BASIC DATA STRUCTURES

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

8: Analysis of Algorithms

Alexandra Papoutsaki
she/her/hers

Today we're going to do some math and some science. Not a lot, but we need to have a scientific basis for understanding the performance of our algorithms to properly develop them and use them in practice.

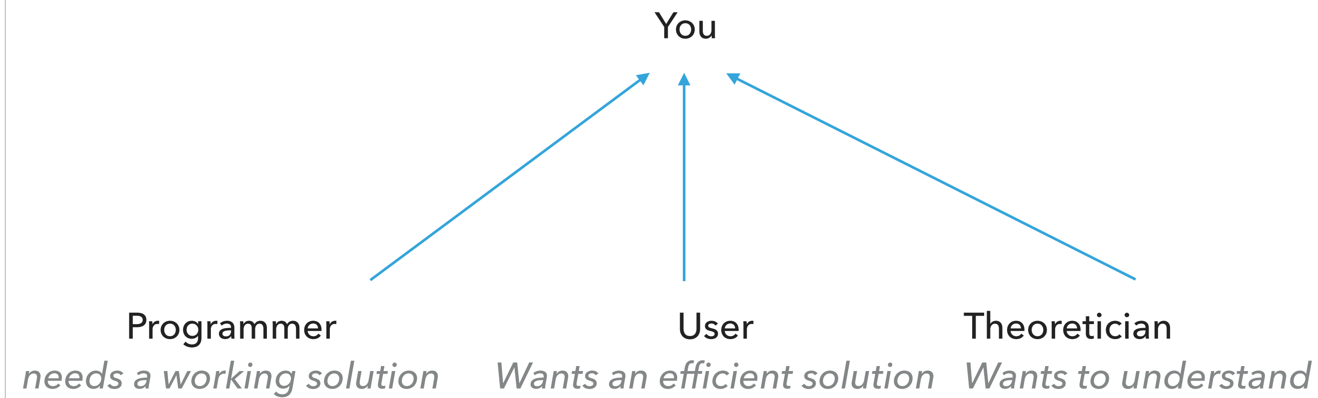
Lecture 8: Analysis of Algorithms

- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of ArrayList operations

Some slides adopted from Algorithms 4th Edition or COS226

We're going to look at how to make mathematical models and how to classify algorithms according to the order of growth of their running time and we'll apply those principles to analyze the running time of array list operations.

Different Roles



To put this all in perspective, we're going to think about these issues from the point of view of different types of characters. The first one is the programmer who needs to solve a problem programmatically. The second one is the user who wants to use whatever program to get the job done efficiently. The third one is the theoretician; that's somebody who really wants to understand what's going on when analyzing an algorithm. There's a little bit of each one of these perspectives in today's lecture. As a student, you have to think that you might be playing any or all of these roles some day. Therefore, it's pretty important to understand these different points of view.

3-SUM: Given n distinct numbers, how many unordered triplets sum to 0?

- Input: 30 -40 -20 -10 40 0 10 5
- Output: 4
  - 30 -40 10
  - 30 -20 -10
  - -40 40 0
  - -10 0 10

For a running example, we're going to use the 3-sum: If you have n distinct integers, how many triplets sum to exactly zero? For example, with these eight integers, there are four triples that sum to zero. Our goal is to write a program that can compute this quantity for any set of n distinct integers. The 3-sum problem is an extremely important computation that's deeply related to many problems in computational geometry which is a branch of computer science that covers the algorithms and underlying science related to graphics, movies, and geometric models of all sorts.

3-SUM: Brute force algorithm

```
public class ThreeSum {

public static int count(int[] a) {
        int n = a.length;
        int count = 0;
        for (int i = 0; i < n; i++) {
             for (int j = i+1; j < n; j++) {
                  for (int k = j+1; k < n; k++) {
                       if (a[i] + a[j] + a[k] == 0) {
                            count++;
                       }
                  }
             }
        }
        return count;
    }
```

```
public static void main(String[] args)  {
        String filename = args[0];
        int fileSize = Integer.parseInt(args[1]);
        try {
             Scanner scanner = new Scanner(new File(filename));
             int intList[] = new int[fileSize];
             int i=0;
             while(scanner.hasNextInt()){
                  intList[i]=scanner.nextInt();
                   i++;
             }
             Stopwatch timer = new Stopwatch();
             int count = count(intList);
             System.out.println("elapsed time = " + timer.elapsedTime());
             System.out.println(count);
        }
        catch (IOException e) {
             throw new IllegalArgumentException("Could not open " + filename, e);
        }
    }
```

Let's assume that the n distinct numbers are given to us in a file. We could come up with a brute force algorithm for solving the 3-sum problem. Our main method accepts two arguments through the args String[]. The first one corresponds to the name of the file that contains one integer per line. The second is the number of lines of the file. We use the Scanner class to read the specified number of lines and we count how much time has elapsed when running our count method. The count method has three nested loops that examine every combination of triples to see which ones add up to 0 and increases a counter accordingly. We are sure that our code works correctly but how much time does it take as a function of n (that is the number of distinct numbers?)

Empirical Analysis

- Input: 8ints.txt
- Output: 4 and 0

- Input: 1Kints.txt
- Output: 70 and 0.081

- Input: 2Kints.txt
- Output: 528 and 0.38

- Input: 2Kints.txt
- Output: 528 and 0.371

- Input: 4Kints.txt
- Output: 4039 and 2.792

- Input: 8Kints.txt
- Output: 32074 and 21.623

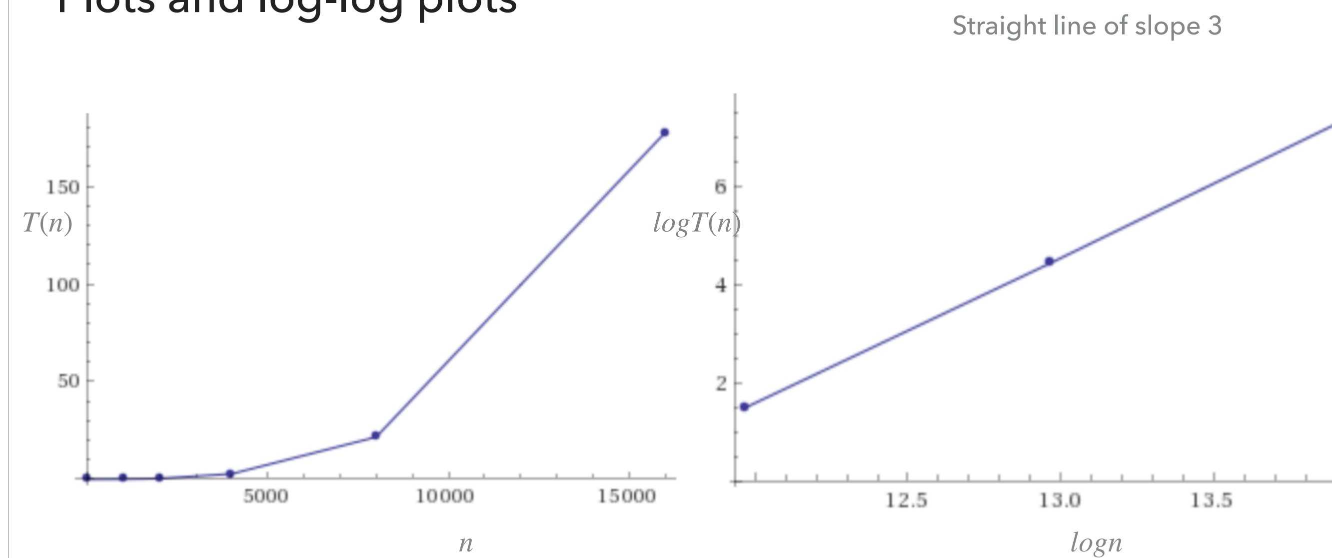
- Input: 16Kints.txt
- Output: 255181 and 177.344

Input size	Time
8	0
1000	0.081
2000	0.38
2000	0.371
4000	2.792
8000	21.623
16000	177.344

The stopwatch will help us with our empirical analysis. We will run our code for various input sizes that will double every time and measure their running time. The results are shown on the table. It’s interesting to note that for the same input, we will have slight variations in measurements every time we run it, but broadly they will show the same trend. What we see is that each time we're doubling the size of the input, our code takes longer to run.

EXPERIMENTAL ANALYSIS OF RUNNING TIME

Plots and log-log plots



- Regression: $T(n) = an^b$ (power-law).
- $\log T(n) = b \log n + \log a$, where b is slope.
- Experimentally: $\sim 0.42 \times 10^{-10} n^3$, in our example for ThreeSum.

We can now take these numbers and plot them. On the left figure, you see that the x-axis has the problem size (that is the number of distinct integers) and y-axis has the corresponding time it took in seconds. The right figure shows the log-log plot, where on the x-axis we have the logarithm of the problem size (the number of integers), and on the y-axis the logarithm of the time it took for each run. Very often, this will result in a a straight line. The slope (the change in y coordinate with respect to the change in x coordinate, m = (y2-y1)/(x2-x1)) of the straight line is the key to what's going on. In this case, the slope of the straight line is 3. We can run a regression to fit a straight line through the data points. We won't do the proof, but if you get a straight line and the slope is b (here, 3), then your function is proportional to an^b. That's called the power law. And that's true of many, many scientific problems including most polynomial algorithms. Experimentally, in our example for our three-sum problem this would result in 0.42x10^{-10}n^3, that is a=0.42 and b=3 (as we've already seen).

Input size	Time
8	0
1000	0.081
2000	0.38
4000	2.792
8000	21.623
16000	177.344

Doubling Hypothesis

- Doubling input size increases running time by a factor of $\frac{T(n)}{T(n/2)}$
- Run program doubling the size of input. Estimate factor of growth:
  - $\frac{T(n)}{T(n/2)} = \frac{an^b}{a(\frac{n}{2})^b} = 2^b$.
- E.g., in our example, for pair of input sizes 8000 and 16000 the ratio $(\frac{177.344}{21.623})$ is 8.2 or ~8 which can be written as 2^3, therefore b is approximately 3.
- Assuming we know b, we can figure out a.
  - E.g., in our example, $T(16000) = 177.34 = a \times 16000^3$.
    - Solving for a we get $a = 0.42 \times 10^{-10}$.

Let's use this to figure out how we can experimentally figure out how the running time of our algorithm grows as the problem size grows. We have already taken various measurements where we double the size of the problem and note how long it takes every time. What we will do is pick two large input sizes, where one is twice as big as the other one and divide their running times. This relationship will result in a number that can be rewritten as 2^b. This is known as the doubling hypothesis.
For example, our code took 177.344 sec and 21.623 seconds respectively for 16000 and 8000 random distinct integers. If we divide 177.344/21.623 we get 8.2 which is roughly equal to 8, which can be written as 2^3, thus b is approximately 3. Now that we know b, we can figure out a. Remember T(n)=an^b. n=16000 and b=3, thus solving for a we get a = 0.42x10^{-10}

PRACTICE TIME

- Suppose you time your code and you make the following observations. Which function is the closest model of $T(n)$?

A. n^2

B. $6 \times 10^{-4} n$

C. $5 \times 10^{-9} n^2$

D. $7 \times 10^{-9} n^2$

Input size	Time
1000	0
2000	0.0
4000	0.1
8000	0.3
16000	1.3
32000	5.1

Let's see whether we know how to apply the doubling hypothesis.

ANSWER

- C. $5 \times 10^{-9} n^2$
- T(32000)/T(16000) is approximately 4, therefore $b = 2$.
- $T(32000) = 5.1 = a \times 32000^2$.
- Solving for $a = 4.98 \times 10^{-9}$.s

Input size	Time
1000	0
2000	0.0
4000	0.1
8000	0.3
16000	1.3
32000	5.1

The correct answer was C

Effects on Performance

- System independent effects: Algorithm + input data
  - Determine b in power law relationships.
- System dependent effects: Hardware (e.g., CPU, memory, cache) + Software (e.g., compiler, garbage collector) + System (E.g., operating system, network, etc).
  - Dependent and independent effects determine a in power law relationships.

- Although it is hard to get precise measurements, experiments in Computer Science are cheap to run.

The running time of a program can be affected by the machine you run it on, but the primary effects are independent of what computer you run it on. Far more important is what algorithm you have chosen and what kind of data you have. These two factors will determine b, the exponent in the power law. Of course the system itself, as what hardware, software, and what other programs you run will affect the performance of your code. This will be captured in the a of the power law.

In modern systems there is so much going on in the hardware and software, it's sometimes difficult to get really precise measurements. But on the other hand we don't have to sacrifice animals, or fly to another planet the way they do in other sciences, we can just run a huge number of experiments and usually take care of understanding these kind of effects.

Lecture 8: Analysis of Algorithms

- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of ArrayList operations

Observing what's happening as we did in the last section, gives us a way to predict performance but it really doesn't help us understand what the algorithms are doing. So next, we're going to look at mathematical modeling, a way to get a better concept of what's really happening.

Total Running Time

- Popularized by Donald Knuth in the 60s in the four volumes of "The Art of Computer Programming".
  - Knuth won the Turing Award (The "Nobel" in CS) in 1974.

- In principle, accurate mathematical models for performance of algorithms are available.

- Total running time = sum of cost x frequency for all operations.
- Need to analyze program to determine set of operations.
- Exact cost depends on machine, compiler.
- Frequency depends on algorithm and input data.

We will focus on the total running time, a concept that was developed and popularized by Donald Knuth starting in the late 60s. At that time, computer systems were really becoming complicated for the first time and computer scientists were concerned about whether we really were going to be able to understand what's going on. Knuth was very direct in saying that this is something that we certainly can do. We can calculate the total running time of a program by identifying all the basic operations, figuring out the cost, figuring out the frequency of execution, and summing up the cost times frequency for all the operations. You have to analyze the program to determine what set of operations. As we saw, the cost depends on the machine and the computer in the system and the frequency on the algorithm and the input data. Knuth has written a series of books that give very detailed and exact analyses within a particular computer model for a wide range of algorithms. So, from Knuth, we know that we can get accurate mathematical models for the performance of algorithms or programs in operation.

Cost of Basic Operations

- Add < integer multiply < integer divide < floating-point add < floating-point multiply < floating-point divide.

Operation	Example	Nanoseconds
Variable declaration	`int a`	c_1
Assignment statement	`a = b`	c_2
Integer comparison	`a < b`	c_3
Array element access	`a[i]`	c_4
Array length	`a.length`	c_5
1D array allocation	`new int[n]`	$c_6 n$
2D array allocation	`new int[n][n]`	$c_7 n^2$
string concatenation	`s+t`	$c_8 n$

For basic operations, like addition etc. we'll postulate that the running time is some constant. If we're going to allocate an int array of size n, it takes time proportional to n because in Java, the default is that all the elements in an int array are initialized to zero. Another important one is string concatenation. If you concatenate two strings, the running time is proportional to the length of the final string. Many novices in programming and Java make the mistake of assuming that that's a constant time operation, when it's not.

Example:1-SUM

- How many operations as a function of n?

```
int count = 0;
for (int i = 0; i < n; i++) {
    if (a[i] == 0) {
        count++;
    }
}
```

Operation	Frequency
Variable declaration	2
Assignment	2
Less than	$n+1$
Equal to	n
Array access	n
Increment	n to $2n$

Let's look into 1-SUM, a very simple variant of a 3-SUM problem, which calculates how many numbers are zero. We can solve this problem with just one for loop: we go through our array and we test if the number is zero and increment our count. By analyzing that code, you can see that:
- i and count have to be declared and then they have to be assigned to zero. That gives us 2 variable declarations and 2 assignments.
- There's n+1 comparison of i against n (i<n).
- There's n comparisons of a[i] against zero (a[i]==0)
- There's n array accesses (a[I]).
- In terms of increment, we have n for the i (i++). count++ will be done 0 to n times depending on how many of our n numbers are 0. So overall, we get n to 2n increments.

Example: 2-SUM

$$1 + 2 + 3 + \ldots + n = n(n+1)/2$$

- How many operations as a function of n?

```
int count = 0;
 for (int i = 0; i < n; i++) {
     for (int j = i+1; j < n; j++) {
         if (a[i] + a[j] == 0) {
             count++;
         }
     }
 }
```

BECOMING TOO TEDIOUS TO CALCULATE

Operation	Frequency
Variable declaration	$n + 2$
Assignment	$n + 2$
Less than	$(n+1)(n+2)/2$
Equal to	$n(n-1)/2$
Array access	$n(n-1)$
Increment	$n(n+1)/2$ to n^2

Let's look at a more complicated problem, the 2-SUM problem which asks how many pairs of integers sum to zero.

- We have 1 declaration for count, 1 for i, and n for j (j gets redeclared every time i increases). That gives n+2 declarations.
- Same idea for assignments.
- Let’s count the frequency of the less than operations
  - The outer circle gives us n+1 such comparisons (i<n).
  - The inner circle gives us n(n+1)/2 such comparisons (j<n). How did we calculate that?

When i=0, we do n comparisons with j
When i=1, we do n-1 comparisons with j
When i=2, we do n-2 comparisons with j
…
When i=n-2, we do 2 comparisons with j
When i=n-1, we do 1 comparison with j

If we add those up we have 1+ 2 + ... + (n-1) +n = n(n+1)/2

Together we have (n+1) +n(n+1)/2= (n+1)(n+2)/2

Alternatively, this can be calculated as $\sum_{i=0}^{n}(1+ \sum_{j=i + 1}^{n} 1) = (n+1)(n + 2)/2$
- Let’s apply the same idea to counting the equal to operation (==).

i=0: n-1
i=1: n-2
i=2: n-3
...
i=n-2: 1
i=n-1: 0
0+1+ 2 + ... (n-2) + (n-1) = (n-1)(n-1+1)/2 = n(n-1)/2

Alternatively \sum_{I=0}^{n - 1}\sum_{j=i+1}^{n-1} = n(n-1)/2
- Let's apply the same idea to counting the array accesses. Remember we have two each time.
i=0: 2*(n-1)
i=1: 2*(n-2)
i=2: 2*(n-3)
…
i=n-3:2*2
i=n-2:2*1
i=n-1: 0
2(1+2+ ... (n-2) + (n-1))= 2(n(n-1)/2) = n(n-1)

Alternatively \sum_{i=0}^{n - 1}\sum_{j=i+ 1}^{n-1}2 = n(n-1)

- For increments:
    - the outer cycle gives n increments for i.
    - The inner cycle gives n(n-1)/2 increments for j.
    - Together n+n(n-1)/2= n(n+1)/2

    Alternatively, sum_{i=0}^{n-1}(1+\sum_{j=i+1}^{n-1}1) = n(n + 1)/2
    - Count++ can be executed from 0 times all the way to n(n-1)/2

    - All the increments together are:
        - at minimum n(n+1)/2 and
        - at maximum n(n+1)/2+ n(n-1)/2=n^2.

This is getting too tedious to have to do for every single operation!

Tilde Notation

- Estimate running time (or memory) as a function of input size n.
- Ignore lower order terms.
  - When n is large, lower order terms become negligible.

- Example 1: $\frac{1}{6}n^3 + 10n + 100 \quad \sim n^3$
- Example 2: $\frac{1}{6}n^3 + 100n^2 + 47 \quad \sim n^3$
- Example 3: $\frac{1}{6}n^3 + 100n^{\frac{2}{3}} + \frac{1/2}{n} \quad \sim n^3$

Instead of doing all these difficult calculations we will use the tilde notation that drops all the lower order terms. We will use it both when analyzing the running time or the memory as a function of the problem input size n. The idea is that when n is large, the lower order terms become negligible. Above you can see three examples of how we can simplify the analysis to showing that all three of these algorithms run in n^3 time.

Simplification

- Cost model: Use some basic operation as proxy for running time. E.g., array accesses
- Combine it with tilde notation.

Operation	Frequency	Tilde notation
Variable declaration	$n + 2$	$\sim n$
Assignment	$n + 2$	$\sim n$
Less than	$(n+1)(n+2)/2$	$\sim n^2$
Equal to	$n(n-1)/2$	$\sim n^2$
Array access	$n(n-1)$	$\sim n^2$
Increment	$n(n+1)/2$ to n^2	$\sim n^2$

- $\sim n^2$ array accesses for the 2-SUM problem

Our cost model will use some basic operation as proxy for running time. E.g., array accesses and combine it with the tilde notation. That means we can simplify the 2-SUM analysis to showing that the frequency of operations is n or n^2. We will choose the number of array accesses, n^2 as a proxy for the total cost of this algorithm.

Back to the 3-SUM problem

- Approximately how many array accesses as a function of input size n?

```
int count = 0;
for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        for (int k = j+1; k < n; k++) {
            if (a[i] + a[j] + a[k] == 0) {
                count++;
            }
        }
    }
}
```

- $\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1}\sum_{k=j+1}^{n-1} 3 = 1/2n(n^2 - 3n + 2) \sim n^3$ array accesses.

We can apply the same logic to figure out that the 3-SUM problem takes n^3 array accesses.

The outer loop runs n times.
The middle loop runs from i+1 to n-1, so it iterates n-1-(i+1) +1 = n-i-1 times for each iteration of the outer loop
The inner loop runs from j+1 to n-1, so it iterates n-1-(j+1)+1 = n-j-1 times for each iteration of the middle loop.
The total number of array accesses is the product of the number of iterations of the three loops multiplied by the number of array accesses inside the innermost loop.
Therefore, the total number of array accesses in terms of input size n is:

\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1}\sum_{k=j+1}^{n-1}3 = 1/2n(n^2 - 3n + 2) which simplifies to n^3

A much more detailed way to solving this would be calculating it as follows:
When:
i = 0, j= 1, k=2…n-1, that is 3*(n-2) array access
i=0, j=2, k= 3…n-1, that is 3*(n-3) array accesses
i=0, j=3, k=4…n-1, that is 3*(n-4) array accesses
…
i=0, j=n-2, k=n-1, that is 3*1 array accesses
Overall 3*(1+2+…+n-2) =3*(n-2)(n-1)/2
When:
i = 1, j=2, k=3…n-1, 3*(n-3) array accesses

…

$i= 1$, $j=n-2$, $k=1$, that is 3 array accesses

Overall $3*(1+2+\ldots+n-3) =3*(n-3)(n-2)/2$

...

We see the trend that for a certain i, we have $3*(n-2-i)(n-1-i)/2$ array accesses.

We have in total $\sum_{i=0}^{n-1} 3*(n-2-i)(n-1-i)/2$ which results to exactly what we calculated above, i.e. $1/2n(n^2-3n+2)$ which simplifies to n^3

Lecture 8: Analysis of Algorithms

- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of ArrayList operations

Fortunately, when we analyze algorithms not too many different functions arise and that property allows us to classify algorithms according to their performance as the problem size grows.

Types of analysis

- Best case: lower bound on cost.
  - What the goal of all inputs should be.
  - Often not realistic, only applies to "easiest" input.
- Worst case: upper bound on cost.
  - Guarantee on all inputs.
  - Calculated based on the "hardest" input.
- Average case: expected cost for random input.
  - A way to predict performance.
  - Not straightforward how we model random input.

In general, theoretical computer scientists perform three types of analysis for algorithms. They look at the best case, that is the lower bound on cost with the goal of what the running time should be for all inputs. But this is often no realistic and it only applies to the easiest input. On the other side, we have the worst case analysis that provides an upper bound on cost. This is a guarantee about the longest our code would take to run for any input at it is calculated on what is the hardest input we would expect. Somewhere in the middle, there is the average case that focuses on the expected cost for a random input. It is the most realistic performance prediction but it is not very easy to figure out how to model random input.

Worst case analysis

- Definition: If $f(n) \sim cg(n)$ for some constant $c > 0$, then the order of growth of $f(n)$ is $g(n)$.
  - Ignore leading coefficients.
  - Ignore lower-order terms.
- We will be using the big-Oh (O) notation. For example:
  - $3n^3 + 2n + 7 = O(n^3)$
  - $2^n + n^2 = O(2^n)$
  - $1000 = O(1)$
- Yes, $3n^3 + 2n + 7 = O(n^6)$, but that's a rather useless bound.

In this class, we will focus on the worst case analysis. We will start with the definition of order of growth, which an approximation of the time required to run a computer program as the input size increases. There is a formal definition but I want you to keep two things. If you calculate that the running time of a program can be described by a certain function, you can ignore the leading coefficients and drop the lower order terms. We will use the Big-Oh notation.

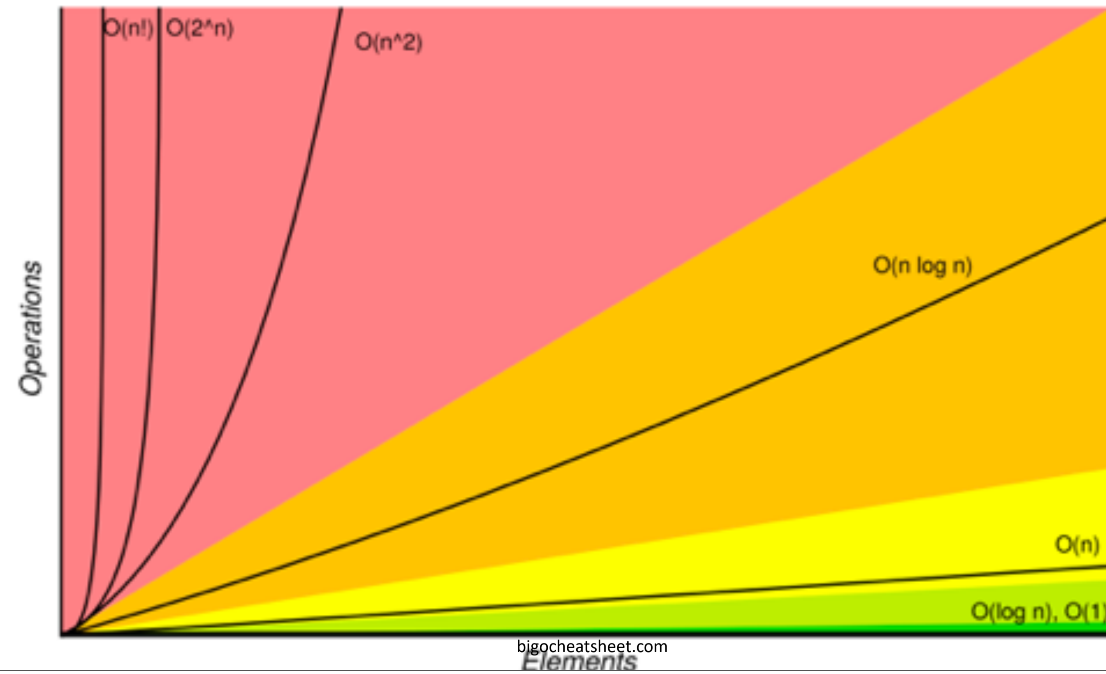
Common order of growth classifications

- Good news: only a small number of function suffice to describe the order-of-growth of typical algorithms.
- 1: constant
  - Doubling the input size won't affect the running time. Holy-grail.
- $\log n$: logarithmic
  - Doubling the input size will increase the running time by a constant.
- n : linear
  - Doubling the input size will result to double the running time.
- $n \log n$: linearithmic
  - Doubling the input size will result to a bit longer than double the running time.
- n^2: quadratic
  - Doubling the input size will result to four times as much running time.
- n^3: cubic
  - Doubling the input size will result to eight times as much running time.
- 2^n: exponential
  - When you increase the input by some constant amount, the time taken is doubled.
- $n!$: factorial
  - Running time grows exponentially with the size of the input.

The good news is there's only a small number of functions that turn up in the algorithms that we are interested in. O(1) is constant: doubling the input size won't affect the running time. That's the holy-grail but also very hard to achieve. O(long) is logarithmic. If we double the input size we will see constant slow down of the running time. Next is O(n), linear. Doubling the input size will result to double the running time. O(nlogn) is linearithmic: Doubling the input size will result to a bit longer than double the running time. Next comes O(n^2) which is quadratic. Doubling the input size will result to four times as much running time. O(n^3), cubic: Doubling the input size will result to eight times as much running time. O(2^n) is factorial: When you increase the input by some constant amount, the time taken is doubled. And finally we have factorial O(n!): Running time grows exponentially with the size of the input.

From slowest growing to fastest growing

- $1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$



Here's a good graphic about what we hope for and what we can be ok with. In general the slowest growing is constant and the fastest growing is factorial. Constant and logarithmic are the idea. We can be ok with linear. Above that, things start slowing down considerably.

Common order of growth classifications

Order-of-growth	Name	Example code	$T(n)/T(n/2)$
1	Constant	`a[i]=b+c`	1
$\log n$	Logarithmic	`while(n>1){n=n/2;…}`	~ 1
n	Linear	`for(int i=0; i<n; i++)`	2

$n \log n$	Linearithmic	`for (i = 1; i <= n; i++){`
		`int x = n;`
		`while (x > 0)`
		`x -= i;`
		`}`
			~ 2

n^2	Quadratic	`for(int i=0; i<n; i++) {` `for(int j=0; j<n; j++){`	4

n^3	Cubic	`for(int i=0; i<n; i++) {`
		`for(int j=0; j<n; j++){`
		`for(int k=0; k<n; k++){`
			8

These are some examples that we might encounter in this class. Note that the last column gives you what the experimental output would be.

I think they are all self-explanatory. For the linearithmic example:
The outer loop runs n times.
For each iteration, the inner loops runs n / i times.

The total number of runs is:
n + n/2 + n/3 + ... + n/n
Asymptotically (ignoring integer arithmetic rounding), this simplifies as

n * (1 + 1/2 + 1/3 + ... + 1/n)
This series loosely converges towards n * log(n).

Useful approximations

- Harmonic sum: $H_n = 1 + 1/2 + 1/3 + \ldots + 1/n \quad \sim \ln n$
- Triangular sum: $1 + 2 + 3 + \ldots + n \quad \sim n^2$
- Geometric sum: $1 + 2 + 4 + 8 + \ldots + n = 2n - 1 \quad \sim n$, when n power of 2.
- Binomial coefficients: $\binom{n}{k} \quad \sim \frac{n^k}{k!}$ when k is a small constant.

- Use a tool like Wolfram alpha.

Here are some useful approximations to be aware of in general when you analyze the running time of code. I don't expect you to have these memorized but be aware they exist and they can simplify sums.

Lecture 8: Analysis of Algorithms

- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of ArrayList Operations

Now that we know how we can analyze the running time of an algorithm both experimentally and mathematically, let's apply these lessons to seeing the running time of array list operations we encountered last time.

Worst-case performance of `add()` is $O(n)$

- Cost model: 1 for insertion, n for copying n items to a new array.
- Worst-case: If ArrayList is full, `add()` will need to call `resize` to create a new array of double the size, copy all items, insert new one.
- Total cost: $n + 1 = O(n)$.

- Realistically, this won't be happening often and worst-case analysis can be too strict. We will use amortized time analysis instead.

The worst-case performance for add is O(n). For our cost model, we will assume that we will pay some constant, let's say for each insertion, and n for when we need to copy n items to a new array. The worst-case scenario is that the array list is full and add calls resize to create a new array of double the size, copy all items, insert new one. The total cost would be n+1 which is O(n). Realistically, this won't be happening often and worst-case analysis can be too strict. We will use amortized time analysis instead.

Amortized analysis

- Amortized cost per operation: for a sequence of n operations, it is the total cost of operations divided by n.

Amortized time analysis says that worst-case analysis is to pessimistic. Instead, we will perform a sequence of n operations and we will take the average (the total total cost of operations divided by n). There are

Amortized analysis for n add() operations

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Insertion Cost	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Copying Cost	0	1	2	0	4	0	0	0	8	0	0	0	0	0	0	0	16
Total Cost	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17

- As the ArrayList increases, doubling happens *half as often* but *costs twice as much.*
- O(total cost)= $\sum$(“cost of insertions”) + $\sum$(“cost of copying”)
- $\sum$(“cost of insertions”) $= n$.
- $\sum$(“cost of copying”) = $1 + 2 + 2^2 + \ldots 2^{\lfloor \log_2 n \rfloor} \leq 2n$.
- O(total cost) $\leq 3n$, therefore amortized cost is $\leq \frac{3n}{n} = 3 = O^+(1)$, but “lumpy”.

Let's see how that would play out for n additions. Remember, we pay 1 for every insertion so the total cost of n insertions is n*1 = n.

We also pay n for every time we need to resize the array. Assume that we start with an array list of 1 capacity. When the first addition happens, we are good, we don't need to copy over. Now a second add comes, we need to copy the 1 element and add 1 element. That will result to cost of 2. When the third addition happens, we will need to copy the two old elements and add a new one so we pay 3 in total. But then we're set until the fifth add. In general, as the ArrayList increases, doubling happens half as often but costs twice as much. We need to copy things logn times if we have n additions. If we add up the cost for insertion and cost for copying, we will get that the total cost is smaller than 3n.

Thus the amortized cost is smaller than 3n/n<=3 which is O^+(1). So overall, we say that the amortized cost for add is constant although we will occasionally pay a lot to copy the old elements.

Amortized analysis for n add() operations when increasing ArrayList by 1.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Insertion Cost	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Copying Cost	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Total Cost	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

- $\sum$("cost of insertions") $= n$.
- $\sum$("cost of copying") $= 1 + 2 + 3 + \ldots + n - 1 = n(n-1)/2$.
- O(total cost) $= n + n(n-1)/2 = n(n+1)/2$, therefore amortized cost is $(n+1)/2$ or $O^+(n)$.
- Same idea when increasing ArrayList size by a constant.
  - This is why in the lab on Friday, we saw that doubling was the fastest and linear(1) the slowest.

Let's see now what it would cost if we increased the capacity by 1 instead of doubling the underlying array, every time it was full. The cost of insertions remains the same. But the cost of copying is quadratic. Therefore, the amortized cost would have been linear. The same idea applies when increasing ArrayList size by a constant. This is why in the lab on Friday, we saw that doubling was the fastest and linear(1) the slowest. In fact, if you remember, we saw that for n additions, the running time was becoming four times slower. That means that on average, for one addition, it would be linear. We have both experimental and mathematical proof of why doubling is a far superior idea!

Lecture 8: Analysis of Algorithms

- Experimental Analysis of Running Time
- Mathematical Models of Running Time
- Order of Growth Classification
- Analysis of ArrayList operations

And that's it for today. From now on, we will be using these tools to analyze the running time of all of the data structures we will encounter.

Readings:

- Recommended Textbook:
  - Chapter 1.4 (pages 172-205)
- Recommended Textbook Website:
  - Resizable arrays (arraylists): https://algs4.cs.princeton.edu/13stacks/
  - Analysis of Algorithms: https://algs4.cs.princeton.edu/14analysis/

Code

- Lecture 8 code

Practice Problems:

- 1.4.1-1.4.9, 1.4.32, 1.4.35-1.4.36