CS 6363.003 Spring 2021 Author. Yaokun Wu (yxw200031)
Homework 3 Problem 1

Describe and analyze a dynamic programming algorithm to compute, given a list of integers separated by + and − signs, the maximum possible value the expression can take by adding parentheses.

Solution: Let dp[i][j] represent maximum possible value for the sequence from index i to j. Then,

dp[i][j] = max(A[i] +/- dp[i+1][j], dp[i][j-1] +/- A[j]), where i < j - 1
or = A[i] +/- A[j], where i = j - 1
or = A[i], where i = j
where +/- is determined by the signs in between integers.

```
MaxPosVal(A[1..n], B[1..n−1]):
  for i ← 1 to n
      dp[i][i] ← A[i]
  for i ← 1 to n−1
      if B[i] =′ +′
          dp[i][i+1] ← A[i] + A[i+1]
      else
          dp[i][i+1] ← A[i] − A[i+1]
  for i ← n down to 1
      for j ← i+2 to n
          if B[i] =′ +′
              sum1 ← A[i] + dp[i+1][j]
          else
              sum1 ← A[i] − dp[i+1][j]
          if B[j−1] =′ +′
              sum2 ← dp[i][j−1] + A[j]
          else
              sum2 ← dp[i][j−1] − A[j]
          dp[i][j] ← max(sum1, sum2)
  return dp[1][n]
```

Since dp[i][j] depends on dp[i+1][j] and dp[i][j-1], the calculation sequence should be i going bottom up and j going left to right. Base case is when i = j - 1 and i = j given in the recursive definition. The solution is in dp[1][n].

Time complexity is $O(n^2)$ since i and j both goes from 1 to n.

■

CS 6363.003 Spring 2021 Author. Yaokun Wu (yxw200031)

Homework 3 Problem 2

(a) Describe a dynamic programming algorithm to compute, given the tree T and an integer r, the size of the smallest subset of vertices whose clustering cost is at most r.

Solution:

```
smallestSize(T, r):
    return smallestSizeHelper(T, 0)
```

```
smallestSizeHelper(T, dist):
    if T = null
        return 0
    if dist > r
        return inf
    if T.size[dist] != null
        return T.size[dist]

    subSize ← min(smallestSizeHelper(T.left, 0) + smallestSizeHelper(T.right, 0),
    smallestSizeHelper(T.left, dist + 1) + smallestSizeHelper(T.right, 0),
    smallestSizeHelper(T.left, 0) + smallestSizeHelper(T.right, dist + 1),
    smallestSizeHelper(T.left, dist + 1) + smallestSizeHelper(T.right, dist + 1))

    if dist = 0
        T.size[dist] ← subSize + 1
    else
        T.size[dist] ← subSize
    return T.size[dist]
```

For the basecase, when the node is null, return 0 meaning no vertices required to be marked and when dist > r, return inf meaning the current clustering cost is invalid.

Assume that smallestSizeHelper(childnode, dist1) return the smallest subset of vertices for the subtree rooted at childnode with distance of dist to its parent. smallestSizeHelper(node, dist2) will give the smallest subset of vertices based on all its children returns in the algorithm. Therefore, smallestSizeHelper(node, dist2) will return the smallest subset of vertices for the subtree rooted at node with distance of dist2 to its parent.

Time complexity is $O(nr)$ since there are n nodes in the tree and the dist can reach up to r. And at each node, a dist will be only calculated once and stored.

■

(b) Describe an algorithm to compute, given the tree T and an integer k, the minimum clustering cost of any subset of k vertices in T.

Solution: The algorithm is shown below.

```
minCost(T, k):
    n gets calculate number of vertices in the T
    left ← 0, right ← n
    while left < right
        mid ← left + ⌊(right − left)/2⌋
        minSize ← smallestSize(T, mid)
        if minSize > k
            left ← mid + 1
        else
            right ← mid
    return left
```

■

Assume that smallestSize(T,mid) give the correct answer. When smallestSize(T,mid) is greater than k meaning that the clustering cost is too low such that a bigger set size is needed. So the left bound should be left = mid + 1. When smallestSize(T,mid) is lower than/equals to k meaning that the clustering cost may be too high such that a smaller set size may achieve. So we shrink the search space by half by using right = mid. (We do not exclude the mid because there are cases when smallestSize(T,mid) is equals to k). In the base case when left equals right, meaning that the exact clustering cost is found and return.

Time complexity is $O(n^2 \log n)$ because a single search takes $O(nr)$ time, where r here is n and the search space starts from n and each time shrink by half.

CS 6363.003 Spring 2021 Author. Yaokun Wu (yxw200031)

Homework 3 Problem 3

Describe and analyze an algorithm to compute an optimal *ternary* prefix-free code for a given array of frequencies $f[1 .. n]$.

Solution: We could build the tree in the following fashion.

```
BuildHuffman(f[1..n]):
  for i ← 1 to n
      L[i] ← 0; M[i] ← 0; R[i] ← 0
      INSERT(i, f[i])
      size = n
  for i ← n to 3n − 1
      if size = 1
          x ← EXTRACTMIN()
          INSERT(i, f[x])
          return
      if size mod 2 = 0
          x ← EXTRACTMIN()
          y ← EXTRACTMIN()
          f[i] ← f[x] + f[y]
          INSERT(i, f[i])
          L[i] ← x; P[x] ← i
          M[i] ← y; P[y] ← i
          size ← size − 2
      else
          x ← EXTRACTMIN()
          y ← EXTRACTMIN()
          z ← EXTRACTMIN()
          f[i] ← f[x] + f[y] + f[z]
          INSERT(i, f[i])
          L[i] ← x; P[x] ← i
          M[i] ← y; P[y] ← i
          R[i] ← z; P[z] ← i
          size ← size − 3
```

■

Greedy proof:

Since the tree node in the optimal tree either has at least two child nodes, otherwise we can always remove that node and move everything up, there are two cases.

When there are less than or equal to three characters, it is the base case and can be merged into a single parent. When there are more than three characters, two cases are shown below.

Case 1: When the number of tree node is odd, it is obvious the optimal code tree should be a complete tree. Let x, y and z be the three least frequent characters. There is an optimal code tree in which x, y and z are siblings.

There are a three leaves a, b and c at depth d. Suppose x is neither a nor b nor c.

Swap leaves a and x to get tree T'.

cost(T')

= cost(T) + f[x](depth(a) - depth(x)) - f[a](depth(a)-depth(x))

= cost(T) + (f[x] - f[a])(depth(a) - depth(x)).

But f[x] - f[a] is non-positive and depth(a) - depth(x) is non-negative, so cost(T') <= cost(T) + 0 = cost(T). So there's an optimal tree T' where x has maximum depth.

If x's new neighbor is not y or z, we can swap the two neighbors with y and z again and the same conclusion holds true.

Case 2: When the number of tree node is even, one of the leaf node should have two child nodes. Let x and y be the two least frequent characters and there is an optimal code tree in which x, y are siblings.

There are a pair of leaves a and b at depth d. Suppose x is neither a nor b.

Swap leaves a and x to get tree T'.

cost(T')

= cost(T) + f[x](depth(a) - depth(x)) - f[a](depth(a)-depth(x))

= cost(T) + (f[x] - f[a])(depth(a) - depth(x)).

But f[x] - f[a] is non-positive and depth(a) - depth(x) is non-negative, so cost(T') <= cost(T) + 0 = cost(T).

So there's an optimal tree T' where x has maximum depth.

If x's new neighbor is not y, we can swap the two nodes again and the same conclusion holds true.

Recursive strategy proof:

For the basecase, if n = 1 or n = 2 or n = 3 then the code is optimal because there is no choices. Otherwise, let f[1 .. n] be the input frequencies.

Case 1:

when n is odd, assume f[1], f[2]. f[3] have the smallest frequencies.

Let T be any code tree for f[1 .. n] where 1, 2, 3 are siblings, and let T' = T exclude 1, 2, 3.

For simplicity, we'll treat the parent of characters 1, 2 and 3 as character n + 1. We'll use f[n+1] := f[1] + f[2] + f[3].

T' is a code tree for f[4 .. n+1].

cost(T) = sum (i=1 to n) f[i] * depth(i)

= sum (i=4 to n+1) depth(i) + f[1] * depth(1) + f[2] * depth(2) + f[3] * depth(3) - f[n + 1] * depth(n+1)

= cost(T') + f[1] * depth(1) + f[2] * depth(1) + f[3] * depth(1) - f[n+1] * (depth(1) - 1)

= cost(T') + f[1] + f[2]+ f[3] + (f[1] + f[2] + f[3] - f[n+1]) * (depth(1) - 1)

= cost(T') + f[1] + f[2] + f[3]

Since f[1], f[2] and f[3] are fixed, cost(T) is minimized when cost(T') is minimized. Also note, when n is odd for tree T, T' is also odd. So by the induction hypothesis, cost(T') is minimized by recursively building the codes for T'.

Case 2: when n is even, assume f[1], f[2] have the smallest frequencies.

Let T be any code tree for f[1 .. n] where 1 and 2 are siblings, and let T' = T exclude 1, 2.

For simplicity, we'll treat the parent of characters 1 and 2 as character n + 1. We'll use f[n+1] := f[1] + f[2].

T' is a code tree for f[3 .. n+1].

cost(T) = sum (i=1 to n) f[i] * depth(i)

= sum (i=3 to n+1) depth(i) + f[1] * depth(1) + f[2] * depth(2) - f[n + 1] * depth(n+1)

= cost(T') + f[1] * depth(1) + f[2] * depth(1) - f[n+1] * (depth(1) - 1)

= cost(T') + f[1] + f[2] + (f[1] + f[2] - f[n+1]) * (depth(1) - 1)

= cost(T') + f[1] + f[2]

Since f[1] and f[2] are fixed, cost(T) is minimized when cost(T') is minimized. Also note, when n is even for tree T, the nodes for T' is odd. We can therefore recursively minimize the cost of T' in case 1.

Time complexity is $O(n \log n)$ if we use a min-heap as the priority queue because it takes O(log n) time to do each queue operation and there are O(n) queue operations total, so the total time to build the tree is $O(n \log n)$.

CS 6363.003 Spring 2021 Author. Yaokun Wu (yxw200031)

Homework 3 Problem 4

Describe and analyze an algorithm to compute the smallest number of times Judy needs to teleport to travel from galaxy s to galaxy t so that the total cost is a multiple of five dollars.

Solution: The algorithm is shown below.

```
numTele(G, W, s, t):
  put(s, 0) in a queue
  mark (s, 0)
  while the queue is not empty
      size = size of queue
      while size > 0
          take (v, cost) from queue
          size−− = 1
          if v = t and cost mod 5 = 0
              break
          else
              for each edge vw in G
                  if (w, cost + W(vw)) is unmarked
                      mark (w, cost + W(vw))
                      put(w, cost + W(vw)) into the queue

      count++ = 1
  return count
```

The algorithm uses breadth first search and the state is represented by (position, cost), anytime a state is reached, it is recorded and will not reached again to prevent infinite traversal. The first time when the point is reached with the required amount of cost will be returned.

Time complexity is O(V' + E') = O(V + E) where V' = 5V and E' = 5E because no matter how the toll is distributed, the back and forth teleport of 5 times always ends up the multiple of 5 as an upper bound. ■