Name: Chirag Telang

ID: 109410601

CSCI 3104, Algorithms **Due April 2, 2021**

Problem Set 9 (50 points) **Spring 2021, CU-Boulder**

Advice 1: For every problem in this class, you must justify your answer: show how you arrived at it and why it is correct. If there are assumptions you need to make along the way, state those clearly.

Advice 2: Verbal reasoning is typically insufficient for full credit. Instead, write a logical argument, in the style of a mathematical proof.

Instructions for submitting your solution:

- The solutions **should be typed** and we cannot accept hand-written solutions. Here's a short intro to Latex.
- You should submit your work through **Gradescope** only.
- The easiest way to access Gradescope is through our Canvas page. There is a Gradescope button in the left menu.
- Gradescope will only accept **.pdf** files.
- It is vital that you match each problem part with your work. Skip to 1:40 to just see the matching info.

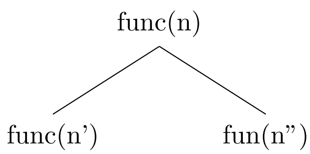
---

1. Recall that the fibonacci numbers form a sequence wherein each number is the sum of the previous two numbers in the sequence.

   (a) Give the recurrence relation for the definition of the fibonacci sequence.

   (b) Assume you have a program that implements the function $fib(n)$ to compute the nth fibonacci number with the recursive approach. Draw the recursion tree of function calls to compute $fib(4)$

   Tree Example of function $func$ using tikz:

   *Note if you want to make the tree with a different program, you can simply embed an image of it in the latex submission. A handwritten tree is also acceptable if it is extremely legible.

   (c) See the following function that uses a dynamic programming trick to implement $fib(n)$ with a faster runtime than the recursive one. Give the time complexity in terms of $\mathcal{O}()$ for the recursive implementation, as well as for Algorithm. 1 below. You do not need to write a proof. Explain why the dynamic programming algorithm is faster.

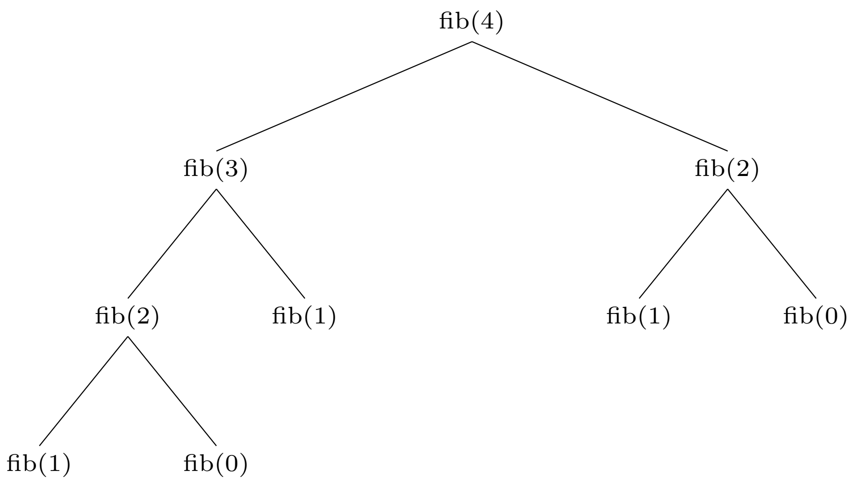
def *fib(n)***:**
  $memo \leftarrow [0, 1]$;
  for i *in* $2..n$ **do**
    $memo[i] \leftarrow memo[i-1] + memo[i-2]$;
  end
  return $memo[n]$

Algorithm 1: Dynamic fibonacci

Solution:

(a) The recurrence relation for the definition of the fibonacci sequence is $F_n = F_{n-1} + F_{n-2}$ – where the initial conditions are $F_0 = 1$ and $F_1 = 1$ and each number in the sequence is the sum of the previous two values.

(b)



(c) For the recursive implementation, the time complexity is $\mathcal{O}(2^n)$ – since the recurrence relation for this method is $T(n) = T(n-1) + T(n-2) + O(1)$. In regards to the dynamic fibonacci implementation, the time complexity is $\mathcal{O}(n)$ – since it only uses a single for loop. Here, the dynamic programming method is faster because its performance is linear, as compared to the recursive implementation, whose performance is exponential.

Name: Chirag Telang

ID: 109410601

CSCI 3104, Algorithms **Due April 2, 2021**

Problem Set 9 (50 points) **Spring 2021, CU-Boulder**

2. Consider the Knapsack problem for the list $A = [(4,3),(1,2),(3,1),(5,4),(6,3)]$ of (weight, value) pairs. The weight threshold is $W = 10$.

   (a) Fill in the table below using the bottom-up DP algorithm.

Weight	Value	$w=0$	1	2	3	4	5	6	7	8	9	10
-	-	0	0	0	0	0	0	0	0	0	0	0
4	3	0	0	0	0	3	3	3	3	3	3	3
1	2	0	2	2	2	3	5	5	5	5	5	5
3	1	0	2	2	2	3	5	5	5	6	6	6
5	4	0	2	2	2	3	5	6	6	6	7	9
6	3	0	2	2	2	3	5	6	6	6	7	9

   (b) Write an algorithm that prints the optimal subset of items once the bottom-up DP algorithm has finished. Your algorithm should only use the filled in table and the inputs to the bottom-up algorithm.

   (c) Highlight in red the numbers in each cell that your algorithm from part (b) visits. Circle each cell that is part of the optimal solution. (Indicate this on the same table from part (a).)

   (d) Does the order that we consider the items change the optimal solution? Explain why or why not.

Solution:

(a) Given above

(b) In the algorithm:
itemNum = the item number
currWeight = weight
tableInput(itemNum, currWeight) = returns the value that is inputted in the table
table = table that is being filled
wt(itemNum) = returns the weight of itemNum

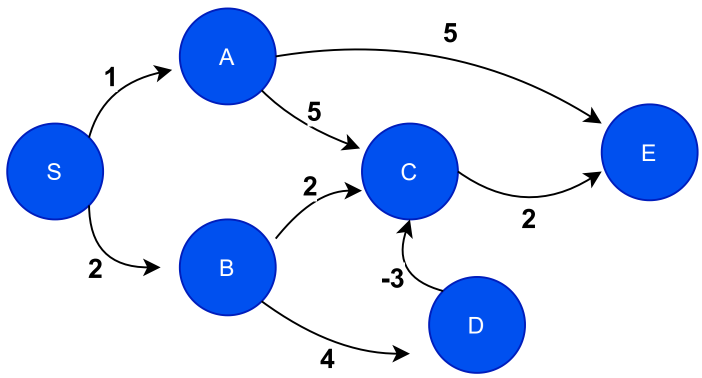
```
def Knapsack(itemNum, currWeight, table):
    if tableInput(itemNum, currWeight) < tableInput(itemNum - 1, currWeight):
        Knapsack(itemNum - 1, currWeight, table)

    if tableInput(itemNum, currWeight) > tableInput(itemNum - 1, currWeight):
        print(itemNum)
        Knapsack(itemNum - 1, wt(i), table)
        return
    else:
        return
```

(c) Given above

(d) No, the order that we consider the items in does not change the optimal solution. This is because dynamic programming starts with the smallest subset of subproblems – so regardless of if the order in which the elements appear changes, the value they return should remain the same.

3. Consider the directed graph $G = (V, E)$, pictured below. We define the minumum cost of a path from vertex u to vertex v to be the minumum of the edge weights along that path. For example, the minimum cost of the path from A to E is 5.



(a) Fill in the table below with the minimum cost to get from each node to every other node. Assume that paths start from the row node (*i.e.* cell (1,2) corresponds to the path starting at node S and ending at node A). If a path between two nodes does not exist, fill the cell with NA.

	S	A	B	C	D	E
S	0	1	2	3	6	5
A	NA	0	NA	5	NA	5
B	NA	NA	0	1	4	3
C	NA	NA	NA	0	NA	2
D	NA	NA	NA	-3	0	-1
E	NA	NA	NA	NA	NA	0

(b) Recall that the Bellman Ford algorithm can find the shortest paths of this graph by iteratively relaxing all edges. Given the order of edges below, show all of the updates that Bellman Ford would make to the cost of each vertex in the graph.
1. (S, A) 2. (S, B) 3. (A, C) 4. (B, C) 5. (A, E) 6. (B, D) 7. (D, C) 8. (C, E)

Fill in here:

- S: 0
- A: ∞, 1
- B: ∞, 2
- C: ∞, 6, 4, 3
- D: ∞, 6
- E: ∞, 6, 5

(c) Consider a cyclic graph (one in which there is a path from some node u that can return to u). Under what circumstances are we unable to define an exact shortest path between two nodes in this graph?

Solution:

(c) We are unable to define an exact shortest path between two nodes in a graph if the graph has a negative weight cycle – such that the sum of edges in that cycle results in a negative value.

4. Consider an algorithm for clustering words together that are likely to be simiilar. One metric for weighing the similarity of words is by their Minimum Edit Distance. Recall this algorithm from lecture, and assume that the operations are weighed as follows:

   - Insertion = 1
   - Deletion = 1
   - Substitution = 2

   (a) Fill in the below table with the edit distance of the two strings, and then specify the minimum edit distance between them.

	#	D	E	F	I	E	S
#	0	1	2	3	4	5	6
F	1	2	3	2	3	4	5
I	2	3	4	3	2	3	4
N	3	4	5	4	3	4	5
E	4	5	4	5	4	3	4

   (b) Assuming we weight edit operations with functions w_i, w_d, and w_s for *insertion* weight, *deletion* weight, and *substitution* weight, respectively, give the local recurrence of the minimum edit distance algorithm of the strings X and Y. You can ignore the cases of X_i, Y_j where i or j are 0.

   Hint: For any cell in the above table beyond the comparisons in the 1st row or column, give the equation that determines the value in the cell, in terms of the previous cells.

Solution:

(a) The minimum edit distance of the two strings is 4.

(b) Local recurrence where I_w= Insertion weight, D_w = Deletion weight, and S_w= Substitution weight.

$$\text{cost}(x, y, i, j) = \text{minimum}\begin{cases} \text{cost}(x, y, i-1, j-1) & \text{when } x_i = y_j, \\ \begin{cases} \text{cost}(x, y, i, j-1) + I_w \\ \text{cost}(x, y, i-1, j) + D_w \\ \text{cost}(x, y, i-1, j-1) + S_w \end{cases} & \text{when } x_i \neq y_j. \end{cases}$$