600.363/463 Algorithms - Fall 2013
Solution to Assignment 3

(120 points)

I (30 points)

(Hint: This problem is similar to parenthesization in matrix-chain multiplication, except the special treatment on the two possible operators.)

The optimal substructure: This problem has the optimal substructure. Suppose that the optimal parenthesization of $a_i \square_i \cdots a_k \square a_{k+1} \cdots a_j$ after a_k, then the parenthesization of both the "prefix" $a_i \square_i \cdots a_k$ and the "suffix" $a_{k+1} \cdots a_j$ must be optimal parenthesized. If $\square_k$ is '+', the "suffix" is maximized when the expression $a_{k+1} \cdots a_j$ is maximized while if $\square_k$ is '−', the expression $a_{k+1} \cdots a_j$ is minimized. Therefore we record both the maximum and minimum values of each subproblem. The computation of the minimization is similar.

Overlapping subproblems: It is obvious that during the recursion, subproblems (max or min) with smaller size will be revisited again and again in the computation of bigger subproblems.

Based on the analysis above, we need to record both the maximum and minimum value of each subproblem. Let $M[i,j]$ denote the maximum value of expression $a_i \square_i \cdots a_j$, and $m[i,j]$ denote the minimum value of $a_i \square_i \cdots a_j$, then recursion is:

$$M[i,j] = \begin{cases} \max_{i \le k < j}\{M[i,k] + M[k+1,j]\} & \text{if } \square_k =' +' \\ \max_{i \le k < j}\{M[i,k] - m[k+1,j]\} & \text{if } \square_k =' -' \end{cases}$$

$$m[i,j] = \begin{cases} \min_{i \le k < j}\{m[i,k] + m[k+1,j]\} & \text{if } \square_k =' +' \\ \min_{i \le k < j}\{m[i,k] - M[k+1,j]\} & \text{if } \square_k =' -' \end{cases}$$

The $M[i,j]s$ and $m[i,j]s$ can be computed in the order of increasing value of $j - i$. The resulting bottom-up algorithms is shown below.

Algorithm 1: MAXIMIZATION-EXPRESSION

Input: A sequence of n positive numbers and $n-1$ operators in the form of $a_1 \square_1 a_2 \square_2 \cdots a_{n-1} \square_{n-1} a_n$

Output: A parenthesised expression that the value is maximized.

```
1  for i ← 1 to n do
2  |  M[i, i] ← a_i;
3  |  m[i, i] ← a_i;
4  end
5  for l ← 2 to n do
6  |  for i ← 1 to n − l + 1 do
7  |  |  j ← i + l − 1;
8  |  |  M[i, j] = −∞;
9  |  |  m[i, j] = +∞;
10 |  |  for k ← i to j − 1 do
11 |  |  |  if □_k ==' +' then
12 |  |  |  |  p ← M[i, k] + M[k + 1, j];
13 |  |  |  |  q ← m[i, k] + m[k + 1, j];
14 |  |  |  end
15 |  |  |  else
16 |  |  |  |  p ← M[i, k] − m[k + 1, j];
17 |  |  |  |  q ← m[i, k] − M[k + 1, j];
18 |  |  |  end
19 |  |  |  if p > M[i, j] then
20 |  |  |  |  M[i, j] ← p;
21 |  |  |  |  s[i, j] ← k;
22 |  |  |  end
23 |  |  |  if q < m[i, j] then
24 |  |  |  |  m[i, j] ← q;
25 |  |  |  end
26 |  |  end
27 |  end
28 end
29 return M and s;
```

The algorithm has 3-levels nested loop structure, and each loop index (l, i, k) takes at most $n-1$ values. Therefore the algorithm takes $O(n^3)$ time. Since matrices M, m, s are of size $n \times n$, it requires $\Theta(n^2)$ space.

II (30 points)

Proof. Let B denote the matrix after phase 1 and C denote the matrix after phase 2. Let $\bullet_{ij}$ denote the element of a matrix at i^{th} row and j^{th} column. We show that for every i and j, $c_{ij} < c_{i,j+1}$, establishing that the rows of C are sorted.
Since collumns of C are sorted, i.e.

$$c_{1,j+1}, c_{1,j+1}, \cdots, c_{i-1,j+1} \leq c_{i,j+1}$$

Each of the above i elements has an element of $j+1^{th}$ column of B. Each of them has a small element in the j^{th} column of B (since rows of B are sorted). Hence there are at least i elements in the j^{th} column of B each of which is less than or equal to $c_{i,j+1}$. Hence c_{ij}, the i_{th} smallest element in the j^{th} column of B, is less than $c_i, j+1$. □

Alternative proof by contradiction:
For any two elements b_{ij} and $b_{i,j+1}$ in the i^{th} row of B, where $1 \leq i \leq m, 1 \leq j < n$, since rows of B are sorted, $b_{ij} < b_{i,j+1}$. Let $c_{i',j+1}$ be the same element as $b_{i,j+1}$, i.e. $c_{i',j+1} = b_{i,j+1}$.

If the order of a row is not maintained after phase 2, let us assume that $c_{i',j} > c_{i',j+1}$, where $c(i', j)$ is the i'^{th} order statistics of column j, so does $c_{i',j+1}$, since the columns in C are sorted. Hence there are i' elements in column $j+1$ of C no greater than $c_{i',j+1}$. Since $c_{i',j+1} = b_{i,j+1}$, and column $j+1$ in matrix B and C contains the same elements, there are i' elements in column $j+1$ of B no greater than $b_{i,j+1}$. Since the rows of B is sorted, column j of B has i' elements, which corresponds to the i' elements in column $j+1$, no greater than $b_{i,j+1}$, and those do not include b_{ij}. Since $b_{ij} < b_{i,j+1}$, then at column j, there are at least $i'+1$ elements no greater than $b_{i,j+1}$, then no greater than $c_{i',j+1}$, as $c_{i',j+1} = b_{i,j+1}$. Since $c_{i',j} > c_{i',j+1}$, there are at least $i'+1$ elements in j^{th} column of C no greater than $c_{i',j}$, which contradicts the fact that $c_{i',j}$ is the i'^{th} order statistics of column j. Hence the rows maintain in sorted order.

III (30 points)

Radix sort approach:

Take each element as the least significant digit and the set index as the next digit. Specifically, for every i, if set S_i contains element j, we create the pair (i, j), then sort the pairs by radix sort in $O(n)$ time. Then going down the list, we output all the js with i as the first element as the sorted S_i.

Algorithm 2: SORTING-SETS-RADIX

Input: m non-empty sets $S_1, S_2, \cdots, S_m$, which are subsets of $\{1, 2, \cdots, n\}$ and the total number of elements is n.

Output: $S_1, S_2, \cdots, S_m$ being sorted in $O(n)$ time.

```
1  k ← 0;
2  for i ← 1 to m do
3  |   for j ← 1 to length(S_i) do
4  |   |   N[k].key[1] ← S_i[j];
5  |   |   N[k].key[2] ← i;
6  |   |   k++;
7  |   end
8  end
9  for d ← 1 to 2 do
10 |   Use a stable sort to sort N on key[d];
11 end
12 k ← 1;
13 j ← 1;
14 for i ← 1 to m do
15 |   if N[k].key[2] == i then
16 |   |   S_i[j] ← N[k].key[1];
17 |   |   k++;
18 |   |   j++;
19 |   end
20 |   else
21 |   |   j ← 1;
22 |   end
23 end
```

Note that $key[1]$ has at most n possible values and $key[2]$ has m possible values. Since the elements are in $\{1, 2, \cdots, n\}$ and the total number of elements is n, $m \leq n$. Therefore sorting in line 10 takes O(n) time for each key, and the loop in lines 9-10 takes $O(n)$ time. The other two loops is simply copying elements hence takes $O(n)$ time, therefore the speed of the algorithm is $O(n)$.

Alternative counting sort approach:

Three auxiliary arrays are introduced: $C[1..n]$ records the counts, $B[1..n]$ save the sorted results of all elements and $T[1..n]$ record the affiliations.

Algorithm 3: SORTING-SETS-COUNTING

Input: m non-empty sets $S_1, S_2, \cdots, S_m$, which are subsets of $\{1, 2, \cdots, n\}$ and the total number of elements is n.

Output: $S_1, S_2, \cdots, S_m$ being sorted in $O(n)$ time.

▷ Initialization.
for $k \leftarrow 1$ *to* n **do**
  $C[k] \leftarrow 0$;
end
▷ Collecting counts.
for $i \leftarrow 1$ *to* m **do**
  for $j \leftarrow 1$ *to* $length(S_i)$ **do**
    $C[S_i[j]] + +$;
  end
end
▷ Accumulating counts.
for $k \leftarrow 2$ *to* n **do**
  $C[k] \leftarrow C[k] + C[k-1]$;
end
▷ Sorting according to counts. Affiliation of each element to S_i is assigned.
for $i \leftarrow 1$ *to* m **do**
  for $j \leftarrow 1$ *to* $length(S_i)$ **do**
    $B[C[S_i[j]]] \leftarrow S_i[j]$;
    $T[C[S_i[j]]] \leftarrow i$;
    $C[S_i[j]] \leftarrow C[S_i[j]] - 1$;
  end
end
▷ Initializing local index of each set.
for $i \leftarrow 1$ *to* m **do**
  $idx[i] \leftarrow 1$;
end
▷ Moving each elements to its affiliated set
for $k \leftarrow 1$ *to* n **do**
  $i \leftarrow T[k]$;
  $S_i[idx[i]] \leftarrow B[k]$;
  $idx[i] + +$;
end
return $S_1, S_2, \cdots, S_m$;

There are 6 loops in all. Four of them are single loops and the loop index is of size n. The other two are two-level nested loop which visit each element from all S_i sets once. Therefore they take $O(n)$ time as well. Hence the algorithm takes $O(n)$ time.

IV (30 points)

The heapifying procedure is as follows: after heapifying the first $i-1$ elements, we start at $A[i]$. We compare $A[i]$ with its parent. If the parent is larger, we stop; otherwise we swap the 2 elements. If a swap happens, we go up one level and repeat the procedure. The algorithms

are specified below.

Algorithm 4: MAX-HEAPIFY-BOTTOMUP(A, i)

Input: $A[1..i-1]$ satisfying a heap and element $A[i]$ to be added to the heap
Output: $A[1..i]$ satisfying a heap

```
1  if i == 1 then
2  |  return A[1];
3  end
4  p ← ⌊i/2⌋ ;
5  if A[p] < A[i] then
6  |  swap A[i] ↔ A[p];
7  |  MAX-HEAPIFY-BOTTOMUP(A, p);
8  end
9  else
10 |  return A[1..i];
11 end
```

Algorithm 5: BUILD-MAX-HEAP(A)

Input: Array A.
Output: Max-heap A.

```
1  n ← length[A] ;
2  for i ← 1 to n do
3  |  MAX-HEAPIFY-BOTTOMUP(A, i);
4  end
5  return A;
```

The heap has depth of $\log n$. So each heapifying any element $A[i]$ takes at most $O(\log n)$ swaps. The loop index takes $O(n)$ time, hence the algorithm takes $O(n \log n)$ time. More precisely, at depth d there are at most 2^{d-1} nodes, and such a node takes at most d swaps, therefore, there are at most

$$\sum_{d=1}^{\log n} d2^{d-1} = O(n \log n)$$

swaps in building an heap of size n. Therefore the speed of the algorithm is $O(n \log n)$.