# DSC 190

## DATA STRUCTURES & ALGORITHMS

Lecture 10 | Part 1

## Today's Lecture

# Beyond Greedy

- Greedy algorithms are typically **fast**, but may not find the optimal answer.
- Brute force guarantees the optimal answer, but is **slow**.
- Can we guarantee the optimal answer and be faster than brute force?

# Today

- The **backtracking** idea.
- It is a useful, general algorithm design technique[1].
- And the foundation of **dynamic programming**.

[1] Commonly seen in tech interviews

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 10 | Part 2

# The 0-1 Knapsack Problem

# 0-1 Knapsack

- Suppose you're a thief.
- You have a knapsack (bag) that can fit 100L.
- And a list of *n* things to possibly steal.

| item | size (L) | price |
|---|---|---|
| TV | 50 | \$400 |
| iPad | 2 | \$900 |
| Printer | 10 | \$100 |
| ⋮ | ⋮ | ⋮ |

- Goal: maximize total value of items you can fit in your knapsack.

# Example

| item | size (L) | price |
|---|---|---|
| 1 | 50 | $40 |
| 2 | 10 | $25 |
| 3 | 80 | $100 |
| 4 | 5 | $10 |
| 5 | 20 | $20 |
| 6 | 30 | $6 |
| 7 | 8 | $32 |
| 8 | 17 | $34 |

In the bag: ______________

Total value: ______________

Space remaining: __________

# Greedy

- Does a greedy approach find the optimal?
- What do we mean by “greedy”?
- Idea #1: take most expensive available that will fit.

# Example

| item | size (L) | price |
|---|---|---|
| 1 | 50 | $40 |
| 2 | 10 | $25 |
| 3 | 80 | $100 |
| 4 | 5 | $10 |
| 5 | 20 | $20 |
| 6 | 30 | $6 |
| 7 | 8 | $32 |
| 8 | 17 | $34 |

In the bag: ____________

Total value: ____________

Space remaining: ________

# Greedy, Idea #2

- We want items with high value for their size.
- Define "price density" =
  `item.price / item.size`
- Idea #2: take item with highest price density.

# Example

| item | size (L) | price |
|---|---|---|
| 1 | 50 | $40 |
| 2 | 10 | $25 |
| 3 | 80 | $100 |
| 4 | 5 | $10 |
| 5 | 20 | $20 |
| 6 | 30 | $6 |
| 7 | 8 | $32 |
| 8 | 17 | $34 |

In the bag: ____________

Total value: ____________

Space remaining: ________

# Greedy is Not Optimal

- Claim: the best possible total value is $157.
  - Items 2, 3, and 7.

# Never Looking Back

- Once greedy makes a decision, it never looks back.
- This is why it may be suboptimal.
- **Backtracking**: go back to reconsider every previous decision.

# DSC 190

## DATA STRUCTURES & ALGORITHMS

Lecture 10 | Part 3

## Backtracking

# Backtracking

- Reconsider every decision.
- If we initially tried including x, also try *not* including x.

# Backtracking

```
def knapsack(items, bag_size):
    # choose item arbitrarily from those that fit in bag
    x = items.arbitrary_item(fitting_in=bag_size)

    # if None, it means there was no item that fit
    if x is None:
        return 0

    # assume x should be in bag, see what we get
    best_with = ...

    # backtrack: now assume x should not be in bag, see what we get
    best_without = ...

    return max(best_with, best_without)
```

# Recursive Subproblems

- What is BEST(`items, bag_size`) if we assume that `x` **is** in the bag?
- Imagine choosing `x`.
  - Your current total value is `x.price`.
  - You have `bag_size - x.size` space left.
  - Items left to choose from: `items - x`.
- Clearly, you want the best outcome for *new* situation (subproblem).
- Answer: `x.price` + BEST(`items - x, bag_size - x.size`)

# Recursive Subproblems

- What is BEST(`items, bag_size`) if we assume that `x` **is not** the bag?

- Imagine deciding `x` is not in the bag.
  - Your current total value is `0`.
  - You have `bag_size` space left.
  - Items left to choose from: `items - x`.

- Clearly, you want the best outcome for *new* situation (subproblem).

- Answer: `0` + BEST(`items - x, bag_size`)

# Backtracking

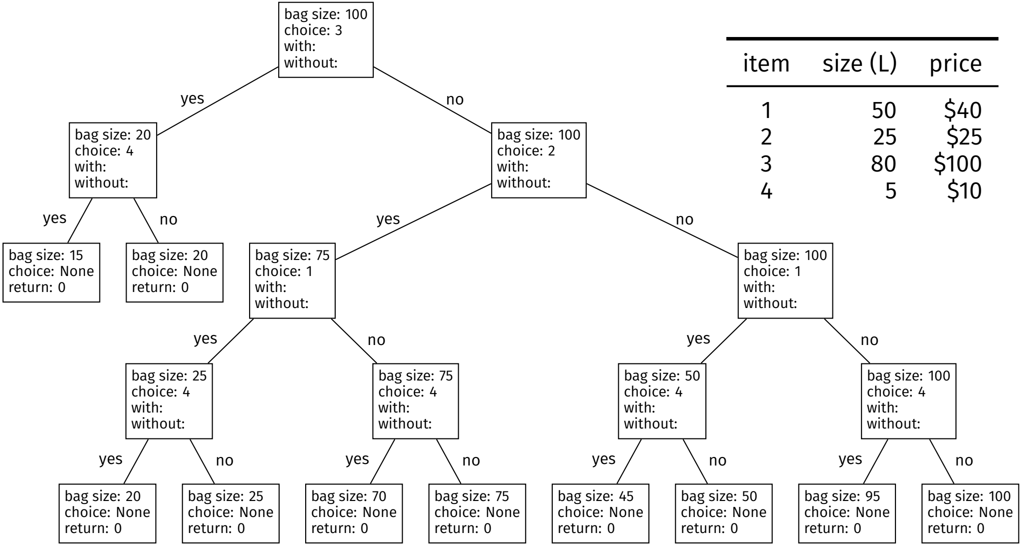
```python
def knapsack(items, bag_size):
    # choose item arbitrarily from those that fit in bag
    x = items.arbitrary_item(fitting_in=bag_size)

    # if None, it means there was no item that fit
    if x is None:
        return 0

    # assume x is in the bag, see what we get
    best_with = # knapsack(items - x, bag_size - x.size)

    # now assume x is not in bag, see what we get
    best_without = # knapsack(items - x, bag_size)

    return max(best_with, best_without)
```

# Backtracking

```python
def knapsack(items, bag_size):
    # choose item arbitrarily from those that fit in bag
    x = items.arbitrary_item(fitting_in=bag_size)

    # if None, it means there was no item that fit
    if x is None:
        return 0

    items.remove(x)
    best_with = knapsack(items, bag_size - x.size)
    best_without = knapsack(items, bag_size)
    items.replace(x)

    return max(best_with, best_without)
```

# Backtracking

- **Backtracking**: go back to reconsider every previous decision.
- Searches the whole tree.
- Can be thought of as a DFS on implicit tree.




| item | size (L) | price |
|---|---|---|
| 1 | 50 | \$40 |
| 2 | 25 | \$25 |
| 3 | 80 | \$100 |
| 4 | 5 | \$10 |

## Exercise

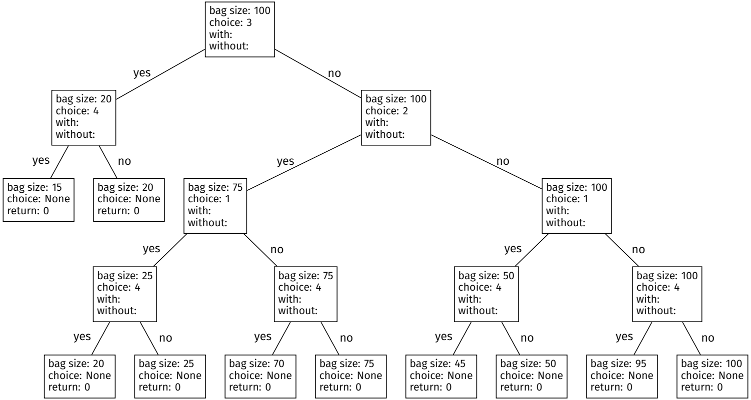
Is the backtracking solution guaranteed to find an optimal solution?

# Yes!

- It tries every **valid** combination and keeps the best.
  - A combination of items is valid if they fit in the bag together.

# Leaf Nodes

- Each leaf node is a different valid combination.



## Exercise

Suppose instead of choosing an arbitrary node we choose most expensive. Does the answer change?

# No!

- The choice of node is arbitrary.
- Call tree will change, but all valid combinations are tried.

## Exercise

How does backtracking relate to the greedy approach? How would you change the code to make it greedy?

# Summary

```python
def knapsack_greedy(items, bag_size):
    # choose greedily
    x = items.most_valuable_item(fitting_in=bag_size)

    # if None, it means there was no item that fit
    if x is None:
        return 0

    # assume x is in the bag, see what we get
    items.remove(x)
    best_with = knapsack(items, bag_size - x.size)

    # in the greedy approach, we don't do this
    # best_without = # knapsack(items - x, bag_size)

    return best_with
```

# DSC 190

## Data Structures & Algorithms

Lecture 10 | Part 4

## Efficiency Analysis

# A Benchmark

- Brute force: try every **possible** combination of items.
  - Even the **invalid** ones whose total size is too big.
  - Why? Hard to know which are invalid without trying them.
- There are $\Theta(2^n)$ possible combinations.
- So brute force takes $\Omega(2^n)$ time. **Exponential :(**

# Time Complexity of Backtracking

```python
def knapsack(items, bag_size):
    # choose item arbitrarily from those that fit in bag
    x = items.arbitrary_item(fitting_in=bag_size)

    # if None, it means there was no item that fit
    if x is None:
        return 0

    items.remove(x)
    best_with = knapsack(items, bag_size - x.size)
    best_without = knapsack(items, bag_size)
    items.replace(x)

    return max(best_with, best_without)
```

$T(n) =$

# Backtracking Takes Exponential Time

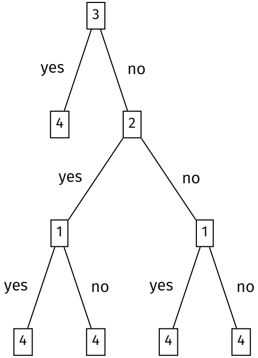
- ...in the worst case.
- This is just as bad as **brute force**.
- So why use it?
- Its worst case isn’t always indicative of its practical performance.

# Intuition

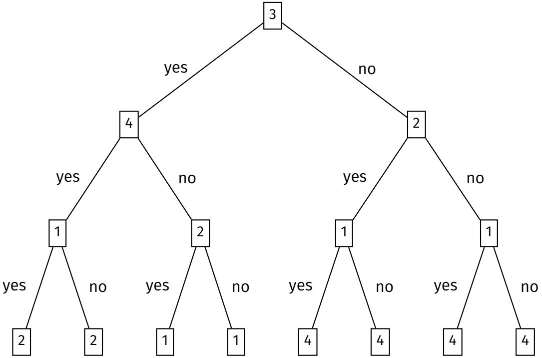
- Brute force tries all **possible** combinations.
- Backtracking tries all **valid** combinations.
- The number of valid combinations can be much less than the number of possible combinations.[2]

[2] Not always true!

# Pruning




backtracking




brute force

# Pruning

- Backtracking **prunes** branches that lead to invalid solutions.

# Example

- 23 items with size/price chosen from Unif([23, ..., 46])
- Bag size is 46
- Brute force: 52 seconds.
- Backtracking: 4 milliseconds.

# Example

- 300 items with size/price chosen from Unif([150, ..., 300])
- Bag size is 600
- Brute force: ? ($\approx 4.6 \times 10^{77}$ years)
- Backtracking: 30 seconds.

# Backtracking Worst Case

- knapsack's **worst case** is when bag size is very large.
- All solutions are valid, aren't pruned.
- But this is actually an easy case!

```python
def knapsack_2(items, bag_size):
    if sum(item.size for item in items) < bag_size:
        return sum(item.price for item in items)

    x = items.arbitrary_item(fitting_in=bag_size)

    if x is None:
        return 0

    items.remove(item)
    best_with = x.price + knapsack_2(items, bag_size - x.size)
    best_without = knapsack_2(items, bag_size)
    items.replace(x)

    return max(best_with, best_without)
```

# Pruning

- This further prunes the tree, resulting in speedup for some inputs.

# DSC 190

## Data Structures & Algorithms

Lecture 10 | Part 5

## Branch and Bound

# Example

- Suppose you have a bag of size 100.
- One of the items is a diamond.
    - Price: \$10,000. Size: 1
- The other 49 items are coal.
    - Price: \$1. Size: 1
- Do you even consider not taking the diamond?

# Idea

- Assume we take the diamond, compute best result.
- Find quick upper bound for not taking diamond.
- If upper bound is less than best for diamond, don’t go down that branch.
- This is **branch and bound**; another way to prune tree.

# Branch and Bound

```
def knapsack_bb(items, bag_size, find_upper_bound):
    # try to make a good first choice
    x = items.item_with_highest_price_density(fitting_in=bag_size)

    if x is None:
        return 0

    items.remove(item)
    best_with = x.price + knapsack_bb(items, bag_size - x.size)

    if find_upper_bound(items, bag_size) < best_with:
        best_without = 0
    else:
        best_without = knapsack_bb(items, bag_size)

    items.replace(x)

    return max(best_with, best_without)
```

# Example

| item | size (L) | price |
|---|---|---|
| 1 | 50 | $40 |
| 2 | 25 | $25 |
| 3 | 95 | $1000 |
| 4 | 5 | $10 |

# Upper Bounds for Knapsack

- How do we get a good upper bound?
- One idea: the solution to the *fractional* knapsack problem upper bounds that for 0/1 knapsack.

# DSC 190

## Data Structures & Algorithms

Lecture 10 | Part 6

## Summary

# Summary

- A backtracking approach is **guaranteed** to find an optimal answer.
- It is typically faster than brute force, but can still take **exponential time**.

# Summary

- We can speed up backtracking by pruning:

- Three ways to prune:
    1. Prune invalid branches (default).

    2. Prune “easy” cases.

    3. Prune by branching and bounding.

# Summary

- Next time: **dynamic programming**.
- We'll see it is just backtracking + memoization.