CSE 202 Homework 2: Solutions
Combinatorial Search and Optimization:
Dynammic Programming and Greedy Algorithms

1. **Bitonic euclidean traveling-salesperson problem: CLR 16-1**

   Solution(s)

   We will consider two approaches to the problem that result in $O(n^2)$ algorithms. The first consists of $O(n^2)$ subproblems that are generated in constant time each, the second has $O(n)$ subproblems that require $O(n)$ time to compute.

   The first solution maintains subproblems, D, where $D(i,j), j < i$, is the length of an optimal bitonic path with endpoints i and j visiting all points $k < i$. Note that this path is unidirectional on points $k > j$ since j is an endpoint. The base case $D(1,2)$ is just the distance $|p_1 - p_2|$. Recursively,

   $D(i,j) = \min\left(D(i,j-1) + |p_j - p_{j-1}|, D(j,j-1) + |p_{j-1} - p_{j+1}| + \sum_{j+1 \le k < i} |p_k - p_{k+1}|\right)$

   Intuitively this compares the case where point $j-1$ lies on the path leading to endpoint j to that where $j-1$ leads to i. Then, the final tour is the minimum over j of $D(n,j)$ with the closing edge added: $T = \min_{j<n} D(n,j) + |p_n - p_j|$.

   To prove this is the optimal tour length we must show that the base case is optimal, the recursive formulation maintains optimality, and the final solution can be obtained from the subproblems. By our definition of the subproblem, $D(1,2) = |p_1 - p_2|$ is clearly optimal since the direct line is the only path to consider. Now, when calculating $D(i,j)$, node k ($j < k < i$) is necessarily connected to $k+1$ because the path is bitonic. The first node we have to make a decision about is $j-1$ which has two choices. Consider the optimal bitonic path with endpoints i and j; point $j-1$ must come between 1 and j or between 1 and i, not both since j and i are endpoints. In the first case, $j-1$ is necessarily directly connected to j and the optimal path $D(i,j-1)$ finishes the path $D(i,j)$. In the second, $j-1$ is connected to $j+1$ and the path is $D(j,j-1)$ with the monotonic path from $j-1$ to n (skipping only j) added. Optimality is inductively maintained. Finally, the optimal tour must necessarily connect $(n, n-1)$ in one direction and some (n,j) in the other and thus follow an optimal bitonic path with endpoints n and j. Minimizing over these path lengths plus the connecting edge (n,j) will give an optimal bitonic tour.

   There are $O(n^2)$ subproblems. Each is evaluated by a constant number of references to other subproblems, a reference to an edge distance, and a monotonic path length (which can be calculated in $O(n^2)$ time beforehand and written down). The evaluation ordering on subproblems (for i = 1 to n (for j = 1 to i) ...) ensures referenced subproblems have already been calculated. So, the total time is $O(n^2)$.

   Another approach (thanks to Nathan Segerlind and crew, and just summarized here) is to consider bitonic paths with endpoints i and $i+1$, containing all points $j < i$. The base case $D(1,2)$ is again just the edge (1,2). Recursively, $D(i,i+1) = \min_j(D(j,j+1) + |p_j - p_i| + |p_{i+1} - p_{i-1}| + \sum_{j+1<k<i} |p_{k-1} - p_k|)$. Intuitively this recursion considers all possible immediate predecessors to i, and completes the bitonic path with the optimal subproblem already known plus the implicit monotonic segments. To find the final solution, the final closing edge must also be considered. The optimality is inductively guaranteed because all possible bitonic tours with endpoints $i, i+1$ have some node $j < i1$ directly connected to i combined with a bitonic tour with endpoints $j, j+1$ and a monotonic path from $j+1$ to $i+1$. Further, all possible bitonic *tours*have some immediate predecessor, j, on the side not through $n-1$ and therefore contain a bitonic *path*with endpoints j and $j+1$. There are $O(n)$ subproblems and each is calculated as the minimum over $O(n)$ previously calculated subproblems. This algorithm is therefore $O(n^2)$ time.

2. Problem 3-6 in Skiena. In the United States, coins are minted with denominations of 1,5,10,25, and 50 cents. Now consider a country whose coins are minted with denominations of $\{d_1, ..., d_k\}$ units. They seek an algorithm that will enable them to make change of n units using the minimum number of coins.

   a) The greedy algorithm for making change repeatedly uses the biggest coin smaller than the amount to be changed until it is zero. Provide a greedy algorithm for making change of n units using US denominations. Prove its correctness and analyze its time complexity.

b) Show that the greedy algorithm does not always give the minimum number of coins in a country whose denominations are $\{1, 6, 10\}$.

c) Give an efficient algorithm that correctly determines the minimum number of coins needed to make change of n units using denominations $\{d_1, ..., d_k\}$. Analyze its running time.

Solution:

a)

Inputs: number of units to make change for n

Outputs: number of half dollars, quarter, dimes, nickels, and pennies to use $(c_{50}, c_{25}, c_{10}, c_5, c_1)$.

MAKECHANGE(n)

(a) $c_{50} = n$ div 50
(b) $leftover = n \bmod 50$
(c) $c_{25} = n$ div 25
(d) $leftover = n \bmod 25$
(e) $c_{10} = n$ div 10
(f) $leftover = n \bmod 10$
(g) $c_5 = n$ div 5
(h) $leftover = n \bmod 5$
(i) $c_1 = n$ div 1
(j) $leftover = n \bmod 1$
(k) return $(c_{50}, c_{25}, c_{10}, c_5, c_1)$

Because the algorithm always performs 10 calculations, it's worst-case running time is $\Theta(1)$.

Proof of Optimality

Assume that the best non-greedy solution for a given instance of the problem is $(b_{50}, b_{25}, b_{10}, b_5, b_1)$, where $n = 50b_{50} + 25b_{25} + 10b_{10} + 5b_5 + b_1$. We show that the greedy solution is as good as or better than the best solution. The greedy solution is $(c_{50}, c_{25}, c_{10}, c_5, c_1)$. We want to show that $c_{50} + c_{25} + c_{10} + c_5 + c_1 \leq b_{50} + b_{25} + b_{10} + b_5 + b_1$.

Since the best solution is not greedy at some point there will be fewer coins of some denomination in the best solution vs. the greedy solution. We will show that any combination of coins with lower denominations which make up for the difference could be replaced with fewer coins. Therefore, the best solution must be equivalent to the greedy solution.

If $b_50 < c_{50}$ then $25b_{25} + 10b_{10} + 5b_5 + b_1 \geq 50$. To satisfy the given inequality b_25 could be ≥ 2. If that is the case then each 2 quarters could be replaced by one half-dollar thus using fewer coins. For this inequality all the cases are listed below.

(a) if $b_{25} \geq 2$, replace with 1 half-dollar
(b) if $b_{25} = 1$ we must also have either 2 dimes and 1 nickel, 1 dime and 3 nickels, etc., any of these combinations can be replaced with 1 half-dollar therefore using fewer coins
(c) if $b_{25} = 0$ we must also have either 5 dimes, 4 dimes and 2 nickels, etc., any of these combinations can be replaced with 1 half-dollar

If $b_{50} = c_{50}$ and $b_{25} < c_{25}$ then $10b_{10} + 5b_5 + b_1 \geq 25$.

(a) if $b_{10} \geq 3$, replace with 1 quarter and 1 nickel
(b) if $b_{10} = 2$ we must also have either 1 nickels or 5 pennies, all of which can be replaced with 1 quarter

(c) if $b_{10} = 1$ we must also have either 3 nickels, 2 nickels and 5 pennies, etc., any of these combinations can be replaced with 1 quarter

(d) if $b_{10} = 0$ we must also have either 5 nickels, 5 nickels and 5 pennies, etc., any of these combinations can be replaced with 1 quarter

The entire proof would continue through the case if $b_{50} = c_{50}$, $b_{25} = c_{25}$, $b_{10} = c_{10}$, and $b_5 < c_5$.

b)

We can show that the greedy algorithm doesn't work for all possible denominations by giving a counter-example. If $n = 12$ and $(d_1, d_2, d_3) = (1, 6, 10)$, then the greedy algorithm would return $(c_{10}, c_6, c_1) = (1, 0, 2)$. However, the optimal solution is $(c_{10}, c_6, c_1) = (0, 2, 0)$.

c)

Given a list of k coin values, $(d_1, d_2, ..., d_k)$, and a number n, we want to find the integers $(c_{d_1}, c_{d_2}, ..., c_{d_k})$ such that $n = \sum_{i=1}^{k} d_i c_{d_i}$ and that $\sum_{i=1}^{k} c_{d_i}$ is minimal.

Our subproblems consist of the optimal change set for 1 through n. To keep track of the optimal solution for each subproblem we will use an array called $sumc$ which is indexed by subproblem. (i.e. $sumc[i]$ contains the least number of coins needed to make change for i). $coin[i]$ designates which coin denomination was last used when making change for i units.

$$sumc[d_1] = 1, sumc[d_2] = 1, ..., sumc[d_k] = 1$$

$$sumc[i] = \min_{1 \le j \le k} sumc[i - d_j] + 1$$

Inputs: denominations $(d_1, d_2, ..., d_k)$, units n

Outputs: the count of each denomination $(c_{d_1}, c_{d_2}, ..., c_{d_k})$.

MAKECHANGE(n,$(d_1, d_2, ..., d_k)$)

(a) // initialization
(b) for $i = 1$ to n
(c) $\quad sumc[i] = \infty$
(d) for $j = 1$ to k
(e) $\quad sumc[d_j] = 1; coin[d_j] = j$
(f) // calculate $sumw[i]$ for $1 \le i \le n$
(g) for $i = 1$ to n
(h) $\quad$ for $j = 1$ to k
(i) $\quad\quad temp = sumc[i - d_j] + 1$
(j) $\quad\quad$ if $temp < sumc[i]$
(k) $\quad\quad\quad sumc[i] = temp; coin[i] = j$
(l) // determine if it is possible to make change
(m) if $sumc[n] = \infty$ return "impossible"
(n) else // generate answer
(o) $\quad$ for $j = 1$ to $sumc[n]$
(p) $\quad\quad c_{d_j} = 0$ // initialization
(q) $\quad$ // traverse through coins used to make best change
(r) $\quad total = n$

(s) while $total > 0$

(t) $\quad c_{d_{coin[total]}} = c_{d_{coin[total]}} + 1$

(u) $\quad total = total - d_{coin[total]}$

(v) return $(c_{d_1}, c_{d_2}, ..., c_{d_k})$

The running time of the above algorithm is $O(nk)$. Notice that n doesn't properly describe our input size. The input parameter n can be represented with $\lg n$ bits. Assume that all d_i denominations are represented with a constant number of bits. The true input size is $O(p + k)$, where $p = \lg n$. In terms of the true input size the running time of the algorithm is $O(2^p k)$.