COL351 Assignment 2

Pratik Nimbalkar, Garvit Dhawan

TOTAL POINTS

47 / 60

QUESTION 1

1 Maximum sum **18 / 18**

✓ **+ 18 pts** *Correct Solution*

QUESTION 2

Forex trading 17 pts

2.1 part 1 **10 / 10**

✓ **+ 6 pts** *Algorithm Correctness*

✓ **+ 3 pts** *Correctness Proof*

✓ **+ 1 pts** *Running Time Analysis Correctness*

2.2 part 2 **7 / 7**

✓ **+ 4 pts** *Algorithm Correctness*

✓ **+ 2 pts** *Algorithm Correctness Proof*

✓ **+ 1 pts** *Time Analysis*

QUESTION 3

3 Disjoint Paths **12 / 25**

✓ **+ 2 pts** *Identifying we need to use DP*

✓ **+ 0 pts** *Incorrect/Unattempted*

+ 10 *Point adjustment*

1 Not a complete binary tree, so height can be O(n)

2 Not clear why cant other cases be possible

3 Why?

COL351 Assignment 2

18th September 2022

Pratik Nimbalkar (2020CS10607)
Garvit Dhawan (2020CS50425)

Q.1 Maximum sum

ALGORITHM:

We make an array A of size 2*n such that the given sectores of disc are repeated twice in A (A=[p1,p2,p3….pn,p1,p2,......pn]).
We define MaxSum[i] which stores the maximum contiguous sum of subarray of A till i.

```
We initialise maxSum[i]= -∞ for all 1<=i<=2n.
Maxoverall=A[1]
OverallMaxInterval = [1,1]
Now we start from the first index
    MaxSum[1] = A[1]
    c=1
    currInterval=[1,1]
    For i in range 2 to 2n:
        If (A[i]>MaxSum[i-1] + A[i]):
            c=1
            MaxSum[i] =  A[i]
            currInterval[0]=i
            currInterval[1]=i
        Else :
            c=c+1
            If (c<=n):
                MaxSum[i] =MaxSum[i-1] + A[i]
                currInterval[1]=i
            Else :
                c=1
                MaxSum[i]=A[i]
                currInterval[0]=i
                currInterval[1]=i
        if(MaxSum[i]>Maxoverall) :
            Maxoverall = MaxSum[i]
            OverallMaxinterval = currInterval

    if(OverallMaxInterval[1]>n) :
            Overall_Interval[1]= Overall_Interval[1] -n
        if(OverallMaxInterval[0]>n) :
            Overall_Interval[0]= Overall_Interval[0] -n
        Return OverallMaxInterval
```

TIME COMPLEXITY :

In each iteration, we do finite number of O(1) operations, thus each iteration takes constant time and we do 2*n iterations. Thus, overall Time Complexity = 2n*O(1) = O(n).

CORRECTNESS PROOF:

We can traverse the complete circle starting from any index i ending at the previous index j.
Case 1: 1<=i<=j<=n
In this case any subarray will lie in (A[I], A[J]), I=i, J=j.

Case 2: 1<=j<i<=n
In this case any subarray will lie in (A[I], A[J]), I=i, J=j+n
Since, j<=n, J=j+n<=n+n=2*n

(i, j are indices of circular disc(D), while I, J are indices of array A)

//
To summarise, there would be 2 cases each having the starting index of array lying between 1 and n and the 1st case having an end before n (case 1) and another having end after n (case2). In both the cases the end would be before 2*n.
For example, in the given example, if the start is i, then the end can go upto index 8 (case1), or if it crosses 8, then it goes upto i-1 (case2) (beyond that sectors will repeat).
//

Hence, finding maximum subarray sum in A gives us maximum sum of contiguous sector collection in D

(Note that in A we can get a subarray whose size is more than n, in that case, we discard that subarray and start counting from that element again. This is handled by the counter c)

Now, we find MaxSum[i] as the maximum sum of contiguous subarray of A ending at index i.

We use the recursive relation MaxSum[i] = max(A[i], MaxSum[i-1]+A[i]).
We prove this by induction

Base Case: i=1
    => MaxSum[i]=A[i] (trivial)

Induction Hypothesis:
    We assume that MaxSum[k] is the correct maximum subarray sum ending at k for k<i

Induction Step :

We observe a basic fact that the maximum subarray sum upto i can be a subarray starting either from index i itself or from a previous index.

When it starts at the ith index itself,

MaxSum[i]=A[i] —(1)

Otherwise,
Let A[j],A[j+1],...,A[i] be any subarray ending at i

Then,

```
        A[j]+...+A[i-1] <= MaxSum[i-1]            (By Induction Hypothesis)
=>      A[j]+...+A[i] <= MaxSum[i-1]+A[i]
```

Hence, MaxSum[i-1]+A[i] is greater than or equal to any subarray ending at i
Therefore,

MaxSum[i] = MaxSum[i-1]+A[i] —(2)

Finally, MaxSum[i] would be the maximum of cases (1) and (2)

Hence, MaxSum[i] = max(MaxSum[i-1]+A[i], A[i])

Now, we have the maximum subarray sum ending at i $\forall$ 1<=i<=2*n

So, the maximum sum subarray of A would be the maximum of MaxSum[i] $\forall$ 1<=i<=2*n
(This is done by maintaining the Maxoverall variable)

With every MaxSum we also maintain the interval using currInterval
By updating the ending index of the interval if MaxSum[i] = MaxSum[i-1]+A[i] (Interval grows) and updating both the start and end index if MaxSum[i]=A[i] (Interval restarts).

And we update OverallMaxInterval if the overall sum upto now has the maximum MaxSum

(Note that to adjust from A to D, if the index is greater than n, we subtract n from it)

Q.2 Forex Trading

Let the given weighted graph be G = (V,E).
We want a cycle c(i)1,c(i)2,.....c(i)k,c(i)(k+1) with product R[i1, i2] · R[i2, i3] · · · R[ik−1, ik] · R[ik, i1] > 1.
Using the fact that x is strictly larger than 1 if and only if log(1/x) < 0, therefore we want log(1/R[i1, i2] · R[i2, i3] · · · R[ik−1, ik] · R[ik, i1]) <0.
=> - log (R[i1, i2] · R[i2, i3] · · · R[ik−1, ik] · R[ik, i1]) <0
=> Σ*j* (-(log(R[i(j), i(j+1)]))<0 (1<=j<=k)

Hence we want a negative weight cycle in the directed graph G' = (V, E') where E' = {e | e ∈ E, wt'(e) = -log(wt(e)) }.

Lemma1:
For any path in G', the minimum weight path connecting any 2 vertices (a,b) is the same as the largest path connecting those 2 vertices in G.

Proof:
Let there be a shortest path (e1,e2,e3,...ej) in G' with weight W. Then (wt'(e1) + wt'(e2) + …. wt'(ej) = W.
=> log(1/(wt(e1). wt(e2)....wt(ej)) = W (Assuming base of log is 10)
=> wt(e1)* wt(e2)* …. *wt(ej) = 10^ (-W) (where e1, e2, e3,,,,,ej are edges in G)
Since W is the minimum weight path between the 2 vertices a and b, then for every path with weight W' in G', W'>= W.
=> 10^(-W) >= 10^(-W') for all W weighted paths connecting a,b in G'. —(1)

Here 10^(-W') is the weight of path in G corresponding to path in G' having weight W' and 10^(-W) is the weight of path in G corresponding to path in G' having weight W.

Therefore the 10^(-W) weighted path in G is the largest weighted path in G. —(from (1))

Hence for any path in G', the minimum weight path connecting any 2 vertices (a,b) is the same as the largest path connecting those 2 vertices in G.

Therefore a positive weight cycle in G is a negative weight cycle in the graph G'.To check whether a negative weight cycle is present in the graph G', we apply Bellman- Ford Algorithm.

1 Maximum sum **18 / 18**

✓ **+ 18 pts** *Correct Solution*

And we update OverallMaxInterval if the overall sum upto now has the maximum MaxSum

(Note that to adjust from A to D, if the index is greater than n, we subtract n from it)

Q.2 Forex Trading

Let the given weighted graph be G = (V,E).
We want a cycle c(i)1,c(i)2,.....c(i)k,c(i)(k+1) with product R[i1, i2] · R[i2, i3] · · · R[ik−1, ik] · R[ik, i1] > 1.
Using the fact that x is strictly larger than 1 if and only if log(1/x) < 0, therefore we want log(1/R[i1, i2] · R[i2, i3] · · · R[ik−1, ik] · R[ik, i1]) <0.
=> - log (R[i1, i2] · R[i2, i3] · · · R[ik−1, ik] · R[ik, i1]) <0
=> Σ*j* (-(log(R[i(j), i(j+1)]))<0   (1<=j<=k)

Hence we want a negative weight cycle in the directed graph G' = (V, E') where E' = {e | e ∈ E, wt'(e) = -log(wt(e)) }.

Lemma1:
For any path in G', the minimum weight path connecting any 2 vertices (a,b) is the same as the largest path connecting those 2 vertices in G.

Proof:
Let there be a shortest path (e1,e2,e3,...ej) in G' with weight W. Then (wt'(e1) + wt'(e2) + …. wt'(ej) = W.
=> log(1/(wt(e1). wt(e2)....wt(ej)) = W   (Assuming base of log is 10)
=> wt(e1)* wt(e2)* …. *wt(ej) = 10^ (-W)   (where e1, e2, e3,,,,,ej are edges in G)
Since W is the minimum weight path between the 2 vertices a and b, then for every path with weight W' in G', W'>= W.
=> 10^(-W) >= 10^(-W') for all W weighted paths connecting a,b in G'.   —(1)

Here 10^(-W') is the weight of path in G corresponding to path in G' having weight W' and 10^(-W) is the weight of path in G corresponding to path in G' having weight W.

Therefore the 10^(-W) weighted path in G is the largest weighted path in G. —(from (1))

Hence for any path in G', the minimum weight path connecting any 2 vertices (a,b) is the same as the largest path connecting those 2 vertices in G.

Therefore a positive weight cycle in G is a negative weight cycle in the graph G'.To check whether a negative weight cycle is present in the graph G', we apply Bellman- Ford Algorithm.

```
For Each v ∈ V :
    D[v]= ∞ and parent[v]=null
D[s]=0
For i = 1 to n-1:
    For Each(x,y) ∈ E'
        If (D[ y] > D[x ] + wt'(x,y) ) then
            D[ y] = D[ x] + wt'(x,y)
            parent[ y] = x
If ( ∃  an edge (x,y) satisfying D[ y] > D[ x] + wt'(x,y) ) then
    Return "Negative-weight cycle found."
Return D, parent.
```

Proof that it correctly detects negative weight cycle:

If G' has no negative weight cycles, then Bellman-Ford correctly computes the minimum weight paths from the start vertex (let's say S) and for each edge (x,y), the following equality holds:
D[y] =D[x] +wt'(x,y)

We know that, without negative-weight cycles, shortest paths are always simple.
(A simple path is one in which no vertices are repeated)
If no vertices are repeated, then any path would have at most |V| vertices and hence, at most |V| -1 edges

We know that Bellman-Ford algorithm runs for |V|-1 iterations and after the ith iteration, all vertices upto level i satisfy D[y] <=D[x] +wt'(x,y) for edge (x,y) —(2)
(vertex v at level i means that shortest path from S to v has at most i edges)

Therefore, after the |V| -1 th iteration, all vertices upto level |V| -1 satisfy the inequality (2).
That is, all paths having at most |V| - 1 edges (which is equivalent to all simple paths) satisfy inequality (2)

If a negative weight cycle exists, it would have more than |V| -1 edges (since the path weight can reduce after moving along the cycle one more time), therefore it will not satisfy inequality (2).

Hence, Bellman-Ford correctly detects the presence of a negative weight cycle.

2. Print the negative weight cycle (if exists):

i) We perform the Bellman Ford Algorithm on G' and check If there exists any negative weight cycle.
ii) If there does not exists such a cycle then we print "No Negative Weight Cycle"
iii) If there does exists such a cycle then we take the vertex y which satisfies the inequality D[y] > D[x] + wt'(x,y).
iv) Now we find the ancestors of y in the array Parent which was calculated in our bellman ford algorithm and store them in array "cycle".

2.1 part 1 **10 / 10**

✓ **+ 6 pts** *Algorithm Correctness*

✓ **+ 3 pts** *Correctness Proof*

✓ **+ 1 pts** *Running Time Analysis Correctness*

```
For Each $v \in V$ :
    D[v]= $\infty$ and parent[v]=null
D[s]=0
For i = 1 to n-1:
    For Each(x,y) $\in$ E'
        If (D[ y] > D[x ] + wt'(x,y) ) then
            D[ y] = D[ x] + wt'(x,y)
            parent[ y] = x
If ( $\exists$  an edge (x,y) satisfying D[ y] > D[ x] + wt'(x,y) ) then
    Return "Negative-weight cycle found."
Return D, parent.
```

Proof that it correctly detects negative weight cycle:

If G' has no negative weight cycles, then Bellman-Ford correctly computes the minimum weight paths from the start vertex (let's say S) and for each edge (x,y), the following equality holds:
D[y] =D[x] +wt'(x,y)

We know that, without negative-weight cycles, shortest paths are always simple.
(A simple path is one in which no vertices are repeated)
If no vertices are repeated, then any path would have at most |V| vertices and hence, at most |V| -1 edges

We know that Bellman-Ford algorithm runs for |V|-1 iterations and after the ith iteration, all vertices upto level i satisfy D[y] <=D[x] +wt'(x,y) for edge (x,y) —(2)
(vertex v at level i means that shortest path from S to v has at most i edges)

Therefore, after the |V| -1 th iteration, all vertices upto level |V| -1 satisfy the inequality (2). That is, all paths having at most |V| - 1 edges (which is equivalent to all simple paths) satisfy inequality (2)

If a negative weight cycle exists, it would have more than |V| -1 edges (since the path weight can reduce after moving along the cycle one more time), therefore it will not satisfy inequality (2).

Hence, Bellman-Ford correctly detects the presence of a negative weight cycle.

2. Print the negative weight cycle (if exists):

i) We perform the Bellman Ford Algorithm on G' and check If there exists any negative weight cycle.
ii) If there does not exists such a cycle then we print "No Negative Weight Cycle"
iii) If there does exists such a cycle then we take the vertex y which satisfies the inequality D[y] > D[x] + wt'(x,y).
iv) Now we find the ancestors of y in the array Parent which was calculated in our bellman ford algorithm and store them in array "cycle".

v) We perform operation iv till we reach y again. After this the array cycle will contain the desired negative weight cycle.
vi) Then we print the array cycle.

<u>Correctness Proof:</u>

From the previous path, we have proved that if there exists a negative weight cycle then its vertices will not satisfy the inequality (2).
Therefore, y will be a vertex of a negative weight cycle.
Then, the parent x of y in the array parent will also be a part of the same cycle since, going from x to y in the cycle would reduce D[y].

<u>Time Complexity:</u>

Bellman-Ford algorithm takes O(mn) time. The size of the array parent will have at maximum n elements. Therefore, the total time to compute the cycle would be:

O(mn) + O(n) = O(mn)

And we know that in the worst case, m =O(n^2)

So, O(mn) <=O(n^3)

Q.3 Disjoint Collection of Paths

We solve this using Dynamic Programming and the sub-problems we consider are the subtrees of the node under investigation in the tree T.

Lemma: If both the end points of a path (a,b) where a and b are the starting and ending vertices of the path (since each path can be identified only by its ending and starting vertices) lie in the subtree of vertex X then all edges of the path (a,b) also lie in the subtree of x.

Proof:
We prove this by contradiction. If all edges don’t lie in the subtree of x, then Then (x,x’) will be repeated twice in the path, where x’ is the parent of X in T. (Since this edge is the only connection of the subtree to the rest of the tree). Then such a path will not exist (since we are only considering simple paths).
This is a contradiction

Let us assume that we have the solution sets for all the descendents of x and A, B be the optimal solution set of the left and right subtrees of any vertex x.

(Note : optimal solution of any vertex v is the optimal solution of the subtree rooted at v with the subset of paths of S such that all paths in that subset lie in the subtree of v)

2.2 part 2 **7 / 7**

- ✓ **+ 4 pts** *Algorithm Correctness*
- ✓ **+ 2 pts** *Algorithm Correctness Proof*
- ✓ **+ 1 pts** *Time Analysis*

v) We perform operation iv till we reach y again. After this the array cycle will contain the desired negative weight cycle.
vi) Then we print the array cycle.

<u>Correctness Proof:</u>

From the previous path, we have proved that if there exists a negative weight cycle then its vertices will not satisfy the inequality (2).
Therefore, y will be a vertex of a negative weight cycle.
Then, the parent x of y in the array parent will also be a part of the same cycle since, going from x to y in the cycle would reduce D[y].

<u>Time Complexity:</u>

Bellman-Ford algorithm takes O(mn) time. The size of the array parent will have at maximum n elements. Therefore, the total time to compute the cycle would be:

O(mn) + O(n) = O(mn)

And we know that in the worst case, m =O(n^2)

So, O(mn) <=O(n^3)

Q.3 Disjoint Collection of Paths

We solve this using Dynamic Programming and the sub-problems we consider are the subtrees of the node under investigation in the tree T.

Lemma: If both the end points of a path (a,b) where a and b are the starting and ending vertices of the path (since each path can be identified only by its ending and starting vertices) lie in the subtree of vertex X then all edges of the path (a,b) also lie in the subtree of x.

Proof:
We prove this by contradiction. If all edges don't lie in the subtree of x, then Then (x,x') will be repeated twice in the path, where x' is the parent of X in T. (Since this edge is the only connection of the subtree to the rest of the tree). Then such a path will not exist (since we are only considering simple paths).
This is a contradiction

Let us assume that we have the solution sets for all the descendents of x and A, B be the optimal solution set of the left and right subtrees of any vertex x.

(Note : optimal solution of any vertex v is the optimal solution of the subtree rooted at v with the subset of paths of S such that all paths in that subset lie in the subtree of v)

The optimal solution of x will have 5 disjoint cases:
i) it does not contain any path which passes through x, or
ii) it contains 1 path in which x is an internal vertex, or
iii) it contains only the path whose vertices are x, any vertex in the left subtree of x, or
iv) it contains only the path whose vertices are x, any vertex in the right subtree of x, or
v) it contains both paths of case iii, iv

Proof that there does not exist any other case:
Any other case will include either edge (x,a) or edge (x,b) or both where a,b are the children of x. then this case will have an overlapping path with either case ii, or case iii or case iv or case v

So, we take the case whose solution set has the maximum cardinality.

Case i: If there are no paths passing through x in the optimal solution, then all paths lie either in the left subtree of x or the right subtree of x (any other paths will lie outside the optimal solution of x)

Therefore, optimal solution set of x = A U B

(Since, A, B are the maximum sized disjoint sets of paths of the left and right subtree respectively)

Case ii: Let (g1,h1) and (g2,h2) be the be the endpoints of the path whose x is an internal node such that height(g1)<height(g2) and height(h1)>height(h2) Or vice-versa such that any other path with x as an internal node have both endpoints at height lower than that of (g1,h1) and (g2,h2) .

We use these because these will give us the maximum cardinality

Lemma 2: The cardinality of the optimal solution of any descendent of a vertex o is always less than that of the the optimal solution of O

Proof: Let d be a descendant of o.Then, the optimal solution of d will only contain the paths which completely lie in the subtree of d while the optimal solution of o will contain all the paths in the optimal solution of d along with paths which are in the subtree of o but not in the subtree of d.
Therefore, |optimal solution of o|>=|optimal solution of d|

Therefore, any other path with x as an internal node will have lower sum of cardinality of both its endpoints.

Let G1,G2,H1,H2 be the optimal solutions of g1,g2,h1,h2 respectively

Then optimal solution of X would be:

(g1,h1) U G1 U H1 if |G1|+|H1| > |G2|+|H2|
(g2,h2) U G2 U H2 otherwise

Case iii : Let (x,a1) be the path with a1 in the left subtree of x such that a1 has the maximum height of all such paths and A1 be the optimal solution of a1

(We take the node at maximum height since this would give us the maximum cardinality(Lemma 2))

Then, optimal solution of x would be : (x,a1) U A1 U B

Case iv: Let (x,b1) be the path with b1 in the right subtree of x such that b1 has the maximum height of all such paths and B1 be the optimal solution of b1

Then, optimal solution of x would be : (x,b1) U B1 U A

Case v: Let (x,a1) be the path with a1 in the left subtree of x such that a1 has the maximum height of all such paths and A1 be the optimal solution of a1 and let (x,b1) be the path with b1 in the right subtree of x such that b1 has the maximum height of all such paths and B1 be the optimal solution of b1

Then, optimal solution of x would be : (x,a1) U A1 U (x,b1) U B1

Therefore, the optimal solution of x is the case which has the maximum cardinality.

Hence, we would start growing the solutions from the leaves and move from the leaves to the root and keep storing them

<u>Time Complexity:</u>

For each node at height i, we look at all paths passing through that node.

At height i, the maximum number of paths in the subtree will be (2^i)C2 = O(4^i)
(C = number of combinations)

And we do this for heights 0 to h=log(n)

Therefore total steps = O(4^0)+O(4^1)+...+O(4^h) = O(4^h+1) = O(4*(2^h)^2) = O(n^2)

3 Disjoint Paths 12 / 25

✓ **+ 2 pts** *Identifying we need to use DP*

✓ **+ 0 pts** *Incorrect/Unattempted*

+ 10 *Point adjustment*

1. Not a complete binary tree, so height can be O(n)

2. Not clear why cant other cases be possible

3. Why?