Assignment 3

Quinn Perfetto, 104026025
60-454 Design and Analysis of Algorithms

March 21, 2017

Question 1 (a).

Idea: Sum consecutive elements of the input array until the sum exceeds M. Once this happens, add the offending index to the subdivision and reset the sum.

Algorithm 1: Subdivide(W, M)

```
Input: W[1..n], 0 ≤ W[i] ≤ M, 1 ≤ i ≤ n
Output: S[1..k] such that S is an optimal subdivision of W

begin
    sum := 0;
    S := [ ];
    for i ← 1 to n do
        sum = sum + W[i];
        if sum > M then
            append(S, i - 1);
            sum := W[i];
        end
    end
end
```

Lemma 1.1. Algorithm Subdivide produces a valid subdivision of the input array W

We shall show this by inductively proving that after the mth iteration of the for loop,

$$S \text{ is a valid subdivision of } W[1..m] \wedge sum = \sum_{j=S_{last}+1}^{m} W[j]$$

Note: We take S_{last} to be the last element in S if it exists, and 0 otherwise.

Proof. (Induction Basis) We first note that sum is initialized to 0. After control reaches line 4 for the first time we have,

$$sum = sum + W[1] \Rightarrow sum = W[1] = \sum_{j=1}^{1} W[j]$$

Note that S was initialized to $[\ ]$. Since $sum = W[1] \leq M$, control will not enter the if statement on line 5, thus S will remain empty and $S_{last} = 0$. Further since $W[1..m = 1]$ is a single element list such that $W[1] \leq M$, $S = [\ ]$ is vacuously a valid subdivision of W.

(Induction Hypothesis) Assume that after k iterations of the for loop,

$$S \text{ is a valid subdivision of } W[1..k] \wedge sum = \sum_{j=S_{last}+1}^{k} W[j]$$

(Induction Step) **Case 1:** $sum > M$

By the induction assumption S is a valid subdivision of $W[1..k]$, by the defintion of a valid subdivision we thus have,

$$\sum_{j=S_{last}+1}^{k} W[j] \leq M \tag{I}$$

After appending $i - 1 = k$ to S, $S_{last} = k$. Therefore (I) is equivalent to,

$$\sum_{j=S_{last-1}+1}^{S_{last}} W[j] \leq M$$

Further since $\sum_{j=S_{last}+1}^{k+1} W[j] = W[k+1] \leq M$ we have S is a valid subdivision of $W[1..k+1]$.

After assigning $sum = W[k+1]$ we also have $sum = \sum_{S_{last}+1}^{k+1} W[j]$.

Case 2: $sum \leq M$

Since by our inductive assumption S is a valid subdivision of $W[1..k]$ and,

$$\begin{aligned} sum &= \sum_{j=S_{last}+1}^{k} W[j] + W[k+1] \\ &= \sum_{j=S_{last}+1}^{k+1} W[j] \\ &\leq M \end{aligned}$$

We have S is a valid subdivision of $W[1..k+1]$. $\square$

Therefore by Lemma 1.1, after n iterations S will be a valid parition of $W[1..n]$.
Hence the algorithm produces a valid subdivision of W.

Lemma 1.2. Algorithm Subdivide produces an optimal subdivision of the input array in terms of size

Proof. (Contradiction) Suppose to the contrary that Algorithm Subdivide does not produce an optimal subdivision of the input array. Let S be the subdivision produced by Algorithm Subdivide for some input array W, and let S' be a valid subdivision of W such that $|S'| < |S|$ (i.e. S' is more optimal than S). Since $|S'| < |S|$, $\exists i_j, i_{j+1} \in S$ and $\exists i_x, i_{x+1} \in S'$ such that $i_x \leq i_j < i_{j+1} < i_{x+1}$ (I). Such indices must exist for if they didn't the solutions would be equal in size. It can be seen that extending any subdivision produced by Algorithm Subdivide would result in an invalid subdivision as,

$$\exists i_k, \sum_{j=i_k+1}^{i_{k+1}} W[j] > M$$

(I) implies that S' contains a subdivision that is an extension of a subdivision of S, and thus has a sum larger than M. Therefore S' is an invalid subdivision. Thus S must be optimal. □

Lemma 1.3. Algorithm Subdivide runs in O(n) time

The for loop iterates over each of the n elements of W, and performs two operations for each iteration (i.e. one addition and one comparison).
We therefore have T(n) = O(2n) = O(n).

Question 1 (b). The greedy algorithm presented above will not produce an optimal subdivision if W contains negative elements. If W contains negative elements, extending a subdivision does not necessarily increase it's sum, and thus greedily ending subdivisions does not guarantee optimality.

Example: W = $[1, 2, 10, -9]$, $M = 10$
$S_{greedy} = [2]$, $S_{optimal} = []$

Question 2 (a). *Proof.* Given $s_i < s_j \wedge h_i < h_j$, by symmetry 3 different cases arise:

Case 1: $s_i < s_j \leq h_i < h_j$ We thus have,

$$|h_j - s_j| + |h_i - s_i| = h_j - s_j + h_i - s_i$$

And,

$$|h_j - s_i| + |h_i - s_j| = h_j - s_i + h_i - s_j$$

I.e.

$$|h_j - s_j| + |h_i - s_i| = |h_j - s_i| + |h_i - s_j|$$

Case 2: $s_i \le h_i \le s_j \le h_j$ We thus have,

$$|h_j - s_j| + |h_i - s_i| = h_j - s_j + h_i - s_i$$

And,

$$|h_j - s_i| + |h_i - s_j| = h_j - s_i + s_j - h_i$$

I.e.,

$$-(s_j - h_i) \le s_j - h_i \qquad\qquad (s_j \ge h_i \Rightarrow s_j - h_i \ge 0)$$

Case 3: $s_i \le h_i < h_j \le s_j$ We thus have,

$$|h_j - s_j| + |h_i - s_i| = s_j - h_j + h_i - s_i$$

And,

$$|h_j - s_i| + |h_i - s_j| = h_j - s_i + s_j - h_i$$

I.e.,

$$-(h_j - h_i) \le h_j - h_i \qquad\qquad (h_j \ge h_i \Rightarrow h_j - h_i \ge 0)$$

The remaining cases can be expressed by simply swapping h_i with s_i and s_j with h_j. Therefore the statement holds true for all cases. □

Question 2 (b).

Idea: Let $D[i, j]$ represent the least difference mapping between $H[i..n]$ and $S[j..m]$. For any $i,\ j$ two options arise:

- Pair H_i with S_j, in which case $D[i,\ j] = |H_i - S_j| + D[i+1, j+1]$
- Don't pair H_i with S_j, in which case $D[i,\ j] = D[i,\ j+1]$

The optimal result is thus the minimum of the two options.

Base cases:

- $D[n, m]$ = the least difference mapping between $H[n..n]$ and $S[m..m]$ = $|H_n - S_m|$
- $D[i,\ m] = \infty$, $1 \le i \le n-1$ since each value in H must map to a distinct value in S
- $D[n+1, i] = 0$, $1 \le i \le m$ since $H[n+1..n]$ is empty

Algorithm 2: LeastDifferenceMapping(H, S)

Input: $H = \{h_j \mid 1 \leq j \leq n\}, S = \{S_j \mid 1 \leq j \leq m\},\ n \leq m$
Output: $min(\sum_{i=0}^{n} |H[i] - S[i]|)$

```
begin
    for i ← 1 to m do
        D[n+1, i] = 0;
    end
    for i ← 1 to n−1 do
        D[i, m] = ∞;
    end
    SortAscending(H);
    SortAscending(S);
    D[n,m] = |H[n] − S[m]| ;
    for i ← n to 1 do
        for j ← m - 1 to 1 do
            D[i,j] = min(|H[i] − S[j]| + D[i+1, j+1], D[i, j+1]);
        end
    end
    return D[1,1]
end
```

Lemma 2.1. Algorithm LeastDifferenceMapping correctly produces the mapping from H to S such that $\sum_{j=1}^{n} |h_j - sj|$ is minimized

(The optimal substructure)
Consider a one-to-one mapping from two sequences sorted in ascending order $h_i h_{i+1} ... h_n$ to $s_j s_{j+1} ... s_m$.

We define the least difference mapping as a mapping that minimizes $\sum_{x=i}^{n} |h_x - s_x|$.

Let $D[i,\ j]$ = the summation of the differences in the least difference mapping of $h_i h_{i+1} ... h_n$ to $s_j s_{j+1} ... s_m$.

In any least difference mapping $h_i h_{i+1} ... h_n$ to $s_j s_{j+1} ... s_m$,

1. If h_i is mapped to s_j, then $h_{i+1} ... h_n$ is mapped to $s_{j+1} ... s_m$ and must be a least difference mapping. Otherwise, a more optimal mapping from $h_{i+1} ... h_n$ to $s_{j+1} ... s_m$ combined with the mapping from h_i to s_j would produce a mapping with a smaller difference, a contradiction!

   It follows that $D[i,j] = |h_i - s_j| + D[i+1,\ j+1]$.

2. If h_i is not mapped to s_j, then $h_i ... h_n$ is mapped to $s_{j+1} ... s_m$ and must must be a least difference mapping. Since the sequences are sorted, by the property proven in 2 (a)

any other mapping could be swapped to decrease the sum, thus it must be a minimum.

It follows that $D[i, j] = D[i, j+1]$.

We thus obtain the following recurrence:

$$\begin{aligned} D[i,j] = min\{|h_i - s_j| &+ D[i+1, j+1], \\ & D[i, j+1]\} \end{aligned}$$

Clearly,

$$\begin{aligned} &D[n,m] = |h_n - s_m| && \text{(Only one possible mapping)} \\ &D[n+1, i] = 0 && \text{(H is empty)} \\ &D[i,m] = \infty && \text{(Not enough elements in S for a distinct mapping)} \end{aligned}$$

Lemma 2.2. Algorithm LeastDifferenceMapping runs in O($mlgm$ + mn) time

Initialization: Initializing the first and second base cases perform m and $n-1$ operations respectively.

Sorting: Sorting H and S requires at most $nlgn$ and $mlgm$ operations respectively.

Filling the table: The outer loop performs n iterations, while the inner loop performs $m-1$ iterations. Each of the $m-1$ iterations performs a constant amount of work, we therefore have $n(m-1) = nm - n = O(mn)$ total operations.

Total: In summation,

$$\begin{aligned} T(n,m) &= O(m) + O(n-1) + O(nlgn) + O(mlgm) + O(mn) \\ &= O(mlgm) + O(mn) && (n \le m) \\ &= O(mlgm + mn) \end{aligned}$$

Question 3 (a).

Proof. Let G = (V, E) be a connected simple graph such that $\exists j, 1 \le j \le d_1{+}1,\ \{v_1, v_j\} \notin E$.

Let $v_j \in V, \{v_1, v_j\} \notin E, 1 \le j \le d_1 + 1$.
Since v_1 is adjacent to d_1 verticies, and $|\{v_i \mid 2 \le i \le d_1 + 1\} - \{v_j\}| = d_1 - 1$, v_1 must be adjacent to a vertex outside of this range. That is,

$$\exists v_\ell \in V,\ d_1 + 1 < \ell \le n, \{v_1, v_\ell\} \in E$$

By transitivity $j < \ell$, therefore $d_j \ge d_\ell$. Further since v_1 is adjacent to v_ℓ and not adjacent to v_j, there must exist some vertex, namely $u(\ne v_j, v_l)$, that is adjacent to v_j and not v_ℓ. Hence we have,

$$\{v_1, v_j\} \notin E \wedge \{u, v_\ell\} \notin E \wedge \{v_1, v_\ell\} \in E \wedge \{u, v_j\} \in E$$

□

Question 3 (b).

Proof. We must construct a graph G' with the same degree sequence as G, but $\{v_1, v_j\} \in E'$.

$\forall v_j \in V,\ \{v_1, v_j\} \notin E,\ 2 \leq j \leq d_1 + 1$ we assume the existance of the corresponding v_ℓ, and u in G by the previously proven theorem in Question 3 a.

E' can then be constructed by performing the following transformations to E:

- Connect v_1 to v_j
- Connect u to v_ℓ
- Disconnect v_1 from v_ℓ (Note that this restores each to their original degrees)
- Disconnect v_j from u (Note that this restores each to their original degrees)

Thus G' contains the same degree sequence as G, but contains an edge from v_1 to v_j. □

Question 3 (c).

Idea: After sorting the degree sequence, we can apply the theorem proved in Question 3 (b) to assert that there exists a graph with an identical degree sequence such that $\forall v_i, \{v_1, v_i\} \in E, 2 \leq i \leq d_1 + 1$. Thus by removing v_1 from the graph, we reduce the degree of the following d_1 vertices by 1. This process can be continued until:

1. We are left with a single vertex of degree 0. In this case we have a valid graph.
2. The degree of the highest vertex is greater than $|V| - 1$. In this case the graph is invalid, as the degree of a single vertex cannot be more than $|V| - 1$ in a simple graph.

The degree sequence shall be stored in a *linked list* to allow for constant time removal and reordering of the elements.

Algorithm 3: IsValidGraph(D)

```
Input: D[1..n] a linked list of integers
Output: Whether or not a graph exists with the degree sequence D

begin
    SortDescending(D);
    return IsValidGraph'(D);
end

Function IsValidGraph'(D_sorted)
    begin
        if D_sorted is empty then
            return true;
        end
        // Remove the first element and return it
        d_1 = pop_first(D_sorted);
        // Pointer to the head of the linked list
        ptr := head(D_sorted);
        for i ← 1 to d_1 do
            if ptr == null then
                return false;
            end
            ptr.value = ptr.value - 1;
            ptr = ptr.next;
        end
        MaintainSort(D_sorted);
        return IsValidGraph'(D_sorted);
    end
```

Lemma 3.1. Algorithm IsValidGraph correctly determines if a graph exists with the provided degree sequence

Note: IsValidGraph simply sorts the degree sequence in descending order and then calls IsValidGraph'. The remainder of this proof will reference IsValidGraph', and assume the initial input was sorted. This sorting will be accounted for in the following time complexity analysis.

We shall show correctness by induction on the input size n.

(Induction Basis) $n = 0$ The null graph is a graph with an empty degree sequence. Therefore there does exist a graph when $n =$. The algorithm will clearly return true as D_{sorted} will be empty upon the initial call. Therefore the algorithm works correctly.

(Induction Hypothesis) Suppose $\forall k < n (n > 0)$, the algorithm correctly determines if there exists a graph with the given degree sequence of size k.

(Induction Step) Let D' be a degree sequence of size n sorted in descending order. Following the call to pop_first on Line 4, d_1 contains the maximum value in D' (since it is assumed to be sorted in descending order), and $|D'| = n - 1$.