CSE 143
Lecture 8

Complexity

slides created by Ethan Apter
http://www.cs.washington.edu/143/

Intuition

- Are the following operations "fast" or "slow"?

array

behavior	fast/slow
add at front	slow
add at back	fast
get at index	fast
resizing	slow
binary search	(pretty) fast

linked list

behavior	fast/slow
add at front	fast
add at back	slow
get at index	slow
resizing	N/A (fast!)
binary search	(really) slow

2

Complexity

- "Complexity" is a word that has a special meaning in computer science
- **complexity**: the amount of computational resources a block of code requires in order to run
- main computational resources:
  - **time**: how long the code takes to execute
  - **space**: how much computer memory the code consumes
- Often, one of these resources can be traded for the other:
  - e.g.: we can make some code use less memory if we don't mind that it will need more time to finish (and vice-versa)

3

Time Complexity

- We usually care more about time complexity
  - we want to make our code run fast!
- But we don't merely measure how long a piece of code takes to determine it's time complexity
  - Why not?
- That approach would have results strongly skewed by:
  - size/kind of input
  - speed of the computer's hardware
  - other programs running at the same time
  - operating system
  - etc

4

Time Complexity

- Instead, we care about the growth rate as the input size increase
- First, we have to be able to measure the input size
  - the number of names to sort
  - the number of nodes in a linked list
  - the number of students in the IPL queue
- We usually call the input size "n"
- What happens if we double the input size (n → 2n)?
  - Will the running time double? quadruple? take forever?

5

Time Complexity

- We can learn about this growth rate in two ways:
  - by examining code
  - by running the same code over different input sizes
- Measuring the growth rate by is one of the few places where computer science is like the other sciences
  - here, we actually collect data
- But this data can be misleading
  - modern computers are very complex
  - some features (code optimizations) interfere with our data

6

Time Complexity

- We'll count most "simple" statements as 1 time unit
  - this includes `i = i + 1`, `int x = elementData[i]`, etc
  - but not loops! (or methods that contain loops!)

7

Time Complexity

- Examples:

1
```
int x = 4 * 10 / 3 + 2 - 10 * 42;
```

100
```
for (int i = 0; i < 100; i++) {
    x += i;
}
```

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

$n^2 + 100 + 1$

8

Optimizing Code

- Many programmers care a lot about efficiency
- But many inexperienced programmers *obsess* about it
  - and the wrong kind of efficiency, at that
- Which one is faster:

```
System.out.println("print");
System.out.println("me");
```

or:

```
System.out.println("print\nme");
```

Who cares? Any difference is insignificant

- If you're going to optimize some code, improve it so that you get a real benefit!

9

Growth Rates

- We care about n as it gets bigger
  - it's a lot like calculus, with n approaching infinity
    - you all know calculus, right?
- So, when we see something complicated like this:

$$\frac{n^3 - 18n^2 + 385n + 708}{0.005n^4 - 13n^2 + 73842}$$

- We can remove all the annoying terms:

$$\frac{n^3}{n^4}$$

- And as n gets really big, this approaches 0

10

Big O Notation

- We need a way to write a growth rate of a block of code
- Computer scientists use big O ("big oh") notation
  - O(n)
  - $O(n^2)$
- In big O notation, we ignore coefficients that are constants
  - 5n is written as O(n)
  - 100n is also written as O(n)
  - $0.05n^2$ is written as $O(n^2)$ and will eventually outgrow O(n)
- Each O(*[something]*) specifies a different complexity class

11

Complexity Classes

Complexity Class	Name	Example
O(1)	**constant time**	**accessing an array element**
O(log n)	**logarithmic time**	**binary search on an array**
O(n)	**linear time**	**scanning all elements of an array**
O(n log n)	**log-linear time**	**binary search on a linked list and good sorting algorithms**
$O(n^2)$	**quadratic time**	**poor sorting algorithms (like inserting n items into `SortedIntList`)**
$O(n^3)$	**cubic time**	**(example later today)**
$O(2^n)$	**exponential time**	**Really hard problems. These grow so fast that they're impractical**

12

Examples of Each Complexity Class's Growth Rate

- Assume that all complexity classes can process an input of size 100 in 100ms

Input Size (n)	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^3)$	$O(2^n)$
100	100ms	100ms	100ms	100ms	100ms	100ms	100ms
200	100ms	115ms	200ms	240ms	400ms	800ms	32.7 sec
400	100ms	130ms	400ms	550ms	1.6 sec	6.4 sec	12.4 days
800	100ms	145ms	800ms	1.2 sec	6.4 sec	51.2 sec	36.5 million years
1600	100ms	160ms	1.6 sec	2.7 sec	25.6 sec	6 min 49.6 sec	$4.21 * 10^{24}$ years
3200	100ms	175ms	3.2 sec	6 sec	1 min 42.4 sec	54 min 36 sec	$5.6 * 10^{61}$ years

13

Case Study: `maxSum`

- Given an array of **`int`**s, find the subsequence with the maximum sum

- Additional information:
  - values in the array can be negative, positive, or zero
  - the subsequence must be contiguous (can't skip elements)
  - you must compute:
    - the value of the sum of this subsequence
    - the starting index (inclusive) of this subsequence
    - the stopping index (inclusive) of this subsequence

- This has been used as a Microsoft interview question!

14

Case Study: `maxSum`

- For example: suppose you were given the following array:

0	1	2	3	4	5	6	7	8	9
14	8	-23	4	6	10	-18	5	5	11

max subsequence

max sum: 4 + 6 + 10 + -18 + 5 + 5 + 11 = 23
starting index: 3
stopping index: 9

- Notice that we included a negative number (-18)!
  – but this also let us include the 4, 6, and 10

15

Case Study: `maxSum`

- First, a simple way to solve this: try every subsequence!

- Psuedo-code:

```
// try every start index, from 0 to size - 1
    // try every stop index, from start index to size - 1
        // compute the sum from start index to stop index
```

- Converted to be part code, part pseudo-code:

```
for (int start = 0; start < list.length; start++) {
    for (int stop = start; stop < list.length; stop++) {
        // compute the sum from start index to stop index
    }
}
```

16

Case Study: maxSum

- Now, we just need to convert this pseudo-code:

```
// compute the sum from start index to stop index
```

- ...into code. Here's one way:

```
int sum = 0;
for (int i = start; i <= stop; i++) {
    sum += list[i];
}
```

- And we need to store this sum if it becomes our max sum:

```
if (sum > maxSum) {
    maxSum = sum;
}
```

17

Case Study: maxSum

- Here's our whole algorithm, with some initialization:

```
int maxSum = list[0];
int maxStart = 0;
int maxStop = 0;
for (int start = 0; start < list.length; start++) {
    for (int stop = start; stop < list.length; stop++) {
        int sum = 0;
        for (int i = start; i <= stop; i++) {
            sum += list[i];
        }
        if (sum > maxSum) {
            maxSum = sum;
            maxStart = start;
            maxStop = stop;
        }
    }
}
```

this is the most frequently executed line of code

18

Case Study: `maxSum`

- What complexity class is the previous algorithm?
  - $O(n^3)$ (cubic time)
- This is pretty slow
  - we recalculate the entire sum every time:
    - calculate the entire sum from index 0 to index 0
    - calculate the entire sum from index 0 to index 1
    - ...
    - calculate the entire sum from index 0 to index 999
- How can we improve it?
  - remember the old sum (values list[start] to list[stop-1])
  - add the single new value (list[stop]) to the old sum

19

Case Study: `maxSum`

- Improved code, now with a running sum:

```
int maxSum = list[0];
int maxStart = 0;
int maxStop = 0;
for (int start = 0; start < list.length; start++) {
    int sum = 0;
    for (int stop = start; stop < list.length; stop++) {
        sum += list[stop];
        if (sum > maxSum) {
            maxSum = sum;
            maxStart = start;
            maxStop = stop;
        }
    }
}
```

these are the most frequently executed lines of code

20

Case Study: `maxSum`

- What complexity class is the previous algorithm?
  - $O(n^2)$ (quadratic time)
- This is a *big* improvement over the old code
  - it now runs much faster for large input sizes
- And it wasn't that hard to convert our first version to this improved version
- But we can still do better
  - if only we can figure out how...

21

Case Study: `maxSum`

- There is a better algorithm, but it's harder to understand
  - (nor do you need to understand it)
- The main idea is that we will find the max subsequence *without* computing all the sums
  - this will eliminate our inner for loop
  - ...which means we can find the subsequence with just a single loop over the array
- We need to know when to reset our running sum
  - this will "throw out" all previous values
  - but we have to know for sure that we don't want them!

22

Case Study: `maxSum`

- Suppose we're about to look at an index greater than 0
  - for example index 10
- If we're going to include previous values, we must include the value at index 9
  - index 9 is immediately before index 10
- We want to use only the best subsequence that ends at 9
- And only if it helps us. When does it help?
  - it helps when the sum of this old subsequence is positive
  - and hurts when the sum of this old subsequence is negative

23

Case Study: `maxSum`

- Best code:

```
int maxSum = list[0];
int maxStart = 0;
int maxStop = 0;
int sum = 0;
int start = 0;
for (int i = 0; i < list.length; i++) {
    if (sum < 0) {
        sum = 0;
        start = i;
    }
    sum += list[i];
    if (sum > maxSum) {
        maxSum = sum;
        maxStart = start;
        maxStop = i;
    }
}
```

these are the most frequently executed lines of code

24

Case Study: `maxSum`

- What complexity class is our best algorithm?
  - O(n) (linear time)
- This is again a big improvement over both other versions
- But let's not just take my word for it
- Let's conduct an experiment (in MaxSum.java -- available on the website)
  - we'll give an array of **`int`**s of some size to each algorithm
  - ...and then give the algorithm an array of *twice* that size
  - ...and then give the algorithm an array of *triple* that size
  - ...and see how long it takes

25

MaxSum.java

- Output for an array of 1500 **`int`**s in the O(n^3) algorithm:

```
How many numbers do you want to use? 1500
Which algorithm do you want to use? 1
Max = 172769
Max start = 677
Max stop = 971
for n = 1500, time = 0.96

Max = 198959
Max start = 1727
Max stop = 1972
for n = 3000, time = 7.543

Max = 614711
Max start = 251
Max stop = 3870
for n = 4500, time = 25.427

Double/single ratio = 7.857291666666667
Triple/single ratio = 26.486458333333335
```

these numbers are close to 8 (2^3) and 27 (3^3) respectively, so this algorithm exhibited O (n^3) growth

26

MaxSum.java

- Output for an array of 30,000 `ints` in the $O(n^2)$ algorithm:

```
How many numbers do you want to use? 30000
Which algorithm do you want to use? 2
Max = 809852
Max start = 10146
Max stop = 19139
for n = 30000, time = 0.988

Max = 2170008
Max start = 9832
Max stop = 25833
for n = 60000, time = 3.935

Max = 4112483
Max start = 74
Max stop = 88871
for n = 90000, time = 8.853

Double/single ratio = 3.9827935222672064
Triple/single ratio = 8.960526315789474
```

these numbers are close to 4 (2^2) and 9 (3^2) respectively, so this algorithm exhibited $O(n^2)$ growth

MaxSum.java

- Output for an array of 5,000,000 `ints` in the $O(n)$ algorithm:

```
How many numbers do you want to use? 5000000
Which algorithm do you want to use? 3
Max = 22760638
Max start = 456
Max stop = 4998134
for n = 5000000, time = 0.016

Max = 27670910
Max start = 1045808
Max stop = 9643590
for n = 10000000, time = 0.031

Max = 28178549
Max start = 239081
Max stop = 8574748
for n = 15000000, time = 0.044

Double/single ratio = 1.9375
Triple/single ratio = 2.75
```

look at how fast it processed 5,000,000, 10,000,000, and 15,000,000 `ints`!

these numbers are close to 2 and 3 respectively, so this algorithm exhibited O(n) growth