Module-4

Brute force approach

It is a fundamental method in problem solving that involves systematically trying every posible solutions to find the correct one.

~~Bo~~ Eg: Pad lock, Password guessing

characteristics of brute force solⁿ:-

(i) exhaustive search
(ii) simplicity
(iii) inefficiency
(iv) guaranteed solⁿ

for loop fn-range

Pad lock

It involves 1000 combinations., 3 digits, 0-9 characters.

Start with 0 0 0

Increment sequencially

Stop when lock opens.

Parameter

```
def Padlock(key)
    for combination in range(1000)
        guess = f"{combination:03}"
        Print(f"trying combination {guess}")
        If guess = = key
            Print(f"padlock open's at {guess}")
            break
```

Pattern instead of series and make it as 3 digits.

Padlock (123)

Out: Trying combination 000
" " 001
" " ~~002~~ 123

Padlock opens at 123

Password guessing

```
import itertools
Import strings
def check_password (Password)
    character = string.printable
    for guess in itertools.product (characters, repeat=3)
    guess = ''.join (guess)
    Print (f"Trying password: {guess}")
    If guess == Password
        Print (f"password found: {guess}")
        return

Password = "Dog"
check_password (password)
```

all strings that are printable are saved here.
instead of range
reset as default as tuple
to convert tuple to string

Out: Trying password : aaa
" " : "
Trying password : Dog
Password found : Dog

Divide and conquer approach.

Solve problems efficiently.

(i) divide (division continues recursively until the sub problems are simple enough to solve directly.
(ii) Conquer
(iii) Merge

Merge and ~~Quick~~ Sort

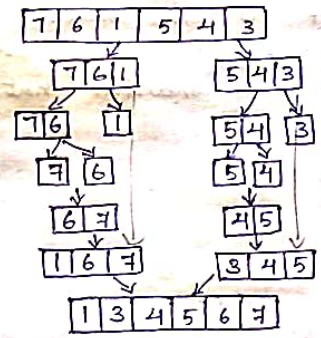
input : an unsorted array of size n

Process

divide array into two halves left and right
Recursively and then sort left and right.
Merge left and right into sorted array.

Output : a sorted array

Eg:



Dynamic Programming

D.P is a method for solving problems by breaking them down into smaller overlaping sub problems, solving each sub problem just once, and storing their solutions. It is particularly useful for optimisation problems where the problem can be divided into simpler sub-problems that are solved independly and combine to form a solution to the original problem.

Key Properties / fundamental Principles of D.P

Optimal Substructure

Overlapping sub problems

Example:- fibinoki series / sequence

To find fibinoki(5), you need the values of fibinoki(4) and fibinoki(3). To compute the fibinoki(4) you need fibinoki(2) and Fibinoki(3). Here, fibinoki(3) is computed multiple times when calculating different fibinoki numbers.

Without dynamic Programming

To compute fibinoki(5), you should calculate fibinoki(3) twice.

This redentency leads to a lot of repeatd works.

With dynamic Programming

To compute fibinoki(3) once and stores its result. whenever you need fibinoki(3) again, you call the stored result instead of recalculating it. This helps to avoid unnecessary calculations and speeds up the process.

Compute and reuse - majorly uses two operations

Steps involved in D.P.

Break down complex problems to sub-problems.
find the optimal solutions to the sub-problem.
store the result of sub problem.
Reuse the result of sub problems to avoid repeated calculations.
Finaly find the result of complex problem.

Approaches in D.P

Memoization (top-down approach)

Tabulation (bottom-up approach)

Greedy Approach

Greedy approach is the most intuitive method in algorithm design. when placed with a problem that requires a series of decisions. A greedy algorithm makes the best choice available at each step, focusing solely on the immediate situation without considering future consequences.

This approach simplifies problem by reducing it into a series of smaller sub-problems each requiring fewer decisions. Greedy algorithm excels in problems with optimal substructure where the problem can be broken into smaller solvable components.

Greedy algorithm approach properties

Greedy solution could be implemented only if the problem statement follows two properties.

1) Greedy choice property
2) Optimal substructure

Key characteristics of greedy approach.

1) Local optimisation (best possible at each step)
2) Irrevokable decisions (no backtracking or reconsidering
3) Problem specific characteristic
4) Optimal solution for a specific problems.
5) Efficiency

Eg:- Coin exchange.

Motivation for greedy approach

Simplicity and ease of implementation.
Straight forward logic
Minimum requirements

2) Efficiency in time and space
   - fast execution
   - low memory usage
3) * optimal solutions for specific problems
   Greedy choice property
   Optimal substructure
4) Real world applicability
   Practical applications
   Quick near optimal solutions.

Advantages and disadvantages of greedy approach:

Advantages - simplicity, speed, optimal for certain problems.

disadvantages - sub optimal solutions

Global optimum can be reached by making local optimum solutions as assumptions.

Irrevokable decisions

lack of backtracking

Eg: task completion problem

Q Given an array of positive integers each indicating completion time for a task. find the maximum number of tasks that can be completed in a limited amount of time that you have.

Eg: completion times = [2, 3, 1, 4, 6]
available time = 8

Ans) Steps involved
Sort task (arrange task in assending order)
Iterate and track (add track sequencially as long as the total time doesn't exceed the limit and update the task count accordingly.)

Add task with time 1 : total time = 1
task count = 1

Add task with time 2 : total time = 3
task count = 2

3 : 3
1+3
5+1=6

Add task with time 1, 3 : total time = 6
task count = 3

Next task with time for will exceed available time, so loop breaks.
Max. no. of tasks that can be completed in 8 units of time is 3.

```
def max_task (completion_times, available time).
    completion_times.sort()  # to sort array
    total time = 0
    task_count = 0

for time in completion_times:
    if total_time + time<= available_time
        total_time += time
        task_count += 1
    else:
        break
return task_count
```