The Knapsack Problem

An example of recursive optimization

"Hard" Problems

"Hard" Problems

- There are many different categories of problems in computer science that are considered to be "hard" to solve.
    - Formally, these are known as "NP-hard" problems. Take CS103 to learn more!

"Hard" Problems

- There are many different categories of problems in computer science that are considered to be "hard" to solve.
  - Formally, these are known as "NP-hard" problems. Take CS103 to learn more!

- For these categories of problems, there exist no known "good" or "efficient" ways to generate the best solution to the problem. The only known way to generate an exact answer is to **try all possible solutions** and select the best one.
  - Often times these problems involve finding permutations (`O(n!)` possible solutions) or combinations (`O(2^n)` possible solutions)

"Hard" Problems

- There are many different categories of problems in computer science that are considered to be "hard" to solve.
    - Formally, these are known as "NP-hard" problems. Take CS103 to learn more!

- For these categories of problems, there exist no known "good" or "efficient" ways to generate the best solution to the problem. The only known way to generate an exact answer is to **try all possible solutions** and select the best one.
    - Often times these problems involve finding permutations (`O(n!)` possible solutions) or combinations (`O(2^n)` possible solutions)

- **Backtracking recursion is an elegant way to solve these kinds of problems!**

Using backtracking recursion

- There are 3 main categories of problems that we can solve by using backtracking recursion:
  - We can generate all possible solutions to a problem or count the total number of possible solutions to a problem
  - We can find one specific solution to a problem or prove that one exists
  - **We can find the best possible solution to a given problem**
- There are many, many examples of specific problems that we can solve, including
  - Generating permutations
  - **Generating subsets (with additional constraints!)**
  - Generating combinations
  - And many, many more

The Knapsack Problem

The Knapsack Problem

- Imagine yourself in a new lifestyle as a professional wilderness survival expert

The Knapsack Problem

- Imagine yourself in a new lifestyle as a professional wilderness survival expert

The Knapsack Problem

- Imagine yourself in a new lifestyle as a professional wilderness survival expert

- You are about to set off on a challenging expedition, and you need to pack your knapsack (or backpack) full of supplies.

The Knapsack Problem

- Imagine yourself in a new lifestyle as a professional wilderness survival expert
- You are about to set off on a challenging expedition, and you need to pack your knapsack (or backpack) full of supplies.
- You have a list full of supplies (each of which has a survival value and a weight associated with it) to choose from.

The Knapsack Problem

- Imagine yourself in a new lifestyle as a professional wilderness survival expert
- You are about to set off on a challenging expedition, and you need to pack your knapsack (or backpack) full of supplies.
- You have a list full of supplies (each of which has a survival value and a weight associated with it) to choose from.
- Your backpack is only sturdy enough to hold a certain amount of weight.

The Knapsack Problem

- Imagine yourself in a new lifestyle as a professional wilderness survival expert
- You are about to set off on a challenging expedition, and you need to pack your knapsack (or backpack) full of supplies.
- You have a list full of supplies (each of which has a survival value and a weight associated with it) to choose from.
- Your backpack is only sturdy enough to hold a certain amount of weight.
- Question: How can you **maximize the survival value** of your backpack?

Solve a small knapsack example

The "Greedy" Approach

What happens if you always choose to include the item with the highest value that will still fit in your backpack?

Rope

- Value: 3
- Weight: 2

Axe

- Value: 4
- Weight: 3

Tent

- Value: 5
- Weight: 4

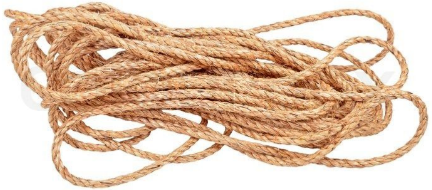
Canned food

- Value: 6
- Weight: 5

The "Greedy" Approach

What happens if you always choose to in hest value that will still fit in your backpack?

Bag is full!



Rope
- Value: 3
- Weight: 2

Axe
- Value: 4
- Weight: 3

Tent
- Value: 5
- Weight: 4

Canned food
- Value: 6
- **Weight: 5**

The "Greedy" Approach

What happens if you always choose to in hest value that will still fit in your backpack?

Why doesn't this work?

Rope

- Value: 3
- Weight: 2

Axe

- Value: 4
- Weight: 3

Tent

- Value: 5
- Weight: 4

Canned food

- Value: 6
- **Weight: 5**

The "Greedy" Approach

What happens if you always choose hest value that will still fit in your backpack?

Items with lower individual values may sum to a higher total value!

Rope

- **Value: 3**
- Weight: 2

Axe

- **Value: 4**
- Weight: 3

Tent

- Value: 5
- Weight: 4

Canned food

- Value: 6
- Weight: 5

The Recursive Approach

Idea: Enumerate all subsets of weight <= 5 and pick the one with best total value.

The Recursive Approach

Idea: **Enumerate all subsets of weight <= 5** and pick the one with best total value.

This is generating combinations!

How do we approach this problem?

Solving backtracking recursion problems

- **Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, pick one best solution)**
- **What are we building up as our “many possibilities” in order to find our solution?**

- What’s the provided function prototype and requirements? Do we need a helper function?
  - What are we returning as our solution?
  - Do we care about returning or keeping track of the path we took to get to our solution? If yes, what parameters are we already given and what others might be useful?

- What are our base and recursive cases?
  - What does my decision tree look like? (decisions, options, what to keep track of)
  - In addition to what we’re building up, are there any additional constraints on our solutions?
  - Does it make sense to use choose/explore/undo OR copy/edit/recurse for the recursion?

Using backtracking recursion

- There are 3 main categories of problems that we can solve by using backtracking recursion:
    - We can generate all possible solutions to a problem or count the total number of possible solutions to a problem
    - We can find one specific solution to a problem or prove that one exists
    - **We can find the best possible solution to a given problem**
- There are many, many examples of specific problems that we can solve, including
    - Generating permutations
    - Generating subsets
    - **Generating combinations**
    - And many, many more

The Recursive Approach

Idea: **Enumerate all combinations** and pick the one with best total value.

The Recursive Approach

Idea: Enumerate all combinations and **pick the one with best total value**.

Our final backtracking use case: "Pick one best solution"!

(i.e. optimization)

The Recursive Approach

Idea: Enumerate all combinations and **pick the one with best total value**.

We'll need to keep track of the total value we're building up, but for this version of the problem, we won't worry about finding the actual best subset of items itself.

Solving backtracking recursion problems

- Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, pick one best solution)
- What are we building up as our "many possibilities" in order to find our solution?

- **What's the provided function prototype and requirements? Do we need a helper function?**
    - What are we returning as our solution?
    - Do we care about returning or keeping track of the path we took to get to our solution? If yes, what parameters are we already given and what others might be useful?

- What are our base and recursive cases?
    - What does my decision tree look like? (decisions, options, what to keep track of)
    - In addition to what we're building up, are there any additional constraints on our solutions?
    - Does it make sense to use choose/explore/undo OR copy/edit/recurse for the recursion?

Problem Setup

```
int fillBackpack(Vector<BackpackItem>& items, int targetWeight);
```

- Assume that we have defined a custom **`BackpackItem`** struct, which packages up an item's **`survivalValue`** (int) and **`weight`** (int).
- We need to return the max value we can get from a combination of **`items`** under **`targetWeight`**.

Problem Setup

```
int fillBackpack(Vector<BackpackItem>& items, int targetWeight);
```

- Assume that we have defined a custom **`BackpackItem`** struct, which packages up an item's **`survivalValue`** (int) and **`weight`** (int).
- We need to return the max value we can get from a combination of **`items`** under **`targetWeight`**.

We need a helper function!

Pseudocode

- We need a helper function!

```
int fillBackpackHelper(Vector<BackpackItem>& items,
                          int capacityRemaining, int curValue);
```

Solving backtracking recursion problems

- Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, pick one best solution)
- What are we building up as our “many possibilities” in order to find our solution?

- What’s the provided function prototype and requirements? Do we need a helper function?
  - What are we returning as our solution?
  - Do we care about returning or keeping track of the path we took to get to our solution? If yes, what parameters are we already given and what others might be useful?

- **What are our base and recursive cases?**
  - What does my decision tree look like? (decisions, options, what to keep track of)
  - In addition to what we’re building up, are there any additional constraints on our solutions?
  - Does it make sense to use choose/explore/undo OR copy/edit/recurse for the recursion?

What defines our knapsack decision tree?

- **Decision** at each step (each level of the tree):
  - Are we going to include a given item in our combination?

- **Options** at each decision (branches from each node):
  - Include element
  - Don’t include element

- Information we need to store along the way:
  - The total value so far
  - The remaining elements to choose from
  - The remaining capacity (weight) in the backpack

What defines our knapsack decision tree?

- **Decision** at each step (each level of the tree):
  - Are we going to include a given item in our combination?

- **Options** at each decision (branches from each node):
  - Include element
  - Don’t include element

- Information we need to store along the way:
  - The total value so far
  - The remaining elements to choose from
  - The remaining capacity (weight) in the backpack

This should look very similar to our previous combinations problem!

Pseudocode

- **Recursive case**:
  - Select an unconsidered item.
  - Recursively calculate the values both with and without the item.
  - Return the higher value.

- **Base cases**:
  - No remaining capacity in the knapsack ➔ return 0
    (not a valid combination with weight <= 5)
  - No more items to choose from ➔ return current value

Let’s code it!

(if time allows)

Challenge extensions on knapsack

Challenge #1: Improving our efficiency

- For efficiency, we'll use an **`index`** to keep track of which items we've already looked at inside **`items`**:

```
int fillBackpackHelper(Vector<BackpackItem>& items,
                       int capacityRemaining, int curValue,
                       int index);
```

Our adjusted pseudocode

- **Recursive case**:
  - Select an unconsidered item **based on the index**.
  - Recursively calculate the values both with and without the item.
  - Return the higher value.

- **Base cases**:
  - No remaining capacity in the knapsack ➔ return 0
    (not a valid combination with weight <= 5)
  - No more items to choose from ➔ return current value

Challenge #2: Tracking our items

- What if we wanted to know what combination of items resulted in the best value?

- Think about which answers to which questions in our recursive backtracking strategy would change.

Takeaways

- Finding the best solution to a problem (optimization) can often be thought of as an additional layer of complexity/decision making on top of the recursive enumeration we've seen before

- For "hard" problems, the best solution can only be found by enumerating all possible options and selecting the best one.

- Creative use of the return value of recursive functions can make applying optimization to an existing function straightforward.