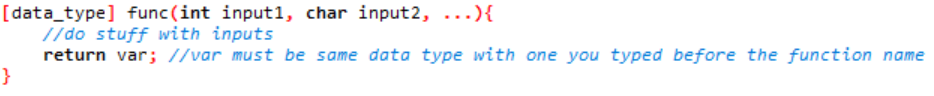
**Functions**

In mathematics, functions are things that accept inputs and produce outputs. Basically the same in OI. You can think of it as tiny programs in your cpp file.

syntax:

return data type can also be void, then it will return nothing

cannot return more than one thing, but you can return pairs, vectors and other containers. (Beware it needs to copy the whole result over to the main program so if you return a big big vector it may TLE)

example:

<https://pastebin.com/vsQ02cD3>

Purposes:

* Reuse functions so you do not need to copy blocks of code
* Easier to debug as it is only one thing
* R E C U R S I O N

Advanced stuff you can google:

c++ lambda functions, c++ pass by reference

**Recursion**

Can you call other functions in your function? Yes!

What if you call the same function in your own function?

First of all, it is possible to call the same function in your own function.

But it doesn’t make any sense!

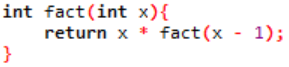
Many people think that functions are fixed things with clear steps on what to do with the input variables, so it may be a little bit difficult to understand.

**Subproblems and base cases**

The most important part when writing a recursive function is to identity the subproblems and how to reduce the current problem into subproblems.

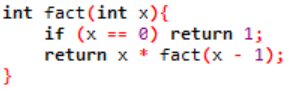
Example 1: factorial

Sure, you can write a for loop, but why not use recursion?

Wow! We have found a method to reduce it to a subproblem our function can solve!

Does this work? No it would fall into an infinite loop.

We need to make a base case so it can know when to stop.



Yay we have finished out first recursive function.

It is easy to see that it has the same time complexity.

good: easy to write

bad: maybe tricky to think, repeated function calls slows down runtime a little bit

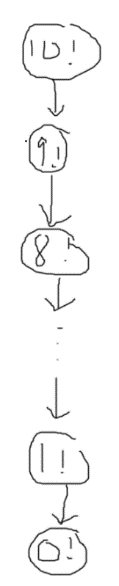
**Ways to think**

Method 1: Black box

* Treat the function as a ‘black box’
* trust it
* trust it that it works
* trust that it will solve the subproblem correctly

In the last example, we ‘trusted’ the function to return the correct value for , so we only needed to reduce the problem.

So debugging a recursive function: your reduction to subproblems is wrong, or your subproblems are fundamentally wrong

Method 2: tree diagram

Draw the subproblems out

Sometimes makes it easier to analyse time complexity

**Reducing time complexity**

Recursion is very useful because clever reductions may speed up the program!

Warning: big brain stuff

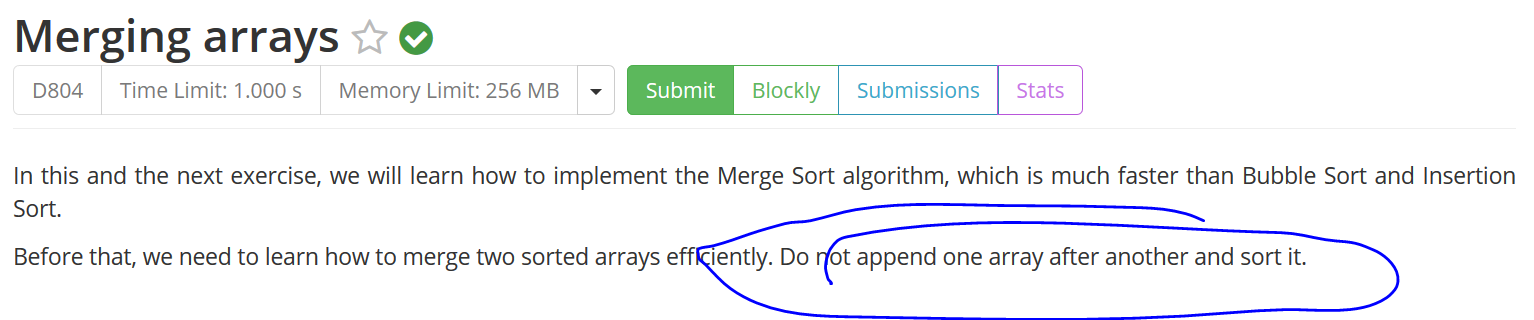
Example 2: merge sort

You want to sort n numbers. How can you reduce it into subproblems?

There are a lot of ways, but the simplest way is just split the array into 2 arrays each with n / 2 numbers.

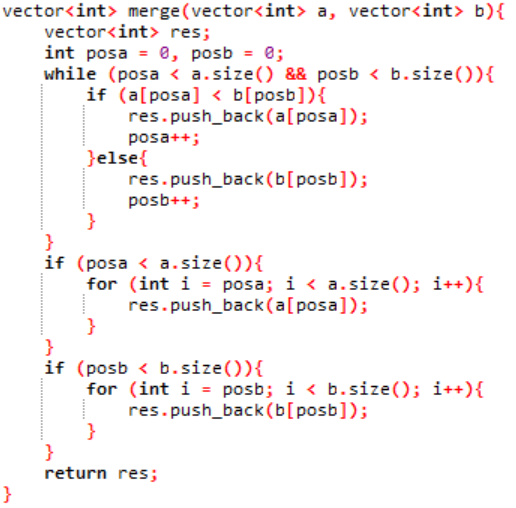
Now, through recursion we have sorted the first and second half of the array (black box)

We need to find a way to combine 2 sorted array such that the final array is also sorted!



Hmm the problem says not to sort it. As you are a good boy of course you won’t do that.

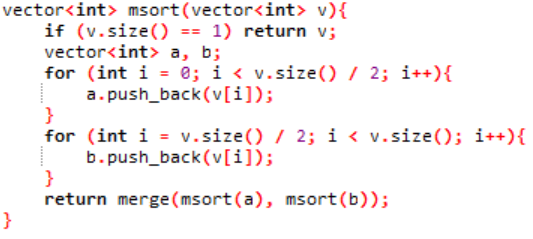
How can we take advantage of the fact that the 2 arrays are sorted?

We can keep 2 pointers!

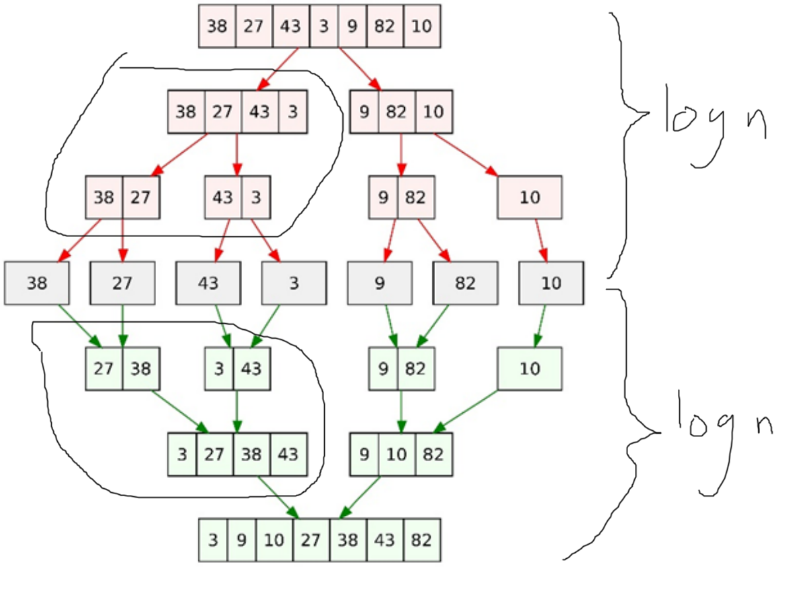
Maintain 2 variables posa and posb. if a[posa] is smaller than b[posb], push a[posa] into the final array, else push b[posb] (increment posa or posb afterwards of course)

Don’t forgot remaining elements.

It is obvious that the time complexity of merging 2 arrays with size n and m is O(n + m).

Don’t forget about the base case! if the array only has 1 element it is already sorted.

Implementation of merge sort:

We have used the ‘black box’ method to design the algorithm. Now let’s use the tree diagram to analyse the time complexity.

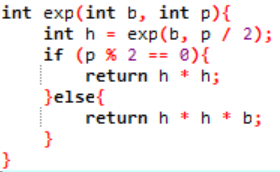
As you can see, every step you split or merge the array into halves. That means you repeatedly divide it by 2, so if you have x layers at the end, that means you divided it into around 2x arrays. Therefore, the number of layers is around . (In OI if not specified its base 2)

To split an array with length len, it takes O(len) time. To merge 2 arrays with total length len, it also takes O(len) time.

Now, in every layer, the total length of all the arrays is n. There are 2 log n layers, so the total time complexity if O(n log n).

Example 3: binary exponentiation

Let’s say we want to calculate . You may say, it takes multiplications so the fastest time complexity should be . WRONG

Again, find subproblems.

If we know , then is just !

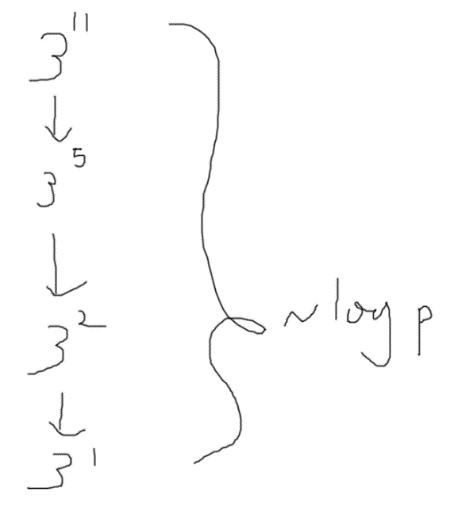
Therefore, 2 cases:

if is even, calculate , return

if is odd, calculate , return

Oh no! it doesn’t work

You forgot base case!

if p is 1 just return b. (or return 1 when p is 0)

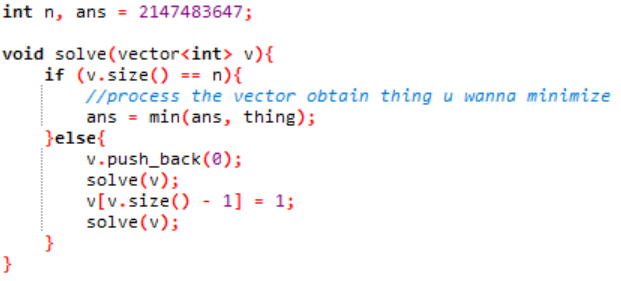
Let’s draw the tree diagram.

Each call we only solve one subproblem, so it’s a line.

Each time we divide p by 2, so time complexity if O(log p)!

Example 4: brute force

Recursion can also be used to brute force stuff. For example, let’s say a problem wants us to find a binary sequence with length n such that something is minimized.



**How to think of solution**

You may also have heard of ‘divide and conquer’. Recursion is a concept where a function calls itself, and divide and conquer is a technique where you divide the main problem into subproblems so you can ‘conquer’ them using recursion, and finally combine them.

1. Think of the subproblems (most cases like to divide by 2)
2. Think of how to combine the solved subproblems
3. Find the time complexity (if you can)

Recursion is a very important concept that is used almost everywhere from advanced data structure to graph traversals, so you must understand it. Ask questions in the group (or pm me) if you are still unclear about some stuff!

Advanced and interesting stuff: Karatsuba multiplication, master theorem (a formula to calculate time complexity of recursive algos)

Practice: D804, D805, D806, 20374, 01003, S002, 01031, 30098, 01037, 01048, S163, 01046