(1)

Problem 8-5. Hidden Palindromes [35 points]

Cayson wants to send hidden messages to his friends Ack and Dirk, via Derrit, an online public message board. Cayson will post to the message board a seemingly random string of lowercase letters, containing a secret message within a palindrome subsequence. A **palindrome** is any string which is the same string when the order of its characters is reversed. String B is a **subsequence** of a string A if the letters of B appear in order in A, possibly with other letters interspersed. Given an online message board post A, Cayson's secret message B will be the **first half** of the longest palindrome subsequence of A. If the longest palindrome subsequence has an odd number of characters, the secret message will include the middle character. You may assume that the longest palindrome subsequence of one of Cayson's online message board posts is unique. For example, if Cayson's message board post is `"nweeyaoslsoitmarawtuitfsjaipdiwi"`, the (unique) longest subsequence palindrome is `"wasitaratisaw"`[2], with the secret message being `"wasitar"`[3].

(a) [10 points] Given a message board post from Cayson containing n characters, describe an $O(n^2)$-time algorithm to decode it and return Cayson's secret message.

(b) [25 points] Implement the decoding algorithm described above in a Python function `decode_message`. You can download a code template containing some test cases from the website. Submit your code online on **Dropbox**.

Subproblems, LPS: Longest Palindromic Subsequence. Define $T(i,j)$: the "half" length of LPS in the substring $A[i:j]$. Here "half" implies that every pair of occurrence of a character in the palindrome is counted as one unit.

Relate, Say S is the LPS in $A[i:j]$ of length l. If $A[i]=A[j]$, then these two characters must be a part of S, otherwise we can make a longer palindrome by including them. Therefore now it suffices to to check the LPS in $A[i+1:j-1]$. The half length of this LPS must be $l-1$ after excluding the repeating occurrence $A[i]$.

If not then S must completely lie in either $A[i+1:j]$ or $A[i:j-1]$. Since in any case the excluded end point is not a part of S, S is indeed a LPS of half length l in one of the two substrings. From these observations we can quickly form the following recurrence,

$$T(i,j)=\begin{cases} 1+T(i+1,j-1) & \text{, if } A[i]=A[j] \\ \max\{T(i+1,j),\ T(i,j-1)\} & \text{, otherwise} \end{cases}$$

Here to maintain some book keeping we will introduce $pi(i,j)$ such that,

$$\pi(i,j)=\begin{cases} (i+1,j-1) & \text{, if } A[i]=A[j] \\ (u,v) & \text{, where } (u,v) \text{ maximizes } T(i,j) \text{ when } A[i]\neq A[j] \end{cases}$$

In each of the relations values larger than i and smaller than j are used in obtaining the ij-th component of the memo table, and hence the relations are acyclic.

Base cases, $T(i,j)=0$ for $i>j$, as there is no possible substring $A[i:j]$. And $T(i,i)=1$ as a single character is a palindrome of half length 1.

Solution, $T(1, n)$ holds the length of the secret message in the entire string. To obtain the string itself we would run a separate loop using the parent memo π. We will initialize an empty string S and start with $p(1, n)$ to recursively set (i, j) to $\pi(i, j)$ until we extract the LPS. If at any stage $A[i] = A[j]$ then it must be a part of the LPS, and to construct the secret message we add one of the characters, say $A[i]$, to the string S. We will stop when the beginning index i surpasses the ending index j, and accordingly return S as the secret message.

Run time, there are n^2 subproblems for each of T and π, as $1 \leq i, j \leq n$. Plus each subproblem is calculated by using a constant number of operations, or in $O(1)$ time. The loop to extract the LPS S runs at most the length of S times and hence is upper bounded by n. Therefore the total run time of the algorithm is $O(n^2) + O(n) = O(n^2)$.

(2)

Problem 8-4. [10 points] **Bottle Breaker**

The hit new videogame *Green Zombie Atonement II* features a new minigame, where you can throw balls at a line of bottles, each labeled with a number (positive, negative, or zero). You can throw as many balls at the bottles as you like, and if a ball hits a bottle, it will fall and shatter on the ground. Each ball will either hit no bottle, exactly one bottle, or two bottles (but only when the two bottles were **adjacent in the original lineup**). If a ball hits two adjacent bottles, you receive a number of points equal to the product of the numbers on the bottles. Otherwise, if a ball hits zero or one bottle, you do not receive any points. For example, if the line of bottle labels is $(5, -3, -5, 1, 2, 9, -4)$, the maximum possible score is 33, by throwing two balls at bottle pairs $(-3, -5)$ and $(2, 9)$. Given a line of bottle labels, describe a efficient dynamic-programming algorithm to maximize your score.

Subproblems, Let $[p_1, p_2, \cdots, p_n]$ be the list of bottle labels on indices $1 \leq k \leq n$. Denote $T(k)$: maximum score possible in the sub-list $[p_1, p_2, \cdots, p_k]$. Clearly then the solution to the problem is $T(n)$.

Relate, if $T(k)$ is as defined then there are two options in regard to p_k. First, p_k had a contribution to the score $T(k)$. Then according to the rules its adjacent bottle p_{k-1} was also hit. And now, to calculate $T(k)$ we can look at the remainder of the chunk $[p_1, \cdots, p_{k-2}]$, obtain the maximum score possible, and add to it $p_{k-1} \cdot p_k$.

If p_k had no contribution to the score then for $T(k)$ just look at the maximum score obtained from $[p_1, \cdots, p_{k-1}]$. Using this observations we can quickly form the recurrence relation,

$$T(k) = \max\{T(k-1),\ \ p_k \cdot p_{k-1} + T(k-2)\}$$

Clearly values smaller than k are involved in calculation of $T(k)$, and hence the above relations are acylic.

Base case, As mentioned in the question there are no points for hitting a single bottle, so the maximum score $T(1)$ on the list $[p_1]$, $T(1) = 0$. Also, in our definition, $T(2) = \max\{T(1),\ \ p_1 \cdot p_2 + T(0)\}$, but as $T(2) = \max\{0, p_1 \cdot p_2\}$ we must have $T(0) = 0$ for consistency (which is intuitive as well).

Run time, as for $T(k)$, $0 \leq k \leq n$, the number of subproblems in $O(n)$. Each subproblem requires obtaining the maximum of two numbers and is hence done is $O(1)$ time. Accordingly, the run time of the algorithm is $O(1) \cdot O(n) = O(n)$.

(3)

Problem 8-3. Circleworld Politics [15 points]

Circleworld, an essentially one-dimensional circular nation surrounding a sun, is home to n residents (where n is odd). There are two political parties in Circleworld: the Maryonettes and the Itnizks, with the Maryonettes currently in power. Each resident is a member of one of these two parties and always votes for their party in any election. Mary Jander, the leader of the Maryonettes, lives at address 1 in Circleworld. Each other resident lives at a unique consecutively increasing integer address starting at Mary's house going around the circular world in one direction, so that Mary's next door neighbor in that direction lives at address 2, and her other next door neighbor in the opposite direction lives at address n.

Circleworld is currently divided into d districts that are each **odd-contiguous**. A district is odd-contiguous if it contains an odd number of residents who live at a contiguous sequence of addresses along the circular world. For example, Mary currently lives in district 1, comprising 7 residents who live at addresses $(n-1, n, 1, 2, 3, 4, 5)$. When an election is held, the residents in each district cast a vote for their political party, and the party receiving the majority of votes within a district wins that district. Being the leader of the party in power, Mary is allowed to completely reassign the d districts in Circleworld prior to the upcoming election, under the condition that each of the new d districts is odd-contiguous, and that every resident is assigned to exactly one district. Given the address and political party of each resident in Circleworld, describe an efficient dynamic-programming algorithm to help Mary reassign districts so as to maximize the number of districts her party can win in the upcoming election.

Preliminaries, denote $L = [p_1, \cdots, p_n]$ be a "circular" list such that p_i denotes the voting preference of resident in address i. Define,

$$p_i = \begin{cases} 1 & \text{if } i \text{ votes Mayonettes} \\ 0 & \text{otherwise} \end{cases}$$

Let $L_{ij} = [p_i, \cdots, p_j]$ is the sublist that traverses from index i to index j in the clockwise direction. Notice that this is possible as the original list is circular and hence if $i > j$ then $L_{ij} = [p_i, \cdots, p_n, p_1 \cdots p_j]$. So,

$$|L_{ij}| = \begin{cases} j - i + 1 & \text{if } j \geq i \\ j - i + 1 + n & \text{if } j < i \end{cases} \quad \text{or, } |L_{ij}| = 1 + [(j-i) \mod n]$$

Now to each L_{ij} we will associate a value V_{ij} defined according to the following rule,

$$V_{ij} = \begin{cases} -\infty & \text{if } 2 \mid |L_{ij}| \\ 1 & \text{if } 2 \nmid |L_{ij}| \text{ and } \sum_{k=i}^{j} p_i \geq \left\lceil \frac{j-i+1}{2} \right\rceil \\ 0 & \text{otherwise} \end{cases}$$

In the above definition V_{ij} denotes a measure of potential win or loss of the Mayonettes if L_{ij} was a district.

Subproblems, define $T(i, j, l)$: maximum number of districts that Mayonettes can win if Mary divides L_{ij} into l valid districts. Clearly, $1 \leq i, j \leq n$, and $l \leq d$. Since the list L is circular then final partition into the districts can begin from any arbitrary position. Therefore the final solution we will be obtained as T-value for d districts corresponding to that position i which maximizes T, i.e. $T(i, i-1, d)$.

Relate, Let P be the optimal partition of L_{ij} into l districts. Clearly the partitions start from address i. The districts being contiguous, the district starting at i must therefore end at some index k, maintaining that L_{ik} is a valid district. From here, after having decided the first district Mary can look for partitions of $L_{k+1,j}$ into $l-1$ districts. In particular, Mary must look at the optimal partition from this set, as doing otherwise would break the optimality of P.

Now, V_{ik} would be the score of Mary from the district L_{ik}, and $T(k+1, j, l-1)$ would be Mary's remaining score from $L_{k+1,j}$. If L_{ik} is not a valid district then Mary's total score would remain $-\infty$ and therefore would not be chosen by Mary. As Mary is trying to maximize the number of districts she wins, we can accordingly form the following recurrence,

$$T(i,j,l) = \max_{\substack{i \le k \le j \\ \text{(in circular order)}}} \{V_{ik} + T(k+1, j, l-1)\}$$

Values smaller than l are used for $T(i,j,l)$, we can can proceed the calculation in ascending order of l, i.e. by filling in the entire row $T(i,j,l-1)$ for all i,j and going to l, and hence the above relation is acyclic.

Base case, Firstly the score possible from L_{ij} if it in itself is a district is by definition V_{ij}, and hence $T(i,j,1) = V_{ij}$. Otherwise the finest valid partition that can be made on L_{ij} is if every district is a single address (therefore odd) of this partition. Or on L_{ij} at most $|L_{ij}|$ districts can be made. Therefore, $T(i,j,l) = 0$ whenever $l > |L_{ij}|$.

Run time, we will individually calculate the run time of the parts of the algorithm.

- For any given i, j, $|L_{ij}|$ can be calculated in constant time.
- For V_{ij} we are required to sum through all elements of L_{ij}, with some additional $O(1)$ operations. But $|L_{ij}| \le n$, and hence this part takes $o(n)$ time. Since there are n^2 possible (i,j) pairs, the first two steps require $O(n^3)$ time.
- For the DP part as $1 \le i,j \le n$, and $1 \le l \le d$, the number of subproblems is dn^2. Each subproblem involves maximizing over $|L_{ij}|$ many elements, and therefore takes $O(n)$ time as $|L_{ij}| \le n$. So the DP component takes $dn^2 \cdot O(n) = O(dn^3)$ time.
- We finally obtain the solution by obtaining the maximum among n numbers, i.e. $T(i, i-1, d)$ in $O(n)$ time.
- Hence the total run time of the algorithm is $O(n^3) + O(dn^3) + O(n) = O(dn^3)$.

(4)

Problem 8-2. Treasureship! [10 points]

The new boardgame Treasureship is played by placing 2×1 ships within a $2 \times n$ rectangular grid. Just as in regular battleship, each 2×1 ship can be placed either horizontally or vertically, occupying exactly 2 grid squares, and each grid square may only be occupied by a single ship. Each grid square has a positive or negative integer value, representing how much treasure may be acquired or lost at that square. You may place as many ships on the board as you like, with the score of a placement of ships being the value sum of all grid squares covered by ships. Design an efficient dynamic-programming algorithm to determine a placement of ships that will maximize your total score.

Subproblems, Let $B = \{b_{uv} \mid 1 \leq p \leq 2, 1 \leq v \leq n\}$ be the original game board, and let B_{ij} for $j \in \{i-1, i, i+1\}$ be the subboard with i boxes in the top row and j in the bottom. Therefore this board consists of $i+j$ boxes out of the original $2n$ boxes. Now define, $T(i,j)$: maximum score possible on B_{ij}. Clearly the required solution is $T(n,n)$.

Relate, Observe that there are three configurations possible for B_{ij}, and we will define a recursion for each. Observe that $|i-j| \leq 1$. Consider the following cases,

1. If the top row has an extra box, i.e., $j = i-1$, then we can either place a horizontal ship at the edge the top row, or discard the extra box can consider the subboard $B_{j,j}$.
2. The bottom row having an extra box, i.e. $j = i+1$, is a mirror image of the previous case. We either place a horizontal ship at the edge of the bottom row or discard the extra box to look at $B_{i,i}$.
3. Otherwise if $i = j$, then we place a vertical ship at the edge of grid, and consider the subboard $B_{i-1,i-1}$. Otherwise, we either skip a board in the top or the bottom row and look at the subboard $L_{i,i-1}$ or $L_{i-1,i}$ respectively.

We can accordingly form the following recurrence to calculate $T(i,j)$,

$$T(i,j) = \begin{cases} \max\{b_{1,i} + b_{1,i-1} + T(i-2, i-1),\ T(i-1, i-1)\ \} & \text{if } j = i-1 \\ \max\{b_{2,i} + b_{2,i+1} + T(i, i-1),\ T(i,i)\ \} & \text{if } j = i+1 \\ \max\{b_{1,i} + b_{2,i} + T(i-1, i-1),\ T(i-1, i),\ T(i, i-1)\ \} & \text{if } j = i \end{cases}$$

Clearly smaller values of i and j are involved in calculation of $T(i,j)$ and hence the above relation is acyclic.

Base case, $T(i,j) = 0$ if $i + j < 2$, not enough space to place a ship.

Run time, As $1 \leq i \leq n$, and $j \in \{i-1, i, i+1\}$ the total number of subproblems is bounded by $3n$. Observe the at each subproblem we identify the case it belongs to and then calculate the maximum of either 2 or 3 quantities; and therefore this can be achieved in $O(1)$ time. Hence the total run time of the algorithm is $3n \cdot O(1) = O(n)$.

(5)

Problem 8-1. [30 points] **Tree Counting**

How many heaps, binary search trees, and AVL trees are there on n distinct integer keys? In this problem, you will describe an algorithm to compute each of these counts. Instead of analyzing each algorithm's running time, please report the asymptotic number of **arithmetic operations** performed by your algorithm (additions, subtractions, multiplications, and divisions).[1]

(a) [10 points] Recall, a **Binary Search Tree** is a binary tree with a key at each node, satisfying the BST property. There are 5 binary search trees on the $n = 3$ distinct keys $\{12, 0, 5\}$. Describe a dynamic-programming algorithm to compute the number of different binary search trees on a given set of n distinct integer keys, using at most $O(n^2)$ arithmetic operations.

(b) [10 points] Recall, an **AVL Tree** is a binary tree with a key stored at each node, satisfying both the BST and AVL properties. There is only 1 AVL tree on the $n = 3$ distinct keys $\{12, 0, 5\}$. Describe an dynamic-progamming algorithm to compute the number of different AVL trees on a given set of n distinct integer keys, using at most $O(n^2 \log n)$ arithmetic operations.

(c) [10 points] Recall, a **Binary Min Heap** is a left-justified, complete binary tree with a key at each node, satisfying the min-heap property. There are 2 binary min heaps on the $n = 3$ distinct keys $\{12, 0, 5\}$. Describe a dynamic-programming algorithm to compute the number of different binary min heaps on a given set of n distinct integer keys, using at most $O(n)$ arithmetic operations.

Let the i-th smallest distinct key be represented as the number i, for $1 \leq i \leq k$. The position of a key in a BST, AVL or min-heap is determined by its relative order with the other keys in the set. The representation suggested is valid as it keeps the order of the original keys intact.

(a) Subproblems, define T_k : the number of BSTs with k keys. Therefore required solution is T_n.

Relate, Observe that there is a BST with these keys that has i in its root. This tree inevitably has $i-1$ keys in its left subtree, and $k-i$ in its right. Each of these subtrees are BSTs in itself, with $i-1$ and $k-i$ keys respectively. Clearly, for each configuration of a left subtree there are T_{k-i} right subtrees. Therefore the total number of BSTs with i in the root is given by $T_{i-1} \cdot T_{k-i}$. Since each i is a valid candidate as a root of the BST, the total number of BSTs with k keys is obtained using the following recurrence,

$$T_k = \sum_{i=1}^{k} T_{i-1} \cdot T_{k-i}$$

Only values smaller than k are used in the calculation of T_k and hence the above relation is acyclic.

Base cases, $T_1 = 1$ is the trivial BST with a single key. However from the above relation we also have $T_1 = {T_0}^2 = 1$, implying that $T_0 := 1$ must be defined for consistency.

Run time, One can see in the recurrence that for a given n there are n subproblems, and the time required for each subproblem is again $O(n)$, as it involves calculating the sum of $n-1$ quantities. Accordingly, the total run time of the algorithm is $O(n^2)$.

(b) Preliminaries, Let h be the height of an AVL tree with k nodes. We will first obtain a lower bound and an upper bound, m_k and M_k, for the values of h.

First of all, a binary tree of height h holds at most $\sum_{i=0}^{h} 2^i = 2^{h+1}-1$ keys, implying $2^{h+1}-1 \geq k$, or $h+1 \geq \log_2(k+1)$, implying $m_k = \lfloor \log_2(k+1) \rfloor - 1$.

Also, if N_h denotes the minimum number of keys in an AVL trees of height h, we must have $N_h = 1 + N_{h-1} + N_{h-2}$, and particularly $N_h > N_{h-1} + N_{h-2}$. In particular at least one key is required to make an AVL tree of height 0 and two are required to make one of height 1, implying, $N_0 = 1 = F_2, N_1 = 2 = F_3$. From the nature of the above equation one can conclude that $N_h > F_{h+2}$, where the right hand side is the $(h+2)$-nd Fibonacci number.

By definition $F_{h+2} = (\phi^{h+2} - \psi^{h+2})/\sqrt{5} > (\phi^{h+1})/\sqrt{5}$, since ϕ is the golden ratio, and $\psi < 0$ is it's conjugate. Clearly, $k > N_h$, therefore, $k > F_{h+2} > (\phi^{h+2})/\sqrt{5}$, or $h+2 < (\log k + \log\sqrt{5})/\log\phi$. But as $(\log\sqrt{5}/\log\phi) < 2$, we can conclude $h \leq \lfloor \log k/\log\phi \rfloor$, and accordingly define $M_k = \lfloor \log_2 k/\log_2\phi \rfloor$.

In particular observe that the number of integers from m_k to M_k is given as,

$$M_k - m_k = \lfloor \frac{\log_2 k}{\log_2 \phi} \rfloor - \lfloor \log_2(k+1) \rfloor + 1 < \left(\frac{1}{\log_2 \phi} - 1 \right) \cdot \log k + 1 \approx 0.44 \log k + 1$$

Subproblems, define $T(k)$: the number of AVL trees with k keys, and $T(k,h)$: the number of AVL trees with k keys of height h. Clearly the required solution is $T(n)$.

Relate, the skew of the root of an AVL tree of height h lies in the set $\{-1,0,1\}$, and therefore the possible configurations for its left subtree and right subtree is limited to the pairs $(h-1,h-2),(h-1,h-1)$ and $(h-2,h-1)$.

Note that one of the subtrees must have a height of $h-1$ for the original tree to be of height h. Also, if the left subtree contains $t-1$ keys, the right is ought to contain $k-t$ keys, thereby maintaining the total keys at $t-1+k-t+1=k$. But t must also satisfy $1 \leq t \leq k$, as the size of left subtree ranges between 0 and $k-1$. Moreover, both the left and right subtrees satisfy AVL properties themselves. Therefore, one can combine the mentioned observations to form the following recurrence for $T(k,h)$,

$$T(k,h) = \sum_{t=1}^{k} \left\{ \begin{array}{rl} & T(t-1,h-1) \cdot T(k-t,h-2) \\ + & T(t-1,h-1) \cdot T(k-t,h-1) \\ + & T(t-1,h-2) \cdot T(k-t,h-1) \end{array} \right\}$$

But, counting AVL trees with k keys involves counting trees of all possible heights h. Since we have already obtained suitable bounds for h, one can quickly for a recurrence for $T(k)$ as,

$$T(k) = \sum_{h=m_k}^{M_k} T(k,h) = \sum_{h=m_k}^{M_k} \sum_{t=1}^{k} \left\{ \begin{array}{rl} & T(t-1,h-1) \cdot T(k-t,h-2) \\ + & T(t-1,h-1) \cdot T(k-t,h-1) \\ + & T(t-1,h-2) \cdot T(k-t,h-1) \end{array} \right\}$$

Clearly again values involved in calculation of $T(k,h)$ are strictly less than k and h, and hence the above relation is acyclic.

Base case, $T(1,0)=1$, but, $T(1,0)=2\cdot T(0,-1)\cdot T(0,-2)+[T(0,-1)]^2=1$, or $T(0,-1)=1$ $[T(h,k)\geq 0$ always$]$, and hence $T(0,-1)=1$ must be set for consistency. Clearly, $T(k,h)=0$ for $k<1$ or $h<m_k$ except when $(k,h)=(0,-1)$.

Run time, Clearly the calculation of $T(n,h)$ involves roughly $3n$ subproblems, with $O(3n)$ time per subproblem. Later for obtaining $T(n)$ the number of subproblems is the sixe of range of h, i.e. M_n-m_n+1. Therefore the total run time of the algorithm is,

$$3n\cdot O(3n)(M_n-m_n+1)=O(n^2)(0.44\log_2 n+2)=O(n^2\log_2 n)$$

(c) Preliminaries, let $l(k)$ and $r(k)$ denote the number of keys in the left subtree and right subtree respectively of a min heap with k keys. First observe that $l(k)+r(k)+1=k$.

Observe that if the min heap of k keys has height $h=h(k)$ it must follow that, $2^h-1<k\leq 2^{h+1}-1$, or $h\leq\log_2(k+1)\leq h+1$, or $h=\lceil\log_2(k+1)\rceil-1$. In particular, if the number of keys in the deepest level is $d=d(k)$, then $d=k-2^h+1$. From here there are two possibilities - first, the left subtree is complete, second, the left subtree is incomplete.

Say the left subtree is complete, i.e. $d\geq 2^{h-1}$. Here it must contain the same number of keys as that of the left subtree of a complete binary tree of height h. Therefore we have, $l=0.5\cdot\left(2^{h+1}-2\right)=2^h-1$.

Otherwise, if the left subtree is incomplete, then we will first calculate the number of keys in the left subtree of a complete binary tree of height $h-1$. Additionally all the elements of the deepest level would be a part of this subtree, and therefore we will have, $l=2^{h-1}-1+d$. And hence,

$$l(k)=\begin{cases}2^{h(k)}-1 & ,d(k)\geq 2^{h(k)-1}\\ 2^{h(k)-1}-1+d(k) & ,\text{otherwise}\end{cases}$$
$$\text{And, } r(k)=k-l(k)-1$$

Subproblems, define $T(k)$: the number of min heaps that can be formed with k keys. The solution to the problem is therefore $T(n)$.

Relate, A min heap with of k elements must have the smallest element 1 at the root. We have calculated that the right and left subtree contain $l(k)$ and $r(k)$ keys respectively. Now, since all the remaining $k-1$ elements are larger than 1, any l of them can be chosen to construct the left subtree, putting the remaining in the right. There are $\binom{k-1}{l}$ such choices. Now, each subtree are min heaps in themselves of size l and r. Therefor, $T(k)$ can be related to the subproblems as,

$$T[k]=\binom{k-1}{l(k)}\;T[l(k)]\cdot T[r(k)]$$

As $l(k),r(k)<k$, the above recurrence is acyclic.

Base case, $T(1) = 1$, clearly. We also know that $T(2) = 1$, as there is a unique min heap with 2 elements. But, $l(2) = 1, r(2) = 0$. Hence $1 = T(2) = \binom{1}{1} \cdot T(1) \cdot T(0)$, implying $T(0) = 1$ must be defined for consistency.

Run time, the smallest element 1 among the n elements can be identified in $O(n)$ time by traversing all the elements once and updating a "minimum" variable. The binomial coefficients can be efficiently calculated in constant time by either using pascal's identity, or by creating a memo table for factorials, i.e. for $k \le n$ recursively calculate $k!$ as,

$$k! = k \cdot (k-1)!$$

After the calculation for all $k \le n$ in $O(n)$ time the coefficients can be calculated in constant time using values from the memo table.

Finally, the values of $h(n), d(n), l(n), r(n)$ all require only a constant number of operations. And finally there are n subproblems with $O(1)$ operation per subproblem. Hence the total run time of the algorithm is $O(n) + O(n) + O(n) + O(1) = O(n)$.