Time, Asymptotics, and Recursion

Instructors: Sam McCauley and Dan Barowy
February 23, 2022
Admin

- Pairs assigned for lab 3 (randomly)

- Some office hours today if you’re still finishing up lab 2

- Important place in the course for you to check in if you’re feeling behind
Time and Space Analysis
How efficient is a given method?

- We saw how to do `contains` in a `Vector`. How many items did we have to look through in the worst case?

- Let's say I'm looking through a literal dictionary. Is my `contains()` method very efficient? Do you have a faster way?

- What if I say I'm a really fast reader. Is your method still faster?
  - Probably
  - Unless the dictionary is really short. A fast reader may be able to read through a dictionary with 10 elements better than a more clever search method

- Idea here: analyze the efficiency of a methodology. Your speed—or your computer’s speed—shouldn’t be a factor.
What do we mean by efficiency?

- Perhaps: how long does a method take to run in seconds?

- How much space does it take? (How many bits do we need to store on our computer during the calculation)?
Algorithmic Efficiency

- We are looking for worst-case guarantees

- When you write a piece of code, the goal here is to say “I promise that my code will always run efficiently.”
  
  - It’s a much more widely applicable statement than “I tested my code out and it seems to run efficiently.”
  
  - What if your tests didn’t take into account a key scenario?
The Challenge of Analyzing Time

- Different computers run at different speeds

- Computers are complicated! Adding two numbers together (for example) can take drastically different times depending on context.

- Good news: often times these details don’t change much

- **Example:** It doesn’t matter (too much) how fast I read if I’m scanning thousands of extra dictionary pages.
Counting up time

• When we look at some Java code, how can we estimate how fast it is?

• Let’s look at the operations the code requires
  • By operations, I mean built-in operations like +, -, ==, if, =, array operations, etc.
  • If any methods are called, should count up their operations as well

• If we sum the time of all operations, we can figure out how long the code takes.

• Let’s do a quick example
Counting up time example

```java
int i = 0;  \text{c1 time (integer variable assignment)}
int count = 0; \text{c2 time (integer variable assignment)}
while(i < arr.length) { \text{c3 time (accessing length and comparing)}
    count += arr[i]; \text{c4 time (array access, addition, and assignment)}
    i++; \text{c5 time (variable assignment and addition)}
}
```

In total, this code takes time \textit{at most}:

\[
c_1 + c_2 + (c_3 + c_4 + c_5) \cdot \text{arr.length}
\]
Counting up time comparison

```c
int count = arr[0] % 27; c6 time
```

In total, this code takes time at most $c_6$.

If our array is at all large, this is going to be faster than the loop on any computer.
Let's formalize this

Goal in analyzing efficiency:

• We don’t care that much about constants

• We care about scaling: what happens when the data in question is fairly large?

• Big-O notation: way of comparing two running times with this in mind
Definition 1

$f(n)$ is $O(g(n))$ if there exist constants $c > 1$ and $n_0$ such that for all $n > n_0$,

$$f(n) \leq c \cdot g(n)$$

That is to say: If $n$ is large enough ($n > n_0$), then ignoring constants (we compare to $c \cdot g(n)$), then $g(n)$ is larger.
Plotting Big-O

Let $g(n)$ be the blue function, and $f(n)$ be the orange function. If $g(n)$ and $f(n)$ continue increasing in the same way, then $f(n) = O(g(n))$. 
Let $g(n)$ be the blue function, and $f(n)$ be the orange function; assume that $f(n)$ is bounded above by the dotted line. Since $f(n) < c \cdot g(n)$, we still have $f(n) = O(g(n))$. 
Continued from last slide: once we multiply $g$ by a constant, we obtain the plot shown in red; this is larger than $f$. 
Proving Big-O

Reminder: \( f(n) \) is \( O(g(n)) \) if there exist constants \( c > 1 \) and \( n_0 \) such that for all \( n > n_0 \), \( f(n) \leq c \cdot g(n) \)

- Let’s say we have two functions \( f(n) \) and \( g(n) \), and we want to show that \( f(n) \) is \( O(g(n)) \).

- We need to come up with a \( c > 1 \) and an \( n_0 \) such that for all \( n > n_0 \), \( f(n) \leq c \cdot g(n) \)
Counting up time example

```java
int i = 0;  c1 time
int count = 0; c2 time
while(i < arr.length) { c3 time
    count += arr[i]; c4 time
    i++;  c5 time
}
```

Let's define \( n = \text{arr.length} \). That is: we are analyzing the running time in terms of the array length.

In total, this code takes time at most:

\[
f(n) = c_1 + c_2 + (c_3 + c_4 + c_5)n
\]

Let's prove that this is \( O(n) \).
Counting up time example

\[ f(n) = c_1 + c_2 + (c_3 + c_4 + c_5)n \]

Let's prove that \( f(n) = O(n) \).

Let's set \( n_0 = 1 \) and \( c = c_1 + c_2 + c_3 + c_4 + c_5 \). Then we want to show that for all \( n > 1 \), \( f(n) \leq c \cdot g(n) \):

\[
\begin{align*}
  f(n) &= c_1 + c_2 + (c_3 + c_4 + c_5)n \\
  &\leq c_1 n + c_2 n + (c_3 + c_4 + c_5)n \\
  &= (c_1 + c_2 + c_3 + c_4 + c_5)n \\
  &= c \cdot n \\
  &= c \cdot g(n)
\end{align*}
\]

So for \( n > n_0 \), \( f(n) \leq c \cdot g(n) \); therefore, \( f(n) = O(n) \)
Counting up time example

```java
int i = 0;  \text{c1 time}
int count = 0;  \text{c2 time}
while(i < arr.length) {  \text{c3 time}
    count += arr[i];  \text{c4 time}
    i++;  \text{c5 time}
}
```

This code takes $O(n)$ time.
Simplifying

• None of the $c_i$ really mattered in the above analysis

• What we want to do: count the *number of operations* in a code segment

• Don’t need to be too careful about it: `count += arr[i]` counting as 1 operation or 3 operations isn’t going to change our final result

• Let’s consider the above code again
Counting up time example

```java
int i = 0;  2 operations
int count = 0;
while(i < arr.length) {  1 operation
    count += arr[i];  4 operations
    i++;
}
```

This code takes $2 + 5n$ operations. Since each operation takes constant time, this is $O(n)$ time.
Counting up time comparison

\[ \text{int count} = \text{arr}[0] \ % \ 27; \ 1 \text{ operation} \]

If \( n \) is the length of the array, how long does this code take?

It takes a constant number of operations. \( O(1) \) time.

(Note: \( O(1) \) is means \textit{bounded above by a constant}—this function does not get larger as \( n \) increases. How does this relate to the definition of big-\( O \)?)
Wrapping Up Asymptotics

- We want to count the amount of time taken by a method.
- Our analysis should apply regardless of how fast the computer is.
- Idea: Look at how many operations are used. Use big-O notation.
  - Ignore constants
  - Only care about sufficiently large inputs.
Recursion
Recursion

- We’ve seen methods call other methods in Java
- Methods can also call themselves. This is called recursion
- Works just like any other method call! Execution continues from the beginning of the method; goes back to previous point after it returns
- Recursion allows for *simpler* and *clearer* code in some cases
  - But not in others
  - Anything that can be solved with recursion can be solved without recursion
  - It’s just one more tool in your toolbox
Classic example: factorial

- The factorial function, written $n!$, is useful in combinatorics
  - (It counts the number of ways to order $n$ objects.)

- $n!$ is the product of the first $n$ numbers:
  \[ n! = n \cdot (n - 1) \cdot (n - 2) \ldots 3 \cdot 2 \cdot 1 \]

- So $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$.

- Can also define $n! = n \cdot (n - 1)!$
Two implementations of factorial

```java
public static int factorial(int n) {
    int ret = 1;
    for(int current = n; current >= 1; current--) {
        ret *= current;
    }
    return ret;
}
```

```java
public static int factorial(int n) {
    if(n == 1) {
        return 1;
    }
    return n * factorial(n-1);
}
```
Factorial discussion

- Which of these methods is better?
  - A matter of taste
- What are some advantages of the recursive method? What are some disadvantages?
  - If this method calls itself, why doesn’t it loop forever?
Recipe for Recursion

- Need a base case: on a sufficiently small input, can easily return the correct solution without a recursive call.

- If we’re not in the base case, can split into smaller instances of the same problem.
Searching a (Physical) Dictionary

- We agreed that my dictionary lookup method wasn’t very effective

- Can we describe a lookup methodology that works faster?

- (This is a method for humans, not code: let’s just talk about how it works on the board.)
Binary Search

- Recursive algorithm for searching in a sorted list

- Can be implemented without recursion! That is to say: you can implement a perfectly good binary search method with a loop instead of recursion

- We’ll be seeing a lot more of binary search soon.
Helper methods

- May be helpful to use extra information when recursing
  - We didn’t just search in the dictionary, we kept track of which *portion* of the dictionary we were recursing on

- Can create *helper methods* that have more parameters

- Let’s say we want to search a dictionary. We can “help” using a method that searches a portion of a dictionary.
Another Recurion Example: Scheduling
Let’s say there are 6 enrolled students enrolled in a course. I want to schedule office hours so that every single student has a chance to attend office hours.

I create a doodle poll with 10 options for my office hours.

Each student states which of the office hours they can attend.

What is the minimum number of office hours I can hold so that every student can make at least one hour?
Creating Office Hours

The possible time slots are \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}.

- Student 1 can make slots \{1, 6, 8\}
- Student 2 can make slots \{2, 5, 8\}
- Student 3 can make slots \{3, 4, 9, 10\}
- Student 4 can make slots \{6, 7, 8, 9\}
- Student 5 can make slots \{2, 3, 4\}
- Student 6 can make slots \{1, 3, 4, 5, 9\}
Creating Office Hours

The possible time slots are \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}.

- Student 1 can make slots \{1, 6, 8\}
- Student 2 can make slots \{2, 5, 8\}
- Student 3 can make slots \{3, 4, 9, 10\}
- Student 4 can make slots \{6, 7, 8, 9\}
- Student 5 can make slots \{2, 3, 4\}
- Student 6 can make slots \{1, 3, 4, 5, 9\}

This is solvable with 3 slots. (I think that’s optimal?)
Solving the Office Hours Problem Recursively

• Where to start?

• Can someone come up with a base case?
  • When there’s only one time slot, our only choice is to take it or not take it
  • Second base case option: if there is one student, if they have an hour that matches with a time slot, then 1 slot is optimal. Otherwise, can’t solve.
  • Another option: zero students or zero time slots
Office Hours Scheduling: Breaking into a Smaller Subproblem

• How can we make this subproblem smaller?

• Let’s look at the first possible time slot

• There are two options: either this time slot is in the solution, or it isn’t

  • Let’s assume we take the first time slot. Then we can remove that time slot from our list, and remove all students who can attend that time slot. That gives us a new instance of office hours scheduling!

  • Let’s assume we don’t take the first time slot. Then we can remove that time slot from our list. That gives us a new instance of office hours scheduling!
Office Hours Scheduling Solution

- If there is only remaining slot, just determine if it meets all students’ needs. Return 1 if so; −1 otherwise.
- Otherwise:
  - Recursively find the office hours scheduling solution with the first slot removed, and with all students whose availability matches that slot removed. Store this optimal solution in solWithSlot
  - Recursively find the office hours scheduling solution with the first slot removed. Store this optimal solution in solWithOutSlot
  - If both solWithSlot and solWithOutSlot are not −1, return the minimum of 1 + solWithSlot and solWithOutSlot
  - If just one is −1, return the other
  - If both are −1, return −1.
Discussion

- Why does this method work? What do we need to guarantee for a recursion to terminate?
  - Need to make progress towards the base case!
  - Each recursive call reduces the number of slots by 1

- Is this method fast? Is that OK?
  - No, this is not fast at all.
  - In algorithms you will learn that this problem is computationally intensive—there’s no solution that’s efficient and always correct