Greedy Algorithms

Knapsack problem:

Input: values $v_1, \dots, v_n$
weights $w_1, \dots, w_n$
max weight W

Output: set of items with maximum value and total weight $\leq W$

Fractional Knapsack

We can take part of an item.

If we take a fraction f of item i, that has value $f \cdot v_i$ and weight $f \cdot w_i$.

Strategy: Select as much as possible of the item with highest v/w. Repeat.

Theorem: There exists an optimal solution that includes as much as possible of the item X with maximum v/w ratio.

Proof: Consider an optimal solution and suppose that it only includes a of X, when it could include b ($0 \leq a < b \leq 1$). Then we can replace $b-a$ of other items with X. Since X has the maximum v/w ratio, the new solution's ~~seo~~ total value is at least as high as the original solution. □

Example:

v	5	6
w	1	6
v/w	5	1
f	1	5/6

$W=6$

Standard Knapsack:

v	5	6
w	1	6
v/w	5	1

$W=6$

Opt is just item 2, with value 6.

- select item 1
- can't select item 2 ⇒ value 5

This strategy does not always give the optimal solution for regular knapsack.

~~In general~~, Greedy algorithms always make the locally optimal ~~solution~~ decision. (They don't "look ahead".)

For some problems, this gives the globally optimal solution, but usually it does not.

To prove that it does:

1. Consider an optimal solution that does not have greedy structure.
2. Change it to have greedy structure.
3. Show that the value of the solution does not decrease.

Huffman Encoding

Suppose we have a message M that we want to encode as a bitstring.

Example: M = AABACAAABD

Encoding 1:
- A → 00
- B → 01
- C → 10
- D → 11

} 20 bits

fixed-length

Encoding 2:
- 6× A → 0
- 2× B → 10
- 1× C → 110
- 1× D → 111

} 16 bits

variable length

We want to use as few bits as possible, while avoiding ambiguity.

Example: M = ABABC

Encoding 1:
- A → 0
- B → 1
- C → 01

M_1 = 010101

ABABAB?

C C C?

ambiguous

Encoding 2:
- A → 0
- B → 10
- C → 11

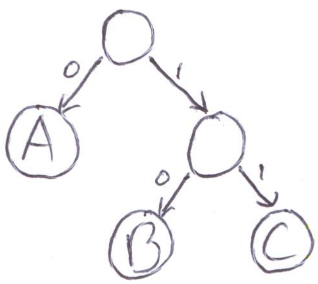
M_2 = 0 10 0 10 11

A B A B C ✓

unambiguous

The key to avoiding ambiguity is to ensure that the encoding is prefix-free: no code is a prefix of another code.

This means that we can represent the encoding as a binary tree:



To decode, we simply follow the appropriate path in the tree.

How do we find the optimal tree?

$$\#\text{bits} = \sum_{s} f(s) \cdot \text{depth}(s),$$ where $f(s)$ is the frequency of symbol s.

Thus, the least frequent symbols should be lowest in the tree.

Strategy: Make the two least frequent $\underbrace{\text{symbols}}_{s_1 \text{ and } s_2}$ leaves of the same internal node. Replace s_1 and s_2 by a merged symbol s_1/s_2 with frequency $f(s_1)+f(s_2)$. Repeat.