Programming Assignment 3: Hash Tables and Hash Functions

Revision: March 29, 2020

Introduction

In this programming assignment, you will practice implementing hash functions and hash tables and using them to solve algorithmic problems. In some cases you will just implement an algorithm from the lectures, while in others you will need to invent an algorithm to solve the given problem using hashing.

Learning Outcomes

Upon completing this programming assignment you will be able to:

1. Apply hashing to solve the given algorithmic problems.
2. Implement a simple phone book manager.
3. Implement a hash table using the chaining scheme.
4. Find all occurrences of a pattern in text using Rabin–Karp's algorithm.
5. Applying hashing to solve other string processing problems.

Passing Criteria: 2 out of 6

Passing this programming assignment requires passing at least 2 out of 6 programming challenges from this assignment. In turn, passing a programming challenge requires implementing a solution that passes all the tests for this problem in the grader and does so under the time and memory limits specified in the problem statement.

Contents

1 Phone book

Problem Introduction

In this problem you will implement a simple phone book manager.

Problem Description

Task. In this task your goal is to implement a simple phone book manager. It should be able to process the following types of user's queries:

- **add** `number name`. It means that the user adds a person with name `name` and phone number `number` to the phone book. If there exists a user with such number already, then your manager has to overwrite the corresponding name.
- **del** `number`. It means that the manager should erase a person with number `number` from the phone book. If there is no such person, then it should just ignore the query.
- **find** `number`. It means that the user looks for a person with phone number `number`. The manager should reply with the appropriate name, or with string "not found" (without quotes) if there is no such person in the book.

Input Format. There is a single integer N in the first line — the number of queries. It's followed by N lines, each of them contains one query in the format described above.

Constraints. $1 \le N \le 10^5$. All phone numbers consist of decimal digits, they don't have leading zeros, and each of them has no more than 7 digits. All names are non-empty strings of latin letters, and each of them has length at most 15. It's guaranteed that there is no person with name "not found".

Output Format. Print the result of each **find** query — the name corresponding to the phone number or "not found" (without quotes) if there is no person in the phone book with such phone number. Output one result per line in the same order as the **find** queries are given in the input.

Time Limits. `C`: 3 sec, `C++`: 3 sec, `Java`: 6 sec, `Python`: 6 sec. `C#`: 4.5 sec, `Haskell`: 6 sec, `JavaScript`: 9 sec, `Ruby`: 9 sec, `Scala`: 9 sec.

Memory Limit. 512MB.

Sample 1.

Input:

```
12
add 911 police
add 76213 Mom
add 17239 Bob
find 76213
find 910
find 911
del 910
del 911
find 911
find 76213
add 76213 daddy
find 76213
```

Output:

```
Mom
not found
police
not found
Mom
daddy
```

76213 is Mom's number, 910 is not a number in the phone book, 911 is the number of police, but then it was deleted from the phone book, so the second search for 911 returned "not found". Also, note that when the daddy was added with the same phone number 76213 as Mom's phone number, the contact's name was rewritten, and now search for 76213 returns "daddy" instead of "Mom".

Sample 2.

Input:

```
8
find 3839442
add 123456 me
add 0 granny
find 0
find 123456
del 0
del 0
find 0
```

Output:

```
not found
granny
me
not found
```

Recall that deleting a number that doesn't exist in the phone book doesn't change anything.

Starter Files

The starter solutions for C++, Java and Python3 in this problem read the input, implement a naive algorithm to look up names by phone numbers and write the output. You need to use a fast data structure to implement

a better algorithm. If you use other languages, you need to implement the solution from scratch.

What to Do

Use the direct addressing scheme.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

2 Hashing with chains

Problem Introduction

In this problem you will implement a hash table using the chaining scheme. Chaining is one of the most popular ways of implementing hash tables in practice. The hash table you will implement can be used to implement a phone book on your phone or to store the password table of your computer or web service (but don't forget to store hashes of passwords instead of the passwords themselves, or you will get hacked!).

Problem Description

Task. In this task your goal is to implement a hash table with lists chaining. You are already given the number of buckets m and the hash function. It is a polynomial hash function

$$h(S) = \left(\sum_{i=0}^{|S|-1} S[i]x^i \bmod p \right) \bmod m \,,$$

where $S[i]$ is the ASCII code of the i-th symbol of S, $p = 1\ 000\ 000\ 007$ and $x = 263$. Your program should support the following kinds of queries:

- **add** `string` — insert `string` into the table. If there is already such string in the hash table, then just ignore the query.
- **del** `string` — remove `string` from the table. If there is no such string in the hash table, then just ignore the query.
- **find** `string` — output "yes" or "no" (without quotes) depending on whether the table contains `string` or not.
- **check** i — output the content of the i-th list in the table. Use spaces to separate the elements of the list. **If i-th list is empty, output a blank line.**

When inserting a new string into a hash chain, you must insert it in the beginning of the chain.

Input Format. There is a single integer m in the first line — the number of buckets you should have. The next line contains the number of queries N. It's followed by N lines, each of them contains one query in the format described above.

Constraints. $1 \le N \le 10^5$; $\frac{N}{5} \le m \le N$. All the strings consist of latin letters. Each of them is non-empty and has length at most 15.

Output Format. Print the result of each of the **find** and **check** queries, one result per line, in the same order as these queries are given in the input.

Time Limits. `C`: 1 sec, `C++`: 1 sec, `Java`: 5 sec, `Python`: 7 sec. `C#`: 1.5 sec, `Haskell`: 2 sec, `JavaScript`: 7 sec, `Ruby`: 7 sec, `Scala`: 7 sec.

Memory Limit. 512MB.

Sample 1.

Input:

```
5
12
add world
add HellO
check 4
find World
find world
del world
check 4
del HellO
add luck
add GooD
check 2
del good
```

Output:

```
HellO world
no
yes
HellO
GooD luck
```

The ASCII code of 'w' is 119, for 'o' it is 111, for 'r' it is 114, for 'l' it is 108, and for 'd' it is 100. Thus, $h(\text{“world"}) = (119 + 111 \times 263 + 114 \times 263^2 + 108 \times 263^3 + 100 \times 263^4 \bmod 1\,000\,000\,007) \bmod 5 = 4$. It turns out that the hash value of *HellO* is also 4. Recall that we always insert in the beginning of the chain, so after adding “world" and then “HellO" in the same chain index 4, first goes “HellO" and then goes “world". Of course, “World" is not found, and “world" is found, because the strings are case-sensitive, and the codes of 'W' and 'w' are different. After deleting “world", only “HellO" is found in the chain 4. Similarly to “world" and “HellO", after adding “luck" and “GooD" to the same chain 2, first goes “GooD" and then “luck".

Sample 2.

Input:

```
4
8
add test
add test
find test
del test
find test
find Test
add Test
find Test
```

Output:

```
yes
no
no
yes
```

Adding “test" twice is the same as adding “test" once, so first **find** returns “yes". After del, “test" is

no longer in the hash table. First time **find** doesn't find "Test" because it was not added before, and strings are case-sensitive in this problem. Second time "Test" can be found, because it has just been added.

Sample 3.

Input:

```
3
12
check 0
find help
add help
add del
add add
find add
find del
del del
find del
check 0
check 1
check 2
```

Output:

```
no
yes
yes
no

add help
```

Explanation:
Note that you need to output a blank line when you handle an empty chain. Note that the strings stored in the hash table can coincide with the commands used to work with the hash table.

Starter Files

There are starter solutions only for C++, Java and Python3, and if you use other languages, you need to implement solution from scratch. Starter solutions read the input, do a full scan of the whole table to simulate each **find** operation and write the output. This naive simulation algorithm is too slow, so you need to implement the real hash table.

What to Do

Follow the explanations about the chaining scheme from the lectures. Remember to always insert new strings in the beginning of the chain. Remember to output a blank line when **check** operation is called on an empty chain.

Some hints based on the problems encountered by learners:

- Beware of integer overflow. Use `long long` type in C++ and `long` type in Java where appropriate. Take everything $\pmod p$ as soon as possible while computing something $\pmod p$, so that the numbers are always between 0 and $p - 1$.

- Beware of taking negative numbers $\pmod{p}$. In many programming languages, $(-2)\%5 \neq 3\%5$. Thus you can compute the same hash values for two strings, but when you compare them, they appear to be different. To avoid this issue, you can use such construct in the code: $x \leftarrow ((a\%p) + p)\%p$ instead of just $x \leftarrow a\%p$.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

3 Find pattern in text

Problem Introduction

In this problem, your goal is to implement the Rabin–Karp's algorithm.

Problem Description

Task. In this problem your goal is to implement the Rabin–Karp's algorithm for searching the given pattern in the given text.

Input Format. There are two strings in the input: the pattern P and the text T.

Constraints. $1 \leq |P| \leq |T| \leq 5 \cdot 10^5$. The total length of all occurrences of P in T doesn't exceed 10^8. The pattern and the text contain only latin letters.

Output Format. Print all the positions of the occurrences of P in T in the ascending order. Use 0-based indexing of positions in the the text T.

Time Limits. `C`: 1 sec, `C++`: 1 sec, `Java`: 5 sec, `Python`: 5 sec. `C#`: 1.5 sec, `Haskell`: 2 sec, `JavaScript`: 3 sec, `Ruby`: 3 sec, `Scala`: 3 sec.

Memory Limit. 512MB.

Sample 1.

Input:

```
aba
abacaba
```

Output:

```
0 4
```

Explanation:
The pattern *aba* can be found in positions 0 (**aba**caba) and 4 (abac**aba**) of the text *abacaba*.

Sample 2.

Input:

```
Test
testTesttesT
```

Output:

```
4
```

Explanation:
Pattern and text are case-sensitive in this problem. Pattern $Test$ can only be found in position 4 in the text $testTesttesT$.

Sample 3.

Input:

```
aaaaa
baaaaaaa
```

Output:

```
1 2 3
```

Note that the occurrences of the pattern in the text can be overlapping, and that's ok, you still need to output all of them.

Starter Files

The starter solutions in C++, Java and Python3 read the input, apply the naive $O(|T||P|)$ algorithm to this problem and write the output. You need to implement the Rabin–Karp's algorithm instead of the naive algorithm and thus significantly speed up the solution. If you use other languages, you need to implement a solution from scratch.

What to Do

Implement the fast version of the Rabin–Karp's algorithm from the lectures.

Some hints based on the problems encountered by learners:

- Beware of integer overflow. Use `long long` type in C++ and `long` type in Java where appropriate. Take everything $\pmod p$ as soon as possible while computing something $\pmod p$, so that the numbers are always between 0 and $p - 1$.
- Beware of taking negative numbers $\pmod p$. In many programming languages, $(-2)\%5 \neq 3\%5$. Thus you can compute the same hash values for two strings, but when you compare them, they appear to be different. To avoid this issue, you can use such construct in the code: $x \leftarrow ((a\%p) + p)\%p$ instead of just $x \leftarrow a\%p$.
- Use operator `==` in Python instead of implementing your own function `AreEqual` for strings, because built-in operator `==` will work much faster.
- In C++, method `substr` of `string` creates a new string, uses additional memory and time for that, so use it carefully and avoid creating lots of new strings. When you need to compare pattern with a substring of text, do it without calling `substr`.
- In Java, however, method `substring` does NOT create a new `String`. Avoid using `new String` where it is not needed, just use `substring`.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

4 Substring equality

Problem Introduction

In this problem, you will use hashing to design an algorithm that is able to preprocess a given string s to answer any query of the form "are these two substrings of s equal?" efficiently. This, in turn, is a basic building block in many string processing algorithms.

Problem Description

Input Format. The first line contains a string s consisting of small Latin letters. The second line contains the number of queries q. Each of the next q lines specifies a query by three integers a, b, and l.

Constraints. $1 \le |s| \le 500\,000$. $1 \le q \le 100\,000$. $0 \le a, b \le |s| - l$ (hence the indices a and b are 0-based).

Output Format. For each query, output "Yes" if $s_a s_{a+1} \dots s_{a+l-1} = s_b s_{b+1} \dots s_{b+l-1}$ are equal, and "No" otherwise.

Time Limits. `C`: 1 sec, `C++`: 1 sec, `Java`: 2 sec, `Python`: 10 sec. `C#`: 1.5 sec, `Haskell`: 2 sec, `JavaScript`: 5 sec, `Ruby`: 5 sec, `Scala`: 5 sec.

Memory Limit. 512MB.

Sample 1.

Input:

```
trololo
4
0 0 7
2 4 3
3 5 1
1 3 2
```

Output:

```
Yes
Yes
Yes
No
```

0 0 7 → `trololo` = `trololo`

2 4 3 → `trololo` = `trololo`

3 5 1 → `trololo` = `trololo`

1 3 2 → `trololo` ≠ `trololo`

What to Do

For a string $t = t_0 t_1 \cdots t_{m-1}$ of length m and an integer x, define a polynomial hash function

$$H(t) = \sum_{j=0}^{m-1} t_j x^{m-j-1} = t_0 x^{m-1} + t_1 x^{m-2} + \cdots + t_{m-2} x + t_{m-1} .$$

Let $s_a s_{a+1} \cdots s_{a+l-1}$ be a substring of the given string $s = s_0 s_1 \cdots s_{n-1}$. A nice property of the polynomial hash function H is that $H(s_a s_{a+1} \cdots s_{a+l-1})$ can be expressed through $H(s_0 s_1 \cdots s_{a+l-1})$ and

$H(s_0 s_1 \cdots s_{a-1})$, i.e., through hash values of two prefixes of s:

$$\begin{aligned} H(s_a s_{a+1} \cdots s_{a+l-1}) &= s_a x^{l-1} + s_{a+1} x^{l-2} + \cdots + s_{a+l-1} = \\ &= s_0 x^{a+l-1} + s_1 x^{a+l-2} + \cdots + s_{a+l-1} - \\ &- x^l (s_0 x^{a-1} + s_1 x^{a-2} + \cdots + s_{a-1}) = \\ &= H(s_0 s_1 \cdots s_{a+l-1}) - x^l H(s_0 s_1 \cdots s_{a-1}) \end{aligned}$$

This leads us to the following natural idea: we precompute and store the hash values of all prefixes of s: let $h[0] = 0$ and, for $1 \le i \le n$, let $h[i] = H(s_0 s_1 \cdots s_{i-1})$. Then, the identity above becomes

$$H(s_a s_{a+1} \cdots s_{a+l-1}) = h[a+l] - x^l h[a]\,.$$

In other words, we are able to get the hash value of any substring of s in just constant time! Clearly, if $H(s_a s_{a+1} \cdots s_{a+l-1}) \neq H(s_b s_{b+1} \cdots s_{b+l-1})$, then the corresponding two substrings ($s_a s_{a+1} \cdots s_{a+l-1}$ and $s_b s_{b+1} \cdots s_{b+l-1}$) are different. However, if the hash values are the same, it is still possible that the substrings are different — this is called a *collision*. Below, we discuss how to reduce the probability of a collision.

Recall that in practice one never computes the exact value of a polynomial hash function: everything is computed modulo m for some fixed integer m. This is done to ensure that all the computations are efficient and that the hash values are small enough. Recall also that when computing $H(s) \bmod m$ it is important to take every intermediate step (rather then the final result) modulo m.

It can be shown that if s_1 and s_2 are two *different* strings of length n and m is a prime integer, then the probability that $H(s_1) \bmod m = H(s_2) \bmod m$ (over the choices of $0 \le x \le m-1$) is at most $\frac{n}{m}$ (roughly, this is because $H(s_1) - H(s_2)$ is a non-zero polynomial of degree at most $n-1$ and hence can have at most n roots modulo m). To further reduce the probability of a collision, one may take two different modulus.

Overall, this gives the following approach.

1. Fix $m_1 = 10^9 + 7$ and $m_2 = 10^9 + 9$.
2. Select a random x from 1 to 10^9.
3. Compute arrays $h_1[0..n]$ and $h_2[0..n]$: $h_1[0] = h_2[0] = 0$ and, for $1 \le i \le n$, $h_1[i] = H(s_0 \cdots s_{i-1}) \bmod m_1$ and $h_2[i] = H(s_0 \cdots s_{i-1}) \bmod m_2$. We illustrate this for h_1 below.

   ```
   allocate h1[0..n]
   h1[0] ← 0
   for i from 1 to n:
     h1[i] ← (x · h1[i − 1] + si) mod m1
   ```

4. For every query (a, b, l):
   (a) Use precomputed hash values, to compute the hash values of the substrings $s_a s_{a+1} \cdots s_{a+l-1}$ and $s_b s_{b+1} \cdots s_{b+l-1}$ modulo m_1 and m_2.
   (b) Output "Yes", if

   $$\begin{aligned} &H(s_a s_{a+1} \cdots s_{a+l-1}) \bmod m_1 = H(s_b s_{b+1} \cdots s_{b+l-1}) \bmod m_1 \text{ and} \\ &H(s_a s_{a+1} \cdots s_{a+l-1}) \bmod m_2 = H(s_b s_{b+1} \cdots s_{b+l-1}) \bmod m_2\,. \end{aligned}$$

   (c) Otherwise, output "No".

Note that, in contrast to Karp–Rabin algorithm, we do not compare the substrings naively when their hashes coincide. The probability of this event is at most $\frac{n}{m_1} \cdot \frac{n}{m_2} \le 10^{-9}$. (In fact, for random strings the probability is even much smaller: 10^{-18}. In this problem, the strings are not random, but the probability of collision is still very small.)

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

5 Longest common substring

Problem Introduction

In the longest common substring problem one is given two strings s and t and the goal is to find a string w of maximal length that is a substring of both s and t. This is a natural measure of similarity between two strings. The problem has applications in text comparison and compression as well as in bioinformatics.

The problem can be seen as a special case of the edit distance problem (where only insertions and deletions are allowed). Hence, it can be solved in time $O(|s| \cdot |t|)$ using dynamic programming. Later in this specialization, we will learn highly non-trivial data structures for solving this problem in linear time $O(|s| + |t|)$. In this problem, your goal is to use hashing to solve it in almost linear time.

Problem Description

Input Format. Every line of the input contains two strings s and t consisting of lower case Latin letters.

Constraints. The total length of all s's as well as the total length of all t's does not exceed 100 000.

Output Format. For each pair of strings s and t_i, find its longest common substring and specify it by outputting three integers: its starting position in s, its starting position in t (both 0-based), and its length. More formally, output integers $0 \le i < |s|$, $0 \le j < |t|$, and $l \ge 0$ such that $s_i s_{i+1} \cdots s_{i+l-1} = t_j t_{j+1} \cdots t_{j+l-1}$ and l is maximal. (As usual, if there are many such triples with maximal l, output any of them.)

Time Limits. `C`: 2 sec, `C++`: 2 sec, `Java`: 5 sec, `Python`: 15 sec. `C#`: 3 sec, `Haskell`: 4 sec, `JavaScript`: 10 sec, `Ruby`: 10 sec, `Scala`: 10 sec.

Memory Limit. 512MB.

Sample 1.

Input:

```
cool toolbox
aaa bb
aabaa babbaab
```

Output:

```
1 1 3
0 1 0
0 4 3
```

Explanation:
The longest common substring of the first pair of strings is `ool`, it starts at the first position in `toolbox` and at the first position in `cool`. The strings from the second line do not share any non-empty common substrings (in this case, $l = 0$ and one may output any indices i and j). Finally, the last two strings share a substring `aab` that has length 3 and starts at position 0 in the first string and at position 4 in the second one. Note that for this pair of string one may output `2 3 3` as well.

What to Do

For every pair of strings s and t, use binary search to find the length of the longest common substring. To check whether two strings have a common substring of length k,

- precompute hash values of all substrings of length k of s and t;
- make sure to use a few hash functions (but not just one) to reduce the probability of a collision;

- store hash values of all substrings of length k of the string s in a hash table; then, go through all substrings of length k of the string t and check whether the hash value of this substring is present in the hash table.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

6 Pattern matching with mismatches

Problem Introduction

A natural generalization of the pattern matching problem is the following: find all text locations where distance from pattern is sufficiently small. This problems has applications in text searching (where mismatches correspond to typos) and bioinformatics (where mismatches correspond to mutations).

Problem Description

Task. For an integer parameter k and two strings $t = t_0t_1 \cdots t_{m-1}$ and $p = p_0p_1 \cdots p_{n-1}$, we say that p occurs in t at position i with at most k mismatches if the strings p and $t[i : i + p) = t_it_{i+1} \cdots t_{i+n-1}$ differ in at most k positions.

Input Format. Every line of the input contains an integer k and two strings t and p consisting of lower case Latin letters.

Constraints. $0 \le k \le 5$, $1 \le |t| \le 200\,000$, $1 \le |p| \le \min\{|t|, 100\,000\}$. The total length of all t's does not exceed 200 000, the total length of all p's does not exceed 100 000.

Output Format. For each triple (k, t, p), find all positions $0 \le i_1 < i_2 < \cdots < i_l < |t|$ where p occurs in t with at most k mismatches. Output l and $i_1, i_2, \ldots, i_l$.

Time Limits. `C`: 2 sec, `C++`: 2 sec, `Java`: 5 sec, `Python`: 40 sec. `C#`: 3 sec, `Haskell`: 4 sec, `JavaScript`: 10 sec, `Ruby`: 10 sec, `Scala`: 10 sec.

Memory Limit. 512MB.

Sample 1.

Input:

```
0 ababab baaa
1 ababab baaa
1 xabcabc ccc
2 xabcabc ccc
3 aaa xxx
```

Output:

```
0
1 1
0
4 1 2 3 4
1 0
```

Explanation:
For the first triple, there are no exact matches. For the second triple, `baaa` has distance one from the pattern. For the third triple, there are no occurrences with at most one mismatch. For the fourth triple, any (length three) substring of p containing at least one `c` has distance at most two from t. For the fifth triple, t and p differ in three positions.

What to Do

Start by computing hash values of prefixes of t and p and their partial sums. This allows you to compare any two substrings of t and p in expected constant time. For each candidate position i, perform k steps of the form "find the next mismatch". Each such mismatch can be found using binary search. Overall, this gives an algorithm running in time $O(nk \log n)$.

Need Help?

Ask a question or see the questions asked by other learners at this forum thread.

7 Appendix

7.1 Compiler Flags

C (`gcc 7.4.0`). File extensions: `.c`. Flags:

```
gcc -pipe -O2 -std=c11 <filename> -lm
```

C++ (`g++ 7.4.0`). File extensions: `.cc`, `.cpp`. Flags:

```
g++ -pipe -O2 -std=c++14 <filename> -lm
```

If your `C/C++` compiler does not recognize `-std=c++14` flag, try replacing it with `-std=c++0x` flag or compiling without this flag at all (all starter solutions can be compiled without it). On Linux and MacOS, you most probably have the required compiler. On Windows, you may use your favorite compiler or install, e.g., `cygwin`.

C# (`mono 4.6.2`). File extensions: `.cs`. Flags:

```
mcs
```

Go (`golang 1.13.4`). File extensions: `.go`. Flags

```
go
```

Haskell (`ghc 8.0.2`). File extensions: `.hs`. Flags:

```
ghc -O2
```

Java (`OpenJDK 1.8.0_232`). File extensions: `.java`. Flags:

```
javac -encoding UTF-8
java -Xmx1024m
```

JavaScript (`NodeJS 12.14.0`). File extensions: `.js`. No flags:

```
nodejs
```

Kotlin (`Kotlin 1.3.50`). File extensions: `.kt`. Flags:

```
kotlinc
java -Xmx1024m
```

Python (`CPython 3.6.9`). File extensions: `.py`. No flags:

```
python3
```

Ruby (`Ruby 2.5.1p57`). File extensions: `.rb`.

```
ruby
```

Rust (`Rust 1.37.0`). File extensions: `.rs`.

```
rustc
```

Scala (`Scala 2.12.10`). File extensions: `.scala`.

```
scalac
```

7.2 Frequently Asked Questions

Why My Submission Is Not Graded?

You need to create a submission and upload the *source file* (rather than the executable file) of your solution. Make sure that after uploading the file with your solution you press the blue "Submit" button at the bottom. After that, the grading starts, and the submission being graded is enclosed in an orange rectangle. After the testing is finished, the rectangle disappears, and the results of the testing of all problems are shown.

What Are the Possible Grading Outcomes?

There are only two outcomes: "pass" or "no pass." To pass, your program must return a correct answer on all the test cases we prepared for you, and do so under the time and memory constraints specified in the problem statement. If your solution passes, you get the corresponding feedback "Good job!" and get a point for the problem. Your solution fails if it either crashes, returns an incorrect answer, works for too long, or uses too much memory for some test case. The feedback will contain the index of the first test case on which your solution failed and the total number of test cases in the system. The tests for the problem are numbered from 1 to the total number of test cases for the problem, and the program is always tested on all the tests in the order from the first test to the test with the largest number.

Here are the possible outcomes:

- `Good job!` Hurrah! Your solution passed, and you get a point!
- `Wrong answer.` Your solution outputs incorrect answer for some test case. Check that you consider all the cases correctly, avoid integer overflow, output the required white spaces, output the floating point numbers with the required precision, don't output anything in addition to what you are asked to output in the output specification of the problem statement.
- `Time limit exceeded.` Your solution worked longer than the allowed time limit for some test case. Check again the running time of your implementation. Test your program locally on the test of maximum size specified in the problem statement and check how long it works. Check that your program doesn't wait for some input from the user which makes it to wait forever.
- `Memory limit exceeded.` Your solution used more than the allowed memory limit for some test case. Estimate the amount of memory that your program is going to use in the worst case and check that it does not exceed the memory limit. Check that your data structures fit into the memory limit. Check that you don't create large arrays or lists or vectors consisting of empty arrays or empty strings, since those in some cases still eat up memory. Test your program locally on the tests of maximum size specified in the problem statement and look at its memory consumption in the system.
- `Cannot check answer. Perhaps the output format is wrong.` This happens when you output something different than expected. For example, when you are required to output either "Yes" or "No", but instead output 1 or 0. Or your program has empty output. Or your program outputs not only the correct answer, but also some additional information (please follow the exact output format specified in the problem statement). Maybe your program doesn't output anything, because it crashes.
- `Unknown signal 6 (or 7, or 8, or 11, or some other).` This happens when your program crashes. It can be because of a division by zero, accessing memory outside of the array bounds, using uninitialized variables, overly deep recursion that triggers a stack overflow, sorting with a contradictory comparator, removing elements from an empty data structure, trying to allocate too much memory, and many other reasons. Look at your code and think about all those possibilities. Make sure that you use the same compiler and the same compiler flags as we do.
- `Internal error: exception...` Most probably, you submitted a compiled program instead of a source code.

- `Grading failed.` Something wrong happened with the system. Report this through Coursera or edX Help Center.

May I Post My Solution at the Forum?

Please do not post any solutions at the forum or anywhere on the web, even if a solution does not pass the tests (as in this case you are still revealing parts of a correct solution). Our students follow the Honor Code: "I will not make solutions to homework, quizzes, exams, projects, and other assignments available to anyone else (except to the extent an assignment explicitly permits sharing solutions)."

Do I Learn by Trying to Fix My Solution?

My implementation always fails in the grader, though I already tested and stress tested it a lot. Would not it be better if you gave me a solution to this problem or at least the test cases that you use? I will then be able to fix my code and will learn how to avoid making mistakes. Otherwise, I do not feel that I learn anything from solving this problem. I am just stuck.

First of all, learning from your mistakes is one of the best ways to learn.

The process of trying to invent new test cases that might fail your program is difficult but is often enlightening. Thinking about properties of your program makes you understand what happens inside your program and in the general algorithm you're studying much more.

Also, it is important to be able to find a bug in your implementation without knowing a test case and without having a reference solution, just like in real life. Assume that you designed an application and an annoyed user reports that it crashed. Most probably, the user will not tell you the exact sequence of operations that led to a crash. Moreover, there will be no reference application. Hence, it is important to learn how to find a bug in your implementation yourself, without a magic oracle giving you either a test case that your program fails or a reference solution. We encourage you to use programming assignments in this class as a way of practicing this important skill.

If you have already tested your program on all corner cases you can imagine, constructed a set of manual test cases, applied stress testing, etc, but your program still fails, try to ask for help on the forum. We encourage you to do this by first explaining what kind of corner cases you have already considered (it may happen that by writing such a post you will realize that you missed some corner cases!), and only afterwards asking other learners to give you more ideas for tests cases.