Greedy (Best First Search)

- A search method of selecting the **Local Optimal** (best local choice) at each step hopes to find a **Global Optimal** solution.

- A greedy algorithm works in phases: At each phase:
  - You take the best you can get right now, without regard for future consequences.
  - You hope that by choosing a local optimum at each step, you will end up at a global optimum.

Greedy Algorithm

Greedy method control abstraction/ general method

```
Algorithm  Greedy(a,n)
// a[1:n] contains the n inputs
{
   solution=  //Initialize solution
   for i=1 to n do
   {
       x:=Select(a);
       if  Feasible(solution,x)  then
             solution=Union(solution,x)
   }
return solution;
}
```

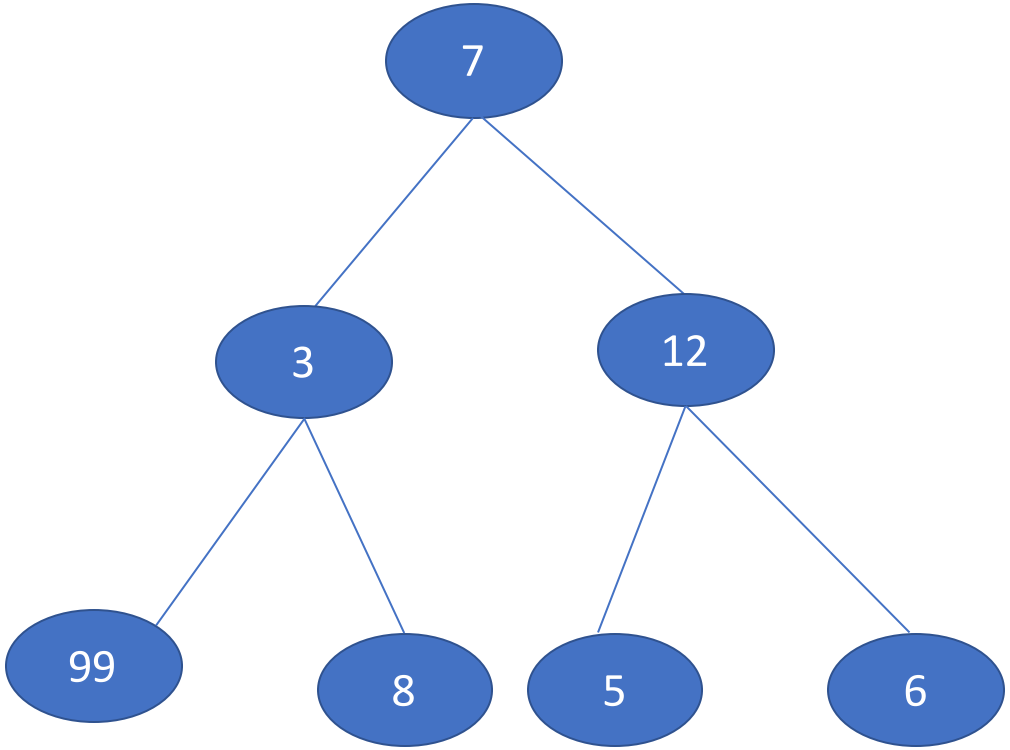
a1	a2	a3	a4

- 1 2 3 4 **(n=4)**

- Note: Greedy has a constraint/condition, goal is to choose the feasible solution(s) of input a, having 1,2,…… with a total of 4 inputs solution, that meets the condition(optimal solution) out of all possible solutions.

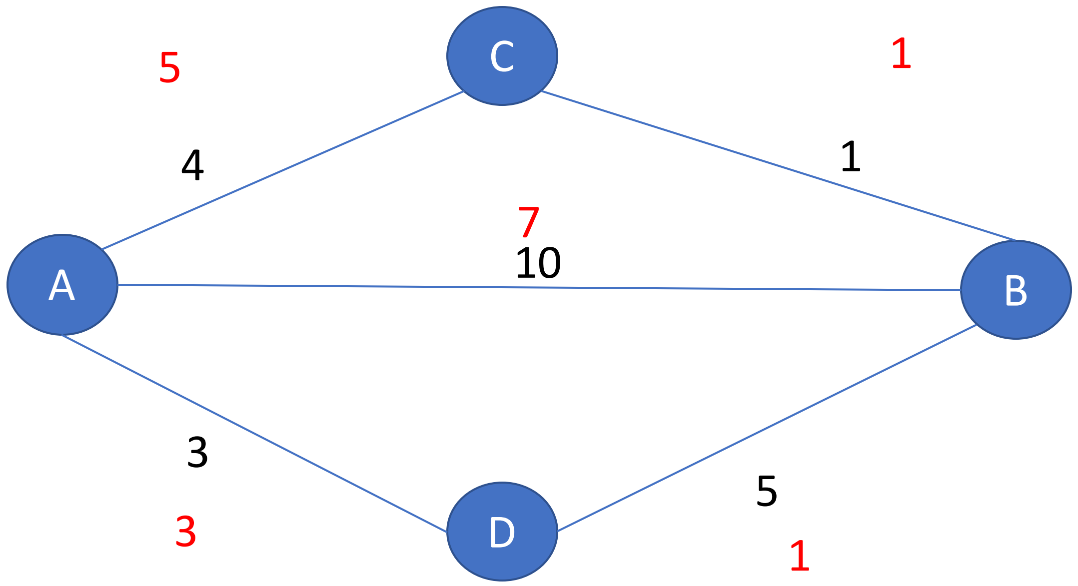
Greedy Algorithm

- E.g.



- Tree Problem, Get to root node: Greedy Algorithm will start with best choice(Largest).
- 7→12(>3)→6(>5) = 7 + 12 + 6 = 25(global solution)
- But Optimal Solution
- 7→ 3 →99= 109.

Greedy Algorithm



- Goal: find shortest path from A to B, use Greedy Technique,?
  - Greedy Algorithm/Best First Search(BFS) Local optimum
  - Greedy Algorithm
    - Shortest distance A → C(4) or A →D(3) or A → B(10).
      - BFS = A →D = (3) Local optimum
      - Next feasible choice is D → B =5
      - Total = 3 + 5 = 8
  - But Optimal Solution is
    - A → C → B = 4 + 1 = 5
    - Therefore Greedy Algorithm didn't produce Optimal solution in this case.
- How about Coloured Red Distances?

Greedy Algorithm

- Feasible
  - Has to satisfy the problem's constraints
- Locally Optimal
  - The greedy part
  - Has to make the best local choice among all feasible choices available on that step
    - If this local choice results in a global optimum then the problem has optimal substructure
- Irrevocable
  - Once a choice is made it can't be un-done on subsequent steps of the algorithm
- Simple examples:
  - Task Scheduling,(Select the tasks, with the earliest **finishing time, earliest task if long, other task will be rejected, or uncompleted.**)
  - Playing chess by making best move without lookahead.
  - Giving fewest number of coins as change.

- Pros – Simple, Easy Implementation and Run time is fast(time complexity)
- Cons- sometimes there is no such guarantee of getting Optimal Solution.

Divide-and-Conquer

- Divide and conquer Algorithm operates in recursive mode, using 3 steps;
  1. **Divide**: break instance of the complex problem into sub problems of **same type.**
  2. **Conquer**: Solve sub problems independently and recursively .
  3. **Combine:** Obtain solution to original (larger) instance by combining these solutions.

Applications of Divide and Conquer Algorithm

- Sorting: merge sort and quicksort
- Binary Sort
- Matrix multiplication: Strassen’s algorithm

Divide and Conquer Algorithm

//*S* is a large problem with input size of *n*

Algorithm divide_and_conquer(*S*)

if (*S* is small enough to handle)

solve it //base case: conquer

else

split *S* into two (equally-sized) subproblems S_1 and S_2

divide_and_conquer(S_1)

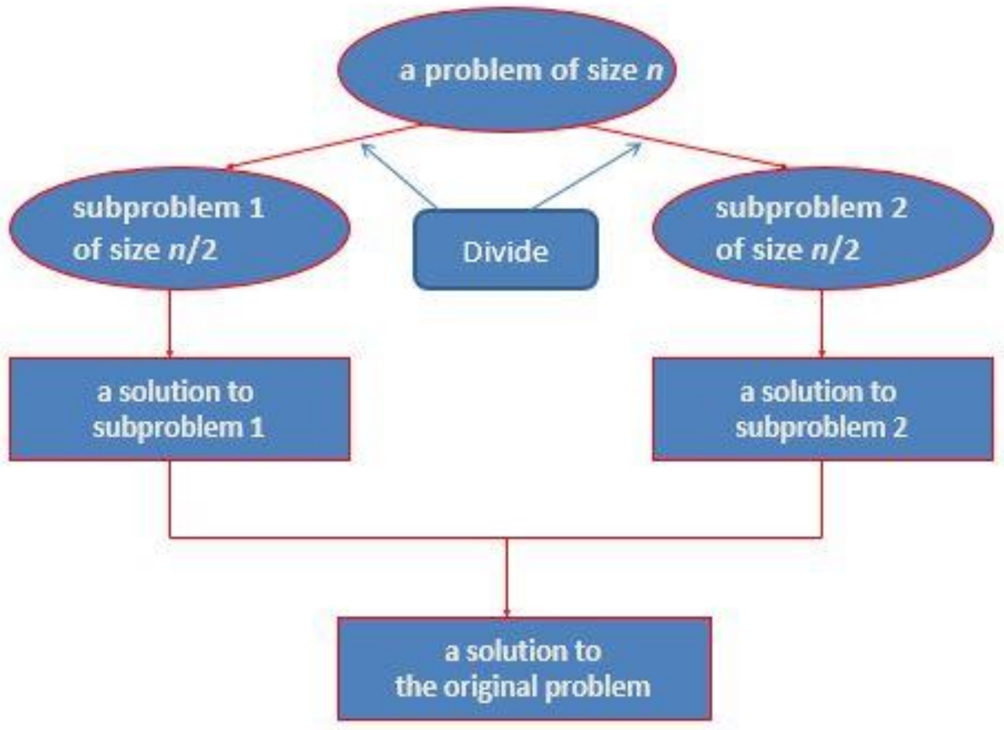
divide_and_conquer(S_2)

combine solutions to S_1 and S_2

endif

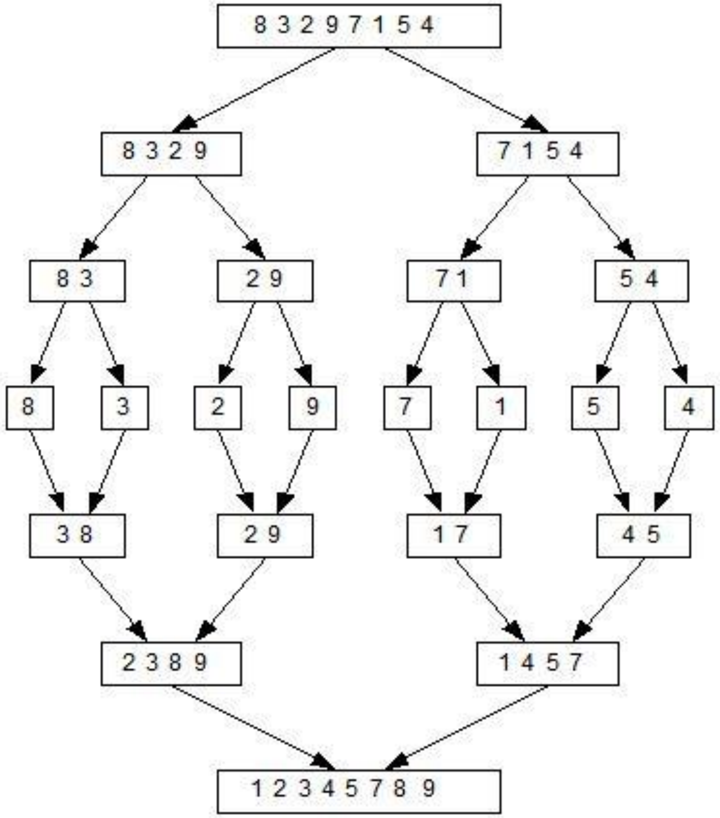
End

- E.g. Merge Sort



Divide and Conquer(Merge Sort)

Mergesort
Example



- Divides problem into independent subproblems-subproblems, until we have unique single element solution, then solutions are merged.
- Pros : solves difficult problems with ease by its division, if it is general case solution like merge sort and they make efficient use of memory caches(parallel Operation).
- Cons: uses recursion that makes it a little slower. If performing a recursion for no. times greater than the stack in the CPU than the system may crash.

DYNAMIC PROGRAMMING

- Dynamic Programming is an algorithm design technique for optimization problems: often minimizing or maximizing. Dynamic Programming is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems.
- Like divide and conquer, DP solves problems by combining solutions to subproblems.
- Divide-&-conquer works best when all subproblems are independent. So, pick partition that makes algorithm most efficient & simply combine solutions to solve entire problem.
- Dynamic programming is needed when subproblems are dependent(subproblems share sub subproblems).
- Note: Both Greedy and Dynamic are used to solve optimization(Minimum or maximum result) problem, although different strategies.

Dynamic Algorithm(Fibonacci Numbers)

procedure *fibonacci*(n: nonnegative integer)
if $n = 0$ **then return** 0
else if $n = 1$ **then return** 1
else return *fibonacci*$(n-1)$ + *fibonacci*$(n-2)$
{output is *fibonacci*(n)}

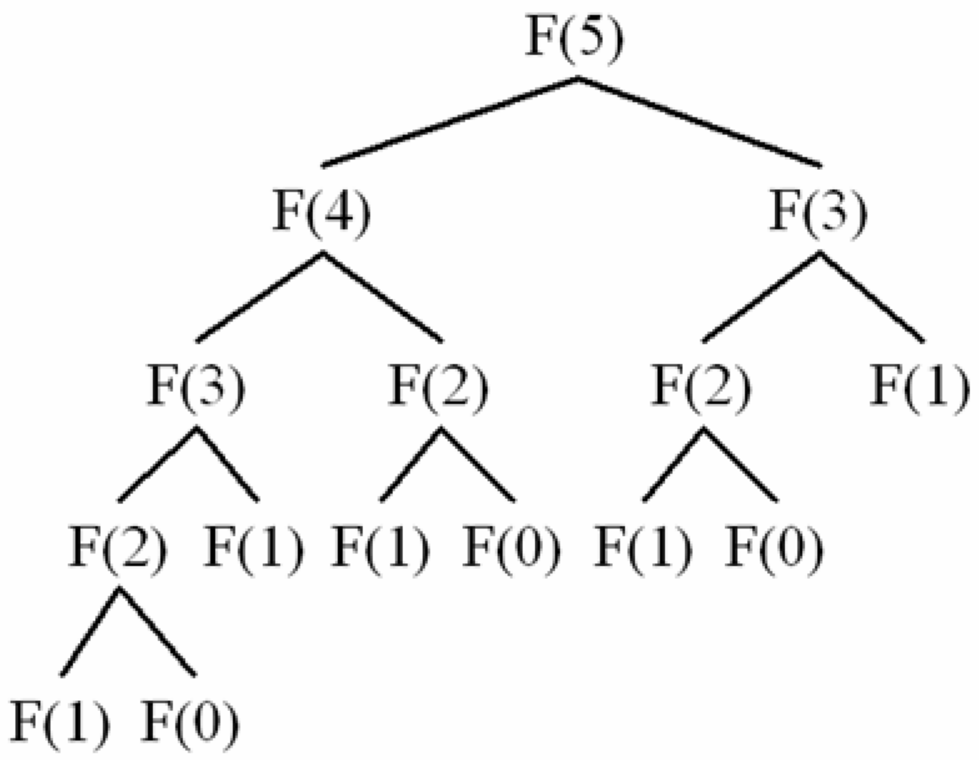
- Fib(n)= {0, 1, F(n-1) + F(n-2)}
  - 0,1,1,2,3,5,8,13 ……. //n>1 every item is obtained by adding previous item
  - F(n)= F(n-1) + F(n-2)
  - F(2) =F(2-1) + f(2-2)
    - 1 + 0 = 1
    - F(2) = 1

F(3) = F(2-1) + F(3-2)

1 + 1 = 2

Note: we didn't need to solve that problem, we just used previous result obtained in solving a subproblem.

Dynamic Algorithm(Fibonacci Numbers)



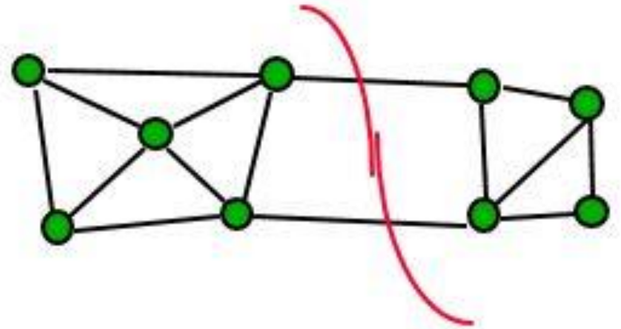
- Dynamic Programming solves the sub-problems bottom up. The problem can't be solved until we find all solutions of sub-problems. Solving some subproblems and reusing the result or functions (stored in DB / memoization or memoisation) of them to solve some more.
- Greedy solves the sub-problems from top down. We first need to find the greedy choice for a problem.
- Dynamic Programming has to try every possibility before solving the problem. It is much more expensive than greedy.

Greedy vs Dynamic

Feature	Greedy	Dynamic
Feasibility	Whatever choice seems best at the moment in the hope that it will lead to global optimal solution.	Decision are made at each step considering current problem and solution to previously solved sub problem to calculate optimal solution .
Optimality	sometimes there **is no such guarantee** of getting Optimal Solution.	There **is guarantee** it will generate an optimal solution as it generally considers **all possible cases** and then **choose the best**.
Recursion	A greedy method follows the problem solving heuristic of making the locally optimal choice at each stage.	It operates based on a recurrent formula(iteration) that uses some previously calculated states(overlap, where we solve same problem more than once).
memoisation	This techniques doesn’t revise or look back to its previous choices, therefore it more efficient in memory utilization	It requires db. table for memoisation and it increases it’s memory complexity.
Time complexity	Greedy method is generally faster	DP is generally slower.

Minimum Cut

- Min Cut Problem – Is used to divide a connected a graph G(V,E) into 2 disjoint graphs(subgraph A & B)
  - AIM: Cut fewest edges



- Contraction: is a technique that merges the endpoints u and v in a G graph to create a new (super) node uv.

Contraction Algorithm Technique

RANDOM CONTRACTION ALGORITHM-David Karger, early 90’s

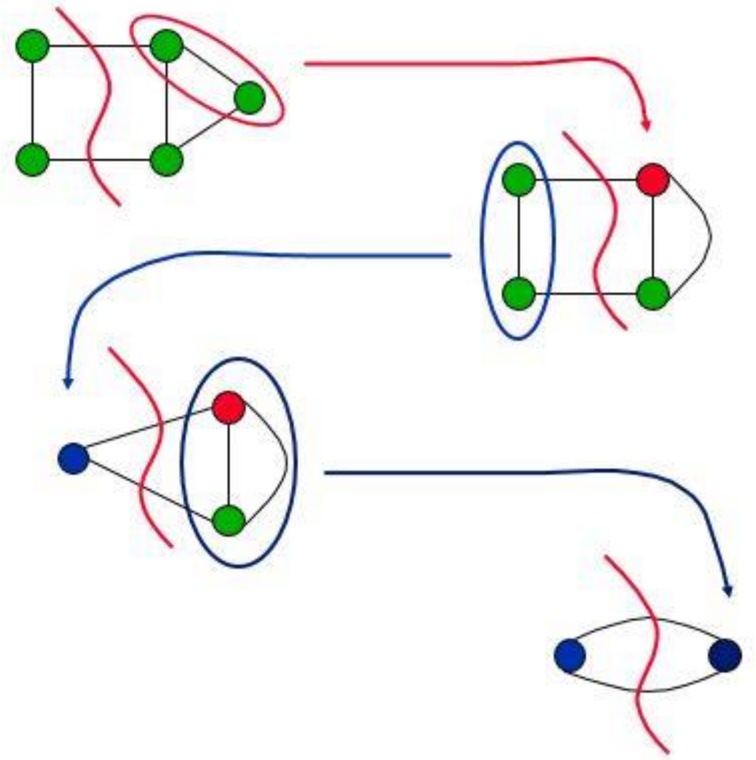
While there are more than 2 vertices:

- Pick a remaining edge (u,v)uniformly at random
- Merge (or “contract”) $u$and $v$into a single vertex
- Remove self-loops

Return cut represented by final 2 vertices

Contract (merge) endpoints to 1 vertex

- O(m) time algorithm to pick edge
- n contractions: O(mn) time for min-cut
  - Note: the edges / connection(s) remains, you only merge vertices/nodes(v).
  - Aim: Get 2 vertices from the original graph



END