1 Dynamic Programming (60 points)

1.1 Basic Case (12 points)

Suppose that we have an $n * n$ matrix filled with integers. Starting from the top left corner, we advance to either the downward or rightward block for each step and finally reach the bottom right corner of the matrix. During this process, we will pass through nodes with different integers. By applying dynamic programming, we can find the path with the largest sum of the passed integers. Write out the recurrence relation.

Solution:

$$\text{dp[i][j]} = \max(\text{dp[i-1][j]}, \text{dp[i][j-1]}) + \text{integers[i][j]},$$

where dp stores the maximum sum of value until we reach $(i,\ j)$.

1.2 Do it twice? (24 points)

Previously, we just go across the matrix for a single time. Assume that after the first travel, we set the visited integer in the matrix to be 0 and go across the matrix, back to the left corner again. How to maximize the sum of the passed integers in the whole procedure (top left -> bottom right -> top left)? In terms of this problem, William and Coned have different ideas again.

1.2.1 Simple repeat (10 points)

Coned thinks that since this problem is quite similar to what we have solved in 1.1, just run dynamic programming twice and add them up, and we will have the correct final result. Do you agree with him? If agree, write the recurrence relation for the second dynamic programming procedure and state whether there is a difference; if not agree, come up with a counter example and explain why it doesn't work.

Solution: No, he is wrong. If we just run dynamic programming twice, we will only obtain the optimal solution for each single procedure. Counterexample:

```
0 2 3
1 5 0
0 7 0
```

If we run dynamic programming twice for this matrix, we will first have $0 \to 2 \to 5 \to 7 \to 0$ as the path with the largest sum of value and then $0 \to 0 \to 3 \to 0 \to 0$ and the result of this solution is 2+5+7+3=17. However, we have $0 \to 1 \to 5 \to 7 \to 0$ and $0 \to 0 \to 3 \to 2 \to 0$ to be a possible solution, which have 1+5+7+2+3=18>17. It indicates that by running dynamic programming twice, we cannot make sure that the obtained solution is optimal.

Notes: actually by running dynamic programming, we just obtain two "partial greedy" solutions, which means that they are the greedy solutions for those single steps. However, summing partial greedy solutions doesn't guarantee you with a greedy solutions from the perspecitve of the whole procedure.

1.2.2 Double the result table (14 points)

William thinks that the dynamic programming strategy for this problem should be modified from the very beginning in this case. He gives out the main function as shown below. However, after testing, he found out there are some problems within this piece of code. Please correct the code between line 17 to line 26.

```
1  int main(){
2      // two paths are considered simultaneously, one at (i, j), the other at (k, l)
3      // integers stored in integer[n][n]
4      int n;
5      cin >> n;
6      int dp[100][100][100][100] = {0};
7      int integers[100][100] = {0};
8      // read all the inputs
9      for (int i = 0; i < n; i++)
10         for (int j = 0; j < n; j++)
11             cin >> integers[i][j];
12
13     dp[0][0][0][0] = integers[0][0]; // start point
14
15     // start dp
16
17     /* modify code within this part */
18     for (int i = 0; i <= n; i++)
19         for (int j = 0; j <= n; j++)
20             for (int k = 0; k <= n; k++)
21                 for (int l = 0; l <= n; l++){
22                     if (i == 0 && k == 0) dp[i][j][k][l] = dp[i][j-1][k][l-1];
23                     else if (i == 0 && l == 0) dp[i][j][k][l] = dp[i][j-1][k-1][l];
24                     else if (j == 0 && k == 0) dp[i][j][k][l] = dp[i-1][j][k][l-1];
25                     else if (j == 0 && l == 0) dp[i][j][k][l] = dp[i-1][j][k-1][l];
26                     else dp[i][j][k][l] = max(dp[i-1][j][k-1][l], dp[i][j-1][k-1][l],
27                                               dp[i-1][j][k][l-1], dp[i][j-1][k][l-1])
28                     dp[i][j][k][l] += integers[i][j]+integers[k][l];
29                     if (i == k && j == l) dp[i][j][k][l] -= integers[i][j];
30     /* modify code within this part */
31
32     cout << dp[n][n][n][n];
33
34 }
```

Notes: changes have been marked as red. Four changes: traverse upperbound, boundary cases, recursive relation, extra process for special case where two points have the same coordinate.
Here dp[n-1][n-1][n-1][n-1] and dp[n][n][n][n] are equal since all the entries are initialized as 0. No deduction if you don't change the traversing upperbound, since I just want to remind you that IO specification is also important.

1.3 How to save memory? (12 points)

Coned takes a look at William's strategy and thinks that its memory usage is too bad. Regardless of correctness, it will have a space complexity of $O(n^4)$, which sounds horrible. Propose a modification to this dynamic programming algorithm so that the space complexity can be reduced to $O(n^3)$ and write out its recurrence relation.
Hint: one way to do so is to reduce the array dp to be 3-dimensional. There should be 1 dimension still for i and 1 dimension still for k.

Solution:
dp[step][i][k]=
max(dp[step-1][i][k],dp[step-1][i-1][k-1],dp[step-1][i-1][k],dp[step-1][i][k-1])
+integers[i][step-i]+integers[k][step-k]

Notes: from observation we can know that for n^{th} step, if we know one dimension of the coordinate, then we can calculate the other one accordingly. Therefore, we can decrease the dimension of dp array from 4 to 3. Notice that if you want to implement dynamic programming with such a setting, you should carefully deal with the special cases. Further detail will be included in my RC recording.

1.4 Does optimization end here? (12 points)

Looking at the modification proposed by Coned, William surprisingly agrees with him. After brainstorming, they find that this strategy can be further optimized in terms of space complexity. Briefly state how you can further reduce its space complexity to $O(2n^2)$.
Hint: Do we need every *dp* value in every iteration?

Solution:
Since we do not need every *dp* value during our enumeration, (for example, when step = 10, we will only use dp[9][i][k] and dp[step][i][k] with step <= 8 will never be used any more later) we do not need to keep the result for every step. Just 2 2-dimensional array is needed: dp[0][i][k] and dp[1][i][k].

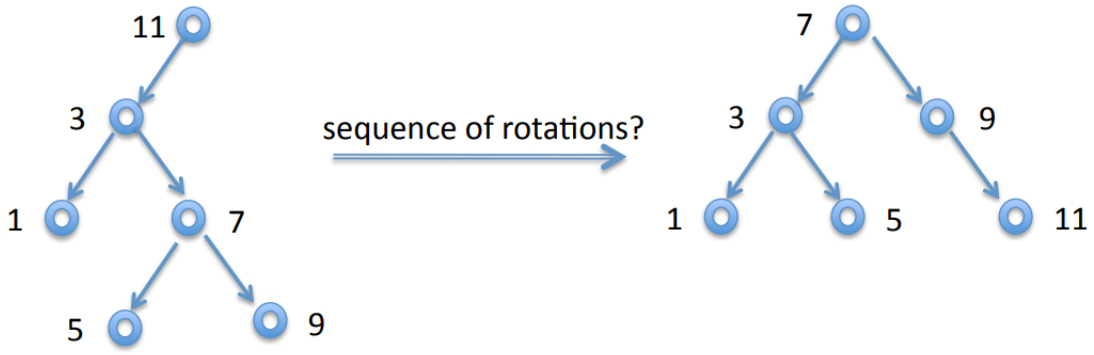
Notes: this solution can be called as a "rolling array" since for each step (we still need to record what is the current step although no dp array dimension is dedicated to do so), we calculate dp[1][i][k] and assign it to dp[0][i][k] for next step.
Notes: Actually we can further reduce its space usage from $O(2n^2)$ to $O(n^2)$ in this problem.
Further notes on the black notes: typically in this problem, as shown in 1.2, we may enumerate from 0 to n. However, based on the solution in 1.4, if we enumerate **from n to 0** for i and k (still 1 to $2n$-2 for step), it is even not necessary to have 2 2-dimensional arrays!

2 Self-balanced tree (40 points)

2.1 AVL Tree (20 points)

2.1.1 Rotate! (6 points)

Give the shortest sequence of single left or right rotations, and the two nodes being rotated, to transform the tree on the left to the tree on the right. Indicate in the rightmost column if the tree **after** the rotation satisfies the balance invariant of the AVL trees and also write out the balance factor of the **root** after rotation.
Notice that there might be unnecessary rotations, but make sure your sequence exactly matches the provided final result. You don't need to fill in all the rows in the table.

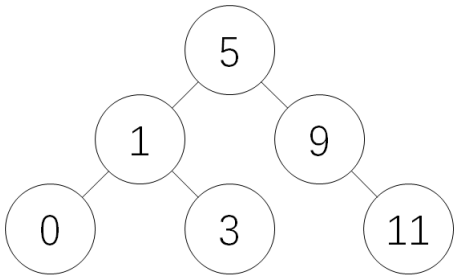


Step	Left or Right?	at nodes	balance factor	AVL?
1	left	3,7	3	no
2	right	7,11	0	yes
3	right	9,11	0	yes
4				
5				

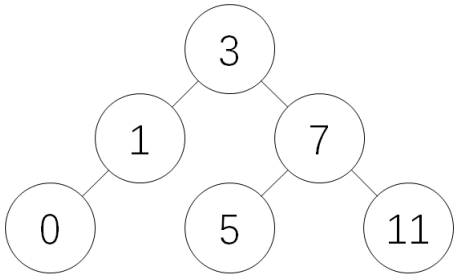
2.1.2 Simulate! (8 points)

Based on the balanced tree in 2.1.1, give a value whose insertion will trigger a left-left rotation. If such a value does not exist, choose a non-leaf node to delete and again give such a value whose insertion will trigger a left-left rotation. Draw your final tree no matter whether a deletion takes place.
Assume that when deleting a non-leaf node, we always use the largest node in the left subtree to replace the deleted node.

Solution:
Delete node 7 and insert 0:



Or delete node 9 and insert 0:



2.1.3 Code! (6 points)

In the lecture slides, the implementation of `Balance()` is given. In this problem, please implement `LLRotation` as well as `LRRotation` to make `Balance()` work properly. Suppose that `RRRotation` and `RLRotation` have been implemented and we follow the definitions in the slides.

```
void LLRotation(node *&n) {
    node * new_parent = n -> left;
    n->left = new_parent->right;
    new_parent->right = n;
}
void LRRotation(node *&n) {
    RRRotation(n->left);
    LLRotation(n);
}
```

2.2 Red Black Tree (20 points)

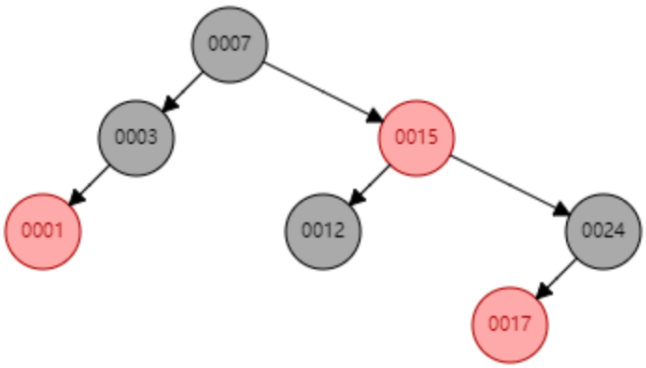
2.2.1 Balance? (4 points)

Briefly explain why red black tree is to some extent balanced.
Hint: what is the characteristic of a "balanced" tree?

Solution: We have proved in the lecture(at least in the slides) that by path rule and red rule, every red-black tree with n nodes has height $\leq 2\log_2(n+1)$ and since we will visit each level at most once for the basic operations, their time complexity is $log(n)$, which means this binary search tree is balanced, at least to some extent.

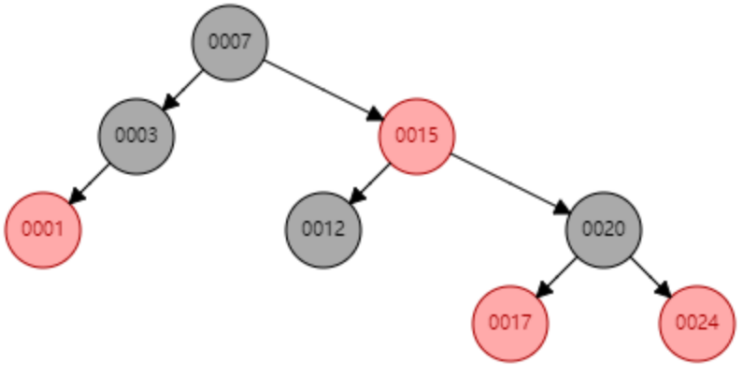
2.2.2 Complex? (8 points)

Given such a red black tree:



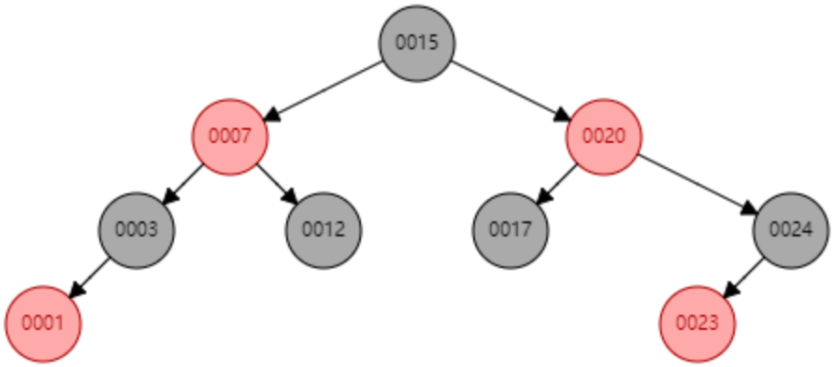
1) Show the result after inserting 20. Color the nodes clearly.

Solution:



2) After 1), show the result after inserting 23. Color the nodes clearly.

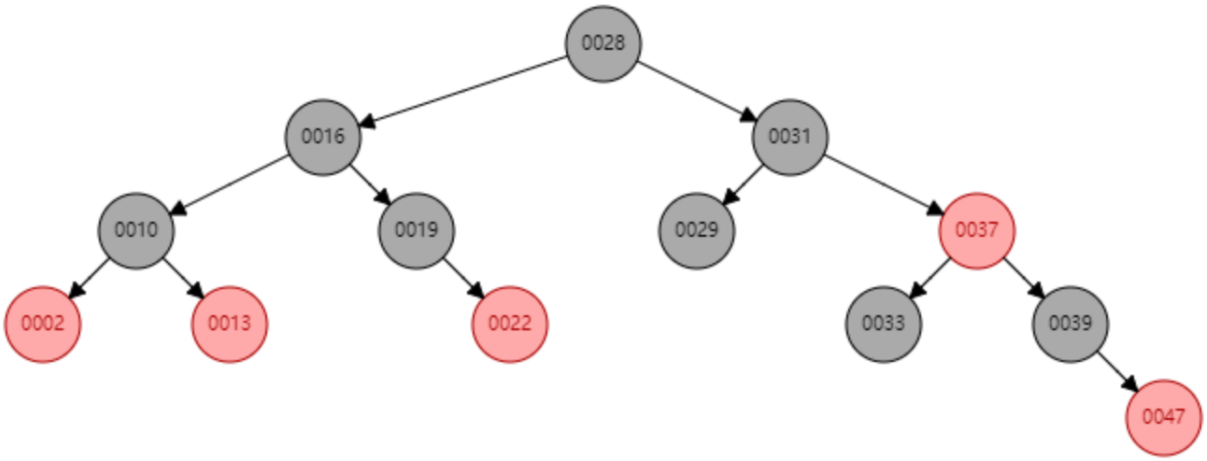
Solution:



2.2.3 Correctness? (8 points)

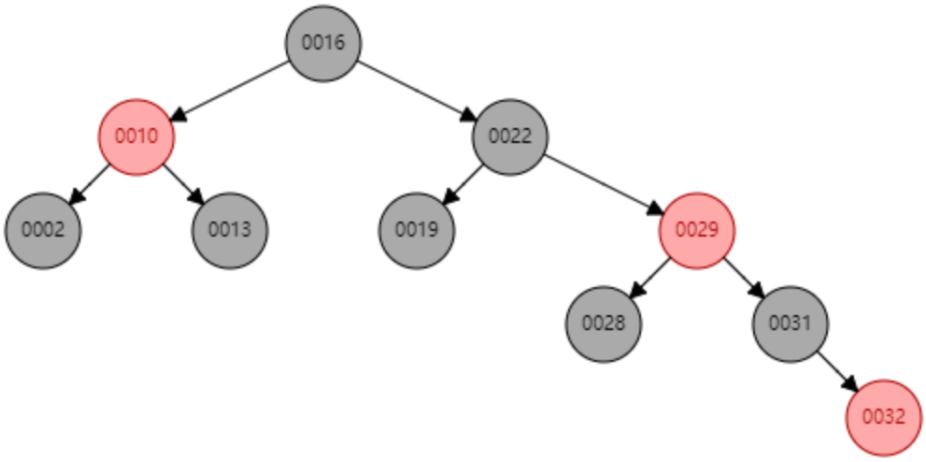
Judge whether the following red black tree is correct. If not correct, please point out which node is wrong and whether a recolor or rotation is needed (no need to draw the tree after fixing it).

a)



Solution: This tree is correct.

b)



Solution: This tree is incorrect. Recoloring node 10 to be black can fix the error.

Notes: In fact, although red black tree has a wide range of good application scenarios, its code implementation is quite complicated compared with other balanced tree such as AVL tree. It is also a reason why this assignment doesn't ask you to write related code in terms of red black tree.

Reference

Homework 6, 15-122, Carnegie Mellon University
Exam 2, 15-122, Carnegie Mellon University