Algorithm Problems

Solutions by Felipe Thomé.

Almost all of these probems came from interviewbit.com.

Summary

Array Anti Diagonals

Given a N by N square matrix, return an array of its anti-diagonals. Look at the example for more details.

Example:

```
Given,
1 2 3
4 5 6
7 8 9

Return
[
  [1],
  [2, 4],
  [3, 5, 7],
  [6, 8],
  [9]
]
```

Time complexity: O(N), where N is the number of elements in the matrix.

Explanation: your diagonals start at each element of your first row, and after those, at the last element of each row after the first one. So just iterate over these elements where the diagonals start and then for each one print the elements in the diagonal decreasing one column and increasing one row each time.

```
vector<vector<int>> diagonal(vector<vector<int>> &a) {
  vector<vector<int>> ans;

  int t = 0, l = 0, dim = a.size();

  while (t < dim) {
    int ct = t, cl = l;
    vector<int> temp;

    while (ct < dim && cl >= 0) {
      temp.push_back(a[ct][cl]);
      ct++;
      cl--;
    }

    if (temp.size()) ans.push_back(temp);

    if (l == dim - 1) t++;
    else l++;
  }

  return ans;
}
```

Array Find Duplicate

Given a read only array of n + 1 integers between 1 and n, find one number that repeats in linear time using less than O(n) space and traversing the stream sequentially O(1) times.

Example:
Given [3 4 1 4 1]
Return 1

If there are multiple possible answers (like in the sample case above), output any one.
If there is no duplicate, output -1

Time complexity: O(N), where N is the size of the array.
Space complexity: O(1).

Explanation: Floyd Cycle Algorithm. Since the size of the array is N + 1 and the numbers in the array are between 1 .. N, we can treat the array as a graph where each node is one if the indices of the array and the edges are given by the values in each one of these indices. Now, when there is a repeated number there will be two edges of our graph arriving at a node/indice. In other words there will be a cycle in our graph. So if we apply the Floyd Cycle Algorithm we can find the starting node of the cycle and this starting node is one of the repeated number in the array.

```
int repeatedNumber(const vector<int> &a) {
  if (a.size() <= 1) return -1;

  int n = a.size(), count = 0, turt = 0, hare = 0, i = 0;

  do {
    turt = a[turt];
```

```
      hare = a[a[hare]];
      count++;
    }
    while (turt != hare && count < n);

    if (count < n) {
      while (i != turt) {
        turt = a[turt];
        i = a[i];
      }

      return i;
    }

    return -1;
  }
```

Array First Missing Integer

```
Given an unsorted integer array, find the first missing positive integer.
```

Example:
```
Given [1,2,0] return 3,
Given [3,4,-1,1] return 2,
Given [-8, -7, -6] returns 1

Your algorithm should run in O(N) time and use constant space.
```

Time complexity: O(N), where N is the number of elements in the array.
Space complexity: O(N).

Explanation: since the exercise ask us to use O(1) space we don't have another choice besides modifying the given array. We also need to sort the array (there is no other way), but the exercise asks for O(N) complexity so we need to take advatange of some condition to be able to find a O(N) sort algorithm. Remember we are looking for the first missing integer, so we can use our array indexation to organize the integers we find in their respective buckets. For example, we want number 1 at index 0, number 2 at index 1, number 3 at index 2 and so on. When we find a number that can't be placed in some place at the array using it index, like 0 or any number bigger than the array size, we just live it there. Eventually this numbers will be replaced by some number that should be in that bucket and these number will end up in the end of the array. Look at:
before [4 2 1], after [1 2 4]
After this process we just need to look for the first integer that is not the same as index + 1. In the above case would be 4 (the correct number at that bucket would be 3), so we return the index + 1, which is 3.

```
  int firstMissingPositive(vector<int> &a) {
    if (a.size() == 0) return 1;

    for (int i = 0; i < a.size(); i++) {
      while (a[i] > 0 && a[i] <= a.size() && a[i] != i + 1 && a[a[i] - 1] != a[i]) {
        swap(a[i], a[a[i] - 1]);
      }
    }

    for (int i = 0; i < a.size(); i++) {
      if (a[i] != i + 1) return i + 1;
    }
    return a.size() + 1;
  }
```

Array Largest Number

Given a list of non negative integers, arrange them such that they form the largest number.

Note: The result may be very large, so you need to return a string instead of an integer.

Example:
Given [3, 30, 34, 5, 9], the largest formed number is 9534330.

Time complexity: O(N*logN*M), where N is the number of elements in the array and M is the average amount of digits in the numbers.

Explanation: first of all we need to manipulate strings because we need to access each digit of these numbers and since our answer will be a string it is easier if we already start our algorithm manipulating strings. We need to implement a custom comparator for the C++ sort() function. Our comparator will receive two elements/strings at each call, let them be called "a" and "b". Then we check if ab > ba, and if it is we return true because we want "a" before "b" (returning true in a comparator means that the first argument "a" must preceed the second argument "b"). When ab is bigger than ba? Starting from the left, when we find ab[i] > ba[i]. Finally, combine this sorted vector in a string and return.

```
bool cmp(string a, string b) {
  string ab = a + b;
  string ba = b + a;

  for (int i = 0; i < ab.size(); i++) {
    if (ab[i] == ba[i]) continue;
    return ab[i] > ba[i] ? true : false;
  }

  return false;
}

string largestNumber(const vector<int> &a) {
  vector<string> b;
  string ans = "";

  for (int i = 0; i < a.size(); i++) b.push_back(to_string(a[i]));

  sort(b.begin(), b.end(), cmp);

  for (int i = 0; i < b.size(); i++) ans += b[i];

  int j = 0;
  while (ans[j] == '0' && j + 1 < ans.size()) j++;

  return ans.substr(j);
}
```

Array Max Absolute Difference

You are given an array of N integers, A1, A2 ,…, AN. Return maximum value of f(i, j) for all 1 ≤ i, j ≤ N. f(i, j) is defined as |A[i] - A[j]| + |i - j|, where |x| denotes absolute value of x.

Example:
Given A = [1, 3, -1]

```
f(1, 1) = f(2, 2) = f(3, 3) = 0
f(1, 2) = f(2, 1) = |1 - 3| + |1 - 2| = 3
f(1, 3) = f(3, 1) = |1 - (-1)| + |1 - 3| = 4
f(2, 3) = f(3, 2) = |3 - (-1)| + |2 - 3| = 5
```
So, we return 5.

Time complexity: O(N), where N is the size of the array.
Space complexity: O(1).

Explanation: My solution is different from the editorial, but has better space complexity. Think about what you are calculating with each of these absolute values in the equation. Each of these values is a distance between two integer numbers. So, if we are able to add the distance |i - j| to the other distance |A - B| we can get an equation that we can solve in O(N) time.
Think about these integers in the x-axis:
x: ... -4 -3 -2 -1 0 1 2 3 4 ...
|i - j| will always be a positive value, which means it will always increase the distance between the integer numbers A and B. So, we have two options: or we add our |i - j| distance to max(A, B) sending our right x-value more to the right of the x-axis or we subtract the distance |i - j| from min(A, B) sending our left x-value more to the left of the x-axis. Lets define our new equation as |C - D|, where C = max(A, B) + |i - j| and D = min(A, B). Notice, that we maximize |C - D| by making C the biggest possible value and D the minimum. If we start from the end of our array we can easily find the maximum C value. For a given i the maximum value C can increase by 1 because i is one unit further than it was before in the previous iteration, or C can become a[i], because the current value a[i] is the biggest value found until now.
We make D the current value a[i] and store in our answer max(ans, C - D).
For example,

```
       0  1   2   3  4
Given -3, 6, -2, -1, 2
i = 4 .. 0
C = 2 (a[4] + 0), 3 (max + 1), 4 (max + 1), 6 (a[1] + 0), 7 (max + 1)
D = 2,           -1,          -2,           6,            -3
ans = 0, 4, 6, 6, 10

i = 0 .. 4 (see why we are going also from 0 .. 4 below)
C = -3 (a[0] + 0), 6 (a[1] + 0), 7 (max + 1), 8 (max + 2), 9 (max + 3)
D = -3,             6,           -2,          -1,            2
ans = 10 10 10 10 10
So, we return 10
```

Notice, we are going from the end of the array to the beginning and this is fine as long as the final minimum value D is at the left of the maximum value in the array. Otherwise, when we found the maximum value C, the minimum value D that would give our answer already passed in our iterations because it was at the right of this final C value. To solve this problem we need to iterate from the beginning of the array to, so we can check for our C value the minimum values that are at its right.

```cpp
int maxArr(vector<int> &a) {
  if (a.size() == 0) return 0;

  int ans = INT_MIN, mx = INT_MIN;

  for (int i = a.size() - 1; i >= 0; i--) {
    mx = max(a[i], mx + 1);
    ans = max(ans, mx - a[i]);
  }

  mx = INT_MIN;
  for (int i = 0; i < a.size(); i++) {
    mx = max(a[i], mx + 1);
    ans = max(ans, mx - a[i]);
  }

  return ans;
}
```

Array Max Non Negative Subarray

Find out the maximum sub-array of non negative numbers from an array.
The sub-array should be continuous. That is, a sub-array created by choosing
the second and fourth element and skipping the third element is invalid.

Maximum sub-array is defined in terms of the sum of the elements in the
sub-array. Sub-array A is greater than sub-array B if sum(A) > sum(B).

Example:
A : [1, 2, 5, -7, 2, 3]
The two sub-arrays are [1, 2, 5] [2, 3].
The answer is [1, 2, 5] as its sum is larger than [2, 3]

Note 1: If there is a tie, then compare with segment's length and return
segment which has maximum length.
Note 2: If there is still a tie, then return the segment with minimum
starting index.

Time complexity: O(N), where N is the number of elements in the array.

Explanation: just iterate over the array. While you are looking at positive
elements store the elements in another array. Also keep a sum of these
elements you are storing. When you find a negative element stop storing and
see if the sum you have now is bigger than the previous sum, if it is store
this array you just made in your answer. When you find another positive
number start the process again. Keep doing this while you still have
elements to look at in the array.

```
vector<int> maxset(vector<int> &a) {
  vector<int> ans;
  int i = 0;
  long long sum = 0;

  while(i < a.size()) {
    vector<int> temp;
    int j = i;
    long long currSum = 0;

    while (j < a.size() && a[j] >= 0) {
      currSum += a[j];
      temp.push_back(a[j]);
      j++;
    }

    if (currSum > sum) {
      ans = temp;
      sum = currSum;
    }
    else if (currSum == sum) {
      if (temp.size() > ans.size()) ans = temp;
    }
    i = j + 1;
  }

  return ans;
}
```

Array Max Sum Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest sum. For this problem, return the maximum sum.

Example:

Given the array [-2,1,-3,4,-1,2,1,-5,4],
the contiguous subarray [4,-1,2,1] has the largest sum = 6.

Time complexity: O(N), where N is the size of the given array.
Space complexity: O(1).

Explanation: the subarray with the maximum sum will never start with a negative number, because you could discard this negative number and start from the next position to make a larger sum. Also, every candidate sum will always try to get as most positive sums as possible. For example, if you have an array like [2, -1, 5] the maximum sum will be 2 + (-1) + 5, so even if we have a negative number in the middle of our sum it is still better to sum 5 with the sum (2 + (-1)) because its result is positive, so we get 1 + 5 = 6. Thinking about all positive numbers in the array as possible starting points for our maximum sum we will not consider the left part (all the numbers to the left of the starting positive number) just if the sum of the left part is negative.

```
// Kadane's algorithm O(N).
int maxSubArray(vector<int> &a) {
  int sum = INT_MIN, acc = 0, i;

  for (i = 0; i < a.size(); i++) {
    // Don't let the subarray start with a negative number. Also, start the
    // sum again when it becomes negative.
    if ((acc + a[i]) > 0) {
      acc += a[i];
      sum = max(sum, acc);
    }
    else {
      acc = 0;
    }

    // In case there are just negative elements in the array.
    if (a[i] < 0 && a[i] > sum) sum = a[i];
  }

  return sum;
}

// // O(N^2).
// int maxSubArray(vector<int> &a) {
//   int sum = INT_MIN;

//   for (int i = 0; i < a.size(); i++) {
//     int acc = 0;
//     for (int j = i; j < a.size(); j++) {
//       acc += a[j];
//       sum = max(sum, acc);
//     }
//   }

//   return sum;
// }
```

Array Merge Intervals

Given a collection of intervals, merge all overlapping intervals.
Make sure the returned intervals are sorted.

Example:

Given [1,3],[2,6],[8,10],[15,18],

Return [1,6],[8,10],[15,18].

Time complexity: O(NlogN), where N is the number of intervals.
Space complexity: O(N), where N is the number of intervals.

Explanation: sort the intervals in ascending order by the starting point. The current interval will be merged while the maximum end point found so far is larger than the starting point of the current interval. When this doesn't happen we insert the merged interval in the answer, create another interval and start to merge again.

```
struct Interval {
  int start;
  int end;
  Interval() : start(0), end(0) {}
  Interval(int s, int e) : start(s), end(e) {}
};

bool cmp(Interval i1, Interval i2) {
  return i1.start < i2.start || (i1.start == i2.start && i1.end < i2.end);
}

vector<Interval> merge(vector<Interval> &a) {
  sort(a.begin(), a.end(), cmp);

  int i = 0;
  vector<Interval> ans;

  while (i < a.size()) {
    Interval *curr = new Interval(a[i].start, a[i].end);

    while (i < a.size() - 1 && curr->end >= a[i + 1].start) {
      curr->end = max(curr->end, a[i + 1].end);
      i++;
    }

    ans.push_back(*curr);
    i++;
  }

  return ans;
}
```

Array Min Steps Infinite Grid

You are in an infinite 2D grid where you can move in any of the 8 directions:
(x,y) to
(x+1, y),
(x - 1, y),
(x, y+1),
(x, y-1),
(x-1, y-1),
(x+1,y+1),
(x-1,y+1),
(x+1,y-1)

You are given a sequence of points and the order in which you need to cover the points. Give the minimum number of steps in which you can achieve it. You start from the first point.

Example:
Input [(0, 0), (1, 1), (1, 2)]
Output 2
It takes 1 step to move from (0, 0) to (1, 1). It takes one more step to move from (1, 1) to (1, 2).

Time complexity: O(N), where N is the number of points.

Explanation: you just need to realize that since you can move diagonally that the minimum distance from one point to another will always be the max(abs(x1 - x2), abs(y1 - y2)) because the diagonals allow you to behave like if you were moving horizontally or vertically to achive the point. For example, suppose we want to know the minimum distance of (0, 0) to (1, 3). Before calculating this distance imagined you wanted to go from point (0, 0) to (0, 3). The minimum distance in this case would be 3 because we just need to walk horizontally until the final point. In our original case the final point is not at the same line or column so we need to wak diagonally. And when we do, we increase x by one and y by one, so it is like if we were in a case where we are at the same line or column of the origin point. It is difficult to explain, but easy to realize if you draw the grid.

```
int coverPoints(vector<int> &x, vector<int> &y) {
  int n = x.size(), ans = 0;

  for (int i = 1; i < x.size(); i++) {
    ans += max(abs(x[i] - x[i - 1]), abs(y[i] - y[i - 1]));
  }

  return ans;
}
```

Array N Over-3 Repetitions

You're given a read only array of n integers. Find out if any integer occurs more than n/3 times in the array in linear time and constant additional space. If so, return the integer. If not, return -1.
If there are multiple solutions, return any one.

Example :

Given [1 2 3 1 1]
Output 1, because 1 occurs 3 times which is more than 5/3 times.

Time complexity: O(N), where N is the size of the given array.

Space complexity: O(1).

Explanation: given k, we want to find out if a number occurs more than n / k times. This is the Boyer-Moore Majority Vote algorithm, but the basic version of this algorithm finds out if a number occur more than n / 2 times. Here, we want n / 3 times, but we will generalize for any integer k.

We need to solve this problem k - 1 variables to store information, so if k is small we can say that this algorithm has linear time complexity, and constant space complexity. If k is large we say we have O(n * k) time complexity and O(k) space complexity.

Lets see how we would approach this problem for k = 2. We have 1 variable, which is a struct with an integer indicating an element, and another integer indicating the number of repetitions for this element. When k = 2, it is easy to see that the majority element will have a count that you could subtract the sum of counts of every other element and you still would have a number larger than 0. Boyer-Moore look at each element just once and if it already saw this number before (the check is performed using our unique variable that stores an element and its count) the algorithm increments the count, otherwise it has two options:

1. Decrement the count if it is larger than 0.
2. If the count is already 0 store the current element.

So, if we have a count of 1 for a particular element, and the current element is different from the one with count 1, we just decrement our count and continue the iteration discarding the current element. The idea is that we are checking if we have more elements different from the one with the count because the majority element will be present at least (n / 2 + 1) times which is larger than the rest of elements (n / 2 - 1).

For example,
Given [1, 2, 3, 4, 1]

#1
element = 1 count = 1

#2
element = 1 count = 0

#3
element = 3 count = 1

#4
element = 3 count = 0

#5
element = 1 count = 1

So, we have 1 as a candidate for the majority element. Notice, we have a candidate and not an actual answer because we could have a situation where we were with a count 0 and it happened that we started our count again for the next element giving a count different from 0, but it doesn't mean it is a majority. For example, the array with three elements [1, 2, 3], would give e = 3 and c = 1, but 3 is not a majority. So, after these steps we need to get our variable e and check if it really occurs more than n / 2 times in the array.

Now, for k = 3 we need two variables, where each is a struct indicating an element and its count. We are looking for an element that occurs more than n / 3 times. So, we are looking for an element that occurs at least (n / 3 + 1) times, so the rest of the elements is (2n / 3 - 1). Since we have two variables we have two possible places to store information and we just get rid of one element when its variable has count 0. That means we have the majority as (n / 3 + 1) and there are just ((2n / 3 - 1) / 2) = (n / 3 - 1) elements that can take place in the variable where the majority would be stored. For example,

```
Given [1, 2, 3, 4, 1]

#1
element1 = 1 count1 = 1
element2 = undefined count2 = 0

#2
element1 = 1 count1 = 1
element2 = 2 count2 = 1

#3 (decrement both)
element1 = 1 count1 = 0
element2 = 2 count2 = 0

#4
element1 = 4 count1 = 1
element2 = 2 count2 = 0

#5
element1 = 4 count1 = 1
element2 = 1 count2 = 1

Now we check elements 4 and 1 and we find that 1 occurs more than n / 3
times.

For k = 4, we are looking for an element that occurs at least (n / 4 + 1)
times, so the rest of the elements is (3n / 4 - 1). We use three variables
so we get (3n / 4 - 1) = (n / 4 - 1). Generalizing this for any integer k,
we need k - 1 variables to execute this approach.
```

```
int repeatedNumber(vector<int> &a) {
  int e1 = -1, c1 = 0, e2 = -1, c2 = 0;

  for (int i = 0; i < a.size(); i++) {
    if (a[i] == e1) c1++;
    else if (a[i] == e2) c2++;
    else {
      if (c1 == 0) {
        e1 = a[i];
        c1 = 1;
      }
      else if(c2 == 0) {
        e2 = a[i];
        c2 = 1;
      }
      else {
        c1--;
        c2--;
        if (c1 == 0) e1 = -1;
        if (c2 == 0) e2 = -1;
      }
    }
  }

  int actualC1 = 0, actualC2 = 0;
  for (int i = 0; i < a.size(); i++) {
    if (e1 == a[i]) actualC1++;
    else if (e2 == a[i]) actualC2++;
  }

  if (actualC1 > a.size() / 3) return e1;
  else if (actualC2 > a.size() / 3) return e2;
  else return -1;
}
```

Array N Over K Repetitions

Given an array and an integer k, find out if an element in the array occurs more than n / k times. In other words, generalize Boyer–Moore Majority vote algorithm.

Time complexity: O(N * K), where N is the size of the given array and K the number of parts we want the array to be divided.
Space complexity: O(K).

Explanation: Look at "N / 3 repetitions in array" problem.

```
int findElementIndex(vector<pair<int, int>> &counts, int b) {
  for (int i = 0; i < counts.size(); i++) {
    if (counts[i].first == b) return i;
  }
  return -1;
}

int findEmptyIndex(vector<pair<int, int>> &counts) {
  for (int i = 0; i < counts.size(); i++) {
    if (counts[i].first == -1) return i;
  }
  return -1;
}

void decreaseCounts(vector<pair<int, int>> &counts) {
  for (int i = 0; i < counts.size(); i++) {
    counts[i].second--;
    if (counts[i].second == 0) counts[i].first = -1;
  }
}

vector<int> findMajorities(const vector<int> &a, int b) {
  vector<pair<int, int>> counts(b, {-1, 0});
  vector<int> ans;

  for (int i = 0; i < a.size(); i++) {
    int idx = findElementIndex(counts, a[i]);
    if (idx != -1) {
      counts[idx].second++;
    }
    else {
      int emptyIdx = findEmptyIndex(counts);
      if (emptyIdx != -1) {
        counts[emptyIdx].first = a[i];
        counts[emptyIdx].second++;
      }
      else {
        decreaseCounts(counts);
      }
    }
  }

  for (int i = 0; i < counts.size(); i++) {
    int actualCount = 0;
    for (int j = 0; j < a.size(); j++) {
      if (counts[i].first == a[j]) actualCount++;
    }
    if (actualCount > a.size() / b) ans.push_back(counts[i].first);
  }

  return ans;
}
```

Array Pascal Triangle Kth Row

Given an index k, return the kth row of the Pascal’s triangle.
Pascal’s triangle : To generate A[C] in row R, sum up A’[C] and A’[C-1] from previous row R - 1.

Note: k is 0 based. k = 0, corresponds to the row [1].

Can you optimize your algorithm to use only O(k) extra space?

Example:

```
k = 3
Output [1, 3, 3, 1]
```

Time complexity: O(K), where K is the row number.
Space complexity: O(K).

Explanation: forget the editorial tip. The best way to calculate this is using a recurrence relation. So, you can make this in linear time.

(n, k) represents Newton's binomial.

```
(n, k) = n! / (n - k)!k!

For k = 1
n! / (n - 1)!1! =
= n * (n - 1)! / (n - 1)! =
= n

For k = 2
n! / (n - 2)!2! =
= n * (n - 1) * (n - 2)! / (n - 2)!2 =
= n * (n - 1) / 2

For k = 3
n! / (n - 3)!3! =
= n * (n - 1) * (n - 2) * (n - 3)! / (n - 3)!3! =
= n * (n - 1) * (n - 2) / 3 * 2
```

So for a row n, its n + 1 elements xi can be computed like:

```
x0 = 1
x1 = x0 * n
x2 = x1 * (n - 1) / 2
x3 = x2 * (n - 2) / 3
x4 = x3 * (n - 3) / 4
x5 = x4 * (n - 4) / 5
...
xn = xn-1 * (n - (n - 1)) / n
```

Notice there is another optimization you could make that is you could calculate just the first n / 2 elements of the row and the other ones are symmetric.

Editorial:
Create a vector with k + 1 positions, all filled with 1. Iterate from 2 to k, including k (start at 2 because row 0 and 1 are [1] and [1, 1], respectively). Start to calculate the new numbers from the end of the new line because we don't want to use extra space. So for any element we perform ans[j] = ans[j] + ans[j - 1]. Calculating from the beginning wouldn't work because we would have modified numbers in our answer that we would need to calculate the next number.

```
vector<int> getRow(int a) {
  vector<int> ans(a + 1);
  int prev = 1;

  for(int i = 0; i <= a; i++){
    ans[i] = prev;
    prev = prev * (a - i) / (i + 1);
  }

  return ans;
}

// vector<int> getRow(int k) {
//   vector<int> ans(k + 1, 1);

//   for (int i = 2; i <= k; i++) {
//     for (int j = i - 1; j > 0; j--) {
//       ans[j] = ans[j] + ans[j - 1];
//     }
//   }

//   return ans;
// }
```

Array Pascal Triangle Rows

Given numRows, generate the first numRows of Pascal’s triangle.
Pascal’s triangle : To generate A[C] in row R, sum up A’[C] and A’[C−1] from previous row R − 1.

Example:
Given numRows = 5,

Return

```
[
     [1],
     [1,1],
     [1,2,1],
     [1,3,3,1],
     [1,4,6,4,1]
]
```

Explanation: the first and last elements of every row are always one. The others you just calculate using the above formula. Just the first row that is [1] needs to be placed manually in your answer, so start with it and then use the formula.

```
vector<vector<int>> generate(int n) {
  vector<vector<int>> ans;

  if (n >= 1) {
    vector<int> row(1, 1);
    ans.push_back(row);
  }

  for (int i = 1; i < n; i++) {
    vector<int> row;
    row.push_back(1);
    for (int j = 1; j <= i - 1; j++) {
      row.push_back(ans[i - 1][j] + ans[i - 1][j - 1]);
    }
    row.push_back(1);
    ans.push_back(row);
  }

  return ans;
}
```

Array Remove Element

Given an array and a value. Remove all the intances of that value from the array and return the new length.

Example:
If the array is [4, 1, 1, 2, 1, 3] and value 1
New legnth will be 3. [4, 2, 3]

Time complexity: O(N), where n is the array size.
Explanation: You need to have a pointer that you increment each time you find an element that is different from value. And each time that occurs you actually assign to the previous pointed element the element you just found that is different from value.

```
int removeElement(vector<int> &a, int b) {
  int count = 0;

  for (int i = 0; i < a.size(); i++) {
    if (a[i] == b) continue;

    a[count] = a[i];
    count++;
  }

  return count;
}
```

Array Rotate Matrix

You are given an n x n 2D matrix representing an image. Rotate the image by 90 degrees (clockwise). You need to do this in place.

Example:

```
If the array is
[
    [1, 2],
    [3, 4]
]
Then the rotated array becomes
[
    [3, 1],
    [4, 2]
]
```

Time complexity: O(N), where N is the number of elements in the matrix.
Space complexity: O(1).

Explanation: they way I initially did was to swap all the elements symmetric in the matrix diagonal, then in another loop swap all the elements symmetric in the line.
There is a easier way. Suppose you have the matrix:

```
a11 a12 a13 a14
a21 a22 a23 a24
a31 a32 a33 a34
a41 a42 a43 a44
```

We can make:

```
#1
temp = a11
a11 = a41
a41 = a44
a44 = a14
a14 = temp

#2
temp = a12
a12 = a31
a31 = a43
a43 = a24
a24 = temp
```

and so on.

To complete the rotation with this process we need to go until line N / 2, not including N / 2, and start at column i (where i is the line) and go until column N - i - 1, not including column N - i - 1. So, in the above example, we would perform this process for lines 0 and 1, and for line 0 we would go from column 0 to 2 (included), and for line 1 we would use just column 1.

```cpp
void rotate(vector<vector<int>> &a) {
  int n = a.size();

  for (int i = 0; i < n / 2; i++) {
    for (int j = i; j < n - i - 1; j++) {
      int temp = a[i][j];
      a[i][j] = a[n - j - 1][i];
      a[n - j - 1][i] = a[n - i - 1][n - j - 1];
      a[n - i - 1][n - j - 1] = a[j][n - i - 1];
      a[j][n - i - 1] = temp;
    }
  }
}

// void rotate(vector<vector<int>> &a) {
//   int n = a.size();

//   // Swap elements symmetric to the diagonal.
//   for (int i = 0; i < n; i++) {
//     for (int j = 0; j < i; j++) {
//       swap(a[i][j], a[j][i]);
//     }
//   }

//   // Swap elements symmetric in a line.
//   for (int i = 0; i < n; i++) {
//     for (int j = 0; j < n / 2; j++) {
//       swap(a[i][j], a[i][n - j - 1]);
//     }
//   }
// }
```

Array Set Zeroes

```
Given an m x n matrix of 0s and 1s, if an element is 0, set its entire row
and column to 0. Do it in place. Note that this will be evaluated on the
extra memory used. Try to minimize the space and time complexity.
```

Example:

```
Given array A as
1 0 1
1 1 1
1 1 1
On returning, the array A should be :
0 0 0
1 0 1
1 0 1
```

Time complexity: O(N), where N is the number of elements in the matrix.
Space complexity: O(1).

Explanation: use the first row and column to store which rows and columns beside the first ones needs to be set to 0. Since you will use the first row and column to store information you need to check if they have 0's before starting to overwrite their information. After, with the information you stored you set the rows and columns to 0. Finally, if you checked that the first row and column used to have zeros you need to also set them to 0.

Notice you don't need to restore the information of the first row/column because if you made one of its elements 0 it is because that row/column needs to be set entirely to 0 including the first row/column.

You could use an intermediary symbol, like the number 2, to mark columns and rows that need to be shifted to 0, but the complexity wouldn't be linear because you would end up repeating the work.

```cpp
void setZeroes(vector<vector<int>> &a) {
  if (a.size() == 0 || a[0].size() == 0) return;

  int nrows = a.size(), ncols = a[0].size();
  bool hasZeroFirstRow = false, hasZeroFirstColumn = false;

  // Does first row have zero?
  for (int j = 0; j < ncols; ++j) {
    if (a[0][j] == 0) {
      hasZeroFirstRow = true;
      break;
    }
  }

  // Does first column have zero?
  for (int i = 0; i < nrows; ++i) {
    if (a[i][0] == 0) {
      hasZeroFirstColumn = true;
      break;
    }
  }

  // find zeroes and store the info in first row and column
  for (int i = 1; i < nrows; ++i) {
    for (int j = 1; j < ncols; ++j) {
      if (a[i][j] == 0) {
        a[i][0] = 0;
        a[0][j] = 0;
      }
    }
  }
```

```
   // set zeroes except the first row and column
   for (int i = 1; i < nrows; ++i) {
     for (int j = 1; j < ncols; ++j) {
       if (a[i][0] == 0 || a[0][j] == 0)  a[i][j] = 0;
     }
   }

   // set zeroes for first row and column if needed
   if (hasZeroFirstRow) {
     for (int j = 0; j < ncols; ++j) a[0][j] = 0;
   }
   if (hasZeroFirstColumn) {
     for (int i = 0; i < nrows; ++i) a[i][0] = 0;
   }
 }
```

Array Spiral Matrix

Given a matrix of m * n elements (m rows, n columns), return all elements of the matrix in spiral order.

Example:
Given the following matrix:

```
[
    [ 1, 2, 3 ],
    [ 4, 5, 6 ],
    [ 7, 8, 9 ]
]
```

You should return: [1, 2, 3, 6, 9, 8, 7, 4, 5]

Time complexity: O(N), where N is the number of elements in the matrix.

Explanation: You have four possible directions you need to follow and in this order: right, bottom, left, top. So we create a variable to control the direction (an integer from 0 to 3). We also have four limits (walls) that we need to update: top = 0, left = 0, bottom = a.size(), right = a[0].size(), being "a" the matrix. Our top and left increased, and our bottom and right decreases until we are at the center of the matrix and top becomes equal bottom and left equal right.
Notice right is equal to a[0].size() and not a.size(), that is because the number of columns may be different from the number of lines.

```
 vector<int> spiralOrder(const vector<vector<int> > &a) {
   vector<int> result;

   if (a.size() == 0 || a[0].size() == 0) return result;

   int t = 0, b = a.size() - 1, l = 0, r = a[0].size() - 1;
   int dir = 0;

   while (t <= b && l <= r) {
     if (dir == 0) {
       for (int i = l; i <= r; i++) result.push_back(a[t][i]);
       t++;
     }
     else if (dir == 1) {
       for (int i = t; i <= b; i++) result.push_back(a[i][r]);
       r--;
     }
```

```
    else if (dir == 2) {
      for (int i = r; i >= l; i--) result.push_back(a[b][i]);
      b--;
    }
    else if (dir == 3) {
      for (int i = b; i >= t; i--) result.push_back(a[i][l]);
      l++;
    }

    dir = (dir + 1) % 4;
  }

  return result;
}
```

Array Spiral Matrix-2

Given an integer n, generate a square matrix filled with elements from 1 to n^2 in spiral order.

Example:

Given n = 3,
You should return the following matrix:
 [[1, 2, 3], [8, 9, 4], [7, 6, 5]]

Time complexity: O(N), where N is the number of elements in the matrix.

Explanation: You have four possible directions you need to follow and in this order: right, bottom, left, top. So we create a variable to control the direction (an integer from 0 to 3). We also have four limits (walls) that we need to update: top = 0, left = 0, bottom = a.size(), right = a[0].size(), being "a" the matrix. Our top and left increased, and our bottom and right decreases until we are at the center of the matrix and top becomes equal bottom and left equal right.
Notice right is equal to a[0].size() and not a.size(), that is because the number of columns may be different from the number of lines.
In this exercise a.size() = a[0].size().

```
vector<vector<int>> generateMatrix(int n) {
  if (n <= 0) return vector<vector<int>>();

  vector<vector<int>> result(n, vector<int>(n, 0));

  int t = 0, l = 0, b = n - 1, r = n - 1;
  int dir = 0, value = 1;

  while (t <= b && l <= r) {
    if (dir == 0) {
      for (int i = l; i <= r; i++) result[t][i] = value++;
      t++;
    }
    else if (dir == 1) {
      for (int i = t; i <= b; i++) result[i][r] = value++;
      r--;
    }
    else if (dir == 2) {
      for (int i = r; i >= l; i--) result[b][i] = value++;
      b--;
    }
    else if (dir == 3) {
      for (int i = b; i >= t; i--) result[i][l] = value++;
      l++;
    }

    dir = (dir + 1) % 4;
  }

  return result;
}
```

Array To Bst

Construct a BST from a sorted array.

Time complexity: O(N), where N is the number of nodes.

Explanation: think about binary search. The principle of a binary search is to visualize an array as a BST. The secret is to get the middle element and insert as a root. Then, the middle element of the right part is the root of the right subtree and the middle element of the left part is the root of the left subtree.

```
TreeNode* _arrayToBst(vector<int> &a, int s, int e) {
  if (s > e) return NULL;

  int middle = (s + e) / 2;

  TreeNode *n = new TreeNode(a[middle]);

  n->left = _arrayToBst(a, s, middle - 1);
  n->right = _arrayToBst(a, middle + 1, e);

  return n;
}

TreeNode* arrayToBst(vector<int> &a) {
  if (a.size() == 0) return NULL;

  TreeNode *root = _arrayToBst(a, 0, a.size() - 1);

  return root;
}

void insert(TreeNode **root, int val) {
  if (*root == NULL) {
    TreeNode *n = new TreeNode(val);
    *root = n;
  }
  else if (val <= (*root)->val) {
    insert(&((*root)->left), val);
  }
  else {
    insert(&((*root)->right), val);
  }
}

TreeNode* search(TreeNode *root, int val) {
  if (root == NULL) return NULL;

  else if (root->val == val) return root;

  else if (val <= root->val) return search(root->left, val);
```

```
    else return search(root->right, val);
  }

  TreeNode* findMin(TreeNode *root) {
    if (root->left == NULL) return root;
    else return findMin(root->left);
  }

  TreeNode* findMax(TreeNode *root) {
    if (root->right == NULL) return root;
    else return findMin(root->right);
  }

  void remove(TreeNode **root, int val) {
    if (*root == NULL) return;
    else if (val < (*root)->val) remove(&((*root)->left), val);
    else if (val > (*root)->val) remove(&((*root)->right), val);
    else {
      if ((*root)->left == NULL && (*root)->right == NULL) {
        delete *root;
        *root = NULL;
      }
      else if ((*root)->left == NULL) {
        TreeNode *temp = *root;
        *root = (*root)->right;
        delete temp;
      }
      else if ((*root)->right == NULL) {
        TreeNode *temp = *root;
        *root = (*root)->left;
        delete temp;
      }
      else {
        TreeNode *min = findMin((*root)->right);
        (*root)->val = min->val;
        remove(&((*root)->right), min->val);
      }
    }
  }

  /*
   * Level order means we print all the nodes in one level, than all the nodes
   * in the following level and so on. To do that just use a queue.
   */
  void levelorder(TreeNode *root) {
    if (root == NULL) return;
    queue<TreeNode *> q;
    q.push(root);

    while (!q.empty()) {
      TreeNode *t = q.front();
      cout << t->val << " ";
      if (t->left != NULL) q.push(t->left);
      if (t->right != NULL) q.push(t->right);
      q.pop();
    }
  }

  /*
   * root, left, right.
   */
  void preorder(TreeNode *root) {
    if (root == NULL) return;

    cout << root->val << " ";
    preorder(root->left);
    preorder(root->right);
  }
```

```
/*
 * left, root, right. The result of this traversal is the nodes in sorted order.
 */
void inorder(TreeNode *root) {
  if (root == NULL) return;

  inorder(root->left);
  cout << root->val << " ";
  inorder(root->right);
}

/*
 * left, right, root.
 */
void postorder(TreeNode *root) {
  if (root == NULL) return;

  postorder(root->left);
  postorder(root->right);
  cout << root->val << " ";
}

/*
 * The height of a tree is the longest path from the root to one of the leaves.
 * Just traverse the tree and return max(left, right) + 1 for each call. Notice,
 * an empty tree (root == NULL) has height equal -1 per definition.
 *
 * And remember that a path consist of edges, so the height of a node is the
 * number of edges from this node to the most distant leaf.
 */
int findHeight(TreeNode *root) {
  if (root == NULL) return -1;

  int left = findHeight(root->left);
  int right = findHeight(root->right);

  return max(left, right) + 1;
}
```

Array Wave

Given an array of integers, sort the array into a wave like array and return it, in other words, arrange the elements into a sequence such that
a1 >= a2 <= a3 >= a4 <= a5.....
Note: If there are multiple answers possible, return the one of the one thats lexicographically smallest. So, in example case, you will return [2, 1, 4, 3].

Example:

Given [1, 2, 3, 4]
One possible answer: [2, 1, 4, 3]
Another possible answer: [4, 1, 3, 2]

Time complexity: O(NlogN), where N is the size of the array. A O(N) solution is possible using the median, though it is not guaranteed to be the lexicographically smallest.
Space complexity: O(1).

Explanation: just sort the array and switch each two neighbors.

Not lexicographically smallest solutions:
Call the median M, half of the array will be smaller or equal to M, and the other half will be larger than M. If we call the smaller elements L (for

```
left) and R (for right) the larger ones we can see an array as something
like:
a = [L, R, R, R, R, L, L, L]
One way to rearrange the elements to obey the wave rule is:
a = [R, L, R, L, R, L, R, L]
So, using the median we can check if the current element E1 is in a position
it could be (R needs to be in even indices and L in odd indices) and if it
is not we just look for the next element that would fit in that place E2.
Then, we swap E1 and E2. Notice, we don't need to go back to the position
where E1 was. We just update our first pointer to position E1 + 2 (plus 2
because the next element to E1 after the swap will definitely be in its
correct position) and we continue to look for suitable elements from the
position of E2.

A better solution there is O(N), but much simpler.
All we need to guarantee is that even positioned elements are bigger than
odd positioned elements. So, we iterate through the even positioned elements
and we do:
1. If current element is smaller than previous odd element, swap previous
and current.
2. If current element is smaller than next odd element, swap next and
current.
```

```cpp
vector<int> wave(vector<int> &a) {
  int n = a.size();

  // Traverse all even elements
  for (int i = 0; i < n; i += 2) {
    // If current even element is smaller than previous
    if (i > 0 && a[i - 1] > a[i]) swap(a[i], a[i - 1]);

    // If current even element is smaller than next
    if (i < n - 1 && a[i] < a[i + 1]) swap(a[i], a[i + 1]);
  }

  return a;
}

// vector<int> wave(vector<int> &a) {
//   int n = a.size(), m = median(a), i = 0, j = 0;

//   while (i < n && j < n) {
//     bool l = i % 2 ? true : false;

//     if (i == j && ((l && a[i] <= m) || (!l && a[i] > m))) {
//       i++;
//       j++;
//     }
//     else {
//       if (l) while (i < n && a[i] > m) i++;
//       else while (i < n && a[i] <= m) i++;

//       if (i < n) {
//         swap(a[i], a[j]);
//         i++;
//         j += 2;
//       }
//     }
//   }

//   return a;
// }

// Lexicographically smallest solution. Sort the away and swap neighbors.
// vector<int> wave(vector<int> &a) {
//   sort(a.begin(), a.end());

//   for (int i = 0; i < a.size() - 1; i += 2) {
```

```
//     swap(a[i], a[i + 1]);
//   }

//   return a;
// }
```

Avl Tree

Impementation of an AVL tree.

```
int height(BstNode *node) {
  if (node == NULL) return -1;
  return node->height;
}

void rotateClockwise(BstNode **root) {
  BstNode *newRoot = (*root)->left;
  (*root)->left = newRoot->right;
  newRoot->right = (*root);

  (*root)->height = 1 + max(height((*root)->left), height((*root)->right));
  newRoot->height = 1 + max(height(newRoot->left), height(newRoot->right));

  *root = newRoot;
}

void rotateCounterClockwise(BstNode **root) {
  BstNode *newRoot = (*root)->right;
  (*root)->right = newRoot->left;
  newRoot->left = (*root);

  (*root)->height = 1 + max(height((*root)->left), height((*root)->right));
  newRoot->height = 1 + max(height(newRoot->left), height(newRoot->right));

  *root = newRoot;
}

void insert(BstNode **root, int data) {
  if (*root == NULL) {
    BstNode *n = new BstNode(data);
    *root = n;
  }
  else if (data <= (*root)->data) {
    insert(&((*root)->left), data);
  }
  else {
    insert(&((*root)->right), data);
  }

  int balance = height((*root)->left) - height((*root)->right);

  if (balance > 1) {
    // LL
    if (height((*root)->left->left) >= height((*root)->left->right)) {
      rotateClockwise(root);
    }
    // LR
    else {
      rotateCounterClockwise(&((*root)->left));
      rotateClockwise(root);
    }
  }
```

```
    else if (balance < -1) {
      // RR
      if (height((*root)->right->right) >= height((*root)->right->left)) {
        rotateCounterClockwise(root);
      }
      // RL
      else {
        rotateClockwise(&((*root)->right));
        rotateCounterClockwise(root);
      }
    }
    else {
      (*root)->height = 1 + max(height((*root)->left), height((*root)->right));
    }
  }

  BstNode* findMin(BstNode *root) {
    if (root->left == NULL) return root;
    else return findMin(root->left);
  }

  BstNode* findMax(BstNode *root) {
    if (root->right == NULL) return root;
    else return findMin(root->right);
  }

  void remove(BstNode **root, int data) {
    if (*root == NULL) return;
    else if (data < (*root)->data) remove(&((*root)->left), data);
    else if (data > (*root)->data) remove(&((*root)->right), data);
    else {
      if ((*root)->left == NULL && (*root)->right == NULL) {
        delete *root;
        *root = NULL;
      }
      else if ((*root)->left == NULL) {
        BstNode *temp = *root;
        *root = (*root)->right;
        delete temp;
      }
      else if ((*root)->right == NULL) {
        BstNode *temp = *root;
        *root = (*root)->left;
        delete temp;
      }
      else {
        BstNode *min = findMin((*root)->right);
        (*root)->data = min->data;
        remove(&((*root)->right), min->data);
      }
    }

    if (*root == NULL) return;

    int balance = height((*root)->left) - height((*root)->right);

    if (balance > 1) {
      // LL
      if (height((*root)->left->left) >= height((*root)->left->right)) {
        rotateClockwise(root);
      }
      // LR
      else {
        rotateCounterClockwise(&((*root)->left));
        rotateClockwise(root);
      }
    }
    else if (balance < -1) {
      // RR
      if (height((*root)->right->right) >= height((*root)->right->left)) {
        rotateCounterClockwise(root);
      }
      // RL
      else {
        rotateClockwise(&((*root)->right));
        rotateCounterClockwise(root);
      }
    }
    else {
      (*root)->height = 1 + max(height((*root)->left), height((*root)->right));
    }
  }

  BstNode* search(BstNode *root, int data) {
    if (root == NULL) return NULL;
    else if (root->data == data) return root;
    else if (data <= root->data) return search(root->left, data);
    else return search(root->right, data);
  }

  void levelorder(BstNode *root) {
    if (root == NULL) return;
    queue<BstNode *> q;
    q.push(root);

    while (!q.empty()) {
      BstNode *t = q.front();
      cout << t->data << " ";
      if (t->left != NULL) q.push(t->left);
      if (t->right != NULL) q.push(t->right);
      q.pop();
    }
  }

  void inorder(BstNode *root) {
    if (root == NULL) return;

    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
  }
```

Backtracking Gray Code

The gray code is a binary numeral system where two successive values differ in only one bit. Given a non-negative integer n representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0.

For example, given n = 2, return [0,1,3,2]. Its gray code sequence is:

```
00 - 0
01 - 1
11 - 3
10 - 2
```

There might be multiple gray code sequences possible for a given n. Return any such sequence.

Time complexity: (2^N), where N is the number of bits
Explanation:

There are two approaches. The iterative one is easier than the recursive one.

Recursive: call the function two times, one when you are setting the bit in a specific position and other when you are not setting this bit. So, the main thought is: replicate all the bits from your last found number in your first call, then in your second call that bit will change to 0 or 1, then you replicate again. Each time you are setting a bit you need a variable to keep your current value and a variable to keep your current power (bit position).

Example:

For n = 2, start with the bits 00:

```
0 0 initially in our vector

0 0 we replicate until we reach the end and start to return

0 1 change one bit, and we are at the end of our positions so we return

1 1 change the bit at the position pointed by the function call we just
    returned and replicate until we reach the end and start to return again

1 0 change one bit, and we are at the end of our positions so we return
```

Iterative:

```
void _grayCode(int curr, int power, vector<int> &ans) {
  if (power == 0) {
    ans.push_back(curr);
    return;
  }

  if ((ans.back() & (1 << (power - 1))) != 0) {
    _grayCode(curr + (1 << (power - 1)), power - 1, ans);
    _grayCode(curr, power - 1, ans);
  }
  else {
    _grayCode(curr, power - 1, ans);
    _grayCode(curr + (1 << (power - 1)), power - 1, ans);
  }
}

vector<int> grayCode(int a) {
  vector<int> ans;

  if (a == 0) return ans;
  ans.push_back(0);

  _grayCode(0, a, ans);
  ans.erase(ans.begin());

  return ans;
}

// vector<int> grayCode(int n) {
//   vector<int> result(1, 0);
//   for (int i = 0; i < n; i++) {
//     int curSize = result.size();
//     // push back all element in result in reverse order
//     for (int j = curSize - 1; j >= 0; j--) {
//       result.push_back(result[j] + (1 << i));
//     }
//   }
//   return result;
// }
```

Backtracking N Queens

```
Given an integer n, return all distinct solutions to the n-queens puzzle.
Observation: the queen can move forward, backward, sideways and diagonally.

For example,
There exist two distinct solutions to the 4-queens puzzle:
[
  [".Q..",  // Solution 1
   "...Q",
   "Q...",
   "..Q."],

  ["..Q.",  // Solution 2
   "Q...",
   "...Q",
   ".Q.."]
]
```

Time complexity: O(N^N to C*N!), where N is the board size.

Explanation: we will have N lines in our chess board. Each queen needs to be at one of these lines, and no more than one can be at the same line. So we iterate over all columns and we keep track of the line we are currently at with a variable in our function signature. While we are iterating over the columns we try to fit the queen in that particular position (notice, we need to have a function that is able to do that based on the current state of our board) and when the queen can be positioned there we call our function again with our new board state. When our function returns it is important that we reinstall the previous state of the board in that call so our loop can try to find a new position for that queen.

```
string emptyLine(int n) {
  char *c = new char[n + 1];
  int i;

  for (i = 0; i < n; i++) {
    c[i] = '.';
  }
  c[i] = '\0';

  return string(c);
}

bool check(int n, int c, vector<string> &curr) {
  int line = curr.size();

  if (line == 0) return true;
  if (line >= n) return false;

  for (int i = 0; i < curr.size(); i++) {
    int ld = c - line + i;
    int rd = c + line - i;

    if ((ld >= 0 && curr[i][ld] == 'Q') || (rd < n && curr[i][rd] == 'Q') ||
      curr[i][c] == 'Q') {
      return false;
    }
  }

  return true;
}

void solve(int n, int l, vector<string> &curr, vector<vector<string>> &ans) {
  if (l >= n) {
    ans.push_back(curr);
    return;
  }

  for (int c = 0; c < n; c++) {
    if (check(n, c, curr)) {
      string line = emptyLine(n);
      line[c] = 'Q';

      curr.push_back(line);
      solve(n, l + 1, curr, ans);
      curr.pop_back();
    }
  }
}

vector<vector<string>> solveNQueens(int n) {
  vector<vector<string>> ans;
  vector<string> curr;

  if (n > 0) solve(n, 0, curr, ans);

  return ans;
}
```

Backtracking Permutations

Given a collection of numbers, return all possible permutations.

Example:
[1,2,3] will have the following permutations:

```
[1,2,3]
[1,3,2]
[2,1,3]
[2,3,1]
[3,1,2]
[3,2,1]
```

Time complexity: O(N!), where N is the size of the given list.

Explanation: for each number in our list we let it assume the position our current function call is dealing with. Then, we call again our function, but we increment the position so the number we just put in that position will not be swapped in this next function call. When we come back from a function is important we swap back the number to its original position in this function call we are now at.

```
void _permute(vector<int> &c, int s, vector<vector<int>> &ans) {
  if (s == c.size() - 1) {
    ans.push_back(c);
    return;
  }

  for (int i = s; i < c.size(); i++) {
    swap(c[s], c[i]);
    _permute(c, s + 1, ans);
    swap(c[s], c[i]);
  }
}

vector<vector<int>> permute(vector<int> &a) {
  vector<vector<int>> ans;

  if (a.size() == 0) return ans;

  sort(a.begin(), a.end());
```

```
    _permute(a, 0, ans);

    return ans;
  }

  // vector<vector<int>> _permute(vector<vector<int>> &c, int a) {
  //   vector<vector<int>> r;
  //   int i, j, k;

  //   for (i = 0; i < c.size(); i++) {
  //     for (j = 0; j < c[i].size() + 1; j++) {
  //       vector<int> t;
  //       for (k = 0; k < c[i].size(); k++) {
  //         if (k == j) t.push_back(a);
  //         t.push_back(c[i][k]);
  //       }
  //       if (k == j) t.push_back(a);
  //       r.push_back(t);
  //     }
  //   }

  //   return r;
  // }

  // vector<vector<int>> permute(vector<int> &a) {
  //   vector<vector<int>> ans;
  //   vector<int> t;

  //   if (a.size() == 0) return ans;

  //   ans.push_back(t);
  //   for (int i = 0; i < a.size(); i++) {
  //     ans = _permute(ans, a[i]);
  //   }
  //   sort(ans.begin(), ans.end());

  //   return ans;
  // }
```

Binary Search Allocate Books

```
N number of books are given. The ith book has Pi number of pages. You have
to allocate books to M number of students so that maximum number of pages
alloted to a student is minimum. All books need to be allocated. A book will
be allocated to exactly one student. Each student has to be allocated at
least one book. Allotment should be in contiguous order, for example: A
student cannot be allocated book 1 and book 3, skipping book 2.

Notes:
1. Return -1 if a valid assignment is not possible
2. Your function should return an integer corresponding to the minimum
number.
```

Example:

```
P = [12, 34, 67, 90]
M = 2

Output 113

There are 2 number of students. Books can be distributed in following
fashion:
1. [12] and [34, 67, 90]
Max number of pages is allocated to student 2 with 34 + 67 + 90 = 191 pages
```

```
2. [12, 34] and [67, 90]
Max number of pages is allocated to student 2 with 67 + 90 = 157 pages
3. [12, 34, 67] and [90]
Max number of pages is allocated to student 1 with 12 + 34 + 67 = 113 pages

Of the 3 cases, Option 3 has the minimum pages = 113.
```

Time complexity: O(logB * N), where B is the number of bits for the variable used to hold the sum of the pages and N is the number of elements in the given array.
Space complexity: O(1).

Explanation: the way this exercise is formulated it is difficult to see what it is really asking. You are not supposed to distribute the books, but to find out what is the minimum sum for the array partition with the maximum sum. So, in other words, we want to find:
Given a number P of pages is it possible with the number of students that was given for every student to read at most P pages, and if it is can we make P smaller?
So, if we have the total number of pages T (a1 + a2 + ... + ai). We can look for P using binary search:

```
     mid = (0 + T) / 2
0 .........................T
```

Now, if we are able to calculate how many student we need to read mid pages, and we find out that is a number less or equal to the given number of students M, then of course any number larger than mid will require the same number of students or even less. So we can discard every number larger than M because we are interested in the minimum number of pages. When we find the smallest possible mid, then we found our answer.

How to find out if the given number of students is enough to read mid pages? Since the distribution needs to be contiguous we just sum the pages in the array until it gets larger than mid. At this point we need to increment the number of students because a student can read at most mid pages, so if it is more than that we need one more student. Now, we start our sum again and perform the same thing. In the end, if the number we got is less than or equal to the given number it is possible that with at most mid pages the students can read everything so we try to make mid smaller.

```cpp
bool isPossible(vector<int> &a, int b, int mid) {
  int sum = 0, count = 1;

  for (int i = 0; i < a.size(); i++) {
    if (a[i] > mid) return false;

    if (sum + a[i] > mid) {
      count++;
      sum = a[i];
      if (count > b) return false;
    }
    else {
      sum += a[i];
    }
  }

  return true;
}

int books(vector<int> &a, int b) {
  if(a.size() < b) return -1;

  int ans = INT_MAX;
  long long low = 0, high = 0, mid;

  for (int i = 0; i < a.size(); i++) high += a[i];
```

```
    while (low <= high) {
      mid = (high + low) / 2;

      if (isPossible(a, b, mid)) {
        ans = min(ans, (int)mid);
        high = mid - 1;
      }
      else {
        low = mid + 1;
      }
    }

    return ans;
  }
```

Binary Search Find Rotated Array

```
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 ).
You are given a target value to search. If found in the array, return its
index, otherwise return -1.

Notes:
1. You may assume no duplicate exists in the array.
2. Think about the case when there are duplicates. Does your current solution
work? How does the time complexity change?
```

Example:

```
Input = [4 5 6 7 0 1 2] and target = 4
Output = 0
```

Time complexity: O(logN), where N is the size of the input array.
Space complexity: O(1).

Explanation: the trick is to use a normal binary search but "rotate" the just calculated middle point. So, we find the rotation point (the index of the minimum element in the array) and after we perform a normal binary search, with normal low, high and middle points, but when we will efective access a the middle element we add the rotation index:
a[mid] becomes a[(mid + n) % n].

Notice that finding the rotation point (minimum element) is also a O(logN) operation. You can find details of how to write this algorithm in the code below.

```
  // Find the minimum element in a sorted array using binary search.
  int findMin(vector<int> &a) {
    int n = a.size(), low = 0, high = n - 1;

    while (low <= high) {
      // If the low element is smaller than the high element so the piece of array
      // we are current looking at is not rotated so we can return low as the
      // minimum element.
      if (a[low] <= a[high]) return low;

      int mid = (low + high) / 2;
      int prev = (mid - 1 + n) % n;
      int next = (mid + 1) % n;

      // If the previous and next element are smaller than the middle one we
      // found the minimum element in the array.
      if (a[mid] <= a[prev] && a[mid] <= a[next]) return mid;

      // If middle element is smaller than high element so we know that all
      // elements after the middle one will continue to increase. So, we discard
      // this part.
      if (a[mid] <= a[high]) high = mid - 1;
      // If the middle element is larger than low element (and we know the array
      // is rotated and we also know we still not found the minimum element), so
      // we can affirm the minimum element will be behind the middle element.
      else if (a[mid] >= a[low]) low = mid + 1;
    }

    return -1;
  }

  int search(vector<int> &a, int b) {
    int n = a.size(), low = 0, high = n - 1, rotation = findMin(a);

    while (low <= high) {
      int mid = (low + high) / 2;

      if (a[(mid + rotation) % n] == b) return (mid + rotation) % n;

      if (a[(mid + rotation) % n] < b) low = mid + 1;
      else high = mid - 1;
    }

    return -1;
  }
```

Binary Search Median Two Arrays

There are two sorted arrays A and B of size M and N respectively. Find the median of the two sorted arrays (The median of the array formed by merging both the arrays).

The overall run time complexity should be O(log(M + N)).

Notes:
1. If the number of elements in the merged array is even, then the median is the average of N / 2 th and N / 2 + 1th element. For example, if the array is [1 2 3 4], the median is (2 + 3) / 2.0 = 2.5

Example:

A = [1 4 5]
B = [2 3]

Output = 3

Time complexity: O(log(M + N)), where M is the size of one array and N is the size of the other array.
Space complexity: O(1), in iterative implementation, O(log(M + N)) using recursion.

Explanation:
We want to find the Kth smallest element. For that, we basically want to find a position i in the first array, and a position j in the second array, so that i + j = k - 1. Since at position i we have i elements smaller than A[i], and at position j we have j elements smaller than B[j]. We need to maintain this property in our iterations.

To solve this problem we can look at the likely Kth smallest elements, which will be at position (A_start|B_start) + i, where i = k / 2, because k / 2

```
elements from the first array plus k / 2 elements of the second array
we respect the property i + j = k - 1.

In practice we can only guaratee that
for k, we have k / 2 elements behind min(A[A_start + i], B[B_start + i]), so
we find k / 2 smallest elements at each iteration:

1. A[A_start + i] < B[B_start + i], then we can discard all elements in A
with indices smaller than i, because A[A_start + i] could be at most the
(k - 1)th smallest element. And we can discard all elements in B at positions
larger than i, since B[B_start + i] will be the kth smallest element or the
(k + x)th smallest element for some x > 0.
2. B[B_start + i] < A[A_start + i], same as above, but change A with B.

So, we are trying to find the Kth smallest element for each k / 2 partition,
where k / 2 is updated recursively, like (((k / 2) / 2) / 2) and so on. This
show us that our time complexity will be O(log(k)) and k can be at most
M + N, so we get O(log(M + N)).

For example, given,
A = [-50, -21, -10]
B = [-50, -41, -40, -19, 5, 21, 28]

We define findKth(A, As, B, Bs, k),
Where A and B are the input arrays, As and Bs are integers that indicate
where we jound start our search in A and B respectively, and k indicates
which kth smallest element we are searching.

We get,
M = 3, N = 7

M + N is even so we need to look for two k's,
k = (M + N) / 2 = 5, k = (M + N) / 2 + 1 = 6

Lets take a look at case k = 6

#1
As = 0, Bs = 0, k = 6
i = k / 2 - 1 = 2 (the -1 comes from the fact that we use 0 based indices)
A[As + i] = -10, B[Bs + i] = -40
B[Bs + i] < A[As + i], so we discard in B, every element up to B[i], and
every element larger than A[As + i], doing k - k / 2 we guarantee that we
will not go further A[As + i], and doing Bs + k / 2 we start our searching
for B in the next iteration beyond B[Bs + i]. Notice that k - k / 2 is not
necessary equal k / 2, it can be also k / 2 + 1, which will be the correct
value for odd k.

#2
As = 0, Bs = 3, k = 3
i = k / 2 - 1 = 0
A[As + i] = -50, B[Bs + i] = -19
A[As + i] < B[Bs + i], so we discard elements up to A[As + i], and elements
after B[Bs + i].

#3
As = 1, Bs = 3, k = 2
i = k / 2 - 1 = 0
A[As + i] = -21, B[Bs + i] = -19
A[As + i] < B[Bs + i], so we discard elements up to A[As + i], and elements
after B[Bs + i].

#4
As = 2, Bs = 3, k = 1
k = 1, so we return min(A[As], B[Bs]), which is -19

Editorial Solution
This problem will use binary search to find the median and avoid
merging the array which would be a linear operation.
```

```
If we can divide both arrays, one at point i and the other at point j, in
a way that all the elements in the left part (left part of first array plus
left part of the second array) are smaller than all the elements in the
right part (right part of first array plus right part of the second array),
so we can find the median as one of the middle left extremes of these parts:

Left           Right
[x1 x2 x3]    [x4 x5]
[y1 y2]       [y3 y4]

If all elements in left are smaller than all elements in right, then the
median needs to be one of x3, y2. That is obvious if you think that this is
exactly what a median does: divies the array into two equal parts, left and
right, where all elements in the left part are smaller than all elements in
the right part (notice the left part can have one more element than the
right part).

The trick is that we can find i with binary search. Once, we have i, j is
just a calculation because for each i we need to guarantee j divides the
second array in a way that left(i) + left(j) = right(i) + right(j), where
left(i or j) and right(i or j) are the number of elements in the left/right
part for a division happening at point i/j. Remembering that actually the
left part can have one more element than the right part if the total number
of elements is odd, we add 1 to the right part to make things equal (and
add 1 in the "even" case will "floor" to the correct value if we didn't have
added 1, because 1 / 2 will give 0.5):

Right 1      Right 2            Left 1          Left 2
(M - i)   +   (N - j)   +   1   =   i       +       j

Which simplifies to j = ((M + N + 1) / 2) - i

Now, the question that emerges is: why we can use binary search?
Since the arrays are sorted to check if we have our condition that all
elements in the left part are smaller than all the elements in the right part
we just need to check the elements in the division indices i and j as:

x4 > y2 and y3 > x3

If this condition is true so all the following elements after y3 and x4
will be larger than x3 and y2 since the arrays are sorted. Now, if the
condition is false it means that for that particular i and j, or x4 is
is smaller than y2 or y3 is smaller than x3, and we can get:

1. x4 < y2, means we can discard all indices smaller than i, because if x4
is smaller than y2, so will be x3, x2, x1.
2. y3 < x3, means we can discard all indices larger than i, because if y3
is smaller than x3 , so it will be smaller than x4 and x5 too.

We just defined our binary search, but there are a couple of things that I
will highlight because they will matter for the implementation:

1. Be sure N is larger than M, because in the equation
j = ((m + n + 1) / 2) - i, (m + n + 1) needs to be larger than i, otherwise
when you subtract i you could get a negative j.
2. i = 0 means the left part of the first array is empty, beacause i means
the amount of elements in the left part. So, m - i, is the amount of elements
in the right part. Same thing goes for j.
3. The median when M + N is even will always be: max(A[i - 1], B[j - 1]).
4. The median when M + N is odd will always be:
max(A[i - 1], B[j - 1]) + min(A[i], B[j])).
```

```cpp
// This function can also be iterative.
int findKth(const vector<int> &A, int A_start, const vector<int> &B, int B_start, i
  // If we don't have any more elements to look at A, find kth directly in B.
  if (A_start >= A.size()) return B[B_start + k - 1];
  // If we don't have any more elements to look at B, find kth directly in A.
  if (B_start >= B.size()) return A[A_start + k - 1];
```

```
    // Our base case.
    if (k == 1) return min(A[A_start], B[B_start]);

    // Our indices is 0 based, but our kth element is 1 based, so we subtract 1.
    int A_key = A_start + k / 2 - 1 < A.size() ? A[A_start + k / 2 - 1] : INT_MAX;
    int B_key = B_start + k / 2 - 1 < B.size() ? B[B_start + k / 2 - 1] : INT_MAX;

    if (A_key < B_key) {
      // If A_key < B_key we can discard all elements from 0 to A_key in A and
      // we can discard all elements larger than B_key in B.
      // Notice that k - k / 2 is not necessarily equal to k / 2, it can also be
      // k / 2 + 1 which is the right choice for odd k.
      return findKth(A, A_start + k / 2, B, B_start, k - k / 2);
    }
    else {
      // Same thing as above but we switch A with B.
      return findKth(A, A_start, B, B_start + k / 2, k - k / 2);
    }
  }

  double findMedianSortedArrays(const vector<int> &A, const vector<int> &B) {
    int len = A.size() + B.size();

    if (len % 2 == 1) {
      return findKth(A, 0, B, 0, len / 2 + 1);
    }
    else {
      return 0.5 * (findKth(A, 0, B, 0, len / 2) +
        findKth(A, 0, B, 0, len / 2 + 1));
    }
  }

  // double findMedianSortedArrays(const vector<int> &A, const vector<int> &B) {
  //   int m = A.size(), n = B.size();

  //   // Be sure array B is larger than array A. In the equation
  //   // j = (m + n + 1) / 2 - i, (m + n + 1) needs to be larger than i, otherwise
  //   // when you subtract i you could get a negative j.
  //   if (m > n) return findMedianSortedArrays(B, A);

  //   int low = 0, high = m;

  //   while (low <= high) {
  //     int i = (low + high) / 2, j = (m + n + 1) / 2 - i;

  //     if (j > 0 && i < m && B[j - 1] > A[i]) low = i + 1;
  //     else if (i > 0 && j < n && A[i - 1] > B[j]) high = i - 1;
  //     else {
  //       // We met our condition that all elements elements in the left part needs
  //       // to be smaller than all the elements in the right part, so lets find
  //       // the median.
  //       int median1 = 0, median2 = 0;

  //       if (i == 0) median1 = B[j - 1];
  //       else if (j == 0) median1 = A[i - 1];
  //       else median1 = max(A[i - 1], B[j - 1]);

  //       // If the number of elements in the array is odd our median is the middle
  //       // element.
  //       if ((m + n) % 2 == 1) return 1.0 * median1;

  //       // If the number of elements in the array is even our median is the
  //       // average of median1 and the element next to it.
  //       if (i == m) median2 = B[j];
  //       else if (j == n) median2 = A[i];
  //       else median2 = min(A[i], B[j]);

  //       return 1.0 * (median1 + median2) / 2.0;
```

```
//     }
//   }

//   // If the median does not exist (empty arrays).
//   return -1.0;
// }
```

Binary Search Tree

Implementation of a Binary Search Tree.

```
void insert(TreeNode **root, int val) {
  if (*root == NULL) {
    TreeNode *n = new TreeNode(val);
    *root = n;
  }
  else if (val <= (*root)->val) {
    insert(&((*root)->left), val);
  }
  else {
    insert(&((*root)->right), val);
  }
}

TreeNode* search(TreeNode *root, int val) {
  if (root == NULL) return NULL;
  else if (root->val == val) return root;
  else if (val <= root->val) return search(root->left, val);
  else return search(root->right, val);
}

TreeNode* findMin(TreeNode *root) {
  if (root->left == NULL) return root;
  else return findMin(root->left);
}

TreeNode* findMax(TreeNode *root) {
  if (root->right == NULL) return root;
  else return findMin(root->right);
}

void remove(TreeNode **root, int val) {
  if (*root == NULL) return;
  else if (val < (*root)->val) remove(&((*root)->left), val);
  else if (val > (*root)->val) remove(&((*root)->right), val);
  else {
    if ((*root)->left == NULL && (*root)->right == NULL) {
      delete *root;
      *root = NULL;
    }
    else if ((*root)->left == NULL) {
      TreeNode *temp = *root;
      *root = (*root)->right;
      delete temp;
    }
    else if ((*root)->right == NULL) {
      TreeNode *temp = *root;
      *root = (*root)->left;
      delete temp;
    }
    else {
      TreeNode *min = findMin((*root)->right);
      (*root)->val = min->val;
      remove(&((*root)->right), min->val);
    }
  }
}

/*
 * Level order means we print all the nodes in one level, than all the nodes
 * in the following level and so on. To do that just use a queue.
 */
void levelorder(TreeNode *root) {
  if (root == NULL) return;
  queue<TreeNode *> q;
  q.push(root);

  while (!q.empty()) {
    TreeNode *t = q.front();
    cout << t->val << " ";
    if (t->left != NULL) q.push(t->left);
    if (t->right != NULL) q.push(t->right);
    q.pop();
  }
}

/*
 * root, left, right.
 */
void preorder(TreeNode *root) {
  if (root == NULL) return;

  cout << root->val << " ";
  preorder(root->left);
  preorder(root->right);
}

/*
 * left, root, right. The result of this traversal is the nodes in sorted order.
 */
void inorder(TreeNode *root) {
  if (root == NULL) return;

  inorder(root->left);
  cout << root->val << " ";
  inorder(root->right);
}

/*
 * left, right, root.
 */
void postorder(TreeNode *root) {
  if (root == NULL) return;

  postorder(root->left);
  postorder(root->right);
  cout << root->val << " ";
}

/*
 * The height of a tree is the longest path from the root to one of the leaves.
 * Just traverse the tree and return max(left, right) + 1 for each call. Notice,
 * an empty tree (root == NULL) has height equal -1 per definition.
 *
 * And remember that a path consist of edges, so the height of a node is the
 * number of edges from this node to the most distant leaf.
 */
int findHeight(TreeNode *root) {
  if (root == NULL) return -1;
  int left = findHeight(root->left);
  int right = findHeight(root->right);

  return max(left, right) + 1;
}
```

Binary Tree All Paths

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum is equal to the given sum.

Time complexity: O(N), where N is the number of nodes.

Explanation: first we need to think how we know we are at the end of a path and that is accomplished checking if both children of a node are NULL. Second, we need to traverse all the paths and we gonna use recursion for that. I choose to first go to the left branch and after to the right branch. At each node we visit we store it in a vector that we use to store our path. We, then, keep a variable of the remaining sum when adding that node to the path (remainingSum = sum - node->val) and call the function for the left and right branchs with remainingSum as input of these calls.
If we reach a leaf node (both children NULL) we store out path in another vector that stores all our paths (so it is a vector of vectors).
When we come back from both of our calls (left and right branch) we pop the node we stored in our vector path at that call. Notice we don't pop our nodes after the left branch call, but after the right branch call (or in another words after both of the calls). This is because the right path can potentially use nodes stored by the left branch calls.

```
void _allPaths(TreeNode *root, int sum, vector<int> &curr, vector<vector<int>> &ans
  if (root == NULL) return;

  curr.push_back(root->val);

  // We want our sum to end in a leaf, so both children must be NULL.
  if (root->left == NULL && root->right == NULL) {
    if (sum == root->val) ans.push_back(curr);
  }
  else {
    int remainingSum = sum - root->val;

    _allPaths(root->left, remainingSum, curr, ans);
    _allPaths(root->right, remainingSum, curr, ans);
  }

  curr.pop_back();
}

vector<vector<int>> allPath(TreeNode *root, int sum) {
  vector<vector<int>> ans;
  vector<int> curr;

  _allPaths(root, sum, curr, ans);

  return ans;
}
```

Binary Tree Build Inorder Postorder

Given inorder and postorder traversal of a tree, construct the binary tree.
Note: You may assume that duplicates do not exist in the tree.

Time complexity: O(N), where N is the number of nodes.

Explanation: the last element of the postorder traversal is the root of the entire tree. So, of course, all the elements before the last element in the postorder vector are children of the root/last element. Now, if we identify how many of these reamining elements are the left children and how many are the right children we can discover which of them are the roots of left and right subtree. The root of the left subtree would be the last element of all the left children (so if we know the number of left children we just count from the beginning of the array to get the last left element). We do the same thing for the right subtree starting our count after the last left element (again supposing we know the number of right children).
We discover the number of left and right children using our inorder traversal. From the beginning of the inorder traversal we start to count and when we find our root element we know the number of left children. Now, we start we get how many elements we have after the element we just found to be our root and we till the end of our inorder vector and then we have the number of right children.
Repeat the process above for left and right subtree updating variables that you keep to indicate the portion of the postorder and inorder vector you want to investigate.

This question has bonus points if you solve without recursion. So you should think about that. You can find the code for the non recursive approach below:

```
// interviewbit.com
TreeNode *buildTree(vector &inorder, vector &postorder) {
  if(inorder.size() == 0)return NULL;
  TreeNode* p;
  TreeNode* root;
  vector vint;
  vector vtn;
  root = new TreeNode(postorder.back());
  vtn.push_back(root);
  postorder.pop_back();
  while (true) {
    if(inorder.back() == vtn.back()->val) {
      p = vtn.back();
      vtn.pop_back();
      inorder.pop_back();
      if(inorder.size() == 0) break;
      if(vtn.size())
          if(inorder.back() == vtn.back()->val)continue;
      p->left = new TreeNode(postorder.back());
      postorder.pop_back();
      vtn.push_back(p->left);
    }
    else {
      p = new TreeNode(postorder.back());
      postorder.pop_back();
      vtn.back()->right = p;
      vtn.push_back(p);
    }
  }
  return root;
}
```

```
void _buildTree(vector<int> &inorder, vector<int> &postorder, int s, int e, int x,
  if (s >= e || root == NULL) return;

  int i;
```

```
    for (i = s; i <= e; i++) {
      if (inorder[i] == postorder[x]) break;
    }

    int leftSize = i - s;
    int rightSize = e - i;

    if (leftSize > 0) root->left = new TreeNode(postorder[x - rightSize - 1]);
    if (rightSize > 0) root->right = new TreeNode(postorder[x - 1]);

    _buildTree(inorder, postorder, s, i - 1, x - rightSize - 1, root->left);
    _buildTree(inorder, postorder, i + 1, e, x - 1, root->right);
  }

  TreeNode* buildTree(vector<int> &inorder, vector<int> &postorder) {
    if (postorder.size() == 0 || postorder.size() != inorder.size()) return NULL;

    TreeNode *root = new TreeNode(postorder.back());

    _buildTree(inorder, postorder, 0, postorder.size() - 1, postorder.size() - 1, roo

    return root;
  }
```

Binary Tree Build Inorder Preorder

Given inorder and preorder traversal of a tree, construct the binary tree.
Note: You may assume that duplicates do not exist in the tree.

Time complexity: O(N), where N is the number of nodes.

Explanation: the first element of the preorder traversal is the root of the entire tree. So, of course, all the elements after the first element in the preorder vector are children of the root/last element. Now, if we identify how many of these reamining elements are the left children and how many are the right children we can discover which of them are the roots of left and right subtree. The root of the left subtree would be the first element of all the left children, so the element just after the root. And the root of the right subtree will be the first element after the last left child.
We discover the number of left and right children using our inorder traversal. From the beginning of the inorder traversal we start to count and when we find our root element we know the number of left children. Now, we start we get how many elements we have after the element we just found to be our root and we till the end of our inorder vector and then we have the number of right children.
Repeat the process above for left and right subtree updating variables that you keep to indicate the portion of the preorder and inorder vector you want to investigate.

```
  void _buildTree(vector<int> &inorder, vector<int> &preorder, int s, int e, int x, T
    if (s >= e || root == NULL) return;

    int i;

    for (i = s; i <= e; i++) {
      if (inorder[i] == preorder[x]) break;
    }

    int leftSize = i - s;
    int rightSize = e - i;
    if (leftSize > 0) root->left = new TreeNode(preorder[x + 1]);
    if (rightSize > 0) root->right = new TreeNode(preorder[x + leftSize + 1]);

    _buildTree(inorder, preorder, s, i - 1, x + 1, root->left);
    _buildTree(inorder, preorder, i + 1, e, x + leftSize + 1, root->right);
  }

  TreeNode* buildTree(vector<int> &inorder, vector<int> &preorder) {
    if (preorder.size() == 0 || preorder.size() != inorder.size()) return NULL;

    TreeNode *root = new TreeNode(preorder.front());

    _buildTree(inorder, preorder, 0, preorder.size() - 1, 0, root);

    return root;
  }
```

Binary Tree Equality

Given two binary trees, write a function to check if they are equal or not. Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

Time complexity: O(N).

Explanation: left and right subtrees must be equal.

```
 bool equal(TreeNode *r1, TreeNode *r2) {
   if (r1 == NULL && r2 == NULL) return true;
   if (r1 == NULL || r2 == NULL) return false;

   if (r1->val != r2->val) {
     return false;
   }
   else {
     bool l = equal(r1->left, r2->left);
     bool r = equal(r1->right, r2->right);

     return l && r;
   }
 }

 /*
  * Check if two trees are equal (subtrees can be equal to their mirror).
  *
  * Time complexity: O(N), where N is the number of nodes.
  *
  * Explanation: left and right subtrees must be equal, but now the subtree can
  * be equal to their mirrors, so this other case is covered with:
  * t1->left == t2->right && t1->right == t2->left.
  */
 bool equalMirror(TreeNode *r1, TreeNode *r2) {
   if (r1 == NULL && r2 == NULL) return true;
   if (r1 == NULL || r2 == NULL) return false;

   if (r1->val != r2->val) {
     return false;
   }
   else {
     return (equal(r1->left, r2->left) && equal(r1->right, r2->right)) ||
       (equal(r1->left, r2->right) && equal(r1->right, r2->left));
   }
 }
```

Binary Tree Lowest Ancestor

Find the lowest common ancestor of a binary tree (not a binary search tree).

Time complexity: O(N), where N is the number of nodes.
Space complexity: O(logN), because of the tree height.

Explanation:
Supposing both of the nodes are present in the tree we can traverse the tree checking if you found one of the nodes you are looking for. If you found return this node. Then, you have three cases:

1. When a function receives NOT NULL from left subtree and NOT NULL from right subtree so the node of that function execution is the ancestor.

2. When the function receives NOT NULL from one of the subtrees (left or right) and NULL from the other, so the current function execution just return the NOT NULL node it received, because this node is one of the nodes we were looking for and it potentially may be the ancestor of itself and the other node.

3. When the function execution receives NULL from both subtrees so we return NULL.

Notice, this algorithm also works for the case one of the nodes we are looking for is the lowest common ancestor (look at case 2 above).

If the tree may not contain both nodes so we can use a find function to check if the nodes exist. The time and space complexity would remain the same, but in reality we would traverse the tree three times (one to find node N1, another to find node N2 and another to check the ancestor). The other option is to use some variables to indicate if we found both nodes. We also continue to traverse the current branch tree when we find one, but still did not find the other.

In a interview you should stick with finding the nodes before applying the LCA algorithm because it is easier, but maybe would be interesting to point that is possible to traverse the tree just one time.

```
TreeNode* lca(TreeNode *root, int n1, int n2) {
  if (root == NULL) return NULL;
  if (root->val == n1 || root->val == n2) return root;

  TreeNode *left = lca(root->left, n1, n2);
  TreeNode *right = lca(root->right, n1, n2);

  if (left != NULL && right != NULL) return root;
  if (left == NULL && right == NULL) return NULL;

  return left != NULL ? left : right;
}

// Lowest ancestor modified to accept the case when the values not necessarily
// exist in the tree.
// int _lca(TreeNode *root, int n1, int n2, TreeNode **f1, TreeNode **f2, int t) {
//   if (root == NULL) return t;
//   if (root->val == n1) {
//     // Store N1, so we know we already found it.
//     *f1 = root;
//     // In case we already found N2, we have two options: N2, is the
//     // ancestor of N1 or it is not. If it is not, the actual ancestor will
//     // receive NOT NULL (actually NOT -1) from both sides and return itself,
//     // if it is we will continue to return N2 till the end.
//     if (*f2 != NULL) return (*f2)->val;
//     // If we didn't find N2, then we need to store N1 in a temp variable,
//     // because we need to keep traversing this branch since N2 can be under
//     // N1, but if it is not we eventually will have root == NULL and in this
//     // case we need to return N1 all the way up. Notice the "t" variable is
//     // copied in all calls so just nodes under N1 will see "t" as N1->val.
//     t = root->val;
//   }
//   if (root->val == n2) {
//     // Same logic as above.
//     *f2 = root;
//     if (*f1 != NULL) return (*f1)->val;
//     t = root->val;
//   }

//   int left = _lca(root->left, n1, n2, f1, f2, t);
//   int right = _lca(root->right, n1, n2, f1, f2, t);

//   if (left != -1 && right != -1) return root->val;
//   if (left == -1 && right == -1) return -1;

//   return left != -1 ? left : right;
// }

// int lca(TreeNode *root, int n1, int n2) {
//   TreeNode *f1 = NULL, *f2 = NULL;

//   int lcaValue = _lca(root, n1, n2, &f1, &f2, -1);

//   if (f1 && f2) return lcaValue;

//   return -1;
// }
```

Binary Tree Max Depth

Given a binary tree, find its maximum depth.

Time complexity: O(N), where N is the number of nodes.

Explanation: the height of a tree is the longest path from the root to one of the leaves. And the height of tree is equal to its maximum depth. Just traverse the tree and return max(left, right) + 1 for each call.

This problem is simple and it can be given as a warmup to the minimum depth of a binary tree problem that is a slightly more complicated problem.

```
int maxDepth(TreeNode *root) {
  if (root == NULL) return 0;

  int left = maxDepth(root->left);
  int right = maxDepth(root->right);

  return max(left, right) + 1;
}
```

Binary Tree Min Depth

Given a binary tree, find its minimum depth.
Note 1: The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.
Note 2: the path has to end in a left node.

Time complexity: O(N), where N is the number of nodes.

Explanation: you must return min(root->left, root->right). The only problem is what you do when left or right subtree is NULL. For example:

```
1
 \
  2
```

Has minimum depth 2 and not 1 (we are counting nodes and not edges here because the exercise stated depth as nodes and not edges, also stated paths needs to terminate in a leaf node). If you just return min(root->left, root->right) you would get 1 as answer. So, to address this corner case we identify that we are in a NULL node returning 0 from it and then we can do "if (left == 0 && right != 0) return right + 1;" (same when left != 0).

```
int minDepth(TreeNode *root) {
  if (root == NULL) return 0;

  int left = minDepth(root->left);
  int right = minDepth(root->right);

  if (left == 0 && right != 0) return right + 1;
  else if (left != 0 && right == 0) return left + 1;
  else return min(left, right) + 1;
}
```

Binary Tree Next Pointers

Given a binary tree:

```
struct TreeLinkNode {
  TreeLinkNode *left;
  TreeLinkNode *right;
  TreeLinkNode *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Note: each "pointer" is already initialized with NULL.

Explanation: the secret is to use the previous level that you just finished completing the "next" pointers to complete the next "next" pointers.
So, you start with the root and you make the next pointer of the left child point to the right child. Then, you move to the next level and you get the nodes of this next level traversing the previous one using the next pointers you just filled.

In the code below I make the "next" pointer of the left child point to the right child in one iteration. Then in this same iteration I store in a variable the address of the "next" pointer of the right child if this right child is not NULL, otherwise I store the address of the "next" pointer of the left child. So, in the next iteration I can use this address to fill this "next" pointer with the node I'm currently at. **Example:**

```
    o       o'
   / \     / \
  l   r   l'  r'

In one iteration we are checking the children of o, in the next one we are
checking the children of o'. So in the next iteration I need to make r point
to l', that is why I store the adress of the "next" pointer of r to fill it
in the next iteration.
```

```
struct TreeNode {
  int val;
  TreeNode *left;
  TreeNode *right;
  TreeNode *next;
  TreeNode(int x) : val(x), left(NULL), right(NULL), next(NULL) {}
};

void populateNextPointers(TreeNode *root) {
  TreeNode *curr = root, *newCurr = NULL;
  TreeNode **n = NULL;

  while (curr != NULL) {
    while (curr != NULL) {
      // Get the first not NULL node of this level.
      if (newCurr == NULL) newCurr = curr->left ? curr->left : curr->right;

      // If we stored a pointer to a "next" pointer try to fill it with a not
      // NULL node.
      if (n != NULL) *n = curr->left ? curr->left : curr->right;

      // If we have a left node try to point it to the right node.
      if (curr->left) curr->left->next = curr->right;

      // Store the pointer to the "next" pointer so we can fill it with the next
      // not NULL node.
      if (curr->right) {
        n = &(curr->right->next);
      }
      else if (curr->left) {
        n = &(curr->left->next);
      }

      curr = curr->next;
    }
    // Move to next level
    curr = newCurr;
    newCurr = NULL;
    n = NULL;
  }
}

void specialTraversal(TreeNode *root) {
  TreeNode *curr = root, *newLeft = NULL;

  while (curr != NULL) {
    newLeft = curr->left;

    while (curr != NULL) {
      cout << curr->val << " ";
      curr = curr->next;
    }

    curr = newLeft;
  }
}
```

Binary Tree Non Recursive Traversal

Write functions to traverse a binary tree in preorder, inorder and postorder traversal without using recursion.

Time complexities: O(N), where N is the number of nodes.

Explanation:

Preorder: use a stack, a while loop. Also use a variable to keep the node you are currently dealing with in each iteration. Print the value of the node you are in and go the left. Do that until you find NULL. Then, you pop one node of your stack and start the process again. Do that while your variable with the current node is not NULL OR your stack is not empty.

Inorder: use a stack, a while loop. Also use a variable to keep the node you are currently dealing with in each iteration. Store your current node and go to the left. Do that until you find a NULL node. Then, you pop one node of your stack, print its value and go to the right. Then, you repeat the process going to the left again and sotring your nodes. Do that while your variable with the current node is not NULL OR your stack is not empty.

Postorder: one easy way to do postorder traversal without using recursion is to do the preorder traversal, store the result in a vector and then print the values of this vector in reversing order.
The complicated way that does not use a vector: use a stack, a while loop. Also use a variable to keep the node you are currently dealing with in each iteration. We initialize our variable with the root. Then, in our while we start to store in our stack the left nodes until we find NULL. Then, we get without pop one node from our stack and get its right node. We start the process again. The tricky part is to recognize when we should print. You should print when you are at a right node that does not have any more left nodes that you need to store and then you pop the node your printed.

```
void preorderIterative(TreeNode *root) {
  stack<TreeNode*> s;
  TreeNode *curr = root;

  do {
    if (curr != NULL) {
      s.push(curr);
      cout << curr->val << " ";
      curr = curr->left;
    }
    else {
      curr = (s.top())->right;
      s.pop();
    }
  } while (!s.empty() || curr != NULL);
}

void inorderIterative(TreeNode *root) {
  stack<TreeNode*> s;
  TreeNode *curr = root;

  do {
    if (curr != NULL) {
      s.push(curr);
      curr = curr->left;
    }
    else {
      curr = s.top();
      s.pop();
      cout << curr->val << " ";
      curr = curr->right;
    }
  } while (!s.empty() || curr != NULL);
}

void postorderIterative(TreeNode *root) {
  if (root == NULL) return;

  stack<TreeNode*> s;
  TreeNode *curr = root, *prevCurr = NULL;

  do {
    if (curr != NULL && curr != prevCurr) {
      s.push(curr);
      curr = curr->left;
    }
    else if (curr == s.top()->right) {
      curr = s.top();
      s.pop();
      cout << curr->val << " ";
      prevCurr = curr;
    }
    else {
      curr = s.top()->right;
    }
  } while (!s.empty());
}
```

Binary Tree Path Sum

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

Example:
Given the below binary tree and sum = 22,

```
            5
           / \
          4   8
         /   / \
        11  13  4
       /  \      \
      7    2      1
```

return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

Time complexity: O(N), where N is the number of nodes.

Explanation: first we need to think how we know we are at the end of a path and that is accomplished checking if both children of a node are NULL.
Second, we need to traverse all the paths and we gonna use recursion for that. I choose to first go to the left branch and after to the right branch.
We, then, keep a variable of the remaining sum when adding that node to the path (remainingSum = sum - node->val) and call the function for the left and right branchs with remainingSum as input of these calls.
If we reach a leaf node (both children NULL) and remaining sum is equal to the node value we return true, else we return false. And our recursion calls are made with an OR, like: hasPathSum(left) || hasPathSum(right).

```
int hasPathSum(TreeNode *root, int sum) {
  if (root == NULL) return 0;
```

```
    // We want our sum to end in a leaf, so both children must be NULL.
    if (root->left == NULL && root->right == NULL) {
      if (sum == root->val) return 1;
      else return 0;
    }

    int remainingSum = sum - root->val;

    return hasPathSum(root->left, remainingSum) ||
      hasPathSum(root->right, remainingSum);
  }
```

Binary Tree Root Leaf Numbers

Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.
An example is the root-to-leaf path 1->2->3 which represents the number 123.
Find the total sum of all root-to-leaf numbers % 1003.

Example:

```
    1
   / \
  2   3
```

The root-to-leaf path 1->2 represents the number 12.
The root-to-leaf path 1->3 represents the number 13.
Return the sum = (12 + 13) % 1003 = 25 % 1003 = 25.

Time complexity: O(N), where N is the number of nodes.

Explanation: the numbers are represented from root to leaf. So, one simple solution is to calculate each of these numbers, sum them, and then take the remainder. The problem with this solution is the overflow. The reason our numbers are stored using tree nodes is probably because they are so big they can not fit in a 32-bit or even 64-bit variable.
So, since we need the sum modulo 1003 we need to take advantage of this.

Our number would be calculated doing:
(currentNumber * 10) + node->val.
If some analysis we find out that we can make:
((currentNumber * 10) + node->val) % 1003.

Now, we have all our numbers modulo 1003, but this is not enough because we can have so many of these numbers that our sum can still overflow. So, we also need to make our sum modulo 1003. So, every time we achieve a leaf we add to our sum like this:
sum = (sum + num) % 1003.

```
  void _allPathSum(TreeNode *root, int num, int *sum) {
    if (root == NULL) return;

    num = ((num * 10) + root->val) % 1003;

    // We want our sum to end in a leaf, so both children must be NULL.
    if (root->left == NULL && root->right == NULL) {
      *sum = (*sum + num) % 1003;
    }
    else {
      _allPathSum(root->left, num, sum);
      _allPathSum(root->right, num, sum);
    }
```

```
}

int allPathSum(TreeNode *root) {
  int sum = 0;
  int num = 0;

  // Notice, num is not passed by reference so we have a distinct num variable
  // to work with at each call of _allPathSum.
  _allPathSum(root, num, &sum);

  return sum;
}
```

Binary Tree Symmetry

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

Example:

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

The above binary tree is symmetric.
But the following is not:

```
    1
   / \
  2   2
   \   \
   3    3
```

Time complexity: O(N), where N is the number of nodes.

Explanation: left nodes must be equal to right nodes. So, traverse the tree with two pointers: one at the left, and another at the right. These nodes must have equal val, then when one recursion call ended, call the function again going now to the right of the left pointer and to the left of the right pointer, again the val must be equal.

```
bool _isSymmetric(TreeNode *left, TreeNode *right) {
  if (left == NULL && right == NULL) return true;
  if (left == NULL || right == NULL) return false;

  if (left->val != right->val) return false;

  return _isSymmetric(left->left, right->right) &&
    _isSymmetric(left->right, right->left);
}

bool isSymmetric(TreeNode *root) {
  if (root == NULL) return true;

  return _isSymmetric(root->left, root->right);
}
```

Binary Tree To Doubly Linked List

```
                             +--------+
                             |        |
                             |   4    |
                             |        |
                  +----------+--------+-------+
                  |                           |
                  |                           |
                  |                           |
              +---v----+                  +---v----+
              |        |                  |        |
              |   2    |                  |   5    |
              |        |                  |        |
    +---------+--------+--------+         +--------+
    |                           |
    |                           |
    |                           |
+---v----+                  +---v----+
|        |                  |        |
|   1    |                  |   3    |
|        |                  |        |
+--------+                  +--------+
```

Transform the sorted binary tree in a circular doubly linked list sorted in increasing order.

Time complexity: O(N), where N is the number of nodes.

Explanation: we need to helper functions. One append a node to a circular doubly linked list, the other joins two doubly linked lists. We traverse the tree in order (first we go all to the left than we start to go to the right). For each of our calls we append the current node in the left list and then we join the left list with the right list. We need to return our left list head in all calls, because the left most node of the tree is the head of our doubly linked list. So, when doing this exercise beside writing the function that does the transformation you also need to wright the append and join helper functions.

```
Node *tree_to_list(Node *node) {
  if (node == NULL) return NULL;

  Node *small = tree_to_list(node->small);
  Node *large = tree_to_list(node->large);
  small = append(small, node);
  small = join(small, large);

  return small;
}

Node* append(Node *head, Node *node) {
  if (head != NULL) {
    Node *last = head->small;

    node->large = head;
    node->small = last;

    last->large = node;
    head->small = node;

    return head;
  }

  node->small = node;
  node->large = node;
```

```
    return node;
  }

  Node* join(Node *head1, Node *head2) {
    if (head1 != NULL && head2 != NULL) {
      Node *last1 = head1->small;
      Node *last2 = head2->small;

      last1->large = head2;
      head2->small = last1;

      last2->large = head1;
      head1->small = last2;

    }

    return head1;
  }

  /* --- Helper functions --- */

  void print_node(Node *node, char line_break) {
    cout << node->val << line_break;
  }

  void print_list(Node *head) {
    Node *curr = head;

    do {
      print_node(curr, ' ');
      curr = curr->large;
    } while (curr != head);

    cout << endl;
  }

  void print_tree(Node *root) {
    Node *curr = root;

    if (root != NULL) {
      print_tree(root->small);
      print_node(root, ' ');
      print_tree(root->large);
    }
  }

  void print_list_two(Node *head) {
    Node *curr = head;
    int count = 0;

    do {
      print_node(curr, ' ');
      curr = curr->large;

      if (curr == head) count++;
    } while (count < 2);

    cout << endl;
  }

  void print_list_two_reverse(Node *head) {
    Node *last = head->small;
    Node *curr = last;
    int count = 0;

    do {
      print_node(curr, ' ');
      curr = curr->small;
      if (curr == last) count++;
    } while (count < 2);

    cout << endl;
  }
```

Binary Tree To List

Given a binary tree, flatten it to a linked list in-place.

Example:

Given,

```
        1
       / \
      2   5
     / \   \
    3   4   6
```

The flattened tree should look like,

```
   1
    \
     2
      \
       3
        \
         4
          \
           5
            \
             6
```

Note that the left child of all nodes should be NULL.

Time complexity: O(N), where N is the number of nodes. Is O(N) because the loop will go through all nodes even if no work is performed in that iteration.

Explanation: first thing to note is that this needs to have an order, and looking at the example we can see that this linked list must be build using preorder traversal. Note that the last node to be printed in a preorder traversal is the right most node.

Based on this, the last node to be printed from the left subtree is the right most node of the left subtree. The next node to be printed would be the first node from the right subtree. So, we can append the entire right subtree in the right most node of the left subtree and then replace the right subtree of our current node (root in the first iteration) with its left subtree. After, we go one node to the right from the current node and do the same thing. We keep doing this until our current node is NULL.

```
void treeToList(TreeNode *root) {
  TreeNode *curr = root;

  while (curr != NULL) {

    if (curr->left != NULL) {
      TreeNode *rightMost = curr->left;

      while (rightMost->right != NULL) {
        rightMost = rightMost->right;
      }

      rightMost->right = curr->right;
      curr->right = curr->left;
      curr->left = NULL;
    }

    curr = curr->right;
  }
}

void specialTraversal(TreeNode *root) {
  TreeNode *curr = root;

  while (curr != NULL) {
    cout << curr->val << " ";
    curr = curr->right;
  }
  cout << endl;
}
```

Binary Tree Vertical Traversal

Given a binary tree, print a vertical order traversal of it.

Notes:
1. If 2 Tree Nodes share the same vertical level then the one with lesser depth will come first.
2. If 2 Tree Nodes share the same line and depth then the one most to the left will come first.

Example:
Given binary tree

```
      6
    /   \
   3     7
  / \     \
 2   5     9
```

Return

```
[
    [2],
    [3],
    [6 5],
    [7],
    [9]
]
```

Time complexity: O(N), where N is the number of nodes in the tree.
Space complexity: O(N).

Explanation: think about this: can you perform level order traversal and while doing that identify the line which a node belongs?

So, we perform level order traversal and for each node at each level we also store its line. Later, we create our answer iterating through our levels and placing each node at the vector storing the nodes for that particular line.

The annoying part that lets the code a little ugly is that to know how many lines we have we need to keep the left most line, and the right most line.

```
If we say that root is line 0, then left most line is the line with minimum
value (this minimum value will be 0 because of the root or less), and the
right most line will be the line with maximum value (will be at least 0
because of the root). Now, notice that lines to the left of the root will
be negative, so we need to offset them to their correct place in the answer
since we don't have negative indexes in our vectors, and for that our
offset = abs(leftMostLine). Our total number of lines will be:
abs(leftMostLine) + rightMostLine + 1.
```

```
vector<vector<int>> verticalOrderTraversal(TreeNode *root) {
  if (root == NULL) return vector<vector<int>>();

  int leftMostLine = INT_MAX, rightMostLine = INT_MIN;
  vector<vector<pair<int, int>>> levelOrder;
  queue<pair<TreeNode *, int>> qe;

  qe.push({root, 0});

  while (!qe.empty()) {
    queue<pair<TreeNode *, int>> nextQe;
    vector<pair<int, int>> currLevel;

    while (!qe.empty()) {
      TreeNode *currNode = (qe.front()).first;
      int currLine = (qe.front()).second;
      qe.pop();

      leftMostLine = min(leftMostLine, currLine);
      rightMostLine = max(rightMostLine, currLine);

      currLevel.push_back({currNode->val, currLine});

      if (currNode->left) nextQe.push({currNode->left, currLine - 1});
      if (currNode->right) nextQe.push({currNode->right, currLine + 1});
    }

    levelOrder.push_back(currLevel);
    qe = nextQe;
  }

  int offset = abs(leftMostLine);
  int totalLines = offset + rightMostLine + 1;
  vector<vector<int>> ans(totalLines, vector<int>());

  for (int i = 0; i < levelOrder.size(); i++) {
    for (int j = 0; j < levelOrder[i].size(); j++) {
      pair<int, int> curr = levelOrder[i][j];
      ans[curr.second + offset].push_back(curr.first);
    }
  }

  return ans;
}

// O(NlogN) time solution. Calculates the lines and depth and then stable_sort
// by lines and if lines are equal by depth.
// struct NodeInfo {
//   int depth, line, val;
//   NodeInfo(int x, int y, int z) : depth(x), line(y), val(z) {}
// };

// void getNodes(TreeNode *root, int depth, int line, vector<NodeInfo*> &nodes) {
//   if (root == NULL) return;

//   NodeInfo *curr = new NodeInfo(depth, line, root->val);
//   nodes.push_back(curr);

//   getNodes(root->left, depth + 1, line - 1, nodes);
// 	getNodes(root->right, depth + 1, line + 1, nodes);
// }

// bool cmp(NodeInfo *n1, NodeInfo *n2) {
//   return n1->line < n2->line ||
//     (n1->line == n2->line && n1->depth < n2->depth);
// }

// vector<vector<int>> verticalTraversal(TreeNode *root) {
//   vector<NodeInfo*> nodes;
//   vector<vector<int>> ans;
//   getNodes(root, 0, 0, nodes);

//   stable_sort(nodes.begin(), nodes.end(), cmp);

//   int i = 0;
//   while (i < nodes.size()) {
//     vector<int> temp;
//     int currLine = nodes[i]->line;

//     while (i < nodes.size() && nodes[i]->line == currLine) {
//       temp.push_back(nodes[i]->val);
//       i++;
//     }

//     if (temp.size() > 0) ans.push_back(temp);
//   }

//   return ans;
// }
```

Binary Tree Zigzag Traversal

Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

Example:
Given binary tree,

```
    3
   / \
  9  20
    /  \
   15   7
```

return,

```
[
  [3],
  [20, 9],
  [15, 7]
]
```

Time complexity: O(N), where N is the number of nodes.

Exaplanation: use two nested while loops, a stack to keep the nodes and a variable to indicate the current direction (left to right or right to left). Also, use a auxiliary stack inside your loop to keep the next nodes. So, you loop through the main stack (that is initialized with root) and print the nodes. For each node you process from the main stack you store its children in the auxiliary stack based on your direction variable. When, the main stack is empty you assign to main stack the auxiliary stack and change direction. If after you assign to the main stack the auxiliary stack the main stack is still empty you stop your loop.

```
void zigzag(TreeNode *root) {
  if (root == NULL) return;

  int dir = 0; // 0 right, -1 left

  stack<TreeNode*> s;
  s.push(root);

  while (true) {
    stack<TreeNode*> t;

    while (!s.empty()) {
      TreeNode* curr = s.top();
      s.pop();

      cout << curr->val << " ";

      if (dir) {
        if (curr->right) t.push(curr->right);
        if (curr->left) t.push(curr->left);
      }
      else {
        if (curr->left) t.push(curr->left);
        if (curr->right) t.push(curr->right);
      }
    }

    dir = ~dir; // 0 to -1, -1 to 0
    s = t;
    if (s.size() == 0) break;
  }

  cout << endl;
}
```

Bits Byte Order

Identify the architecture endianess.

Time complexity: O(1).

Explanation: use a union with 2 bytes. Store in each of these bytes different numbers like 1 and 2. In your union keep an array of chars with two positions so you can read them. If the first byte in this char array is 1 and the second 2 we have a big-endian architecture, little-endian otherwise.

Bits Count-1 Bits

Write a function that takes an unsigned integer and returns the number of 1 bits it has.

Time complexity: O(N), where N is the number of bits.

Explanation: You can AND the number with 1 and check the result. If is > 0 then increment a counter. at each iteration shift the number 1 bit to right. Another solution, x & (x - 1) unset the lowest non-zero bit of a number, because x - 1 is like x with the the lowest non-zero bit turned to 0 and all the other ones before it turned to 1. So, while x is not zero increment a counter.

```
int numSetBits(unsigned int a) {
  unsigned int x = a;
  unsigned int c = 0;

  /*
   * x & (x - 1) unset the lowest non-zero bit.
   */
  do {
    x = x & (x - 1);
    c++;
  } while (x);

  return c;
}

// int numSetBits(unsigned int a) {
//   unsigned int x = a;
//   unsigned int c = 0;

//   while (x) {
//     if (x & 1) c++;
//     x = x >> 1;
//   }

//   return c;
// }
```

Bits Count Diff Bits

We define f(X, Y) as number of different corresponding bits in binary representation of X and Y. For example, f(2, 7) = 2, since binary representation of 2 and 7 are 010 and 111, respectively. The first and the third bit differ, so f(2, 7) = 2.

You are given an array of N positive integers, A1, A2 ,…, AN. Find sum of f(Ai, Aj) for all pairs (i, j) such that 1 ≤ i, j ≤ N. Return the answer modulo 10^9+7.

Example:
A = [1, 3, 5]

We return
f(1, 1) + f(1, 3) + f(1, 5) +
f(3, 1) + f(3, 3) + f(3, 5) +
f(5, 1) + f(5, 3) + f(5, 5) =

0 + 1 + 1 +
1 + 0 + 2 +
1 + 2 + 0 = 8

Time complexity: O(M * N), where N is the size of the given array and M the number of bits. If you consider the number of bits a small constant so this algorithm has linear time complexity.
Space complexity: O(1).

```
Explanation: what this problem is asking is the sum of Hamming Distances.

Check one bit position at a time for each element. So we check
bit at position 1 for all elements, then bit at position 2 for all elements.
And for each of these bits we count how many of them are 1's and how many
are 0's, then the number of different bits for that position is
zeroBitCount * oneBitCount. For example:

Given A = [1, 3, 5]
pos: 0 1 2
     -----
     0 0 1
     0 1 1
     1 0 1

For position 0 we get 2 zeros and 1 one, so 2 * 1 different bits for
position 0. For position 1, 2 zeros and 1 one, so 2 * 1 different bits.
For position 2, 3 ones and 0 zeros, so 3 * 0 different bits. So, our answer
is (2 + 2 + 0) * 2 = 8. Notice we are multiplying by 2 because this exercise
considers f(X, Y) different of f(Y, X) though they are equal.
```

```
#define MOD 1000000007LL
#define INT_SIZE 32

int cntBits(vector<int> &a) {
  long long ans = 0;

  for (int i = INT_SIZE - 1; i >= 0; i--) {
    long long zeroCount = 0, oneCount = 0;
    for (int j = 0; j < a.size(); j++) {
      if (a[j] & (1 << i)) oneCount++;
      else zeroCount++;
    }
    ans = (ans + zeroCount * oneCount) % MOD;
  }

  return (2LL * ans) % MOD;
}

// Naive O(N^2) time solution using XOR.
// int count1Bits(int n) {
//   int count = 0;

//   while (n) {
//     n = n & (n - 1);
//     count++;
//   }

//   return count;
// }

// int cntBits(vector<int> &a) {
//   int ans = 0;

//   for (int i = 0; i < a.size(); i++) {
//     for (int j = i; j < a.size(); j++) {
//       if (i == j) continue;
//       int bits = a[i] ^ a[j];
//       ans = (ans + 2 * count1Bits(bits)) % MOD;
//     }
//   }

//   return ans;
// }
```

Bits Divide Integers

```
Divide two integers without using multiplication, division and mod operator.
Return the floor of the result of the division.

Notes:
1. Also, consider if there can be overflow cases. For overflow case,
return INT_MAX.
```

Example:

```
5 / 2 = 2
```

Time complexity: O(K), where K is the number of bits, and can be considered constant.
Space complexity: O(1).

Explanation: the naive way is to keep subtracting the divisor from the dividend until divided is smaller than divisor. But will have linear time complexity to the dividend.

The better way is to apply the same algorithm we use to divide base-10 numbers on the paper. We start from the end of the dividend and accumulate the bits in a variable called currDividend until we get a number less greater or equal to the divisor. At this moment we add the bit 1 to our answer and update the currDividend to currDividend - divisor. For example, lets see the division of 11 by 2 in decimal and binary bases:

```
Decimal

'
11 | 2
------
1 is smaller than 2 so append 0 to the answer and keep going.

''
11 | 2
------
11 is greater than 2 and we know 2 * 5 is the closest we can get to 11, so
append 5 to the answer and we are done because there are no other numbers.

Binary

'
1011 | 0010
1 is smaller than 2 so append 0 to the answer and keep going.
ans = 0

''
1011 | 0010
2 is equal to 2, so append 1 to the answer and 2 - 2 = 0 so make
currDividend = 0.
ans = 1

  '
1011 | 0010
1 is smaller the 2 so append 0 to the answer and keep going.
ans = 10

  ''
1011 | 0010
3 is larger then 2 so append 1 to the answer and make
currDividend = 3 - 2 = 1.
ans = 101

Notice that in the decimal case we need to find out the number that is
closest to our currDividend because we have 9 options (1 to 9), but in the
```

binary case we just have one option (just 1).

In our algorithm we need to perform this operations with positive numbers, so get the absolute values of the input and keep the sign. Also, the description of this problem is misleading because it states it wants the Floor of the divisions, but Floor(-10 / 3) is -4 though the official solution returns -3, so actually the problem wants the Ceil for negative answer and the Floor for positive answer.

Don't forget to check for overflows. Work with "long long" and after check if the answer is larger or equal to abs(INT_MIN).

```
int divide(int a, int b) {
  if (b == 0) return INT_MAX;

  long long ans = 0, dividend = a, divisor = b, currDividend = 0;
  int sign = dividend < 0 ^ divisor < 0 ? -1 : 1;

  dividend = abs(dividend), divisor = abs(divisor);

  for (long long i = 31; i >= 0; i--) {
    bool bit = dividend & (1LL << i);
    currDividend = (currDividend << 1LL) | bit;
    ans = ans << 1LL;

    if (currDividend >= divisor) {
      currDividend = currDividend - divisor;
      ans = ans | 1LL;
    }
  }

  ans = sign * ans;

  // This should be necessary to get a properly Floor as the exercise states,
  // but actually the official solution is looking for an algorithm that is
  // a Floor for positive answer, and Ceil for negative answer.
  // if (ans < 0 && currDividend != 0) ans--;

  return ans >= 2147483648 ? INT_MAX : ans;
}

// int divide(int dividend, int divisor) {
//   if (divisor == 0) return INT_MAX;
//   if (dividend == -2147483648 && divisor == -1) return INT_MAX;
//   if (dividend == -2147483648 && divisor == 1) return INT_MIN;

//   long long ans = 0, sign = 1;

//   if ((dividend >= 0 && divisor < 0) || (dividend < 0 && divisor > 0)) {
//     sign = -1;
//   }

//   long long a = abs((long long)dividend);
//   long long b = abs((long long)divisor);

//   while (a >= b) {
//     ans++;
//     a -= b;
//   }

//   return (int)(sign * ans);
// }
```

Bits Min Xor Pair

Given an array of N integers, find the pair of integers in the array which have minimum XOR value. Report the minimum XOR value.

Constraints:
2 <= N <= 100 000
0 <= A[i] <= 1 000 000 000

Examples:
Input
0 2 5 7
Output
2 (0 XOR 2)

Input
0 4 7 9
Output
3 (4 XOR 7)

Time complexity: O(NlogN) or O(N) depending on implementation, where N is the size of the given array.
Space complexity: O(1) or O(N) depending on implementation.

Explanation: there are two nice solutions for this problem.

First approach:
Sort the elements, iterate over the array performing XOR for each consecutive pair and keep the smallest. Sorting makes the elements with most similar bits stay side by side and it makes sense since XOR is binary sum without carry.

Second approach:
Use a trie. This solution uses more memory, but has linear time complexity. Our alphabet for this trie is 0 and 1 only. If we insert in the trie the bits starting from the MSB (Most Significant Bit) the most similar element B that we can find in this trie for another element A is the smallest XOR that we could find for A. So we iterate over the array looking in the trie for the most similar element to the current element. Every time we get a mismatch (the bit we are current looking is not present in the trie for the prefix we are) we just check the other bit. Notice we just insert the current element after we looked in the trie for its most similar element, because, of course, if the current element was already in the trie then it would be the most similar element to itself.

```
#define INT_SIZE 32

int findMinXor(vector<int> &a) {
  int ans = INT_MAX;
  sort(a.begin(), a.end());

  for (int i = 0; i < a.size() - 1; i++) {
      ans = min(ans, a[i] ^ a[i + 1]);
  }

  return ans;
}

// Solution using Trie. Time complexity: O(N), space complexity: O(N).
// struct TrieNode {
//   int val;
//   vector<TrieNode*> keys;
//   TrieNode() : val(0), keys(2, NULL) {}
// };

// void insert(TrieNode *root, int val) {
//   TrieNode *curr = root;
//   for (int i = (INT_SIZE - 1); i >= 0; i--) {
//     bool currBit = val & (1 << i);
//     if (curr->keys[currBit] == NULL) {
//       curr->keys[currBit] = new TrieNode();
//     }
//     curr = curr->keys[currBit];
//   }

//   curr->val = val;
// }

// int findMinXorUtil(TrieNode *root, int val) {
//   TrieNode *curr = root;

//   for (int i = (INT_SIZE - 1); i >= 0; i--) {
//     bool currBit = val & (1 << i);
//     if (curr->keys[currBit] != NULL) {
//       curr = curr->keys[currBit];
//     }
//     else if (curr->keys[!currBit] != NULL) {
//       curr = curr->keys[!currBit];
//     }
//   }

//   return val ^ curr->val;
// }

// int findMinXor(vector<int> &a) {
//   int ans = INT_MAX;
//   TrieNode *root = new TrieNode();

//   insert(root, a[0]);

//   for (int i = 1; i < a.size(); i++) {
//     ans = min(ans, findMinXorUtil(root, a[i]));
//     insert(root, a[i]);
//   }

//   return ans;
// }
```

Bits Reverse

Reverse bits of an 32 bit unsigned integer.

Example:
00000000000000000000000000000011 =>
11000000000000000000000000000000

Time complexity: O(1). Or O(logN), if you want to consider the amount of bits as a variable N.

Explanation: divide and conquer approach:

```
             01101001

            /        \

         0110       1001

       /   \        /   \

     01    10    10    01
```

```
       /\     /\    /\     /\

      0  1   1  0  1  0   0  1

Reverse the bottom bits with:
x = ((x & 0x55555555) << 1) | ((x & 0xAAAAAAAA) >> 1);
then the bits in the previous level with:
((x & 0x33333333) << 2) | ((x & 0xCCCCCCCC) >> 2);
etc...
```

```
unsigned int reverse(unsigned int a) {
  unsigned int x = a;

  x = ((x & 0x55555555) << 1) | ((x & 0xAAAAAAAA) >> 1);
  x = ((x & 0x33333333) << 2) | ((x & 0xCCCCCCCC) >> 2);
  x = ((x & 0x0F0F0F0F) << 4) | ((x & 0xF0F0F0F0) >> 4);
  x = ((x & 0x00FF00FF) << 8) | ((x & 0xFF00FF00) >> 8);
  x = (x << 16) | (x >> 16);

  return x;
}

// unsigned int reverse(unsigned int a) {
//   unsigned int d = a;
//   unsigned int b = 0;
//   unsigned int r = 0;
//   unsigned int p = 2147483648;
//   unsigned int c = 0;

//   while (c < 32) {
//       b = d % 2;
//       d = d / 2;
//       if (b) r += p;
//       p /= 2;
//       c++;
//   }

//   return r;
// }
```

Bits Single Number-2

Given an array of integers, every element appears thrice except for one which occurs once. Find that element which does not appear thrice.
Note: Your algorithm should have a linear runtime complexity and not use extra memory.

Time complexity: O(N), where N is the number of bits.

Explanation: for each number of the array loop through all his bits (yes, I agree this is not O(N), but this is the official solution so...) and count how many times the bit one appeared. You can use bitmasks for that. So, create a mask called "ones" and another called "twos". When the jth number appeared one time set the correspond position in the "ones" mask, when it appeared two set the correspond bit in the "twos" mask and unset in the "ones" mask.

```
int singleNumber(const int *a, int n1) {
  int ones = 0, twos = 0;
  int i, j, bit;
  for (i = 0; i < n1; i++) {
    for (j = 0; j < 32; j++) {
      // If the jth bit is 0 skip this iteration
      if (!(a[i] & (1 << j))) continue;

      if (ones & (1 << j)) {
        twos = twos | (1 << j);
        ones = ones & ~(1 << j);
      }
      else if (twos & (1 << j)) {
        ones = ones & ~(1 << j);
        twos = twos & ~(1 << j);
      }
      else {
        ones = ones | (1 << j);
      }
    }
  }

  return ones;
}
```

Bst Find Sum

```
Given a binary search tree T, where each node contains a positive integer,
and an integer K, you have to find whether or not there exist two different
nodes A and B such that A.value + B.value = K.

Return 1 to denote that two such nodes exist. Return 0, otherwise.

Notes:
Your solution should run in linear time and not take memory more than
O(height of T).
Assume all values in BST are distinct.
```

Example:

```
Input 1:

T :       10
         / \
        9   20

K = 19

Return: 1

Input 2:

T:        10
         / \
        9   20

K = 40

Return: 0
```

Time complexity: O(N), where N is the number of nodes.

Explanation: the tricky is to think about the BST as a sorted array. How do you find a sum in a sorted array? You use two pointers: one at the beginning (i) and the other at the end (j) of the array. If a[i] + a[j] > sum decrement j, else if a[i] + a[j] < sum increment i, else it is equal and you

found your sum. Now, apply the same idea for the BST using log N extra memory.
Use two stacks, one for the left nodes (l) and the other for the right nodes (r). Initialize l with all left nodes from root, and r with all right nodes from root.
If l.top() + r.top() > sum pop the right stack and re-fill it with r.top()->left nodes including r.top()->left. Else if l.top() + r.top() < sum pop the left stack and re-fill it with l.top()->right including l.top()->right.

```
void fillLeftStack(TreeNode *root, stack<TreeNode*> &s) {
  TreeNode *temp = root;

  while (temp) {
    s.push(temp);
    temp = temp->left;
  }
}

void fillRightStack(TreeNode *root, stack<TreeNode*> &s) {
  TreeNode *temp = root;

  while (temp) {
    s.push(temp);
    temp = temp->right;
  }
}

bool check2Sum(TreeNode *root, int sum) {
  if (root == NULL) return false;

  stack<TreeNode*> sl;
  stack<TreeNode*> sr;

  fillLeftStack(root, sl);
  fillRightStack(root, sr);

  TreeNode *l = sl.top();
  TreeNode *r = sr.top();

  while (l && r) {
    if (l != r) {
      if (l->val + r->val > sum) {
        sr.pop();
        fillRightStack(r->left, sr);
        r = sr.size() ? sr.top() : NULL;
      }
      else if (l->val + r->val < sum) {
        sl.pop();
        fillLeftStack(l->right, sl);
        l = sl.size() ? sl.top() : NULL;
      }
      else {
        return true;
      }
    }
    else {
      sr.pop();
      fillRightStack(r->left, sr);
      r = sr.size() ? sr.top() : NULL;
    }
  }

  return false;
}
```

Bst Fix Swapped Elements

Two elements of a binary search tree (BST) are swapped by mistake.
Tell us the 2 values swapping which the tree will be restored.

Note:
A solution using O(n) space is pretty straight forward. Could you devise a constant space solution?

Time complexity: O(N), where N is the number of nodes.

Explanation: the trick is to traverse the tree inorder. This give us the elements in sorted order. We need to look for anomalies. So we are looking for elements were currentNode->val < previousNode->val, because this should not happen in inorder traversal. When we find this we store our previousNode in a variable node1 and currentNode in a variable node2 and we keeping traverse and if we find this again we replace node2 with the new currentNode.

1, 3, 5, 7, 9, 11, 12, 15, 14, 13, 17

We want to have node1 = 15 and node2 = 13. Note that when we find the anomaly 15 > 14 we make node1 = 15 and node2 = 14, because when you switch adjacent elements you would not find any element anymore smaller than its previous element.

1, 3, 5, 7, 9, 11, 13, 12, 14, 15, 17

We want to have node1 = 13 and node2 = 12, but note that from 12 onwards everything is normal. Note too that if you find another anormaly after your first one it means you must replace node2 with your new currentNode.

```
void findMistake(TreeNode *root, TreeNode **prev, TreeNode **n1, TreeNode **n2) {
  if (root == NULL) return;

  findMistake(root->left, prev, n1, n2);

  if (*prev && root->val < (*prev)->val) {
    if (*n1 == NULL) {
      *n1 = *prev;
      *n2 = root;
    }
    else {
      *n2 = root;
    }
  }

  *prev = root;

  findMistake(root->right, prev, n1, n2);
}

void fixMistake(TreeNode *root) {
  if (root == NULL) return;

  TreeNode *n1 = NULL, *n2 = NULL, *prev = NULL;

  findMistake(root, &prev, &n1, &n2);

  if (n1 && n2) {
    int temp = n1->val;
    n1->val = n2->val;
    n2->val = temp;
  }
}
```

Bst Invert

Invert a tree. It is like if you were looking at it in the mirror.

Time complexity: O(N), where N is the number of nodes in the tree.
Space complexity: O(logN).

Explanation: to do that just change left and right pointers at each call and then call the function again recursively.

```
void invert(TreeNode *root) {
  if (root == NULL) return;

  TreeNode *temp = root->left;
  root->left = root->right;
  root->right = temp;

  invert(root->left);
  invert(root->right);
}
```

Bst Is Balanced

Check if a binary tree is balanced. A binary tree is balanced if the heights of left and right subtree don't differ by more than 1.

Time complexity: O(N), where N is the number of nodes.

Explanation: use an auxiliary function to check the heights of the subtrees bottom-up. A empty subtree has height -1. Find the height for the left and right subtree than check if their absolute difference is greater than 1. If it is, mark a variable that you should pass by reference as false. On your main function return this variable.

```
int _isBalanced(TreeNode *root, int *ans) {
  if (root == NULL) return -1;

  int leftHeight = _isBalanced(root->left, ans);
  int rightHeight = _isBalanced(root->right, ans);

  if (abs(leftHeight - rightHeight) > 1) *ans = 0;

  return max(leftHeight, rightHeight) + 1;
}

int isBalanced(TreeNode *root) {
  if (root == NULL) return 1;

  int ans = 1;

  _isBalanced(root, &ans);

  return ans;
}
```

Bst Is Valid

Check if a binary tree is a valid binary search tree.

Time complexity: O(N), where N is the number of nodes.

Explanation 1: Traverse the tree in inorder traversal because this would give us the nodes in sorted order. Instead of using a list to keep the nodes and then check if the list is sorted pass to each call the previous node. So, the current node just needs to be smaller than the previous node if it was called from a left subtree, or bigger if it was called from a right subtree (so you also will need a variable to check if the current node is in a left or right branch).

Expanation 2: Use a range to check if the current node is valid. The root node can be in the range (-infinity, +infinity). Then, the left node from root node can be in range (-infinity, root->val), and the right node can be in the range (root->val, +infinity). Do that recursively updating the range for each call.

```
// Method 2
bool _isBST(TreeNode *root, int minValue, int maxValue) {
  if (root == NULL) return true;

  if (root->val > minValue && root->val <= maxValue) {
    bool left = _isBST(root->left, minValue, root->val);
    bool right = _isBST(root->right, root->val, maxValue);

    return left && right;
  }
  else {
    return false;
  }
}

bool isBST(TreeNode *root) {
  return _isBST(root, INT_MIN, INT_MAX);
}

// Method 1
// bool _isBST(TreeNode *root, TreeNode *prev, bool left) {
//   if (root == NULL) return true;

//   bool c = false;
//   bool l = _isBST(root->left, root, true);

//   if (prev == NULL) {
//     c = true;
//   }
//   else if (left) {
//     if (root->val <= prev->val) c = true;
//   }
//   else {
//     if (root->val > prev->val) c = true;
//   }

//   bool r = _isBST(root->right, root, false);

//   return c && l && r;
// }

// bool isBST(TreeNode *root) {
//   return _isBST(root, NULL, true);
// }
```

Bst Iterator

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.
The first call to next() will return the smallest number in BST. Calling next() again will return the next smallest number in the BST, and so on.

Notes:
1. next() and hasNext() should run in average O(1) time and uses O(h) memory, where h is the height of the tree.
2. Try to optimize the additional space complexity apart from the amortized time complexity.

Time complexity: Amortized O(1), Worst case O(h), where h is the height of the tree.

Explanation: the first thing to notice is that they want the amortized complexity to be O(1), which basically means that sometimes will not be O(1). So we can use a stack and in construction time we fill it with all the left nodes from root, which give us a total of O(h) extra memory, where h is the height of the tree. At each time we retrieve a node we re-fill the stack with the right nodes of this just retrieved node.

```
class BSTIterator {
private:
  stack<TreeNode*> s;

public:
  BSTIterator(TreeNode *root) {
    fillStack(root);
  }

  bool hasNext() {
    return !s.empty();
  }

  int next() {
    TreeNode *temp = s.top();
    s.pop();

    if (temp->right) fillStack(temp->right);

    return temp->val;
  }

  void fillStack(TreeNode *root) {
    TreeNode *curr = root;
    while (curr) {
      s.push(curr);
      curr = curr->left;
    }
  }
};
```

Bst Kth Smallest Element

Given a binary search tree, write a function to find the kth smallest element in the tree.

```
NOTE: You may assume 1 <= k <= Total number of nodes in BST.
```

Example:

```
Given,
  2
 / \
1   3

and k = 2

Return 2, as 2 is the second smallest element in the tree.
```

Time complexity: O(N), where N is the number of nodes.

Explanation: just perform inorder traversal. In the place in the code where you would print your nodes in an inorder traversal you start to decrement your k. When k is zero you keep that value and return.

```
void _kthsmallest(TreeNode* root, int *k, int *val) {
  if (root == NULL) return;

  _kthsmallest(root->left, k, val);
  (*k)--;
  if (*k == 0) {
    *val = root->val;
    return;
  }
  _kthsmallest(root->right, k, val);
}

int kthsmallest(TreeNode* root, int k) {
  int j = k, val;

  _kthsmallest(root, &j, &val);

  return val;
}
```

Bst Successor

Given a BST and a value V, find the successor of this value. Don't use recursion.

Time complexity: O(logN), where N is the number of nodes.

Explanation: use two BST nodes, one to traverse the tree and another one to keep the successor. If the node you are current in has a value larger than the given value V store this node as the successor and go to the left. If the current node is smaller or equal to V go to the right.

```
TreeNode* getSuccessor(TreeNode *root, int val) {
  TreeNode *s = NULL, *t = root;

  while (t != NULL) {
    if (t->val > val) {
      s = t;
      t = t->left;
    }
    else if (t->val <= val) {
      t = t->right;
    }
  }

  return s;
}
```

Bubble Sort

```
void bubbleSort(vector<int> &a) {
  int temp;
  bool sorted = true;

  for (int i = 0; i < a.size(); i++) {
    // The largest element in one complete execution of the inner loop will
    // be in its final position. Then, the element before the last in the
    // second complete execution and so on. That is why we subtract i, so we
    // don't take into consideration elements that are already in their final
    // position.
    for (int j = 0; j < a.size() - 1 - i; j++) {
      if (a[j] > a[j + 1]) {
        swap(a[j], a[j + 1]);
        sorted = false;
      }
    }

    // If there were no swaps, so the list is already sorted
    if (sorted) break;
    sorted = true;
  }
}
```

Cartesian Tree From Inorder

Given an inorder traversal of a cartesian tree, construct the tree.

Cartesian tree: is a heap ordered binary tree, where the root is greater than all the elements in the subtree.

Note: You may assume that duplicates do not exist in the tree.

Example:
Given, [1 2 3]

Return,
```
          3
         /
        2
       /
      1
```

Time complexity: O(N), where N is the number of nodes.

Explanation: the first thing to notice is that inorder traversal here doesn't means the elements in the array is in sorted order because this is not a binary search tree. Nonetheless, whoever is our root all the elements to the left of it are their left children and all the elements to its right are their right children. So, we can use recursion with two variables in our function call keeping which portion of the array we are dealing with. At each

call we find out who is our root with a "for" loop. Then we can our function again updating our limits variable to be:
start .. root index - 1
root index + 1 .. end

```
TreeNode* _buildCartesianTree(vector<int> &inorder, int s, int e) {
  if (s > e) return NULL;

  int i, maxValue = -1, maxIdx;

  for (i = s; i <= e; i++) {
    if (inorder[i] > maxValue) {
      maxIdx = i;
      maxValue = inorder[i];
    }
  }

  TreeNode *root = new TreeNode(inorder[maxIdx]);

  int leftSize = maxIdx - s;
  int rightSize = e - maxIdx;

  if (leftSize > 0) {
    root->left = _buildCartesianTree(inorder, s, maxIdx - 1);
  }
  if (rightSize > 0) {
    root->right = _buildCartesianTree(inorder, maxIdx + 1, e);
  }

  return root;
}

TreeNode* buildCartesianTree(vector<int> &inorder) {
  if (inorder.size() == 0) return NULL;

  return _buildCartesianTree(inorder, 0, inorder.size() - 1);
}
```

Circular List

Implementation of a Circular Linked List.

```
ListNode* create_node(int val) {
  ListNode *node = (ListNode *)malloc(sizeof(ListNode));

  node->val = val;
  node->next = NULL;

  return node;
}

ListNode* get_last(ListNode *head) {
  ListNode *curr = head;

  while (curr->next != head) {
    curr = curr->next;
  }

  return curr;
}
```

```
void append(ListNode *head, int val) {
  ListNode *last = get_last(head);
  ListNode *node = create_node(val);

  last->next = node;
  node->next = head;
}

void push(ListNode **head, int val) {
  ListNode *node = create_node(val);
  ListNode *last = get_last(*head);

  node->next = *head;
  *head = node;

  last->next = node;
}

void insert_after(ListNode *previous_node, int val) {
  ListNode *new_node = create_node(val);
  ListNode *previous_next = previous_node->next;

  previous_node->next = new_node;
  new_node->next= previous_next;
}

// int remove_node(ListNode **head, ListNode* node) {
//   ListNode *prev = NULL;
//   ListNode *curr = *head;

//   while (curr != node && curr != NULL) {
//     prev = curr;
//     curr = curr->next;
//   }

//   if (curr == node) {
//     if (prev == NULL) {
//       *head = curr->next;
//     }
//     else {
//       prev->next = curr->next;
//     }
//     free(curr);

//     return 1;
//   }

//   return 0;
// }

/*
 * Optimized remove_node(). Instead of pointing to nodes we point to the
 * "next" pointers (or in other words to the pointers that points to nodes).
 */
int remove_node(ListNode **head, ListNode* node) {
  ListNode **curr = head;
  ListNode *last = get_last(*head);

  while (*curr != node && (*curr)->next != *head) {
    curr = &((*curr)->next);
  }

  if (*curr == node) {
    *curr = node->next;
    if (*curr == *head) last->next = node->next;
    free(node);
    return 1;
  }
```

```
    return 0;
  }

  void free_list(ListNode *head) {
    ListNode *curr = head, *tmp = NULL;

    while (curr->next != head) {
      tmp = curr;
      curr = curr->next;
      free(tmp);
    }
  }

  /* --- Helper functions --- */

  ListNode* get_node(ListNode *head, int n) {
    ListNode *curr = head;
    int i;

    for (i = 0; i < n && curr->next != head; i++) {
      curr = curr->next;
    }

    if (i == n) {
      return curr;
    }
    else {
      return NULL;
    }
  }

  void append_nodes(ListNode *head, int n) {
    ListNode *last = get_last(head);
    ListNode *curr = last;

    for (int i = 1; i <= n; i++) {
      curr->next = create_node(i);
      curr = curr->next;
    }

    curr->next = head;
  }

  void print_node(ListNode *node, char line_break) {
    cout << node->val << line_break;
  }

  void print_list(ListNode *head) {
    ListNode *curr = head;

    do {
      print_node(curr, ' ');
      curr = curr->next;
    } while (curr != head);

    cout << "\n";
  }
```

Circular List Split In Half

Split a circular list in two equal size halves.

Time complexity: O(N), where N is the number of nodes of the given list.
Space complexity: O(1).

Explanation. you need to get to the middle node and its previous node and the last node. Then, you can make previous node point to head, and last node point to the middle one.
To get to the middle you need to have two pointers, one that will be the middle node and another one that will be the last node. "Middle" pointer needs to advance with half the velocity of the "last" pointer.

```
ListNode* split_in_half(ListNode *head) {
  ListNode *mid = head, *last = head, *prev = NULL;
  int count = 0;

  if (head == NULL) return NULL;

  while (last->next != head) {
    count++;
    if (count % 2) {
      prev = mid;
      mid = mid->next;
    }
    last = last->next;
  }

  if (prev == NULL) return NULL;

  last->next = mid;
  prev->next = head;

  return mid;
}
```

Doubly Linked List

Implementation of a Doubly Linked List.

```
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

ListNode* partition(ListNode *head, ListNode *last) {
  int pivot = last->val;
  ListNode *wall = head->prev;

  for (ListNode *curr = head; curr != last; curr = curr->next) {
    if (curr->val <= pivot) {
      wall = (wall == NULL) ? head : wall->next;
      swap(&(curr->val), &(wall->val));
    }
  }

  wall = (wall == NULL) ? head : wall->next;
  swap(&(last->val), &(wall->val));

  return wall;
}

void _quicksort(ListNode *head, ListNode *last) {
  if (last != NULL && head != last && head != last->next) {
    ListNode *pivot_ptr = partition(head, last);
```

```
      _quicksort(head, pivot_ptr->prev);
      _quicksort(pivot_ptr->next, last);
    }
  }

  void quicksort(ListNode *head) {
    ListNode *last = get_last(head);
    _quicksort(head, last);
  }

  ListNode* create_node(int val) {
    ListNode *node = (ListNode *)malloc(sizeof(ListNode));

    node->val = val;
    node->prev = NULL;
    node->next = NULL;

    return node;
  }

  ListNode* get_last(ListNode *head) {
    ListNode *curr = head;

    while (curr->next != NULL) {
      curr = curr->next;
    }

    return curr;
  }

  void append(ListNode *head, int val) {
    ListNode *last = get_last(head);
    ListNode *node = create_node(val);

    node->prev = last;
    last->next = node;
  }

  void push(ListNode **head, int val) {
    ListNode *node = create_node(val);

    (*head)->prev = node;
    node->next = *head;
    *head = node;
  }

  void insert_after(ListNode *previous_node, int val) {
    ListNode *new_node = create_node(val);
    ListNode *previous_next = previous_node->next;

    previous_next->prev = new_node;
    new_node->prev = previous_node;

    previous_node->next = new_node;
    new_node->next= previous_next;
  }

  // int remove_node(ListNode **head, ListNode* node) {
  //   ListNode *curr = *head;

  //   while (curr != node && curr != NULL) {
  //     curr = curr->next;
  //   }

  //   if (curr == node) {
  //     /* Remove head */
  //     if (curr->prev == NULL) {
  //       *head = curr->next;
  //       (*head)->prev = NULL;
```

```
//     }
//     /* Remove last */
//     else if(curr->next == NULL) {
//       curr->prev->next = NULL;
//     }
//     else {
//       curr->prev->next = curr->next;
//       curr->next->prev = curr->prev;
//     }
//     free(curr);

//     return 1;
//   }

//   return 0;
// }

/*
 * Optimized remove_node(). Instead of pointing to nodes we point to the
 * "next" pointers (or in other words to the pointers that points to nodes).
 */
int remove_node(ListNode **head, ListNode* node) {
  ListNode **curr = head;

  while (*curr != node && *curr != NULL) {
    curr = &((*curr)->next);
  }

  if (*curr == node) {
    *curr = node->next;
    if (node->next != NULL) node->next->prev = node->prev;
    free(node);

    return 1;
  }

  return 0;
}

void free_list(ListNode *head) {
  ListNode *curr = head, *tmp = NULL;

  while (curr != NULL) {
    tmp = curr;
    curr = curr->next;
    free(tmp);
  }
}

/* --- Helper functions --- */

ListNode* get_node(ListNode *head, int n) {
  ListNode *curr = head;
  int i;

  for (i = 0; i < n && curr != NULL; i++) {
    curr = curr->next;
  }

  if (i == n) {
    return curr;
  }
  else {
    return NULL;
  }
}

void append_nodes(ListNode *head, int n) {
  ListNode *last = get_last(head);
```

```
    ListNode *prev = last;
    ListNode *new_node = NULL;

    for (int i = 1; i <= n; i++) {
      new_node = create_node(i);
      new_node->prev = prev;
      prev->next = new_node;
      prev = new_node;
    }
  }

  void print_node(ListNode *node, char line_break) {
    cout << node->val << line_break;
  }

  void print_list(ListNode *head) {
    ListNode *curr = head;

    while (curr != NULL) {
      print_node(curr, ' ');
      curr = curr->next;
    }
    cout << "\n";}

  void print_list_reverse(ListNode *head) {
    ListNode *last = get_last(head);
    ListNode *curr = last;

    while (curr != NULL) {
      print_node(curr, ' ');
      curr = curr->prev;
    }
    cout << "\n";
  }
```

Dp All Palindrome Partitions

```
Given a string s, partition s such that every string of the partition is a
palindrome. Return all possible palindrome partitioning of s.

For example, given s = "aab",
Return
[
  ["a","a","b"]
  ["aa","b"],
]
```

Time complexity: probably exponential.

Explanation: Walk through the string. At each character position you are, divide the string in a left part and a right part. For example "a" and "ab". Check if the left part is a palindrome and if it is call the function again for the right part. When you arrive at the end of the substring you currently are analysing return a vector> that will be empty if nothing were a palindrome at that particular call. When you return this vector> to a previous call of your function walk through it and for each vector insert at the beginning the left string of that function and then push_back this vector in the vector> of the function.

I will try to explain better why this "concatenation" of the vector and vector> works. Suppose, you returned v':
v' = [["a", "a"], ["aa"]]
And that your prefix (left part) at the current execution of your function is

```
"b". So, you can insert your prefix at the beginning of each vector:
v' = [ ["b", a", "a"], ["b", aa"] ]
And then push_back each vector of v' in the vector> V
of your function:
V = [ ["b", a", "a"], ["b", aa"] ]
So, trying to summarize, for each prefix (left part) you checked that is a
palindrome you will have a vector> v' returned with the
possible partitions, so you put your prefix together and push_back in your
vector> V.

Remember of checking after your loop if the whole substring you are analysing
at that call is a palindrome. The whole substring will not be analysed in the
loop because it can't be divided in left and right parts.
```

```
string strrev(string a) {
  reverse(a.begin(), a.end());
  return a;
}

string copystr(string a) {
  char *c = new char [a.size()+1];
  copy(a.begin(), a.end(), c);
  c[a.size()] = '\0';

  return string(c);
}

bool check(string a) {
  if (a != "" && a == strrev(a)) return true;
  return false;
}

vector<vector<string>> partition(string a) {
  vector<vector<string>> v;

  if (a.size() == 1) {
    vector<string> t;
    t.push_back(a);
    v.push_back(t);
    return v;
  }

  for (int i = 1; i < a.length(); i++) {
    string l = copystr(a).erase(i);
    l.erase(i);
    string r = a.substr(i);

    if (check(l)) {
      vector<vector<string>> v2 = partition(r);

      for (int j = 0; j < v2.size(); j++) {
        vector<string> t;
        t.push_back(l);
        t.insert(t.end(), v2[j].begin(), v2[j].end());
        v.push_back(t);
      }
    }
  }

  if (check(a)) {
    vector<string> t;
    t.push_back(a);
    v.push_back(t);
  }

  return v;
}
```

Dp Arrange Horses

You are given a sequence of black and white horses, and a set of K stables numbered 1 to K. You have to accommodate the horses into the stables in such a way that the following conditions are satisfied:

You fill the horses into the stables preserving the relative order of horses. For instance, you cannot put horse 1 into stable 2 and horse 2 into stable 1.

No stable should be empty and no horse should be left unaccommodated.

Take the product (number of white horses * number of black horses) for each stable and take the sum of all these products. This value should be the minimum among all possible accommodation arrangements.

Note: If a solution is not possible, then return -1,

Example:
Input: {WWWB} , K = 2
Output: 0, because we have 3 choices {W, WWB}, {WW, WB}, {WWW, B}
for first choice we will get 1*0 + 2*1 = 2,
for second choice we will get 2*0 + 1*1 = 1,
for third choice we will get 3*0 + 0*1 = 0.
Of the 3 choices, the third choice is the best option.

Time complexity: O(N * M), where N is the number of horses and M is the number of stables.
Space complexity: O(M).

Explanation: think about the recursive approach and then use memoization. The recursive approach is to call the function for each stable and then try all the combinations of horses for that stable. The combination is easy, you just need a for loop because we need to maintain the order of the horses (we can't permutate them). So, in our funcition signature we also will need to keep where in our vector of horses we must start at that call. If our first stable is trying to keep for example horses 0 and 1, in our next call/stable we need to start from the horse 2. Basically, we have:
for(i = start to the end) arrange(horses, stables - 1, i + 1)
Now, if you think a little you will notice that we are repeating a lot of calculations. When we start an iteration from index 2 to 5 and 2 stables, in our next call for 2 stables (that comes from the "for" loop of 3 stables), we gonna iterate from 3 to 5, so we are recalculating the results for horses of indices 3, 4, 5. This would be exponential, but if we memoize we can make this polynomial.

```
int _arrange(string a, int b, int start, map<pair<int, int>, int> &cache) {
  pair<int, int> key = make_pair(start, b);
  if (cache.find(key) != cache.end()) return cache[key];

  int nb = 0, nw = 0, product, temp, ans = INT_MAX;

  for (int i = start; i < a.size(); i++) {
    if (a[i] == 'W') nw++;
    else if (a[i] == 'B') nb++;
    product = nw * nb;

    if (b > 1 && i + 1 < a.size()) {
      temp = _arrange(a, b - 1, i + 1, cache);
      if (temp < INT_MAX) ans = min(ans, temp + product);
      cache[make_pair(i + 1, b - 1)] = temp;
    }
  }
  if (b == 1) ans = product;

  return ans;
}

int arrange(string a, int b) {
  if (b == 0 || a.size() == 0 || b > a.size()) return -1;

  map<pair<int, int>, int> cache;

  return _arrange(a, b, 0, cache);
}

static uint64_t now() {
  struct timeval t;
  gettimeofday(&t, NULL);
  return ((uint64_t) t.tv_sec) * 1000000 + (uint64_t) t.tv_usec;
}
```

Dp Different Bst

Given N, how many structurally unique BST's (binary search trees) that store values 1...N?

Example:
Given N = 3, there are a total of 5 unique BST's.

```
   1         3     3      2      1
    \       /     /      / \      \
     3     2     1      1   3      2
    /     /       \                 \
   2     1         2                 3
```

Time complexity: O(N^2), where N is the given number.
Space complexity: O(N).

Explanation: we start from the base case where N = 1, and the number of different BST's is one. Give N bigger than 1 we need to calculate all the answers for i < N, so we can finally calculate N. Notice that at a given node value i, this node can assume all positions in the tree. So, when our current node i is at position j, we want to check how many different BST's we have above i at position j, and how many we have under i at position j. We multiply these two number of possibilities to find the total number of combinations. We then accumulate the result for all positions j < i.

This exercise has the same approach as the exercise "Number of chords in a circle that don't intersect", including the same results:

```
for N: 1  2  3   4   5 ...
ways:  1  2  5  14  42 ...
```

```
int numTrees(int n) {
  if (n <= 0) return 0;

  vector<int> ans;
  ans.push_back(1);
  ans.push_back(1);

  for (int i = 2; i <= n; i++) {
    int sum = 0;
    for (int j = 0; j < i; j++) {
      sum += (ans[j] * ans[i - 1 - j]);
    }
    ans.push_back(sum);
  }

  return ans.back();
}

static uint64_t now() {
  struct timeval t;
  gettimeofday(&t, NULL);
  return ((uint64_t) t.tv_sec) * 1000000 + (uint64_t) t.tv_usec;
}
```

Dp Edit Distance

Given two words A and B, find the minimum number of steps required to convert A to B. Each operation is counted as 1 step.
You have the following 3 operations permitted on a word:
Insert a character
Delete a character
Replace a character

Example:
edit distance between "Anshuman" and "Antihuman" is 2.
Operation 1: Replace s with t.
Operation 2: Insert i.

Time complexity: O(M * N), where M is the length of string 1 and N is the length of string 2.
Space complexity: O(M * N).

Explanation: our dp variable will have size M * N, where M is the length of string 1 and N is the length of string 2. The idea is that for each pair of characters that we are looking at we decide which operation we want to perform (insertion, deletion, replace). For example, imagine we have:

```
    0 1 2 3
s1: a a b
s2: b b b a
```

And we are looking to the pair (s1[2] = b, s2[3] = a).
We can insert "a" (s2[3]) and reuse the result we have for *(aab, bbb), or we can delete "b" (s1[2]) and reuse the result we have for **(aa, bbba), or we can replace "b" with "a" (s1[2] with s2[3]) and reuse the result we have for ***(aa, bbb).
*: dp[i - 1][j]
**: dp[i][j - 1]
***: dp[i - 1][j - 1]
The results above suppose the columns of the dp matrix are composed by s1, and the rows by s2.
Notice you need to add 1 for the results above because you need to take into account you are performing an operation.
Notice too that if characters of the pair we are looking at are equal the result is a little different, because in the case of the replace operation you don't need to add 1.

The complete dp matrix for the example above would be:

```
     a a b
   0 1 2 3
b: 1 1 2 2
b: 2 2 2 2
b: 3 3 3 2
a: 4 3 3 3
```

The first column and first row are for empty strings, where the number of operations is decided by the size of the non-empty string.

```
int minDistance(string a, string b) {
  vector<vector<int>> dp(a.size() + 1, vector<int>(b.size() + 1, 0));

  for (int i = 0; i <= a.size(); i++) dp[i][0] = i;
  for (int i = 0; i <= b.size(); i++) dp[0][i] = i;

  for (int i = 1; i <= a.size(); i++) {
    for (int j = 1; j <= b.size(); j++) {
      if (a[i - 1] == b[j - 1]) {
        dp[i][j] = min(dp[i - 1][j - 1], min(dp[i - 1][j], dp[i][j - 1]) + 1);
      }
      else {
        dp[i][j] = min(dp[i - 1][j - 1], min(dp[i - 1][j], dp[i][j - 1])) + 1;
      }
    }
  }

  return dp[a.size()][b.size()];
}
```

Dp Equal Average Partition

Given an array with non negative numbers, divide the array into two parts such that the average of both the parts is equal. Return both parts (if they exist). If there is no solution. return an empty list.

Example:
Given,
[1 7 15 29 11 9]

Output [9 15] [1 7 11 29]

The average of part is (15+9)/2 = 12,
average of second part elements is (1 + 7 + 11 + 29) / 4 = 12

Note 1: If a solution exists, you should return a list of exactly 2 lists of integers A and B which follow the following condition:
numElements in A <= numElements in B
If numElements in A = numElements in B, then A is lexicographically smaller than B

Note 2: If multiple solutions exist, return the solution where length(A) is minimum. If there is still a tie, return the one where A is lexicographically smallest.

Note 3: Array will contain only non negative numbers.

Time complexity: O(N^2 * S), where N is the number of elements in the array and S is the maximum value the sum of the elements of this array can be. So, this time complexity is pseudo-polynomial.
Space complexity: O(N^2 * S).

Explanation: this exercise is tough depending on how you approach it. First, you need to realize a mathematical fact, after you need to speed up the algorithm and I think this is tough because if you try a bottom-up approach you will not get your answer fast enough, so you need to use memoization! I tried to use the dp approach for the knapsack problem and it is fast to find an answer, but will not necessarily be the correct answer because this exercise asks for the lexicographic smaller solution. So, once you complete

```
the knapsack table you would need to get all the possible answers to get the
smallest one and this process is exponential. So, to get the right answer
you need to sort the array and do a top-down approach because during the
process you naturally would find the smallest ones first.
The mathematical fact is:

SUM_of_Set1 / size_of_set1 = SUM_of_Set2 / size_of_set2
SUM_of_Set1 = SUM_of_Set2 * (size_of_set1 / size_of_set2)

total_sum = Sum_of_Set1 + Sum_of_Set2
and size_of_set2 = total_size - size_of_set1

Sum_of_Set1 =
  (total_sum - Sum_of_Set1) * (size_of_set1 / (total_size - size_of_set1))
OR on simplifying,
  total_sum / Sum_of_Set1 - 1 = (total_size - size_of_set1) / size_of_set1
  total_sum / Sum_of_Set1 = total_size / size_of_set1
  Sum_of_Set1 / size_of_set1 = total_sum / total_size

Now we can just iterate on size_of_set1, and we would know the required
Sum_of_Set1.

To summarize, the above is saying the average of the whole list is equal
the average of the parts if the average of the parts are equal.

Now that you know what you need to know (use memoization and the
mathematical fact) you need to implement a recursive functin to calculate
Sum_of_Set1. This recursive function return true if the sum exists, false
otherwise.

isPossible(index, sum, size):
  // Include the current element
  isPossible(index + 1, sum - array[index], size - 1) OR
  // Or don't include it
  isPossible(index + 1, sum, size)

Memoize the above function with a 3-dimensional vector and you are done.
Notice, you need in this function body to store the values that are part of
your group 1 (I call group 1, the group that gives Sum_of_Set1). Look at
the code to understand exactly how this is done, but it is simple. Just
push_back your values and pop_back them if you returned false.
```

```cpp
vector<vector<vector<bool> > > dp;
vector<int> res;
vector<int> original;
int total_size;

bool possible(int index, int sum, int sz) {
  if (sz == 0) return (sum == 0);
  if (index >= total_size) return false;

  if (dp[index][sum][sz] == false) return false;

  if (sum >= original[index]) {
    res.push_back(original[index]);
    if (possible(index + 1, sum - original[index], sz - 1)) return true;
    res.pop_back();
  }

  if (possible(index + 1, sum, sz)) return true;

  return dp[index][sum][sz] = false;
}

vector<vector<int> > avgset(vector<int> &Vec) {
  sort(Vec.begin(), Vec.end());
  original.clear();
  original = Vec;
```

```
    dp.clear();
    res.clear();

    int total_sum = 0;
    total_size = Vec.size();

    for(int i = 0; i < total_size; ++i) total_sum += Vec[i];

    dp.resize(original.size(), vector<vector<bool> > (total_sum + 1, vector<bool> (to

    // We need to minimize size_of_set1. So, lets search for the first
    // size_of_set1 which is possible.
    for (int i = 1; i < total_size; i++) {
      // Sum_of_Set1 has to be an integer
      if ((total_sum * i) % total_size != 0) continue;

      int Sum_of_Set1 = (total_sum * i) / total_size;

      if (possible(0, Sum_of_Set1, i)) {
        // Lets get the elements in group 2 now that we have group 1.
        int ptr1 = 0, ptr2 = 0;
        vector<int> res1 = res;
        vector<int> res2;
        while (ptr1 < Vec.size() || ptr2 < res.size()) {
          if (ptr2 < res.size() && res[ptr2] == Vec[ptr1]) {
            ptr1++;
            ptr2++;
            continue;
          }
          res2.push_back(Vec[ptr1]);
          ptr1++;
        }

        vector<vector<int> > ans;
        ans.push_back(res1);
        ans.push_back(res2);
        return ans;
      }
    }

    // Not possible.
    vector<vector<int> > ans;
    return ans;
  }
```

Dp Intersecting Chords

```
Given a number N, return number of ways you can draw N chords in a circle
with 2*N points such that no 2 chords intersect. Two ways are different if
there exists a chord which is present in one way and not in other.
Return answer modulo 109+7.
```

Example:

```
N = 2
If points are numbered 1 to 4 in clockwise direction, then different ways to
draw chords are: {(1-2), (3-4)} and {(1-4), (2-3)}

So, we return 2.

Note 1: 1 ≤ N ≤ 1000
```

Time complexity: O(N^2), where N is the number of chords.
Space complexity: O(N).

Explanation: the way I did this exercise maybe is not the simplest one, but once you get it, it is easy. The first thing I realized was that if we enumerate the cords in increasing order and in clockwise orientation the chord that has 1 as its beginning point will be connected to all even numbers after it. It needs to be even numbers because we need to have a even amount of points between the chord we connect to be sure these points in between that are left can all be connected. For example, 1 -> 4, left us with 2 and 3, so they can be connected. 1 -> 6, left us with 2, 3, 4, and 5 so they can be connected. Notice, that now we know one chord that will be present in each possible combination. For example, if N is 3 so we know that in each group we gonna have one of the following chords 1 -> 2, 1 -> 4 or 1 -> 6, since we need to connect all points. So, what we need to figure it out is how many combinations we can do with the points we are left with when we connect 1 -> 2 or 1 -> 4, 1 -> 6, etc. That is where we use DP. If we know the base case that is 1 (when we have N = 1) we can find the others. For example, in N = 2 we have 4 points (1, 2, 3, 4). If we connect 1 -> 2, we are left with 3 and 4, since they are just two points they can just be connected in one way. If we connect 1 -> 4, we are left with 2 and 3 and again they can just be connected in one way. This give us for N = 2, 2 possible combinations. Now, lets look at N = 3:

1 -> 2, left with 3, 4, 5, 6. They are 4 points and from our previous calculation we already know that for 4 points we have 2 possible combinations.

1 -> 4, left with 2, 3 | 5, 6. This "|" symbol is used to indicate I have a point being used in the middle that is 4. So we need to look at 2 and 3 separately to 5 and 6. 2 and 3 = 1 combination, 5 and 6 = 1 combination. So, we have 1 * 1 = 1 (Note the multiplication here).

1 -> 6, left with 2, 3, 4, 5. They are four points so we know we can make 2 combinations.

Summing up: 2 + 1 + 2 = 5. And we keep doing this until we calculated our N. Notice, that when we have groups that need to be looked separately like above (2, 3 | 5, 6) we need to multiply the numbers we get to have the correct result of combinations of that iteration.

Notice too our answers for different N's will be:
N = 1, 1
N = 2, 2
N = 3, 5
N = 4, 14
N = 5, 42
I'm pointing that out because there are another DP exercises where you will find the answers being these numbers and all of them can be solved in the same way.

```
int chordCnt(int n) {
  vector<int> counts;
  counts.push_back(0);
  counts.push_back(1);

  for (int i = 2; i <= n; i++) {
    int chordCount = 0;
    long long left = 1, right = 1;
    for (int j = 2; j <= 2 * i; j += 2) {
      if ((2 * i) - j >= 2) {
        left = counts[((2 * i) - j) / 2];
      }

      if (j - 2 >= 2) {
        right = counts[(j - 2) / 2];
      }

      chordCount = (chordCount + ((left * right) % 1000000007)) % 1000000007;
    }
    counts.push_back(chordCount);
  }

  return counts.back();
}
```

Dp Jump Game

Given an array of non-negative integers, you are initially positioned at the first index of the array. Each element in the array represents your maximum jump length at that position. Determine if you are able to reach the last index.

Example:
A = [2, 3, 1, 1, 4], return 1 (true).

A = [3, 2, 1, 0, 4], return 0 (false).

Return 0/1 for this problem.

Time complexity: O(N), where N is the size of the array.
Space complexity: O(1).

Explanation: we start from the end of our array and we keep a variable that tell us the minimum index in the array we need to achive to be sure we can arrive at the end of the array in the end of the game (call this variable minIndex). At the beginning this variable is the index of the last element. We start our iteration checking the element before the last element and we see if the number of maximum steps we can take is enough (this means: current i + steps needs to be larger or equal) to arrive at minIndex. If it is we update minIndex assigning the index of the element we just checked. We keep doing this until we are at index 0 and as before we check if we can arrive at minIndex, and if we can we return 1, otherwise 0.

```
int canJump(vector<int> &a) {
  if (a.size() == 0 || a.size() == 1) return 1;

  int minIndexPossible = a.size() - 1;

  for (int i = a.size() - 2; i >= 0; i--) {
    if (a[i] + i >= minIndexPossible) {
      if (i == 0) return 1;
      minIndexPossible = i;
    }
  }

  return 0;
}
```

Dp Longest Subsequence

Find the longest increasing subsequence of a given sequence / array.

In other words, find a subsequence of array in which the subsequence’s elements are in strictly increasing order, and in which the subsequence is as long as possible. This subsequence is not necessarily contiguous, or unique. In this case, we only care about the length of the longest increasing subsequence.

Time complexity: O(N), where N is the number of elements in the array.
Space complexity: O(N).

Explanation: start at the end of the array. The last element is our base case and has subsequence size of 1. The element next to it will have subsequence size 1 or 2, 2 if it is smaller than the last element. We, continue this process and at each element we are we can not just check the element at its right, but all the elements to its right to the end of the array. For that particular element the longest subsequence will be 1 plus the largest size already calculated for elements at its right that are bigger than it.

```
int lis(const vector<int> &a) {
  if (a.size() == 0) return 0;

  int m = 1;
  vector<int> ans(a.size(), 0);
  ans[ans.size() - 1] = 1;

  for (int i = a.size() - 2; i >= 0; i--) {
    int t = 1;
    for (int j = i + 1; j < ans.size(); j++) {
      if (a[j] > a[i]) t = max(t, ans[j] + 1);
    }
    ans[i] = t;
    m = max(m, t);
  }

  return m;
}
```

Dp Longest Valid Parentheses

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.

Example:
For "(()", the longest valid parentheses substring is "()", which has length = 2.
Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.

Time complexity: O(N), where N is the length of the string.
Space complexity: O(N).

Explanation: there are two ways that I know to solve this exercise. One, uses a stack, the other one, dynamic programming. First of all, notice that it is simple to know if the sequence is valid with a stack, but find out the length of the sequence using the stack approach is trickier.

Initialize a dp variable with the size as the length of the string and 0 as the initial values. For each parenthesis in the input string we check if it is a closing parenthesis ')', because if it is we need to check if it is part of a valid sequence and update the dp variable with the length of this valid sequence. To do that we need to identify the opening parenthesis of this sequence and (if this opening parenthesis exist) sum with the length of the valid sequence we found previously to this opening parenthesis. Notice, that for each valid parenthesis pair we add 2, because they are composed by two characters ('(' and ')'). For example:

```
     0 1 2 3 4 5 6
```

```
 s = ( ( ( ) ( ) )
dp = 0 0 0 0 0 0 0
At s[3] we have a ')', so we get the opening parenthesis using the length
of the valid sequence we had before s[3], or in other words, we get the value
of dp[2]. In this case is 0, so the opening parenthesis for s[3] should be at
s[2], which is in fact a '(' making a valid pair, so we add dp[2] + 2. Then,
we add the length of the valid sequence we had before s[2], dp[1], which is
again 0. We get:
     0 1 2 3 4 5 6
dp = 0 0 0 2 0 0 0
At s[5] we have again a ')', so we get the opening parenthesis checking the
length of dp[4], which is 0, hence the opening parenthesis for s[5] should be
at s[4], which is in fact a '(', so we do dp[4] + 2. Now, we add the length
of the sequence prior to the opening parenthesis, which is dp[3] = 2. We get:
     0 1 2 3 4 5 6
dp = 0 0 0 2 0 4 0
Repeating the process at s[6], we get:
     0 1 2 3 4 5 6
dp = 0 0 0 2 0 4 6

Return the maximum value of the dp variable.
```

```cpp
int longestValidParentheses(string a) {
  vector<int> dp(a.size(), 0);
  int ans = 0;

  for (int i = 1; i < a.size(); i++) {
    int opening = i - (dp[i - 1] + 1);
    if (opening >= 0 && a[i] == ')' && a[opening] == '(') {
      dp[i] = dp[i - 1] + 2 + (opening - 1 >= 0 ? dp[opening - 1] : 0);
    }
    ans = max(ans, dp[i]);
  }

  return ans;
}

// // This solution uses a stack and is also O(N) time.
// int longestValidParentheses(string a) {
//   // Create a stack and push -1 as initial index to it.
//   stack<int> stk;
//   stk.push(-1);

//   // Initialize ans
//   int ans = 0;

//   // Traverse all characters of given string
//   for (int i = 0; i < a.size(); i++) {
//     // If opening bracket, push index of it
//     if (a[i] == '(') stk.push(i);
//     else { // If closing bracket, i.e.,a[i] = ')'
//       // Pop the previous opening bracket's index
//       stk.pop();

//       // Check if this length formed with base of
//       // current valid substring is more than max
//       // so far
//       if (!stk.empty()) ans = max(ans, i - stk.top());
//       // If stack is empty. push current index as
//       // base for next valid substring (if any)
//       else stk.push(i);
//     }
//   }

//   return ans;
// }
```

Dp Max Coin Sum

You are given a set of coins S. In how many ways can you make sum N assuming you have infinite amount of each coin in the set.

Note : Coins in set S will be unique. Expected space complexity of this problem is O(N).

Time complexity: O(N * size of S).
Space complexity: O(N).

Explanation: we need two loops. One to go coin by coin and another one to ivestigate all the sums from 0..N. Our base case is the sum 0, where we always have one way of doing it and it is by picking no coin.
So, we create an array of size N + 1 (the +1 is to accomodate our base case). Then, we start with the first coin and for each possible sum we check in our array the element a[currentSum - currentCoin], where currentSum is the current sum we are looking at in our loop. The number of ways of doing this currentSum will be the already stored value in this currentSum that is a[currentSum], plus the value at a[currentSum - currentCoin] indicating the number of ways we can make the sum "currentSum - currentCoin".

Example:
S = [1, 2, 3]
N = 4

```
indices/sums: 0 1 2 3 4
initial   a = 1 0 0 0 0
coin 1    a = 1 1 1 1 1
coin 2    a = 1 1 2 2 3
coin 3    a = 1 1 1 3 4
```

Our answer is the last element of this array, that is 4.

```
int coinchange(vector<int> &s, int n) {
  if (s.size() == 0 || n == 0) return 1;

  vector<int> ans(n + 1, 0);

  ans[0] = 1;

  for (int i = 0; i < s.size(); i++) {
    for (int j = 1; j < ans.size(); j++) {
      if (j < s[i]) continue;
      ans[j] = (ans[j] + ans[j - s[i]]) % 1000007;
    }
  }

  return ans[ans.size() - 1];
}
```

Dp Max Not Adjacent Sum

Given a 2 * N Grid of numbers, choose numbers such that the sum of the numbers is maximum and no two chosen numbers are horizontally, vertically or diagonally adjacent, and return it.

Example:

```
Given the grid
  1 2 3 4
  2 3 4 5
Return 8, because we choose 3 and 5 so sum will be 8.
```

Time complexity: O(N), where N is the number of columns of the grid.
Space complexity: O(1).

Explanation: first you need to realize that not get an element adjacent to another in this problem means that for a chosen element you can only get another element after the next column of this chosen element, or in other words, you need to have one column of space between chosen elements. Second, at each column we are at the element that matters to us is always the largest one. Third, lets see with an example how we can reuse previous calculated information to get our answer:

```
Given,
9, 12, 4, 10, 8, 11
4, 6, 13, 24, 1, 15
```

Create an empty array with the number of columns of the grid, call it b.

At the beginning, we look at the first column and since this array we created is empty we store into it max(9, 4), which is 9. After, at the second position we store max(9, 12, 6), which is 12.

So right now we have: 9 12 _ _ _ _

Now, at the third column we can realize our first sum. So, we check if 9 + max(13, 4) is bigger than 12, and it is, so we store 9 + 13: 9 12 22 _ _ _

At the fourth column we again use the previous information stored in the array to get our largest sum. We check max(10, 24) + 12 > 22, and it is, so we store 24 + 12: 9 12 22 36 _ _
And we keep doing that until we checked the entire grid.

What we are doing is: if we already calculated what is the largest sum for, lets suppose our 3 first columns, when looking at the fourth column we know that we already know the largest sum for 2 and 3 first columns, so we check if using the maximum element of the fourth column with the largest sum for the 2 first columns is larger than the sum for the first 3 columns. If it is not we just propagate the sum of the first 3 columns.

Notice we can use constant space, because instead of this array we can just use variables to keep the two last sums we calculated.

```cpp
int adjacent(vector<vector<int>> &a) {
  if (a.size() == 0 || a[0].size() == 0) return 0;

  int m, s1 = 0, s2 = 0;

  for (int i = 0; i < a[0].size(); i++) {
    int curr = max(a[0][i], a[1][i]);

    if (curr + s1 >= s2) {
      int temp = s1;
      s1 = s2;
      s2 = curr + temp;
    }
    else {
      s1 = s2;
    }
  }

  return s2;
}
```

Dp Max Product Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest product. Return an integer corresponding to the maximum product possible.

Example:
Given [2, 3, -2, 4]
Return 6. Possible with [2, 3]

Time complexity: O(N), where N is the number of elements in the array.
Space complexity: O(1).

Explanation: this array has positive and negative numbers and zeros. Lets think about what happens when we find a 0. When we are looking at a contiguous subarray and we find a zero it means we need to start all our process again after this zero because any subarray containing 0 will have product 0. Lets think about the negative numbers. Imagine you are looking at a contiguous subarray that do not contain 0. You have two options, and two options only, or you find an odd number of negative numbers or you find an even number of negative numbers. If the number of negative signs are even the maximum product of that contiguous subarray without zeros will be the whole contiguous subarray. If you have an odd number of negative signs so you have two options: or you have your maximum product for that subarray before the first negative number, or after it.
So, for each subarray without zeros we need to keep two products: one for the whole subarray and another one that we start to count after the first negative number. At each iteration we store in a variable the maximum of these two products and our previous maximum.
Example:
2, 3, -2, 4
a[i] = 2 p1: 2 p2: 1 negativeFound: false max: 2
a[i] = 3 p1: 6 p2: 1 negativeFound: false max: 6
a[i] = -2 p1: -12 p2: 1 negativeFound: true max: 6
a[i] = 4 p1: -48 p2: 4 negativeFound: true max: 6

```
int maxProduct(const vector<int> &a) {
  if (a.size() == 0) return 0;

  int m = a[0], p1 = 1, p2 = 1;
  bool firstNeg = false;

  for (int i = 0; i < a.size(); i++) {
    m = max(m, a[i]);

    if (a[i] != 0) {
      p1 = p1 * a[i];
      m = max(m, p1);

      if (firstNeg == false && a[i] < 0) firstNeg = true;
      else if (firstNeg) {
        p2 = p2 * a[i];
        m = max(m, p2);
      }
    }
    else {
      firstNeg = false;
      p1 = 1;
      p2 = 1;
    }
  }

  return m;
}

int bruteMaxProduct(const vector<int> &a) {
  if (a.size() == 0) return 0;

  int mp = a[0], acc;

  for (int i = 0; i < a.size(); i++) {
    acc = a[i];
    mp = max(mp, acc);
    for (int j = i + 1; j < a.size(); j++) {
      acc *= a[j];
      mp = max(mp, acc);
    }
  }

  return max(mp, acc);
}
```

Dp Max Rectangle Binary Matrix

```
Given a 2D binary matrix filled with 0’s and 1’s, find the largest rectangle
containing all ones and return its area.

Bonus if you can solve it in O(n^2) or less.
```

Example:

```
A = [
       1 1 1
       0 1 1
       1 0 0
    ]

Output 4 as the max area rectangle is created by the 2x2 rectangle created
by (0,1), (0,2), (1,1) and (1,2).
```

Time complexity: O(N^2), where N is the number of elements in the matrix.
Space complexity: O(N).

Explanation: first thing we need is a matrix telling us the maximum number of consecutive 1's to the left of our current element, including the element (call this matrix "c"). I also used a matrix that tells me exactly how many consecutive 1's we have above the current element, including the current element, though this matrix is not necesary (see note 1). Now we iterate over our elements and for each element that is a 1 we check what is the maximum area of the rectangles that have this element as their bottom right corner. Calculate this area is simple, start from the row with the element and start to go up, incrementing your height by 1 every time. The width of the rectangle is the min between the previous calculated width and the value you can found in your matrix "c" for the same column of your element and the row you are currently checking.
Note1: You don't need this matrix of consecutives 1 in the y-direction because you could just go until you are at row with index 0 or when the column of your element at the row you are at is 0 (a[i - y][j] == 0), because this means it is impossible to find another rectangles that have the element a[i][j] as the bottom right corner.
Note 2: After calculating the matrix of consecutive 1's in the x-direction you could apply the algorithm of "max rectangle area in a histogram" to find the answer.

```
 int maximalRectangle(vector<vector<int>> &a) {
   if (a.size() == 0 || a[0].size() == 0) return 0;

   int area = 0;
   vector<vector<int>> r(a.size(), vector<int>(a[0].size(), 0));
```

```
    vector<vector<int>> c(a.size(), vector<int>(a[0].size(), 0));

    for (int i = 0; i < a.size(); i++) {
      for (int j = 0; j < a[0].size(); j++) {
        if (a[i][j]) {
          c[i][j] = j - 1 >= 0 ? 1 + c[i][j - 1] : 1;
          r[i][j] = i - 1 >= 0 ? 1 + r[i - 1][j] : 1;

          int y = 1, x = a[0].size();
          while (y <= r[i][j]) {
            x = min(x, c[i - (y - 1)][j]);
            area = max(area, x * y);
            y++;
          }
        }
      }
    }

    return area;
  }
```

Dp Max Stock Profit

Say you have an array for which the ith element is the price of a given stock on day i.
If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

Example:
Given [1 2]
Return 1

Time complexity: O(N), where N is the number of elements in the array.
Space complexity: O(1).

Explanation: we want max(a[j] - a[i]), where j is bigger than i. The maximum profit will be the biggest value - smallest value. So, we can initialize a variable (call it "smallest") with the first element and we update the value every time we find an a[i] smaller than this variable. When the element is not smaller than this variable we store in another variable: maxProfit = max(maxProfit, a[i] - smallest).

```
  int maxProfit(vector<int> &a) {
    if (a.size() <= 1) return 0;

    int sml = a[0], mp = 0;

    for (int i = 0; i < a.size(); i++) {
      if (a[i] < sml) {
        sml = a[i];
      }
      else {
        mp = max(mp, a[i] - sml);
      }
    }

    return mp;
  }
```

Dp Max Stock Profit-2

Say you have an array for which the ith element is the price of a given stock on day i.
Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Example:
Given [1 2]
Return 1

Time complexity: O(N), where N is the number of elements in the array.
Space complexity: O(1).

Explanation: we start at the last index and start to look for the element that has a neighbor that is smaller than it. For example, consider the array [4, 1, 3, 2]. In this array we would find the element 3 because it is the largest element from the end of the array that we could get until we find an element that is smaller (in this case, element 1). Now, we continue our loop until we find an element where its left neightbor is larger than it. So, in our example we would find 1, because it is smaller than 4. Now, we add the difference 3 - 1 to our answer. We continue the process until the loop is finished.

I will try to explain better the process described above. Look at:
[1, 3, 2, 10]
The answer here is (10 - 2) + (3 - 1) = 10, because it is better than 10 - 1 = 9. The process described above works because we are looking for big elements that can give us profit, so we first select 10 because we know that its left neighbor is smaller than it so we can get profit using 10, then when we find an element like 2 that has a neighbor 3 that is bigger than it, is when we can complete the transaction to get our profit. We can do this because even if we could have more profit looking for a smaller element than 2 we can recover and even get more money using our 3 element (think about it, if you would get more money using 10 would be with a number smaller than 2 like 1, so you would get 1 unit more of profit, but if you complete the transation at 2 and use 3 for a new transaction you have 2 units more than 1).

The official answer for this problem, kinda use the fact I descibed above, but in a more simple way:
Note 1: I will never buy a stock and sell it in loss.
Note 2: If A[i] < A[i+1], I will always buy a stock on i and sell it on i+1.

```
int maxProfit(vector &prices) {
  int total = 0, sz = prices.size();
  for (int i = 0; i < sz - 1; i++) {
    if (prices[i+1] > prices[i]) total += prices[i+1] - prices[i];
  }
  return total;
}
```

```
int maxProfit(const vector<int> &a) {
  if (a.size() <= 1) return 0;

  int bst = -1, mp = 0;

  for (int i = a.size() - 1; i >= 0; i--) {
    if (a[i] >= bst) {
      bst = a[i];
    }
    else {
      if (i == 0 || (i > 0 && a[i - 1] >= a[i])) {
        mp = mp + (bst - a[i]);
        bst = -1;
      }
    }
  }

  return mp;
}
```

Dp Max Stock Profit-3

Say you have an array for which the ith element is the price of a given stock on day i. Design an algorithm to find the maximum profit. You may complete at most two transactions.

Note: you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Example:
Given [1 2 1 2]
Output 2

Day 1: Buy
Day 2: Sell
Day 3: Buy
Day 4: Sell

Time complexity: O(N), where N is the number of elements in the array.
Space complexity: O(N).

Explanation: I solved this exercise in a different way from the editorial. I realized that at each day "i" that you are looking at you have a transaction that can happen until this day, and another one that can happen after this day (since you are allowed to make just two transactions). So the brute force way would be to look at each day and check what would be the maximum transaction at its left and at its right. The maximum profit would be max(current, left + right). So we need to speed up this process and for that we calculate an array for all the maximum profits to the left of day "i" in O(N) time. After we calculate an array starting at the end of our input that has the maximum profit at the right of day "i", also in O(N) time. Finally, iterate over these arrays looking for max(current, left + right).

```
int maxProfit(const vector<int> &a) {
  if (a.size() <= 1) return 0;

  int ans = 0, bst = INT_MIN, sml = INT_MAX, m = 0;
  int left[a.size()], right[a.size()];

  for (int i = 0; i < a.size(); i++) {
    if (a[i] < sml) sml = a[i];
    else m = max(m, a[i] - sml);
    left[i] = m;
  }

  m = 0;

  for (int i = a.size() - 1; i >= 0; i--) {
    if (a[i] > bst) bst = a[i];
    else m = max(m, bst - a[i]);
    right[i] = m;
  }

  for (int i = 0; i < a.size(); i++) {
    ans = max(ans, left[i] + right[i]);
  }

  return ans;
}

// int maxSingleTransactionProfit(const vector<int> &a, int s, int e) {
//   if (a.size() <= 1) return 0;

//   int sml = a[s], mp = 0;

//   for (int i = s; i < e; i++) {
//     if (a[i] < sml) {
//       sml = a[i];
//     }
//     else {
//       mp = max(mp, a[i] - sml);
//     }
//   }

//   return mp;
// }

// int maxProfit(const vector<int> &a) {
//   if (a.size() <= 1) return 0;

//   int ans = 0;

//   for (int i = 0; i < a.size(); i++) {
//     int l = maxSingleTransactionProfit(a, i, a.size());
//     int r = maxSingleTransactionProfit(a, 0, i + 1);
//     ans = max(ans, l + r);
//   }

//   return ans;
// }
```

Dp Max Sum Binary Tree

Given a binary tree, find the maximum path sum. The path may start and end at any node in the tree.

Example:

Given the below binary tree,

```
       1
      / \
     2   3
```
Return 6.

Time complexity: O(N), where N is the number of nodes in the tree.
Space complexity: O(logN), where logN is the height of the tree.

Explanation: first notice this is not exactly a path. You can go up the node you start at, but once you start to go down you can't go up again. In the above example the path would be: 2, 1, 3. So, what we want is the maximum of left, right, parent, left + parent, right + parent, left + parent + right. We start at the bottom left-most node and go up and we always return to the parent node max(left + parent, right + parent). And we keep a variable passed by reference to store the answer.

```
int _maxSum(TreeNode *r, int *ans) {
  int left = 0, right = 0;

  // Current, parent
  *ans = max(*ans, r->val);

  if (r->left) {
    left = _maxSum(r->left, ans);
    // Current, left, left + parent
    *ans = max(*ans, max(left, left + r->val));
  }

  if (r->right) {
    // Current, right, right + parent
    right = _maxSum(r->right, ans);
    *ans = max(*ans, max(right, right + r->val));
  }

    // Current, left + parent + right
  if (r->left && r->right) *ans = max(*ans, left + r->val + right);

  if (r->left == NULL) return r->val + right;
  else if (r->right == NULL) return r->val + left;

  return r->val + max(left, right);
}

int maxPathSum(TreeNode *r) {
  if (r == NULL) return 0;

  int ans = INT_MIN;

  _maxSum(r, &ans);

  return ans;
}

// void _maxSum(TreeNode *r, int acc, int *ans) {
//   if (r == NULL) return;

//   if (r->left == NULL && r->right == NULL) *ans = max(*ans, acc + r->val);

//   _maxSum(r->left, acc + r->val, ans);
//   _maxSum(r->right, acc + r->val, ans);
// }

// int maxSum(TreeNode *r) {
//   if (r == NULL) return 0;

//   int acc = 0, ans = INT_MIN;

//   _maxSum(r, acc, &ans);

//   return ans;
// }
```

Dp Min Sum M Partitions

Given an array with N integers and an integer M, find the minimum sum of the maximum elements of M contiguous partitions of the array.

Example:

Given [1, 2, 3, 4] and M = 2 (the array is not necessarily sorted)
Output 5, because possible partitions are
[1] [2, 3, 4] => 1 + 4 = 5
[1, 2] [3, 4] => 2 + 4 = 6
[1, 2, 3] [4] => 3 + 4 = 7

Time complexity: O(N^2 * M), were N is the size of the given array and M is the number of partitions.
Space complexity: O(N * M).

Explanation: to solve this problem we can use dynamic programming. The brute force solution would be to try all possible combinations which would be O(N^M). Lets see some important notes to get our DP solution supposing we start our algorithm from the end of the array to the beginning:

1. We want to find DP(i, j), where DP(i, j) is the the minimum sum of the array with "i" partitions and starting at element A[j].

2. If we decide that we want to partition the array at point "j" and we current have to perform "i" partitions, then the minimum sum will be A[j] (that is the maximum element because this partition has just one element) plus DP(i - 1, j + 1). Notice that when i = 0 we have just one partition because we are using zero based indexes and we need to consider the entire array, so the minimum sum will be the maximum element found so far. In other words, DP(0, j) will be the maximum element we found from A[A.size() - 1] to A[j].

3. The item 2 is half of our logic, the other half is that we need to figure out point j. For that we iterate from j to A.size() - i (we need to subtract "i" because one partition can use all elements in the array, but two partitions can use at most A.size() - 1, three, A.size() - 2, and so on). In each iteration we keep the maximum element we have so far for this partition that starts at j and we get:
min(DP(i, j), currMax + DP(i - 1, j + 1)).

For example,

Given A = [1, 3, 5, 2, 4, 8, 6, 1, 0, 2, 9, 1] and M = 3, our result DP table would be:

i/j	0	1	2	3	4	5	6	7	8	9	10	11
0	9	9	9	9	9	9	9	9	9	9	9	1
1	10	10	10	10	10	10	10	10	9	10	10	INF
2	11	13	15	12	14	17	15	10	10	12	INF	INF

And our answer would be DP[M - 1][0] = DP[2][0] = 11.

This table give us information like this: if our array starts at index 8 and we need to perform 2 partitions what is the minimum sum? It is DP[1][8] = 9. If we need to find DP[2][6], so we want to find the minimum sum if our array starts at index 6 and we need to have 3 partitions, then we check every element from indexes 6 to 9 keeping the maximum and for each of them we check DP[i - 1][j + 1] which are:

DP[1][7] = 10, DP[1][8] = 9, DP[1][9] = 10, DP[1][10] = 10,

So we get the minimum as currMax + DP[1][8] = 15 (where currMax = A[6] = 6). In other words, our minimum sum starting at 6 happens when we make the third partition at element 7:

[6, 1], [0], [2, 9, 1]

```
int minSum(vector<int> &a, int b) {
  int n = a.size();
  vector<vector<int>> dp(b, vector<int>(n, INT_MAX));
```

```
    dp[0][n - 1] = a.back();
    for (int i = n - 2; i >= 0; i--) dp[0][i] = max(a[i], dp[0][i + 1]);

    for (int i = 1; i < b; i++) {
      for (int j = n - i - 1; j >= 0; j--) {
        int currMax = INT_MIN;
        for (int k = j; k < n - i; k++) {
          currMax = max(currMax, a[k]);
          dp[i][j] = min(dp[i][j], currMax + dp[i - 1][k + 1]);
        }
      }
    }

    return dp[b - 1][0];
  }
```

Dp Min Sum Matrix

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

Example:
Input,

```
    [  1 3 2
       4 3 1
       5 6 1
    ]
```

Output 8, representing 1 -> 3 -> 2 -> 1 -> 1

Time complexity: O(M * N), where M is the number of rows and N the number of columns.
Space complexity: O(M * N).

Explanation: we gonna need a matrix of the same size of our input matrix. We initialize this matrix with infinite. Now, at each element of this matrix we check min(dp[i + 1][j], dp[i][j + 1]) + a[i][j], since we can just go right or down at each iteration.

```
 int minPathSum(vector<vector<int> > &a) {
   if (a.size() == 0 || a[0].size() == 0) return 0;

   vector<vector<int>> dp;
   int m = a.size();
   int n = a[0].size();

   for (int i = 0; i < m; i++) {
     vector<int> t;
     for (int j = 0; j < n; j++) t.push_back(INT_MAX);
     dp.push_back(t);
   }
   dp[m - 1][n - 1] = a[m - 1][n - 1];

   for (int i = m - 1; i >= 0; i--) {
     for (int j = n - 1; j >= 0; j--) {
       if (i + 1 < m) dp[i][j] = min(dp[i][j], a[i][j] + dp[i + 1][j]);
       if (j + 1 < n) dp[i][j] = min(dp[i][j], a[i][j] + dp[i][j + 1]);
     }
   }
```

```
  return dp[0][0];
}
```

Dp Min Sum Triangle

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below (with adjacent they mean one number to the right of the current number).
Bonus point if you are able to do this using only O(N) extra space, where N is the number of rows in the given triangle.

Time complexity: O(N * M), where N is the number of rows and M is the average length of rows.
Space complexity: O(N).

Example:
Given,

```
[
     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]
]
```

The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).

Explanation: we are asked to use O(N) space, where N is the number of rows. This N is also the number of elements in the last row since this is a triangle. So we initialize our dp variable with the last row and then iterate from the row before the last to the first row. At each row i, at each column j store in dp[j] = a[i][j] + min(dp[j], dp[j + 1]). Return dp[0] (since it is a triangle the first minimum path from the first row, that has just one element, is at dp[0]).

```
int minimumTotal(vector<vector<int> > &a) {
  if (a.size() == 0 || a[a.size() - 1].size() == 0) return 0;

  int nr = a.size();
  vector<int> dp(a[nr - 1]);

  for (int i = nr - 2; i >= 0; i--) {
    for (int j = 0; j < a[i].size(); j++) {
      dp[j] = a[i][j] + min(dp[j], dp[j + 1]);
    }
  }

  return dp[0];
}
```

Dp Palindrome Partitioning-2

Given a string s, partition s such that every substring of the partition is a palindrome. Return the minimum cuts needed for a palindrome partitioning of s.

Example:
Given,

```
s = "aab",
Return 1 since the palindrome partitioning ["aa","b"] could be produced
using 1 cut.
```

Time complexity: O(N^2), where N is the length of the string.
Space complexity: O(N).

Explanation: iterate over the string. For each character "i" check all the substrings made by the union of "i" with "j = i .. 0" are palindrome and when they are get min(current, dp[j - 1]). For example:

```
s = abba
i = 0, j = 0: dp = [0]
i = 1, j = 1 .. 0:
  substr(1 to 1) = b, so it is palindrome and gives us minCuts = 1
  substr(0 to 1) = ab, it is not a palindrome
  dp = [0 1]
i = 2, j = 2 .. 0:
  substr(2 to 2) = b, so it is palindrome and give us minCuts = 2
  substr(2 to 1) = bb, so it is palindrome and give us minCuts = 1
  substr(2 to 0) = abb, it is not a palindrome
  dp = [0 1 1]
i = 3, j = 3 .. 0:
  substr(3 to 3) = a, so it is palindrome and give us minCuts = 2
  substr(3 to 2) = ab, it is not a palindrome
  substr(3 to 1) = abb, it is not a palindrome
  substr(3 to 0) = abba, it is a palindrome and give us minCuts = 0
  dp = [0 1 1 0]
```

Now that we know our algorithm there is room for more improvements. Especificaly we need to make the algorithm that check if a string is palindrome faster. And we can use more DP for that. We need a square matrix N x N, where N is the length of the string where we gonna fill just half of if (we gonna fill the diagonal and all the elements under it). This is because this half matrix is enough to give us all the possible palindromes of a string. For example:

```
0 1 2 3
a b b a
```

Give us: 0, 1, 2, 3, 0->1, 0->2, 0->3, 1->2, 1->3, 2->3. Notive how j is never bigger than i, otherwise we would have substrings like 3->2, which is the same as 2->3.
Our diagonal elements in the matrix are i = j, and of course they are true because every single letter (substr(i, i) = one letter) in the string is a palindrome. Now we check if a[i] == a[j] AND isPalindrome[i - 1][j + 1] is true. This "if" statement is basically saying that if the letter at the beginning and end of the substring are equal and the substring from j + 1 to i - 1 is a palindrome (this j + 1 to i - 1 substring is the middle, or in other words, everything between the beginning and end of the current substring) so this current substring is also a palindrome. Notice I'm not wrong saying j + 1 to i - 1 substring, because this is what the element at isPalindrome[i - 1][j + 1] needs to represent.

```cpp
int minCut(string s) {
  int n = s.size();
  if (!n) return 0;

  vector<vector<bool> > isPalindrome(n, vector<bool>(n, false));
  int dp[n];

  for (int i = 0; i < n; ++i){
    isPalindrome[i][i] = true;
    dp[i] = i;
  }

  for (int i = 1; i < n; ++i) {
    for (int j = i; j >= 0; --j) {
      if (s[i] == s[j] && (i - j < 2 || isPalindrome[i - 1][j + 1])) {
        if (j == 0) {
          dp[i] = 0;
        }
        else {
          isPalindrome[i][j] = true;
          dp[i] = min(dp[j - 1] + 1, dp[i]);
        }
      }
    }
  }

  return dp[n - 1];
}

// bool isPalindrome(string &str, int s, int e) {
//   if (str.size() == 0) return true;

//   for (int i = s, j = 0; i < e; i++, j++) {
//     if (str[i] != str[e - j - 1]) return false;
//   }

//   return true;
// }

// int minCut(string str) {
//   if (str.size() <= 1) return 0;

//   vector<int> dp;

//   for (int i = 0; i < str.size(); i++) {
//     int j = i, mc = INT_MAX;

//     while (j >= 0) {
//       if (isPalindrome(str, j, i + 1)) {
//         if (j == 0) mc = -1;
//         else mc = min(mc, dp[j - 1]);
//       }
//       j--;
//     }
//     dp.push_back(mc + 1);
//   }

//   return dp.back();
// }

// bool isPalindrome(string &str, int s, int l) {
//   if (str.size() == 0) return true;

//   for (int i = s, j = 0; i < (s + l); i++, j++) {
//     if (str[i] != str[(s + l) - j - 1]) return false;
//   }

//   return true;
// }

// void partition(string &str, int cuts, int *ans, int s) {
//   if (s == str.size()) *ans = min(*ans, cuts);

//   for (int i = s; i < str.size(); i++) {
//     if (isPalindrome(str, s, i - s + 1)) {
//       partition(str, cuts + 1, ans, i + 1);
//     }
//   }
// }

// int minCut(string str) {
//   if (str.size() <= 1) return 0;

//   int ans = INT_MAX;

//   partition(str, 0, &ans, 0);
```

```
//    return ans - 1;
// }
```

Dp Regex Match

Implement wildcard pattern matching with support for '?' and '*'.
'?' : Matches any single character.
'*' : Matches any sequence of characters (including the empty sequence).
Note: The pattern must match the entire string.

```
Examples :
isMatch("aa","a") → 0
isMatch("aa","aa") → 1
isMatch("aaa","aa") → 0
isMatch("aa", "*") → 1
isMatch("aa", "a*") → 1
isMatch("ab", "?*") → 1
isMatch("aab", "c*a*b") → 0
```

Return 1/0 for this problem.

Time complexity: O(N*M), where N is the length of the string, and M is the length of the pattern.
Space complexity: O(M), where M is the length of the pattern.

Explanation: I used a different approach. The '*' in the pattern are not important to tell if there will be a match or not because they can match the empty string or anything else. What really tells you if a pattern will occur are the characters and '?' in the pattern. So, I divided the pattern in pieces at each '*', and each piece is made only of characters and '?'. Then, I try to check if all of these pieces exist in the input string. A problem with this approach is that it could be tricky to check the beginning and end of our input string, because I'm looking for substrings in the input, which can return false positives when we find a substring that we wanted to start at the very beginning of the input, example:
input: ba
pattern: a
This would return true, because "a" is a substring of "ba", but it should return false because the "a" we matched should be at the beginning. To fix these cases I insert in the pattern and input string the characters ^ and $ like this:
input: ^ba$
pattern: ^a$
That fix the problem because "a" is still a substring of "ba", but now we need to find "^a$" which will never happen unless "a" was the first and last character of the input string.

```
int checkStrings(string str, string substr, int start) {
  if (substr.size() == 0) return start;

  for (int i = start; i < str.size(); i++) {
    int count = 0, k = i;
    for (int j = 0; j < substr.size(); j++) {
      if (substr[j] == '?' || str[k] == substr[j]) {
        count++;
        if (count == substr.size()) return k;
        else k++;
      }
      else {
        break;
      }
    }
  }

  return -1;
}

int isMatch(const string &s, const string &p) {
  string str = "^" + s + "$";
  string ptr = "^" + p + "$";
  vector<string> pieces;
  int i = 0, j = 0;

  while (j <= ptr.size()) {
    if (j < ptr.size() && ptr[j] != '*') j++;
    else {
      if (j - i > 0) pieces.push_back(ptr.substr(i, j - i));
      j++;
      i = j;
    }
  }

  i = 0;
  j = 0;

  while (i < str.size() || j < pieces.size()) {
    int lastIndex = checkStrings(str, pieces[j], i);

    if (lastIndex > -1) i = lastIndex + 1;
    else return 0;

    j++;
  }

  return 1;
}

// int _isMatch(const string &s, const string &p, int i, int j, vector<vector<int>>
//   if (i < s.size() && j < p.size() && cache[i][j] != -1) {
//     return cache[i][j];
//   }

//   if (i == s.size() && j == p.size()) {
//     return 1;
//   }

//   int ans = 0;

//   if (j < p.size() && p[j] == '*') {
//     int b1 = 0, b2 = 0;
//     if (i < s.size()) b1 = _isMatch(s, p, i + 1, j, cache);
//     if (j < p.size()) b2 = _isMatch(s, p, i, j + 1, cache);
//     ans = b1 || b2;
//   }
//   else if (i < s.size() && (p[j] == '?' || s[i] == p[j])) {
//     int b = _isMatch(s, p, i + 1, j + 1, cache);
//     ans = b;
//   }

//   if (i < s.size() && j < p.size()) cache[i][j] = ans;

//   return ans;
// }

// int isMatch(const string &s, const string &p) {
//   vector<vector<int>> cache;

//   for (int i = 0; i < s.size(); i++) {
//     vector<int> temp;
//     for (int j = 0; j < p.size(); j++) {
```

```
 //       temp.push_back(-1);
 //     }
 //     cache.push_back(temp);
 //   }

 //   return _isMatch(s, p, 0, 0, cache);
 // }

 // int isMatch(const string &s, const string &p) {
 //   stack<pair<int, int>> stk;
 //   int i = 0, j = 0;

 //   while (i < s.size() || j < p.size()) {
 //     if (p[j] != '*') {
 //       if (i < s.size() && (p[j] == '?' || s[i] == p[j])) {
 //         i++;
 //         j++;
 //       }
 //       else if (stk.size() != 0) {
 //         pair<int, int> indices = stk.top();
 //         stk.pop();
 //         i = indices.first;
 //         j = indices.second;
 //       }
 //       else {
 //         return 0;
 //       }
 //     }
 //     else {
 //       j++;
 //       while (i < s.size()) {
 //         stk.push(make_pair(i, j));
 //         i++;
 //       }
 //     }
 //   }

 //   return 1;
 // }
```

Dp Rod Cutting Minimize Cost

There is a rod of length N lying on x-axis with its left end at x = 0 and right end at x = N. Now, there are M weak points on this rod denoted by positive integer values(all less than N) A1, A2, …, AM. You have to cut rod at all these weak points. You can perform these cuts in any order. After a cut, rod gets divided into two smaller sub-rods. Cost of making a cut is the length of the sub-rod in which you are making a cut.

Your aim is to minimise this cost. Return an array denoting the sequence in which you will make cuts. If two different sequences of cuts give same cost, return the lexicographically smallest.

Notes:
1. Sequence a1, a2 ,…, an is lexicographically smaller than b1, b2 ,…, bm, if and only if at the first i where ai and bi differ, ai < bi, or if no such i found, then n < m.
2. N can be upto 109.

Example:
N = 6
A = [1, 2, 5]

```
If we make cuts in order [1, 2, 5], let us see what total cost would be.
For first cut, the length of rod is 6.
For second cut, the length of sub-rod in which we are making cut is 5
(since we already have made a cut at 1).
For third cut, the length of sub-rod in which we are making cut is 4
(since we already have made a cut at 2).
So, total cost is 6 + 5 + 4.

Cut order           | Sum of cost
(lexicographically  | of each cut
 sorted)            |
____________________|______________
[1, 2, 5]           | 6 + 5 + 4 = 15
[1, 5, 2]           | 6 + 5 + 4 = 15
[2, 1, 5]           | 6 + 2 + 4 = 12
[2, 5, 1]           | 6 + 4 + 2 = 12
[5, 1, 2]           | 6 + 5 + 4 = 15
[5, 2, 1]           | 6 + 5 + 2 = 13

So, we return [2, 1, 5].
```

Time complexity: O(M^3), where M is the number of points we have to cut.
Space complexity: O(M^2).

Explanation: This is a dynamic programming problem and since the exercise asks for the lexicographically smallest solution it is better if we use memoization.
The main idea is that at each point "i" where we cut, will be created to rods with length 0 .. i and i .. N. For these two rods we can check their minimum costs independent from each other, in other words, the cuts that will be performed in rod 0 .. i don't depend from the cuts that will be performed in rod i .. N.
Notice, that you can never cut a rod with length <= 1. Also, we need to insert points 0 and N in our cuts array (rods can, of course, start at 0 and end at N).
So, we want our dp(i, j) to be the minimum cost for rod i .. j. When a rod is defined to be i .. j we are not interested in the points i and j because we are supposing that already exists a rod from i .. j (like in the beginning case where we have the rod that goes from 0 .. N, but we do not manipulate 0 and N because it doesn't make sense since we can't perform cuts in these points). And, if we have a rod from i .. j its cost will be the sum of minimum costs of inside rods, but if there is only on point we can't define a rod so we always need to add the cost of j - i. For example, we want to know the minimum cost for rod 2 .. 6, and we can perform according to our "cuts array" a cut at point 3, but point 3 alone doesn't make a rod, but we know that a cut at 3 will have cost 6 - 2 = 4.

Now, with the above facts we can define our recursion to be:
dp[l][r] = min(dp[l][k] + dp[k][r] + (cuts[r] - cuts[l])), for every cut point k inside this rod l .. r.

Our dp variable contains the minimum costs for rods i .. j, but we need to know here the cuts happened. For that, we store which index k in the above recursion gave us the minimum cost for rod i .. j. Since we are using memoization once we store the index k, we do not store any other k that is able to give a cost equal to the minimum cost and then we can guarantee the lexicographically smallest solution.

Notice that if the exercise was asking just for the minimum cost it would be possible to implement in O(M^2) time because we would need to calculate the minimium cost just for the rod 0 .. N (because we don't need to know where the cuts happend there is no need to calculate costs for intermediary rods i .. j, where i is greater than 0 so we just calculate a matrix with minimum costs for rods 0 .. i, with i being all the cut points, including N).
In this case dp[i][j] means the minimum cost for a rod starting at 0 and ending at i, with possible cuts being the range 0 .. j from the "cuts array".

Solution with the cut points based on:
http://qa.geeksforgeeks.org/4063/minimize-the-cutting-cost-latest-google-question

```
vector<vector<int>> dp;
vector<vector<int>> parent;

// Solve for dp(l, r).
int rec(int l, int r, vector<int> &b) {
  if (l + 1 >= r) return 0;

  if (dp[l][r] != -1) return dp[l][r];

  dp[l][r] = INT_MAX;
  int minCutIndex;

  for(int i = l + 1; i < r; i++) {
    // Recurrence.
    int possibleCost = rec(l, i, b) + rec(i, r, b) + (int)b[r] - (int)b[l];

    // Update index. Notice that we choose lexicographically smallest index
    // if multiple solutions give the same cost.
    if(possibleCost < dp[l][r]) {
      dp[l][r] = possibleCost;
      minCutIndex = i;
    }
  }

  // Store parent of (l, r).
  parent[l][r] = minCutIndex;

  return dp[l][r];
}

// Function for building solution.
void buildSolution(int l, int r, vector<int> &b, vector<int> &ans){
  if(l + 1 >= r) return;

  // First choose parent of (l, r).
  ans.push_back(b[parent[l][r]]);

  // Call recursively for two new segments. Calling left segment first
  // because we want lexicographically smallest.
  buildSolution(l, parent[l][r], b, ans);
  buildSolution(parent[l][r], r, b, ans);
}

vector<int> solve(int a, vector<int> &b) {
  vector<int> ans;

  // Insert A and 0.
  b.push_back(a);
  b.insert(b.begin(), 0);
  int n = b.size();

  // (Re)Initialise dp array.
  dp.resize(n);
  parent.resize(n);
  for (int i = 0; i < n; i++){
    dp[i].resize(n);
    parent[i].resize(n);
    for (int j = 0; j < n; j++) dp[i][j] = -1;
  }

  int best = rec(0, n - 1, b);
  buildSolution(0, n - 1, b, ans);

  return ans;
}

// If we just wanted the minimum cost a O(M^2) solution is possible.
// int minCost(int a, vector<int> &b) {
```

```
//   int n = b.size() + 1;
//   vector<vector<int>> dp(n, vector<int>(n, 0));

//   vector<int> m = b;
//   m.push_back(a);

//   for (int i = 1; i < n; i++) dp[i][0] = m[i];

//   for (int i = 1; i < n; i++) {
//     for (int j = 1; j < i; j++) {
//       dp[i][j] = min(dp[i][j - 1] + (m[i] - m[j - 1]), m[i] + dp[j][j - 1]);
//     }
//   }

//   return dp[n - 1][n - 2];
// }
```

Dp Stairs Climb

You are climbing a stair case. 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?

Example:
Given 3
Return 3. Because the steps would be: [1 1 1], [1 2], [2 1].

Time complexity: O(N), where N is the number of steps.
Space complexity: O(1) if optimized since this is just Fibonacci in the end.

Explanation: each time you can do two things: climb 1 step or climb two. So,
you need to check the two possibilities. The number of ways for a given N, is
the number of ways for N - 1 plus the number of ways for N - 2. Since, we
can take 1 or two steps we have the initial conditions:
N = 0 => 1
N = 1 => 1
And because of that our answer is just a Fibonacci number.

For example,
```
indices: 0 1 2 3
answer:  1 1 2 3
```

answer[3] = answer[1] + answer[2].

```
int climbStairs(int n) {
  vector<int> ans(n + 1, 0);

  ans[0] = 1;
  ans[1] = 1;

  // The answer will always be a Fibonacci number (1, 1, 2, 3, 5, 8, ...). So,
  // you can optimize the space complexity to O(1).
  for (int i = 2; i < ans.size(); i++) {
    ans[i] = ans[i - 1] + ans[i - 2];
  }

  return ans[ans.size() - 1];
}
```

Dp Unique Path Grid

Given a grid of size m * n, lets assume you are starting at (1,1) and your goal is to reach (m,n). At any instance, if you are on (x,y), you can either go to (x, y + 1) or (x + 1, y).

Now consider if some obstacles are added to the grids. How many unique paths would there be? An obstacle and empty space is marked as 1 and 0 respectively in the grid.

Example:
There is one obstacle in the middle of a 3x3 grid as illustrated below.

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

The total number of unique paths is 2.

Note: m and n will be at most 100.

Time complexity: O(M * N), where M is the number of rows and N is the number of columns.
Space complexity: O(M * N).

Explanation: this is a famous dp problem. We initialize our dp[M][N] variable with zeros and we start to iterate the grid from the bottom right element. When we are at an element that is not an obstacle we add dp[i + 1][j] + dp[i][j + 1]. We return dp[0][0].

```
int uniquePathsWithObstacles(vector<vector<int> > &a) {
  if (a.size() == 0 || a[0].size() == 0) return 0;

  int m = a.size(), n = a[0].size();
  vector<vector<int>> dp;

  for (int i = 0; i < m; i++) {
    vector<int> t(n, 0);
    dp.push_back(t);
  }

  dp[m - 1][n - 1] = !a[m - 1][n - 1];

  for (int i = m - 1; i >= 0; i--) {
    for (int j = n - 1; j >= 0; j--) {
      if (a[i][j] == 0) {
        int r = 0, b = 0;
        if (i + 1 < m) b = dp[i + 1][j];
        if (j + 1 < n) r = dp[i][j + 1];
        if (r != 0 || b!= 0) dp[i][j] = r + b;
      }
    }
  }

  return dp[0][0];
}
```

Dp Ways To Decode

A message containing letters from A-Z is being encoded to numbers using the following mapping:

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

Given an encoded message containing digits, determine the total number of ways to decode it.

Example:
Given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12).
The number of ways decoding "12" is 2.

Time complexity: O(N), where N is the length of the message.
Space complexity: O(1).

Explanation: this problem is kinda like Fibonacci. For each character of your encoded string you have two options: or you use this character alone and check it against your map (that would give you numbers from 1-9), or you get this character plus its neighbor (that would give you numbers from 10-26). So, we start from the end of our array and for each character/number of our encoded string we get the number of ways the substring after this character can be decoded plus the number of ways the substring after this character and its neighbor can be decoded. The answer would be a Fibonacci number of it wasn't the fact that at each time we are checking our character and its number we need to be sure the number is between 1 and 26, otherwise if it is zero we have zero ways to decode that substring and if it is larger than 26 we don't add the number of ways the substring after the current character and its neightbor can be decoded, but instead just the number of ways the substring after this current character can be decoded.

```
int numDecodings(string a) {
  if (a.size() == 0 || (a.size() == 1 && a[0] == '0')) return 0;
  if (a.size() == 1) return 1;

  vector<int> ans(a.length() + 1, 0);

  ans[0] = 1;

  for (int i = a.size() - 1, j = 1; i >= 0; i--, j++) {
    if (stoi(a.substr(i, 1)) >= 1) {
      if (i + 1 < a.size() && stoi(a.substr(i, 2)) <= 26) {
        ans[j] = ans[j - 1] + ans[j - 2];
      }
      else {
        ans[j] = ans[j - 1];
      }
    }
  }

  return ans[ans.size() - 1];
}

/**
 * This version uses constant space.
 */
int numDecodings2(string a) {
  if (a.size() == 0 || (a.size() == 1 && a[0] == '0')) return 0;
  if (a.size() == 1) return 1;

  int secondToLast = 1, thirdToLast, ans = 0;

  for (int i = a.size() - 1; i >= 0; i--) {
    if (stoi(a.substr(i, 1)) >= 1) {
      if (i + 1 < a.size() && stoi(a.substr(i, 2)) <= 26) {
        ans = secondToLast + thirdToLast;
      }
      else {
        ans = secondToLast;
      }
    }
    else {
      ans = 0;
    }

    thirdToLast = secondToLast;
    secondToLast = ans;
  }

  return ans;
}
```

Dp Word Break

```
Given a string s and a dictionary of words dict, determine if s can be
segmented into a space-separated sequence of one or more dictionary words.
Return 0 / 1 ( 0 for false, 1 for true ) for this problem
```

Example:

```
s = "myinterviewtrainer",
dict = ["trainer", "my", "interview"].
Return 1 ( corresponding to true ) because "myinterviewtrainer" can be
segmented as "my interview trainer".
```

Time complexity: O(N^3), where N is the length of the string.
Space complexity: probably O(N^3)

Explanation: this problem is simpler than word-break-II, we just need a vector dp because we are not interested in all the solutions, but if it is possible to have a valid sentence. Traverse the string from the end to the beginning with a variable i. At each position i, try all the possible words from i till the end of the string. When you find a match for one of these words in your dictionary make dp[i] = dp[j] (j is the index where the word ends + 1). Then if dp[i] became true stop your inner loop (the one looking for possible words that start at i) because what matters it that at i we have a valid sentence. Return dp[0].

Check the solution for word-break-II if you want more details.

```
int wordBreak(string a, vector<string> &b) {
  unordered_set<string> dict(b.begin(), b.end());
  vector<bool> dp(a.length() + 1, false);

  // initialize the valid values
  dp[a.length()] = 1;

  // generate solutions from the end
  for(int i = a.length() - 1; i >= 0; i--) {
    // Try all possible words from i till the end of the string, and as soon
    // as we find a valid combination break the loop.
    for(int j = i + 1; j <= a.length() && dp[i] == false; j++) {
      string word = a.substr(i, j - i);
      if (dict.find(word) != dict.end()) dp[i] = dp[j];
    }
  }

  return dp[0];
}
```

Dp Word Break-2

```
Given a string s and a dictionary of words dict, add spaces in s to construct
a sentence where each word is a valid dictionary word.
Return all such possible sentences.

Examples:
s = "catsanddog",
dict = ["cat", "cats", "and", "sand", "dog"].
A solution is ["cats and dog", "cat sand dog"]
```

Time complexity: O(N^3), where N is the length of the string.
Space complexity: probably O(N^3)

Explanation: first of all this is dynamic programming so we gonna have a variable vector>dp. Traverse the string from the end to the beginning with a variable i. At each position i, try all the possible words from i till the end of the string. When you find a match for one of these words in your dictionary combine this matched word with all the strings you were able to assemble at the position where this matched word ends + 1 using your dp variable. Store the result of these combinations in your dp variable at position i.

For example, suppose you have:
a: "dogcatsand"
dict: "dog", "cat", "cats", "and", "sand"

And at some point of our running we have:
Matched word "dog", starts at position i = 0 and ends at position 2.
In our dp we have at position 3 ["cat sand", "cats and"].
So we combine "dog" with our dp at position 3 getting
["dog cat sand", "dog cats and"]

```cpp
vector<string> wordBreak(string a, vector<string> &b) {
  unordered_set<string> dict(b.begin(), b.end());
  vector<vector<string>> dp(a.length() + 1, vector<string>(0));

  // initialize the valid values
  dp[a.length()].push_back("");

  // generate solutions from the end
  for(int i = a.length() - 1; i >= 0; i--) {
    vector<string> t;

    // Try all possible words from i till the end of the string
    for(int j = i + 1; j <= a.length(); j++) {
      string word = a.substr(i, j - i);

      if (dict.find(word) != dict.end()) {
        // If we find a word at position j - 1 we need to combine it with all
        // the strings that we were able to assemble at position j
        for (int k = 0; k < dp[j].size(); k++) {
          t.push_back(word + (dp[j][k].empty() ? "" : " ") + dp[j][k]);
        }
      }
    }

    dp[i] = t;
  }

  return dp[0];
}

// bool findWord(vector<string> &b, string a, int s, int e) {
//   for (int i = 0; i < b.size(); i++) {
```

```
//     int j = 0, k = s;

//     while (j < b[i].size() && k < e) {
//       if (b[i][j] != a[k]) break;
//       j++;
//       k++;
//     }

//     if (j == b[i].size() && k == e) return true;
//   }

//   return false;
// }

// vector<string> wordBreak(string a, vector<string> &b) {
//   if (a.size() == 0 || b.size() == 0) return vector<string>();

//   vector<string> ans;
//   vector<vector<int>> dp;
//   dp.push_back(vector<int>(1, a.size()));

//   for (int i = a.size() - 1; i >= 0; i--) {
//     vector<vector<int>> nextDp;

//     for (int j = 0; j < dp.size(); j++) {
//       vector<int> t;

//       if (dp[j].back() == 0) nextDp.push_back(dp[j]);

//       for (int k = dp[j].back() - 1; k >= 0; k--) {
//         if (findWord(b, a, k, dp[j].back())) t.push_back(k);
//       }

//       for (int k = 0; k < t.size(); k++) {
//         vector<int> t2 = dp[j];
//         t2.push_back(t[k]);
//         nextDp.push_back(t2);
//       }
//     }

//     if (nextDp.size()) dp.swap(nextDp);
//   }

//   for (int i = 0; i < dp.size(); i++) {
//     string t = "";
//     if (dp[i].back() == 0) {
//       for (int j = 1; j < dp[i].size(); j++) {
//         if (j > 1) t = a.substr(dp[i][j], dp[i][j - 1] - dp[i][j]) + " " + t;
//         else t = a.substr(dp[i][j], dp[i][j - 1] - dp[i][j]);
//       }
//       ans.push_back(t);
//     }
//   }

//   sort(ans.begin(), ans.end());

//   return ans;
// }
```

Graph Bellman Ford

Single Source Shortest Path Bellman-Ford algorithm.

Time complexity: O(|V| * |E|), where |V| is the number of vertices and |E| is

the number of edges.
Space complexity: O(|V|).

Explanation: similar to Dijkstra Algorithm we start our distance array with 0 distance from the source to itself and everything else is Infinity. Then, we visit all edges |V| - 1 times and update the distance array with dist[v] = min(dist[v], dist[u] + weight(u, v)).
Bellman-Ford has two advantages over Dijkstra (though it is slower):
1. It can calculate shortest paths in graphs with negative weights.
2. It can detect negative cycles.
The algorithm is executed |V| - 1 times, that is the maximum number of edges between a vertex and another one (assuming there is no self loops).
Lets see why we need to execute |V| - 1 times. Suppose you have the graph:

```
    2      4       1
0 ----> 1 ----> 2 ----> 3
```

And the order we gonna visit the edges is: 2->3, 1->2, 0->1. At the beginning our distance array is dist = [0, INF, INF, INF]. So, when we check the edge 2->3 we will not update anything because we didn't discover yet a path to 2. So, in the first iteration we just update the edge 0->1 and we get dist = [0, 2, INF, INF]. Continuing the process you can see that you will need more 2 iterations, that is the total is |V| - 1 = 3, to get the shortest paths.

```
struct Edge {
  int from, to, weight;
  Edge(int x, int y, int z): from(x), to(y), weight(z) {}
};

vector<int> bellmanFord(vector<Edge> &g, int src, int numVertices) {
  vector<int> parents(numVertices, -1);
  vector<int> dist(numVertices, INT_MAX);

  dist[src] = 0;

  for (int i = 0; i < numVertices - 1; i++) {
    for (int j = 0; j < g.size(); j++) {
      int u = g[j].from, v = g[j].to, w = g[j].weight;
      if (dist[u] != INT_MAX && dist[u] + w < dist[v]) {
        dist[v] = dist[u] + w;
        parents[v] = u;
      }
    }
  }

  for (int j = 0; j < g.size(); j++) {
    int u = g[j].from, v = g[j].to, w = g[j].weight;
    if (dist[u] != INT_MAX && dist[u] + w < dist[v]) {
      cout << "Negative cycle found." << endl;
      return vector<int>(numVertices, -1);
    }
  }

  return parents;
}
```

Graph Breadth First Search

Breadth First Search for a graph represented as Adjacency List.

Time complexity: O(|V| + |E|), where |V| is the number of vertices and |E| is

the number of edges.
Space complexity: O(|V|).

Explanation: use a queue. Breadth First Search can use much more memory than Depth First Search (in trees), but BFS can give the optimal solution when you are looking if a vertex has a path to another and you want the shortest path (and the graph is unweighted), while DFS, though it also can find a path it is not guaranteed it will be the shortest.

```
struct Edge {
  int to;
  Edge(int x): to(x) {}
};

vector<int> breadthFirstSearch(vector<vector<Edge>> &g, int v) {
  int numVertices = g.size();
  vector<int> ans;
  vector<bool> visited(numVertices, false);
  queue<int> q;

  q.push(v);
  visited[v] = true;

  while (!q.empty()) {
    int first = q.front();
    q.pop();
    ans.push_back(first);
    for (int i = 0; i < g[first].size(); i++) {
      int u = g[first][i].to;
      if (!visited[u]) {
        // It is better to mark as visited here to avoid unnecessary work,
        // though I believe the answer would be the same if we mark the node
        // as visited in the beginning of the while loop. Marking it here we
        // don't add the same vertex multiple times to the queue saving memory
        // and increasing performance.
        visited[u] = true;
        q.push(u);
      }
    }
  }

  return ans;
}
```

Graph Capture Regions Board

Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'. A region is captured by flipping all 'O's into 'X's in that surrounded region.

Note:
1. The board can have different x and y dimensions.

Example:

```
X X X X
X O O X
X X O X
X O X X
```

After running your function, the board should be:

```
X X X X
X X X X
X X X X
X O X X
```

Time complexity: O(N^2), where N is the number of places in the board.
Space complexity: O(N).

Explanation: use BFS or DFS to find connected components of O's. When performing the BFS keep all the vertices you visited because you may need to switch them back to its original value when the connected component you traversed is not completely surrounded by X's. A connected component will not be completely surrounded when one of its O's is in one of the borders of the board (i = 0, j = 0, i = N - 1, j = N - 1). Also notice that if you switch a not valid connected component back to O's in the BFS function you will end up investigating the same component more than once. For example,

```
O O X
X X X
X X X
```

If you don't mark in some way that the vertex (0, 1) was already visited you will cal BFS for it again. For that you can use a intermediary symbol or a "visited" variable. I used an intermediary symbol 'F' to save some extra memory, so in the end of my algorithm I need to switch back the F's to O's.

```cpp
void bfs(vector<vector<char>> &a, int i, int j) {
  int n = a.size(), m = a[0].size();
  bool valid = true;
  vector<pair<int, int>> visited;

  queue<pair<int, int>> q;
  q.push({i, j});
  a[i][j] = 'X';

  while (!q.empty()) {
    pair<int, int> curr = q.front();
    q.pop();
    int r = curr.first, c = curr.second;

    if (r == 0 || r == n - 1 || c == 0 || c == m - 1) valid = false;

    if (r - 1 >= 0 && a[r - 1][c] == 'O') {
      q.push({r - 1, c});
      a[r - 1][c] = 'X';
    }
    if (c - 1 >= 0 && a[r][c - 1] == 'O') {
      q.push({r, c - 1});
      a[r][c - 1] = 'X';
    }
    if (r + 1 < n && a[r + 1][c] == 'O') {
      q.push({r + 1, c});
      a[r + 1][c] = 'X';
    }
    if (c + 1 < m && a[r][c + 1] == 'O') {
      q.push({r, c + 1});
      a[r][c + 1] = 'X';
    }

    visited.push_back({r, c});
  }

  if (!valid) {
    for (pair<int, int> p : visited) a[p.first][p.second] = 'F';
  }
}

void solve(vector<vector<char>> &a) {
  if (a.size() == 0 || a[0].size() == 0) return;
  int n = a.size(), m = a[0].size();

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (a[i][j] == 'X' || a[i][j] == 'F') continue;
      bfs(a, i, j);
    }
  }

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (a[i][j] == 'F') a[i][j] = 'O';
    }
  }
}
```

Graph Clone Undirected Graph

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors. The graph node is defined as:

```
struct UndirectedGraphNode {
 int label;
 vector neighbors;
 UndirectedGraphNode(int x) : label(x) {}
};
```

Time complexity: O(|V| + |E|), where |V| is the number of vertices and |E| is the number of edges.
Space complexity: O(|V|).

Explanation: you need to perform BFS. The trick here is that you need to keep a map of the nodes you are creating because you will need to retrieve them eventually when checking the neighbors of the vertices. For example, given the graph:

```
1---2
|   |
4---3
```

Suppose you start from vertex 1, you will need to create vertex 2 and 4. Later, when checking vertex 2 you will need to retrieve it, and when checking its neighbors you will need to retrieve vertices 1 and 4. So, instead of using a visited variable in your BFS use a map.
You start inserting in this map the source node, after for each vertex that your current vertex is connected you check if it exists in the map, if not, you create it and add to the map.
Notice, the key must be the address of the node and not its label, because the exercise didn't say the label would be unique, but we know the address is.

```
struct UndirectedGraphNode {
  int label;
  vector<UndirectedGraphNode *> neighbors;
  UndirectedGraphNode(int x) : label(x) {}
};

UndirectedGraphNode* cloneGraph(UndirectedGraphNode *src) {
  unordered_map<UndirectedGraphNode *, UndirectedGraphNode *> nodeMap;
  queue<UndirectedGraphNode *> q;

  q.push(src);
  nodeMap[src] = new UndirectedGraphNode(src->label);
  while (!q.empty()) {
    UndirectedGraphNode *u = q.front();
    q.pop();

    for (UndirectedGraphNode *v : (u->neighbors)) {
      if (nodeMap.find(v) == nodeMap.end()) {
        nodeMap[v] = new UndirectedGraphNode(v->label);
        q.push(v);
      }

      (nodeMap[u]->neighbors).push_back(nodeMap[v]);
    }
  }

  return nodeMap[src];
}
```

Graph Commutable Islands

There are n islands and there are many bridges connecting them. Each bridge has some cost attached to it. We need to find bridges with minimal cost such that all islands are connected.
It is guaranteed that input data will contain at least one possible scenario in which all islands are connected with each other.

Time complexity: O(|E|log|E|), where |E| is the number of edges and this complexity comes from the sorting.
Space complexity: O(|E| + |V|), where |V| is the number of vertices.

Explanation: apply Kruskal's algorithm.

```
/* --- Disjoint Sets --- */
struct Node {
  int rank, val;
  Node *parent;
  Node(int x, int y, Node *z) : rank(x), val(y), parent(z) {}
};

unordered_map<int, Node*> nodeMap;

Node *makeSet(int val) {
  Node *n = new Node(0, val, NULL);
  n->parent = n;
  nodeMap[val] = n;
  return n;
}

Node *findSetUtil(Node *n) {
  if (n == n->parent) return n;
  n->parent = findSetUtil(n->parent);
  return n->parent;
}

Node *findSet(int val) {
  if (nodeMap.find(val) != nodeMap.end()) {
    return findSetUtil(nodeMap[val]);
  }
  return NULL;
}

void join(int val1, int val2) {
  Node *n1 = findSet(val1);
  Node *n2 = findSet(val2);

  if (n1 == n2 || !n1 || !n2) return;

  if (n1->rank >= n2->rank) {
    if (n1->rank == n2->rank) (n1->rank)++;
    n2->parent = n1;
  }
  else {
    n1->parent = n2;
  }
}

/* --- Kruskal's Algorithm --- */
struct Edge {
  int from, to, weight;
  Edge(int x, int y, int z) : from(x), to(y), weight(z) {}
};

bool cmp(Edge ed1, Edge ed2) {
  return ed1.weight <= ed2.weight;
}

vector<Edge> kruskal(vector<Edge> &edges, vector<int> &vertices) {
  vector<Edge> ans;

  sort(edges.begin(), edges.end(), cmp);

  for (int v : vertices) makeSet(vertices[v]);

  for (Edge ed : edges) {
    Node *n1 = findSet(ed.from);
    Node *n2 = findSet(ed.to);

    if (n1 != n2) {
      join(ed.from, ed.to);
      ans.push_back(ed);
    }
  }

  return ans;
}
```

Graph Connected Black Shapes

```
Given N * M field of O's and X's, where O = white, X = black. Return the
number of black shapes. A black shape consists of one or more adjacent X's
(diagonals not included).

Notes:
1. We are looking for connected shapes.
```

Example:

```
OOOXOOO
OOXXOXO
OXOOOXO

Answer is 3 shapes are:
(i)    X
     X X
(ii)
      X
```

```
(iii)
     X
     X
```

Time complexity: O(N * M).

Explanation: At each time we find a X we increment our count and then we apply BFS or DFS to eliminate the connected component.

```
void bfs(vector<string> &a, int i, int j) {
  int n = a.size(), m = a[0].size();
  queue<pair<int, int>> q;
  q.push({i, j});
  a[i][j] = 'O';

  while (!q.empty()) {
    pair<int, int> curr = q.front();
    q.pop();

    int r = curr.first, c = curr.second;

    if (r + 1 < n && a[r + 1][c] == 'X') {
      q.push({r + 1, c});
      a[r + 1][c] = 'O';
    }
    if (c + 1 < m && a[r][c + 1] == 'X') {
      q.push({r, c + 1});
      a[r][c + 1] = 'O';
    }
    if (r - 1 >= 0 && a[r - 1][c] == 'X') {
      q.push({r - 1, c});
      a[r - 1][c] = 'O';
    }
    if (c - 1 >= 0 && a[r][c - 1] == 'X') {
      q.push({r, c - 1});
      a[r][c - 1] = 'O';
    }
  }
}

int black(vector<string> &a) {
  if (a.size() == 0 || a[0].size() == 0) return 0;

  int n = a.size(), m = a[0].size(), ans = 0;

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (a[i][j] == 'O') continue;
      bfs(a, i, j);
      ans++;
    }
  }

  return ans;
}
```

Graph Depth First Search

Depth First Search for a graph represented as Adjacency List.

Time complexity: O(|V|), where |V| is the number of ans.

Space complexity: O(log|V|)

Explanation: use a stack. The easiest ways is to use recursion.

```
struct Edge {
  int to;
  Edge(int x): to(x) {}
};

void dfsUtil(
  vector<vector<Edge>> &g, int v, vector<bool> &visited, vector<int> &ans
) {
  visited[v] = true;
  ans.push_back(v);

  for (int i = 0; i < g[v].size(); i++) {
    int u = g[v][i].to;
    if (!visited[u]) dfsUtil(g, u, visited, ans);
  }
}

vector<int> depthFirstSearch(vector<vector<Edge>> &g, int v) {
  int numVertices = g.size();
  vector<int> ans;
  vector<bool> visited(numVertices, false);

  dfsUtil(g, v, visited, ans);

  return ans;
}
```

Graph Dijkstra

Dijkstra Single Source Shortest Path algorithm.

Time complexity: O(|E| * log|V|), where |E| is the number of edges and |V| is the number of vertices. This complexity assume you are using a priority queue.
Space complexity: O(|V|).

Explanation: Dijkstra algorithm calculate the distance from a given vertex to all the other vertices in the graph. It begins assuming that the distance from this source vertex to every other node is Infinity (except for the source node itself where the distance is 0). After, using a priority queue (this queue is initialized with {source, 0}) the algorithm looks for the shortest distance until that moment, get the vertex where we have have this shortest distance, mark it as visited and check all of its edges updating distances and pushing these distances to the priority queue. Since Dijkstra algorithm always get from the priority queue the shortest distance to a vertex it can mark this vertex as visited and never check it again.
For example,
If we 5 vertices and a distance vector like this:

```
indices: 0 1 2 3 4
dist:    5 4 3 2 1
```

It will get the vertex 4 and mark it as visited because if we try to reach vertex 4 with any other path we will have a larger distance since the values 2, 3, 4, and 5 are all larger than 1 already (assuming that we have only positive weights). When the priority queue is empty the algorithm is done.

```
struct Edge {
  int to, weight;
```

```
    Edge(int x, int y) : to(x), weight(y) {}
  };

  // SSSP Dijkstra algorithm for a graph represented with Adjacency List.
  // Time complexity: O(|E| * log|V|).
  // Space complexity: O(|V|).
  vector<int> dijkstra(vector<vector<Edge>> &g, int src) {
    int numVertices = g.size();
    // dist[i] keeps the shortest distance from src to a vertice i.
    vector<int> dist(numVertices, INT_MAX);
    // parents[i] keeps the node we must come from to get to i with the shortest
    // distance.
    vector<int> parents(numVertices, -1);
    // The priority queue. It is better to use a set than the priority_queue
    // from STL because we can delete nodes from set. In the priority_queue
    // we would need to repeat a vertex every time we find a shortest distance.
    // A pair is made as {vertex, distance}.
    set<pair<int, int>> pq;

    // The source has a distance of 0 to itself.
    dist[src] = 0;
    pq.insert({src, 0});

    while (!pq.empty()) {
      // Get the vertex with the shortest distance until now.
      int from = pq.begin()->first;
      // Erase it from the queue marking it as visited. erase(iterator) has
      // time complexity O(1).
      pq.erase(pq.begin());

      // If our queue gave us a node with Infinity weight it means we don't have
      // anymore nodes that we can get to from source.
      if (dist[from] == INT_MAX) break;

      // For each vertex that we can get from vertex "from".
      for (Edge ed : g[from]) {
        // If to get from node u to v has a shortest distance than we had before
        // update our variables.
        if (dist[ed.to] > dist[from] + ed.weight) {
          // Erase the previous pair {vertex, distance} from our queue so we
          // don't have useless entries in our queue. erase(value) has time
          // complexity O(logN).
          pq.erase({ed.to, dist[ed.to]});
          dist[ed.to] = dist[from] + ed.weight;
          pq.insert({ed.to, dist[ed.to]});

          // Update from where we came from to get to this current node with
          // a shortest distance.
          parents[ed.to] = from;
        }
      }
    }

    return parents;
  }

  int findMinIndex(vector<int> &dist, vector<bool> visited) {
    int minIndex = -1;

    for (int i = 0; i < dist.size(); i++) {
      if (minIndex == -1 && !visited[i]) minIndex = i;
      if (!visited[i] && dist[i] < dist[minIndex]) minIndex = i;
    }

    return minIndex;
  }

  // SSSP Dijkstra algorithm for a graph represented with Adjacency Matrix and
  // not using a priority queue.
```

```
// Time complexity: O(|V|^2).
// Space complexity: O(|V|).
vector<int> dijkstraSlow(vector<vector<int>> &g, int src) {
  int numVertices = g.size();
  vector<int> dist(numVertices, INT_MAX);
  vector<int> parents(numVertices, -1);
  vector<bool> visited(numVertices, false);

  dist[src] = 0;

  for (int i = 0; i < numVertices - 1; i++) {
    int u = findMinIndex(dist, visited);
    visited[u] = true;

    for (int v = 0; v < numVertices; v++) {
      if (g[u][v] == 0 || dist[u] == INT_MAX) continue;

      if (!visited[v] && dist[u] + g[u][v] < dist[v]) {
        dist[v] = dist[u] + g[u][v];
        parents[v] = u;
      }
    }
  }

  return parents;
}
```

Graph Disjoint Sets

```
Disjoint Sets With Path Compression.
```

Explanation: disjoint sets are normally used in Kruskal's Algorithm and to find cycles in a graph. It consists of of Abstract Data Type with 3 operations:
1. makeSet(val): make a set from a single element.
2. findSet(val): returns the representative element from a set.
3. union(val1, val2): join the sets that val1 and val2 belongs.

This is normally achieved with a tree with variable number of children and the basic structure of a Node in a set is:

```
Node {
  int rank;     // rank here means the depth of the tree and it is important
                // just for the root node.
  int val;      // the data of the node.
  Node *parent; // a pointer to the parent node so we can get to the root
                // node
}
```

The path compression is performed when findSet() is called and what we want to achive is that every child node points directly to the representative node of the set, in other words, we want one parent whith a lot of direct children. For example:
If we have:

```
      1
     /
    2
   /
  3
```

We want to have instead:

```
      1
     / \
    2   3
```

Where 1 is the representative node.

Notice that to get a node in O(1) time when the user call one of our 3 operations in O(1) time we use a hashmap, but to get the root node (representative node) it will not necessarily be constant time if path compression was not performed yet for that node because in this case we gonna use recursion with the parent node to get to the root.

Notice too that a rank (the depth of the tree) just grows if both of the settings we are joining have the same rank. In this case we increment rank by 1. If one of the ranks is larger than the other we just use the larger one, so the rank will be the same. For example:

```
     1              4
    /      and       \
   2                  5
  /
 3
```

Rank for the left tree is 2 and for the right tree is 1 so we get:

```
     1
    / \
   2   4
  /     \
 3       5
```

And the larger rank is still 2.

```
struct Node {
  int rank;
  int val;
  Node *parent;
  Node(int x, int y, Node *z) : rank(x), val(y), parent(z) {}
};

unordered_map<int, Node*> nodeMap;

Node *makeSet(int val) {
  Node *n = new Node(0, val, NULL);
  n->parent = n;
  nodeMap[val] = n;

  return n;
}

Node *findSetUtil(Node *n) {
  if (n == n->parent) return n;
  n->parent = findSetUtil(n->parent);
  return n->parent;
}

Node *findSet(int val) {
  if (nodeMap.find(val) != nodeMap.end()) {
    return findSetUtil(nodeMap[val]);
  }
  return NULL;
}

void join(int val1, int val2) {
  Node *n1 = findSet(val1);
  Node *n2 = findSet(val2);

  if (n1 == n2 || !n1 || !n2) return;

  if (n1->rank >= n2->rank) {
    if (n1->rank == n2->rank) (n1->rank)++;
    n2->parent = n1;
  }
  else {
    n1->parent = n2;
  }
}
```

Graph Fibonnaci Numbers

What is the minimum amount of Fibonacci numbers required so their sum is equal to a given Number N? N is a positive number.

Note: repetition of number is allowed.

Example:

```
N = 4
Fibonacci numbers : 1 1 2 3 5 .... so on
here 2 + 2 = 4, so minimum numbers will be 2
```

Time complexity: O(NlogN), where N is is the given number and logN comes from the fact that there are logN Fibonacci numbers smaller than N. This complexity is for the greedy approach.
Space complexity: O(logN).

Explanation: "shortest path", "dynamic programming coin changing", "greedy", are all possible approaches for this problem.

This exercise is based on Zeckendorf's Theorem that states that every positive integer larger than 2 can be represented as the sum of non-neighbor Fibonacci numbers. From this theorem we can deduce that if we get the largest Fibonacci number smaller or equal to N (call it fib(i)), then N - fib(i) will have its own representation as a sum of Fibonacci numbers. As we are always getting the largest possible ones, the amount of numbers will be minimum. Also, notice that we will never use Fibonacci neighbors because if we use fib(i) and fib(i - 1) we could use instead fib(i + 1), so actually fib(i + 1) would be the largest number smaller or equal to N.

For the shortest path approach you need to build a graph where the vertices are all the numbers 0 .. N. The edges will all have weight 1 and they will connect a number u to all the possible numbers v that you can get subtracting from u a Fibonacci number. After, you get the shortest path from N to 0. The time and space complexity is this case would be O(N^2)

```
#define INF 999999

vector<int> fib(int n) {
  vector<int> ans;
  int f1 = 1, f2 = 1;
  ans.push_back(1);

  while (f1 + f2 <= n) {
    ans.push_back(f1 + f2);
    int temp = f1;
    f1 = f2;
    f2 = temp + f2;
  }

  return ans;
}

int getLargestFib(vector<int> &f, int n) {
  int prev = 0;

  for (int i = 0; i < f.size(); i++) {
    if (f[i] > n) return f[prev];
    else prev = i;
  }

  return f[prev];
```

```
}

int fibsum(int n) {
  if (n == 1 || n == 2) return 1;

  vector<int> f = fib(n);
  int ans = 0;

  while (n > 0) {
    int larger = getLargestFib(f, n);
    n = n - larger;
    ans++;
  }

  return ans;
}

/* --- Shortest Path approach --- */
// vector<int> fib(int n) {
//   vector<int> ans;
//   if (n == 0) return ans;

//   int f1 = 1, f2 = 1;
//   ans.push_back(1);

//   while (f1 + f2 <= n) {
//     ans.push_back(f1 + f2);
//     int temp = f1;
//     f1 = f2;
//     f2 = temp + f2;
//   }

//   return ans;
// }

// int getLargestFibIndex(vector<int> &f, int n) {
//   int prev = -1;

//   for (int i = 0; i < f.size(); i++) {
//     if (f[i] > n) return prev;
//     else prev = i;
//   }

//   return prev;
// }

// int shortestPath(vector<vector<int>> &g, int src, int dest) {
//   int n = g.size();
//   queue<int> q;
//   vector<bool>visited(n, false);
//   vector<int> dist(n, INF);

//   q.push(src);
//   visited[src] = true;
//   dist[src] = 0;

//   while (!q.empty()) {
//     int curr = q.front();
//     q.pop();

//     if (curr == dest) return dist[dest];

//     for (int i = 0; i < g[curr].size(); i++) {
//       int v = g[curr][i];
//       if (!visited[v]) {
//         dist[v] = min(dist[v], dist[curr] + 1);
//         visited[v] = true;
//         q.push(v);
//       }
//     }
//   }

//   return dist[dest];
// }

// int minFibNumbers(int n) {
//   vector<vector<int>> g(n + 1, vector<int>());
//   vector<int> f = fib(n);

//   for (int i = 1; i <= n; i++) {
//     for (int j = getLargestFibIndex(f, i); j >= 0; j--) {
//       g[i].push_back(i - f[j]);
//     }
//   }

//   return shortestPath(g, n, 0);
// }
```

Graph Floyd Warshall

Floyd Warshall All-Pairs Shortest Path.

Time complexity: O(|V|^3), where |V| is the number of vertices.

Explanation: the main idea is to check if going from vertex i to j is shorter if we use an intermediary vertex k, intead of the direct path i to j. So, we want dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]). Notice that this algorithm as it is work for Adjacency Matrices and that an edge that does not exist must to be represented as Infinity (and take care because the above equation can overflow if you use INT_MAX).

```
vector<vector<int>> floydWarshall(vector<vector<int>> &g) {
  int numVertices = g.size();
  vector<vector<int>> dist(numVertices, vector<int>(numVertices, 0));

  // Initialize dist matrix. This is unnecessary if the input graph is a matrix
  // where a edge that does not exist is represented with Infinity weight and
  // the distance from a vertex to itself is 0.
  for (int i = 0; i < numVertices; i++) {
    for (int j = 0; j < numVertices; j++) {
      if (i == j) continue;
      dist[i][j] = g[i][j] == 0 ? INT_MAX : g[i][j];
    }
  }

  for (int k = 0; k < numVertices; k++) {
    for (int i = 0; i < numVertices; i++) {
      for (int j = 0; j < numVertices; j++) {
        // Avoid overflow.
        if (dist[i][k] == INT_MAX || dist[k][j] == INT_MAX) continue;
        dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
      }
    }
  }

  return dist;
}
```

Graph Kruskal

Kruskal's algorithm for a graph represented as Edges List.

Time complexity: O(|E|log|E|), where |E| is the number of edges and this complexity comes from the sorting.
Space complexity: O(|E| + |V|), where |V| is the number of vertices.

Explanation: sort the edges in ascending order and iterate through them. Using the Disjoint Sets to know if two vertices are already connected or not. For that, at each edge check if the "from" and "to" nodes belongs to the same set with findSet() and if they do ignore, otherwise joing the sets and add this edge to the answer.

```
struct Edge {
  int from, to, weight;
  Edge(int x, int y, int z) : from(x), to(y), weight(z) {}
};

bool cmp(Edge ed1, Edge ed2) {
  return ed1.weight <= ed2.weight;
}

vector<Edge> kruskal(vector<Edge> &edges, vector<int> &vertices) {
  vector<Edge> ans;

  sort(edges.begin(), edges.end(), cmp);

  for (int v : vertices) makeSet(vertices[v]);

  for (Edge ed : edges) {
    Node *n1 = findSet(ed.from);
    Node *n2 = findSet(ed.to);

    if (n1 != n2) {
      join(ed.from, ed.to);
      ans.push_back(ed);
    }
  }

  return ans;
}

/* --- Disjoint Sets --- */
unordered_map<int, Node*> nodeMap;

Node *makeSet(int val) {
  Node *n = new Node(0, val, NULL);
  n->parent = n;
  nodeMap[val] = n;

  return n;
}

Node *findSetUtil(Node *n) {
  if (n == n->parent) return n;
  n->parent = findSetUtil(n->parent);
  return n->parent;
}

Node *findSet(int val) {
  if (nodeMap.find(val) != nodeMap.end()) {
    return findSetUtil(nodeMap[val]);
  }
  return NULL;
}
```

```
void join(int val1, int val2) {
  Node *n1 = findSet(val1);
  Node *n2 = findSet(val2);

  if (n1 == n2 || !n1 || !n2) return;

  if (n1->rank >= n2->rank) {
    if (n1->rank == n2->rank) (n1->rank)++;
    n2->parent = n1;
  }
  else {
    n1->parent = n2;
  }
}
```

Graph Largest Distance

```
Find largest distance. Given an arbitrary unweighted rooted tree which
consists of N (2 <= N <= 40000) nodes. The goal of the problem is to find
largest distance between two nodes in a tree. Distance between two nodes is
a number of edges on a path between the nodes (there will be a unique path
between any pair of nodes since it is a tree). The nodes will be numbered 0
through N - 1.

The tree is given as an array P, there is an edge between nodes P[i] and
i (0 <= i < N). Exactly one of the i’s will have P[i] equal to -1, it will
be root node.
```

Example:

```
If given P is [-1, 0, 0, 0, 3], then node 0 is the root and the whole tree
looks like this:
          0
       /  |  \
      1   2   3
               \
                4
One of the longest path is 1 -> 0 -> 3 -> 4 and its length is 3.
```

Time complexity: O(N), where N is the maximum number of nodes in the tree.
In other words, N is the array length.
Space complexity: O(N).

Explanation: to get linear time complexity we need to rebuild the graph using a Adjacency List. Now, we perform Depth First Search to get the distance of a pair of nodes. The maximum distance will always be made of two parts p1 and p2. These parts are the maximum distance and the second maximum distance from a node to another node.
In my code, I get the distance (or in other words, the height of that node) from bottom-up returning maxCurrDistance + 1 of each node. And I keep two variables at each function call (notice that each function call is responsible for a single node) to store p1 and p2. Also, at each function call after the "for" loop that investigates the edges of each vertex, I check for the maximum p1 and p2 if the current answer is smaller than p1 + p2 and then update answer.

```
int dfsUtil(vector<vector<int>> &a, int v, vector<bool> &visited, int &ans) {
  visited[v] = true;
  int count = 0, p1 = 0, p2 = 0;

  for (int i = 0; i < a[v].size(); i++) {
    int u = a[v][i];
    if (!visited[u]) {
      int currDist = dfsUtil(a, u, visited, ans);

      if (currDist > p1 || currDist > p2) {
        p1 = max(p1, p2);
        p2 = currDist;
      }
    }
  }

  ans = max(ans, p1 + p2);

  return max(p1, p2) + 1;
}

int solve(vector<int> &a) {
  int src, ans = 0;
  vector<bool> visited(a.size(), false);
  vector<vector<int>> graph(a.size(), vector<int>());

  for (int i = 0; i < a.size(); i++) {
    if (a[i] != -1) graph[a[i]].push_back(i);
    else src = i;
  }

  dfsUtil(graph, src, visited, ans);

  return ans;
}

// The solution below doesn't rebuild the graph. Instead it uses the edge
// list the exercise gives, so the space complexity would be logN, but the
// time complexity would be N^2.
// int dfsUtil(vector<int> &a, int v, vector<bool> &visited, int &ans) {
//   visited[v] = true;
//   int count = 0, p1 = 0, p2 = 0;

//   for (int i = 0; i < a.size(); i++) {
//     if (a[i] == v && !visited[i]) {
//       int currDist = dfsUtil(a, i, visited, ans);

//       if (currDist > p1 || currDist > p2) {
//         p1 = max(p1, p2);
//         p2 = currDist;
//       }
//     }
//   }

//   ans = max(ans, p1 + p2);

//   return max(p1, p2) + 1;
// }

// int solve(vector<int> &a) {
//   int src, ans = 0;
//   vector<bool> visited(a.size(), false);

//   for (int i = 0; i < a.size(); i++) {
//     if (a[i] == -1) {
//       src = i;
//       break;
//     }
//   }

//   dfsUtil(a, src, visited, ans);

//   return ans;
// }
```

Graph Level Order

```
Given a binary tree, return the level order traversal of its nodes’ values.
(ie, from left to right, level by level).

Example :
Given binary tree,
    3
   / \
  9  20
    /  \
   15   7
return its level order traversal as:
[
  [3],
  [9,20],
  [15,7]
]
```

Time complexity: O(N), where N is the number of nodes in the tree.
Space complexity: O(M), where M is the number of nodes in the last level of the tree, which is 2^logN.

Explanation: just use a queue to save each level. Since the exercise is asking for the answer as a vector of vectors, where each inner vector is a level, we need an auxiliary to save all the nodes in a level without interfering with the nodes that are already in the queue and that are part of the same level. In the end we replace the main queue with the auxiliary queue. In other words, process a level and in another queue save all the children of this level, then, after processing this level, replace the main queue with the auxiliary queue with all children because they are the next level.

```cpp
vector<vector<int>> levelOrder(TreeNode *root) {
  vector<vector<int>> ans;

  if (root == NULL) return ans;

  queue<TreeNode *> q;
  q.push(root);

  while (!q.empty()) {
    vector<int> temp;
    queue<TreeNode *> nextQ;

    while (!q.empty()) {
      TreeNode *t = q.front();
      q.pop();
      temp.push_back(t->val);

      if (t->left != NULL) nextQ.push(t->left);
      if (t->right != NULL) nextQ.push(t->right);
    }

    if (temp.size()) ans.push_back(temp);
    q = nextQ;
  }

  return ans;
}
```

Graph Prim

```
Prim's Algorithm for a graph represented as Adjacency List.
```

Time complexity: O(|E| * log|V|), where |E| is the number of edges and |V| is the number of vertices.
Space complexity: O(|E| + |V|).

Explanation: you will need a Min Heap with Decrease operation and the C++ standard library doesn't support this operation, so you need to implement your own or use a set (personally in a competition I would go with the set).
Prim's Algorithm, different from Kruskal's Algorithm, has a source node for the MST. The idea is to check from which of the vertices already visited we have the edge with the smallest weight.
We initialize our Heap with 0 cost for the source vertex and Infinity for every other vertex in the graph. Then, We start from the source and check its edges, after we pick the edge with smallest weight (notice, we don't have another choice and an edge needs to be picked because we need to start our MST from the source vertex). In the next iteration we look at the edges that comes from the "to" vertex of the just picked edge and do the same thing. After this second check we need, again, to get the edge with the smallest weight (as before an edge needs to be picked so we can continue to build our MST). We keep doing this until there are no more nodes in Heap.
For example:

```
Given,
      1
  a ---- c
  |     /
2 |   / 3
  | /
  b

Source: a

#1 Iteration
Heap:       Map Vertex->Edge:    Result:
a - 0
b - Inf
c - Inf

#2 Iteration
Heap:       Map Vertex->Edge:    Result:
c - 1       c -> (a, c)          (a, c)
b - 2       b -> (a, b)

#3 Iteration
Heap:       Map Vertex->Edge:    Result:
            c -> (a, c)          (a, c)
b - 2       b -> (a, b)
In this iteration "c" just has one edge to b and it is not smaller than the
already visited (a, b) so we don't do anything.

Heap:       Map Vertex->Edge:    Result:
            c -> (a, c)          (a, c), (a, b)
            b -> (a, b)
```

Notice, the map Vertex->Edge is used to identify the edge that gives the weight we currently have in the heap for that vertex. So, every time we remove a vertex from the heap because it has the smallest edge with extractMin() we go to this map to get the edge that we need to add to our result.

```cpp
struct Edge {
  int to, weight;
  Edge(int x, int y) : to(x), weight(y) {}
};

vector<pair<int, int>> prim(vector<vector<Edge>> &g, int src) {
  int numVertices = g.size();
  // MinHeap with decrease operations and vertices identified by integers.
  MinHeap<int> heap;
  // Map of vertex to Edge, so we know which edge has the smallest weight to
  // this vertex.
  vector<pair<int, int>> edges(numVertices);
  // The edges in the MST.
  vector<pair<int, int>> ans;

  // Initialize heap with 0 cost to the source vertex and Infinity to every
  // other vertex.
  for (int i = 0; i < numVertices; i++) {
    if (i == src) heap.push(i, 0);
    else heap.push(i, INT_MAX);
  }

  while (!heap.empty()) {
    // First integer is the vertex, second is the weight. Extract the node with
    // the smallest weight.
    pair<int, int> node = heap.extractMin();
    int v = node.first;

    // Add edge that goes to the current vertex with smallest weight.
    if (v != src) ans.push_back(edges[v]);

    // Iterate through all the edges of a given vertex v.
    for (int i = 0; i < g[v].size(); i++) {
      int u = g[v][i].to, w = g[v][i].weight;
      // If the vertex is in the heap AND the weight to get to it is greater
      // than the weight of the current edge.
      if (heap.contains(u) && heap.getWeight(u) > w) {
        heap.decreaseKey(u, w);
        edges[u] = {v, g[v][i].to};
      }
    }
  }

  return ans;
}
```

Graph Topological Sort

Topological Sort for a graph represented as Adjacency List.

Time complexity: O(N), where N is the number of nodes.
Space complexity: O(N).

Explanation: for each every u->v u must be placed before v. Topological sort is commonly used in build systems where v has a dependecy of u so u must be built before v. Topological Sort uses Depth First Search to get to a node with no edges so we can insert it into a stack (this stack is our answer).

```
struct Edge {
  int to;
  Edge(int x): to(x) {}
};

void topologicalSortUtil(
  int v, vector<vector<Edge>> &g, unordered_set<int> &visited, stack<int> &stk
) {
  visited.insert(v);

  for (int i = 0; i < g[v].size(); i++) {
    int u = g[v][i].to;
    if (visited.find(u) != visited.end()) continue;
    topologicalSortUtil(u, g, visited, stk);
  }

  stk.push(v);
}

vector<int> topologicalSort(vector<vector<Edge>> &g) {
  int numVertices = g.size();
  stack<int> stk;
  unordered_set<int> visited;
  vector<int> ans(numVertices, -1);

  for (int i = 0; i < numVertices; i++) {
    if (visited.find(i) != visited.end()) continue;
    topologicalSortUtil(i, g, visited, stk);
  }

  for (int i = 0; i < numVertices; i++) {
    ans[i] = stk.top();
    stk.pop();
  }

  return ans;
}
```

Greedy Distribute Candy

There are N children standing in a line. Each child is assigned a rating value. You are giving candies to these children subjected to the following requirements:
1. Each child must have at least one candy.
2. Children with a higher rating get more candies than their neighbors.

What is the minimum candies you must give?

Example:
Given,
Ratings [1 2]
return 3, because the candidate with 1 rating gets 1 candy and candidate with rating cannot get 1 candy as 1 is its neighbor. So rating 2 candidate gets 2 candies. In total, 2+1 = 3 candies need to be given out.

Time complexity: O(N), where N is the size of the given array.
Space complexity: O(N), but can be done in O(1) if use a arithmetic progression sum).

Explanation: remember that every child must have at least one candy, and that if a child has a neighbor with a bigger rating this neighbor needs to receive more candies. So, what happens if we have:
ratings: 3 6 9
We start with 1 candy and then we increment until the last child:
ratings: 3 6 9
candies: 1 2 3
And if we have:
ratings: 9 6 3
Now, if we start from the end of the array to the beginning we would also start with 1 candy and then increment:

```
ratings: 9 6 3
candies: 3 2 1
```

So, lets use DP. We first iterate from the beginning of our loop looking for increasing sequences and for each a[i - 1] < a[i] we make
dp[i] = dp[i - 1] + 1
Then, after the above loop (and not nested) we start from the end of the array looking for increasing sequences again (increasing looking from right to left) and we make
dp[i] = dp[i + 1] + 1
Finally, get the sum of you DP vector and you are done.

```
int candy(vector<int> &a) {
  int ans = 0, n = a.size();
  vector<int> dp(n, 1);

  for (int i = 1; i < n; i++) {
    if (a[i - 1] < a[i]) dp[i] = max(dp[i], dp[i - 1] + 1);
    // else if (a[i - 1] == a[i]) dp[i] = max(dp[i], dp[i - 1]);
  }

  for (int i = n - 2; i >= 0; i--) {
    if (a[i + 1] < a[i]) dp[i] = max(dp[i], dp[i + 1] + 1);
    // else if (a[i + 1] == a[i]) dp[i] = max(dp[i], dp[i + 1]);
  }

  for (int i = 0; i < n; i++) {
    ans += dp[i];
  }

  return ans;
}
```

Greedy Gas Station

There are N gas stations along a circular route, where the amount of gas at station i is gas[i]. You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station i to its next station (i+1). You begin the journey with an empty tank at one of the gas stations.

Return the minimum starting gas station's index if you can travel around the circuit once, otherwise return -1.

Note 1: You can only travel in one direction.
i to i + 1, i + 2, ... n - 1, 0, 1..
Note 2: Completing the circuit means starting at i and ending up at i again.

Example:
Given,
Gas: [1, 2]
Cost: [2, 1]
Output: 1

If you start from index 0, you can fill in gas[0] = 1 amount of gas. Now your tank has 1 unit of gas. But you need cost[0] = 2 gas to travel to station 1.
If you start from index 1, you can fill in gas[1] = 2 amount of gas. Now your tank has 2 units of gas. You need cost[1] = 1 gas to get to station 0. So, you travel to station 0 and still have 1 unit of gas left over. You fill in gas[0] = 1 unit of additional gas, making your current gas = 2. It costs you cost[0] = 2 to get to station 1, which you do and complete the circuit.

Time complexity: O(N), where N is the size of the arrays.

Space complexity: O(1).

Explanation: think about the fuel tank, while it is positive you can keep adding stations because even if there are negative pumps in your way when you find a positive pump you still would end up with more fuel than if you have started at this just found positive pump (of course, right? because you are adding the left fuel in your tank to this positive number). Now, what happens when you tank becomes negative? Well, this means no pump before this pump that just made your tank negative will be able to complete the trip. So, when your tank becomes negative, just restart it to zero and try to complete the trip from the next pump. In other words, the only possible pumps to fulfill the task are the ones that follow a pump that is making your tank negative.

```
int canCompleteCircuit(vector<int> &gas, vector<int> &cost) {
  int sumGas = 0, sumCost = 0, start = 0, tank = 0;
  for (int i = 0; i < gas.length; i++) {
    sumGas += gas[i];
    sumCost += cost[i];
    tank += gas[i] - cost[i];
    if (tank < 0) {
      start = i + 1;
      tank = 0;
    }
  }

  return sumGas < sumCost ? -1 : start;
}

// int canCompleteCircuit(const vector<int> &a, const vector<int> &b) {
//   int n = a.size();

//   if (n == 0) return -1;
//   if (n == 1) return 0;

//   int start = -1, tank = 0, i = 0, count = 0;

//   // You need at most 2n executions of this loop to find an answer.
//   while (i != start && count < 2 * n) {
//     tank += (a[i] - b[i]);

//     if (tank < 0) {
//       tank = 0;
//       start = -1;
//     }
//     else if (start == -1) {
//       start = i;
//     }

//     i = (i + 1) % n;
//     count++;
//   }

//   return count < 2 * n ? start : -1;
// }
```

Greedy Highest Product

Given an array of integers, return the highest product possible by multiplying 3 numbers from the array.

Example:

```
[0, -1, 3, 100, 70, 50] => 70*50*100 = 350000
```

Time complexity: O(NlogN), where N is the size of the array.
Space complexity: O(1).

Explanation: sort the array in increasing order. Then, you have two possible cases:
1. NEGATIVE * NEGATIVE * POSITIVE, where the largest result possible for the multiplication NEGATIVE * NEGATIVE is of course at the beginning of the array.
2. LARGEST * SECOND_LARGEST * THIRD_LARGEST, where these elements are the largest ones so they are at the end of the array.

```
int maxp3(vector<int> &a) {
  sort(a.begin(), a.end());

  int n = a.size();

  // Case 1: NEGATIVE * NEGATIVE * POSITIVE
  int p1 = a[0] * a[1] * a[n - 1];
  // Case 2: LARGEST * SECOND_LARGEST * THIRD_LARGEST
  int p2 = a[n - 1] * a[n - 2] * a[n - 3];

  return  max(p1, p2);
}
```

Greedy Light Bulbs

N light bulbs are connected by a wire. Each bulb has a switch associated with it, however due to faulty wiring, a switch also changes the state of all the bulbs to the right of current bulb. Given an initial state of all bulbs, find the minimum number of switches you have to press to turn on all the bulbs. You can press the same switch multiple times.

Note : 0 represents the bulb is off and 1 represents the bulb is on.

Example:

```
Given [0 1 0 1], return 4
  press switch 0 : [1 0 1 0]
  press switch 1 : [1 1 0 1]
  press switch 2 : [1 1 1 0]
  press switch 3 : [1 1 1 1]
```

Time complexity: O(N), where N is the size of the array.
Space complexity: O(1).

Explanation: since just the bulbs at the right of the current bulb have their state changed, there is no way to change the first 0 bulb to 1 unless we perform that change when we are at it. So, just do that. Iterate from the beginning of the array and at each time you find a "off" bulb turn it on and look for the next "off" bulb.

```
int bulbs(vector<int> &a) {
  bool invert = false;
  int m = 0;

  for (int i = 0; i < a.size(); i++) {
    if (invert == false) {
      if (a[i] == 0) {
        m++;
        invert = true;
      }
    }
    else {
      if (a[i] == 1) {
        m++;
        invert = false;
      }
    }
  }

  return m;
}
```

Greedy Majority Element

Given an array of size n, find the majority element. The majority element is the element that appears more than floor(n/2) times.
You may assume that the array is non-empty and the majority element always exist in the array.

Example:
Given [2, 1, 2], return 2, which occurs 2 times which is greater than 3/2.

Time complexity: O(N), where N is the size of the array.
Space complexity: O(1).

Explanation: it can be done easily in O(N^2), where you count each element to see if the number of repetitions is bigger than n / 2, but instead of this brute force approach you should use Moore's Voting Algorithm.

```
int majorityElement(vector<int> &a) {
  int count = 1, elem = a[0];

  for (int i = 1; i < a.size(); i++) {
    count = elem == a[i] ? count + 1 : count - 1;
    if (count == 0) {
      elem = a[i];
      count = 1;
    }
  }

  return elem;
}
```

Greedy Mice Holes

There are N Mice and N holes are placed in a straight line. Each hole can accomodate only 1 mouse. A mouse can stay at his position, move one step right from x to x + 1, or move one step left from x to x - 1. Any of these moves consumes 1 minute. Assign mice to holes so that the time when the last mouse gets inside a hole is minimized.

Note: The final answer will fit in a 32 bit signed integer.

Example:
positions of mice are: 4 -4 2
positions of holes are: 4 0 5

```
Assign mouse at position x = 4 to hole at position x = 4: Time taken is 0
Assign mouse at position x = -4 to hole at position x = 0: Time taken is 4
Assign mouse at position x = 2 to hole at position x = 5: Time taken is 3
After 4 minutes all of the mice are in the holes.
```

Since, there is no combination possible where the last mouse's time is less than 4, answer = 4.

Time complexity: O(NlogN), where N is the size of the arrays.
Space complexity: O(1).

Explanation: to solve this exercise we need to assign mice to their closest holes. This way we minimize the time the mouse that is furthest to a hole will take to get into one. To do that we just need to sort both of these arrays so mice will be assigned to their closest holes. The maximum time taken by a mouse to get into a hole is max of abs(a[i] - b[i]), where "a" is mice's positions and b is the positions of the holes.

```
int mice(vector<int> &a, vector<int> &b) {
  sort(a.begin(), a.end());
  sort(b.begin(), b.end());

  int m = 0;

  for (int i = 0; i < a.size(); i++) {
    m = max(m, abs(a[i] - b[i]));
  }

  return m;
}
```

Greedy Seats

There is a row of seats. Assume that it contains N seats adjacent to each other. There is a group of people who are already seated in that row randomly. i.e. some are sitting together & some are scattered. An occupied seat is marked with a character 'x' and an unoccupied seat is marked with a dot ('.').
Now your target is to make the whole group sit together i.e. next to each other, without having any vacant seat between them in such a way that the total number of hops or jumps to move them should be minimum.

Return minimum value % MOD where MOD = 10000003.

Example:

```
Here is the row having 15 seats represented by the String -
             . . . . x . . x x . . . x . .

Now to make them sit together one of approaches is -
             . . . . . . x x x x . . . . .

Following are the steps to achieve this -
1. Move the person sitting at 4th index to 6th index -
  Number of jumps by him =   (6 - 4) = 2

2. Bring the person sitting at 12th index to 9th index -
  Number of jumps by him = (12 - 9) = 3

So now the total number of jumps made = ( 2 + 3 ) % MOD = 5 which is the
minimum possible jumps to make them seat together.
```

There are also other ways to make them sit together but the number of jumps will exceed 5 and that will not be minimum.

For example, bring them all towards the starting of the row i.e. start placing them from index 0. In that case the total number of jumps will be (4 + 6 + 6 + 9) % MOD = 25 which is very costly and not an optimized way to do this movement.

Time complexity: O(N), where N is the length of the string.
Space complexity: O(1).

Explanation: my solution is different from the editorial, but I believe it is easier to read and understand. Besides, it has O(1) space complexity.
You need to realize three things:
1. the empty seats right at the beginning of a row will never be used, because the cheapest solution will begin at most at the first occupied spot.
2. Once you move a group to the left you will never move this group to the right, because if you move a person 2 seats to the left and then you at some point you move this person 3 seats to the right would be cheapier if you just had left this person where she was. Same thing goes if you move to the right.
3. Now that you know the two facts above, for every group you just need to decide if this group must stay where it is, must go to the left to join the other groups on the left, or must go to the right to joing the other groups on the right.

It turns out the decision pointed out on item 3 is easy to make. You just need the total number of people. If the number of people on the left is smaller than the number of people on the right, then you move all people on the left to the right. If the number of people on the left is bigger than the number of people on the right, then you move all people on the right to the left.
I will try to explain better what I said above. If you move people that are on the right to the left, you will need to keep moving this group of people to the left in other iterations because of item 2. So, if you have more people in total (with total I mean all people that are on the right, even if there are empty spots between them) on the right you will end up making more movies than you could if you had done the opposite, that is, move people on the left to the right.
Another thing, notice that if you have for example 3 empty seats and you are about to move a group of 3 people to the left/right, what you are actually doing is moving everybody to the left/right of "i" to this new spot at 3 seats to the left/right (and not just this group). So, we count the number of empty seats at that iteration, and we decide if we gonna move totalPeopleOnTheLeft to the right, or if we gonna move totalPeopleOnTheRight to the left, and then we add to our answer
numberOfEmptySeats * totalPeopleOnTheLeft or
numberOfEmptySeats * totalPeopleOnTheRight
and reset numberOfEmptySeats to 0, so we can start the count on the next ieration.

For example,
```
  Given,                       ...x..xx...x.......xxxx...xx..
  Move first group to the right .....xxx...x.......xxxx...xx..
  Move first group to the right ........xxxx.......xxxx...xx..
  Move first group to the right ...............xxxxxxxx...xx..
  Move last group to the left   ...............xxxxxxxxxx.....
```

And an example of the fact that you do not only move a single group, but all the poeple on the left/right of "i",
```
  Given, xxxx....x.x.x
  Since your first group has size of 4 we have more people on the left of
  "i" (i right now is at index 4, on the ".""" right after the last "x" of
  this first group). So we need to move people that are on the right side
  of "i" to the left. So we get,
  xxxxx.x.x....
  Notice, how we not just moved the group at index 8, but everybody to the
```

left. This is important because it makes the implementation much simpler
than trying to do one group at a time.

```
int seats(string a) {
  int totalPeople = 0;
  for (int i = 0; i < a.size(); i++) {
    if (a[i] == 'x') totalPeople++;
  }

  int i = 0, peopleOnTheLeft = 0, moves = 0, MOD = 10000003;

  while (a[i] == '.') i++;

  while (i < a.size()) {
    int emptySeats = 0;

    for (; i < a.size() && a[i] == 'x'; i++) peopleOnTheLeft++;
    for (; i < a.size() && a[i] == '.'; i++) emptySeats++;

    int peopleOnTheRight = (totalPeople - peopleOnTheLeft) % MOD;
    peopleOnTheLeft %= MOD;
    emptySeats %= MOD;

    if (peopleOnTheLeft < peopleOnTheRight) {
      moves = (moves + ((emptySeats * peopleOnTheLeft) % MOD)) % MOD;
    }
    else {
      moves = (moves + ((emptySeats * peopleOnTheRight) % MOD)) % MOD;
    }
  }

  return moves;
}
```

Hash Colorful Number

For Given Number N find if its COLORFUL number or not. Return 0/1.

A number can be broken into different contiguous sub-subsequence parts.
Suppose, a number 3245 can be broken into parts like:
3 2 4 5 32 24 45 324 245.
And this number is a COLORFUL number, since product of every digit of a
contiguous subsequence is different.

Example:
N = 23
2 3 23
2 -> 2
3 -> 3
23 -> 6
this number is a COLORFUL number since product of every digit of a
sub-sequence are different.
Output : 1

Time complexity: O(N), where N is the number of digits.
Explanation:

```
vector<int> getDigits(int a) {
  vector<int> digits;

  while (a > 0)  {
    digits.push_back(a % 10);
    a /= 10;
  }

  return digits;
}

int colorful(int a) {
  vector<int> d = getDigits(a);
  unordered_set<int> s;

  for (int i = 0; i < d.size(); i++) {
    int product = 1;

    for (int j = i; j >= 0; j--) {
      product *= d[j];
      if (s.find(product) != s.end()) return 0;
      s.insert(product);
    }
  }

  return 1;
}
```

Hash Diff Elements Equal K-2

```
Given an array 'A' of integers and another non negative integer k,
find if there exists 2 indices i and j such that A[i] - A[j] = k, i != j.
```

Example:
```
A : [1 5 3]
k : 2

Output : True, because 3 - 1 = 2
```

Time complexity: O(N), where N is the size of the array.
Explanation: Iterate over your list. Since we know the value k, at each value of our array we have:
```
aj = arr[i] - k
ai = arr[i] + k
```
So, we keep a unordered_set for the values of our array we already checked with our iteration. If we find aj or ai in our set, so return true.

```
  int diffPossible(const vector<int> &num, int diff) {
    if (num.size() < 2 || diff < 0) return false;

    unordered_set<int> S;

    for (int i = 0; i < num.size(); i++) {
      int aj = num[i] - diff;
      int ai = num[i] + diff;

      if (S.find(aj) != S.end()) return true;
      if (S.find(ai) != S.end()) return true;

      S.insert(num[i]);
    }

    return false;
  }

  // int diffPossible(const vector<int> &a, int b) {
```

```
//   if (a.size() <= 1) return 0;

//   unordered_map<int, int> m;

//   for (int i = 0; i < a.size(); i++) {
//     m.insert(make_pair(a[i], i));
//   }

//   for (int i = 0; i < a.size(); i++) {
//     int diff = a[i] - b;
//     unordered_map<int, int>::const_iterator it = m.find(diff);
//     if (it != m.end() && it->second != i) return 1;
//   }

//   return 0;
// }
```

Hash Distinct Numbers In Window

You are given an array of N integers, A1, A2 ,…, AN and an integer K.
Return the of count of distinct numbers in all windows of size K.
Formally, return an array of size N-K+1 where i’th element in this array
contains number of distinct elements in sequence Ai, Ai+1 ,…, Ai+k-1.

Note:
If K > N, return empty array.

Example,
A = [1, 2, 1, 3, 4, 3] and K = 3
All windows of size K are:
[1, 2, 1]
[2, 1, 3]
[1, 3, 4]
[3, 4, 3]
So, we return an array [2, 3, 3, 2].

Time complexity: O(N), where N is the size of the given array.

Explanation: start with an empty hashmap where the keys are the elements of the array and the values are counters for how many times these elements appeared. Each time you arrive at the start of a new window you will decrement the first element of the previous window and add (or increment if it already exists in the hashmap) the the last element of the current window, both, in the hashmap. For each window the number of distinct elements is the size of the hashmap. When you are doing the process of removing or decrementing the first element of the previous window don't forget that when the counter becomes zero you must delete the entry from the hashmap.

```
vector<int> dNums(vector<int> &a, int k) {
  vector<int> ans;
  unordered_map<int, int> m;

  if (k <= 0 || k > a.size()) return ans;

  for (int i = 0; i < k; i++) {
    if (m.find(a[i]) != m.end()) m[a[i]] += 1;
    else m[a[i]] = 1;
  }

  ans.push_back(m.size());

  int prevFirst = a[0];
```

```
    for (int i = 1; i <= a.size() - k; i++) {
      int last = a[i + k - 1];

      if (m.find(prevFirst) != m.end()) {
        m[prevFirst] -= 1;
        if (m[prevFirst] <= 0) m.erase(prevFirst);
      }

      if (m.find(last) != m.end()) m[last] += 1;
      else m[last] = 1;

      ans.push_back(m.size());

      prevFirst = a[i];
    }

    return ans;
  }
```

Hash Equal Sum In Array

```
Given an array A of integers, find the index of values that satisfy
A + B = C + D, where A,B,C & D are integers values in the array. If no
solution is possible, return an empty list.

Notes:
1) Return the indices `A1 B1 C1 D1`, so that
 A[A1] + A[B1] = A[C1] + A[D1]
 A1 < B1, C1 < D1
 A1 < C1, B1 != D1, B1 != C1

2) If there are more than one solutions,
then return the tuple of values which are lexicographical smallest.

Assume that if we have two solutions
S1 : A1 B1 C1 D1 ( these are values of indices int the array )
S2 : A2 B2 C2 D2

then S1 is lexicographically smaller than S2 iff
A1 < A2 OR
A1 = A2 AND B1 < B2 OR
A1 = A2 AND B1 = B2 AND C1 < C2 OR
A1 = A2 AND B1 = B2 AND C1 = C2 AND D1 < D2
```

Example:
Input: [3, 4, 7, 1, 2, 9, 8]
Output: [0, 2, 3, 5] (O index)

Time complexity: O(N^2), where N is the length of the array.
Explanation: notice, that B1 can be larger than C1 and D1. So, we iterate over the array with two loops and we keep a map>.
The outer loop using a variable, lets say, "i" starts at 0, and the inner loop with a variable, lets say, "j", starts at i + 1. The loops are basically selecting all the possible values for A and B of the equation. The key of the map, is the sum of two values in the array, the value of the map is a pair with the indexes this values are. Each time we find a sum we check to see if the answer we just fuond is lexicographically smaller than the one we had before. Notice, that when we find a sum in the map we do not update our map because the sum we found before has smaller indexes than the one we just found.

```
  void storeAns(vector<int> &ans, vector<int> &v) {
    if (ans.size() != 0) {
      if(v[0] < ans[0] ||
        (v[0] == ans[0] && v[1] < ans[1]) ||
        (v[0] == ans[0] && v[1] == ans[1] && v[2] < ans[2]) ||
        (v[0] == ans[0] && v[1] == ans[1] && v[2] == ans[2] && v[3] < ans[3])) {
        ans[0] = v[0];
        ans[1] = v[1];
        ans[2] = v[2];
        ans[3] = v[3];
      }
    }
    else {
      ans.push_back(v[0]);
      ans.push_back(v[1]);
      ans.push_back(v[2]);
      ans.push_back(v[3]);
    }
  }

  vector<int> equal(vector<int> &a) {
    if (a.size() < 4) return vector<int>();

    vector<int> ans;
    unordered_map<int, pair<int, int>> m;

    for (int i = 0; i < a.size() - 1; i++) {
      for (int j = i + 1; j < a.size(); j++) {
        int sum = a[i] + a[j];
        unordered_map<int, pair<int, int>>::iterator it = m.find(sum);

        if (it != m.end()) {
          pair<int, int> prevIdx = it->second;
          if (prevIdx.first < i && prevIdx.second != i && prevIdx.second != j) {
            vector<int> t = {prevIdx.first, prevIdx.second, i, j};
            storeAns(ans, t);
          }
        }
        else {
          m[sum] = make_pair(i, j);
        }
      }
    }

    return ans;
  }
```

Hash Fraction

Given two integers representing the numerator and denominator of a fraction, return the fraction in string format. If the fractional part is repeating, enclose the repeating part in parentheses.

Example:
Given numerator = 1, denominator = 2, return "0.5"
Given numerator = 2, denominator = 1, return "2"
Given numerator = 2, denominator = 3, return "0.(6)"

Time complexity: O(N), where N is the total number of digits the number will have.

Explanation: calculate digit by digit, where
integer part = numerator / denominator and
decimals = initialize "rest" with (numerator % denominator) and then perform
  (rest * 10) % denominator

```
  rest = rest % denominator
  until rest == 0
```

The secret to identify the repeating part is to use a hashmap to store our previously seen "rest's", because when you find a "rest" that you already saw before the calculations (rest / denominator) and (rest % denominator) will start to repeat of course. So, use a hashmap to store your "rest's" and the position where they were first seen (you gonna need this position to insert the opening parenthesis "(").

```
string getDecimals(long long r, long long d) {
  string decimals = "";
  unordered_map<long long, int> st;

  while (r != 0) {
    unordered_map<long long, int>::iterator it = st.find(r);

    // When a number was already seen the digits will start to repeat because
    // when you perform r / d and r % d you will get the same sequence of
    // digits.
    if (it != st.end()) {
      int idx = it->second;
      return (decimals.substr(0, idx) +
        "(" + decimals.substr(idx, decimals.length() - idx) + ")");
    }

    // Remember the current position so we can insert "(".
    st[r] = decimals.length();
    // Get the next digit.
    r *= 10;
    string digit = to_string(r / d);
    decimals += digit;

    r %= d;
  }

  return decimals;
}

string fractionToDecimal(int numerator, int denominator) {
  if (numerator == 0) return "0";

  long long n = numerator;
  long long d = denominator;

  string sign = "";
  if ((n < 0 && d > 0) || (n > 0 && d < 0)) sign = "-";
  n = abs(n);
  d = abs(d);

  string ans = "";
  long long integerPart = n / d;

  ans += to_string(integerPart);

  if (n % d) {
    string decimals = getDecimals(n % d, d);
    if (decimals != "") ans += '.' + decimals;
  }

  return sign + ans;
}
```

Hash Longest Substring Without Repeat

Given a string, find the length of the longest substring without repeating characters.

Example:
The longest substring without repeating letters for "abcabcbb" is "abc", which the length is 3.

For "bbbbb" the longest substring is "b", with the length of 1.

Time complexity: O(N), where N is the length of the given string.

Explanation: iterate over the string and keep two variables: a variable for the start of your substring (call it s), and another one for the end (call it e). Also, keep a map. The key of the map store a character and the value store the index where that value was first found.

When you, during your iteration, find a character that is already in the map you need to:
1. Check if the substring you just found is larger than the one you had before.
2. Update your start variable s to the next index from where you found the character for the first time (in other words, to the int value from your map plus 1).
3. Make e = s.
4. Clear your map.

When you don't find the current character in your map you add the character to your map with the character as the key and its index as value, and increment e.

Stop the iteration when s or e are equal or larger to the original string length.

```
int lengthOfLongestSubstring(string a) {
  unordered_map<char, int> m;
  int ans = 0, s = 0, e = 0;

  while (s < a.length() && e < a.length()) {
    unordered_map<char, int>::iterator it = m.find(a[e]);

    if (it != m.end()) {
      if (e - s > ans) ans = e - s;

      s = it->second + 1;
      e = s;
      m.clear();
    }
    else {
      m.insert(make_pair(a[e], e));
      e++;
    }
  }

  return max(e - s, ans);
}
```

Hash Substring Of All Words

You are given a string, S, and a list of words, L, that are all of the same length. Find all starting indices of substring(s) in S that is a concatenation of each word in L exactly once and without any intervening characters.

Example :
S: "barfoothefoobarman"
L: ["foo", "bar"]
You should return the indices: [0,9].
(order does not matter).

Time complexity: O(N * M), where N is the string length and M is the number of words.

Explanation: suppose n = S.length(), m = words.size(), l = words[0].length(). We need a hashmap to count the frequency of our words, because they can repeat in our words list. So our hashmap has the from .
We iterate though our words list counting the frequency. Then, we iterate through S from 0 to n - (m * l), because it's not possible to have a substring from all words after n - (m * l). Call the variable controlling this loop "i". Each "i" represents the beginning of a possible substring. So, for each "i" we start another loop where we try to find all occurrences of our words. This internal loop is incremented by the word size in every iteration. We need to use another hashmap in this loop to help us count the occurrences of words, we also need a variable to count how many words we already found because when this variable is equal m we found a substring and we need to add it to our answer. Notice, that when we found more occurrences of a particular word than our original frequency hashmap allows we need to break this internal loop because that "i" will not be a possible beginning of a substring. Also, we need to break when we didn't find the current word in our frequency hashmap.

```
vector<int> findSubstring(string s, vector<string>& words) {
  int n = s.length(), m = words.size(), l = words[0].length();
  int ctrl = 0, nextCtrl = 0;
  vector<int> ans;
  unordered_map<string, pair<int, int>> map;

  if (n == 0 || m == 0 || l == 0) return ans;
  if (n - (m * l) < 0) return ans;

  for (int i = 0; i < words.size(); i++) {
    unordered_map<string, pair<int, int>>::iterator it = map.find(words[i]);
    if (it != map.end()) (it->second.first)++;
    else map.insert(make_pair(words[i], make_pair(1, 0)));
  }

  for (int i = 0; i <= n - (m * l); i++) {
    int j = i;
    int internalCount = 0;

    while ((j - i) < (m * l)) {
      string curr = s.substr(j, l);
      unordered_map<string, pair<int, int>>::iterator it = map.find(curr);
      j += l;

      if (it != map.end()) {
        pair<int, int> *p = &(it->second);

        if (p->first == p->second - ctrl) {
          break;
        }
        else {
          if (p->second < ctrl) p->second = ctrl + 1;
          else (p->second)++;

          nextCtrl = max(nextCtrl, p->second);
          internalCount++;
          if (internalCount == m && (j - i) == (m * l)) {
            ans.push_back(i);
          }
        }
      }
    }

    ctrl = nextCtrl;
  }

  return ans;
}

// vector<int> findSubstring(string s, vector<string>& words) {
//   int n = s.length(), m = words.size(), l = words[0].length();
//   vector<int> ans;
//   unordered_map<string, int> map;

//   if (n == 0 || m == 0 || l == 0) return ans;
//   if (n - (m * l) < 0) return ans;

//   for (int i = 0; i < words.size(); i++) {
//     unordered_map<string, int>::iterator it = map.find(words[i]);
//     if (it != map.end()) (it->second)++;
//     else map.insert(make_pair(words[i], 1));
//   }

//   for (int i = 0; i <= n - (m * l); i++) {
//     unordered_map<string, int> currMap;
//     int j = i;
//     int internalCount = 0;

//     while ((j - i) < (m * l)) {
//       string curr = s.substr(j, l);
//       unordered_map<string, int>::iterator it = map.find(curr);
//       j += l;

//       if (it != map.end()) {
//         unordered_map<string, int>::iterator it2 = currMap.find(curr);

//         if (it2 != currMap.end()) {
//           if (it2->second == it->second) break;
//           else (it2->second)++;
//         }
//         else {
//           currMap.insert(make_pair(curr, 1));
//         }

//         internalCount++;
//         if (internalCount == m && (j - i) == (m * l)) ans.push_back(i);
//       }
//     }
//   }

//   return ans;
// }
```

Integer Subsets

```
Given a set of distinct integers, S, return all possible subsets.
Conditions:
Elements in a subset must be in non-descending order.
The solution set must not contain duplicate subsets.
Also, the subsets should be sorted in ascending (lexicographic) order.
The given list is not necessarily sorted.
```

Example:

```
If S = [1,2,3], a solution is:
[
  [],
  [1],
  [1, 2],
  [1, 2, 3],
  [1, 3],
  [2],
  [2, 3],
  [3],
]
```

Time complexity: Without taking into consideration the sorting process, O(2^(N + 1)) in the recursion case, O(2^N * N) in the iterative case, where N is the size of the give list.

Explanation:

Use a sort algorithm to sort the given list.

Iterative: we have 2^N possible different subsets, so we can iterate from 0 to (2^N - 1) using a variable "i" and check the setted bits of "i". Then, we iterate from 0 to N - 1 using a variable "j". "j" gives us the a index from our original set, and the setted bits (bits with the value 1) from "i" give us all the possible combinations of subsets, where each bit position is a index from our original set. So we check to see if (1 << j) is setted in "i" and if it is we add to our current subset.

Recursive: we have two options for each element in the set: or we add it, or we not. Make a recursive call for each one of these possibilities.

Finally, use a sort algorithm to sort the answer.

```cpp
void _subsets(vector<int> &a, int i, vector<int> &c, vector<vector<int>> &ans) {
  if (i >= a.size()) {
    ans.push_back(c);
    return;
  }

  _subsets(a, i + 1, c, ans);

  c.push_back(a[i]);
  _subsets(a, i + 1, c, ans);
  c.pop_back();
}

vector<vector<int>> subsets(vector<int> &a) {
  vector<vector<int>> ans;
  vector<int> c;

  sort(a.begin(), a.end());
  _subsets(a, 0, c, ans);
  sort(ans.begin(), ans.end());

  return ans;
}

// vector<vector<int>> subsets(vector<int> &a) {
//   vector<vector<int>> ans;
//   int n = a.size();
//   sort(a.begin(), a.end());

//   for (int i = 0; i < (1 << n); i++) {
//     vector<int> t;

//     for (int j = 0; j < n; j++) {
//       if (i & (1 << j)) t.push_back(a[j]);
//     }

//     ans.push_back(t);
//   }

//   sort(ans.begin(), ans.end());

//   return ans;
// }
```

Knuth Morris Pratt

Check if a substring exists in a string using the Knuth–Morris–Pratt algorithm.

Time complexity: O(N + M), where N is the length of the string and M the length of the substring.

Explanation: the Knuth–Morris–Pratt algorithm uses the idea that when a mismatch occurs when comparing the pattern and the string the pattern itself encodes enogh information about where in the pattern we should start to try to match characters again. So it generates a table of length M, where M is the length of the pattern with this information.

Table: we always start our table with the value zero for the first character and we use two pointer, i and j, to keep track of our prefix and suffixes. Initiate i with 0, and j with 1. When i and j are equal you increment both. When they are not equal, if j – 1 is bigger than 0 you make j = table[j – 1], else you push 0 in the array and increment i.

Example:

```
a a b a a b a a a
0 1 0 1 2 3 4 5 2
```

How to use the table: the table information is saying that when a mismatch occurs (s[i] != p[j]) look at the previous index of our table (table[j – 1]) to see where we should start the match again in our pattern, withou having to go back in our current string position. For example, lets look at the first three characters of our above pattern with a hypothetical string:

```
p: a a b ...
   0 1 0
s: a a c ...
```

In "c", a mismatch occurs so we look at the previous index in our table and check where we should start the match again, in this case the table says index 1. So we try to match "c" with "a" and so on. This makes sense, because when we compared "b" with "c" and had a mismatch we had already matched two "a"s, so we are saying: "aac" is not equal "aab", but MAYBE "ac" will be equal to "aa" (that we will see in the next comparison that is actually not equal).

Note: when j, the pointer that points to the pattern characters and that we also use to control our table information, is 0 and we have a mismatch (s[i] != p[j]), we just increment i, because of course, our substring doesn't start at s[i].

```
vector<int> buildTable(string &sub) {
  int j = 0, i = 1;
  vector<int> table;

  table.push_back(0);

  while (i < sub.length()) {
    if (sub[j] == sub[i]) {
      table.push_back(j + 1);
      i++;
      j++;
    }
    else {
      if (j - 1 < 0) {
        table.push_back(0);
        i++;
      }
      else {
        j = table[j - 1];
      }
    }
  }

  return table;
}

bool findSubstr(string &str, string &sub) {
  if (sub.length() > str.length() || sub.length() == 0) return false;

  int i = 0, j = 0;
  vector<int> table = buildTable(sub);

  while (i < str.length() && j < sub.length()) {
    if (str[i] == sub[j]) {
      i++;
      j++;
    }
    else {
      if (j == 0) {
        i++;
      }
      else {
        j = table[j - 1];
      }
    }
  }

  if (j == sub.length()) return true;
  return false;
}
```

Knuth Morris Pratt All

Find all substrings occurrences in a string using the Knuth-Morris-Pratt algorithm.

Time complexity: O(N + M), where N is the length of the string and M the length of the substring.

Explanation: the Knuth-Morris-Pratt is the same. What changes is that this time we gonna use the table.back() element (that is useless when we are looking for the first occurrence of the pattern in our string) to reinitiate the matching process. So, each time we find the pattern we push the index where the pattern started in our string in our answer and we reinitiate j, the pointer that points to our pattern characters and that we also use to control our table information, to table.back(). It makes sense, because this means: when you find a substring use the information of the last character of the pattern to tell us how much we can reuse of our just matched pattern to find another pattern.

```
vector<int> buildTable(string &sub) {
  int j = 0, i = 1;
  vector<int> table;

  table.push_back(0);

  while (i < sub.length()) {
    if (sub[j] == sub[i]) {
      table.push_back(j + 1);
      i++;
      j++;
    }
    else {
      if (j - 1 < 0) {
        table.push_back(0);
        i++;
      }
      else {
        j = table[j - 1];
      }
    }
  }

  return table;
}

vector<int> findAllSubstr(string &str, string &sub) {
  vector<int> ans;

  if (sub.length() > str.length() || sub.length() == 0) return ans;

  int i = 0, j = 0;
  vector<int> table = buildTable(sub);

  while (i < str.length() && j < sub.length()) {
    if (str[i] == sub[j]) {
      i++;
      j++;

      if (j == sub.length()) {
        ans.push_back(i - sub.length());
        j = table.back();
      }
    }
    else {
      if (j == 0) {
        i++;
      }
      else {
        j = table[j - 1];
      }
    }
  }

  return ans;
}
```

Linked List

Implementation of a Linked List.

```
ListNode* create_node(int val) {
  ListNode *node = (ListNode *)malloc(sizeof(ListNode));

  node->val = val;
  node->next = NULL;

  return node;
}

ListNode* get_last(ListNode *head) {
  ListNode *curr = head;

  while (curr->next != NULL) {
    curr = curr->next;
  }

  return curr;
}

void append(ListNode *head, int val) {
  ListNode *last = get_last(head);
  ListNode *node = create_node(val);

  last->next = node;
}

void push(ListNode **head, int val) {
  ListNode *node = create_node(val);

  node->next = *head;
  *head = node;
}

void insert_after(ListNode *previous_node, int val) {
  ListNode *new_node = create_node(val);
  ListNode *previous_next = previous_node->next;

  previous_node->next = new_node;
  new_node->next= previous_next;
}

// int remove_node(ListNode **head, ListNode* node) {
//   ListNode *prev = NULL;
//   ListNode *curr = *head;

//   while (curr != node && curr != NULL) {
//     prev = curr;
//     curr = curr->next;
//   }

//   if (curr == node) {
//     if (prev == NULL) {
//       *head = curr->next;
//     }
//     else {
//       prev->next = curr->next;
//     }
//     free(curr);

//     return 1;
//   }
// return 0;
// }

/*
 * Optimized remove_node(). Instead of pointing to nodes we point to the
 * "next" pointers (or in other words to the pointers that points to nodes).
 */
int remove_node(ListNode **head, ListNode* node) {
  ListNode **curr = head;

  while (*curr != node && *curr != NULL) {
    curr = &((*curr)->next);
  }

  if (*curr == node) {
    *curr = node->next;
    free(node);
    return 1;
  }

  return 0;
}

void free_list(ListNode *head) {
  ListNode *curr = head, *tmp = NULL;

  while (curr != NULL) {
    tmp = curr;
    curr = curr->next;
    free(tmp);
  }
}

/* --- Helper functions --- */

ListNode* get_node(ListNode *head, int n) {
  ListNode *curr = head;
  int i;

  for (i = 0; i < n && curr != NULL; i++) {
    curr = curr->next;
  }

  if (i == n) {
    return curr;
  }
  else {
    return NULL;
  }
}

void append_nodes(ListNode *head, int n) {
  ListNode *last = get_last(head);
  ListNode *curr = last;

  for (int i = 1; i <= n; i++) {
    curr->next = create_node(i);
    curr = curr->next;
  }
}

void print_node(ListNode *node, char line_break) {
  cout << node->val << line_break;
}

void print_list(ListNode *head) {
  ListNode *curr = head;

  while (curr != NULL) {
    print_node(curr, ' ');
    curr = curr->next;
  }
  cout << "\n";
}
```

List Cycle

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

Example:

```
                  ______
                 |      |
                 \/     |
        1 -> 2 -> 3 -> 4
```

Return 3

Time complexity: O(N), where N is the size of the list.

Explanation: My first approach was to create a dummy node and make the "next" pointers of the nodes I already visited point to this dummy node. This solves the question and it is really easy to think about, but probably the interviewer would not like this since I'm destroying the data structure.
The right approach is to use the Floyd's Cycle Finding Algorithm.
In this algorithm we have two pointers: a hare, and a turtoise. The hare, walks at two times the speed of the turtoise. When there is a cycle eventualy they will meet (and if they do not meet before the last point of the list there is no cycle). Now, that we know if there is a cycle we need to find the start node. To do that we can:

1. Find out the size of the loop walking with one of your pointers until you meet the other one again, call this size (distance) k. Now put a pointer at the head of the list and another one k nodes distant. Walk with them one node at a time and when they meet you have the start node.

2. This another approach inveolves more math and it is less intuitive, but once reveealled it is really easy to implement. Let's see the math:
c = distance from list's head to to cycle's start node.
L = length of the loop
k = distance from meeting point to the cycle's start node.
t = number of times the turtoise ran in loop before meeting with the hare.
h = number of times the hare ran in loop before meeting with the turtoise.

Number of steps travelled by the turtoise:
s = c + tL + k
Number of steps travelled by the hare:
2s = c + hL + k
Solving to L:
L = (c + k) / (h - 2t) or
c + k = (h - 2t)L

So c + k is a multiple of L. Let's suppose h - 2t = 1, then: c = L - k.
Let's suppose h - 2t = 2, then: c = 2L - k. Now with 3, then 4, etc.
It turns out 2L, 3L, 4L are equivalent to L. Conclusion: use a pointer at the list's head and another in the meeting node. Walk with them one node at a time and when they meet you have the start node.

```
                        c
                     _______
                    |       |
                0 -> 0 -> -> 0 -> 0
                             |    |
                L - k    0    0
                         |    |
       meeting node ---> 0 <- 0
```

It is difficult to draw L - k, but it is the two nodes from the metting node to the cycle's start node.

See in the draw above how L - k = 2 nodes, and c = 2 nodes.

```
ListNode* detectCycle(ListNode* a) {
  if (a == NULL || a->next == NULL) return a;

  ListNode *t1 = a, *t2 = a, *h = a;

  do {
    t1 = t1->next;
    if (h->next) h = h->next->next;
  } while (h != NULL && h->next != NULL && h != t1);

  if (h && h->next) {
    while (t2 != t1) {
      t1 = t1->next;
      t2 = t2->next;
    }

    return t2;
  }

  return NULL;
}

// ListNode* detectCycle(ListNode* a) {
//   ListNode *cn = a, *pn = NULL, *nn = NULL;
//   ListNode *dummy = new ListNode(0);

//   while (cn != NULL && cn != dummy) {
//     nn = cn->next;
//     cn->next = dummy;
//     pn = cn;
//     cn = nn;
//   }

//   if (cn == NULL) {
//     cout << "no cycle" << '\n';
//     return NULL;
//   }
//   else {
//     cout << "pn: " << pn->val << '\n';
//     return pn;
//   }
// }
```

List Get Middle Element

Get the middle element of a linked list.

Time complexity: O(N), where N is the number of nodes in the linked list.
Space complexity: O(1).

Explanation: instead of counting the number of nodes and then iterating again over the list until you reach (count / 2)th node, use two pointers. One you use to traverse the list one node at a time, the other you traverse two nodes at a time. When the second pointer arrives to the end of the list the first pointer will at the middle. Return the first point.

```
ListNode* get_middle(ListNode *head) {
  ListNode *last = head, *middle = head;

  while (last != NULL && last->next != NULL) {
    /* Move "last" to the next node of the next node. */
    last = last->next->next;
    /* Move "middle" to the next node. */
    middle = middle->next;
  }

  return middle;
}
```

List Insertition Sort

Sort a linked list using insertion sort.

Time complexity: O(N^2), where N is the size of the list.

Explanation: We need to keep a pointer to the last element of the sorted part, call it sorted and initialize it with the head of the list. The element to be sorted in a given iteration will then be sorted->next. We can't go from right to left in a singly linked list, so we need to always go from the head until the unsorted element. While doing this when (curr->val > unsorted->val) we swap the value of the elements. Again, we are swapping values and not nodes. That means we are always swapping the last swapped value with values that are larger than it. Look:

```
Start of the inner loop:
c    s    u
|    |    |
3 -> 4 -> 2 -> 8

First swap:
    c,s   u
     |    |
2 -> 4 -> 3 -> 8

Second swap:
     s   c,u
     |    |
2 -> 3 -> 4 -> 8
```

Inner loop is now complete because current = unsorted.

With c = current, s = sorted, u = unsorted (sorted->next).

```
void swap(ListNode *a, ListNode *b) {
  if (a && b) {
    int t = a->val;
    a->val = b->val;
    b->val = t;
  }
}

ListNode* insertionSortList(ListNode* a) {
  if (!a) return a;
  ListNode *sorted = a;

  while (sorted->next != NULL) {
    ListNode *unsorted = sorted->next, *curr = a, *prev = NULL;

    while (curr != unsorted) {
      if (curr->val > unsorted->val) swap(curr, unsorted);
      curr = curr->next;
    }

    sorted = sorted->next;
  }

  return a;
}
```

List Intersection Node

Write a program to find the node at which the intersection of two singly linked lists begins.

Notes:
If the two linked lists have no intersection at all, return null.
The linked lists must retain their original structure.
You may assume there are no cycles anywhere in the entire linked structure.
Your code should preferably run in O(n) time and use only O(1) memory.

Example:

```
A:          a1 → a2
                    ↘
                      c1 → c2 → c3
                    ↗
B:     b1 → b2 → b3
```

Return c1.

Time complexity: O(N), where N is the size of the list.

Explanation: The lists may have different sizes. So, we first calculate the size of the lists. If there is a bigger one we get a pointer and walk through the list just enough to be in a node where if this node were the head the two lists would have the same size. Now, we just loop through both lists while their pointers are different. If we find a equal one we return it, otherwise we return NULL.

```
ListNode* getIntersectionNode(ListNode* a, ListNode* b) {
  ListNode *c1 = a, *c2 = b, *r = NULL;
  int n1 = 0, n2 = 0, diff = 0;

  while (c1 != NULL || c2 != NULL) {
    if (c1) {
      c1 = c1->next;
      n1++;
    }

    if (c2) {
      c2 = c2->next;
      n2++;
    }
  }
  diff = n2 - n1;
  c1 = a;
  c2 = b;

  if (diff < 0) {
    diff *= -1;
    while (diff) {
       diff--;
      c1 = c1->next;
    }
  }
  else {
    while (diff) {
       diff--;
      c2 = c2->next;
    }
  }

  while (c1 != c2) {
    c1 = c1->next;
    c2 = c2->next;
  }

  if (c1) return c1;
  else return NULL;
}
```

List Quick Sort

```
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

ListNode* partition(ListNode *head, ListNode *prev, ListNode *last) {
  int pivot = last->val;
  ListNode *wall = prev;

  for (ListNode *curr = head; curr != last; curr = curr->next) {
    if (curr->val <= pivot) {
      wall = (wall == NULL) ? head : wall->next;
      swap(&(curr->val), &(wall->val));
    }
  }

  ListNode *pivot_ancestor = wall;
  wall = (wall == NULL) ? head : wall->next;
  swap(&(last->val), &(wall->val));

  return pivot_ancestor;
}

void _quicksort(ListNode *head, ListNode *prev, ListNode *last) {
  if (last != NULL && head != last && head != last->next) {
    ListNode *pivot_ancestor = partition(head, prev, last);
    ListNode *pivot = (pivot_ancestor == NULL) ? head : pivot_ancestor->next;
    _quicksort(head, prev, pivot_ancestor);
    _quicksort(pivot->next, pivot, last);
  }
}
```

```
void quicksort(ListNode *head) {
  ListNode *last = get_last(head);
  _quicksort(head, NULL, last);
}
```

List Remove Duplicates Sorted

Given a sorted linked list, delete all duplicates such that each element appear only once.

Example:
Given 1->1->2, return 1->2.
Given 1->1->2->3->3, return 1->2->3.

Time complexity: O(N), where N is the size of the list.

Explanation: Walk over the list. At each iteration keep the previous value. If the previous value is different from the current value keep the current value.

```
ListNode* deleteDuplicates(ListNode* a) {
  if (a == NULL) return a;

  ListNode *head = a, *curr = a;
  int prev = curr->val;

  a = a->next;

  while (a != NULL) {
    if (prev != a->val) {
      curr->next = a;
      curr = curr->next;
    }
    prev = a->val;
    a = a->next;
  }

  curr->next = NULL;

  return head;
}
```

List Remove Nth Node

Given a linked list, remove the nth node from the end of list and return its head.

Example:
Given linked list: 1->2->3->4->5, and n = 2.
The linked list becomes 1->2->3->5.

Time complexity: O(N), where N is the size of the list.

Explanation: Walk over the list and count its size. Then subtract from the size the value n, call the result c. Then, walk c nodes and remove this node.

Another approach, use two pointers pointing to the beginning of the list, call them i and j. Walk with i n nodes. Then, in another loop, walk with i and j simultaneously until i reach the end of the list. Remove the node where j stopped.

```
ListNode* removeNthFromEnd(ListNode* a, int b) {
  if (a == NULL) return a;

  ListNode **curr = &a;
  int size = 0, c;

  while (*curr != NULL) {
    size++;
    curr = &((*curr)->next);
  }

  curr = &a;
  c = max(0, size - b);

  while(c) {
    curr = &((*curr)->next);
    c--;
  }

  *curr = (*curr)->next;

  return a;
}
```

List Reverse

Reverse a linked list from position m to n. Do it in-place and in one-pass.

Example:
Given 1->2->3->4->5->NULL, m = 2 and n = 4,
return 1->4->3->2->5->NULL.

Time complexity: O(N), where N is the size of the list.

Explanation: Take as an example the list: 1->2->3->4->5->NULL. The reversed list can be seen as: NULL<-1<-2<-3<-4<-5. So we can use a strategy of keeping variables to the previous node, current node, and next node. Then at each iteration we make the current node point to the previous node. Now, we just need to adapt this to reverse a part of the list instead of the whole list. We just need to walk in the list until we find the first node of our partial list and then we reverse until we are at the last of our partial list. Notice that the previous node of the first node of our partial list will need to have its "next" pointer pointing to the correct node (that is the last node of our partial list).

```
ListNode *reverseBetween(ListNode *a, int m, int n) {
  if (!a) return a;

  ListNode *prev_node = NULL, *curr_node = a, *next_node = NULL;
  ListNode **before_start = &a;
  int count = 1;

  while (count < m) {
    count++;
    before_start = &(curr_node->next);
    curr_node = curr_node->next;
  }
```

```
    while (count <= n) {
      count++;
      next_node = curr_node->next;
      curr_node->next = prev_node;
      prev_node = curr_node;
      curr_node = next_node;
    }

    (*before_start)->next = next_node;
    *before_start = prev_node;

    return a;
  }

  // ListNode ** _rev(ListNode *c, ListNode **s, ListNode **e, int x, int n) {
  //   ListNode **next_p = NULL;

  //   if (x == n) {
  //     *s = c;
  //     *e = c->next;
  //     cout << "c: " << c->val << '\n';
  //   }
  //   else {
  //     next_p = _rev(c->next, s, e, x + 1, n);
  //     *next_p = c;
  //     cout << "c: " << c->val << '\n';
  //     cout << "next_p: " << (*next_p)->val << '\n';
  //   }

  //   return &(c->next);
  // }

  // ListNode *reverseBetween(ListNode *a, int m, int n) {
  //   int x = 1;
  //   ListNode *c = a, *s = NULL, *e = NULL;

  //   while (x < m - 1) {
  //     c = c->next;
  //     x++;
  //   }

  //   ListNode *first = c->next;

  //   cout << "x: " << x << " c: " << c->val << '\n';

  //   _rev(c->next, &s, &e, x + 1, n);
  //   c->next = s;
  //   first->next = e;

  //   cout << "s: " << s->val << " e: " << e->val << '\n';
  //   cout << "first: " << first->val << '\n';

  //   return a;
  // }
```

List Reverse Recursively

Reverse a linked list with recursion.

Example:
Given 1->2->3->4->5->NULL,
return 5->4->3->2->1->NULL.

Time complexity: O(N), where N is the size of the list.

Explanation: My strategy is to return a pointer to the "next" pointer of each node.
I stop the recursion when the "next" pointer of my current node is NULL, or in other words when the current node is the last node. Then, I start to return the "next" pointer. And I make the "next" pointer point to the current node. But there is a problem: the head needs to point to the last node of the list, so I also pass a pointer to a pointer that will point to the new head and in the base case (when we are at the last node) I make this pointer point to the last node.

```
ListNode **_rev(ListNode *a, ListNode **h) {
  if (a->next == NULL) {
    *h = a;
    return &(a->next);
  }
  else {
    ListNode **pn = _rev(a->next, h);
    *pn = a;
    return &(a->next);
  }
}

ListNode *reverseList(ListNode *a) {
  if (a == NULL) return a;
  else {
    ListNode *h = NULL;
    ListNode **pn = _rev(a, &h);
    *pn = NULL;
    return h;
  }
}
```

List Sum Numbers Defined With List

You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

Example:
Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)
Output: 7 -> 0 -> 8

Time complexity: O(N), where N is the size of the largest list.

Explanation: just add the numbers and keep a "carry" variable. Notice, that the result may be longer than the lists you are working with so you need to be sure that carry is zero before exiting the loop. Also, one of the lists can be larger than the other so you will use an OR and not an AND in the loop condition. Return the head of the new list.

```
ListNode *addTwoNumbers(ListNode *l1, ListNode *l2) {
  if(!l1) return l2;
  if(!l2) return l1;

  int carry = (l1->val + l2->val) / 10;
  ListNode *l3 = new ListNode((l1->val + l2->val) % 10);
  ListNode *tail = l3;
  l1 = l1->next;
  l2 = l2->next;
```

```
    while(l1 || l2 || carry) {
      int sum = ((l1 ? l1->val : 0) + (l2 ? l2->val : 0) + carry);
      ListNode *t  = new ListNode(sum % 10);
      carry = sum / 10;

      if(l1) l1 = l1->next;
      if(l2) l2 = l2->next;
      tail->next = t;
      tail = t;
    }

    return l3;
  }

  /*
   * In the following solution I reused the nodes of one of the lists. The
   * exercise actually wants a new list as the result.
   */
  // ListNode* addTwoNumbers(ListNode* a, ListNode* b) {
  //   ListNode **i = &a, **j = &b;
  //   int c = 0, p;

  //   while (*i != NULL && *j != NULL) {
  //     p = (*i)->val;
  //     (*i)->val = ((*i)->val + (*j)->val + c) % 10;
  //     c = (p + (*j)->val + c) / 10;
  //     i = &((*i)->next);
  //     j = &((*j)->next);
  //   }

  //   if (*i == NULL && *j != NULL) *i = *j;

  //   while (*i != NULL) {
  //     p = (*i)->val;
  //     (*i)->val = ((*i)->val + c) % 10;
  //     c = (p + c) / 10;
  //     i = &((*i)->next);
  //   }

  //   if (c > 0) {
  //     *i = b;
  //     (*i)->val = c;
  //     (*i)->next = NULL;
  //   }

  //   return a;
  // }
```

List Swap Nodes In Pairs

Given a linked list, swap every two adjacent nodes and return its head. Your algorithm should use only constant space. You may not modify the values in the list, only nodes itself can be changed.

For example,
Given 1->2->3->4, you should return the list as 2->1->4->3.

Time complexity: O(N), where N is the size of the list.
Explanation: Keep a pointer to the "next" pointer of the second node of a pair of nodes, call it c. At each iteration we swap the "next" pointers of our pair and also the content of the "next" pointer we are pointing with c.

c (so *c = address of the first node)

```
|
h -> next -> next -> next -> next
      1       2       3       4

              c (so *c = address pointed by the
              |  "next" pointer of the second node)
              |
h -> next -> next -> next -> next
      1       2       3       4
```

```
// Obs: You can use a dummy node for the head to get rid of the special case.
ListNode* swapPairs(ListNode* a) {
  if (!a) return a;

  ListNode **c = &a;

  while (*c != NULL && (*c)->next != NULL) {
    ListNode *l = *c, *r = (*c)->next, *t = NULL;

    *c = r;
    l->next = r->next;
    r->next = l;

    c = &((*c)->next->next);
  }

  return a;
}
```

List To Binary Tree

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

A height balanced BST: a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

Example:

Given A = 1 -> 2 -> 3
Return a height balanced BST

```
    2
  /   \
 1     3
```

Time complexity: O(N), where N is the number of nodes in the given list.
Space complexity: O(N).

Explanation: the inorder traversal of a binary tree gives the values in ascending order. The list is sorted so if we perform a inorder traversal of the tree before it exists and then construct the tree from bottom to top we get our answer.
To perform the inorder traversal we need to know how many nodes we have in the list and then we know that since the tree must be balanced that there will be N / 2 nodes to the left and N / 2 nodes to the right, where N is the number of nodes in the given list. So, if use a variable to control if ew are at the leaf level we can start to return and build our tree getting one node at a time using our inorder traversal.

```
TreeNode* constructTree(ListNode **curr, int s, int e) {
  if (s == 0 && e == 1) {
    TreeNode *n = new TreeNode(-1);
    n->val = (*curr)->val;
    *curr = (*curr)->next;
    return n;
  }
  if (e - s <= 1) return NULL;

  int m = (s + e) / 2;
  TreeNode *n = new TreeNode(-1);

  n->left = constructTree(curr, s, m);

  n->val = (*curr)->val;
  *curr = (*curr)->next;

  n->right = constructTree(curr, m, e);

  return n;
}

TreeNode* sortedListToBST(ListNode *head) {
  int n = 0;
  ListNode *curr = head;

  while (curr != NULL) {
    n++;
    curr = curr->next;
  }

  curr = head;

  return constructTree(&curr, 0, n);
}
```

Lru Cache

```
Design and implement a data structure for Least Recently Used (LRU) cache.
It should support the following operations: get and set.

get(key) - Get the value (will always be positive) of the key if the key
exists in the cache, otherwise return -1.
set(key, value) - Set or insert the value if the key is not already present.
When the cache reaches its capacity, it should invalidate the least recently
used item before inserting the new item.

The LRUCache will be initialized with an integer corresponding to its
capacity. Capacity indicates the maximum number of unique keys it can hold at
a time.
```

Example:

```
capacity = 2
set(1, 10)
set(5, 12)
get(5)        returns 12
get(1)        returns 10
get(10)       returns -1
set(6, 14)    this pushes out key = 5 as LRU is full.
get(5)        returns -1
```

Time complexity: O(1) for both operations "set" and "get".

Explanation: use a hashmap and a doubly linked list. The hashmap will allow us to get the elements in O(1), but the hashmap alone would not allow us to set a value in O(1), because we would need to look for the LRU element. If we use a doubly linked list to store a values we can easily reorder our elements in constant time each time we set or we get a value. The hashmap then needs to store the pointer to the linked list node instead of the value itself. The reorder of our linked list is just putting the just used element in the front of the list for a LRU cache. We could use the same approach for a LFU cache and just the reorder would change.

```
class LRUCache {
private:
  int c;
  list<pair<int, int>> l;
  unordered_map<int, list<pair<int, int>>::iterator> m;

public:
  LRUCache(int capacity) {
    c = capacity;
    print();
  }

  int get(int key) {
    unordered_map<int, list<pair<int, int>>::iterator>::iterator mit;
    mit = m.find(key);

    if (mit == m.end()) return -1;

    list<pair<int, int>>::iterator lit = mit->second;
    l.push_front(make_pair(key, mit->second->second));
    l.erase(lit);
    mit->second = l.begin();
    print();

    return mit->second->second;
  }

  void set(int key, int value) {
    unordered_map<int, list<pair<int, int>>::iterator>::iterator mit;
    mit = m.find(key);

    if (mit != m.end()) {
      list<pair<int, int>>::iterator lit = mit->second;
      l.erase(lit);
      l.push_front(make_pair(key, value));
      mit->second = l.begin();
    }
    else {
      if (l.size() == c && l.size() != 0) {
        list<pair<int, int>>::iterator eit = l.end();
        --eit;
        m.erase(eit->first);
        l.erase(eit);
      }
      l.push_front(make_pair(key, value));
      list<pair<int, int>>::iterator fit = l.begin();
      m[key] = fit;
    }
    print();
  }

  void print() {
    list<pair<int, int>>::iterator it = l.begin();

    cout << "List: ";
    for (; it != l.end(); it++) {
      cout << it->first << ":" << it->second << " ";
    }
    cout << endl;

    cout << "Map size: " << m.size() << endl;
  }
};
```

Math All Factors

Given a number N, find all factors of N.

Example:

N = 6
factors = {1, 2, 3, 6}
Make sure the returned array is sorted.

Time complexity: O(sqrt(N)), where N is the given input.

Explanation: the algorithm is based on the fact that you can find all the factors of a number N looking at its possible divisors up to sqrt(N). The other factors will then be the result of the division of N by these factors up to sqrt(N). For example,

N = 6

sqrt(6) = 2
factors up to 2 = 1, 2
remaining factors = 6 / 2, 6 / 1 = 3, 6
result = 1, 2, 3, 6

Just don't forget that when the number has a whole square root that you need to be sure to not include the same factor twice with the above process.

```
vector<int> allFactors(int a) {
  vector<int> ans;

  if (a < 2) {
    ans.push_back(a);
    return ans;
  }

  int r = (int)sqrt(a);

  for (int i = 1; i <= r; i++) {
    if (a % i == 0) ans.push_back(i);
  }

  for (int i = ans.size() - 1; i >= 0; i--) {
    if (a != ans[i] * ans[i]) ans.push_back(a / ans[i]);
  }

  return ans;
}
```

Math Excel Column

Given a column title as appears in an Excel sheet, return its corresponding column number.

Example:

```
A -> 1
B -> 2
C -> 3
...
Z -> 26
AA -> 27
AB -> 28
```

Time complexity: O(logN), where N is the given number and log is base 26.
Space complexity: O(1).

Explanation: base 26, transform each letter to its respective number and multiply by the correct base, then add the current result to the answer.
Since it is base 26, bases will be = 26^0, 26^1, 26^2, and so on.

```
int convertToNumber(string a) {
  int ans = 0, base = 1;

  for (int i = a.size() - 1; i >= 0; i--) {
    ans += (a[i] - 'A' + 1) * base;
    base *= 26;
  }

  return ans;
}
```

Math Excel Title

Given a positive integer, return its corresponding column title as appear in an Excel sheet.

Example:

```
1 -> A
2 -> B
3 -> C
...
26 -> Z
27 -> AA
28 -> AB
...
52 -> AZ
...
78 -> BZ
```

Time complexity: O(logN), where N is the given number and log is base 26.
Space complexity: O(1).

Explanation: this is base 26, but with one difference, that is our number do not start at 0, but at 1. Our 0th symbol is A and our 25th symbol is Z, so we need to subtract 1 from our current input (1 becomes 0, which is A, 26 becomes 25 which is Z, 52 becomes AZ, and so on).
Notice that using the ASCII values to get the letters is better than using a dicitionary.

```
string convertToTitle(int n) {
    string ans = "";

    while (n > 0) {
      ans.pus_back((char)((n - 1) % 26 + 'A'));
      n = (n - 1) / 26;
    }

    reverse(ans.begin(), ans.end());

    return ans;
  }
```

Math Factorial Trailling Zeros

Given an integer N, return the number of trailing zeros in n!.

Notes:
1. Your solution should be in logarithmic time complexity.

Example:
N = 5
N! = 120
Number of trailing zeros = 1, so, return 1

Time complexity: O(logN), where is the given number and log is base 5.
Space complexity: O(1).

Explanation: factorial of N is the multiplication of all numbers from 1 to N. So, to add a zero to the answer we need to multiply by 10 and number 10's come from the multiplication of an even number multiplied by 5. We don't need to worry about even numbers because there will always be plenty of them to multiply our numbers 5, so what we need to do is to count the number of 5's is the multiplication. For example, for N = 100 we have:

100 / 5 = 20, so there are 20 multiples of 5 from 1 to 100.
And for this 20 multiples there are 20 / 5 = 4 another multiples of 5.
So, we get 24 zeros.

If we expand the multiplication we can see easily see the above fact:
1 * 2 * ... * 10 * 11 * ... 25 ... 100
1 * 2 * ... * (2 * 5) * 11 * ... (5 * 5) ... 100

```
  int trailingZeroes(int a) {
    int ans = 0;

    while (a >= 5) {
      a /= 5;
      ans += a;
    }

    return ans;
  }
```

Math Find Primes Up To N

Find all primes up to a number N.

Example:
Given N = 10, return [2, 3, 5, 7]

Time complexity: O(N * log(log(N))), where N is the size of the array. The tricky part is to find a sum of primes (2 + 3 + 5 + ...). I believe this is the complexity because the algorithm is similar to Sieve of Eratosthenes.
Space complexity: O(N).

Explanation: keep a list of primes. Iterate from 2 .. N and keep a list of the primes found so far. For each number i try to divide i by all the primes that you already found. One optimization is:
If we have primes = [2, ..., Xi, ...], j = Xi and we are checking a number N bigger than Xi, we already know in our algorithm that N is not multiple of any prime from 2 to Xi-1. So:
  1. If Xi is a factor of N, so N / Xi >= Xi.
  2. If Xi * Xi > N then N / Xi < Xi, so the above condition is not true, hence N is prime (because any other prime larger than Xi should respect the above condition which will not, since Xk > Xi will give a smaller result for the division).

Observation: you can make Sieve of Eratosthenes in O(N) time complexity using more memory. The main idea is to cross each number just once keeping a list of the smallest prime factor of each number up to N.

```
vector<int> findPrimes(n) {
  vector<int> primes;

  for (int i = 2; i <= n; i++) {
    for (int j = 0; j < primes.size(); j++) {
      if (primes[j] * primes[j] > i) {
        primes.push_back(i);
        break;
      }

      if (i % primes[j] == 0) break;
    }

    if (j == primes.size()) primes.push(i);
  }

  return primes;
}

// Sieve of Eratosthenes
vector<int> sieve(n) {
  vector<bool> isPrime(n + 1, true);
  isPrime[0] = false;
  isPrime[1] = false;

  for (int i = 2; i <= n; i++) {
    if (!isPrime[i]) continue;
    if (i > n / i) break;
    for (int j = 2 * i; j <= n; j += i) isPrime[j] = false;
  }

  vector<int> primes;
  for (int i = 2; i <= n ; i++) {
    if (isPrime[i]) primes.push_back(i);
  }

  return primes
}
```

Math Gcd

Given 2 non negative integers m and n, find gcd(m, n)

GCD of 2 integers m and n is defined as the greatest integer g such that g is a divisor of both m and n. Both m and n fit in a 32 bit signed integer.

Example:
m = 6
n = 9
Return 3.

Time complexity: O(M + N), where M and N are the inputs.
Space complexity: O(1), if not using recursion.

Explanation: Euclidian's Algorithm. Keep doing the mod operation until the smaller value is 0. For example,

1112 mod 695 = 417
695 mod 417 = 278
417 mod 278 = 139
278 mod 139 = 0
So we return 139.

```
int gcd(int m, int n) {
  if (m < n) {
    int temp = m;
    m = n;
    n = temp;
  }

  if (n == 0) return m;
  return gcd(m % n, n);
}
```

Math Modular Exponentiation

Implement pow(a, b) % c recursively.

Example:
Input : a = 2, b = 3, c = 3
Return : 2

Time complexity: O(log n), where n is equal to the exponent.
Explanation: The entire problem is based on the fact that:
(x * y) % m = ((x % m) * (y % m)) % m
So, we can think in our exponention in terms of even and odd exponents.
If the exponent is even we have:
(a^(b / 2) % m) * (a^(b / 2) % m)
If the exponent is odd we have:
(a % m) * (a^(b - 1) % m)
And our base case is the exponent equal 0, in such case we return 1 % m.
Obs: notice, the % operator in C/C++ is actually the reminder operator and not the modulo operator. They differ when applied to negative numbers. So, the reminder operator can be turned into the modulo operator like this:
r = x % y, r < 0 ? r + y : r

```
long long int _mod(long long int a, int b) {
  long long int r = a % b;
  return r < 0 ? r + b : r;
}

int Mod(int a, int b, int c) {
  if (b == 0) return _mod(1, c);
  else if ((b % 2) == 0) {
    long long int y = Mod(a, b / 2, c);
    return _mod((y * y), c);
  }
  else {
    return _mod((_mod(a, c) * Mod(a, b - 1, c)), c);
  }
}

// long long int Mod(long long int x, long long int y, long long int p) {
//   long long int res = 1; // Initialize result

//   x = x % p; // Update x if it is more than or equal to p

//   while (y > 0) {
//     // If y is odd, multiply x with result
//     if (y & 1) res = _mod((res*x), p);

//     // y must be even now
//     y = y>>1; // y = y/2
//     x = _mod((x*x), p);
//   }

//   return res;
// }
```

Math Newton Square Root

Newton method applied to calculate square roots.

```
int c = 0;

double nSqrt(double a, double guess) {
  if (a == 0) return 0;
  if (a == 1) return 1;

  double ans = guess, prevAns;
  double threshold = 0.001;

  do {
    c++;
    prevAns = ans;
    ans = 0.5 * (prevAns + (a / prevAns));
  } while (abs(ans - prevAns) > threshold);

  return ans;
}
```

Math Numbers Length N Value Less K

Given a set of digits (A) in sorted order, find how many numbers of length B are possible whose value is less than number C.

Notes:
1. All numbers can only have digits from the given set.

Constraints: 1 <= B <= 9, 0 <= C <= 1e9 & 0 <= A[i] <= 9

Example:
```
 Input:
   3 0 1 5
   1
   2
 Output:
   2 (0 and 1 are possible)

 Input:
   4 0 1 2 5
   2
   21
 Output:
   5 (10, 11, 12, 15, 20 are possible)
```

Time complexity: O(B), where B is the maximum number of digits. The size of the array can be considered constant since will always be at most 10.
Space complexity: O(B), but the dynamic programming approach can be optimized to O(1).

Explanation: there are three cases:

1. B is larger than the number of digits in C. In such case we return 0.

2. B is smaller than the number of digits in C. In such case we return n^b if we don't have a zero in the array, otherwise if B is larger than 1 then (n - 1) * n^(b - 1) because our possible numbers can't start with zero. n is the size of the array A.

3. B is equal to the number of digits in C. In such case we need to count the leaf nodes of our tree if possibilities. The tricky part is how to do that without making mistakes.

A tree looks like:
A = [1, 2]
B = 2
C = 22

```
                     +----------+
          +----------+Empty root+---------+
          |          +----------+         |
          |                               |
          |                               |
       +--v--+                         +--v--+
       |     |                         |     |
   +---+  1  +--+                  +---+  2  |
   |   |     |  |                  |   |     |
   |   +-----+  |                  |   +-----+
   |            |                  |
+--v--+      +--v--+            +--v--+
|     |      |     |            |     |
|  1  |      |  2  |            |  1  |
|     |      |     |            |     |
+-----+      +-----+            +-----+
```

Let's define some things:

1. First(i) is the number formed by the first i digits of C (where the first digit is the one at the largest power (the one at the left)).

2. digit[i] is an array with the digit at position i of C (again, at position 1 we have the digit with largest power of C).

3. lower[i] is an array where each element is the number of elements in array A smaller than i. We fill this array for all the possible digits, so from 0 to 9.

4. DP[i] is an array where each element denotes the total numbers of length i which are less than first i digits of C. Initialized with zeros.

Now, notice that DP[0] is always 0, and that we need to take care to not take into account for DP[1] the number zero when B is larger than 1, because number with more than one digit can't start with 0.

DP[i] can be filled checking two cases:

1. For all the Numbers whose First(i − 1) is less than First(i − 1) of C, we can put any digit at i'th index. Hence, DP[i] += (DP[i − 1] ∗ n). Where n is the size of A.

2. For all the Numbers whose First(i − 1) is same as First(i − 1) of C, we can only put those digits which are smaller than digit[i]. Hence, DP[i] += lower[digit[i]].

Case 1 we execute for every DP[i], but case 2 we just add if using the numbers in the array we are able to make First(i − 1) of C.

Notice, it is really easy to make this solution use O(1) space because we just need the previous answer DP[i − 1] to make the next DP[i].

```cpp
int solve(vector<int> &a, int b, int c) {
  if (a.size() == 0 || b <= 0 || c <= 0) return 0;

  int ans = 0, n = a.size(), len = 0;
  vector<int> digits;

  while (c > 0) {
    digits.insert(digits.begin(), c % 10);
    c /= 10;
    len++;
  }

  if (b > len) {
    return 0;
  }
  else if (b < len) {
    return a[0] == 0 && b > 1 ? (n - 1) * pow(n, b - 1) : pow(n, b);
  }
  else {
    int partialC = 0, partialV = 0;
    vector<int> dp(b + 1, 0), lower(10, 0);

    for (int i = 1, j = 0; i < lower.size(); i++) {
      while (j < a.size() && a[j] < i) j++;
      lower[i] = j;
    }

    for (int i = 1; i < dp.size(); i++) {
      dp[i] += (dp[i - 1] * n);
      if (partialV == partialC) dp[i] += lower[digits[i - 1]];
      if (i == 1 && a[0] == 0 && b > 1) dp[i] = max(dp[i] - 1, 0);

      partialC = partialC * 10 + digits[i - 1];
      for (int j = 0; j < a.size(); j++) {
        if (a[j] == digits[i - 1]) partialV = partialV * 10 + a[j];
      }
    }

    return dp[b];
  }
  return ans;
}

// My original solution fixed and working. It is basically the same thing as
// the editorial solution.
// int solve(vector<int> &a, int b, int c) {
//   if (a.size() == 0 || b <= 0 || c <= 0) return 0;

//   int ans = 0, n = a.size(), len = 0, rev = 0;

//   while (c > 0) {
//     len++;
//     rev = (rev * 10) + (c % 10);
//     c /= 10;
//   }

//   if (b > len) {
//     ans = 0;
//   }
//   else if (b < len) {
//     ans = a[0] == 0 && b > 1 ? (n - 1) * pow(n, b - 1) : pow(n, b);
//   }
//   else {
//     int partialC = 0, partialV = 0;

//     for (int i = 0; i < len; i++) {
//       int digit = rev % 10;
//       rev /= 10;

//       ans = ans * n;

//       int firstSmall = 0;
//       for (int j = 0; j < a.size() && a[j] < digit; j++) firstSmall++;

//       if (partialV == partialC) ans += firstSmall;

//       partialC = partialC * 10 + digit;
//       for (int j = 0; j < a.size(); j++) {
//         if (a[j] == digit) partialV = partialV * 10 + a[j];
//       }

//       if (i == 0 && a[0] == 0 && b > 1) ans = max(0, ans - 1);
//     }
//   }

//   return ans;
// }
```

Math Palindrome Integer

Determine whether an integer is a palindrome. Do this without extra space.
A palindrome integer is an integer x for which reverse(x) = x where
reverse(x) is x with its digit reversed. Negative numbers are not
palindromic.

Example:

```
Given 12121
Output true

Given 123
Output false
```

Time complexity: O(N), where N is the number of digits the input has.
Space complexity: O(1).

Explanation: reserse the number and take care to not overflow. If the number overflows or if the number is negative, so it is not a palindrome.

```
int reverse(int a) {
  int r = 0;

  while (a > 0) {
    int digit = a % 10;

    if (r > INT_MAX / 10 || r * 10 > INT_MAX - digit) return -1;

    r = r * 10 + digit;
    a /= 10;
  }

  return r;
}

bool isPalindrome(int a) {
  if (a < 0) return false;
  return reverse(a) == a;
}
```

Math Pow Mod

Implement pow(x, n) % d. In other words, given x, n and d, find (x^n % d)

Note that remainders on division cannot be negative so make sure the answer you return is non negative.

Example:

Given, x = 2, n = 3, d = 3
Output 2, because 2^3 % 3 = 8 % 3 = 2.

Time complexity: O(logN), where N is the exponent.
Space complexity: O(1).

Explanation: to understand this problem you need to think about the binary form of the exponent. For example, the binary form of 200 is: 11001000, which is 128 + 64 + 8. So, if our base is 2 (it could be any other integer), then 2^200 = 2^128 * 2^64 * 2^8
Now, notice that
2^2 = 2 * 2 = 4
2^4 = 4 * 4 = 16
2^8 = 16 * 16 = 256
So, we can reuse the previous result to get the next result when our exponent is a power of 2 (2, 4, 8, ...).

To solve this exercise we just want to look at the binary form of our exponent and check which bits are 1. When the bit is 0 we just update a variable "curr" to be our base to the power of the current binary position. When the bit is 1 it means we need to include in our answer "curr", so we multiply "curr" by the answer we have so far.

In our example, we would eventually get 2^8 and we would check that the bit for 8 is 1 so we would include in our answer 2^8. After some more iterations we would have 2^64 and we would check that the bit for 64 is 1 in our exponent so we would include 2^64 in our answer that currently is 2^8, so we

```
would get
ans = 2^8 * 2^64 =>
ans = ans * curr = 2^72
We would do the same thing for 2^128 and finally get
ans = 2^72 * 2^128 =>
ans = ans * curr = 2^200

Of course, this algorithm works for any base and any positive exponent.
```

```
int pow(int x, int n, int d) {
  int base = x, ans = 1;

  while (n > 0) {
    // Check if the last bit is 1, which means we need to include in our answer
    // the base^(current binary power).
    if (n & 1) {
      ans = (ans * base) % d;
      n--;
    }
    // If it is not 1 we just update our base to the next binary power
    // base^(binary power), which is current base * current base.
    else {
      base = (base * base) % d;
      n = n >> 1;
    }
  }

  if (ans < 0) return (ans + d) % d;
  return ans;
}
```

Math Prime Sum

Given an even number (greater than 2), return two prime numbers whose sum will be equal to given number.

Note 1: A solution will always exist. read Goldbach's conjecture
Note 2: If there are more than one solutions possible, return the lexicographically smaller solution.

Example:
Given 4, return [2, 2], since 2 + 2 = 4.

Time complexity: O(N * time complexity of isPrime()). If you use Sieve of Eratosthenes will be O(N * log(log(N))). If you use a simple method will be O(N * sqrt(N)).
Space complexity: O(N), if you use Sieve of Eratosthenes, O(1) if you use the common method.

Explanation: iterate from 2 to N. Check if the number i is prime and if it is check if (N - i) is also prime, and if it is return [i, N - i].

Observation: Check find-prime-numbers-up-to-n problem to see Sieve of Eratosthenes.

```
int isPrime(int n) {
  if (n < 2) return 0;

  int upperLimit = (int)(sqrt(n));
  for (int i = 2; i <= upperLimit; i++) {
    if (i < n && n % i == 0) return 0;
  }

  return 1;
}

vector<int> primesum(int a) {
  vector<int> ans;

  for (int i = 2; i <= a; i++) {
    if (isPrime(i) && isPrime(a - i)) {
      ans.push_back(i);
      ans.push_back(a - i);
      return ans;
    }
  }

  return ans;
}
```

Math Reverse Integer

Reverse digits of an integer.

Example:
x = -123,
return -321

Return 0 if the result overflows and does not fit in a 32 bit signed integer.

Time complexity: O(N), where N is the number of digits in the number.

Explanation: check the sign and store it is a variable (1 positive, -1 negative). Reverse the number digit by digit using mod and division. At each iteration you multiply your current number by 10 and add the mod by 10 of your number, then divide the number by 10 and keep doing it until your number is 0. The overflow checking is easy. If INT_MAX / 10 is bigger than your current number you know the number will overflow in the next iteration. Or if current_value * 10 is bigger than INT_MAX - a % 10 you also know the next iteration will overflow.

```
  int reverse(int a) {
    if (a == 0) return 0;

    int sign = a < 0 ? -1 : 1;
    int b = abs(a), ans = 0;

    while (b != 0) {
      if (ans > INT_MAX / 10 || ans * 10 > INT_MAX - (b % 10)) return 0;

      ans = (ans * 10) + (b % 10);
      b /= 10;
    }

    return ans * sign;
  }
```

Math Verify Prime

```
int isPrime(int n) {
  if (n < 2) return 0;

  int upperLimit = (int)(sqrt(n));
  for (int i = 2; i <= upperLimit; i++) {
    if (i < n && n % i == 0) return 0;
  }

  return 1;
}
```

Math Write As Power

Given a positive integer which fits in a 32 bit signed integer, find if it can be expressed as A^P where P > 1 and A > 0. A and P both should be integers.

Example:

Given 4
Output True, because 2^2 = 4.

Time complexity: O(NlogN), where N is the given number. I believe it is NlogN because we iterate over all possible bases from 2..N - 1 and for all these bases we multiply them until we possibly get the result. The number of times we will perform multiplications for a possible base will be log for that base of N.
Space complexity: O(1).

Explanation: Iterate through the all numbers from 2..N - 1. For each of these numbers multiply it by itself until you get N or a number larger than N.

```
bool isPower(int n) {
  if (n == 0) return false;
  if (n == 1) return true;

  for (int base = 2; base < n; base++) {
    int curr = 1;

    while (curr < n) {
      curr *= base;
      if (curr == n) return true;
    }
  }

  return false;
}
```

Merge K Sorted Array

Merge k sorted linked lists and return it as one sorted list.

Example:
1 -> 10 -> 20
4 -> 11 -> 13
3 -> 8 -> 9

```
Will result in:
1 -> 3 -> 4 -> 8 -> 9 -> 10 -> 11 -> 13 -> 20
```

Time complexity: O(N*logM), where N is the total numbers of nodes and M is the number of lists.

Explanation: organize the head nodes of the list in a min heap (priority_queue with custom comparator in C++) and then start to create a new list removing the root node and inserting the next node of the head you just removed in the heap. Do that until you have no more nodes.

```cpp
struct CompareNode {
  bool operator()(ListNode* const &p1, ListNode* const &p2) {
    return p1->val > p2->val;
  }
};

ListNode* mergeKLists(vector<ListNode*> &a) {
  ListNode *dummy = new ListNode(0);
  ListNode *tail = dummy;
  priority_queue<ListNode*, vector<ListNode*>, CompareNode> q;

  for (int i = 0; i < a.size(); i++) {
    if (a[i] != NULL) q.push(a[i]);
  }

  while (!q.empty()) {
    tail->next = q.top();
    q.pop();
    tail = tail->next;

    if (tail->next) {
      q.push(tail->next);
    }
  }

  return dummy->next;
}

ListNode* get_last(ListNode *head) {
  ListNode *curr = head;

  while (curr->next != NULL) {
    curr = curr->next;
  }

  return curr;
}

void append(ListNode *head, int val) {
  ListNode *last = get_last(head);
  ListNode *node = new ListNode(val);

  last->next = node;
}

int remove_node(ListNode **head, ListNode* node) {
  ListNode **curr = head;

  while (*curr != node && *curr != NULL) {
    curr = &((*curr)->next);
  }

  if (*curr == node) {
    *curr = node->next;
    free(node);
    return 1;
  }

  return 0;
}

void free_list(ListNode *head) {
  ListNode *curr = head, *tmp = NULL;

  while (curr != NULL) {
    tmp = curr;
    curr = curr->next;
    free(tmp);
  }
}

void print_node(ListNode *node, char separator) {
  cout << node->val << separator;
}

void print_list(ListNode *head) {
  ListNode *curr = head;

  while (curr != NULL) {
    print_node(curr, ' ');
    curr = curr->next;
  }
  cout << endl;
}
```

Merge Sorted Array

Given two sorted integer arrays A and B, merge B into A as one sorted array.
You must modify array A to be the result.

Example:
Input : A : [1 5 8] B : [6 9]
Modified A : [1 5 6 8 9]

Time complexity: O(N), where N is the length of array B.
Explanation: keep two pointers, one at the beginning of the first array, call it i, another one at the beginning of the second array, call it j. Iterate through the second array with j, if a[i] < b[j] increment i, else (greater or equal) insert the element b[j] at position i and then increment i and j.

```
  void merge(vector<int> &a, vector<int> &b) {
    int i = 0, j = 0;

    while (j < b.size()) {
      if (a[i] < b[j] && i < a.size()) i++;
      else {
        a.insert(a.begin() + i, b[j]);
        i++;
        j++;
      }
    }
  }
```

Merge Two Sorted Lists

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists, and should also be sorted.

Example:
5 -> 8 -> 20
4 -> 11 -> 15
Merge list: 4 -> 5 -> 8 -> 11 -> 15 -> 20

Time complexity: O(N), where N is the size of the longest list.
Explanation: keep two pointers at the beginning of the lists, call them i and j. When j->val < i->val add j to the list and go forward with j.
When i->val <= j->val add i to the list and go forward with i.

```
ListNode* mergeTwoLists(ListNode* a, ListNode* b) {
  ListNode *i = a, *j = b, *curr = NULL, *head = NULL;

  // Add the head
  if (a == NULL) return b;
  if (b == NULL) return a;

  if (i->val < j->val) {
    head = i;
    i = i->next;
  }
  else {
    head = j;
    j = j->next;
  }

  curr = head;

  // Merge the elements
  while (i != NULL && j != NULL) {
    if (j->val < i->val) {
      curr->next = j;
      j = j->next;
    }
    else if (i->val <= j->val) {
      curr->next = i;
      i = i->next;
    }
    curr = curr->next;
  }

  // Add the tail
  if (i) curr->next = i;
  else curr->next = j;

  return head;
}

ListNode* create_node(int val) {
  ListNode *node = (ListNode *)malloc(sizeof(ListNode));

  node->val = val;
  node->next = NULL;

  return node;
}

ListNode* get_last(ListNode *head) {
  ListNode *curr = head;
```

```
      while (curr->next != NULL) {
        curr = curr->next;
      }

      return curr;
    }

    void append(ListNode *head, int val) {
      ListNode *last = get_last(head);
      ListNode *node = create_node(val);

      last->next = node;
    }

    void push(ListNode **head, int val) {
      ListNode *node = create_node(val);

      node->next = *head;
      *head = node;
    }

    void insert_after(ListNode *previous_node, int val) {
      ListNode *new_node = create_node(val);
      ListNode *previous_next = previous_node->next;

      previous_node->next = new_node;
      new_node->next= previous_next;
    }

    int remove_node(ListNode **head, ListNode* node) {
      ListNode **curr = head;

      while (*curr != node && *curr != NULL) {
        curr = &((*curr)->next);
      }

      if (*curr == node) {
        *curr = node->next;
        free(node);
        return 1;
      }

      return 0;
    }

    void free_list(ListNode *head) {
      ListNode *curr = head, *tmp = NULL;

      while (curr != NULL) {
        tmp = curr;
        curr = curr->next;
        free(tmp);
      }
    }

    /* --- Helper functions --- */

    ListNode* get_node(ListNode *head, int n) {
      ListNode *curr = head;
      int i;

      for (i = 0; i < n && curr != NULL; i++) {
        curr = curr->next;
      }

      if (i == n) {
        return curr;
      }
      else {
```

```
    return NULL;
  }
}

void append_nodes(ListNode *head, int n) {
  ListNode *last = get_last(head);
  ListNode *curr = last;

  for (int i = 1; i <= n; i++) {
    curr->next = create_node(i);
    curr = curr->next;
  }
}

void print_node(ListNode *node, char line_break) {
  printf("%d%c", node->val, line_break);
}

void print_list(ListNode *head) {
  ListNode *curr = head;

  while (curr != NULL) {
    print_node(curr, ' ');
    curr = curr->next;
  }
  printf("\n");
}
```

Segment Tree Order People By Height

You are given the following:
A positive number N
Heights: A list of heights of N persons standing in a queue
Infronts: A list of numbers corresponding to each person (P) that gives the number of persons who are taller than P and standing in front of P

You need to return list of actual order of persons’s height

Note: consider that heights will be unique.

Time complexity:

Explanation: what is the place for the smallest person in the queue? To find that we just need to count the number of people in front of him starting at the very beginning of the array. And for the second smallest person? Again, it is the number of people in front of him counted from the very beginning of the array, but this time we need to realize that we should just count free spots remaining in our answer. So if in our count we fall into a position where the first smallest person was placed we don't take it into account.

Based on that, we just need to sort our array of heights and do the process described above. The sorting process give us a O(logN) time complexity, where N is the number of elements in the array.

Now, can we do something to improve our code? Yes, we can. When we are counting free spots to find out where our element should be placed in our answer we take O(N) time, but if we use a segment tree we can find this spot in O(logN) time. So, that is what we do. We build a segment tree with information about free spots in our answer.

```
bool isPowerOfTwo(int n) {
  return (n & (n - 1)) == 0;
```

```
}

// Build the segment tree that contains the amount of free spaces in our
// answer vector.
void buildTree(vector<int> &tree, int treeSize, int initialValue) {
  int value = initialValue, rest;

  tree.push_back(value);

  // Build tree top down.
  for (int i = 0; i < (treeSize / 2); i++) {
    // This rest variable is here in case the number is odd because then one
    // of our branches needs to contain one more space. In this code I chose
    // the left branch to have this extra space.
    rest = (tree[i] % 2) && tree[i] > 1 ? 1 : 0;
    value = tree[i] / 2;
    tree.push_back(value + rest);
    tree.push_back(value);
  }
}

int getIndex(vector<int> &tree, int infront, int i, int s, int e) {
  // Depending on your input you would need to check if the place you are is
  // valid (there is one empty space) with "if (tree[i] == 1)".
  if (s == e) {
    tree[i] = 0;
    return s;
  }

  // In top down approach decrement the amount of free spaces. If the tree
  // becomes invalid and this function return -1 we should exit the program,
  // that is why we can go top down.
  tree[i]--;

  int mid = (s + e) / 2;

  // Check if the left branch has enough free spaces.
  if (tree[2 * i + 1] > infront) {
    return getIndex(tree, infront, 2 * i + 1, s, mid);
  }
  else {
    // When going to the right branch we need to subtract from our infront
    // value the amount of free spaces we have in the left branch.
    return getIndex(tree, infront - tree[2 * i + 1], 2 * i + 2, mid + 1, e);
  }
}

vector<int> orderPeople(vector<int> &heights, vector<int> &infronts) {
  vector<pair<int, int>> hi;
  vector<int> ans(heights.size(), -1), tree;
  int treeSize;

  // Build a vector with height and infront together (with height in the lead).
  for (int i = 0; i < heights.size(); i++) {
    hi.push_back(make_pair(heights[i], infronts[i]));
  }

  // Sort the vector by height.
  sort(hi.begin(), hi.end());

  // Our segment tree has size (nearestPowerOfTwo * 2) - 1. A segment tree is
  // always a complete binary tree, because it is stored in an array and to find
  // children of a give parent in an array we use formulas like 2i and 2i + 1.
  // If it wasn't a complete binary tree the formulas wouldn't work.
  if (!isPowerOfTwo(heights.size())) {
    int nextPowerOfTwo = pow(2, ceil(log((double)heights.size()) / log(2)));
    treeSize = (2 * nextPowerOfTwo) - 1;
  }
  else {
    treeSize = (2 * heights.size()) - 1;
  }

  buildTree(tree, treeSize, heights.size());

  for (int i = 0; i < hi.size(); i++) {
    int height = hi[i].first;
    int infront = hi[i].second;
    int idx = getIndex(tree, infront, 0, 0, hi.size() - 1);

    if (idx > -1) ans[idx] = height;
  }

  return ans;
}
```

Sort Insertion Sort

Insertition Sort Algorithm.

Time complexity: O(N^2), where N is the size of the array to be sorted.
Space complexity: O(1).

Explanation: the idea is to divide the array into two parts: a sorted part, and a unsorted part. We want to move elements one by one from the unsorted part to the sorted part.

So, for each i from 1 .. N - 1 we iterate from i to 0 (we start from the second element, index 1, because the 0 element is already sorted since it is just one element). When iterating from i..0 we are looking for a place to move the first element from the unsorted part to the sorted part.

```
void insertionSort(vector<int> &a) {
  for (int i = 1; i < a.size(); i++) {
    int element = a[i], j = i;

    // Move elements one by one to the right until we find the new position
    // for "element".
    while (j > 0 && a[j - 1] > element) {
      a[j] = a[j - 1];
      j--;
    }

    a[j] = element;
  }
}
```

Sort Merge Sort

```
 for (i = 0; i < n1; i++) L[i] = a[s + i];
 for (j = 0; j < n2; j++) R[j] = a[m + 1 + j];


  i = 0;
  j = 0;
  k = s;
  while (i < n1 && j < n2) {
      if (L[i] <= R[j]) {
        a[k] = L[i];
        i++;
      }
      else {
        a[k] = R[j];
        j++;
      }
      k++;
    }

    // Copy the remaining elements of L[], if there are any.
    while (i < n1) {
      a[k] = L[i];
      i++;
      k++;
    }

    // Copy the remaining elements of R[], if there are any.
    while (j < n2) {
      a[k] = R[j];
      j++;
      k++;
    }
  }

  void mergeSort(vector<int> &a, int s, int e) {
    if (s < e) {
      // Same as (s + e) / 2, but avoids overflow for large s and e.
      int m = s + (e - s) / 2;

      // Sort first and second halves
      mergeSort(a, s, m);
      mergeSort(a, m + 1, e);

      merge(a, s, m, e);
    }
  }
```

Sort Quick Sort

Quick Sort Algorithm.

Time complexity: O(NLogN) average case and O(N^2) worst case, where N is the size of the array to be sorted.
Space complexity: O(N) in naive approach, O(logN) Sedgewick 1978.

Explanation: the idea is that for each partition we choose a PIVOT and we want to find the final position of this pivot.

So, we want everybody to the left of this position to be smaller than pivot, and everybody to the right to be larger than the pivot. For that we will use the concept of a WALL. We use this wall to find the position of the pivot. At the end we want wall to be the position where the pivot must be placed. So, when the current element is smaller than the pivot we swap current element with the wall and move the wall to the right.

```
  void quickSortUtil(vector<int> &a, int left, int right) {
    int wall = left - 1, curr = left, pivot = right;

    if (left < right) {
```

```
    for (int curr = left; curr < right; curr++) {
      if (a[curr] <= a[pivot]) {
        wall++;
        swap(a[wall], a[curr]);
      }
    }

    wall++;
    swap(a[wall], a[pivot]);

    quickSortUtil(a, left, wall - 1);
    quickSortUtil(a, wall + 1, right);
  }
}

void quickSort(vector<int> &a) {
  quickSortUtil(a, 0, a.size() - 1);
}
```

Sort Selection Sort

Selection Sort Algorithm.

Time complexity: O(N^2), where N is the size of the array to be sorted.
Space complexity: O(1).

Explanation: for each index i (0..N - 1) we look for the minimum element after this index. Then, we swap a[i] with a[min].

```
void selectionSort(vector<int> &a) {
  for (int i = 0; i < a.size() - 1; i++) {
    int min = i;

    for (int j = i + 1; j < a.size(); i++) {
      if (a[j] < a[min]) min = j;
    }

    if (min != i) swap(a[min], a[i]);
  }
}
```

Stack Check Parentheses

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.

Return 0 / 1 (0 for false, 1 for true) for this problem.

Time complexity: O(N), where N is the length of the string.

Explanation: Just use a stack to keep the openning characters: '(', '[', '{'. When you find a closing character, pop() one character from the stack and see if they match (a '(' from the stack matches with a ')', etc). If the stack is empty and you iterated over the entire string you have a valid string.

```
int isValid(string a) {
  stack<char> s;

  for (int i = 0; i < a.length(); i++) {
    if (a[i] == '(' || a[i] == '[' || a[i] == '{') {
      s.push(a[i]);
    }
    else {
      if (s.size()) {
        char c = s.top();
        if (c == '(' && a[i] != ')') return 0;
        if (c == '[' && a[i] != ']') return 0;
        if (c == '{' && a[i] != '}') return 0;
        s.pop();
      }
      else {
        return 0;
      }
    }
  }

  if (s.size() == 0) return 1;
  else return 0;
}
```

Stack Evaluate Expression

Evaluate the value of an arithmetic expression in Reverse Polish Notation. Valid operators are +, -, *, /. Each operand may be an integer or another expression.

Example:
["2", "1", "+", "3", "*"] -> ((2 + 1) * 3) -> 9
["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6

Time complexity: O(N), where N is the size of the array.
Explanation: Use a stack to store the integers. Each time you find a operand use your stack to make the operation. Store the result of the operation at the top of the stack. This way, the top of the stack os always your right operand and the element before the top is your left operand. Once you process all the array the stack must be left with just one element that is your answer, so return the top of the stack.

```
int _eval(int l, int r, string o) {
  int res = 0;

  if (o == "*") {
    res = l * r;
  }
  else if (o == "/") {
    res = l / r;
  }
  else if (o == "+") {
    res = l + r;
  }
  else if (o == "-") {
    res = l - r;
  }

  return res;
}
```

```
int evalRPN(vector<string> &a) {
  if (a.size() == 1) return atoi(a[0].c_str());

  stack<int> s;
  int l, r;

  for (int i = 0; i < a.size(); i++) {
    if (a[i] != "*" && a[i] != "/" && a[i] != "+" && a[i] != "-") {
      s.push(atoi(a[i].c_str()));
    }
    else {
      r = s.top();
      s.pop();
      l = s.top();
      s.pop();
      s.push(_eval(l, r, a[i]));
    }
  }

  return s.top();
}
```

Stack Max Area Histogram

```
int largestRectangleArea(vector<int> &a) {
  stack<int> posStack;
  int height, width, area, maxArea = 0;

  for (int i = 0; i < a.size();) {
    if (posStack.empty() || a[posStack.top()] <= a[i]) {
      posStack.push(i++);
    }
    else {
      while (!posStack.empty() && a[posStack.top()] > a[i]) {
        height = a[posStack.top()];
        posStack.pop();
        width = !posStack.empty() ? i - (posStack.top() + 1) : i;
        area = width * height;
        maxArea = max(maxArea, area);
      }
    }
  }

  while (!posStack.empty()) {
    height = a[posStack.top()];
    posStack.pop();
    width = !posStack.empty() ? a.size() - (posStack.top() + 1) : a.size();
    area = width * height;
    maxArea = max(maxArea, area);
  }

  return maxArea;
}
```

Stack Nearest Smaller Element

```
Given an array, find the nearest smaller element G[i] for every element A[i]
in the array such that the element has an index smaller than i.

More formally,
G[i] for an element A[i] is equal an element A[j] such that
  j is maximum possible AND
  j < i AND
  A[j] < A[i]
```

Example:
Input : A : [4, 5, 2, 10]
Return : [-1, 4, -1, 2]

Time complexity:

Explanation: use a stack to keep candidates. Start to iterate over the array. If the stack is empty there are no candidates so just add -1 to your answer. While the stack is not empty try to find an element in the stack that is smaller than your current element. If you find, add this element to your answer. At the end of each iteration add the current element to the stack.

This approach is possible because if an element in the stack is bigger than the current element you are analysing, so this element in the stack is definetely not a better candidate than your current one for the next iteration, so you can remove it.

```
vector<int> prevSmaller(vector<int> &a) {
  vector<int> r;
  stack<int> s;

  for (int i = 0; i < a.size(); i++) {
    while (s.size()) {
      if (s.top() < a[i])  {
        r.push_back(s.top());
        break;
      }
      s.pop();
    }

    if (s.empty()) r.push_back(-1);
    s.push(a[i]);
  }

  return r;
}
```

Stack Redundant Braces

Write a program to validate if the input string has redundant braces?
Return 0 / 1 , 0 -> NO 1 -> YES
Input will be always a valid expression and operators allowed are only:
+ , * , - , /

Example:
((a + b)) has redundant braces so answer will be 1
(a + (a + b)) doesn't have have any redundant braces so answer will be 0

Time complexity: O(N), where N is the length of the string.
Explanation: All non-redundant braces must have at least one operator inside of it. So we use a stack to put all the openning braces and operators. Each time we find a closing brace ')' we start to pop() the stack until the element in the stack is a '('. When that happens we check to see if before

the '(' element on the stack there were operators. If there were not we return 1. When we get to the end of the string since the exercise guarantee it is valid expression we just return 0.

```
int braces(string a) {
  stack<char> s;

  for (int i = 0; i < a.length(); i++) {
    if (a[i] == '(' || a[i] == '+' || a[i] == '*' || a[i] == '/'
      || a[i] == '-') {
      s.push(a[i]);
    }
    else if (a[i] == ')') {
      if (s.top() == '(') return 1;
      else {
        while (!s.empty() && s.top() != '(') {
          s.pop();
        }

        if (s.top() == '(') s.pop();
      }
    }
  }

  return 0;
}
```

Stack With Min Element

```
class MinStack {
  public:
    MinStack();
    void push(int x);
    void pop();
    int top();
    int getMin();
    void print(stack<int> &p);
};

stack<int> s;
stack<int> m;
int cmin;

MinStack::MinStack() {
  while (!s.empty()) {
    s.pop();
  }
  while (!m.empty()) {
    m.pop();
  }
}

void MinStack::push(int x) {
  if (s.empty()) {
    cmin = x;
  }
  else if (cmin >= x) {
    m.push(cmin);
    cmin = x;
  }
```

```
    s.push(x);
  }

  void MinStack::pop() {
    if (s.empty()) return;

    if (!m.empty() && cmin == s.top()) {
      cmin = m.top();
    }

    if (!m.empty() && m.top() == cmin) {
      m.pop();
    }

    s.pop();
  }

  int MinStack::top() {
    if (s.empty()) return -1;
    return s.top();
  }

  int MinStack::getMin() {
    if (s.empty()) return -1;
    return cmin;
  }

  void MinStack::print(stack<int> &p) {
    cout << "Stack: ";

    for (std::stack<int> dump = p; !dump.empty(); dump.pop())
      std::cout << dump.top() << ' ';

    cout << '\n';
  }
```

String Add Binary

Given two binary strings, return their sum (also a binary string).

Example:
a = "100"
b = "11"
Return a + b = “111”.

Time complexity: O(N), where N is the length of the string (and if we consider the push_back function of a string object as an O(1) operation).

Explanation: iterate from the end of the input strings. Get the integer values from these characters (doing something like character - '0'). Now, you have bit1 and bit2. You also need a "carry" variable. Perform the XOR operation to get the resulting bit from that iteration doing bit1 ^ bit2 ^ carry, and a sum followed by a shift to get the new carry value, (carry + bit1 + bit 2) >> 1. Do that while you still have bits to check in one of your inputs or while carry is equal 1.

```
  string addBinary(string a, string b) {
    int i = a.size() - 1, j = b.size() - 1, carry = 0;
    string ans = "";

    while (i >= 0 || j >= 0 || carry) {
      int b1 = i >= 0 ? a[i] - '0' : 0;
      int b2 = j >= 0 ? b[j] - '0' : 0;

      if (b1 ^ b2 ^ carry) ans.push_back('1');
      else ans.push_back('0');

      carry = (b1 + b2 + carry) >> 1;

      i--;
      j--;
    }

    reverse(ans.begin(), ans.end());

    return ans;
  }
```

String Atoi

```
Implement atoi to convert a string to an integer.
```

Example:

```
Given "9 2704"
Return 9

Questions:
Q1. Does string contain whitespace characters before the number?
A. Yes
Q2. Can the string have garbage characters after the number?
A. Yes. Ignore it.
Q3. If no numeric character is found before encountering garbage characters,
what should I do?
A. Return 0.
Q4. What if the integer overflows?
A. Return INT_MAX if the number is positive, INT_MIN otherwise.
```

Time complexity: O(N), where N is the length of the string.
Space complexity: O(1).

Explanation: there is not much to say here. Iterate through the string convert each number character to a number and add to your answer (ans * 10 + number). Take care of the overflow case in each iteration checking if you are in a case where adding the next number will be larger than INT_MAX (or smaller than INT_MIN).

```
int atoi(const string &a) {
  int sign = 1, ans = 0, i = 0;

  // Discard initial empty spaces, check if there is a sign to update our sign
  // variable and discard it too.
  while (a[i] == ' ') i++;
  if (a[i] == '-' || a[i] == '+') {
    sign = a[i] == '-' ? -1 : 1;
    i++;
  }

  // While we have numbers. The '/0' is not in this range so we don't need to
  // check the string size.
  while (a[i] >= '0' && a[i] <= '9') {
    // Check for overflow. The max value for negative integers is one unit
    // larger (in absolute value) than the max value for positive integers.
    if (ans > INT_MAX / 10 || (ans == INT_MAX / 10 && a[i] - '0' > 7)) {
      return INT_MAX;
    }
    else if (ans < INT_MIN / 10 || (ans == INT_MIN / 10 && a[i] - '0' > 8)) {
      return INT_MIN;
    }
    // Convert the current character to a number based on their ASCII values.
    // The ASCII code for a character that represents a number N minus the
    // character that represents 0 gives N. Multiply by the sign here because
    // this way "ans" will during the whole process be negative if the sign is
    // negative. This is important for the overflow check.
    ans  = 10 * ans + ((a[i] - '0') * sign);
    i++;
  }

  return ans;
}
```

String Count And Say

The count-and-say sequence is the sequence of integers beginning as follows:
1, 11, 21, 1211, 111221, ...
1 is read off as one 1 or 11.
11 is read off as two 1s or 21.
21 is read off as one 2, then one 1 or 1211.

Given an integer n, generate the nth sequence.

Note: The sequence of integers will be represented as a string.

Example:

if n = 2,
the sequence is 11.

Time complexity: hard to tell, but will be something like O(N * M), where N is the given input and M is the length of the largest string.

Explanation: I did a straight forward solution. Just count how many times each digit occurred and build a new string for each i until i is N.

```
string countAndSay(int n) {
  if (n == 0) return "";

  string ans = "1";

  for (int i = 1; i < n; i++) {
    string t = "";
    for (int j = 0; j < ans.size();) {
      int k = j, r = 0;
      char first = ans[k];

      while (ans[k] == first) {
        k++;
        r++;
      }

      j = k;
      t += to_string(r) + first;
    }
    swap(ans, t);
  }
```

```
    return ans;
  }
```

String Integer To Roman

```
Given an integer, convert it to a roman numeral, and return a string
corresponding to its roman numeral version.
Input is guaranteed to be within the range from 1 to 3999.
```

Example:
```
Given 5, return "V".
Given 14, return "XIV".
```

Time complexity: O(1), it is proportional to the number of 10th powers, but since we have at most 10^3 numbers it is constant.

Explanation: of course we need a dicitionary to get our symbols. We need to get with this dicitionary the roman strings of each of our 10th powers.

```
string intToRoman(int a) {
  // 1000, 2000, 3000
  string M[] = {"", "M", "MM", "MMM"};
  // 100, 200, 300, .. 900
  string C[] = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};
  // 10, 20, ... 90
  string X[] = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};
  // 1, 2, ... 9
  string I[] = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};

  return M[a / 1000] + C[(a % 1000) / 100] + X[(a % 100) / 10] + I[a % 10];
}
```

String Longest Common Prefix

```
Write a function to find the longest common prefix string amongst an array
of strings. Longest common prefix for a pair of strings S1 and S2 is the
longest string S which is the prefix of both S1 and S2.
As an example, longest common prefix of "abcdefgh" and "abcefgh" is "abc".

Given the array of strings, you need to find the longest S which is the
prefix of ALL the strings in the array.
```

Example:
```
Given the array,
[
  "abcdefgh",
  "aefghijk",
  "abcefgh"
]

The answer would be “a”.
```

Time complexity: O(N^2), where N is the length of the largest string.
Space complexity: O(N).

Explanation: iterate from 0 to the length of the string with the smallest length. For each character a[i][0] check if every other character a[i][j]

is equal. If all of them are equal add this character to you answer,
otherwise return the answer.

```
string longestCommonPrefix(vector<string> &a) {
  int minLen = INT_MAX;
  string result = "";

  for (int i = 0; i < a.size(); i++) minLen = min(minLen, (int)a[i].size());

  for (int i = 0; i < minLen; i++) {
    int count = 0;

    for (int j = 0; j < a.size(); j++) {
      if (a[j][i] == a[0][i]) count++;
      else break;
    }

    if (count == a.size()) result.push_back(a[0][i]);
    else break;
  }

  return result;
}
```

String Longest Palindrome

Given a string S, find the longest palindromic substring in S.

Substring of string S:
S[i...j] where 0 <= i <= j < len(S)

Palindrome string:
A string which reads the same backwards. More formally, S is palindrome if
reverse(S) = S.

Incase of conflict, return the substring which occurs first (with the least
starting index).

Example:

Given "aaaabaaa"
Output "aaabaaa"

Time complexity: O(N^2), where N is the length of the string.
Space complexity: O(1).

Explanation: we expand from the center of the palindrome, because from its
extremes would result in a lot of useless work. For example, if we are
checking the string "abad" if we expand from the extremes we would check
"a" with "d" and see it is not a palindrome, then "a" with "a". So, we can
see that if we start from the center the check of "a" with "d" would be
unnecessary.

In my code I decided to run a loop from 0 .. 2 * N - 1, this is because
there are two cases:

1. When our palindrome will have odd length and then the center itself is
made of just one character and doesn't need to be checked
2. When the palindrome length will be even and the we check the center with
its right neighbor. For example "bbbb", when we are at index 1 as center
we compare a[1] with a[2].

So I iterate double of the length of the string and for odd i I defined I'm in the case where the palindrome has even length. The center will always be i / 2.

```
string longestPalindrome(string a) {
  int n = a.size(), start = 0, end = 0;

  for (int i = 0; i < 2 * n - 1; i++) {
    int center = i / 2, currStart = 0, currEnd = 0;
    // If i is odd I consider the palindrome to have even length.
    int left = i % 2 ? center : center - 1, right = center + 1;

    while (left >= 0 && right < n && a[left] == a[right]) {
      currStart = left;
      currEnd = right;
      left--;
      right++;
    }

    // Just "larger", and not "larger or equal" because we want to preserve the
    // palindrome that starts at the lowest possible index.
    if (currEnd - currStart > end - start) {
      start = currStart;
      end = currEnd;
    }
  }

  return a.substr(start, (end - start) + 1);
}
```

String Pretty Json

Pretty print a JSON object using proper indentation. Your solution should return a list of strings, where each entry corresponds to a single line. The strings should not have “\n” character in them.

1. Every inner brace should increase one indentation to the following lines.
2. Every close brace should decrease one indentation to the same line and the following lines.
3. The indents can be increased with an additional ‘\t’
4. [] and {} are only acceptable braces in this case.

Assume for this problem that space characters can be done away with.

Example:

```
Given {A:"B",C:{D:"E",F:{G:"H",I:"J"}}}
Output,
{
    A:"B",
    C:
    {
        D:"E",
        F:
        {
            G:"H",
            I:"J"
        }
    }
}

Given ["foo", {"bar":["baz",null,1.0,2]}]
```

```
Output,
[
    "foo",
    {
        "bar":
        [
            "baz",
            null,
            1.0,
            2
        ]
    }
]
```

Time complexity: O(N), where N is the length of the given string.
Space complexity: O(N).

Explanation: nothing much to talk about here, just take care with some possible cases:

1. Ignore spaces, the exercise ask you for doing that, so don't forget.
2. Add/remove indentation.
3. Properly add commas that can come after closing braces.
4. Remove one indentation before inserting a closing brace.

```
vector<string> prettyJSON(string a) {
  vector<string> ans;
  int i = 0, level = 0;

  while (i < a.size()) {
    // Ignore white spaces
    if (a[i] == ' ') {
      i++;
      continue;
    }

    string line = "";
    // Insert the proper indentation.
    for (int i = 0; i < level; i++) line.push_back('\t');

    // For opening braces we increment the indentation and add it to the answer.
    if (a[i] == '{' || a[i] == '[') {
      line.push_back(a[i] == '{' ? '{' : '[');
      level++;
      i++;
    }
    // For closing braces we decrement the indentation and add it to the answer.
    // Remove one indentation because the closing braces come one indentation
    // before the previous content.
    else if (a[i] == '}' || a[i] == ']') {
      line.pop_back();
      line.push_back(a[i] == '}' ? '}' : ']');
      level--;
      i++;
    }
    else {
      // Insert everything that we can into our line and increment i. Here, we
      // also need to ignore spaces.
      while (i < a.size() && a[i] != ',' && a[i] != '}' && a[i] != ']' &&
        a[i] != '{' && a[i] != '[') {
        if (a[i] != ' ') line.push_back(a[i]);
        i++;
      }
    }

    // Commas can come after a value or after a closing brace, in both ways
    // they are part of the current line, so we check for it here instead of
    // inside the "else" above.
```

```
      if (i < a.size() && a[i] == ',') {
        line.push_back(',');
        i++;
      }

      ans.push_back(line);
    }

    return ans;
  }
```

String Reverse Words

```
Given an input string, reverse the string word by word.

Notes:
1. A sequence of non-space characters constitutes a word.
2. Your reversed string should not contain leading or trailing spaces, even
if it is present in the input string.
3. If there are multiple spaces between words, reduce them to a single space
in the reversed string.
```

Example:

Given s = "the sky is blue",

return "blue is sky the".

Time complexity: O(M * N), where N is the length of the given string and M is the length of the largest word in this string.
Space complexity: O(N).

Explanation: there are multiple ways of performing this, but all are similar. I decided to start at the beginning of the given string and create word by word. When I find a space it is time to insert the created word in the beginning of the answer (in C++ I used insert()).

```
void reverseWords(string &a) {
  string ans = "", word = "";
  int i = 0;

  while (i < a.size()) {
    // Create a word.
    if (a[i] != ' ') {
      word.push_back(a[i]);
    }
    // Found a space so insert the word in the answer.
    else if (word.size() > 0) {
      // If the answer is not empty we need to insert a space after the current
      // word, otherwise it's the last word in the answer so no space is needed.
      if (ans.size() > 0) word.push_back(' ');
      ans.insert(0, word);
      word = "";
    }
    i++;
  }

  // Insert the last word because if there is no space after it the "while"
  // loop will not insert it.
  if (word.size()) {
    if (ans.size() > 0) word.push_back(' ');
    ans.insert(0, word);
  }

  a = ans;
}
```

String Roman To Integer

Given a roman numeral, convert it to an integer. Input is guaranteed to be within the range from 1 to 3999.

Read more details about roman numerals at Roman Numeric System

Example:

Given "XIV"
Return 14
Given "XX"
Output 20

Time complexity: O(N), where N is the length of the given string.
Space complexity: O(1).

Explanation: you need a dictionary to return map the letters to their respective number. There are two cases:
1. There is a letter that has a smaller value than the following letter. In this case you need to add to the result the difference between the larger and the smaller value. For example, "IV", we add to the result 5 - 1 = 4.
2. The above condition is not true, so we just add to the result the value of the current letter.

```
int getNumber(char c) {
  switch (c) {
    case 'I': return 1;
    case 'V': return 5;
    case 'X': return 10;
    case 'L': return 50;
    case 'C': return 100;
    case 'D': return 500;
    case 'M': return 1000;
    default: return 0;
  }
}

int romanToInteger(string a) {
  int result = 0;
  char prev = '0';

  for (int i = a.size(); i >= 0; i--) {
    if (getNumber(a[i]) < getNumber(prev)) result = result - getNumber(a[i]);
    else result = result + getNumber(a[i]);
    prev = a[i];
  }

  return result;
}
```

Sum Of Subsets

Given an array of numbers make an array containing all the possible sums of the elements of this array.

Time complexity: O((2^N) * N), where N is the size of the array.

```
void sum_of_all_subset(std::vector<int> s) {
  int n = s.size();
  int results[(1 << n)]; // (1 << n) = 2^n

  // initialize results to 0
  memset(results, 0, sizeof(results));

  // iterate through all subsets
  for (int i = 0 ; i < (1 << n); ++i) { // for each subset, O(2^n)
    for (int j = 0; j < n; ++j) { // check membership, O(n)
      if ((i & (1 << j)) != 0) // test if bit ‘j’ is turned on in subset ‘i’?
        results[i] += s[j]; // if yes, process ‘j’
    }
  }
}
```

Trie Shortest Unique Prefix

```
void insertWord(TrieNode **root, string str) {
  if (*root == NULL) *root = new TrieNode();

  TrieNode *curr = *root;

  for (int i = 0; i < str.length(); i++) {
    curr->freq++;

    if (curr->keys.find(str[i]) != curr->keys.end()) {
      curr = (curr->keys.find(str[i]))->second;
    }
    else {
      curr->keys[str[i]] = new TrieNode();
      curr = curr->keys[str[i]];

      if (i == str.length() - 1) curr->endOfWord = true;
    }
  }
}

vector<string> findUniquePrefixes(vector<string> &a) {
  vector<string> ans;

  if (a.size() == 0) return ans;

  TrieNode *root = NULL;

  for (int i = 0; i < a.size(); i++) {
    insertWord(&root, a[i]);
  }

  for (int i = 0; i < a.size(); i++) {
    string temp = "";
    TrieNode *curr = root;

    for (int j = 0; j < a[i].size(); j++) {
      temp += a[i][j];
      curr = (curr->keys.find(a[i][j]))->second;
      if (curr->freq <= 1) {
        ans.push_back(temp);
        break;
      }
    }
  }

  return ans;
}
```

Twop Container Max Area

```
Given n non-negative integers a1, a2, ..., an, where each represents a point
at coordinate (i, ai). 'n' vertical lines are drawn such that the two
endpoints of line i is at (i, ai) and (i, 0). Find two lines, which together
with x-axis forms a container, such that the container contains the most
water. In other words, given an array of non-negative integers, where each
element represents a point (i, a[i]) find the square with the largest area
represented by the lines drawn with this points and x axis.

Your program should return an integer which corresponds to the maximum area
of water that can be contained ( Yes, we know maximum area instead of maximum
volume sounds weird. But this is 2D plane we are working with for
simplicity).

Notes:
1. You may not slant the container.
```

Example:

```
Input [1, 5, 4, 3]
Output 6, because 5 and 3 are distance 2 apart. So size of the base = 2.
Height of container = min(5, 3) = 3. So total area = 3 * 2 = 6
```

Time complexity: O(N), where N is the size of the given array.
Space complexity: O(1).

Explanation: You start with a pointer at the begining of the array and one at the end. Since the beginning and end represent the longest base you could have, for the next area to be larger needs to have a height longest than the previous one (and height = min(a[start], a[end])). So, you have:
1. If a1 < aN, then the problem reduces to solving the same thing for a2, aN.
2. Else, it reduces to solving the same thing for a1, aN-1

```
int calcArea(vector<int> &a, int i, int j) {
  return (j - i) * min(a[i], a[j]);
}

int maxArea(vector<int> &a) {
  int area = 0, i = 0, j = a.size() - 1;

  while (i < j) {
    area = max(area, calcArea(a, i, j));

    if (a[i] < a[j]) {
      i++;
    }
    else {
      j--;
    }
  }
  return area;
}
```

Twop Diff Elements Equal K

Given an array ‘A’ of sorted integers and another non negative integer k,
find if there exists 2 indices i and j such that A[i] - A[j] = k, i != j.

Example:
A : [1 3 5]
k : 4

Output : True, because 5 - 1 = 4

Time complexity: O(N), where N is the size of the array.

Explanation: Start with a pointer at the beginning of the array, call it i, and another one at beginning + 1, call it j. Make the diff between these two elements, if the diff is bigger than k so we increment j, otherwise we increment i.

```
/**
 * @input A : Integer array
 * @input n1 : Integer array's ( A ) length
 * @input B : Integer
 *
 * @Output Integer
 */
int diffPossible(int* a, int n1, int k) {
  int i = 1, j = 0, diff;

  while (i != j && i < n1 && j < n1) {
    diff = a[i] - a[j];

    if (diff > k && j + 1 != i) j++;
    else if (diff > k && j + 1 == i) {j++; i++;}
    else if (diff < k) i++;
    else return 1;
  }

  return 0;
}
```

Twop Intersection Sorted Arrays

Find the intersection of two sorted arrays. Assume that elements that appear more than once must be included more than once.

Example:
Input : A : [1 2 3 3 4 5 6] B : [3 3 5]
Output : [3 3 5]

Time complexity: O(N), where N is the biggest length between the two arrays.

Explanation: keep two pointers, one at the beginning of the first array, call it i, and other one at the beginning the the second array, call it j. When a[i] < b[j], increment i, when b[j] < a[i] increment j, else (they are equal) push the element into a result array.

```
vector<int> intersect(const vector<int> &a, const vector<int> &b) {
  vector<int> r;
  int len_a =  a.size();
  int len_b = b.size();
  int i = 0, j = 0;

  while (i < len_a && j < len_b) {
    if (a[i] < b[j]) i++;
    else if (b[j] < a[i]) j++;
    else {
      r.push_back(a[i]);
      i++;
      j++;
    }
  }

  return r;
}
```

Twop Remove Duplicates

Given a sorted array, remove the duplicates in place such that each element appears only once and return the new length. Note that even though we want you to return the new length, make sure to change the original array as well in place.
Do not allocate extra space for another array, you must do this in place with constant memory.

Example:
Given input array A = [1,1,2],
Your function should return length = 2, and A is now [1,2].

Time complexity: O(N^2 / M), where N is the size of the array and M is the average number of duplicates.
Space complexity: O(1).
Obs: the erase() function has time complexity Linear on the number of elements erased (destructions) plus the number of elements after the last element deleted (moving).

Explanation: use two pointers. Iterate from the end of the array (most of the times when you are willing to delete elements from an array you should start at the end). Use a variable to keep track of the number of duplicates. While you are finding elements that are equal to its predecessor increment your count, otherwise delete all elements from i + 1 (since i is the index of the element that is different from the others) to i + 1 + count, and restart your count making it 0.

```
int removeDuplicates(vector<int> &a) {
  int i, j = 0;

  for (i = a.size() - 2; i >= 0; i--) {
    if (a[i] == a[i + 1]) j++;
    else {
      if (j) a.erase(a.begin() + i + 1, a.begin() + i + 1 + j);
      j = 0;
    }
  }
  if (j) a.erase(a.begin() + i + 1, a.begin() + i + 1 + j);

  return a.size();
}
```

Twop Sort By Color

Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue. Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

Example:
Input : [0 1 2 0 1 2]
Modify array so that it becomes : [0 0 1 1 2 2]

Time complexity: O(N), where N is the length of the array.

Explanation: Keep a counter to each color. Then overwrite the array using the counter. For example, if one of the counter is named red_counter. Decrement the red counter and assign 0 (the integer representing the red color) to the array. Another approach is, swap the 0s to the beginning of the array keeping a pointer and 2s the the end also keeping a pointer. 1s will automatically be at the right place.

```
void sortColors(vector<int> &a) {
  int cr = 0, cw = 0, cb = 0, i = 0;

  if (a.size() == 1) return;

  while (i < a.size()) {
    if (a[i] == 0) cr++;
    else if (a[i] == 1) cw++;
    else cb++;
    i++;
  }

  i = 0;

  while (cr--) {
    a[i] = 0;
    i++;
  }

  while (cw--) {
    a[i] = 1;
    i++;
  }

  while (cb--) {
    a[i] = 2;
    i++;
  }
}
```