# Homework 6

| | |
|---|---|
| Tags | Homework |
| Date | |

**Problem 1 — Exercise 19 on page 329**

ALGORITHM

```
input: s, x, y
function:
  let n = length of s
  initialize 2D array arr[0][0] to 1

  if length of x + length of y /= n // /= is not equal
    return false

  for all k from 1 to n
    for all i,j such that i+j = k:
      let s' = s[i+j]
      if arr[i][j-1] = 1 and s' = y'[j] OR arr[i-1][j] = 1 and s' = x'[i]
        arr[i][j] = 1
      else
        arr[i][j] = 0

  if for some i,j s.t. i+j=n and arr[i][j] = 1
    return true
```

PROOF

- A string s is interleaving of *x* and *y* if its symbols can be partitioned into two (not necessarily contiguous) subsequences *s′* and *s′*
- base case:
  - let string s,x, and y be an empty string.
  - the length of x + y = s, and x,y are partitioned in that they are empty sequences of an empty sequence s, so  x,y are interleaving of s by definition.
  - therefore arr[0][0] = 1 as shown in the algorithm, therefore the base case is true

- induction step:
  - we must prove that if the algorithm is true for k, the algorithm must remain true for k-1.
  - let s be the interleaving of x’ and y’ for some sequence of letters from 0 to i and i to k, x’ and y’ respectively
  - since s is the interleaving of of these two sequences, we know that the last letter of s must come from either of the sequences. remove that letter.
  - by removing that letter, we have k-1 letters and we are left with a subproblem defined by the prefixes of x’ and y’
  - by definition, if s from 1 to i + j is an interleaving of x’ from 1 to i or y’ from 1 to j then we have arr[i][j] = 1 and once again see that the last letter of s is equal to the last letter of x or y
  - therefore, the algorithm retains true for k-1 subproblems. By induction, the algorithm correctly identifies whether the string s is an interleaving of strings x and y

RUNTIME

- In a worst case scenario, there are n computations of the first loop and at max n computations found by the pairs (i,j) s.t. i+j = k
- therefore at most, there will be n*n iterations in the loops giving us a time complexity of **O(n^2)**

**Problem 2 — Exercise 21 on page 330**

ALGORITHM

```
let p(i,j) = function that returns p(j) - p(i)
let o[i,j] = array of optimal maximum profits by any transaction
let n be the consecutative days of a given stock

for i to n-1
  initialize o[i,i+1] - 1000*p(i+1, i)

for i from i to n-1
```

```
  for j from 1 to n-i
    o[i,j] = maximum between 1000*p(i,j), o[i, j-1] or o[i+1, j]

let m be the m shot strategies s.t. 1 <= m <= k
let d be the number of days s.t. 1 <= d <= n
let k[m,d] = maximum return obtained by k-shot strategies
initialize k[1,1] = 0

for d from 2 to n
  k[1,d] = q[1,d]

for m from 1 to k
  k[m,0] = 0

for m from 1 to k-1
  for d from 1 to n
    for i from 1 to d-1
      for j from i+1 to d
        k[m,d] = maximum between current k[m+1,d] and q[i,j] + k[m-1, i-1]

return maximum where m < k =  k[m,n]
```

PROOF

- The first part of the solution allows us to determine the best profit from a single transaction — we compare each of the days in the period from days i to j and build up a table of all the profits and losses computed by p(i,j)
- this algorithm finds the best profit because it takes the current profit and calculates all other profits that could be find by buying a selling a day after or buying a day earlier, for each possible iteration that can be found
- then using the maximum returns that could be found on the days, we enact the idea of the k-shot strategy by finding the profit obtained from m buy-sell transactions
- that is, for each possible number of buys that can be enacted from any to all days, we compare the maximum between the current maximum largest m-shot return and each consecutive added return by selling a day later and buying a day earlier

RUNTIME

- To find the optimal maximum profits for any transaction, we have n iterations and every iteration from 1 to n nested within that loop, therefore we have a runtime of O(n^2)

- Then for the second part of implementing the k-shot strategy, we have each k iterated over and each n day iterated over from 1 to n, then each iteration from 1 to d-1 and each iteration of that value from i+1 to d, which gives us every iteration from n nested 3 times, leaving us with a runtime of O(kn^3)
- Therefore we are left with O(kn^3 + n^2) which equates to **O(kn^3)**

**Problem 3 — Exercise 23 on page 331**

**a)**

ALGORITHM

```
for all edges e(v,w)
  if d(v) > d(w) + c_e
    return false

complete a bfs of the graph
  if each node has a path to node t
    return true
  else
    return false
```

PROOF

- the first part of the algorithm has to reject if d(v) > d(w) + c_e. Assume such a node v results a value d(v) > d(w) + c_e. That would mean that there is a path from v to t that passes through w with cost d(w) + c_e
- This would imply that the minimum distance from v to t is  d(w) + c_e, which would imply that d(v) is not the cost of the minimum cost path from node v to sink t. therefore, by proof of contradiction, the first loop of the algorithm must be true
- next, completing a traversal of the graph will show us each traversal of every node. if at least one node doesn't have a path to t, by definition it's minimum cost path will be the last node that node traverses to before it ends at a node before t.
- hence, d(v) would then not be cost of the minimum-cost path because it would have a cost that ends at t which would be greater than the node that ends before t
- therefore, by proof of contradiction, we have shown that the algorithm correctly identifies the cost of the minimum-cost path from v to t

RUNTIME

- the runtime is only exacerbated by the fact that a breadth first search traversal is required, which by definition since it traverses each node and edge along the graph V,E and determines whether a path exists, the runtime is **O(V + E)**

**b)**

ALGORITHM

```
let c'e[v] be an array of all modified costs for all nodes v
for each edge(v,w)
  set new distances c'e = c_e - d(v) + d(w)
run djikstra's algorithm from t' to compute shortest path
return shortest path
```

PROOF

- We must show that setting the new distances c’e does not change the minimum cost paths
- Assume that we can have negative paths, if c’e < 0, that would imply that d(v) < d(w) + c_e
- That cannot be true because that would imply that d does not reflect correct distances to t
- Therefore, we must consider whether c’e and c retain us the same minimum paths.
- From above, we know that if c’e > 0, then the minimum cost of a path p in (a) is represented by c(p) and minimum cost of path p found by this algorithm is c’(p) from node v to w
    - total min c(p) = c(p)
    - total min c’(p) = c(p) + d(w) - d(v) because all but the first and last node in the path p occur once positively and once negatively in the sum, resulting in the addition of d(w) - d(v)
- Therefore, our algorithm correctly modifies the cost function and computes the distances to a different sink t’ accurately.

RUNTIME

- The loop runs for each edge and node and calculates a new cost, which runs in O(m + n)
- We use djikstra's algorithm which computes the shortest path in O(mlog(n)), meaning we are left with a total of O(m + n + mlog(n)) ⇒ **O(mlog(n))**

**Problem 4 —** Exercise 9 on page 419

ALGORITHM

```
input: n injured people, k hospitals
function:
  let s be the source node, and t be the sink node
  let the network be defined by n, k, s, and t

  initialize the flow in all edges defined the graph to 0

  while there is an augmenting path between s and t
    if flow (v,w) is <= n/k and v is within half hour distance of w
      add edge e to the flow

  let f = max flow in the network
  if f != n
    return false

  return true
```

PROOF

- Let the network created by adding the edges be a residual graph. If there is a path from the source to sink in the residual graph, then it is possible to add an edge e to add flow
- If there is no edge between two nodes, that means either the flow is over the capacity of n/k for the hospital or the distance between the person and the hospital is greater than a half hour
- that means that an edge can only be added if the conditions in the prompt are met- therefore the algorithm creates the optimal residual graph. by way of contradiction, we show that our graph correctly creates a network
- using that network, assume the max flow found from this optimal residual graph is less than n, that means the amount of injured people cannot be served by all the

hospitals and therefore we must return false

- therefore, by way contradiction we know the max flow must equal the number of people n in order for valid assignment

RUNTIME

Each time a path is augmented, we increase the total flow, so at worst the number of times we find an augmenting path is **O(E * f)** where E is the number of edges traversed while augmenting and f is the value of the max flow found.

**Problem 5 — Exercise 11 on page 420**

PROOF

- The statement is false.
- The friend's argument is that on every instance of the maximum-flow problem, their algorithm finds a flow of value of at least 1/b where b > 1, s.t. the flow value is at least the maximum flow value * 1/b
- However, let the maximum flow be defined by f such that f is greater than the size of the input, let it be called n
- if f > n, then a minimum cut of edges out of a node n_i will have n edges deleted which results in a maximum flow of at most f-n
- then for the subgraph of the node n_i, each of the edges connected to n_j to n_i have at most n edges deleted, therefore leaving a minimum cut of f-n.
- for the maximum flow of f-n, that means that their algorithm is sure to find a value (f-n) * 1/b $\Rightarrow$ 1 > (1-n)/b while the actual minimum cut has a value f-n that is not equal (1-n)/b

**Problem 6 — Given a sequence of numbers find a subsequence of alternating order, where the subsequence is as long as possible. (that is, find a longest subsequence with alternate low and high elements).**

ALGORITHM

```
input: arr
function:
  let n be the length of the array
  let max = length of longest subsequence, initialize to 0
  let lengths[n][2] = 2d array of length n by 2
  for all i from 0 to n, initialize arr[i][0] and arr[n][1] with 1

  for all i from 1 to n-1
    for all j from 0 to i-1
      endI = lengths[i]
      endJ = lengths[j]
      if arr[j] > arr[i]
        set endI[1] to maximum between endI[1] and endJ[0]+1

      if arr[j] < arr[i]
        set endI[0] to maximum between endI[0] and endJ[1]+1

    set local_max to maximum between lengths[i][1] and lengths[i][0]
    set max to maximum between max and local_max

  return max
```

PROOF

- Base Case:
  - let the array be an array with 2 elements that is alternating — take [0,1] which has a  longest alternating sequence of 1
  - the only iteration in the loops will be comparing j = 0 and i = 1. since arr[1] = 1 > arr[0] = 0, we set the max to the maximum between the default value of 1 and the current max here, which is also 1. therefore the maximum is 1, proves the base condition is true
- Induction case:
  - we must prove if the algorithm retains true for k elements, it must also be give the max if there were k-1 elements
  - let the loop run i be the k-2’th index in a list of k elements, then for each subsequence from 0 to k-3th index, we check whether the longest alternating subsequence ending at i where the last element is greater than the previous element is greater or lesser than the longest alternating sequence ending at index i where the last element is lesser than its previous element.

- that must give us the maximum of the alternating subsequence for lengths[k-2][1] and and lengths[k-2][0], and getting the maximum from that must the maximum subsequent array including the k-2th element. then checking with the global maximum of whether that is the maximum or not gives us the ultimate maximum we are looking for
- therefore by induction, the algorithm gives us the longest alternating subsequence.

RUNTIME

The algorithm iterates over each element in the input array, which is a total of n times multiplied by each possible subsequence between 0 to n (i.e. 0 to 1, 0 to 2… 0 to n). Therefore there are a total of n * n iterations until we can return the maximum, which gives us a worst case time complexity of **O(n^2)**