Algorithms and Data Structures

Lecture notes

Pavel Mavrin

August 17, 2022

An Awesome Publisher

Preface

Pavel Mavrin

Contents

1 Introduction

1.1 What is an Algorithm?

In this section we'll start from very simple things. We'll discuss what is an algorithm and how to measure its efficiency. So what is an algorithm? Algorithm is a formalized description of a solution to some problem. Basically, you take the solution to some problem, and split it into some elementary steps.

Algorithms are used in many different areas. In this course we will mostly talk about data processing algorithms. That means, the algorithm will take some data as input, processes that data somehow, and produce some data as a result.

PICTURE

If you're familiar with programming contests like Codeforces[1], it's basically how most of the Codeforces problems work: in each problem you read some input data, solve the problem, and output result. That's how most of the algorithms in this course will look like.

1: Codeforces. Programming competitions and community. https://codeforces.com

For example let's consider the following problem. You have an array of n integers, and you need to find their sum. Here the input data is the array of integers, and the output data is the calculated sum.

You probably solved this problem before, the algorithm looks something like this.

```
input: array a
s = 0
for i = 0 .. n - 1:
    s += a[i]
print s
```

Here we described the algorithm using some *pseudocode*. Pseudocode is a language used do describe algorithms. It usually looks like programming code, but it much less formal, because it meant to be read by people, and not by computer. Scientists may use many different pseudocodes, the idea here is to keep balance between formality and readability.

The pseudocode we used above looked mostly like Python, but not exactly. For example, the `for` loop looks different (more like in Ruby), just because it's easy to read it this way.

1.2 Good and bad algorithms

There may be multiple algorithms that solve the same problem. Some of them might be better than another. How to find out if the algorithm you use is good or bad? The main characteristic used to measure the quality of algorithms is *time complexity*. Time complexity of the algorithm

is basically the time that your algorithm spend to solve the problem. The lower it is, the better.

Let's think, what units will should we use to measure the time complexity. In physics time is measured in seconds. Why can't we measure time in seconds when we talk about algorithms? Because different processors have different speed, the same algorithm may spend different time on different processors[2]. So instead of physical time we will measure the logical time, in the *number of operations*.

2: Even if you fix the real processor, for example let's measure time complexity on Intel Core i7 processor, the time spend by your algorithm may be different because it depends on so many details of your hardware and environment

Now to measure the number of operations we need to fix the set of elementary operations. For example,

like uh like what operations are here i don't know maybe this one operation maybe it's 10 operations maybe not so to measure number of operations you need to fix the environment you work with so this environment is called a computational model so computational model it is a mathematical abstraction uh uh that you use to calculate the time complexity and not only time complexity memory complexity and so on so you might you analyze your algorithm using uh some but some computational model it's a mathematical model uh that works kinda like a real processor but not exactly like very simplified version of the processor and there are different computational models uh and we will discuss different models during this course a very basic computational model which is like the most common one it's called the ram model so it's ram machine or rare model

ram model is basically the model which simulates the usual processor you have in your computer so it's somehow the simplified version of the regular processor so how it works you have some

memory now let's ask here that does anyone know what what is ram stands for so

so these letters what are these stands for hmm

render maximum good so simple random access memory what does it mean it means that you have memory and you can access any cell of that memory in constant time so let's see if this is our memory so this is memory

uh we will think about memory like one big race so we have one bigger a it's split in some cells so each cell have some address so that's the basis from zero uh let's say to minus one like m is the total memory so in one operation you can get any element from your memory so you can

by this number i you can go to memory and take this value from the memory and you can put value in the memory so in one operation you can take any any element from the memory or put any element to the memory so this cost one operation ah and this this is the most fundamental stuff about remembering not not all computational models have these uh operations to the memory so in some different models you may not have access to any element in constant time but in this model you have uh now uh beside this this this memory operations you have some regular operations like you have all the let's say arithmetic operations you have cycles ifs and so on so all all usual operations you have in your programming languages we will not specify that exactly but it's kind of intuitive so e

if if you are good enough so actually if you very very very precise when you specify the model you need to specify all the types of operations you you can use in your program but in real model you just can use any operations you have in any like regular programming language so we'll not specify this exactly but we will just think about like a usual program you have in your favorite programming language

uh uh-uh

i don't know what's buffering if it is buffering maybe the video quality is too big i'm not sure how twitch works not sure about this

my channel seems good so i if it's buffering it's something your side i think

good now

we have this computational model let's go back to this algorithm and calculate its time complexity so time complexity again it's the number of operations your algorithm spent to solve the problem uh this number of operations depend on the input yeah so if if this array is big then the time will be bigger so but when you increase the size of the array the time will be increased as well so time you already spend is some function of input

let's see so if we have array of size n so we have n elements in the ring what will be the total number of operations here

let's go so uh what is the total number of operations so here we have one operation to to initialize the variable s

here we have cycle what number operations we need to to go to the next iteration of the cycle let's say we have here look it may be different again some detail so here we need to increase the value of i then compare i to n check if i is less than n and then go solids let's say two operations per cycle two elevations spent here

uh here we have some plus equal it means we need to load this element from the array then calculate this sum and then put this sum into variable s so let's say we have like three operations per cycle so on each iteration we spend three operations here um so total number of operations we spend here will be free and

something like that and finally we need one operations to output

so the total time complexity of this algorithm will be the function of n

and it will be like 1 2 plus 5 m

3n it's just maybe we need to load this from array then calculate this sum and then put this sum into variable s back so we need like about three operations it's not it's not very important we'll talk about later

yeah so when we calculate a number of operations we have something like that so this is some function which explains how how the time grows when you increase the size of the input

now let's take this formula and remove everything which is not important

let's look for example this plus 2 is not important why it is not important because this 5n is much bigger than this 2.

we will talk about master theorem in the end but not not now this five n is much bigger than this two so this two is not very important let's remove it

now let's look at this 5n in this 5n this constant 5 is actually not very important why it is not important because it actually depends on very different details so these five may be different if we fix another set of operations so if let's say we have operation plus equal in one operation we'll have node three and but n here so it will be not 5 and but say 3n and so on so so this constant 5 is not the property of the algorithm is property of its realization so if we make this cycle in very in some different way we will we may have different constant here so this constant is not the property of the algorithm it's just how good we're in programming so it is not proportional so we

but this n this n is important because this n is a property of algorithm so if you make slightly different cycle and like calculated in different in different order and so on you still will have some function which has this n you may have different constant here different constant here but this n stands the same and to describe situations like this there is a special special theory so it's written like this is big o

ah you may see this so again if you take part in programming competitions and you read some tutorials you might be familiar with this uh sign big o so what this big o stands for uh because this stands for this following so it's just the short

symbol to describe the following situation so if one function is big o of another function

uh what does it actually mean it means that after some value and zero i will use some pointers here so it means exists so so there exists some constant m0 and some constant c

such that after point and zero so if so for all n greater equal than n0

this function f

is no more than c multiplied by function

again this is the formal definition of this big o sign so when whenever we use this big o sign we actually mean this so for example

i don't understand what's going on

okay i don't see any questions here again

if you have any questions just post them in the chat so don't spam too much but if you have questions just post it immediately

let's prove that this function is actually big o of n let's do it so let's let's do it here

so we want to prove that 2 plus 5 m is big o of n

how to prove this uh we need to find some constants n 0 and c such that this holds

okay any guesses

try to find values of n 0 and c for these functions again here function f is 2 plus 5 n and function g is simply n and we need five find constant n0 and c such that for big number values of n f of n like this function is no more than c multiplied by j

mhm

come on any guesses so we need to find where you on zero and value of c

five forward variables we have two constants and zero and c

five is a good number but for what for what constant

ah c equal to five and equal to one let's check and one is equal to five

let's check so we need to prove that 2 plus 5 n is less or equal than constant c which is 5 multiplied by function g which is n

that's obviously not true so 2 plus 5 n is greater than 5 n so this pair of constant just doesn't work

n equal to pi of c equal to 2 now if if you have c equal to 2

then you have constant 2 here

that's even worse so now you have 2 plus 5 n less than 2 n but if n is big then this is actually much bigger than this so again it's not working c equal to 6 is much better so if c equal to 6

let's see so if if we take c equal to 6 then we have this integration in this equation uh let's just uh decrease so let's move it to the right remove this we will have something like angle two so if we take n zero equal to

two

we will have the true equation so again if n is greater or equal than 2 then we have this inequality this 2 plus 5 n is less or equal than 6 n

so if you take these two constants

you have the correct proof or you can oh yes you you can take one one and seven that's so that's all it also works constants one and seven also works

okay now we proved our first time complexity so 10 complexity of this algorithm is big o of n good now let's move to some difference so um what i want to talk about so ah this big orientation is actually the upper bound of the complexity so it's not the real value of complex but upper bound for example let's prove that n is big o of n squared

uh let's prove this now it's easy so for example let's take n zero equal to one c equal to one so we have n is less or equal than n squared uh that is true if n is greater or equal than one so if n is greater or equal than one then n is less original than n square so formally n is big o of n square

so this big or notation shows not the real asymptotics but the upper bound of this asymptotics so when you when when you prove that your algorithm works in big o of n time you prove that your algorithm is not slower than the girlfriend

sometimes so when you want to prove your algorithm is fast enough you say it you prove that your algorithms works working time is big o of

some function sometimes you want to prove that your algorithm is not not fast enough so you want to show that that time complexity is big

for these cases there is some other notation annotation is not big o but big omega so sometimes we need to use like this big omega

big omega is the same thing but not less or equal but greater or equal so it is again uh there exists some constant n 0 and c so for all and after this f of n is greater or equal than c multiplied by

so this big o is a upper bound for the complexity and this big omega is the lower bound of reflex of the complexity for example let's show for the same function

that the time complexity of this algorithm is big omega prime

again let's get guess some numbers and zero so we need to fix numbers and zero and and c such that 2 plus 5 n is greater or equal than some constant multiplied by n any guess again

very simple

one and one good one and one works so if you have one here one here then you you need to prove that two plus five n is greater than n that's obviously true cool so now we prove that the time complexity

is big o of n and the time complexity is big omega and when you have situation like this like you have uh uh the same upper and lower bound on the complexity uh they have a special symbol for this so when we have these two equations then you have c of n equal to big data

so that's a special sign which which means that you have lower button bound asymptotic bounds are the same

cool so if you read some special articles about algorithms then usually when someone invents some new algorithm they try to not only prove their upper bound but also proved lower bound so the real result of the real article is usually something like that

but in this course i i will almost never use lower bound i will usually i will prove only the upper bound of the complexity just because to make the course more simple because it it is more important to prove the upper bound when you prove again when you prove the upper bound you show that your algorithm is fast enough when you prove the lower bound

uh you show that your algorithm is not is

not fast enough so so it's slow enough

and usually when you design algorithm you want to prove your algorithm is good so to prove your algorithm is good you need only upper bound

so in this course i will mostly always prove only the upper bound but in in real world you

in most cases you want to prove both both these bounds just if you're if you're writing the real article you want to prove all the both bounds and get something like this good

now how to how to how to calculate this number of operations for your algorithm there are different cases so

so sometimes it's very simple for example if you have like two cycles you have cycle i from zero to n minus one and then inside you have cyclone trade

what is the time complexity

l squared so time complexity this algorithm is n squared

uh that's quite trivial because this line is executed exactly n squared time so we have n square time executing this line uh if you have something more complicated let's say we have not not to n minus one but to i minus one here

uh if you have something like this let's again we we can just calculate the number of times we execute this line uh this line will be executed so on the first iteration is good once then you have two times and so on so the number of times you visit this line is like one plus two plus and so on plus n minus 1 and minus n n right no n minus 1

because it is from 0 from 0. so on that when i equal to 0 you have 0 iterations here good so this sum is actually n multiplied by n minus 1 divided by 2

so this is again it's big o of n square

so when you have four cycles it's usually simple so when you in in your algorithm you have only four cycles four cycles are very easy it's little bit more complicated when you have wild cycles and usually you have some of them so for example let's say let's say we have i equal to zero and while uh i multiplied by i is less than n i plus plus

what will be the time complexity here

so in this algorithm you increase the value of i until i square will be greater or equal than n

yeah good so the number of iterations so you will end this cycle when i square so you end the cycle when i square will be greater or equal than n means that i will be greater or equal than square root of n and each time you increase the value of high so the total time you in you increment the value of i will be up to square root of n so complexity of this order will be square root of n good now let's make something

let's say let's make something like this

let's say we have equal to one and while y less than n multiplied by

two

so we just increase the value of i twice on each iteration until i will be greater or equal than n what is the time complexity of disorder

go here we have a look again we will end this cycle then two in the power of uh if we have if if we make k iterations let's see

iterations

i will be equal to n power of k so the number of iterations will be such that 2 n power of k is greater radical n it means that k will be equal to logarithm of base 2. so the total number of iterations will be up to a logarithm of friend so time complexity will be like this

good now when you have logarithms in your asymptotics i i will usually not write this base of the logarithm so i will usually write something like this

uh that's basically because this constant doesn't affect the asymptotics so if you put any other base here it only affects the constant not asymptotic so for example

so we have something log base a of n is big o of log base b of n for any a and b greater equal greater than one

so all logarithms only changed by their constant so they all grows in the same speed so when i will write in the future when i will write uh some big o of some logarithm i will adjust i need this base of the logger cool

yeah nice timing so next

now let's talk about recursive algorithms

some algorithms we'll talk about even in this lecture are recursive so they have some function which calls itself so let's write something

ah for example let's say we have f of n like in python

so we have some function

and it do something like this up let's say if n equal to zero then just

and if it is greater than zero then we will call f of n minus one so what happened if we call f of n here now to calculate the time complexity of the recursive algorithm to calculate capacity of recursive algorithm we need to calculate the number of times you call the recursive function and for each call of recursive function add the time you spend when you make this recursive function so now each call of the recursive function works in constant time we don't have any cycles here so this all works in constant time so we said this this all works in constant time so to calculate the total complexity into total number of operations we need to calculate the number of times we call this function so what will be total time so when we call f of n

what will happen we call f of n it will call f of n minus one

it will call f of n minus two

and so on until we call f of zero

so the total number of recursive calls will be m so we have and recursive calls and each recursive call costs you constant number of operations so the total number of operations will be linear

it is clear enough so again if you have any questions just ask them immediately good

now let's make something more less real uh if you have let's see

here if you have not n minus one but let's say and divide by two

what will be let's say this integer division so you you divide it like integers just without the remainder

here again what happened you call f of n now from f of n you call f of n over 2 from this you call f of n over 4 and so on so until you equal f of 0

and when you make integer divisions each time the total number of divisions you can have until you reach 0 will be up to logarithms so

this depth of recursion will be logan

let's end base two just for clearing this up yeah the total number of operations here will be up to logan okay and now finally let's make something more interesting let's add more here so let's just uh let's have two recursive calls on cover two and

another now what will happen here

now it's a little bit more more complicated now you have not the sequence of recursive calls but some trigger recursive calls so you have some main recursive call f of n

from this call you have two independent recursive calls so you call f of n over two and then again call f of n over them

now each of these recursive calls will also make two recursive calls so from here you have call of n over four and again

and here

and so on so here here you have another two poles

so we have this this three of recursive calls let's calculate the total number of recursive calls

how to calculate the total number of recursive calls we need to calculate number of nodes in this tree to complete number of nodes of this tree we need to know its length so what is the height of this tree so what is what will be the case of the stream the head of three will be the deepest recursive call so the number of recursive call in in every branch of this string will be up to log n like before yeah so we'll make f of n f of n over 2 and over 4 and so on until we reach f of 0 in the end

so so number of

calls on each branch of this tree will be login

and now you want to calculate the number of nodes in the tree of height equal to logan what is the number of trees and let's say since h equal to again so if you have tree of hate h what will be the total number of elements in this string it will be about 2 in the power of logan so total number of elements here will be 2 in power of h

so this height of the 3 is log n so we need to put this logan here so total number of elements in this tree will be 2 in the power of log n base 2

which is n

so the time complexity of this algorithm is linear

so this happens sometimes so you have some strange recursive recursive function and when you calculate total number of recursive calls you may you have something similar in there so um you don't ask questions that make me suspicious you don't know the same thing well that's the question okay let's

if you don't ask any questions i don't understand if you live with me or not so let's see if you have three so we have some first layer second layer third layer and so on

so on the first layer you have one node on the second node layer you have two nodes you have four nodes here eight nodes here and so on so in the last level you will have two power h nodes

now we need to calculate sum of all these elements so we need to calculate sum one plus two plus four plus and so on plus 2 and power of h this is the sum of geometric progression so the sum of these powers of 2 will be like 2 and power h minus 1 minus 1.

and this is big o of twin power of h

2 plus 1

so when you have 3 of 8 2 in power of h total number of elements in this tree will be big o of 2 and power of h

is it more clear now okay i hope now it's more clear

and now here the height is logarithm base 2 so if you have 2 in the power of logarithm base 12 n you will have n cool cool now let's let's complicate it a little bit more let's let's add like third recursive call why not

what happened if we add third recursive call here

okay it's your turn like try to do everything the same but if three recursive calls what will happen

we have we call f of n and then we have three recursive calls here each pole is n over two

then again we have three recursive calls here three three three three three curves equals uh n over four here and so on

yes the height of the three will be the same the height is log base two of n

cool so but but yes the height will be the same but the number of elements will be different because the branching factor is not two but three so in the bottom layer we have not two in the power of h but three in the power of h because each time we split into three branches so total number of operations here will be three and power of h

and now we need to calculate three in power of log base two it's not n because this base is different so this actually will be equal to n in the power log base 2 of 3. so this number is bigger than n but less than n square so so we have n in some strange part so what what is the log of 3 base 2 it's one it's about 1.7 i think so it's about and one point

maybe less no no no it's about one point seven not exactly so it is n in some strange power so it's not not linear not n square but something in the middle and actually sometimes it happens so we will discuss some algorithms which have some some strange complexity like this

8 will be log base 2 because each time we split in 2 so we have each time we have n over 2 here so each time n is decreased twice yeah that's a good question since let's i'll add this in the in the practice section

that's that that's nice exercise try to solve it good

oh we spent a whole hour for this okay now let's move to some real algorithms so now we will talk about sorting algorithms

that all was like uh introduction about how to how to measure the complexity of the of the algorithm so time complexity of the august measures something like that so now let's move to some real options ah that's not all that's not

we need to discuss one more thing and we will do it now

now we will talk about sorting algorithms

what is the sourcing algorithm is basically the algorithm you take the array and you return the sorted ring so you have input like array n a and output uh let's say array b

same as is

what's up

okay and the first sorting algorithms i want to talk about is the insertion zone

ah the insertion software works like this you have some array and you move from left to right and try to put each element in its correct place so you move from left to right on each step you have some sorted prefix you take the next element this element

and move it to the left until it reaches correct position so you take this element and then try to move it to the left until it finds the correct position of this element

cool

let's go so let's write the code so we have let's say we have 4 i from 0 to n minus 1. let's say this element is index i and now we need to move this element to the right or the left how we'll do it let's say let's say j is the index of the current element we are moving so first it's i and now we move it to the left until we reach the left border off or we reach element which is less than so while our j is greater than zero and

we can move we can move it to the left if it is less than element of electron if this element is less than element to the left

we will move this element one position to the left so we swap element a j with element a j minus one

and and decreasing

looks fine so this is the basic insertion sort algorithm

is it clear it's working or should we prove it

let's prove let's prove it's algorithm works so how to prove this algorithm works we will prove it using the volume invariant so we will prove that after each iteration of this cycle we have the sorted prefix of the array

so here here we have sorted

elements from zero to i

let's prove this works uh if it was sorted on previous iteration then on the next iteration what happens we take the next element move it to the

left until so we put we have situation like this so we move it somewhere here uh and if this vial stops it means that uh this j uh this aj is greater or equal than previously so it's greater equal than this and since we moved it on the previous iteration it means that we have less here

so we have less oracle here we have less here and this part is sorted this part is sorted so we have this all elements in sorted order so these are sorted and these are sorted this is less ripple this is what that's all it's quite easy ah now let's talk about what is the time complexity of this algorithm to calculate time complexity we need to calculate the number of again number of operations with algorithm makes to sort the array and what is the tricky part the tricky part is that the number of iterations depends on the input so for summaries this algorithm may work much longer than for other arrays what is the best case for this album source what is the best array for this album

again to the proof uh no again uh less proof by by induction so so let's prove that in the end of this iteration so after iteration i you have the certain prefix of elements from 0 to i why is it sorted because it was sorted so it was on the on the previous iteration we have sorted elements from zero to i minus one so on the next iteration we take this element this element and move it to the left to some position

we have this part sorted this part sorted and here since we stopped moving we have less or equal here and we have less here so the whole this prefix is sorted so when you move from iteration i minus 1 to iteration i you have the sort prefix of length i so in the end after the last iteration you have sorted prefix of the whole array so all the rays sorted

yeah what i want to talk about the time complexity of this algorithm actually depends on what input you give to him so for some inputs it's fast for some inputs it's slow what is the best case the best case is when the array is already sorted so if you have array like this

what will be the total number of iterations uh let's see so each time you go into this one you check that element is less than the previous element have false here so you don't go inside this cycle so on each iteration you make only constant number of operations and move to the next iteration so the total time complexity sorted array will be linear

and what is the worst case

uh the worst case will be then the other that will have the same array but reverse yes when we have array n n minus one and so on one uh then we each time we go into this while and make this swap so we first move this element to the left this element move twice this one and move three times and so on last element will be moved to the first position so the time complexity here will be n squared

and for for the most logarithmic in this course uh things

like this there are some good cases and some bad cases and to analyze this is important part to analyze the time complexity of the algorithm we will talk only about worst case so in most for the most obvious for some algorithms will be different but most of the time we will talk only about the worst case so

so if i don't specify any different when i talk about time complexity algorithm i talk about the worst time complex time complexity so we find the worst possible input and calculate the time complexity on this worst possible input

okay so time complexity of insertion sort is n square it means that it's time complexity in the worst case okay good

now let's move to really finally let's uh finally let's get some something interesting so finally let's talk about merced let's

now we're finally ready to make something not that trivial okay mercedes

another sorting algorithms with time complexity less than n square spoilers uh so how does the mercer work so u the central element of the merge solved algorithm is the merge operation

now separation works like this you have two sorted arrays let's make salted array eight sorted and b

to save space let's move it here

so you're given two sorted arrays

and it outputs the merged version of this array so we have all elements of this arrays we merge them into one big sorcery

let's say c so let's put some real elements here so we have some elements and one five and ten here and some two four and six here so and then we'll have something like one two four five six and ten

non-decreasing it's it's like small details if you have non-decreasing or increasing it doesn't affect any anything else i will say increasing but it's actually the same if you have the non-degrees and also if you have equal elements it doesn't affect algorithm as well so how to merge two sorted rates into one sort three

let's let's see so

you have two sort of tricks

let's draw a little bit bigger

and you want to put all these elements into one big ray insert to other let's go from left to right

let's go from left to right we'll talk about quick sorts on next section not next in the lecture in two lectures i think so we'll talk about not today quick sort is in the plan but not today

again we want to merge these two arrays into one degree

uh what happens so what let's find the first element we need to put in the in this mercury so this leftmost element must be the minimum elements of all both arrays so we have some elements here some elements here you need to find the minimal element and put it in the first position so how can we find the minimal element minimal element is in this element or this element

so this is the minimal element in the first array this is the minimal element in the second trick if you want to find the minimal element offers you take the minimum of these two elements

so let's take minimum of these two elements it will be one and take this one and put it on the first position in the result ring put one here

now remove this one

and do the same again now to find the second element we need to find the minimum of these two elements so we take minimum of two and five see that two is less so we take this two put it on the second position

remove this from there and so on each time we have two elements in the race one one element in the first array one element in the secondary we take the minimal element and move it on the next position in ray c yeah so

let's write the quote

if we have the same elements you can pick any element so if you have same elements in left or so you could take any element so they they will be equal so you can put them in any order

let's see so we'll have three variables so first variable will be the index of the first element of the first array so i uh here we have index j it will be the index of the first element in the secondary and we'll have this index k will be the first empty position in racing

let's say n is the length of r and m is the length length of b so this is the length of these four so this is array of of size n this is the rate of size m

now let's initialize our variables so we have i equal in the beginning we have i equal to zero j equal to zero and k equal to zero

they may have different sizes

that's fine if they these two arrays may have different size it doesn't matter

all right you discussed the quick sort quick sort we'll discuss not today but in the future lectures we'll discuss quite shortly how to pick the pivot element and so on ah

now we need to merge these two arrays so we take each time we take the minimum of these two elements and put them into racing let's say so there are different ways to implement this i i will implement like this while both arrays are more empty less m or j is less than m so why i have more elements i will put one element from a right to here let's say so when i need to move element from the first array i need to move element if second array is empty so if b equal to m or if first element is less than this element so or if i less than n and

i is less than vg

spend too much space on this on this

board

go so if we have something like this so if we have secondaries empty or if the first element is less than second element then we move this element from array a to array c let's say c k plus plus equal to a i plus plus

here i use plus plus like in c plus plus there's no plus plus in python but i like how it works when you add element into array so i will use plus plus in this silver code again this code is just combination of all good techniques in different programming languages

cool so we move this element to array c and increment k and increment i else we need to move element from array b so we can do the same move element

from array that's all

yes we can yes we don't talk about memory complexity here yes here here we need uh to sort to merge this to arrays we need to create another big ray this is not exactly optimal way to do it you can do the same without additional memory but we will not talk about it today it's kind of complicated so we will create another array and just put these elements into different array just spread another and memory to create another array it's not optimal but fine for today

where

seagull i didn't create rayc but i think it's like you you need to create ray c of size n plus m somewhere here let's say c is array of size n plus m something like that

just create there initialize the zero something

c is the ring basically you need to initialize somehow yes i agree good

now let's calculate the time complexity of this algorithm now it's a little bit more tricky again when you have a while cycle it's usually more tricky to calculate the time complexity to calculate time complexity you need to calculate the number of iterations of this cycle and usually the best way to calculate number of iterations is to make some uh half invariants so let's look on some value uh which increases every time you make another iterations of the cycle for example in this cycle each time if you look on the value i plus j

and on each iteration you either increase i or j so on each iteration this sum will increase by one and in the beginning you you have i plus j equal to zero in the end you have i plus j equal to n plus m and on each iteration this sum i plus j increased by one

so the total number of iterations will be equal to n plus m now i believe there is easier way to to to prove this but the more intuitive way to prove this is like every time you pick the element from the array and move it so total number of elements is n plus m so if you take if you remove one element each time the total number of iterations will be n plus m so it's not it's not not that difficult actually okay so we can merge two arrays a and b of size n and m in total time and plus this pretty good code actually you you you can program it like this it's it's actually pretty efficient

okay now uh

jacob of course of course yes j i i j that's right

small bug okay actually thank you for pointing

good now what we need to do we need to uh using this procedure make a sorting algorithm how to create a sorting algorithm when you can

merge two sort trays uh you can do it while we we can we will make the algorithm using the divide and conquer technique

special name for for these techniques okay how to sort there so you have everything this is merged this is so uh sort of a we have array a and we need to sort it

how divide and conquer technique works you have some array and you need to perform some procedure you split this array into several parts in our case in two parts so we'll split this array into two halves place half

let's call them b and c just for simplicity so b is the left part of the array and c is the right one

uh now we will call the same recursive function for both halves of the array so we will call a preparation sort for array b and operation software array abc so we'll have two sort of trays we'll have sort ray b and sword racing and in the end we will merge these two arrays into one big array so we will call this merge procedure and merge these two arrays into one degree

so we'll have one big sorcery

that's the plan again what is the plan i'll just write the quote here it will be very easy so so you have array a you split it into two halves let's say b is the left half say a from zero to n divided by 2 minus 1 and c is the right part so it's a from n divided by 2 to n minus 1. so these are two these two helps now you we call the same function for these two arrays

and then we merge these two halves into one bigger

oh my god uh if you if you make function like this it it will never end so if you make if you make recursive function you need to stop it somehow here i always have this recursive calls so i this recursion will never stop you need to stop the recursion at some point and here let's say if

fix small fix up

let's say if a length of array a is less than 2 then just hit ok

now it's fine so again how it works if if array is small it's one element or it's empty we just return the same element it's already sorted if it is it has at least two elements we can split it into two parts so we split in left half and right half uh then we sort both halves using the same algorithm just call this this function sort is the same as this function so we can just call the same function here and then merge these two arrays into one big ring cool

looks fine yeah okay

good and final thing let's calculate time complexity now using all we learn today we need to calculate the total number of operations of all this recursive function together in this

start

so how to calculate the time complexity of this function again this is a recursive function we already did similar thing so we have recursive function we have two recursive calls

let's say that t of n is the time complexity of array of size n so what happened here here

here we have two recursive calls but each recursive call have array of twice smaller size so we have two recursive calls each call spend time t of n over two half t one over two here

and now let's calculate the time complexity of all other lines here uh so we need to split array here and we need to return merge here so time complexity of all other lines is linear so we need to spend big o of n time here to calculate this merge and we need to spend we go off and time here just to make this splits so total time complexity will be two multiplied by n over two plus some linear stuff here so plus let's say some c multiplied by n

good

yeah we had almost the same recursive function before but what different here uh here is this part so here we have additional linear processing time in each recursive calls so uh again what happened that's right

so what's happening here so we have a recursive call of all the array of size n and then we have two recursive calls of size n over two then we have four recursive holes of size n r4 and so on

but now we need not calculate the number of recursive calls but we need to locate the total number spent in all recursive calls and in each recursive course we spent not the constant time but linear time so so in this in the main recursive call we make these two recursive calls and then spend another end time to calculate this merge

now we make two recursive calls in each call we spend another n over time two or n over two time to calculate emerging each two sections so we calculate merge here in n over to time calculate merge here in our time so we spend and over two time here and over and over the input here so we need not to calculate the number of nodes here but calculate the sum of all elements here so let's calculate the sum of all numbers in this tree how to calculate this sum

easiest way is just to look on this tree and calculate it so on each level uh the sum of all numbers is n so we spend end time on this level here we have n n over two here and one or two here the sum is n

here we have four recursive calls each recursive call make a n over four operations so total number of operations in all these four recursive calls will be n and so on so on each layer we have sum of all these recursive calls will be n and the total number of these layers will be login so the sum of all these layers will be multiplied by log

it means the total complexity is n log

this is like more intuitive approach to to calculate the content complexity you can perform more mathematicians mathematical approach to calculate the complexity and you can do it using the mathematical induction so how to calculate uh let's prove by induction that t of n is less or equal than we have c here let's use another constant well let's say it says c and logan

let's prove that time complexity is no more than c multiplied by n log n let's prove it by induction so suppose it is correct for the smaller n how to prove it for bigger m we need to make a transition from small and bigger so we put this l into here we have t of m t of n is 2 times t of n over 2 t of n over 2 is

so we have 2 multiplied by this time so c r n over 2 log of n over 2 plus c n

and now we just simplify this so we can get rid of these twos log of n over two is n minus one so we have c of c n logan minus one plus cm when you simplify this you have c n log n minus c n plus plus c n so it is so you can prove it like this just just by induction you suppose it's working for small n and then make the proof for the beginning n log logan is bigger than end yeah that's correct

okay

uh-huh looks fine oh yeah perfect timing yes we i plan to end about this time so any more questions okay the final thing i want to talk about is about master theorem just to mention it so uh the formulas like this are usually happening when you have algorithm which divides the input in some parts and call the recursive function so there are a lot of algorithms which use the divide and conquer technique but there's some different constants uh let's write it here

recommend the book i don't know good book there is a corman book classical common book about algorithms but

not something wrong let's click on excess we'll talk about coding exercises i think i'll make a post of code forces and we'll discuss what is the best way to make practice good final thing so uh we have hormones like this in various different algorithms so when you have algorithm which splits the input into let's say b chunks and then make a recursive calls

let's say you have algorithm like this you're given some input of size n uh you split it into b parts and make a recursive calls so you have

you have a recursive calls here

uh your usually a is bigger than b because when you split the input you need to make at least one recursive but you know not always but usually a is bigger than n now in in the end you make some some additional operations just to combine all these results and this costs you

already have f

let's rename the function

i want to use f

and in the end you spend another time just to combine these results into one so we have some f of n here

now how to calculate time complexity of such algorithms so what is the time complexity of this algorithm let's say t of n so we have a recursive calls of size n over b so we have a multiplied by t of n over b plus some african

so situations like this happens all the time so so they make a special theorem about situations like this it's called the master theory master theme is the theorem which uh

calculates the time complexity of algorithms of this form let's go so let's start from simple cases so if if oh cool so if you remove this f of n so if you if you say f of n equal to one

so what will be the time complexity again what will happen you have you have some tree which have branch factor equal to a so each time you have a recursive calls

so each time you have a recursive calls and the height of the tree will be equal to log and base b right so the total number of elements in this tree will be a power log b n like this is equal to n in power log b a

again we won't discuss this like we discussed when you have b equal to 2 and a equal to 3 but it's just generalization this so when you have branch factor equal to 2 and you divide input by b then the height of this 3 will be equal to log base b and total number of recursive calls will be n in the power of log base b okay and now no no no no

what the theorem says theorem says that if this function f

is much less than this n in the power of plus b for example if f of n is big o of again it will be very simplified version of master theorem we'll just do not try to make more difficult also let's see if f n is big o of n in power like here log base b of a minus epsilon

so it is some polynomial polynomial of power less than log of a or base b so it is much less than this perimeter then the time complexity will be this

basically what happens then this number of recursive calls just is much more than the total time you need to spend on this uh f function so so the most time you're gonna spend just making this recursive calls

uh on the other hand if f n is big o of some bigger function so it's log

let's say plus epsilon so if this function f is much more than this function then the time complexity will be the same as function

it's f exactly the truth

better write this

smaller so the time complexity will be the same as as the function a

and finally if the complexity of function f is the same as this

then the total complexity of the algorithm will be like here it will be multiplied by a

logarithm

so this is basically how the master theorem works so if this function f is slower than this recursive calls then you the total complexity will be the same as this function f if it is faster than this this uh recursive calls then the complexity will be the same as this recursive cost without fast function if it is the same as this recursive calls then you will add logarithm factor here

that's basically all any more questions

no no more questions how to let's use master master theory to calculate time complexity of the of the

merchant

so in our case what happens so you have function like this with a equal to two and b equal to two so we have a equal to two b equal to two so we have n in power of logarithm of a and base b so logarithm of a b is b equal to 1 because it's logarithm of 2 base 2.

so we have this third case and this function f

is c multiplied by n

uh and this function f is big o of n

so we have this third case and the power of one

so we have this third case so in the first case uh the time complexity will be n in the power of log b this n and the power of one multiplied by log sometime complexity will be n and the power of one multiplied by logan

yes yes you you can use static here like i said in the beginning of the lecture i will most usually use big o instead of big data just because i'm lazy and i don't want to prove lower bounds but yes in the regional master theorem you can use static here in all places it will be it will be even more accurate if that's correct that's correct

okay thank you for watching so that's all for today see you next week i will make a special post about this course on the court forces and we'll discuss the best format for the practice problems okay so see you next week

you