Introduction

This guide covers some USACO basics, a review of common data structures and algorithms, and some tips and tricks for the USACO contests.

Like our C++ doc, it is a work in progress. If you find that you would like more detail on one of these topics, or on a topic that's not covered here, please let Ian know.

Running Time (Big O Notation)

Generally, C++ programs in USACO are allowed to run for at most 2 seconds. How can we guess whether a potential solution is actually fast enough?

One helpful tool is Big O notation. For our purposes, Big O is a mathematical representation of how an algorithm's worst-case runtime grows as the size of the input data grows. (”Size” here may refer to the amount of data, and/or to the data values themselves.) We calculate the runtime based on the number of *operations* needed to run the program, rather than the number of seconds needed.

For example, consider a function that takes in an integer n and prints out all the numbers from 1 to n. This involves n print operations, but that's not all – the computer needs to do some work under the hood to prepare those numbers for printing. Perhaps you have a for loop that's incrementing a value, and so we need to count the addition operations needed to do this. Fine – we may not know the exact total number of operations, or even how fast print operations are relative to addition operations (probably the former is much slower!) But we can agree that the total is probably roughly a multiple of n – or, in Big O notation, $O(n)$. We will sometimes say ”*linear* in n” to refer to $O(n)$.

The beauty (and whole point!) of Big O is that it ignores constant factors and lower-order terms. A function that takes in an integer n and prints out all the numbers from 1 to $2n$ would still have a runtime of $O(n)$. What if we also had it print ”BESSIE” at the end? You might be inclined to change the runtime to $O(n) + 1$ or perhaps $O(n + 1)$ to include this constant work, but the 1 is a lower-order term that we can handwave away – the asymptotic runtime is still $O(n)$. (Think of ”asymptotic” as just meaning ”as n gets very large”). Similarly, $O(n^2 + 500n)$ is really just $O(n^2)$, because you can imagine an n large enough that n^2 is enormously larger than $500n$.

As a practical competitive programmer, you may be protesting that our algorithms won't be running on enormously large values of n, and so we can't really just sweep constant factors like that 500 under the rug. True enough. However, Big O gives us a way to discuss roughly how an algorithm will scale with increasing data size, and we'll see that even something like $O(n^2)$ can be too slow for USACO.

First let's consider how fast some common individual operations are:

Runtimes of Common Data Structures

Data Structure	Runtime
`vector: push_back, pop_back`	$O(1)$
`vector: insert`	$O(n)$
`sort(v.begin(), v.end())`	$O(n \log n)$
`map: set, find, erase`	$O(\log n)$
`set: insert, find, erase`	$O(\log n)$
`unordered_map: set, find, erase`	$O(1)$
`unordered_set: insert, find, erase`	$O(1)$
`queue: push, pop`	$O(1)$
`stack: push, pop`	$O(1)$
`priority_queue: push, pop`	$O(\log n)$
`deque: push_back, push_front, pop_back, pop_front`	$O(1)$

If you haven't seen $\log n$ before, you can think of it as being somewhere between constant (1) and n, but *much* closer to the former. $O(n \log n)$, for instance, is much much better than $O(n^2)$.[1] However...

Note that while the `unordered_map` and `unordered_set` have constant time for `set`, `find`, and `erase`, the constant may be large enough that it is slower than the logarithmic time of the ordered `map` and `set`. Typically, the ordered data structures are faster when doing many lookups, while the unordered data structures are faster when doing many insertions.

Maximum Input Size

Oh, right, our original reason for being interested in big-O was to get a good sense of what solutions are viable in competition. A good rule of thumb for some back-of-the-napkin math is that the USACO judge computer will perform about 10^8 operations in 1 second.

Input Size	Worst-Case Time Complexity
any	$O(\log n)$ or $O(1)$
$n \leq 10^7$	$O(n)$
$n \leq 10^4$	$O(n^2)$
$n \leq 550$	$O(n^3)$
$n \leq 25$	$O(2^n)$
$n \leq 10$	$O(n!)$

Keep in mind that constant factors play a large role, especially at the upper bound of each runtime. For example, with an input size of $n = 10^4$, if your program happens to have a constant factor of more than 2 and a complexity of $O(n^2)$, it will not run in two seconds. It's common for USACO problems to have a simple $O(n^2)$ solution, but the problem authors will construct the test cases to ensure that such a solution passes few, if any, cases.

Arrays and Linked Lists

An array is a contiguous (i.e., all back-to-back and in the same place) block of memory locations. These locations can store a fixed-size collection of elements that are all of the same data type.[2] An element is accessed using an index, which is an integer value representing the position of the element in the array. The index starts at 0 and ends at the size of the array minus one.

[1] Strictly speaking, it's even better than $O(n^{1.00001})$, but now we're a little too far into the realm of theory...

[2] If they could be of different data types with different sizes, the computer would not be able to quickly calculate how to jump to a particular entry in the array.

Arrays are good for ”random access” – which means ”getting values from wherever in the array we want” – because accessing an element takes constant time, O(1). Arrays are also good for storing sequential data, such as strings, because they can be easily accessed and manipulated.

However, arrays have some limitations. The size of an array is fixed at the time of creation, so it cannot be resized dynamically. This means that if you need to store more elements than the array can hold, you must create a new, larger array and copy the old elements to the new array. This can be inefficient if the array is large or if it needs to be resized frequently. Additionally, arrays are not good for inserting or deleting elements in the middle of the array because it requires shifting the elements to make room or close gaps, which can also be inefficient.

A linked list, on the other hand, is a dynamic data structure that can grow or shrink as needed. A linked list is made up of *nodes*, each of which contains a piece of data and a reference to the next node in the list.[3] The first node is called the head, and the last node is called the tail.

Linked lists are good for inserting or deleting elements – from anywhere in the list! – because this only requires updating the references to the affected nodes, which can be done in constant time, O(1). That is, suppose we have an existing node X pointing to another existing node Y, and we want to insert a new node Z between X and Y. All we need to do is tell Z that Y is its successor, and then tell X that Z is now its successor. Deletion is similar, though it’s important to make these changes in the correct order.

However, linked lists have some limitations as well. Unlike arrays, they are not good for random access, because accessing an element requires traversing the list from the head all the way to the desired node. This can take time linear ($O(n)$) in the number of nodes in the list. Linked lists are also not as compact in memory as arrays, because each node requires an additional memory allocation for the reference to the next node.

C++ Vectors

In C++, a `vector` is a dynamic array that can grow or shrink as needed. The memory representation is similar to that of a regular array. It stores the elements in a contiguous block of memory, and each element is accessed using an index. If more elements are added to the `vector` than the allocated memory can hold, the `vector` automatically allocates a larger block of memory and copies the existing elements to the new block. (It’s important to remember that even though C++ does this resizing and copying for you under the hood, you still pay the cost of it!)

Some competitive programmers use arrays with a defined maximum size instead of `vector`s. This runs slightly faster, as it avoids reallocation of memory and other overhead. However, `vector`s are more convenient overall, as they have more built-in functions than arrays. Using an array instead of a vector will rarely mean the difference between passing and failing a test case.

Binary Search

Binary search is a technique that can be used to find the first value in some search space S that satisfies some condition $f(x)$ in $O(\log N)$ time. The idea is to split S in half each time, and to check if the value at the midpoint satisfies the condition. If it does, then we can search the left half of S, and if it doesn’t, then we can search the right half of S.

This is a little abstract, so let’s give a concrete example. You can play the following guessing game with a friend: ask them to secretly think of a number between 1 and 1000, and claim you will guess their number after asking at most ten questions of the form: is it larger than, smaller than, or equal to ¡some number¿? [4] Our strategy is to use

[3]This is a *singly-linked* list. In a doubly linked list, each node also has a reference back to its predecessor. Sometimes, when designing a data structure that involves a linked list, you will find that you need this property.

[4]We leave it as an exercise to figure out how the ”ten” and 1000 are related. One elegant way to look at it is: what are all the possible sequences of responses that your friend could give you, and how many of those are there? Could we even offer our friend a range larger than $[1, 1000]$, while still guaranteeing that we only need at most ten guesses?

each guess to rule out as much of the search space as possible, which means making a guess in the middle of the current search space. For instance, asking ”is it larger than, smaller than, or equal to 2” would probably get us almost no useful information, since our friend’s number is almost certainly greater than 2.

For example, suppose that (unbeknownst to us) our friend chooses 777. We start by guessing 500, since it is about in the middle of the range $[1, 1000]$. Our friend says ”higher”. Now we know the number is not in the range $[1, 500]$, so the search space has shrunk to $[501, 1000]$. The midpoint of this is 750.5, so we pick (say) 750 for our next guess, and so on. Inevitably, we either guess 777 or end up with a search space so small that 777 is the only possible answer.

Back to generalities! For a binary search to work, S must be sorted and $f(x)$ must be monotonic. That is, there exists some x such that for all $y \geq x$, $f(y)$ is true and for all $y < x$, $f(y)$ is false.

Example Implementation

Suppose we need to search for the smallest value in a sorted vector v that is greater than or equal to some value x.

The below code implements the algorithm from the previous section. At every step, we check the value of the midpoint is in relation to x, and change the bounds of our search space accordingly.

```
int lo = 0, hi = v.size() - 1;
while (lo < hi) {
  int mid = lo + (hi - lo) / 2;
  if (v[mid] < x) lo = mid + 1;
  else hi = mid;
}

// lo is now the index of the first element in v that is >= x
// if no such element exists, then lo == v.size()
// we should also check if v[lo] == x, if that's necessary
```

We use `lo + (hi - lo) / 2` instead of `(lo + hi) / 2` to avoid overflow, as `lo + hi` could be greater than `INT_MAX`.

Built-in Functions

There are two functions in C++ that are very similar to binary search, called `lower_bound` and `upper_bound`. `lower_bound` returns an iterator[5] to the first element in a sorted vector that is greater than or equal to some value x, and `upper_bound` returns an iterator to the first element in a sorted vector that is greater than x.

```
int index = lower_bound(v.begin(), v.end(), x) - v.begin();

// upper_bound syntax is the same as lower_bound
```

These functions can help in simple binary searches, but typing out the whole algorithm can help modify or debug more complicated binary searches.

[5]Iterators are memory addresses to elements in a vector. To get the index of an iterator, you can subtract the iterator from the beginning of the vector.

Thanks to our sponsors:

Jane Street

D E Shaw & Co

scale

CITADEL

SIG
SUSQUEHANNA