1 A Classic Problem and a Greedy Approach

A classic problem for which one might want to apply a greedy algo is *knapsack*.

- Given: a knapsack of capacity $M$, and $n$ items.
- Item $i$ has weight $w_i > 0$, value $v_i > 0$.
- *Problem*: choose contents for the knapsack so that the total weight is at most $M$ and total value is maximized.
- (To make this interesting, we assume that $\sum_i w_i > M$, so we cannot choose everything.)
- Many versions of this problem exist, but let’s look at two.

First variant: *fractional knapsack*

- Can take any real-valued amount up to $w_i$ of item $i$.
- **Examples**: gold dust, gasoline, cocaine . . .
- Could also model return on activities, e.g. time spent coding vs time spent writing grants vs time spent reading papers.
- Suggestions? *(Wait)*
- **Intuition**: to maximize value, we want to take items with greatest “value density.”
- Define $d_i = \frac{v_i}{w_i}$.
- Density measures “bang for buck” from taking a fixed amount of a given item.

OK, so let’s design a (greedy) algorithm.

- Sort items in decreasing order of value density.
- Initial weight of knapsack is 0 (empty).
- For each item $i$ in this order, add item to knapsack until it is used up or until total weight reaches $M$.
- Cost is trivially $O(n \log n)$. Is it correct?
- Let’s formulate and prove our key properties!
2 Proof that Fractional Knapsack is Optimal

- **Greedy Choice**: Consider a knapsack instance \( P \), and let item 1 be item of highest value density. Then there exists an optimal solution to \( P \) that uses as much of item 1 as possible (that is, \( \min(w_1, M) \)).

- **Pf**: suppose we have a solution \( \Pi \) that uses weight \( w < \min(w_1, M) \) of item 1. Let \( w' = \min(w_1, M) - w \).

- \( \Pi \) must contain at least weight \( w' \) of some other item(s), since it never pays to leave the knapsack partly empty.

- Construct \( \Pi^* \) from \( \Pi \) by removing \( w' \) worth of other items and replacing with \( w' \) worth of item 1.

- Because item 1 has max value density, \( \Pi^* \) has total value at least as big as \( \Pi \). QED

One down, two to go.

- **Inductive Structure**: after making greedy first choice for \( P \), we are left with a smaller problem instance \( P' \) with no external constraints.

- **Pf**: We are left with a knapsack of capacity \( M' < M \) (possibly 0) and a collection of remaining items to fill it. Any feasible knapsack for \( P' \) may be combined with the remaining weight \( M - M' \) of item 1 to form a feasible knapsack for \( P \).

Two down, one to go.

- **Optimal Substructure**: optimal solution to \( P' \) combined with greedy choice yields optimal solution to \( P \).

- **Pf**: suppose we find optimal solution \( \Pi' \) for \( P' \) and combine with greedy choice to get solution \( \Pi \) to \( P \). Let \( \hat{v} \leq v_1 \) be value associated with initial greedy choice.

- Then \( \text{value}(\Pi) = \text{value}(\Pi') + \hat{v} \).

- If \( \Pi \) is not optimal, let \( \Pi^* \) be an optimal solution that makes greedy choice, i.e. uses as much of item 1 as possible. Remainder of knapsack after this item is removed has value greater than \( \text{value}(\Pi') \), which is impossible. QED

- **Note**: that last bit of argument gets repetitive to write. It is enough for your homework to recognize that

\[
\text{value}(\Pi) = \text{value}(\Pi') + \hat{v},
\]

that is, that the value of the full solution is the value of the subproblem’s solution plus that of the greedy choice.
3 And Now, a Classic Failure

Lest you get all excited and think that greed always works...

- 0-1 knapsack problem does not permit items to be subdivided.
- **Example**: gold bar, painting, SD card full of purchased songs
- Each item still has weight $w_i$ and value $v_i$.
- Goal is to maximize value of knapsack without going over weight $M$.

Fractional algo makes no sense in this context. What to do?

- Well, we can still assign value density $d_i$ to each item.
- Intuitively, items of high value density are more attractive (diamond vs an equal-sized chunk of coal).
- **Suggestion**: sort items by decreasing value density as before, then choose items of highest density until next item would exceed total weight of $M$.
- Does this work? *(wait)*
- Counterexample with 3 items and $M = 5$:

  - values = 20, 30, 40; weights = 2, 3, 3.5
  - densities are 10, 10, 11.4
  - Greedy algo picks item of weight 3.5 first, then stops with value 40.
  - Optimal solution would take other two items for total value 50.

What broke?

- There is *no* optimal solution that contains the greedy choice!
- Hence, greedy choice property fails for this problem.
- (In fact, it is NP-hard, but we don’t know that yet!)
4 Set up for Huffman Coding

We’re now going to look at a very important application of greedy algorithm design. In fact, you probably use it every day at least a few times without knowing it.

- **Setting**: data compression (ZIP, gzip, etc)
- Suppose we have a text consisting of a collection of letters.
- We want to encode this text efficiently on a computer.
- If there are at most $2^k$ distinct letters, each one can be represented by a $k$-bit code. For $n$-letter text, total size in bits is $kn$.
- However, what if letter frequencies are very unequal?
- **Example**: in *Moby Dick*, there are 117194 occurrences of ‘e’, but only 640 occurrences of ‘z’. Other letters are in-between.
- **Idea**: encode common letters with fewer bits!
- **Example** on text of 100,000 letters from 6-letter alphabet:
  - Smallest fixed-length encoding uses 3 bits per letter, hence 300,000 total bits.
  - Specified encoding uses only 224,000 bits.

Wait, are variable-length encodings possible?

- **Stupid example**: consider the following encoding of a 3-letter alphabet:
  - $a \to 0$
  - $b \to 1$
  - $c \to 01$
  - Hence, the message “ababc” would be encoded as “010101”
  - Anyone see a problem here?
  - “ccc” is *also* encoded as “010101.” If you receive these bits, which message was sent?
  - This is the *ambiguity problem*.
• In contrast, first code is *unambiguous* – no encoded message can be decoded as two different strings.

• *Sufficient condition*: no code word is a prefix of any other code word. (Pf left as exercise)

• Such codes are called *prefix-free*, or simply *prefix codes*.

We will look at *Huffman coding*, a technique for optimal prefix code generation.

• **Problem**: given a sequence $T$ built from letters $X = \{x_1 \ldots x_n\}$, such that $x_i$ occurs $f_i$ times in $T$.

• Produce an encoding function $C(x)$ that maps characters to bit strings, s.t.
  1. $C(X)$ is a prefix code.
  2. Total number of bits used to represent $T$ is minimized. That is, minimize $B(T) = \sum_i f_i \cdot |C(x_i)|$. 