CIS 211

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)

Intuition

• Are the following operations “fast” or “slow”?

<table>
<thead>
<tr>
<th></th>
<th>array</th>
<th>linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td>behavior</td>
<td>fast/slow</td>
<td>behavior</td>
</tr>
<tr>
<td>add at front</td>
<td>slow</td>
<td>add at front</td>
</tr>
<tr>
<td>add at back</td>
<td>fast</td>
<td>add at back</td>
</tr>
<tr>
<td>get at index</td>
<td>fast</td>
<td>get at index</td>
</tr>
<tr>
<td>resizing</td>
<td>slow</td>
<td>resizing</td>
</tr>
<tr>
<td>binary search</td>
<td>(pretty) fast</td>
<td>binary search</td>
</tr>
</tbody>
</table>

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
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?

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

Time Complexity

• We’ll count most “simple” statements as 1 time unit
  – this includes \( i = i + 1, \) \( \text{int } x = \text{elementData}[i], \) etc
  – but not loops! (or methods that contain loops!)

Time Complexity

• Examples:

\[
\begin{align*}
\text{int } x &= 4 \times 10 / 3 + 2 - 10 \times 42; \\
\text{for } (\text{int } i = 0; i < 100; i++) \{ \\
& \quad \text{x } += \text{i;} \\
\} \\
\text{for } (\text{int } i = 0; i < n; i++) \{ \\
& \quad \text{for } (\text{int } j = 0; j < n; j++) \{ \\
& \quad \quad \text{x } += \text{i } + \text{j;} \\
& \quad \} \\
& \} \\
\end{align*}
\]
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!

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

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

Complexity Classes

<table>
<thead>
<tr>
<th>Complexity Class</th>
<th>Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>constant time</td>
<td>accessing an array element</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>logarithmic time</td>
<td>binary search on an array</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>linear time</td>
<td>scanning all elements of an array</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>log-linear time</td>
<td>binary search on a linked list and good sorting algorithms</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>quadratic time</td>
<td>poor sorting algorithms (like inserting $n$ items into SortedIntList)</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>cubic time</td>
<td>(example later today)</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>exponential time</td>
<td>Really hard problems. These grow so fast that they’re impractical</td>
</tr>
</tbody>
</table>
Examples of Each Complexity Class's Growth Rate

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

<table>
<thead>
<tr>
<th>Input Size (n)</th>
<th>O(1)</th>
<th>O(log n)</th>
<th>O(n)</th>
<th>O(n log n)</th>
<th>O(n^2)</th>
<th>O(n^3)</th>
<th>O(2^n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100ms</td>
<td>100ms</td>
<td>100ms</td>
<td>100ms</td>
<td>100ms</td>
<td>100ms</td>
<td>100ms</td>
</tr>
<tr>
<td>200</td>
<td>100ms</td>
<td>115ms</td>
<td>200ms</td>
<td>400ms</td>
<td>800ms</td>
<td>32.7 sec</td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>100ms</td>
<td>130ms</td>
<td>400ms</td>
<td>550ms</td>
<td>6.4 sec</td>
<td>12.4 days</td>
<td></td>
</tr>
<tr>
<td>800</td>
<td>100ms</td>
<td>145ms</td>
<td>800ms</td>
<td>6.4 sec</td>
<td>51.2 sec</td>
<td>36.5 million years</td>
<td></td>
</tr>
<tr>
<td>1600</td>
<td>100ms</td>
<td>160ms</td>
<td>1.6 sec</td>
<td>25.6 sec</td>
<td>6 min</td>
<td>4.21 * 10^24 years</td>
<td></td>
</tr>
<tr>
<td>3200</td>
<td>100ms</td>
<td>175ms</td>
<td>3.2 sec</td>
<td>1 min</td>
<td>54 min</td>
<td>5.6 * 10^61 years</td>
<td></td>
</tr>
</tbody>
</table>

Case Study: maxSum

- Given an array of ints, 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!

Case Study: maxSum

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

  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

Case Study: maxSum

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

  Psuedo-code:
  ```java
  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
    }
  }
  ```

  Converted to be part code, part pseudo-code:
  ```java
  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
    }
  }
  ```
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;
  }

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;
      }
    }
  }

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

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;
      }
    }
  }
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...

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!

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

Case Study: maxSum
• Best code:

```java
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
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 ints 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

MaxSum.java

- Output for an array of 1500 ints 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

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 = 4121483
  Max start = 74
  Max stop = 88871
  for n = 90000, time = 8.853

  Double/single ratio = 4.4
  Triple/single ratio = 9

  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

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

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