Lec 13: Recursion Contd.

Sam McCauley

March 12, 2026

Admin

- Lab today!
    - Not due until tonight
    - Start early!

Tracing Execution of Recursive Programs

Example: `printNums`

```java
public static void printNums(int num) {
    if (num == 0) {
        return;
    }
    System.out.println(num);
    printNums(num - 1);
}
```

- What does `printNums(3)` print?
- Prints `num`, then calls itself with `num - 1`
- Output: 3, 2, 1 (descending)

Example 2: `printNumsAgain`

```
1 public static void printNumsAgain(int num) {
2     if (num == 0) {
3         return;
4     }
5     printNumsAgain(num - 1);
6     System.out.println(num);
7 }
```

- Same code, but `println` is *after* the recursive call

- What does `printNumsAgain(4)` print?

Tracing `printNumsAgain(4)`

- `printNumsAgain(4)` first calls `printNumsAgain(3)`, which calls `printNumsAgain(2)`, ...all the way to the base case
  - No printing yet!

- Then, as each call returns, it prints *its own* `num`

- Let's track these calls on the board. As each recursive call is made we'll write it down, and cross it out as each completes.

- Output: 1, 2, 3, 4

Binary Search

- How can we do `contains(int element, int start, int end)` on a *sorted* `ArrayList<Integer>`?

- Does someone have an idea for a high-level strategy?
  - Compare to element in middle slot: `(end + start)/2`. Recurse on one side or the other depending on if it is larger or smaller

  - In pairs: how can we write this method recursively?

  - At home: can you write this method with a loop instead? How does the code change?

A scheduling problem: Creating Office Hours

- 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 $\{1, \mathbf{2}, 3, 4, 5, \mathbf{6}, 7, 8, \mathbf{9}, 10\}$.

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

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
  - What happens in each case? How do we make a recursive call?
  - 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 one 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 $_{\text{known}}$ solution that's efficient and always correct

Creating Office Hours

Let's look at what our algorithm does on a smaller instance.

The possible time slots are $\{1, 2, 3\}$.

- Student 1 can make slots $\{1, 2\}$
- Student 2 can make slots $\{2, 3, \}$
- Student 3 can make slots $\{3\}$
- Student 4 can make slots $\{1\}$

Smaller example

Let's look at what our algorithm does on a smaller instance.
The possible time slots are $\{1, 2, 3\}$.

- Student 1 can make slots $\{1, 2\}$
- Student 2 can make slots $\{2, 3, \}$
- Student 3 can make slots $\{3\}$
- Student 4 can make slots $\{1\}$

- If there is only one remaining slot, determine if it suffices for all students; otherwise:
    - Recursively find solution with the first slot removed, and with all students whose availability matches that slot removed.
    - Recursively find solution with the first slot removed. Store this optimal solution in `solWithOutSlot`

Time and Space Analysis

How efficient is a given method?

- We saw how to do `contains()` and `indexOf()` in an unsorted `ArrayList`. 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.
- This is why we focused on traversals until now.