

CS502 Short Notes Mid term

Lec 1 to22

BY vuonlinehelp.blogspot.com

YOUTUBE LINK:

WEBSITE LINK:

[HTTPS://VUONLINEHELP.BLOGSPOT.COM/](https://vuonlinehelp.blogspot.com/)

IMPORTANT NOTE

You should never copy paste the same answer which we are providing. Please do make some changes in the solution file. If you appreciate our work, please share our website vuonlinehelp.blogspot.com with your friends.

آپ کو کبھی بھی وہی جواب کاپی پیسٹ نہیں کرنا چاہیے جو ہم فراہم کر رہے ہیں۔ براہ کرم حل فائل میں کچھ تبدیلیاں کریں۔ اگر آپ ہمارے کام کی تعریف کرتے ہیں تو براہ کرم ہماری ویب سائٹ vuonlinehelp.blogspot.com کو اپنے دوستوں کے ساتھ شیئر کریں

More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglhAA1SG3>

Chapter 1

1st Lecher

Origin of word: Algorithm

The word Algorithm comes from the name of the muslim author Abu Ja'far Mohammad ibn Musa al-Khowarizmi. He was born in the eighth century at Khwarizm (Kheva), a town south of river Oxus in present Uzbekistan. Uzbekistan, a Muslim country for over a thousand years, was taken over by the Russians in 1873.

It has been established from his contributions that he flourished under Khalifah Al-Mamun at Baghdad during 813 to 833 C.E. Al-Khwarizmi died around 840 C.E.

Much of al-Khwarizmi's work was written in a book titled al Kitab al-mukhtasar fi hisab al-jabr wa'l-muqabalah (The Compendious Book on Calculation by Completion and Balancing).

Algorithm: Informal Definition

An algorithm is any well-defined computational procedure that takes some values, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into output.

Algorithms, Programming

A good understanding of algorithms is essential for a good understanding of the most basic element of computer science: programming. Unlike a program, an algorithm is a mathematical entity, which is independent of a specific programming language, machine, or compiler.

Implementation Issues

One of the elements that we will focus on in this course is to try to study algorithms as pure mathematical objects, and so ignore issues such as programming language, machine, and operating system.

Analyzing Algorithms

Design good algorithms, we must first agree the criteria for measuring algorithms. The emphasis in this course will be on the design of efficient algorithm, and hence we will measure algorithms in terms of the amount of computational resources that the algorithm requires.

Model of Computation

Another goal that we will have in this course is that our analysis be as independent as possible of the variations in machine, operating system, compiler, or programming language. Unlike programs, algorithms to be understood primarily by people (i.e. programmers) and not machines. mathematical model of computation. A RAM is an idealized machine with an infinitely large random-access memory. Instructions are executed one-by-one.

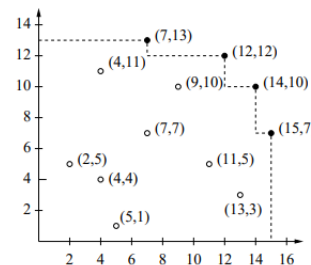
Basic operations include things like assigning a value to a variable, computing any basic arithmetic operation (+, -, $\times$, integer division).

Chapter 1

2nd Lecher

2-dimensional Maxima:

Given a set of points $P = \{p_1, p_2, \dots, p_n\}$ in 2-space, each represented by its x and y integer coordinates, output the set of the maximal points of P, that is, those points p_i , such that p_i is not dominated by any other point of P.



Observe that our description of the problem so far has been at a fairly mathematical level. These would normally be important in a software specification.

Chapter 1

3rd Lecture

Brute-Force Algorithm

More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglhAA1SG3>

A simple brute-force algorithm, with no thought to efficiency. Let $P = \{p_1, p_2, \dots, p_n\}$ be the initial set of points.

MAXIMA (int n , Point $P[1 \dots n]$)

1. for $i \leftarrow 1$ to n
2. do maximal $\leftarrow$ true
3. for $j \leftarrow 1$ to n
4. do
5. if $(i \neq j)$ and $(P[i].x = P[j].x)$ and $(P[i].y = P[j].y)$
6. then maximal $\leftarrow$ false; break
7. if (maximal = true)
8. then output $P[i]$

Running Time Analysis

The running time of an implementation of the algorithm would depend upon the speed of the computer, programming language, optimization by the compiler etc. Although important, we will ignore these technological issues in our analysis.

Worst-case time

Is the maximum running time over all (legal) inputs of size n . Let I denote an input instance, let $|I|$ denote its length, and let $T(I)$ denote the running time of the algorithm on

input I . Then

$$T_{\text{worst}}(n) = \max_{|I|=n} T(I)$$

Average-case time

Is the average running time over all inputs of size n . Let $p(I)$ denote the probability of seeing this input. The average-case time is the weighted sum of running times with weights being the probabilities:

$$T_{\text{avg}}(n) = \sum_{|I|=n} p(I)T(I)$$

Chapter 1

4th lecture

More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglAA1SG3>

Analysis of the brute-force maxima algorithm

MAXIMA (int n, Point P [1 . . . n])

1. for i $\leftarrow$ 1 to n n times
2. do maximal $\leftarrow$ true
3. for j $\leftarrow$ 1 to n n times
4. do
5. if (i $\neq$ j) & (P[i].x = P[j].x) & (P[i].y = P[j].y) 4 accesses
6. then maximal $\leftarrow$ false break
7. if maximal
8. then output P[i].x, P[i].y 2 accesses

Thus we might express the worst-case running time as a pair of nested summations, one for the i-loop and the other for the j-loop:

$$\begin{aligned} T(n) &= \sum_{i=1}^n \left(2 + \sum_{j=1}^n 4 \right) \\ &= \sum_{i=1}^n (4n + 2) \\ &= (4n + 2)n = 4n^2 + 2n \end{aligned}$$

For small values of n, any algorithm is fast enough. Running time does become an issue. When n is large, n^2 term will be much larger than the n term and will dominate the running time.

The analysis involved computing a summation. Summation should be familiar but let us review a bit here. Given a finite sequence of values $a_1, a_2, \dots, a_n$, their sum $a_1 + a_2 + \dots + a_n$ is expressed in summation notation as

$$\sum_{i=1}^n a_i$$

If $n = 0$, then the sum is additive identity, 0.

$$\sum_{i=1}^n c a_i = c \sum_{i=1}^n a_i$$

$$\sum_{i=1}^n (a_i + b_i) = \sum_{i=1}^n a_i + \sum_{i=1}^n b_i$$

Some facts about summation: If c is a constant

Some important summations that should be committed to memory.

- **Arithmetic series**

$$\sum_{i=1}^n c a_i = c \sum_{i=1}^n a_i$$

$$\sum_{i=1}^n (a_i + b_i) = \sum_{i=1}^n a_i + \sum_{i=1}^n b_i$$

- **Quadratic series**

$$\begin{aligned} \sum_{i=1}^n i^2 &= 1 + 4 + 9 + \dots + n^2 \\ &= \frac{2n^3 + 3n^2 + n}{6} = \Theta(n^3) \end{aligned}$$

- **Geometric series**

$$\begin{aligned} \sum_{i=1}^n x^i &= 1 + x + x^2 + \dots + x^n \\ &= \frac{x^{(n+1)} - 1}{x - 1} = \Theta(n^2) \end{aligned}$$

If $0 < x < 1$ then this is $\Theta(1)$, and if $x > 1$, then this is $\Theta(x^n)$.

Harmonic series For $n \geq 0$

$$\begin{aligned} H_n &= \sum_{i=1}^n \frac{1}{i} \\ &= 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \approx \ln n \\ &= \Theta(\ln n) \end{aligned}$$

Analysis: A Harder Example

Let us consider a harder example.

NESTED-LOOPS ()

1. for $i \leftarrow 1$ to n
2. do
3. for $j \leftarrow 1$ to $2i$
4. do $k = j \dots$

More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglhAA1SG3>

5. while (k = 0)
6. do k = k - 1 . . .

Its running time is determined by i. Let M(i) be the time spent in the for loop:

$$\begin{aligned}
 M(i) &= \sum_{j=1}^{2i} I(j) \\
 &= \sum_{j=1}^{2i} (j + 1) \\
 &= \sum_{j=1}^{2i} j + \sum_{j=1}^{2i} 1 \\
 &= \frac{2i(2i + 1)}{2} + 2i \\
 &= 2i^2 + 3i
 \end{aligned}$$

Finally the outer-most for loop

NESTED-LOOPS()

1. for i ? 1 to n
2. do for j ? 1 to 2i
3. do k = j
4. while (k = 0)
5. do k = k - 1

Let T() be running time of the entire algorithm:

$$\begin{aligned}
 T(n) &= \sum_{i=1}^n M(i) \\
 &= \sum_{i=1}^n (2i^2 + 3i) \\
 &= \sum_{i=1}^n 2i^2 + \sum_{i=1}^n 3i \\
 &= 2 \frac{2n^3 + 3n^2 + n}{6} + 3 \frac{n(n + 1)}{2} \\
 &= \frac{4n^3 + 15n^2 + 11n}{6} \\
 &= \Theta(n^3)
 \end{aligned}$$

2-dimension Maxima Revisited

Recall the 2-d maxima problem: Let a point p in 2-dimensional space be given by its integer coordinates, $p = (p.x, p.y)$. A point p is said to dominated by point q if $p.x \leq q.x$ and $p.y \leq q.y$.

More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglhAA1SG3>

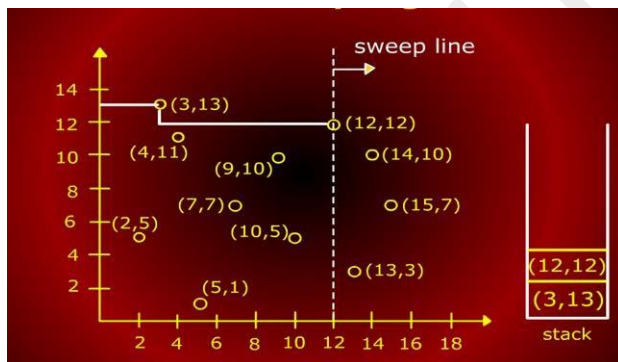
5th lectures

Plane-sweep Algorithm

We will sweep a vertical line across the 2-d plane from left to right. As we sweep, we will build a structure holding the maximal points lying to the left of the sweep line. The points that p_i dominates will appear at the end of the list because points are sorted by x-coordinate.

Plane sweep

We will build a structure holding the maximal points lying to the left of the sweep line. When the sweep line reaches the rightmost point of P , then we will have constructed the complete set of maxima. This approach of solving geometric problems by sweeping a line across the plane is called plane sweep. We need some way to perform the plane sweep in discrete steps. To do this, we will begin by sorting the points in increasing order of their x-coordinates.



Here is the algorithm.

PLANE-SWEEP-MAXIMA (n , P [1... n])

1. sort P in increasing order by x ;
2. stack s ;
3. for $i \leftarrow 1$ to n
4. do
5. while ($s.\text{notEmpty}()$ & $s.\text{top}().y = P[i].y$)
6. do $s.\text{pop}()$;
7. $s.\text{push}(P[i])$;

More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglhAA1SG3>

8. 8 output the contents of stack s;

Analysis of Plane-sweep Algorithm

Sorting takes $\Theta(n \log n)$; we will show this later when we discuss sorting. The for loop executes n times. The inner loop (seemingly) could be iterated $(n - 1)$ times. It seems we still have an $n(n - 1)$ or $\Theta(n^2)$ algorithm.

We pop an element off the stack each time we go through the inner while-loop. It is impossible to pop more elements than are ever pushed on the stack.

The for-loop iterates n times and the inner while-loop also iterates n time for a total of $\Theta(n)$. Combined with the sorting, the runtime of entire plane-sweep algorithm is $\Theta(n \log n)$.

Comparison of Brute-force and Plane sweep algorithms

Consider the ratio of running times: $\frac{n^2}{n \log n} = \frac{n}{\log n}$

n	log n	$\frac{n}{\log n}$
100	7	15
1000	10	100
10000	13	752
100000	17	6021
1000000	20	50171

Chapter 2 Asymptotic Notation

6th, 7th lecher

6th lecher

Asymptotic Notation

Continue to use the notation $\Theta()$ but have never defined it. Let's remedy this now. Given any function $g(n)$, we define $\Theta(g(n))$ to be a set of functions that asymptotically equivalent to $g(n)$.

Formally:

$\Theta(g(n)) = \{f(n) \mid \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 < c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$

This is written as “ $f(n) \in \Theta(g(n))$ ” That is, $f(n)$ and $g(n)$ are asymptotically equivalent. This means that they have essentially the same growth rates for large n .

- Lower bound: $f(n)$ grows asymptotically at least as fast as n^2 .
- Upper bound: $f(n)$ grows asymptotically no faster than n^2 .

The definition of Θ -notation relies on proving both a lower and upper asymptotic bound. Sometimes we are only interested in proving one bound or the other. The O -notation is used to state only the asymptotic upper bounds.

$O(g(n)) = \{f(n) \mid \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0\}$

The Ω -notation allows us to state only the asymptotic lower bounds.

$\Omega(g(n)) = \{f(n) \mid \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 < c g(n) \leq f(n) \text{ for all } n \geq n_0\}$

7th lecher

The three notations:

$\Theta(g(n)) : 0 < c_1 g(n) \leq f(n) \leq c_2 g(n)$
 $O(g(n)) : 0 \leq f(n) \leq c g(n)$
 $\Omega(g(n)) : 0 < c g(n) \leq f(n)$
for all $n \geq n_0$

These definitions suggest an alternative way of showing the asymptotic behavior. We can use limits to define the asymptotic behavior. Limit rule for Θ -notation:

More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglhAA1SG3>

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c,$$

for some constant $c > 0$ (strictly positive but not infinity) then $f(n) \in \Theta(g(n))$. Similarly, the limit rule for O -notation is

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c,$$

for some constant $c \geq 0$ (nonnegative but not infinite) then $f(n) \in O(g(n))$ and limit rule for Ω -notation:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \neq 0,$$

Here is a list of common asymptotic running times:

- $\Theta(1)$: Constant time; can't beat it!
- $\Theta(\log n)$: Inserting into a balanced binary tree; time to find an item in a sorted array of length n using binary search.
- $\Theta(n)$: About the fastest that an algorithm can run.
- $\Theta(n \log n)$: Best sorting algorithms.
- $\Theta(n^2)$, $\Theta(n^3)$: Polynomial time. These running times are acceptable when the exponent of n is small or n is not too large, e.g., $n \leq 1000$.
- $\Theta(2^n)$, $\Theta(3^n)$: Exponential time. Acceptable only if n is small, e.g., $n = 50$.
- $\Theta(n!)$, $\Theta(nn)$: Acceptable only for really small n , e.g. $n = 20$.

Chapter 3

Divide and Conquer Strategy

The ancient Roman politicians understood an important principle of good algorithm design. The main elements to a divide-and-conquer solution are

- Divide: the problem into a small number of pieces
- Conquer: solve each piece by applying divide and conquer to it recursively
- Combine: the pieces together into a global solution.

Merge Sort

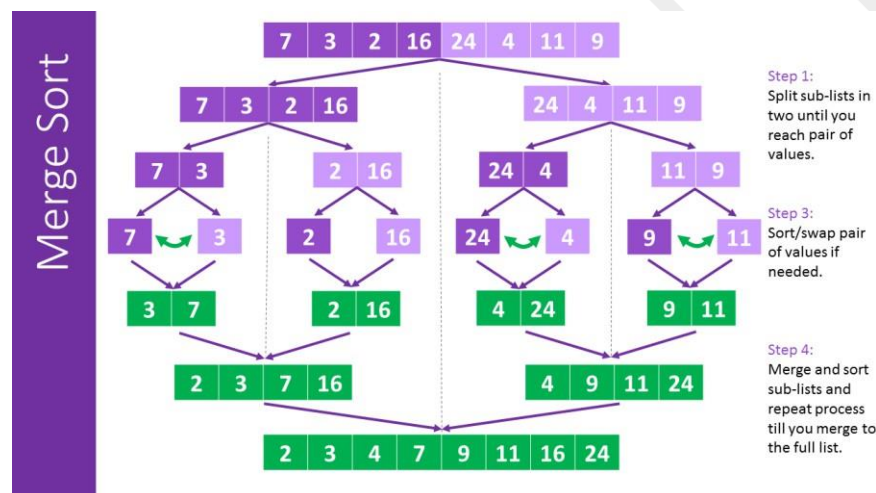
More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglhAA1SG3>

Divide and conquer strategy is applicable in a huge number of computational problems. A simple and efficient sorting procedure called we are given a sequence of n numbers A . assumes are stored in an array $A[1..n]$. The merge sort algorithm works:

- (Divide:) split A down the middle into two subsequences, each of size roughly $n/2$
- (Conquer :) sort each subsequence by calling merge sort recursively on each.
- (Combine :) merges the two sorted subsequences into a single sorted list.

MERGE-SORT(array A , int p , int r)

1. if ($p < r$)
2. then
3. $q \leftarrow (p + r)/2$
4. MERGE-SORT (A , p , q) // sort $A[p..q]$
5. MERGE-SORT(A , $q + 1$, r) // sort $A[q + 1..r]$
6. MERGE(A , p , q , r) // merge the two pieces



Chapter 3

8th lecher

Analysis of Merge Sort

More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglhAA1SG3>

The procedure Merge (A, p, q, r). Let $n = r - p + 1$ denote the total length of both the left and right sub-arrays. The running time to Merge n items is $\Theta(n)$. Let us write this without the asymptotic notation, simply as n .

a function that is defined recursively in terms of itself. To avoid circularity, the recurrence for a given value of n is defined in terms of values that are strictly smaller than n . Finally, a recurrence has some basis values (e.g. for $n = 1$).

$T(n)$ denote the worst case running time of MergeSort on an array of length n . MergeSort with an array containing a single item ($n = 1$) then the running time is constant. We can just write $T(n) = 1$, ignoring all constants. For $n > 1$, MergeSort splits into two halves, sorts the two and then merges them together. The left half is of size $\lfloor n/2 \rfloor$ and the right half is $\lfloor n/2 \rfloor$.

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ T(\lfloor n/2 \rfloor) + T(\lfloor n/2 \rfloor) + n & \text{otherwise} \end{cases}$$

In conclusion we have:

This is called recurrence relation, i.e., a recursively defined function. Divide-and-conquer is an important design technique, and it naturally gives rise to recursive algorithms. It is thus important to develop mathematical techniques for solving recurrences, either exactly or asymptotically.

Let's expand the terms.

$$T(1) = 1$$

$$T(2) = T(1) + T(1) + 2 = 1 + 1 + 2 = 4$$

$$T(3) = T(2) + T(1) + 3 = 4 + 1 + 3 = 8$$

$$T(4) = T(2) + T(2) + 4 = 4 + 4 + 4 = 12$$

$$T(5) = T(3) + T(2) + 5 = 8 + 4 + 5 = 17$$

...

$$T(8) = T(4) + T(4) + 8 = 12 + 12 + 8 = 32$$

...

$$T(16) = T(8) + T(8) + 16 = 32 + 32 + 16 = 80$$

...

$$T(32) = T(16) + T(16) + 32 = 80 + 80 + 32 = 192$$

Let's consider the ratios $T(n)/n$ for powers of 2:

$$\begin{array}{ll} T(1)/1 = 1 & T(8)/8 = 4 \\ T(2)/2 = 2 & T(16)/16 = 5 \\ T(4)/4 = 3 & T(32)/32 = 6 \end{array}$$

This suggests $T(n)/n = \log n + 1$ Or, $T(n) = n \log n + n$ which is $\Theta(n \log n)$ (using the limit rule).

Chapter 3

9th lecher

The Iteration Method for Solving Recurrence Relations

Floor and ceilings are a pain to deal with. If n is assumed to be a power of 2 ($2^k = n$), this

will simplify the recurrence to

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

The iteration method turns the recurrence into a summation. Let's see how it works. Let's expand the recurrence:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(2T(n/4) + n/2) + n \\ &= 4T(n/4) + n + n \\ &= 4(2T(n/8) + n/4) + n + n \\ &= 8T(n/8) + n + n + n \\ &= 8(2T(n/16) + n/8) + n + n + n \\ &= 16T(n/16) + n + n + n + n \end{aligned}$$

If n is a power of 2 then let $n = 2^k$ or $k = \log n$.

$$\begin{aligned} T(n) &= 2^k T(n/(2^k)) + \underbrace{(n + n + n + \dots + n)}_{k \text{ times}} \\ &= 2^k T(n/(2^k)) + kn \\ &= 2^{(\log n)} T(n/(2^{(\log n)})) + (\log n)n \\ &= 2^{(\log n)} T(n/n) + (\log n)n \\ &= nT(1) + n \log n = n + n \log n \end{aligned}$$

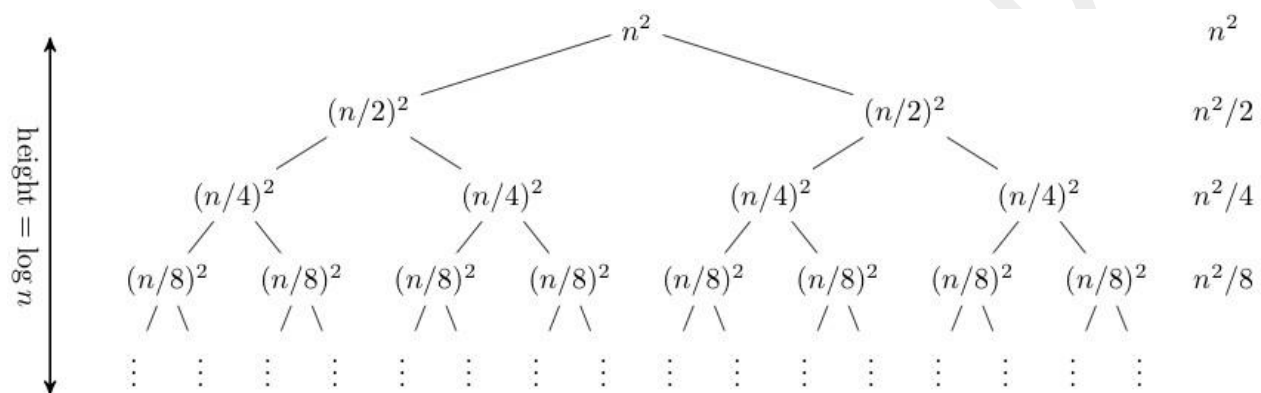
Visualizing Recurrences Using the Recursion Tree

Iteration is a very powerful technique for solving recurrences. Any recurrence in terms of a tree, where each expansion of the recurrence takes us one level deeper in the tree.

Recall that the recurrence for MergeSort (which we simplified by assuming that n is a power of 2, and hence could drop the floors and ceilings).

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

Merge two lists; we label that node of the tree with the time it takes to perform the associated (nonrecursive) merge. Recall that to merge two lists of size $m/2$ to a list of size m takes $\Theta(m)$ time, which we will just write as m . below is an illustration of the resulting recursion tree.



Selection Problem

Define the **rank** of an element to be one plus the number of elements that are smaller. Thus, the rank of an element is its final position if the set is sorted. The minimum is of **rank 1** and the **maximum** is of **rank n** . Consider the set: {5, 7, 2, 10, 8, 15, 21, 37, and 41}. The rank of each number is its position in the sorted order.

position	1	2	3	4	5	6	7	8	9
Number	2	5	7	8	10	15	21	37	41

For example, rank of 8 is 4, one plus the number of elements smaller than 8 which is 3. The selection problem is stated as follows:

Median

Given a set A of n distinct numbers and an integer k , $1 \leq k \leq n$, output the element of A of rank k . Of particular interest in statistics is the median. If n is odd then the median is defined to be element of rank $(n + 1)/2$.

When n is even, there are two choices: $n/2$ and $(n + 1)/2$. In statistics, it is common to return the average of the two elements.

Chapter 3

10th lecher

Sieve Technique

Introducing this algorithm is that it illustrates a very important special case of divide-and-conquer. We think of divide-and-conquer as breaking the problem into a small number of smaller sub problems, which are then solved recursively. The sieve technique is a special case, where the number of sub problems is just 1. The sieve technique works in phases as follows. It applies to problems where we are interested in finding a single item from a larger set of n items. We do not know which item is of interest, however after doing some amount of analysis of the data, taking say $\Theta(n^k)$ time.

Applying the Sieve to Selection

Sieve technique works, let us apply it to the selection problem. We will begin with the given array $A[1..n]$. We will pick an item from the array, called the pivot element which we will denote by x . We will talk about how an item is chosen as the pivot later; it as a random element of A .

We then partition A into three parts.

1. $A[q]$ contains the pivot element x ,
2. $A[1..q - 1]$ will contain all the elements that are less than x and
3. $A[q + 1..n]$ will contains all elements that are greater than x .

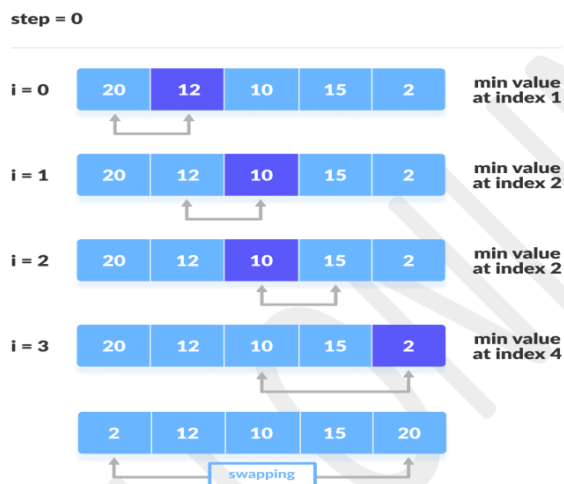
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Selection Algorithm

More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglhAA1SG3>

It is easy to see that the rank of the pivot x is $q - p + 1$ in $A[p..r]$. Let $\text{rank } x = q - p + 1$. If $k = \text{rank } x$ then the pivot is k^{th} smallest. If $k < \text{rank } x$ then search $A[p..q - 1]$ recursively. If $k > \text{rank } x$ then search $A[q + 1..r]$ recursively. Find element of rank $(k - q)$ because we eliminated q smaller elements in A . **SELECT** (array A , int p , int r , int k)

1. if ($p = r$)
2. then return $A[p]$
3. else $x \leftarrow \text{CHOOSE PIVOT}(A, p, r)$
4. $q \leftarrow \text{PARTITION}(A, p, r, x)$
5. $\text{rank } x \leftarrow q - p + 1$
6. if $k = \text{rank } x$
7. then return x
8. if $k < \text{rank } x$
9. then return **SELECT**($A, p, q - 1, k$)
10. else return **SELECT**($A, q + 1, r, k - q$)



Chapter 3

11th lecher

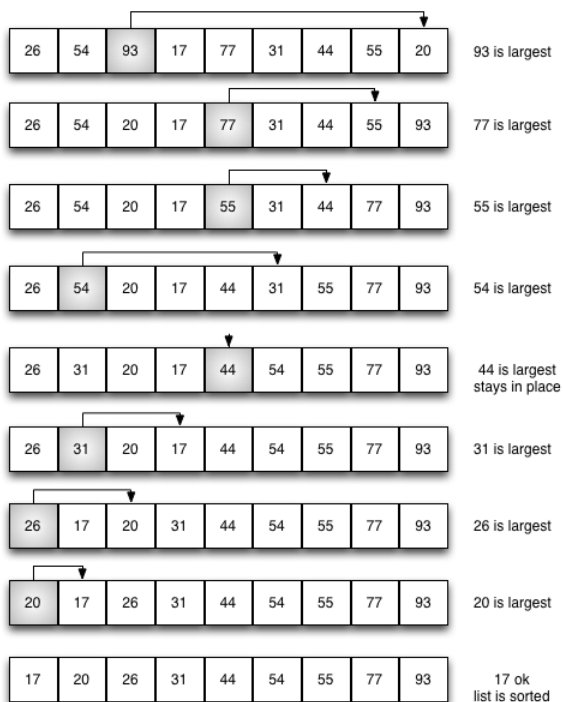
Selection Algorithm

It is easy to see that the rank of the pivot x is $q - p + 1$ in $A[p..r]$. Let $\text{rank } x = q - p + 1$. If $k = \text{rank } x$ then the pivot is k^{th} smallest. If $k < \text{rank } x$ then search $A[p..q - 1]$ recursively. If $k > \text{rank } x$ then search $A[q + 1..r]$ recursively. Find element of rank $(k - q)$ because we eliminated q smaller elements in A .

SELECT(array A , int p , int r , int k)

More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglhAA1SG3>

1. if ($p = r$)
2. then return $A[p]$
3. else $x \leftarrow \text{CHOOSE PIVOT}(A, p, r)$
4. $q \leftarrow \text{PARTITION}(A, p, r, x)$
5. $\text{rank } x \leftarrow q - p + 1$
6. if $k = \text{rank } x$
7. then return x
8. if $k < \text{rank } x$
9. then return $\text{SELECT}(A, p, q - 1, k)$
10. else return $\text{SELECT}(A, q + 1, r, k - q)$



SELECTION PROBLEM

This means that we recurse on the remaining $n/2$ elements. This leads to the following recurrence:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ T(n/2) + n & \text{otherwise} \end{cases}$$

If we expand this recurrence, we get

$$\begin{aligned}
 T(n) &= n + \frac{n}{2} + \frac{n}{4} + \dots \\
 &\leq \sum_{i=0}^{\infty} \frac{n}{2^i} \\
 &= n \sum_{i=0}^{\infty} \frac{1}{2^i}
 \end{aligned}$$

Recall the formula for infinite geometric series; for any $|c| < 1$,

$$\sum_{i=0}^{\infty} c^i = \frac{1}{1-c}$$

Using this we have $T(n) \leq 2n \in \Theta(n)$.

$\Theta(n)$ algorithm for selection. Normally, a $\Theta(n)$ time algorithm would make a single or perhaps a constant number of passes of the data set. In this algorithm, we make a number of passes. In fact it could be as many as $\log n$.

We eliminate a constant fraction of the array with each phase, we get the convergent geometric series in the analysis. the total running time is indeed linear in n .

Chapter 4

Sorting

Sorting is one of the few problems where provable lower bounds exist on how fast we can sort. In sorting, we are given an array $A[1..n]$ of n numbers. We are to reorder these elements into increasing (or decreasing) order.

Slow Sorting Algorithms

There are a number of well-known slow $O(n^2)$ sorting algorithms.

- **Bubble sort:**
Scan the array. Whenever two consecutive items are found that are out of order, swap them. Repeat until all consecutive items are in order.
- **Insertion sort:**
Assume that $A[1..i-1]$ have already been sorted. Insert $A[i]$ into its proper position in this sub array. Create this position by shifting all larger elements to the right.
- **Selection sort:**
Assume that $A[1..i-1]$ contain the $i-1$ smallest elements in sorted order. Find the smallest element in $A[i..n]$. Swap it with $A[i]$.

More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglhAA1SG3>

These algorithms are easy to implement. But they run in $T(n^2)$ time in the worst case.

Sorting in $O(n \log n)$ time

Mergesort sorts an array of numbers in $\Theta(n \log n)$ time.

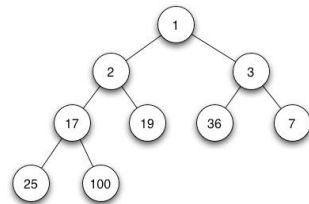
Chapter 4

12th lecher

Heaps

A heap is a left-complete binary tree that conforms to the heap order. The heap order property: in a (min) heap, for every node X , the key in the parent is smaller than or equal to the key in X .

In a max heap, the parents has a key larger than or equal both of its children Thus the smallest key is in the root in a min heap; in the max heap, the largest is in the root.



A min-heap

The number of nodes in a complete binary tree of height h is

$$n = 2^0 + 2^1 + 2^2 + \dots + 2^h = \sum_{i=0}^h 2^i = 2^{h+1} - 1$$

h in terms of n is

$$h = (\log(n + 1)) - 1 \approx \log n \in \Theta(\log n)$$

One of the clever aspects of heaps is that they can be stored in arrays without using any pointers. This is due to the left-complete nature of the binary tree. We store the tree nodes in level-order traversal.

A node involves simple arithmetic operations:

- Left (i): returns $2i$, index of left child of node i .

- Right (i): returns $2i + 1$, the right child.
- Parent (i): returns $\lfloor i/2 \rfloor$, the parent of i.
- The root is at position 1 of the array

Heapsort Algorithm

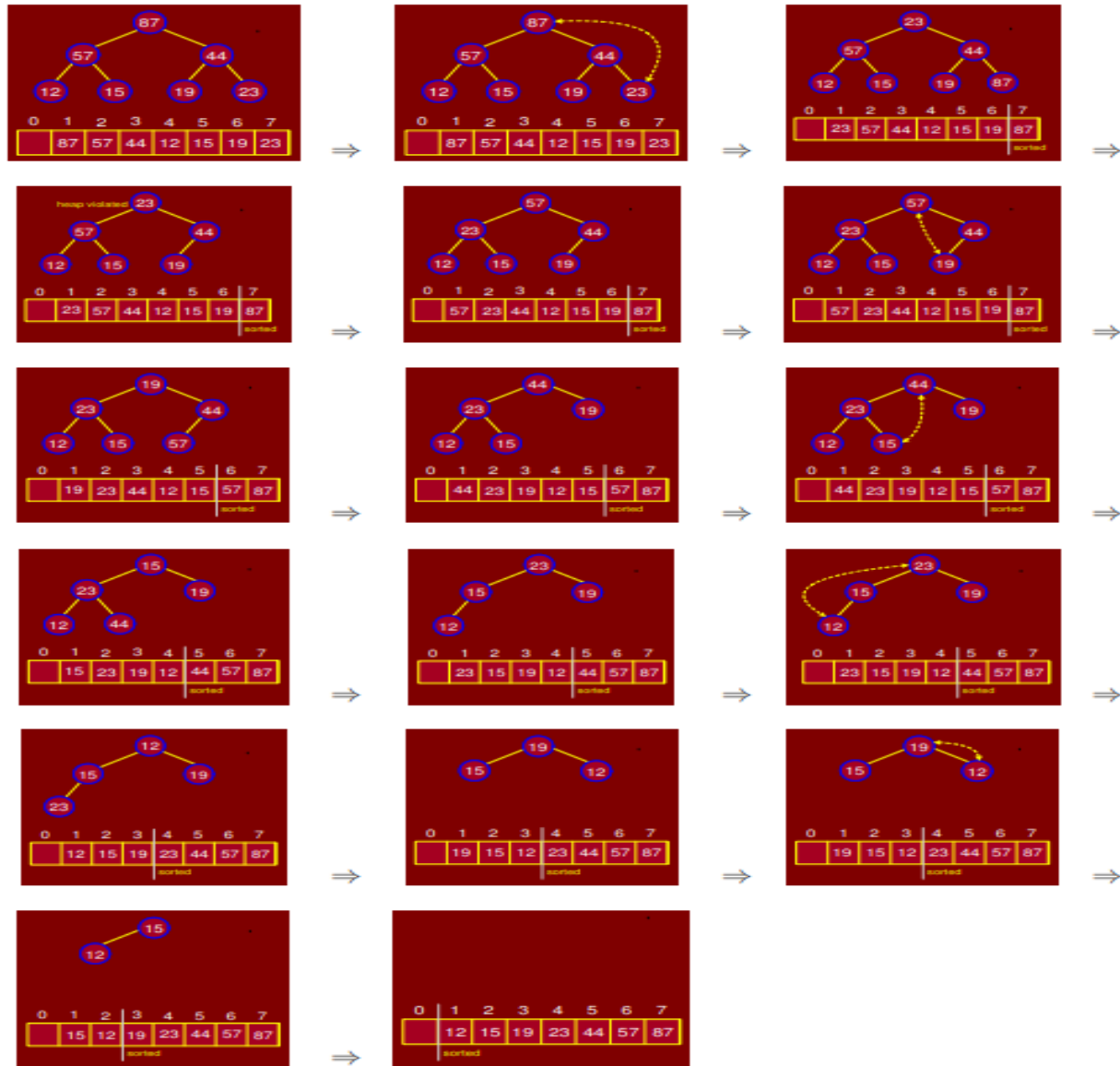
Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.

Heap sort basically recursively performs two main operations

- Build a heap H, using the elements of array.
- Repeatedly delete the root element of the heap formed in 1st phase.

HEAPSORT (array A, int n)

1. BUILD-HEAP(A, n)
2. $m \leftarrow n$
3. while ($m > 1$)
4. do SWAP(A[1], A[m])
5. $m \leftarrow m - 1$
6. HEAPIFY(A, 1, m)



Heapify Procedure

One principal operation for maintaining the heap property It is called Heapify. Other books it is sometimes called sifting down. In particular this root element may be too small. To fix this we “sift” it down the tree by swapping it with one of its children. The larger of the two children to satisfy the heap ordering property this continues recursively until the element is either larger than both its children or until its falls all the way to the leaf level.

Algorithm

It is given the heap in the array A, and the index i of the suspected element, and m the current active size of the heap. The element A[max] is set to the maximum of A[i] and its two children. If $\text{max} \neq i$ then we swap A[i] and A[max] and then recurse on A[max].

HEAPIFY (array A, int i, int m)

1. $l \leftarrow \text{LEFT}(i)$
2. $r \leftarrow \text{RIGHT}(i)$
3. $\text{max} \leftarrow i$
4. if $(l = m) \text{ and } (A[l] > A[\text{max}])$
5. then $\text{max} \leftarrow l$
6. if $(r = m) \text{ and } (A[r] > A[\text{max}])$
7. then $\text{max} \leftarrow r$
8. if $(\text{max} \neq i)$
9. then SWAP(A[i], A[max])
10. HEAPIFY(A, max, m)

Analysis of Heapify

The maximum levels an element could move up are $\Theta(\log n)$ levels. At each level, we do simple comparison which $\Theta(1)$. The total time for heapify is thus $\Theta(\log n)$.

BuildHeap

The "BuildHeap" operation is an efficient way to create a heap from an array in-place, with a time complexity of $O(n)$, where n is the number of elements in the array. The algorithm works as follows:

1. Start with an unsorted array.
2. Iterate over the array from the last non-leaf node to the first node (index 0).
3. For each node at index i, perform the "Heapify" operation to restore the heap property.
 - Compare the value of the current node with its children.
 - If the heap property is violated, swap the node with the larger (in a max heap) or smaller (in a min heap) child.
 - Repeat this process recursively for the affected child node until the heap property is restored.
4. Continue the iteration until the first node (index 0) is reached.
5. At this point, the array has been transformed into a binary heap.

The entire array, the parameter m for Heapify, which indicates the current heap size will be equal to n , the size of array A , in all the calls.

BUILDHEAP (array A , int n)

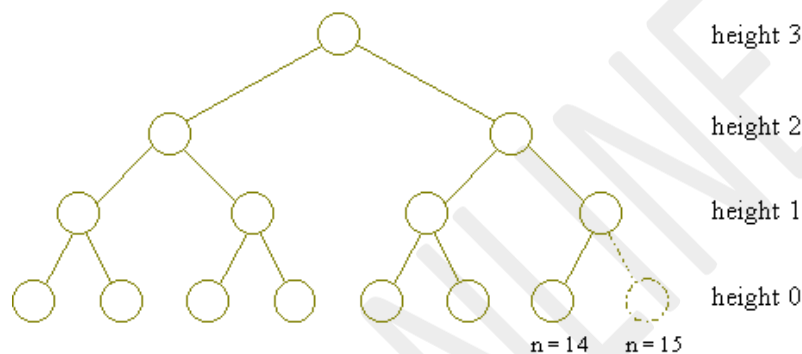
- 1 for $i \leftarrow n/2$ downto 1
- 2 do
- 3 HEAPIFY(A , i , n)

Analysis of BuildHeap

Each Heapify takes $O(\log n)$ time, and there are $O(n)$ calls to Heapify, so

$T(n) = O(n) \cdot O(\log n) = O(n \log n)$. But a more detailed analysis yields a tighter bound.

Heapify(i) takes time proportional to the height of node i . So count the individual steps:



Height	Maxnodes	Cost
3	$\leq \frac{n+1}{16}$	$\leq 4 \cdot \frac{n+1}{16}$
2	$\leq \frac{n+1}{8}$	$\leq 3 \cdot \frac{n+1}{8}$
1	$\leq \frac{n+1}{4}$	$\leq 2 \cdot \frac{n+1}{4}$
0	$\leq \frac{n+1}{2}$	$\leq 1 \cdot \frac{n+1}{2}$

- The number of nodes at height i is at most $\frac{n+1}{2^{i+1}}$.
 - The cost of Heapify at height i is at most $i+1$ per node.
- So the cost of BuildHeap is

$$\begin{aligned}
T(n) &\leq \sum_{i=1}^{\lfloor \log n \rfloor} \left(\frac{n+1}{2^{i+1}} \right) (i+1) \\
&= (n+1) \sum_{i=1}^{\lfloor \log n \rfloor} \frac{i+1}{2^{i+1}} \\
&= (n+1) \sum_{i=2}^{\lfloor \log n \rfloor + 1} \frac{i}{2^i} \\
&< (n+1) \sum_{i=0}^{\infty} \frac{i}{2^i} \\
&= 2(n+1) \quad \text{since } \left(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \dots \right) = 2
\end{aligned}$$

Therefore, $T(n) \in O(n)$.

There are a total of $n \approx 2^{h+1}$ nodes. The number of nodes at the bottom three levels alone is

$$2^h + 2^{h-1} + 2^{h-2} = \frac{n}{2} + \frac{n}{4} + \frac{n}{8} = \frac{7n}{8} = 0.875n$$

Almost 90% of the nodes of a complete binary tree reside in the 3 lowest levels. Thus, algorithms that operate on trees should be efficient (as BuildHeap is) on the bottom-most levels since that is where most of the weight of the tree resides.

Analysis of Heapsort

Heapsort calls BuildHeap once. This takes $\Theta(n)$. Heapsort then extracts roughly n maximum elements from the heap. Each extract requires a constant amount of work (swap) and $O(\log n)$ heapify. Heapsort is thus $O(n \log n)$.

Chapter 4

13th lecher

Quicksort

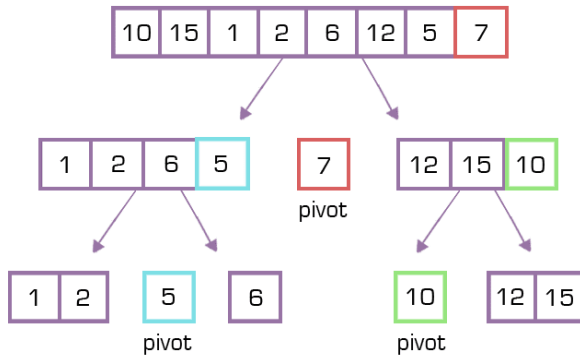
Sorting algorithm is Quicksort. It is one of the fastest sorting algorithms known and is the method of choice in most sorting libraries. Quicksort is based on the divide and conquer strategy. Here is the algorithm:

QUICKSORT (array A, int p, int r)

1. if ($r > p$)
2. then
3. $i \leftarrow$ a random index from $[p..r]$

More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglhAA1SG3>

4. swap $A[i]$ with $A[p]$
5. $q \leftarrow \text{PARTITION}(A, p, r)$
6. $\text{QUICKSORT}(A, p, q - 1)$
7. $\text{QUICKSORT}(A, q + 1, r)$

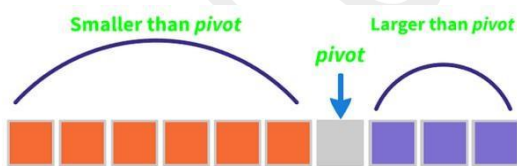


Partition Algorithm

Suppose you're given a collection of elements (unsorted). And your goal is to pick one element (from this collection) and "put it in its correct position".

For simplicity, let's say that its "correct position" (of the picked element) is defined as a position that all smaller elements will be on its left and all greater elements will be on its right.

Meaning, we're simply taking some collection of elements (for example, you may be familiar with "arrays" or "tuples") and then partitioning (or splitting!) this collection into two parts — left part and right part.



Distinguish between "right" and "left" sides

The partition should be done in such a way that after it's completed. We will be able to distinguish between the right side and the left side (based on some rule/criteria).

We can say that all elements on the left of some point are smaller than all elements on the right of that point.

Partition Algorithm [Step by Step]

Now, once we know what the Partition Problem is, let's try to solve it by writing a

More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglhAA1SG3>

suitable algorithm that can solve it.

Our Algorithm is just some “steps” to be taken, such that after they’re taken — we know that the chosen pivot is at it’s correct position (as previously explained).

Step 1: Get our collection of elements

There is a given collection of elements (numbers, etc) on which we would like to apply the “Partition Algorithm”.



Step 2: Choosing a PIVOT

Pick an element (the PIVOT) from the given collection of elements.

This PIVOT will be used for splitting the array into parts (smaller than the pivot and greater than pivot).

Usually, we’ll pick the pivot as the first element (leftmost element) of a given collection.

pivot — an arbitrary element of a given collection that we would like to put in it’s correct position.



Step 3: Set up “left” and “right” cursors

At this step, we will define 2 cursors/indicators which will help us in implementing this algorithm.

- “left cursor” — will indicate the “leftmost” element of a given collection.
- “right cursor” — will indicate the “rightmost” element of a given collection.

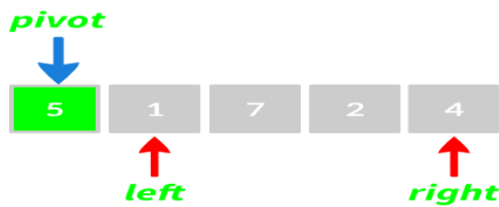


Step 4: Move “right cursor” to the left

Move the right cursor from its current position to the left (one step at a time) — until the element that the cursor points to is smaller than the pivot.

Simply, move the right cursor to the left until you see the first element which is smaller than the PIVOT.

Once you see such an element — STOP there (!) and move on to the next step (Step 5).



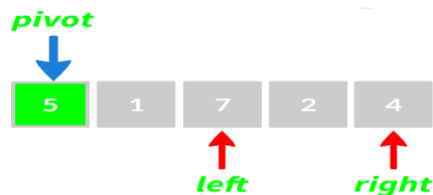
“4” is already smaller than the pivot “5” — so no need to move it now

Note: If at any time during this Step 4 the “right” and “left” cursors overlap — jump to Step 7.

Step 5: Move “left cursor” to the right

Move the left cursor to the right (one step at a time). Until the element that the cursor points to is greater than the pivot.

Moving the left cursor to the right until we encounter with such an element that is greater than the PIVOT Once you see such an element . STOP there (!) and proceed for the next step (Step 6).

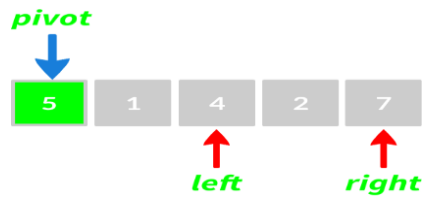


“7” is greater than the pivot

Note: If at any time during this Step 4 the “right” and “left” cursors overlap — jump to Step 7.

Step 6: Swap between “right” and “left” elements

Once the “right cursor” points to the element smaller than the pivot and the “left cursor” points to the element greater than the pivot — SWAP between these two elements.



After Swap between “4” and “7”

So now, after the swap operation, on the “right side”, where the “right cursor” points to — will be the element greater than the pivot and vice versa.

Just to remind, we follow these “Step4”, “Step5” and the “swap” operation in order to “keep” all greater elements (than the pivot) on the right and all smaller elements (than the pivot) on the left.

Once these steps (step 6) are complete — REPEAT the process from Step 4

Step 7: Final Step — Swap with the Pivot

This is the final step of the “Partition Algorithm”.

You should come to this step only when the “right cursor” points to the same element as the “left cursor” — meaning the “left” and “right” overlap.

At this moment, both “right” and “left” cursors point to the same element — meaning we’ve checked all the elements of the collection.

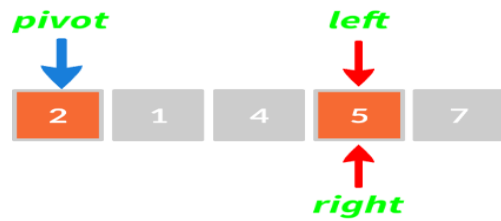


“Left” and “Right” cursors point to the same element

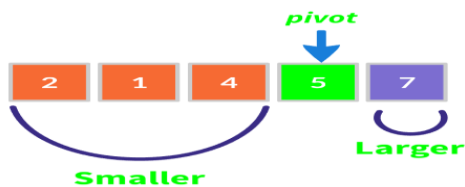
Now we would like to use the “swap” operation for the last time to complete this algorithm.



So we’ll swap between the PIVOT and the element both of the cursors point to.



After Swap with the Pivot



After you've successfully completed Step 7 — you should see that the pivot is exactly at its correct position:

- All elements on the left of the pivot — smaller than the pivot.
- All elements on the right of the pivot — greater than the pivot.

Analysis of Quicksort

The running time of quicksort depends heavily on the selection of the pivot. If the rank of the pivot is very large or very small then the partition (BST) will be unbalanced. Since the pivot is chosen randomly in our algorithm, the expected running time is $O(n \log n)$. The worst case time, however, is $O(n^2)$.

Worst Case Analysis of Quick Sort

Suppose that we are sorting an array of size n , $A[1 : n]$, and further suppose that the pivot that we select is of rank q , for some q in the range 1 to n . It takes $\Theta(n)$ time to do the partitioning and other overhead, and we make two recursive calls. The first is to the subarray $A[1 : q - 1]$ which has $q - 1$ elements, and the other is to the subarray $A[q + 1 : n]$ which has $n - q$ elements. So if we ignore the $\Theta(n)$ (as usual) we get the recurrence:

$$T(n) = T(q - 1) + T(n - q) + n$$

This depends on the value of q . To get the worst case, we maximize over all possible values of q . Putting it together, we get the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \max_{1 \leq q \leq n} (T(q - 1) + T(n - q) + n) & \text{otherwise} \end{cases}$$

More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglhAA1SG3>

Would plug in the value $q = 1$, $q = n$, and $q = n/2$ and work each out. The worst case happens at either of the extremes. If we expand the recurrence for $q = 1$, we get:

$$\begin{aligned} T(n) &\leq T(0) + T(n-1) + n \\ &= 1 + T(n-1) + n \\ &= T(n-1) + (n+1) \\ &= T(n-2) + n + (n+1) \\ &= T(n-3) + (n-1) + n + (n+1) \\ &= T(n-4) + (n-2) + (n-1) + n + (n+1) \end{aligned}$$

$$= T(n-k) + \sum_{i=1}^{k-2} (n-i)$$

For the basis $T(1) = 1$ we set $k = n-1$ and get

$$\begin{aligned} T(n) &\leq T(1) + \sum_{i=1}^{n-3} (n-i) \\ &= 1 + (3+4+5+\dots+(n-1)+n+(n+1)) \\ &\leq \sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2} \in \Theta(n^2) \end{aligned}$$

Chapter 4

14th lecher

Average-case Analysis of Quicksort

The algorithm has n random choices for the pivot element, and each choice has an equal probability of $1/n$ of occurring. So we can modify the above recurrence to compute an average rather than a max, giving:

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \frac{1}{n} \sum_{q=1}^n (T(q-1) + T(n-q) + n) & \text{otherwise} \end{cases}$$

The time $T(n)$ is the weighted sum of the times taken for various choices of q . I.e.,

$$\begin{aligned} T(n) &= \left[\frac{1}{n} (T(0) + T(n-1) + n) \right. \\ &\quad + \frac{1}{n} (T(1) + T(n-2) + n) \\ &\quad + \frac{1}{n} (T(2) + T(n-3) + n) \\ &\quad \left. + \dots + \frac{1}{n} (T(n-1) + T(0) + n) \right] \end{aligned}$$

More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglhAA1SG3>

Attempt a constructive induction to solve it. We know that we want a $\Theta(n \log n)$. Let us assume that $T(n) \leq cn \log n$ for $n \geq 2$ where c is a constant. For the base case $n = 2$ we have

$$\begin{aligned} T(n) &= \frac{1}{2} \sum_{q=1}^2 (T(q-1) + T(2-q) + 2) \\ &= \frac{1}{2} [(T(0) + T(1) + 2) + (T(1) + T(0) + 2)] \\ &= \frac{8}{2} = 4. \end{aligned}$$

For the induction step, we assume that $n \geq 3$ and The induction hypothesis is that for any $n_0 < n$, we have $T(n_0) \leq cn_0 \log n_0$. We want to prove that it is true for $T(n)$. By expanding $T(n)$ and moving the factor of n outside the sum we have

$$\begin{aligned} T(n) &= \frac{1}{n} \sum_{q=1}^n (T(q-1) + T(n-q) + n) \\ &= \frac{1}{n} \sum_{q=1}^n (T(q-1) + T(n-q)) + n \\ &= \frac{1}{n} \sum_{q=1}^n T(q-1) + \frac{1}{n} \sum_{q=1}^n T(n-q) + n \\ T(n) &= \frac{1}{n} \sum_{q=1}^n T(q-1) + \frac{1}{n} \sum_{q=1}^n T(n-q) + n \end{aligned}$$

Same values $T(0) + T(1) + \dots + T(n-1)$. One counts up and other counts down. Thus

we can replace them with $2 \sum_{q=0}^{n-1} T(q)$. We will extract $T(0)$ and $T(1)$ and treat them specially. These two do not follow the formula.

$$\begin{aligned}
 T(n) &= \frac{2}{n} \left(\sum_{q=0}^{n-1} T(q) \right) + n \\
 &= \frac{2}{n} \left(T(0) + T(1) + \sum_{q=2}^{n-1} T(q) \right) + n
 \end{aligned}$$

We will apply the induction hypothesis for $q < n$ we have

$$\begin{aligned}
 T(n) &= \frac{2}{n} \left(T(0) + T(1) + \sum_{q=2}^{n-1} T(q) \right) + n \\
 &\leq \frac{2}{n} \left(1 + 1 + \sum_{q=2}^{n-1} (cq \log q) \right) + n \\
 &= \frac{2c}{n} \left(\sum_{q=2}^{n-1} (cq \ln q) \right) + n + \frac{4}{n}
 \end{aligned}$$

We have never seen this sum before:

$$S(n) = \sum_{q=2}^{n-1} (cq \ln q)$$

Recall from calculus that for any monotonically increasing function $f(x)$

$$\sum_{i=a}^{b-1} f(i) \leq \int_a^b f(x) dx$$

The function $f(x) = x \ln x$ is monotonically increasing, and so

$$S(n) = \sum_{q=2}^{n-1} (cq \ln q) \leq \int_2^n x \ln x dx$$

We can integrate this by parts:

$$\int_2^n x \ln x dx = \frac{x^2}{2} \ln x - \frac{x^2}{4} \Big|_{x=2}^n$$

$$\begin{aligned}
 \int_2^n x \ln x dx &= \frac{x^2}{2} \ln x - \frac{x^2}{4} \Big|_{x=2}^n \\
 &= \left(\frac{n^2}{2} \ln n - \frac{n^2}{4} \right) - (2 \ln 2 - 1) \\
 &\leq \frac{n^2}{2} \ln n - \frac{n^2}{4}
 \end{aligned}$$

We thus have

$$S(n) = \sum_{q=2}^{n-1} (cq \ln q) \leq \frac{n^2}{2} \ln n - \frac{n^2}{4}$$

Plug this back into the expression for $T(n)$ to get

$$T(n) = \frac{2c}{n} \left(\frac{n^2}{2} \ln n - \frac{n^2}{4} \right) + n + \frac{4}{n}$$

$$\begin{aligned}
 T(n) &= \frac{2c}{n} \left(\frac{n^2}{2} \ln n - \frac{n^2}{4} \right) + n + \frac{4}{n} \\
 &= cn \ln n - \frac{cn}{2} + n + \frac{4}{n} \\
 &= cn \ln n + n \left(1 - \frac{c}{2} \right) + \frac{4}{n}
 \end{aligned}$$

$$T(n) = cn \ln n + n \left(1 - \frac{c}{2} \right) + \frac{4}{n}$$

To finish the proof, we want all of this to be at most $c \ln n$. For this to happen, we will need to select c such that

$$n\left(1 - \frac{c}{2}\right) + \frac{4}{n} \leq 0$$

If we select $c = 3$, and use the fact that $n \geq 3$ we get

$$\begin{aligned} n\left(1 - \frac{c}{2}\right) + \frac{4}{n} &= \frac{3}{n} - \frac{n}{2} \\ &\leq 1 - \frac{3}{2} = -\frac{1}{2} \leq 0. \end{aligned}$$

From the basis case we had $c \geq 2.88$. Choosing $c = 3$ satisfies all the constraints. Thus

$$T(n) = 3n \ln n \in \Theta(n \log n).$$

Chapter 4

15th lecher

In-place, Stable Sorting

An in-place sorting algorithm is one that uses no additional array for storage. A sorting algorithm is stable if duplicate elements remain in the same relative position after sorting.

9 3 3' 5 6 5' 2 1 3''	unsorted
1 2 3 3' 3'' 5 5' 6 9	stable sort
1 2 3' 3 3'' 5' 5 6 9	unstable

Bubble sort, insertion sort and selection sort are in-place sorting algorithms. Bubble sort and insertion sort can be implemented as stable algorithms but selection sort cannot. Mergesort is a stable algorithm but not an in-place algorithm. It requires extra array storage. Quicksort is not stable but is an in-place algorithm. Heapsort is an in-place algorithm but is not stable.

Lower Bounds for Sorting

the lower bound for sorting is $\Omega(n \log n)$ for comparison-based sorting algorithms, implying that any algorithm attempting to sort a set of elements by making comparisons must make at least this many comparisons in the worst case.

It's important to note that the lower bound applies specifically to comparison-based sorting algorithms. There exist non-comparison-based sorting algorithms, such as counting sort and radix sort, which can achieve linear time complexity $O(n)$ in certain

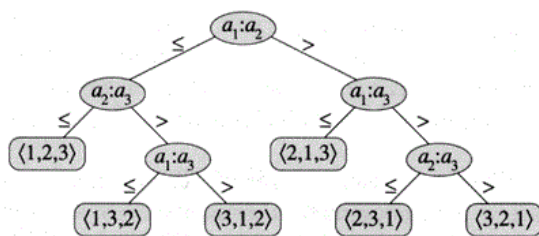
More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglAA1SG3>

scenarios. However, these algorithms have limitations and require specific properties of the input elements.

Consider sorting three numbers a_1, a_2, a_3 . There are $3! = 6$ possible combinations:

$(a_1, a_2, a_3), (a_1, a_3, a_2), (a_3, a_2, a_1)$
 $(a_3, a_1, a_2), (a_2, a_1, a_3), (a_2, a_3, a_1)$

One of these permutations leads to the numbers in sorted order. The comparison based algorithm defines a decision tree. Here is the tree for the three numbers.



The decision tree for insertion sort operating on three elements. There are $3! = 6$ possible permutations of the input elements, so the decision tree must have at least 6 leaves.

Any binary tree of height $T(n)$ has at most $2^{T(n)}$ leaves. Thus a comparison based sorting algorithm can distinguish between at most $2^{T(n)}$ different final outcomes. So we have

$$2^{T(n)} \geq n! \quad \text{and therefore} \\ T(n) \geq \log(n!)$$

We can use Stirling's approximation for $n!$:

$$n! \geq \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

Therefore

$$\begin{aligned} T(n) &\geq \log\left(\sqrt{2\pi n} \left(\frac{n}{e}\right)^n\right) \\ &= \log(\sqrt{2\pi n}) + n \log n - n \log e \\ &\in \Omega(n \log n) \end{aligned}$$

We thus have the following theorem.

Linear Time Sorting

We have sorting algorithms that can sort "n" numbers in $O(n \log n)$ time. Merge Sort and Heap Sort achieve this upper bound in the worst case, and Quick Sort achieves this on Average Case. Merge Sort, Quick Sort and Heap Sort algorithm share an interesting property: the sorted order they determined is based only on comparisons between the input elements. We call such a sorting algorithm "**Comparison Sort**".

There is some algorithm that runs faster and takes linear time such as Counting Sort, Radix Sort, and Bucket Sort but they require the special assumption about the input sequence to sort.

Counting Sort Algorithm

In this article, we will discuss the counting sort Algorithm. Counting sort is a sorting technique that is based on the keys between specific ranges. In coding or technical interviews for software engineers, sorting algorithms are widely asked. So, it is important to discuss the topic.

This sorting technique doesn't perform sorting by comparing elements. It performs sorting by counting objects having distinct key values like hashing. After that, it performs some arithmetic operations to calculate each object's index position in the output sequence. Counting sort is not used as a general-purpose sorting algorithm.

Counting sort is effective when range is not greater than number of objects to be sorted. It can be used to sort the negative input values.

Now, let's see the algorithm of counting sort.

Algorithm

- I. `countingSort(array, n)` // 'n' is the size of array
- II. `max` = find maximum element in the given array
- III. create count array with size `maximum + 1`
- IV. Initialize count array with all 0's
- V. **for** `i = 0` to `n`
- VI. find the count of every unique element and
- VII. store that count at `i`th position in the count array
- VIII. **for** `j = 1` to `max`
- IX. Now, find the cumulative sum and store it in count array
- X. **for** `i = n` to `1`

- XI. Restore the array elements
- XII. Decrease the count of every restored element by 1
- XIII. end countingSort

Working of counting sort Algorithm

To understand the working of the counting sort algorithm, let's take an unsorted array. It will be easier to understand the counting sort via an example.

Let the elements of array are -

2	9	7	4	1	8	4
---	---	---	---	---	---	---

1. Find the maximum element from the given array. Let **max** be the maximum element.

max						
9	2	7	4	1	8	4

2. Now, initialize array of length **max + 1** having all 0 elements. This array will be used to store the count of the elements in the given array.

0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---

3. Count array

Now, we have to store the count of each array element at their corresponding index in the count array.

The count of an element will be stored as - Suppose array element '4' is appeared two times, so the count of element 4 is 2. Hence, 2 is stored at the 4th position of the count array. If any element is not present in the array, place 0, i.e. suppose element '3' is not present in the array, so, 0 will be stored at 3rd position.

Given array	2	9	7	4	1	8	4
-------------	---	---	---	---	---	---	---

	0	1	2	3	4	5	6	7	8	9
Count array	0	1	1	0	2	0	0	1	1	1

Count of each stored element

Now, store the cumulative sum of **count** array elements. It will help to place the elements

	0	1	2	3	4	5	6	7	8	9
	0	1	2	0	2	0	0	1	1	1

at the correct index of the sorted array.

	0	1	2	3	4	5	6	7	8	9
	0	1	2	2	2	0	0	1	1	1

2+0=2

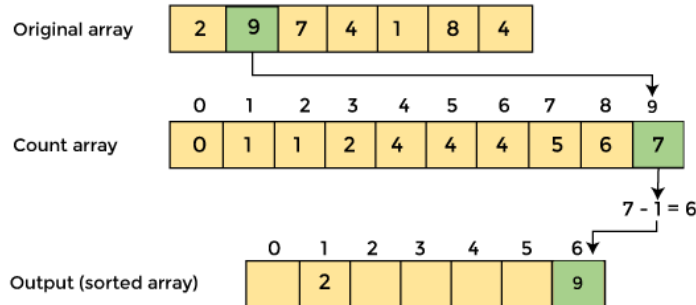
Similarly, the cumulative count of the count array is -

0	1	2	3	4	5	6	7	8	9
0	1	2	2	4	4	4	5	6	7

Cumulative count

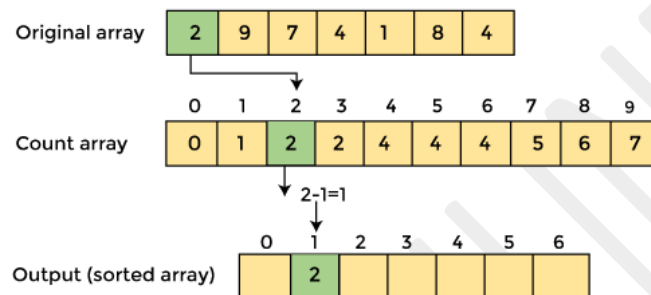
4. Now, find the index of each element of the original array

For element 9



After placing element at its place, decrease its count by one. Before placing element 2, its count was 2, but after placing it at its correct position, the new count for element 2 is 1.

For element 2



Similarly, after sorting, the array elements are -

Output (sorted array)

1	2	4	4	7	8	9
---	---	---	---	---	---	---

Now, the array is completely sorted.

Chapter 4

16th lecher

Bucket Sort Algorithm

The data items in the bucket sort are distributed in the form of buckets. In coding or technical interviews for software engineers, sorting algorithms are widely asked. So, it is important to discuss the topic.

Bucket sort is a sorting algorithm that separate the elements into multiple groups said to be buckets. Elements in bucket sort are first uniformly divided into groups called buckets,

More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglAA1SG3>

and then they are sorted by any other sorting algorithm. After that, elements are gathered in a sorted manner.

The basic procedure of performing the bucket sort is given as follows -

- First, partition the range into a fixed number of buckets.
- Then, toss every element into its appropriate bucket.
- After that, sort each bucket individually by applying a sorting algorithm.
- And at last, concatenate all the sorted buckets.

The advantages of bucket sort are

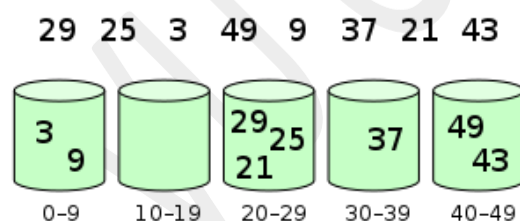
- Bucket sort reduces the no. of comparisons.
- It is asymptotically fast because of the uniform distribution of elements.

The limitations of bucket sort are

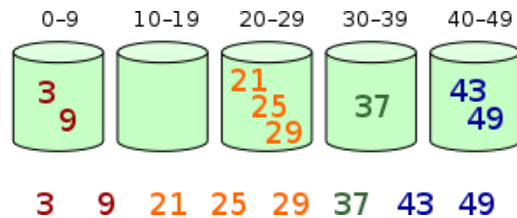
- It may or may not be a stable sorting algorithm.
- It is not useful if we have a large array because it increases the cost.
- It is not an in-place sorting algorithm, because some extra space is required to sort the buckets.
- The best and average-case complexity of bucket sort is $O(n + k)$, and the worst-case complexity of bucket sort is $O(n^2)$, where n is the number of items.

Bucket sort is commonly used

- With floating-point values.
- When input is distributed uniformly over a range.



Elements are distributed among bins



Then, elements are sorted within each bin

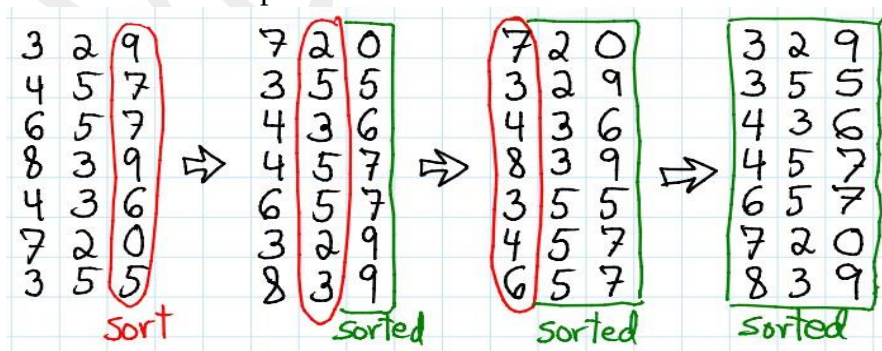
Radix Sort Algorithm

Radix sort is the linear sorting algorithm that is used for integers. In Radix sort, there is digit by digit sorting is performed that is started from the least significant digit to the most significant digit.

The process of radix sort works similar to the sorting of students names, according to the alphabetical order. In this case, there are 26 radix formed due to the 26 alphabets in English. In the first pass, the names of students are grouped according to the ascending order of the first letter of their names. After that, in the second pass, their names are grouped according to the ascending order of the second letter of their name. And the process continues until we find the sorted list.

Algorithm

1. radixSort(arr)
2. max = largest element in the given array
3. d = number of digits in the largest element (or, max)
4. Now, create d buckets of size 0 - 9
5. **for** i -> 0 to d
6. sort the array elements using counting sort (or any stable sort) according to the digit at the i^{th} place



Fibonacci sequence

The Fibonacci sequence is a set of integers (the Fibonacci numbers) that starts with a zero, followed by a one, then by another one, and then by a series of steadily increasing numbers. The sequence follows the rule that each number is equal to the sum of the preceding two numbers.

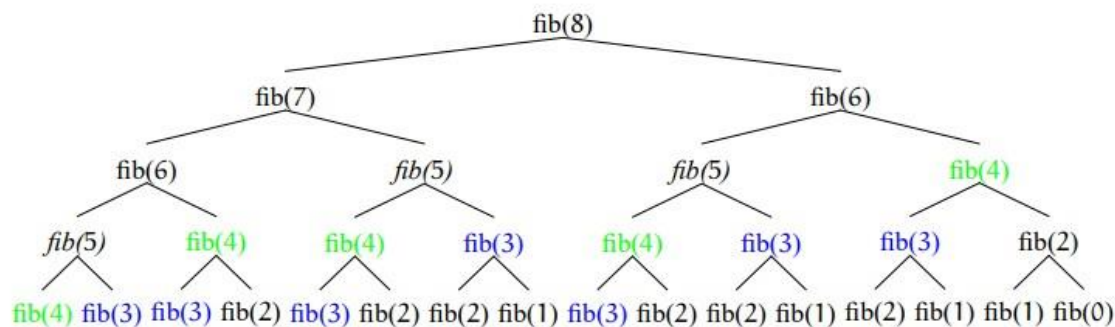
The Fibonacci sequence begins with the following 14 integers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233...

Each number, starting with the third, adheres to the prescribed formula. For example, the seventh number, 8, is preceded by 3 and 5, which add up to 8.

Fibonacci sequence Rule

- $x_n = x_{n-1} + x_{n-2}$
where:
- x_n is term number "n"
- x_{n-1} is the previous term (n-1)
- x_{n-2} is the term before that (n-2)



Recursive calls during computation of Fibonacci number

Memorization

We can avoid these unnecessary repetitions by writing down the results of recursive calls and looking them up again if we need them later. This process is called memorization.

Dynamic Programming

Dynamic programming is essentially recursion without repetition. Developing a dynamic programming algorithm generally involves two separate steps:

- Formulate problem recursively. Write down a formula for the whole problem as a simple combination of answers to smaller sub problems.
- Build solution to recurrence from bottom up. Write an algorithm that starts with base cases and works its way up to the final solution.

Dynamic programming algorithms need to store the results of intermediate sub problems.

Chapter 5

17th lecher

Edit Distance

In computational linguistics and computer science, edit distance is a string metric, i.e. a way of quantifying how dissimilar two strings (e.g., words) are to one another that is measured by counting the minimum number of operations required to transform one string into the other.

Where a point mutation is one of:

- change a letter,
- insert a letter or
- delete a letter

Edit Distance: Applications

Edit Distance, or Levenshtein Distance, has numerous applications in various fields. Here are some common applications:

➤ Spelling Correction:

Edit Distance is commonly used in spelling correction systems. By comparing a misspelled word with a dictionary of correctly spelled words, the algorithm can suggest the most likely correct word based on the minimum number of operations required to transform the misspelled word into a valid word.

➤ DNA Sequence Alignment:

Edit Distance is employed in bioinformatics to align DNA sequences. It helps identify similarities and differences between genetic sequences, enabling researchers to study genetic mutations, evolutionary relationships, and gene function.

➤ **Plagiarism Detection:**

Edit Distance can be used in plagiarism detection systems to compare and measure the similarity between two pieces of text. By calculating the edit distance between texts, it becomes possible to identify similarities and potential instances of plagiarism.

➤ **Natural Language Processing (NLP):**

In NLP applications, Edit Distance is used for tasks like automatic summarization, machine translation, and named entity recognition. It can help in identifying similar words or phrases between different languages or summarizing text by finding the most relevant sentences.

➤ **OCR (Optical Character Recognition) Error Correction:**

Edit Distance plays a crucial role in OCR systems. It assists in correcting errors by comparing recognized text with the original document. The algorithm determines the minimum number of operations required to transform the recognized text into the correct text.

Chapter 5

18th lecher

Edit Distance Algorithm

The algorithm works by constructing a matrix where the rows represent the characters of the first string and the columns represent the characters of the second string. Each cell of the matrix represents the minimum edit distance between the substrings formed by the characters up to that cell.

To compute the minimum edit distances, we start by initializing the first row and column of the matrix. The first row represents the transformation of an empty string to the second string, and the first column represents the transformation of the first string to an empty string. The values in the first row and column are incremented sequentially from 0 to the length of the respective strings.

The time complexity of the Edit Distance algorithm is $O(m \cdot n)$, where m and n are the lengths of the input strings. This is because we need to fill in the entire matrix of size $(m+1) \times (n+1)$. The space complexity is also $O(m \cdot n)$ as we need to store the matrix.

Edit Distance: Dynamic Programming

The algorithm utilizes a dynamic programming table to compute the minimum edit

distances efficiently. The table is a 2D matrix where each cell represents the minimum edit distance between substrings of the two input strings.

To calculate the edit distance, we initialize the first row and first column of the table. The first row represents the transformation from an empty string to the characters of the second string, and the first column represents the transformation from the characters of the first string to an empty string. The values in the first row and first column are incremented sequentially from 0 to the length of the respective strings.

We then iterate through the remaining cells of the table, filling them based on the following rules:

- If the characters at the current positions in both strings are equal, we take the value from the diagonal cell (top-left).
- If the characters are different, we consider three operations:
 - a) Deletion: We take the value from the cell above (representing deleting a character from the first string).
 - b) Insertion: We take the value from the cell on the left (representing inserting a character into the first string).
 - c) Substitution: We take the value from the diagonal cell and increment it by 1 (representing substituting a character in the first string with a character from the second string).
- After filling all the cells, the bottom-right cell of the table represents the minimum edit distance between the two strings.

The time complexity of the Edit Distance algorithm is $O(m * n)$, where m and n are the lengths of the input strings. This is because we need to fill in the entire matrix of size $(m+1) \times (n+1)$. The space complexity is also $O(m * n)$ as we need to store the matrix.

By using the Edit Distance algorithm, we can solve a variety of problems that involve measuring the similarity or difference between strings, such as spell checking, DNA sequence alignment, plagiarism detection, and more. The dynamic programming approach allows for an efficient computation of the edit distance, making it widely applicable in many domains.

Chapter 5

19th lecher

Analysis of DP Edit Distance

More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglhAA1SG3>

There are $\Theta(n^2)$ entries in the matrix. Each entry $E(i, j)$ takes $\Theta(1)$ time to compute. The total running time is $\Theta(n^2)$.

Matrix chain multiplication

Matrix chain multiplication (or the matrix chain ordering problem) is an optimization problem concerning the most efficient way to multiply a given sequence of matrices. The problem is not actually to perform the multiplications, but merely to decide the sequence of the matrix multiplications involved. The problem may be solved using dynamic programming.

A $p \times q$ matrix A can be multiplied with a $q \times r$ matrix B . The result will be a $p \times r$ matrix C . In particular, for $1 \leq i \leq p$ and $1 \leq j \leq r$,

$$C[i, j] = \sum_{k=1}^q A[i, k]B[k, j]$$

There are $(p \cdot r)$ total entries in C and each takes $O(q)$ to compute.

$$C[i, j] = \sum_{k=1}^q A[i, k]B[k, j]$$

items:

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

This is related to a famous function in combinatorics called the *Catalan numbers*. Catalan numbers are related the number of different binary trees on n nodes. Catalan number is given by the formula:

$$C(n) = \frac{1}{n+1} \binom{2n}{n}$$

In particular, $P(n) = C(n-1)$ $C(n) \in \Omega(4^n/n^{3/2})$ The dominating term is the exponential 4^n thus $P(n)$ will grow large very quickly. So this approach is not practical.

There are many options because matrix multiplication is associative. In other words, no matter how the product is parenthesized, the result obtained will remain the same. For example, for four matrices A , B , C , and D , there are five possible options:

$$((AB) C)D = (A (BC)) D = (AB) (CD) = A ((BC) D) = A (B (CD)).$$

Although it does not affect the product, the order in which the terms are parenthesized affects the number of simple arithmetic operations needed to compute the product, that is, the computational complexity. The straightforward multiplication of a matrix that is $X \times Y$ by a matrix that is $Y \times Z$ requires XYZ ordinary multiplications and $X(Y-1) Z$

Ordinary additions. it is typical to use the number of ordinary multiplications as a measure of the runtime complexity.

If A is a 10×30 matrix, B is a 30×5 matrix, and C is a 5×60 matrix, then

- computing (AB)C needs $(10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500$ operations, while
- Computing A (BC) needs $(30 \times 5 \times 60) + (10 \times 30 \times 60) = 9000 + 18000 = 27000$ operations.

Chapter 5

20th lecher

Chain Matrix Multiplication-Dynamic Programming Formulation

The Chain Matrix Multiplication problem involves finding the most efficient way to multiply a chain of matrices. The goal is to determine the order in which to perform the matrix multiplications so as to minimize the total number of scalar multiplications required.

To solve this problem using dynamic programming, we can use a table to store the intermediate results and gradually build up the solution. Let's assume we have a chain of matrices represented by dimensions $[d_0, d_1, d_2, \dots, d_n]$. Here, the matrix A_i has dimensions $d[i-1] \times d[i]$ for $i = 1$ to n .

We can define a 2D table, M, of size $n \times n$, where $M[i][j]$ represents the minimum number of scalar multiplications needed to compute the matrix product $A[i \dots j]$. The table is initially filled with zeros.

1. To fill in the table, we use the following dynamic programming formulation:
 - For each chain length $l = 2$ to n ($l = j - i + 1$):
 - For each starting matrix $i = 1$ to $n - l + 1$:
 - Compute the corresponding ending matrix $j = i + l - 1$.
 - Initialize $M[i][j]$ with a large value.
 - For each $k = i$ to $j - 1$:
 - Compute the number of scalar multiplications for the left and right subchains:
 - $\text{left_mult} = M[i][k]$
 - $\text{right_mult} = M[k+1][j]$
 - Compute the total number of scalar multiplications for the current split:

- $\text{current_mult} = \text{left_mult} + \text{right_mult} + d[i-1] * d[k] * d[j]$
 - Update $M[i][j]$ with the minimum value between current_mult and $M[i][j]$.
2. The final result is stored in $M[1][n]$, which represents the optimal number of scalar multiplications needed to compute the full chain product.

By applying this dynamic programming approach, we can efficiently solve the Chain Matrix Multiplication problem and obtain the minimum number of scalar multiplications required for multiplying the given chain of matrices.

The time complexity of this algorithm is $O(n^3)$, where n is the number of matrices in the chain. The space complexity is also $O(n^2)$ for the table M .

Chapter 6

21th lecher

0/1 Knapsack problem

Here knapsack is like a container or a bag. Suppose we have given some items which have some weights or profits. We have to put some items in the knapsack in such a way total value produces a maximum profit.

For example, the weight of the container is 20 kg. We have to select the items in such a way that the sum of the weight of items should be either smaller than or equal to the weight of the container, and the profit should be maximum.

There are two types of knapsack problems:

- 0/1 knapsack problem
- Fractional knapsack problem

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely.

Example of 0/1 knapsack problem

Consider the problem having weights and profits are:

Weights: {3, 4, 6, 5}

- Profits: {2, 3, 1, 4}

The weight of the knapsack is 8 kg

The number of items is 4

The above problem can be solved by using the following method:

- $x_i = \{1, 0, 0, 1\}$
- $x_i = \{0, 0, 0, 1\}$
- $x_i = \{0, 1, 0, 1\}$

The above are the possible combinations. 1 denotes that the item is completely picked and 0 means that no item is picked since there are 4 items so possible combinations will be $2^4 = 16$ so. Here are 16 possible combinations that can be made by using the above problem. Once all the combinations are made, we have to select the combination that provides the maximum profit.

0/1 Knapsack Problem: Dynamic Programming Approach

If $B[i][j] = B[i-1][j]$, the package i is not selected. $B[n][W]$ is the optimal total value of package put into the knapsack.

Steps for tracing:

- Step 1: Starting from $i = n, j = M$.
- Step 2: Look in column j , up from bottom, you find the line i such that $B[i][j] > B[i-1][j]$. Mark selected package i : Select $[i] = \text{true}$;
- Step 3: $j = B[i][j] - W[i]$. If $j > 0$, go to step 2, otherwise go to step 4
- Step 4: Based on the table of options to print the selected packages.

Chapter 6

22th lecher

Constructing the Optimal Solution

The algorithm for computing $V[i, j]$ does not keep record of which subset of items gives the optimal solution. To compute the actual subset, we can add an auxiliary Boolean array $keep[i, j]$ which is 1 if we decide to take the i^{th} item and 0 otherwise. We will use all the values $keep[i, j]$ to determine the optimal subset T of items to put in the knapsack as follows:

More Files Join VU Group: <https://chat.whatsapp.com/BL4ok4jnM9iJiglhAA1SG3>

- If $\text{keep}[n, W]$ is 1, then $n \in T$. We can now repeat this argument for $\text{keep}[n - 1, W - w_n]$.
- If $\text{keep}[n, W]$ is 0, then $n \notin T$ and we repeat the argument for $\text{keep}[n - 1, W]$.

KNAPSACK (n, W)

1. for $w = 0, W$
2. do $V[0, w] \leftarrow 0$
3. for $i = 0, n$
4. do $V[i, 0] \leftarrow 0$
5. for $w = 0, W$
6. do if ($w_i = w$ & $v_i + V[i - 1, w - w_i] > V[i - 1, w]$)
7. then $V[i, w] \leftarrow v_i + V[i - 1, w - w_i]$; $\text{keep}[i, w] \leftarrow 1$
8. else $V[i, w] \leftarrow V[i - 1, w]$; $\text{keep}[i, w] \leftarrow 0$
9. // output the selected items
10. $k \leftarrow W$
11. for $i = n$ downto 1
12. do if ($\text{keep}[i, k] = 1$)
13. then output i
14. $k \leftarrow k - w_i$