NOI.PH Training: Algorithms 1

Algorithm Basics

Robin Yu

Contents

1 Analysis of algorithms

In competitive programming, time is of the essence. This is true in more ways than one; our program needs to run *fast enough*, *within the time limit* given by the problem, but at the same time we as contestants are constantly battling the clock as the competition time runs down.

Hence, we do not have the time to implement all the algorithms we might think of just to see if they would pass the time limits given by the problem. Instead, before we even code something, which is usually a grueling process taking minutes to hours of competition time, we should first *convince ourselves* that this algorithm is fast enough, to avoid wasting time coding things that will be too slow.

We don't need very detailed analyses to convince ourselves that something might be too slow. Complex analyses take time, so we just want to get into the right ballpark of how long an algorithm might take. Luckily for us, there is a rather systematic way of doing it, and this is where big $\mathcal{O}$ analysis[1] comes in.

1.1 Big $\mathcal{O}$ notation

Big $\mathcal{O}$ notation allows us to describe, in asymptotic terms, how a function $f(x)$ *grows*. Formally, we write

$$f(x) = \mathcal{O}(g(x))$$

if there exists some positive constant c such that $|f(x)| \leq c|g(x)|$ for all sufficiently large x.

For example, if $f(x) = 10x^2 + x + 5$ and $g(x) = x^2$, then we can see that $f(x) = \mathcal{O}(g(x))$, because if we take $c = 11$, then for all sufficiently large x ($x \geq 2.79129\ldots$), $|f(x)| \leq 11|g(x)|$.[2]

On the other hand, if $f(x) = x^2$ and $g(x) = x$, then $f(x) \neq \mathcal{O}(g(x))$, because regardless of what c we take, for really large x, we will always eventually have $|f(x)| > c|g(x)|$.

Thus big $\mathcal{O}$ notation captures the idea of some function growing "slower than" or "around as fast as" another function asymptotically. To say that $f(x) = \mathcal{O}(g(x))$ is really to say that, if we ignored all the constants and took only the fastest-growing term, then $f(x)$ grows slower than or as fast as $g(x)$.

1.1.1 The use of big $\mathcal{O}$ in algorithm analysis

In practice, we use the big $\mathcal{O}$ notation to find an asymptotic upper bound for the number of operations an algorithm requires, or the amount of space an algorithm needs, almost always in the worst case[3].

For example, consider the program in Listing 1. How many operations does this program require as a function of n?

You might reason like this:

> "Well, it creates a variable, so that's 1 operation. Then it reads n, which is another 1 operation. It creates and initializes a variable i; is that 2 operations or 1? Let's say it's 2. Then, it does a loop from 0 to n; in each iteration, we first have to check

[1] https://en.wikipedia.org/wiki/Big_O_notation

[2] Of course, there may be other cs that work, such as $c = 420$.

[3] In competitive programming, we usually assume that the problemsetter is sadistic and gives input such that our algorithm takes as long as possible or requires as much memory as possible. Exceptions can be made for e.g. probabilistic or randomized algorithms where the problemsetter can't reasonably predict your program's particular constants and hence create worst-case inputs; however, with the advent of "adaptive checker problems", even this assumption can now sometimes be called into question.

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n;
    cin >> n;
    int i = 0;
    while (i < n)
        i++;
    i = 0;
    while (i < n)
        i++;
    cout << "wow\n";
}
```

Listing 1: C++ program that does a whole lot of nothing

> the condition and then increment, so that should be $2n$ operations. Then we set i to zero again, so that's another 1 operation. Then the loop is another $2n$ operations. Then, we print a string... is that 1 operation, or 4, one for each character? I think it should be 4..."

Adding it all up, you'll find that the program takes $4n + 9$ operations. Or maybe $4n + 5$ or $4n + 6$, or even something else, depending on how you counted the operations. This "works", but notice just how *tedious* and *error-prone* that process was, and how you will get different results if you count some operations differently.[4]

And, recall that our goal is to analyze algorithms for *programs we haven't written yet*, so this approach isn't exactly useful!

In our case, it's *much much easier and useful* to just say that the program takes $\mathcal{O}(n)$ operations. We just simply ignore all constant factors and slower growing terms.

Let's consider another program, Listing 2.

```cpp
#include <bits/stdc++.h>
using namespace std;

int arr[100005];

int main() {
    int n, sum = 0;
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> arr[i];
    for (int i = 0; i < n; i++)
        for (int j = i+1; j < n; j++)
            sum += arr[i]*arr[j];
    cout << sum << '\n';
}
```

Listing 2: C++ program that does a whole lot of something

[4] Furthermore, not all operations are the same cost, so the "total number of operations" doesn't even need to exactly represent the running time, which means getting the actual exact count is useless.

How many operations does it take, as a function of n?

If you were to follow the previous process in detail, you would get something like an^2+bn+c for some integers a, b, c, which will depend on how you counted the operations[5]—very tedious! But notice that, regardless of b and c, if a is positive (and it will be), this will just simplify to $\mathcal{O}(n^2)$ if we used big $\mathcal{O}$ notation. In general, we should just look at the fastest growing term; in this case, that's the n^2 term.

When you look at this program, your attention should immediately focus on the two nested loops; that's where the bottleneck is going to be. Since those two loops go to n, you know that the algorithm is immediately going to be $\mathcal{O}(n^2)$[6], so you need not focus on the other parts. In practice, this is how you might analyze algorithms in your head; think of where the bottleneck is going to be, and then determine, ignoring constants, the number of operations, as a function of your inputs, that bottleneck part is going to take.

We usually say that an algorithm that takes $f(n)$ operations takes $\mathcal{O}(f(n))$ *time*, or *runs in* $\mathcal{O}(f(n))$. These are simply other ways of saying the same thing.

Likewise, the notion of big $\mathcal{O}$ can be extended to memory; a program that uses $f(n)$ bytes of memory is usually said to *use* $\mathcal{O}(f(n))$ *memory*.

Let's run through a couple of examples:

1. An algorithm that calculates the sum $1+2+3+\cdots+n$ by looping through them all takes $\mathcal{O}(n)$ time[7] and uses $\mathcal{O}(1)$ memory[8].
2. An algorithm that uses the formula $n(n+1)/2$ to calculate the same sum takes $\mathcal{O}(1)$ time and uses $\mathcal{O}(1)$ memory.
3. An algorithm that receives as input an array of length n, then computes the sum of products of all unordered pairs of integers by trying them all (Listing 2) takes $\mathcal{O}(n^2)$ time and uses $\mathcal{O}(n)$ memory.
4. An algorithm that receives as input an array of length n, then prints all of its permutations takes $\mathcal{O}(n \times n!)$ time and uses $\mathcal{O}(n)$ memory.
5. An algorithm that factorizes a number n using optimized trial division takes $\mathcal{O}(\sqrt{n})$ time and uses $\mathcal{O}(1)$ memory.

Note that it is technically also correct, by the definition, to say that some algorithm that runs in $\mathcal{O}(n)$ also runs in, say, $\mathcal{O}(n^2)$. However, in general, we are interested in the *tightest possible* upper bound, because that is what will allow us to estimate best whether or not an algorithm can pass the time limit for a given problem.

[5] I couldn't be bothered to do it myself, and we would probably get different results anyway.

[6] Obviously there are some exceptions.

[7] We generally consider the usual operations (addition, multiplication, division, ...) to be $\mathcal{O}(1)$, since we are normally just dealing with 32-bit or 64-bit integers; the 32 and 64 there are considered as constants. (We will explain in more detail later what 32- and 64-bit integers are, if you are not familiar with them.)

[8] Again, the memory is technically not $\mathcal{O}(1)$, but we usually assume that it fits inside a 32-bit or 64-bit integer.

2 Basic algorithms

In this section, we will describe algorithms for some basic problems. We will also describe basic algorithm *patterns* to help you come up with your own algorithms.

2.1 Sorting algorithms

One of the common tasks we face in competitive programming, and indeed programming in general, is sorting a list of n elements. This is a standard problem and many classic algorithms have been invented to solve it; here, we go through some of the most basic ones.

2.1.1 The slow $\mathcal{O}(n^2)$ algorithms

The simplest and most straightforward algorithms tend to be the slowest. This is a recurring theme in competitive programming; faster algorithms always need more observations. This is true even in sorting algorithms.

Selection sort is perhaps one of the simplest sorting algorithms. This algorithms sorts a list by finding the minimum element and swapping it with the first element, then finding the minimum element in the remaining list and swapping it with the second element, and so on.

Figure 1 is an illustration demonstrating a couple of steps in selection sort.

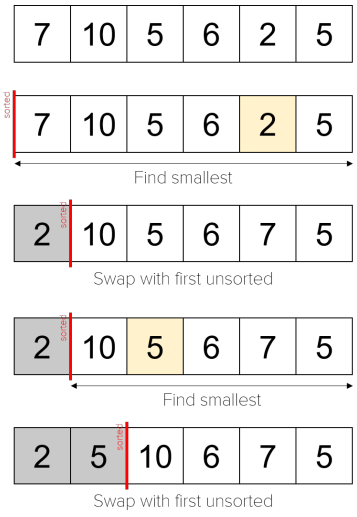

Figure 1: Selection Sort

This algorithm is clearly $\mathcal{O}(n^2)$, because it does around n comparisons the first time, then $n-1$, then $n-2$, and so on until 1, and $1+2+3+\cdots+n = \mathcal{O}(n^2)$.

Insertion sort is similar in that it also maintains a partially sorted list as a prefix as it sorts the list. In this scenario, however, we repeatedly take the next unsorted element and look for where it goes in the sorted prefix. This is illustrated in Figure 2.

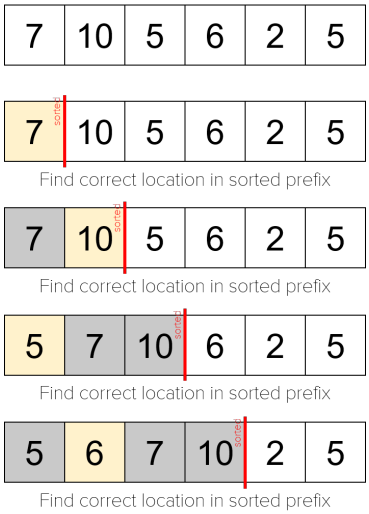

Figure 2: Insertion Sort

This also runs in $\mathcal{O}(n^2)$ in the worst case, because we need to shift up to n elements for each of the n elements we add to maintain the sorted prefix.

Besides these two, there are also other slow $\mathcal{O}(n^2)$ sorting algorithms, such as the **cocktail shaker sort**, **odd–even transposition sort**, **gnome sort**, and so on. Of course, these other sorting algorithms are generally not used in serious competitive programming, so you do not really need to learn them.

2.1.2 Merge sort

Merge sort is an $\mathcal{O}(n \log n)$ recursive divide and conquer sorting algorithm. In simple terms, it sorts a list by dividing it into two lists, sorting these two lists using merge sort, and then merging these two sorted lists in linear time. If a list only contains one element then it is already sorted, so nothing needs to be done.

The division step proceeds like so:

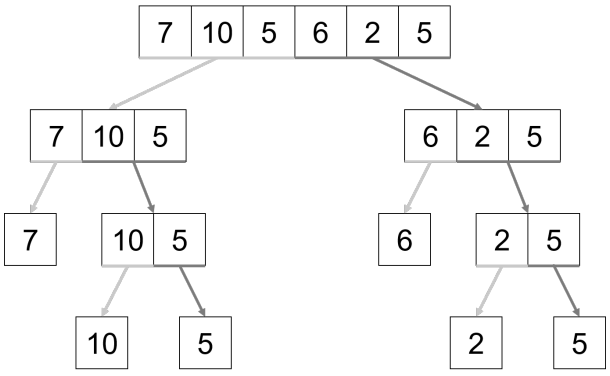

Figure 3: Merge sort: divide

The merge step proceeds like so:

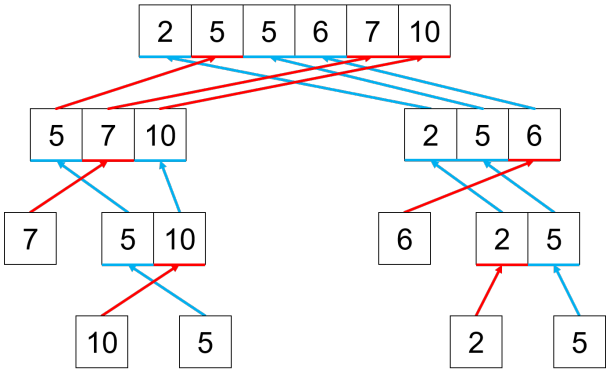

Figure 4: Merge sort: merge

Note that, as in typical recursive algorithms, these steps are intertwined. Of course, this would be hard to illustrate only with static images. Websites like Visualgo[9] allow you to see—interactively—how these algorithms proceed, step-by-step.

How do we merge two sorted lists into one big list in linear time? Consider the two sorted lists as two queues. Then, we just repeatedly do the following: consider the front elements of the queues, and take the smaller one and add it to the end of the big list. Continue this way until either queue becomes empty, then just dump the rest of the non-empty queue into the big list. This clearly runs in $\mathcal{O}(n)$ time, and is quite intuitively correct. Figure 4 attempts to illustrate this process using red and blue arrows, but you may prefer the interactive Visualgo website instead.

Notice that, in the tree structure above, every layer contains at most $\mathcal{O}(n)$ elements, and there are $\mathcal{O}(\log n)$ layers. It is not hard to see, then, that the algorithm runs in $\mathcal{O}(n \log n)$ time.

2.1.3 Quicksort

Quicksort is another example of a recursive divide and conquer algorithm. In simple terms, it sorts a list by first choosing some element in the list to be a pivot; then, it puts all the elements *smaller* than the pivot as a prefix of the list, then all the elements *larger* than the pivot as a suffix of the list, then all the elements equal to the pivot in the middle. Then, it recursively sorts both the prefix and suffix. See Figure 5.

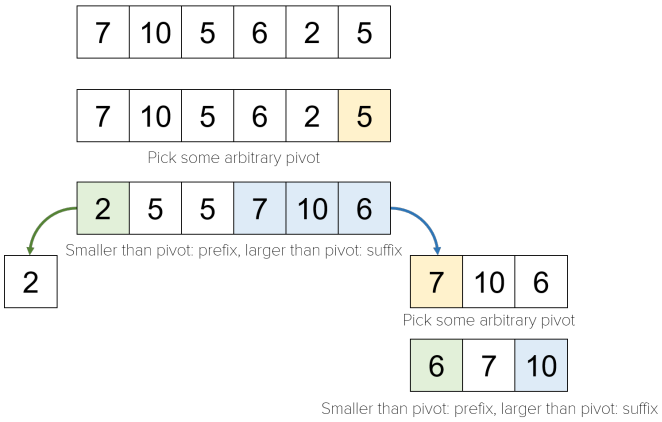

Figure 5: Quicksort

Again, you may prefer to refer to the Visualgo website, as this static picture cannot do the

[9] https://visualgo.net/bn/sorting

algorithm much justice.

Algorithm 1 shows a simple-to-understand pseudocode for the Quicksort algorithm. It is neither the shortest nor the fastest, but it should be easy to digest.[10]

Algorithm 1 QuickSort

```
function QuickSort(list A)
    if A is empty then
        return A
    else
        // Choose a pivot
        p ← random integer from 0 to |A| − 1
        // Partition the list A using the pivot A[p]
        Create three new empty lists: L, M and R
        for each v in A do
            if v < A[p] then
                Append v to L
            else if v = A[p] then
                Append v to M
            else
                Append v to R
            end if
        end for
        // Recursively sort L and R
        L ← QuickSort(L)
        R ← QuickSort(R)
        return the concatenation of L, M and R
    end if
end function
```

If the pivots are chosen well, that is, chosen such that the array is split "roughly in half" each time, then by a similar analysis as merge sort it can be seen that Quicksort also runs in $\mathcal{O}(n \log n)$. If the pivots are chosen badly, however, the array is almost never split and we just remove one element each time, then performance degrades to $\mathcal{O}(n^2)$. In practice, however, this does not happen as long as the pivot is chosen randomly, so we tend to assume that the runtime is $\mathcal{O}(n \log n)$. In fact, there are more sophisticated methods to *guarantee* $\mathcal{O}(n \log n)$ runtime for Quicksort, such as the median of medians pivot strategy, but they also increase the constant factor and are usually not necessary.

Quickselect is a selection algorithm that builds off of the idea of Quicksort. It solves a simpler problem: given a list of n elements, can you find the kth element if this list were sorted? Of course, it is easy to just sort the list and solve this problem in $\mathcal{O}(n \log n)$ time; however, Quickselect gives an *expected* $\mathcal{O}(n)$ algorithm.

The idea is the same as Quicksort. Pick a random pivot, then partition all the elements to smaller and larger. After this, however, we will know *how many* elements are smaller than the pivot, how many elements are equal to the pivot, and how many are larger than the pivot. Hence, we will know which of the three groups the kth element must be; we then just recurse to that group (if the group is the one equal to the pivot, we're actually done), and ignore the rest.

[10] Real-world implementations of quick sort tend to perform quick-sort "in place", that is, by simply reusing the input array and moving things around, since that's faster in practice. We've chosen to include a slower but conceptually simpler pseudocode to communicate the algorithm better.

It's essentially Quicksort but it only "sorts" one of the two partitions. If we assume that the pivots are chosen nicely such that the array is halved each time, then the first time we need approximately n operations, the next time $n/2$, the next time $n/4$, and so on; $n + \frac{n}{2} + \frac{n}{4} + \cdots \approx 2n = \mathcal{O}(n)$, so Quickselect runs in $\mathcal{O}(n)$ time, which is faster than sorting in $\mathcal{O}(n \log n)$. Problems involving Quickselect tend to be few and far between, especially in modern programming contests, because it is usually hard to distinguish between $\mathcal{O}(n)$ and a "fast" $\mathcal{O}(n \log n)$. It is still useful to know it, however.

2.1.4 Special-purpose sorting algorithms

It has been proven that any comparison sort—that is, any algorithm which sorts a list by comparing elements to determine which one is smaller, such as all the algorithms above—take at least $\Omega(n \log n)$ time[11]. We've shown in the previous section two algorithms that do achieve this lower bound; hence they are already asymptotically optimal.

If our input has some special properties, however, we can avoid using a comparison sort and hence achieve potentially faster runtimes.

Counting sort is a very simple sorting algorithm if the numbers are small. For example, if you want to sort an array of n numbers, but all the elements are positive integers up to 1000 only, then just create 1000 "buckets", and throw each of the n elements into the buckets, then go through the buckets in order and then retrieve the elements afterwards. This is clearly $\mathcal{O}(n + c)$, where c is the largest element (as we will make c buckets).

Bucket sort is a generalization of counting sort, and works if c is slightly too large to just make c buckets. Instead of the ith bucket just representing the value i, we can let it represent a "range" of values, e.g. bi to $b(i + 1) - 1$ for some b. For example, if we let $b = 100$, then the 0th bucket might represent 0 to 99, the 1st bucket 100 to 199, and so on. We throw all the elements into their corresponding buckets, and then, for each bucket, sort the elements in it, possibly using another bucket sort. After all buckets are sorted, we can easily reconstruct the sorted array.

Radix sort is similar to bucket sort, but the buckets represent digits. Let the number of digits of the largest number be d. Then, append leading zeroes to all other numbers so that they also have d digits. Now, consider the first digits of all the numbers; throw those with digit 0 into one bucket, those with digit 1 into another, those with digit 2 into another, and so on. Now, all those in the 0th bucket are smaller than those in the 1st bucket, which are themselves smaller than those in the 2nd bucket, and so on. So, sort each bucket again using radix sort, but now using the second digit, and so on. This takes $\mathcal{O}(dn)$ time, and can be faster than $\mathcal{O}(n \log n)$ if d is small.

As a conclusion to this subsection, remember that C++, Java and Python all have *fast built-in sorting functions*, which usually implement one of the $\mathcal{O}(n \log n)$ sorting algorithms above[12], and we usually do not have to reinvent the wheel. Still, knowing how these sorting algorithms, particularly the faster ones, work can be useful, as there are certain insights which can be gleaned from these classical sorting algorithms that might reoccur in other aspects of other, completely different algorithms.

[11] $\Omega(f(n))$ is the opposite of $\mathcal{O}(f(n))$—it is a lower bound, rather than an upper bound.

[12] or modified versions of them; for example, C++ uses Introsort, which is a sort of hybrid of Quicksort and another sorting algorithm, Heapsort.

2.2 Searching algorithms

Another common task we face in competitive programming is searching. Given a list of n elements, can you determine whether some number x is in that list?

If we have just one x to check, then clearly we cannot do better than the following algorithm: go through all the n elements and check whether any of them are equal to x. This is a **linear search** algorithm, and if our list doesn't have any special properties, then this is usually the best we can do.

If we have multiple, say q, values x to check, this will take $\mathcal{O}(nq)$ time, which might be too slow. In this case, we might choose to preprocess the list. We can sort the list using one of the faster sorting algorithms discussed in the previous section, and then use the **binary search** algorithm on it.

What is the binary search algorithm? Well, suppose the list is sorted. Consider the middle element in this list. Is it equal to x? If not, it is either less than x or greater than x. If it is less than x, then we know we need to continue our search on the right side of the array, since all the elements to the left will also be less than x and hence cannot be equal to x; otherwise, we need to continue our search on the left side, since all the elements to the right will also be greater than x and hence cannot be equal to x. Repeatedly querying the middle of the array like this allows us to split the search range in half each time, thus we can determine whether or not x exists in the list in $\mathcal{O}(\log n)$ time. (Once there is only one element left in our search range, and it is not equal, then we know that x is not in the list.)

Figure 6 should explain it clearly. A typical implementation of binary search looks something like Listing 3.

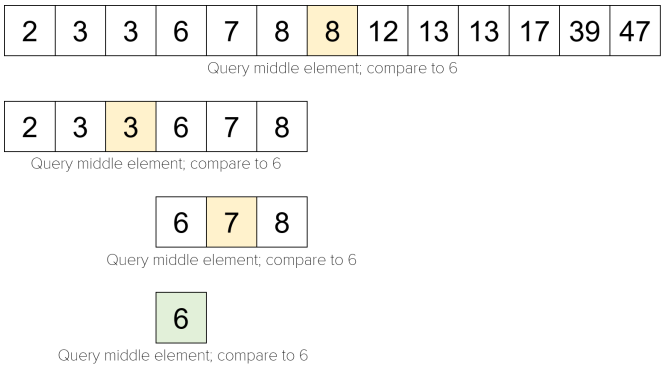

Figure 6: Binary search

It might be more natural to implement it recursively, which we invite[13] the interested reader to do.

We can also modify the binary search to locate the *first instance* of an element x; see Listing 4.

2.2.1 Binary search the answer

Although binary search is presented here as a seemingly standalone algorithm on an array, its ubiquity cannot be understated! Binary search shows up almost *everywhere*, especially in competitive programming.

Whenever you have any arbitrary predicate $f(x)$ that satisfies $x \leq y \implies f(x) \leq f(y)$ (or $x \leq y \implies f(x) \geq f(y)$), and you want to find, say, the smallest x satisfying $f(x) \geq p$ (or

[13] not require

```cpp
#include <bits/stdc++.h>
using namespace std;

int arr[1000005];

int main() {
    int n, q, x;
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> arr[i];
    sort(arr, arr+n);
    cin >> q;
    while (q--) {
        cin >> x;
        int lo = 0;
        int hi = n-1;
        bool found = false;
        while (lo <= hi) {
            int mid = (lo + hi)/2;
            if (x == arr[mid]) {
                found = true;
                break;
            }
            else if (x > arr[mid])
                lo = mid+1;
            else
                hi = mid-1;
        }
        if (found)
            cout << x << " is found!\n";
        else
            cout << x << " is not found.\n";
    }
}
```

Listing 3: C++ implementation of binary search

$f(x) \leq p$) for some p, you can do binary search on that predicate!

For example, consider the problem Country Story,[14] from the NOI.PH 2018 Fun-Filled Christmas Practice Contest Extravaganza[15]:

Problem 2.1. You have an $R \times C$ grid. Some cells have trees on them, and you can cut down at most K trees. What is the maximum possible size of a square subgrid containing no trees? Assume $R, C \leq 2000$.

Let's invert the problem, so that it instead asks: what is the maximum size of a square subgrid with at most K trees?

Then, we can define $f(x)$ as the minimum number of trees across all $x \times x$ subgrids. It should be clear that f satisfies $x \leq y \Leftrightarrow f(x) \leq f(y)$; that is; if you increase the size of your subgrid, then the minimum number of trees across all subgrids of that size must also increase (or stay the same).

[14] https://www.hackerrank.com/contests/noi-ph-2018-practice/challenges/playfish-2

[15] I will never not take the opportunity to say that contest's full name.

```cpp
#include <bits/stdc++.h>
using namespace std;

int arr[1000005];

int main() {
    int n, q, x;
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> arr[i];
    sort(arr, arr+n);
    cin >> q;
    while (q--) {
        cin >> x;
        int lo = 0;
        int hi = n-1;
        int location = -1;
        while (lo <= hi) {
            int mid = (lo + hi)/2;
            if (x > arr[mid]) {
                lo = mid+1;
            } else {
                location = mid;
                hi = mid-1;
            }
        }
        if (location != -1)
            cout << x << " is found, leftmost occurrence at " << location << '\n';
        else
            cout << x << " is not found.\n";
    }
}
```

Listing 4: C++ implementation of modified binary search

Hence, we can simply *binary search* for the *largest* x satisfying $f(x) \leq K$; we can compute $f(x)$ for arbitrary x in $\mathcal{O}(RC)$[16], and x itself can only be up to the smaller of R and C, so we have an algorithm that runs in $\mathcal{O}(RC \log \min(R, C))$ time, which is "basically $\mathcal{O}(RC)$", and is surely fast enough!

The domain of the function f need not even be integers, and p need not be an integer. Consider the following problem:

Problem 2.2. There are N circles at different points on a 2D plane, each with radius r. What is the minimum r such that there exists a path from any circle to any other circle that is always in at least one circle at any point?

One straightforward way to solve this problem is to define $f(x)$ as 1 if there exists a path from any circle to any other circle that is always on at least one circle at any point if all the radii are x, and 0 otherwise.

Computing $f(x)$ for a fixed x is actually not hard, and can be done in $\mathcal{O}(N^2)$ using some

[16]This can be done using a so-called summed-area table, a relatively simple and intuitive data structure. It's basically the 2D version of the "prefix sum array".

graph algorithms[17].

We are simply looking for the smallest x such that $f(x) = 1$, and hence we can binary search for x.

How might one implement this sort of "continuous binary search"? Typically, we do not want to check if $l \leq r$ as floating-point arithmetic is imprecise by nature, and our algorithm might never terminate. Instead, we can just have the loop run a fixed number of times; usually 50 is enough to get a very accurate answer, enough for most problems.

In this way, we can perform binary search even on predicates whose domains are real numbers.

2.2.2 Avoiding off-by-one errors

Binary search is conceptually simple. However, its implementation is somewhat tricky, especially if you haven't done it a lot—it's easy to fall into all sorts of off-by-one errors because of all the variables keeping track of ranges and whatnot.

Like many things in programming, the best way to avoid these problems is *practice*. Nevertheless, it doesn't hurt to learn some tips and tricks from more experienced programmers. Below, I'll give some advice on how to reduce the risk of mistakes. I'll also provide a way to implement binary search that's a little different from the ones discussed above.

In general, binary search involves a predicate, say $good(x)$, which is either true or false. This doesn't always need to be an actual function in your code, but if you're having trouble with implementation, I recommend explicitly defining this function.

One requirement for this predicate is that it should be *monotonic*, that is, if $good(x)$ is true, then $good(y)$ must also be true for all $y > x$. We can then say that "binary search" is a procedure that finds the smallest y such that $good(y)$ is true. Equivalently, it finds the largest x such that $good(x)$ is false. Of course, we must have $y - x = 1$.

Now, suppose you know two integers $L < R$ such that $good(L)$ is definitely false, and $good(R)$ is definitely true. Then the way I usually implement binary search is to *repeatedly halve the size of $R - L$ while maintaining the invariant that "$L < R$, $good(L)$ is false, and $good(R)$ is true"*. I stop when $R - L$ becomes 1, *not* 0.[18]

All in all, this is how my implementation usually looks like:

```
int L = ..., R = ...;
// we assume that L < R, good(L) is false, and good(R) is true.
while (R - L > 1) {
    int M = (L + R) / 2;
    if (good(M))
        R = M;
    else
        L = M;
}
// R is now the smallest integer y such that good(y) is true
// L is now the largest integer x such that good(x) is false
```

I prefer this implementation because there are very little index antics needed; no `+1` or `-1`.

[17] This can be done by modeling each circle as a vertex and connecting them by an edge if and only if their corresponding circles intersect, then checking if the graph is connected using a simple DFS or BFS. This will be covered in later weeks if you do not know them already, so do not despair.

With a more advanced techniques from computational geometry, the problem can actually be solved in $\mathcal{O}(N \log N)$ time. These techniques won't be discussed here, but feel free to ask in Discord.

[18] Of course, if $R - L = 0$, then it's impossible to satisfy the invariant.

Now, suppose you're debugging your solution. To help with this, I recommend adding a bunch of `assert` statements as sanity checks, like this:

```
int L = ..., R = ...;
assert(L < R);
assert(!good(L));
assert(good(R));
while (R - L > 1) {
    int M = (L + R) / 2;
    if (good(M))
        R = M;
    else
        L = M;
    assert(L < R);
    assert(!good(L));
    assert(good(R));
}
assert(R - L == 1);
```

These extra calls to the function **good** introduce a running time overhead, especially if **good** is expensive to compute, but this is fine for debugging purposes; just remove them in your final submission.[19]

There's another potential issue with this—what if the function $good(x)$ only makes sense when x is within a certain interval, say $[U, V]$? For example, what if $good(U-1)$ and $good(V+1)$ don't make sense? In this case, the way I usually deal with this is to simply *extend* the definition of $good(x)$ to allow for arguments x outside the interval. To maintain monotonicity, we simply return *false* if $x < U$, and *true* if $x > V$. Then we can start our binary search with the pair $(L, R) = (U-1, V+1)$.

```
int U = ..., V = ...;
bool _good(int x) {
    // actual implementation of the "good" function.
    // It's okay for this to only make sense if U <= x <= V
    ...
}

bool good(int x) {
    if (x < U) return false;
    if (x > V) return true;
    return _good(x);
}

int main() {
    ...
    int L = U-1, R = V+1;
    while (R - L > 1) {
        int M = (L + R) / 2;
        if (good(M))
            R = M;
        else
            L = M;
    }
    ... // binary search done
}
```

[19] If you want, you can keep the "cheap" assert statements such as `assert(R - L == 1)` or `assert(L < R)` since they're, well, cheap.

2.3 Greedy algorithms (are dangerous)

One common type of problem you will encounter is the *optimization problem*. Optimization problems ask you to minimize or maximize a particular value.

Optimization problems can sometimes be solved with a **greedy algorithm**, which is simply an algorithm that makes decisions without ever reconsidering past decisions. However, *often they can't*! The purpose of this subsection is mostly to *discourage* you from immediately jumping to an intuitive greedy algorithm, which unfortunately many beginners do in our experience. Heed our advice, so you don't have to learn this lesson the hard way—that is, by getting stuck trying to "debug" a doomed solution and burning all your remaining contest time as a result.

2.3.1 Greedy is often bad

To illustrate this dangerous pitfall, consider the following problem, called the **Coin Change Problem**:

> **Problem 2.3** (Coin Change Problem)**.** In a strange country called Noippon, the currency is *pesu*, and there are only three types/denominations of coins: a 1-pesu coin, a 10-pesu coin, and a 25-pesu coin. What is the minimum number of coins needed to make a change of n pesus?

This is an optimization problem—we are trying to minimize a particular value, namely "the number of coins" needed to make change.

Now, because people have real-life experience with coins, there's a very natural, tempting "solution" to this problem:

1. First, use as many 25-pesu coins as you can.
2. Next, use as many 10-pesu coins as you can.
3. Next, use as many 1-pesu coins as you can.

This is a "greedy" algorithm since it tries to perform the seemingly "most optimal action at the moment" without looking back. Because "obviously" it's always optimal to use the largest denomination as much as you can, right?

Many beginners immediately arrive at this solution, and immediately get convinced that it's correct (or rather, never bother checking whether it's correct or not); after all, this is how you normally make change in the real world in the most efficient way, and in your experience, this always leads to the optimal solution (that is, the minimum number of coins). They will then try to implement this, submit, and get faced with the dreaded "Wrong Answer" verdict. Finally, since beginners aren't the most confident with their implementation skills, they'll think "maybe there's just a bug in my implementation", so they will spend the remaining contest trying to "fix" it.

You don't want to be that contestant!

In fact, this solution is wrong. The smallest counterexample is $n = 30$. The greedy algorithm produces the decomposition $30 = 25 + 1 + 1 + 1 + 1 + 1$ with 6 coins, but in fact, it can be done with 3 coins: $30 = 10 + 10 + 10$. There are many more counterexamples such as $130, 230, 330, \ldots$; in fact, there are infinitely many counterexamples. Thus, trying to fix this solution is hopeless since it's fundamentally broken!

"But why does it work with real-world coins?" you ask. In fact, it's not a coincidence, but a design choice: coin denominations in the real world are *designed* to make the greedy algorithm

work. So this is no accident, since as we've seen above, it doesn't always work with any set of denominations. Your real-life experience has fooled you!

The main takeaway here is that you shouldn't assume that a greedy solution is automatically correct just because it "feels correct". You should try to find a reasonable argument convincing yourself that it's correct. As you get more experience, you'll come to realize that you don't necessarily always have to do this, but if you're just starting out, I recommend not jumping to the greedy solution first, especially if you don't have a good reason to believe that it's correct.

2.3.2 Greedy is sometimes good (but often bad)

Warning: If you're on the Bootcamp track, I recommend skipping this part for now.

That being said, there are problems where the greedy algorithm works. For example, consider the following very easy optimization problem:

Problem 2.4. There are N items in a store. The ith item costs C_i dollars. You have B dollars. What's the maximum number of items you can buy?

In this case, the "obvious answer" is to sort all the items by cost, then proceed to buy them from the cheapest one forward until we can't buy any more items. This is an example of a greedy algorithm, since it just repeatedly takes the cheapest item, never reconsidering past decisions.

This solution is "intuitively correct", but to prove it formally, we can use a classic proof technique called the **exchange argument**, which finds use in proving many (simpler) greedy algorithms.

The exchange argument typically proceeds as follows: consider the solution your algorithm produces, A, and suppose we have any arbitrary optimal solution, O. Then, we can manipulate O to A using a series of moves, such that at each step O remains optimal. This will show that A is also optimal.

Let's see how we might prove our greedy algorithm using this technique.

Proof. Without loss of generality, assume that $C_1 \leq C_2 \leq \cdots \leq C_N$. If not, just sort C and just change the indices of our optimal solution.

Suppose we have any optimal solution that purchases k items $\{a_1, a_2, \ldots, a_k\}$. Without loss of generality, assume the a_is are increasing, that is $a_1 < a_2 < \cdots < a_k$, and hence, $C_{a_1} \leq C_{a_2} \leq \cdots \leq C_{a_k}$.

Now, note that if $a_1 = 1, a_2 = 2, \ldots, a_k = k$, then this is precisely the solution our greedy solution produces, so in this case our greedy solution is optimal.

Otherwise, there exists some i where $a_i \neq i$; consider the smallest such i. It should be clear, that $i < a_i$.

But by our assumption, we have $C_i \leq C_{a_i}$, so we can always choose *not to buy* C_{a_i} and instead buy C_i (this is true regardless of B), while keeping the solution optimal—the number of items is still the same!

Repeatedly doing this "exchange" will eventually yield the greedy solution. Since we assumed our original solution was optimal, this shows that our greedy solution is also optimal; simply, any optimal solution can be changed to our greedy solution without losing optimality. □

In practice, it is usually hard and often tedious to prove that a greedy algorithm is correct,

especially under the time pressure of the contest.[20] In practice, you might want to refer to a guide like this:

1. If the greedy algorithm will not take to long to code ($5 \sim 10$ minutes), just code it and submit.[21] Remember that the IOI (and the NOI.PH) does not penalize wrong submissions, so at worst you just lose a couple of minutes.

2. If the greedy algorithm will take a bit longer to code, try (many) cases and make sure your greedy algorithm passes all of them before coding. **Important Tip:** *Be extra cautious* if you find yourself having to consider all sorts of special cases to make your algorithm work, *especially* if you are just coming up with patchwork remedies whenever you encounter them; once your algorithm starts looking less like an algorithm and more like a clobbering of special cases, you should start to wonder whether your algorithm is actually correct. **Do not** start coding an algorithm like this when you have not yet ironed out all the details! There are few feelings worse than finding a fatal flaw 30 minutes into coding a greedy algorithm.

3. If the greedy algorithm will take a very long time to code, you may want to try proving it formally, or at least having a general gist of a proof of why it should work.

There are many optimization problems in recent contests, including the NOI.PH and the IOI, where the solution uses a greedy algorithm, but this greedy algorithm tends to be hard to spot and hard to prove. Indeed, missing a greedy solution can be a costly mistake, and can cause one to just lose out on a medal. Thus, it may be a good use of your time to practice your intuition with greedy algorithms.

2.4 Divide and conquer algorithms

Another kind of algorithm which is more rare in recent contests are the **divide and conquer algorithms**. We've shown a few examples of these above; the merge sort and Quicksort algorithms are two of the standard divide and conquer algorithms. The "pure" divide and conquer principle rarely appears in modern contests[22], but when they do the solution is often novel and beautiful.

The basic principle of divide and conquer algorithms is to divide your problem into (usually two) smaller *independent* problems, solve those subproblems (usually recursively), then *combine* the solutions of those two problems into a solution for the big problem.

Let's consider the classic 1D **max subarray sum problem**:

Problem 2.5. You have an array A of N elements, some of which might be negative. Find a contiguous subarray of A with the maximum sum.

There is a standard greedy $\mathcal{O}(N)$ solution to this problem[23], but did you know that there is also a divide and conquer solution to this problem?

Consider the middle element A_m. There are two cases:

[20] Of course, there are some exceptions: you started doing math at age one and trading stocks at age three, you won second prize in the math category at the largest science fair in the world for a paper you co-wrote, etc.

[21] Make sure to code it carefully! This way, if it is wrong, you know it is your algorithm that has the issue and not your implementation.

[22] This is primarily because it is hard to come up with novel divide and conquer problems.

[23] Kadane's algorithm; as usual, greedy algorithms need a proof of correctness.

1. A_m is not in the optimal solution. This means that the optimal solution must be completely in either $A[0, m-1]$ or $A[m+1, N-1]$. But these are two separate cases! Just solve them recursively and consider the better one.

2. A_m is part of the optimal solution. Now, the answer will consist of a *suffix* (possibly empty) of $A[0, m-1]$, followed by A_m, followed by a *prefix* (possibly empty) of $A[m+1, N-1]$. But it is obvious that we should just take the best prefix and the best suffix, so we can consider this case in just $\mathcal{O}(N)$!

Now, we just take the better of first and second cases.

How long does it run? It should not be hard to discover that this algorithm is $\mathcal{O}(N \log N)$; there are $\mathcal{O}(\log N)$ "layers" (as the array sizes are always being divided by 2, until they become 1 or 2, which we can consider base cases), and in every "layer" we are just doing a total of $\mathcal{O}(N)$ work.

This is "worse" than the $\mathcal{O}(N)$ greedy solution, but you can see that this is somehow more versatile, and the general technique might be useful in more scenarios.

2.5 Brute force algorithms

It happens. You've spent the last three hours thinking about that one killer problem without making any progress. You've thrown extremely elaborate greedy heuristic solutions at it. You've tried reducing the problem, à la divide and conquer, to smaller independent subproblems. You've tried approaching it from all sorts of different angles. Nothing seems to work. It is time to give up your hopes of scoring full points on this problem and settle for *something*. It is time to attack this problem the only way you know how: brute force.

It is not always straightforward to write a brute force solution. If you're lucky, it's as simple as writing a couple of nested loops, but often, especially in more difficult problems, even writing a brute force solution can be tricky. Here, we introduce a powerful tool that lets us write brute force solutions to many kinds of problems.

2.5.1 Recursive backtracking

Let's consider the problem Nyeh Heh Heh!, from the NOI.PH 2018 Fun-Filled Christmas Practice Contest Extravaganza[24]:

Problem 2.6. Find B distinct positive integers below K such that their sum is N or say that it is not possible.

There is a nice and clean solution to this problem which requires a number of insights. But let's say this problem came out in a contest, and let's say, god forbid, that you've spent a long time on this problem without making any progress. On the verge of giving up, you notice that one subtask has $K \leq 15$, and is worth 12 points.

Now, 12 points isn't a lot, but it's better than nothing—besides, a couple of points here and there can determine the difference between ranks, and sometimes even between medals, as I'm sure someone would have experienced firsthand in the NOI.PH 2018 finals.

How might we write a brute force for this problem? It is not immediately obvious. With these kinds of problems, however, we can often define a recursive function to do it for us.

[24] Yeah, I meant it.

Let $f(s)$ be true if we can select a total of B numbers from 1 to K whose total sum is N, *considering that we have already selected some numbers* s, and false otherwise. Now, all we want to know is whether $f(\varnothing)$ is true.

How do we compute $f(s)$ for some s? Well, we usually first want to define some *base cases* for our function:

- If there are more than B numbers in s, then it is false; no matter what we add, we can never have B numbers.
- If the sum of the numbers in s is more than N, then it is false; no matter what we add, we can never have a sum of N.
- If there are exactly B numbers in s, and the sum is exactly N, then it is true; we have found a correct set. We should take note of the current set s and keep it as an answer.

What if it doesn't satisfy any of these conditions? Then:

- Suppose 1 isn't added yet. Is $f(s \cup 1)$ true? If so, then $f(s)$ must be itself true.
- Otherwise, suppose 2 isn't added yet. Is $f(s \cup 2)$ true? If so, then $f(s)$ must be itself true.
- Otherwise, suppose 3 isn't added yet. Is $f(s \cup 3)$ true? If so, then $f(s)$ must be itself true.
- $\vdots$
- Otherwise, suppose K isn't added yet. Is $f(s \cup K)$ true? If so, then $f(s)$ must be itself true.
- Otherwise, nothing we add could eventually lead to a valid solution, so $f(s)$ must be false.

These can be implemented quite straightforwardly using a recursive backtracking routine; see Listing 5.

If you actually code this and try some case with $K = 15$, however, you will find that it takes forever (and indeed it doesn't pass even the first subtask). It is hard to analyze the runtime of such solutions, but this solution, at least, is *very slow*. It is considering the same set in many different orders, and so is doing a lot of unnecessary recursion. It is potentially considering all 15! orders of a set of 15 elements, for example, which is very wasteful.

Instead, let's notice that the order of picking elements doesn't actually matter. Let's force our solution to pick only the elements in increasing order. So, if the last element picked is x, it only considers $x + 1$ to K as the next element. This is much faster.

Also, passing the entire vector by value is bad; we should pass it by reference instead, so we don't make a copy of the vector each time (which is expensive). We just need to remember to return the vector to its original state since it's now being shared across multiple function calls.

Listing 6 shows an implementation.

This now passes the first subtask; it only considers every set exactly once, so it is much faster. This demonstrates that even with our brute force solutions, some thought needs to go in to make sure that it will be fast enough.

The best way to learn how to code recursive backtracking solutions quickly and without bugs is, surprise, to practice coding them; don't underestimate the versatility of recursive backtracking solutions and their ability to net you free points!

```cpp
#include <bits/stdc++.h>
using namespace std;

int n, k, b;
vector<int> ans;

bool try_all(vector<int> s) {
    if (s.size() > b) {
        return false;
    } else {
        int sum = 0;
        for (int v : s) {
            sum += v;
        }
        if (sum > n) {
            return false;
        } else if (sum == n && s.size() == b) {
            ans = s;
            return true;
        } else {
            for (int i = 1; i <= k; i++) {
                if (find(s.begin(), s.end(), i) == s.end()) {
                    s.push_back(i);
                    if (try_all(s))
                        return true;
                    s.pop_back();
                }
            }
            return false;
        }
    }
}

int main() {
    int tc;
    cin >> tc;
    while (tc--) {
        cin >> n >> k >> b;
        vector<int> empty;
        if (try_all(empty)) {
            for (auto it = ans.begin(); it != ans.end(); it++) {
                if (it != ans.begin())
                    cout << ' ';
                cout << *it;
            }
            cout << '\n';
        } else
            cout << "-1\n";
    }
}
```

Listing 5: C++ recursive backtracking solution to Nyeh Heh Heh!

```cpp
#include <bits/stdc++.h>
using namespace std;

int n, k, b;
vector<int> ans;

bool try_all(vector<int> &s) {
    if (s.size() > b) {
        return false;
    } else {
        int sum = 0;
        for (int v : s) {
            sum += v;
        }
        if (sum > n) {
            return false;
        } else if (sum == n && s.size() == b) {
            ans = s;
            return true;
        } else {
            int last = (s.empty() ? 0 : s.back());
            for (int i = last+1; i <= k; i++) {
                s.push_back(i);
                if (try_all(s))
                    return true;
                s.pop_back();
            }
            return false;
        }
    }
}

int main() {
    int tc;
    cin >> tc;
    while (tc--) {
        cin >> n >> k >> b;
        vector<int> empty;
        if (try_all(empty)) {
            for (auto it = ans.begin(); it != ans.end(); it++) {
                if (it != ans.begin())
                    cout << ' ';
                cout << *it;
            }
            cout << '\n';
        } else
            cout << "-1\n";
    }
}
```

Listing 6: Better C++ recursive backtracking solution to Nyeh Heh Heh!

3 Number systems

3.1 Other bases

You are probably familiar with *binary numbers*. If not, simply recall that in our usual number system, the *decimal* number system, the "value" v of any particular number with $k+1$ digits $n_k n_{k-1} \dots n_0$ is simply:

$$v = 10^k \times n_k + 10^{k-1} \times n_{k-1} + \cdots + 10^0 \times n_0.$$

You might still remember your preschool days, where you would count in units, tens, hundreds, and so on, with each digit having a particular "place value". One might notice, however, that this choice of 10 as a base is quite arbitrary. Why not, for example, use 2, or 16 or even 31? This use of 10 as a base has become so ingrained from childhood that we often don't even realize we're using it.

In the *binary number* system, the only difference is that 2 is used as a base instead, thus, binary is also called "base 2". Now, the "value" of any particular number with $k+1$ digits $n_k n_{k-1} \dots n_0$ is simply:

$$v = 2^k \times n_k + 2^{k-1} \times n_{k-1} + \cdots + 2^0 \times n_0. \tag{1}$$

Note that, in this case, the digits n_i can only be either 0 or 1, unlike in the decimal number system where they can be from 0 to 9.

The 2's in there are in decimal, and the entire calculation will be computed "as usual"; like how we evaluate any expression. This will give us some decimal number v, which represents the "value" of some binary number in decimal.

Since we already tend to do all of our calculations in decimal anyway, this definition already allows us to "convert" any number from binary to decimal! We simply evaluate the expression to obtain v.

For example, consider the binary number[25] $(100111)_2$. Writing it out in the form above, we get:

$$\begin{aligned} v &= (2^5 \times 1) + (2^4 \times 0) + (2^3 \times 0) + (2^2 \times 1) + (2^1 \times 1) + (2^0 \times 1) \\ &= 2^5 + 2^2 + 2^1 + 2^0 \\ &= 32 + 4 + 2 + 1 \\ &= 39. \end{aligned}$$

This means that $(100111)_2 = (39)_{10}$; that is, 100111 in binary is 39 in decimal.

In general, in base b, the "value" v of any particular number with $k+1$ digits $n_k n_{k-1} \dots n_0$ is simply

$$v = b^k \times n_k + b^{k-1} \times n_{k-1} + \cdots + b^0 \times n_0, \tag{2}$$

where the digits $n_0, n_1, \dots, n_k$ are all between 0 and $b-1$. Note that we might have $b > 10$; in this case, the "digits" in base b might correspond to more than one "decimal digit". For example, the hexadecimal number system uses base $b = 16$; in this case, the digits can be up to 15. To avoid confusion, with, for example, using the "digit" 12 (which is two decimal digits, but just one hexadecimal digit), we normally represent $10, 11, \dots, 15$ with $\mathsf{A}, \mathsf{B}, \dots, \mathsf{F}$. Note that this choice of letters is completely arbitrary, and may differ in other sources.

Note that we only concern ourselves with **positive integer bases** b. There are a number of very interesting things that happen when we try other kinds of bases, such as fractional bases,

[25] We use the notation $(n)_b$ to denote a number n in base b.

negative bases, and so on—but they are rarely relevant in competitive programming, and even less so in the IOI, so we will not discuss them here. Instead, we leave it to the interested reader to discover these by themselves. For the rest of the discussion, when we refer to a base b we will take it to mean a positive integer base $b \geq 2$.

3.2 Base conversion

We already know how to convert any base b number to decimal; simply compute its value v using (2). But what if we want to convert from decimal to a base b number?

There are a couple of approaches we might use.

The first approach is a **greedy** approach. Notice that, for any base, all natural numbers have a unique representation in that base. Also, notice that $b^k \geq b^{k-1} \times (b-1) + b^{k-2} \times (b-1) + \cdots + b^0 \times (b-1)$; that is, any digit, assuming it is nonzero, will always have a greater value than the sum of all the values of the digits to its right.

This gives us a straightforward greedy algorithm to convert a number to any base b. Suppose we want to convert n from decimal to base b. Then, we should repeatedly find the largest k such that $b^k \leq n$, and "take as much of b^k as possible". This is because, if "we can still take one more b^k", and choose not to, then we will *never* be able to form n, because the maximum possible sum of all the values of the digits to the right is less than b^k, and so is also less than n.

To see this algorithm in action, let's convert $(51966)_{10}$ to hexadecimal.

1. Notice that $16^3 \leq 51966 < 16^4$. So, we know that the highest place value we can take is 16^3.
   - On the other hand, $16^3 \times 12 \leq 51966 < 16^3 \times 13$, so we know that we must have $16^3 \times 12 = 49152$, so $n_3 = 12$.
2. This leaves us with $51966 - 49152 = 2814$ left for the smaller digits. We have $16^2 \leq 2814 < 16^3$, so the next highest place value we can take is 16^2.
   - On the other hand, $16^2 \times 10 \leq 2814 < 16^2 \times 11$, so we know that we must have $16^2 \times 10 = 2560$, so $n_2 = 10$.
3. We have $2814 - 2560 = 254$ left for the smaller digits. We have $16^1 \leq 254 < 16^2$, so the next highest place value we can take is 16^1.
   - On the other hand, $16^1 \times 15 \leq 254 < 16^1 \times 16$, so we know that we must have $16^1 \times 15 = 240$, so $n_1 = 15$.
4. Finally, we have $254 - 240 = 14$ left for the last digit. We have $16^0 \leq 14 < 16^1$, which indeed confirms that this is the last digit.
   - On the other hand, $16^0 \times 14 \leq 14 < 16^0 \times 15$, so we know that we must have $16^0 \times 14 = 14$, so $n_0 = 14$.
5. We get $14 - 14 = 0$, and we are done.

Thus, we have $n_3 = 12$, $n_2 = 10$, $n_1 = 15$ and $n_0 = 14$. Using the convention of using the letters $\mathsf{A}, \mathsf{B}, \ldots, \mathsf{F}$, for the digits $10, 11, \ldots, 15$, respectively, we get $(51966)_{10} = (\mathsf{CAFE})_{16}$.

This approach is more suitable for "eyeballing" a conversion; that is, it is easy for humans to do it mentally, especially when the numbers to convert are rather small. For example, one can determine almost instantaneously that $(33)_{10} = (100001)_2$ without doing any calculations;

one simply has to notice that 32 is a power of two, then subtract 32 from 33 and be left with 1. This approach, however, can be rather tricky to implement in code, and might be more human error-prone when the numbers start to get larger.

The second approach is the more standard **mathematical** one, and is generally more widespread because of its ease of implementation and more "methodological" approach. You are highly recommended to use this method over the former, especially in your code, as it is shorter and more straightforward.

Suppose we want to convert n from decimal to base b. Notice that $n_0 = n \bmod b$. This is easy to see if you consider, for example, decimal—the last digit of 1841 in decimal is simply $n_0 = 1841 \bmod 10 = 1$. It is also true in other bases. For instance, the last digit of any even binary number will always be 0, which you can see because $n_0 = n \mod 2 = 0$ whenever n is even.

Now, divide n by b using integer division. This effectively "kills" the b^0 power, and brings down all the other powers from b^k to b^{k-1}. Now, we can extract $n_1 = n \bmod b$. We can do this repeatedly to extract all the digits, until n becomes zero!

To see this algorithm in action, let's again convert $(51966)_{10}$ to hexadecimal.

1. Compute 51966 mod 16 = 14. Hence, $n_0 = 14$.
   - Then, take $\lfloor \frac{51966}{16} \rfloor = 3247$.
2. Compute 3247 mod 16 = 15. Hence, $n_1 = 15$.
   - Then, take $\lfloor \frac{3247}{16} \rfloor = 202$.
3. Compute 202 mod 16 = 10. Hence, $n_2 = 10$.
   - Then, take $\lfloor \frac{202}{16} \rfloor = 12$.
4. Compute 12 mod 16 = 12. Hence, $n_3 = 12$.
   - Then, take $\lfloor \frac{12}{16} \rfloor = 0$.
5. Now $n = 0$, and we are done.

After this process, we also know that we have 4 digits. We can hence tell that $n_3 = 12$, $n_2 = 10$, $n_1 = 15$, and $n_0 = 14$. Hence the final result also turns out to be $(\texttt{CAFE})_{16}$, which is the same as what we computed earlier.

Notice how much simpler this algorithm is to describe and illustrate; if the numbers are too big to use the "eyeballing" method above, you are recommended to stick to this one, as it is harder to go wrong with it.

3.3 Computer number formats

You might wonder how computers represent numbers, particularly integers, internally. Although whenever you print numbers to the screen, input data into your program, or write constants in your program, you are using decimal numbers, it might be a bit surprising to find that internally, the computer uses a different system.

Indeed, computers store numbers, internally, in binary and not in decimal. This is an artifact of computer organization, which delves more into the hardware of computers (and is thus outside our scope). There are, however, some practical considerations. This is why some integer variables are "32-bit" and others are "64-bit", and this is why `unsigned int` can only

store values in the range 0 to $2^{32} - 1$, for example. This is because, if we give ourselves 32 binary digits, we are able to represent precisely the nonnegative integers up to $2^{32} - 1$.

You might now be wondering: how about *negative* integers? How are they represented in the computer?

3.3.1 Negative numbers

There are a number of ways one might represent negative numbers (particularly integers) in a computer. The most straightforward one would be the **sign and magnitude** representation. In this representation, we simply reserve one bit to be the "sign bit"; if our 32 binary bits are $b_{31}b_{30}\ldots b_0$, then just sacrifice b_{31}; let it be 0 if the number is positive and 1 if the number is negative. Now, we only have 31 bits, each of which has a positive and negative version.

For example, to write 13 as an 8-bit sign and magnitude number, we simply convert it to binary to get $(00001101)_{\texttt{s\&m}}$. However, if we wanted to write -13, we would just flip the first bit, and write $(10001101)_{\texttt{s\&m}}$. Hence, if we have b bits, our computer can now represent the whole range $-(2^{b-1} - 1)$ to $2^{b-1} - 1$.

If you have a keen eye, you would have noticed that this representation is a bit problematic, almost too simplistic to be useful. One, it is hard to do arithmetic with these numbers. While it is straightforward to add two positive binary numbers, it actually fails when we add negative numbers into the mix. For example, $-2 + 1 = -1$, but if we tried to add [26] the 8-bit sign and magnitude representations of these numbers, $(10000010)_{\texttt{s\&m}}$ and $(00000001)_{\texttt{s\&m}}$, we would get $(10000011)_{\texttt{s\&m}} = -3$, which is obviously wrong.

Second, there are two zeros; a positive zero and a negative zero. Both $(00000000)_{\texttt{s\&m}}$ and $(10000000)_{\texttt{s\&m}}$ are zeros in the 8-bit sign and magnitude representation, and this can again cause more complications. For example, when we compare if two numbers are equal, do we now have to make a special exception for zero?

Because of these two issues, sign and magnitude is not used in modern computers.

The **ones' complement** of a binary number represents negative numbers by taking the binary representation of the positive number and flipping all the bits. For example, the 8-bit ones' complement representation of -39 can be found by writing out 39 in binary, as $(00100111)_2$, then flipping every bit, to get $(11011000)_{\texttt{ones'}}$. Note that, like sign and magnitude, we can still use the first bit to determine whether or not a number is negative.

It turns out that the math *usually checks out* when adding numbers in ones' complement representation. For example, in our earlier example, if we were to add -2 and 1 using 8-bit ones' complement representations of these numbers, we would get $(11111101)_{\texttt{ones'}} + (00000001)_{\texttt{ones'}} = (11111110)_{\texttt{ones'}}$, which is indeed -1.

There are still some peculiar issues, though. When there is an overflow bit, that is, a carry bit comes out of the most significant bit, we need to do add the bit back in (end-around carry) to keep the result correct. For example, if we were to add 5 and -2 instead, we would have gotten $(00000101)_{\texttt{ones'}} + (11111101)_{\texttt{ones'}} = (00000010)_{\texttt{ones'}}$ with an overflow bit. We need to add this bit back in to get the correct result of $(00000011)_{\texttt{ones'}}$.

Also, it does not solve the issue of having two zeroes at all, because $(11111111)_{\texttt{ones'}}$ is still a negative zero of the 8-bit ones' complement representation.

Luckily, it turns out that there is a quick fix to both of these two issues.

Enter the **two's complement** representation. The two's complement representation of a negative number n is simply the ones' complement representation of the number $n + 1$. So,

[26] For brevity, we did not discuss how one might add two numbers of arbitrary bases, but it is very similar to decimal addition. The Wikipedia article explains it in more detail.

-5 in 8-bit two's complement can be found by taking -4 in 8-bit ones' complement, which is $(11111011)_{\texttt{ones'}}$. Hence, -5 is $(11111011)_{\texttt{two's}}$.

This elegantly solves the negative zero problem; now, $(11111111)_{\texttt{two's}}$ is -1. It removes the redundancy and hence affords us one more value, -128, which is $(10000000)_{\texttt{two's}}$ which was previously unrepresentable in the sign and magnitude and ones' complement systems.

Additionally, it also removes the awkward end-around carry issue. Now, overflow bits can simply be ignored. If we add 5 and -2 in two's complement, we would have $(00000101)_{\texttt{two's}} + (11111110)_{\texttt{two's}} = (00000011)_{\texttt{two's}}$ with an overflow bit which is simply ignored; the result is 3, which is correct.

Indeed, the two's complement representation is so elegant and magical that it is no surprise that it is the preferred representation in modern computer systems. This explains why a 32-bit signed integer, for example, can represent the values -2^{32} to $2^{32} - 1$; this is because they are represented using the two's complement representation.

4 Bits and pieces

All the above is well and good, but in what ways might knowing how numbers are stored internally be useful to us in the context of competitive programming?

There are certainly a couple of instances when having this knowledge is useful. Let's consider the following *classic* motivational problem:

Problem 4.1. You have a set of N integers. Determine whether there exists a non-empty subset of these N integers whose sum is zero. Assume N is up to 20, and the integers can be up to 10^9 in absolute value.

This is the classic **subset sum problem**; all you should know about this problem is that there is currently no known "fast" solution for it, and so in most cases some sort of brute force is your only solution.[27]

How might one formulate this brute force?

4.1 Binary representations as subsets

One approach is to use the fact if we consider every integer from 0 to $2^N - 1$, each of these integers can be mapped to exactly one subset of the N elements[28].

How do we do this mapping? Consider, for example, $N = 3$. There are $2^3 = 8$ subsets of 3 elements. Now, consider the binary representations of the first $2^3 - 1 = 7$ nonnegative integers:

$$
\begin{aligned}
(0)_{10} &= (000)_2 \\
(1)_{10} &= (001)_2 \\
(2)_{10} &= (010)_2 \\
(3)_{10} &= (011)_2 \\
(4)_{10} &= (100)_2 \\
(5)_{10} &= (101)_2 \\
(6)_{10} &= (110)_2 \\
(7)_{10} &= (111)_2
\end{aligned}
$$

We've added leading zeros to some of the binary representations for clarity.

Notice that, if we look at the binary representations, we can easily deduce a one-to-one relationship between them and subsets of the 3 elements. Let every bit represent an element. Now, a 0 bit simply represents that the element is not taken, and a 1 bit represents that the element is taken.

Traditionally, we associate the ith element with the ith bit *from the right*. This way, we can easily say that an integer s corresponds to the subset where the ith element is included if and only if the coefficient of 2^i (i.e. n_i) in the binary representation of s, as in (1), is 1. (Note that we are using zero-indexed elements here.)

Hence we can reduce the problem to: how do we extract the ith bit of some number s in binary?

[27] Note that there is also a dynamic programming solution if the integers are small, but that's not what we want here. Dynamic programming will be covered in more detail in later weeks.

[28] Integers, but we use "elements" from now on to avoid confusion.

4.2 Bit tricks

Obviously, one way to do that will be to actually write a decimal to binary converter using one of the two algorithms we've discussed earlier, and convert the numbers explicitly. But that is tedious, and adds huge constant factors to our already bad "brute force" algorithm.

Instead, let's exploit the fact that numbers are already represented internally in binary anyway. While neither C++, Java nor Python have a built-in "get ith bit in binary" function, they do have some bitwise operators that let us concoct such a function ourselves.

For example, C++ offers us the following bitwise operators:

1. Bitwise AND, which gives the result if the AND operator ($\wedge$) was applied to every bit of two given integers.
2. Bitwise OR, which gives the result if the OR operator ($\vee$) was applied to every bit of two given integers.
3. Bitwise XOR, which gives the result if the XOR operator ($\oplus$) was applied to every bit of two given integers.
4. Bitwise XOR, which gives the result if the NOT operator ($\neg$) was applied to every bit of a given integer.
5. Bitwise shift left, which shifts all the bits in the binary representation by a given number of bits to the left.[29]
6. Bitwise shift right, which shifts all the bits in the binary representation by a given number of bits to the right.

Java and Python offer similar operations, but the details may be a bit different. We highly recommend looking through the details of these operators in your favorite language and being highly intimate with them to avoid being burned when using them during an actual competition.

An illustration of these bitwise operations in C++ is shown below:

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    cout << " (5 & 6) = " <<  (5 & 6) << '\n'; // 5 AND 6
    cout << " (5 | 6) = " <<  (5 | 6) << '\n'; // 5 OR 6
    cout << " (5 ^ 6) = " <<  (5 ^ 6) << '\n'; // 5 XOR 6
    cout << "    (~7) = " <<     (~7) << '\n'; // NOT 7
    cout << "(5 << 1) = " << (5 << 1) << '\n'; // 5 SHIFT LEFT 1
    cout << "(5 >> 1) = " << (5 >> 1) << '\n'; // 5 SHIFT RIGHT 1
}
```

Listing 7: C++ bitwise operators

Run this code on your local machine, and find out what results are printed, and, in particular, *why* those results are printed.

Let's go back to our problem. How could we "extract" the ith bit of some number s?

[29] A word of warning: care must be used when doing bit shifts, as in some cases, an arithmetic shift will be used, and on others, a logical shift will be used. Sometimes, this behavior can even be undefined. We should prefer sticking only to "obvious" use cases for this reason to avoid nigh impossible to detect bugs.

Well, notice that `1 << i` gives us a single bit shifted to the right i times. If we take `s & (1 << i)`, then if the ith bit of s (from the right) is not 1, then this whole thing evaluates to zero; otherwise, it evaluates to precisely 2^i (or `1 << i`). So we can now easily "query" whether a particular bit in the binary representation of some number is 0 or 1!

Thus, we can now code the brute-force solution to this problem:

```cpp
#include <bits/stdc++.h>
using namespace std;

long long int arr[25];
int main() {
    bool found = false;
    int n;
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> arr[i];
    // do not include the empty subset s = 0
    for (int s = 1; s < (1 << n); s++) {
        long long int sum = 0;
        for (int i = 0; i < n; i++) {
            if (s & (1 << i))
                sum += arr[i];
        }
        if (sum == 0) {
            found = true;
            cout << "A zero-sum subset exists:";
            for (int i = 0; i < n; i++) {
                if (s & (1 << i))
                    cout << ' ' << arr[i];
            }
            cout << '\n';
            break;
        }
    }
    if (!found)
        cout << "No zero-sum subset exists.\n";
}
```

Listing 8: C++ solution to subset sum problem

By the way, this technique of using the binary representation of integers to represent subsets is an application of a well-known technique called bitmasking, where we manipulate the binary representation of numbers, usually to have them represent something, in this case subsets. You will encounter other uses of this in the future; for example, bitmasks can represent the states of items in dynamic programming algorithms.

Note that this algorithm is *very slow*. It examines all $\mathcal{O}(2^N)$ subsets, and for each subset does around $\mathcal{O}(N)$ operations, so we would expect this to take around $\mathcal{O}(N2^N)$ time. In this case, however, N is only up to 20, so this is just $20 \times 2^{20} \approx 2 \times 10^7$ which should pass in a typical competitive programming setting.

4.3 Meet in the middle

Let's make the problem a bit harder. What if N is up to 40? Our previous algorithm will surely fail; there are up to $2^{40} \approx 10^{12}$ subsets to examine, so it will be too slow.

Notice the following, however. Suppose that we split the set into two sets, the first one containing the first $N/2$ elements and the second containing the rest. (If N is odd, throw the last element in either one.) Now, consider the set of all possible sums s_1 formed by non-empty subsets of the first $N/2$ elements, and the set of all possible sums s_2 formed by non-empty subsets of the latter $N/2$ elements. It is clear that for some sum $x \in s_1$, it should be paired with some sum $-x \in s_2$ to get a sum of zero.

But suppose we sorted all the elements in s_1, and sorted all the elements in s_2. Then finding this other element is as simple as doing a *binary search*! Note that both s_1 and s_2 each contain around $2^{N/2}$ elements; we know how to sort in $\mathcal{O}(n \log n)$ time, where n is the number of elements, so in our case, sorting takes $\mathcal{O}(2^{N/2} \log 2^{N/2}) = \mathcal{O}(N2^{N/2})$ time, which is quite reasonable. Then, for every element $x \in s_1$ (of which there are $2^{N/2}$ of) we need to binary search for the corresponding element in s_2, which takes $\mathcal{O}(\log 2^{N/2}) = \mathcal{O}(N)$ time. So the total runtime of this solution is just $\mathcal{O}(N2^{N/2})$, which is loads faster than $\mathcal{O}(N2^N)$ and suffices for even this harder variation.

Note that there is a minor special case to consider when implementing this solution; particularly, if $x \in s_1$ is already 0, then we don't need to find a corresponding sum in s_2, and vice-versa.

The approach of splitting the problem size into two parts that can somehow be combined is called meet in the middle and has many interesting applications beyond this one.

5 Exercises

For the programming exercises, simply link to your corresponding submission.

For the short response items, no need to be overly formal :)

Programming exercises

1. [100★] Solve A Game of Numbers.
2. [100★] Solve Apt. 102.
3. [100★] Solve Clementi.
4. [100★] Solve Papers, Please.
5. [100★] Solve Serena and Desserts.
6. [100★] Solve Start the Day on Wings.
7. [100★] Solve Trick or Treat?.

Short response exercises

8. [10★] Write $3n^3 + 49n^2\sqrt{n} + 200n$ in big $\mathcal{O}$ notation with the lowest complexity.
9. [25★] Determine the runtime of the following algorithm in big $\mathcal{O}$ notation with the lowest complexity:

```cpp
#include <bits/stdc++.h>
using namespace std;

int arr[100005];

// Given an array, determine whether there exist two distinct elements summing
↪  to x
int main() {
    int n, x;
    cin >> n >> x;
    for (int i = 0; i < n; i++)
        cin >> arr[i];
    sort(arr, arr+n);
    bool found = false;
    for (int i = 0, j = n-1; i < n; i++) {
        while (i == j || (j >= 0 && arr[i]+arr[j] > x))
            j--;
        if (arr[i]+arr[j] == x) {
            found = true;
            break;
        }
    }
    cout << (found ? "yes" : "no") << "\n";
}
```

10. Consider the Quicksort algorithm as implemented in Algorithm 1. In particular, consider the line "$p \leftarrow$ random integer from 0 to $|a| - 1$".

(a) [10★] Replace that line with "$p \leftarrow 0$".
Now, design an array of length n such that this Quicksort takes $\mathcal{O}(n^2)$ time.

(b) [50★] Replace that line with "$p \leftarrow \lfloor \frac{|a|-1}{2} \rfloor$".
Now, design an array of length n such that this Quicksort takes $\mathcal{O}(n^2)$ time.

11. [15★] Consider $n = 39^{39^{39}}$ (in decimal). What is the last digit of n when expressed in binary?[30]

12. Consider the base 36 system, which uses the letters `A` through `Z` for the digits 10 through 35, respectively.

(a) [15★] Write down the decimal value of $(\texttt{NOIPH})_{36}$.

(b) [25★] Write down the base 36 representation of $(1630603463)_{10}$.

13. Consider how the negative integer -42 might be represented differently in different computers.

(a) [10★] Find the 16-bit sign and magnitude representation of this number and write it down in hexadecimal.

(b) [15★] Find the 16-bit ones' complement representation of this number and write it down in hexadecimal.

(c) [20★] Find the 16-bit two's complement representation of this number and write it down in hexadecimal.

14. [5★] What six integer results[31] are printed by Listing 7?

15. Consider the subset sum problem described in section 4.

(a) [50★] Describe an $\mathcal{O}(2^n)$ algorithm to generate the sums of all subsets of a list of n integers in sorted order.

(b) [30★] Describe an $\mathcal{O}(n)$ algorithm that, given two sorted lists a and b each of length n, determines whether or not there exists some element in a equal to some element in b.

(c) [20★] Hence, describe an $\mathcal{O}(2^{n/2})$ algorithm for the subset sum problem.

[30] Note that a^{b^c} is evaluated as $a^{(b^c)}$ and not as $(a^b)^c$.
[31] integers appearing on the right hand side of the equations