Lecture 12: Binary Search; Complexity

07/20/22
Reminders

• A2 Resubmission due Wednesday 7/20 @ 11:59pm
• Checkpoint 5 due Sunday 7/24 @ 11:59 pm
Midterm on Friday

• We give lots of partial credit! Write down everything you know.
  • No credit for pseudocode or comments – only for real code
  • Your code doesn’t have to be complete to earn credit
    • If you know you need a while loop, but don’t exactly know what the condition is, write down the while loop anyway
    • etc.

• Manage your time well.
  • Move on to the next question if you feel like you’re stuck

• Don’t write before or after time is called – nothing you write is worth -10 points
Midterm on Friday

• Bring your Husky ID

• We will start at 10:50 sharp to give you the full 60 minutes.
  • Arrive early!

• Make sure you sleep!
Complexity / Efficiency

• Finally!
• Best of CS
Sum up numbers 1 to n

• Let’s write a method to calculate the sum from 1 to some \( n \):

```java
public static int sum1(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    return sum;
}
```

Which one is more efficient?

• Gauss also has a way of solving this:

```java
public static int sum2(int n) {
    return n * (n + 1) / 2;
}
```

\[
1 + 2 + \ldots + n = \frac{n(n + 1)}{2}
\]
Efficiency

• **Efficiency**: measure of computing resources used by code.
  • Resources:
    • Time
    • Space
    • Energy
    • ...
  • Most commonly refers to **time**

• We want to be able to compare different algorithms to see which is more efficient
Runtime Efficiency Try 1

• Let's time the methods!

<table>
<thead>
<tr>
<th>n</th>
<th>sum1 took</th>
<th>sum2 took</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 1</td>
<td>0ms</td>
<td>0ms</td>
</tr>
<tr>
<td>n = 5</td>
<td>0ms</td>
<td>0ms</td>
</tr>
<tr>
<td>n = 10</td>
<td>0ms</td>
<td>0ms</td>
</tr>
<tr>
<td>n = 100</td>
<td>0ms</td>
<td>0ms</td>
</tr>
<tr>
<td>n = 1,000</td>
<td>1ms</td>
<td>0ms</td>
</tr>
<tr>
<td>n = 10,000,000</td>
<td>8ms</td>
<td>0ms</td>
</tr>
<tr>
<td>n = 100,000,000</td>
<td>43ms</td>
<td>0ms</td>
</tr>
<tr>
<td>n = 2,147,483,647</td>
<td>804ms</td>
<td>0ms</td>
</tr>
</tbody>
</table>
Runtime Efficiency Try 1

• Let’s time the methods!

\[
\begin{array}{lll}
n = 1 & \text{sum1 took } 0ms, \text{sum2 took } 0ms \\
n = 5 & \text{sum1 took } 0ms, \text{sum2 took } 0ms \\
n = 10 & \text{sum1 took } 0ms, \text{sum2 took } 0ms \\
n = 100 & \text{sum1 took } 0ms, \text{sum2 took } 0ms \\
n = 1,000 & \text{sum1 took } 0ms, \text{sum2 took } 0ms \\
n = 10,000,000 & \text{sum1 took } 10ms, \text{sum2 took } 0ms \\
n = 100,000,000 & \text{sum1 took } 47ms, \text{sum2 took } 0ms \\
n = 2,147,483,647 & \text{sum1 took } 784ms, \text{sum2 took } 0ms \\
\end{array}
\]
## Runtime Efficiency Try 1

- Let's time the methods!

<table>
<thead>
<tr>
<th>n</th>
<th>Sum1 took</th>
<th>Sum2 took</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 1</td>
<td>0ms</td>
<td>0ms</td>
</tr>
<tr>
<td>n = 5</td>
<td>0ms</td>
<td>0ms</td>
</tr>
<tr>
<td>n = 10</td>
<td>0ms</td>
<td>0ms</td>
</tr>
<tr>
<td>n = 100</td>
<td>0ms</td>
<td>0ms</td>
</tr>
<tr>
<td>n = 1,000</td>
<td>1ms</td>
<td>0ms</td>
</tr>
<tr>
<td>n = 10,000,000</td>
<td>3ms</td>
<td>0ms</td>
</tr>
<tr>
<td>n = 100,000,000</td>
<td>121ms</td>
<td>0ms</td>
</tr>
<tr>
<td>n = 2,147,483,647</td>
<td>1750ms</td>
<td>0ms</td>
</tr>
</tbody>
</table>

- Different computers give different results
- The same computer gives different results!!! D:<
Runtime Efficiency Try 2

• Count number of “simple steps” our algorithm takes to run

• Assume the following:
  • Statement: any single statement 1 step to run
    • `int x = 5;`
    • `boolean b = (5 + 1 * 2) < 15 + 3;`
    • `System.out.println("Hello");`

• Loop: number of times the loop runs * the steps in the body

• Method call: total number of steps inside the method's body
public static void method1(int N) {
    statement1;
    statement2;
    statement3;
}

for (int i = 1; i <= N; i++) {
    statement4;  // 1
}

for (int i = 1; i <= N; i++) {
    statement5;
    statement6;
    statement7;
}

3 + N + 3N
How many “steps” are in this method?

```java
public static void method2(int N) {
    for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++) {
            statement1;
        }
    }
    for (int i = 1; i <= N; i++) {
        statement2;
        statement3;
        statement4;
        statement5;
    }
}
```

The total number of steps is $N^2 + 4N$. 
Sum: how many steps in each?

\[ f(N) = N + 2 \]

```java
public static int sum1(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    return sum;
}
```

\[ g(N) = 1 \]

```java
public static int sum2(int n) {
    return n * (n + 1) / 2;
}
```
Big-O

We report runtime efficiency in terms of the general growth rate of an algorithm.

- **N**: size of the input data
- **Growth rate**: change in runtime as N changes.

Big-O is our notation for reporting this growth rate!

- We only care about the general growth rate
- We do not care about specific scaling factors
Big-O

Say an algorithm runs $0.4N^3 + 25N^2 + 8N + 17$ statements.

• Example:
  • $N = 1$ trillion = $1,000,000,000,000$
  • $N^3 = 1$ undecillion = $1,000,000,000,000,000,000,000,000,000,000,000,000$
  • The constants and lower-order terms don’t matter! The highest-order term ($N^3$) dominates the overall runtime.
  • We say that this algorithm runs "on the order of" $N^3$.
  • or $O(N^3)$ for short ("Big-Oh of N cubed")
Big-O

Say an algorithm runs $0.4N^3 + 25N^2 + 8N + 17$ statements.

• Example:
  
  • $N = 1$ trillion $= 1,000,000,000,000$
  
  • $N^3 = 1$ undecillion $= 1,000,000,000,000,000,000,000,000,000,000,000,000$

• The constants and lower-order terms don’t matter! The highest-order term ($N^3$) dominates the overall runtime.

• We say that this algorithm runs "on the order of" $N^3$.

• or $O(N^3)$ for short ("Big-Oh of $N$ cubed")
public static int sum1(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    return sum;
}

public static int sum2(int n) {
    return n * (n + 1) / 2;
}
Sum: Big-O

// Original sum2 implementation
public static int sum2(int n) {
    return n * (n + 1) / 2; 1
}

// Another sum2 implementation
public static int sum2(int n) {
    int temp1 = n + 1;
    int temp2 = n * temp1;
    int temp3 = temp2 / 2;
    return temp3;
}

O(1)

O(1)
What is the Big-O efficiency of this method?

```
public void method(int n) {
    int value = 0;
    for (int i = 0; i < 7; i++) {
        for (int j = 0; j < n; j++) {
            value += j;
        }
    }
    return value + n / 2;
}
```

- $O(1)$
- $O(n)$
- $O(7n)$
- $O(7n + 4)$
- $O(n^2)$
- $O(n^3)$
# Complexity Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Big-O</th>
<th>If you double N...</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>$O(1)$</td>
<td>unchanged</td>
</tr>
<tr>
<td>logarithmic</td>
<td>$O(\log N)$</td>
<td>increases slightly</td>
</tr>
<tr>
<td>linear</td>
<td>$O(N)$</td>
<td>doubles</td>
</tr>
<tr>
<td>log linear</td>
<td>$O(N \log N)$</td>
<td>slightly more than doubles</td>
</tr>
<tr>
<td>quadratic</td>
<td>$O(N^2)$</td>
<td>quadruples</td>
</tr>
<tr>
<td>cubic</td>
<td>$O(N^3)$</td>
<td>multiplies by 8</td>
</tr>
<tr>
<td>exponential</td>
<td>$O(2^N)$</td>
<td>multiplies drastically</td>
</tr>
</tbody>
</table>
# Complexity Comparison

<table>
<thead>
<tr>
<th>N (input size)</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>100 ms</td>
<td>100 ms</td>
<td>100 ms</td>
<td>100 ms</td>
<td>100 ms</td>
<td>100 ms</td>
<td>100 ms</td>
</tr>
<tr>
<td>200</td>
<td>100 ms</td>
<td>115 ms</td>
<td>200 ms</td>
<td>240 ms</td>
<td>400 ms</td>
<td>800 ms</td>
<td>32.7 sec</td>
</tr>
<tr>
<td>400</td>
<td>100 ms</td>
<td>130 ms</td>
<td>400 ms</td>
<td>550 ms</td>
<td>1.6 sec</td>
<td>6.4 sec</td>
<td>12.4 days</td>
</tr>
<tr>
<td>800</td>
<td>100 ms</td>
<td>145 ms</td>
<td>800 ms</td>
<td>1.2 sec</td>
<td>6.4 sec</td>
<td>51.2 sec</td>
<td>36.5 million years</td>
</tr>
<tr>
<td>1600</td>
<td>100 ms</td>
<td>160 ms</td>
<td>1.6 sec</td>
<td>2.7 sec</td>
<td>25.6 sec</td>
<td>6 min 49.6 sec</td>
<td>42.1 * 10^{24} years</td>
</tr>
<tr>
<td>3200</td>
<td>100 ms</td>
<td>175 ms</td>
<td>3.2 sec</td>
<td>6 sec</td>
<td>1 min 42.4 sec</td>
<td>54 min 36 sec</td>
<td>5.6 * 10^{61} years</td>
</tr>
</tbody>
</table>
Sequential Search

• Remember writing `indexOf` in `ArrayIntList`?
• Sequential search: start at the beginning, examine each element until you find what you want (or reach the end)

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| value | -4|  2|  7| 10| 15| 20| 22| 25| 30| 36| 42 | 50 | 56 | 68 | 85 | 92 | 103|

i
Sequential Search

• What is its complexity class (Big-O)?

```
public int indexOf(int value) {
    for (int i = 0; i < size; i++) {
        if (elementData[i] == value) {
            return i;
        }
    }
    return -1;  // not found
}
```
What if the array was sorted?

- How could we perform a faster search?

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>
Binary Search

- **Algorithm:** Examine the middle element of the array.
  - If it is too big, eliminate the right half of the array and repeat.
  - If it is too small, eliminate the left half of the array and repeat.
  - Else it is the value we're searching for, so stop.
Binary Search Time Complexity

• For an array of size \( N \), it eliminates \( \frac{1}{2} \) until 1 element remains.
  • \( N, N/2, N/4, N/8, \ldots, 4, 2, 1 \)
  • How many divisions does it take?

• Think of it from the other direction: How many times do I have to multiply by 2 to reach \( N \)?
  • 1, 2, 4, 8, \ldots, N/4, N/2, N
  • Call this number \( x \)

\[
2^x = N
\]

\[
x = \log_2 N
\]

\( \bigO(\log N) \)
Time Complexity of Collections

contains(value)

- ArrayList: $O(N)$
- LinkedList: $O(N)$
- TreeSet: $O(\log N)$
- HashSet: $O(1)$
countUnique – Using a List

public static int countUnique(Scanner input) {
    List<String> list = new ArrayList<>();
    while (input.hasNext()) {
        String word = input.next();
        if (!list.contains(word)) {
            list.add(word);
        }
    }
    return list.size();
}

What is the time complexity of this code?

$N = \# \text{ words}$

$1 + 2 + 3 + \ldots + N = \frac{N(N+1)}{2} = \frac{N^2 + N}{2}$

$O(N^2)$
countUnique – Using a Set

public static int countUnique(Scanner input) {
    Set<String> set = new HashSet<>();
    while (input.hasNext()) {
        String word = input.next();
        set.add(word);
    }
    return set.size();
}

What is the time complexity of this code?

$O(N)$