

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.

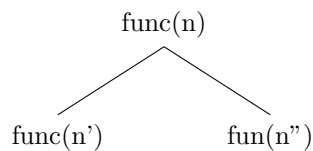
Instructions for submitting your solution:

- 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.
-

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

- Give the recurrence relation for the definition of the fibonacci sequence.
- Assume you have a program that implements the function $fib(n)$ to compute the n th 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.

- 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 ← [0, 1];
    for i in 2..n do
        | memo[i] ← memo[i - 1] + memo[i - 2];
    end
    return memo[n]
  
```

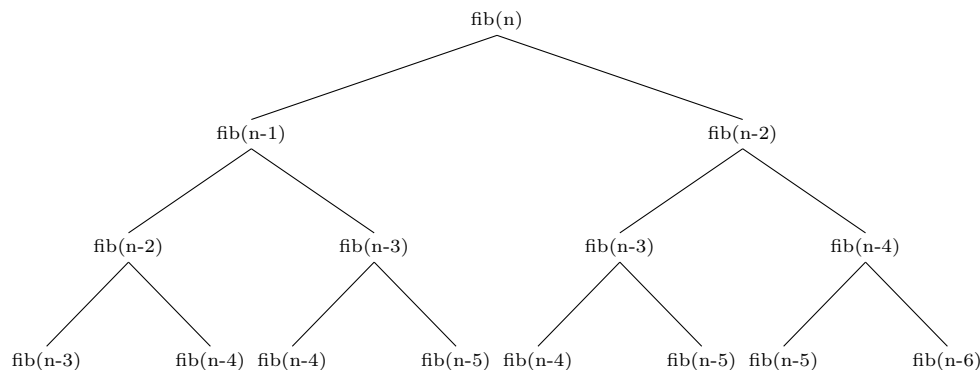
Algorithm 1: Dynamic fibonacci

Solution:

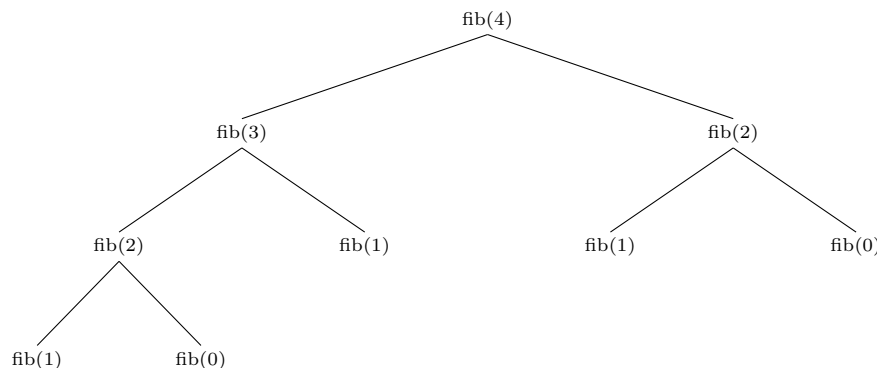
(a) Recurrence relation for the definition of the Fibonacci sequence:

$$T(n) = \begin{cases} T(n-1) + T(n-2), & n > 1 \\ 1, & n = 1 \\ 0, & n = 0 \end{cases}$$

(b) Recursion tree of function calls to compute $\text{fib}(n)$:



Recursion tree of function calls to compute $\text{fib}(4)$:




(c) The runtime is exponential for the recursive implementation, which is $\mathcal{O}(2^n)$. For the given function that uses dynamic programming, its time complexity is linear which is $\mathcal{O}(n)$. The dynamic programming version is much faster because it uses memoization which lowers a function's time cost in exchange for space cost. In other words, it stores the results of expensive function calls and returns the cached result when the same inputs occur again. This approach breaks down the problem into multiple subproblems. If the subproblem has already been solved it will reuse the answer produced. Else, it will solve the subproblem and store the result. This method avoids recomputing the subproblems, which can make the runtime much slower. In the recursive version, it will re-calculate the same problem that have already occurred, and this repetitiveness is why the runtime is exponential.

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	6	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.  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:

- (b)

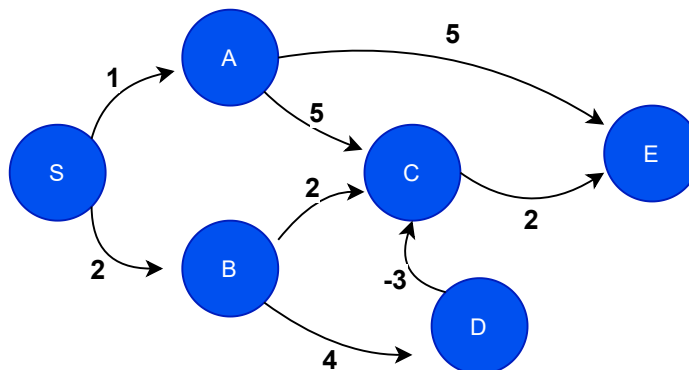
```

1 def printOpt(tableRes, A, W):
2     result = []
3     i = len(A)
4     j = W
5     while tableRes[i][j] > 0:
6         if(tableRes[i][j] == tableRes[i-1][j]):
7             continue
8         else: // cell above doesn't match current
9             add A[i-1] to result[] // weight-value pair part of soln
10            j = j - A.weight[i-1] // subtract weight to visit next cell column
11            i = i - 1 // go up a row
12    print result // print optimal subset from result
13    return result

```

- (d) The order that we consider the items does not change the optimal solution because order only affects how the table appears and their values should remain the same. The only thing that changes is the way we navigate the solution in the table, however the optimal solution is unchanged and is correct regardless of the order.

3. Consider the directed graph $G = (V, E)$, pictured below. We define the minimum cost of a path from vertex u to vertex v to be the minimum 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) If there is a cycle in the graph with a negative net weight < 0 , Bellman Ford is unable to define an exact shortest path between two nodes because there won't be a shortest path. This is because it will result in an infinite loop where it will iterate over and over again and the cycle will keep reducing the weight of the path.

4. Consider an algorithm for clustering words together that are likely to be similar. 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) *view table with edit distance of the two strings above*

Minimum edit distance = 4

- (b)

$$E(i, j) = \min \begin{cases} E(i, j-1) + w_i \\ E(i-1, j) + w_d \\ E(i-1, j-1) + w_s \end{cases}$$

Keep in mind that w_s depends on whether $X[i] = Y[j]$. If they equal, the substitution weight is 0. If they don't equal, the weight is 2.