Online Contests Solutions

Saman Saadi

Contents

Chapter 1

LeetCode

1.1 Medium

1.1.1 Longest Substring Without Repeating Characters

Find the length of the longest substring without repeating characters. For example in "abcabcbb" the answer is "abc".

We can use mathematical induction. Assume $s[i..j-1]$ is a valid substring (it doesn't have any repetitive characters). Now we want to add $s[j]$ to it. If $s[j]$ is not in $s[i..j-1]$, then we can easily add $s[j]$ to the list and claim $s[i..j]$ is also valid. Otherwise there is exactly one $i \leq k \leq j-1$ that $s[k] = s[j]$.

Because of $s[j]$, we cannot extend $s[i..j-1]$ anymore, but we may be able to extend $s[k+1..j]$. The running time of the following implementation is $O(n)$:

```
int lengthOfLongestSubstring(string s) {
  int res = 0;
  unordered_map<char, int> mp;
  for (int i = 0, j = 0; i < s.length() && j < s.length(); ++j)
  {
    auto iter = mp.find(s[j]);
    if (iter != mp.end())
    {
      //Note that after mp.erase, iter will be invalid!
      //So we store the index in k
      int k = iter->second;
      for (; i <= k; ++i)
        mp.erase(s[i]);
    }
    mp[s[j]] = j;
    res = max(res, j - i + 1);
  }
  return res;
}
```

1.1.2 Longest Palindromic Substring

Given a string s, return the longest palindromic substring in s. For another variation of this problem refer to 3.1.1.

Assume $S[i..j] = s_i s_{i+1} \dots s_j$. We define $S'[i'..j'] = s_{i'} s_{i'+1} \dots s_{j'}$ in such a way $(s_j = s_{i'}) \wedge (s_{j-1} = s_{i'+1}) \wedge \dots \wedge (s_i = s_{j'})$. Let's Assume string P is a palindromic string and c is an arbitrary character. We can break P into:

$$P[i..j] = \begin{cases} S[i..m]S'[m+1..j] & (j-i+1) \text{ is even} \\ S[i..m-1]cS'[m+1..j] & (j-i+1) \text{ is odd} \end{cases}$$

It's obvious that $m = \lfloor \frac{i+j}{2} \rfloor$. To solve this problem we need to find the place of m that can be $0 \leq m < n$. n is the length of the string. The running time of this algorithm is $O(n^2)$:

```
string longestPalindrome(string s) {
  int len = 0;
  int index = -1;

  const auto findLongest = [&](int l, int r)
  {
    for (; l >= 0 && r < s.length() && s[l] == s[r]; --l, ++r)
    {
      if ((r - l + 1) > len)
      {
        len = r - l + 1;
        index = l;
      }
    }
  };

  for (int m = 0; m < s.length(); ++m)
  {
    //Check for S S_r (length is even)
    findLongest(m, m + 1);
    //Check for S c S_r (length is odd)
    findLongest(m, m);
  }
  if (index == -1)
    return "";
  return s.substr(index, len);
}
```

1.1.3 Container With Most Water

You are given an integer array *hieght* of length n. There are n vertical lines drawn such that the two endpoints of i^{th} line are $(i, 0)$ and $(i, height[i])$. Find two lines that together with the x-axis form a container, such that the container contains the most water. You may not slant the container.

We use mathematical induction. Suppose we know the answer for the number of lines less than n. Now we consider n lines. We consider the first and n^{th} lines and call them l_1 and l_n:

1. $height[1] < height[n]$: We find the best solution for l_1. Because for $2 \leq k \leq n-1$, the container l_1 and l_k cannot have more water than l_1 and l_n. We can safely remove l_1 and use mathematical induction to find the optimal solution. Then we need to compare that solution with l_1 and l_n and choose the maximum
2. $height[1] > height[n]$: We find the best solution for l_n, like above we can safely remove l_n and find the best solution and compare it with l_1 and l_n
3. $height[1] = height[n]$ We find the best solution for both l_1 and l_n. We can safely remove both of them and find the optimal solution and compare it with l_1 and l_n.

The running time of this algorithm is $O(n)$.

```
int maxArea(vector<int>& height) {
  int left = 0, right = height.size() - 1;
  int res = 0;
  while (left < right)
  {
    if (height[left] < height[right])
    {
      res = max(res, (right - left) * height[left]);
      ++left;
    }
    else if (height[left] > height[right])
    {
      res = max(res, (right - left) * height[right]);
      --right;
    }
    else
    {
      res = max (res, (right - left) * height[right]);
      ++left;
      --right;
    }
  }
  return res;
}
```

1.1.4 Two Sum II - Input Array Is Sorted

Given a sorted array, return two indices i, j such that $numbers[i]+numbers[j] = target \wedge i < j$. The indices start from 1.

For similar questions, refer to 1.1.5. For a general solution refer to 1.1.6

We use mathematical induction. We assume we know the solution for all sorted arrays with length less than n. Now consider a sorted array with length n:

1. $numbers[0]+numbers[n-1] < target$: Since the array is sorted, $numbers[n-1]$ is the maximum and $numbers[0]$ is the minimum. So $numbers[0]$ cannot be in the solution. We use mathematical induction to get the answer from $numbers[1..n-1]$

2. $numbers[0] + numbers[n-1] > target$: With a similar argument we can say $numbers[n-1]$ cannot be in the solution. So we use the induction to solve $numbers[0..n-2]$

3. $number[0] + numbers[n-1] == target$: We have the solution.

```
vector<int> twoSum(vector<int>& numbers, int target) {

  int left = 0, right = numbers.size() - 1;
  while (left < right)
  {
    const auto sum = numbers[left] + numbers[right];
    if (sum < target)
      ++left;
    else if ( sum > target)
      --right;
    else
      return {left + 1, right + 1};
  }
  throw 1;
}
```

1.1.5 3Sum

Given an integer array nums, return all triplets $[nums[i], nums[j], nums[k]]$ such that $i \neq j \wedge i \neq k \wedge j \neq k$. Notice that the solution set must not contain **duplicate triplets**.

This is the extension of 1.1.4. For general solution refer to 1.1.6

The tricky part here is how to avoid duplicates. We assume $nums$ is sorted, then we use mathematical induction. We assume we know how to solve it for less than n sorted numbers. Now we want to solve n sorted numbers. We consider the first element $nums[0]$. Now consider $nums[j]$ such that $j = k + 1$. We define k as $\{nums[0], nums[1], \ldots, nums[k]\}$ such that $nums[m] = nums[m+1]$ for $0 \leq m \leq k-1$. We use mathematical induction to find the answers for $nums[j..n-1]$.

Note that we cannot use induction for subproblems $nums[m..n-1]$ for $1 \leq m \leq k$ because it's possible they have triplets that start with $nums[0]$. As mentioned in problem statement, duplicates are not allowed.

Then we need to find the answers that contain $nums[0]$. For doing that we should find set S:

$$S = \{\{p, q\} : 1 \leq p < q \leq n-1 \wedge nums[p] + nums[q] = -nums[0]\}$$

Note that S shouldn't have any duplicates. Also it's important that we should find the pairs in $nums[1..n-1]$ (not in $nums[j..n-1]$. For example $[0, 0, 0]$ is a valid triplet). This is two sum II problem (refer to 1.1.4), but we should handle duplicates. There are two solutions to implement two sum:

```
vector<vector<int>> threeSum(vector<int>& nums) {
  vector<vector<int>> res;
```

```
  sort(nums.begin(), nums.end()); //O(nlogn)
  unordered_set<int> visited;
  for (int i = 0; i < nums.size() && nums[i] <= 0; ++i)
  {
    // To find subprolem nums[j..n - 1]
    //note that i points to the first repetitive character
    if (i > 0 && nums[i] == nums[i - 1])
      continue;
    visited.clear();
    for (int j = i + 1; j < nums.size(); ++j)
    {
      //make sure j points to the last repetitive character:
      if (j + 1 < nums.size() && nums[j] == nums[j + 1])
      {
        // to handle cases like [0, 0, 0]
        visited.insert(nums[j]);
        continue;
      }
      auto target = -nums[i] - nums[j];
      if (visited.count(target) > 0)
        res.push_back({nums[i], target, nums[j]});
      visited.insert(nums[j]);
    }
  }
  return res;
}
```

The second solution:

```
vector<vector<int>> threeSum(vector<int>& nums) {
  vector<vector<int>> res;

  sort(nums.begin(), nums.end()); //O(nlogn)
  for (int i = 0; i + 1 < nums.size() && nums[i] <= 0; ++i)
  {
    // To find subprolem nums[j..n - 1]:
    if (i > 0 && nums[i] == nums[i - 1])
      continue;
    int target = -nums[i];
    int left = i + 1, right = nums.size() - 1;
    while (left < right)
    {
      auto sum = nums[left] + nums[right];
      if (sum < target)
        ++left;
      else if (sum > target)
        --right;
      else
      {
        res.push_back({nums[i], nums[left], nums[right]});
        ++left;
        while (left < right && nums[left] == nums[left - 1])
          ++left;
        --right;
      }
    }
  }
```

```
  return res;
}
```

1.1.6 4sum

Given an array $nums$ of n integers, return an array of all the **unique** quadruplets $[nums[a], nums[b], nums[c], nums[d]]$ such that $a < b < c < d$ and $nums[a] + nums[b] + nums[c] + nums[d] = target$.

The base case for this solution is 2sum. For more information refer to 1.1.4. It's a good idea to look at 3sum in 1.1.5

We can solve it similar to 3sum (for more information refer to 1.1.5). We suppose the array is sorted. We use mathematical induction, so we know how to solve it for less than n sorted numbers. Now we consider n sorted numbers. We choose the first two numbers $nums[0]$ and $nums[1]$. We choose subproblem $nums[j..n-1]$ such that $nums[m] = nums[0] \vee nums[m] = nums[1]$ for $2 \leq m \leq j-1$. As explained in section 1.1.5, this is required to avoid duplication. We use mathematical induction to find quadruplets in $nums[j..n-1]$. Then for finding quadruplets that contain $nums[0]$ and $nums[1]$, we use 2sum (refer to section 1.1.4) algorithm on $nums[2..n-1]$ (not $nums[j..n-1]$. For example $[0,0,0,0]$ is a valid quadruplet if $target = 0$).

```
vector<vector<int>> fourSum(vector<int>& nums, int target) {
  vector<vector<int>> res;
  const auto size = nums.size();
  sort(nums.begin(), nums.end());
  for (int i = 0; (i + 3) < size; ++i)
  {
    if (i > 0 && nums[i] == nums[i - 1])
      continue;
    for (int j = i + 1; (j + 2) < size; ++j)
    {
      if (j > i + 1 && nums[j] == nums[j - 1])
        continue;
      int t = target - nums[i] - nums[j];
      int left = j + 1;
      int right = size - 1;
      while (left < right)
      {
        int sum = nums[left] + nums[right];
        if (sum < t)
          ++left;
        else if (sum > t)
          --right;
        else
        {
          res.push_back({nums[i], nums[j], nums[left], nums[right]})
              ;
          ++left;
          while (nums[left] == nums[left - 1] && left < right)
            ++left;
          --right;
        }
```

```
      }
    }
  }
  return res;
}
```

To solve $ksum$ problem, we need $k-2$ for loops to find $[nums[0], nums[1], \ldots, nums[k-3]]$ and we use 2sum algorithm to find $nums[k-2]$ and $nums[k-1]$. We can use a recursive function to solve the problem dynamically during the runtime.

```
vector<vector<int>> twosum(vector<int>& nums, int left, int target)
{
  int right = nums.size() -1;
  vector<vector<int>> res;
  while (left < right)
  {
    auto sum = nums[left] + nums[right];
    if (sum < target)
      ++left;
    else if (sum > target)
      --right;
    else
    {
      res.push_back({nums[left], nums[right]});
      ++left;
      while (nums[left] == nums[left - 1] && left < right)
        ++left;
      --right;
    }
  }
  return res;
}
vector<vector<int>> ksum(vector<int>& nums, int start, int k, int
    target)
{
  vector<vector<int>> res;
  if (start == nums.size())
    return res;
  int average = target / k;
  //nums[start] * k > target
  if (nums[start] > average)
    return res;
  //nums.back() * k < target
  if (nums.back() < average)
    return res;
  if (k == 2)
    return twosum(nums, start, target);

  for (int i = start; i < nums.size(); ++i)
  {
    if (i > start && nums[i] == nums[i - 1])
      continue;
    auto partial = ksum(nums, i + 1, k - 1, target - nums[i]);
    for (auto& val : partial)
    {
      val.insert(val.begin(), nums[i]);
      res.push_back(val);
```

```
    }
  }
  return res;
}

vector<vector<int>> fourSum(vector<int>& nums, int target) {
  sort(nums.begin(), nums.end());
  return ksum(nums, 0, 4, target);
}
```

1.1.7 Generate Parentheses

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

We use mathematical induction. Let's say S_n is the set of all valid n pairs. We assume we know how to generate S_k for $k < n$. It's obvious that $S_1 = \{"()"\}$ and $S_0 = \{""\}$. Now we consider S_n. let's assume $p \in S_i$ and $q \in S_{n-1-i}$ for $0 \leq i \leq n-1$. It's obvious that "(p)q" is a a valid parentheses with n pairs $(i + n - 1 - i + 1 = n)$. We define

$$(S_i) = \{(p) : p \in S_i\}$$

Also we define Cartesian product as follow (pq means the concatenation of strings p and q):

$$S_i \times S_j = \{pq : p \in S_i \wedge q \in S_j\}$$

So we can use the induction to generate S_n as follow:

$$S_n = \bigcup_{i=0}^{n-1} (S_i) \times S_{n-1-i}$$

```
vector<string> generateParenthesis(int n) {
  vector<string> res;
  if (n == 0)
    return {""};
  for (int i = 0; i < n; ++i)
  {
    auto set_left = generateParenthesis(i);
    auto set_right = generateParenthesis(n - i - 1);
    for (const auto& p : set_left)
    {
      string left_str = "(" + p + ")";
      for (const auto& q : set_right)
        res.push_back(left_str + q);
    }
  }
  return res;
}
```

1.1.8 Divide Two Integers

Divide two integers **without** using multiplication, division, and mod operator. We can only use **32-bit signed integers**.

We know that the range of a 32-bit signed integer is $[-2^{31}, 2^{31} - 1]$. We should convert negative numbers to positive ones. The tricky part is how to handle -2^{31}. Because we cannot fit it inside a positive 32-bit integer and we are not allowed to use *unsigned* variables.

Method 1

We use mathematical induction. Suppose $a = bq + r$. We define $div(a, b) = q$ and $mod(a, b) = r$. Suppose we know how to calculate div and mod for less than a. Now we want to calculate for a:

$$div(a,b) = \begin{cases} 2 \times div(k,b) + div(2 \times mod(k,b), b) & a = 2k \\ 2 \times div(k,b) + div(2 \times mod(k,b) + 1, b) & a = 2k+1 \\ 0 & a < b \\ 1 & a = b \\ a & b = 1 \\ base_{div}(a,b) & k \le b \end{cases}$$

$$mod(a,b) = \begin{cases} mod(2 \times mod(k,b), b) & a = 2k \\ mod(2 \times mod(k,b) + 1, b) & a = 2k+1 \\ a & a < b \\ 0 & a = b \vee b = 1 \\ base_{mod}(a,b) & k \le b \end{cases}$$

For calculating $base$ we use a straightforward approach. Since we cannot use multiplication and division, we can use $2 \times a = a + a$ and we use shift operator to do division by 2 ($a >> 1$).

Let's consider $a = -2^{31}$. Let's assume $q = div(2^{31}, |b|)$ and $r = mod(2^{31}, |b|)$:

$$|div(-2^{31}, |b|)| = \begin{cases} q & r + 1 < |b| \\ q + 1 & r + 1 \ge |b| \end{cases}$$

Method2

This is easier to implement. We assume $a = bq + r$. We define $div(a, b) = q$. Like previous method, suppose we know how to calculate div for less than a. Now we want to calculate for a. Suppose $b \times 2^k \le a < b \times 2^{k+1}$. So we can say $a = b \times 2^k + r$. If $r < b$ we find the answer; otherwise we use the induction to find $r = bq' + r'$. So the answer is:

$$\begin{aligned} a &= b \times 2^k + r \\ &= b \times 2^k + bq' + r' \\ &= b \times (2^k + q') + r' \end{aligned}$$

We can summarize it as:

$$div(a,b) = \begin{cases} 2^k + div(a - b \times 2^k, b) & a \geq b \\ 0 & a < b \end{cases}$$

Let's consider $a = -2^{31}$. We cannot store it as positive in a 32-bit signed integer. Let's assume $a = bq + r$:

$$\begin{aligned} & b(q-1) + r \\ & = bq + r - b \\ & = a - b \\ & \implies a - b = b(q-1) + r \end{aligned}$$

So $div(|2^{31}|, |b|) = div(|2^{31}| - |b|, |b|) + 1$:

```
int div(int a, int b)
{
  if (a < b)
    return 0;
  const int half_of_a = a >> 1;
  int q = 1, bq = b;
  for (; bq <= half_of_a; bq <<= 1, q <<= 1);
  return q + div(a - bq, b);
}

int divide(int a, int b) {
  static const int min_int = numeric_limits<int>::min();
  static const int max_int = numeric_limits<int>::max();
  int sign = 1;
  int res = 0;

  if (b == min_int)
    return a == min_int ? 1 : 0;
  if (b == 1)
    return a;
  if (b == -1)
    return a == min_int ? max_int : -a;

  if (b < 0)
    sign = -sign, b = -b;
  if (a == min_int)
    a += b, res = 1;
  if (a < 0)
    sign = -sign, a = -a;

  res += div(a, b);

  return res * sign;
}
```

1.1.9 Next Permutation

Implement std::next_permutation for $nums[0..n-1]$! For example if $nums = [1,3,2]$, the answer is $[2,1,3]$

Suppose p and q are two permutations of $nums[0..n-1]$. If $p < q$, then we have a $0 \leq i < n$ such that $p[k] = q[k]$ for $0 \leq k < i$ and $p[i] < q[i]$. For solving this question we should find the smallest q that is greater than p.

Assume i is the maximum index such that $nums[i] < nums[i+1]$ for $0 \leq i \leq n-2$. If such i does not exist it means we have the last permutation. So we should return the first one. Since i is the maximum index, it implies $nums[k] \geq nums[k+1]$ for all $i+1 \leq k \leq n-2$. In other words, $nums[i+1..n-1]$ is in non-ascending order.

Let's assume $j \geq i+1$ is the maximum index such that $nums[i] < nums[j]$. Since $nums[i+1..n-1]$ is in non-ascending order, that means $nums[j]$ is the smallest element that is bigger than $nums[i]$ in that range. We should swap $nums[i]$ and $nums[j]$ and reverse $nums[i+1..n-1]$ to make it in non-descending order. This is the smallest permutation greater than $nums$.

To summarize we break $nums[0..n-1]$ into $A = nums[0..i]$ and $B = nums[i+1..n-1]$. The former is in non-descending and the latter in non-ascending order. We define $B' = \{nums[j] \in B : nums[j] > nums[i]\}$. We find the smallest element in B' and swap it with $nums[i]$. Then we should sort B in non-descending order.

The implementation is based on possible implementation of next_permutation. To understand how reverse_iterator::base ($i.base()$) works, refer to B.5.

```
void nextPermutation(vector<int>& nums) {
  auto i = is_sorted_until(nums.rbegin(), nums.rend());
  if (i != nums.rend())
  {
    auto j = upper_bound(nums.rbegin(), i, *i);
    iter_swap(i, j);
  }
  //https://en.cppreference.com/w/cpp/iterator/reverse_iterator/base
  reverse(i.base(), nums.end());
}
```

1.1.10 Search in Rotated Sorted Array

There is an integer array $nums$ sorted in ascending order (with distinct values). For example let's assume we have $[0,1,2,4,5,6,7]$. We might rotate at pivot index 3 so it becomes $[4,5,6,7,0,1,2]$. Write an $O(\log n)$ algorithm to find the index of $target$.

In the non-rotated versions, the minimum number starts at index 0. We use the following algorithm to find the minimum index in the rotated one. We assume $middle = \lfloor \frac{left+right}{2} \rfloor$

$$f(left, right) = \begin{cases} f(middle+1, right) & nums[middle] > nums[right] \\ f(left, middle) & nums[middle] < nums[right] \\ left & left = right \end{cases}$$

Let's say $k = f(0, n-1)$. Then we have two ascending lists $A = nums[0..k-1]$

and $B = nums[k..n-1]$. Based on $target$ value we choose A or B to do a binary search to see whether $target$ is in the list.

```
int search(vector<int>& nums, int target) {
  int left = 0;
  const int n = nums.size();
  int right = n - 1;
  int k;
  while (left <= right)
  {
    if (left == right)
    {
      k = left;
      break;
    }
    int middle = (left + right) / 2;
    if (nums[middle] > nums[right])
      left = middle + 1;
    else
      right = middle;
  }
  if (target >= nums[k] && target <= nums[n - 1])
    left = k, right = n - 1;
  else
    left = 0, right = k - 1;
  while (left <= right)
  {
    if (left == right)
      return nums[left] == target ? left : -1;
    int middle = (left + right) / 2;
    if (nums[middle] < target)
      left = middle + 1;
    else
      right = middle;
  }
  return -1;
}
```

1.1.11 Find First and Last Position of Element in Sorted Array

Given an array of integers $nums$ sorted in non-decreasing order, find the starting and ending position of a given $target$ value.

Refer to 1.1.26 for a another variation of this problem. We define $f(i, j)$ that has the index of left-most $target$ in $nums[i..j]$. On the other hand we define $g(i, j)$ that keeps the index of the right-most $target$ in $nums[i..j]$.

Let's assume mid is index of the middle element. Both f and g acts similarly when $nums[mid] < target$ or $nums[mid] > target$. Let's assume $nums[mid] = target$. Since f should choose the left-most target, it tries to find the answer in $nums[0..mid]$. There is a possibly that other elements with value $target$ exists in $nums[0..mid]$. On the other hand, g should find the rightmost $target$. So it chooses $nums[mid..n-1]$.

We assume $mid_f = \lfloor \frac{left+right}{2} \rfloor$:

$$f(left, right) = \begin{cases} f(mid_f + 1, right) & nums[mid_f] < target \\ f(left, mid_f) & nums[mid_f] \geq target \\ left & left = right \wedge nums[left] = target \\ -1 & otherwise \end{cases}$$

We define $mid_c = \lceil \frac{left+right}{2} \rceil$:

$$g(left, right) = \begin{cases} g(mid_c, right) & nums[mid_c] \leq target \\ g(left, mid_c - 1) & nums[mid_c] > target \\ left & left = right \wedge nums[left] = target \\ -1 & otherwise \end{cases}$$

Finding mid becomes tricky when the size of sub-problem is 2. Let's say we are at $nums[i..i+1]$. We know $mid_f = \lfloor \frac{i+i+1}{2} \rfloor = i$ and $mid_c = \lceil \frac{i+i+1}{2} \rceil = i+1$. For f choosing mid_c leads to an infinite loop and for g it's mid_f:

$$f(i, i+1) = \begin{cases} f(mid_f + 1, i+1) = f(i+1, i+1) & nums[mid_f] < target \\ f(i, mid_f) = f(i, i) & nums[mid_f] \geq target \end{cases}$$

$$f(i, i+1) = \begin{cases} f(mid_c + 1, i+1) = f(i+2, i+1) & nums[mid_c] < target \\ f(i, mid_c) = f(i, i+1) & nums[mid_c] \geq target \end{cases}$$

Also we can think that f wants to find the leftmost $target$, so it makes sense to choose mid_f, but g wants to find the rightmost $target$ so it should choose mid_c. Now we can use $f(0, n-1)$ and $g(0, n-1)$ to find the solution:

```
vector<int> searchRange(vector<int>& nums, int target) {
  int n = nums.size();
  int left = 0, right = n - 1;

  while (left < right)
  {
    int mid = (left + right) / 2;
    if (nums[mid] < target)
      left = mid + 1;
    else
      right = mid;
  }
  if (left > right || nums[left] != target)
    return {-1, -1};
  vector<int> res(1, left);
  right = n - 1;
  while (left < right)
  {
    int mid = (left + right + 1) / 2;
    if (nums[mid] <= target)
      left = mid;
    else
```

```
      right = mid - 1;
  }
  res.push_back(left);
  return res;
}
```

1.1.12 Valid Sudoku

```
bool isValidSudoku(vector<vector<char>>& board) {
  array<array<bool, 10>, 9> row_seen{}, col_seen{};
  array<array<array<bool, 10> ,3>,3> box_seen{};

  for (int r = 0; r < 9; ++r)
  {
    for (int c = 0; c < 9; ++c)
    {
      if (board[r][c] == '.')
        continue;
      int d = board[r][c] - '0';
      int box_row = r / 3;
      int box_col = c / 3;
      auto& rs = row_seen[r][d];
      auto& cs = col_seen[c][d];
      auto& bs = box_seen[box_row][box_col][d];
      if (rs || cs || bs)
        return false;
      rs = cs = bs = true;
    }
  }
  return true;
}
```

1.1.13 Multiply Strings

Given two non-negative integers $num1$ and $num2$ represented as strings, return the product of num1 and num2, also represented as a string.

Let's assume the size of $num1$ is $n1$ and the size of $num2$ is $n2$. The result has at most $n1 + n2$ digits. We use mathematical induction. We assume we know how to solve the problem for every $num1$ of any size and $num2$ for sizes less than $n2$. Now we want to calculate the result when $num2$ has size $n2$. We use mathematical induction to calculate $num1[0..n1-1]$ and $num2[1..n2-1]$. Let's assume the answer is in $res[1..n1+n2-1]$. We assume $res[0] = 0$. Now we use the induction to calculate the result of $num1[0..n1-1]$ and $num2[0..n2-1]$. For $0 \leq i \leq n1-1$ we have:

$$res[i+1] = (res[i+1] + num1[i] \times num2[0]) \bmod 10$$

$$res[i] = res[i] + \lfloor \frac{res[i+1] + num1[i] \times num2[0]}{10} \rfloor$$

Note that $res[i]$ keeps the carry. It's possible it goes above 9 but after the algorithm finishes, it has the correct digit.

```
string multiply(string num1, string num2) {
  string res(num1.length() + num2.length(), '0');
  for (int i = num1.length() - 1; i >= 0; --i)
  {
    for (int j = num2.length() - 1; j >= 0; --j)
    {
      int t = (num1[i] - '0') * (num2[j] - '0') +
              res[i + j + 1] - '0';
      res[i + j + 1] = t % 10 + '0';
      res[i + j] += t / 10;
    }
  }
  int i = 0;
  for (;i < res.length() - 1 && res[i] == '0'; ++i);
  res = res.substr(i);
  return res;
}
```

1.1.14 Jump Game

You are given an integer array nums. You are initially positioned at the array's first index, and each element in the array represents your maximum jump length at that position. Return true if you can reach the last index, or false otherwise.

For explanation refer to 1.1.15.

```
bool canJump(vector<int>& nums) {
  int u;
  int upper_index = nums[0];
  for (u = 1; u <= upper_index && u < nums.size(); ++u)
    upper_index = max(upper_index, u + nums[u]);
  return upper_index >= nums.size() - 1;
}
```

1.1.15 Jump Game II

Given an array of non-negative integers *nums*, you are initially positioned at the first index of the array. Each element in the array represents your maximum jump length at that position. Your goal is to reach the last index in the minimum number of jumps. You can assume that you can always reach the last index.

This is one application of BFS. The graph is created in a special way that it's easy to implement it in $O(V)$ instead of $O(V + E)$.

If we run a BFS on this graph, we will have a BFS tree. The level of the root (index 0) is 0. The level of the childern of the root are 1 and so on. The answer is to find the level of the last element in a BFS tree.

We define $upper\_index[i]$ as the rightmost index of level i:

- level 0 starts at $upper\_index[0]$ and ends at $upper\_index[0]$ ($upper\_index[0] = 0$)
- level 1 starts at $upper\_index[0] + 1$ and ends at $upper\_index[1]$

- level $i > 0$ starts at $upper\_index[i-1]+1$ and ends at $upper\_index[i]$

We know $upper\_index[0] = 0$. For $i \geq 1$ we assume $upper\_index[i-1]+1 \leq j \leq upper\_index[i]$, then we have:

$$upper\_index[i+1] = \max(nums[j])$$

Since $upper\_index$ increases for next levels, we can assume $0 \leq j \leq upper\_index[i]$. So for caluclating $upper\_index[i+1]$ we just need to know $upper\_index[i]$. That means we can avoid an array and store them in two variables $upper\_index$ and $next\_upper\_index$:

```
int jump(vector<int>& nums) {
  int upper_index = 0, next_upper_index = 0;
  int d = 0;
  for (int u = 0; u < nums.size(); ++u)
  {
    if (u > upper_index)
    {
      ++d;
      upper_index = next_upper_index;
    }
    next_upper_index = max(next_upper_index, u + nums[u]);
  }
  return d;
}
```

1.1.16 Rotate Image

You are given an $n \times n$ 2D *matrix* representing an image, rotate the image **in-place** by 90 degrees (clockwise)

Let's consider a 3×3 image and rotate it 90 degrees:

$$\begin{matrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{matrix} \Longrightarrow \begin{matrix} a_{20} & a_{10} & a_{00} \\ a_{21} & a_{11} & a_{01} \\ a_{22} & a_{12} & a_{02} \end{matrix}$$

Let's assume the image is m and the rotated image is r, then we have:

$$m[i][j] = r[n-1-j][i]$$

For solving this problem first we do a transpose and then a reflect:

$$m[i][j] = m'[j][i] = r[n-1-j][i]$$

```
void rotate(vector<vector<int>>& matrix) {
  const auto n = matrix.size();
  for (int i = 0; i < n; ++i)
    for (int j = i; j < n; ++j)
      swap(matrix[i][j], matrix[j][i]);
  const int half = n / 2;
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < half; ++j)
      swap(matrix[i][j], matrix[i][n - 1 - j]);
}
```

1.1.17 Maximum Subarray

Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

We define $sums[i]$ as the contiguous subarray that ends to $nums[i]$ such that the sum of its elements are maximum:

$$sums[i] = \begin{cases} \max(sum[i-1] + nums[i], nums[i]) & i > 0 \\ nums[0] & i = 0 \end{cases}$$

So the answer is $\max_{i=0}^{n-1}(sums[i])$:

```
int maxSubArray(vector<int>& nums) {
  const int n = nums.size();
  auto sum = vector<int>(n);
  sum[0] = nums[0];
  for (int i = 1; i < n; ++i)
    sum[i] = max(sum[i - 1] + nums[i], nums[i]);
  int res = -10000;
  for (int i = 0; i < n; ++i)
    res = max(res, sum[i]);
  return res;
}
```

1.1.18 Spiral Matrix

Given an $m \times n$ matrix, return all elements of the matrix in spiral order.

This is an implementation question. For another variant look at Spiral Matrix II

```
// right, down, left, up:
// dir[i][0]: delta for row
// dir[i][1]: delta for col
int dir[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
// dir[i][0] = top, dir[i][1] = left,
// dir[i][2] = bottom and dir[i][3] = right
int box[4][4] = {{1, 0, 0, 0}, {0, 0, 0, -1}, {0, 0, -1, 0}, {0, 1,
    0, 0}};

vector<int> spiralOrder(vector<vector<int>>& matrix) {
  int rows = matrix.size();
  int cols = matrix[0].size();
  const int cells = rows * cols;
  int top = 0, bottom = rows - 1, left = 0, right = cols - 1;
  int cur_cell = 0;
  int r = 0, c = -1;
  vector<int> res;
  while (cur_cell < cells)
  {
    for (int i = 0; i < 4; ++i)
    {
      int nr = r + dir[i][0];
      int nc = c + dir[i][1];
      while (nr <= bottom && nr >= top && nc <= right && nc >= left)
```

```
      {
        ++cur_cell;
        r = nr, c = nc;
        res.push_back(matrix[r][c]);
        nr += dir[i][0];
        nc += dir[i][1];
      }
      top += box[i][0];
      left += box[i][1];
      bottom += box[i][2];
      right += box[i][3];
    }
  }
  return res;
}
```

1.1.19 Insert Interval

You are given an array of non-overlapping intervals that is sorted in ascending order by start points. You are given a new interval. Insert it into intervals such that they remain sorted and without overlaps.

For another variant of this problem refer to 4.1. The tricky part of this question is to find out when two intervals i_1 and i_2 have overlap. To find the answer it's easier to find out when they don't overlap:

$$\neg overlap(i_1, i_2) = \\ right(i_1) < left(i_2) \vee left(i_1) > right(i_2)$$

So we can calculate $overlap(i_1, i_2)$ as:

$$overlap(i_1, i_2) = \\ right(i_1) \geq left(i_2) \wedge left(i_1) \leq right(i_2)$$

For another similar problem look at Merge Intervals. The implementation:

```
vector<vector<int>> insert(vector<vector<int>>& intervals, vector<
    int>& newInterval) {
  int i;
  vector<vector<int>> res;
  for (i = 0; i < intervals.size() && intervals[i][1] < newInterval
      [0]; ++i)
    res.push_back(intervals[i]);
  auto left = newInterval[0];
  auto right = newInterval[1];
  //intervals[i][1] >= newInterval[0]
  for (; i < intervals.size() &&
         intervals[i][0] <= right; ++i)
  {
    left = min(left, intervals[i][0]);
    right = max(right, intervals[i][1]);
  }
  res.push_back({left, right});
  for (; i < intervals.size(); ++i)
```

```
    res.push_back(intervals[i]);
  return res;
}
```

1.1.20 Rotate List

Given the head of a linked list, rotate the list to the right by k places.

This is an implementation problem. We can easily find out the k^{th} element from the end of the list will become the first element after the rotation ($k = 1$ means last element). So we just need to find the last element and the parent of new head and modify them.

```
tuple<int, ListNode*> find_size_and_last_node(ListNode* node)
{
  int len = 0;
  ListNode* parent = nullptr;
  for (; node != nullptr; ++len, node = node->next)
    parent = node;
  return make_tuple(len, parent);
}

ListNode* find_from_last(ListNode* node, int size, int k)
{
  for (int i = 0; node != nullptr; ++i, node = node->next)
    if (size - i == k)
      return node;
  return nullptr;
}
ListNode* rotateRight(ListNode* head, int k) {
  if (head == nullptr)
    return head;
  int size;
  ListNode* last_node;
  tie(size, last_node) = find_size_and_last_node(head);
  k = k % size;
  if (k == 0)
    return head;
  auto new_head_parent = find_from_last(head, size, k + 1);
  auto new_head = new_head_parent->next;
  new_head_parent->next = nullptr;
  last_node->next = head;
  return new_head;
}
```

1.1.21 Unique Paths

There is a robot on an $m \times n$ grid. The robot is initially located at the top-left corner (i.e., grid[0][0]). The robot tries to move to the bottom-right corner (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time. Given the two integers m and n, return the number of possible unique paths that the robot can take to reach the bottom-right corner.

There are two solutions:

Dynamic programming

Let's say cnt[r][c] is the number of ways to reach from $(0, 0)$ to (r, c). We have:

$$cnt[r, c] = \begin{cases} cnt[r-1, c] + cnt[r, c-1] & r > 0 \wedge c > 0 \\ 1 & r = 0 \vee c = 0 \end{cases}$$

So the implementation is:

```
int uniquePaths(int m, int n) {
  auto cnt = vector<vector<int>>(m, vector<int>(n, 1));
  for (int r = 1; r < m; ++r)
    for (int c = 1; c < n; ++c)
    {
      cnt[r][c] = cnt[r - 1][c] + cnt[r][c - 1];
    }
  return cnt[m - 1][n - 1];
}
```

It's trivial to optimize memory and convert *cnt* to a 1-dimensional array:

```
int uniquePaths(int m, int n) {
  auto cnt = vector<int>(n, 1);
  cnt[0] = 1;
  for (int r = 1; r < m; ++r)
    for (int c = 1; c < n; ++c)
    {
      cnt[c] += cnt[c - 1];
    }
  return cnt[n - 1];
}
```

Permutations with repetition

We can simply calculate to find the answer. Let's assume $m \geq n$:

$$\begin{aligned} ans &= \binom{m+n-2}{m-1, n-1} \\ &= \frac{(m+n-2)!}{(m-1)! \times (n-1)!} \\ &= \frac{(m+n-2) \times \cdots \times (m-2)}{(n-1)!} \end{aligned}$$

To avoid overflow, we should do more. For simplicity consider the following permutation (again $m \geq n$):

$$\begin{aligned} &\binom{m+n}{m, n} \\ &= \frac{(m+n) \times (m+1) \times \cdots \times m!}{m! \times n!} \\ &= \frac{(m+1) \times (m+2) \times \ldots (m+n)}{1 \times \cdots \times n} \end{aligned}$$

Since $m+1$ to $m+n$ are n consecutive numbers, there is at least one number among them that is divisible by $1 \leq i \leq n$. For $1 \leq i \leq n$ we have:

$$\binom{m+i}{m,i} = \frac{(m+1) \times \ldots \ (m+i)}{1 \times \cdots \times i}$$

To avoid overflow we can start from $i = 1$ and increment i until we reach n. The division shouldn't have any remainder:

```
int uniquePaths(int M, int N) {
  auto m = max(M, N) - 1;
  auto n = min(M, N) - 1;
  const auto mn = m * n;

  // for values like m = 51 and n = 9 we cannot use 32-bit integer:
  int64_t res = 1;
  for (int i = 1; i <= n; ++i)
  {
    res *= m + i;
    res /= i;
  }
  return static_cast<int>(res);
}
```

1.1.22 Unique Paths II

You are given an m x n integer array grid. There is a robot initially located at the top-left corner (i.e., grid[0][0]). The robot tries to move to the bottom-right corner (i.e., grid[m-1][n-1]). The robot can only move either down or right at any point in time. An obstacle and space are marked as 1 or 0 respectively in grid. A path that the robot takes cannot include any square that is an obstacle. Return the number of possible unique paths that the robot can take to reach the bottom-right corner.

We use dynamic programming. We assume $a[0..m-1, 0..n-1]$ is obstacleGrid. We define $dp[0..m, 0..n]$ as:

$$dp[i,j] = \begin{cases} dp[i-1,j] + dp[i,j-1] & a[i,j] \neq 1 \\ 0 & a[i,j] = 1 \\ 1 & i = 1 \wedge j = 1 \wedge a[1,1] \neq 1 \\ 0 & i = 0 \vee j = 0 \end{cases}$$

So the answer for $a[i,j]$ is $dp[i+1, j+1]$. Since the destination is cell $m-1, n-1$, the final answer is $dp[m,n]$:

```
int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
  const int m = obstacleGrid.size();
  const int n = obstacleGrid[0].size();
  auto dp = vector<vector<int>>(m + 1, vector<int>(n + 1, 0));
  //dp[1][1] should be 1, unless there is an obstacle
  dp[0][1] = 1;
```

```
  for (int i = 0; i < m; ++i)
    for (int j = 0; j < n; ++j)
    {
      if (obstacleGrid[i][j] == 1)
        dp[i + 1][j + 1] = 0;
      else
        dp[i + 1][j + 1] = dp[i][j + 1] + dp[i + 1][j];
    }
  return dp[m][n];
}
```

As before we can save memory:

```
int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
  const int m = obstacleGrid.size();
  const int n = obstacleGrid[0].size();
  auto dp = vector<int>(n + 1, 0);
  dp[1] = 1;
  for (int i = 0; i < m; ++i)
    for (int j = 0; j < n; ++j)
    {
      if (obstacleGrid[i][j] == 1)
        dp[j + 1] = 0;
      else
        dp[j + 1] += dp[j];
    }
  return dp[n];
}
```

1.1.23 Minimum Path Sum

Given a $m \times n$ 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. You can only move either down or right at any point in time.

Instead of Dijkstra shortest path algorithm, we can use a simple dynamic programming approach. Suppose we have $grid[0..m-1, 0..n-1]$. We define $d[0..m, 0..n]$ as:

$$d[i,j] = \begin{cases} \min(d[i-1,j], d[i,j-1]) + grid[i-1,j-1] & i > 0 \wedge j > 0 \\ \infty & i = 0 \vee j = 0 \\ grid[0,0] & i = 1 \wedge j = 1 \end{cases}$$

The implementation:

```
int minPathSum(vector<vector<int>>& grid) {
  const int m = grid.size();
  const int n = grid[0].size();
  const int inf = m * n * 100 + 1;
  auto d = vector<vector<int>>(m + 1, vector<int>(n + 1, inf));
  // To make sure d[1][1] = grid[0][0]:
  d[0][1] = d[1][0] = 0;
  for (int i = 0; i < m; ++i)
    for (int j = 0; j < n; ++j)
```

```
      d[i + 1][j + 1] = min(d[i][j + 1], d[i + 1][j]) + grid[i][j];
  return d[m][n];
}
```

To save memory:

```
int minPathSum(vector<vector<int>>& grid) {
  const int m = grid.size();
  const int n = grid[0].size();
  const int inf = m * n * 100 + 1;
  auto d = vector<int>(n + 1, inf);
  // To make sure d[1][1] = grid[0][0]:
  d[1] = 0;
  for (int i = 0; i < m; ++i)
    for (int j = 0; j < n; ++j)
      d[j + 1] = min(d[j + 1], d[j]) + grid[i][j];
  return d[n];
}
```

1.1.24 Simplify Path

This is an implementation problem. To solve this problem we are going to tokenize the input. The delimiter is '/' character:

```
string simplifyPath(string path) {
  int i = 0;
  vector<string> tokens;
  while (i < path.length())
  {
    if (path[i] == '/')
    {
      ++i;
      continue;
    }
    auto pos = path.find('/', i);
    if (pos == path.npos)
      pos = path.length();
    auto token = path.substr(i, pos - i);
    if (token == "..")
    {
      if (!tokens.empty())
        tokens.erase(prev(tokens.end()));
    }
    else if (token != ".")
      tokens.push_back(token);
    i = pos + 1;
  }
  string res;
  for (const auto& token : tokens)
    res += "/" + token;
  return res.empty() ? "/" : res;
}
```

1.1.25 Set Matrix Zeroes

$O(m+n)$ space complexity

```
void setZeroes(vector<vector<int>>& matrix) {
  const auto m = matrix.size();
  const auto n = matrix[0].size();

  auto row_zero = vector<bool>(m, false);
  auto col_zero = vector<bool>(n, false);

  for (int r = 0; r < m; ++r)
    for (int c = 0; c < n; ++c)
      if (matrix[r][c] == 0)
        row_zero[r] = col_zero[c] = true;
  for (int r = 0; r < m; ++r)
    for (int c = 0; c < n; ++c)
      if (row_zero[r] || col_zero[c])
        matrix[r][c] = 0;
}
```

$O(1)$ space complexity

```
void setZeroes(vector<vector<int>>& matrix) {
  const auto m = matrix.size();
  const auto n = matrix[0].size();
  //col0 plus matrix[0][1..n - 1] handle columns
  //matrix[0..m-1][0] handle rows
  int col0 = matrix[0][0];

  for (int r = 0; r < m; ++r)
  {
    //Handling column 0:
    if (matrix[r][0] == 0)
      col0 = 0;
    // We start from c = 1 to handle
    // rows 0 to m - 1 and columns 1 to n - 1
    for (int c = 1; c < n; ++c)
      if (matrix[r][c] == 0)
        matrix[r][0] = matrix[0][c] = 0;
  }
  // We start from m - 1 to 0 to avoid resetting
  // The first row too soon. As an example:
  // 0 1
  // 1 1
  // If we start first from r = 0, all elements of the first row
  // become zero, so the entire elements would become zero
  for (int r = m - 1; r >= 0; --r)
  {
    // We handle c = 0 later
    for (int c = 1; c < n; ++c)
      if (matrix[r][0] == 0 || matrix[0][c] == 0)
        matrix[r][c] = 0;
    // Note that we should handle matrix[r][0] after we handled
    // matrix[r][c] for c > 0
```

```
    if (matrix[r][0] == 0 || col0 == 0)
      matrix[r][0] = 0;
  }
}
```

1.1.26 Search a 2D Matrix

Given a 2D array $rows \times columns$. Each row is sorted in non-descendaing order. The first element of the next row is greater than the last element in current row.

For a similar problem reft to 1.1.11.

Method 1

We define $r(t, b)$ that knows in which row the target is located. Similarly we define $c(l, r)$ that knows in which column the target is located. Both r and c are defined like f in 1.1.11. We assume $m_r = \lfloor \frac{t+b}{2} \rfloor$:

$$r(t,b) = \begin{cases} r(t, m_r) & target \leq matrix[m_r, columns - 1] \\ r(m_r + 1, b) & target > matrix[m_r, columns - 1] \\ m_r & matrix[m_r][0] \leq target \leq matrix[m_r][columns - 1] \end{cases}$$

Let's assume $q = r(0, rows - 1)$ and $m_c = \lfloor \frac{l+r}{2} \rfloor$:

$$c(l,r) = \begin{cases} c(l, m_c) & target \leq matrix[q, m_c] \\ c(m_c + 1, r) & target > matrix[q, m_c] \\ m_c & l = r \wedge target = matrix[q, m_c] \end{cases}$$

```
bool searchMatrix(vector<vector<int>>& matrix, int target) {
  const auto rows = matrix.size();
  const auto cols = matrix[0].size();
  int top_r = 0, bottom_r = rows - 1;
  while (top_r < bottom_r)
  {
    auto mid = (top_r + bottom_r) / 2;
    if (target <= matrix[mid][cols - 1])
      bottom_r = mid;
    else
      top_r = mid + 1;
  }
  int left = 0, right = cols - 1;
  while (left < right)
  {
    auto mid = (left + right) / 2;
    if (target <= matrix[bottom_r][mid])
      right = mid;
    else
      left = mid + 1;
  }
  return matrix[bottom_r][right] == target;
}
```

Method 2

We consider $matrix[0..rows-1][0..columns-1]$ as a one dimentional matrix $g[0..rows*columns-1]$:

```
bool searchMatrix(vector<vector<int>>& matrix, int target) {
  const auto rows = matrix.size();
  const auto cols = matrix[0].size();
  const auto extract_row_col = [rows, cols](int index)
  {
    int row = index / cols;
    int col = index % cols;
    return make_tuple(row, col);
  };
  int left = 0, right = rows * cols - 1;
  //Since we compare for equality inside the loop,
  //we use <= instead of < operator:
  while (left <= right)
  {
    auto mid = (left + right) / 2;
    int r, c;
    tie(r, c) = extract_row_col(mid);
    if (target == matrix[r][c])
      return true;
    else if (target < matrix[r][c])
      right = mid - 1;
    else
      left = mid + 1;
  }
  return false;
}
```

1.1.27 Combinations

Given two integers n and k, return all possible combinations of k numbers out of the range $[1, n]$.

We can use mathmatical induction. Suppose we know the answer $c(n, m)$ for $m < k$ and for all possible n. $m = 1$ is trivial. Now consider $c(n, k)$. We use the hypothesis and find all the answers for $c(n-1, k-1)$. The maximum element in each of these combination is no more than $n-1$, so we can easily extend them to k elements.

Recursion

The tricky part is the range of valid numbers in each depth in recursive tree. Let's say $c = [n_{i_0}, \ldots, n_{i_{k-1}}]$ is an arbitrary combination. $depth_j$ is responsible to find n_{i_j}. After n_{i_j}, we should find $n_{i_{j+1}}$ to $n_{i_{k-1}}$. So there are $(k-1)-(j+1)+1 = k-j-1$ reamining elements. So $n_{i_j}+k-j-1 \leq n \implies n_{i_j} \leq n-k+j+1$

```
class Solution {
  public:
  void f(int start, int depth)
  {
```

```
    if (depth == k)
    {
      res.push_back(cur);
      return;
    }
    for (int i = start; i <= (n - k + depth + 1); ++i)
    {
      cur[depth] = i;
      f(i + 1, depth + 1);
    }
  }
  vector<vector<int>> combine(int n, int k) {
    res.clear();
    cur = vector<int>(k);
    this->n = n;
    this->k = k;
    f(1, 0);
    return res;
  }
  vector<vector<int>> res;
  vector<int> cur;
  int n, k;
};
```

Iterative

Suppose we are in i step. After i there are $k-1-(i+1)+1 = k-i-i$ remaining elements. So we have $c[i] + k - i + 1 \leq n \implies c[i] \leq n - k + i + 1$:

```
vector<vector<int>> combine(int n, int k) {
  vector<int> c(k);
  vector<vector<int>> res;
  int i = 0;
  c[0] = 0;
  while (i >= 0)
  {
    if (i == k)
    {
      res.push_back(c);
      --i;
    }
    else if (c[i] < n - k + i + 1)
    {
      ++c[i];
      if (i + 1 < k)
        c[i + 1] = c[i];
      ++i;
    }
    else
      --i;
  }
  return res;
}
```

1.2 Hard

1.2.1 Longest Valid Parentheses

Given a string containing just the characters '(' and ')', return the length of the longest valid (well-formed) parentheses substring.

Dynamic programming approach

Let's assume s is the string and $dp[i]$ is the length of maximum valid parentheses that ends with $s[i]$. We use mathematical induction. We assume we know the answers for all s with length less than i and now we want to find the answer for all strings of length i. There are two cases that we need to consider:

$$dp[i] = \begin{cases} dp[i-2]+2 & s[i] =) \wedge s[i-1] = (\\ dp[i-1]+2+dp[i-dp[i-1]-2] & s[i] =) \wedge s[i-1] =) \wedge s[i-dp[i-1]-1] = (\\ 0 & i \leq 0 \end{cases}$$

An example of the first case is "()()" and an example of the second case is "(())".

Stack approach

We assume the bottom of stack keeps the index of the last unmatched ")". The rest are all unmatched "(". For simplicity we assume all strings has a ")" at index -1. Here is the algorithm:

- If $s[i] = ($, then we add it on top of the stack
- If $s[i] =)$, then we can match it to the last unmatched "(" which may be on top of the stack. For doing that first we pop the last element from the stack:
  - If the stack is empty, then it means that there is no more "(" and we popped the last unmatched ")" so we need to insert the index of the current unmatched ")" into the stack. Please not that the the bottom element of the stack should always be the index of the last unmatched ")"
  - If the stack is non-empty, then it means that the element that we popped was the last unmatched "(". We need to check the difference between $s[i]$ and the index of the top element of stack with the current answer

Two-pass linear scan

Let's assume we want to find the longest valid parentheses in $s[0..n]$. Let's assume $s[i..j]$ is a non-extendable valid parentheses for $i > 0 \wedge j < n \wedge i < j$. We need to consider 4 cases:

1. $(s[i..j])$: This is in contrast that $s[i..j]$ is non-extendable. Because $(s[i..j])$ is a valid and longer parentheses
2. $(s[i..j]($: The first and last open parentheses should remain unmatched. Otherwise, $s[i..j]$ is extendable
3. $)s[i..j])$: The first and last open parentheses should remain unmatched. Otherwise, $s[i..j]$ is extendable
4. $)s[i..j]($: The first and last open parentheses should remain unmatched. Otherwise, $s[i..j]$ is extendable

We know that in a valid parentheses, the number of left and right ones are equal. To solve cases 3 and 4 we need to start scaning s from left to right and start counting left and right ones. If they are equal, it's a potential answer. If the number of right parentheses is more than left, we find an umatched right parenthesis so we should reset both left and right. This phase can detect solutions for case 3 and 4.

For detecting case 2, We need to scan from right to left. If the number of left and right parentheses are equal, then it's a potential solution. If the number of left parentheses is greater than right, then we find an unmathced left parenthesis, so we should reset the counters.

Chapter 2

HackerRank

2.1 New Year Chaos

We define $index_i$ as the current index for person i. For example if we have $1, 2, 3, 4$ and 4 bribes 3, the queue looks like $1, 2, 4, 3$. So $index_4 = 3$. Since no body can bribe more than 2 times, $index_i \geq i - 2$ for $1 \leq i \leq n$. Consider person n. No body can bribe that person. So $n - 2 \leq index_n \leq n$. After we retruned that person to his actual place we can consider $n - 1$. So we have $n - 3 \leq index_{n-1} \leq n - 1$ (note that at this moment $index_n = n$).

```
void minimumBribes(vector<int> q) {

    const auto& n = q.size();
    int res = 0;
    for (int num = n; num > 0; --num)
    {
        for (int i = max(0, num - 3); i < num - 1; ++i)
        {
            if (q[i] == num)
            {
                ++res;
                swap(q[i], q[i + 1]);
            }
        }
        if (q[num - 1] != num)
        {
            cout << "Too chaotic" << endl;
            return;
        }
    }
    cout << res << endl;
}
```

2.2 Minimum Swaps 2

Note that this solution is based on Selection Sort in which the number of swaps are minimum. According to Wikipedia: "One thing which distinguishes selection sort from other sorting algorithms is that it makes the minimum possible number of swaps, n − 1 in the worst case.". Altourh Selection sort has minimum number of swaps among all sorts agorithms, it has $O(n^2)$ comparisons. Since the final result is $\{1, 2, \ldots, n\}$, it's like we have the set in sorted order so we can bypass comparisons and use Selection Sort advantage which is the minimum number of swaps.

We define $index_i$ as the current index of number i. Suppose we have n numbers, so $1 \leq index_i \leq n$. The goal is to have $index_i = i$. Without losing generality suppose $i < j \wedge index_i = j$. There are two cases to consider:

1. If $index_j = i$, then by swapping arr_i and arr_j, we put both i and j in their corresponding positions.

2. If $index_j = k \wedge k \neq i \wedge k \neq j$. In this case by swapping arr_i and arr_j we only put i in its corresponding position. So we need to do an extra swap to put j in its correct position.

We can start from $i = 1$ to $i = n$ and make sure i is in correct position; otherwise we perform a swap. In each iteration we fix the position of one or two numbers. A good example is $\{4, 3, 2, 1\}$.

```
    int minimumSwaps(vector<int> arr) {

    const auto& n = arr.size();
    vector<int> index(n + 1);

    for (int i = 0; i < n; ++i)
        index[arr[i]] = i;
    int cnt = 0;
    for (int num = 1; num <= n; ++num)
    {
        if (index[num] != num - 1)
        {
            ++cnt;
            index[arr[num - 1]] = index[num];
            swap(arr[index[num]], arr[num - 1]);
            index[num] = num - 1;
        }
    }
    return cnt;
}
```

2.3 Count Triplets

We use dynamic programming to solve it. For mathematical induciton we define $cnt[num][n]$ like this:

$cnt[a_{i_1}][0] = |\{a_{i_0} \in arr \mid a_{i_1} = a_{i_0} \times r \ \wedge \ i_1 < i_2\}|$
$cnt[a_{i_2}][1] = |\{(a_{i_0}, a_{i_1}) \in arr \times arr \mid a_{i_k} = a_{i_{k-1}} \times r \ \wedge \ i_{k-1} < i_k \text{ for } 1 \leq k \leq 2\}|$

So the final answer is:

$$\sum_{n \in arr} cnt[n][1]$$

Then for each number n we have

$$cnt[n \times r][0] = cnt[n \times r][0] + 1$$
$$cnt[n \times r][1] = cnt[n \times r][1] + cnt[n][0]$$

Since $r = 1$, the order of assignments are very important.

```
long countTriplets(vector<long> arr, long r) {
    const auto n = arr.size();
    unordered_map<long, array<long, 2>> cnt;
    //cnt[a[j]][0] = |{a[i]}| in which i < j and
    //               a[j] = a[i] * r
    //cnt[a[k]][1] = |{a[i], a[j]}| in which
    //               i < j < k and
    //a[k] = a[j] * r and a[j] = a[i] * r

    long res = 0;
    for (const auto& num : arr)
    {
        res += cnt[num][1];
        const auto next = num * r;
        cnt[next][1] += cnt[num][0];
        ++cnt[next][0];
    }
    return res;
}
```

2.4 Fraudulent Activity Notifications

Basically we want a $O(nlogn)$ algorithm to find median of a sequuence, when we removed the first element and add another one. So we need two binary search trees. In the first one the maximum element is the median itself and in the secon one the minimum element is the second median in case of $d = 2k$ or a value greater than median when $d = 2k + 1$. So if $d = 2k$ both of these binary search trees always have k element. When $d = 2k + 1$, the first one always has $k + 1$ elements and the second one has k elements. Let's call them $lessEqual$ and $greaterEqual$.

If both removing element and new element belong to the same tree, nothing extra is required. So we only need to remove one element and add the new one. If the removing element is from $lessEqual$, we must remove the minimum

element from *greaterEqual* and add it to *lessEqual*. If the removing element is from *greaterEqual*, we must remove the maximum element from *lessEqaul* and add it to *greaterEqual*. By doing that the maximum element is *lessEqual* is median. In case of $d = 2k$, the minimum element in *greaterEqual* is the second median. The running time of this algorithm is $O(nlogn)$.

```
int activityNotifications(vector<int> expenditure, int d)
{
    multiset<int, greater<int>> lessEqual;
    multiset<int> greaterEqual;

    vector<int> init(d);
    copy(expenditure.begin(), expenditure.begin() + d,
         init.begin());
    sort(init.begin(), init.end());

    const bool isEven = (d & 1) == 0;

    int medianIndex = (d - 1) / 2;
    int i;
    for (i = 0; i <= medianIndex; ++i)
        lessEqual.insert(init[i]);
    for (; i < d; ++i)
        greaterEqual.insert(init[i]);

    int res = 0;
    for (int i = d; i < expenditure.size(); ++i)
    {
        const int median1 = *lessEqual.begin();
        if (isEven)
        {
            const int median2 = *greaterEqual.begin();
            if (expenditure[i] >= (median1 + median2))
                ++res;
        }
        else
        {
            if (expenditure[i] >= 2 * median1)
                ++res;
        }

        const auto removed = expenditure[i - d];

        if (removed <= median1 &&
            expenditure[i] <= median1)
        {
            lessEqual.erase(lessEqual.find(removed));
            lessEqual.insert(expenditure[i]);
        }
        else if (removed > median1 &&
                 expenditure[i] > median1)
        {
          greaterEqual.erase(greaterEqual.find(removed));
          greaterEqual.insert(expenditure[i]);
        }
        else if ( removed <= median1)
        {
```

```
            //For handling d=1, it should first:
            greaterEqual.insert(expenditure[i]);
            lessEqual.erase(lessEqual.find(removed));
            lessEqual.insert(*greaterEqual.begin());
            greaterEqual.erase(greaterEqual.begin());
        }
        else
        {
          //For handling d=1, it should be first:
          lessEqual.insert(expenditure[i]);
          greaterEqual.erase(greaterEqual.find(removed));
          greaterEqual.insert(*lessEqual.begin());
          lessEqual.erase(lessEqual.begin());
        }
    }
    return res;
}
```

2.5 Merge Sort: Counting Inversions

Problem Statement

We can solve it using merge sort. Suppose we have $arr[left..right]$. We break it into two subproblem $arr[left..mid]$ and $arr[mid+1..right]$. Both of them are sorted. According to merge sort algorithm $left \leq i \leq mid$ and $mid+1 \leq j \leq right$. In other words we already put $arr[left..i-1]$ and $arr[mid+1..j-1]$ into their correct positions. So when $arr[j] < arr[i]$, it means $arr[j] < arr[i] \leq arr[x]$ in which $i+1 \leq x \leq mid$. So we need to have $mid - i + 1$ swaps.

```
long mergeSort(vector<int>& arr, int leftIndex,
               int rightIndex)
{
    if (leftIndex == rightIndex)
        return 0;

    long res = 0;
    int midIndex = (leftIndex + rightIndex) / 2;
    res = mergeSort(arr, leftIndex, midIndex);
    res += mergeSort(arr, midIndex + 1, rightIndex);

    vector<int> sorted(rightIndex - leftIndex + 1);
    int i, j, k;
    for (i = leftIndex, j = midIndex + 1, k = 0;
         i <= midIndex && j <= rightIndex;)
    {
        if (arr[i] <= arr[j])
            sorted[k++] = arr[i++];
        else
        {
            res += midIndex - i + 1;
            sorted[k++] = arr[j++];
        }
    }
```

```
    if (i <= midIndex)
        copy(arr.begin() + i,
             arr.begin() + midIndex + 1,
             sorted.begin() + k);
    else
        copy(arr.begin() + j ,
             arr.begin() + rightIndex + 1,
             sorted.begin() + k);
    copy(sorted.begin(), sorted.end(),
         arr.begin() + leftIndex);
    return res;
}
// Complete the countInversions function below.
long countInversions(vector<int> arr) {
    return mergeSort(arr, 0, arr.size() - 1);
}
```

Chapter 3

TopCoder

3.1 SRM 428

3.1.1 ThePalindrome

For another variation refer to 1.1.2. We want to add the minimum number of characters to the end of string to make it a palindrome. The straightforward approach is to try add the first i characters in reverse for all $0 \leq i \leq n-1$ in which n is the length of string. So we start from $i = 0$ and check whether the string is palindrome. If it's not we check for $i = 1$ and so on. The running time of this algorithm is $O(n^2)$.

```
bool isPalindrome(const string& str)
{
  int left = 0, right = str.length() - 1;
  for (; left < right && str[left] == str[right];
         ++left, --right);
  return left >= right;
}

int find(string s)
{
  for (int i = 0; i < s.length(); ++i)
  {
    string tmp = s + string(s.rend() - i, s.rend());
    if (isPalindrome(tmp))
      return tmp.length();
  }
  throw 1;
}
```

Since $n \leq 50$, this algorithm is fast. We can make it $O(n \log_2 n)$ if we use binary search tree to find the minimum i.

There is another approach. Let's assume we have string $S = s_1 s_2 \dots s_n$. We define $S' = s_n \dots s_2 s_1$. Suppose we can write string S as QP. In other words, S is the concatenation of two strings Q and P. We assume P is palindrome but

Q is not. We can make S palindrome if we convert QP to QPQ' we call this new String Z. Z is palindrome because if we reverse it we have:

$$\begin{aligned} Z: &\quad QPQ' \\ Z': &\quad QP'Q' \end{aligned}$$

Since we want the length of Q be minimum, we must find the maximal P:

```
bool isPalindrome(const string& str, int start)
{
  int left = start, right = str.length() - 1;
  for (; left < right && str[left] == str[right];
         ++left, --right);
  return left >= right;
}

int find(string s)
{
  for (int i = 0; i  < s.length(); ++i)
  {
    if (isPalindrome(s, i))
      return s.length() + i;
  }
  throw 1;
}
```

As the previous implementation the running time is $O(n^2)$ but it's easy to convert it to $O(n \log_2 n)$ using binary search.

This implementation has a unique feature. We can convert it to an $O(n)$ algorithm using KMP algorithm. Suppose $S = QP$ where P is a palindrome. We want to find palidnrome postfix P which its length is maximum among all palindrome post-fixes. We need to run KMP pre-compute calculation on $S' = P'Q'$. Then we run KMP algorithm as if we want to find whether S' is a substring of S. Suppose we use i as an index for S and j as an index for S'. The algorithm start with $i = 0$ and ends when $i = len(S)$ in which $S'[0..j-1]$ is P' or P (since it's palindrome).

```
vector<int> calculateNext(const string& B)
{
  vector<int> next(B.length());
  next[0] = -2;
  next[1] = -1;
  int i, j;
  for (i = 2; i < B.length(); ++i)
  {
    j = next[i - 1] + 1;
    for (; j >= 0 && B[j] != B[i - 1]; j = next[j] + 1);
    next[i] = j;
  }
  return next;
}

int find(string A)
{
  const string B = string(A.rbegin(), A.rend());
```

```
  const auto next = calculateNext(B);
  int i, j;
  for (i = 0, j = 0; i < A.length() && j < B.length();)
  {
    if (A[i] == B[j])
      ++i, ++j;
    else if ((j = next[j] + 1) < 0)
    {
      //Since B.front() == A.back(), it's impoosible
      //i == A.length() here:
      ++i, j = 0;
      j = 0;
    }
  }
  int palindromeLen = j;
  return A.length() + A.length() - palindromeLen;
}
```

Chapter 4

Miscellaneous

4.1 Painting a pole

Suppose we have a pole of length 10. Each time we select an interval of length 1 and paint the entire interval. The goal is to paint the entire pole. For example we select interval $[1.5, 2.5]$ and paint it and so on. We have a stream of paint operation. Each paint operation choose an interval and paint it. So we should have at least tow functions $voidpaint(doublepaint_point)$ and $boolis_painted()$. The former select an interval and paint it. The latter examine whether the entire pole is painted.

This is a special case of interval merging problem. For more information refer to 1.1.19. We can call *paint* many times. The trivial solution is each time this function is called, we try to add the interval and merge with others if possible. It's easier this set of intervals is sorted based on the start point of intervals. Since we can add new interval using *paint* function, keep the list sorted can be challenging.

Since the length of pole is 10 and each interval has the length of 1, we can create 10 segments (buckets) ($\frac{10}{1} = 10$. If the interval length is 2, then we create 5 segments: $\frac{10}{2} = 5$).

Since the length of both intervals and segments are 1, each segment has at most two intervals. For i^{th} segment we define the following variables:

- $left_i$: means $[left_i, i + 1]$ is painted
- $right_i$: means $[i, right_i]$ is painted

Before painting starts, the i^{th} segment has values $left_i = i + 1$ and $right_i = i$. We can examine it's entirely painted if $left_i \leq right_i$.

```
struct Segment
{
  //From left to the end of segment:
  int left;
  //From the beginning to the right of segment
```

```
  int right;
};

vector<Segment> segments(10);

void init()
{
  for (int i = 0; i < segments.size(); ++i)
  {
    segments[i].left = i + 1;
    segments[i].right = i;
  }
}

void paint(double paint_point)
{
  if (paint_point < 0 || paint_point > 10)
    return;
  int segment_index = floor(paint_point);
  int next_segment_index = floor(paint_point + 1);
  auto& left = segments[segment_index].left;
  if (paint_point < left)
    left = paint_point;
  if (next_segment_index >= 10)
    return;
  auto& right = segments[next_segment_index].right;
  if ((paint_point + 1) > right)
    right = paint_point + 1;
}

bool is_painted()
{
  for (const auto& segment : segments)
    if (segment.left > segment.right)
      return false;
  return true;
}
```

4.2 Average of last k numbers

Suppose we have a list of integers of size n. We want to calculate the average of the last k numbers. You should define two functions *insert* and *average*. The former is called n times. The latter can be called arbitary. The running time of both function should be $O(1)$ and the memory complexity shouldn't be higher than $O(k)$.

To calculate the sum of the last k numbers in real time, first we initialize the array with 0. Before inserting the new value, we subtract the current value from sum and then add the new value to sum:

```
class List
{
public:
  List(std::size_t size): size{size},
```

```
    numbers{std::vector<int>(size, 0)},
    begin{0},
    end{0},
    sum{0} {}

  void insert(int val)  //O(1)
  {
    sum -= numbers[end];
    sum += val;
    numbers[end] = val;
    end = (end + 1) % size;
    if (end == begin)
      begin = (begin + 1) % size;
  }

  double average()  //O(1)
  {
    auto cur_size = end > begin ? end - begin : size;
    return static_cast<double>(sum) / cur_size;
  }
private:
  std::size_t size, begin, end;
  std::vector<int> numbers;
  int sum;
};

void test1()
{
  std::vector<int> numbers{1, 2, 3, 4};
  auto l = List(1);
  for (const auto val : numbers)
    l.insert(val);
  assert(l.average() == 4);
}

void test2()
{
  std::vector<int> numbers{1, 2, 3, 4};
  auto l = List(2);
  for (const auto val : numbers)
    l.insert(val);
  assert(l.average() == 3.5);
}

int main()
{
  test1();
  test2();
}
```

Appendix A

Algorithms

A.1 Dynamic Programming

A.1.1 Knapsack problem

Given a set of items, each with a weight and a value, determine which items to include in the collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

0-1 Knapsack

In this variation an item can only be taken once or not. More precisely, given a set of n items numbered from 1 to n, each with a weight w_i and a value v_i, along with a maximum weight capacity W. We want to maximize $\sum_{i=1}^{n} v_i \times x_i$ subject to $\sum_{i=1}^{n} v_i \times x_i \leq W \wedge x_i \in \{0, 1\}$.

Here x_i represents the number of instances of item i to include in the knapsack. Informally, the problem is to maximize the sum of the values of the items in the knapsack so that the sum of the weights is less than or equal to the knapsack's capacity.

To solve this problem we assume $dp_{i,j}$ is the maximum result using the first i items that are indexed from 0 to $i-1$ with $W = j$. We started index from 0 because it's eaiser to implement it in a programming language like C++

$$dp_{i,j} = \begin{cases} max(dp_{i-1,j}, dp_{i-1,j-w_{i-1}} + v_{i-1}) & j \geq w_{i-1} \\ dp_{i-1,j} & j < w_{i-1} \\ 0 & i = 0 \vee j = 0 \end{cases}$$

For implementation we can optimize the memory usage. There is no need to have a two-dimenstional array for dp. For calculating $dp_{i,j}$, we need to refer to $i - 1^{\text{th}}$ row. Let's define $dp[j]$ which has the maximum result for $0 \leq j \leq W$ for the first $i - 1$ items. Now we want to update it for the first i items. Please note that it's very important the update should be done from **right to left**. Otherwise, it's not equvalent to above formula.

```cpp
int knapsack(const vector<int>& w, const vector<int>& v, int W)
{
        vector<int> dp(W + 1, 0);
        for (size_t i = 0; i < w.size(); ++i)
          for (size_t j = W; j >= w[i]; --j) // From right to left
            dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
        return dp[W];
}
```

The time complexity is $O(n \times W)$ and memory complexity is $O(W)$. n is the number of items and W is weight capacity.

Unbounded Knapsack (UKP)

It places no upper bound on the number of copies of each kind of item. More preceisely, we want to maximize $\sum_{i=1}^{n} v_i \times x_i$ subject to $\sum_{i=1}^{n} w_i \times x_i \leq W \wedge x_i \in \mathbf{N}$.

Let's define $dp_{i,j}$ like the one that we defined for 0-1 Knapsack. It has the maximum result from the first i items that are indexed from 0 to $i-1$:

$$dp_{i,j} = \begin{cases} \max(dp_{i-1,j}, dp_{i,j-w_{i-1}} + v_{i-1}) & j \leq w_{i-1} \\ dp_{i-1,j} & j > w_{i-1} \\ 0 & i = 0 \vee j = 0 \end{cases}$$

Like 0-1 knapsack, we can optmize the memory usage. We define $dp[j]$ that has the maximum result for $0 \leq j \leq W$ for the first $i-1$ items, we want to update it for the first i items. In the the above formula, $dp_{i,j}$ access an element in the $i-1^{\text{th}}$ row or an element in the left side of the current row. **Unlike the 0-1 knapsack**, it's important that we update from **left to right**.

```cpp
int knapsack(const vector<int>& w, const vector<int>& v, int W)
{
        vector<int> dp(W + 1, 0);
        for (size_t i = 0; i < w.size(); ++i)
          for (size_t j = w[i]; j <= W; ++j) // From left to right
            dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
        return dp[W];
}
```

The only difference between this implementation and 0-1 knapsack is **the inner loop is from left to right!**

The time complexity is $O(n \times W)$ and memory complexity is $O(W)$. n is the number of items and W is weight capacity.

Bounded Knapsack (BKP)

It removes the restriction that there is only one of each item, but restricts the numberx_i of copies of each kind of item to a maximum non-negative integer value c. More precisely, we want to maximize $\sum_{i=1}^{n} v_i \times x_i$ subject to $\sum_{i=1}^{n} w_i \times x_i \leq W \wedge x_i \in \{0, 1, 2, \ldots, c_i\}$.

Let's define $dp_{i,j}$ like the one that we defined for 0-1 Knapsack. It has the maximum result from the first i items that are indexed from 0 to $i-1$:

$$dp_{i,j} = \begin{cases} \max_{k=1}^{c_{i-1}} (dp_{i-1,j}, dp_{i-1,j-k\times w_{i-1}} + k \times v_{i-1}) & j \geq k \times w_{i-1} \\ dp_{i-1,j} & j < w_{i-1} \\ 0 & i = 0 \vee j = 0 \end{cases}$$

Like 0-1 knapsack, we can optmize the memory usage. We define $dp[j]$ that has the maximum result for $0 \leq j \leq W$ for the first $i-1$ items, we want to update it for the first i items. In the the above formula, $dp_{i,j}$ access an element in the $i-1^{\text{th}}$. **Like the 0-1 knapsack**, it's important that we update from **right to left**.

```
int knapsack(const vector<int>& w, const vector<int>& v, const
    vector<int>& c, int W)
{
        vector<int> dp(W + 1, 0);
        for (size_t i = 0; i < w.size(); ++i)
          for (size_t j = W; j >= w[i]; --j) // From right to left
            for (size_t k = 1; k <= c[i] && j >= (k * w[i]); ++k)
              dp[j] = max(dp[j], dp[j - k * w[i]] + k * v[i]);
        return dp[W];
}
```

The time complexity is $O(n \times W \times C)$. n is the number of items. W is the weight capacity. $C = \max_{i=0}^{n-1} c_i$. The memory complexity is $O(W)$.

Bounded Kanspack through binary decomposition

As explaind in previous seciton, $O(n \times W \times C)$ is slow. We can improve bounded knapsack solution using binary decomposition and convert the problem into 0-1 knapsack.

By looking at $\max_{k=1}^{c_{i-1}} (dp_{i-1,j}, dp_{i-1,j-k\times w_{i-1}} + k \times v_{i-1})$, we can realize that it's similair to 0-1 knapsack if we break iterm $i-1$ into c_{i-1} items such as $w'_j = j \times w_{i-1}$ and $v'_j = j \times v_{i-1}$ for all $1 \leq j \leq c_{i-1}$.

Instead of breaking item $i-1$ into c_{i-1} items, we can break it into $\lfloor \log_2(c_{i-1}) \rfloor$.

Let's assume $s_k = \sum_{i=0}^{k} 2^i$. Let's say $S = s_k$ in such a way that $s_{k+1} > c_{i-1}$. With $b_k \times 2^k + b_{k-1} \times 2^{k-1} \ldots b_1 \times 2 + b_0$ assuming $b \in \{0, 1\}$ we can create all numbers that are less than or equal to S. Let's assume $r = c_{i-1} - S$. Let's consider $S + q$ for $1 \leq q \leq r$ copies of item $i-1$ is required to get the maximum solution. Since $S + q - r \leq S$, we can create it using $b_k \times 2^k + \cdots + b_0$. Then we add r to it to create $S + q$.

For example for $c_{i-1} = 10$, we have $10 = 1+2+4+3$. Note that we shouldn't add 8 to the set because it's possible we exceed 10.

```
int knapsack(const vector<int>& w, const vector<int>& v, const
    vector<int>& c, int W)
{
        vector<int> dp(W + 1, 0);
        for (size_t i = 0; i < w.size(); ++i)
```

```
        {
                int r = c[i];
                for (int k = 1; k <= r; r -= k, k << 1)
                {
                        int new_w = w[i] * k;
                        int new_v = v[i] * k;
                        for (int j = W; j >= new_w; --j) // from
                            right to left
                        dp[j] = max(dp[j], dp[j - new_w] + new_v);
                }
                if (r > 0)
                {
                        int new_w = w[i] * r;
                        int new_v = v[i] * r;
                        for (int j = W; j >= new_w; --j) // from
                            right to left
                        dp[j] = max(dp[j], dp[j - new_w] + new_v);
                }
        }
        return dp[W];
}
```

The time complexity is $O(n \times W \times \log_2 C)$. n is the number of items. W is the weight capacity. $C = \max_{i=0}^{n-1} c_i$. The memory complexity is $O(W)$.

Appendix B

C++ refresher

B.1 std::tuple

B.1.1 std::tie

Creates a tuple of lvalue references to its arguments or instances of std::ignore

B.2 Associative containers

B.2.1 std::set

```
#include <iostream>
#include <algorithm>
#include <cassert>
#include <set>
#include <tuple>
#include <array>

int main()
{
  std::set<int> s({1, 2, 3, 4, 5, 6, 7});
  assert(s.count(7) == 1);
  assert(s.count(8) == 0);
  bool result;
  std::set<int>::iterator iter;
  std::tie(iter, result) = s.insert(8);
  assert(result == true && *iter == 8 && s.count(8) == 1);
  std::tie(std::ignore, result) = s.insert(9);
  assert(result = true && s.count(9) == 1);

  std::array<int, 6> a = {10, 11, 12, 13, 14, 15};
  s.insert(a.begin(), a.end());
  assert(s.count(10) == 1);

  iter = s.find(13);
  assert(iter != s.end());
```

```
  iter = s.lower_bound(4);
  assert(*iter == 4);
  iter = s.upper_bound(4);
  assert(*iter == 5);

  s = {1, 2, 4, 5};
  iter = s.lower_bound(3);
  assert(*iter == 4);
  iter = s.upper_bound(3);
  assert(*iter == 4);
}
```

B.3 Inserting into a container

The following code uses most insert operations:

```
#include <iostream>
#include <vector>
#include <deque>
#include <iterator>
#include <algorithm>

template<typename Container>
void print(const Container& c)
{
  static int id = 1;
  std::cout << id++ << ": ";
  std::for_each(c.begin(), c.end(), [](const auto& val) {
                std::cout << val << ' ';
                });
  std::cout << std::endl;
}

int main()
{
  std::vector<int> base{0, 1, 2, 3, 4};
  auto v(base);
  std::vector<int> v2{100, 200, 300};

  auto i = std::next(v.begin()); // v.begin() + 1
  i = v.insert(i, v2[0]);
  i = v.insert(i, v2[1]);
  i = v.insert(i, v2[2]);
  print(v); // 1: 0 300 200 100 1 2 3 4
  // Note that after
  // v.insert(i, v[0]);
  // i becomes invalid. So this is undefined behavior:
  // v.insert(i, v[1]);
  v = base;
  i = std::next(v.begin()); // v.begin() + 1
  i = v.insert(i, v2.begin(), v2.end());
  print(v); // 2: 0 100 200 300 1 2 3 4

  v = base;
```

```
  std::copy(v2.begin(), v2.end(),
            std::inserter(v, std::next(v.begin(), 1)));
  print(v); // 3: 0 100 200 300 1 2 3 4

  v = base;
  std::copy(v2.begin(), v2.end(), std::back_inserter(v));
  print(v); // 4: 0 1 2 3 4 100 200 300

  //std::vector doesn't have push_front:
  std::deque<int> d(base.begin(), base.end());
  std::copy(v2.begin(), v2.end(), std::front_inserter(d));
  print(d); // 1: 300 200 100 0 1 2 3 4
}
```

B.4 std::vector iterators

It uses LegacyRandomAccessIterator. For more information refer to Iterator library.

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <vector>

template<typename Container>
void print(const Container& c)
{
  const auto m_print = [](const auto& val)
  {
    std::cout << val << ' ';
  };
  std::for_each(c.begin(), c.end(), m_print);
  std::cout << std::endl;
}

//my implementation of std::reverse
template<typename Container>
void my_reverse(Container& c)
{
  auto left = c.begin();
  auto right = std::prev(c.end());
  for (; left < right; ++left, --right)
    std::iter_swap(left, right);
}

template<typename Container>
void my_reverse2(Container& c)
{
  auto left = 0;
  auto right = std::distance(c.begin(), std::prev(c.end()));
  for (; left < right; ++left, --right)
    std::swap(c[left], c[right]);
}

int main()
```

```
{
  std::vector<int> v{0, 1, 2, 3, 4};
  my_reverse(v);
  print(v); // 4 3 2 1 0
  my_reverse2(v);
  print(v); // 0 1 2 3 4
}
```

B.5 std::reverse_iterator

To understand how reverse_iterator::base works, see the following snippet. For more information refer to std::reverse_iterator.

```
int main()
{
  using iter = std::vector<int>::iterator;
  using r_iter = std::vector<int>::reverse_iterator;

  std::vector<int> v {0, 1, 2, 3, 4};
  r_iter res = std::find(v.rbegin(), v.rend(), 2);
  iter base = res.base();
  assert(*res == 2 && *base == 3);
}
```

B.6 std::mismatch

```
#include <iostream>
#include <algorithm>
#include <string>
#include <cassert>
#include <vector>
#include <iterator>

bool is_palindrome(const std::string& str)
{
  auto res = std::mismatch(str.begin(), str.end(),
                           str.rbegin(), str.rend());
  return res.first == str.end();
}

int main()
{
  assert(is_palindrome("aba"));
  assert(is_palindrome("abba"));
  assert(!is_palindrome("abcd"));

  std::vector<int> v1{1, 2, 3, 4};
  std::vector<int> v2{1, 2};
  auto res = std::mismatch(v1.begin(), v1.end(),
                           v2.begin(), v2.end());
  //C++14 and above:
  assert(res.first == std::next(v1.begin(), v2.size()) &&
```

```
          res.second == v2.end());
}
```

B.7 Boyer–Moore string-search algorithm

Note that unlike KMP, the worst case is still $O(mn)$. It's implemented since C++17. Another variation is Boyer–Moore–Horspool algorithm.

```
#include <iostream>
#include <algorithm>
#include <functional> //std::boyer_moore_searcher
#include <string>
#include <cassert>
#include <iterator>

int main()
{
  std::string s1 = "This is a test string";
  std::string s2 = "test";
  std::string before = "This is a ";
  auto res = std::search(s1.begin(), s1.end(),
                std::boyer_moore_searcher(s2.begin(), s2.end()));
  assert(res == std::next(s1.begin(), before.length()));
}
```

B.8 std::array

Note that if you don't use {}, both std::array and c-style array are not initialized when you are using primitive type. So they can have arbitrary values.

```
#include <iostream>
#include <array>
#include <algorithm>
#include <cassert>
#include <cstring>

int main()
{
  // a members are not initialized:
  // std::array<int, 7> a;
  // c_a members are not initialized:
  // int c_a[7];
  //a1 members are initialized to default:
  std::array<int, 7> a1{};
  // c_a1 members are initiliazed to default:
  int c_a1[7] {};
  for (auto i = 0; i < a1.size(); ++i)
    assert(a1[i] == 0 && c_a1[i] == 0);

  a1.fill(7);
  std::fill(c_a1, c_a1 + 7, 7);
  for (auto i = 0; i < a1.size(); ++i)
```

```
    assert(a1[i] == 7 && c_a1[i] == 7);

  std::array<std::array<int, 2>, 3> a2{};
  int c_a2[3][2] = {{1, 2}, {3, 4}};

  std::memcpy(a2.data(), c_a2, sizeof(c_a2));
  for (auto i = 0; i < a2.size(); ++i)
    for (auto j = 0; j < a2[0].size(); ++j)
      assert(a2[i][j] == c_a2[i][j]);

  assert(a2.size() == 3);
  assert(a2[0].size() == 2);

  a2.fill({4, 5});
  for (const auto& val : a2)
    assert(val[0] == 4 && val[1] == 5);
  std::array<int, 3> a3 {1, 2};
  assert(a3[0] == 1 && a3[1] == 2 && a3[2] == 0);
}
```

B.9 Initializing arrays with std::fill, memcpy and memset

Note that memset fills every byte in the buffer. If it's an array of elements bigger than 1-byte (e.g. array of integers), it does make sense to use it just for 0 or -1 (in a 32-bit signed integer -1 = 0xffffffff).

Unless it's an array of elements of 1-byte (e.g. char), it's wrong to use sizeof for std::fill. Note that the first two parameters of std::fill are two pointers. One points to the first element and the other to one after the last. On the other hand we should use sizeof for memcpy and memset.

```
#include <iostream>
#include <array>
#include <algorithm>
#include <cassert>
#include <cstring>
#include <iterator>

int main()
{
  std::array<int, 3> a1{};
  assert(sizeof(a1) == sizeof(int) * 3);
  std::fill(a1.begin(), a1.end(), -20);
  for (const auto& val : a1)
    assert(val == -20);

  std::array<std::array<int, 2>, 3> a2{};
  int c_a2[3][2] = {{1, 2}, {3, 4}};

  std::fill(&a2[0][0], &a2[1][0], 7);
  std::fill(&a2[1][0], &a2[2][0], 8);
  std::fill_n(&a2[2][0], 2, 9);
  for (int i = 0; i < a2.size(); ++i)
```

```
    if (i == 0)
      assert(a2[i][0] == 7 && a2[i][1] == 7);
    else if (i == 1)
      assert(a2[i][0] == 8 && a2[i][1] == 8);
    else
      assert(a2[i][0] == 9 && a2[i][1] == 9);

  std::fill(a2.begin()->begin(),
            std::next(a2.begin())->begin(),
            10);
  std::fill(std::next(a2.begin())->begin(),
            std::next(a2.begin(), 2)->begin(),
            11);
  std::fill(std::next(a2.begin(), 2)->begin(),
            std::prev(a2.end())->end(),
            12);
  for (int i = 0; i < a2.size(); ++i)
    if (i == 0)
      assert(a2[i][0] == 10 && a2[i][1] == 10);
    else if (i == 1)
      assert(a2[i][0] == 11 && a2[i][1] == 11);
    else
      assert(a2[i][0] == 12 && a2[i][1] == 12);

  assert(sizeof(a2) == sizeof(int) * 2 * 3);
  // Note that &a2[0] is a pointer to std::array<int, 2>
  std::fill(&a2[0][0], &a2[0][0] + 2 * 3, 15);
  for (const auto& val : a2)
    assert(val[0] == 15 && val[1] == 15);

  std::memcpy(a2.data(), c_a2, sizeof(c_a2));
  for (auto i = 0; i < a2.size(); ++i)
    for (auto j = 0; j < a2[0].size(); ++j)
      assert(a2[i][j] == c_a2[i][j]);

  std::fill_n(a2.begin()->begin(), 2 * 3, 16);
  for (const auto& val : a2)
    assert(val[0] == 16 && val[1] == 16);

  std::copy_n(&c_a2[0][0], 2 * 3, a2.begin()->begin());
  for (auto i = 0; i < a2.size(); ++i)
    for (auto j = 0; j < a2[0].size(); ++j)
      assert(a2[i][j] == c_a2[i][j]);

  std::memset(a2.data(), -1, sizeof(a2));
  for (const auto& val : a2)
    assert(val[0] == -1 && val[1] == -1);

  std::memset(a2.data(), 1, sizeof(a2));
  for (const auto& val : a2)
    assert(val[0] == 0x01010101 && val[1] == 0x01010101);
}
```

B.10 Hash combine

STL doesn't offer hash for containers. We need to consdier this fact that the following criterea:

1. The order of elements are important
2. The values of the lements are important

For example if we have a class with two attributes name and family, it's not a good idea to concatnate them and create a hash from it. For example if we have $name = foo$ and $family = bar$, then all of the following values have the same hash:

- name=foo, family=bar
- name=f, family=oobar
- name=fo, family=obar
- name=foob, family=ar
- ...

It's not a good idea to use operators that are **commutative** (e.g. $a \oplus b = b \oplus a$) or **associative** (e.g. $a \oplus (b \oplus c) = (a \oplus b) \oplus c$). Otherwise it's not a good distribution and the likelihood of collision is high. Both XOR and addition are commutative, but usually we use XOR because it's faster. An easy hash function in C++ can be

```
// Custom hash can be a standalone function object.
struct MyHash
{
        std::size_t operator()(const S& s) const noexcept
        {
                std::size_t h1 = std::hash<std::string>{}(s.
                    first_name);
                std::size_t h2 = std::hash<std::string>{}(s.
                    last_name);
                return h1 ^ (h2 << 1); // or use boost::hash_combine
        }
};
```

For more information please refer to std::hash.

since XOR is commutative, it's not order-sensitive so we should combine it with other operators. $boost :: hash_combine$ uses the following formula:

```
template<typename T> void hash_combine(size_t & seed, T const& v)
{
        seed ^= hash_value(v) + 0x9e3779b9 + (seed << 6) + (seed >>
            2);
}
```

$0x9e3779b9$ or 2654435769 in decimal is golden ratio. It's calculated:

$$\begin{aligned}\varphi &= \frac{1+\sqrt{5}}{2} \\ &\approx 1.61803398875\end{aligned}$$

So the golden ratio is

$$\frac{2^{32}}{\varphi} = 2654435769.5$$

I used hash combine formuala for this question

```
template<>
struct hash<array<int, 26>>
{
        size_t operator()(const array<int, 26>& v) const
        {
                size_t h = 0;
                for (const auto& item : v)
                h ^= hash<int>{}(item) + 0x9e3779b6 + (h << 6) + (h
                    >> 2);
                return h;
        }
};
unordered_map<array<int, 26>, vector<string>> anagram;
```