$O(n)$ runtime, $O(n)$ space – Minor space optimization:

If a new element is larger than the current minimum, we do not need to push it on to the min stack. When we perform the pop operation, check if the popped element is the same as the current minimum. If it is, pop it off the min stack too.

```java
class MinStack {
   private Stack<Integer> stack = new Stack<>();
   private Stack<Integer> minStack = new Stack<>();

   public void push(int x) {
      stack.push(x);
      if (minStack.isEmpty() || x <= minStack.peek()) {
         minStack.push(x);
      }
   }

   public void pop() {
      if (stack.pop().equals(minStack.peek())) minStack.pop();
   }

   public int top() {
      return stack.peek();
   }

   public int getMin() {
      return minStack.peek();
   }
}
```

40. Evaluate Reverse Polish Notation

Code it now: https://oj.leetcode.com/problems/evaluate-reverse-polish-notation/ Difficulty: Medium, Frequency: Low

Question:

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are +, -, *, /. Each operand may be an integer or another expression.

Some examples:

["2", "1", "+", "3", "*"] -> ((2 + 1) * 3) -> 9

["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6

Example Questions Candidate Might Ask:

Q: Is an empty array a valid input?
A: No.

Solution:

The Reverse Polish Notation (RPN) is also known as the postfix notation, because each operator appears *after* its operands. For example, the infix notation “3 + 4” is expressed as “3 4 +” in RPN.

The brute force approach:

We look for the simplest RPN sequence that could be evaluated immediately, that is: Two successive operands followed by an operator such as “4 2 +”. We replace this sequence with the expression’s value and repeat until there is only one operand left.

For example,

[“4”, **“13”, “5”, “/”**, “+”]

➔ **[“4”, “2”, “+”**]

➔ [“6”]

How would you do the replacement? You could do it in-place with the input array, but it would result in quadratic runtime as elements have to be shifted every time a replacement occurs.

A workaround is to copy the input array to a doubly linked list. The replacement is efficient and the next scan begins with the replaced value’s previous element. This results in an algorithm with linear runtime and linear space. Although this works, the implementation is complex and is far from ideal in an interview session.

The optimal approach:

Observe that every time we see an operator, we need to evaluate the last two operands. Stack fits perfectly as it is a Last-In-First-Out (LIFO) data structure.

We evaluate the expression left-to-right and push operands onto the stack until we encounter an operator, which we pop the top two values from the stack. We then evaluate the operator, with the values as arguments and push the result back onto the stack.

For example, the infix expression "8 – ((1 + 2) * 2)" in RPN is:

8 1 2 + 2 * –

Input	Operation	Stack	Notes
8	Push operand	8	
1	Push operand	1 8	
2	Push operand	2 1 8	
+	Add	3 8	Pop two values 1, 2 and push result 3
2	Push operand	2 3 8	
*	Multiply	6 8	Pop two values 3, 2 and push result 6
–	Subtract	2	Pop two values 8, 6 and push result 2

After the algorithm finishes, the stack contains only one value which is the RPN expression's result; in this case, 2.

```
private static final Set<String> OPERATORS =
      new HashSet<>(Arrays.asList("+", "-", "*", "/"));

public int evalRPN(String[] tokens) {
   Stack<Integer> stack = new Stack<>();
   for (String token : tokens) {
      if (OPERATORS.contains(token)) {
         int y = stack.pop();
         int x = stack.pop();
         stack.push(eval(x, y, token));
      } else {
         stack.push(Integer.parseInt(token));
      }
   }
   return stack.pop();
}

private int eval(int x, int y, String operator) {
   switch (operator) {
      case "+": return x + y;
      case "-": return x - y;
      case "*": return x * y;
      default:  return x / y;
   }
}
```

Further Thoughts:

The above code contains duplication. For example, if we decide to add a new operator, we would need to update the code in two places – in the set's initialization and the switch statement. Could you refactor the code so it is more extensible?

You are probably not expected to write this refactored code during an interview session. However, it will make you a stronger candidate if you could make this observation and point this out, as it shows to the interviewer that you care about clean code.

In Java, create an interface called Operator and map each operator string to an implementation of the Operator interface. For other languages such as C++, each operator will be mapped to a function pointer instead.

```
interface Operator {
   int eval(int x, int y);
}

private static final Map<String, Operator> OPERATORS =
      new HashMap<String, Operator>() {{
            put("+", new Operator() {
               public int eval(int x, int y) { return x + y; }
            });
            put("-", new Operator() {
               public int eval(int x, int y) { return x - y; }
            });
            put("*", new Operator() {
               public int eval(int x, int y) { return x * y; }
            });
            put("/", new Operator() {
               public int eval(int x, int y) { return x / y; }
            });
      }};

public int evalRPN(String[] tokens) {
   Stack<Integer> stack = new Stack<>();
   for (String token : tokens) {
      if (OPERATORS.containsKey(token)) {
         int y = stack.pop();
         int x = stack.pop();
         stack.push(OPERATORS.get(token).eval(x, y));
      } else {
         stack.push(Integer.parseInt(token));
      }
   }
   return stack.pop();
}
```

41. Valid Parentheses

Code it now: https://oj.leetcode.com/problems/valid-parentheses/ Difficulty: Easy, Frequency: High

Question:

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(]" and "([)]" are not.

Example Questions Candidate Might Ask:

Q: Is the empty string valid?
A: Yes.

Solution:

To validate the parentheses, we need to match each closing parenthesis with its opening counterpart. A Last-In-First-Out (LIFO) data structure such as stack is the perfect fit.

As we see an opening parenthesis, we push it onto the stack. On the other hand, when we encounter a closing parenthesis, we pop the last inserted opening parenthesis from the stack and check if the pair is a valid match.

It would be wise to avoid writing multiple if statements when matching parentheses, as your interviewer may think that you are writing sloppy code. You could use a map, which is more maintainable.

```
private static final Map<Character, Character> map =
      new HashMap<Character, Character>() {{
            put('(', ')');
            put('{', '}');
            put('[', ']');
      }};

public boolean isValid(String s) {
   Stack<Character> stack = new Stack<>();
   for (char c : s.toCharArray()) {
      if (map.containsKey(c)) {
         stack.push(c);
      } else if (stack.isEmpty() || map.get(stack.pop()) != c) {
         return false;
      }
   }
   return stack.isEmpty();
}
```

Chapter 8: Dynamic Programming

42. Climbing Stairs

Code it now: https://oj.leetcode.com/problems/climbing-stairs/ Difficulty: Easy, Frequency: High

Question:

You are climbing a staircase. It takes *n* steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Solution:

***O*(*n*) runtime, *O*(1) space – Dynamic programming:**

This is a classic Dynamic Programming problem.

Define:

$f(n)$ = number of ways you can climb to the nth step.

To reach to the n^{th} step, you have only two choices:

1. Advance one step from the $n - 1^{th}$ step.
2. Advance two steps from the $n - 2^{th}$ step.

Therefore, $f(n) = f(n - 1) + f(n - 2)$, which is the exact same recurrence formula defined by the Fibonacci sequence (with different base cases, though).

Set base cases $f(1) = 1$, $f(2) = 2$ and you are almost done.

Now, we could calculate f(n) easily by storing previous values in an one dimension array and work our way up to n. Heck, we can even optimize this further by storing just the previous two values.

```java
public int climbStairs(int n) {
    int p = 1, q = 1;
    for (int i = 2; i <= n; i++) {
        int temp = q;
        q += p;
        p = temp;
    }
    return q;
}
```

Combinatorics:

Interestingly, this problem could also be solved using combinatorics.

Warning: Math-y stuff ahead, feel free to skip this section if you are not interested.

For example, let's assume $n = 6$.

Let:

x = number of 1's,
y = number of 2's.

We could reach the top using one of the four combinations below:

x	y	
6	0	=> 1) Six single steps.
4	1	=> 2) Four single steps and one double step.
2	2	=> 3) Two single steps and two double steps.
0	3	=> 4) Three double steps.

For the first combination pair $(x,y) = (6,0)$, there's obviously only one way of arranging six single steps.

For the second combination pair (4,1), there's five ways of arranging (think of it as slotting the double step between the single steps).

Similarly, there are six ways C(4,2) and one way $\binom{3}{3}$ of arranging the third and fourth combination pairs respectively.

Generally, for pair (x,y), there are a total of $\binom{x+y}{y} = \frac{(x+y)!}{x!y!}$ ways of arranging the 1's and 2's.

The total number of possible ways is the sum of all individual terms,

$f(6) = 1 + 5 + 6 + 1 = 13$.

Generalizing for all n's (including odd n),

$$f(\mathrm{n}) = \binom{n}{0} + \binom{n-1}{1} + \binom{n-2}{2} + \cdots + \binom{ceil\left(\frac{n}{2}\right)}{floor\left(\frac{n}{2}\right)}$$

43. Unique Paths

Code it now: https://oj.leetcode.com/problems/unique-paths/ Difficulty: Medium, Frequency: Low

Question:

A robot is located at the top-left corner of a $m \times n$ grid (marked 'Start' in the diagram below). The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below). How many possible unique paths are there?

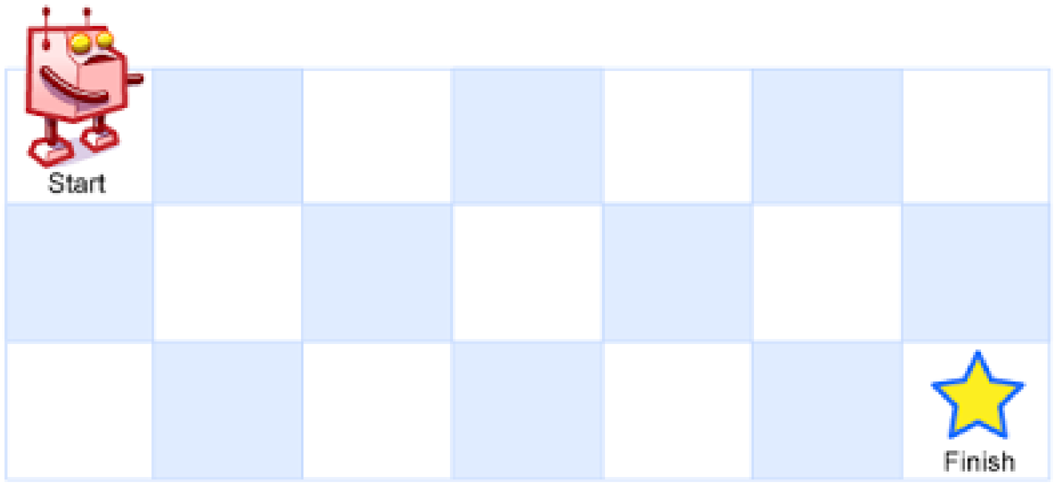


Figure 5: The grid above is 7×3, and is used to illustrate the problem.

$O\left(\binom{m+n}{m}\right)$ runtime, $O(m+n)$ space – Backtracking:

The most direct way is to write code that traverses each possible path, which can be done using backtracking. When you reach *row* = *m* and *col* = *n*, you know you've reached the bottom-right corner, and there is one additional unique path to it. However, when you reach *row* > *m* or *col* > *n*, then it's an invalid path and you should stop traversing. For any grid at *row* = *r* and *col* = *c*, you have two choices: Traverse to the right or traverse to the bottom. Therefore, the total unique paths at grid (*r*, *c*) are equal to the sum of total unique paths from the grid to the right and the grid below.

Deriving the runtime complexity is slightly tricky. Observe that the robot must go right exactly *m* times and go down exactly *n* times. Assume that the right movement is 0 and the down movement is 1. We could then represent the robot's path as a binary string of length = *m* + *n*, where the string contains *m* zeros and *n* ones. Since the backtracking algorithm is just trying to explore all possibilities, its runtime complexity is equivalent to the total permutations of a binary string that contains *m* zeros and *n* ones, which is $\binom{m+n}{m}$.

On the other hand, the space complexity is $O(m+n)$ due to the recursion that goes at most *m* + *n* level deep.

Below is the backtracking code in just five lines of code:

```
private int backtrack(int r, int c, int m, int n) {
   if (r == m - 1 && c == n - 1)
      return 1;
   if (r >= m || c >= n)
      return 0;

   return backtrack(r + 1, c, m, n) + backtrack(r, c + 1, m, n);
}

public int uniquePaths(int m, int n) {
   return backtrack(0, 0, m, n);
}
```

Improved Backtracking Solution using Memoization:

Although the above backtracking solution is easy to code, it is very inefficient in the sense that it recalculates the same solution for a grid over and over again. By caching the results, we prevent recalculation and only calculate when necessary. Here, we are using a dynamic programming (DP) technique called memoization.

```
private int backtrack(int r, int c, int m, int n, int[][] mat) {
   if (r == m - 1 && c == n - 1)
      return 1;
   if (r >= m || c >= n)
      return 0;

   if (mat[r + 1][c] == -1)
      mat[r + 1][c] = backtrack(r + 1, c, m, n, mat);
   if (mat[r][c+1] == -1)
      mat[r][c + 1] = backtrack(r, c + 1, m, n, mat);

   return mat[r + 1][c] + mat[r][c + 1];
}

public int uniquePaths(int m, int n) {
   int[][] mat = new int[m + 1][n + 1];
   for (int i = 0; i < m + 1; i++) {
      for (int j = 0; j < n + 1; j++) {
         mat[i][j] = -1;
      }
   }
   return backtrack(0, 0, m, n, mat);
}
```

$O(mn)$ runtime, $O(mn)$ space – Bottom-up dynamic programming:

If you notice closely, the above DP solution is using a *top-down* approach. Now let’s try a *bottom-up* approach. Remember this important relationship that is necessary for this DP solution to work:

The total unique paths at grid (*r*, *c*) are equal to the sum of total unique paths from grid to the right (*r*, *c* + 1) and the grid below (*r* + 1, *c*).

How can this relationship help us solve the problem? We observe that all grids of the bottom edge and right edge must all have only one unique path to the bottom-right corner. Using this as the base case, we can build our way up to our solution at grid (1, 1) using the relationship above.

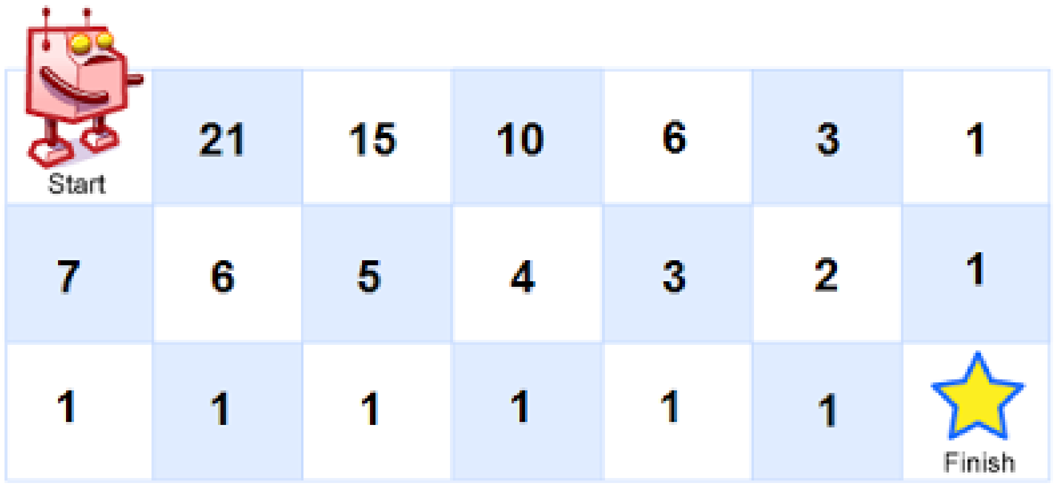


Figure 6: The total unique paths at grid (r, c) are equal to the sum of total unique paths from grid to the right (r, c + 1) and the grid below (r + 1, c).

```java
public int uniquePaths(int m, int n) {
   int[][] mat = new int[m + 1][n + 1];
   mat[m - 1][n] = 1;
   for (int r = m - 1; r >= 0; r--) {
      for (int c = n - 1; c >= 0; c--) {
         mat[r][c] = mat[r + 1][c] + mat[r][c + 1];
      }
   }
   return mat[0][0];
}
```

Combinatorial Solution:

It turns out this problem could be solved using combinatorics, which no doubt would be the most efficient solution. In order to see it as a combinatorial problem, there are some necessary observations. Look at the 7×3 sample grid in the picture above. Notice that no matter how you traverse the grids, you always traverse a total of 8 steps. To be more exact, you always have to choose 6 steps to the right (R) and 2 steps to the bottom (B). Therefore, the problem can be transformed to a question of how many ways can you choose 6R‘s and 2B‘s in these 8 steps. The answer is $\binom{8}{2}$ or $\binom{8}{6}$. Therefore, the general solution for an $m \times n$ grid is $\binom{m+n-2}{m-1}$.

Further Thoughts:

Now consider if some obstacles are added to the grids marked as ‘X’. How many unique paths would there be? A combinatorial solution is difficult to obtain, but the DP solution can be modified easily to accommodate this constraint. See Question [44. Unique Paths II].

44. Unique Paths II

Code it now: https://oj.leetcode.com/problems/unique-paths-ii/ Difficulty: Medium, Frequency: High

Question:

Similar to Question [43. Unique Paths], but now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space are marked as 1 and 0 respectively in the grid.

For example,

There is one obstacle in the middle of a 3×3 grid as illustrated below.

```
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
```

The total number of unique paths is 2.

Solution:

O(*mn*) runtime, *O*(*mn*) space – Dynamic programming:

It turns out to be really easy to extend from the [Bottom-up dynamic programming] approach above. Just set the total paths to 0 when you encounter an obstacle.

```
public int uniquePathsWithObstacles(int[][] obstacleGrid) {
   int m = obstacleGrid.length;
   if (m == 0) return 0;
   int n = obstacleGrid[0].length;
   int[][] mat = new int[m + 1][n + 1];
   mat[m - 1][n] = 1;
   for (int r = m - 1; r >= 0; r--) {
      for (int c = n - 1; c >= 0; c--) {
         mat[r][c] = (obstacleGrid[r][c] == 1) ? 0 : mat[r][c+1] + mat[r+1][c];
      }
   }
   return mat[0][0];
}
```

45. Maximum Sum Subarray

Code it now: https://oj.leetcode.com/problems/maximum-subarray/ Difficulty: Medium, Frequency: High

Question:

Find the contiguous subarray within an array (containing at least one number) that has the largest sum.

For example, given the array [2, 1, –3, 4, –1, 2, 1, –5, 4],

The contiguous array [4, –1, 2, 1] has the largest sum = 6.

Solution:

***O*(*n* log *n*) runtime, *O*(log *n*) stack space – Divide and Conquer:**

Assume we partition the array A into two smaller arrays S and T at the middle index, M. Then, $S = A_1 \dots A_{M-1}$, and $T = A_{M+1} \dots A_N$.

The contiguous subarray that has the largest sum could either:

i. Contain the middle element:

    a. The largest sum is the maximum suffix sum of $S + A_M$ + the maximum prefix sum of T.

ii. Does not contain the middle element:

    a. The largest sum is in S, which we could apply the same algorithm to S.

    b. The largest sum is in T, which we could apply the same algorithm to T.

```java
public int maxSubArray(int[] A) {
   return maxSubArrayHelper(A, 0, A.length - 1);
}

private int maxSubArrayHelper(int[] A, int L, int R) {
   if (L > R) return Integer.MIN_VALUE;
   int M = (L + R) / 2;
   int leftAns = maxSubArrayHelper(A, L, M - 1);
   int rightAns = maxSubArrayHelper(A, M + 1, R);
   int lMaxSum = 0;
   int sum = 0;
   for (int i = M - 1; i >= L; i--) {
      sum += A[i];
      lMaxSum = Math.max(sum, lMaxSum);
   }
   int rMaxSum = 0;
   sum = 0;
   for (int i = M + 1; i <= R; i++) {
      sum += A[i];
      rMaxSum = Math.max(sum, rMaxSum);
   }
   return Math.max(lMaxSum + A[M] + rMaxSum, Math.max(leftAns, rightAns));
}
```

The runtime complexity could be expressed as $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$, which is $O(n \log n)$. We will not attempt to prove it here; you could read up any advanced algorithm textbooks to learn the proof.

***O*(*n*) runtime, *O*(1) space – Dynamic programming:**

To devise a dynamic programming formula, let us assume that we are calculating the maximum sum of subarray that ends at a specific index.

Let us denote that:

f(*k*) = Maximum sum of subarray ending at index *k*.

Then,

f(*k*) = *max*(*f*(*k*-1) + A[*k*], A[*k*])

Using an array of size *n*, We could deduce the final answer by as *f*(*n* – 1), with the initial state of *f*(0) = *A*[0]. Since we only need to access its previous element at each step, two variables are sufficient. Notice the difference between the two: *maxEndingHere* and *maxSoFar*; the former is the maximum sum of subarray that *must* end at index *k*, while the latter is the global maximum subarray sum.

```
public int maxSubArray(int[] A) {
   int maxEndingHere = A[0], maxSoFar = A[0];
   for (int i = 1; i < A.length; i++) {
      maxEndingHere = Math.max(maxEndingHere + A[i], A[i]);
      maxSoFar = Math.max(maxEndingHere, maxSoFar);
   }
   return maxSoFar;
}
```

46. Maximum Product Subarray

Code it now: https://oj.leetcode.com/problems/maximum-product-subarray/ Difficulty: Medium, Frequency: High

Question:

Find the contiguous subarray within an array of integers that has the largest product. For example, given the array [2,3,-2,4], the contiguous subarray [2,3] has the largest product = 6.

Example Questions Candidate Might Ask:

Q: Could the subarray be empty?
A: No, the subarray must contain at least one number.

Solution:

This problem is very similar to Question [45. Maximum Sum Subarray]. There is a slight twist though. Besides keeping track of the largest product, we also need to keep track of the smallest product. Why? The smallest product, which is the largest in the negative sense could become the maximum when being multiplied by a negative number.

Let us denote that:

$f(k)$ = Largest product subarray, from index 0 up to k.

Similarly,

$g(k)$ = Smallest product subarray, from index 0 up to k.

Then,

$f(k) = max(f(k\text{-}1) * \mathrm{A}[k], \mathrm{A}[k], g(k\text{-}1) * \mathrm{A}[k])$

$g(k) = min(g(k\text{-}1) * \mathrm{A}[k], \mathrm{A}[k], f(k\text{-}1) * \mathrm{A}[k])$

There we have a dynamic programming formula. Using two arrays of size n, we could deduce the final answer as $f(n\text{-}1)$. Since we only need to access its previous elements at each step, two variables are sufficient.

```
public int maxProduct(int[] A) {
   assert A.length > 0;
   int max = A[0], min = A[0], maxAns = A[0];
   for (int i = 1; i < A.length; i++) {
      int mx = max, mn = min;
      max = Math.max(Math.max(A[i], mx * A[i]), mn * A[i]);
      min = Math.min(Math.min(A[i], mx * A[i]), mn * A[i]);
      maxAns = Math.max(max, maxAns);
   }
   return maxAns;
}
```

47. Coins in a Line

Code it now: Coming soon! Difficulty: Hard, Frequency: N/A

Question:

There are *n* coins in a line. (Assume *n* is even). Two players take turns to take a coin from one of the ends of the line until there are no more coins left. The player with the larger amount of money wins.

1. Would you rather go first or second? Does it matter?
2. Assume that you go first, describe an algorithm to compute the maximum amount of money you can win.

Figure 7: U.S. coins in various denominations in a line. Two players take turn to pick a coin from one of the ends until no more coins are left. Whoever with the larger amount of money wins.

Hints:

If you go first, is there a strategy you can follow which prevents you from losing? Try to consider how it matters when the number of coins is odd vs. even.

Solution:

Going first will guarantee that you will not lose. By following the strategy below, you will always win the game (or get a possible tie).

1. Count the sum of all coins that are odd-numbered. (Call this X)
2. Count the sum of all coins that are even-numbered. (Call this Y)
3. If X > Y, take the left-most coin first. Choose all odd-numbered coins in subsequent moves.
4. If X < Y, take the right-most coin first. Choose all even-numbered coins in subsequent moves.
5. If X == Y, you will guarantee to get a tie if you stick with taking only even-numbered/odd-numbered coins.

You might be wondering how you can always choose odd-numbered/even-numbered coins. Let me illustrate this using an example where you have 10 coins:

If you take the coin numbered 1 (the left-most coin), your opponent can only have the choice of taking coin numbered 2 or 10 (which are both even-numbered coins). On the other hand, if you choose to take the coin numbered 10 (the right-most coin), your opponent can only take coin numbered 1 or 9 (which are odd-numbered coins).

Notice that the total number of coins change from even to odd and vice-versa when player takes turn each time. Therefore, by going first and depending on the coin you choose, you are essentially forcing your opponent to take either only even-numbered or odd-numbered coins.

Now that you have found a non-losing strategy, could you compute the maximum amount of money you can win?

Hints:

One misconception is to think that the above non-losing strategy would generate the maximum amount of money as well. This is probably incorrect. Could you find a counter example? (You might need at least 6 coins to find a counter example).

Assume that you are finding the maximum amount of money in a certain range (ie, from coins numbered i to j, inclusive). Could you express it as a recursive formula? Find ways to make it as efficient as possible.

Solution for (2):

Although the simple strategy illustrated in Solution (1) guarantees you not to lose, it does not guarantee that it is optimal in any way.

Here, we use a good counter example to better see why this is so. Assume the coins are laid out as below:

{ 3, 2, 2, 3, 1, 2 }

Following our previous non-losing strategy, we would count the sum of odd-numbered coins, X = 3 + 2 + 1 = 6, and the sum of even-numbered coins, Y = 2 + 3 + 2 = 7. As Y > X, we would take the last coin first and end up winning with the total amount of 7 by taking only even-numbered coins.

However, let us try another way by taking the first coin (valued at 3, denote by (3)) instead. The opponent is left with two possible choices, the left coin (2) and the right coin (2), both valued at 2. No matter which coin the opponent chose, you can always take the other coin (2) next and the configuration of the coins becomes: { 2, 3, 1 }. Now, the coin in the middle (3) would be yours to keep for sure. Therefore, you win the game by a total amount of 3 + 2 + 3 = 8, which proves that the previous non-losing strategy is not necessarily optimal.

To solve this problem in an optimal way, we need to find efficient means in enumerating all possibilities. This is when Dynamic Programming (DP) kicks in and become so powerful that you start to feel magical.

First, we would need some observations to establish a recurrence relation, which is essential as our first step in solving DP problems.

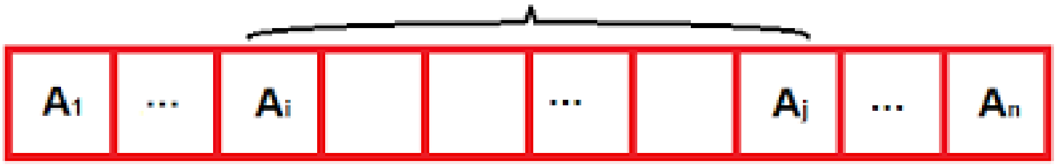


Figure 8: The remaining coins are { Ai ... Aj } and it is your turn. Let P(i, j) denotes the maximum amount of money you can get. Should you choose Ai or Aj?

Assume that P(i, j) denotes the maximum amount of money you can win when the remaining coins are { A_i, …, A_j }, and it is your turn now. You have two choices, either take Ai or Aj. First, let us focus on the case where you take Ai, so that the remaining coins become { A_{i+1} … A_j }. Since the opponent is as smart as you, he must choose the best way that yields the maximum for him, where the maximum amount he can get is denoted by P(i + 1, j).

Therefore, if you choose Ai, the maximum amount you can get is:

$$P_1 = \sum_{k=i}^{j} A_k - P(i+1, j)$$

Similarly, if you choose A_j, the maximum amount you can get is:

$$P_2 = \sum_{k=i}^{j} A_k - P(i, j-1)$$

Therefore,

$$\begin{aligned} P(i,j) &= \max(P_1, P_2) \\ &= max\left(\sum_{k=i}^{j} A_k - P(i+1,j), \sum_{k=i}^{j} A_k - P(i,j-1)\right) \end{aligned}$$

In fact, we are able to simplify the above relation further to (Why?):

$$P(i,j) = \sum_{k=i}^{j} A_k - min\big(P(i+1,j), P(i,j-1)\big)$$

Although the above recurrence relation is easy to understand, we need to compute the value of $\sum_{k=i}^{j} A_k$ in each step, which is not very efficient. To avoid this problem, we can store values of $\sum_{k=i}^{j} A_k$ in a table and avoid re-computations by computing in a certain order. Try to figure this out by yourself. (Hint: You would first compute P(1,1), P(2,2), … P(n, n) and work your way up).

A Better Solution:

There is another solution that does not rely on computing and storing results of $\sum_{k=i}^{j} A_k$, therefore is more efficient in terms of time and space. Let us rewind back to the case where you take Ai, and the remaining coins become { A_{i+1} … A_j }.

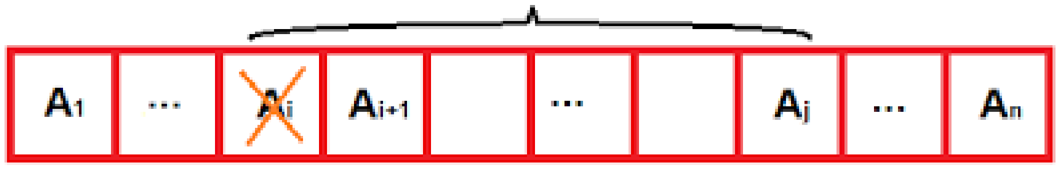


Figure 9: You took Ai from the coins { Ai … Aj }. The opponent will choose either Ai+1 or Aj. Which one would he choose?

Let us look one extra step ahead this time by considering the two coins the opponent will possibly take, A_{i+1} and A_j. If the opponent takes A_{i+1}, the remaining coins are { A_{i+2} … A_j }, which our maximum is denoted by $P(i + 2, j)$. On the other hand, if the opponent takes A_j, our maximum is $P(i + 1, j – 1)$. Since the opponent is as smart as you, he would have chosen the choice that yields the minimum amount to you.

Therefore, the maximum amount you can get when you choose Ai is:

$$P_1 = A_i + min\big(P(i+2,j), P(i+1,j-1)\big)$$

Similarly, the maximum amount you can get when you choose A_j is:

$$P_2 = A_j + min\big(P(i+1,j-1), P(i,j-2)\big)$$

Therefore,

$$P(i,j) = \max(P_1, P_2)$$
$$= max\left(A_i + min\big(P(i+2,j), P(i+1,j-1)\big), A_j + min\big(P(i+1,j-1), P(i,j-2)\big)\right)$$

Although the above recurrence relation could be implemented in few lines of code, its complexity is exponential. The reason is that each recursive call branches into a total of four separate recursive calls, and it could be *n* levels deep from the very first call). Memoization provides an efficient way by avoiding re-computations using intermediate results stored in a table. Below is the code which runs in $O(n^2)$ time and takes $O(n^2)$ space.

The code contains a function *printMoves* which prints out all the moves you and the opponent make (assuming both of you are taking the coins in an optimal way).

```
const int MAX_N = 100;

void printMoves(int P[][MAX_N], int A[], int N) {
  int sum1 = 0, sum2 = 0;
  int m = 0, n = N-1;
  bool myTurn = true;
  while (m <= n) {
    int P1 = P[m+1][n]; // If take A[m], opponent can get...
    int P2 = P[m][n-1]; // If take A[n]
    cout << (myTurn ? "I" : "You") << " take coin no. ";
    if (P1 <= P2) {
      cout << m+1 << " (" << A[m] << ")";
      m++;
    } else {
      cout << n+1 << " (" << A[n] << ")";
      n--;
    }
    cout << (myTurn ? ", " : ".\n");
    myTurn = !myTurn;
  }
  cout << "\nThe total amount of money (maximum) I get is " << P[0][N-1] << ".\n";
}

int maxMoney(int A[], int N) {
  int P[MAX_N][MAX_N] = {0};
  int a, b, c;
  for (int i = 0; i < N; i++) {
    for (int m = 0, n = i; n < N; m++, n++) {
      assert(m < N); assert(n < N);
      a = ((m+2 <= N-1)             ? P[m+2][n] : 0);
      b = ((m+1 <= N-1 && n-1 >= 0) ? P[m+1][n-1] : 0);
      c = ((n-2 >= 0)               ? P[m][n-2] : 0);
      P[m][n] = max(A[m] + min(a,b),
                    A[n] + min(b,c));
    }
  }
  printMoves(P, A, N);
  return P[0][N-1];
}
```

Further Thoughts:

Assume that your opponent is so dumb that you are able to manipulate him into choosing the coins you want him to choose. Now, what is the maximum possible amount of money you can win?

Chapter 9: Binary Search

48. Search Insert Position

Code it now: https://oj.leetcode.com/problems/search-insert-position/ Difficulty: Medium, Frequency: Low

Question:

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You may assume no duplicates in the array.

Here are few examples.
[1,3,5,6], 5 → 2
[1,3,5,6], 2 → 1
[1,3,5,6], 7 → 4
[1,3,5,6], 0 → 0

Solution:

This problem is a direct application of Binary Search, as you can spot it easily by the keywords *sorted* and *finding target*. The requirements seem complex, but let's first start with something we're already familiar with – The raw binary search algorithm.

Let's start with defining two variables, *L* and *R* representing its lowest and highest inclusive indices that are searched, which are initialized to 0 and *n* – 1 respectively.

```
int L = 0, R = A.length - 1;
while (L < R) {
   int M = (L + R) / 2;
   // TODO: Implement conditional checks.
}
```

Code 1: Getting started with a Binary Search template.

Now, the key part of the binary search – We look at the middle element, and ask: "Is the middle element smaller than the target element?" If this is true, then it means all elements from *L* up to *M* inclusive could be excluded from the search. Otherwise, the middle element is greater or equal to the target element and that means all elements from *M* + 1 up to *R* could be excluded.

```
int L = 0, R = A.length - 1;
while (L < R) {
   int M = (L + R) / 2;
   if (A[M] < target) {
      L = M + 1;
   } else {
      R = M;
   }
}
```

Code 2: Filling out the key part of the Binary Search algorithm.

A good thing to verify your above binary search does not stuck in an infinite loop is to test with input containing two elements, e.g., [1,3] and test with *target* = 0 and 1. Here, our binary search works properly, but if we were to define *M* as the *upper* middle, that is: *M* = (*L* + *R* + 1) / 2, then it will stuck in an infinite loop.

We've now reached the final step. When the while loop ends, *L* must be equal to *R* and it is a valid index. Obviously, if *A*[*L*] is equal to target, we return *L*. If *A*[*L*] is greater than *target*, that means we are inserting *target* ***before*** *A*[*L*], so we return *L*. If *A*[*L*] is less than *target*, that means we insert *target* ***after*** A[L], so we return *L* + 1.

```
public int searchInsert(int[] A, int target) {
   int L = 0, R = A.length - 1;
   while (L < R) {
      int M = (L + R) / 2;
      if (A[M] < target) {
         L = M + 1;
      } else {
         R = M;
      }
   }
   return (A[L] < target) ? L + 1 : L;
}
```

49. Find Minimum in Sorted Rotated Array

Code it now: https://oj.leetcode.com/problems/find-minimum-in-rotated-sorted-array/ Difficulty: Medium, Frequency: High

Question:

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

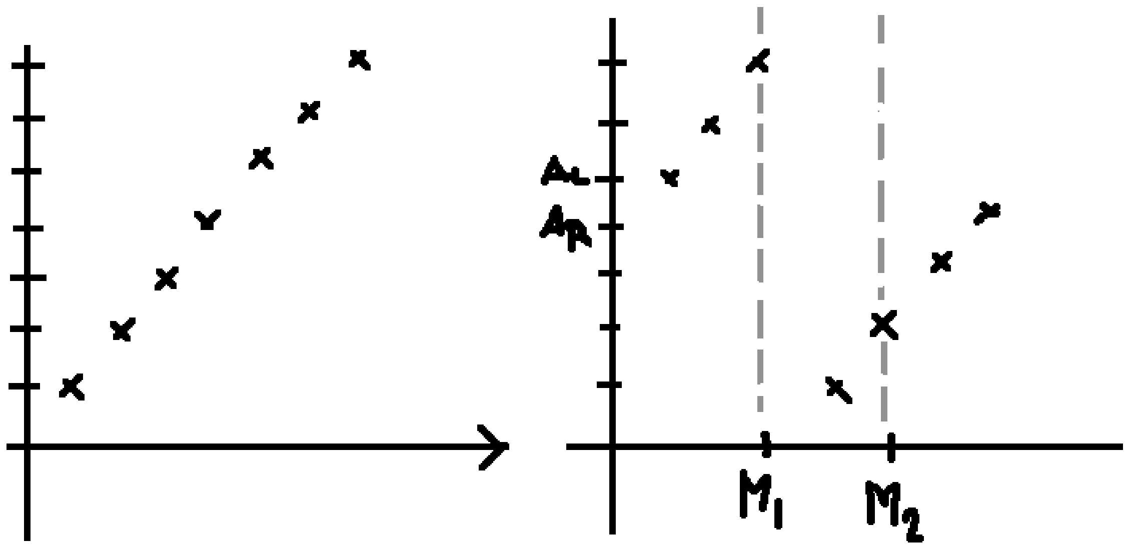
(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the minimum element.

You may assume no duplicate exists in the array.

Solution:

The minimum is at A_i where $A_{i-1} > A_i$. Notice that if we subdivide the array into two, one will always be sorted, while the other contains the minimum.



Imagine we have an array [1,2,3,4,5,6,7] (See graph 1) which was being rotated 3 steps to the right [5,6,7,1,2,3,4] (See graph 2). Let's say we subdivide the array at point *k* to two subarrays [A_L, A_{L+1}, …, A_k], [A_{k+1}, …, A_R].

If the sorted array is not rotated, then $A_L < A_R$ then we could return A_L as the minimum immediately.

Otherwise for a sorted array that was rotated at least one step, A_L must always be greater than A_R.

Let's assume we choose M_1 as the dividing point. Since $A_{M1} > A_R$, we know that each element in [A_L … A_{M1}] is greater than A_R (Remember that $A_L > A_R$?). Therefore, the minimum value must locate in [A_{M1+1} … A_R].

On the other hand, let's assume we choose M_2 as the dividing point. Since $A_{M2} \le A_R$, we know that each element in [A_{M2+1} … A_R] is greater than A_{M2}. Therefore, the minimum point must locate in [A_L … A_{M2}].

As we are discarding half of the elements at each step, the runtime complexity is $O(\log n)$.

To understand the correct terminating condition, we look at two elements. Let us choose the lower median as M = (L + R) / 2. Therefore, if there are two elements, it will choose A_L as the first element.

There are two cases for two elements:

A = [1,2]

B = [2,1]

For A, $1 < 2 \Rightarrow A_M < A_R$, and therefore it will set R = M => R = 0.

For B, $2 > 1 \Rightarrow A_M > A_R$, and therefore it will set L = M + 1 => L = 1.

Therefore, it is clear that when L == R, we have found the minimum element.

```
public int findMin(int[] A) {
   int L = 0, R = A.length - 1;
   while (L < R && A[L] >= A[R]) {
      int M = (L + R) / 2;
      if (A[M] > A[R]) {
         L = M + 1;
      } else {
         R = M;
      }
   }
   return A[L];
}
```

Further Thoughts:

If the rotated sorted array could contain duplicates? Is your algorithm still $O(\log n)$ in runtime complexity?

50. Find Minimum in Rotated Sorted Array II – with duplicates

Code it now: https://oj.leetcode.com/problems/find-minimum-in-rotated-sorted-array-ii/ Difficulty: Hard, Frequency: Medium

Question:

If the rotated sorted array could contain duplicates? Is your algorithm still *O*(log *n*) in runtime complexity?

Solution:

For case where $A_L == A_M == A_R$, the minimum could be on A_M's left or right side (eg, [1, 1, 1, 0, 1] or [1, 0, 1, 1, 1]). In this case, we could not discard either subarrays and therefore such worst case degenerates to the order of *O*(*n*).

```java
public int findMin(int[] A) {
   int L = 0, R = A.length - 1;
   while (L < R && A[L] >= A[R]) {
      int M = (L + R) / 2;
      if (A[M] > A[R]) {
         L = M + 1;
      } else if (A[M] < A[L]) {
         R = M;
      } else {   // A[L] == A[M] == A[R]
         L = L + 1;
      }
   }
   return A[L];
}
```