CSE 12 – Basic Data Structures

Prof. Christine Alvarado
[Slides borrowed/adapted from slides by Cynthia Lee]
Announcements

1. HW2: How’s it going?
   A. I haven’t even read it
   B. I’ve read it, but haven’t started coding
   C. I’ve just started coding
   D. I’m well on my way
   E. I’m done!
Today’s Topics

1. Running time: How long does a program take to run
2. Best case and worst case analysis
3. Big-O, Big-Omega, Big-Theta
You have an unsorted array of integers, size 20.

You want to find the index where 15 is stored

How many places do you have to look?

A. 1
B. 10
C. 15
D. 20
E. Other/none/more than one
You have an unsorted array of integers, size 20: But I’m not telling you what they are

<table>
<thead>
<tr>
<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>
<th>17</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
</table>

- You want to find the index where 15 is stored
- How many places do you have to look in the **BEST** case?

A. 1  
B. 10  
C. 15  
D. 20  
E. Other/none/more than one
You have an unsorted array of integers, size 20

You want to find the index where 15 is stored

How many places do you have to look in the WORST case?

A. 1
B. 10
C. 15
D. 20
E. Other/none/more than one
You have an unsorted array of integers, size 20

A. 1
B. 10
C. 15
D. 20
E. Other/none/more than one

We also sometimes consider average case, but that's somewhat tricky.
Running time: What version of the problem are you analyzing

- One part of figuring out how long a program takes to run is figuring out how “lucky” you got in your input.
  - You might get lucky (best case), and require the least amount of time possible
  - You might get unlucky (worst case) and require the most amount of time possible
  - Or you might want to know “on average” (average case) if you are neither lucky or unlucky, how long does an algorithm take.

Almost always, what we care about is the WORST CASE or the AVERAGE CASE. Best case is usually not that interesting.

In CSE 12 when we do analysis, we are doing WORST CASE analysis unless otherwise specified.
boolean find( int[] theList, int toFind ) {
    for ( int i = 0; i < theList.length; i++ ) {
        if ( theList[i] == toFind )
            return true;
    }
    return false;
}

How many instructions do you have to execute to find out if the element is in the list in the worst case, if n represents the length of the list?
Analyzing the worst case

```java
boolean find( int[] theList, int toFind ) {
    for ( int i = 0; i < theList.length; i++ ) {
        if ( theList[i] == toFind )
            return true;
    }
    return false;
}

boolean slowFind( int[] theList, int toFind ) {
    for ( int i = 0; i < theList.length; i++ ) {
        System.out.println( "Looking for " + toFind );
        if ( theList[i] == toFind )
            return true;
    }
    return false;
}
```

Which method is faster?
A. find
B. find2
C. They are about the same
Analyzing the worst case

```java
boolean find( int[] theList, int toFind ) {
    for ( int i = 0; i < theList.length; i++ ) {
        if ( theList[i] == toFind )
            return true;
    }
    return false;
}

boolean fastFind( int[] theList, int toFind ) {
    return false;
}
```

Which method is faster?
A. find
B. find2
C. They are about the same
2n vs n vs 1
\( f_2 \) is* \( O(f_1) \)

A. TRUE
B. FALSE

Why or why not?

* You can't actually tell if you don't know the function, because it could do something crazy just off the graph, but we'll assume it doesn't.
Big-O

We say a function $f(n)$ is “big-O” of another function $g(n)$, and write $f(n) = O(g(n))$, if there are positive constants $c$ and $n_0$ such that:
- $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

In other words, for large $n$, can you multiply $g(n)$ by a constant and have it always be bigger than or equal to $f(n)$.
\[ f(n) = O(g(n)), \text{ if there are positive constants } c \text{ and } n_0 \text{ such that } f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0. \]

- Obviously \( f_2 = O(f_1) \) because \( f_1 > f_2 \) (after about \( n=10 \), so we set \( n_0 = 10 \))
- \( f_1 \) is clearly an \textit{upper bound} on \( f_2 \) and that’s what big-O is all about
\( f_1 \) is \( \mathcal{O}(f_2) \)

A. TRUE
B. FALSE

Why or why not?

In other words, for large \( n \), can you multiply \( f_2 \) by a constant and have it always be bigger than \( f_1 \)
\( f(n) = \mathcal{O}(g(n)), \) if there are positive constants \( c \) and \( n_0 \) such that \( f(n) \leq c \cdot g(n) \) for all \( n \geq n_0 \).

- Obviously \( f_2 = \mathcal{O}(f_1) \) because \( f_1 > f_2 \) (after about \( n=10 \), so we set \( n_0 = 10 \))
  - \( f_1 \) is clearly an upper bound on \( f_2 \) and that’s what big-O is all about
- But \( f_1 = \mathcal{O}(f_2) \) as well!
  - We just have to use the “\( c \)” to adjust so \( f_2 \) that it moves above \( f_1 \)
f(n) = O(g(n)), if there are positive constants c and n₀ such that f(n) ≤ c * g(n) for all n ≥ n₀.

- Obviously f₂ = O(f₁) because f₁ > f₂ (after about n=10, so we set n₀ = 10)
  - f₁ is clearly an upper bound on f₂ and that’s what big-O is all about
- But f₁ = O(f₂) as well!
  - We just have to use the “c” to adjust so f₂ that it moves above f₁
$f_1$ is $\Omega(f_2)$

A. TRUE  
B. FALSE

Why or why not?
We say a function $f(n)$ is “big-omega” of another function $g(n)$, and write $f(n) = \Omega(g(n))$, if there are positive constants $c$ and $n_0$ such that:

- $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

In other words, for large $n$, can you multiply $g(n)$ by a constant and have it always be smaller than or equal to $f(n)$.
$f_2$ is $\Omega(f_1)$

A. TRUE  
B. FALSE

Why or why not?

In other words, for large $n$, can you multiply $f_2$ by a positive constant and have it always be smaller than $f_1$
$f_1$ is $\Theta(f_2)$

A. TRUE

B. FALSE

Why or why not?
$f_1$ is $O(f_3)$

A. TRUE
B. FALSE

Why or why not?
$f_1$ is $\Theta(f_3)$

A. TRUE
B. FALSE

Why or why not?
\( f_1 = O(f_3) \text{ but } f_3 \neq O(f_1) \)

- There is no way to pick a \( c \) that would make an \( O(n) \) function \( f_1 \) stay above an \( O(n^2) \) function \( f_3 \).
Shortcuts for calculating

Big-O analysis starting with a function characterizing the growth in cost of the algorithm
Let $f(n) = 3 \log_2 n + 4 n \log_2 n + n$

Which of the following is true?

A. $f(n) = O(\log_2 n)$
B. $f(n) = O(n \log_2 n)$
C. $f(n) = O(n^2)$
D. $f(n) = O(n)$
E. Other/none/more
Let $f(n) = 546 + 34n + 2n^2$

Which of the following is true?

A. $f(n) = O(2^n)$
B. $f(n) = O(n^2)$
C. $f(n) = O(n)$
D. $f(n) = O(n^3)$
E. Other/none/more
Let \( f(n) = 2^n + 14n^2 + 4n^3 \)

Which of the following is true?

A. \( f(n) = O(2^n) \)
B. \( f(n) = O(n^2) \)
C. \( f(n) = O(n) \)
D. \( f(n) = O(n^3) \)
E. Other/none/more
Let $f(n) = 100$

Which of the following is true?

A. $f(n) = O(2^n)$
B. $f(n) = O(n^2)$
C. $f(n) = O(n)$
D. $f(n) = O(n^{100})$
E. Other/none/more
Next time

- More running time analysis, and calculating it from code