Design and Analysis of Algorithms

Instructor: Sharma Thankachan

Lecture 10: Greedy Algorithm

Slides modified from Dr. Hon, with permission
About this lecture

• Introduce *Greedy Algorithm*

• Look at some problems solvable by *Greedy Algorithm*
Coin Changing

• Suppose that in a certain country, the coin denominations consist of:
  $1, $2, $5, $10

• You want to design an algorithm such that you can make change of any $x$ dollars using the fewest number of coins
Coin Changing

• An idea is as follows:
  1. Create an empty bag
  2. while \((x > 0)\) {
    Find the largest coin \(c\) at most \(x\);
    Put \(c\) in the bag;
    Set \(x = x - c\);
  }
  3. Return coins in the bag
Coin Changing

• It is easy to check that the algorithm always return coins whose sum is $x$.

• At each step, the algorithm makes a greedy choice (by including the largest coin) which looks best to come up with an optimal solution (a change with fewest #coins).

• This is an example of Greedy Algorithm.
Coin Changing

- Is Greedy Algorithm always working?
- No!
- Consider a new set of coin denominations: $1, $4, $5, $10

- Suppose we want a change of $8
- Greedy algorithm: 4 coins (5,1,1,1)
- Optimal solution: 2 coins (4,4)
Greedy Algorithm

• We will look at some non-trivial examples where greedy algorithm works correctly

• Usually, to show a greedy algorithm works:
  • We show that some optimal solution includes the greedy choice
    ➔ selecting greedy choice is correct
  • We show optimal substructure property
    ➔ solve the subproblem recursively
Activity Selection

• Suppose you are a freshman in a school, and there are many welcoming activities.

• There are $n$ activities $A_1, A_2, \ldots, A_n$.

• For each activity $A_k$, it has
  • a start time $s_k$, and
  • a finish time $f_k$.

Target: Join as many as possible!
Activity Selection

- To join the activity $A_k$,
  - you must join at $s_k$;
  - you must also stay until $f_k$

- Since we want as many activities as possible, should we choose the one with
  (1) Shortest duration time?
  (2) Earliest start time?
  (3) Earliest finish time?
Activity Selection

• Shortest duration time may not be good:
  \[ A_1 : [4:50, 5:10), \]
  \[ A_2 : [3:00, 5:00), \quad A_3 : [5:05, 7:00), \]

• Though not optimal, \#activities in this solution \( R \) (shortest duration first) is at least half \#activities in an optimal solution \( O \):
  • One activity in \( R \) clashes with at most 2 in \( O \)
  • If \( |O| > 2|R| \), \( R \) should have one more activity
Activity Selection

• Earliest start time may even be worse:
  \[A_1 : [3:00, 10:00),\]
  \[A_2 : [3:10, 3:20), \quad A_3 : [3:20, 3:30),\]
  \[A_4 : [3:30, 3:40), \quad A_5 : [3:40, 3:50) \ldots\]

• In the worst-case, the solution contains 1 activity, while optimal has \(n-1\) activities
Greedy Choice Property

To our surprise, earliest finish time works!

We actually have the following lemma:

Lemma: For the activity selection problem, some optimal solution includes an activity with earliest finish time

How to prove?
Proof: (By “Cut-and-Paste” argument)

- Let $OPT$ = an optimal solution
- Let $A_j$ = activity with earliest finish time
- If $OPT$ contains $A_j$, done!
- Else, let $A'$ = earliest activity in $OPT$
  - Since $A_j$ finishes no later than $A'$, we can replace $A'$ by $A_j$ in $OPT$ without conflicting other activities in $OPT$
  - an optimal solution containing $A_j$
    (since it has same #activities as $OPT$)
Optimal Substructure

Let $A_j$ = activity with earliest finish time
Let $S$ = the subset of original activities that do not conflict with $A_j$
Let $OPT$ = optimal solution containing $A_j$

Lemma:

$OPT - \{ A_j \}$ must be an optimal solution for the subproblem with input activities $S$
Proof: (By contradiction)
• First, \( \text{OPT} - \{ A_j \} \) can contain only activities in \( S \)
• If it is not an optimal solution for input activities in \( S \), let \( C \) be some optimal solution for input \( S \)
  \( \implies C \) has more activities than \( \text{OPT} - \{ A_j \} \)
  \( \implies C \left[ \{ A_j \} \right] \) has more activities than \( \text{OPT} \)
  \( \implies \text{Contradiction occurs} \)
Greedy Algorithm

The previous two lemmas implies the following **correct** greedy algorithm:

\[ S = \text{input set of activities}; \]

\[ \text{while (S is not empty)} \ { } \{ \]

\[ A = \text{activity in } S \text{ with earliest finish time}; \]

Select \( A \) and update \( S \) by removing activities having conflicts with \( A \);

\} 

If finish times are sorted in input, running time = \( O(n) \)
0-1 Knapsack Problem

- Suppose you are a thief, and you are now in a jewelry shop (nobody is around!)
- You have a big knapsack that you have “borrowed” from some shop before
  - Weight limit of knapsack: $W$
- There are $n$ items, $I_1, I_2, \ldots, I_n$
  - $I_k$ has value $v_k$, weight $w_k$

Target: Get items with total value as large as possible without exceeding weight limit
0-1 Knapsack Problem

• We may think of some strategies like:
  (1) Take the most valuable item first
  (2) Take the densest item (with $v_k/w_k$ is maximized) first

• Unfortunately, someone shows that this problem is very hard (NP-complete), so that it is unlikely to have a good strategy

• Let’s change the problem a bit...
Fractional Knapsack Problem

• In the previous problem, for each item, we either take it all, or leave it there
  • Cannot take a fraction of an item

• Suppose we can allow taking fractions of the items; precisely, for a fraction $c$
  • $c$ part of $I_k$ has value $cv_k$, weight $cw_k$

Target: Get as valuable a load as possible, without exceeding weight limit
Fractional Knapsack Problem

• Suddenly, the following strategy works:
  Take as much of the densest item (with $v_k/w_k$ is maximized) as possible

• The correctness of the above greedy-choice property can be shown by cut-and-paste argument

• Also, it is easy to see that this problem has optimal substructure property
  \[ \implies \text{implies a correct greedy algorithm} \]
Fractional Knapsack Problem

• However, the previous greedy algorithm (pick densest) **does not work** for 0-1 knapsack
• To see why, consider $W = 50$ and:
  
  $I_1 : v_1 = 60, \ w_1 = 10 \ (\text{density: 6})$
  
  $I_2 : v_2 = 100, \ w_2 = 20 \ (\text{density: 5})$
  
  $I_3 : v_3 = 120, \ w_3 = 30 \ (\text{density: 4})$

• Greedy algorithm: $160 \ (I_1, I_2)$
• Optimal solution: $220 \ (I_2, I_3)$
Encoding Characters

- In ASCII, each character is encoded using the same number of bits (8 bits)
  - called *fixed-length* encoding
- However, in real-life English texts, not every character has the same frequency
- One way to encode the texts is:
  - Encode frequent chars with few bits
  - Encode infrequent chars with more bits
  => called *variable-length* encoding
Encoding Characters

- **Variable-length** encoding may gain a lot in storage requirement

**Example:**
- Suppose we have a 100K-char file consisted of only chars a, b, c, d, e, f
- Suppose we know a occurs 45K times, and other chars each 11K times

⇒ Fixed-length encoding: 300K bits
Encoding Characters

Example (cont):

Suppose we encode the chars as follows:

\[ a \rightarrow 0, \quad b \rightarrow 100, \quad c \rightarrow 101, \]
\[ d \rightarrow 110, \quad e \rightarrow 1110, \quad f \rightarrow 1111 \]

- Storage with the above encoding:

\[ (45 \times 1 + 33 \times 3 + 22 \times 4) \times 1K \]
\[ = 232K \text{ bits (reduced by 25\% !!)} \]
Encoding Characters

Thinking a step ahead, you may consider an even "better" encoding scheme:

\[ a \rightarrow 0, \quad b \rightarrow 1, \quad c \rightarrow 00, \]
\[ d \rightarrow 01, \quad e \rightarrow 10, \quad f \rightarrow 11 \]

- This encoding requires less storage since each char is encoded in fewer bits …

- What’s wrong with this encoding?
Prefix Code

Suppose the encoded texts is: 0101
We cannot tell if the original text is
abab, dd, abd, aeb, or ...

• The problem comes from:
  one codeword is a prefix of another one
Prefix Code

- To avoid the problem, we generally want each codeword not a prefix of another
- called prefix code, or prefix-free code
- Let $T$ = text encoded by prefix code
- We can easily decode $T$ back to original:
  - Scan $T$ from the beginning
  - Once we see a codeword, output the corresponding char
  - Then, recursively decode remaining
Prefix Code Tree

- Naturally, a prefix code scheme corresponds to a prefix code tree
  - Each char $\rightarrow$ a leaf
  - Root-to-leaf path $\rightarrow$ codeword
- E.g., $a \rightarrow 0$, $b \rightarrow 100$, $c \rightarrow 101$, $d \rightarrow 110$, $e \rightarrow 1110$, $f \rightarrow 1111$
Optimal Prefix Code

Question: Given frequencies of each char, how to find the optimal prefix code scheme (or optimal prefix code tree)?

Precisely:

Input: \( S = \) a set \( n \) chars, \( c_1, c_2, \ldots, c_n \) with \( c_k \) occurs \( f_{c_k} \) times

Target: Find codeword \( w_k \) for each \( c_k \) such that \( \sum_k |w_k| f_{c_k} \) is minimized
Huffman Code

In 1952, David Huffman (then an MIT PhD student) thinks of a greedy algorithm to obtain the optimal prefix code tree.

Let \( c \) and \( c' \) be chars with least frequencies. He observed that:

**Lemma:** There is some optimal prefix code tree with \( c \) and \( c' \) sharing the same parent, and the two leaves are farthest from root.
Proof: (By “Cut-and-Paste” argument)

• Let $\text{OPT} =$ some optimal solution
• If $c$ and $c'$ as required, done!
• Else, let $a$ and $b$ be two bottom-most leaves sharing same parent (such leaves must exist... why??)
  • swap $a$ with $c$, swap $b$ with $c'$
  • an optimal solution as required
    (since it at most the same $\sum_k |w_k| f_k$ as OPT ... why??)
Graphically:

If this is optimal

then this is optimal

Bottom-most leaves
Optimal Substructure

Let $OPT$ be an optimal prefix code tree with $c$ and $c'$ as required.

Let $T$ be a tree formed by merging $c$, $c'$, and their parent into one node.

Consider $S' = \text{set formed by removing } c \text{ and } c' \text{ from } S$, but adding $X$ with $f_X = f_c + f_{c'}$.

Lemma:

$T$ is an optimal prefix code tree for $S'$. 
Graphically, the lemma says:

If this is optimal for $S$ and the parent node then this is optimal for $S'$.

Merging $c$, $c'$ and the parent node.

Here, $f_X = f_c + f_{c'}$.

Merged node.
Huffman Code

Questions:

Based on the previous lemmas, can you obtain Huffman’s coding scheme?
(Try to think about yourself before looking at next page...)

What is the running time?

$O(n \log n)$ time, using heap (how??)
Huffman($S$) // build Huffman code tree

1. Find least frequent chars $c$ and $c'$
2. $S' = \text{remove } c \text{ and } c' \text{ from } S,$
   but add char $X$ with $f_X = f_c + f_{c'}$
3. $T' = \text{Huffman}(S')$
4. Make leaf $X$ of $T'$ an internal node by connecting two leaves $c$ and $c'$ to it
5. Return resulting tree

}