CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 5
Submit via Gradescope

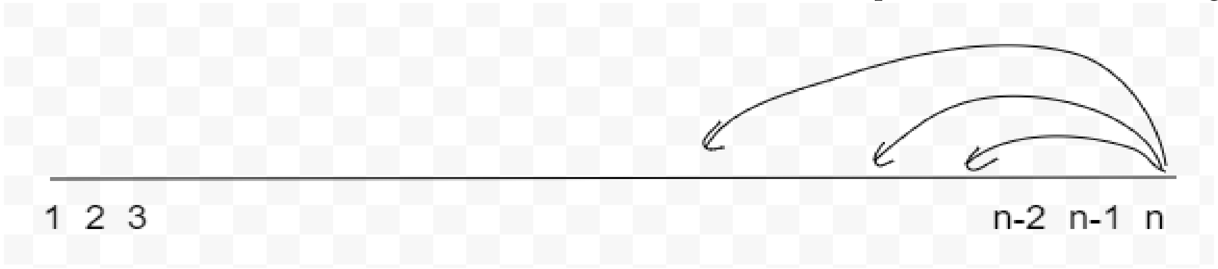
Name: Xuran Feng
Collaborators:

Instructions:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- Please review the grading policy outlined in the course information page.
- You must also write down with whom you worked on the assignment. If this changes from problem to problem, then you should write down this information separately with each problem.
- Problem numbers (like Exercise 3.1-1) are corresponding to CLRS 3^{rd} edition. While the 2^{nd} edition has similar problems with similar numbers, the actual exercises and their solutions are different, so make sure you are using the 3^{rd} edition.

1. ***(15 points)*** *Exercise 15.4-5.*

Solution:
Can use dynamic programming to solve the longest non-decreasing sequence of n numbers.
This problem has optimal subproblem solution property. For a given sequence of n numbers, if we can divide somewhere in the middle, the optimal solution of left subsequence and the optimal solution of right subsequence can be combined together to produce an optimal solution of the original sequence.
Let A[] be the array of given n numbers.
The recurrence equation is $C[n] = max_{k<=n-1, A[n]>=A[n-k]}\{1 + C[n-k]\}$, in English, C[n] represents the maximum length of non-decreasing sequence before nth (inclusive) number. If the nth number can form a longer non-decreasing subsequence with previous subsequence, then C[n] needs to increment by 1 based on the length of previous subsequence, nth number needs to iterate all previous subsequence to find the maximum one. At last, the maximum number of C[] needs to be picked as returned answer.
The visual of problem and subproblem is as below: there are arrows pointing from n leftwards until the leftmost end if A[n] satisfies >= A[n-k]. The bottom-up order is from left to right.



Pseudocode:

```
for i=1 to i=n:
    C[i]=1
    S[i]=0
for i=2 to i=n:
    best=0
    bestk=-1
    for k=1 to k=i-1:
        if(A[i]<A[i-k])
            skip
        if(1+C[i-k]>best)
            best=1+C[i-k]
            bestk=k
    C[i]=best
    S[i]=bestk
max=1
for i=2 to i=n:
    if(C[i]>C[max])
        max=i

Print(max):
    if(S[max]=0)
        return
    Print(S[max])
    output(A[max])
```
The total running time of this algo is $\theta(n^2)$ because there is a nested double loop and space complexity is $\theta(n)$.

2. ***(15 points)*** *Exercise 15.2-1.*

Solution:

According to the dynamic programming pseudocode in textbook, a m and a s table can be derived. The first table is a m table which tracks the minimum number of doing multiplications for $A_i...A_j$.

	1	2	3	4	5	6
6	2010	1950	1770	1860	1500	0
5	1155	2850	1080	3000	0	
4	405	330	180	0		
3	330	360	0			
2	150	0				
1	0					

$(A_1(A_2((A_3A_4)(A_5A_6)))) = 3\times12\times5+5\times50\times6+3\times5\times6+10\times3\times6+5\times10\times6 = 2250$
$(A_1(A_2(A_3(A_4(A_5A_6))))) = 5\times50\times6+12\times5\times6+3\times12\times6+10\times3\times6+5\times10\times6 = 2556$
$(((((A_1A_2)A_3)A_4)A_5)A_6) = 5\times10\times3+5\times3\times12+5\times12\times5+5\times5\times50+5\times50\times6 = 3380$
$((A_1((A_2A_3)(A_4A_5)))A_6) = 10\times3\times12+12\times5\times50+10\times12\times50+5\times10\times50+5\times50\times6 = 13360$
The second table s tracks the split point for a sequence of matrices.

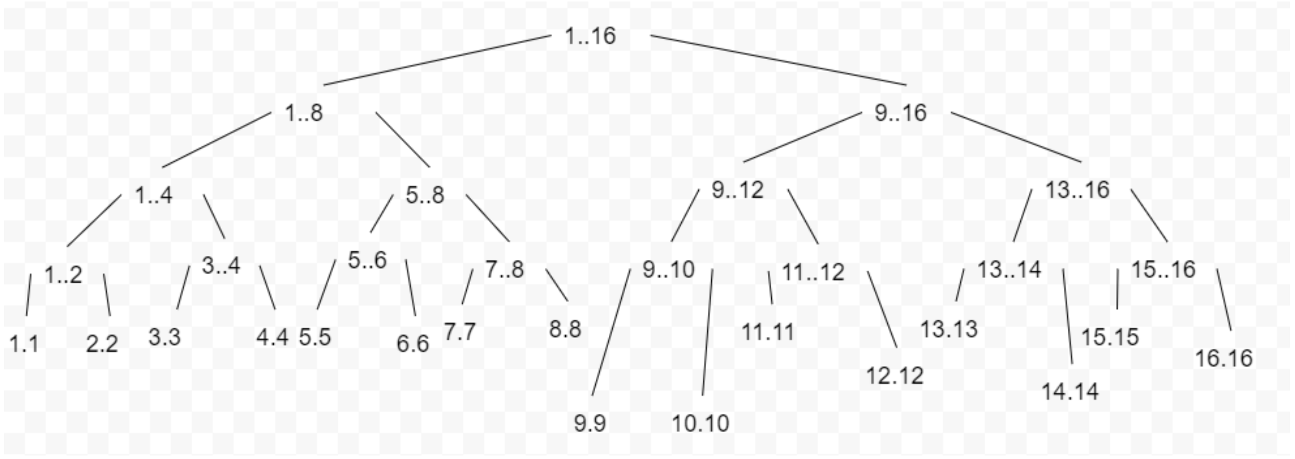
	1	2	3	4	5
6	2	2	4	4	5
5	4	2	4	4	
4	2	2	3		
3	2	2			
2	1				

The six matrices are A1: 5×10, A2: 10×3, A3: 3×12, A4: 12×5, A5: 5×50, A6: 50×6, according to s table (1,6) cell, the first split point is 2 meaning A1 and A2 are in a pair; the next subgroup is from A3 to A6, according to (3,6) cell, the second split point is 4 meaning A3 and A4 will be in a pair; the last subgroup is from A5 to A6, since this subproblem only contains 2 matrices, A5 and A6 will be in a pair. The final trace output is ((A1A2)((A3A4)(A5A6))) with 2010 scalar multiplications.

3. ***(15 points)*** *Exercise 15.3-2.*

Solution:

The recursion part of mergesort is drawn below. Memoization is used to speed up recursion problem that has a lot of overlapping subprolems. By recording answers of subproblems, a recursion tree of a repeating subproblem doesn't need to be gone through again and again, we can directly obtain answer from the memo table. Since mergesort algo doesn't have overlapping subproblem from the drawn recursion tree, each subproblem only hangs below one parent, so memoization is no use here to speed up running time.

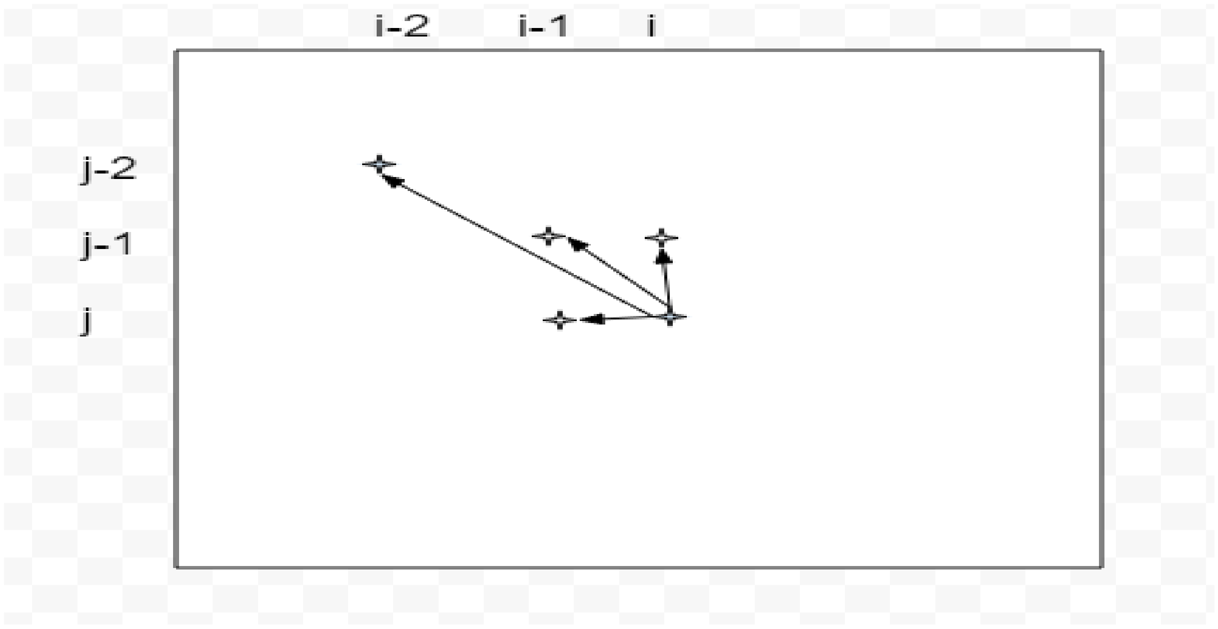


4. ***(20 points)*** *Exercise 15.3-5.*

Solution:
According to Section 15.1, the two pieces of rod-cutting problem after first cut are subproblems that should be solved independently, the overall solution is just to incorporate two optimal solutions of two subproblems. However, if we expose limits on the number of pieces of rod length, we lose the independence property between subproblems, when we want to solve the right subrod, we need to consider how the left subrod is cut to make sure we still have quota of some length to apply for right subrod. In other words, we cannot do divide and conquer strategy because now subproblems are strictly limited by each other, we need to solve the whole rod at one time.

5. ***(20 points)*** *Problem 15-5.*

Solution:
(a)The problem has optimal subproblem solution property. If we divide somewhere in the middle of sequence x and in the middle of sequence y, for example, there is an optimal operation set to transform the first i characters of x to the first j characters of y, there is also an optimal operation set to transform left characters of x to the left characters of y, combining the two subproblem optimal solutions together we can find a global solution of the original two sequences.
We need to reform the original problem. The original problem is to transform sequence x into sequence y, now the problem becomes to transform a subsequence of x to a subsequence y so the recurrence equation is $C[i,j] = min\{C[i-1,j-1]+cost(copy)$ if x[i]=y[j], $C[i-1,j-1]+cost(replace)$, $C[i-1,j]+cost(delete)$, $C[i,j-1]+cost(insert)$, $C[i-2,j-2]+cost(twiddle)$ if x[i]=y[j-1] and x[i-1]=y[j], $cost(kill)$ if j=0, $C[i,j]+cost(kill)$ if j=n and i>=n$\}$. C[i,j] represents the edit distance from x[1..i] to y[1..j] and it depends on subproblems of different operations. The visual can be drawn as below: the arrows are pointing from right bottom to upper left and the bottom-up order is from upper left to bottom right.



Pseudocode:

```
let C[][] be a 2D dimension array with (n+1)*(m+1) size.
let S[][] be a 2D dimension array with (n+1)*(m+1) size.
let O[][] be a 2D dimension array with (n+1)*(m+1) size storing operation name.
C[0,0]=0
for i=1 to i=m:
    C[i][0]=min{cost(kill),i*cost(delete)}
    S[i][0]=[i-1,0] if C[i][0] is cost(delete)
    S[i][0]=[0,0] if C[i][0] is cost(kill)
    O[i][0]=operation name of C[i][0]
for j=1 to j=n:
    C[0][j]=j*cost(insert)
    S[0][j]=[0,j-1]
    O[0][j]='insert'
for i=1 to i=m:
    for j=1 to j=n:
        copyCost=∞
        if(x[i]=y[j])
            copyCost=C[i-1,j-1]+cost(copy)
        replaceCost=C[i-1,j-1]+cost(replace)
        deleteCost=C[i-1,j]+cost(delete)
        insertCost=C[i,j-1]+cost(insert)
        twiddleCost=∞
        if(i> 1 and j>1 and x[i]=y[j-1] and x[i-1]=y[j])
            twiddleCost=C[i-2,j-2]+cost(twiddle)
        killCost=∞
        if(j=n and i>=n)
            killCost=C[i,j]+cost(kill)
        minCost=min{copyCost, replaceCost,deleteCost,insertCost,twiddleCost,killCost}
        minOperation=operation name of minCost
        C[i,j]=minCost
        S[i,j]=[i-1,j-1] if minCost is copyCost or replaceCost, [i-1,j] if minCost is deleteCost, [i,j-1] if minCost is insertCost...
        O[i,j]=minOperation
Print(m,n):
    if(m=0 and n=0)
        return
    Print(first element of S[m,n], second element of S[m,n])
    output(O(m,n))
```

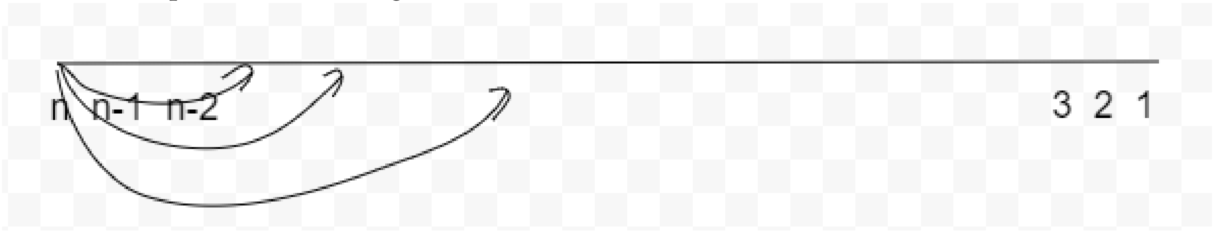
The total running time is $\theta(mn)$ because of the nested double for loop and the space requirement is $\theta(mn)$ because every intersection needs a cell to track.

(b) Because we are trying to align two DNA sequences, kill and twiddle are two operations that cannot be considered for they changing contents of DNA sequences and relative order of DNA sequences, thus will be always given ∞ values to make sure they won't appear as the final minCost choice. Because edit distance computes total positive cost between two sequences, we need to flip the score assigning rule, so when x'[j]=y'[j], the score is 1 so the cost of 'copy' is -1; when x'[j]$\neq$ y'[j], the score is -1 so the cost of 'replace' is 1; when either x'[j] or y'[j] is a space, the score is -2 so the cost will be 2, however, a character is not inserted or deleted as in part(a), instead, a space will be inserted or deleted to work as alignment, when a space is tried in x'[j], the operation is 'insert' because C[i,j-1] will be considered; when a space is tried in y'[j], the operation is 'delete' because C[i-1,j] will be considered. After confirming the cost of each operation, we plug into the part(a) to find the minimum cost of edit distance between two DNA sequences. Thus DNA alignment becomes an application of edit distance problem by assigning operations specific score number.

6. ***(30 points)*** *(Note: you should decide to use Greedy or DP on this problem) Prof. Curly is planning a cross-country road-trip from Boston to Seattle on Interstate 90, and he needs to rent a car. His first inclination was to call up the various car rental agencies to find the best price for renting a vehicle from Boston to Seattle, but he has learned,much to his dismay, that this may not be an optimal strategy. Due to the plethora of car rental agencies and the various price wars among them, it might actually be cheaper to rent one car from Boston to Cleveland with Hertz, followed by a second car from Cleveland to Chicago with Avis, and so on, than to rent any single car from Boston to Seattle.*
Prof. Curly is not opposed to stopping in a major city along Interstate 90 to change rental cars; however, he does not wish to backtrack, due to time contraints. (In other words, a trip from Boston to Chicago, Chicago to Cleveland, and Cleveland to Seattle is out of the question.) Prof. Curly has selected n major cities along Interstate 90 and ordered them from East to West, where City 1 is Boston and City n is Seattle. He has constructed a table T[i, j] which for all i < j contains the cost of the cheapest single rental car from City i to City j. Prof. Curly wants to travel as cheaply as possible.Devise an algorithm which solves this problem, argue that your algorithm is correct,and analyze its running time and space requirements. Your algorithm or algorithms should output both the total cost of the trip and the various cities at which rental cars must be dropped off and/or picked up.

Solution:
Should use dynamic programming to solve this problem because we cannot make decision before

looking at all subproblems. If we use greedy algorithm, we may choose the cheapest $/mile to begin with regardless of the second city we arrive from Boston. However, after arriving at the second city, we may find any rental car from this second city is very expensive regardless of the third city we choose to arrive, a better way is we should skip this second city at all. The greedy fails because now solution of this subproblem is not optimal.
The problem has optimal subproblem solution property. We can find some city in the middle of Interstate 90, the right trip before this city has an optimal solution and the left trip after this city can also has an optimal solution, combining two together, we can get a global optimal solution.
The recurrence equation is $C(n) = min_{k<=n-1}\{T(k,n)+C[n-k]\}$ where C[n] represents the minimum cost of arriving at city n. To obtain the minimum cost of arriving at city n, we need to iterate all cities before nth city. The cost from kth city to nth city is the minimum cost of arriving at city k plus any cost from k to n according to T table. The final minimum cost is simply C[n].
The visual of this problem is drawn below: there are arrows pointing from left to right since Boston is on the east while Seattle is on the west. There is an arrow between n and each previous city and the bottom-up order is from right to left.



Pseudocode:

```
for i=1 to i=n:
    C[i]=0
    S[i]=0
for i=2 to i=n:
    best=∞
    bestk=-1
    for k=1 to k=i-1:
        if(T(k,i)+C[i-k]<best)
            best=T(k,i)+C[i-k]
            bestk=k
    C[i]=best
    S[i]=bestk

Print(n):
    while(S[n]!=0)
        output S[n]
        n=S[n]
```

The total running time of this algo is $\theta(n^2)$ because there is a nested double loop and space complexity is $\theta(n)$.