int overflow, array bounds special cases (n=1?) do smth instead of nothing and stay organized WRITE STUFF DOWN DON'T GET STUCK ON ONE APPROACH

Memory usage code:

#include <unistd.h>

#include <ios>

#include <iostream>

#include <fstream>

#include <string>

using namespace std;

void mem\_usage(double& vm\_usage, double& resident\_set) {

   vm\_usage = 0.0;

   resident\_set = 0.0;

   ifstream stat\_stream("/proc/self/stat",ios\_base::in); //get info from proc

   directory

   //create some variables to get info

   string pid, comm, state, ppid, pgrp, session, tty\_nr;

   string tpgid, flags, minflt, cminflt, majflt, cmajflt;

   string utime, stime, cutime, cstime, priority, nice;

   string O, itrealvalue, starttime;

   unsigned long vsize;

   long rss;

   stat\_stream >> pid >> comm >> state >> ppid >> pgrp >> session >> tty\_nr

   >> tpgid >> flags >> minflt >> cminflt >> majflt >> cmajflt

   >> utime >> stime >> cutime >> cstime >> priority >> nice

   >> O >> itrealvalue >> starttime >> vsize >> rss; // don't care

   about the rest

   stat\_stream.close();

   long page\_size\_kb = sysconf(\_SC\_PAGE\_SIZE) / 1024; // for x86-64 is configured

   to use 2MB pages

   vm\_usage = vsize / 1024.0;

   resident\_set = rss \* page\_size\_kb;

}

int main() {

   double vm, rss;

   mem\_usage(vm, rss);

   cout << "Virtual Memory: " << vm << "\nResident set size: " << rss << endl;

}

[**http://www.cs.ecu.edu/karl/3300/spr16/Notes/C/Array/heap.html**](http://www.cs.ecu.edu/karl/3300/spr16/Notes/C/Array/heap.html)

[**http://jsteemann.github.io/blog/2016/06/14/how-much-memory-does-an-stl-container-use/**](http://jsteemann.github.io/blog/2016/06/14/how-much-memory-does-an-stl-container-use/)

[**https://lemire.me/blog/2016/09/15/the-memory-usage-of-stl-containers-can-be-surprising/**](https://lemire.me/blog/2016/09/15/the-memory-usage-of-stl-containers-can-be-surprising/)

**11**

A std::vector<element> typically takes 3 machine words ***in total*** + sizeof(element) \* capacity() of memory. For typical implementations, the overhead consist of pointers to the beginning, end and current size of the vector. The elements themselves are stored in contiguous memory. capacity() typically has room for up to twice the actual number of elements.

A std::map<element, int> typically takes about 2 machine words **in total** + 3 machine words ***per element*** + [ sizeof(element) +sizeof(int) ] \* num\_elements of memory. For typical implementations, the overhead consists of pointers to the stored elements. The elements themselves are stored in a binary tree, with pointers to its parent and two children.

With these rules of thumb, all you need to know is the average number of characters per string and the total number of strings to know total memory consumption.

**Memory!**

|  |  |
| --- | --- |
| Storage cost in bytes per 32-bit entry | |
| STL container | Storage |
| std::vector | 4 |
| std::deque | 8 |
| std::list | 24 |
| std::set | 32 |
| std::unordered\_set | 36 |

**Beginner**: Linked List, Stack, Queue, Binary Search Tree.  
  
**Intermediate**: Heap, Priority Queue, Huffman Tree, Union Find, Tries, Hash Table, Tree Map.

**Proficient**: Segment Tree, Binary Indexed Tree, Suffix Array, Sparse Table, Lowest Common Ancestor, Range Tree.

**Expert**: Suffix Automaton, Suffix Tree, Heavy-Light Decomposition, Treap, Aho-Corasick, K-d tree, Link-Cut Tree, Splay Tree, Palindromic Tree, Rope, Dancing Links, Radix Tree, Dynamic Suffix Array.

I don't know that there is much to explain... You can do closer to 2 \* 10^8 operations per second in C++ if those operations are low-level operations (array accesses, additions, bitshifts, multiplies, subtractions, xors, et cetera). Mods and divisions are slightly slower. If you are doing things like hashmap lookups, reading input, or printing output, you can do about 10^6 of those per second. If you use Java, the cut-off is maybe 70% what it is in C++, since a lot of things have more overhead, but of course it really depends on what the operations are; sometimes it is closer to 30% of what C++ can do.

It completely depends on the type of operations that you are going to use. For example (%) module operation takes much more time than (+) sum operation. If your operations are simple (like **+** and **×**. But not **/** and **%**) C++ can do even more than 1e9 operations per second. The second thing that matters is your memory usage. If CPU can store all of the data in cache memory then your code will run way faster than the case that CPU should use RAM!

C++ Speed Optimizations

1. **/\* stuff you should look for**
2. **\* int overflow, array bounds**
3. **\* special cases (n=1?)**
4. **\* do smth instead of nothing and stay organized**
5. **\* WRITE STUFF DOWN**
6. **\* DON'T GET STUCK ON ONE APPROACH**
7. **\*/**

Use macros #define LIMIT 5

- corresponds with int LIMIT = 5;

- not sure what the difference is

// allows floats to be printed to standard output (stdout) up to your precision of choosing (up to the precision limit of the type allows, of course)

template <typename T>

std::string to\_string\_with\_precision(const T a\_value, const int n = 6)

{

std::ostringstream out;

out.precision(n);

out << std::fixed << a\_value;

return out.str();

}

#pragma GCC optimize("Ofast")

#pragma GCC target("avx,avx2,fma")

// helpful sidenote: C++ clock cycles per second is 1,000,000 !

There are at least three simple things that you can do with I/O in order to make your I/O experience in C++ faster. Be advised though, that whatever you do, using the [C](https://open.kattis.com/help/c) stdio routines (e.g. scanf and printf) is still likely to be faster .

One problem is that the cin and cout are synchronized with the C stdio functions. In order to disable this, use ios::sync\_with\_stdio(false);.

Another problem is that cin and cout are tied, which means that as soon as you read from cin, cout is flushed. While this may be desirable when running a program interactively, it slows things down, especially if there are many alternating reads and writes. In order to disable this, use the command cin.tie(NULL);. Also, be aware that sync\_with\_stdio() resets this setting, so make sure you do things in the right order!

Third, it is good to know that the cout << endl also flushes cout. As far as we are aware of, this cannot be disabled, so if you are anticipating a lot of output, you should probably refrain from using endl, and use "\n" instead.

1. Use prefix(++i) instead of postfix(i++) increment/decrement, as postfix uses a temporary variable to save the initial state of the variable.  
   2. Use short hand assignment operators (+=, -=), they don’t create a temporary variable.  
   3. Use bit-wise operators where ever it is possible. Operations on bits is always more efficient than directly on decimal. Using x >> 1 inste
2. cin.tie(NULL);
3. cout.tie(NULL);
4. This ensures that both the Input and Output streams namely cin and cout are untied which has a side effect of significant decrement in execution time if the code deals with extensively heavy I/O streams and large quantity of input data. The decrement ranges upto the order of mili seconds or can even reach the order of seconds.
5. The second thing which you can use is printf and scanf commands instead of cout and cin which is also considered as a time friendly practice in C++. It has been proved that printf and scanf outperform their counterparts on experimental basis, not significant though…
6. Now if you need extreme speeds of inputs then you can write your own code for taking inputs which primarily involves usage of gechar\_unlocked().. You may Google for more details abt this method.. This outstands all input methods as seen in results of main I/O tests.

In order to minimize overhead when reading input, you need to avoid calling a function every time you need the next byte or the next chunk of input. There are two methods you can use that involve only a single system call: either you can read/fread the entire input file into an array, or you can use mmap to map the contents of the file into memory, which you can then treat as an array (the kernel will perform the actual input on demand when you attempt to read a page from the mapping). It’s then up to you to parse the bytes yourself.

In practice, the gains from doing input this way can be significant when the input consists of a large number of integers, because it avoids the overhead of a call to a library function for *every* integer. However, when the input consists mostly of long strings, the gains from reading the input this way instead of just using fgets are negligible.

1. #define ll long long int
2. #define ff first
3. #define ss second
4. #define pb push\_back
5. #define mp make\_pair
6. Most of the people use std::make\_pair or mp where #define mp make\_pair. There is a better alternative for this :
7. map<string, pair<int, int>> mis;
8. for (int i = 0; i < n; ++i)
9. {
10. cin >> x >> y >> str;
11. mis.insert(mp(str, mp(x, y)));
12. // Alternative
13. mis.insert({ str, {x, y} });
14. }

You can see which one is short and more readable. (This requires C++ 11)

2. Count the number of zeroes in MOD

1. auto MOD = 10000000000007L;

12? 13? 14? Now count :

1. auto MOD = 10'000'000'000'007L;

If you counted 12, then you'r right. (This requires C++ 14). This can be used with floating point numbers :

1. auto Float = 0.000'69'000'69;

Ability to read and understand other participants’ code is paramount in competitive programming in my humble opinion. The list below may be too excessive, but I still recommend that you know all the following:

* Basic syntax. Functions. Recursion.
* Bit operations.
* Pointers.
* Operator precedence. Competitive programming participants have a bad habit of skipping parentheses.
* Macros like

1. #define rep(i, a, b) for(i = (a); i < (b); ++i)

If you are not planning to use them, it is still useful to know how they work, when reading others’ codes.

* How to implement structs, classes, templates, and how to work with them.
* Stl containers: vector, list, set, map, etc. How to work with them, how they are implemented, and time complexity of each operation.
* Stl algorithms: sort, unique, random\_shuffle, etc. How to work with them, how they are implemented, and time complexity.
* How to implement BigInteger in C++ as a struct.

C++ benefits:

1. Code is usually shorter than Java. For example, instead of writing ArrayList<Integer> arr = new ArrayList<Integer>();  
   You can write vector<int> arr;
2. Strings are mutable. You can append to them and grow them as needed.
3. Easy printing formats. By using printf you can select the format you want to use for your output. This comes in handy especially when you’re trying to set the level of precision for doubles.
4. Nice utility functions like: [next\_permutation](http://www.cplusplus.com/reference/algorithm/next_permutation/), [rotate](http://www.cplusplus.com/reference/algorithm/rotate/?kw=rotate), [random\_shuffle](http://www.cplusplus.com/reference/algorithm/random_shuffle/?kw=random_shuffle), [reverse](http://www.cplusplus.com/reference/algorithm/reverse/?kw=reverse), [lower/upper\_bound](http://www.cplusplus.com/reference/algorithm/lower_bound/), [pair data structure](http://www.cplusplus.com/reference/utility/pair/?kw=pair), etc. These make solving some problems easier.
5. Easy to set up macros to make code shorter by using [#define](http://en.cppreference.com/w/cpp/preprocessor/replace#.23define_directives). For example, [this is the template](https://github.com/andmej/competitive_programming/blob/b9acac8a2b567b7087bfa26ce6deb0fa72764ffa/template.cpp#L28) me and my team used while we were competing.
6. Namespaces allow you to write code and reuse variable names within the same file, making it less bug prone.
7. And what I think is the most important reason of all: My ICPC teammates also used C++. Even though we all knew Java, our preferred language was C++. This made it easy to think of solutions together, debug each other’s code and do pair programming for the harder problems.

AFAIK the fastest way to clear a vector is to std::swap it (std::swap(vec, vector<long>()). Could be wrong though. imho without knowing more about your algorithm, it's hard to say, as Justin says.

12

I have a clear preference for small scopes (i.e. declaring the variable in the innermost loop if it’s only used there) but for large sizes this could cause a lot of allocations.

So *if* this loop is a performance problem, try declaring the variable outside the loop and merely clearing it inside the loop – however, this is only advantageous if the (reserved) size of the vector stays identical. If you are resizing the vector, then you get reallocations anyway.

Don’t use a raw array – it doesn’t give you any advantage, and only trouble.

clear kills the size but doesn’t actually free up any memory. Hence, reallocation doesn’t happen. clear followed by resize reinitialises the vector’s elements without costly heap allocations.

USE STATIC ARRAYS FOR STATIC MEMORY, USE VECTORS FOR DYNAMIC MEMORY WITH PUSH\_BACK CALLS

