Analysis of Algorithms

Alvin Lin (axl1439) and William Leuschner (wel2138)

August 2017 - December 2017

Problem 1

Algorithm Description

For this problem, we implemented a dynamic programming algorithm to find the maximum length of the longest convex subsequence in a sequence of numbers. The heart of the solution is described as follows:

$S[i][j]$ = the maximum length of a convex subsequence ending with
a_i where the second to last element in the subsequence is a_j

$$S[i][j] = \begin{cases} i+1 & if\ i < 3 \\ 2 & if\ j = 1 \\ 3 & if\ i = 3 \wedge a_1 \dots a_3 \text{ is convex} \\ 2 & if\ i = 3 \wedge a_1 \dots a_3 \text{ is not convex} \\ max(S[i]) & if\ i = j \\ max(S[i][j], S[j][k] + 1, 2)\ \forall k\ (k \mid 1 \leq k < j) & otherwise \end{cases}$$

The solution is yielded by taking the maximum of the elements along the diagonal of the solution matrix ($max(S[i][j])$ where $i = j$).

Running Time Estimate

The dynamic programming fills a two dimensional solution matrix, and requires a linear pass to fill each element of the solution matrix and to find the final solution, which makes the runtime $n^3 + n$, which is $O(n^3)$.

Psuedocode

```
def solve(sequence):
  let n = sequence.length
  let s = matrix of size n x n
  for i = 0 to n:
    for j = 0 to n:
      if i < 2:
        s[i][j] = i + 1
      else if j == 0:
        s[i][j] = 2
      else if i == 2:
        if sequence[0], sequence[1], sequence[2] is convex:
          s[i][j] = 3
        else:
          s[i][j] = 2
      else if i == j:
        s[i][j] = max(s[i])
      else:
        for k = 0 to j:
          if sequence[k], sequence[j], sequence[i] is convex:
            s[i][j] = max(s[i][j], s[j][k] + 1)
          else:
            s[i][j] = max(s[i][j], 2)
  result = 0
  for i = 0 to n:
    result = max(result, solution[i][i])
  return result
```

Problem 2

Algorithm Description

For this problem, we implemented a dynamic programming algorithm to find the maximum cost attainable with a sample of items placed in two backpacks. The

heart of the solution is described as follows:

$$S[k][w_1][w_2] = \text{the maximum cost of a subset of the first } k \text{ items where the weight of items in the subset is distributed in two backpacks with weight limits } w_1 w_2$$

$$S[k][w_1][w_2] = \begin{cases} 0 & if\ k = 0, w_1 = 0, w_2 = 0 \\ \texttt{max}(S[k-1][w_1][w_2], \\ \quad c_k + S[k-1][w_1 - w_k][w_2], \\ \quad c_k + S[k-1][w_1][w_2 - w_k]) & if\ w_k \leq w_1 \wedge w_k \leq w_2 \\ \texttt{max}(S[k-1][w_1][w_2], \\ \quad c_k + S[k-1][w_1 - w_k][w_2]) & if\ w_k \leq w_1 \\ \texttt{max}(S[k-1][w_1][w_2], \\ \quad c_k + S[k-1][w_1][w_2 - w_k]) & if\ w_k \leq w_2 \\ S[k-1][w_1][w_2] & otherwise \end{cases}$$

With n items and respective weight limits of W_1, W_2, the solution is thus yielded by:

$$S[n][W_1][W_2]$$

This algorithm works by trying to place the item into the two bags. If it fits in both, then it will use the maximum cost attained by placing it either, or not placing it at all. If it only fits in one, then it takes the maximum cost attained by placing it in that bag or not placing it in the bag.

Running Time Estimate

Since the dynamic programming solution matrix is a three dimensional array with dimensions $n \times W_1 \times W_2$, our running time is $O(nW_1W_2)$ since we need to populate the entire solution matrix.

Psuedocode

```
def solve(items, W1, W2):
  let n = items.length
  let s = matrix of size (n+1) x (W1+1) x (W2+1)
  for k = 0 to n + 1:
    for w1 = 0 to W1:
      for w2 = 0 to W2:
        if k == 0:
          s[k][w1][w2] = 0
        else:
          let item = items[k - 1]
          let previous = s[k - 1]
          if item.weight <= w1 and item.weight <= w2:
            solution[k][w1][w2] = max(
              previous[w1][w2],
              item.cost + previous[w1 - item.weight][w2],
              item.cost + previous[w1][w2 - item.weight]
            )
          else if item.weight <= w1:
            solution[k][w1][w2] = max(
              previous[w1][w2],
              item.cost + previous[w1 - item.weight][w2]
            )
          else if item.weight <= w2:
            solution[k][w1][w2] = max(
              previous[w1][w2],
              item.cost + previous[w1][w2 - item.weight]
            )
          else:
            solution[k][w1][w2] = previous[w1][w2]
  return solution[n][W1][W2]
```

Problem 3

Algorithm Description

For this problem, we extended it off of problem 2's dynamic programming solution matrix. We iterate backwards over the items array and pull out the item's weight and cost and the current value in the solution matrix for that item. We can subtract the value in the solution matrix from the previous value in the solution matrix to determine if the item was included in the cost calculation during the construction of the solution matrix, and if it was, we can add it to the knapsack as a item included in the valid solution.

Argument of Correctness

This algorithm reverses the construction of the optimal cost entry in the solution matrix, so we are guaranteed that the items it produces is part of the optimal set of items to maximize cost. By subtracting entries in the solution matrix, we can determine the cost of the item that was added.

Running Time Estimate

Since this algorithm relies on the solution matrix to construct the solution, it is $O(nW_1W_2)$. The solution construction simply needs to go through every item, so the construction is $O(n)$. The entire runtime is upper bounded by the time it takes to generate the solution matrix.

Pseudocode

```
def solve(solution, items, W1, W2):
  let bag1 = []
  let bag2 = []
  let w1 = W1
  let w2 = W2
  for i = items.length - 1 to 0:
    let item = items[i]
    let current = solution[i + 1][w1][w2]
    let prev = solution[i]
    if w1 >= item.weight and
        current == item.cost + prev[w1 - item.weight][w2]:
      bag1.append(i + 1)
      w1 -= item.weight
    elif w2 >= item.weight and
        current == item.cost + prev[w1][w2 - item.weight]:
      bag2.append(i + 1)
      w2 -= item.weight
  return [bag1, bag2]
```

Problem 4

Algorithm Description

For this problem, we implemented a dynamic programming algorithm to solve for the minimum number of operations needed to multiply the optimally parenthesized groups of matrices. While populating the solution array, we also stored the groups

of parenthesized matrices so that we could recursively reconstruct the solution. The heart of the solution is as follows:

$$\begin{aligned} S[l][r] = &\text{ the minimum number of steps required to multiply the} \\ &\text{matrices from index } l \text{ to } r \text{ and the grouping that} \\ &\text{yielded the resulting minimum number of steps} \\ S[l][r] = &\ min(S[l,k] + S[k+1][r] + a_{l-1}a_k a_r)\ \forall k(1 \leq k \leq r-1) \\ &\text{and store } [l,k,r] \text{ in } groupings[l][k] \end{aligned}$$

The final result for the minimum number of steps is given by:

$$S[1][n]$$

We can take the $[l,k,r]$ grouping stored at $S[1][n]$ and recursively expand it into solution string.

Running Time Estimate

Populating the solution matrix requires $O(n^3)$ time, but recursively reconstructing the solution only requires $O(n)$ time because a recursive expansion can occur at most n times.

Pseudocode

```
def solve(dimensions):
  let n = dimensions.length - 1
  let solution = list of length n
  let groupings = matrix of size n x 3
  for d = 1 to n:
    for l = 0 to n - d:
      let r = l + d
      solution[l][r] = infinity
      for k = l to r:
        let tmp = solution[l][k] + solution[k + 1][r] +
          dimensions[l] * dimensions[k + 1] * dimensions[r + 1]
        if tmp < solution[l][r]:
          solution[l][r] = tmp
          groupings[l][r] = [l, k, r]
  return solution, groupings
```

```
def reconstruct(groupings):
  let n = groupings.length
  return reconstructRecursive(groupings, groupings[0][n - 1])

def reconstructRecursive(groupings, solutionGroup):
  let l, k, r = solutionGroup
  if l + 1 == r:
    return "( A%d x A %d )".format(l + 1, r + 1)
  elif l == k:
    return "( A%d x %s )".format(l + 1,
      reconstructRecursive(groupings[k + 1][r]))
  elif k + 1 == r:
    return "( %s x A%d )".format(
      reconstructRecursive(groupings[l][k]), r + 1)
  else:
    return "( %s x %s )".format(
      reconstructRecursive(groupings[l][k]),
      reconstructRecursive(groupings[k + 1][r]))
```

Problem 5

Algorithm Description

For this problem, we implemented a dynamic programming algorithm to solve for the minimum triangulation length of a convex polygon. The heart of the solution is as follows:

$$S[i][j] = \text{the minimum length of the triangulation of the polygon vertices formed by } i \text{ through } j$$

$$S[i][j] = \begin{cases} 0 & if\ j - i < 2 \\ min\{S[i][t] + S[t][j] + dist(a_i, a_j) \mid i < t < j\} & otherwise \end{cases}$$

with the solution being $S[0][n-1] - dist(a_0, a_{n-1})$. This algorithm calculates the shortest possible triangulation for every consecutive group of k vertices and uses that value to construct the minimum possible triangulations for larger groups of vertices.

Running Time Estimate

This algorithm populates an $n \times n$ solution matrix. Since it requires linear time to populate each entry in the solution matrix, this algorithm runs in $O(n^3)$.

Pseudocode

```
def solve(coordinates):
  let n = coordinates.length
  let s = matrix of size n x n
  for i = 0 to n:
    let j = 0
    for k = i to n:
      let length = distance(coordinates[i], coordinates[k])
      if k < j + 2:
        s[j][k] = 0
      else:
        s[j][k] = infinity
        for t = j + 1 to k:
          s[j][k] = min(s[j][t] + s[t][k] + length, s[j][k])
        if j == 0 and k == n - 1:
          s[j][k] -= length
      j += 1
  return s[0][n-1]
```

References

For problem 5, we got stuck and looked up advice for dynamic programming on GeeksForGeeks:
http://www.geeksforgeeks.org/minimum-cost-polygon-triangulation/