Thinking Recursively
Part IV
Outline for Today

● *Recap From Last Time*
  ● Where are we, again?

● *More on Tug-of-War*
  ● Addressing some points from last time.

● *Shrinkable Words*
  ● A little word puzzle!
Recap from Last Time
Enumeration and Optimization

- An *enumeration* problem is one where the goal is to list all objects of some type.
- An *optimization* problem is one where the goal is to find the best object of some type.
- If you can enumerate all solutions to a problem, with a few quick code tweaks you can convert what you have into a solution to an optimization problem.
You want to organize a tug-of-war match as a morale-building exercise for your team.

You’d like the match to be as fair as possible, and you have a rough estimate of how much force everyone can pull with.

What’s the fairest way to divvy people up into teams?
List all ways to split \{A, B, C\} into two teams.
New Stuff!
Answering Your Questions
Question 1:

What happens if we make a bad decision early on? Won’t we be stuck committed to the wrong solution?
Teams bestTeamsRec(const Set<Person>& remaining, const Teams& soFar) {
    if (remaining.isEmpty()) {
        return soFar;
    } else {
        Person curr = remaining.first();

        /* Option 1: Put this person on Team 1. */
        Teams best1 = bestTeamsRec(remaining – curr, {
            soFar.one + curr, soFar.two });

        /* Option 2: Put this person on Team 2. */
        Teams best2 = bestTeamsRec(remaining – curr, {
            soFar.one, soFar.two + curr });

        if (imbalanceOf(best1) < imbalanceOf(best2)) {
            return best1;
        } else {
            return best2;
        }
    }
}
Teams bestTeamsRec(const Set<Person>& remaining, const Teams& soFar) {
    if (remaining.isEmpty()) {
        return soFar;
    } else {
        Person curr = remaining.first();

        /* Option 1: Put this person on Team 1. */
        Teams best1 = bestTeamsRec(remaining - curr,
                                    { soFar.one + curr, soFar.two });

        /* Option 2: Put this person on Team 2. */
        Teams best2 = bestTeamsRec(remaining - curr,
                                    { soFar.one, soFar.two + curr });

        if (imbalanceOf(best1) < imbalanceOf(best2)) {
            return best1;
        } else {
            return best2;
        }
    }
}
Perspective 1: *Trace the Recursion*
```java
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr, soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
```
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr, soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr, soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
                              { soFar.one + curr,  soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
                              { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr,  soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two +  curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr, { soFar.one + curr, soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr, { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr, soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr, soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr, soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr,  soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two +  curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr, soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr, soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
                                { soFar.one + curr,  soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
                                { soFar.one, soFar.two +  curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr, soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr,  soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
```java
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr, soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
```
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();

    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr, soFar.two });

    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two + curr });

    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr, soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr,  soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
  return soFar;
} else {
  Person curr = remaining.first();
  Teams best1 = bestTeamsRec(remaining – curr,
      { soFar.one + curr, soFar.two });
  Teams best2 = bestTeamsRec(remaining – curr,
      { soFar.one, soFar.two + curr });
  if (imbalanceOf(best1) < imbalanceOf(best2)) {
    return best1;
  } else {
    return best2;
  }
}
if (remaining.isEmpty()) {
  return soFar;
} else {
  Person curr = remaining.first();
  Teams best1 = bestTeamsRec(remaining – curr,
    { soFar.one + curr, soFar.two });
  Teams best2 = bestTeamsRec(remaining – curr,
    { soFar.one, soFar.two + curr });
  if (imbalanceOf(best1) < imbalanceOf(best2)) {
    return best1;
  } else {
    return best2;
  }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr, soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining - curr,
        { soFar.one + curr, soFar.two });
    Teams best2 = bestTeamsRec(remaining - curr,
        { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr, soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
if (remaining.isEmpty()) {
    return soFar;
} else {
    Person curr = remaining.first();
    Teams best1 = bestTeamsRec(remaining – curr,
        { soFar.one + curr, soFar.two });
    Teams best2 = bestTeamsRec(remaining – curr,
        { soFar.one, soFar.two + curr });
    if (imbalanceOf(best1) < imbalanceOf(best2)) {
        return best1;
    } else {
        return best2;
    }
}
Teams bestTeamsRec(const Set<Person>& remaining, 
    const Teams& soFar) {
    if (remaining.isEmpty()) {
        return soFar;
    } else {
        Person curr = remaining.first();

        /* Option 1: Put this person on Team 1. */
        Teams best1 = bestTeamsRec(remaining - curr, 
            { soFar.one + curr, soFar.two });

        /* Option 2: Put this person on Team 2. */
        Teams best2 = bestTeamsRec(remaining - curr, 
            { soFar.one, soFar.two + curr });

        if (imbalanceOf(best1) < imbalanceOf(best2)) {
            return best1;
        } else {
            return best2;
        }
    }
}
Teams bestTeamsRec(const Set<Person>& remaining, const Teams& soFar) {
  if (remaining.isEmpty()) {
    return soFar;
  }
  Person curr = remaining.first();

  /* Option 1: Put this person on Team 1. */
  Teams best1 = bestTeamsRec(remaining – curr,
                               { soFar.one + curr, soFar.two });

  /* Option 2: Put this person on Team 2. */
  Teams best2 = bestTeamsRec(remaining – curr,
                               { soFar.one, soFar.two + curr });

  if (imbalanceOf(best1) < imbalanceOf(best2)) {
    return best1;
  }
  else {
    return best2;
  }
}

This just kicks the answer one level higher up. It doesn’t end the recursive exploration.
Perspective 2: *Think Abstractly*
Without looking at the implementation, can you explain what this function does?
Teams bestTeamsRec(const Set<Person>& remaining, const Teams& soFar);

... you can make from these people ...

... given that some people are already placed on those teams?
Teams bestTeamsRec(const Set<Person>& remaining, const Teams& soFar) {
    if (remaining.isEmpty()) {
        return soFar;
    } else {
        Person curr = remaining.first();

        /* Option 1: Put this person on Team 1. */
        Teams best1 = bestTeamsRec(remaining – curr,
                                    { soFar.one + curr, soFar.two });

        /* Option 2: Put this person on Team 2. */
        Teams best2 = bestTeamsRec(remaining – curr,
                                    { soFar.one, soFar.two + curr });

        if (imbalanceOf(best1) < imbalanceOf(best2)) {
            return best1;
        } else {
            return best2;
        }
    }
}
Teams bestTeamsRec(const Set<Person>& remaining,
        const Teams& soFar) {
    if (remaining.isEmpty()) {
        return soFar;
    } else {
        Person curr = remaining.first();

        /* Option 1: Put this person on Team 1. */
        Teams best1 = bestTeamsRec(remaining – curr,
                                    { soFar.one + curr, soFar.two });

        /* Option 2: Put this person on Team 2. */
        Teams best2 = bestTeamsRec(remaining – curr,
                                    { soFar.one, soFar.two + curr });

        if (imbalanceOf(best1) < imbalanceOf(best2)) {
            return best1;
        } else {
            return best2;
        }
    }
}

What are the best teams you can form if everyone is already assigned to a team?
Thinking Recursively

• When writing recursive functions, always ask yourself this question:

  *Can you describe what the function does purely by reading the signature?*

• If so, great! That will guide your coding effort.

• If not, pause and think it through a bit. It’s hard to write a function correctly when you can’t explain what it’s supposed to do!
Question 2:
We’re generating duplicate solutions!
How do we fix that?
List all ways to split \{A, B, C\} into two teams.

{A, B, C}
{}
Breaking Symmetry

• In many enumeration and optimization problems, there may be many solutions that are equivalent to one another.
  • Here, swapping Team 1 and Team 2 doesn’t change anything.

• In some cases, you can break symmetries by committing to some fixed decision up front.
  • Here, forcing the first person to be on Team 1.

• In others, you’ll need to rethink your recursive approach.
  • For example, finding a different decision tree.
Question 3:
That code is really long! Can we make it shorter and prettier?
The Wonderful **auto** Keyword

- In C++, you need to assign a type to each variable.

- In the case where you define a variable and give it an initial value, you can write **auto** instead of the name of a type to have C++ figure out the type for you.

  ```
  auto variable = expression;
  ```

- Use this when you are declaring a variable whose value can unambiguously be determined from the expression initializing it.
The Wonderful ?: Operator

• In C++, the *ternary conditional operator* can be used to select one of two expressions.

• The syntax is

\[
\text{expression} \ ? \ \text{if-true} : \ \text{if-false}
\]

• This shows up all the time in recursive optimization problems.
Question 4:

Why do we even need recursion at all here? Can’t we just iterate over the combinations and take the best?

Great exercise: Solve this problem without using recursion. How will you enumerate all the possible ways of splitting folks into teams?
Time-Out for Announcements!
Research Office Hours

- Two of our amazing PhD students – including one who’s a former section leader – are holding research office hours twice a week.
- Have questions about what it’s like to do research in CS? Head to *Gates B02* on
  
  **Mondays, 1:30PM - 2:30PM**

  or

  **Fridays, 10:00AM - 11:00AM.**
• Stanford’s Society of Women Engineers (SWE) is hosting a conference on diversity in engineering.

• Includes a keynote by the Provost and a pretty impressive panel!

• It’s this upcoming Saturday, February 2nd from 10:00AM – 3:00PM in the d.school.

• RSVP using this link.
lecture.resume();

(The old, janky Java way of telling a thread that’s been paused to start again. Basically no one uses this syntax any more.)
A Little Word Puzzle
“What nine-letter word can be reduced to a single-letter word one letter at a time by removing letters, leaving it a legal word at each step?”
The Startling Truth

STARTLING
The Startling Truth
The Startling Truth

STARING
The Startling Truth

STRING
The Startling Truth

STING
The Startling Truth

S I N G
The Startling Truth
The Startling Truth
The Startling Truth
Is there really just one nine-letter word with this property?
"Cart" is shrinkable… because "art" is shrinkable …

... because "at" is shrinkable …

... because "a" is a single-letter word.
All Possible Paths
All Possible Paths

“Up” is not shrinkable... because neither P nor U are words.
All Possible Paths

“Cup” is not shrinkable…

... because none of these are shrinkable words.

CUP

USP

US

SP

UP

CUS

US

CS

CU

CUSP

UP

CP

CU

CUS
"Cusp" is not shrinkable…

… because none of these are shrinkable words.
Shrinkable Words

- Let's define a *shrinkable word* as a word that can be reduced down to one letter by removing one character at a time, leaving a word at each step.

- **Base Cases:**
  - A string that is not a word is not a shrinkable word.
  - Any single-letter word is shrinkable (A, I, and O).

- **Recursive Step:**
  - A multi-letter word is shrinkable if you can remove a letter to form a shrinkable word.
  - A multi-letter word is not shrinkable if no matter what letter you remove, it’s not shrinkable.
Your Action Items

- **Read Chapter 9 of the textbook.**
  - There’s tons of cool backtracking examples there, and it will help you prep for Friday.

- **Keep working on Assignment 3.**
  - If you’re following our timetable, you should be done with the Sierpinski triangle at this point and have started Human Pyramids.
  - Aim to complete Human Pyramids and to have started work on Shift Scheduling by Friday.
Next Time

- **More Backtracking**
  - Techniques in searching for feasibility.
- **Closing Thoughts on Recursion**
  - It’ll come back, but we’re going to focus on other things for a while!