Twitter

433. Minimum Genetic Mutation

Example 2:

```
Input: start = "AACCGGTT", end = "AAACGGTA", bank =
["AACCGGTA","AACCGCTA","AAACGGTA"]
Output: 2
```

BFS

```
class Solution {
    Character[] list = {'A','C','G','T'};
    public int minMutation(String start, String end, String[] bank) {
        Set<String> visited = new HashSet<>();
        Set<String> bankSet = new HashSet<>();
        for(String each:bank){
            bankSet.add(each);
        }
        Deque<String> queue = new LinkedList<>();
        queue.offer(start);
        visited.add(start);
        int step = 0;
        while(!queue.isEmpty()){
            int size = queue.size();
            while(size-->0){
                String curr = queue.poll();
                if(curr.equals(end)) return step;
                for(int i=0;i<curr.length();i++){
                    for(int j=0;j<4;j++){
                        String newStr = generateNewStr(curr,i,j);
                        if(newStr.equals("")||!bankSet.contains(newStr)||visited.contains(newStr)) continue;
                        visited.add(newStr);
                        queue.offer(newStr);
                    }
                }
            }
            step++;

        }
        return -1;
    }
    public String generateNewStr(String str,int i,int j){
        char curr = str.charAt(i);
        if(curr==list[j]) return "";
        return str.substring(0,i)+list[j]+str.substring(i+1,str.length());
    }
}
```

1348. Tweet Counts Per Frequency

需求分析：

- 记录 **user** 和其对应的推文 **list -> HashMap**
- 给出时间上下界计算 list 数量 （二分搜索 /TreeMap）
  - TreeMap<time,count>

```java
class TweetCounts {
    Map<String,TreeMap<Integer,Integer>> userToTweet;
    public TweetCounts() {
        userToTweet = new HashMap<>();
    }
    public void recordTweet(String tweetName, int time) {
        if(!userToTweet.containsKey(tweetName)){
            userToTweet.put(tweetName,new TreeMap<Integer,Integer>());
        }
        userToTweet.get(tweetName).put(time,userToTweet.get(tweetName).getOrDefault(time,0)+1);
    }
    public List<Integer> getTweetCountsPerFrequency(String freq, String tweetName, int startTime, int endTime) {
        List<Integer> res = new ArrayList<>();
        if(!userToTweet.containsKey(tweetName)) return res;
        TreeMap<Integer,Integer> tree = userToTweet.get(tweetName);
        int gap = getGap(freq);
        int start = startTime;
        int end = Math.min(endTime,start+gap-1);
        Map.Entry<Integer,Integer> entry = tree.ceilingEntry(startTime);
        while(start<=end){
            int count = 0;
            while(entry!=null&&entry.getKey()<=end){
                count+=entry.getValue();
                entry = tree.higherEntry(entry.getKey());
            }
            res.add(count);
            start = end+1;
            end =  Math.min(endTime,start+gap-1);
        }
        return res;
    }
    public int getGap(String freq){
        int gap = 1;
        switch(freq){
            case "minute":
                gap = 60;
                break;
            case "hour":
                gap = 60*60;
                break;
            case "day":
                gap = 60*60*24;
                break;
        }
        return gap;
    }
```

- TreeMap的ceilingEntry,higherEntry方法
- start/end界定

635. Design Log Storage System

- transform string into int
  - string -> string[] -> int[]
    - Arrays.stream(timestamp.split(":")).mapToInt(Integer::parseInt).toArray();
  - lower bound/upper bound

```
    public long convert(int[] st) {
        st[1] = st[1] - (st[1] == 0 ? 0 : 1);
        st[2] = st[2] - (st[2] == 0 ? 0 : 1);
        return (st[0] - 1999L) * (31 * 12) * 24 * 60 * 60 + st[1] * 31 * 24 * 60 * 60 + st[2] *
24 * 60 * 60 + st[3] * 60 * 60 + st[4] * 60 + st[5];
    }
    public long getTime(String[] arr){
        long time = 0L;
        time+=(Integer.parseInt(arr[0])-1999)*366*24*60*60;
        if(Integer.parseInt(arr[1])!=0) time+=(Integer.parseInt(arr[1])-1)*31*24*60*60;
        if(Integer.parseInt(arr[2])!=0) time+=(Integer.parseInt(arr[2])-1)*24*60*60;
        time+=(Integer.parseInt(arr[3])-0)*60*60;
        time+=(Integer.parseInt(arr[4])-0)*60;
        time+=(Integer.parseInt(arr[5])-0);
        return time;
    }
    public long getGranularity(String str,String granularity,boolean isStart){
        String[] num = str.split(":");
        String[] res = {"1999","00","00","00","00","00"};
        int gran = reference.get(granularity);
        for(int i=0;i<=gran;i++){
            res[i] = num[i];
        }
        // if(!isStart) res[gran]=String.valueOf(Integer.parseInt(res[gran])+1);
        // return getTime(res);
        int[] t = Arrays.stream(res).mapToInt(Integer::parseInt).toArray();
        if (!isStart) t[gran]++;
        return convert(t);
    }
}
```

```
TreeMap<Long,Integer> tree;
Map<String,Integer> reference;
public LogSystem() {
    tree = new TreeMap<>();
    reference = new HashMap<>();
    reference.put("Year",0);
    reference.put("Month",1);
    reference.put("Day",2);
    reference.put("Hour",3);
    reference.put("Minute",4);
    reference.put("Second",5);
}
public void put(int id, String timestamp) {
    int[] st = Arrays.stream(timestamp.split(":")).mapToInt(Integer::parseInt).toArray();
    tree.put(convert(st), id);
}
public List<Integer> retrieve(String start, String end, String granularity) {
    List<Integer> res = new ArrayList<>();
    long i = getGranularity(start,granularity,true);
    long j = getGranularity(end,granularity,false);
    Map.Entry<Long,Integer> entry = tree.ceilingEntry(i);
    while(entry!=null&&entry.getKey()<j){
        res.add(entry.getValue());
        entry = tree.higherEntry(entry.getKey());
    }
    return res;
}
```

780. Reaching Points

1️⃣ brute force

Java Python Copy

```java
class Solution {
    public boolean reachingPoints(int sx, int sy, int tx, int ty) {
        if (sx > tx || sy > ty) return false;
        if (sx == tx && sy == ty) return true;
        return reachingPoints(sx+sy, sy, tx, ty) || reachingPoints(sx, sx+sy, tx, ty);
    }
}
```

Complexity Analysis

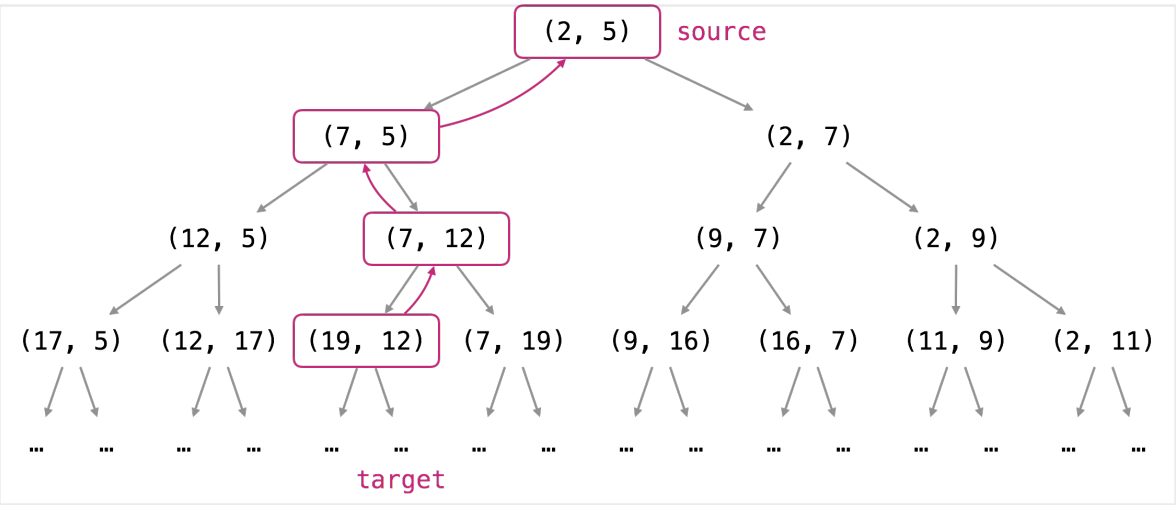
- Time Complexity: $O(2^{tx+ty})$, a loose bound found by considering every move as `(x, y) -> (x+1, y)` or `(x, y) -> (x, y+1)` instead.
- Space Complexity: $O(tx * ty)$, the size of the implicit call stack.

2 DP (enhancement of BF)

```java
import java.awt.Point;

class Solution {
    Set<Point> seen;
    int tx, ty;

    public boolean reachingPoints(int sx, int sy, int tx, int ty) {
        seen = new HashSet();
        this.tx = tx;
        this.ty = ty;
        search(new Point(sx, sy));
        return seen.contains(new Point(tx, ty));
    }

    public void search(Point P) {
        if (seen.contains(P)) return;
        if (P.x > tx || P.y > ty) return;
        seen.add(P);
        search(new Point(P.x + P.y, P.y));
        search(new Point(P.x, P.x + P.y));
    }
}
```

- Pruning
- Memo

3 Work Backwards



Java | Python

```java
class Solution {
    public boolean reachingPoints(int sx, int sy, int tx, int ty) {
        while (tx >= sx && ty >= sy) {
            if (sx == tx && sy == ty)
                return true;
            if (tx > ty) tx -= ty;
            else ty -= tx;
        }
        return false;
    }
}
```

Complexity Analysis

- Time Complexity: $O(\max(tx, ty))$. If say `ty = 1`, we could be subtracting `tx` times.
- Space Complexity: $O(1)$.

4️⃣ Work Backwards (Modulo Variant)

```java
class Solution {
    public boolean reachingPoints(int sx, int sy, int tx, int ty) {
        while (tx >= sx && ty >= sy) {
            if (tx == ty) break;
            if (tx > ty) {
                if (ty > sy) tx %= ty;
                else return (tx - sx) % ty == 0;
            } else {
                if (tx > sx) ty %= tx;
                else return (ty - sy) % tx == 0;
            }
        }
        return (tx == sx && ty == sy);
    }
}
```

Complexity Analysis

- Time Complexity: $O(\log(\max(tx, ty)))$. The analysis is similar to the analysis of the Euclidean algorithm, and we assume that the modulo operation can be done in $O(1)$ time.
- Space Complexity: $O(1)$.

341. Flatten Nested List Iterator

1️⃣ BF - Make a Flat List with Recursion

```java
public class NestedIterator implements Iterator<Integer> {

    private List<Integer> integers = new ArrayList<Integer>();
    private int position = 0; // Pointer to next integer to return.

    public NestedIterator(List<NestedInteger> nestedList) {
        flattenList(nestedList);
    }

    // Recursively unpacks a nested list in DFS order.
    private void flattenList(List<NestedInteger> nestedList) {
        for (NestedInteger nestedInteger : nestedList) {
            if (nestedInteger.isInteger()) {
                integers.add(nestedInteger.getInteger());
            } else {
                flattenList(nestedInteger.getList());
            }
        }
    }

    @Override
    public Integer next() {
        // As per Java specs, we should throw an exception if no more ints.
        if (!hasNext()) throw new NoSuchElementException();
        // Return int at current position, and then *after*, increment position.
        return integers.get(position++);
    }

    @Override
    public boolean hasNext() {
        return position < integers.size();
    }
}
```

- Time complexity:

  We'll analyze each of the methods separately.

  - **Constructor:** $O(N + L)$.

    The constructor is where all the time-consuming work is done.

    For each list within the nested list, there will be one call to `flattenList(...)`. The loop within `flattenList(...)` will then iterate n times, where n is the number of integers within that list. Across all calls to `flattenList(...)`, there will be a total of N loop iterations. Therefore, the time complexity is the number of lists plus the number of integers, giving us $O(N + L)$.

    Notice that the maximum depth of the nesting does not impact the time complexity.

  - **next():** $O(1)$.

    Getting the next element requires incrementing `position` by 1 and accessing an element at a particular index of the `integers` list. Both of these are $O(1)$ operations.

  - **hasNext():** $O(1)$.

    Checking whether or not there is a next element requires comparing the length of the `integers` list to the `position` variable. This is an $O(1)$ operation.

- Space complexity : $O(N + D)$.

  The most obvious auxiliary space is the `integers` list. The length of this is $O(N)$.

  The less obvious auxiliary space is the space used by the `flattenList(...)` function. Recall that recursive functions need to keep track of where they're up to by putting stack frames on the runtime stack. Therefore, we need to determine what the maximum number of stack frames there could be at a time is. Each time we encounter a nested list, we call `flattenList(...)` and a stack frame is added. Each time we finish processing a nested list, `flattenList(...)` returns and a stack frame is removed. Therefore, the maximum number of stack frames on the runtime stack is the maximum nesting depth, D.

  Because these two operations happen one-after-the-other, and either could be the largest, we add their time complexities together giving a final result of $O(N + D)$.

2️⃣ Stack

```java
// In Java, the Stack class is considered deprecated. Best practice is to use
// a Deque instead. We'll use addFirst() for push, and removeFirst() for pop.
private Deque<NestedInteger> stack;

public NestedIterator(List<NestedInteger> nestedList) {
    // The constructor puts them on in the order we require. No need to reverse.
    stack = new ArrayDeque(nestedList);
}


@Override
public Integer next() {
    // As per java specs, throw an exception if there's no elements left.
    if (!hasNext()) throw new NoSuchElementException();
    // hasNext ensures the stack top is now an integer. Pop and return
    // this integer.
    return stack.removeFirst().getInteger();
}


@Override
public boolean hasNext() {
    // Check if there are integers left by getting one onto the top of stack.
    makeStackTopAnInteger();
    // If there are any integers remaining, one will be on the top of the stack,
    // and therefore the stack can't possibly be empty.
    return !stack.isEmpty();
}


private void makeStackTopAnInteger() {
    // While there are items remaining on the stack and the front of
    // stack is a list (i.e. not integer), keep unpacking.
    while (!stack.isEmpty() && !stack.peekFirst().isInteger()) {
        // Put the NestedIntegers onto the stack in reverse order.
        List<NestedInteger> nestedList = stack.removeFirst().getList();
        for (int i = nestedList.size() - 1; i >= 0; i--) {
            stack.addFirst(nestedList.get(i));
        }
    }
}
```

Complexity Analysis

Let N be the total number of *integers* within the nested list, L be the total number of *lists* within the nested list, and D be the maximum nesting depth (maximum number of lists inside each other).

- Time complexity.
  - **Constructor:** $O(N + L)$.

    The worst-case occurs when the initial input nestedList consists entirely of integers and empty lists (everything is in the top-level). In this case, every item is reversed and stored, giving a total time complexity of $O(N + L)$.

  - **makeStackTopAnInteger():** $O(\frac{L}{N})$ or $O(1)$.

    If the top of the stack is an integer, then this function does nothing; taking $O(1)$ time.

    Otherwise, it needs to process the stack until an integer is on top. The best way of analyzing the time complexity is to look at the total cost across all calls to `makeStackTopAnInteger()` and then divide by the number of calls made. Once the iterator is exhausted `makeStackTopAnInteger()` must have seen every integer at least once, costing $O(N)$ time. Additionally, it has seen every list (except the first) on the stack at least once also, so this costs $O(L)$ time. Adding these together, we get $O(N + L)$ time.

    The amortized time of a single `makeStackTopAnInteger` is the total cost, $O(N + L)$, divided by the number of times it's called. In order to get all integers, we need to have called it N times. This gives us an amortized time complexity of $\frac{O(N + L)}{N} = O(\frac{N}{N} + \frac{L}{N}) = O(\frac{L}{N})$.

  - **next():** $O(\frac{L}{N})$ or $O(1)$.

    All of this method is $O(1)$, except for possibly the call to `makeStackTopAnInteger()`, giving us a time complexity the same as `makeStackTopAnInteger()`.

  - **hasNext():** $O(\frac{L}{N})$ or $O(1)$.

    All of this method is $O(1)$, except for possibly the call to `makeStackTopAnInteger()`, giving us a time complexity the same as `makeStackTopAnInteger()`.

- Space complexity : $O(N + L)$.

  In the worst case, where the top list contains N integers, or L empty lists, it will cost $O(N + L)$ space. Other expensive cases occur when the nesting is very deep. However, it's useful to remember that $D \leq L$ (because each layer of nesting requires another list), and so we don't need to take this into account.

3️⃣ two stack

```java
public class NestedIterator implements Iterator<Integer> {

    private Deque<List<NestedInteger>> listStack = new ArrayDeque<>();
    private Deque<Integer> indexStack = new ArrayDeque<>();

    public NestedIterator(List<NestedInteger> nestedList) {
        listStack.addFirst(nestedList);
        indexStack.addFirst(0);
    }
    @Override
    public Integer next() {
        if (!hasNext()) throw new NoSuchElementException();
        int currentPosition = indexStack.removeFirst();
        indexStack.addFirst(currentPosition + 1);
        return listStack.peekFirst().get(currentPosition).getInteger();
    }
    @Override
    public boolean hasNext() {
        makeStackTopAnInteger();
        return !indexStack.isEmpty();
    }
    private void makeStackTopAnInteger() {

        while (!indexStack.isEmpty()) {

            // If the top list is used up, pop it and its index.
            if (indexStack.peekFirst() >= listStack.peekFirst().size()) {
                indexStack.removeFirst();
                listStack.removeFirst();
                continue;
            }

            // Otherwise, if it's already an integer, we don't need to do anything.
            if (listStack.peekFirst().get(indexStack.peekFirst()).isInteger()) {
                break;
            }

            // Otherwise, it must be a list. We need to update the previous index
            // and then add the new list with an index of 0.
            listStack.addFirst(listStack.peekFirst().get(indexStack.peekFirst()).getList());
            indexStack.addFirst(indexStack.removeFirst() + 1);
            indexStack.addFirst(0);
        }
    }
}
```

Complexity Analysis

Let N be the total number of *integers* within the nested list, L be the total number of *lists* within the nested list, and D be the maximum nesting depth (maximum number of lists inside each other).

- Time complexity:
  - **Constructor:** $O(1)$.

    Pushing a list onto a `stack` is *by reference* in all the programming languages we're using here. This means that instead of creating a new list, some information about how to get to the existing list is put onto the stack. The list is not traversed, as it doesn't need reversing this time, and we're not pushing the items on one-by-one. This is, therefore, an $O(1)$ operation.
  - **makeStackTopAnInteger() / next() / hasNext():** $O(\frac{L}{N})$ or $O(1)$.

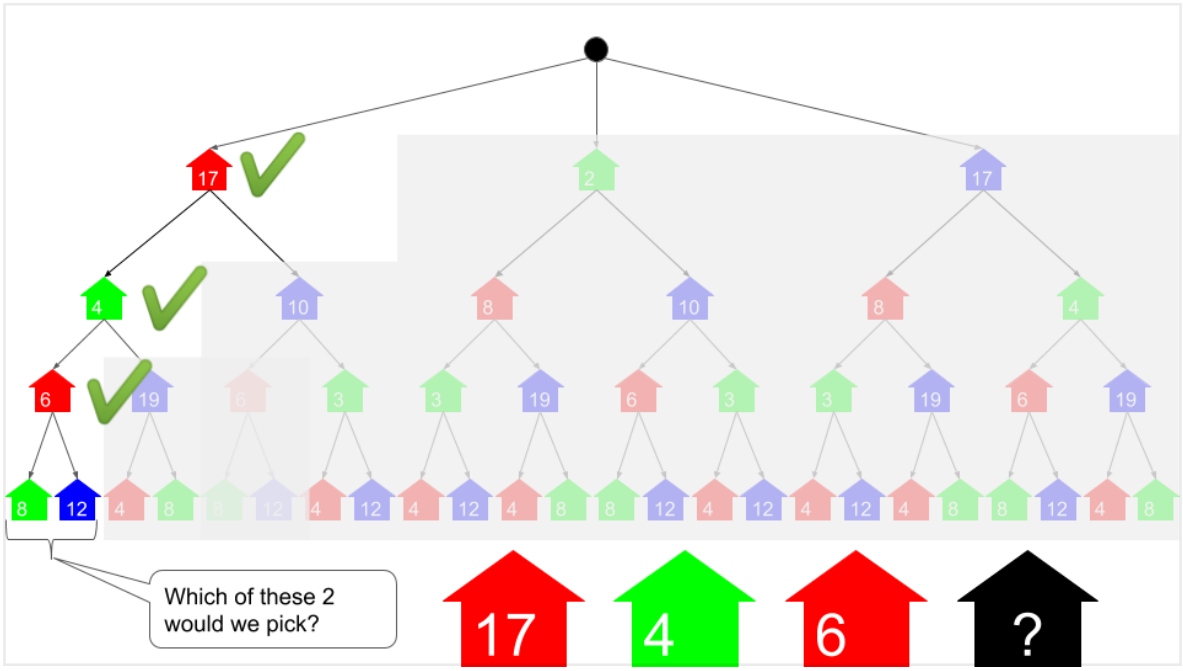
    Same as Approach 2.
- Space complexity : $O(D)$.

  At any given time, the stack contains only *one* nestedList reference for each level. This is unlike the previous approach, wherein the worst case we need to put almost all elements onto the stack.

  Because there's one reference on the stack at each level, the worst case is when we're looking at the deepest leveled list, giving a space complexity is $O(D)$.

256. Paint House

1️⃣ Brute force with a Recursive Tree




```
private int[][] costs;

public int minCost(int[][] costs) {
    if (costs.length == 0) {
        return 0;
    }
    this.costs = costs;
    return Math.min(paintCost(0, 0), Math.min(paintCost(0, 1), paintCost(0, 2)));
}

private int paintCost(int n, int color) {
    int totalCost = costs[n][color];
    if (n == costs.length - 1) {
    } else if (color == 0) { // Red
        totalCost += Math.min(paintCost(n + 1, 1), paintCost(n + 1, 2));
    } else if (color == 1) { // Green
        totalCost += Math.min(paintCost(n + 1, 0), paintCost(n + 1, 2));
    } else { // Blue
        totalCost += Math.min(paintCost(n + 1, 0), paintCost(n + 1, 1));
    }
    return totalCost;
}
```

Time complexity : O(2^n)

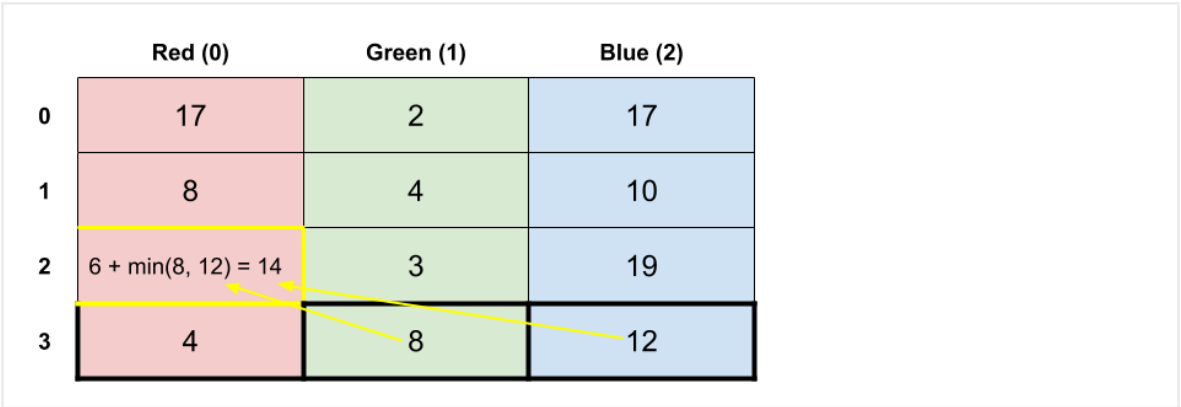
Space complexity : $O(n)$.

2️⃣ Memoization

```java
class Solution {

    private int[][] costs;
    private Map<String, Integer> memo;

    public int minCost(int[][] costs) {
        if (costs.length == 0) {
            return 0;
        }
        this.costs = costs;
        this.memo = new HashMap<>();
        return Math.min(paintCost(0, 0), Math.min(paintCost(0, 1), paintCost(0, 2)));
    }

    private int paintCost(int n, int color) {
        if (memo.containsKey(getKey(n, color))) {
            return memo.get(getKey(n, color));
        }
        int totalCost = costs[n][color];
        if (n == costs.length - 1) {
        } else if (color == 0) { // Red
            totalCost += Math.min(paintCost(n + 1, 1), paintCost(n + 1, 2));
        } else if (color == 1) { // Green
            totalCost += Math.min(paintCost(n + 1, 0), paintCost(n + 1, 2));
        } else { // Blue
            totalCost += Math.min(paintCost(n + 1, 0), paintCost(n + 1, 1));
        }
        memo.put(getKey(n, color), totalCost);

        return totalCost;
    }

    private String getKey(int n, int color) {
        return String.valueOf(n) + " " + String.valueOf(color);
    }
}
```

Time complexity : $O(n)$
Space complexity : $O(n)$.

3️⃣ DP bottom-up



```
ass Solution {
  public int minCost(int[][] costs) {

      for (int n = costs.length - 2; n >= 0; n--) {
          // Total cost of painting the nth house red.
          costs[n][0] += Math.min(costs[n + 1][1], costs[n + 1][2]);
          // Total cost of painting the nth house green.
          costs[n][1] += Math.min(costs[n + 1][0], costs[n + 1][2]);
          // Total cost of painting the nth house blue.
          costs[n][2] += Math.min(costs[n + 1][0], costs[n + 1][1]);
      }

      if (costs.length == 0) return 0;

      return Math.min(Math.min(costs[0][0], costs[0][1]), costs[0][2]);
  }
```

Time complexity : $O(n)$
Space complexity : $O(1)$

4️⃣ Dynamic Programming with Optimized Space Complexity

```
  public int minCost(int[][] costs) {

      if (costs.length == 0) return 0;

      int[] previousRow = costs[costs.length -1];

      for (int n = costs.length - 2; n >= 0; n--) {

          /* PROBLEMATIC CODE IS HERE
           * This line here is NOT making a copy of the original, it's simply
           * making a reference to it Therefore, any writes into currentRow
           * will also be written into "costs". This is not what we wanted!
           */
          int[] currentRow = costs[n];

          // Total cost of painting the nth house red.
          currentRow[0] += Math.min(previousRow[1], previousRow[2]);
          // Total cost of painting the nth house green.
          currentRow[1] += Math.min(previousRow[0], previousRow[2]);
          // Total cost of painting the nth house blue.
          currentRow[2] += Math.min(previousRow[0], previousRow[1]);
          previousRow = currentRow;
      }

      return Math.min(Math.min(previousRow[0], previousRow[1]), previousRow[2]);
  }
```

57. Insert Interval

Greedy

```java
public int[][] insert(int[][] intervals, int[] newInterval) {
    LinkedList<int[]> output = new LinkedList<>();
    int nStart = newInterval[0],nEnd = newInterval[1];
    int n = intervals.length;
    int idx = 0;
    while(idx<n && intervals[idx][0]<nStart){
        output.add(intervals[idx++]);
    }

    int[] interval = new int[2];
    if(output.isEmpty()||output.getLast()[1] < nStart){
        output.add(newInterval);
    }else{
        interval = output.removeLast();
        interval[1] = Math.max(nEnd,interval[1]);
        output.add(interval);
    }
    while(idx<n){
        int[] curr = intervals[idx++];
        if(curr[0]>output.getLast()[1]){
            output.add(curr);
        }else{
            interval = output.removeLast();
            interval[1] = Math.max(interval[1],curr[1]);
            output.add(interval);
        }
    }
    return output.toArray(new int[output.size()][2]);
}
```

1297. Maximum Number of Occurrences of a Substring

only focus on minSize!!

HashMap

```
String s;
int minSize;
Map<String,Integer> record;
public int maxFreq(String s, int maxLetters, int minSize, int maxSize) {
    int n = s.length();
    this.s = s;
    this.minSize = minSize;
    record = new HashMap<>();
    int res = 0;
    for(int i=0;i<=n-minSize;i++){
        String sub = s.substring(i,i+minSize);
        if(getNum(sub)>maxLetters) continue;
        record.put(sub,record.getOrDefault(sub,0)+1);
    }
    for(int each:record.values()) res = Math.max(res,each);
    return res;
}
public int getNum(String str){
    Set<Integer> count = new HashSet<>();
    for(char ch:str.toCharArray()){
        count.add(ch-'a');
    }
    return count.size();
}
```

1817. Finding the Users Active Minutes
Get to know what the problem is

```
public int[] findingUsersActiveMinutes(int[][] logs, int k) {
    int[] res = new int[k];
    Map<Integer,Set<Integer>> map = new HashMap<>();
    for(int[] each:logs){
        map.putIfAbsent(each[0],new HashSet<Integer>());
        map.get(each[0]).add(each[1]);
    }
    for(Set<Integer> each:map.values()){
        res[each.size()-1]++;
    }
    return res;
}
```

453. Minimum Moves to Equal Array Elements

In one move, you can increment n - 1 elements of the array by 1

==

decrement 1 elements of the array by 1

```java
public int minMoves(int[] nums) {
    PriorityQueue<Integer> pq = new PriorityQueue<>();
    int min = Integer.MAX_VALUE;
    for(int each:nums) min = Math.min(min,each);
    int res = 0;
    for(int each:nums) res += each-min;
    return res;
}
```

647. Palindromic Substrings

1️⃣ **BF -** Check All Substrings
2️⃣ DP

```java
public int countSubstrings(String s) {
    int n = s.length(), ans = 0;

    if (n <= 0)
        return 0;

    boolean[][] dp = new boolean[n][n];

    // Base case: single letter substrings
    for (int i = 0; i < n; ++i, ++ans)
        dp[i][i] = true;

    // Base case: double letter substrings
    for (int i = 0; i < n - 1; ++i) {
        dp[i][i + 1] = (s.charAt(i) == s.charAt(i + 1));
        ans += (dp[i][i + 1] ? 1 : 0);
    }

    // All other cases: substrings of length 3 to n
    for (int len = 3; len <= n; ++len)
        for (int i = 0, j = i + len - 1; j < n; ++i, ++j) {
            dp[i][j] = dp[i + 1][j - 1] && (s.charAt(i) == s.charAt(j));
            ans += (dp[i][j] ? 1 : 0);
        }

    return ans;
}
```

Time Complexity: O(N^2)
Space Complexity: O(N^2)

3️⃣ Expand Around Possible Centers

```java
public int countSubstrings(String s) {
    int ans = 0;

    for (int i = 0; i < s.length(); ++i) {
        // odd-length palindromes, single character center
        ans += countPalindromesAroundCenter(s, i, i);

        // even-length palindromes, consecutive characters center
        ans += countPalindromesAroundCenter(s, i, i + 1);
    }

    return ans;
}

private int countPalindromesAroundCenter(String ss, int lo, int hi) {
    int ans = 0;

    while (lo >= 0 && hi < ss.length()) {
        if (ss.charAt(lo) != ss.charAt(hi))
            break;      // the first and last characters don't match!

        // expand around the center
        lo--;
        hi++;

        ans++;
    }

    return ans;
}
```

Further Thoughts

Better approaches do exist to solve this problem in sub-quadratic time, however those are significantly complex and impractical to implement in most interviews.

Some known approaches are:

1. **Binary Search with a fast rolling hash algorithm (like Rabin-Karp).** This approach tries to optimize Approach #3 by speeding up the time to figure out the largest palindrome for each of the $2N - 1$ centers in logarithmic time. This approach counts all palindromic substrings in $O(N \log N)$ time. Here's a Quora answer by T.V. Raziman which explains this approach well.

2. **Palindromic trees (also known as EERTREE).** It is a data structure invented by Mikhail Rubinchik which links progressively larger palindromic substrings within a string. The tree construction takes linear time, and the number of palindromic substrings can be counted while constructing the tree in $O(N)$ time. Additionally, the tree can be used to compute how many distinct palindromic substrings are in a string (it's just the number of nodes in the tree) and how frequently each such palindrome occurs. This blog post does a good job of explaining the construction of a palindromic tree.

3. **Suffix Arrays with quick Lowest common Ancestor (LCA) lookups.** This approach utilizes Ukonnen's algorithm to build suffix trees for the input string and its reverse in linear time. Subsequently, quick LCA lookups can be used to find maximum palindromes, which are themselves composed of smaller palindromes. This approach can produce a count of all palindromic substrings in $O(N)$ time. The original paper describes the algorithm, and this Quora answer demonstrates an example.

4. **Manacher's algorithm.** *It's basically Approach #3, on steroids.*™ The algorithm reuses computations done for previous palindromic centers to process new centers in sub-linear time (which reduces progressively for each new center). This algorithm counts all palindromic substrings in $O(N)$ time. This e-maxx post provides a fairly simple implementation of this algorithm.

358. Rearrange String k Distance Apart

Heap -> HashMap traverse -> addAll

Queue -> k non-repeated

```
Map<Character,Integer> map = new HashMap<>();
public String rearrangeString(String s, int k) {
    PriorityQueue<Character> pq = new PriorityQueue<>((a,b)->map.get(b).compareTo(map.get(a)));
    for(char each:s.toCharArray()){
        map.put(each,map.getOrDefault(each,0)+1);
    }
    pq.addAll(map.keySet());
    StringBuffer sb = new StringBuffer();
    Queue<Character> queue = new LinkedList<>();
    while(!pq.isEmpty()){
        char curr = pq.poll();
        int freq = map.get(curr);
        map.put(curr,freq-1);
        sb.append(curr);
        queue.offer(curr);
        if(queue.size()>=k){
            char next = queue.poll();
            if(map.get(next)>0){
                pq.offer(next);
            }
        }
    }
    return sb.length()==s.length()?sb.toString():"";
}
```

1492. The kth Factor of n

1️⃣ Brute Force, O(*N*) time

```java
public int kthFactor(int n, int k) {
    for (int x = 1; x < n / 2 + 1; ++x) {
        if (n % x == 0) {
            --k;
            if (k == 0) {
                return x;
            }
        }
    }

    return k == 1 ? n : -1;
}
```

2 Heap

```java
public int kthFactor(int n, int k) {
    PriorityQueue<Integer> pq = new PriorityQueue<Integer>(new Comparator<Integer>(){
        public int compare(Integer a,Integer b){
            return b-a;
        }
    });
    for(int i=1;i<=Math.sqrt(n);i++){
        if(n%i==0){
            pq.offer(i);
            if(i*i!=n) pq.offer(n/i);
            while(pq.size()>k) pq.poll();
        }
    }
    return k == pq.size() ? pq.poll() : -1;
}
```

Complexity Analysis

- Time Complexity: $\mathcal{O}(\sqrt{N} \times \log k)$.
- Space Complexity: $\mathcal{O}(\min(k, \sqrt{N}))$ to keep the heap of size k.

3 Math

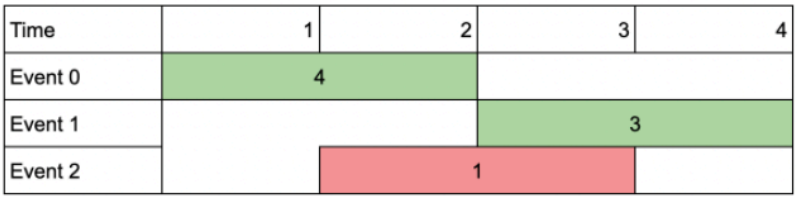
```java
public int kthFactor(int n, int k) {
    List<Integer> divisors = new ArrayList();
    int sqrtN = (int) Math.sqrt(n);
    for (int x = 1; x < sqrtN + 1; ++x) {
        if (n % x == 0) {
            --k;
            divisors.add(x);
            if (k == 0) {
                return x;
            }
        }
    }

    // If n is a perfect square
    // we have to skip the duplicate
    // in the divisor list
    if (sqrtN * sqrtN == n) {
        ++k;
    }

    int nDiv = divisors.size();
    return (k <= nDiv) ? n / divisors.get(nDiv - k) : -1;
}
```

Complexity Analysis

- Time Complexity: $\mathcal{O}(\sqrt{N})$.
- Space Complexity: $\mathcal{O}(\min(k, \sqrt{N}))$ to store the list of divisors.

1751. Maximum Number of Events That Can Be Attended II

Example 1:



Input: events = [[1,2,4],[3,4,3],[2,3,1]], k = 2
Output: 7
Explanation: Choose the green events, 0 and 1 (0-indexed) for a total value of 4 + 3 = 7.

```
    int[][] events;
    int[][] memo;
    int k;
    int res;
    public int maxValue(int[][] events, int k) {
        Arrays.sort(events,new Comparator<int[]>(){
            public int compare(int[] a,int[] b){
                if(a[1]==b[1]) return b[0]-a[0];
                else return a[1]-b[1];
            }
        });
        this.memo = new int[events.length][k+1];
        this.events = events;
        this.k = k;
        for(int i=0;i<memo.length;i++) Arrays.fill(memo[i],-1);
        // dp(events.length-1,k);
        return dp(events.length-1,k);
    }
    public int dp(int index,int k){
        if(k==0) return 0;
        if(index==-1) return 0;
        if(memo[index][k]!=-1) return memo[index][k];
        int currStart = events[index][0];
        int last = -1;
        for(int i=index-1;i>=0;i--){
            if(events[i][1]<currStart){
                last = i;
                break;
            }
        }
        int maxValue = Math.max(dp(index-1,k),dp(last,k-1)+events[index][2]);
        memo[index][k] = maxValue;
        return maxValue;

    }
```

1248. Count Number of Nice Subarrays

preSum array

```
public int numberOfSubarrays(int[] nums, int k) {
    int count = 0;
    int[] cnt = new int[nums.length+1];
    int res = 0;
    cnt[0]++;
    for(int i=0;i<nums.length;i++){
        count += nums[i]&1;
        if(count>=k) res+=cnt[count-k];
        cnt[count]++;
    }
    return res;
}
```