Competitive Programming

Richard Wang

Spring 2024

1 Preface

This document is written as part of Graduate Independent Study (CS 6993) conducted with Professor Mark Floryan at the University of Virginia in the Spring 2024 semester. It aims to go over several interesting/advanced techniques or algorithms which have applications in programming competitions as well as offer general guidance for those potentially interested in participating in these competitions.

2 About the Author

My name is Richard Wang and at the time of writing this I am currently doing a Masters in Computer Science at the University of Virginia. I started getting into programming competitions in the summer of 2020 and have since reached the rank of Master on Codeforces (Profile) and have also qualified for the ICPC World Finals for the 2023 season, placing 14th at the North American Championship. I've also solved an extensive number of problems on the site Kattis in preparation for ICPC, and you can see my profile here

3 Introduction

Programming competitions can be a great way to enrich your problem solving and thinking skills. Depending on the type of contest you are participating in, the formats can vary. Some of the main competitions include the International Collegiate Programming Competition (ICPC) for college students where teams of 3 compete in several stages, including Regionals, Nationals, and then Worlds. There are also online contest sites such as Codeforces and Atcoder, which are individual and have no restriction on who can participate. The problems that you typically see usually require you to make some observations and reductions before possibly applying some algorithms or data structures to solve within the time limit. Although knowing algorithms and data structures is important, going through and memorizing a bunch of them is by no means the most efficient way to practice. The more recent problems especially place a greater emphasis

on making observations and logical thinking which you can improve simply by just solving more problems.

If you are on the fence about whether you want to participate or not, I think there is no harm in trying it out and seeing whether you like it or not. I tried it out in the summer of 2020 after graduating high school during the COVID lockdown with no expectations whatsoever, and it ended up being one of my main hobbies for my college years. If you really find that you don't enjoy it and have to force yourself just to solve problems, there is no harm in dropping it if it just isn't for you.

If you are a UVA student looking to get some experience with programming competitions, you can join the ICPC club at UVA where you can participate in weekly practices as well as discuss problems in the ACM Discord.

4 Resources

In this section, I will highlight the main resources I use when preparing for programming competitions. Pretty much all of the problems that I link will be taken from one of these sources.

- Codeforces - Contains editorials for almost all of the problems which can be helpful when you are stuck
- Atcoder - Problems tend to be more focused on making observations and math. You can find a problem visualizer here which includes problem difficulties to help you pick the right problems to work on
- Kattis - If you are thinking of participating in the ICPC, you can find many past ICPC problems here. Keep in mind a lot of the problems may not have editorials so you may need to dig online to see if you can find them.
- USACO - Contains pretty detailed editorials for all of the problems. Recent problems have been quite good and also the topics which can show up are pretty limited in the lower divisions so there is a much greater emphasis on thinking rather than knowing algorithms.
- CSES - Contains a collection of more standard/well known problems. Do not confuse standard/well known to mean easy, it just means they might be a direct application of some technique or data structure, which in it of itself may not be that easy.
- USACO Guide - Includes a topic list for topics needed for each division of USACO as well as practice problems and guides for each. Although I personally wouldn't recommend practicing specifically by topic too much, this guide is quite useful for learning a specific topic that you may come across and get stuck at.

- CP Algorithms - A website which includes tutorials and linked practice problems for a lot of major algorithms and data structures. I wouldn't recommend reading through the entire page, but instead use it as a guide similar to USACO guide when you come across something while doing problems that you don't know.

5 Practice Guide

If the resources list is overwhelming, then a simple place to start would be to go to Codeforces, navigate to the problemset tab, then enter in some difficulty in the filter and solve down the list. You may want to pick a difficulty where you can get maybe around half the problems within 45 minutes without any help and move up when you feel more comfortable with that difficulty. You can also choose a range of difficulties since Codeforces difficulties have a lot of variance. It may take some experimenting to see which difficulty is right for you if you are just starting out. If you are stuck and have not made any new ideas for say around 20 minutes in a row, you might want to look at the editorial. If you encounter something that you don't know, like some data structure, you can do some research to learn it using some of the links I mentioned earlier such as USACO guide or CP algorithms. This is by no means the way you have to practice, but just something you can try and adapt if you are just starting out. The main advice I can say is that you will improve a lot faster if you just enjoy solving the problems. After reading the editorial, I think it is important to make sure you implement the solution so you really understand what is going on. It is also useful to look at other people's implementations for problems even if you solve it so that you can potentially learn implementation tricks and get better at it.

Another guide for practicing that I think is quite a valuable read is this (make sure to also read the comments section as the blog author includes more points there), which goes over a lot of potential pitfalls that you may find yourself in while practicing so being able to learn to avoid those early on is quite useful.

I will also recommend especially if you are just starting out to use C++ as your main language. It is the most popular language for programming competitions meaning the vast majority of guides and templates are written in C++. On top of this, code written in C++ runs a lot faster than alternative languages like Java or Python, which can be useful for problems which do not have different time limits depending on which language you use. In fact, for ICPC, it is only guaranteed that a solution exists in C++ or in Java (although it may require more work to get under the limit). It is possible that even the intended solution, when coded in Python, is just not fast enough to pass.

tl;dr - Just focus on solving problems at an appropriate difficulty level, look at editorial/other people's solutions when stuck or after you've solved the problem,

and use C++.

6 Mental Health

I also want to make an important point that like a lot of competitions, there will inevitably be points where you become frustrated due to poor performance, sometimes happening multiple times in a row. This is completely normal and has happened a lot to me. The important thing is to make sure you don't attribute this to mean that you yourself are somehow inferior as a result of it happening. While you continue to practice, you are always improving your thinking skills even if the performance in contests don't seem to indicate this. Sometimes you just don't have a good day or the problem which showed up in a contest happened to be something weird, so things like the results in contests or rating on Codeforces are too variable to be a true indication of your ability. Instead, focus on solving problems and the rating will catch up in the long run. If you are worried you aren't improving because you are somehow doing something wrong, refer to the blog post I linked in the previous section, and if you feel like you keep performing poorly in contests and it is getting to you, it is perfectly fine to take a break.

7 Disclaimer

The remaining content in this document can be quite abstract and difficult. It is mainly topics that I haven't learned yet that I am trying to get a better grasp on. Especially if you are just starting out, realize that you can get very far in competitive programming without ever needing to learn any of these, so don't think that not knowing them is what's holding you back. However, if you (like me) just find them interesting, then feel free to read about it.

8 Note

In each section, I will usually present some algorithm/data structure as well as a few problems and applications of it. After reading about the topic, I recommend that you look at the problems first and try to give them some thought and come up with the solution yourself before reading my solution. I will also be using 0 indexing like in actual programming unless stated otherwise. I will also be writing all my code in C++.

Strings

The next few sections will cover various algorithms related to strings. I've found that competitive programmers tend to be scared of string algorithms. Most of the time, they really only learn hashing, which although can solve the majority

of string related problems, can be annoying to implement as it requires a lot of steps and indexing. I was on the same boat as well, but when I finally decided to suck it up and try and understand some of these algorithms, I realized the way they work is actually very unique and some can even solve problems that hashing can't.

9 Z Array

9.1 Prerequisites

I would say the only prerequisite that you really need for these sections would probably be knowing that Rabin Karp String hashing (polynomial hashes) exist and what they can do as well as basic string terminology (prefix, suffix, etc.).

9.2 Introduction

Suppose we are given a string s. We want to compute the z function for this string. For a specific index i, the z function at position i is the longest substring starting at that position that matches with a prefix of the string. By convention, we say that $z[0] = 0$, although it can be argued that it is n, the length of the entire string, although this is up to you. The z array is just the array with the z function values for every single index. As an example, consider the string *abab*. The z array will look like $[0, 0, 2, 0]$, whereas for the string *aaaaa*, the z array will look like $[0, 4, 3, 2, 1]$.

9.3 Code

Now that I've defined the z array, let us look at implementing it. We start with the simplest implementation:

Listing 1: Naive algorithm for z array

```
vector<int> getZ(string s) {
  int n = s.size();
  vector<int> z(n);
  for (int i = 1; i < n; i++) {
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
      z[i]++;
    }
  }
  return z;
}
```

Initially $z[i] = 0$ for all i. At each position, we compare positions $z[i]$ and $i+z[i]$ while they match and continue to increment $z[i]$ until we hit a mismatch or go off the end of the string. Basically, we compare positions i and 0, $i + 1$ and

1, etc. until one of those conditions happens. The worst case runtime for this algorithm is $O(n^2)$. The easiest case to see when this worst case occurs is the string of all a's since the z array looks something like $[0, n-1, n-2, ..., 1]$, so in total the while loop runs $1 + 2 + 3 + ... + n - 1 = \frac{n \cdot (n-1)}{2}$ times for $O(n^2)$ runtime. We obviously want to do better than the naive algorithm, so let me introduce a few more variables. We will keep track of some variables l and r, which are initially both set to 0. As we iterate through the loop, we will set l and r to store the rightmost interval of indices which match up with a prefix of the string s. l will be inclusive and r will be exclusive. This means that if $l = 7$ and $r = 11$ for example, then we know $s[7] = s[0]$, $s[8] = s[1]$, $s[9] = s[2]$, and $s[10] = s[3]$. The new modified code looks like the following:

Listing 2: z array algorithm slightly modified

```
vector<int> getZ(string s) {
  int n = s.size();
  vector<int> z(n);
  int l = 0, r = 0;
  for (int i = 1; i < n; i++) {
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
      z[i]++;
    }
    if (i + z[i] > r) {
      r = i + z[i];
      l = i;
    }
  }
  return z;
}
```

By the definition of the z function, if the z function at position i is $z[i]$, then we know that the indices $[i, i + z[i])$ match up with the prefix of s, so we can use that to update l and r accordingly as shown in the code. However, this code is still $O(n^2)$ because even though we set these two new variables, we never used them. Now, I will actually incorporate these two variables in the code to get the finalized algorithm for the z array.

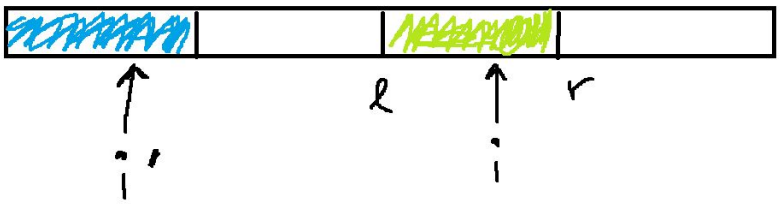
Listing 3: Final z array algorithm

```
vector<int> getZ(string s) {
  int n = s.size();
  vector<int> z(n);
  int l = 0, r = 0;
  for (int i = 1; i < n; i++) {
    if (i < r) {
      z[i] = min(r - i, z[i - l]);
    }
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
      z[i]++;
    }
    if (i + z[i] > r) {
      r = i + z[i];
      l = i;
    }
  }
  return z;
}
```

Notice the only thing I changed from this code to the previous is before running the while loop inside the outer for loop, I set $z[i]$ to some initial value. However, this code not only computes the correct answer for the z array, but also has an overall runtime of $O(n)$ amortized and I had absolutely no idea why it worked for the longest time, but now I will try to explain it to the best of my abilities. First, I will start off with the proof of correctness, for why this will arrive at the right answer, and then go over why it runs in only $O(n)$.

9.4 Proof of Correctness

Let us first consider cases for i at any point in time. Notice if $i \geq r$, then $z[i]$ will not be set to any value and kept at 0. The iterations in the while loop then run exactly like in the naive algorithm. Now suppose that $i < r$ in some iteration. Notice that $i > l$ will always hold since l was set in a previous iteration to a smaller index. I've drawn the following diagram to help illustrate this case.



In this diagram, l and r are exactly as described earlier, so the green interval they represent matches with the blue interval which is a prefix of the string. i is the current position we are checking and i' is the characters corresponding

location on the prefix, since we know that the green and blue intervals are the same string. Because the interval $[l, r)$ matches with a prefix of the string, this means that $i' = i - l$. We can use the fact that because the blue and green intervals are the same, and assuming we have already computed the z values correctly for indices before i, this means we can initialize the answer for $z[i]$ to equal $z[i'] = z[i - l]$ since those characters are the corresponding characters in two identical strings. We can see how this would work by imagining running the naive algorithm starting at both positions i and i' simultaneously.

However, there is a catch. We only know that running the naive algorithm starting at positions i and i' should behave the same until we go out of the interval, since the characters will be different starting then. This is why we need to *min* the starting value of $z[i]$ with $r - i$, since we do not yet know the behavior after we go out of the interval. However, this means that we can still initialize z this way at the start of each iteration and we will never end up getting a higher z value to start leading to an incorrect answer.

9.5 Proof of runtime

The proof that the overall runtime of this algorithm is $O(n)$ uses the fact that r will never be greater than n. By the way it is defined, we can never go past the original string so at most the rightmost boundary of some interval which matches a prefix of the string is n. You can also see this by reading the code and seeing that r will only ever get set to $i + z[i]$ for some i and the while loop terminates as soon as $i + z[i] >= n$. On top of this, r never gets decreased in any way so it can only increase until it hits n.

Now that we know r will only increase and is bounded by n, let's analyze some cases as we want to make sure that the while loop does not run too many iterations.

Case 1 : $i \geq r$

If $i \geq r$, then $z[i]$ starts at 0. Notice that in this case, $i + z[i] \geq r$ as well. This means that for every iteration of the while loop which occurs, $z[i]$ increases, meaning $i + z[i]$ increases as well. Since it is already $\geq r$ to begin with, it means every iteration of the while loop will contribute to increasing r by 1 when we update l and r after it is done running.

Case 2: $z[i] = r - i$

This case is similar to the previous case in that if $z[i] = r - i$ to start, then $i + z[i] = r$. Once again, every iteration of the while loop will end up contributing to increasing r by 1 when we update it afterwards.

Case 3: $z[i] = z[i-l]$

Assuming $z[i-l] < r-i$, this would mean that if you consider the green and blue intervals, a mismatch occurred before going past the interval so there was no need to even clip the initial value by $r-i$. For the while loop, this would mean $z[i]$ is already initialized to the point in which a mismatch occurred so it will immediately terminate when it sees the characters differ.

This shows that every time the while loop runs, it either immediately terminates because of a mismatch, or it causes r to increase by 1 but r can only go to at most n so the while loop only runs at most $O(n)$ times. I am aware that this can be a lot to take in so if you have some confusion I recommend rereading parts of this section and maybe drawing some cases down to see how the indexing works.

9.6 Problems

To me, this algorithm is absolutely beautiful since we take a naive algorithm which is already super short and add like 7 lines of code which make it run fast. Now that we have seen how to implement the z array, let's look at a few applications and problems where we can apply it.

Problem 1 - Finding Borders

Link: https://cses.fi/problemset/task/1732/

Solution: Given string s of length n, we can just compute the z array directly on s. For each index, check if the $i + z[i] = n$ as this means that the suffix starting at index i also matches with a prefix of s.

Code

Problem 2 - String Matching

Link: https://cses.fi/problemset/task/1753

Solution: Suppose our string is s and out pattern is p, with lengths n and m respectively. Let us first create a new string $t = p + s$, where $+$ represents concatenation. Now, p is a prefix of this new string. Next, we compute the z array for t and check which positions have a z value of at least m since that means starting in that position it matches with at least m of the prefix which includes the entire pattern. Be careful to only consider indices which were originally part of string s and not p.

Code

Problem 3 - Finding Periods

Link: https://cses.fi/problemset/task/1733

Solution: Once again, we compute the z array for the string. However, computing the solution once we have those values still requires one additional trick, which is the fact that the sum of a harmonic series, $\sum_{i=1}^{n} \frac{n}{i}$ is actually $O(nlogn)$. You can read more about this here. Now that we have the z array values, we can iterate for each length l and check if a length is valid. For any given length l, we can check indices $l, 2l, 3l, ...$ and see if the z value at those indices are at least l. We just have to be careful to account for when a string gets cut out at the end of the string. Overall, the number of iterations is $\frac{n}{1} + \frac{n}{2} + ... + \frac{n}{n}$ which is $O(nlogn)$ as discussed earlier.

Code

Problem 4 - MUH and Cube Walls

Link: https://codeforces.com/problemset/problem/471/D

We can do something similar to the solution for Problem 2 - String Matching, however now there can be some offset with the heights. To handle this, we can instead take the difference between each successive element in both the original array and the pattern since the difference between successive elements remains constant even with an offset added. Afterwards, we can do the same solution that we did for string matching to find the answer. Be careful of the case for when $w = 1$, since the difference array is empty, so just output n when that happens. This problem also demonstrates that you can use z array not just for strings but for things like arrays of numbers.

Code

The main takeaway from this is you can notice how much cleaner code using z array is compared to using hashing. This can be extremely important in cases such as ICPC, where you don't get to use pre-written templates that you can copy and paste, and your team only gets 1 computer so maximiizing your efficiency for keyboard time is essential to performing well. It is also less error prone and if you consider the case of Problem 4 - MUH and Cube Walls, many hashing implementations usually involve some form of polynomial hash where the prime base is chosen to be slightly larger than the alphabet size, however, in the case of this problem, the alphabet size can be very large which could actually make hashing even more complicated since the inputs are arrays of integers up to 10^9 rather than strings of alphabetical characters.

10 Manacher's Algorithm

10.1 Prerequisites

The main prerequisite to understanding Manacher's Algorithm is understanding Z array first

10.2 Introduction

You've probably seen problems such as finding the longest palindromic substring in a given string done in $O(n^2)$, in fact, this is even a pretty well known interview problem on Leetcode. To do this in $O(n^2)$, the idea is to fix the middle then iterate to the left and right simultaneously while the values match up until you either go off the string or find a mismatch. You can see a simple example of a worst case occurring when the string is n copies of the same character, so the palindrome centered around each position will hit the end of the string every time. This then begs the question, can we do better? Using Manacher's algorithm, we can not only find the longest palindromic substring of a string in $O(n)$ time, but even find every single palindromic substring in $O(n)$. You may be wondering how we could find every single palindromic substring of a given string in $O(n)$ if there can be up to $O(n^2)$ palindromic substrings such as in the worst case of n repeats of a single character, but the idea for Manacher's is rather than storing each individual substring, we store for each position the longest palindrome centered about that position. For example, notice if we have a palindrome of length 5, with characters $c_1c_2c_3c_4c_5$. By definition of a palindrome, this means we can simplify the string to be $c_1c_2c_3c_2c_1$. If we remove both ends and consider the middle 3 characters, we have $c_2c_3c_2$ which is also a palindrome. Taking this one step further gives us c_3 which is once again a palindrome of length 1. Thus, we are allowed to store the length of the longest palindrome centered around a certain position since we can't have for example a length 7 palindrome centered about a certain position but the middle 3 characters are not a palindrome.

10.3 Code

Now that I've gotten the basic overview of Manacher's out of the way, let's look at how we could go about coding this. You'll notice that most of the ideas end up being very similar to Z array. Something else to be aware of is because Manacher's returns the longest palindrome centered about a certain position, we need to do separate cases for even and odd length palindromes. For now, I will give the code and tutorial for odd lengths and then extend it to work with even lengths as well.

Like Z array, we start with code for the naive algorithm and try to expand upon it.

Listing 4: Naive Manacher's Algorithm for Odd Palindromes

```
vector<int> manacherOdd(string s) {
  int n = s.size();
  vector<int> p(n);
  for (int i = 0; i < n; i++) {
    while (i - p[i] >= 0 && i + p[i] < n
        && s[i - p[i]] == s[i + p[i]]) {
      p[i]++;
    }
  }
  return p;
}
```

If you trace the code for a particular index, we first compare positions i and i, then $i-1$ and $i+1$, then $i-2$ and $i+2$, etc. until we either go off the string or find a mismatch in characters. Note that the actual value stored in $p[i]$ is the half length of the palindrome rounded up. So if the longest palindrome centered at a position was length 7, $p[i]$ would store $\lceil \frac{7}{2} \rceil = 4$. This also means to retrieve the actual length of the palindrome, we would take $2 * p[i] - 1$.

Next, I will once again introduce two new variables, l and r. Previously for Z array, l and r were used to store the rightmost interval which matched a prefix of the string. This time for Manacher's, l and r will store the rightmost interval which is a palindrome. For my implementation, I have exclusive l and r. This means the actual interval of indices which is our palindrome is $[i+1, r-1]$. Our code looks like the following.

Listing 5: Manacher's Algorithm for odd palindromes slightly modified

```
vector<int> manacherOdd(string s) {
  int n = s.size();
  vector<int> p(n);
  int l = -1, r = 0;
  for (int i = 0; i < n; i++) {
    while (i - p[i] >= 0 && i + p[i] < n
        && s[i - p[i]] == s[i + p[i]]) {
      p[i]++;
    }
    if (i + p[i] > r) {
      r = i + p[i];
      l = i - p[i];
    }
  }
  return p;
}
```

Because $p[i]$ stores the half length of the longest palindrome centered at position i, we can see the left and right exclusive bounds will be $i - p[i]$ and $i + p[i]$,

which is how we update l and r in our code after computing $p[i]$.

Once again, this is still $O(n^2)$ because we haven't used l and r in any way other than setting them. Like the Z array I will now add some magic lines to make it run in $O(n)$.

Listing 6: Final Manacher's Algorithm for odd palindromes

```
vector<int> manacherOdd(string s) {
  int n = s.size();
  vector<int> p(n);
  int l = -1, r = 0;
  for (int i = 0; i < n; i++) {
    if (i < r) {
      p[i] = min(r - i, p[l + r - i]);
    }
    while (i - p[i] >= 0 && i + p[i] < n
        && s[i - p[i]] == s[i + p[i]]) {
      p[i]++;
    }
    if (i + p[i] > r) {
      r = i + p[i];
      l = i - p[i];
    }
  }
  return p;
}
```

As usual, the only difference between this code and the previous is right before the while loop, $p[i]$ is initialized to some starting value depending on i relative to l and r. Let's see why this gives the correct output and runs fast enough.

10.4 Proof of Correctness

The proof of correctness is pretty similar to the proof of correctness for the Z array. If $i \geq r$, then $p[i]$ starts at 0 and the while loop is essentially just running the naive algorithm so there is no reason for the output to be incorrect. Otherwise, i is in the range (l, r), which implies that it is part of a substring which we know to be a palindrome. You'll notice that I consider the value $p[l+r-i]$. This is because if i is part of the palindrome in the range (l, r), then its corresponding mirrored index will just be $l + r - i$. Notice that $l + r - i < i$ because l and r are set in a previous iteration centered around some index which is less than the current i. Now, you can once again imagine running the naive algorithm starting at positions i and positions $l+r-i$ and see that the behavior should be the exact same provided you are still within the interval (l, r). This is also why we once again minimize $p[i]$ with $r - i$ because we still do not yet know the behavior of the naive algorithm outside of the range.

10.5 Proof of Runtime

The proof of runtime is pretty much identical to the proof of runtime for Z array. r can only increase to at most n and never gets decreased in any way. Depending on the value that $p[i]$ is initialized to, either an iteration of the while loop contributes to increasing r or $p[i]$ is already initialized to the point in which a mismatch occurs so it immediately terminates. If this is still not clear, I recommend reviewing the proof of runtime in the Z array section since that is much more detailed.

10.6 Even Lengths

Now that we have the code for odd length palindromes, we want to see if we can also do the same for even length palindromes. I would even recommend trying to come up with this yourself as an exercise before reading on to make sure you really understand how this works since it is not very different from the odd case at all. The main difference between the even and odd case is that for odd palindromes, the center is the middle character, but the center for an even palindrome is between the two middle characters. We will basically run the exact same algorithm, but this time $p[i]$ will store the answer for the gap between the i-th and $i+1$-th characters in the string. What else does this change? Well in the while loop for the odd case, we start by comparing $i - p[i]$ and $i + p[i]$. When $p[i]$ starts at 0, we start by comparing characters at positions i to i, $i-1$ to $i+1$, etc. Now for even lengths, we want to start by comparing i and $i+1$, then $i-1$ and $i+2$, then $i-2$ and $i+3$, etc. So instead of doing $i - p[i]$ and $i + p[i]$, we now compare $i - p[i]$ and $i + p[i] + 1$. If you trace the execution of some example, you will see that $p[i]$ will store exactly half the length of the palindrome centered around gap i, so you can find the actual length of the palindrome by taking $2 * p[i]$. The proof of correctness and runtime are the exact same as the odd case.

Listing 7: Final Manacher's Algorithm for even palindromes

```
vector<int> manacherEven(string s) {
  int n = s.size();
  vector<int> p(n);
  int l = -1, r = 0;
  for (int i = 0; i < n; i++) {
    if (i < r) {
      p[i] = min(r - i, p[l + r - i]);
    }
    while (i - p[i] >= 0 && i + p[i] + 1 < n
        && s[i - p[i]] == s[i + p[i] + 1]) {
      p[i]++;
    }
    if (i + p[i] > r) {
      r = i + p[i];
      l = i - p[i];
    }
  }
  return p;
}
```

You can see that there are very small differences between the even and odd cases. In fact, it would probably make more sense to write Manacher's in a way which generalizes more. Using the fact that in C++, taking the $!x$ gives 1 when $x = 0$ and 0 otherwise, we get get a compact Manacher's algorithm as follows.

Listing 8: Final Manacher's Algorithm

```
vector<vector<int>> manachers(string s) {
  int n = s.size();
  vector<vector<int>> p(2, vector<int>(n));
  for (int type = 0; type < 2; type++) {
    int l = -1, r = 0;
    for (int i = 0; i < n; i++) {
      if (i < r) {
        p[type][i] = min(r - i, p[type][l + r - i]);
      }
      while (i - p[type][i] >= 0 && i + p[type][i] + !type < n
        && s[i - p[type][i]] == s[i + p[type][i] + !type]) {
        p[type][i]++;
      }
      if (i + p[type][i] > r) {
        r = i + p[type][i];
        l = i - p[type][i];
      }
    }
  }
  return p;
}
```

This code returns two vectors, $p[0]$ holds the vector of answers for the even case and $p[1]$ holds the vector of answers for the odd case. This is chosen intentionally so that it matches the parity of the type of palindromes you are looking at. As mentioned before, I use $!type$ which is either 0 or 1 so that the algorithm runs the same as when we looked at the even and odd cases individually. This code can look pretty confusing with all of the weird indexing, but try comparing it to when we consider the even and odd case separately and you will see that it only builds off of those slightly.

10.7 Problems

Now that we understand how Manacher's works, let's apply it to a few problems.

Problem 1 - Longest Palindrome

Link: https://cses.fi/problemset/task/1111

Solution: This is pretty much the classic problem for Manacher's. Simply run the algorithm and check which palindrome is the longest and print it. Note that in C++, the substr function arguments are the starting index and the length of the substring, so some fiddling with the indexing may be required to actually print out the longest palindromic substring.

Code

Problem 2 - Prefix Suffix Palindromes

Link: https://codeforces.com/contest/1326/problem/D2

Suppose our optimal solution involves taking the prefix of length 6 and suffix of length 2. Let these strings be *abcddc* and *ba* respectively. By concatenating them, we get that *abcddcba* which is a palindrome. Notice this means that the suffix of length 2 and prefix of length 2 must be mirrors of each other so concatenating just them would be a palindrome (*abba*), and then the remaining part in the prefix (*cddc*) has to also be a palindrome. Thus, any valid string can be split into three parts, the prefix, middle, and suffix, where the prefix and suffix are also those of the original string and must be mirrors of each other and then middle part which must be a palindrome. Thus, we can run Manacher's Algorithm to find for each midpoint the longest palindrome that is centered around that point. Afterwards, we just need to check if the endpoints of the palindrome are closer to the beginning or the end of the string, and see if the prefix and suffix of the original string with that distance as the length are mirrors of each other. If they are, we can update the answer accordingly.

Code

11 Segment Trees

11.1 Prerequisites

Prefix sums since most problems that deal with segment trees also involve this

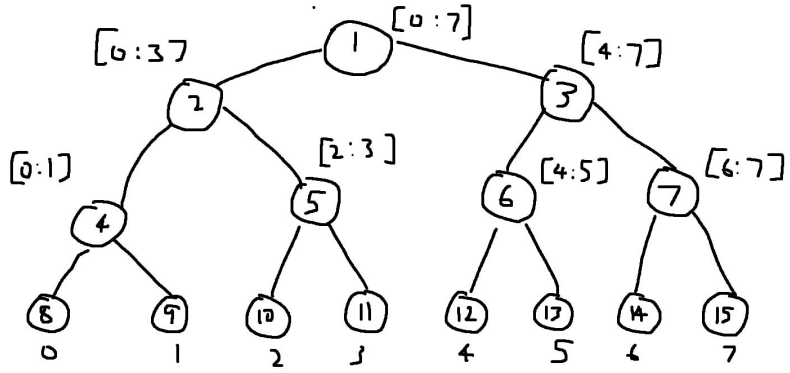
11.2 Introduction

Suppose you are given an array of elements and queries of the form for a given range $[l, r]$, find the sum of elements on this range. This alone can be done quite easily with each query answered in $O(1)$ using the prefix sum technique. Now, we want to add another type of query, which given an index and value, set the element at that index to the given value. When trying to think about this in terms of prefix sums, it does not seem immediately obvious how we can handle this kind of updating in less than $O(n)$. However, this is a classic problem for the segment tree data structure, which can handle both querying the sum on a range and updating some element in $O(log(n))$ time.

11.3 Visualization

I've included a visualization of a segment tree for an array of length 8. I deliberately chose 8 since it is a power of 2, and the illustration will show why this

makes things nicer.



You can see from the illustration the segment tree is basically just a binary tree which is specifically used for querying and updates. The numbers on the node indicate the node index, so when we code the segment tree, that will be the index in the array for that specific node on the binary tree. Note that this indexing is done so for any given node i, the two children will have indices $2 * i$ and $2*i+1$, and finding the parent of some node i is just $\frac{i}{2}$ (floor division). The ranges outside each node show what values each node consists of. Right now we are specifically looking at segment trees which support range sum queries, so node 2 will have the value $a[0] + a[1] + a[2] + a[3]$ if the array is called a. The bottom row of the segment tree is basically the original array. Now what happens if the array length is not a power of 2? In that case the easiest way to deal with that is just pad it until it becomes a power of 2. This won't change the runtime complexity since if the array has length n, then the smallest power of 2 at least n is at most $2n$, so it might make the constant factor a bit worse, but I have never run into a problem where this becomes an issue. If we let the next largest power of 2 be x, which ends up being the length of the bottom layer of the segment tree, for a given index i we can find its corresponding node in the bottom of the segment tree as $i + x$. You can see in the illustration the node for element 0 has index 8, the node for element 1 has index 9 and so on.

11.4 Updating

Now how does the segment tree make updating fast? You can notice that suppose we update element 2. First, we find the corresponding node on the bottom level which in the case of the illustration is node 10. When we change this value, which nodes' values actually change? It turns out the only nodes whose values will change in this case are nodes which have element 2 in the range, or the nodes from the path from the bottom level node to the root. In the case of updating element 2, we would start at node 10 and continue to update nodes 5, 2, and 1. You can see we are basically just walking up the tree and only updating 1 node per level, and there are only $log(n)$ levels, so we only

need to do $log(n)$ changes so the complexity of update is $O(log(n))$.

11.5 Querying

Suppose we are given some range such as $[2:6]$ and we want to find the sum. Looking at what ranges each node in our segment tree represents, we can just find the smallest amount of nodes needed to cover exactly this range. In the case of our illustration, we would only consider nodes 5, 6, and 14 for this range, as this gives us $[2:3]+[4:5]+6$ which is the range we are looking for. The proof for why this is not $O(n)$ is not as intuitive as updating, but one way could be to notice that in each level of the segment tree, you will only need to care about at most 2 nodes. For example, you might need nodes 5 and 6 in our illustration. Because a range query is consecutive, if you have 3 nodes in a single level, then 2 of them will definitely be children of the same parent so you could just consider the parent instead. For example, if you add node 4 to nodes 5 and 6, then you would have taken nodes 2 and 6 instead since node 2 is the sum of nodes 4 and 5.

11.6 Implementation

Now that we have seen how it works, lets look at how we can implement a segment tree. First, let's look at the node.

Listing 9: Node Struct for Segment Tree

```
struct node {
  int l = 0, r = 0;
  long long val = 0;
  node(int _l = 0, int _r = 0) {
    l = _l, r = _r;
  }
};
```

The node contains 3 values, the left and right endpoint as well as the value stored, all of which are the same as defined earlier. Next, we initialize the segment tree.

Listing 10: Segment Tree Initialization

```
int N = 1;
vector<node> tree;
segtree(int _n) {
  while(N < _n) N <<= 1;
  tree.resize(2 * N);
  build();
}
void build() {
  for (int i = 0; i < N; i++) {
    tree[i + N] = node(i, i + 1);
  }
  for (int i = N - 1; i > 0; i--) {
    tree[i] = node(tree[2 * i].l, tree[2 * i + 1].r);
  }
}
```

First, we declare N which will represent the size of the bottom layer of the segment tree. You can see we get N by finding the first power of 2 larger than the size we declare which is $_n$, which will usually be the size of your array. We then resize the tree vector, which stores each of our nodes to $2 * N$ and then call the build function. The build function initializes the left and right indices for each of the nodes in the same way as mentioned in the visualization section. Keep in mind that the illustration used left and right inclusive indices, while this implementation is left inclusive but right exclusive. Next, let's look at the update function

Listing 11: Segment Tree Update

```
void update(int i, long long v) {
  i += N;
  tree[i].val = v;
  while (i > 1) {
    i >>= 1;
    tree[i].val = tree[2 * i].val + tree[2 * i + 1].val;
  }
}
```

Pretty self explanatory, it does exactly what I described in the update section. If you are not familiar with binary operators, the >> operator represents bit-shifting so doing $i >>= 1$ is equivalent to taking floor division of i by 2. Finally, we have the query function.

Listing 12: Segment Tree Query

```
long long query(int v, int l, int r) {
  if (l <= tree[v].l && tree[v].r <= r) {
    return tree[v].val;
  }
  if (tree[v].l >= r || tree[v].r <= l) {
    return 0;
  }
  return query(2 * v, l, r) + query(2 * v + 1, l, r);
}
```

In this recursive function, v is the index of the current node you are on, so you always call this starting with $v = 1$ since that is the root, and l and r are the interval you want to query. For this implementation, it will be left inclusive and right exclusive just like the node ranges. The first if statement checks if the current node is fully contained in the interval of interest and if so you can just immediately return the value of the node. Otherwise, the second if statement checks if the node is completely outside the interval of interest and if so returns some identity element which doesn't affect the answer. For sum it would be 0. If your segment tree was doing say range minimum queries, you would return some very large value instead. Otherwise, if the current node is party within the interval and partly outside, we just recursively call the left and right child and combine the values. I usually like to include a second query function which automatically calls the one above with $v = 1$ so I only need to specify the interval of interest.

Listing 13: Segment Tree Query Helper

```
long long query(int l, int r) {
  return query(1, l, r);
}
```

Something else that may be useful is you might have a segment tree which does not necessarily do range sum queries. In this case instead of manually putting addition as the combiner function in query or when doing an update, you might want to specify some specific combiner function so you can just modify that like the code shown below.

Listing 14: Combiner Function

```
long long comb(long long a, long long b) {
  return a + b;
}
```

This way you can use it in the query and update functions and only have to modify it in case you want a segment tree other than range sum. Of course, you may still need to modify the return value in the query function when you are outside the interval.

Putting all this together, we can solve the following problem with the following code. By having an explicit combiner function, you can easily modify the segment tree to support other types of operations, such as this problem using this code. Notice all I changed was returning $min(a, b)$ in comb instead of $a + b$ and changing the return to INT_MAX in the case the node was out of range during query.

12 More Interesting Segment Tree Applications

12.1 Prerequisites

Segment trees

12.2 Introduction

So far we have only considered segment trees which just store some value in each node which represents the answer to some simple function applied to that range, such as min or sum, however, we can use segment trees to do more sophisticated queries. This section will mainly consist of example problems and an explanation of their solution and how to use segment trees to efficiently handle it.

12.3 Problems

Problem 1 - Prefix Sum Queries

Link: https://cses.fi/problemset/task/2166

Solution: The query we are being presented with here is very interesting and the simple types of queries we saw earlier will not be enough to solve this problem. So how do we approach this? Let's look at what the combine function actually does. On the surface it just takes two values and returns their combined value depending on what we want the segment tree to output. But what do the values passed to the combine function actually represent? These aren't just arbitrary values, but rather the answers for a left and right interval on the segment tree which are adjacent to each other. We then call combine on these answers to get what the answer would be if we were to concatenate the right interval to the left interval. For example, when we do a sum segment tree, say we call combine on the values for intervals $[0 : 2]$ and $[3 : 5]$. The sum of the values would then be the answer for the interval $[0 : 5]$. Thus, to handle these weird kinds of queries, we need to consider what information we would need to know about two intervals in order to merge them efficiently.

First, suppose our intervals have two intervals and we assume we already have the optimal prefix for each of these intervals. When we merge the two intervals, what could our resulting answer be? Well first, the optimal answer could still

just be the answer for the left interval independent of the right interval. Example: Consider the two intervals have values $[2, -1]$ and $[-5, -4]$. The best prefix for the left is just 2 and even when concatenating the two intervals the answer is still just 2 from the left interval. You can see that if the best answer for the concatenated intervals doesn't go past the left interval it will just be the answer for the left interval. Otherwise, the new best answer could go past the left interval and take some prefix of the right interval instead. For example, if the intervals are $[2, -1]$ and $[5, 4]$, the answer for the left is just 2 but concatenating the two tells us that the new best prefix is actually to take $2, -1, 5$ for an answer of 6. In the case that we go past the left interval, notice the resulting answer will be the sum of all of the elements in the left interval and the best prefix for the right interval. Thus, we need to store two values for each node, the total sum of elements as well as the best prefix on that interval for the node. When we merge two nodes, we can see that the new sum will just be the sum of the two intervals, and the best prefix will be the max of either the left prefix or the left sum + right prefix. The only other thing to keep in mind is when we update a position with some value, we will first set the node that only contains that specific index. We can see that if the value is ≥ 0, the best prefix will be the value, otherwise it will be 0, and the sum should just be the value the node is being set to. I also created a custom struct to store as the value in each node so it may not necessarily just be integers we deal with.

Code

Problem 2 - Subarray Sum Queries

Link: https://cses.fi/problemset/task/1190

Now, let's do a slightly more complicated example which follows from the previous problem. Once again, we just need to look at what information to store so we can do a merge efficiently. First, we should always store the actual answer or the best subarray sum. Now what happens when we merge to intervals. First, the new best subarray could just be the best of either the left or right interval. Alternatively, it could be part of both intervals. For example, merging $[-2, 5]$ and $[4, -3]$, we get the new best subarray is $[5, 4]$ with a sum of 9. Notice when this happens, the new subarray will be the best suffix of the left interval + the best prefix of the right interval, so we also need to store best prefix and best suffix sum. From the previous problem, we can see that updating the new best prefix or new best suffix will require us to store the total sum on the interval as well. Thus, each node stores 4 values, the best subarray, best prefix, best suffix, and total sum. Updating total sum in a merge is easy, and we've seen how to update best prefix already. Updating best suffix is pretty much the same, except this time we look at the max of the sum of right segment + best suffix of left and best suffix of right. As for subarray sum, as described earlier, it will be the max of either the left subarray sum, right subarray, or best suffix of the left segment + best prefix of the right segment.

Code

Problem 3 - Triptik

Link: https://open.kattis.com/problems/triptik

This problem has other steps on top of directly using a segment tree, so I will go over those other steps relatively quickly since they are not the main focus. Basically, we want to do a bfs where each node in our graph stores the middle point of interest and zoom level, since we only need to consider up to 30 different zoom levels. When transitioning to other nodes, we can increase or decrease our zoom level which is easy to do, or we can transition to another visible point of interest. This last transition is more difficult, but what we can do is coordinate compress the locations of the points of interests and then keep track of a segment tree to efficiently handle querying up to the largest 4 values in a range. Thus, each node in our segment tree will be a vector storing up to 4 of the largest values in the range the node keeps track of, and when merging two nodes, we can just take the best 4 out of the combined values so we only need to maintain at most 4 of the best values at any point in time. Note that the segment tree is also used when updating distances for the answer so that we can check if the current midpoint is actually visible or not.

Code

13 Mergesort Trees

13.1 Prerequisites

Segment Trees

13.2 Introduction

Mergesort trees are a pretty natural extension to segment trees. In the mergesort sorting algorithm, we divide up our current range into two halfs, recursively call the sort on those two halves then merge the two sorted halves to get the fully sorted range. For a mergesort tree, we will basically do the same thing. Earlier when learning segment trees, if a node had for example the range $[0:3]$, then this node would contain some combined value of elements 0 through 3 such as their sum. In a mergesort tree, instead the node will store elements 0 through 3 in sorted order, meaning we literally have a vector with a copy of elements 0 through 3 sorted in whichever way we care about. Knowing this, we need to also guarantee that we can do this and still have our code run in time. What you will notice is that every element will have a copy in exactly $log(n)$ levels,

as an individual element will only appear in the nodes leading from its location at the bottom of the tree up to the root. Put another way, if you look at every individual level of the segment tree, if you take all the nodes in the same level, they will be comprised of all of the elements of the initial array. Since there are only $log(n)$ levels, this means we only have to store $log(n)$ copies of the array total, meaning the total memory complexity is $nlog(n)$.

Now what about queries? Well similar to segment trees, every range we query on will be comprised of at most $log(n)$ nodes we visit. Typically, however, since each node now stores a range of values, finding the answer on an individual node might also take $log(n)$ time, for example if we were querying the number of values on a range smaller then a given value, then on each individual node we might do a binary search which takes $log(n)$ time, but since we are only doing this to at most $log(n)$ nodes, this means the complexity of this query is still $log^2(n)$, which is usually fast enough.
Let's look at a few example problems so that the previous section makes more sense.

13.3 Problems

Problem 1 - Smaller Sum

Link: https://atcoder.jp/contests/abc339/tasks/abc339$_g$

This is a pretty classic application of mergesort trees. Notice that we are forced to do the queries online, which means in the order which they are given and that there are no updates. For each node, we store both the elements in its range in sorted order, and also a prefix sum of those elements in sorted order. Thus, for an individual node when we are given some value x, we can do a binary search to figure out how many elements are $\leq x$ in the node, then return the value of the prefix sum at that index. In my implementation, I did most of the code in the main function using lambda functions instead of creating a mergesort tree struct, but the code is pretty much the exact same.

Code

Problem 2 - Triangular Logs

Link: https://open.kattis.com/problems/triangularlogs

This is an infamous problem from the 2022 ICPC North American Championship (NAC). The first observation needed is to think about how many trees in some area guarantee that a triangle can be formed. If we consider the worst case, the lengths of the trees will be $1, 1, 2, 3, 5, etc.$, with each new log being exactly equal to the sum of the two previous to just barely not be a valid triangle. This follows the well known Fibonacci sequence, which grows exponentially.

Given that the length of each log is at most 10^9, this means that if an area has more than around 50 trees we can assume automatically that a triangle can be formed. Its actually slightly lower but I just chose 50 to be safe. Thus, in each query we return up to any 50 trees in the area, and then we sort the values. We can check if a triangle is possible by fixing each element i as the largest length in the triangle, and we only need to check that elements $i-1$ and $i-2$ sum up to a larger value than element i to satisfy the triangle inequality.

The question then becomes how can be actually return the list of up to 50 trees in an area? During the contest, many teams actually moved towards using something like a 2D segment tree which is quite difficult to understand and implement, but we can just do this with a mergesort tree. First, we sort each point by their x coordinate and coordinate compress these x coordinates. Thus, each node in the bottom layer of the segment tree represents a different x coordinate and the vector in these nodes will store all trees with that x coordinate in sorted order by y coordinate. Afterwards, we simply construct the mergesort tree like shown previously. To answer a query, we find the coordinate compressed range for the x_{low} and x_{high} values so we know which range to query. For a specific node, we can binary search the first value $\geq y_{low}$ and keep adding trees to our resulting answer list while the length of the tree does not exceed y_{high}, making sure to break if we get more than 50 trees in our list. Finally, to merge the results, we can simply add the values in the second list to the first, clipping the length to 50.

Code

14 Persistent Segment Trees

14.1 Prerequisites

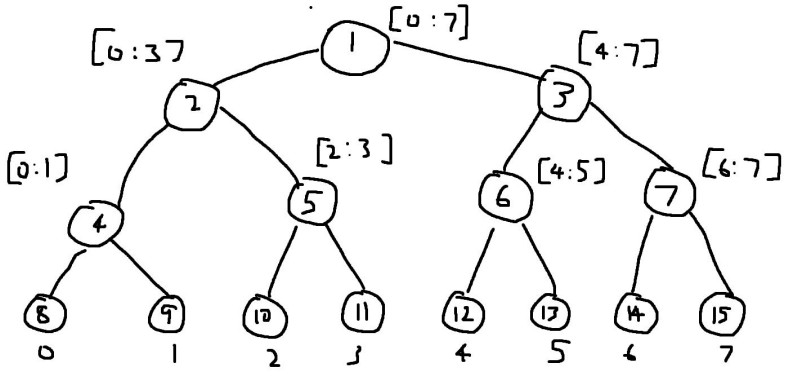
Strong understanding of segment trees and recursion

14.2 Introduction

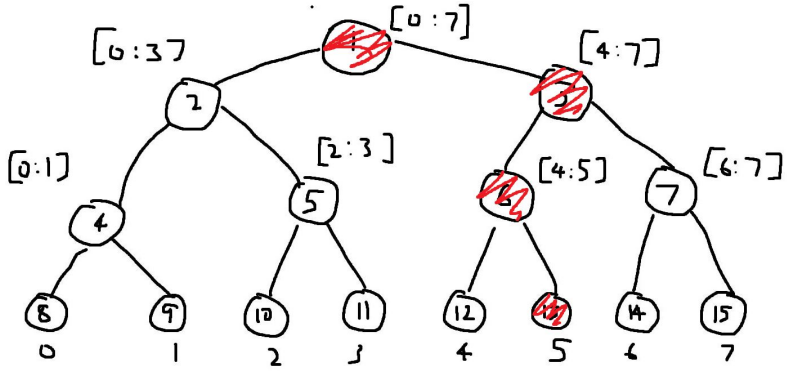
Persistent segment tree are a pretty interesting extension to the standard segment trees (ones which support updates/range sums, etc.). The reason they are called persistent is because you can query the segment tree at different points in time so the updates you make will persist through time. We can think of each update as creating a new version of the segment tree without actually physically copying the entire tree so we can efficiently do queries on different versions of the tree.

14.3 Main Idea

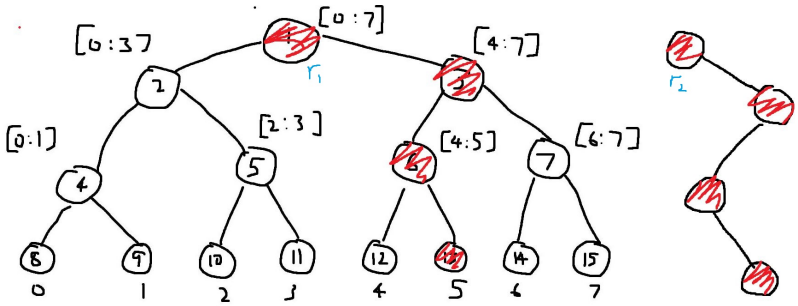
Consider the picture of the segment tree we had before with 8 nodes in the bottom layer.



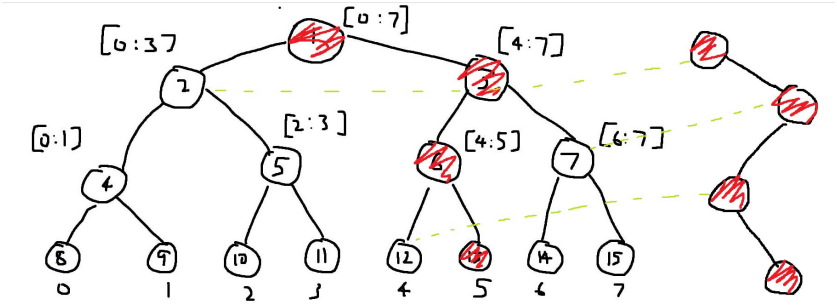
Now, suppose we decide to update index 5. Using what we know about segment trees, the nodes which need to change are colored in red.



The idea behind how persistent segment trees work is even though a new version of the segment tree is created as a result of this update, only the $log(n)$ nodes which are marked in red in this case actually change. In the normal segment tree, we simply override the values as we go up the tree and update all of the values. However, in this case, we want to leave the original nodes so that they persist and we can refer back to them later, and create new nodes which represent the nodes we had to update.



You can see in the diagram above, we create a copy of the nodes which are updated which are shown in the right. Notice that the topmost node out of the copied nodes creates a new root, labelled $r2$. So if we wanted to in the future query the updated tree, we could call query starting at this new root, otherwise if we want to call a query to the original tree, we call it on the starting root, which is labelled as $r1$. In order to actually be able to call queries on any version of the tree, we also need to connect the copied nodes to the proper children. The diagram below shows which nodes are children of each of the copied nodes, with an edge denoted in a dashed green line.



For every one of the copied node which was not a leaf, it should have two children. In one of the subtrees, no update occured so the edge is between the copied node and the original node. The other subtree will have an update so the edge is to another copied node, which is illustrated in the above diagram. If this is difficult to see, try looking at the above diagram and mapping each of the copied node to its original location in the tree and seeing which nodes the green edge goes to. The interesting thing about this construction is it also allows us to do updates on different versions of the tree. We can create a new version by saying update index 4 from segment tree version 3 for example, and create another new version.

Now that we have seen how the persistent segment tree works, lets try and implement it. In our previous segment tree implementations, we have a node which stores the answer for its range as well as the left and right positions for the range this node cares about. This was because when dealing with children, we could simply take some node i and say $2 \cdot i$ and $2 \cdot i + 1$ are the two children.

However, as you can see when we create copies, we have to create new nodes which didn't previously exist so the indexing will no longer be this simple. In my version of the implementation, I now store the left and right child id of each node as well as the value, and the left and right bounds for the range of each node will be handled as parameters in the recursive function.

Listing 15: Persistent Segtree Node

```
struct node {
  long long val;
  int lc, rc;
};
```

In this case, *val* is the answer and *lc* and *rc* are the left child and right child index respectively. Next, we initialize the tree.

Listing 16: Persistent Segtree Initialization

```
vector<node> tree;
int n;
psegtree(int _n) {
  n = _n;
}
long long comb(long long a, long long b) {
  return a + b;
}
```

We still have the *comb* function to make modifying the tree easier, and in this case we are doing sum as our query. The main difference between this and the normal segment tree is that we don't initialize the *tree* array to have size $2 * n$. Instead, it starts off completely empty and we create nodes on the fly. Next, we want to create functions which can create new nodes whenever necessary. There are two cases, the first one being when we create new leaves (nodes which are at the bottom layer and have no children).

Listing 17: Leaf node creation

```
int newLeaf(long long v) {
  int p = tree.size();
  node nd = {v, 0, 0};
  tree.push_back(nd);
  return p;
}
```

First, we take p which is the current size of our tree array. This will be the new index we assign the node we are creating. We can just set *lc* and *rc* to any value since we will never visit them. The parameter the function takes is the value, which we initialize into the node. Next, for the non-leaf nodes, we create a function which takes the indices of the two children we want this node to be

a parent of, and create connections that way.

Listing 18: Parent node creation

```
int newParent(int left, int right) {
  int p = tree.size();
  node nd = {comb(tree[left].val, tree[right].val), left, right};
  tree.push_back(nd);
  return p;
}
```

We do not need to pass in the value we want the node to have since we can just calculate that from using the *comb* function on the two values stored on the children, which are given as $left$ and $right$ in the function parameter, so we initialize the node in this way.

Now that we have set up some functions to help us create new nodes, we now need the fundamental functions which a segment tree uses. Being proficient at recursion will definitely be very useful in understanding why the following functions work.

Listing 19: Persistent Segtree Build

```
int build(vector<long long>& sum, int lo, int hi) {
  if (lo == hi) {
    return newLeaf(sum[lo]);
  }
  int mid = (lo + hi) / 2;
  return newParent(build(sum, lo, mid), build(sum, mid + 1, hi));
}
```

The first function is the build function. In the normal segment tree, the build function was used to initialize all of the proper ranges for each node since we knew those were all fixed beforehand. For the persistent segment tree, our build function takes in the initial array and builds the starting nodes of that array. The variables lo and hi represent the current bounds of the range we are looking at. I mentioned previously that now the left and right bounds will be handled as function parameters, and these are lo and hi. On top of constructing the initial segment tree, the return value of build at any call is the index of the new node created. This allows us to use build recursively to create the entire segment tree as well as set the children of each subtree accordingly.

The first if statement looks at if $lo == hi$. If this happens, then this means that the range includes a single element, which means it is a leaf, so we call the $newLeaf$ function. The parameter we pass will just be the element in the array at position lo since the range includes just that element. Otherwise, what we can do is for any range, we can find the midpoint by doing $\frac{(lo+hi)}{2}$, and the left

child will have range $[lo : mid]$ and the right child will have range $[mid+1 : hi]$. We can instead call the *newParent* function to create a new parent node if the current range is not a leaf. We need to specify the indices of the children, which we conveniently can do by just recursively calling *build* on the left and right ranges. The final return value is the root created during initialization, so we can call the function like:

int initialRoot = build(arr, 0, n - 1);

Next up we will look at the query function.

Listing 20: Persistent Segtree Query

```
long long query(int p, int l, int r, int lo, int hi) {
  if (l <= lo && hi <= r) {
    return tree[p].val;
  }
  if (hi < l || lo > r) {
    return 0;
  }
  int lc = tree[p].lc, rc = tree[p].rc;
  int mid = (lo + hi) / 2;
  return comb(query(lc, l, r, lo, mid), query(rc, l, r, mid + 1, hi));
}
```

p is the current index of the node we are on, while l and r is the range we are querying and want to know the answer for. Finally, lo and hi are defined similarly as previously, representing the range that the node we are currently on holds. The first two if statements are very similar to the normal segment trees, checking if the current node is completely in or out of the range we are querying and returning the answer accordingly. Otherwise, the current node we are on and the range we are querying intersect in some way, so we can just recurse on both the left and right child. In a normal segment tree, we would be able to have the children as $2 \cdot p$ and $2 \cdot p + 1$, but this might not be the case for the persistent segment tree. However, the node struct stores the index of the left and right child, so we can just use that to find their respective indices, then recurse to those nodes. Once again, we need to find the midpoint to get the ranges of the child nodes, and call *comb* on the answers to get the result. During queries, we do not need to create any new nodes so there are no calls to *newLeaf* or *newParent*. We can call a query as follows:

long long ans = query(root, l, r, 0, n - 1);

Finally, we look at the update function.

Listing 21: Persistent Segtree Update

```
int update(int i, long long x, int p, int lo, int hi) {
  if (lo == hi) {
    return newLeaf(x);
  }
  int mid = (lo + hi) / 2;
  int lc = tree[p].lc, rc = tree[p].rc;
  if (i <= mid) {
    return newParent(update(i, x, lc, lo, mid), rc);
  } else {
    return newParent(lc, update(i, x, rc, mid + 1, hi));
  }
}
```

The update function shares a lot of similarities with the build function, and the return value is also the index of the new node created during the current function call. This time i is the index we are updating, x is the value we are updating the index to, and p, lo, and hi are defined the same as in query. Once again, when $lo == hi$, we create a new leaf node with the value x as the set value of that node. Otherwise, we need to find the midpoint and the left and right child indices of our current node. In the first case when $i \leq mid$, this means that we need to call update recursively on the left subtree while the right subtree is unchanged. This is why we can call newParent with the left child specified as a recursive call to update on the left subtree, while the right child is the same as the original right child because the right half is unchanged, and vice versa for the case where $i > mid$. We can call an update like the following:

int newRoot = update(i, x, root, 0, n - 1);

If the recursive functions are still difficult to grasp, I recommend first learning more about how to implement segment trees in a more recursive fashion as shown in the following tutorial, as the persistent segment tree builds upon this kind of implementation.

14.4 Problems

Problem 1 - Range Queries and Copies

Link: https://cses.fi/problemset/task/1737

This problem is a very direct application of the persistent segment tree. To solve it, I created a separate list to store all of the roots for each of the versions of the segment tree. The second operation, which is the query, I just look at the corresponding root in the location in the list and call the query function. For the first type, I find the corresponding root at position k, and set it equal to the new root returned by calling update. Finally, to deal with the third query, I

find the root id at position k and push a copy of it to the back of the list. Even if there are multiple of the same root ids in the list at the same time, calling update on any one of them won't affect the others because update will always create an entirely new set of nodes without overriding any of the original values.

Code

Problem 2 - Closest Equals

Link: https://codeforces.com/contest/522/problem/D

The problem can be solved using a normal segment tree using the fact that queries can be answered offline (out of order). However, using persistent segment trees, we can solve a harder version of this problem where we have to answer the queries online (in order). First, for each element, find the previous element in the array which has the same value as well as the difference in their positions. If this does not exist, just set it to some large value INF. When querying a range, we can consider an element "valid" if it and this previous element are both within the range. We only want to query the min of all differences in a range which are "valid".

In order to ignore the non "valid" elements in a range when querying, we can start with an array initialized to INF in our persistent segment tree then update each value in decreasing order of previous. When we get a query with ranges l and r, we can binary search the root where all elements have previous greater than or equal to l, which takes us to the version of the segment tree where only those elements whose previous are at least l are initialized in the segment tree, and then just call the normal range query for that range. This ensures that only "valid" elements are queried.

Note: Because the intended solution of the problem is to answer queries offline and use a normal segment tree, the constraints are higher which makes the memory limit tight. Our normal implementation will MLE due to the overhead of using *push_back* in vectors. To deal with this, I modified the implementation to utilize C arrays and initialize a large one at the beginning along with an int to keep track of the current index to avoid having to resize.

Code

Problem 3 - Army Creation

Link: https://codeforces.com/contest/813/problem/E

This problem shares similarities with the previous ones, however the main difference is this time we are required to answer the queries online. For any query, we only need to keep track of the first k elements of each kind since any after

that should not be counted. Conveniently, k is fixed at the start and not per query. This allows us to use the fact that an element is "valid" in a range if we find the kth previous element of the same value and it is strictly lower than the left value of the range, since that means this element is one of the first k of its type in the range. We can preprocess this kth previous for each element at the start.

This time because we are counting the total number in a range, we start with an array of all 0s and an update will set an element in the array to 1 so that when we do a query we can get the sum. We do updates in increasing order of kth previous. That way when we are given a query l and r, we can binary search the roots we got from the updates to find the point where all updates had a kth previous strictly less than l, then do the range query from l to r since this ensures that all elements queried are "valid".

Code

Bonus: This problem can also be done in $log^2(n)$ per query using mergesort trees. Can you see how?

15 SOS DP

15.1 Prerequisites

Bitmask DP + Proficiency with DP

15.2 Introduction

Sum over Subsets (SOS) DP is typically used when we need to find information on various bitmasks fast. One classic example of this is suppose we are given an array of numbers, with each number being less than 2^{20}. We want to figure out for every single bitmask from 0 to $2^{20}-1$, how many elements in the array occur which are a submask of this bitmask. I will introduce some problems later which show why computing information like this could be useful, but for now let's go over some ways to compute this and their respective runtime complexities.

15.3 Naive Algorithm

Understanding the naive algorithms for summing over all subsets is actually not super important for understanding SOS DP so I will just briefly skim over it. The following code iterates over each mask, and the inner loop will iterate over every single submask of the current bitmask we are on. We assume we are considering bitmasks of length n, and when we visualize how this works, we pad any mask with a binary representation of length less than n with 0s in the front to make it length n.

Listing 22: Slow SOS

```
vector<int> f(1 << n);
for (int mask = 0; mask < (1 << n); mask++) {
  for (int i = mask; i > 0; i = (i - 1) & mask) {
    f[mask] += a[i];
  }
}
return f;
```

In the above code, f stores the answers for each mask we are looking for, and a stores what we are interested in, for our current example a stores the frequency of each number in our array. This code will compute the values we want mentioned in the introduction. What is the runtime of the following code? A mask that has k one bits will have 2^k submasks. When we iterate for every mask from 0 to $2^n - 1$, there are a total of $\binom{n}{k}$ bitmasks with k one bits, so the total submasks over every mask will be $\sum_{k=0}^{n} \binom{n}{k} \cdot 2^k$. Using binomial theorem (under the statement section), we can see that this summation follows the exact form of the expanded expression of $(1+2)^n = 3^n$, so the runtime is $O(3^n)$. This means realistically this code will only work for $n \leq 18$ or so, but there problems which may require $n \leq 24$. Although this does not seem like that much of a difference, because our runtimes are typically exponential, 3^{24} would be over 700 times slower than 3^{18}. For any given bitmask, the number of times it appears as a submask for some other bitmask is 2^z where z is the number of zero bits, which means we can be visiting a single bitmask quite a lot of times in the above code. If we reframe the dynamic programming to limit how many times each mask gets visited, then we can make the code much more efficient.

15.4 SOS

Let's define a new recurrence for our DP. We now have two values in each state, a mask as well as some other number i which represents which bits can be different (0 indexed). What does this mean? Suppose our mask is 0110101 and i is 2. This would mean the bits at positions 0, 1, and 2, or the rightmost three bits (since leftmost bits are the highest order) can be different while all of the higher order bits have to be the same. This means $f(2, 0110101) = a[0110101] + a[0110100] + a[0110001] + a[0110000]$, where a is our base case. Notice the first 4 that the last 3 bits which are allowed to be different in this case still have to be a submask of the original mask.

Now that we have defined our dp, we need to look at transitions. For any given mask and i, we are interested in the ith order bit in mask. If the ith bit is 0, then we can only add $dp[i-1][mask]$, since the ith bit will have to be a 0 to preserve the mask being a submask, so we add the state where we have one more fixed bit. Otherwise, if the ith bit is a 1, then when we decide to fix the ith bit, we can make that bit either a 0 or 1 and it will still preserve the submask property. Thus, in that case we get $dp[i-1][mask] + dp[i-1][mask - 2^i]$. Putting this in

code, we get the following:

Listing 23: SOS

```
vector<vector<int>> f(n, vector<int>(1 << n));
for (int mask = 0; mask < (1 << n); mask++) {
  f[-1][mask] = a[mask];
}
for (int i = 0; i < n; i++) {
  for (int mask = 0; mask < (1 << n); mask++) {
    f[i][mask] = f[i - 1][mask];
    if (mask & (1 << i)) {
      f[i][mask] += f[i - 1][mask ^ (1 << i)];
    }
  }
}
return f[n - 1];
```

First, you can see negative indices at the beginning. That is intentional to just show that the state before $i = 0$ is just the base case. Afterwards, the code shows the transitions as described. There are a total of $n \cdot 2^n$ states, and there are at most 2 transitions from each state, so this gives us a total runtime of $O(n \cdot 2^n)$. However, this also uses $O(n \cdot 2^n)$ memory, which may be too much despite the runtime being fast enough for some problems. Thus, we can use the fact that the value for i is iterated over in the outer for loop, meaning we are finishing computing all the states for a lower order bit before we move on to a higher order bit. Using this, we can completely remove one of the dimensions in our dp. This is also in part due to how we can't subtract $1 << i$ from a single mask more than once when looking at position i, so no double counting will occur. This gives rise to the following code:

Listing 24: SOS Clean

```
vector<int> f(1 << n);
f = a;
for (int i = 0; i < n; i++) {
  for (int mask = 0; mask < (1 << n); mask++) {
    if (mask & (1 << i)) {
      f[mask] += f[mask ^ (1 << i)];
    }
  }
}
return f;
```

The final code ends up being very short, and the best part is typically when doing SOS problems, once you think about what you want the SOS DP to calculate and store, you can just jump straight to the final code and there is no need to even think about the two dimensional derivation. The other thing you can do with this is suppose instead of computing some sum over submasks, but instead you want to compute some sum over supermasks, you can literally change one character in the above implementation.

Listing 25: SOS for Supermasks

```
vector<int> f(1 << n);
f = a;
for (int i = 0; i < n; i++) {
  for (int mask = 0; mask < (1 << n); mask++) {
    if (~mask & (1 << i)) {
      f[mask] += f[mask ^ (1 << i)];
    }
  }
}
return f;
```

The only thing changes is the if statement now has a tilde in from of mask which represents bit inverse, so now we check if the ith bit is a 0 instead of a 1. The reason this works is because if some mask is a supermask of another mask, then if you invert all of the bits in both masks, it will now become a submask. So we are basically doing the same thing as the submask case but just consider the bits as opposite. Next, I will go over some problems which use SOS DP to hopefully illustrate the kinds of problems you can use it in.

15.5 Problems

Problem 1 - Bit Problem

Link: https://cses.fi/problemset/task/1654/

Let's look at what each of the staements is actually saying. The first statement means that y is a submask of x, so we are counting the number of submasks. The second statement means that y is a supermask of x. The last statement is a bit tricky, but if $x\&y \neq 0$, we can count the number of y such that $x\&y = 0$, and then subtract it from n. If $x\&y = 0$, this means they share no 1 bit in common so if we invert the bits of x, then y is a submask of the inverted value. Because each value is at most 10^6, we can count the number of submasks and supermasks assuming up to 20 bits using the values described earlier and then output the values as described.

Code

Problem 2 - Compatible Numbers

Link: https://codeforces.com/contest/165/problem/E

Once again, we use the idea that if $x\&y = 0$, then y is a submask of the inverted x. However, this time we just want to print any value which satisfies this property, not count the number of values. When we say sum over subsets, the sum doesn't necessarily mean addition, but rather some form of aggregation over all subsets. Thus, what we can do is initialize every mask for the base case to either be -1 if that mask does not exist in the array or the mask itself if it does exist in the array. Our "sum" function can be the max over all submasks so at the end if the result for some bitmask is -1, we know that there are no submasks of that bitmask in the array since all of the values are positive, otherwise we have some submask value which we can use to compute the answer by inverting x and finding the answer at that position.

Code

Problem 3 - Vowels

Link: https://codeforces.com/contest/383/problem/E

First, the problem tells us that only the first 24 letters of the alphabet appears, so our bitmasks will be over the character positions in the alphabet. We can assign each letter to a different bit, so for example, the string *abd* will be given the mask $2^0 + 2^1 + 2^3$. Now, consider a single string, suppose it is *abd*. We want this string to add 1 to every single bitmask which shares at least 1 character with this string. Looking at the character's position, we can add 1 to the mask 2^0, then add 1 to the mask 2^1 and then add 1 to the mask 2^3. However, notice that if we do the SOS DP for submask totals, this will result in overcounting, since the mask $2^0 + 2^1$ will now get a total of 2 contributed from a single string. To deal with this, we need to use inclusion-exclusion.

For any given string, we can iterate over each nonzero bitmask of the string and

consider only those characters, construct the bitmask based on the positions of those characters in the alphabet, then add 1 to that mask if the number of characters is odd, otherwise subtract 1 (this is how inclusion-exclusion works in this case). So for our example of *abd*, we would first add 1 to masks 2^0, 2^1, and 2^3. Next, we subtract 1 from masks $2^0 + 2^1$, $2^0 + 2^3$, and $2^1 + 2^3$. Finally, we add 1 again to mask $2^0 + 2^1 + 2^3$. Afterwards, we do the submask sum for SOS DP. If these steps are confusing, it may be more useful to refer to the code. I also removed duplicates from the string first before doing the inclusion-exclusion.

Code

Problem 4 - On-Call Team

Link: https://open.kattis.com/problems/oncallteam

This problem showed up on the 2023 ICPC NA South Regionals. Notice there are only up to 20 different services, so if we could somehow check if every single bitmask of services had a proper matching assignment to engineers or not, we could pretty easily compute the answer from that. To check if a proper matching occurs for a bitmask, we need to use Hall's Marriange Theorem. In particular, look at the Graph theoretic formulation section.

To put it simply, if we considered some subset of services, and drew an edge between each service in the subset and every engineer who was familiar with that service, then we can consider the subset good if the number of engineers we connect an edge to is at least the number of services in our subset. In order to have a matching, then this condition needs to satisfy for every single submask of our subset as well.

Using this, we can first find for each subset of services the number of engineers we can connect an edge to, aka the number of engineer bitmasks given in the input which share at least 1 one bit with our service bitmask. If x is our service bitmask and y is our engineer mask, this is equivalent to counting for x how many y there are such that $x\&y \neq 0$. We've already seen how to do this with SOS from one of the previous problems. For each service mask, once we've found this count, we consider the difference of this count minus the number of one bits in the mask. Under Hall's Marriage Theorem, this difference has to be ≥ 0 for all submasks of a particular mask for there to be a matching for that mask. Thus, we can do a second SOS DP which finds for each mask the minimum value of this difference over all submasks. If we get a result that is less than 0, then we know that this subset of services does not have a matching and we can use that to find the answer.

Code

Problem 5 - All Colourings

Link: https://open.kattis.com/problems/allcolourings

This problem uses some knowledge of inclusion exclusion alongside SOS DP. First, notice the number of edges is small. This means we can do our bitmasks over the edges. I defined a 1 bit for an edge to mean that the two nodes it connects have equal colors, otherwise a 0 bit means the two nodes have different colors. When we are checking a specific mask, we can unite all of the edges which have value 1 since we know all of those nodes in the connected components formed will be the same color. We can then calculate the answer for the mask as k^c, where c is the number of resulting connected components. However, this isn't exactly correct as it ends up overcounting. Basically, by doing our calculation this way, all of the 1 edges in the bitmask will be satisfied, but there is a possibility that a 0 edge ends up actually connecting the same colors which would actually make it a 1 edge. This would mean that the answer we found for a specific mask isn't the answer for just that mask, but actually the total of the answers for this mask and all of its supermasks.

Once we have found the answers like above, we are given a list of values where the value for a given mask is the sum of answers for that mask and all of its supermasks, so we need some way to get the answer for each mask by itself. Fortunately, this requires a very minor modification to our SOS DP algorithm. All we have to do is take our SOS DP code which given just the values for each mask, finds the total sum of each mask and all of its supermasks and change the plus to a minus in the for loop where we update values. This allows us to reverse the problem into finding the original values for each mask given the sum of each mask and its supermasks.

Code