COL351: Analysis and Design of Algorithms **Fall 2024**

Tutorial Sheet 8

Announced on: Sept 19 (Thurs)

- This tutorial sheet contains problems in dynamic programming (DP). When presenting your solutions, please define the DP table clearly. Don't forget to write the base case. The correctness and running time arguments can be brief (1-2 sentences).
- Problems marked with ($\star$) will not be asked in the tutorial quiz.

1. Suppose you own two stores, A and B. Each day, you can be either at A or B. If you are currently at store A (respectively, B) then moving to store B (respectively, A) the next day will cost C amount of money. For each day $i \in \{1, ..., n\}$, we are also given the profits $P_A(i)$ (respectively, $P_B(i)$) that you will make if you are at store A (respectively, B) on day i. Give a schedule which tells where you should be on each day so that the overall money earned, i.e., the profit minus the cost of moving between the stores, is maximized.

   Informal Idea: $dp_{i,j}$ is the maximum overall profit that can be collected in the first i days such that you end at store j on day i. Note that $1 \leqslant i \leqslant n$ and $j \in \{A, B\}$.
   Recurrence Relation:

   $$dp_{i,A} = P_A(i) + \max(dp_{i-1,A}, dp_{i-1,B} - C), \quad 2 \leqslant i \leqslant n$$
   $$dp_{i,B} = P_B(i) + \max(dp_{i-1,A} - C, dp_{i-1,B}), \quad 2 \leqslant i \leqslant n$$

   Base Case:

   $$dp_{1,A} = P_A(1)$$
   $$dp_{1,B} = P_B(1)$$

   While calculating these values of $dp_{i,j}$, we also store which store (either A or B) should be chosen on day $i-1$ in order to maximize the overall profit at day i. This can be done by tracking the decisions taken during the calculation, allowing us to reconstruct the optimal schedule afterward.

ALGORITHM 1: Optimal Schedule with maximizing overall profit

Input: Profits $P_A(i)$ and $P_B(i)$ for each day i, cost of switching C, number of days n

Output: Maximum total profit and optimal schedule

```
Let $dp_{1,A} = P_A(1)$ and $dp_{1,B} = P_B(1)$
Let $choice[1, A] = A$ and $choice[1, B] = B$
for $i = 2$ to $n$ do
    $dp_{i,A} = P_A(i) + \max(dp_{i-1,A}, dp_{i-1,B} - C)$
    if $dp_{i-1,A} \geqslant dp_{i-1,B} - C$ then
        $choice[i, A] = A$
    else
        $choice[i, A] = B$
    end
    $dp_{i,B} = P_B(i) + \max(dp_{i-1,A} - C, dp_{i-1,B})$
    if $dp_{i-1,B} \geqslant dp_{i-1,A} - C$ then
        $choice[i, B] = B$
    else
        $choice[i, B] = A$
    end
end
Let $final\_store = \arg\max(dp_{n,j}), j \in \{A, B\}$
Let $max\_profit = \max(dp_{n,A}, dp_{n,B})$
Let $schedule[n] = final\_store$
for $i = n - 1$ down to $1$ do
    $schedule[i] = choice[i + 1, schedule[i + 1]]$
end
return $max\_profit$ and $schedule$
```

Proof of Correctness:

The algorithm uses dynamic programming to solve this problem, and we will prove its correctness using **induction**.

Inductive Hypothesis:

$dp_{i,j}$ is the maximum overall profit upto day i if you are at store j on day i, where $1 \leqslant i \leqslant n$ and $j \in \{A, B\}$.

Base Case:

For $i = 1$, the solution is straightforward:

$$dp_{1,A} = P_A(1)$$
$$dp_{1,B} = P_B(1)$$

This case is trivial, the only possible profit on the first day is simply the profit earned by being at store A or store B.

Inductive Step:

For $dp_{i,A}$:

$$dp_{i,A} = P_A(i) + \max(dp_{i-1,A}, dp_{i-1,B} - C)$$

If you are at store A on day i, the profit for day i is $P_A(i)$, and the total profit is the sum of $P_A(i)$ and the maximum of:

- $dp_{i-1,A}$, which is the maximum profit up to day $i-1$ if you stayed at store A on day $i-1$,
- $dp_{i-1,B} - C$, which is the maximum profit up to day $i-1$ if you were at store B on day $i-1$ and incurred the switching cost C to move to store A on day i.

Thus, $dp_{i,A}$ correctly captures the maximum profit possible up to day i if you are at store A on day i.
Similarly we can argue for $dp_{i,B}$.
Time Complexity: It can be seen that the *for* loop runs $O(n)$ times, with each iteration taking constant time. Hence, the algorithm runs overall in $O(n)$ time.

- Written "I do not know how to approach this problem" - 0.6 points
-
  - Correct base case for the recurrence - 0.25 points
  - Correct recurrence relation - 1 point
  - Brief justification of the recurrence relation - 0.5 points
  - Mentioning the correct order of filling the DP table - 0.25 points
  - Outputting the optimal schedule (not just the optimal profit) - 0.5 points
  - Brief justification of the time complexity - 0.5 points

2. Given a tree T where the vertices have weights, an *independent set* is a subset of vertices such that there is no edge joining any two vertices in this set. Give an efficient algorithm to find an independent set of maximum total weight.

Informal Idea: Let $dp[v][0]$ the maximum total weight of the independent set of the subtree rooted at vertex v not including v in the set and $dp[v][1]$ the maximum total weight of the independent set of the subtree rooted at vertex v including v in the set. The recurrence relation is as follows:

$$dp[v][1] = w_v + \sum_{u \in \text{children}(v)} dp[u][0]$$
$$dp[v][0] = \sum_{u \in \text{children}(v)} \max(dp[u][0], dp[u][1])$$

The base case would be for leaves as follows:

$$dp[v][1] = weight[v]$$
$$dp[v][0] = 0$$

Thus the maximum total weight of an independent set would be $\max(dp[root][0], dp[root][1])$.

ALGORITHM 2: Maximum Weight Independent Set in a Tree

```
Input: A tree $T$ with $n$ vertices, each having a weight $w_v$ for vertex $v$
Output: Maximum total weight of an independent set
Let $dp[v][0]$ and $dp[v][1]$ be arrays of size $n$ initialized for each vertex $v$
Let $adj[v]$ be the adjacency list of vertex $v$
Let $visited[v]$ be a boolean array initialized to $false$
Function DFS($v$):
    $visited[v] = true$
    $dp[v][1] = weight[v]$
    $dp[v][0] = 0$
    for each neighbor $u$ of $v$ in $adj[v]$ do
        if $visited[u] = false$ then
            DFS($u$)
            $dp[v][1] = dp[v][1] + dp[u][0]$
            $dp[v][0] = dp[v][0] + \max(dp[u][0], dp[u][1])$
        end
    end
Let $root$ be any arbitrary vertex of tree $T$
DFS($root$)
return $\max(dp[root][0], dp[root][1])$
```

Try it yourself: The above algorithm only outputs the weight of the optimal set. To output the optimal set explicitly, we additionally store for each vertex v the children of v that are included in the optimal solution to the subproblem corresponding to $dp[v][i]$. Then the optimal set can be easily constructed using another DFS run.

Proof Of Correctness:
We will prove the correctness using **strong induction**.

Inductive Hypothesis: Given a subtree rooted at vertex v, the algorithm correctly computes $dp[u][0]$ and $dp[u][1]$ for all vertices u in subtree of v.

Base Case: For leaf nodes:

$$dp[v][1] = weight[v] \quad \text{(include the leaf)}$$
$$dp[v][0] = 0 \quad \text{(exclude the leaf)}$$

This is clearly correct since the maximum weight independent set can only consist of the leaf itself when included, and 0 when excluded.

Inductive Step: Assume the values of $dp[u][0]$ and $dp[u][1]$ are correct for all children u of v.

a) Including vertex v: If vertex v is included in the independent set, then none of its children can be included. Thus, the maximum weight is:

$$dp[v][1] = w_v + \sum_{u \in \text{children}(v)} dp[u][0]$$

This correctly adds the weight of v and the maximum weight of independent sets that exclude each child u, following our hypothesis.

b) Excluding vertex v: If vertex v is excluded, then we can either include or exclude each child u. Hence, the maximum weight is:

$$dp[v][0] = \sum_{u \in \text{children}(v)} \max(dp[u][0], dp[u][1])$$

This correctly calculates the maximum possible weight from either including or excluding each child based on the previously computed values, consistent with our hypothesis.

Time Complexity: $O(n)$ (DFS on tree) where n is the number of vertices in the tree.

3. (At most one of the two parts should be asked in the tutorial quiz.)

Recall the following problem from Tutorial Sheet 6:

You are given as input n jobs, each with a start time s_j and a finish time t_j. Two jobs *conflict* if they overlap in time—if one of them starts between the start and finish times of the other. The goal is to select a maximum-size subset of jobs that have no conflicts. (For example, given three jobs consuming the intervals $[0, 3]$, $[2, 5]$, and $[4, 7]$, the optimal solution consists of the first and third jobs.)

(a) Design an efficient dynamic programming algorithm for this problem.

(b) Consider a generalization of the above problem wherein job j has a non-negative weight w_j. Design an efficient dynamic programming algorithm to select a maximum weight subset of mutually non-conflicting jobs.

Informal Idea: We solve directly the (b) part, since part (a) can be solved by setting all weights to 1.
The goal is to find a maximum-weight subset of non-conflicting jobs, where each job j has a start time s_j, finish time t_j, and weight w_j. Two jobs conflict if their time intervals overlap.
We solve this using dynamic programming. The key idea is to process jobs in increasing order of their finish times, and for each job j, decide whether to include it in the solution or not. This decision is based on whether we include the job or take the optimal solution without it.
We define $dp[j]$ as the maximum weight subset of jobs from the first j jobs. The recurrence relation is:

$$dp[j] = \max(w_j + dp[p(j)], dp[j-1])$$

where $p(j)$ is the largest index of a job that finishes before job j starts. If job j is included, we add its weight and combine it with the optimal solution of non-conflicting jobs before $p(j)$. If not, we just take $dp[j-1]$.
(Bonus) We can make this efficient via binary search

ALGORITHM 3: Maximum Weight Subset of Non-Conflicting Jobs

```
Input: $n$ jobs, each with a start time $s_j$, finish time $t_j$, and weight $w_j$
Output: Maximum weight subset of mutually non-conflicting jobs
Sort the jobs by increasing order of finish times: $(s_1, t_1, w_1), (s_2, t_2, w_2), \ldots, (s_n, t_n, w_n)$
Let $dp[0] = 0$
for each $j$ from $1$ to $n$ do
    Find $p(j)$, the largest index $i < j$ such that $t_i \leqslant s_j$ (no conflict with job $j$)
    $dp[j] = \max(w_j + dp[p(j)], dp[j-1])$
end
return $dp[n]$
```

Try it yourself: The above algorithm only outputs the weight of the optimal set. To output the optimal set explicitly, we additionally store for each index j the last job (i.e., the job with the highest finish time) included in the optimal solution to the subproblem corresponding to $dp[j]$. Then the optimal set can be easily constructed using a single pass over the list of jobs.

Proof of Correctness
We prove correctness by *strong induction* on the number of jobs.
Base case: For $j = 0$, there are no jobs, so $dp[0] = 0$ is correct.

Inductive step: Assume that for all jobs up to $j-1$, the dynamic programming solution correctly computes the maximum weight subset of non-conflicting jobs. Now, consider job j:

- If we include job j, the total weight is $w_j + dp[p(j)]$, which is correct since the jobs before $p(j)$ are non-conflicting with job j.
- If we exclude job j, the optimal solution remains $dp[j-1]$, which by the inductive hypothesis is correct.

Thus, $dp[j] = \max(w_j + dp[p(j)], dp[j-1])$ correctly computes the maximum weight subset of non-conflicting jobs. By induction, the algorithm is correct for all jobs.

Time Complexity Analysis
The time complexity of the algorithm is determined by two main steps:

- **Sorting the jobs:** We first sort the jobs by their finish times, which takes $O(n \log n)$, where n is the number of jobs.
- **Computing the dynamic programming table:** For each job j, we compute $dp[j]$. This requires finding the index $p(j)$, which can be done using binary search in $O(\log n)$ if the jobs are sorted. Therefore, each state $dp[j]$ can be computed in $O(\log n)$ time.

Thus, the total time complexity of the algorithm is $O(n \log n)$ due to sorting the jobs and performing binary search for each job to compute the optimal solution.

4. You are given n boxes, where box i has height h_i, width w_i and length ℓ_i. Box i can be stacked on top of box j if $w_i < w_j$ and $\ell_i < \ell_j$. Give an algorithm for finding a stacking of a subset of boxes of maximum total height.

Informal Idea: We will first sort all the boxes in the non-increasing order of their width. Then for every box i we will maintain dp_i which denotes the maximum height of the stack that can be created in which box i is at the top.
The recurrence relation for dp_i is as follows:

$$dp_i = \max_{\substack{j<i \\ w_j<w_i,\ l_j<l_i}} (dp_j + h_i)$$

The maximum total height would be max over all dp_i.

ALGORITHM 4: Box Stacking Problem

Input: A list of n boxes where each box i has dimensions (h_i, w_i, ℓ_i)
Output: Maximum total height of a stack of boxes

```
Sort the boxes in non-increasing order of width $w_i$
for $i = 1$ to $n$ do
    $dp_i = h_i$
end
for $i = 2$ to $n$ do
    for $j = 1$ to $i - 1$ do
        if $w_j > w_i$ and $\ell_j > \ell_i$ then
            $dp_i = \max(dp_i, dp_j + h_i)$
        end
    end
end
return $\max(dp_i)$ for $i = 1$ to $n$
```

Try it yourself: The above algorithm only outputs the height of the optimal stacking. How can you output optimal stacking explicitly? (Hint: Use ideas similar to previous questions)

Proof Of Correctness:
We will prove the correctness using **strong induction**.

Inductive Hypothesis: dp_i is the maximum height of the stack in which box i is at the top.

Base Case: For $i = 1$

$$dp_1 = h_1$$

This case is trivial because we can only take the first box in the stack.

Inductive Step: Suppose $dp_1, dp_2, ...dp_{i-1}$ are all correct. Now at index i, one possible stack is just the box i itself, which has height h_i. The other kind of possibility is a stack that starts earlier and ends with box i at top, with the box prior to index i included being j. This can only occur if $w_j < w_i$ and $l_j < l_i$, and if so, its maximum stack height would be $dp_j + h_i$ because we added the box i on the top of maximum height stack ending at index j. The algorithm takes the maximum over these possibilities, so dp_i is correct.

Time Complexity: $O(nlogn)$(sorting) + $O(n^2)$(finding maximum stack height).

Try it yourself: Solve the above problem in $O(n \log n)$.
Hint 1: Can we modify the problem instance so that length and width always lie in the range $[1, n]$?
Hint 2: If Hint 1 is true, then we can store for each length j the maximum height which can be obtained if length j box is at the bottom. Can we then maintain these values efficiently using some data structure, like segment trees?

- Written "I do not know how to approach this problem" - 0.6 points
-
  - Sorting in non increasing order (or non decreasing order if ans is calculated for stacking big box on small box) on w_i or l_i or $w_i \times l_i$
  - Correct base case for the recurrence - 0.25 points
  - Correct recurrence relation - 1 point
  - Brief justification of the recurrence relation - 0.25 points
  - Mentioning the correct order of filling the DP table - 0.25 points
  - Outputting the optimal stacking (not just the optimal height) - 0.5 points
  - Brief justification of the time complexity - 0.5 points

5. Suppose you are taking n courses this semester. It is nearing the end of the semester and each course has a final project that still needs to be completed (started?). Each project will be graded on the following scale: It will be assigned an integer number on a scale of 1 to $g \geqslant 1$, higher numbers denoting better grades. Your goal, of course, is to maximize your average grade on the n projects. You have a total of $H > n$ hours in which to work on the n projects cumulatively, and you want to decide how to divide up this time. For simplicity, assume H is a positive integer, and you will spend an integer number of hours on each project. To figure out how best to divide up your time, you have come up with a set of functions $\{f_1, f_2, \ldots, f_n\}$ (rough estimates, of course) for each of your n courses, defined as follows: If you spend $h \leqslant H$ hours on the project for course i, you'll get a grade of $f_i(h)$. You may assume that the functions f_i are non-decreasing, i.e., if $h < h'$, then $f_i(h) \leqslant f_i(h')$.

   Given these functions $f_1, f_2, \ldots, f_n$, decide how many hours to spend on each project (in integer values only) so that your average grade, as computed according to f_i, is as large as possible. In order to be efficient, the running time of your algorithm should be polynomial in n, g, and H. None of these quantities should appear as an exponent in your running time.

High level idea: We maintain two 2-d arrays of dimensions $n \times H$.

- `max_grade`$[i][h]$ stores the maximum sum of grades we can achieve in the first i courses if we spend a total of h hours.
- `hours_spent`$[i][h]$ stores the number of hours spent on task i to achieve the max grade that is stored in `max_grade`$[i][h]$.

Algorithm:

ALGORITHM 5: Algorithm for calculating the dps

```
Input: The functions $f_1, f_2, \dots, f_n$
1 Assign max_grade$[i][h]$ := 0 for all $0 \leqslant i \leqslant n, 0 \leqslant h \leqslant H$
2 Assign hours_spent$[i][h]$ := 0 for all $1 \leqslant i \leqslant n, 0 \leqslant h \leqslant H$
3 for $i = 1$ to $n$ do
4     for $h = 0$ to $H$ do
5         for $h' = 0$ to $h$ do
6             grade_sum := $f_i(h')$ + max_grade$[i-1][h-h']$
7             ' if grade_sum > max_grade$[i][h]$ then
8                 max_grade$[i][h]$ := grade_sum
9                 hours_spent$[i][h]$ := $h'$
```

ALGORITHM 6: Algorithm for obtaining the answer

```
Input: The dp hours_spent
1 Assign Ans$[i]$ := 0 for all $1 \leqslant i \leqslant n$
2 Assign hours_left := $H$
3 for $i = n$ to 1 do
4     Ans$[i]$ := hours_spent$[i]$[hours_left]
5     hours_left := hours_left − Ans$[i]$
6 Return Ans
```

Proof of correctness: The Algorithm 5 maintains the following invariant:

$$\texttt{max\_grade}[i][h] = \max_{0 \leqslant h' \leqslant h} \left(f_i(h') + \texttt{max\_grade}[i-1][h-h']\right)$$

This means that we try spending h' hours on the i^{th} task and then optimally spend the remaining $h - h'$ hours on the remaining $i - 1$ tasks before i. Then we take the maximum value we can get over all values of h', which would give us the maximum sum of grades possible for the first i tasks utilizing h hours.
As f_i are all non-decreasing, there exists an optimal solution utilizing all H hours, so the optimal sum of grades that we can obtain is `max_grade`$[n][H]$. Hence when we calculate the answer using the `hours_spent` matrix we can start with `hours_spent`$[n][H]$.

Time Complexity: In the Algorithm 5, we have on loop of order n and then two nested loops bounded by order H. Hence the Algorithm 5 has complexity $O(nH^2)$. Then the Algorithm 6 takes just $O(n)$ time.

Hence, the total time taken is $O(nH^2)$, which is polynomial in n, g and H.

6. Recall the following problem from Tutorial Sheet 6:

   Imagine you have a set of n course assignments given to you today. For each assignment i, you know its deadline d_i and the time ℓ_i it takes to finish it. With so many assignments, it may not be possible to finish all of them on time. If you finish an assignment after its deadline, you get zero marks. Therefore, you must either complete the assignment by the deadline or not at all. How can you determine the maximum number of assignments you can complete within their deadlines?

   Design an efficient dynamic programming algorithm for this problem.

Assumption: We assume that the completed assignments only refers to the assignments completed withing their deadlines, i.e., an assignment with finish time larger than its deadline will be considered non-completed.
Also, by exchange argument, we can always ensure that there exists an optimal solution in which the completed assignments are scheduled in increasing order of their deadlines **(Try it Yourself)**. So, WLOG, we assume that the completed assignments are always scheduled in increasing order of their deadlines.

Informal Idea: We first sort the course assignments in increasing order of their deadlines. We maintain a dp table where $dp[i]$ stores a tuple $(c[i], t[i])$, where $c[i]$ represents *maximum number of assignments that can be completed from the first i assignments, if assignment i is included* and $t[i]$ represents *minimum total time required to complete* $c[i]$ *assignments from the first i assignments, if assignment i is included.* $dp[i]$ is updated using $dp[j]$ as follows, where j goes from 0 to $i-1$:

$$dp[i] = \begin{cases} (dp[j].c + 1, dp[j].t + \ell_i) & \text{if } dp[j].c + 1 > dp[i].c \text{ and } dp[j].t + \ell_i < d_i \\ (dp[i].c, dp[j].t + \ell_i) & \text{if } dp[j].c + 1 = dp[i].c \text{ and } dp[j].t + \ell_i < dp[i].t \end{cases}$$

Here, the first case updates $dp[i]$ if including assignment i leads to a greater number of assignments completed than the current maximum, while also ensuring that the total time taken does not exceed the deadline for assignment i. The second case updates dp_i to ensure that, when the count of assignments remains the same, we also track the minimum time required.

ALGORITHM 7: Maximum Number of Assignments within Deadlines

Input: A list of assignments with deadlines d_i and times ℓ_i

Output: Maximum number of assignments that can be completed within their deadlines

```
1  Sort assignments by deadlines $d_i$ and finish times $\ell_i$
2  Initialize $dp$ array where $dp[i] = (0, \infty)$ for all $i$
3  $dp[0] \leftarrow (0, 0)$
4  for $i \leftarrow 1$ to $n$ do
5      for $j \leftarrow 0$ to $i - 1$ do
6          if $dp[j].c + 1 > dp[i].c$ and $dp[j].t + \ell_i < d_i$ then
7              $dp[i] \leftarrow (dp[j].c + 1, dp[j].t + \ell_i)$
8          else if $dp[j].c + 1 = dp[i].c$ and $dp[j].t + \ell_i < dp[i].t$ then
9              $dp[i] \leftarrow (dp[i].c, dp[j].t + \ell_i)$
10 Return max$(dp[i].c)$ for all $i$
```

Try it yourself: The above algorithm only outputs the number of assignments in the optimal schedule. How can you output the optimal schedule explicitly? (Hint: Use ideas similar to previous questions)

Proof of Correctness:
We will prove the correctness using *strong induction*.

Inductive Hypothesis: $dp[i]$ maintains the maximum number of assignments that can be completed including assignment i using minimum total time.

Base Case: For the first assignment, $dp[0]$ is initialized to $(0, 0)$ since no assignments can be completed.

Inductive Step: Assume that the algorithm correctly calculates $dp[j]$ for all assignments $j < i$. When processing assignment i the algorithm check for each $j < i$ that if including assignment i in $dp[j]$ gives a better result than the current solution else it checks whether including i does not increase the count of completed assignments but reduces the time required, it updates $dp[i]$ accordingly. Thus, finally in $dp[i]$ we have the maximum number of assignments that can be completed including assignment i using minimum total time.

Time Complexity: Since for calculating every $dp[i]$ value we need to iterate over all $j < i$. Therefore, the entire algorithm runs in $O(n^2)$.

Try it Yourself: Can you create the dp table in $O(n \log n)$ time?

7. Given integers n and k, along with $p_1, \ldots, p_n \in [0, 1]$, you want to determine the probability of obtaining exactly k heads when n biased coins are tossed independently at random, where p_i is the probability that the i^{th} coin comes up heads. Give an $\mathcal{O}(n^2)$ algorithm for this task. Assume you can multiply and add two numbers in $[0, 1]$ in $\mathcal{O}(1)$ time.

Informal Idea: Suppose the n^{th} coin turns up heads, then we need to find the probability of obtaining $k-1$ heads from the first $n-1$ coins. If n^{th} coin turns up tails, then we find the probability of obtaining k heads from first $n-1$ coins. We use DP to find these probabilities efficiently.

Algorithm: Let P be a 2D array, whose entries are set in the following manner (intuitively, $P[i][j]$ will denote the probability of obtaining exactly $j \in [0,k]$ heads when the first $i \in [0,n]$ biased coins are tossed independently):

- Base Cases:
  - When $i = 0$ and $j > 0$: No coins are tossed, making the probability of getting at least one head to be 0 i.e. $P[0][j] = 0 \; \forall \; j \in [1,k]$.
  - When $i = 0$ and $j = 0$: Probability of getting 0 heads when no coins are tossed is 1 i.e. $P[0][0] = 1$.
  - When $i < j$: When less coins are tossed compared to the number of heads required, results in 0 probability i.e. $P[i][j] = 0 \; \forall \; i < j$
- Recursive Step: When $i > 0$ and $j > 0$:

  There are two possibilities in the i^{th} toss:

  a) Heads comes with probability p_i: We require exactly $j-1$ heads in the previous $i-1$ tosses.

  b) Tails comes with probability $1-p_i$: We require exactly j heads in the previous $i-1$ tosses.

$$P[i][j] = \begin{cases} p_i * P[i-1][j-1] + (1-p_i) * P[i-1][j] & \text{if } j > 0, \\ (1-p_i) * P[i-1][j] & \text{if } j = 0. \end{cases} \quad (1)$$

  It can be seen that the table P can be filled in row-wise manner, so that entries of $(i-1)^{th}$ row are available while calculating the entries for i^{th} row.

Time Complexity: Since there is an array of size $(n+1) \times (k+1)$ and each location assignment occurs only once, there are $O(nk)$ assignments. Each location assignment runs in constant time (it is essentially a lookup). Thus, the entire algorithm runs in $O(nk)$ time.

- Correct base case for the recurrence - 0.5 points
- Correct recurrence relation - 1 point
- Brief justification of the recurrence relation - 0.5 points
- Mentioning the correct order of filling the DP table - 0.5 points

- Brief justification of the time complexity - 0.5 points

8. ($\star$) A subsequence is *palindromic* if it is the same whether read left to right or right to left. For instance, the sequence

$$A, C, G, T, G, T, C, A, A, A, A, T, C, G$$

has many palindromic subsequences, including A, C, G, C, A and A, A, A, A (on the other hand, the subsequence A, C, T is not palindromic). Devise an algorithm that takes a sequence $x[1 \ldots n]$ and returns the longest palindromic subsequence as well as its length. Its running time should be $\mathcal{O}(n^2)$.

ALGORITHM 8: Longest Palindromic Subsequence

```
Input: A list $x[1, ..., n]$ of characters.
Output: The longest palindrome subsequence of $x$.
Let $P$ be a $n \times n$ matrix such that the $P[i][j]$ represents the longest palindromic
  subsequence in the sequence $x[i], x[i+1], \ldots, x[j]$
Base condition 1: $P(i, j)$ is empty string whenever $i > j$
Base condition 2: $P(i, i)$ is $x_i \forall i \in [1, n]$
for $length \leftarrow 2$ to $n$ do
    for $start \leftarrow 1$ to $n - length + 1$ do
        $end \leftarrow start + length - 1$
        if $x_{start} == x_{end}$ then
            $P(start, end) \leftarrow x_{start} + P(start+1, end-1) + x_{end}$
        else
            if $P(start, end-1).length > P(start+1, end).length$ then
                $P(start, end) \leftarrow P(start, end-1)$
            else
                $P(start, end) \leftarrow P(start+1, end)$
return $P(1, n)$
```

Proof of Correctness: We need to prove $|OPT(i,j)| = |P(i,j)|$ where $|OPT(i,j)|$ refers to the longest palindrome subsequence in the sequence $x[i], x[i+1], \ldots, x[j]$. It is sufficient to prove $|OPT(i,j)| \geqslant |P(i,j)|$ and $|OPT(i,j)| \leqslant |P(i,j)|$.
To show $|OPT(i,j)| \geqslant |P(i,j)|$, we use the fact that palindrome subsequence $|P(i,j)|$ is feasible solution. Thus, it is clear that $OPT(i,j)| \geqslant |P(i,j)|$.

To prove that $|OPT(i,j)| \leqslant |P(i,j)|$, we perform an induction on the length of the sequence, i.e., an induction on $j - i + 1$.
Base Case ($j = i + 1$): The longest palindrome is of length 1 or 2 depending on whether $x_i = x_j$. This implies that $|OPT(i,j)| \leqslant max\{P(i, j-1), P(i+1, j), 2 + P(i+1, j-1)\} = |P(i,j)|$.

Induction Hypothesis: $|OPT(i,j)| \leqslant |P(i,j)| \; \forall \; i, j$ s.t $j - i + 1 = k$

Consider the solution $OPT(i,j)(\forall\ i,j \text{ s.t } j-i+1 = k+1)$. We have the following cases:

- $x_i \in OPT(i,j) \wedge x_j \notin OPT(i,j)$: We can conclude that $OPT(i,j) = OPT(i,j-1)$.
- $x_i \notin OPT(i,j) \wedge x_j \in OPT(i,j)$: We can conclude that $OPT(i,j) = OPT(i+1,j)$.
- $x_i \in OPT(i,j) \wedge x_j \in OPT(i,j)$: We can conclude that

$$OPT(i,j) = x_i.OPT(i+1,j-1).x_j$$

Now we have,

$$|OPT(i,j)| \leqslant max\{|OPT(i,j-1)|, |OPT(i+1,j)|, 2+|OPT(i+1,j-1)|\}$$

By induction hypothesis we have $|OPT(i,j)| \leqslant |P(i,j)|$.

Time Complexity: Since there is an array of size $n \times n$ and each location assignment occurs only once, there are $O(n^2)$ assignments. Each location assignment runs in constant time (it is essentially a lookup). The return is merely a table lookup, so it runs in constant time. Thus, the entire algorithm runs in $O(n^2)$ time.