Greedy Algorithms

Optimization problem: find the best way to do something.
- E.g. match up two strings (LCS problem).

Search techniques look at many possible solutions.
- E.g. dynamic programming or backtrack search.

A greedy algorithm
- Makes choices along the way that seem the best.
- Sticks with those choices.

For some problems, greedy approach always gets optimum.
For others, greedy finds good, but not always best.
- If so, it’s called a greedy heuristic or approximation.

For still others, greedy approach can do very poorly.
The problem of giving change

Vending machine has huge supply of quarters, dimes and nickels.

Customer needs N cents change (N is multiple of 5).

Machine wants to give out few coins as possible.

Greedy approach:

while (N > 0) {
    give largest denomination coin \leq N;
    reduce N by value of that coin;
}

Does this return the fewest number of coins?

Aside: Using division, it could make decisions faster.

More on giving change

Thm: Greedy algorithm always gives minimal # of coins.

Proof:
- Optimum has \leq 2 dimes.
  - Quarter and nickel better than 3 dimes.
- Optimum has \leq 1 nickel
  - Dime better than 2 nickels.
- Optimum doesn't have 2 dimes + 1 nickel
  - It would use quarter instead.
- So optimum & greedy have at most $0.20 in non-quarters.
  - That is, they give the same number of quarters.
- Optimum & greedy give same on remaining \leq$0.20 too.
  - Obviously.
More on giving change

Suppose we run out of nickels, put pennies in instead.
- Does greedy approach still give minimum number of coins?

Formally, the Coin Change problem is:

Given \( k \) denominations \( d_1, d_2, \ldots, d_k \) and given \( N \),
find a way of writing \( N = i_1 d_1 + i_2 d_2 + \ldots + i_k d_k \) such
that \( i_1 + i_2 + \ldots + i_k \) is minimized.
“Size” of problem is \( k \).

Is the greedy algorithm always a good heuristic?
That is, is there exists a constant \( c \) s.t. for all instances of
Coin Change, the greedy algorithm gives at most \( c \) times
the optimum number of coins?

How do we solve Coin Change exactly?

Coin Change by Dynamic Programming

Let \( C(N) = \min \# \) of coins needed to give \( N \) cents.

Detail: If \( N < 0 \), define \( C(N) = \infty \)

Optimal substructure: If you remove 1 coin, you must
have minimum solution to smaller problem.

So \( C(N) = 1 + \min \{ C(N-5), C(N-10), C(N-25) \} \)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
<th>35</th>
<th>40</th>
<th>45</th>
<th>50</th>
<th>55</th>
<th>60</th>
<th>65</th>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Complexity of Coin Change

Greedy algorithm (non-optimal) takes $O(k)$ time.

Dynamic Programming takes $O(kN)$ time.

- This is NOT necessarily polynomial in $k$!
  - Better way to define “size” is the number of bits needed to specify an instance.
  - With this definition, $N$ can be almost $2^{\text{size}}$.
  - So Dynamic Programming is exponential in size.

- In fact, Coin Change problem is NP-hard.
  - So no one knows a polynomial-time algorithm for it.
Linear Partition Problem

Given a list of positive integers, s₁, s₂, ..., sₙ, and a bound B, find smallest number of contiguous sublists s.t. each sum of each sublist ≤ B.

I.e.: find partition points 0 = p₀, p₁, p₂, ..., pₖ = N such that for j = 0, 1, ..., k-1,

\[ \sum_{i=p_j+1}^{p_{j+1}} s_i \leq B \]

**Greedy algorithm:**

Choose \( p_1 \) as large as possible.

Then choose \( p_2 \) as large as possible. Etc.

---

**Greedy is optimal for linear partition**

**Thm:** Given any valid partition \( 0 = q₀, q₁, ..., qₖ = N \), then for all j, \( q_j \leq p_j \). (The \( p_j \)'s are greedy solution.)

**Proof:** (by induction on \( k \)).

**Base Case:** \( p₀ = q₀ = 0 \) (by definition).

**Inductive Step:** Assume \( q_j \leq p_j \).

We know \( \sum_{i=q_j+1}^{q_{j+1}} s_i \leq B \) (since \( q \)'s are valid).

So \( \sum_{i=p_j+1}^{p_{j+1}} s_i \leq B \) (since \( q_j \leq p_j \)).

So \( q_{j+1} \leq p_{j+1} \) (since Greedy chooses \( p_{j+1} \) to be as large as possible subject to constraint on sum).
Variant on Linear Partitioning

New goal: partition list of N integers into exactly k contiguous sublists to so that the maximum sum of a sublist is as small as possible.

Example: Partition <16, 7, 19, 3, 4, 11, 6> into 4 sublists.
- We might try 16+7, 19, 3+4, 11+6. Max sum is 16+7=23.

Try out (at board):
- Greedy algorithm: add elements until you exceed average.
- Divide-and-conquer: break into two nearly equal sublists.
- Reduce to previous problem: binary search on B.
- Dynamic programming.

Scheduling Unit time Tasks

Given N tasks (N is problem size):
- Task i must be done by time d_i.
- Task i is worth w_i.

You can perform one task per unit time. If you do it before its deadline d_i, you get paid w_i.

Problem: Decide what to do at each unit of time.
- Aside: This is an off-line scheduling problem: You know entire problem before making any decisions.
- In an on-line problem, you get tasks one-at-a-time, and must decide when to schedule it before seeing next task.
- Typically, it’s impossible to solve an on-line problem optimally, and the goal is to achieve at least a certain % of optimal.