Problem Set 3

Graded

Student

Anton Melnychuk

Total Points

96 / 100 pts

Question 1

Q0: Your Information

1 / 1 pt

✔ **– 0 pts** No parts missing

Question 2

Q1: Treasure Hunter

18 / 19 pts

2.1 **Q1.1: Path**

8 / 9 pts

✔ **+ 9 pts** Correct

– 1 pt Point adjustment

1 dp[1] might not equal ruins[1]. If ruins[0] is bigger, it should equal that.

2.2 **Q1.2: Cycle**

10 / 10 pts

Reused Part 1 algorithm correctly

✔ **+ 3 pts** Correct algorithm (assuming Part 1 is correct)

✔ **+ 4 pts** Proof of correctness

✔ **+ 3 pts** Runtime analysis

Question 3

Q2: Monge Arrays **20** / 20 pts

3.1 **Q2.1: non-decreasing** **8** / 8 pts

✔ **+ 8 pts** Correct proof

2 Great case work

3.2 **Q2.2: algorithm** **12** / 12 pts

Algorithm Description

✔ **+ 4 pts** Correct Algorithm

Proof of correctness

✔ **+ 4 pts** Proof of correctness

Runtime analysis

✔ **+ 2 pts** Correct recursive relation

✔ **+ 2 pts** Correct solution and proof of recursive relation

Question 4

Q3: Maximum Self-Match

Resolved **17** / 20 pts

- ✔ **+ 7 pts** Correct algorithm
- ✔ **+ 10 pts** Proof of correctness
- ✔ **+ 3 pts** Running time analysis
- ✔ **− 2 pts** Insufficient runtime analysis
- **− 1 pt** Algorithm: We do not want to include the x^0 coefficient, so your max computation should not include this power. Runtime: Building the polynomials takes $O(kn)$ time, summing the polynomials takes $O(kn)$ time, and you do not account for the contribution of computing the maximum.

Regrade Request **Submitted on: Nov 10**

Hi, I would be extremely grateful for your time to overlook this problem. I apologize for any confusion I may have caused with it; I'm having difficulty understanding why the summation/multiplication of the polynomial takes O(kn) with the FFT Algorithm for each letter (k). Could you by chance provide more clarification on this, please? I indeed overlooked finding the maximum value. Regarding x^0, I included it for the sake of simplifying the solution, as I thought its presence did not impact the time complexity. Let me know if I am wrong. Thank you so much, Best regards, Anton. Have a nice day!

There are k polynomials, each of which contains $O(n)$ terms, thus summing the k polynomials is equivalent to adding $k\cdot O(n) = O(nk)$ terms, thus the summation takes $O(kn)$ time.

Reviewed on: Nov 13

Question 5

Q4: 1-D k-means **20** / 20 pts

5.1 Q4.1: Cost **2** / 2 pts

- ✔ **− 0 pts** Correct

5.2 Q4.2: Recursion **4** / 4 pts

- ✔ **− 0 pts** Correct

5.3 A4.3 AUnit **6** / 6 pts

- ✔ **− 0 pts** Correct

5.4 A4.4: Dynamic programming **8** / 8 pts

- ✔ **− 0 pts** Correct

Question 6

Maximum-Cost Independent Sets on Trees **20** / 20 pts

6.1 **Q5.1: Path** **6** / 6 pts

- ✔ **+ 2 pts** Recursive relation
- ✔ **+ 1 pt** Proof of recursive relation
- ✔ **+ 2 pts** Algorithm (which includes the order in which the dynamic program's states are computed)
- ✔ **+ 1 pt** Running time analysis

6.2 **Q5.2: Tree O(n^2)** **6** / 6 pts

- ✔ **+ 2 pts** Recursive relation
- ✔ **+ 1 pt** Proof of recursive relation
- ✔ **+ 2 pts** Algorithm (which includes the order in which the dynamic program's states are computed)
- ✔ **+ 1 pt** Running time analysis

6.3 **Q5.3: Tree O(n)** **8** / 8 pts

- ✔ **+ 4 pts** Correct algorithm
- ✔ **+ 4 pts** Running time analysis

Question assigned to the following page: 1

CPSC/ECON 365: Algorithms Yale University

Solution to Problem Set 3

Student Name: Anton Melnychuk Due: Friday, October 13, 2023 at 2:30 pm ET

1 Your Information

(a) Your name.

Anton Melnychuk

(b) Your SID.

923819040

(c) A list your collaborators and any outside resources you consulted for this problem set.

ULAs: Caleb, Anuj, Lucas, Dylan, Ryan, Christian

(d) Copy: "I have followed the academic integrity and collaboration policy as written above."

I have followed the academic integrity and collaboration policy as written above

(e) How many hours did you spend in this problem set?

35 hours

(f) Did you complete the **anonymous September Feedback Form**? It is available here:

https://forms.gle/xTDzrV1p68Rb8p1o6

Please certify your completion with a screenshot of the submitted screen and a copy of the following statement "I completed the feedback form thoroughly and completely."

I completed the feedback form thoroughly and completely.

Question assigned to the following page: 1

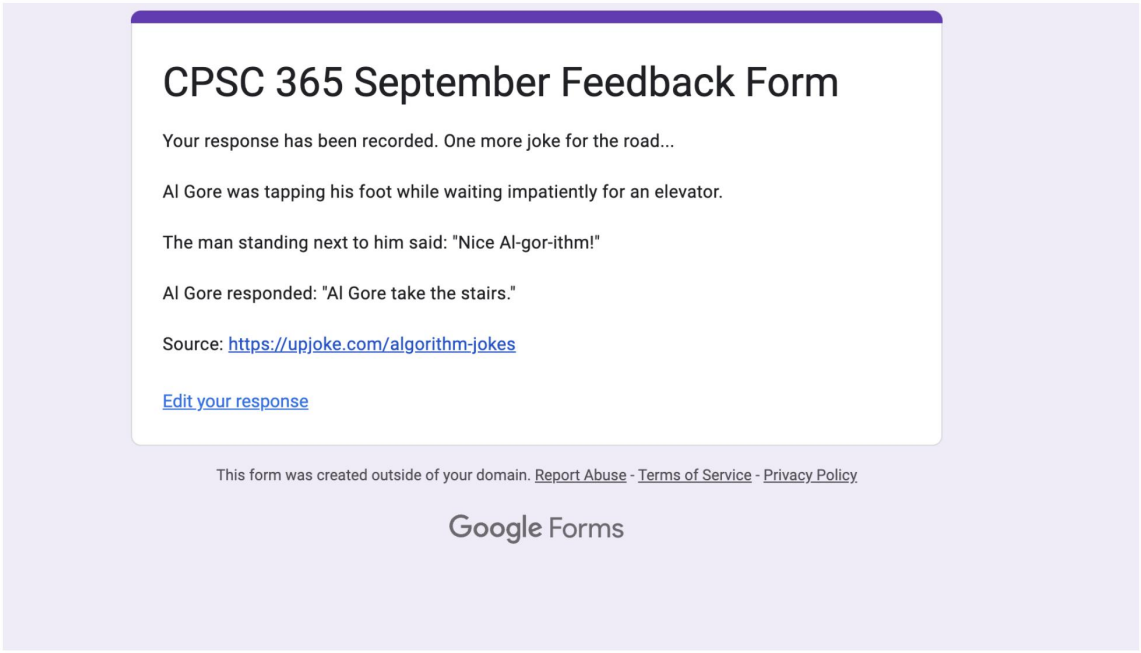

Figure 1: Anonymous September Feedback Form

No questions assigned to the following page.

2 Treasure Hunter

You are a professional treasure hunter exploring a mysterious island filled with ancient ruins (you can safely assume ruins always exist). Each ruin contains a treasure chest with a certain amount of gold coins. However, the ruins are protected by an ancient curse: *if you loot two adjacent ruins, a trap will be triggered, and you'll be caught.*

1. In the first part, assume the ruins are arranged in a path. Design an algorithm that, given an integer array `coins` denoting the number of gold coins in each ruin, returns the maximum amount of gold you can collect tonight without triggering any traps. To get full credits, your algorithm should run in $O(n)$ time.

The next night, you go to another mysterious island filled with ancient ruins. The difference is that on this island all ruins are arranged in a cycle. (Where the first ruin is the neighbor of the last ruin.) The same rule holds: *if you loot two adjacent ruins, a trap will be triggered, and you'll be caught.*

2. Design an algorithm that, given an integer array `coins` containing the number of gold coins in each ruin, returns the maximum amount of gold you can collect tonight without triggering any traps. To get full credits, your algorithm should run in $O(n)$ time. (*Hint:* You can use the algorithm from part 1 as a subroutine.)

In each part, justify the correctness of your algorithm and analyze its running time.

Question assigned to the following page: 2.1

Solution.

Part #1. Algorithm

```
getMaxCoinsLine(ruins):
    # how many ruines
    n <- amounf of ruins (size of an array)

    if (n == 1):
         # how many ruins are in the first ruin
        return ruins[0]
    else if (n == 2):
        # b/c adjacent cannot be chosen, pursue one with max coins
        return max(ruins[0], ruins[1])

    # define array size of n, zero in each
    #define vector/arrar called dp, size of n, all element set to 0
    vector <int> dp(n, 0)

    # base cases (save amount of coins in the first and second ruins)
    dp[0] = ruins[0]
    dp[1] = ruins[1]

    # run though each ruin and
    for (int i = 2; i < n; i++) {
        # pursue the path with greatest amount of coins
        dp[i] = max(dp[i-1], dp[i-2] + ruins[i])
    }

    # return the best path
    return dp[n-1]
```

Time Complexity

The algorithm runs through each ruin in order of their position exactly once, which contributes to a linear time complexity of $O(n)$, where n is the number of ruins in the island. The insertion of values into the *dp* array is also an $O(1)$ operation on average, and the computations involving maximizing coins is a constant-time operation. Therefore, the overall time complexity of this algorithm is $O(n)$.

Question assigned to the following page: 2.1

Proof By Strong Induction

Notations:

Let n $\in \mathbb{N}$ be the amount of *ruin* (notation) given by the problem. By definition, we know that n ≥ 1. Similarly, let *dp* (notation) be an dependency array, size of n, to hold the greatest amounts of coins can be looted till some k $\in \{1, 2, ..., n\}$, index of a ruin.

Base Case:

if n = 1 $\implies$
return ruin[0] coins, because no other choice exists.
The algorithms works. ✓

if n = 2 $\implies$
return max(ruin[1], ruin[2]) coins, because no adjacent ruins can be selected (by definition) and only one, with the greatest amount of coins, can be a solution to this case.
The algorithms works. ✓

Induction Hypothesis:
Let dp[0] = ruin[0] and dp[1] = ruin[1]. Now, assume that for some arbitrary ruin index k $\leq$ n, for any ruin m: k $\geq$ m $\geq$ 2, my algorithm correctly identifies the path with greatest amount of coins in total. [Want to show that my algorithm will also correctly predict the optimal route for k+1 ruin]

Induction Step: By definition two adjacent ruins cannot be looted, which implies that every time we consider a new induction step, we have two distinct cases:

The hunter either pursues the leftwards ruin path (in coins) to keep the greatest benefit from there Or the ruin path (in coins) on the right side and get extra, currently located-at, ruin[k+1] treasure.

To maximize the total amount of output we take max of these values and iterate the process over. Thus, we know that for any k $\leq$ n, the algorithm works as well. ✓

Question assigned to the following page: 2.2

Part #2. Algorithm:

```
# This algorithm uses the function
# from the previous problem and
# assumes that ruins are enumerated

getMaxCoinsCycle(ruins):
    # how many ruines exist
    n <- length(ruins)

    # same base cases
    if (n == 1):
        return ruins[0]
    else if (n == 2):
        return max(ruins[0], ruins[1])

    # define array dp size of n, zero coins in each
    vector <int> dp(n, 0)

    # save the first and second ruins treasure amount
    dp[0] = ruins[0]
    dp[1] = ruins[1]

    # consider cycle as two lines with either left ruine started or right
    # apply similar problem from part a in these two subcases
    int maxWithoutFirst = getMaxCoinsLine (vector<int>(ruins.begin(), ruins.end() - 1))
    int maxWithoutLast = getMaxCoinsLine (vector<int> (ruins.begin() + 1, ruins.end()))

    # return one with greatest amount of coines
    return max(maxWithoutFirst, maxWithoutLast)
```

Time Complexity

After considering two cases of the cycle as a line, the algorithm runs through each ruin in order of their position two times, which contributes to a linear time complexity of $O(2n)$, where n is the number of ruins in the island. The insertion of values into the *dp* array is also an $O(1)$, and the computations involving maximizing rewards is a constant-time operation. Therefore, the overall time complexity of this algorithm is $O(n)$.

Question assigned to the following page: 2.2

Direct Proof

Notes:

Working with a cycle, we might encounter the case where two ruins on the beginning and the end of induction (till not repeated) are adjacent to each other. To ovoid this, we can consider the fact that doing induction in a cycle we will also have either selected left or right element (by the definition of adjacent ruins.), so let's denote the following cycle as two separate cases: in first case eliminate the last ruin (before finding one that is visited), range [1, n-1]; in the other eliminate the first one (after starting with some arbitrary ruin), range [2, n].

Base Case:

if n = 1 $\implies$
return ruin[0] coins, because no other choice exists.
The algorithms works. ✓

if n = 2 $\implies$
return max(ruin[1], ruin[2]) coins, because no adjacent ruins can be selected (by definition) and only one, with the greatest amount of coins, can be a solution to this case.
The algorithms works. ✓

Induction Hypothesis:
Using the function *getMaxCoinsLine(ruins)* from the previous part of the question, let's brute-force denoted subsequences of the cycle represented as a set of two lines:

1. Range [1, n-1]:
getMaxCoinsLine[array(ruins.begin(), ruins.end() - 1)]

2. Range [2, n]:
getMaxCoinsLine[array(ruins.begin() + 1, ruins.end())]

Considering that we have found all possible ways the maximum-coins path could be built in the cycle, let's manually compare the final two outputs from the *getMaxCoinsLine()* functions and return the greatest one to get the maximum amount in total. ✓

No questions assigned to the following page.

3 Monge Arrays

Let M be an $m \times n$ matrix such that for any indices i, j, k, ℓ with $1 \leq i < k \leq m$ and $1 \leq j < \ell \leq n$ we have

$$M_{ij} + M_{k\ell} \leq M_{i\ell} + M_{jk}.$$

For each row $1 \leq r \leq n$, let $f(r)$ be the index of the leftmost minimum element in that row. For example, if

$$M = \begin{bmatrix} 10 & 17 & 13 & 28 \\ 17 & 22 & 17 & 29 \\ 24 & 28 & 22 & 34 \\ 11 & 13 & 6 & 5 \end{bmatrix}$$

then $f(1) = 1$, $f(2) = 1$, $f(3) = 3$, and $f(4) = 4$.

1. Prove that f is non-decreasing, i.e., $f(1) \leq f(2) \cdots \leq f(m)$.
2. Design an $O(n \log m + m)$ time algorithm to compute $f(r)$ for all the rows. Justify the correctness of the algorithm and analyze its running time.

Question assigned to the following page: 3.1

Solution.

Part 1. To prove that function f is non-decreasing or $\forall a, b \in [1, n] : a < b.\ f(a) \leq f(b)$, let's, by contradiction, assume that $\exists a < b\ s.t.\ f(a) > f(b)$. Then by the rule this matrix is defined on, $M_{a,f(a)} + M_{b,f(b)} \leq M_{a,f(b)} + M_{b,f(a)}$ *(1)*. Thus, the new elements $M_{a,f(b)}$ and $M_{b,f(a)}$ are also part of the row a, b respectively. Then, by definition of function f: $\forall x, y \in [1, m]$. $M_{a,f(a)} \leq M_{a,x} \wedge M_{b,f(b)} \leq M_{b,y}$ *(2)*. Therefore, because $M_{a,f(b)}$ and $M_{b,f(a)}$ are part of this range, we know that $M_{a,f(a)} \leq M_{a,f(b)} \wedge M_{b,f(b)} \leq M_{b,f(a)}$ *[from (2)] (3)*. Consider the following two cases:

1. If $M_{a,f(a)} + M_{b,f(b)} < M_{a,f(b)} + M_{b,f(a)}$, then, by plugging it in the statement *(3)* (as both $M_{a,f(a)}$ and $M_{b,f(b)}$ are at most what we're replacing), we have $M_{a,f(b)} + M_{b,f(a)} < M_{a,f(b)} + M_{b,f(a)}$, which leads to the contradiction that some number is greater than itself. $\therefore$ Contradiction

2. If $M_{a,f(a)} + M_{b,f(b)} = M_{a,f(b)} + M_{b,f(a)}$ *(4)*, then $(M_{a,f(a)} = M_{a,f(b)}) \wedge (M_{b,f(b)} = M_{b,f(a)})$. To prove that, let's assume, for the sake of contradiction, that $(M_{a,f(a)} \neq M_{a,f(b)}) \wedge (M_{b,f(b)} \neq M_{b,f(a)})$, then consider the following four cases:

a. If $M_{a,f(a)} > M_{a,f(b)}$; means $M_{a,f(b)}$ is lower than the minimum value (in the row a), which is $M_{a,f(a)}$. This leads to the contradiction as there does not exist strictly lower value than actual min from the statement: $M_{a,f(a)}$. $\therefore$ Contradiction

b. If $M_{b,f(b)} > M_{b,f(a)}$; means $M_{b,f(a)}$ is lower than the minimum value (in the row b), which is $M_{b,f(b)}$. This leads to the contradiction as there does not exist strictly lower value than actual min from the statement: $M_{b,f(b)}$. $\therefore$ Contradiction

c. If $M_{a,f(a)} < M_{a,f(b)}$; means $M_{a,f(b)}$ is greater than the minimum value (in the row a), which is $M_{a,f(a)}$. However, because of the relationship from the statement *(4)*, we know that now $M_{b,f(a)} = (M_{a,f(a)} + M_{b,f(b)}) - M_{a,f(b)} < M_{b,f(b)}$ or just $M_{b,f(a)} < M_{b,f(b)}$ (row b). This leads to the same contradiction described in part b. $\therefore$ Contradiction

d. If $M_{b,f(b)} < M_{b,f(a)}$; means $M_{a,f(b)}$ is greater than the minimum value (in the row b), which is $M_{b,f(b)}$. However, because of the relationship from the statement *(4)*, we know that now $M_{a,f(b)} = (M_{a,f(a)} + M_{b,f(b)}) - M_{b,f(a)} < M_{a,f(a)}$ or just $M_{a,f(b)} < M_{a,f(a)}$ (row a). This leads to the same contradiction described in part a. $\therefore$ Contradiction

Because, from the Case #2, we have shown that $(M_{a,f(a)} = M_{a,f(b)}) \wedge (M_{b,f(b)} = M_{b,f(a)})$, then in row a we have found another contradiction. In particular, that function f returns index of the *leftmost minimum element* in the row. However, from the assumptions, $f(a) > f(b)$, we have shown two element $M_{a,f(a)}$ and $M_{a,f(b)}$ such that $M_{a,f(a)}$ is minimum, but with the index greater than the other value equal to it and located leftwards. $\therefore$ Contradiction

Therefore, because both cases are false, function f is indeed non-decreasing.

Question assigned to the following page: 3.2

Part 2. Algorithm:

Similarly to the discussion section, the general idea is to prevent the algorithm run through each element ($O(n^2)$ complexity), but efficiently constrain the search between already found indexes of min around it, thanks to the non-decreasing property of f (use divide-and-conquer).

Let M' be the array consisting of only odd-indexes rows of M. We then call our subroutine on M' and compute $m_r \in M$ for all odd values of r. Now, to compute the values of m_r for *even* r (located in between), we do the following:

Let $r = 2k$ (notation for the even rows). Then, we search for the minimum element in row r but constrained to $[m_{2k-1},\ m_{2k+1}]$. In case $m = 2k$ (the last line in the table is even) then set $m_{2k+1} = n$ (b/c no m+1 row exists). We set $f(2k)$ to be the index of such minimum.

Finally, in the case where there is only one row, we simply iterate through it and compute the minimum index (as a base case for divide-and-conquer algorithm).

Time Complexity. Let m, n be the number of rows and columns at any point. When computing m_r for even r, we go through each pair of adjacent odd values (m_{2k-1}, m_{2k+1}) and check $m_{2k+1} - m_{2k-1} + 1$ values. Then, this computation has runtime at most:

$$\sum_{1 \le 2k+1 \le m} (m_{2k+1} - m_{2k-1} + 1) \le f(m) - f(1) + \frac{m}{2} \le \frac{m}{2} + n \tag{1}$$

We modify the number of elements we run though the row, but keep the general amount of columns (n) as a constant. Then, the runtime for the algorithm in terms of m (# of rows) becomes:

$$\begin{aligned} T(n,m) &= T(n,m/2) + (\frac{m}{2} + n) \\ T(n,m/2) &= T(n,m/4) + (\frac{m}{4} + n) \\ T(n,m/4) &= T(n,m/8) + (\frac{m}{8} + n) \\ &\vdots \\ T(n,m) &= T(n,m/2^k) + (\sum_m \frac{m}{2^k} + nk), \text{where } k = \log_2 m \end{aligned} \tag{2}$$

considering the fact that the amount of times we can divide n by 2 (even/odd) is $\log_2 m$.

$$\therefore T(n,m) = O(n) + O(m + n\log(m)) = O(m + n(\log(m) + 1)) \tag{3}$$

Therefore, if we initially have $n \times m$ table, we obtain a runtime of $T(n) = O(nlogm + m)$.

Question assigned to the following page: 3.2

Proof. By Strong Induction. Because n (the number of columns) is constant thoughout the algorithm, we consider the induction on the terms of m (number of rows); then:

Base Case: For the smallest matrix $M_{1,n}$ ($1 \leq i$ by def), my algorithm:
Finds the only odd row 1 and then iterate over it from start to the end, finding a minimum.

Inductive Step: Now, assume that the algorithm holds for all $M_{m_r,n}$ with all odd values of r on each subproblem. Want to prove that it also holds for the parent submatrix $M_{m,n}$.

Induction Step Let's break down the proof into two parts:

1. Correctness of the algorithm for odd-indexed rows of $M_{m,n}$ ($M'_{m_r,n}$):
   Because $r \in [1, m] \implies M_{m_r,n} \in M_{m,n}$, we assume that the algorithm is correct for the submatrix M', as we assumed it previously by the induction hypothesis (strong induction).

2. Correctness of the algorithm for even-indexed rows:
   For even-indexed rows ($2k$), the algorithm searches for the minimum element in row r but constrained to $[m_{2k-1}, m_{2k+1}]$. The correctness follows from the fact that when we compute m_r for odd r, the minimum index overall is the same as the minimum index constrained to $m_{2k-1} \leq m_{2k} \leq m_{2k+1}$ b/c of the monotonicity of m_i proofed in part #1 of this problem.

Therefore, by strong induction, the algorithm correctly computes $f(r)$ for all rows in the table $M_{m,n}$ with $O(n \log m + m)$ time complexity, as was needed to show in the induction step.

No questions assigned to the following page.

4 Maximum Self-Match

Let S be a string with n characters taken from an alphabet A of size k. For any $1 \leq i \leq n$, define D_i to be the number of characters that match between S and $S[i\colon]$ (where $S[i\colon]$ is the substring of S that does not include the first i characters of S). For example, if $S =$ ababa then $D_1 = 0$ and $D_2 = 3$. Design an $O(kn \log n)$ time algorithm to compute the largest match D_i. Justify the correctness of the algorithm and analyze its running time.

Question assigned to the following page: 4

Solution. Notes: My approach revolves around a key observation: when multiplying a polynomial like $f(x) = x + x^2 + x^3$ with another polynomial containing negated powers, e.g., $g(x) = x^{-1} + x^{-2} + x^{-3}$, it is effectively highlight new positions via cross correlation. This technique is often done to measure the similarity or degree of match between two arrays or polynomials. At each position or offset, the cross-correlation produces a value that indicates how similar two polynomials are when aligned at that specific position. When the two polynomials are perfectly aligned, the cross-correlation reaches its maximum value. This might be helpful to further calculate the amount of times letters match between each other to get D_i max (in depth explained below).

Notations and SetUp: Let's denote each alphabet character's position in the string with a polynomial called f_a, where $a \in A$ is the letter from the alphabet. For each letter a found in the word, let's denote its position n, of the same letters a in S, as x^n. For instance, if S is "ababa," we create $f_a = x^1 + x^3 + x^5$ (indicating positions $n = 1$ and $n = 3$) and $f_b = x^2 + x^4$. Further use:

$$f_a = \sum_{i=1}^{n} x^n, \text{ if } a_n \text{ exists.} \tag{4}$$

Let g_a will represent a cross-correlated polynomial to f_a with all powers to be negated:

$$g_a = \sum_{i=1}^{|n|} x^{-n}, \text{ if } a_n \text{ exists.} \tag{5}$$

The function g_a is built on the way to find how far away all letters a are located from each other in the convolution (explain below in the proof).

Algorithm:

1. For each unique letter in the alphabet size k (loop), find its position in the word, and build two polynomials f_a and g_a (demonstrating component with shift) described above.

2. Apply FFT polynomial multiplication algorithm described in the class before, to efficiently multiply them to the new polynomial $h_a(x)$ (notation) with $O(n \log n)$ time. Append to array L.

3. For each of these polynomials (at most k of them), sum them together to find the most common optimal shift in of all letters in the word S:

$$h(x) = \sum_{i=1}^{|L|} f_a, \text{ for each } f_a \text{ in L} \tag{6}$$

4. Run (for loop) though $h(x)$ and find the element x with the power of i and set D_i equal to x^i coefficient, if exists. Otherwise, return D_i equal to 0 (matchings).

Question assigned to the following page: 4

Time Complexity: For each unique letter in the alphabet (with a total of k unique letters), the algorithm locates occurrences of that letter within the word S, which takes $O(kn)$ time. Building the two polynomials f_a and g_a for each letter is a one-time operation and contributes $O(k)$ at most because the polynomials are unique for each letter. Then, applying the Fast Fourier Transform (FFT) algorithm for polynomial multiplications on f_a and g_a takes $O(n\log(n))$ time per letter, resulting in a total time complexity of $O(k(n\log(n)))$ (from the class). Finally, summing up the k resulting polynomials (as they are unique) to obtain the final polynomial $h(x)$ takes $O(k)$ time.

Therefore, **total time-complexity** of this algorithm:

$$\begin{gathered} O(kn) + O(kn\log n) + O(k) = O(k(n+1)) + O(kn\log n) = \\ O(kn(\log n + 1)) = O(kn\log n) \end{gathered} \tag{7}$$

Claim: For a given letter a in the alphabet, the output of the Fast Fourier Transform multiplication $h_a(x) = f_a(x) \cdot g_a(x)$, where:

$$\begin{aligned} f_a(x) &= x^{p_1} + x^{p_2} + x^{p_3} + \cdots + x^{p_j} \\ g_a(x) &= x^{-p_1} + x^{-p_2} + x^{-p_3} + \cdots + x^{-p_j} \end{aligned} \tag{8}$$

is related to all possible matchings between two substrings of S.

Here p denote the letter a index in the string.

Proof of the Claim:

1. Consider the polynomials $f_a(x)$ and $g_a(x)$ for a specific letter a in the word S. The powers in $f_a(x)$ represent the positions of the occurrences of letter a in the string S (by set-up).

2. When the algorithm multiply $f_a(x)$ and $g_a(x)$ using FFT-based polynomial multiplication, it computes their convolution $h_a(x)$ (subroutine, proofed in the class):

$$h_a(x) = x^{p_1-p_1} + x^{p_1-p_2} + x^{p_1-p_3} + \cdots + x^{p_{j-1}-p_j} + x^{p_j-p_j}$$

3. The resulting polynomial $h_a(x)$ will have terms of the form $x^{p_f-p_g}$, where p_f and p_g correspond to powers of functions f and g, respectively. Each $|p_f - p_g|$ term of h_a represents how far away matching characters were apart. **To prove that**, let's show that for arbitrary $x^{p_f} \cdot x^{p_g}$ s.t. $p_f - p_g = i$, (note: we do not consider case $p_f - p_g = -i$ as working with non-negative integers i ($i \geq 0$) by definition), is bijective with a set of shifted letters of string S in the following form:

$$S[p_f] = S[p_g + i]$$

- $\Longrightarrow$: if $p_f - p_g = i$, then $S[p_f] = S[p_g + (p_f - p_g)]$. So $S[p_f] = S[p_f]$ and $S[p_f] = S[p_g + i]$ (b/c $p_f - p_g = i$), what we needed to prove.

Question assigned to the following page: 4

- $\Longleftarrow$: if $S[p_f] = S[p_g + i]$, consider their representation in the polynomial: $x^{p_f} \cdot x^{-(p_g - i)}$, then by the properties of exponentiation, $x^{p_f - (p_g - i)}$. So, b/c we expect same position difference equal to 0, $x^{p_f - p_g + i} = x^0 \implies p_f - p_g + i = 0$, then $p_f - p_g = i$, what we needed to prove.

4. If multiple terms in $h_a(x)$ have the same power, it implies that there are multiple pairs of characters at the same distance from each other within the substrings. This represents the number of matches that occur at each possible shift of the substrings (within coefficient).

Finally, summing up all these polynomials $h_a(x)$ for all letters $a \in A$ effectively accumulates the count of matches at each possible shift for the entire string S. This is because each term in the resulting polynomial represents a specific shift, but not an index of the letter a anymore. Thus, its coefficient represents the total number of matches at the shift i.

Therefore, by computing and summing the polynomials $h_a(x)$ for all letters a in the word S, the algorithm obtains the largest match D_i for all possible shifts i, which is the solution.

No questions assigned to the following page.

5 1-D k-means

In the k-means problem, we are given a set of n points $X \coloneqq \{x_1, x_2, \ldots, x_n\} \subseteq \mathbb{R}^d$ (for some $d \geq 1$) and an integer $k \geq 1$, and the goal is to partition the points into k (disjoint) clusters such that the sum of squared-distances from each point x_i to the "center" of the cluster that x_i is assigned to is minimized (as illustrated in Figure 2 for $d = 2$).

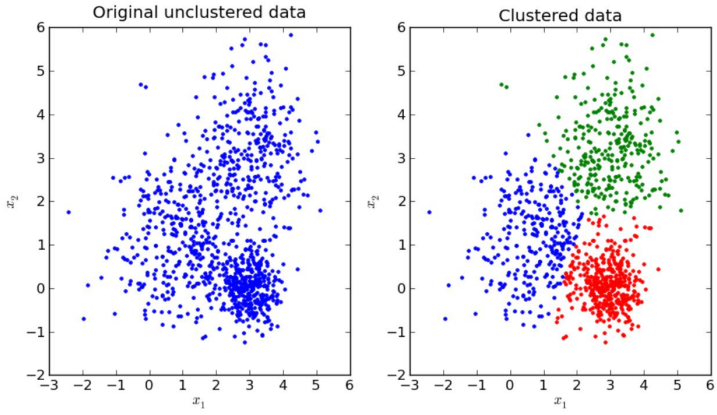

Figure 2: Left side is the original 2D data, right side is the visualization after clustering; different colors denote different clusters. Adapted from https://mubaris.com/posts/kmeans-clustering/.

Definition of clustering. Consider clusters $C_1, C_2, \ldots, C_k \subseteq \{1, 2, \ldots, n\}$. These clusters are said to be **valid** if:

1. they are disjoint (i.e., for any distinct i and j, $C_i \cap C_j = \emptyset$); and
2. they cover all points (i.e., $\bigcup_{i=1}^{k} C_i = \{1, 2, \ldots, n\}$).

Each cluster C_ℓ specifies the points assigned to the ℓ-th partition of the dataset $\{x_i : i \in C_\ell\}$.

The **center** c_ℓ of each cluster C_ℓ is defined as the average of all points in C_ℓ, i.e.,

$$c_\ell \coloneqq \frac{1}{|C_\ell|} \sum_{i \in C_\ell} x_i.$$

Formal statement. The **k-means clustering problem** is defined as follows: Given a set of n points $X \coloneqq \{x_1, x_2, \ldots, x_n\}$ in $\mathbb{R}^d$, and an integer $k \geq 1$, output a valid set of k clusters $C_1, C_2, \ldots, C_k$ that minimizes the following **cost**:

$$\texttt{cost}(C_1, C_2, \ldots, C_k) = \sum_{\ell=1}^{k} \sum_{i \in C_\ell} (x_i - c_\ell)^2 .$$

No questions assigned to the following page.

In general, when $d > 1$ (where d is the dimension of the space in which the points lie), it is difficult to solve the clustering problem exactly, and heuristics are used in practice. In this problem, we design an efficient algorithm in the special case when $d = 1$, i.e., when all points lie on the real number line as shown below. We assume that the points $x_1, x_2, \cdots, x_n$ are sorted in non-decreasing order.

Let us introduce some quantities that will be helpful. For each $i \in \{1, 2, \ldots, n\}$ and $\ell \in \{1, 2, \ldots, k\}$, define $\texttt{cost}(i, \ell)$ to be the *minimum* cost of partitioning points $x_1, x_2, \cdots, x_i$ into ℓ valid clusters. Note that in the k-means problem, we want to compute $\texttt{cost}(n, k)$.

For each distinct $i, j \in \{1, 2, \ldots, n\}$, define $\texttt{unit}(i, j)$ as the cost of placing points $x_i, x_{i+1}, \cdots, x_j$ into one cluster, i.e.,

$$\texttt{unit}(i, j) \coloneqq \sum_{k=i}^{j} (x_k - \mu_{ij})^2 \quad \text{where} \quad \mu_{ij} \coloneqq \frac{1}{j - i + 1} \sum_{k=i}^{j} x_k.$$

Problem. Our aim is to design an algorithm for the 1D k-means problem.

1. For each $i \in \{1, 2, \ldots, n\}$, express $\texttt{cost}(i, 1)$ in terms of $\{\texttt{unit}(j, \ell) \colon j, \ell \in \{1, 2, \ldots, n\}\}$.

2. Given an n and k, express a recursive relation for $\texttt{cost}(n, k)$ in terms of $\{\texttt{unit}(j, \ell) \colon 1 \leq j, \ell \leq n\}$ and $\{\texttt{cost}(j, \ell) \colon 1 \leq j < i \text{ and } 1 \leq \ell < k\}$.

3. Design an algorithm that given X and $1 \leq i, j \leq n$, outputs $\texttt{unit}(i, j)$ in $O(1)$ time. The algorithm is allowed to do $O(n)$ pre-processing (without the knowledge of i and j).

4. Using all of the above parts design a dynamic-programming-based algorithm that, given X and k, outputs an optimal set of k clusters $C_1, C_2, \ldots, C_k$ for the 1D k-means problem. Justify the correctness of the algorithm and analyze its running time. To get full credits, your algorithm should run in $O(n^2 k)$ time.

Questions assigned to the following page: 5.2 and 5.1

Solution. Question #1:

Consider an observation that *cost* function is the minimum cost to split $\{x_1, x_2, \ldots, x_n\}$ into k clusters, where $k = 1$ for this specific case. Then, to express cluster in terms *unit* cost of a single cluster, that is enough to say that any $cost(i, 1) == unit(1, i)$ because *unit* is applied within a single cluster by definition.

Solution. Question #2:

$$\texttt{cost}(n, k) = \min_{j=k-1}^{n-1} \{\texttt{cost}(j, k-1) + \texttt{unit}(j+1, n)\}$$

Justification: The recurrence expresses the minimum cost of partitioning the first n points into k clusters by considering all possible break points between the last two clusters (note: if we start with $j < k - 1$ there would be a cluster with 0 points). For each break point j, it calculates the cost of partitioning the points before j into $k-1$ clusters (which is $\texttt{cost}(j, k-1)$) and the cost of placing the points from $j+1$ to n into a single cluster (which is $\texttt{unit}(j+1, n)$). The algorithm takes the minimum of the sum of these costs to find the optimal cost for partitioning the first n points into k clusters.

Proof. By contradiction, suppose there exists a better partition that results in a lower cost than the one obtained by the recurrence. In other words, there is some j^* such that

$$\texttt{cost}(n, k) > \texttt{cost}(j^*, k-1) + \texttt{unit}(j^*+1, n)$$

But this contradicts the correctness of the recurrence because the recurrence explores all possible break points and if there would exist the other group smaller than k-1 clusters, the algorithm would captured it by the recurrence.

Question assigned to the following page: 5.3

Solution. Question #3:

Idea: Using dynamic programming, create a helper arrays S, P to hold a cumulative sum of elements from 1 to any element $e \in 1, 2, \ldots, n$ ($O(n)$ time). Thus, sum of elements between any i, j could be done with $O(1)$ time by taking the difference between $S[j] - S[i]$ or $P[j] - P[i]$.

1. Initialize array S, P, where $S[i] = \sum_{r=1}^{i} x_k$ and $P[i] = \sum_{r=1}^{i} x_k^2$ for $1 \leq r \leq n$. This can be done in linear time by iterating through the elements of X. In particular the algorithm would be: 1. set $S[1] = x_1$ and $P[1] = x_1^2$. 2. for the following t: $2 \leq t \leq n$, set $S[t] = S[t-1] + x_t$ and $P[t] = P[t-1] + x_t^2$, where x_t might be accessed in $O(1)$ time (suppose from the dictionary).

3. Now, to calculate $\texttt{unit}(i, j)$, consider the observation that

$$\texttt{unit}(i,j) \coloneqq \sum_{k=i}^{j}(x_k - \mu_{ij})^2 = \sum_{k=i}^{j}(x_k^2 - 2x_k\mu_{ij} + \mu_{ij}^2) = \sum_{k=i}^{j}(x_k^2) - \sum_{k=i}^{j}(2x_k\mu_{ij}) + \sum_{k=i}^{j}(\mu_{ij}^2)$$

Here, μ_{ij} is the mean of the elements in the range $[i, j]$ and can be calculated using the cumulative sums (using part # 1) such that

$$\sum_{k=i}^{j} x_k = \sum_{k=1}^{j} x_k - \sum_{k=1}^{i-1} x_k \text{ and } \sum_{k=i}^{j} x_k^2 = \sum_{k=1}^{j} x_k^2 - \sum_{k=1}^{i-1} x_k^2$$

$$\mu_{ij} = \frac{S[j] - S[i-1]}{j - i + 1}$$

So, to compute $\texttt{unit}(i, j)$:

$$\texttt{unit}(i,j) = \sum_{k=i}^{j}(x_k^2) - 2\left(\frac{S[j] - S[i-1]}{j-i+1}\right)\sum_{k=i}^{j} x_k + \left(\frac{1}{j-i+1}\right)(S[j] - S[i-1])^2 =$$

$$(P[j] - P[i-1]) - 2\left(\frac{S[j] - S[i-1]}{j-i+1}\right)(S[j] - S[i-1]) + \left(\frac{1}{j-i+1}\right)(S[j] - S[i-1])^2 =$$

$$(P[j] - P[i-1]) - \left(\frac{(S[j] - S[i-1])^2}{j-i+1}\right)$$

Time Complexity

Creating Dependency Arrays: The time complexity to create the dependency arrays S and P is $O(n)$. This is because for each index t in the range $2 \leq t \leq n$, we compute $S[t] = S[t-1] + x_t$ and $P[t] = P[t-1] + x_t^2$. This is linear as it involves iterating through the elements of the input array X. The mathematical calculations are $O(1)$ time. Therefore, the time complexity is $O(n)$ for preprocessing and $O(1)$ for other evaluations.

Question assigned to the following page: 5.4

Quesrion # 4. Algorithm

1. Initialize the dependency array *cost*, where $cost[i][j]$ represents the minimum cost of partitioning the first i data points into j clusters. For each cluster count j, $2 \leq j \leq k$, and each data point count i, $j \leq i \leq n$, initialize $cost[i][j]$ to positive infinity, indicating that the minimum cost has not been determined yet.

2. For a base case, calculate $cost(i, 1)$ for all i using the algorithm designed in Part 3. These values represent the cost of having a single cluster for data points x_1 through x_i. This is achieved using Part 3 and its relationship to *unit* and *cost* if $k = 1$.

3. Enter a nested loop, iterating over potential break points within the data points from $j - 1$ to i. For each break point x, calculate the cost of forming the current cluster (j) by taking the minimum between the current $cost[i][j]$ and the sum of the cost of the previous cluster ($cost[x][j-1]$) and the cost of forming the current cluster ($unit[x+1][i]$). This iterative process systematically identifies the optimal way to partition the data into j clusters, integrating Part 2's recurrence relation.

4. Identify optimal clusters and preserve data points that belong to each cluster by keeping track of data points after calculating $cost[i][j]$ during the iterative process.

Justification

This is proof by induction (please, look Part #2). The induction step has already been shown in the Part 2, while the base case is in Part 1 and 3. In general, Part 2, we demonstrated that the recurrence relation correctly calculates the minimum cost of partitioning the data into j clusters by considering all possible break points between (j-1) and (i-1) in the table. Thus, running through all data points within the nested loop in Part 4 ensures that the algorithm exhaustively evaluates every possible combination given X and k.

Time Complexity

1. Initializing the *cost* array (preprocessing) takes constant time as the size of these arrays is determined by the input size and the number of clusters, which are constants.

2. Calculating $unit[i][1]$ for all i using the algorithm from Part 3 takes $O(n)$ time.

3. The outer loop iterates through the number of clusters k, and the inner loop iterates through the data points i. In each iteration of the outer loop (for a given k), we go through all data points, leading to n iterations of the inner loop. Since this process happens for each value of k from 2 to the maximum, it leads to a maximum of k iterations for the outer cluster loop.

Therefore, combining these loops, in the worst case, takes $O(n^2k) + O(n) = O(n^2k)$ time.

No questions assigned to the following page.

6 Maximum-Cost Independent Sets on Trees

Suppose we have a graph $G = (V, E)$ with n vertices, where each vertex $v \in V$ has a **reward** $r_v \in \mathbb{R}$, which may be positive, zero, or negative. We define the **reward** $R(S)$ of a subset $S \subseteq V$ to be the sum of rewards of all vertices in S, i.e.,

$$R(S) \coloneqq \sum_{v \in S} r_v.$$

A subset of vertices $S \subseteq V$ is defined be an **independent set** if and only if no two vertices $u, v \in S$ are connected via an edge in E.

Suppose we want to choose S to maximize the reward $R(S)$, but we cannot pick two vertices which are connected by an edge. This means we want to choose an independent set S to maximize $R(S)$. We will see how to compute such S for special types of graphs G.

1. For this subpart, suppose that G is a path graph (see Figure 3). Design an $O(n)$ time algorithm that, given $G = (V, E)$, outputs the maximum value of reward $R(S)$ among all the independent sets S of G.

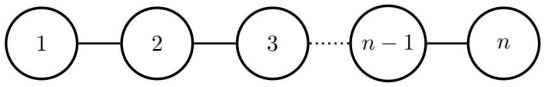

Figure 3: A path graph with n nodes.

2. For this subpart, suppose that G is a tree (see Figure 4). Design an $O(n^2)$ time algorithm that, given $G = (V, E)$, outputs the maximum value of reward $R(S)$ among all the independent sets S of G.

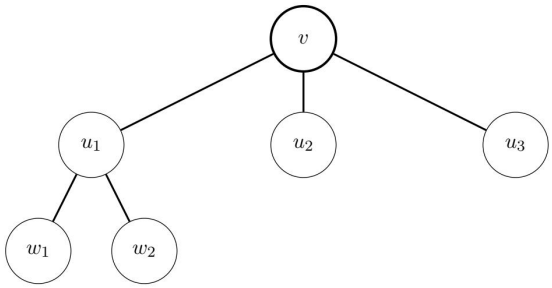

Figure 4: A tree with 6 vertices.

3. Repeat the exercise in part (2) but design an algorithm that runs in $O(n)$ time.

No questions assigned to the following page.

For each part, justify the correctness of your algorithm and analyze its running time.

Question assigned to the following page: 6.1

Part #1. Algorithm

```
getMaxRewardPart1(G):
    n <- number of vertices in V

    if (n == 0):
        return 0
    else if (n == 1):
        return max(R(V[1]), 0) # 0 if negative

    # Define an array called dp with size n to store the maximum rewards
    vector<int> dp(n, 0)

    # Initialize the first two values using the rewards of the first two vertices
    dp[1] = max(R(V[1]), 0)
    dp[2] = max(R(V[1]), R(V[2]), 0)

    # Iterate through the vertices
    for (int i = 3; i <= n; i++) {
        # Choose the maximum reward between taking the current vertex and not taking it
        dp[i] = max(dp[i-1], dp[i-2] + R(V[i]), 0) # 0 if negative
    }

    # The final element of dp[] will contain the maximum reward achievable
    return dp[n]
```

Time Complexity

The algorithm runs through each vertex in order of their position exactly once, which contributes to a linear time complexity of $O(n)$, where n is the number of vertices in the tree. The insertion of values into the *dp* array is also an $O(1)$ operation on average, and the computations involving maximizing rewards and recursive calls for each node are constant-time operations. Therefore, the overall time complexity of the algorithm is $O(n)$.

Question assigned to the following page: 6.1

Proof By Strong Induction

Notations:
Let n $\in \mathbb{N}$ be the amount of *vertices* (notation) given by the problem. By definition, we know that n ≥ 1. Similarly, let *dp* (notation) be an dependency array, size of n, to hold the greatest amounts of reward output, till some k $\in \{1, 2, ..., n\}$, index of a vertex.

Base Case:
if n = 1 $\implies$
return max(R(V[1]), 0)) rewards, because no other choice exists. (Returns 0 if reward is negative)

if n = 2 $\implies$
return max(R(V[1]), R(V[2]), 0) coins, because no adjacent vertices can be selected (by definition) and only one, with the greatest amount of coins, can be a solution to this case.

Induction Hypothesis:
Let dp[1] = max(R(V[1]), 0) and dp[2] = max(R(V[1]), R(V[2]), 0). Now, assume that for some arbitrary vertex index k $\leq$ n, for any vertices m: k $\geq$ m ≥ 2, my algorithm correctly identifies the path with greatest amount of reward in total. [Want to show that my algorithm will also correctly predict the optimal route for k+1]

Induction Step: By definition of the independent set, any two adjacent vertices u and v, where $(u, v) \in E$, cannot be taken at the same time, which implies that every time we consider a new induction step, we have two distinct cases:

1. The algorithm either pursues not to take the current vertex $k + 1$ to keep the greatest benefit from before (e.g. code: $dp[i - 1]$).
2. Or take the vertex it's currently located-at, R(V[k+1]) reward, while ensuring reward is non-negative (e.g. code: $dp[i - 2] + R(V[i])$). Otherwise, not include it (part 1).

To maximize the total reward for k+1 vertices, we update dp[k+1] as follows: $dp[k + 1] = max(dp[k], max(R(V[k+1]), 0))$. Therefore, by considering all possible cases, we know that for any k $\leq$ n, the algorithm works.

Question assigned to the following page: 6.2

Part 2. Algorithm

Let's start from the root node of the tree and recursively traverse each node in the tree once. For each node, we iterate through its children (treating them as grandchildren). This means we iterate through all the children of each node regardless of whether we have already calculated their values. Thus we can achieve $O(n^2)$ time complexity.

```
def maxRewardPart2(node):
    # stop recursion
    if not node:
        return 0 # (rewards)

    # two cases: including and excluding the current node
    include_current = max(node.reward, 0) # reward from current node
    exclude_current = 0 # reward without it

    # Iterate over the children of the current node (reach grandchildren)
    for child in node.children:
        exclude_current += maxRewardPart2(child)  # Exclude the current node
        for grand/child in node.children:
            include_current += maxRewardPart2(grand/child)
            # Include the current node and exclude its grandchildren

    # Return the maximum reward between including and excluding the current node
    return max(include_current, exclude_current, 0)
```

Time Complexity

1. We visit each node in the tree exactly once during the depth-first traversal. This contributes $O(n)$ to the time complexity, where n is the number of vertices in the tree.

2. For each node, we iterate through its children. The number of children is $deg(v)$, where v is the current node. Then, for each child of the current node, we iterate through its children (grandchildren). The number of grandchildren of each child is $\sum deg(u)$, where u belongs to the children of the current node.

In the worst case, when the tree is linear, with potentially n levels, the overall time complexity $O(n * \sum deg(u))$, which simplifies to $O(n^2)$ with the total number of descendants equal to n - 1, where n represents the number of vertices. Therefore, the time complexity of this algorithm:

$$O(\sum_{v \in V}(deg(v) + \sum deg(u))), \text{ where } u \in child(v) = O(n^2) \text{ at most} \tag{9}$$

Question assigned to the following page: 6.2

Proof By Strong Induction

Base Case: For a tree with one vertex ($n = 1$), the algorithm trivially works correctly as there is only one choice to consider, which is to include the single vertex in the independent set. The algorithm returns the correct reward in this case it is non-negative value.

Inductive Hypothesis: Assume that the algorithm correctly finds the maximum reward among all independent sets for all trees with k vertices, where $k \leq n$ for some positive integer n and correctly handles the condition of not selecting two adjacent vertices (e.g., not selecting two vertices u and v if $(u, v) \in E$) and negative rewards. [Want to show that the algorithm also works correctly for a tree with $(k + 1)$ vertices]

Inductive Step: By definition of the independent set, any two adjacent vertices u and v, where $(u, v) \in E$, cannot be taken at the same time, which implies that every time we consider a new induction step, we have two distinct cases. The algorithm starts from the root node and explores its children (if exist. Otherwise, skip to return), treating them as grandchildren. It calculates the maximum reward for these two cases: including (if non-negative) and excluding the current node. If node is included, then it pursues path with the grandchildren to be excluded as we cannot take reward from any two adjacent vertices u and v, where $(u, v) \in E$ (by def); plus, the parent node of grandchildren (if non-negative). If not, it sums up all the rewards from grandchildren paths already being checked for not negative rewards (set-up in the base case and induction step of including parent node).

For each child of the current node, the algorithm applies the same logic, treating the grandchildren as if they are the children of the child node. This recursive process continues until all nodes in the tree are considered.

Since we have shown that the algorithm works correctly for the base case ($n = 1$) and have demonstrated that if it works for n, it also works for $(k + 1)$, we have proven by strong induction that the algorithm correctly finds the maximum reward among all independent sets for all trees.

Question assigned to the following page: 6.3

Part 3. Algorithm

```
def maxRewardPart3(tree):
    dp = {} # Initialize a dependancy array to store each node.

    # Define a DFS (Depth-First Search) function to compute dp values.
    def dfs(node):

        # Initialize values for the two cases: node is not selected (0) or selected (1).
        # Check if reward is non-negative
        dp[node] = [0, max(R[node], 0)]

        for neighbor in tree[node]:
            # Recursively compute dp values for the subtree rooted at neighbor.
            dfs(neighbor)

            # When the current node is selected, add the maximum rewards
            #from subtrees where neighbors are not selected.
            dp[node][1] += dp[neighbor][0]

            # When the current node is not selected, add the maximum rewards
            # from subtrees where neighbors can be selected or not.
            dp[node][0] += max(dp[neighbor][0], dp[neighbor][1], 0)

    # Start the DFS. Can choose any node as the root.
    node = tree.keys()[0] # let it be 0
    dfs(node)

    # The final answer in dp is the maximum reward
    # achievable when the root is either selected or not.
    return max(dp[node][0], dp[node][1])
```

Time Complexity

The algorithm runs through each vertex in the tree exactly once during the Depth-First Search traversal, which contributes to a linear time complexity of $O(n)$, where n is the number of vertices in the tree. The insertion of values into the *dp* array is also an $O(1)$ operation on average, and the computations involving maximizing rewards and recursive calls for each node are constant-time operations. Therefore, the overall time complexity of the algorithm is $O(n)$.

Question assigned to the following page: 6.3

Proof By Strong Induction

Notations:
Let n $\in \mathbb{N}$ be the amount of *verticies* (notation) in the tree given by the problem. Let *dp* (notation) be an dependency array, size of $n \times 2$, to hold the greatest amounts of reward output, till some k $\in \{1, 2, ..., n\}$, index of a verticy.

Base Case:
if n = 1 $\implies$
We can either take this node or not take it. The algorithm takes *max* of this choices and because it does not see anymore any childs of this node, this is the final result (if non-negative. otherwise 0). ✓

Induction Hypothesis:
Now, assume that for some arbitrary vertix index k $\leq$ n, for any node m: k $\geq$ m $\geq$ 2, the algorithm correctly identifies the path with greatest amount of reward in total and handles the condition of not selecting two adjacent vertices. [Want to show that the algorithm will also correctly predict the optimal route for k+1 vertix]

Induction Step: By definition, a reward from any two adjacent vertices u and v, where $(u, v) \in E$, cannot be taken at the same time. This implies that every time we consider a new induction step on neighbors connected by edges in the graph, we have two distinct binary cases:

1. The algorithm either pursues the path that includes this vertix (i.e. dp[node][1]), ahead of time checking if it's non-negative, and not its neighbors (i.e. dp[neighbor][0]).

2. Otherwise, it does not take this vertex (i.e. dp[node][0]) and preserve reward from the neighbors either taking them or not, similarly to the problem # 1 (b/c treasure from two adjacent vertices may be left not taken).

To maximize the total output in case # 2, the algorithm takes max of *taking or not neighbors values* (i.e. max(dp[neighbor][1], dp[neighbor][0])). By doing this, it consideres the most favorable outcome based on the assumed shortest past till k from the indaction hypothesis.

Thus, we know that for any k $\leq$ n, the algorithm works by considering all possible cases with $2 * n$ dependency array.