NOI.PH 2018 Training Week 10

Josh Quinto, JD Dantes, Kevin Atienza

May 4, 2018

Contents

1 Introduction

This week, we'll discuss a bunch of different techniques, mostly unrelated to each other, and also go through some important tips, tricks, and advice, based on our experience which you might find valuable.

All problems mentioned here will be worth some number of points.

2 Tips and Tricks

In the event that you find yourself stuck (e.g. do not know where to start, or do not know how to improve TLE or WA verdicts), here's a quick checklist which you might find useful:

- **Try special or smaller cases.** Usually, for math, DP, combinatorics, or pattern finding problems, the (possibly very short) recurrence or solution may not come to you immediately. In this case, it is advisable to try small, trivial, or base cases so that you have somewhere to start. This would usually mean finding the solution for $N = 0$ or $N = 1$, then possibly moving on to $N = 2, 3, 4...$ until you hopefully find the larger or general recurrence.

  Problems to try:

  - **Algolympics 2018: Religious War Prevention.** For this problem, there is a critical insight about the edges and nodes which you may need to be confident of your solution. In case it takes you a while to observe that, you may still be able to code an AC solution by simulating and finding the solutions for extreme cases (e.g. a purely linear tree, a star-shaped configuration, etc.). Once you find the solutions to these, you will probably observe a pattern which may help you in coding the solution.

    Link: Religious War Prevention

- **Check if sorting or order matters.** For a lot of problems, the straightforward complete search solution would be too slow (i.e., receive TLE) because of the bounds. However, one rule of thumb is to check if N is around 100,000. For such inputs, the solutions would usually be $O(n)$, or at most $O(n \log n)$. Thus, if an $O(n \log n)$ solution is acceptable, then we could formulate an algorithm that makes use of sorting.

  Here are some places where you could try problems which involve sorting:

  - **Greedy problems on uHunt.** In Chapter 3 (Problem Solving Paradigms) of uHunt, there is a section dedicated for just greedy problems. For these, you would usually start by sorting the input, and by doing so the solution would become easier to spot.

  - **Algolympics 2018: Triangles**. For this problem, the straightforward method of calculating the solution will work only for significantly smaller inputs. However, since N is

in the order of 10^5, an $O(n \log n)$ solution is acceptable and we can sort the input. After doing this, it will be easier to form cumulative sums, and the succeeding steps can actually be done in $O(n)$.

Link: Triangles

- **Make use of rubber duck debugging.** In IOI, you may bring a rubber duck or stuffed toy so that you could do what is called "rubber duck debugging", where you try to debug by talking and explaining your code and algorithm to the rubber duck. Others also do this by imagining and talking to a bird or daemon on their shoulder. This way, you are able to make use of the same benefits that you get whenever you teach a concept to someone or explain something to a five year old (i.e. ELI5): since you are verbalizing your thoughts, your ideas are organized and concretized in a step-by-step manner. By slowing down, you are able to deconstruct your algorithm into sentences, and it is easier to question and examine which statements (and implications of those statements) are fully correct and find out which ones may need improvement.

- **Precompute things–trade space for speed.**

  - **Precompute and store values which will be frequently used.** Recall the 1D and 2D cumulative sums. These are classic problems in learning dynamic programming, and are frequently used as a quick and easy tool for a larger problem. For instance, a 2D cumulative sum can be used to check if all cells in a subrectangle are filled out. This is useful in problems such as Codeforces 372B: Counting Rectangles is Fun (200 Points).

  - **Compute and store factorials. Reduce modulo operations.** Similar to the 1D sum and $O(n)$ Fibonacci, it may be wise to precompute the factorials if the highest n is known. This may be very significant when the answers are expressed remainder a modulo, and both the factorials and their inverses are needed (which is usually the case in combinatorics problems). This has happened in an actual contest—in Algolympics 2018: Bananagrams, a team got the correct formulation but got TLE, only getting AC when they went ahead and decided to precompute all the factorials and their inverses.

- **Try to approach the problem from a reverse direction. See if you can do offline processing.** For some problems, there might be a number of queries in the range of 10^5, limiting even an overall $O(n^2)$ approach. However, as with the tip earlier, we may be able to do something by sorting. In this case, we take in all the queries ahead of time and then go through them in a sorted order. By going through the queries in an increasing order, we may be able to get to the solutions within the time limit. We then just need to keep track of the original order of the queries, so that we could output the answers accordingly. Thus, we avoid needing to solve the problems online, but are able to solve everything by considering everything **offline**.

  You may try it on this problem: Zognoid Eggs.

- **Avoid floating point.** As much as possible, avoid working with floating point (fractional decimal) numbers, as their representation in the computer and resulting accuracy is limited.

Thus, if you could work with integers, it is probably better to do so. If the problem says that the input will be real numbers, yet indicates that there are a fixed number of decimal places (e.g. 2 digits after the decimal point), then you may still be able to work with purely integers by scanning the whole and fractional parts separately.

More details, tips and tricks about working with floating points are described in its own section later in this chapter.

- **Make use of typedefs, macros, and other shortcuts.** To save coding time in a programming contest setting, you will probably want to make use of several shortcuts or templates so that you can focus on implementing your algorithm, rather than typing redundant code. Here are some suggestions:
  - **Import everything with `#include <bits/stdc++.h>` (C++).** When working with actual software projects, it is understandable and good practice not to import everything, as that will pollute the namespace. However, in a contest environment, this trick will be useful since you are the only one working on a single file, and you do not want to spend your time repeatedly typing to include common utilities (e.g. STL `stack`, `map`, `utility`) but rather should focus on being able to code the solution itself.
  - **Use typedefs.** To keep code short, you may want to make use of typedefs frequently. Common ones may be:
    * `typedef pair<int,int> pi;`
    * `typedef tuple<int,int,int> t3;`
    * `typedef long long ll;`
  - **Use the `auto` keyword (C++11 and above).** By doing this, it will be easier to change things around when we find ourselves needing to later on. For example, we might need to change from a `pair<int,int>` to a `tuple<int,int,int>`. By using `auto`, the lines that we would need to change will be kept to a minimum.
  - **Use macros or `defines`.** Make sure that the parameters of the macro are appropriately wrapped in parentheses, so that the intended operator precedence is preserved. Some common ones:
    * Infinity. Make sure to check for the appropriate upper bounds. `#define INF (1<<28)`
    * **Take care of `long long` literals.** `#define INFL (1LL<<60)`
    * Left child of a node in a binary tree. `#define lchild(x)((x)<<1)+1)`
    * Right child of a node in a binary tree. `#define rchild(x)((x)<<1)+2)`
    * Dummy values if you want to make your code more readable. `#define UNVISITED -1`

    * `for` loops. I personally don't use this, but a lot of users do: `#define rep(i, l, r) for ((i)= (l); (i)< (r); (i)++)`

  – **Compete and then look at the code of top contestants.** After joining frequent contests on platforms like Codeforces, you may want to take a look at the solutions of the top-ranked users for the contest. By doing so, not only will you be able to compare and see if they had a more elegant solution, you will probably end up seeing the common templates and shortcuts that they use.

- **Review operator precedence.** When using one-liners or macros that involve bit shifting and bitwise operators, make sure that the precedence of operators are as intended. Arithmetic expressions go before the shifts. Single bitwise operators (e.g. `&` and `|`) come rather late—they go before their double operator counterparts (e.g. `&&` and `||`) and after the equality checks (i.e. `==` and `!=`). You may review everything here. When in doubt, just use parentheses to be safe.

- **Debugging checklist.** If you think that your code (or at least, your algorithm) is correct but do not get AC, go over the following:

  – **Arithmetic overflow.** Check the data types (integers vs longs), the sizes of the inputs, the typecasts (if any), and make sure that intermediate operations (e.g. addition, multiplication) do not overflow.

  – **Out of bounds.** Make sure that your array sizes are set correctly. It is common practice to exceed the needed size by a few elements (e.g. by ten or so) to avoid segmentation faults and unexpected behavior. If you try to access addresses beyond the size of the array, you are "lucky" because your program will crash (RE) as a result of a segmentation fault. Another thing that may happen and mislead you is if you have declared two consecutive arrays—if you cross over the bounds of the first one, it will spill over and overwrite the elements of the second one. Thus, your program will not crash, but instead lead to a confusing WA since the other values have been overwritten.

  – **Off-by-one, edge, and tricky cases.** When getting WA, make sure that your code works for the edge cases. This usually means testing on $N = 0, 1$, the maximum values, and other inputs that represent degenerate cases. When you are working with a team, you will have the advantage of having someone to think up of the test cases to break your code. However, when you are working individually, you may have no option but to think of those code-breaking cases yourself, or generate all possible inputs and see if your code produces the expected output.

  – **Recursions that are too deep may crash.** If the input is too large, a recursive approach may go over the stack limit, resulting to a crash (RE). To see how deep you can go, consider testing a simple or plain recursion and checking the maximum number of calls it can reach before it crashes. When this happens, you may either attempt to prune your solution, or recode and switch to a purely iterative approach, which will usually also be faster.

- **Improve typing speed!** This is also pointed out in Halim's book. It is not uncommon for

submissions to be received near the end of the contest, resulting to very critical changes in rankings by the time it ends. Thus, having (in the case of a team contest) or being a very fast coder will of course have its advantages. There are a lot of places where you can track, practice, and improve your typing speed, such as http://play.typeracer.com.[1] Alternatively, you may just want to keep on solving problems (pure "coder" or not) as fast as possible, so that you could get a good mix of letters, numbers, and other symbols that are commonly used in programming contests.

Of course, it will not do to be very fast at typing if you are not able to come up with a solution in the first place, so do not neglect that part. In any case, being fast will not hurt—after all, contests like the ACM ICPC do give awards and prizes for those who are First to Solve for the different problems. For the first (and thus, usually easiest) problem to be solved in the whole contest, submissions are usually received within the first five or so—if not the first—minutes. So do read and code fast.

- **At the same time, have fast but good reading comprehension.** Make sure that you have correctly understood what the problem is asking for. Ideally, you should also confirm it through the sample input and output. This way, you won't be wasting time trying to come up with a solution, coding it, only to find out later that you have misunderstood the problem. The actual problem may be much easier or much harder than you thought, depending on how you (mis)understood it.
  - **Do not be misled by intimidating titles, drawings, or figures.** The problem setters and writers may intentionally disguise the problem through humor or wit. Check Algolympics 2017: Never Forget the C. The description talks about and shows a couple of integrals at the start of the problem. By doing this, a lot of competitors may easily end up skipping the problem, thinking that it might require lots of high mathematical insights (also applicable to computational geometry problems—a lot may skip a problem just from seeing polygons). In this case, the problem is actually not about math, but involves a graph—a 2D grid! A lot of contestants may have solved this one if only they didn't skip it because of the initial integrals in the description. It also helps to really check the sample input and output to ensure what the problem really is about.
  - **At the same time, check if you have not missed a relevant detail that is part of the story.** As a concrete example, take a look at this problem: The Chenes of the Chorva. For this, the fact that the spageti goes down and then up can be easily missed, but is actually a very important detail when coming up with a solution that will pass the time limit.
- **Make sure to read or go through all problems at least once.** Again, be wary about skipping some problems that may appear to be more difficult than they actually are. At the same time, watch out that you do not miss the easy ones because of problems that appear to be easy. In Algolympics 2018, a lot of teams were misled by attempting Triangles, as it was very easy to code a correct but slow (TLE) solution. In comparison, Zoolander was a lot easier (BFS variant), but had less attempts as people were focused on solving Triangles instead.

[1] Another fun one that one of the authors is fond of: http://phoboslab.org/ztype/v1.html

- **Make sure that you have taken the bounds into account.** A "hard" problem may be solvable if the bounds are set to a very small number. On the other hand, some problems may be "simple", but actually require more complicated insights for intentionally higher bounds, as in Algolympics 2016: Godlike Multiplication.

  - If you have immediately thought of a "simple" or straightforward brute force solution that you think will pass the time limit, do not hesitate to code it immediately, and fast. For now, it does not matter if there is a much better or elegant solution—you will have time to learn and discuss that after the contest. During the contest, try to get "easier" problems out of the way as much as possible, to also give you more time to think about the more difficult ones.

  - As an example, this means that if there is a shortest path problem in which you may want to use Dijkstra's for efficiency, coding a 4-liner Floyd-Warshall's might be the better way to go about it.

  - For some problems, it may be possible to "break" the intended solution of the judges by preprocessing for a while (i.e. for a couple of seconds to a few minutes) to solve and store all possible answers for all possible inputs. This strategy is more applicable at the start of the contest when there still a lot of time—you can code and let it run in the background, while working on the other problems.

    This is also a strategy pointed out by Halim. From experience, this may net you another problem, possibly giving you a significant ranking advantage over the other competitors.

    Take a look at this problem: Clash of the Brainf---s. There is a better ($O(n)$) solution, but you may be able to preprocess all values with an $O(n^2)$ approach. Formulate, try and verify it for the first values, then let it run for a while to generate the answers for the whole range of inputs. During the actual contest, another platform was used, so there was no source code limit. However, for HackerRank there may be a limit, so you might not be able to submit such a solution this way.

- **Don't give up until the end!** As mentioned before, it happens that game or tie breaking solutions are submitted minutes, if not seconds before the contest ends. Thus, if it seems that at the start your competitors have solved a couple more problems than you, do not lose hope. It is not uncommon for a contestant or team to lead in the initial parts of the contest, but eventually end up being overtaken just by one problem at the end. Again, you may just need one more problem over your opponent to win, so do try your best and use all of your time to come up with solutions until the end of the contest.

3 Applications of Binary Search

In the Tips and Tricks section before, we have mentioned the use of sorting to give us a different perspective in approaching the problem. In particular, when the input is sorted, an $O(\log n)$ binary search can be done to arrive at a particular value in the array. Aside from this, in this section we explore two other applications of binary search. The first is a method known as "binary search the answer", in which we repeatedly assume that an answer is correct, and continuously adjust this answer until we get to the optimal solution. In the second subsection, we are able to apply binary search since we are keeping an ascending list of increasing subsequences (technically just the last element of the subsequences, as you will see later).

3.1 Binary Search the Answer

For some problems, a complete search may not be fast enough, and there may not be (or possibly just hard to find) any greedy solutions. In this case, consider "binary search the answer", in which instead of coming up with an algorithm to arrive at the optimal solution, we keep on **assuming** the correct answer, and then seeing if the current answer fits the description or constraints of the problem. We then adjust this current answer lower or higher depending on the problem. If the answer has a maximum value of K, then we will be able to arrive at the optimal solution in $O(\log K)$ steps. If checking whether the current answer is valid or not takes $O(f(\cdot))$ time, then the overall complexity for this approach will be $O(f(\cdot) \log K)$.

3.1.1 Exercises

You may try this approach in the following problems:

- GCJ 2018 1A: Bit Party (100 Points). The analysis (solution) is also included in the link.
- Chapter 3 (Problem Solving Paradigms) on uHunt. There is a Divide and Conquer section that has problems involving Binary Search the Answer. (200 Points each)
- Chapter 8 on uHunt. There is a section dedicated to mixing Binary Search with another technique. For some problems, there may be other approaches (e.g. a pure math formulation), but using binary search may be an easier or faster approach. (200 Points each)

3.2 Longest Increasing Subsequences (LIS) in $O(n \log n)$

We can also use binary search to find the LIS.

We have gone over the LIS problem before in the chapters which covered DP. To recap, we need to look for the longest (most number of elements) subsequence (a subset of the array whose elements

are not necessarily contiguous) where, from left to right (i.e. lower indices to higher indices), the elements are increasing (could be strictly increasing or non-decreasing, depending on the problem).

To find the LIS, we can do an $O(n^2)$ solution, which is also a good example for learning DP. However, there is a much better $O(n \log n)$ approach. For this, we keep note of different "stacks" or "decks". Each deck corresponds to an increasing sequence (IS). The catch is, for each deck that we note, we only keep track of the last element in the increasing subsequence that it represents. Also, we keep this decks in an increasing array of its own, such that the first deck is an IS of length 1, the second deck is an IS of length 2, the third deck has 3 elements, and so on.

For simplicity, let's work with unique elements first. Consider the following array:

```
int input[] = {3,7,1,8,4,5,6};
int N = sizeof(input)/sizeof(input[0]);  // N elements
```

At first, we have an empty list of decks:

```
vector<int> decks;  // Initially empty
```

We then loop through all N elements of our `input` array.

At the start, we can build an IS of length one, using the first element.

```
for(int i = 0; i<N; i++){
    if(decks.empty()){
        decks.push_back(i);  // We store the index,
        // rather than the value of the actual element.
        // This will be convenient later on.
    }
}
```

Our list of decks now looks like this:

```
decks[0] -> {3}

// That is, the first element (i.e. IS with length 1) is {3}.
```

We proceed with the next element, look at all (currently just one) our `decks`, and see which one we can extend. Here, we see that we can extend the IS of length one (i.e. `decks[0]`) which ends with the value "3". Thus, we now have an IS of length two (i.e. `decks[1]`) ending with a value "7":

```
decks[0] = 0 (index 0, value 3) -> {3}
decks[1] = 1 (index 1, value 7) -> {3, 7}
```

On to the third element. In general, when we insert an element into `decks`, we want to extend or

append it next to an IS which, as much as possible, is just below it. In other words, we find the index j in `decks` where the last value of the IS `decks[j]` is the largest, but still smaller than the current (i^{th}) element (i.e. `input[decks[j]] < input[i]`, `input[decks[j]]` is as large as possible).

This will be clearer as we proceed with the example. For now, we can separate the said approach into three cases:

- **Case 1:** We can extend the longest deck (i.e. IS with length equal to `decks.size()`). If this is so, we form another IS by extending the longest IS (i.e. that at `decks[decks.size()-1]` and appending it to `decks`. This is what we did when we extended `decks[0] -> {3}` into `decks[1] -> {3, 7}`. In code:

```
    if(input[i] > input[decks[decks.size()-1]]){
        decks.push_back(i);  // Form a longer IS
    }
```

- **Case 2:** The general case. We want to find the "closest" (refer to definition above) deck which we can extend with the current element. We can do a simple loop:

```
    for(int j = decks.size()-2; j>=0; j--){
        if(input[i] > input[decks[j]]){
            decks[j+1] = i;  // Extend the IS with length j+1 (i.e. decks[j])
            // to form an IS of length j+2 (i.e. decks[j+1]).
            // Note that this means that we are replacing the IS
            // with length j+2 (i.e. decks[j+1])
            // with a newer IS whose last value is input[i].
            // This newer input[i] should be less than the previous value
            // input[decks[j+1]]. This way, the IS with length j+2
            // has a last element that is as small as possible,
            // which is a greedy method of making sure that we can
            // extend that IS with the next elements as much as possible.
            break;
        } else {
            continue;
        }
    }
```

- **Case 3:** If `input[i]` is less than the first entry (i.e. `input[decks[0]]`). This is simply just like Case 2, but we are essentially extending an empty IS (length zero) to form a better IS of length one. As before, we want to keep the last (and only element) of the IS to be as small as possible, so that there are more chances of extending it to form increasing subsequences of length two and above. Code:

```
    if(input[i] < input[decks[0]]){  // Make the IS of length one
        decks[0] = i;                // as small as possible
    }
```

Putting it all together:

```
    for(int i = 0; i<N; i++){
        if(decks.empty()){  // Initial case
            decks.push_back(i);
        } else if(input[i] > input[decks[decks.size()-1]]) { // Extend largest IS
            decks.push_back(i);
        } else if (input[i] < input[decks[0]] ) {  // Extend empty IS
            decks[0] = i;
        } else {  // Find the deck with the "closest" last element
            for(int j = decks.size()-2; j>=0; j--){
                if(input[i] > input[decks[j]]){
                    decks[j+1] = i;
                    break;
                } else {
                    continue;
                }
            }
        }
    }
```

With the above in mind, let's proceed with the rest of the array.

For `i=2, value="1"`, the third case is applicable, as "1" is less than `input[decks[0]]=3`. Thus, we will have:

```
    decks[0] = 2 (value=1) -> {1}  // Update
    decks[1] = 1 (value=7) -> {3,7}  // Unchanged
```

Notice that we no longer keep track of the IS ending with "3". However, what we formed earlier still remains, and we can keep track of it even if we just know that `decks[1]` ends with "7" (see the exercise about printing below). The initial IS of length one ending at "3" is no longer relevant from this point on—we have formed all the relevant subsequences that we could from it (for this example, just {3, 7}). From this point on, we will be able to consider more increasing subsequences by considering the IS of length one to be {1}. Thus, if we want to make an IS of length two, for example by using a higher-indexed element with value "2", we will be able to do so. In this case, had we kept the earlier value of `input[decks[0]]=3`, we would not be able to form this IS of length two ending with value "2".

Next, `i=3, value="8"`. We can extend the LIS so far ({3,7}), so we just need to append it to `decks` (case 1). We now have:

```
    decks[0] = 2 (value=1) -> {1}
    decks[1] = 1 (value=7) -> {3,7}
    decks[2] = 3 (value=8) -> {3,7,8}  // New LIS!
```

`i=4, value="4"`. The general (second) case applies here. Between the decks (ISs) ending at 1, 7, and 8, the "closest" would be to extend {1} to form the IS {1,4}:

```
    decks[0] = 2 (value=1) -> {1}
    decks[1] = 4 (value=4) -> {1,4}  // Old IS {3,7} deleted!
    decks[2] = 3 (value=8) -> {3,7,8}
```

Thus, for the IS with length two, we are now concerned with {1,4} and do away with {3,7}, as the former ends in a lower number and is more likely to be extended. Note that while {3,7} has been "deleted", our work from it is still embedded in the remaining IS {3,7,8}.

`i=5, value="5"`. With case 2 once more, we can extend `decks[1]` to form a better 3-length IS {1,4,5}.

```
    decks[0] = 2 (value=1) -> {1}
    decks[1] = 4 (value=4) -> {1,4}
    decks[2] = 5 (value=5) -> {1,4,5}  // Old IS {3,7,8} deleted
```

`i=6, value="6"`. Finally, for this last element case 1 applies once more:

```
    decks[0] = 2 (value=1) -> {1}
    decks[1] = 4 (value=4) -> {1,4}
    decks[2] = 5 (value=5) -> {1,4,5}
    decks[3] = 6 (value=6) -> {1,4,5,6}  // New LIS!
```

At this point, we have gone through all the N elements in our array. The final length of `decks` is also the length of our LIS. If we used a predecessor array, we would be able to print the elements of our formed LIS.

Note that since we have an inner **for** loop, our implementation has an overall $O(n^2)$ complexity. However, since every time we insert an element, we are putting it in its "closest" place, `decks` is always sorted. Thus, we don't actually have to do a linear search every time—we can do a binary search to speed up the insertion! This way, the overall complexity is brought down to only $O(n \log n)$. See the code below:

```
    vector<int> decks;
    for(int i = 0; i<N; i++){
        if(decks.empty()){
            decks.push_back(i);
        } else if (input[i] > decks[decks.size()-1]) {
            decks.push_back(i);
        } else {
            // Do a binary search for the speed-up
            int lo = 0;
            int hi = N-1;

            while(lo <= hi){
                int mid = (lo+hi)/2;
                if(input[i] < input[decks[mid]]){
                    hi = mid-1;
                } else {
```

```
17                          lo = mid+1;
18                      }
19                  }
20                  decks[lo] = i; // Confirm why this is so
21              }
22      }
```

3.2.1 Exercises

- Verify that the above binary search formulation is correct. (100 Points)
- Print the elements of the LIS that you found. You can do the usual way of using a predecessor array. (100 Points)
- What will change if the elements of our `input` array are no longer unique? (100 Points)
- What will change if we want nondecreasing (i.e., not strictly increasing) sequences instead? (100 Points)
- Confirm that your implementation is correct by going through the LIS problems and variants in Chapter 3 of uHunt.
  - UVa 481 - What Goes Up. Direct $O(n \log n)$ LIS with printing of solution. (200 Points)
  - As the LIS also has an $O(n^2)$ solution, for the problems in this part, please send your code as well so that we could see how you implemented your $O(n \log n)$ solution. (200 Points each)
- The LIS can also be found in $O(n \log n)$ using segment trees. Explore this approach. (100 Points)

4 Hashing

In this section, we will learn all about **hashing**.

Hashing can be easily summarized: It is simply the technique of mapping objects from a large set to a (usually) smaller set. That's it! Now, clearly this needs some elaboration, i.e., why is that useful? We'll go into that soon enough.

A **hash function** is a function from some large set to some other (usually smaller) set. Note that we didn't impose any other restrictions on hash functions. We just give them a special name "hash function" since we'll be using them for hashing, but any function can technically be considered a hash function, although as we shall see later on, some functions are better than others as hash functions. This will be made more precise later on.

Let $h : S \to T$ be a hash function. If $x \in S$, then we say that $h(x)$ is the **hash of** x (with respect to h). Any value in T is called a **hash**, or a **hash value**.

The key property of hash functions that we will use is this:

Property 4.1. *The hashes of two equal objects are equal. In other words, for any hash function h, if $x = y$, then $h(x) = h(y)$.*

This is of course a trivial assertion, but it emphasizes one of the important uses of hashing. Hashing is generally used as a substitute (or partial substitute) to *equality checking*. The idea is that if we want to check whether two things x and y are equal, we first check if their hashes, $h(x)$ and $h(y)$, are equal. If not, then x and y are definitely not equal!

Obviously, the other direction is not true; two objects are not necessarily equal even if they have the same hash. When two unequal objects have the same hash, we call it a **collision**. If we assume that $|S| > |T|$, then collisions are inevitable; this is basically the pigeonhole principle.

We start with one of the most basic usages of hashing: hash tables.

4.1 Hash tables

Let's say we want to implement a familiar abstract data type called **map**. We define a map to be a data type consisting of a collection of key-value pairs such that each key appears at most once, and we define the operations to be:

- **insert** k v. Insert a key-value pair with key k and value v, if k doesn't already exist as a key.
- **contains** k. Check whether k exists as a key.
- **get** k. Return the value v associated with key k if k exists as a key.

- **delete** k. Remove the entry with key k if it exists.

There's a related abstract data type called **set** which is similar to a map but only has keys instead of values. A set can trivially be implemented with a map, so we may simply focus on maps.

You've probably already used a map in the form of C++'s `map`, and you've probably already learned that they can implemented with *binary search trees*.[2] However, they can also be implemented in a different way using hashing. The resulting structure will be called **hash tables**.

In general, we can build a hash table for any type of key as long as they're *hashable to integers*, i.e., there is a hash function that takes a key to some integer. But for now, let's assume that the keys will be integers.

To start with, let's consider the case where the integer keys are bounded above by some number, say m. In this case, we can easily implement a map by just using an array of size m. The code would look very simple, like the following:

```
const int m = 100000;
// assumes that the values are strings.
class Map {
    string table[m];
    bool present[m];
public:
    Map() {
        for (int i = 0; i < m; i++) present[i] = false;
    }
    void insert(int key, string value) {
        table[key] = value;
        present[key] = true;
    }
    bool contains(int key) {
        return present[key];
    }
    string get(int key) {
        return table[key];
    }
    void erase(int key) {
        table[key] = string();
        present[key] = false;
    }
};
```

This clearly works. However, there are a few downsides:

- It doesn't work for keys not in the range $[0, m-1]$.
- Even if the keys are in that range, this method wastes a lot of space if only a few keys are inserted. It would be desirable to have a structure whose size scales proportionally with the number of keys.

[2] In fact, C++'s `map` is (usually) implemented internally using binary search trees.

- Obviously, if the keys are not integers anymore, then this doesn't work.

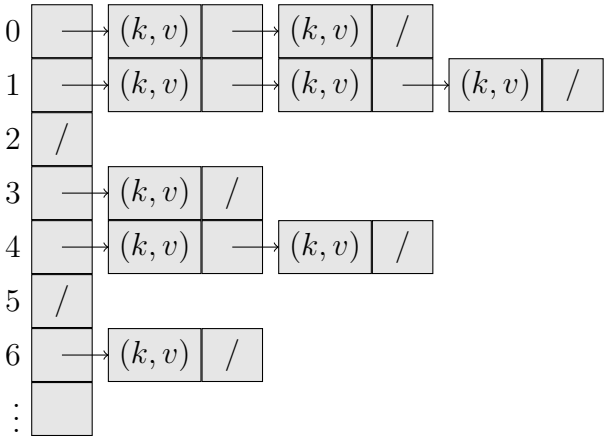
At this point, we can "fix" this by using hashing. The idea is to use a hash as a substitute for our key/index. Specifically, if S is the set of possible keys, and h is a hash function that takes elements from S to the set $\{0, 1, \ldots, m-1\}$, then the above code can be modified to:

```
const int m = 100000;
int h(int x) {
    ... // implementation of h. It must return an integer in the range [0, m).
}
class Map {
    string table[m];
    bool present[m];
public:
    Map() {
        for (int i = 0; i < m; i++) present[i] = false;
    }
    void insert(int key, string value) {
        int hash = h(key);
        table[hash] = value;
        present[hash] = true;
    }
    bool contains(int key) {
        return present[h(key)];
    }
    string get(int key) {
        return table[h(key)];
    }
    void erase(int key) {
        int hash = h(key);
        table[hash] = string();
        present[hash] = false;
    }
};
```

We can then view the earlier implementation as a special case in which we use the identity hash function, i.e., $h(x) = x$.

This solution looks a little better since we're not anymore assuming that the keys are bounded (or even integers), although we will now require a hash function to be present. However, one major downside is that there will be *collisions*. As mentioned before, this is unavoidable if the set of keys is larger than the set of hashes. Thus, this solution will sometimes be incorrect since it will consider objects with the same hash the same object.

There's an obvious fix here. Instead of storing only a single value for every hash, we store a *list* of key-value pairs for every hash. And whenever two keys have the same hash, we revert to linear search.



Each list is called a *chain*, and the hash function is now used to map each key to the correct chain. Each chain can be implemented as a normal list structure.

With this change, we finally get a correct implementation a map. This technique is called **chaining**.[3]

There's one more issue we need to address. Our structure assumes that the number of chains, m, is fixed. Which m should we use? It should be clear that there's no single m that satisfies our requirements, because:

- If m is too small, then the pigeonhole principle guarantees that there will be a chain that's at least $\lceil n/m \rceil$ elements long, where n is the number of key-value pairs present. Thus, the operations will be slow, especially those on that chain.
- If m is too large, then we can't make too many hash tables since they will take up too much space.

What we really need is to make our structure's size *dynamic*, i.e., its size should depend on the number of keys present. To do so, we will need two things:

- A *resizing scheme*, i.e., a rule that determines when and how we will perform a resize.
- A *hash family*. Since m now varies, a single hash function h is not sufficient anymore; we must now have access to a collection of hashes $\{h_m\}$, one for every distinct size m.

Resizing scheme. We would like the number of chains m to depend on the number of key-value pairs, say n. Ideally, we would like $m = O(n)$ so that a hash table takes up $O(n)$ space.

For this, we can simply do something similar to the `vector` resizing scheme; if the ratio of the number of keys and the number of chains goes above some threshold U or below some threshold L, then we resize by some constant factor.

[3]There are other techniques to handle collisions, e.g., *open addressing*, but those are more confusing to implement. You may choose to learn it from the Wikipedia page: https://en.wikipedia.org/wiki/Hash_table#Open_addressing

The following code shows one way to do it. It tries to maintain that the *load factor* $\frac{n}{m}$ lies between two fractions L and U.

```
int m;// current number of chains
int n;// current number of key-value pairs
double L = 1/3., M = 1/2., U = 2/3.;

void resize(int new_m) {
    ... // do the resizing here, from m to new_m.
    m = new_m;
}

void try_rehash() {
    double load = double(n)/m;
    if (!(L <= load && load <= U)) {
        int new_m = max(1, int(n/M)); // we want n/m ~ M (and also m != 0).
        if (m != new_m) resize(new_m);
    }
}
```

Similarly to vectors, one can show that the amortized running time of inserts and deletes becomes $O(1)$, assuming that $L < M < U$ and that `resize` takes $O(m + new\_m)$ time.

Hash family. Next, we need a *hash family*. We define a **hash family** on the set S as an infinite sequence $(h_1, h_2, h_3, \ldots)$ of hash functions such that h_m is a hash function from S to the set $\{0, 1, \ldots, m-1\}$.

For example, one particularly simple hash family on the integers is as follows: $(h_1, h_2, h_3, \ldots)$ where h_m is defined as $h_m(x) = x \bmod m$.

This sounds terrifying since we now also need to create a family of hashes instead of just a single hash! But luckily, there's a particularly simple way of turning a single hash function into a family of hash functions. Given a single hash function h that hashes objects into (possibly unbounded) integers, we can turn it into hash family using the modulo operator. Given h, we define its *modulo family* to be the hash family $(h_1, h_2, h_3, \ldots)$ where h_m is defined as $h_m(x) = h(x) \bmod m$. Thus, for our purposes, we still want to talk about only requiring a single hash function h; our hash table implementation itself can just turn it into a hash family by reducing the hashes modulo m.

Performance. We would now like to analyze the performance of this implementation. However, it's pretty clear that the performance depends heavily on the choice of hash function h. A poor choice of function h could lead to a bad performance since it's possible for several distinct keys to have the same hash. In the extreme case, if h is a *constant* function, say $h(x) = 0$, and this will cause *all* keys to end up at a single chain! Thus, the operations become $O(n)$ in the worst case.

So how do we fix this? One idea would be to find a *really good hash function* that doesn't trigger this worst case. What we want is for the keys to somehow be uniformly distributed among the hash

values $0, 1, \ldots, m-1$. That way, inserting n distinct keys will give us a maximum chain length of $\lceil n/m \rceil$, which if you remember we're trying to maintain $O(1)$.

But there's a big problem: there's no such hash function! To see this, suppose that there are s theoretically distinct keys possible. (s may be infinite.) Then the pigeonhole principle guarantees that for any hash function to the set $\{0, 1, \ldots, m-1\}$, there will be some hash value such that at least $\lceil s/m \rceil$ keys map to that hash value. Since s is usually very large, even infinite, this means that any hash function is (theoretically) breakable: we can always find several distinct keys, all with the same hash.

In the case of our hash table, this means that for any hash function, the worst case $O(n)$ per operation can always be triggered (by a particular set of keys).

Instead, what we want is for our hash values to be "unpredictable" or "random" in some way, that is, we want collisions to be hard to find so the worst case is hard to trigger. For example, the simple hash family $h_m(x) = x \bmod m$ doesn't satisfy our requirements since it's quite simple to trigger collisions. (How?)

To be more precise, we would like each hash function h_m to satisfy the following:

- *h_m uniformly distributes the set of keys to the set of hashes* $\{0, 1, \ldots, m-1\}$. In other words, if we randomly choose a key x from the set of all possible keys, then its hash can be anything in $\{0, 1, \ldots, m-1\}$ with (roughly) equal probability.

  Note that, in practice, this is not a strict requirement; we can still use hash families that don't distribute the keys uniformly. However, the further it is from uniform, the worse the performance of the hash table will get.

- *It is difficult to find collisions.* Here, "difficult" depends on context. Usually, it means it is *computationally difficult* to find collisions, i.e., it takes an huge amount of computational power to find a collision, and even then, "huge" is deliberately vague and depends on the application.

  For example, if you're working on a high-security application, then maybe you'd want it to take years before finding even a single collision, but if you're, say, competing in Codeforces, then maybe it's enough to ensure that finding a collision is not possible within a few hours.

Let's call a hash function satisfying both properties above a **good hash function**.

Now, given a good hash function, we can usually assume that the set of input keys are essentially "random", since there's no way for us to relate any two distinct keys together. We will assume that this is the case during the analysis. This is called the simple uniform hashing assumption. One simple consequence of this assumption is that given any two random keys x and x', the probability that $h(x) = h(x')$ can be assumed to be $\frac{1}{m}$.

So let's assume we can come up with a hash family consisting of good hash functions, and let's again try analyzing the running time of our dynamically-resizing hash table. For now, we can ignore the cost of dynamically resizing since they can be *amortized away* just like in vectors. Thus, we can now consider the cost of any operation on key x to be proportional to the length of the chain

corresponding to its hash $h(x)$. Let's now compute the *expected* length of the chain.

Suppose that there are n distinct keys inserted to our table: $x_1, x_2, \ldots, x_n$. Then the expected length of the chain can be computed using the linearity of expectation as

$$\underbrace{\frac{1}{m} + \frac{1}{m} + \ldots + \frac{1}{m}}_{n} = \frac{n}{m}.$$

But remember that the structure tries to maintain that $\frac{n}{m} \leq U$, therefore, the expected length of the chain is $O(1)$! This is why we usually say that hash maps run in $O(1)$ expected time.

Most programming languages have a built-in implementation of hash tables. For example, C++ has `unordered_set` and `unordered_map` and Java has `HashSet` and `HashMap`. Python's built-in `set` and `dict` are also implemented using hash tables.

Note that maps implemented with binary search trees require the keys to be totally ordered by some comparison operation `<`. The hash table doesn't require comparison; instead, it requires the keys to be *hashable* to integers, i.e., you need to implement hash functions from your data type to $\mathbb{Z}$ such that two equal objects have the same hash. The way to do that depends on the language:

- For C++, you need to implement `std::hash<T>` where `T` is your data type. You also need to implement equality checking by overloading **`operator`**`==`.[4]
- For Java, you need to implement the `hashCode` method and the equality check method `equals`.
- For Python, you need to implement the magic method `__hash__` and the equality check magic method `__eq__`.

For your custom types, it's not hard to come up with a hash function. However, it's usually better to delegate the hash computation to standard ones, e.g., lists, which in most languages already have built-in hash functions that we can trust to be of good quality.

4.2 List/string hashing

Hashing techniques are applicable well beyond hash tables. Hashing allows us to improve algorithms that require comparing many elements for equality. We usually do this by replacing comparison of objects by comparison of their hashes.

We start with a simple case: hashing *lists*. We will actually be discussing *strings* since a string is just a list of characters, hence a special case, but the techniques will also apply to lists of any (hashable) type. We also get to learn some standard applications of hashing.

[4] See http://en.cppreference.com/w/cpp/utility/hash for more details.

4.2.1 A standard problem: comparing substrings

Consider the following (standard) problem. Suppose we are given some string s of length n. We denote by $s_{i...j}$ the substring starting from index i and ending at index j, inclusive. We are asked to answer q queries. In each query, we are given four integers i_1, j_1, i_2, j_2, and we are asked to determine whether $s_{i_1...j_1}$ is equal to $s_{i_2...j_2}$.

The naive solution simply compares both substrings manually. Since substrings can be up to n in length, this takes $O(n)$ per query and $O(qn)$ overall, which is pretty slow.

We can slightly improve this by noticing that we don't need to compare two strings of different lengths. In other words, if $\mathit{length}(s_{i_1...j_1}) \neq \mathit{length}(s_{i_2...j_2})$, then we don't have to compare them at all since we already know they're not equal. Additionally, $\mathit{length}(s_{i...j})$ is simply $j - i + 1$, so this allows us to answer some queries in $O(1)$!

We can restate this improvement in terms of hashes. Consider *length* to be a hash function. Then this hash function allows us to reduce the number of queries to answer by simply not comparing substrings with different hashes.

Unfortunately, it may happen that all pairs of query strings are of equal length, so the worst-case running time is not improved and is still $O(qn)$. The main reason for this is that *length* is not a particularly good hash function; collisions are easy to find! For example, "`ab`" $\neq$ "`cd`" but length("`ab`") = length("`cd`"). So clearly, we want to use a hash function that's better than *length*.

Let's use a different one, then. Let's define *sum* as the function that returns the sum of the ASCII values of the letters of the string. For example,

$$\mathit{sum}(\text{“}\texttt{banana}\text{”}) = 98 + 97 + 110 + 97 + 110 + 97 = 609.$$

Now, consider using *sum* as the hash function. Well, this seems better than *length*; unfortunately it's still not good enough since it's still not hard to break, that is, it's not hard to find collisions! (How?)

We can also use a hash called *sumlength* that's basically a combination of *sum* and *length* by defining it as follows:

$$\mathit{sumlength}(s) = \mathit{sum}(s) \cdot 2^{32} + \mathit{length}(s).$$

This hash function has the property that $\mathit{sumlength}(s_1) = \mathit{sumlength}(s_2)$ if and only if $\mathit{sum}(s_1) = \mathit{sum}(s_2)$ and $\mathit{length}(s_1) = \mathit{length}(s_2)$ (at least in cases we're interested in) and so is strictly stronger than both *sum* and *length*. However, it's still quite easy to break. (How?)

4.2.2 The hashing solution

Whatever function h we choose, our general strategy seems to look like the following:

- Fix a hash function h that takes strings as argument.
- For every query (i_1, j_1, i_2, j_2):

- If $h(s_{i_1 \ldots j_1}) \neq h(s_{i_2 \ldots j_2})$, then output **no**.
- Otherwise, compare $s_{i_1 \ldots j_1}$ and $s_{i_2 \ldots j_2}$ manually. If they are equal, output **yes**, otherwise output **no**.

Ideally, h should be a good-enough hash function, i.e., it should detect unequal strings fairly well so that we minimize the number of times we have to manually compare the strings. I say "good enough" since we are still open to the possibility of *collision*.

The best case is when there are no collisions, but as I've said before, hash functions without collisions are impossible if the set of hash values is smaller than the set of inputs. Of course, we can get rid of collisions totally if we allow our hash values to be arbitrarily large, but such functions are not useful since they are usually slow to compute; indeed, if h has no collisions, then $h(t)$ must essentially encode all information needed to reconstruct t, and that usually means the hash value will be proportional in length to t! Thus, while we get rid of collisions, the running time isn't actually better than simply comparing the strings. As an example, we could choose our hash function to be the identity $h(t) = t$, and we get rid of collisions, but notice that the algorithm reduces to brute force.

Thus, we impose the additional constraint that $h(s_{i \ldots j})$ can be computed quickly. Note that this also implies that the hash values are "small". Ideally, we'd like $O(1)$, but even something like $O(\log n)$ should be fine.

However, there's a problem with this approach even if we have a hash function satisfying all these properties: the worst case running time is still $O(qn)$ since it may happen that all pairs of query strings are actually equal! This is true no matter what h is.

Let's try to fix this by using the following alternative "solution":

- Fix a hash function h that takes strings as argument.
- For every query (i_1, j_1, i_2, j_2):
  - If $h(s_{i_1 \ldots j_1}) \neq h(s_{i_2 \ldots j_2})$, then output **no**.
  - Otherwise, output **yes**.

In other words, we completely skip the manual comparison step and use the hash function as a complete substitute for comparison!

The good news is that this is now pretty fast; specifically, the worst case cost becomes $O(q \cdot t(n))$ where $t(n)$ is the cost of computing the hash of a single substring. The bad news is that this is no longer completely correct because of collisions! And as we have seen, collisions are pretty much unavoidable.

But how bad is it, exactly? Well, let's see. Assume that the hash function h is also a *good* hash function in the sense described in Section 4.1. Thus, we can assume that two distinct strings hashing to the same value only happens by chance,[5] since one of the requirements of being *good* is that

[5] and not, say, because of the problem setter triggering it with suitably-chosen test data.

collisions are hard to find. And if we assume that the hash function takes strings to integers in, say, $\{0, 1, \ldots, m-1\}$, then by the uniformity property of h (part of being *good* again), the chance of two unequal strings getting the same hash is only $\frac{1}{m}$. Obviously, our algorithm never incorrectly says that two equal strings are unequal. This means that the probability of answering a single query correctly is $1 - \frac{1}{m}$, and assuming independence of queries, the probability of correctly answering all queries is

$$\left(1 - \frac{1}{m}\right)^q.$$

Clearly, this is strictly less than 1. But if we assume that $m \approx 10^9$ and $q \approx 10^5$, then the chance of succeeding is[6]

$$\approx \left(1 - \frac{1}{10^9}\right)^{10^5} \approx 1 - \frac{10^5}{10^9} \approx 99.99\%,$$

which is great! Furthermore, if there are 30 test cases, then the probability of our submission getting all test cases correct is

$$\approx \left(\left(1 - \frac{1}{10^9}\right)^{10^5}\right)^{30} \approx 1 - \frac{3 \cdot 10^6}{10^9} \approx 99.7\%,$$

which is a pretty good bet, if you ask me! Thus, the algorithm above will get accepted with a very high probability. We just have to live with the fact that it has a tiny chance of failing.

The only thing remaining is to actually choose our hash function. As we have seen, *length*, *sum* and *sumlength* are not good. The problem with *length* is that it doesn't encode any information about the characters of the string, it only encodes length (obviously), and the problem with *sum* is that, while it does encode information about the characters, it doesn't encode anything about their positions in the string.

We need a hash function that's strong enough that we can say it's *good*, but also with the property that the hash of substrings can be computed easily. Luckily, there exists such a hash function!

Fix two numbers b and m, and a function v that maps characters to integers (a simple example is the map that takes a character to its ASCII value). We define the **polynomial hash function** $P_{b,m,v}$ as the hash function that takes the string s to

$$P_{b,m,v}(s) = (v(s_0) + v(s_1)b + v(s_2)b^2 + v(s_3)b^3 + \ldots) \bmod m = \left(\sum_{i=0}^{|s|-1} v(s_i)b^i\right) \bmod m.$$

In other words, we consider the characters of s as the coefficients of some polynomial (after converting them to integers via v), and evaluate that polynomial on b modulo m.

The nice thing about the polynomial hash is that it's strong enough that we can consider it *good*,[7] but that it's nice enough that there's a way to preprocess the string so that polynomial hashes can be computed in $O(1)$! We will leave the details of the latter to the reader.

[6] using the approximation $(1 + x)^n \approx 1 + nx$ which is pretty good if $|nx| \ll 1$.

[7] Unfortunately, it is very difficult to formally prove that polynomial hashes are good, and in fact, it heavily depends on b and m. But you should be convinced of that intuitively by accepting that polynomials are "random enough" for many choices of b and m. Sadly, this is all I can offer for now.

Thus, by using the polynomial hash, we end up with an algorithm that answers all queries in $O(n+q)$ time.

One final thing: the quality of the polynomial hash depends strongly on the choice of b and m (and to a lesser extent, v). For example, choosing $b = 1$, the polynomial hash reduces to *sum*, which is pretty bad. Ideally, we want a b such that the sequence $b^0, b^1, b^2, \ldots$ doesn't repeat early modulo m. We also want b to not be too small, e.g., $b = 2$ is a bad choice. (Why?) The choice of m is also important. First of all, it affects the probability of success, so we want to choose m that's sufficiently large. There are other considerations, e.g., it turns out that a prime m is a better choice, since other natural choices like 2^{64} turn out to be somewhat weak, and breaking cases for them are quite well-known now; see http://codeforces.com/blog/entry/4898. There might be other considerations, but in general, it's very hard to say whether a choice of b and/or m is good or bad; I mean, it should be pretty clear that analyzing this further leads us deeper into number theory!

Further reading on string hashing: https://www.mii.lt/olympiads_in_informatics/pdf/INFOL119.pdf

4.2.3 Another standard problem: counting substrings of length k

To familiarize ourselves further with hashing, and to also illustrate a common pitfall with hashing techniques, let's consider another standard problem: Given a string s and an integer k, how many distinct substrings of length k are there?

The most obvious solution is the following:

- Let s be a set, initially empty.
- For each substring t of length k, insert t into s.
- Return the size of s.

This is clearly correct, but it is slow. There are $n - k + 1$ substrings of length k, and if $k \approx \frac{n}{2}$, then $n-k+1$ and k are both $\Theta(n)$, so the total complexity is $O(n^2 \log n)$ where the log factor comes from the usage of the set. Even if we use a hash table with a good hash function for our set, this is still $O(n^2)$ expected time!

Now, if we have a hash function, then we have the following faster "solution":

- Let s be a set, initially empty.
- Let h be a hash function that maps strings to integers.
- For each substring t of length k, insert its hash $h(t)$ into s.
- Return the size of s.

The idea is that even though the strings are large, the hashes themselves are small. Unfortunately,

this is not entirely correct because of collisions!

How can we fix this? Well, maybe we can choose to h to be a *perfect* hash function, i.e., one that has no collisions. However, as explained earlier, such a hash function might take too long to compute, so this is probably no better than brute force; we've just made it more complicated.

A better solution would be to use a *good* hash function. Thus, we get a solution that's correct with some probability. However, it can't be just any good hash function; it must also have the property that we can compute the hashes of all length-k substrings quickly.

Clearly, the *polynomial hash* function still works since it allows us to compute the hash of any substring in $O(1)$ time. This gives us an algorithm that works in $O(n \log n)$ time. But actually, the requirements in this particular problem are a bit weaker, since we only need to compute the hashes of length-k substrings. Specifically, it is enough for our hash function h to satisfy the following properties:

- **Right insertion.** If s is a string and c is a character, then $h(sc)$ can be computed quickly from $h(s)$.
- **Left deletion.** If s is a string and c is a character, then $h(s)$ can be computed quickly from $h(cs)$.

If h satisfies these properties, then we can easily compute the hashes of all length-k substrings by repeatedly inserting and deleting letters on both sides and computing the updated hash. We call a hash function satisfying the above a **rolling hash** function.

A simple example of a rolling hash function is *sumlength*, but as we've seen, this is not a good hash function. Luckily, the polynomial hash function is also a rolling hash:

- **Right insertion.** If we insert c to the right, we only need to add $v(c)b^{|s|}$ modulo m to the original hash, i.e., $h(sc) = \left(h(s) + v(c)b^{|s|}\right) \bmod m$. This can be done in $O(\log |s|)$ with fast exponentiation, or even $O(1)$ by precomputing the powers of b (since it is fixed).
- **Left deletion.** Similarly, we find out that $h(cs) \equiv v(c)+b \cdot h(s) \pmod m$, thus, we can compute $h(s)$ from $h(cs)$ as follows: $h(s) = b^{-1}\left(h(cs) - v(c)\right) \bmod m$, where b^{-1} is the modular inverse of b mod m. This only works if b is invertible mod m, but that's easy to guarantee; choosing a prime m is enough. Also, b^{-1} can be precomputed, so computing $h(s)$ from $h(cs)$ only takes $O(1)$ as well!

Hence, by using polynomial hashes and a good choice of b and m, we can find the number of distinct substrings of length k in $O(n \log n)$ time!

4.2.4 The probability of failure

Now that we've come up with an algorithm, and assuming we've selected b and m so that the corresponding polynomial hash is good, do we stop there? Unfortunately, the answer is **no**: We can't

just assume that the probability of success is high, even if our hash function is good!

To see why, let's try to compute the probability of success of our algorithm for the previous problem. Clearly, we will only succeed if, among all length-k substrings, there are no two strings with the same hash. Let's compute that probability in the worst case. For simplicity, let's assume $n = 10^5$ and $m = 10^9$. Note that there are n distinct substrings in the worst case.

This problem turns out to be well-known. This is commonly called the **birthday paradox** and is usually stated as follows: What are the chances that, among n people, at least one pair of people have the same birthday? We say it's a paradox because of the surprising/counterintuitive result that the probability can be very high even with just a few people. Indeed, for just $n = 23$, this probability is greater than 50%, and for $n = 60$, the probability is greater than 99%. This is in spite of the fact that there are 365 days in a year!

Generalizing, we may assume that there are m days in a year, and similarly it turns out that even when $n \ll m$, the probability of failing can get pretty high. To approximate the probability, we assume that all comparisons among the $\binom{n}{2}$ pairs are independent.[8] The probability that all comparisons are all different becomes

$$\approx \left(1 - \frac{1}{m}\right)^{\binom{n}{2}}.$$

If we choose $n = 10^5$ and $m = 10^9$, we get the probability of success as[9]

$$\left(1 - \frac{1}{10^9}\right)^{\binom{10^5}{2}} \approx \left(1 - \frac{1}{10^9}\right)^{10^{10}/2} \approx e^{-5} \approx 0.67\%,$$

which is very bad! This will be even worse if you consider that there are multiple test cases; the probability of getting even just 3 test cases right is

$$\approx \left(e^{-5}\right)^3 \approx 0.0000306\%,$$

which is basically zero. Thus, our solution will almost certainly get rejected, even assuming that our hash function is *good*.

The lesson here is to always compute the chances of your submission being correct when using hashing techniques, even just approximately.

To fix this, we simply have to realize that $m \approx 10^9$ is too low. We need to increase it so that the probability of succeeding becomes higher. The obvious way is to make it so that $m \approx 10^{18}$, that way, by doing the same derivation above, we get the probability of succeeding as

$$\approx \left(1 - \frac{1}{10^{18}}\right)^{10^{10}/2} \approx 1 - \frac{10^{10}/2}{10^{18}} \approx 99.9999995\%,$$

which is basically 100%.

The downside of using $m \approx 10^{18}$ is that computing polynomial hashes will give you some nasty overflow issues, since multiplying two 64-bit integers will probably overflow! Instead, a different

[8] this approximation is pretty good if $n \ll m$

[9] using the approximation $\frac{1}{e} \approx \left(1 - \frac{1}{n}\right)^n$ which is quite good when n is large

solution is to simply use *two* polynomial hashes, h_1 and h_2, with different moduli m_1 and m_2, both $\approx 10^9$, and combine both hashes into one (say by forming a pair, or combining them like $h_1(x) \cdot 2^{32} + h_2(x)$). Thus, the probability of failing is $\frac{1}{m_1 m_2} \approx \frac{1}{10^{18}}$ and so we again get the high probability of succeeding as above. But now, the moduli are small, so there are no overflows!

The main lesson is that you should always check if the probability of your hashing solution succeeding is high enough. As a rule of thumb, if your solution requires c comparisons and the probability of a comparison failing, i.e., saying **yes** when it should be **no**, is p, then you should expect the probability of succeeding to be $(1-p)^c$. Typically, $p = \frac{1}{m}$ where m is the number of distinct hashes of of your hash function. But note that c can be really huge, as seen in the example above. As a rule of thumb, you can estimate c to be $\approx n$ if you have to perform n independent pomparisons, but $\approx \frac{n^2}{2}$ if you have to compare n objects overall, say when counting the number of distinct objects.

4.3 Set hashing

Aside from strings/lists, we also sometimes need to hash *sets*. Obviously, there are simple ways of hashing a set, e.g., we can just sort the values and compute the list hash, but in some applications, we would like our hash function to have other nice properties, e.g., something similar to the *rolling* property.

For example, consider the following problem. You are given a set, initially empty. You are also given q operations. Each operation either inserts an element to or deletes an element from the set. Your task is to find the number of distinct *states* that the set has been in over the course of the q operations.

This is very similar to the distinct-substring problem, except we're dealing with sets. This time, we would like our hash function to satisfy the following properties (on top of being *good* as well):

- **Insert.** If we insert an element, then the new hash can be computed quickly.
- **Delete.** If we delete an element, then the new hash can be computed quickly.

Using such a hash function, we can easily solve the problem. (How?)

All that remains is finding a hashing scheme that allows us to do the above. Try it yourself first before proceeding!

4.3.1 Maintaining sorted hashes

One natural solution would be to start with the set hash I mentioned earlier, which is sorting the values and computing the list hash of the sorted list.

For the list hash, we can again use the polynomial hash. To handle updates, we need to make it dynamic, i.e., we need to be able to update the polynomial hash when we insert or delete elements.

However, this time, the insertion/deletion can be anywhere in the list, not just at the ends, so the rolling property can't be used anymore.

It turns out that we can dynamically maintain the polynomial hash by building a *segment tree* on top of the sorted list! This will give us an $O(\log n)$-time algorithm to maintain and update the hash of the set. I will leave the (somewhat complicated) details to the reader.

One advantage of this algorithm is that it can also be applied to *multisets*, i.e., "sets" that allow duplicate elements.

4.3.2 Zobrist hashing

It turns out that there's a much simpler hashing method than the previous one in the case of sets.

The idea is to first map all possible values to some arbitrary number, say a 64-bit integer. Let's say the value x maps to $v[x]$. Then the hash of a set is simply the XOR of all $v[x]$ for all elements x of the set!

Such a hash is very easy to update. For example, if we insert a new element x that doesn't exist in the set yet, then we simply XOR $v[x]$ with the current hash to get the updated hash. Similarly, if we delete an element x that exists in the set, then we also XOR $v[x]$ with the current hash to get the updated hash. Thus, deletion and insertion are the same operation, and they can be performed very quickly, i.e., $O(1)$!

What's more, this hashing is very good against collisions! If we map each element to a random 64-bit integer at the beginning, then the probability of two distinct sets getting the same hash becomes $\approx \frac{1}{2^{64}}$.

This is called **Zobrist hashing** and it's actually used by some programs that play chess and Go to compute hashes of game states; the ability of the hash to be updated efficiently makes it suitable for it.[10]

Note that this algorithm can't be applied to multisets as it is, but there's a way to modify it so it can also support multisets. I leave it to the reader as exercise.

4.4 Tree hashing

On some occasions, you might find yourself needing to hash *trees* as well. Hashing trees (and graphs in general) is easy if we take the labels into account, since a tree is just a set of nodes V and a set of pairs of nodes E, and we already know how to hash sets.

However, in some problems, we would like to find a hash function that returns the same hash for *isomorphic trees*, regardless of how they're labelled/presented. An example is in the problem Tree

[10] https://en.wikipedia.org/wiki/Zobrist_hashing

Isomorphism.

There are different kinds of trees. For example, we have unrooted and rooted trees. Furthermore, there are ordered and unordered rooted trees. We would like to come up with a hashing algorithm for each of them.

4.4.1 Ordered tree hashing

An *ordered tree* is a rooted tree where the children of each node are ordered in some way. We would like to find a way to hash ordered trees.

Obviously, comparing two ordered trees is pretty simple; just recursively check if the corresponding children are equal. This runs in optimal $O(n)$ time. However, just like in the case of strings, there is still some value in knowing how to hash ordered trees.

It turns out that there's a pretty simple hash for an ordered tree that runs in $O(n)$! I will leave it to the reader to discover as an exercise. *Hint:* Consider the *list* of children.

4.4.2 Unordered tree hashing

On the other hand, in *unordered* trees, the children are not ordered. In other words, we don't distinguish trees even if we rearrange their children.

First, how do you even compare two unordered trees without hashing? Clearly, you can't compare corresponding children anymore since they might not be in the same order in both trees.

To correctly compare two unordered trees, let's first ask: what do you call a collection whose ordering doesn't matter? And how do you compare whether two such collections are the same? The answer is a *multiset*, and to compare them, we simply sort them and compare the corresponding elements!

Now, this idea also gives us a method of hashing the tree. Since we already know how to hash multisets, this gives us a hash for unordered trees as well.

To summarize, to hash an unrooted tree, we first recursively hash the children, and then place the resulting hashes in a multiset, and then return the hash of that multiset (say by sorting and computing the list hash). This runs in $O(n \log n)$ time.

4.4.3 Unrooted tree hashing

What about unrooted trees? Unfortunately, the previous recursive methods will not work anymore since there's no root! Given two unrooted trees, how do we even begin comparing them?

Well, the key here is to use a standard trick when dealing with unrooted trees: root them first! This

is especially helpful since we already know how to hash rooted (unordered) trees. However, which nodes should we choose as roots? Unfortunately, we can't root them arbitrarily since we also need the roots to correspond to each other.

One insight is that we could look for some easily identifiable node in the tree and use that as the root. The easiest one would be the *tree center*: If two unrooted trees are the same and they have unique centers, then rooting them both at the centers will yield equal rooted trees!

This corresponds to a procedure of comparing two unrooted trees, and also a procedure to hash an unrooted tree: find its center, root the tree at the center, and hash the resulting rooted tree.

Unfortunately, this doesn't always work since there are cases when there are two centers! In that case, which center should we pick? Actually, the simplest way would be to simply try *both* centers and return the *smaller* hash! (Can you show why that works?)

The running time is basically the same as that for unordered rooted trees since the center can be found in $O(n)$ time.

4.5 Hashing in cryptography

Beyond hash tables and competitive programming, hashing also plays a role in cryptography. A **cryptographic hash** is simply a hash function that has some properties that make it desirable for use in cryptography. Some examples of useful properties are as follows:

- Collisions are *very* hard to find. In some security applications such as digital signatures, this is important. Usually, the requirements for the difficulty here are much stronger than usual, since we're dealing with security here. In fact, in some modern cryptographic hashes currently in use, no single collision has ever been found![11]

- It's easy to compute, but very hard to *invert*. In other words, given some arbitrary hash value v, it's very hard to find an x whose hash is v, even with the knowledge that such an x exists. This is important in some security applications.

We will not go into more detail regarding the uses of hashing in cryptography since it is beyond the scope of the training material; instead, we refer the interested reader to Wikipedia.[12]

[11] For example, there are no known collisions in the SHA-256 algorithm; any instance of a collision would be very noteworthy and probably worthy of publication. Further reading: https://crypto.stackexchange.com/questions/52578/are-there-any-well-known-examples-of-sha-256-collisions . Obviously, collisions exist, they're just really hard to find.

[12] https://en.wikipedia.org/wiki/Cryptographic_hash_function

4.6 Problems on hashing

1. In our resizing scheme above for hash tables, show that a sequence of n inserts and deletes runs in $O(n)$, starting with an empty table. In other words, the resizing cost is amortized $O(1)$. *Hint:* Assume that $0 < L < M < U$.

2. Show that $\mathit{sumlength}(s_1) = \mathit{sumlength}(s_2)$ if and only if $\mathit{sum}(s_1) = \mathit{sum}(s_2)$ and $\mathit{length}(s_1) = \mathit{length}(s_2)$. State your assumptions.

3. Provide 10^5 distinct strings of lowercase letters with the same *sumlength* hashes. The total length of the strings must not be ridiculously large. Then explain how you found those strings. You may choose to only send me the generator program, but explain how it works.

4. Give an algorithm that takes a string of lowercase letters as input and outputs a different string of lowercase letters with the same *sumlength* hash if and only if such a different string exists. *Bonus:* Implement it!

5. What happens to the polynomial hash if $b = m$? What happens if $b = m - 1$?

6. Provide 10^5 distinct strings of lowercase letters with the same polynomial hashes, for each of the (b, m) choices below (counted as one problem each) and where v takes letters to their ASCII values. The total length of the strings must not be ridiculously large. Then explain how you found those strings. You may choose to only send me the generator program, but explain how it works.

   (a) $b = 2$ and $m = 10^9 + 7$.

   (b) $b = 25$ and $m = 10^9 + 7$.

   (c) $b = 112345$ and $m = 10^9 + 9$.

   (d) $b = 577245688$ and $m = 10^9 + 9$.

   (e) $b = 88876050$ and $m = 10^9 + 7$.

   (f) $b = 123456789$ and $m = 10^9 + 9$.

7. Show how to preprocess a string in $O(n)$ time so that the polynomial hash of any substring can be computed in $O(1)$ time. *Bonus:* Show that this preprocessing can be done without using the modular inverse of b.

8. Show how to preprocess a string in $O(n)$ time to answer the following query in $O(1)$ time with high probability: given a substring, check if it is a palindrome. *Bonus:* Implement it!

9. Show how to preprocess a string in $O(n)$ time to answer the following query in $O(\log n)$ time with high probability: given an index i, find the largest palindromic substring centered at index i. *Bonus:* Implement it!

10. Show an $O(n \log n)$-time algorithm to find the longest palindromic substring with high probability. Be careful: The longest palindromic substring can have even length! *Bonus:* Implement it.

11. Find a good-enough hash function on ordered trees and that can be computed in $O(n)$ time. Explain why it's "good enough".

12. Modify Zobrist hashing so that it can also hash multisets. It should be computable in $O(n)$ expected time. Explain why it's "good enough".

13. Find a good-enough hash function on unordered rooted trees and that can be computed in $O(n)$ expected time. Explain why it's "good enough".

14. Show that we can hash an unrooted tree with two centers by rooting it at both centers, hashing both rooted trees, and returning the smaller hash. (In other words, show that two isomorphic trees get the same hash this way.)

More problems to submit:

1. Spy Syndrome 2
2. Restoring the Expression
3. Tree Isomorphism
4. Palindromic characteristics
5. Sereja and Permutations
6. Chef and Isomorphic Array

5 Rooting and Flattening the Tree

For some graph problems, you may notice that there would be N nodes and $N-1$ edges, indicating that it is a tree. Thus, in these cases, the first step would be to root the tree by calling a DFS from an arbitrary node, if it is not rooted yet. The problem would then usually fall under one which could be solved with the DP on tree/DP on DAG techniques which we have covered in Week 8.

For cases where treating it as a DP problem does not seem to yield results, we can try **flattening the tree**, which may also be known as performing an **Euler tour** on the tree. We can do this by running a DFS from the root, and then noting down each time a node is entered and exited. The tour may look something like the following:

```
int N;  // Number of elements
vector<vector<int>> adjList;  // N x number of children per node

vector<int> flattened;
void f(int node){  // Euler tour/flattening the tree
    flattened.push_back(node);

    for(auto child : adjList[node]){
        f(child);
        flattened.push_back(node);
    }
}

int main(){
    // Assume tree (adjList) has been built
    int ROOT = 0;  // Arbitrary root

    f(ROOT);  // Flatten the tree, to prepare it for further processing

    return 0;
}
```

How long will the `flattened` array be at the end of the tour? Note that for each node, we are pushing to the array twice—on entry and on exit—except for the root, which is not called by any parent. Thus, the total length of `flattened` will be $2N-1$.

We have actually seen this done before. If we keep track of the depths at which the nodes were first discovered, then in the part of the tour where a node u was first discovered up to the part where another node v is reached, we'll be able to pinpoint the closest node which is a direct/indirect predecessor for both—in other words, their lowest common ancestor (LCA)! By doing an Euler tour, we were able to reduce the problem of finding the LCA into a standard one of finding the range minimum query (RMQ), applied on the flattened array.

With the LCA added to our toolbelt, we are able to answer problems which involve questions like the shortest path or distance between two nodes in the tree. However, the LCA is just one of many use cases for tree flattening. A lot of tree data structures problems usually involve many updates

and queries that need to be done efficiently, as in the next example.

5.1 Example: Inversion/Toggling Subtrees

Consider the following scenario:

- You have a rooted tree with N nodes. Each node has a value of either zero or one.
- A node can be toggled, flipping (zero becomes one, one becomes zero) the value of that node and all its descendants (i.e., the whole subtree).
- You also want to be able to know the sum of a subtree (i.e. the total value of a node and all its descendants).

Now let's say that you have Q queries (including both the toggles/updates and the queries asking for the total sum). Typically, the bounds would be set so that an $O(NQ)$ solution is not feasible (e.g. both N and Q are in the order of 10^5. With such bounds, we'd usually want to be able to do the updates or queries in logarithmic time, giving an acceptable overall complexity of $O(Q \log N)$.

Once more, note how we were able to reduce the LCA problem into the RMQ problem, which can be done in $O(N \log N)$ or better by using segment trees or sparse tables. Thus, the first step would be to flatten it using an Euler tour, reducing it into a standard inversion/flipping problem done on 1D arrays.

5.1.1 Inversion on 1D array

For now, assume that the tree has already been flattened, so that we could focus on the subproblem of doing the inversion updates and summing queries in $O(\log N)$ time. Thus, for a given array, we need to be able to:

- `toggle(l,r)` - Flip the elements from `l` to `r`, inclusive. Ones would become zeroes, zeroes would be ones.
- `count(l,r)` - Get the total sum of the elements from `l` to `r`.

One way to go about this is to implement a segment tree with lazy propagation. Lazy propagation would be good here, since flipping a segment twice effectively reverts the elements to their original values.

A possible implementation is shown below. Comments have been included as a guide.

```
1 #define lchild(x) (((x)<<1)+1)
2 #define rchild(x) (((x)<<1)+2)
3
4 int input[100000+10];  // Globals are initially set to zero
```

```
int arr[400000+10];  // Segment tree array set to a rough upper bound of 4N
int lazy[400000+10];  // Practically a boolean type

// After updating/visiting a node, arr[node] should have been
// set appropriately.
void update(int node, int curL, int curR, int uL, int uR) {
    // Account for the lazy propagation
    if(lazy[node]){
        lazy[node] = 0;
        arr[node] = curR - curL + 1 - arr[node];

        if(curL != curR){
            lazy[lchild(node)] ^= 1;
            lazy[rchild(node)] ^= 1;
        }
    }

    // Out of bounds
    if(uR < curL || curR < uL){
        return;
    }

    // Completely within wanted range
    if(uL <= curL && curR <= uR){
        arr[node] = curR-curL+1 - arr[node];

        if(curL != curR){
            lazy[lchild(node)] ^= 1;
            lazy[rchild(node)] ^= 1;
        }
    } else {  // Need to split

        if(curL != curR){
            update(lchild(node), curL, (curL+curR)/2, uL, uR);
            update(rchild(node), (curL+curR)/2+1, curR, uL, uR);

            arr[node] = arr[lchild(node)] + arr[rchild(node)];
        } else {  // Just one element
            arr[node] = 1 - arr[node];
        }
    }
}
```

In this implementation, `toggle()` was implemented as `update()`, to use usual segment tree terminology. `curL` and `curR` refer to the range (left to right indices, inclusive) that we are looking at, and `uL` and `uR` refer to the range that we want to update (toggle/flip).

The `query()` (`count()`) function is very similar, and is shown below:

```
int query(int node, int curL, int curR, int qL, int qR){
    if(lazy[node]){
        lazy[node] = 0;
        arr[node] = curR-curL+1 - arr[node];
```

```
5
6           if(curL != curR){
7               lazy[lchild(node)] ^= 1;
8               lazy[rchild(node)] ^= 1;
9           }
10      }
11
12      if(qR < curL || curR < qL){
13          return 0;
14      }
15
16      if(qL <= curL && curR <= qR){
17          return arr[node];
18      } else {
19          if(curL != curR){
20              int ans = query(lchild(node), curL, (curL+curR)/2, qL, qR)
21              + query(rchild(node), (curL+curR)/2+1, curR, qL, qR);
22
23              return ans;
24          } else {
25              return arr[node];
26          }
27      }
28  }
```

With this implementation in hand, we can go back to solving the problem on the original tree.

5.1.2 Applying the Euler Tour

To use the above implementation, we need to flatten the tree using a DFS, as shown earlier. Modify the DFS (Euler tour) code above so that we could keep track of the indices in `flattened` that correspond to the times when the nodes were first visited. This will be used to convert an update/query on a certain node or subtree of the original tree into a left-to-right operation on the flattened array.

Try to implement the concepts on your own. To see if your implementation was correct, as well as for additional practice, please take a look and try submitting your code for the problems listed below.

5.2 Exercises

- CodeChef: Flipping Coins (200 Points)
- Codeforces 877E: Danil and a Part-time Job (200 Points)
- Codeforces 620E: New Year Tree (200 Points). After flattening the array, you may end up with a solution that would be similar to your solution for CodeChef: Flipping Coins.
- CodeChef: Pishty and Tree (200 Points)

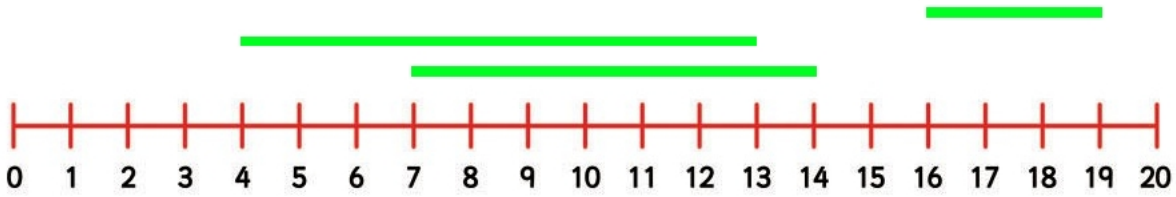
- CodeChef: So Close Yet So Far (200 Points)

6 Coordinate Compression

In some grid-type problems, the entire grid itself is too large to store in an array. Take this problem as an example.

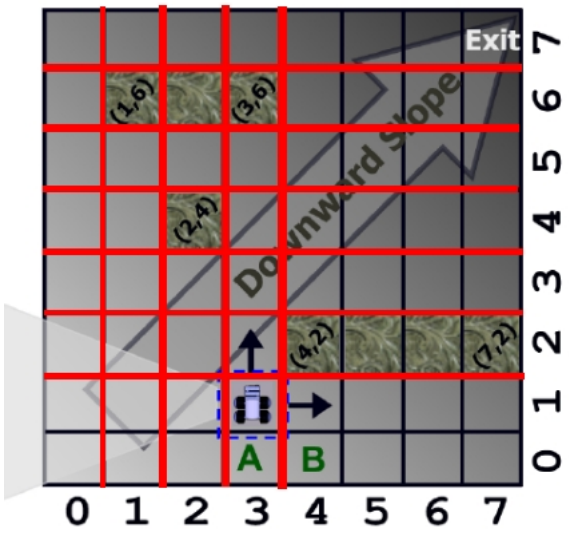
To summarize, we're given a grid, a robot that can only go right or up, and some horizontal walls of thickness 1. We're asked to count the number of starting points for the robot wherein it can't reach the exit (upper-rightmost corner) without going through walls. The grid has size at most $10^6 \times 10^6$, or at most 10^{12} cells. Storing each cell requires an equivalent of 1 Terabyte (TB) of memory – simply too much for any normal computer to handle! However, looking at the constraints, there are at most 1000 walls. Can we somehow make use of this constraint to solve the problem?

Let's take a look at a simpler single-dimensional problem. If we have a 1-dimensional range, and a couple of line segments, how can we represent this more space-efficiently? A picture of the scenario is shown below. The green boxes represent the line segments we're interested in. For this example, we only consider the points at integer positions, and not in between.

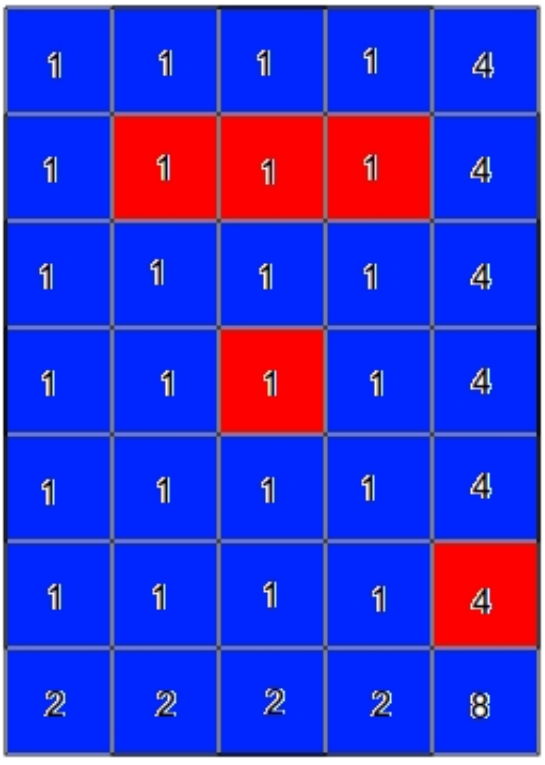


The simple answer to the question is: take note of the endpoints of our line segments, and divide our space accordingly. For the example above, we can notice that for the range $[0, 3]$, there are no line segments covering it; for the range $[4, 6]$, we have one segment covering; for $[7, 13]$, there are two; for $[14, 14]$, there's just one, and so on. All the points within a range we've segmented are uniform, meaning they have exactly the same line segments covering them. With this, we can split our line into $O(N)$ ranges, where N is the number of segments there are within the line.

Going back to our problem before, what can we do for our two-dimensional grid? Since walls can only be horizontal, then it's simple – we can take note of which rows have walls, and which columns do the walls start/end. We then **split** our grid accordingly. The figure below shows how we can split the sample figure given in the problem.



After splitting the grid, each section is uniform – it's either full of walls, or completely empty. The next step is to **compress** each section to a single cell, containing all necessary information. What information to note will depend on the problem, but in our case the type of the cells (walls or empty) and the number of cells within the section should suffice. If we compress the above splitting, then we should have something like the figure below. The numbers within each section indicate how many cells it covers. Blue sections denote empty cells, and red sections denote cells full of walls.



With our now-compressed grid, that's most of the problem down. After compression, we only have

$O(W)$ sections to deal with. The rest of the problem should be easy to solve, and it is left to the reader to provide a solution.

6.1 Exercises

6.1.1 Non-Coding Problems

- In the problem shown above (Tracking Bio Bots), prove that with our splitting scheme, we will end up with at most $O(W)$ sections, where W denotes the number of walls. (100 pts)
- In the same problem, if our walls could be also be vertical, will the number of sections still be $O(W)$ in the worst case? If not, what is it's worst case complexity in terms of W? Include your assumptions, if any. (100 pts)

6.1.2 Coding Problems

- ACM ICPC Live Archive - Tracking Bio Bots (100 pts)
- Hacker Rank - Grand NOI and ICPC Battle (300 pts)

7 Floating Point

7.1 Preface

Open up a Python shell and type the following:

```
>>> 0.3 + 0.15 == 0.45
```

Did the output match your expected answer? For those without access to a shell right now, typing in the above will result to `False`. If we just key in the left-hand side of the expression, we get:

```
>>> 0.3 + 0.15
0.44999999999999996
```

As we see, inaccuracies result because of the way floating points are represented in a computer. These may lead to lots of Wrong Answers in a programming setting. To find out where these come from and what to do about them, read on below.

7.2 Introduction

Floating point types are basic data types that support numbers with fractional parts. In C++, these are your **`float`** and **`double`** types. These numbers can be as small as 0.57, or as large as 10^{10} or even larger (and of course they can also be negative). However, floating points must be handled differently than integer data types. To better understand their differences, it might benefit us to learn how floating point types are stored and handled in memory.

Let's say that we have the number (say 0.375) that we want to store as a variable x. In C++, it's a simple statement: "**`double`** `x = 0.375;`". However, it doesn't store this number the way we see it, which is in *decimal*. Recall that computers store and handle data in a **binary** format, and so representing numbers in a decimal format wouldn't make sense for the computers. Instead, it stores it in a binary format, similar to how integers are stored.

7.3 Recall: Integer Binary Representation

Recall that given any integer, we can express it as a sum of unique powers of two. More formally, an integer x can be represented uniquely by the sequence $a_0, ..., a_{N-1}$, where

$$x = \sum_{i=0}^{N-1} a_i \cdot 2^i$$

and $a_i \in \{0, 1\}$. Given this sequence, the binary representation of x is then $a_{N-1}a_{N-2}\ldots a_1a_0$, as most of you should be familiar with. Here, N depends on how large the integer type is (32-bit `int` has $N = 32$, and 64-bit `long long` has $N = 64$). If our data type is `signed`, then the first bit (a_{N-1}) instead represents the sign of the number (1 if negative, 0 if positive), and so if we denote the sign bit as $s = a_{N-1}$, then we have

$$x = (-1)^s \cdot \sum_{i=0}^{N-2} a_i \cdot 2^i$$

7.4 IEEE 754 Standard

Floating point types are also stored in a binary format, but they follow a different format that the integer binary representation shown above. They follow the IEEE 754 Standard for both representation and arithmetic operations. There are a lot of special number representations in the IEEE 754 standard, such as the representation for $\pm\infty$, NaN, zero, and subnormal[13] numbers. For this document, we will only focus on *finite normal* number representations, with an emphasis on `double` types, as is what we're mostly concerned with in competitive programming.

7.4.1 Representation

A `double` data type has 64 bits. The first bit is the sign bit, the next 11 bits is exponent, and the last 52 bits is called the **significand** or the **mantissa**. If we denote s as the sign bit value, c as the significand value, and e as the exponent value, then the represented number of the double is:

$$x = (-1)^s \cdot c \cdot 2^{e-1024}$$

To compute for c, let's denote the bits of the significand to be c_{51} to c_0 (from the most to the least significant bit). Then c can be calculated using the following equation:

$$c = 1 + \sum_{i=0}^{51} c_i \cdot 2^{i-51}$$

For example, if we are to represent 1.375, then $c_1 = c_2 = 1$, and the rest are 0. Note that the significand value assumes an implicit 1 to be added.

Another way to look at it is to consider it as an integer, and taking note that it should be shifted to the right 52 times. This way, our equation will look like this:

[13] Numbers with very very small absolute values. For `double` types, subnormal numbers have absolute values less than 2^{-1022}.

$$c = \frac{2^{52} + \sum_{i=0}^{51} c_i \cdot 2^i}{2^{52}}$$

An exception to the rules above is 0, which is represented with all-zeros in both the significand and the exponent.

7.4.2 Arithmetic Operations

When multiplying and dividing floating point types, the process is very similar to how the operations are done with integers.

To multiply floating point numbers, we first multiply (XOR) the sign bits. Then, we simply add the exponents to get the product's exponent. Next, we multiply the significand values (represented using integers to-be-shifted, as shown above), and then shift it to the right 52 times. If the result of this multiplication has 54 significant bits, then we shift to the right once, incrementing the product's exponent by one. Finally, the product's significand is the result's bits, omitting the most significant bit.

Dividing floating point numbers is similar to multiplying them, except that instead of adding the exponents, we subtract, and instead of shifting the result to the right (and incrementing the exponent), we shift to the left until it has 53 significant bits, decrementing the exponent for each step.

When adding, the *signed* significand values are first taken (incorporating the sign bit), then the one with the lower exponent is shifted to the right, as many times as the difference between exponents. The larger exponent is taken as the sum's exponent. The two values are then added, and if the resulting value has 54 bits, it is shifted to the right once, and the sum's exponent is incremented. The sum's sign bit is taken from the sign bit of the result, and the sum's significand is the 52 least significant bits of the result (again, omitting 1 bit). Subtraction is simply addition after negating the second number.

7.5 Implications and Limitations

With all that information, let's look at some implications that we can observe.

7.5.1 Precision

If we are to use a `double` data type, we are only limited to 53 significand bits. This accounts to around 15 decimal digits after the leading zeros. If we do a square root operation on our number, the precise decimal digits is halved to around 7. If we do a cube root operation, it's reduced to around just 5. Usually, problems have special checkers that accept values that differ from the hidden IO's answer to a certain threshold, or some ask for only the first k digits after the decimal point, and so

such problems are solvable given enough care. However, if you feel that precision is what's keeping your solution from getting accepted, you can opt to use the 128-bit `long double` data type, which has a 113-bit significand, amounting to around 33 decimal digits after the leading zeros.

7.5.2 Negative Zeros

A negative 0 is represented by a significand with value 0, but having the sign bit on. This can happen in several occasions, such as when you divide a small negative number with a very large number.

When compared to the normal positive zero, they are considered equal. In fact, in almost all operations, the sign doesn't make a difference. However, if we are to print this value, the negative sign is visible, which can lead to a Presentation Error (or most likely just a WA).[14] Such a case is shown below.

```
#include <iostream>
using namespace std;

int main() {
    printf("%.10lf\n",-1/1e10);
    cout << -0.00 << endl;
}
```

Usually, problems expecting floating point outputs use custom checkers that should be able to catch these types of errors, and consider them correct. However, if you think that this can be a possible source of error, you can check if the absolute value of a number is less than a certain threshold, and if so we can consider them 0. For example, if we are asked for the first 5 digits after the decimal point, then if our number (possibly after adding a very small ε value) is greater than -5×10^{-6}, but less than 0, then we can replace it with a 0 instead.

7.5.3 Multiplication and Division

With multiplication and division of floating point types, the process is very similar to multiplying/-dividing integers. And so, we only lose as much precision as when we multiply/divide integers, which is good news! However, we still have to be careful, as the limitations of multiplying/dividing integers also apply. For one, if we have the choice between multiplying or dividing first, then we should multiply first before dividing. There are a few exceptions to this rule, such as when multiplying exceeds the limit of finite normal numbers, but this rarely occurs.

[14] This can also happen when we only ask for the first few digits after the decimal point, and we have a very small negative number.

7.5.4 Addition and Underflow

When adding and subtracting floating point types, we first have to align the exponents by shifting one of them to the right. If the exponent difference is large enough, the significand value of the lower exponent would have been shifted out until it loses lots of its information, effectively having little to no effect due to what is called an underflow. A trivial example of how an underflow can occur is shown below.

```
#include <iostream>
using namespace std;

int main() {
    double x = 1e20;

    for(int i=0; i<1000000; i++)
        x += 1.00;

    cout << fixed << x << endl;
}
```

Though the example above seems silly and simply useless, another example of when underflow can occur is when computing the probability of events. Let's take a look at the equation below:

$$A = \sum_{a_0}^{N_0} \sum_{a_1}^{N_1} \cdots \sum_{a_k}^{N_k} \left[\prod_{i=0}^{k} P_i(a_i) \right]$$

This equation may seem abstract and seems to serve no purpose, but if we say that P_i is a discrete probability distribution, then the equation above looks like the sum of the probabilities of all possible outcomes of k (seemingly) independent events.

And so how can we try to compute this? One way is to use recursive backtracking, as shown in the following sample code.

```
#include <iostream>
using namespace std;

#define MAX_K 100
#define MAX_N 100

double N[MAX_K+1];
double P[MAX_K+1][MAX_N+1];
int K;
double A;

void f(int k, double curr) {
    if(k>K) {
        A += curr;
        return;
```

```
    }

    for(int i=0; i<=N[k]; i++)
        f(k+1,curr*P[k][i]);
}

int main() {
    //initialize K, N, and P

    A = 0;
    f(0,1);

    cout << A << endl;
}
```

While the above code seems to be correct, it's prone to underflow – since P is a probability distribution, then we'll be multiplying lots of numbers less than one, creating very very small numbers. This means we'll be adding lots of these very small numbers, similar to our example earlier.

A better approach to such scenarios is to use a "grouping" approach in dealing with these types of problems, as shown below.

```
#include <iostream>
using namespace std;

#define MAX_K 100
#define MAX_N 100

double N[MAX_K+1];
double P[MAX_K+1][MAX_N+1];
int K;
double A;

double f(int k, double curr) {
    if(k>K) {
        return curr;
    }

    double sum = 0;
    for(int i=0; i<=N[k]; i++)
        sum += f(k+1,curr*P[k][i]);
    return sum;
}

int main() {
    //initialize K, N, and P

    A = f(0,1);
    cout << A << endl;
}
```

In theory they should work the same way, but for the case above, we're adding them together by

grouping, and so the difference between addends should be significantly smaller. The lesson here is that when you're dealing with the sum of really small numbers, grouping them together (perhaps recursively, as was shown) can avoid underflow. This concept can be applied to Divide-and-Conquer and Dynamic Programming solutions that deal with small floating point numbers.

7.5.5 Subtraction and Catastrophic Cancellation

Due to our limited amount of significand bits, we can only approximate numbers to a certain degree. If we are careful, we still have a lot of significant bits to work with. However, when we subtract two very similar numbers, there is bound to be a significant loss of information. For example, if I were to perform the subtraction between 0.123456789123 and 0.123456789120, then the result will be 0.000000000003 – only one digit left, or around 2 significant bits! This phenomenon is what's called a Catastrophic Cancellation.

There is no one-size-fits-all solution to avoid this phenomenon, but a general rule is to try to rewrite your solution in order to avoid subtracting very similar numbers, or use standard libraries that does so for you. For example, if we are to try to implement $e^x - 1$ naively, then we might experience Catastrophic Cancellation when x is very small. A better solution is to use C's standard `expm1` function that computes exactly that, but avoids loss of too much information.

Another example is shown in this post. Here, if we are trying to implement the equation

$$\frac{1 - \sqrt{1 - t^2}}{t}$$

, then it is better to implement the equivalent yet stable equation

$$\frac{t}{1 + \sqrt{1 - t^2}}$$

Another, and very recent, example of where it can occur is demonstrated in Kevin Atienza's submissions for the recent Code Forces 975E. Here, we have two submissions, both theoretically correct: Solution A, and Solution B.

The only difference between the two is their preprocessing inside the main function. Though similar, Solution B managed to avoid Catastrophic Cancellation and got AC, while Solution A suffered precision errors and got WA.

7.5.6 Comparison

Not all rational numbers can be represented by powers of two. Take $\frac{1}{3}$ for instance. We can't represent $\frac{1}{3}$ as the exact sum of powers of two, but we can approximate. For example, $\frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \frac{1}{256} =$

0.33203125. If we add more powers of two, then we can approximate more precisely. In other words, although we can't represent it exactly, we can approximate it with a precision depending on the number of significand bits we have.

What this also implies is that two variables having the same theoretical value may not be equal in the strictest sense if they were computed using different means. Since equality (==) checks whether all bits are equal[15], then these two variables will be reported as not equal, even if they approximate the same number.

To alleviate this, we can modify our equality check to allow some leeway. We do this by checking whether the difference is less than a set ε value, and if so we consider them equal. This has a couple of consequences, such as modifying the inequality operators. An example of some of these modified comparison functions is shown below.

```
#include <cmath>
using namespace std;

#define EPS 1e-9

bool equal(double a, double b) {
    return abs(a-b)<EPS;
}

bool less(double a, double b) {
    return !equal(a,b) && a<b;
}

bool less_or_equal(double a, double b) {
    return a<b+EPS;
}
```

The choice of ε value depends on the problem and perhaps personal experience. A choice of $\varepsilon = 10^{-9}$ is usually a good rule of thumb in most problems. If a program computes roots a lot, then one could argue that epsilon should be a little larger, say 10^{-6}. There is no right answer to people's choice of ε, and so in some problems that require strict precision, some people tend to experiment on the "right" value for ε as a last resort.

[15] With the exception of zero and negative zero, and possibly some other special numbers