CMPS 6610 Problem Set 04

Answers

Name: Areen Khalaila

Place all written answers from `problemset-04.md` here for easier grading.

- **1d.**

```
File         | Fixed-Length Coding|Huffman Coding | Huffman vs. Fixed-Length
----------------------------------------------------------------------
f1.txt       |       1340         |     826       |    0.6164179104477612
alice29.txt  |       1039367      |    676374     |    0.6507557003445367
asyoulik.txt|       876253       |    606448     |    0.6920923523229022
grammar.lsp  |       26047        |    17356      |    0.6663339348101509
fields.c     |       78050        |    56206      |    0.7201281229980782
```

Across all the 5 files, Huffman coding consistently required fewer bits than fixed-length coding. The ratio Huffman/Fixed ranged from about 0.62 to 0.72 (~28–38% savings).

- **1e.**

If every character has the same frequency, Huffman coding assigns nearly equal code lengths to all symbols.
Let $m = |\Sigma|$. When frequencies are uniform, the average bits per character is about $\log_2 m$,
equal to the fixed-length cost if m is a power of two.
If not, a few symbols get codes one bit longer, so the cost is slightly less than the fixed-length cost.
The result is consistent across documents with the same alphabet size.

- **2a.** All elements after index n/2 - 1 are leaves, and leaves already satisfy the heap property (since they have no children). So, we start from the last internal node (index n/2 - 1) and move backward to index 0. We sift down each node. So if the parent is larger than either child, we swap it with the smaller child. We keep moving it down until it's smaller than both children or becomes a leaf.

```
siftDown(A, i, n):
    while True:
        l = 2*i+ 1
        r = 2*i+ 2
        smallest = i
        if l < n and A[l] < A[smallest]:  smallest = l
        if r < n and A[r] < A[smallest]:   smallest = r
        if smallest == i:
            break
        swap(A[i],A[smallest])
        i = smallest
```

- **2b.**

Span is the height of tree so O(log n)

- **3a.** Repeat: take the largest coin whose value is <= the remaining amount.

Steps: 1. Let N be the remaining amount. 2. While N > 0: - Let c = $2^{\log_2 N}$ (largest power of two <= N) - Take t = N/c coins of value c - Set N = N % c 3. Output all chosen coins (or just the count)

- **3b.**

Claim (Greedy Choice):
Let the denominations be powers of two: $1, 2, 4, ..., 2^k$. For any amount N>0, an optimal solution must use the largest coin $c = 2^{log_2 N}$ with c <= N.

Proof:
All available coins are <= c. In particular, every coin is <= c/2.
Any set of coins that reaches total value at least c without using c must contain at least two coins (since each is <= c/2). Replacing those =>2 coins by the single coin c yields the same or fewer coins. Therefore every optimal solution uses coin c

Claim (Optimal Substructure):
After taking one coin c, the remaining subproblem is to make change for N = N − c using the same denominations, and an optimal solution for N consists of c plus an optimal solution for N'.

Proof: By the greedy-choice claim, some optimal solution for N includes c. Removing that c leaves a solution for N'. If that leftover set were not optimal for N', we could replace it by a better one and improve the total number of coins for N, contradicting optimality. Thus the remainder must be optimal.

- **3c.** Work: O(log N) — since the number of steps is at most the number of bits in N.
  Span: O(log N) — the steps depend on each other, so they happen one after another.

- **4a.**

Let's say the coin denominations are {1, 3, 4} and N = 6. According the greedy algorithm we defined in the previous questions: Take the largest coin <= 6 -> 4 (Remaining = 2)

Take the largest coin <= 2 -> 1 (Remaining = 1)

Take the largest coin <= 1 -> 1 (Remaining = 0)

Greedy solution: 4 + 1 + 1 -> 3 coins

But optimal solution is 3 + 3 -> 2 coins

- **4b.**

Optimal substructure property: An optimal way to make change for amount N can be built from optimal solutions to smaller subproblems.
If the first coin chosen is d, then the remaining amount is N - d.
The optimal solution for N is therefore $d \cup$ (optimal solution for N - d).

Proof: If the solution for N - d were not optimal, we could replace it with a better one and get fewer coins for N, which contradicts optimality.
Hence, each optimal solution for N contains an optimal solution for a smaller amount.

Recursive idea: To make any amount: 1. Start from amount 0, which needs 0 coins.
2. For each higher amount, find the fewest coins by trying all possible first coins d and choosing the option that uses the fewest total coins.

- **4c.** Using the optimal substructure property from 4b, we can design a dynamic programming algorithm to find the fewest coins needed to make change for any amount N using denominations $D = d_1, d_2, ..., d_k$.

Algorithm (bottom-up):
1. Create an array A[0..N], where A[i] = minimum number of coins to make amount i.
2. Set A[0] = 0.
3. For each amount i from 1 to N:
- Set A[i] = ∞ initially.
- For each coin d in D:
- If $d \leq i$, update A[i] = min(A[i], 1 + A[i - d]).
4. The answer is A[N] (or report "no solution" if it remains ∞).

Example:
For $D = 1, 3, 4$ and $N = 6$:
A[6] = min(1 + A[5], 1 + A[3], 1 + A[2]) = 2,
corresponding to coins 3 + 3.

Work: $O(N \cdot k)$: we compute each entry once and check all k coins.
Span: $O(N)$: each entry depends on smaller ones, so the computation is sequential.

- **5a.**

We sort tasks by finish time. For task j, let p(j) be the index of the last task that finishes on or before s_j (or 0 if none).

An optimal schedule for the first j tasks must do one of two things:

1. Exclude task j Then the best value is exactly the best value achievable using only the first j—1 tasks.
2. Include task j Then we gain v_j and may only add tasks among the first p(j) (those that finish before s_j).
   So the best value in this case is v_j plus the best value achievable using tasks up to p(j).

Therefore, the best value for the first j tasks is the maximum of: - the best value for the first j—1 tasks, and - v_j plus the best value for tasks up to p(j).

This shows optimal substructure: the optimal solution for j is built from optimal solutions to smaller sub-problems (j—1 or p(j)).

- **5b.**

Tasks: A=(1,3,50), B=(3,10,60), C=(1,10,120).
Greedy picks A then B -> total 110, but taking C alone gives 120 (which is better).
So earliest finish fails with weights.

- **5c.**

1. Sort all tasks by their finish times.

2. For each task j, find p(j) - the index of the last task that finishes before task j starts (or 0 if there is none).

3. Create an array A[0...n] where A[j] stores the best total value achievable using the first j tasks

4. Fill the array with this rule:
   - A[0] = 0
   - For each task j from 1 to n:
     – A[j] =max(A[j - 1], v_j + A[p(j)])
       * Either skip task j, or take it and add its value plus the best non-overlapping total before it.

5. The final answer is A[n].

Work: Sorting takes $O(n \log n)$, finding all p(j) values takes $O(n \log n)$, and filling the array takes O(n)
So total work is $O(n \log n)$.

Span:
The steps depend on previous results in the array, so the DP part runs sequentially with O(n) span.