COMP3711 Assignment 3 - Double Sized Homework

TANUWIJAYA, Randy Stefan
(20582731)
rstanuwijaya@connect.ust.hk

Department of Physics - HKUST
Department of Computer Science and Engineering - HKUST

April 11, 2021

Problem 1: Huffman Coding

i	1	2	3	4	5	6	8	8	9	10	11	12	13	14	15
a_i	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
$f(a_i)$	36	6	9	10	58	7	17	29	45	2	3	12	15	26	28

Table 1.1: Frequency Distribution

The table above lists a vocabulary of 15 characters (a to o) and their frequencies in a document. Apply Huffman coding to construct an optimal codebook. Your solution should contain three parts:

(a) A full Huffman Tree.

Figure 1.1 illustrates the Huffman Encoding heap tree.

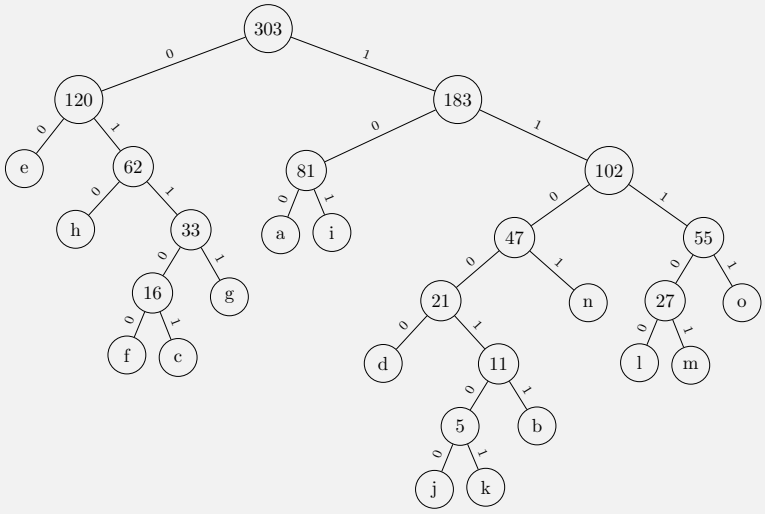

Figure 1.1: Huffman Encoding Heap

(b) The final codebook that contains the binary codewords representing a to o (1111sorted in the alphabetical order on a to o

The alphabetically sorted codebook is given in Figure 1.2

word	encoding
a	100
b	110011
c	01101
d	11000
e	00
f	01100
g	0111
h	010
i	101
j	1100100
k	1100101
l	11100
m	11101
n	1101
o	1111

Table 1.2: Codebook sorted by word

(c) The codebook from part (b) but now sorted by the increasing lengths of the codewords.

The alphabetically sorted codebook is given in Figure 1.3

word	encoding
e	00
a	100
h	010
i	101
g	0111
n	1101
o	1111
c	01101
d	11000
f	01100
l	11100
m	11101
b	110011
j	1100100
k	1100101

Table 1.3: Codebook sorted by length of codeword and alphabetic order as tiebreaker

Problem 2: Optimum Binary Search Trees

You are given an alphabet A containing 8 characters (in alphabetical order) and their associated frequencies in the table below. Create an optimal binary search tree of A to minimize the expected time required to search for a character in this alphabet.

i	1	2	3	4	5	6	7	8
a_i	A	B	C	D	E	F	G	H
$f(a_i)$	5	8	1	6	4	7	3	2

Table 2.1: Frequency Distribution

(a) Present your result by showing the final $e[i, j]$ and $root[i, j]$ tables.

$[i, j]$	1	2	3	4	5	6	7	8
1	5	18	20	33	45	66	75	83
2	0	8	10	23	33	51	60	68
3	0	0	1	8	16	33	40	46
4	0	0	0	6	14	30	37	43
5	0	0	0	0	4	15	21	27
6	0	0	0	0	0	7	13	19
7	0	0	0	0	0	0	3	7
8	0	0	0	0	0	0	0	2

Table 2.2: $e[i, j]$ matrix

$[i, j]$	1	2	3	4	5	6	7	8
1	1	2	2	2	2	4	4	4
2	0	2	2	2	4	4	4	4
3	0	0	3	4	4	5	6	6
4	0	0	0	4	4	5	6	6
5	0	0	0	0	5	6	6	6
6	0	0	0	0	0	6	6	6
7	0	0	0	0	0	0	7	7
8	0	0	0	0	0	0	0	8

Table 2.3: $root[i, j]$ matrix

(b) What is the minimum search time cost of a BST for A?

The minimum cost is given by the top right element on $e[i, j]$ matrix, namely $e[1, 8]$ which yield 83

(c) Execute the "Construct-BST" algorithm. Write down your function calls and draw the final optimal Binary Search Tree Result.

The recursive function calls is given by Algorithm 1.

Algorithm 1: Function call of Construct-BST

```
Construct-BST(root, 1, 8)
  Construct-BST(root, 1, 3)
    Construct-BST(root, 1, 1)
      Construct-BST(root, 1, 0)
      Construct-BST(root, 2, 1)
    Construct-BST(root, 3, 3)
      Construct-BST(root, 3, 2)
      Construct-BST(root, 4, 3)
  Construct-BST(root, 5, 8)
    Construct-BST(root, 5, 5)
      Construct-BST(root, 5, 4)
      Construct-BST(root, 6, 5)
    Construct-BST(root, 7, 8)
      Construct-BST(root, 7, 6)
      Construct-BST(root, 8, 8)
        Construct-BST(root, 8, 7)
        Construct-BST(root, 9, 8)
```

An optimized BST is illustrated in Figure 2.1

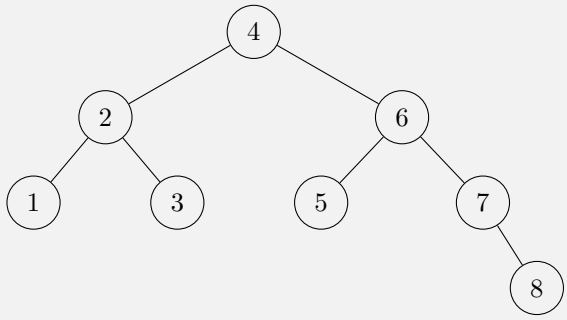

Figure 2.1: A Optimal Tree

Problem 3: Chain Matrix Multiplication

Recall that the cost of multiplying matrices together is the total number of scalar multiplications used.

Let $\text{cost}((A_1A_2)A_3)$ be the cost of first multiplying A_1A_2 and then multiplying their result by A_3, Let $\text{cost}(A_1(A_2A_3))$ be the cost of first multiplying A_2A_3 and then multiplying their result by A_1. In class we saw an example in which

$$\frac{\text{cost}(A_1(A_2A_3))}{\text{cost}((A_1A_2)A_3)} = 10$$

Multiplying in the "correct" order could reduce the total cost by 90%. For this problem, for every integer $t > 1$ define a quadruple $\left(p_0^{(t)}, p_1^{(t)}, p_2^{(t)}, p_3^{(t)}\right)$ such that, if A_i has dimensions $p_{i-1}^{(t)} \times p_i^{(t)}$ then,

$$\frac{\text{cost}(A_1(A_2A_3))}{\text{cost}((A_1A_2)A_3)} \geq t. \tag{1}$$

This shows that the gain of using the "correct" order is unbounded. Your solution should be in two parts:

(a) Give the formulas for $\left(p_0^{(t)}, p_1^{(t)}, p_2^{(t)}, p_3^{(t)}\right)$

The cost ratio is given by:

$$\frac{\text{cost}(A_1(A_2A_3))}{\text{cost}((A_1A_2)A_3)} = \frac{p_0p_1p_3 + p_1p_2p_3}{p_0p_1p_2 + p_0p_2p_3} = \frac{(p_0 + p_2)}{p_0p_2}\frac{p_1p_3}{(p_1 + p_3)} \geq t$$

The following conditions should suffice for the inequality to hold:

$$\begin{cases} \dfrac{(p_0 + p_2)}{p_0p_2} \geq 1 \\ \dfrac{p_1p_3}{(p_1 + p_3)} \geq t \end{cases} \iff \begin{cases} \dfrac{1}{p_0} + \dfrac{1}{p_2} \geq 1 \\ \dfrac{1}{p_1} + \dfrac{1}{p_3} \leq \dfrac{1}{t} \end{cases}$$

For the first inequality, we can take:

$$\frac{1}{p_0} + \frac{1}{p_2} \geq 1 \Leftarrow ((p_0 \leq 2) \wedge (p_2 \leq 2))$$

For the second inequality, we can take:

$$\frac{1}{p_1} + \frac{1}{p_3} \leq \frac{1}{t} \Leftarrow ((p_1 \geq 2t) \wedge (p_3 \geq 2t))$$

Note that $p_i \in \mathbb{Z}^+$. Thus, $\forall t > 1, t \in \mathbb{Z}^+$, we can take $\left(p_0^{(t)}, p_1^{(t)}, p_2^{(t)}, p_3^{(t)}\right)$ to be $(1, 2t, 1, 2t)$ such that the gain is greater or equal to t.

(b) Prove the correctness of align* 1 for your given values.

For $t = 2$, the gain is $4 \geq t$. In general, $\forall t > 1, t \in \mathbb{Z}^+$, the gain is $2t \geq t$.

(c) Given a chain of 7 matrices $\langle A_1, A_2, \ldots, A_7\rangle >$ and their dimensions (as provided in the table below) fully parenthesize the product $A_1A_2 \ldots A7$ in a way that minimizes the number of scalar multiplications required to multiply them all together

Matrix	A_1	A_2	A_3	A_4	A_5	A_6	A_7
Dimension	6×7	7×3	3×1	1×2	2×4	4×5	5×3

Table 3.1: Matrix Dimensions

(i) Present your result by showing the final $m[i, j]$ and $s[i, j]$ tables.

The m and s tables are shown below.

$m[i,j]$	1	2	3	4	5	6	7
1	0	126	63	75	95	121	124
2		0	21	35	57	84	85
3			0	6	20	43	52
4				0	8	28	43
5					0	40	70
6						0	60
7							0

Table 3.2: $m[i, j]$ table

$s[i,j]$	1	2	3	4	5	6	7
1		1	1	3	3	3	3
2			2	3	3	3	3
3				3	3	3	3
4					4	5	6
5						5	6
6							6
7							

Table 3.3: $s[i, j]$ table

(ii) What is the minimum number of scalar multiplications needed to compute the product $A_1 A_2 \dots A_7$?

The minimum number of scalar multiplication is given by the top right element of m matrix, namely $m[1, 7] = 124$.

(iii) Present the optimal parenthesized sequence for multiplying $A_1 A_2 \dots A_7$

The optimal parenthesization is given by:

$$\begin{aligned}
\left(A_{1..s[1,5]} A_{s[1,5]+1..5}\right) &= (A_{1..3} A_{4..7}) \\
&= ((A_1 (A_2 A_3)) (((A_4 A_5) A_6) A_7)) \\
&= ((A_1 A_{2..3}) A_{4..7}) \\
&= ((A_1 (A_2 A_3)) A_{4..7}) \\
&= ((A_1 (A_2 A_3)) (A_{4..6} A_7)) \\
&= ((A_1 (A_2 A_3)) ((A_{4..5} A_6) A_7)) \\
&= ((A_1 (A_2 A_3)) (((A_4 A_5) A_6) A_7))
\end{aligned}$$

Problem 4: More Optimum Binary Search Trees

In class we developed a $O(n^3)$ time solution for the optimal Binary Search tree problem. In this problem you need to modify that solution to construct an optimal tree of given restricted height. The height of a tree T is:

$$height(T) = \max_{1 \le t \le n} d(a_t)$$

where $d(a_t)$ is the depth of the node corresponding to a_t in tree T.

The optimal tree constructed by the algorithm could have height as large as $n-1$. One standard constraint that occurs in practice is to restrict the tree height. That is, given h, to find a tree T0 such that $height(T') \le h$ and

$$B(T') = \min \{B(T) \ : \ T \text{ is a BST satisfying } height(T) \le h\}$$

Do the following.

(a) For $1 \le i \le j \le n$ and $0 \le k \le h$, set

$$e[i, j : k] = \begin{array}{l}\text{the minimum cost of any BST } T \\ \text{on } a_i \ldots a_j \text{ with } height \le k\end{array} \tag{2}$$

Give a Dynamic Programming recurrence align for $e[i, j : k]$. Don't forget to provide the initial conditions.

The recurrence relation is given by: initialization:

$$\begin{aligned} w[i, j] &= \sum_{p=i}^{j} f(p) \\ e[i, i : 0] &= f(i) && \forall \{i, i \in [1, n]\} \\ e[i, j : 0] &= 0 && \forall \{(i, j), i > j\} \\ e[i, j : 0] &= \infty && \forall \{(i, j), i < j\} \end{aligned}$$

recurrence relation:

$$e[i, j : k] = \min_{i \le z \le j} e[i, z-1 : k-1] + e[z+1, j : k-1] + w[i, j] \qquad \forall \{k, 0 < k \le h\}$$

(b) Justify (prove) the correctness of the recurrence relation from part (a).

Base Case $k = 0$: $e[i, j : 0]$ is correct, as the only possible way to construct a tree with height 0 is by having the root as the only node in the tree. This $e[i, i : 0]$ matrix only has one diagonal nonzero and non-infinite entry at $i = j$.

Base Case $k = 1$: $e[i, j : 1]$ is correct. Observe that the matrix has similarities to the original $e[i, j]$. This matrix $e[i, i : 1]$ now has three nonzero and non-infinite matrix at $j = i$, $j = i + 1$, $j = i + 2$. The entries at $j = i$ yield $f[i]$, the entries at $j = i + 1$ yield $e[i, i + 1]$ from the original e matrix described in the class. The entries at $j = i + 2$ is restricted, as the only possible configuration to make a BST with three nodes and height one is by having one node as root, and two nodes as left and right children.

General case $k \geq 0$: assume the induction hypothesis $I(k')$: "$e[i, j : k']$ The algorithm correctly returns the minimum cost of any BST T on $a_i \dots a_j$ with $height < k$" is correct for all $k' < k$.
$e[i, j : k]$ return an optimal tree for with $height \leq k$. Then, the left and right subtree must be an optimal tree with $height \leq k - 1$.
From the class: the cost of tree $B(T_{i,j})$ with root z is given by:

$$B(T_{i,j}) = B(T_{i,z-1}) + B(T_{k+1,j}) + w[i, j]$$

Then, the optimal tree with $height \leq k$ is:

$$e[i, j : k] = \min_{i \leq z \leq j} e[i, z - 1 : k - 1] + e[z + 1, j : k - 1] + w[i, j]$$

By the induction hypothesis and the lemma, $e[i, j : k]$ correctly return the optimal tree on $a_i \dots a_j$ with $height < k$. The induction hypothesis $I(k)$ is true for all $k \geq 0$

(c) Give documented pseudocode for an $O(n^3 h)$ time algorithm to calculate $e[1, n : h]$

Algorithm 2: Initialize-Optimized-Cost-Matrix($f(a_i), n$)

```
let e[1..n, 1..n : 0..n − 1], w[1..n, 1..n] be new arrays of all 0
// initialize weight matrix
for i ← 1 to n do
    w[i, i] ← f(a_i) for j = i + 1 to n do
        w[i, j] ← w[i, j − 1] + f(a_j)
    end
end
// initialize e matrix for the upper right to be infinite except the
   diagonal
for i ← 0 to n do
    for j ← i to n do
        if i = j then
            e[i, j : 0] = f(a_i)
        else
            e[i, j : 0] = ∞
        end
    end
end
// dynamic programming for e matrix
for k ← 1 to n − 1 do
    for l ← 1 to n do
        for i ← 1 to n − l + 1 do
            j ← i + l − 1
            e[i, j : k] ← ∞
            // minimize the cost by considering the tree with smaller height
            for z ← i to j do
                t ← e[i, z − 1 : k − 1] + e[z + 1, j : k − 1] + w[i, j]
                if t < e[i, j : k] then
                    e[i, j] ← t
                end
            end
        end
    end
end
return e[1..n, 1..n : 0..n − 1]
```

(d) Explain why your algorithm runs in the required time.

The running time of computing $e[i, j : k]$, given that $e[i, j : k-1]$ is known is $O(n^3)$. Therefore, to compute $e[i, j : h]$ for $0 \leq k \leq n-1$, the total running time required is $O(n^3 h)$

Problem 5: Dynamic Programming

The input to this problem is a directed "triangle graph" $G = (V, E)$ with h levels. It contains $n = h(h+1)/2$ nodes:

$$V = \{(i, j) : i \in [1, h], j \in [1, i]\}$$
$$E = \bigcup_{\substack{(i,j)\in V \\ i<h}} [((i, j), (i + 1, j)), ((i, j), (i + 1, j + 1))]$$

Intuitively, node (i, j) on level $i < h$ points to two nodes on level $i+1$, $(i+1, j)$ to its left (L) and $(i+1, j+1)$ to its right (R).

Each node in the tree has a given positive integral value, $A[i, j]$

A path is a walk down the graph starting at $(1, 1)$: It is described by a length $j < h$ string $w_1, w_2, \ldots, w_j \in \{L, R\}^j$. The path starts at the top node $(1, 1)$ and, from level i it walks down to level $i + 1$ either by going LEFT (if $w_i = L$) or RIGHT (if $w_i = R$).

- The *value* of a path is the sum of the values on the path.
- A turn on a path is a L followed by an R or a R followed by an L. For example $LLLRRRRLRR$ has three turns.
- A path is balanced if the number of even integers on the path is equal to the number of odd integers on the path.

Each one of (a), (b), (c) should be split into the following five separate parts

(i) Give the recurrence upon which your DP algorithm is based.

(ii) Explain how, after filling in your table, you can use the information in the table to solve the stated problem.

(iii) Justify (prove) the correctness of the recurrence relation from part (i)

(iv) Give documented psuedocode for your algorithm.

(v) Explain why your algorithm runs in the required time.

Solve the following three problems using dynamic programming. The solutions for each of (a), (b) and (c) should start on a new page.

(a) Find the maximum value of a path in the graph. Your algorithm should run in $O(n)$ time.

(i) Basis of algorithm
Define the maximum value of a path ending at $[i, j]$ as $M[1..h, 0..h+1]$.

$$M[i, j] = \text{maximum value of a path ending at } i, j$$

Then, the maximum value path of that ends at level i (length $i-1$) is:

$$W^{(i)} = \max_{1 \le k \le i} \{M[i, k]\}$$

The recurrence of $M[i, j]$ is given by:

$$M[i, j] = \max\{(M[i-1, j]), (M[i-1, j-1])\} + A[i, j]$$

Initial Condition:

$$M[i, j] = -\infty \quad \text{initialization for all } i, j$$
$$M[1, 1] = A[1, 1]$$

(ii) Solving the problem
First note the increasing property of $M[i+1, j] > M[i, j]$, therefore the maximum turns should reach the leaves of the tree. To find the maximum value of a path in a graph, we need to find $W^{(h)} = \max_{1 \le k \le h} \{M[h, k]\}$

(iii) Proof of correctness

- **Base Case** $h = 1$: $W^{(1)} = M[1, 1]$ is correct as $M[1, 1] = A[1, 1]$ is the only element in the matrix.
- **Base Case** $h > 1$: For $h' < h$, consider the induction hypothesis $I(h')$: "M is the maximum value of a path ending at i, j". Then the only two possible path to (i, j) is by passing through vertex $(i-1, j)$ or $(i-1, j-1)$. The maximum value of the path ending at (i, j) maximum of $M[i-1, j] + A[i, j]$ or $M[i-1, j-1] + A[i, j]$. Therefore the induction hypothesis $I(h)$ is correct for all $h \ge 1$.
  The algorithm correctly return the maximum value of a path with length h, $W^{(h)} = \max_{1 \le k \le h} \{M[h, k]\}$.

(iv) Documented pseudocode

Algorithm 3: Find-Maximum-Value(A, h)

```
let M[1..h, 0..h+1] be new array of all −∞ and W = −∞
// Initialize the recurrence
M[1,1] ← A[1,1]
for i ← 2 to h do
    for j ← 1 to i do
        // Find minimum value for each node
        for z ← 0 to 1 do
            t ← M[i−1, j−z] + A[i,j]
            if t > M[i,j] then M[i,j] = t
        end
    end
end
for j ← 1 to h do
    t ← M[h,j]
    if t > W then W ← t
end
return W
```

(v) Running Time Analysis

The running time of filling the $M[i, j]$ for all i, j is $O(h(h+1)) = O(n)$. The running time of finding the maximum element is $O(h) = O(\sqrt{n})$. Therefore the total running time is $O(n)$

(b) Find the maximum value of a balanced path in the graph. Your algorithm should run in $O(n^{3/2})$ time.

(i) Basis of algorithm

Define the maximum value of a path ending at $[i, j]$ with p number of odd element as $M[1..h, 0..h+1 : -1..h]$.

$$M[i, j : p] = \text{maximum value of a path ending at } i, j \text{ with } p \text{ odd element}$$

Then, the maximum value path of that ends at level i (length $i-1$) with p odd element is:

$$W_p^{(i)} = \max_{1 \le k \le i} \{M[i, k : p]\}$$

The recurrence of $M[i, j]$ is given by:

$$M[i, j : p] = \begin{cases} \max\{(M[i-1, j : p]), (M[i-1, j-1 : p])\} + A[i, j] & \text{if } A[i, j] \text{ is even} \\ \max\{(M[i-1, j : p-1]), (M[i-1, j-1 : p-1])\} + A[i, j] & \text{if } A[i, j] \text{ is odd} \end{cases}$$

Initial Condition:

$$M[i, j : p] = -\infty \quad \text{initialization for all } i, j, p$$

$$M[1, 1 : 0] = \begin{cases} A[1, 1] & \text{if } A[1, 1] \text{ is even} \\ -\infty & \text{if } A[1, 1] \text{ is odd} \end{cases}$$

$$M[1, 1 : 1] = \begin{cases} -\infty & \text{if } A[1, 1] \text{ is even} \\ A[1, 1] & \text{if } A[1, 1] \text{ is odd} \end{cases}$$

(ii) Solving the problem

First note the increasing property of $M[i+1, j] > M[i, j]$, therefore the maximum turns should reach the leaves of the tree. To find the maximum value of a balanced path in a graph, assuming h is even, the number of odd element must be equal to the number of even element, $p = \lfloor h/2 \rfloor$, either on $i = h$ (if h is even) or $i = h-1$ (if h is odd). Therefore, we return $W^{(h)}_{\lfloor h/2 \rfloor} = \max_{1 \le k \le h} \{M[h, k, : \lfloor h/2 \rfloor]\}$ as the problem description requires.

(iii) Proof of correctness

Correctness of $M[i, j : p]$ matrix:

- **Base Case** $h = 1$: $M[1, 1 : 0..1]$ are correct as $A[1, 1]$ is either odd or even.
- **Base Case** $h > 1$: For $h' < h$, consider the induction hypothesis $I(h')$: "M is the maximum value of a path ending at i, j with p odd element". Then, the only two possible path to (i, j) is by passing through vertex $(i-1, j)$ or $(i-1, j-1)$.
  If $A[i, j]$ is even, then $M[i, j : p]$ is preceded with maximum path to either vertex $(i-1, j)$ or $(i-1, j-1)$ with exactly p odd elements. Otherwise, if $A[i, j]$ is odd, then $M[i, j : p]$ is preceded with maximum path to either vertex $(i-1, j)$ or $(i-1, j-1)$ with exactly $p-1$ odd elements.
  Therefore, the induction hypothesis $I(h)$ is correct for all $h \ge 1$.

Correctness of algorithm:

- If h is even, the algorithm will correctly return the maximum value balanced path, $W^{(h)}_{h/2} = \max_{1 \le k \le h} \{M[k, h : h/2]\}$
- If h is odd, the algorithm will correctly return the maximum value balanced path, $W^{(h)}_{\lfloor h/2 \rfloor} = \max_{1 \le k \le h} \{M[k, h : \lfloor h/2 \rfloor]\}$

Therefore, the algorithm is correct for all $h \in \mathbb{Z}$.

(iv) Documented pseudocode

Algorithm 4: Find-Maximum-Value-Balanced(A, h)

```
let M[1..h, 0..h+1 : −1..h] be new array of all −∞ and W = −∞
if A[1,1] is even then
|  M[1, 1 : 0] ← A[1, 1]
else
|  M[1, 1 : 1] ← A[1, 1]
end
for i ← 2 to h do
|  for j ← 1 to i do
|  |  for p ← 0 to i do
|  |  |  // Find maximum value for each cases
|  |  |  if A[i, j] is even then
|  |  |  |  // for even node, store to the same p
|  |  |  |  for z ← 0 to 1 do
|  |  |  |  |  t ← M[i − 1, j − z : p] + A[i, j]
|  |  |  |  |  if t > M[i, j : p] then M[i, j : p] = t
|  |  |  |  end
|  |  |  else
|  |  |  |  // for odd node, store to the p+1
|  |  |  |  for z ← 0 to 1 do
|  |  |  |  |  t ← M[i − 1, j − z : p − 1] + A[i, j]
|  |  |  |  |  if t > M[i, j : p − 1] then M[i, j : p] = t
|  |  |  |  end
|  |  |  end
|  |  end
|  end
end
// Get the optimal cost for balanced node
for j ← 1 to h do
|  if i is even then
|  |  t ← M[h, j : ⌊h/2⌋]
|  else
|  |  t ← M[h − 1, j : ⌊h/2⌋]
|  end
|  if t > W then W ← t
end
return W
```

(v) Running Time Analysis

For a particular (i, j), the running time of filling the $M[i, j : 0..i]$ is $O(i)$. The total running time of filling all $M[1..h, 0..h + 1, 0..h]$ is $O(h^3) = O(n^2)$. The running time of finding the maximum element is $O(h) = O(\sqrt{n})$. Therefore the total running time is $O(n^{3/2})$

(c) Find the maximum value of a path containing exactly $\lfloor h/2 \rfloor$ turns in the graph. Your algorithm should run in $O(n^{3/2})$ time.

(i) Basis of algorithm
Define the maximum value of a path ending at $[i, j]$ with $q \in [0..h-1]$ turns, and last direction as enumeration $r \in \{L, R\}$ as $M[1..h, 0..h : 0..h-2, \{L, R\}]$

$M[i, j : p, r]$ = maximum value of a path ending at i, j with q turns and last direction of r

Furthermore, define NOT operator on $\neg r$, $\neg(L, R) = (R, L)$

Then, the maximum value path that ends at level i (length $i-1$) with q turns:

$$W_q^{(i)} = \max_{\substack{1 \le k \le i \\ r \in \{L,R\}}} \{M[i, k : q, r]\}$$

The recurrence of $M[i, j]$ is given by:

$$M[i, j : q, r] = \max \begin{Bmatrix} (M[i-1, j : q, r]), & (M[i-1, j-1 : q, r]), \\ (M[i-1, j : q-1, \neg r]), & (M[i-1, j-1 : q-1, \neg r]) \end{Bmatrix} + A[i, j]$$

Initial condition:

$$\begin{aligned} M[i, j : q, r] &= -\infty \quad \text{initialization for all } i, j, q, r \\ M[1, 1 : 0, r] &= -\infty \quad \text{turns is not defined for } h = 1 \\ M[2, 1 : 0, L] &= A[1, 1] + A[2, 1] \\ M[2, 2 : 0, R] &= A[1, 1] + A[2, 2] \end{aligned}$$

(ii) Solving the problem
To find the maximum value of a path in the graph with $\lfloor h/2 \rfloor$ turns, we need to find $W^{(h)}_{\lfloor h/2 \rfloor} = \max_{\substack{1 \le i \le h, 1 \le j \le i \\ r \in \{L,R\}}} \{M[h, k : \lfloor h/2 \rfloor, r]\}$.

(iii) Proof of correctness
Correctness of $M[i, j : q, r]$ matrix:

- **Base Case** $h = 2$: $M[2, 1 : 0, L]$ and $M[2, 2 : 0, R]$ are correct trivially.
- **Base Case** $h > 2$: For $h' < h$, consider the induction hypothesis $I(h')$: "M is the maximum value of a path ending at i, j with q turns and last path of r". Then, the only two possible path to (i, j) is by passing through vertex $(i-1, j)$ or $(i-1, j-1)$. Furthermore, for each preceding vertex, the path could be pointed to the same or opposite direction to compared to the previous one. If the path is pointing to the opposite direction, then the number of count q increases.
  For a given $M[i, j : q, r]$ there are 4 possibilities of preceding path, which are:
  - Coming from left, previous direction from the left, $(M[i-1, j-1 : q, r])$
  - Coming from left, previous direction from the right, $(M[i-1, j-1 : q-1, \neg r])$
  - Coming from right, previous direction from the left, $(M[i-1, j : q-1, \neg r])$
  - Coming from right, previous direction from the right, $(M[i-1, j : q, r])$

  Therefore, $M[i, j : q, r]$ correctly record the maximum value between these four possibilities in addition to the its value $A[i, j]$. Therefore, the induction hypothesis $I(h)$ is correct for all $h \ge 2$.

Correctness of algorithm: The maximum value of a path is strictly increasing, the maximum value on the leaves of the tree. The algorithm correctly return the maximum value of a path which ends at the h level in the graph with $q = \lfloor h/2 \rfloor$ turns for any last direction r. Therefore, the algorithm is correct for all $h \ge 2$.

(iv) Documented pseudocode

Algorithm 5: Find-Maximum-Value-Turn(A, h)

```
let M[1..h, 0..h+1 : 0..h−1, L, R] be new array of all −∞ and W = −∞
// Initialize the recurrence
M[2, 1 : 0, L] ← A[1, 1] + A[2, 1]
M[2, 2 : 0, R] ← A[1, 1] + A[2, 2]
// Fill M matrix
for i ← 3 to h do
    for j ← 1 to i do
        for q ← 0 to h − 1 do
            // Find maximum value for each allowed configuration
            foreach r in {L, R} do
                foreach (i′, j′ : q′, r′) in [
                    (i − 1, j : q, r),
                    (i − 1, j : q − 1, ¬r),
                    (i − 1, j − 1 : q, r),
                    (i − 1, j − 1 : q − 1, ¬r)
                ] do
                    t ← M[i′, j′ : q′, r′] + A[i, j]
                    if t > M[i, j : q, r] then M[i, j : q, r] = t
                end
            end
        end
    end
end
// Get the maximum value for exactly ⌊q/2⌋ turns for all direction
for i ← 1 to h do
    for j ← 1 to i do
        foreach r in {L, R} do
            t ← M[h, j : ⌊h/2⌋, r]
            if t > W then W ← t
        end
    end
end
return W
```

(v) Running Time Analysis

For a particular (i, j) and any r, the running time of filling the $M[i, j : 0..h-1, r]$ is $O(i)$. The total running time of filling all $M[1..h, 0..h+1, 0..h-1, \{L, R\}]$ is $O(h^3) = O(n^2)$. The running time of finding the maximum element is $O(h^2) = O(n)$. Therefore the total running time is $O(n^{3/2})$

Problem 6: Edit Distance

In this problem you must describe a dynamic programming algorithm for the *minimum edit distance* problem.

Background: The goal of the algorithm is to find a way to transform a "source" string $x[1 \dots m]$ into a new "target" string $y[1 \dots n]$ using any sequence of operations, *operating on the source string from left to right.* The *copy* operation copies the first remaining character in the source string to the target string, and deletes it from the source string. The *insert* operation adds one character to the end of the current target string. The *delete* operation deletes the first remaining character from the source string.

Each of the operations has an associated cost, **C** for copy, **I** for insert, and **D** for delete. The cost of a given sequence of transformation operations is 10the sum of the costs of the individual operations in the sequence.

Given two sequences $x[1 \dots m]$ and $y[1 \dots n]$ and a given set of operation costs **C**, **D**, and **D**, the *minimum edit distance* from x to y is the cost of the *least expensive transformation* sequence that converts x to y.

(a) Set $X[i] = x[1..i]$ and $Y[j] = y[1..j]$. Define the cost matrix as:

$$D[i, j] = \text{minimum edit distance to transform } X[i] \text{ to } Y[j]$$

Give a Dynamic Programming recurrence align* for $D[i, j]$. Don't forget to provide the initial conditions.

The recurrence relation of the cost matrix $D[i, j]$ is given by:

$$D[i,j] = \begin{cases} D[i-1, j-1] + \mathbf{C} & \text{if } x[i] = y[j] \\ \min\{D[i-1, j] + \mathbf{D}, D[i, j-1] + \mathbf{I}\} & \text{if } x[i] \neq y[j] \end{cases}$$
$$\text{for } \{(i, j) : 0 \leq i \leq m, 0 \leq j \leq n\}$$

Initial condition:

$$\begin{aligned} &D[0, 0] = 0 \\ &D[i, 0] = i\mathbf{D} \quad \text{for } \{(i, j) : 0 \leq i \leq m, 0 \leq j \leq n\} \\ &D[0, j] = j\mathbf{I} \end{aligned}$$

(b) Justify (prove) the correctness of the recurrence relation from part (a).

- **Base Case** $i = 0$ **or** $j = 0$:
  - Suppose the source string is an empty string. Then the minimum edit distance is the number of **I** to create the target string. Therefore, the algorithm correctly return the minimum edit distance for $D[0, j]$ for all $j \geq 0$.
  - Suppose the target string is an empty string. Then the minimum edit distance is the number of **I** to delete all source string. Therefore, the algorithm correctly return the minimum edit distance for $D[i, 0]$ for all $j \geq 0$.

- **General Case** $i > 0$: Consider the induction hypothesis $I(i', j')$: "$D[i', j']$ yield the minimum edit distance to transform $X[i']$ to $Y[j']$", for $i' < i$ and $j' < j$.
  - If $x[i] \neq y[j]$ and $\min\{D[i-1, j] + \mathbf{D}, D[i, j-1] + \mathbf{I}\} = D[i-1, j] + \mathbf{D}$: Then, it is required to call **D**, which add **D** additional cost.
  - If $x[i] \neq y[j]$ and $\min\{D[i-1, j] + \mathbf{D}, D[i, j-1] + \mathbf{I}\} = D[i, j-1] + \mathbf{I}$: Then, it is required to call **I**, which add **I** additional cost.
  - If $x[i] = y[j]$ Then, by the given relation $\mathbf{C} < \mathbf{D} + \mathbf{I}$, Then, it is required to call **C**, which add **C** additional cost.

  Therefore, $D[i, j]$ yield the minimum edit distance to transform $X[i]$ to $Y[j]$ for all i, j. $I(i, j)$ is true for $0 \leq i \leq m, 0 \leq j \leq n$

(c) Give documented pseudocode for an $O(mn)$ time algorithm to calculate $D[m, n]$. Your documentation should make clear what each section is doing.

Algorithm 6: Minimum-Cost-Matrix(x, y, m, n)

```
// Initialize empty array
let D[0..m, 0..n] be an array of 0 for all element
// Initialize base case for recurrence
D[0, 0] = 0
for i ← 1 to m do
|  D[i, 0] ← iD
end
for j ← 1 to n do
|  D[0, j] ← jI
end
// Get the recurrence for general case
for i ← 0 to m do
|  for j ← 0 to n do
|  |  if x[i] = y[j] then
|  |  |  D[i, j] = D[i − 1, j − 1] + C
|  |  else
|  |  |  D[i, j] = min{D[i − 1, j] + D, D[i, j − 1] + I}
|  |  end
|  end
end
// Return D
return D[1..m, 1..n]
```

(d) Explain why your algorithm runs in the required time.

The initialization takes $O(n + m)$, and the recurrence will iterate through $i \in [0, m]$ first and then $j \in [0, n]$. The total running time is $O((m + 1)(n + 1)) = O(mn)$.

Problem 7: Greedy Algorithms

Give an $O(n \log n)$ algorithm for assigning location of n files on a magnetic tapes that minimizes the expected time necessary to read a file. As input you are given the size of the files and the probability that each file will be accessed.

Suppose $L[i]$ is the size of file i. The time T_i it takes to read file i is the sum of the sizes of the files located before file i plus the size of file i. Suppose that you know that file i will be accessed with probability p_i and the ordering in which the files are on the tape. The expected (average) time to read a file will be:

$$\sum_i p_i T_i$$

Input is an array of $L[1..n]$ of the n files sizes and a list $p_1, p_2, \ldots, p_n$ of the probability of accessing each file. The problem is to design an $O(n \log n)$ algorithm that outputs an optimal ordering of the files on the tape, i.e., an ordering that minimizes the expected time to read a file. To do this answer the following questions:

(a) First assume that all the files are equally likely, i.e., for all files $p_i = 1/n$. Prove that in this case, inserting the files in order of increasing size gives an optimal solution.

Define the average cost C as: $C = \sum_i p_i T_i = \sum T_i / n$.

Let the ordering of the files on the tape as $X[1..n] = \{i \mid i \in [1, n] \wedge X[i] = X[j] \iff i = j\}$.

We can define the recurrence relation of T_i as:

$$\begin{aligned} T[X[i]] &= T[X[i-1]] + L[X[i]] \quad \text{for } i > 1 \\ T[X[1]] &= L[X[1]] \end{aligned}$$

The cost average cost is:

$$C = \sum_i T[i]/n = \sum_i T[X[i]]/n = \frac{1}{n} \sum_i (n - i + 1) * L[X[i]]$$

Therefore, to minimize the average cost, we need $L[X[i]] \leq L[X[i+1]]$, as smaller i yield larger weight in the sum. In other word, the ordering must be sorted in increasing order. One can use $O(n \log n)$ (stable) sorting algorithm such as merge sort to achieve this, by sorting $X[i]$ with key $L[X[i]]$.

(b) Show that in the general case, sorting the files by size and inserting them on the tape in order of increasing size does not have to give an optimal solution.

Suppose one file is short but will never or rarely get accessed, then it is better to store this file in the last entry rather than the first entry.

For example suppose $L[i] = [2, 4, 6]$ with probability $p_i = [0, 0.5, 0.5]$.

If we store in sorted order, $X[i] = [1, 2, 3]$ the total cost is $C = 9$. If suppose we store 1 at the end, $X[i] = [2, 3, 1]$ the total cost is: $C = 7$ which is smaller than previous ordering.

(c) Describe a rule that does solve the problem.

First, consider a case contrasting (a), where all $L[i] = 1$ for all i. Then, $X[i]$ must be sorted in descending order or the item that is least likely to be accessed placed in the last position. In other word, can propose to sort the output array by $1/p_i$ if the length of the all input is the same.

For general case, we can consider a greedy algorithm that combines both properties by sorting by $L[i]/p_i$. A correctness proof will be given in the following sub problem.

(d) Provide a formal proof that the rule that you gave in part (c) outputs an optimal ordering.

Suppose we sort the array by the key of $L[X_i]/p_{X_i}$, such that $L[X_i]/p_{X_i} \leq L[X_j]/p_{X_j}$ for all $i < j$. Then consider i, j where $L[i]/p_i < L[i]/p_j$. Then, there are two possible way to arrange these two elements:

$$C = \begin{cases} L[i]p_i + (L[i] + L[j])p_j \\ L[j]p_j + (L[j] + L[i])p_i \end{cases}$$
$$= \begin{cases} p_i L[i] + p_j L[j] + L[i]p_j & (*) \\ p_i L[i] + p_j L[j] + L[j]p_i \end{cases}$$

Note that since $L[i]/p_i < L[i]/p_j$, then the minimum cost is the first case (marked with $(*)$).

Then, note that we can "combine" $L[i..j]/p_{i..j}$ to extend the prove.

To prove the correctness, we can use this fact to check the first two sorted elements in the tape is optimal, and treat the first two elements as one element and compare it with the third element to prove the first three elements are optimal, and repeat this process to prove for all elements.

(e) Explain how this gives you an $O(n \log n)$ greedy algorithm for solving the problem.

We need to sort the array by the key of $L[i]/p_i$. Suppose we use merge sort to sort the entries, the running time of the sorting is $O(n \log n)$ time.

Problem 8: Greedy Agorithms

Consider a long river, along which n houses are located. You can think of this river as an x-axis; the house locations are given by their coordinates on this axis in a sorted order. The inputs are $x_1 < x_2 < \cdots < x_n$:

Your company wants to place cell phone base stations along the river, so that every house is within 10 kilometers of one of the base stations.

(a) Give an $O(n)$-time algorithm greedy algorithm that minimizes the number of base stations used.

Provide both documented pseudocode and an explanation in words as to what it does.

The algorithm can be described as "Start from x_1, place a tower at maximum distance to the house at $x_1 + 10km$, then find the next house that is not in the coverage of any tower x_i, and place a tower in $x_i + 10$, and repeat until all houses are in coverage"

Algorithm 7: Place-Cell-Phone-Base($x_1 \dots x_n$, $r = 10km$)

```
let i ← 1, count ← 0
while true do
    place a tower at x_i + r
    count ← count + 1
    if x_i + 2 * r > x_n then
        // All towers is covered
        return count
    end
    for j ← i to n do
        if x_j > x_i + 2 * r then
            // Find next tower which is not covered
            i ← j
            break
        end
    end
end
```

(b) Provide a formal proof of your algorithm's correctness, i.e., that it outputs a minimal size solution.

(This must be formal. Make every assumption used explicit and justify every step in your argument. We will deduct points for ambiguity.)

Claim: Greedy Solution is optimal.

Proof. Suppose greedy solution is different from OPT. Let k be any number between $[1, n]$. Suppose OPT placement is the same as greedy for $i \leq k$. Suppose the next tower is placed differently at $x_t \neq x_i + r$, then there are different possibilities:

- $x_t < x_i + r$, as all previous tower is already covered, moving the tower to $x_i + r$ is also optimal.
- $x_t > x_i + r$, then x_i is not covered, therefore this configuration is not permitted.

Therefore, we can move the next tower to $x_i + r$ and the solution is still optimal. By repeating this method until the last tower, the optimal solution will be the same as greedy solution. Therefore greedy solution is an optimal solution too.

Problem 9: Dynamic Programming

Now consider a modification of the previous problem. You still have a long river on which house locations are given by their coordinates on this axis in a sorted order. The inputs are still $x_1 < x_2 < \cdots < x_n$.

Your company still wants to place cell phone base stations along the river to cover every house. The difference is that you now have t different types of base stations. A base station of type i costs c_i dollars to install and can cover all houses within k_i kilometers of its location. A base station of type i located at y is denoted by pair (y, i).

A house at location x is covered by base station (y, i), if $x \in [y - k_i, y + k_i]$.

A set

$$S = \{(y_1, i_1), (y_2, i_2), \ldots, (y_s, i_s)\}$$

is a cover of the input, if it covers all of the houses, i.e.,

$$\{x_1, \ldots, x_n\} \subset \bigcup_{j=1}^{s} \left[y_j - k_{i_j}, y_j + k_{i_j}\right]$$

The *cost* of a set of base stations is $cost(S) = \sum_{j=1}^{s} c_{i_j}$: Design an $O(tn)$ dynamic programming algorithm to find the cheapest (i.e., least expensive) cost of a set of base stations that covers all of the houses.

(a) Give the recurrence upon which your DP algorithm is based. Before stating the recurrence, define (using English and math symbols) the meaning of each entry. Don't forget to completely specify the initial conditions of your recurrence relation.

Define the minimum cost of covering the first i houses as $M[i]$. The recurrence relation for $M[i]$ is given by:

$$M[i] = \min\{c_{p_i} + M[j]\} \quad \text{where } \{j \geq j' \; \forall (j' < i) \wedge x_j \not\subset [x_i - 2 * k_{p_i}, x_i]\}$$

The initial condition is given by:

$$M[0] = 0 \quad x_0 = -\infty$$

The recurrence relation can be described as:

- Initialize the recurrence by assuming there is a house at $x_0 = -\infty$ which does not have to be covered $M[0] = 0$.
- For every next house located at x_i, assume a tower of type p will be placed at $x_i - k_{p_i}$ which will cover $[x_i - 2 * k_{i_j}, x_i]$.
- Add the previous optimal cost of the last house that cannot be covered by the placed tower $M[j]$.
- Add minimum cost for $M[i]$ to the table, which is the minimum of $c_{i_p} + M[j]$.
- Repeat until the table is fully filled.

Next, we define a relational matrix $R[i : p]$ which will indicate the farthest house after i can be covered by using a single tower of type p.

$$R[i : p] = \min\{j \ldots R[i + 1]\} \quad \text{where } j \leq j' \; \forall (j' < i) \wedge x_i - x_j \leq 2 * k_p$$

The initial condition is:

$$R[n : p] = n$$

There are some important properties of the relational matrix, which are:

- If the tower placed in i cannot cover $j < i$, then it will not be able to cover $j' < j$.
- If a tower placed in $i + 1$ can cover $j < i$, then a tower at i must be able to cover j too.

These properties will be used later to initialize the R matrix within $O(tn)$ time.

(b) Explain how, after filling in your table, you can use the information in your table to solve the stated problem.

First, suppose we can fill the relational matrix $R[i, j : p]$ within the $O(tn)$ number of comparison. The $M[i]$ matrix can also be filled within $O(tn)$ time (will be elaborated later).

Then the minimum cost to cover all n houses is $M[n]$.

(c) Prove the correctness of the recurrence relation from part (a). (This must be formal. Make every assumption used explicit, and justify every step in your argument. We will deduct points for ambiguity.)

Prove of correctness for $M[i]$.

- Base case $i = 0$: Correct by definition.
- Base case $i = 1$: Then $j = 0 \iff M[j] = 0$ for all p. The minimum cost to cover the first tower only is the minimum cost of the tower of type p.
- General case $i > 1$: Consider the strong induction hypothesis $I(i')$ that the M matrix is correct for all $i' < i$. Then, the optimal cost to cover the first i house $M[i]$.
  The optimal cost to cover the first i house is the minimum cost of covering the first j house $M[j]$ (from induction hypothesis) and covering $j+1$ to i house with a tower of type p.
  Suppose the last tower placed is type p, which cover $j + 1$'th to i'th house. Then, the optimal cost *if the last tower is type* p is $M[j_p] + c_{p_i}$. As we have t type of tower, the optimal cost is the minimum of $M[j_p] + c_{p_i}$ for all p.
  Thus the induction induction hypothesis is true for $I(i)$ for all $i > 0$ and the algorithm return the correct answer.

(d) Give documented psuedocode for your algorithm.

Documented pseudocode for initializing $R[i:p]$ matrix, running time $O(tn)$

Algorithm 8: Initialize-Relational-Matrix($x_1..x_n, k_1..k_t$)

```
let R[1..n : 1..t] be a matrix of all zeros
for p ← 1 to t do
    for i ← n down to 1 do
        // A tower must be able to cover one house
        R[i : p] ← i
        // If house at i+1 can cover house j < i, then a house at i can
           cover it too.
        j ← R[i + 1 : p]
        while j ≥ 0 and x_i − x_j ≤ 2 ∗ k_p do
            // If i cannot cover a house at j < i, then any house before j
               cannot be covered too
            R[i : p] ← j
            j ← j − 1
        end
    end
end
return R[1..n : 1..t]
```

Documented pseudocode for initializing $M[i]$ array and getting the optimal cost, running time $O(tn)$

Algorithm 9: Get-Optimal-Cost($x_1..x_n, k_1..k_t, c_1..c_t$)

```
R[1..n : 1..t] ← Initialize-Relational-Matrix(x_1..x_n, k_1..k_t)
let M[0..n] be a matrix of all zeros
for i ← 1 to n do
    M[i] ← ∞
    for p ← 1 to t do
        // get the previous house which cannot be covered by i for using p
           tower
        j ← R[i : p] − 1
        // the cost is the sum of the cost to cover j+1 to i plus the
           optimal cost to cover j towers
        t ← c_p + M[j]
        if t < M[i] then
            M[i] ← t
        end
    end
end
// return the optimal cost for all houses
return M[n]
```

(e) Explain why your algorithm runs in the required time.

The algorithm can be broken down into two parts. The first one is initializing the relational array $R[1..n : 1..t]$, which takes $O(tn)$ time. There are three loop blocks inside the algorithm, but actually the the while loop inside only do $O(2n)$ operations in total for all i iterations because the properties we mentioned in part (a). Therefore, the total running time for initializing $R[1..n : 1..t]$ is $O(t(n + 2n)) = O(tn)$

The second one is initializing the minimum cost array $M[1..n]$ which also takes $O(tn)$ time. There are two loop blocks in the algorithm so the total running time must be $O(tn)$.

Therefore the total running time of the algorithm is also $O(tn)$.