on its head or its tail. Imagine the coin is unbiased and is equally likely to land either heads or tails. If we want to know what the chances are of the coin landing on its head, *we will count the outcomes that result in heads and divide the count by the total number of possible outcomes*. In the described scenario, there are only two outcomes possible; we get either heads or tails on a single flip. The list of all possible outcomes is called the sample space, denoted by S.

> $$S = H, T$$

The sample space S consists of all the possible outcomes of the experiment we are interested in. Note that the events in the sample space are mutually

exclusive, meaning that if the flip results in heads then it can't also be tails. If today is Friday then it can't be Sunday at the same time. The days of the week are mutually exclusive.

Note that at least one of the events in the sample space must happen, i.e. the probability

$$P(S) = 1$$

If you flip a coin then either heads or tails must show up.

Q

If we flip our fair coin twice, what is the probability of getting atleast two heads ?

Check Answers

Random Variables

This chapter discusses random variables.

What are Random Variables?

In probability theory, we use *random variables* to represent the outcomes of random processes. Don't confuse them with variables you define in your code or the variables you read about in algebra class. These variables are traditionally denoted by upper case letters. We call them random because they represent the outcomes of random processes such as flipping a coin, throwing dice or the possibility of rain on a given day. As an example, we define two random variables below:

$$X = Number\ of\ heads\ in\ 3\ flips\ of\ a\ coin$$

$$Y = Faceup\ value\ of\ a\ dice\ greater\ than\ 2\ on\ a\ roll\ of\ dice$$

Note that in both the above cases X and Y can take on different values depending on the outcome of the experiment.

What are Discrete Random Variables

The two random variables we discussed are examples of *discrete* random variables. Discrete means *distinct* or *separate* values. We call the discussed variables discrete because these variables can take on a *countable* number of values. Sometimes the list of countable values can be infinite, but the random variables will be discrete for as long as you can count them (even if that means counting forever).

The other type of random variables are continuous ones. For instance, the weight of your friend can be between 10lbs and 500lbs. However, if you pick any two values in that range, there is an infinite number of values between the two picked values. If you pick 120lbs and 121lbs, there is an infinite number of values between those two numbers that a random variable can take on e.g. 120.01, 120.001, 12.0001 so and so forth. The litmus test to

determine if a random variable is continuous is to pick two values and verify if an infinite number of values exist between them. If you can't list or count all the possible values then you are likely dealing with a continuous random variable. In the case of a discrete random variable, you can count a finite number of values between your two chosen values.

In the study of algorithms, we will only concern ourselves with discrete random variables and their distributions, but its important to understand the distinction between the two types.

What are Indicator Random Variables

The indicator variable for an event A is a variable having value 1 if the A happens, and 0 otherwise. The number of times something happens can be written as a sum of indicator variables. For example, we may define I_6 as an indicator variable when 6 appears face-up on a roll of a fair die. If we roll the die three times, we can express the number of times 6 appears on the die as:

$$I = I_{6_1} + I_{6_2} + I_{6_3}$$

We could have also declared a random variable representing the number of times 6 appears in three rolls of a die. However, sometimes it is easier to reason about problems using indicator random variables than complicated random variables.

Probability Distribution

This chapter discusses the concept of probability distribution.

Probability Distribution

When working with random variables, we can ask questions like "what is the probability that X equals 2 heads in three coin flips, or that Y equals a value greater than 2 on a die roll?". We'll work these example scenarios below.

Example 1

If we flip a coin three times, the sample space will look something like below:

$$S = HHH, HHT, HTH, HTT, THH, TTH, THT, TTT$$

Now what is P(X=2)? In other words, what is the probability that X (the number of heads in three coin flips) is exactly equal to two? The outcomes that satisfy exactly two heads include: HHT, HTH and THH. Therefore we can say

$$P(X = 2) = \frac{number\ of\ occurrences\ of\ desired\ event}{total\ occurrences}$$

$$P(X = 2) = \frac{3}{8}$$

Example 2

Now let's try to work out the dice example. If we roll a die, it can come face up with the following values:

$$S = 1, 2, 3, 4, 5, 6$$

We defined the random variable Y as the value that shows face-up on the die. The outcomes satisfying Y > 2 include 3, 4, 5, and 6.

For a fair die, the probability of P(Y= any value in S) is ⅙ since any of the six values are equally likely to show face-up. However, if we ask what the probability is of Y > 2 happening, it would be equivalent of asking,

$$P(Y = 3 \; or \; 4 \; or \; 5 \; or \; 6) = ? = \frac{4}{6} = \frac{2}{3}$$

We saw how to calculate the probabilities of random variables taking on different values. When we examine probabilities for the possible outcomes of a given random variable, we are working with *probability distribution* for that random variable. Let's go back to our examples.

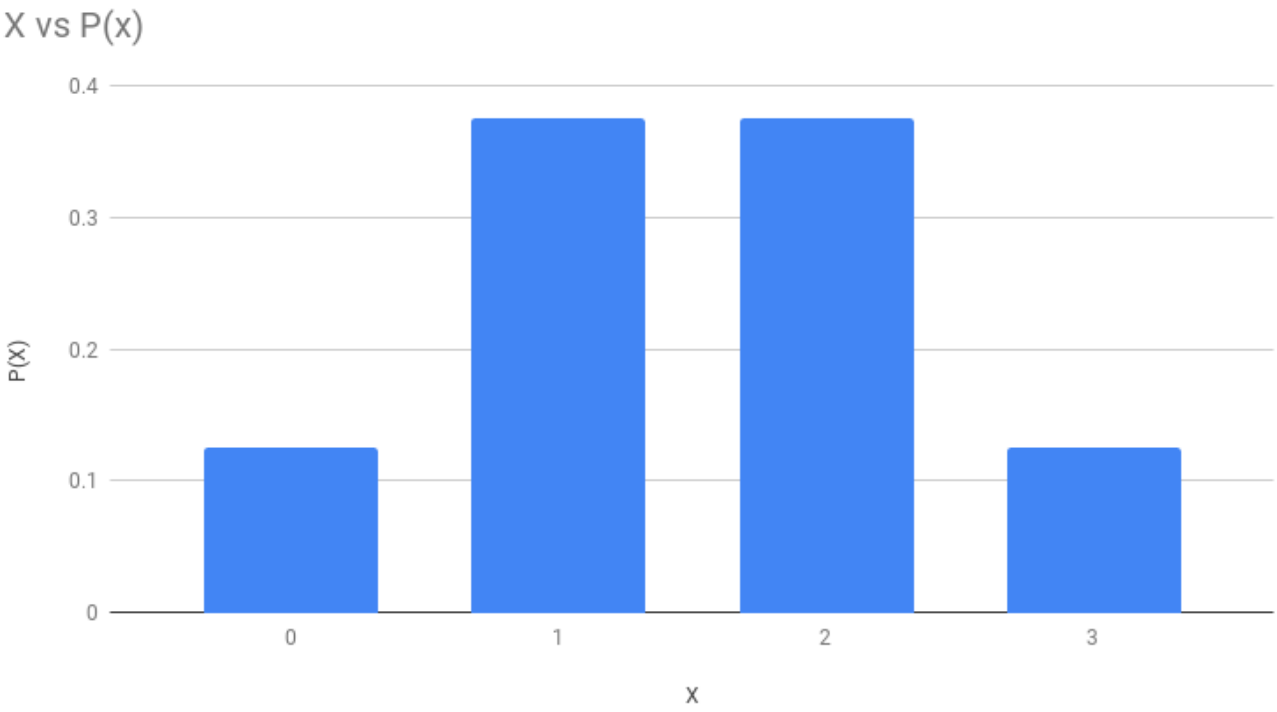
In case of the coin flip, we introduced the random variable X, which represented the number of times head shows up in 3 flips of a coin. What are various values that X can take on? Heads can show up 0 times, 1 time, 2 times, or 3 times in a 3-flip coin experiment. The distribution of probabilities for these various values that X can take on will constitute the *probability distribution* for X.

Values X can take on	Probability of X taking on value in first column
0	{ TTT }⅛ = ⅛
1	{ HTT THT TTH }⅛ = ⅜
2	{ HHT HTH THH }⅛ = ⅜
3	{ HHH }⅛ = ⅛

You can see from the table that X takes on various values with different

probabilities. The probability varies depending on how often the desired outcome occurs in the entire sample space. In our example, head occurring exactly once or twice happens more often than head not occurring at all, thus the corresponding probabilities are also higher.

If we plot the above table as below, it'll represent the probability distribution for the random variable X.

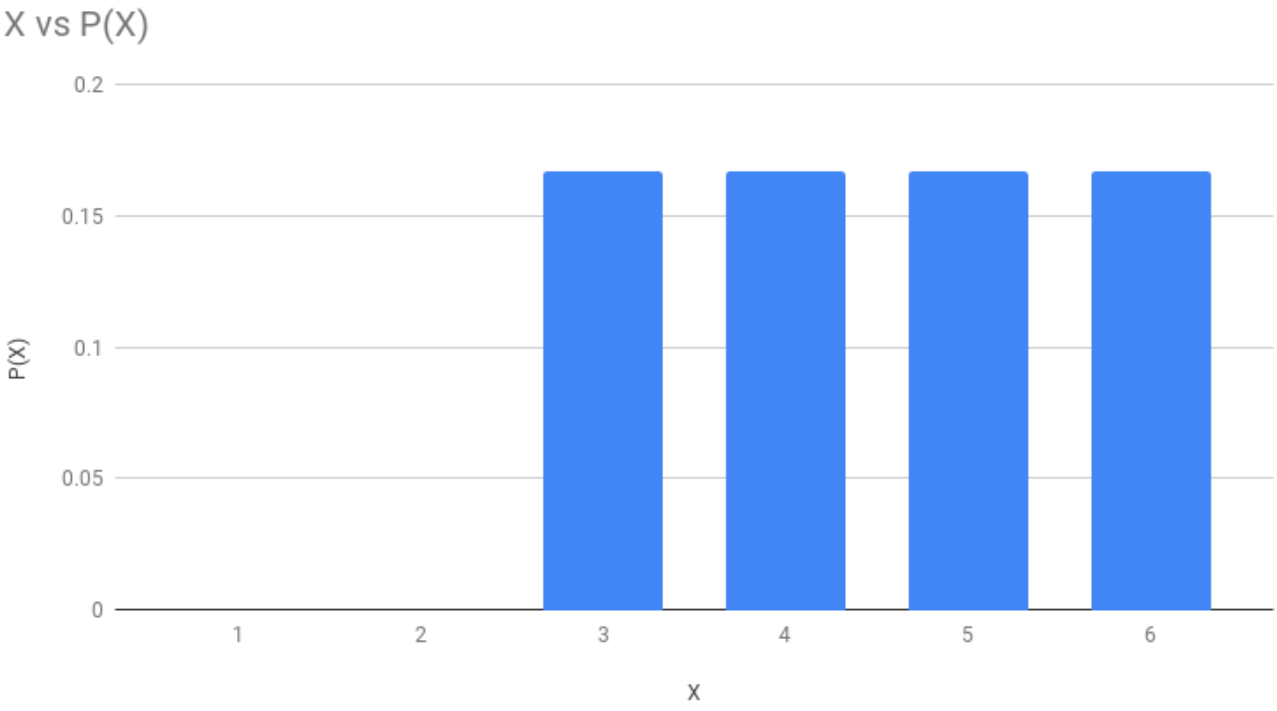


Now we'll contrast the coin flip example with the die example. The random variable Y was defined as the die showing a value greater than 2 on a roll.

Values Y can take on	Probability of Y taking on value in first column
1	0
2	0
3	{3}⁄6 = 1⁄6
4	{4}⁄6 = 1⁄6

5	$\{5\}/6 = 1/6$
6	$\{6\}/6 = 1/6$

Note that we have listed probabilities for 1 and 2 as 0 since, by definition, Y can't take on those values. In contrast to X, one can see that Y has the same probability for the different values it can legally take on. This is an example of a *uniform probability distribution*.



Expected Value of a Random Variable

This chapter discusses the expected values of random variables and how to calculate them.

Expected Value of a random variable

We all understand the concept of average. The average test-score of a class is the sum of each individual student's score divided by the total number of students in the class. The *expected* value of a random variable is somewhat a similar concept. However, note that the outcomes that a random variable can take on don't happen with the same frequency. An outcome that happens more frequently should get a higher weight when computing the "average" for a random variable. The weight here is the probability with which each outcome occurs.

Take the case of the random variable X which we define as the number of heads that occur in 3 flips of a coin. The expected number of heads seen when the experiment of flipping a coin thrice is repeated many many times is shown below:

$$E[X] = P(X = 0) * 0 + P(X = 1) * 1 + P(X = 2) * 2 + P(X = 3) *$$

$$E[X] = \frac{1}{8} * 0 + \frac{3}{8} * 1 + \frac{3}{8} * 2 + \frac{1}{8} * 3$$

$$E[X] = 0 + \frac{3}{8} + \frac{6}{8} + \frac{3}{8}$$

$$E[X] = \frac{12}{8}$$

$$E[X] = 1.5$$

Thus we can make a claim: if we do several experiments of flipping a coin three times and noting down the results from each experiment, we can expect to see heads appearing 1.5 times across all the experiments.

The Tinder Problem

This lesson discusses a sample problem and works out the complexity using probabilistic analysis.

Meet Neha. She's a millennial and an avid Tinder user. She swipes Tinder daily and dumps her latest boyfriend if she finds a better match than her current boyfriend (don't judge). In her world, there's never a tie between two potential boyfriends (i.e. she can always decide that one is better than the other). However, dating isn't easy or cheap. When Neha finds a mutual match, she'll go out on a date with the new guy and pay for it herself - which usually costs her about $30. If she finds she prefers the new match she'll immediately break-up with her current boyfriend over text while also sending him a $100 gift card at the same time to smooth out any grudges.

There are two costs in Neha's approach: one when she goes out on a date, and one when she breaks-up with someone. Let's see what the costs look like in the worst and best cases over ten consecutive mutual matches.

In the best case, she finds the best boyfriend on the first match and only incurs the cost of dating the remaining 9 matches without breaking up with

her first boyfriend. This brings her total to $30 * 10 = $300.

In the worst case, each new match that she dates turns out to be better than her current boyfriend. Over the course of 10 matches, she will have broken up with 9 guys and dated all 10 of them. Her cost, in the worst case, comes out to be $300 + $900 = $1200 ! This is much worse than her best case cost, or four-fold of that former amount.

It is unlikely that she'll hit the best or the worst case, although she can. The question is, what would be her average or expected cost since the matches she receives are guaranteed to be in a random order?

Solution

We can rank each of the 10 matches using an integer from 1 to 10. A match represented by a higher integer is considered to be better than the matches represented by the lower integers. We can simply use an array of 10 integers, representing the 10 matches for Neha, as input to the algorithm that outputs her final cost. Note that the array can have 10! permutations, and each permutation can represent a particular order in which she encounters her 10 matches and dates them. Below is the algorithm followed by the expected cost analysis.

Let's start with calculating the probability that she prefers the 5th match more than the previous four? It's simply $\frac{1}{5}$. Generalizing, when she dates the *kth*

than the previous four? It's simply $\frac{1}{5}$. Generalizing, when she dates the *kth* match, the probability is $\frac{1}{k}$ that she'll prefer the ***kth*** match more than the previous ***k-1*** matches and breaks-up to be with the ***kth*** match. We can now model the possibility that Neha prefers the ***kth*** match more than the previous ***k-1*** matches as an indicator random variable:

> ***I_k = 1 if kth match is best so far or,***
> ***= 0 if a better match existed in previous k-1 dates***

If we want to know how many times is Neha *expected* to break-up, It's simply a summation of all the indicator variables for each of k matches.

$$X = X_1 + X_2 + X_3 \cdots X_k$$

The above equation is essentially counting the number of times Neha starts a relationship when she goes through k tinder matches. We can now apply expectation on both sides of the equation:

$$E[X] = E[X_1 + X_2 + X_3 \cdots X_k]$$

$$E[X] = E[X_1] + E[X_2] + E[X_3] \cdots E[X_k]$$

Expected value of a random indicator variable is just the probability of the event it represents happening.

$$E[X] = 1 + \frac{1}{2} + \frac{1}{3} \cdots \frac{1}{k}$$

Most people would not know the summation of the above series off the top of their heads, but if one can solve to this point, one has learned complexity analysis well. The above series mathematically sums to:

$$E[X] = lnk$$

$$E[X] = lnk$$

Hence we can say that Neha is expected to break-up with ***lnk*** number of guys, so her expected cost over dating ***k*** tinder matches can be represented as:

$$T(k) = 30 * k + 100lnk$$

That is roughly 530, much better than our expected worst case cost of 1200!

Why should I bother?

This chapter gives an introduction to complexity theory.

In the previous sections, we studied problems which didn't require brute force search to find solutions. However, there are problems whose solutions can't be found without examining the entire possible search space. Take the case of prime factorization, for example. Below are the prime factors for 21:

$$7 * 3 = 21$$

It is easy to verify the solution; we just multiply 7 and 3, and check if the product equals 21. However, when we want to do the reverse, i.e. find the prime factors for 21, we need to go through trial division starting from 1, till we finally find the prime factors.

$$? * ? = 21$$

For smaller numbers like 21 a brute force algorithm would finish in reasonable time, but what about really large numbers like the one below?

$$? * ? = 213423567,890182215,329123767,110140235$$

A brute force algorithm would not be able to come up with prime factors for the above number in a reasonable amount of time, especially with today's computer technology. You may wonder why this matters or is even important. Well, enter cryptography! Encryption algorithms make use of the fact that it is easy to multiply two very large prime numbers, but un-assembling the resulting product back into its constituent prime numbers isn't easy. If an easy

resulting product back into its constituent prime numbers isn't easy. If an easy solution was found for prime factorization, then a lot of the heavily relied upon encryption algorithms used in the industry would fall apart.

The consequences aren't just limited to cryptography. Thus far, even while using existing algorithm design techniques, humans have been unable to find efficient solutions to many problems. If efficient solutions to these problems were found, it would result in major advancements in fields as far apart as medical science and AI.

Complexity theorists define various classes of complexity depending on how hard a problem is to solve. We'll briefly describe the important classes in the upcoming sections. However, before we embark upon our study of complexity classes, we'll discuss *decision problems* which help us in defining some of the complexity classes.

Decision Problems

A decidable problem is one for which we can answer yes or no. Algorithms aren't decidable problems. For instance, the breadth first search algorithm isn't decidable in any sense. Complexity classes that we'll discuss for the purposes of this course only include decidable problems. Fortunately, we can cast algorithms as decidable problems in order to reason about their complexity. For instance, for depth first search, we can ask: can all the nodes of a graph (with no cycles) be traversed by visiting at most each node once? We can answer this question by running DFS, and keeping a count of visited nodes and answering yes or no at the end.

Optimization Problems

Optimization problems are interesting problems from the perspective of complexity theory. For instance, find the shortest distance between two vertices in a given graph. There may be multiple paths between the two vertices, but the optimum one needs to be the shortest. Also, there could be multiple shortest paths. We can transform the shortest path question into a decision problem by asking if there is a path between two vertices of a graph consisting of less than k edges?

If we can answer the decision version of an optimization problem, we can also answer the optimization problem itself. Consider starting with an arbitrary

answer the optimization problem itself. Consider starting with an arbitrary value of k edges. The decision version of the problem will spit out either a yes or a no, and (depending on the answer) we can tweak the value of k and recursively apply the binary search to zoom in on the optimum answer. Essentially, we are wrapping our decision-problem solution with binary search to get to the optimized solution.

P and NP classes

In this lesson, we discuss the two most important complexity classes P and NP.

P - Polynomial Time Problems

The *P* stands for *polynomial time*. **Problems which can be solved in n^k, where k is some constant from the set of natural numbers (1, 2, 3 ...), lie in the complexity class P. More simply, problems to which we can find solutions in polynomial time belong to the complexity class P**. Note that when we say *problems* we are talking about decision problems. All the problems in P are decision problems. Strictly speaking, algorithms such as sorting, binary search, and tree-travels are algorithms that can be used to solve decision problems. However, we can always *find* a problem that a given algorithm will solve. For instance, you may pose a decision problem if an array is sorted. You can either scan over it and verify elements appear in ascending order, or you may just run the fastest sorting algorithm on a copy of the array and compare the result with the given array.

- Multiplication can be solved in polynomial time. The decision version of the problem will be if we multiply x and y, is the result z? We will simply multiply x and y and compare the product with z.
- Finding the greatest common divisor is also a problem in the P class.
- Is a given string S a palindrome? This decision problem can be solved in polynomial time.
- Given two strings *p* and *q*, is p a substring of q?

Problems solved on whiteboards during interviews can be either algorithms or decision problems. Sorting, binary search, tree traversals are all polynomial time algorithms but not decision problems. Whenever talking about problems in complexity classes, we are referring to decision problems. So if you state that selection sort is in complexity class P, your statement isn't

valid. However, the algorithm can be run in polynomial time to answer a

decision problem that insertion sort solves, and that decision problem would be considered to be in P.

NP - Non-deterministic Problems

NP stands for nondeterministic polynomial time. Go back to the factorization example at the start of the section and note it's hard to come up with prime factors for a given number. That being said, if the prime factors are already given to you, then its very easy to verify if the solution is correct. You simply multiply the two factors and verify if the product equals the given number. **The class of NP problems consists of those decision problems whose solution can be verified in polynomial time. The problems themselves may or may not be solvable in polynomial time.** We don't care about how we get the solution, but once we have the solution, we can prove it is correct in polynomial time. A few examples appear below:

- **Graph coloring problem**: Given an undirected graph, can we color nodes using only three colors in such a way that no two connected nodes have the same color? Note given a coloring, we can just run a graph traversal algorithm to verify that the solution is correct.
- **Subset sum problem**: Given a set of integers and a target integer, can we choose any combination of integers from the set in such a way that their sum equals the target? Note to answer this question, we'll need to exhaustively search the entire problem space, i.e. find all combination and see if the sum of any one of them equals the target. On the contrary, if given a combination, its very easy to *verify* in polynomial time if the given combination's sum equals target.
- **String Matching**: Don't forget that string matching is in P as well as in NP. We can verify if a string p is a substring of another string q in polynomial time by simply running any string matching algorithm and verifying the claim.

For the first two examples we don't know of a polynomial time solution that will solve either of the problems. However, that doesn't mean that no such algorithm exists. Theorists have not been able to prove otherwise either. Just

algorithm exists. Theorists have not been able to prove otherwise either. Just because no one has been able to come up with efficient solutions to these problems isn't proof enough that no polynomial algorithm exists that will solve these problems. A formal proof that a solution does or doesn't exist has been lacking thus far. It is our inability to provide such a proof that makes the study of complexity intriguing.

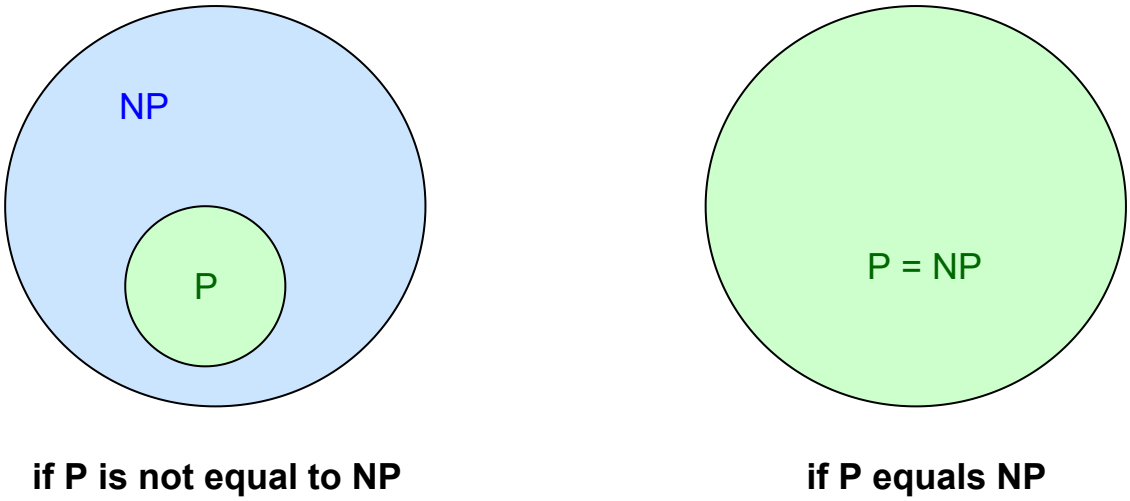
Get Rich with Complexity

In this lesson, we discuss the famous is P=NP dilemma.

P equals NP or P doesn't equal NP?

Since we can solve problems in P in polynomial time, we can also verify them in polynomial time. Thus we can say ***every problem in P is also in NP***. Note we haven't made the converse claim whether problems in **NP** are in **P** or not. Or said another way, is **P** a proper or improper subset of **NP**? We'll come back to that later.

One of the most important open questions in theoretical computer science is whether **P=NP**. It's so important and famous that Clay Institute includes it as one of the millennium problems, offering $1 million dollars in prize money for a solution! If **P** equals **NP** then the two classes would collapse into one. A pictorial representation is shown below



P vs NP

It's very important to understand that just because we can't prove **P=NP**, it doesn't imply that **P≠NP**. We require a proof to conclusively declare one or the other. Most theorists, however, believe that **P≠NP**. If it were proved **P=NP**, then it would mean that all problems that we can't solve today in polynomial time will have efficient solutions.

NP-Complete and NP-Hard

In this chapter we further the discussion on complexity theory with NP-complete and NP-hard complexity classes.

NP Complete

Without getting into the mathematical details and jargon, we'll stick to informally defining the complexity class NP-complete. Problems in the NP-complete class are **those problems in NP that are *as hard as* any other problems in NP**. This may sound confusing but, before we get a chance to define *NP-hard* problems, one can safely assume that NP-complete problems are the hardest problems to solve in the complexity class NP.

By definition of NP, solutions for NP-complete problems can be verified in polynomial time. The wonderful property about problems in NP-complete class is that if we find a polynomial solution for any one of them, we will have found a solution for all of the NP-complete problems - thus implying P=NP. Take a minute to think through what we just stated. It is a very powerful assertion, claiming that the solution for one problem can unlock solutions to all other problems in NP-complete!

Conversely, if we can prove that no polynomial solution exists for any one of the problems in NP-complete complexity class, then we can say that no polynomial time solution exists for all of the problems in NP-complete - thus implying P≠NP.

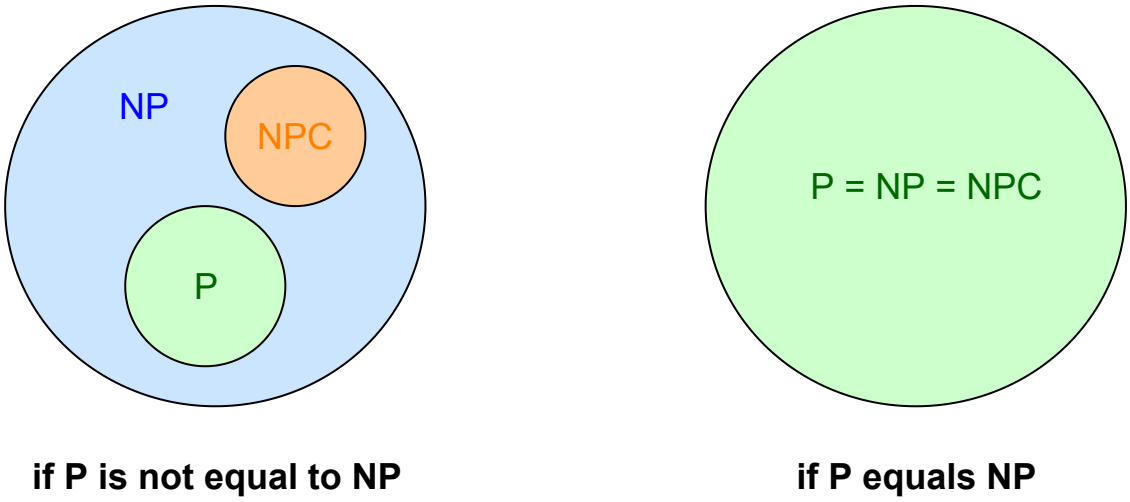
NP-complete problems exhibit two properties:

1. They are in NP, i.e. given a solution one can quickly (in polynomial time) verify that the solution is correct.
2. Every problem in NP can be converted or reduced to a problem in NP-complete. Note that since all NP-complete problems are also in NP, by definition NP-complete problems can be reduced to each other also.

The second property allows us to say that if we can solve a problem ***A*** quickly, and another problem ***B*** can be transformed into problem ***A***, then we can also solve ***B*** quickly. The caveat here is that the transformation itself should take polynomial time. If the transformation takes superpolynomial time then it defeats the purpose of transforming in the first place.

Both the graph coloring and subset sum problems are NP-complete, but string-matching isn't. Other notable NP-complete problems include:

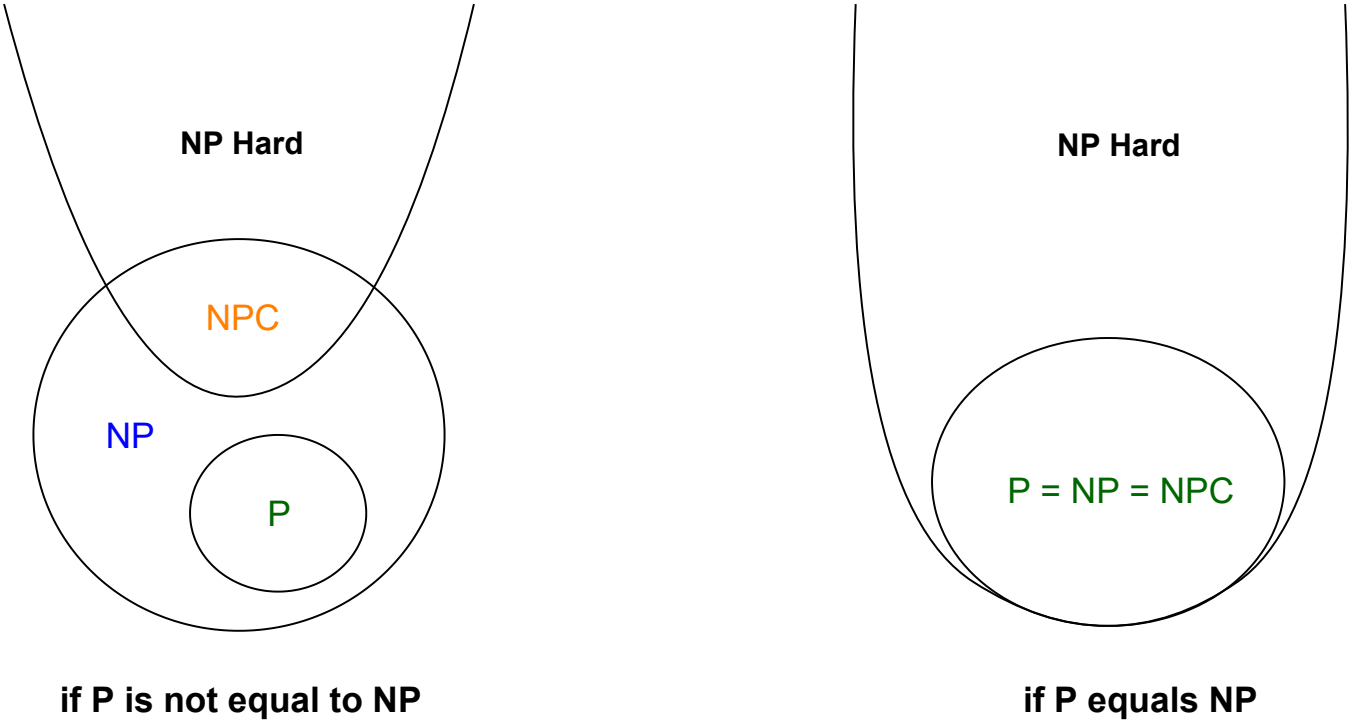
- Clique problem: A clique consists of a subset of those vertices in a graph, where each vertex is connected to every other vertex within the set. The goal is to find the maximum clique in a graph.
- Hamiltonian path problem: Given an undirected graph is there a path in the graph that visits each vertex exactly once?
- Sudoku: Solving a sudoku puzzle isn't polynomial time activity, whereas if given a solution to a Sudoku puzzle, the correctness can be verified in polynomial time.



P vs NP

NP hard

NP-hard problems are those that may not necessarily have a solution, verified for correctness in polynomial time. However, all problems in NP can be reduced or transformed into problems that are NP-hard. **NP-complete problems are in fact NP-hard and NP**. *Not all NP-hard problems are NP-complete though.* The pictorial representation appears below:



P vs NP

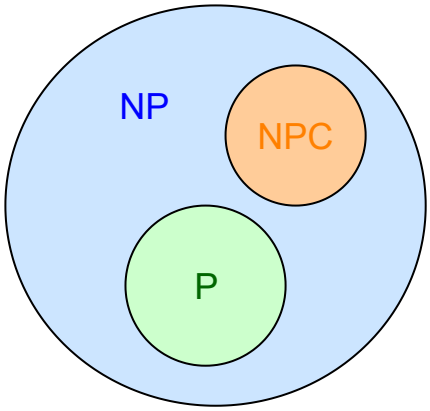
All problems in NP are decision problems with a yes or no outcome. Problems that are NP-hard but not NP-complete are those whose solutions can't be verified in polynomial time, let alone get solved in polynomial time. If you remember, we cast optimization problems into their decision versions so that their solutions can be verified in polynomial time. Optimization problems such as below are examples of NP-hard problems that are not NP-complete.

- Find all the max cliques in a given graph. Even if I give you a set of cliques and claim they represent all the max cliques in the graph, you can't verify in polynomial time if my claim is true or not. To verify, you need to yourself find all the max cliques and compare it with the ones I provide.
- Think about chess. If I tell you that a given move is the best or winning move, you have no way to verify my claim in polynomial time. You'll need to go through all the game trees and verify that what I claim is indeed true. The number of possible game trees is exponential.
- Minimum Vertex Cover: Find the minimum set of vertices in a graph such that every edge in the graph has at least one vertice from the set as an endpoint. This is an optimization problem. Even if provided a solution,

you may verify if it provides a cover but can't verify in polynomial time if it provides the minimum cover.

Between P and NP ?

Do any problems exist that aren't NP-complete and not in P?



Assume P is not equal to NP

NP-Intermediate

Given the above diagrams and assuming P≠NP, one may wonder if there are any problems that are in NP but not in NPC or P? Note that we have assumed **P≠NP**. Problems that exist in this class are called ***NP-intermediate***. Again, very carefully note that NP-intermediate exists only if P≠NP! For the interested, *Ladner's theorem* proves that if P≠NP then NP-intermediate is non-empty.

Prime Factorization

Prime factorization is ***suspected*** to be NP-intermediate problem. Note the use of the word suspected! No one has been able to come up with a polynomial time algorithm to find prime factors for a given integer, but that doesn't mean none exist! The converse hasn't been proven either, that no polynomial time algorithm exists for prime factorization of integers. Moreover, no one has been able to reduce it to a NP-complete problem, and thus prime factorization can't be placed in the NP-complete complexity class. As a result, integer factorization is thought to likely be an NP-intermediate problem.

Beyond NP-Hard

There are several other complexity classes based on different criterion; e.g. PSPACE, 2-EXPTIME, PCP and a bunch more are listed here. However, as software engineers in the industry, knowledge of P, NP and NP-complete problems is sufficient for most purposes. It helps to recognize when a problem may not yield to any polynomial algorithm design technique. Likewise, even informally proving the problem to be NP-complete helps direct energies towards an approximation algorithm or a heuristic rather than mindlessly banging one's head against the wall for a polynomial solution. Some of the problems that one can see as professional software engineers and are NP-complete include

- Scheduling problems, e.g. doctors' duty roster at a hospital. These problems usually involve combinatorial complexity.
- Google Maps generating best route for multiple destinations. Read more here.
- Development of compilers and languages involves questions like if a grammar can generate an ambiguous language, which can't be solved.

Advancements are always afoot in the research community to better our understanding of computational complexity of algorithms. As a recent example, determining if a given integer is prime was thought to be an NP-intermediate decision problem until 2012. At that time, three Indian researchers were able to show that there did indeed exist a polynomial time algorithm for primality test, and the problem was moved to the P complexity class.

Problem Set 6

Questions to test understanding of P/NP problems.

1. Are all decision problems in the complexity class NP?

2. Can optimization problems be in NP?

3. Travelling salesman problem is defined as a salesman planning to visit a bunch of cities to sell his merchandise. The salesman wants to cut costs and needs to follow a route which allows him to visit every city but at the same time is the shortest route among all possible routes. Is this problem in NP-complete?

4

If P=NP is proven, will integer factorization be still in complexity class NP-intermediate?

5

NP-complete problems can’t be solved at all?

Check Answers

Cheat Sheet

This is a compilation of worst-case complexities for various data-structures and algorithms.

Data-Structures

<table>
<tr><th>Data Structure</th><th colspan="2">Worst Case Complexity</th><th>Notes</th></tr>
<tr><td>Array</td><td colspan="2"><table><tr><td>Insert</td><td>O(1)</td></tr><tr><td>Retrieve</td><td>O(1)</td></tr></table></td><td rowspan="2">Note that if new elements are added at the head of the linkedlist then insert becomes a O(1) operation.</td></tr>
<tr><td>Linked List</td><td colspan="2"><table><tr><td>Insert at Tail</td><td>O(n)</td></tr><tr><td>Insert at Head</td><td>O(1)</td></tr><tr><td>Retrieve</td><td>O(n)</td></tr></table></td></tr>
<tr><td>Binary Tree</td><td colspan="2"><table><tr><td>Insert</td><td>O(n)</td></tr><tr><td>Retrieve</td><td>O(n)</td></tr></table></td><td>In worst case, the</td></tr>
</table>

	Insert	*O(n)*	binary tree becomes a linked-list.
	Retrieve	*O(n)*	
Dynamic Array	Insert	*O(1)*	Note by retrieving it is implied we are retrieving from a specific index of the array.
	Retrieve	*O(1)*	
Stack	Push	*O(1)*	There are no complexity trick questions asked for stacks or queues. We only mention them here for completeness. The two data-structures are more important from a last-in last-out (stack) and first in first out (queue) perspective.
	Pop	*O(1)*	
Queue	Enqueue	*O(1)*	
	Dequeue	*O(1)*	

Priority Queue (binary heap)	Insert Delete Get Max/Min	*O(lgn)* *O(lgn)* *O(1)*	
Hashtable	Insert Retrieve	*O(n)* *O(n)*	Be mindful that a hashtable's average case for insertion and retrieval is O(1)
B-Trees	Insert Retrieve	*O(logn)* *O(logn)*	
Red-Black Trees	Insert Retrieve	*O(logn)* *O(logn)*	

Algorithms

<table>
<tr><th>Category</th><th>Worst Case Complexity</th><th>Notes</th></tr>
<tr><td>Sorting</td><td><table>
<tr><td>Bubble Sort</td><td>O(n²)</td></tr>
<tr><td>Insertion Sort</td><td>O(n²)</td></tr>
<tr><td>Selection Sort</td><td>O(n²)</td></tr>
<tr><td>Quick Sort</td><td>O(n²)</td></tr>
<tr><td>Merge Sort</td><td>O(nlgn)</td></tr>
</table></td><td>Note, even though worst case quicksort performance is O(n²) but in practice quicksort is often used for sorting since its average case is O(nlgn).</td></tr>
<tr><td>Trees</td><td><table>
<tr><td>Depth First Search</td><td>O(n)</td></tr>
<tr><td>Breadth First Search</td><td>O(n)</td></tr>
<tr><td>Pre-order,</td><td>O(n)</td></tr>
</table></td><td>n is the total number of nodes in the tree. Most tree-traversal algorithms will end up seeing every node in the tree and their complexity in the worst case is thus O(n).</td></tr>
</table>

		In- order, Post- order Traver sals	

Epilogue

Getting a handle on the time and space complexity of algorithms and data-structures takes experience and effort. However, it is a must-have skill to master if writing software is your intended line of profession. Invariably, you will be pulled into discussions and reviews at work, and you will need to draw upon your knowledge of analyzing complexity to pick the right solution for the problem at hand. This course is an attempt to impart enough knowledge to the reader to help her with tough technical choices in everyday work, without requiring mathematical genius. Remember that in most real-life scenarios ***there's no perfect algorithm or data-structures, there are only tradeoffs.***

Please feel free to reach out to me on linked-in or on here for feedback, errors, omissions or just about anything :)

With best wishes.

C. H. Afzal.
29th Nov, 2018

penned with ♡at coffee shops across the bay-area, US of A.

Credits

Every great product is a result of team-effort and so is this course. Our team members include:

- Sana Bilal (Content enhancement and development)
- Ahsan Khalil (Illustrations and graphic design)

- Educative's Proofreading Ninjas

We are also thankful to our readers and acknowledge them below for pointing out omissions and errors in the course:

- vijshank
- Dragos
- Keith Fung
- Sonjeet C Paul
- Volodymyr Shulga
- Lance Wang