Maha Mamdot

INTRO TO ALGORITHM

</>

Algorithms

① Intro

→ Sorting : to rearrange the elements of a collection into ascending / descending order.

↳ helps to make searching easier since elements in an array are organised.

Types of sorting :-

1) Selection sort
2) Insertion sort
3) Bubble sort
4) Quick sort
5) merge sort
6) linear search
7) binary search

- Selection sort : pick 1 element in the array and swap it with another element. Aim is to sort either in ascending / descending order.

Step 1 : 7 5 4 2 (swap) → 2 | 5 4 7
↳ min element is the smallest element in the array
2 : sorted, 5 4 7 : unsorted

Step 2: 2 5 4 7 (swap) → 2 4 | 5 7
↳ min
2 4 : sorted, 5 7 : unsorted

Step 3: 2 4 5 7 → 2 4 5 | 7
↳ min
2 4 5 : sorted, 7 : unsorted

↳ sort will keep continuing until all elements have had their turns.

Step 4: 2 4 5 7 → 2 4 5 7
↳ min
2 4 5 7 : sorted ✓

pseudocode:

```
for (j = 0; j < n-1; j++)
    int iMin = j;
    for (i = j+1; i < n; i++)
        if (a[i] < a[iMin])
            iMin = i;
        If (iMin != j)
            swap(a[j], a[iMin]);
```

complexity = $O(n^2)$.

- Insertion sort: start from the beginning and compare to the element on the left to see if its bigger or smaller. (straightforward)

First pass: 23 1 10 5 2 → 23 1 10 5 2
↳ no element on the left so remains the same.

second pass: 23 1 10 5 2 → 1 23 10 5 2

third pass: 1 23 10 5 2 → 1 10 23 5 2

fourth pass: 1 10 23 5 2 → 1 5 10 23 2
- 5 is less than 23 so move to the next element 10 and swap with that.
if 5 was greater than 23 then it would not be swapped.
order is ascending. (making sure its in the correct position)

fifth pass: 1 5 10 23 2 → 1 2 5 10 23
Sorted ✓

pseudocode:

```
for i: 1 to length(A)-1
    j = i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j = j-1
```

complexity = $O(n^2)$; worst case time complexity.

Bubble sort

2 8 5 3 9 41 → 2 5 8 3 9 41

2 5 8 3 9 41 → 2 5 3 8 9 41

correct positions

2 5 3 8 9 41 → 2 3 5 8 9 41

sorted ✓

pseudocode:

```
for i from 1 to N
    for j from 0 to N-1
        if a[j] > a[j+1]
            swap (a[j], a[j+1])
```

complexity = $O(n^2)$; worst case.

→ Complexity of an algorithm is its cost expressed as a function of the size of the input.

↳ Time complexity : running time

Space complexity : memory usage

Time complexity

↙ ↘

Worst case

↳ function that performs the max number of steps on input data of size (n).

Best case

↳ function that performs the min number of steps on input data of size (n).

→ larger inputs require more time.
→ we prefer to work with worst case complexity time is because:
1) need to know the max steps on input data.
2) much easier to work out.
3) closely related to the average case.

- Big O notation

→ describes the limiting behaviour of a function when the argument tends towards a particular value.
↳ aka asymptotic complexity.
↳ simplifies calculations of algorithms efficiency.
↳ analyse both time and space.

Different types of Time complexities:-

1) O(1) → constant time
↳ independant of input size (n) e.g:
```
x = 5+(15x20)
y = 15-2
print x+y
```
total time: O(1)+O(1)+O(1)
= 3xO(1) = O(1) ans
↳ drop constant
↳ takes same amount of time always.

2) O(n) → linear time e.g:
```
y = 5+(15x20);        → O(1)
for x in range (0,n):  } → O(n)
    print x;
```
total time: O(1)+O(n) = O(n) ans
↳ used in sequential search. time taken is directly prop to input size.

3) $O(n^2)$ → Quadratic time e.g:
↳ used in nested loops or sorting methods.
```
y = 5+(15x20);
for x in range (0,n):
    print x;
    for y in range (0,n):  } → O(n^2)
    for x in range (0,n):  }
        print xy;
```

4) $O(\log n)$: logarithmic time
5) $O(2^n)$: exponential time
6) $O(n^3)$: cubic time
7) $O(n \log n)$: linearithmic time
8) $O(n!)$: Factorial time
9) $O(n^k)$: polynomial function. k: largest power of n that occurs.

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$

Tractability : a problem is called tractable if there is an efficient algorithm that solves it.
Intractability : a problem is called intractable if there is no efficient algorithm that can solve it.
Infeasibility : key to security.

② Iteration

↳ repeating steps over and over again (looping).
→ Loop invariant : used for demonstrating partial correctness
↳ seen before and after execution of the loop.
↳ when algorithm terminates, it produces a desired result.
→ Loop control variables : keeps track of when to exit the loop.
→ Order of growth (big O notation) : to measure complexity.

establish invariant
while condition
  we have; invariant and condition
    loop body
  establish : invariant
→ on exiting the loop one of these must be true:
1) $i = n$ and x is not in the array
2) $0 \leq i < n$ and $a[i] = x$ and x is not in $a(0)$..

```
max ← a[0]
i ← 1
establish invariant : 1 ≤ i ≤ n
while i < n
  we have : 1 ≤ i < n
  if a[i] > max
    max ← a[i]
  i ← i+1
  establish : 1 ≤ i < n
  we have : i = n
return max
```

- Nested loops: loop within a loop.
  ↳ we use loop control variables to keep track.
  → we can use upper bound on the number of remaining iterations as a function of the value of the loop control variable.

- To analyse algorithms we look at:
1) input size: how many elements there are.
2) counting steps: number of steps the algorithm takes.
3) comparing cost functions: cost of the function. smaller cost is better.
4) order of growth notation: relation b/w the input size and the time of growth.

(3) Simple Sorting Methods

→ Adaptive sort: performs better on partially sorted inputs and takes O(n) time on sorted or almost sorted inputs.

↳ insertion sort is adaptive because if the initial array is already ordered the algorithm only does n-1 which is O(n).

```
i ← 1
while i < n
    tmp ← a[i]
    j ← i
    while j > 0 and a[j-1] > tmp
        a[j] ← a[j-1]
        j ← j-1
    a[j] ← tmp
    i ← i+1
```

array which holds an element that will be moved along the array.

→ Stable sort: preserves the relative ordering of elements that compare as equal.

↳ you can sort in first name and surname but then sort them in surname - first name order.

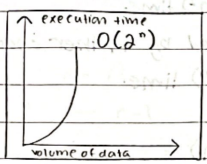
↳ selection sort is neither adaptive or stable because order of elements can change.

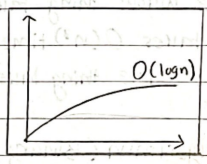
```
i ← 0
while i < n-1
    min ← i
    j ← i+1
    while j < n
        if a[j] < a[min]
            min ← j
        j ← j+1
    swap (a, i, min)
    i ← i+1
```

(4) Exponential and logarithmic complexity

→ exponential time: very bad, found in algorithms that perform an exhausted search.

→ logarithmic time: very good, found in algorithms that cut size of the problem at each step.





→ Satisfiability checking problem is to determine whether, given a formula, there is an assignment of truth values to the variables that make it true.

↳ If any table gives true then that problem will be satisfiable.

→ 2^n = rows

$2^3 = 2 \times 2 \times 2 = 8$ rows in table.

→ for subset sum algorithms, the largest one fits the best.

e.g weights: 23, 39, 57 and capacity = 100.

selection	sum
23+39+57	0
23+39+57	57
23+39+57	39
23+39+57	23
23+39+57	96 ✓ best
23+39+57	80
23+39+57	62
23+39+57	119 X too big

- $2^a \times 2^b = 2^{a+b}$
- $(2^a)^b = 2^{a \times b}$

- Function

→ f(n) belongs to complexity class $O(g(n))$ if there is a size N and a constant c such that for all $n > N$, $f(n) \leq c\, g(n)$.

N = only the behaviour for large n matters.

C = we ignore the constant factors.

$$O(1) \subset O(n) \subset O(n^2) \subset O(n^3) \subset \cdots \subset O(n^k) \subset \cdots \subset O(2^n)$$

- Operations on order function

↳ Simplifies order notation. One common operation arises when an algorithm does one thing after another.

e.g: if $f_1(n)$ and $f_2(n)$ are both $O(g(n))$, then $f_1(n) + f_2(n)$ is also $O(g(n))$

e.g 2: $4n^3 + 6n^3$ is $O(n^3)$ because both functions are $O(n^3)$.

eg 3: $3n^{10} + 2^n$ is $O(2^n)$ because both functions are $O(2^n)$ the faster growing 2^n dominates n^{10}.

★ if a method takes $O(n^2)$ time followed by another operation taking $O(n^2)$ time, the whole thing takes $O(n^2)$ time.

★ if an operation takes $O(n^3)$ time followed by another operation taking $O(2^n)$ time, the whole thing takes $O(2^n)$ time.

- Logarithmic Algorithm (Binary Search)

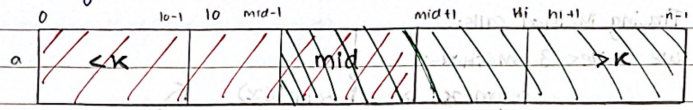
→ uses the rule of divide and conquer and requires the array to be sorted before searching.

↳ it splits the array from the middle then checks which half to look at next and then repeats this till it has found the element its searching for.

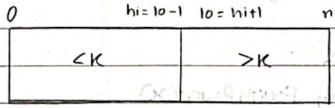
→ Binary search uses:-

1) 0-n numbers of elements.
2) 'Mid' which shows the middle of the array so it can be halved.
3) Lo = less than the mid value.
4) Hi = greater than the mid value.

} control variables.



- if mid < K then eliminate everything from mid → 0 (red)
- if mid > K then eliminate everything from mid → n-1 (green)
- if K and mid are equal then we have found the entry.
- if the key were looking for isn't in the array then both lo and hi will be together.




```
lo ← 0
hi ← n-1
  while lo ≤ hi
      mid ← (lo+hi) DIV 2
      IF a[mid] = K
          return mid
      IF a[mid] < K
          lo ← mid+1
      else
          hi ← mid-1
  return -1
```

Analysing BSA:-

↳ 2 control variables (lo and hi)

↳ size of the input = length of the list n.

↳ on each iteration of loop we halve hi-lo+2 until it reaches 1.

↳ divide and conquer is a good time complexity O(log n)

→ only takes one comparison each time.

⑤ Recursive Algorithms

→ recursion is a process through which a function calls itself directly or indirectly again and again. aka divide and conquer approach.

→ to break larger problem into smaller and smaller ones.

- Tracing Method calls.

→ we have 3 methods;

```
return x * x   } square(x)

y ← square(x)  } cube(x)
Return y * x

y ← cube(x)    } FourthPower(x)
Return y * x.
```

cube(x) calls square(x)

FourthPower(x) calls cube(x)

Suppose $x = 2$

step 1: start from FourthPower(x)

$y = cube(x) \rightarrow y = cube(2)$

step 2: call cube(x)

$y = square(x) \rightarrow y = square(2)$

step 3: call Square(x)

it returns $x * x \rightarrow 2 \times 2 = 4$

$\therefore square(2) = 4$

step 4: Back to cube(x)

$y = square(x) \rightarrow y = square(2) \therefore y = 4.$

it returns $y \times x \rightarrow 4 \times 2 = 8$

step 5: Back to fourthPower(x)

$y = 8$ so returns $8 \times 2 = 16$ ans.

- Implementation of methods (the stack)

→ During the execution of a method, the runtime system stores:-

a) value of the parameters passed to the method.

b) the return address, location in the program from where the method was called.

c) local variables defined inside the method.

d) other house keeping information.

↳ These are called an activation record or stack frame.

Parameters return address local variables	activation record for FourthPower(x)
Parameters RA LV	// // for Cube(x)
Parameters RA LV	// // // Square(x)

↳ when a method is called, a new activation record is added to the stack.

↳ when a method returns, its activation record is discarded, so that the stack shrinks.

↳ objects created with 'new' are stored in a different area called a heap.

- Computing Powers

* eg: Power(2, 3). x = 2 K = 3

```
IF K ≤ 0
    return 1
ELSE
    y ← Power(x, K-1)
    return x x y
```

y = Power(x, K-1) → y = Power(2, 2^(3-1))

y = // // → y = Power(2, 2^(2-1))

y = // // → y = Power(2, 0)

↳ K = 0

- y = Power(2,0) → y = 1 because K = 0
- y = Power(2,1) → y = 1 so x x y = 2x1 = 2
- y = Power(2,2) → y = 2 so x x y = 2x2 = 4
- y = Power(2,3) → y = 4 so 2x4 = 8 ∴ computed $2^3 = 8$ ans

↳ stack implementation :-

x : 2, K : 3 } Power (2,3)
rA

x : 2, K : 2 } Power (2,2)
rA

x : 2, K : 1 } Power (2,1)
rA

x : 2, K : 0 } Power (2,0)
rA

complexity : O(K) time
O(K) space (amount of stack required at deepest point)

→ recursion is an iteration when it calls itself, this is called tail recursion and is equivalent to a loop.

e.g :

```
pow ← 1
i ← 0
while i < K
    pow ← pow x x
    i ← i+1
return pow.
```

↳ iterative power.

complexity : O(K) time
O(1) space

★ iterative algorithm is superior because it avoids using O(n) stack space. but recursive algorithms are preferred more.

★ $x^0 = 1$

$x^{2K} = x^K \times x^K$

$x^{2K+1} = x^K \times x^K * x$

- Divide and conquer strategy

→ if the input is small then compute the answer using a direct method,
otherwise : a) split the problem into smaller ones
b) solve each subproblem recursively
c) combine the solution.

e.g: closest pair problem.
2D space of points, find the a closest points to each other.

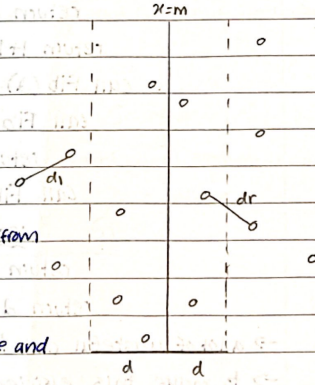
↳ we loop through each point and check it with each point.

```
best ← something huge
For p in P
  For q in P - p
    IF dist(p,q) < best
        first ← p
        second ← q
        best ← dist(p,q)
  Return (p, q, best)
```

Complexity : $O(n^2)$ time which is very long when using large data.

→ so instead of this we can use divide and conquer algorithm.

```
IF P is small
    Return simpleClosestPoint(P)
m ← x-coordinate dividing P in half
d1 ← FastClosestPoints (left half of P)
dr ← // (right half of P)
d ← smaller of d1 and dr
check if the middle strip has two points from
opposite halves that are closer than d.
Return closest distance.
```



→ divides data set in 2 parts. compares each side and solve to find the closest point.

→ total time. = $O(n \log n)$.

- Excessive Function

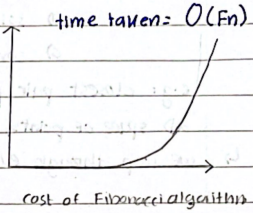
→ Fibonacci function : each term is the sum of the two before it;

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$, for $n \geq 1$.

```
IF n = 0
    Return 0
IF n = 1
    Return 1
Return Fibonacci (n-1) + Fibonacci (n-2)
```




```
Fib (4)
  call Fib (3)
    call Fib (2)
      call Fib (1)
        return 1
      call Fib (0)
        return 0
      return 1+1 = 2
    call Fib (2)
      call Fib (1)
        return 1
      call Fib (0)
        return 0
      return 1+0 = 1
    return 2+1 = 3
```

→ a lot of wasteful calls/calculations.

→ to make this efficient use a strategy called dynamic programming where you solve first for the small values and remember the answers so we don't need to compute.

⑥ Divide and Conquer Sorting

→ recursively breaks down a problem into 2 or more sub problems of the same or related type until these become simple enough to be solved directly.

→ asymptotic refers to what happens when the input size becomes large which we express using big O notation.

. Divide and conquer is seen in:-

1) Quick sort : divide phase does most of the work and the combine phase is trivial

2) Merge sort: divide phase is trivial, most of the work is done in the combine phase.

. Quick Sort

↳ we divide the array in two and this is done by :-

1. pick an element and re-arrange the array so that all the elements smaller than the chosen element are moved to left.

2. The larger elements on the right.

↳ this is called partitioning the array and chosen element is called the pivot.

↳ complexity of $O(n \log n)$ — logarithmic linear.

↳ worst case complexity when ordering an array that is already ordered and will take $O(n^2)$.

lo	0	1	2	3	4	5	6	7	8	9	hi
a	10	16	8	12	15	6	3	9	5	infinite	
pivot = i	↳pivot									j	

→ element greater than lo increment. Decrement j until element is smaller

→ >lo ... → < infinite

10 16i 8 12 15 6 3 9 5j

swap

10 5 8 12i 15 6 3 9j 16

>lo, <inf

swap

10 5 8 9 15i 6 3j 12 16

both i and j so swap with pivot

10 5 8 9 3 6j 15i 12 16 → both i and j crossed so we swap j with pivot

lo infinite 6 5 8 9 3 10 15 12 16 >lo

e.g 2 · smaller items = left

larger items = right

· 3 = pivot point

2 6 5 0 8 7 1 3 — false = swap

(6 < 3 ×, 5 ×, 8 > 3 ×)

first item from left — first item from right

swap

j — i

2 1 5 0 8 7 6 3

(5 < 3 ×, 0 < 3 ✓, 8 > 3 ✓, 7 > 3 ✓, 6 > 3 ✓)

left — right (less than three)

swap

2 1 0 5 8 7 6 3

(5 < 3 ×, 0 < 3 ✓, 8 > 3 ✓, 7 > 3 ✓, 6 > 3 ✓)

item from left (5) has greater index than right (0) so we know its sorted ∴ swap with pivot 3

→ stop when index of item from left > index of item from right.

↳ and swap left with pivot.

2 1 0 3 8 7 6 5

conditions need to be satisfied to see if answer is correct::

1) correct position in final, sorted array.
2) items on the left are smaller than pivot value.
3) " " " right are larger " " ".

→ ~~because we said its recursive we can take the items on the left and organise.~~

→ by recursive we can continue sorting and arrange in an organised manner.

Rough: larger = right

smaller = left

j — i

2 6 5 0 8 7 1 3

(6 < 3 ×, 5 < 3 ×, 0 < 3 ✓, 8 > 3 ✓, 7 > 3 ✓, 1 > 3 ✓) ↳ pivot

2 1 5 0 8 7 6 3

(5 < 3 ×, 0 < 3 ✓, 8 > 3 ✓, 7 > 3 ✓, 6 > 3 ✓)

2 1 5 0 8 7 1 3

(5 < 3 ×, 0 < 3 ✓, 8 > 3 ✓, 7 > 3 ✓, 1 > 3 ✓)

2 1 0 5 8 7 1 3

```
p ← a[lo]
i ← lo+1
j ← hi
    while i ≤ j
        while i ≤ j and a[i] ≤ p
            i ← i+1
        while j ≥ 1 and a[j] ≥ p
            j ← j-1
        IF i < j
            swap (a, i, j)
            i ← i+1
            j ← j-1
        swap (a, lo, j)
        Return j
```

```
void quicksort (int arg[], int low, int hi)
{
    if (low < hi)
    {
    int pi = partition (arr low, hi);

    quicksort (arg, low, pi-1);
    quicksort (arg, pi+1, hi);
    }
}
```

→ algorithm resets i and j when they cross each other.

→ where the pivot belongs.

→ swapping of i and j so the elements move.

easier way to do Quicksort.

step 1: bring pivot to its appropriate position such that left of the pivot is smaller and right is greater.

step 2: QS left part

step 3: // right part

→ last element is selected as the pivot.

Step 1:

	<70✓ 0	<70X 1	<70✓ 2	<70X 3	4	<70✓ 5	≥70✓ 6	
a	10	80	30	90	40	50	70 pivot	i = index
		i	j					j = loop variable

10 < 70 ✓ → check first and last first if true then nothing changes but i++ so i=0, j=1

80 < 70 X → no action, j = 2

30 < 70 ✓ → i++ : i=1, swap with that (30 ↔ 80), j = 3 true for i and j

	10	30	80	90	40	50	70
			<70X	<70✓	<70✓	<70✓	
				i			j

90 < 70X → j = 4

40 < 70 ✓ → i++ : i = 2, swap (80 ↔ 40), j = 5 :

50 < 70 ✓ → i++ : i = 3, swap (50 ↔ 90), j = 6 :

	0	1	2	3	4	5	6
	10	30	40	90	80	50	70
	10	30	40	50	80	90	70

0	1	2	3	4	5	6
10	30	40	50 i	80	90	70 j

i=3 we now swap arr[i+1] and pivot

< 70				> 70		
10	30	40	50	70	90	80

Step2: Quick sort the right part

10	30	40	50	70	90	80
					↑ pivot	

50 is already at correct position so next element

40 " " " " " " " "

30 " " " " " " " "

step 3: 80 is pivot, 90 > 80 so, swap.

10 30 40 50 70 80 90 — ans

- Merge Sort
  - → the division does not depend on the data in the array, so we can make it as even as possible.
  - → usually done recursively
  - → divide and conquer scheme.

in such a case where it can't be divided equally we:

ex1: 38 27 43 3 9 82 10

dividing into sub-problems.

38 27 43 3 | 9 82 10

38 27 | 43 3 | 9 82 | 10

individual Separating

38 | 27 | 43 | 3 | 9 | 82 | 10

merging →

27 38 | 3 43 | 9 82 | 10

Swapped → depending on the order

3 27 38 43 | 9 10 82

3 9 10 27 38 43 82

sorted ✓

ex2: 2 8 5 3 9 4 1 7

2 8 5 3 | 9 4 1 7

2 8 | 5 3 | 9 4 | 1 7

2 | 8 | 5 | 3 | 9 | 4 | 1 | 7

2 | 8 | 5 | 3 | 9 | 4 | 1 | 7

28 35 | 49 17

2358 | 1479

1 2 3 4 5 7 8 9

sorted ✓

→ The time is proportional to the lengths of the lists being merged.
→ 2 arrays can never be merged together ∴ we create a new one.

```
m ← a new array of length nl+nr
i ← 0
j ← 0
m contains the first i+j elements of the merge of l and r.
while i < nl OR j < nr
    IF i < nl AND (j = nr OR l[i] ≤ r[j])
        m[i+j] ← l[i]
        i ← i+1
    ELSE
        m[i+j] ← r[j]
        j ← j+1
    Return m
```

merge sort algorithm

```
IF lo ≥ hi
    Return a[lo .. hi]
mid ← (lo + hi + 1) Div 2
l ← Merge Sort (a, lo, mid-1)
r ← Merge Sort (a, mid, hi)
Return Merge (l, r).
```

Recap

	extra space	avg time	worst-t	adaptive?	stable?
Selection sort	$O(1)$	$O(n^2)$	$O(n^2)$	×	×
insertion sort	$O(1)$	$O(n^2)$	$O(n^2)$	✓	✓
shell sort	$O(1)$	near optimal	near optimal	×	×
Quicksort	$O(\log n)$	$O(n \log n)$	$O(n^2)$	×	×
mergesort	$O(n)$	$O(n \log n)$	$O(n \log n)$	✓	✓
heap sort	$O(1)$	$O(n \log n)$	$O(n \log n)$	×	×

⑦ Limits of Algorithms

→ used when ..

1) problems can't be solved in a reasonable amount of time.
2) problems that can't be solved.

• Tractability = tractable when there are algorithms to solve the problem

→ problems that have algorithms with polynomial worst case time complexity (P) ie $O(n^k)$. these problems might be solvable in reasonable amount of time.

P = can ~~be solved~~ solve problems for which the time required to solve it grows at a reasonable rate as the problem size increases. It can have efficient solutions available.

→ problems like colouring problem, Bin packing problem, time tabling problem etc can not be solved (intractable) as no one knows the polynomial-time algorithm or if the algorithm exists or not that is we don't know whether they are in P or not.

- Exploring problem space

Two things to remember:-

1) NP (nondeterministic Polynomial time) is a class of search problems for which constructing and checking a single possible answer can be done on polynomial time.
problems mentioned previously are all NP. for satisfiability problem we just guess boolean values for each variable and to check it we can evaluate the boolean expression.
time is proportional to the size of the expression ie linear time, which is polynomial.

2) NP consists of problems for which a potential solution can be verified efficiently.
we know problems outside of NP are intractable.

- Reduction of Problems

→ we can solve a problem by reducing it to a problem for which we have a good algorithm.

↳ problem A can be reduced to a problem B if we can use an algorithm for A that calls for B. problem B can be solved by using ~~that solution~~ the solution that was used to solve A.

↳ reduction provides a mapping in such a way that the solution to problem B can be used to obtain a solution to problem A.

ex:- Find the median in an array of numbers.
↳ can be ~~solved by reducing~~ reduced by sorting first
↳ taking out the middle value
↳ middle value would be the median.

→ problem is NP-hard = can be reduced to it in polynomial time.
→ problem is NP-complete = if it is NP-hard and is in NP.

P- class: easily solvable problems and can be easily checked.
e.g: 2x2=4, sorting

NP- class: problems that are harder than P.
↳ hard to solve, but answers are easy to check.
↳ NPC (NP-complete) are the hardest problem in the class.

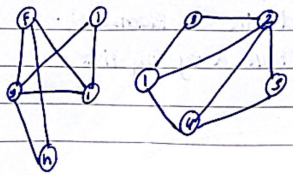
can't find a solution without trial and error.
try many possible answers before getting the correct one.
like: database problems, circuit design, satisfiability.

* is P = NP = ? (remains open)
↳ if true then it means that efficient algorithms exist for all NP problems.
↳ if false then problem within NP are inherently difficult to solve.

• Grey Area Problem
→ grey area between P and NP-complete. Known to NP but not NP-complete. situation if P ≠ NP.
e.g: Integer Factorization Problem
→ Give an n-bit whole number, find two smaller numbers that multiply together to give that number.

56
7 x 8 = 56

e.g: Graph Isomorphism Problem.
→ Is there a one-to-one matching of nodes between graphs.



- Application : Public key encryption (RSA)

→ RSA algorithm will be used b/w two people who dont want their messages to be read by a third-party.
↳ involves generating public and private keys.
↓ ↳ decrypt messages.
encrypt messages

$$C = m^{e} \text{ MOD } N \leftrightarrow m = C^{d} \text{ MOD } N$$

(e: public key; d: private key; N: public key)

↳ cipher; ↓ plain

N, d and e such that for any number $0 \leq m < N$.
↓ plaintext

→ anyone can use e and N to encypta message and can be done in a reasonable time.
→ only bob can decrypt d and know the message.
→ N = much bigger so we combine a block of characters to avoid repetations.

- Generate the keys

$$m = m^{de} \text{ MOD } N$$

RSA method :-

1) Find two large prime numbers p and q
2) Compute $N = p\,q$
3) find a number e that is smaller than $(p-1)(q-1)$ having no CF.
4) find a number d such that $d\ e \text{ MOD } (p-1)(q-1) = 1$.

→ A publishes N and A keeps p and q secret.
→ if B wants to generate the key d, she will factorise N.
→ N will have many digits so it will take a very long time.

- Noncomputability

→ Some problems are worse and no algorithm can solve it.

1) The halting problem
2) Decision problem for first order logic: determines whether a predicate logic formula is valid (∀, ∃).
3) Program equivalence Problem: a terminating programs produce the same results.

⑧ Regular expressions + finite machines

→ we use simplified view of languages called formal languages.

↳ we are only concerned about the surface structure (syntax) of languages instead of their meaning (semantic).

→ an Alphabet is written as Σ. is a finite set of symbols or letters: $\Sigma_{bits} = \{0, 1\}$ or $\Sigma_{digits} = \{1, 2, 3, 4, 5 \ldots\}$

→ a string over an alphabet is written as Σ^* and is a finite sequence of symbols from the alphabet Σ. , Σ^* is an infinite set.

↳ an empty string is written as ε (Epsilon). : 'strings of bits: ε, 0, 01, 10110' or 'string of letters: ε, a, b, c, d, bad.'

→ A language over an alphabet Σ is a finite/infinite set of strings over the alphabet, that makes it the subset of Σ^*. :-

a) $\emptyset$, empty set of strings.
b) $\{\varepsilon\}$, set containing only the empty string can be written as λ.
c) $\{\varepsilon, a, b, aa, ab, ba, bb\}$, strings of lengths at least at most 2. ~~infinite~~.

→ recogniser is used for a language L over an alphabet Σ which is a machine that can be used to decide whether a string over Σ or L.

- The Chomsky hierarchy

→ many ways to define particular formal languages. The types in the table were enumerated by linguist Naom Chomsky. each type is a restriction of the earlier one.

Type	language	Description	Recogniser
0	Decidable	general re-writing systems	Turning machine
1	Context-sensitive	context-sensitive grammars	linear-bounded turning machine.
2	Context-free	context-free grammars	Pushdown machine.
3	regular	regular expression	Finite state machine.

Computer language processing relies on these two.

- Operations on formal languages

1) The union of two languages L_1 and L_2 over the same alphabet Σ is written as $L_1 \cup L_2 :- \{a, ab\} \cup \{a, bc\} = \{a, ab, bc\}$.
   ↳ all the elements combined.

2) The intersection of two languages L_1 and L_2 over the same alphabet Σ is written as $L_1 \cap L_2 :- \{a, ab\} \cap \{a, bc\} = \{a\}$.
   ↳ only the common element.

3) The complement of a language L over the alphabet Σ is written as $\Sigma^* - L$:
   if $\Sigma = \{a\}$
   - $\overline{\{\varepsilon, a, aa\}} = \{aaa, aaaa, aaaaa, \dots\}$
   - $\overline{\{\varepsilon, aa, aaaa, \cdots\}} = \{a, aaa, aaaaa, \cdots\}$

4) The concatenation of L_1 and L_2 over Σ is written as a set of strings formed by concatenating a string from L_1 with one from L_2 :-

$L_1 L_2 = \{ uv \mid u \in L_1 \wedge v \in L_2 \}$.

ex:- $\{a, bc\}\{de, f\} = \{ade, af, bcde, bcf\}$

ex2:- $\{\varepsilon, a\}\{a, bc\} = \{a, bc, aa, abc\}$

* 5) The Kleene closure of all languages L over the alphabet Σ, written as Σ^*, is the set of strings of the form $w_1 w_2 \cdots w_n$ where $w_i \in L$, for $n \geq 0$. L^* is the smallest set satisfying $L^* = \{\varepsilon\} \cup LL^*$.

ex: $\{a\}^* = \{\varepsilon, a, aa, aaa, aaaa, \cdots\}$

ex2: $\{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb\}$

↳ all possible outcomes.

- Regular expressions.

→ way of describing a simple language.

1. $\emptyset$ = empty set
2. ε = the language $\{\varepsilon\}$
3. S = the language $\{S\}$ where s is a symbol from Σ.
4. $R|S$ = union of languages described by R and S. → expression from Σ.
5. RS = concatenation of languages described by R and S.
6. R^* = Kleene closure of the languages described by R.
7. (R) = language described by R.

	RExp	language..
ex1:	a	$\{a\}$
	$a \mid ba$	$\{a, ba\}$
	$(a \mid bb)^*$	$\{\varepsilon, a, bb, abb, bba, bbbb \cdots\}$
	$a(a \mid bb)^* a$	$\{aa, aaa, abba, aaaa, aabba, abbaa \cdots\}$
	$a(a \mid ab)^*$	$\{a, aa, ab, aab, aba, abb \cdots\}$
	$b^* a b^*$	$\{a, ab, ba, abb, bab, bba \cdots\}$
	$((a \mid b)(a \mid b))^*$	$\{\varepsilon, aa, ab, ba, bb, aaaa, aaab, aaba \cdots\}$

- Searching with extended regular expressions.

→ common ones:-

a) [abc] → (a|b|c)

b) [^abc] → any single character except a, b or c.

c) . → any single character.

d) R? → (|R) e.i: zero or one R

e) R+ → short for RR*

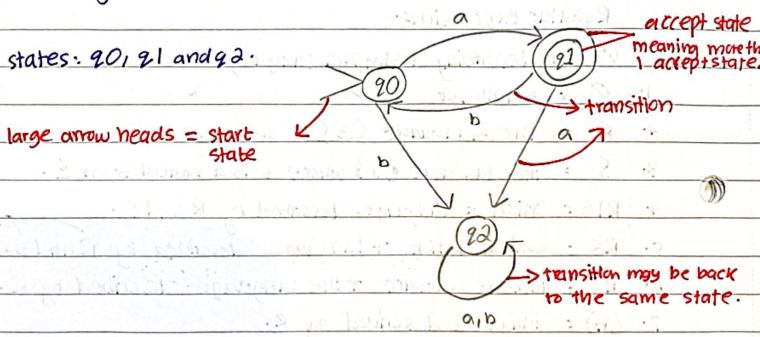
f) \c → literal instance of a special character (. or *).

- Finite state machines.

→ simple machine with which we decide whether a string belongs to a particular regular language. e.g: alphabet $\Sigma = \{a, b\}$ and consider the regular expression a(ba)*, which describes the language {a, aba, ababa, abababa}.

→ also a recogniser for this language.

→ three states: q0, q1 and q2.



→ we use the machine to scan the string e.g: ababa

$$q0 \xrightarrow{a} q1 \xrightarrow{b} q0 \xrightarrow{a} q1 \xrightarrow{b} q0 \xrightarrow{a} q1$$

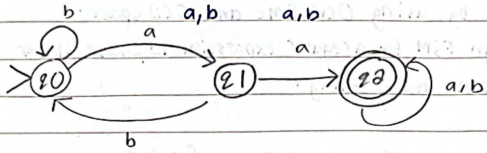
↳ start state

→ string abab would be rejected as we have ended up in state q0 which is not an accept state.

ex2: input aaba

↳ $q0 \xrightarrow{a} q1 \xrightarrow{a} q2 \xrightarrow{b} q2 \xrightarrow{a} q2$

ex3:



Show the sequence of steps:

i) bbaab

↳ $q0 \xrightarrow{b} q0 \xrightarrow{b} q0 \xrightarrow{a} q1 \xrightarrow{a} q2 \xrightarrow{b} q2$ (accept)

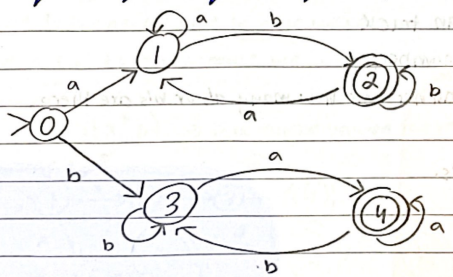
ii) abaa

↳ $q0 \xrightarrow{a} q1 \xrightarrow{b} q0 \xrightarrow{a} q1 \xrightarrow{a} q2$ (accept)

iii) abba

↳ $q0 \xrightarrow{a} q1 \xrightarrow{b} q0 \xrightarrow{b} q0 \xrightarrow{a} q1$ (reject)

ex4:



Show sequence:-

i) ababbababa

↳ $q0 \xrightarrow{a} q1 \xrightarrow{b} q2 \xrightarrow{a} q1 \xrightarrow{b} q2 \xrightarrow{b} q2 \xrightarrow{a} q1 \xrightarrow{b} q2 \xrightarrow{a} q1 \xrightarrow{b} q2 \xrightarrow{a} q1$

(reject)

ii) bbaa

↳ $q0 \xrightarrow{b} q3 \xrightarrow{b} q3 \xrightarrow{a} q4 \xrightarrow{a} q4$ (accept)

- FSMs and regular expressions

→ we can decide whether a string of length n belongs to a particular regular language by using O(n) time and O(1) space.

→ one way to design FSM for a regular expression to choose states to remember certain inputs we can use counting.

→ language of strings over the alphabet $\Sigma = \{a, b\}$ containing exactly 2 b's i.e: the language described by the regular expression $a^* ba^* ba^*$

↳ use states to keep track of how many b's we've seen.

- $q0$ no b's $\rightarrow a^*$
- $q1$ 1 b $\rightarrow a^* ba^*$
- $q2$ 2 b $\rightarrow a^* ba^* ba^*$
- $q3$ 3 or more b $\rightarrow a^* ba^* ba^* b(a|b)^*$

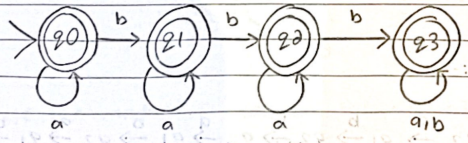
- Limitations:-

↳ Finite machines can track

↳ count up to fixed number

↳ Finite machines can not count how many a's or b's are there.

ex1. FSM containing 2 b's.



→ accept states are different. To accept strings with more than 2 b's i.e: zero, one or two b's, we make $q0$, $q1$ and $q2$ accept states. (two circles).

(9) Context - free grammar

→ allows us to express more complex languages that are used to define the syntax of most programming languages.

→ used at a higher level of structure:-

1) For simpler regular expression, we call the elements of Σ letters, and the string of the language words.

2) For richer context free expression, we call the elements of Σ words, and the string of the language sentences.

• Approximating natural languages

→ we can simplify the complex grammatical rules that people speak and write by approximating them for a simple subset.

↳ {sentence}, {subject}, {predicate}, {noun-phrase}, {article}, {modified-noun}, {adjective}, {noun} and {verb}.

↳ Syntactic catagory.

• Formal Grammar

→ provide rules for expanding these catagaries in terms of other catagories and concrete words like:-

1) <sentence> → <subject><predicate>
2) <subject> → <noun-phrase>
3) <predicate> → <verb><noun-phrase>
4) <noun phrase> → <article><modified noun>|<noun> → does not depend on any words that surround it.
5) <noun> → apple | bear | dog
6) <verb> → eats | hugs
7) <adjective> → ~~running~~ wriggly | tasty
8) <article> → a | ~~an~~ | the

- CFG defination

→ context free grammar is a collection of four things:-

1) alphabet Σ are called terminals from which strings in a language will be constructed.
2) seperate set of symbols are called non-terminals or variables.
3) distinguished non-terminal are called start symbol S.
4) finite set of productions or rules of the form A → B where A = non-terminal and B = finite sequence of zero or more terminal or non-terminal symbols.

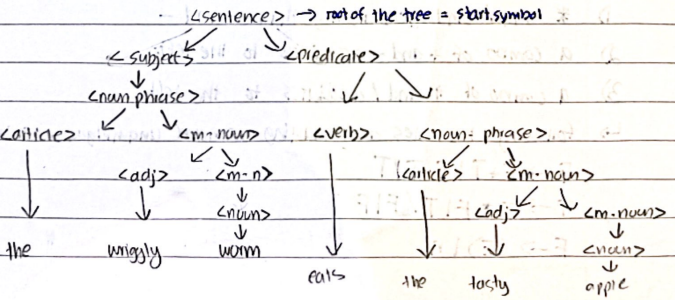
- Alternative Notation aka Backus-Naur form (BNF)

→ more compact notation is used by combining rules with the same left HS and seperating the RHS with vertical bars::

1) <sentence> → <subject><predicate>
2) <subject> → <noun-phrase>
3) <predicate> → <verb><noun-phrase>
4) <noun-phrase> → <article><modified-noun>
5) <modified noun> → <adjective><modified noun> | <noun> (→ or)
6) <noun> → apple | bear | dog
7) <verb> → eats | eats | hugs
8) <adjective> → wriggly | tasty
9) <article> → a | an | the

- Deriving a sentence

→ we make a derivation tree with start symbol <sentence> and expanding a non-terminal symbol <noun> until only terminal symbols remain. The wriggly worm eats the tasty apple



- Algorithms

→ recogniser for the program is an algorithm that tells us if an input string belongs to the language (derivation → grammar).

→ parsing is when one wants to go in the opposite direction starting with a string and working out how it can be derived from the start symbol.

Parser for an ~~algorithm~~ CFG is an algorithm that string as an input and either:

1) returns a parse tree for the string.
2) announces that no parse tree exists for that string.

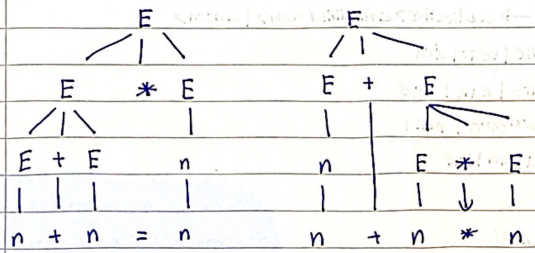
→ parsing algorithm parses a string of length n in $O(n^3)$ time.

- Ambiguity.

→ ~~string~~ CFG is ambiguous if some string has more than one parse tree.
or derived in multiple ways.

e.g: $E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid n$

↳ two parse trees for the string $n+n*n$



→ we tighten the grammar so it chooses the parse tree one would expect using rule:

1) * and / bind more tightly than + and −
2) a combo of + and − associates to the left.
3) a combo of * and / associates to the left.

↳ following the rules and making grammar unambiguous or inherently ambiguous.

$E \rightarrow E+T \mid E-T \mid T$

$T \rightarrow T*F \mid T/F \mid F$

$F \rightarrow (E) \mid n$