**Program efficiency**

Efficiency is all about time and space complexity.

**Dynamic programming**

**Greedy approach:**

In greedy approach whatever is the solution for the problem give at the first go is fixed as the final solution

**Note:** This is not the best approach for all the scenarios however is also work for some cases.

In dynamic programming we will be finding or we will find out all the possible solution for the given problem out of which the best will be selected or picked.

**Time and space complexity:**

**Aymptotic notation:**

* Big O
* Omega
* Theta

**Program-1:**

Swajith has Rs. 1 lakh in his bank account, with a 12% annual interest rate. Swajith takes a withdrawal of 25,000 rupees in the fifth month to buy a gift for a loved one. His second loved one deposits $10,000 into his account in the ninth month. How much money Swajith has left over in his account at the end of the financial year.

**Space complexity**

**Structure:**

it allocates space for each individual member

**Union:**

it allocates space of the largest member.

**Time complexity:**

A for loop executes n+1 times



10-



5-



10 20

For (i=0;i<n;i+2){

Statements;

}

n/2

f(n)=n/2

Degree of polynomial is n

So n/ anything is n

So here also

O(n)

So irrespective of iteration time is gonna be same o(n)

**Nested loops**

For(i=0;i<n; i++){ …n+1 times

For(j=0;j<n; j++){ n\*(n+1)

Statements; n\*n---- square

}

}

Time complexity = o(n square)

**Q) Implement a 2d array rotate the array 90 degrees.**

**Nested loops continue**

For(i=0;i<n;i++){

For(j=0;j<n;j++){

Statements;

}

}

I=0 then 0<0 no I j

1. nothing
2. o will execure

will stop

1. will stop

For (i=1;i<n;i\*2){

Statements;

}

**Analyze**

I=1 1 time

I=2 2 times (1\*2)

I=3 4 times (1\*2)\*2=2 power 2

I=4 8 times (1\*2)\*2)\*2=2 power 3

So when stopes i>=n

I=2 power k

2 power k>=n

K=log n base 2

So when complexity O(log n base 2)

**Create an array (1d), it should contain number b/w 10 to 30, in this array extract and print**

1. **Even numbers**
2. **power values**

* **Constant Time complexity:** O(1)…
* **Linear Time complexity** :O(n)…
* **Logarithmic Time complexity**: O(log n)…
* **Quadratic Time complexity** :O(n2)…
* **Exponential Time complexity:** O(2n)…

**FINAL SUMMARY:**

I++ i- - i+2 O(n)

I\*2 i/2 log(n) base 2

P=0

For(i=0;P<n; i++) O(sqrt(n))

P+=i

def generate\_lists(n):

    table\_list=[]

    for num in range(n):

        row=[]

        for i in range(n):

            row.append(i)

        table\_list.append(row)

    return table\_list

print(generate\_lists(10))

it is called log-linear complexity

**Polynomial complexity:**

Space complexity grows proportionally square of input O(n2).

**Assignment day-2:**

What is stack and heap memory. Which languages are using heap.

What is quick sort?

**Precedence:**

**Q) 10\*4/6+3-1%2**

Ans:

1. 10\*4=40
2. 40/6=6.66
3. 1%2=1
4. 6.66+3=9.66
5. 9.66-1=8.66

**Q) 7+2&4+3&9**

Ans :

1. 7+2=9
2. 4+3=7
3. 9&7=1
4. 1&9=1

**Operator precedence table**

|  |  |  |
| --- | --- | --- |
| Category | Operator | Associativity |
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)\* & sizeof | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %=>>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

**Note :** ~ -> & -> ^ -> |

**Q) 6|3&9+6**

Ans:

1. 9+6=15
2. 3&15=3
3. 6|3=7

**Q) ~9+4&6**

Ans:

1. 9+4=13
2. ~13=2
3. 2&6=2

**Bit manipulation tricks**

**Xor ^**

Even 1’s :0

Odd 1’s:1

Xor of number itself is 0

Xor of number with 0 is number itself

**Ex:** 4^6^5

0100

0110

0101

0111=7

**Right shift :**

1) 5>>1

5/2 ans 2

5>>2 = 2

**Left shift :**

Int 32 bits

1) 5 – 0101

5<<2

5\*power(2,2) ans 20

2) 10<<3

10\*power(2,3)=80

**XOR programs:**

**Q1) Find element that repeat only once in a list using xor.**

def findsingle(ar,n):

    res=ar[0]

    for i in range(1,n):

        res=res^ar[i]

        print(res)

    return res

ar=[2,3,5,4,5,3,4,2,88]

print(findsingle(ar,len(ar)))

**Q2) Swap two number using xor.**

a=100

b=200

a=a^b

b=a^b

a=a^b

print(f"a:{a} b:{b}")

**Q3) For the given number n check the kth bit is set or not.**

# using bitwise

n=int(input("enter the number:"))

p=int(input("enter the position of bit:"))

res=n&(1<<(p-1))

if res==0:

    print("not set")

else:

    print("set")

**Q) Write a program to find the xor of all the numbers in the given range.**

def xor(n):

    if n==0:

        return 0

    return n^xor(n-1)

l=int(input("enter lower range:"))

r=int(input("enter upper range:"))

res=xor(r)^xor(l-1)

print(res)

**Tower of Hanoi:**

**Challenge:**

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

|  |  |  |
| --- | --- | --- |
| 6 | 1 | 8 |
| 7 | 5 | 3 |
| 2 | 9 | 4 |

The sum of all the rows and columns and diagonals should be same .

**Steps:**

1. The sum of total number should be divided by 3 and each row and column should get the sum of this.
2. The middle index should be the median of the given number
3. The smallest and largest number should not be in the corners.
4. The odd numbers should not be in the corners.

**Challenge:**

|  |  |  |  |
| --- | --- | --- | --- |
| **X** | **Q2** | **X** | **X** |
| **X** | **X** | **X** | **Q3** |
| **Q1** | **X** | **X** | **X** |
| **X** | **X** | **Q4** | **X** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **X** | **Q2** | **x** | **x** | **x** | **x** | **x** | **X** |
| **X** | **X** | **X** | **Q4** | **X** | **x** | **x** | **x** |
| **Q1** | **X** | **X** | **X** | **X** | **x** | **x** | **x** |
| **X** | **X** | **Q3** | **X** | **x** | **x** | **x** | **x** |
| **X** | **X** | **X** | **X** |  | **x** | **x** |  |
| **X** | **X** | **x** | **X** | **X** |  | **x** | **X** |
| **X** | **x** | **x** | **x** |  | **x** |  | **X** |
| **x** | **X** | **X** | **X** |  |  | **X** | **X** |

**Recursion:**

* Recursion is a fundamental concept in computer science
* It involves a function repeatedly calling itself until a base case is reached.
* Recursion is used to solve problems by breaking them down into smaller, similar sub-problems
* It can significantly reduce code length and complexity compared to iteration.
* Recursion is an optimal approach for tasks like tree and graph traversals and finding the greatest common divisor(GCD) using Euclid’s algorithm.

**Backtracking:**

* Backtracking is a technique for finding solutions to problems throught recursive exploration
* It involves undoing recursive changes if certain conditions are not met and discarding less optimal solutions
* Backtracking is more efficient than brute force in solving complex problems, although it can have exponential time complexity.
* It excels in solving challenge problems like the N-queens problem and the travelling salesman problem(TSP)
* It explores various choices and systematically backtracks when necessary to arrive at the best solution.

**What is Recursion**

Recursion is a programming technique in compute science where a function solves a problem by breaking it down into similar sub-problems and calling itself to solve them. This process continues until it reaches a base case, where the problem becomes simple enough to be solve directly without further recursive calls.

The base case is like a stop sign in a recursive function. It’s a condition where we already know the answer, so we don’t need to keep making more calls.

**What is Backtracking**

Backtracking is a problem solving technique where we explore different paths to find solutions to a problem. It’s like trying to solve a maze. You start by taking a path, and if it leads to a dead end, you backtrack(go back to where you made the last choice) and try a different path

Here’s how it works:

1. You break the problem into smaller parts sub-problems
2. You try to solve each sub-problem step by step.
3. If a sub-problem doesn’t work our or doesn’t meet the conditions you need, you backtrack (go back) to the previous step and try a different approach.
4. You keep doing this until you find a solution that satisfies all the problem’s requirements.

Its like solving a puzzle piece by piece, discarding pieces that don’t fit, and adjusting your choices as you go wrong. This way, you eventually arrive at the best solution while avoiding unnecessary work or dead ends. Backtracking is especially useful for solving complex problems where you need to explore many possibilities.