# GREEDY ALGORITHMS

→ Optimizing problems

general structure

```
getOptimal (Item arr[], int n){
    ① Initialize res = 0
    ② while (All items are not considered)
        {
            i = selectAnItem().
            if (feasible(i))
                res = res + i;
        }.
    ③ Return Res
}.
```

* Greedy algorithms may not work all the time

Activity Selection problem

I/P → {(2,3), (1,4), (5,8), (6,10)}

O/P → 2

machine can do only 1 activity at a time and we have to count max. number of activities happen on single machine

I/P → {(1,3), (2,4), (3,8) (10,11)}.

O/P → 3

Greedy → sort according to finish time
→ Initialize the solution with first activity (i.e min finish time)
→ DO the following for remaining
① If current activity overlaps with the last picked activity, ignore the current activity
② Else add the current activity

I/P → {(3,8), (2,4), (1,3), (10,11)}

sorted → {(1,3), (2,4), (3,8), (10,11)}

ans {(1,3), (3,8) (10,11)}

---

Implementation

```
int maxActivities (pair<int,int> arr[], int n){
    sort(arr, arr+n, mycmp);
    int prev = 0
    int res = 1;
    for (int curr = 1; curr < n ; curr++){
        if (arr[curr].first ≥ arr[prev].second)
        {
            res++;
            prev = curr;
        }
    }
    return res;
}
bool mycmp (pair<int, int> a , pair<int, int> b)
    return a.second < b.second;
```

# FRACTIONAL KNAPSACK

- collect maximum value in knapsack.

I/P

| weight | 50 | 20 | 30 |
|---|---|---|---|
| values | 600 | 500 | 400 |

knapsack capacity = 70

O/P → $I_1 + I_3 + \frac{20}{50} \times 600$

= 500 + 400 + $\frac{20}{50} \times 600$

= 1140

## Algorithm

① calculate ratio (value/weight) for every item.

② sort all items in decreasing order.

③ initialise res = 0, curr_cap = given_cap.

④ Do the following for every item I in sorted order

```
if (I.weight <= curr_cap) {
        curr_cap -= I.weight
        res += I.value;
}
else {
        res += (curr_cap / I.weight) * I.value;
        return res.
}
```

⑤ return res.

# Job Sequencing

I/P →

| | $J_0$ | $J_1$ | $J_2$ | $J_3$ |
|---|---|---|---|---|
| deadline | 4 | 1 | 1 | 1 |
| profit | 70 | 80 | 30 | 100 |

O/P → 170

I/P →

| | $J_0$ | $J_1$ | $J_2$ | $J_3$ | $J_4$ |
|---|---|---|---|---|---|
| deadl. | 2 | 1 | 2 | 1 | 3 |
| profit | 100 | 50 | 10 | 20 | 30 |

→ 180

→ one unit by every job

→ only one job can be assigned at a time

→ time start with 0

→ maximize the profit

## Algorithm

① sort jobs in decreasing order of profit

② initialize the result as first job in the sorted list

③ Do following for the remaining (n-1) jobs.

(a) if this job can not be added, ignore it

(b) else add it to the latest possible slot.

| 4 | 1 | 1 | 5 | 5 |
|---|---|---|---|---|
| 50 | 5 | 20 | 10 | 80 |

⇒

| $J_4$ | $J_0$ | $J_2$ | $J_3$ | $J_1$ |
|---|---|---|---|---|
| 5 | 4 | 1 | 5 | 1 |
| 80 | 50 | 20 | 10 | 5 |

| $J_2$ | | $J_3$ | $J_0$ | $J_4$ |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

80 + 50 + 20 + 10

# Huffman Coding

→ used for lossless compression
→ variable length coding

**Example problem** → "abaacbaca....".

100 chars

Frequency

| | |
|---|---|
| a → 70 | 00 |
| b → 20 | 01 |
| c → 10 | 10 |

fixed length encoding → 2 × 100 = 200

variable length encoding
↳ different characters might have different length codes

**Greedy Idea**
most frequent chars has smallest code

**prefix Requirement for decompression**
no code should be prefix of any other.

| | | | | |
|---|---|---|---|---|
| a → 70 | 0 | 1 | 0 | 1 |
| b → 20 | 10 | 01 | 0 | 0 |
| c → 10 | 11 | 00 | 01 | 11 |
| | prefix free | | not prefix free | |

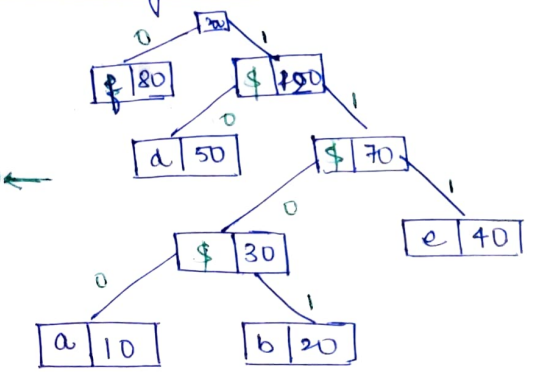
111
↓
aca
or
ca
or
ac

Space → 70×1 + 20×2 + 10×2
= 130 < 200 ✓

# Huffman algorithm

I/P → ['a', 'b', 'c', 'd', 'f']
[10, 50, 20, 40, 80].

1) Build a binary tree




full binary tree

- Every input character is a leaf
- Every left child edge is labelled as 0 and right is labelled as 1
- Every root to leaf path represent Huffman code

a → 0 0 1 1
b → 1 0 1 1
c → 1 1 1
d → 0 1
f → 0

**Implementation**

① we will use min heap data structure here
② create the leaf nodes for every element of an array and put them in min heap
③ now while (h.size() > 1)
  ④ left = h.extractMin();
  ⑤ right = h.extractMin();
  ⑥ create a new node with
    → character '$'
    → frequency as left.freq + right.freq.
    → left and right children as left & right
  ⑦ insert new node into h

* The only node left in h is required binary tree.

```
void printCodes (root, str=" ") {
    if (root == null)
        return ;
    if (root.data != '$')
        print (root.ch + " " + str)
        return
    printCodes (root.left , str + "0");
    printCodes (root.right , str + "1");
};
```

Implementation

```
struct Node {
    int freq;
    char ch;
    Node *left , *right;
    Node (int f , char c , Node *l = NULL,
                           Node *r = NULL)
    {
        freq = f;
        ch = c;
        left = l
        right = r;
    }
};

void createBtree (int arr[] , int freq[] , int n){
    priority-queue < Node* , vector<Node*>,
                                compare > h;
    for (int i = 0; i<n ; i++)
        h.push (new Node (freq[i], arr[i]));
    while (h.size()>1) {
        Node *l = h.top(); h.pop();
        Node *r = h.top(); h.pop();
        Node *node = new Node ('$', l->freq + r->freq , l, r);
        h.push (node);
    }
    printCodes (h.top(), " ");
};

void printCodes() ← already written

struct compare {
    bool operator() (Node* l , Node* r) {
        return l->freq < r->freq;
    }
};
```

time comp → O(NlogN) + O(NlogN) + O(logN).

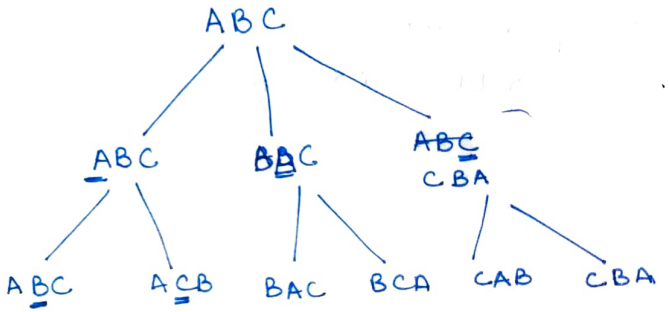
## Backtracking

Given a string, print all the permutation which do not contain "AB" as a substring

string → "ABC"

### Naive solution

generate all permutations and before printing we check if 'ab' is a substring




Fix a character and swap the next one with the remaining of string.

curr_index ↓ , ↓ length

```
void permute (string str, int l, int r){
    if (l == r){
        print (str)
        return;
    }
    else {
        for (int i = l; i <= r; i++){
            swap ( str[i], str[l]);
            permute ( str, l+1, r);
            swap ( str[i], str[l];
        }.
}
```

we can not call recursion after the currently generated string contains 'AB' ~~not~~ in it.

It can reduce a lot of recursive calls.

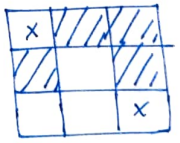
Hence before calling swap & permute we can just include bool is_safe

```
bool is_safe (string str, int l, int i, int r) {
    if ( l != 0 && str[l-1] == 'A' && str[i] == 'B')
        return false;
    if (r = l+1 && str[i] == 'A' && str[i] == 'B')
        return false;
    Return true;
}.
```

⟹ Cut down Recursive calls

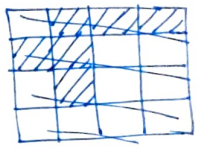
## Rat in a Maze

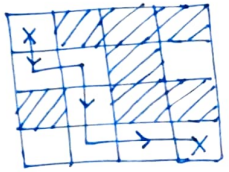
I/P → {{1, 0, 0},
{0, 1, 0},
{1, 1, 1}}



(NO)

I/P → {{1, 0, 0, 0}
{1, 1, 0, 1}
{0, 1, 0, 0}
{1, 1, 1, 1}}





(Yes)

only two moves req.

→ (i+1)(j)

~~(i+1)(j)~~

→ (i)(j+1)

↓

right & ~~left~~ down

⇒ boolean solveMaze() {

```
if (solveMazeRec(0,0) == false)
        return false;
    else {
        print(sol);
        return true;
    }
```

⇒ bool isSafe(int i, int j) {

```
return (i<n && j<N && m[i][j]==1)
```

}.

⇒ boolean solveMazeRec(int i, int j) {

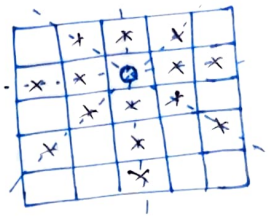
```
if (i == N-1 && j == N-1){sol[i][j]=1;
                          return true;}

if (isSafe(i,j) == true) {
    sol[i][j] = 1;
    if (solveMazeRec(i+1, j) == true)
        return true;
    if (solveMazeRec(i,j+1) == true)
        return true;
    sol[i][j] = 0;
}
return false
```

}.

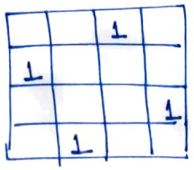
## N Queen Problem

- placing n queens so that no two can attack each other.
- A queen can attack horizontally, vertically and diagonally



N = 4

O/P → Yes



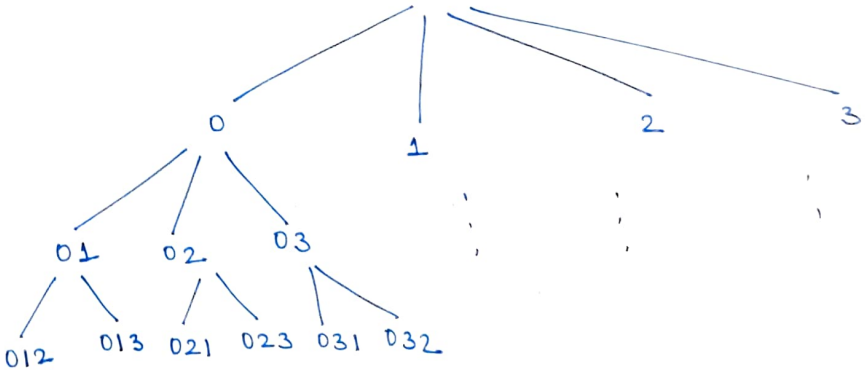

N = 3 O/P → NO

### Super Naive Solution

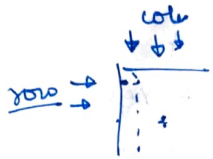
generate all possible permutations i.e. ${}^{n^2}C_n$.

### Naive Solution

* placing queens in different columns. hence generate permutations of in their columns itself.



```
⇒ bool solve() {
    if (solveRec(0) == false)
        return false;
    else {
        printMatrix(board);
        return true;
    }
}
```




```
⇒ bool solveRec(int col) {
    if (col == N) return true;
    for (int i=0; i<N; i++) {
        if (isSafe(i, col)) {
            board[i][col] = 1;
            if (solveRec(col+1))
                return true;
            board[i][col] = 0; }
    }
    return false;
}
```

```
⇒ bool isSafe(int row, int col) {

horizontal check { for(i→n)
        if(board(row)(i)) return false;

upper diagonal { for(i=row, j=col;
        i>=0 && j>=0; i--, j--)
    if(board(i)(j)) return false.

lower diagonal { for(i=row, j=col; j>=0 && i<n; i++, j--)
        if(board(i)(j)) return false;
return true;
```

# SUDUKO PROBLEMS

The size of suduko is always a square
4x4, 9x9, 16x16

① Numbers we are gonna fill is always from
1 → N

- Every number in every row is different
- Every number in every column is different.
- Every number in size subgrid 2x2 must also be different.

```
bool isSafe(int i, int j, int n) {
    for (int k=0; k<n; k++) {
        if (grid[k][j] == n || grid[i][k] == n)
            return false;

    int s = sqrt(n);
    int rs = i - i%s;
    int cs = j - j%s;
    for (int i=0; i<s; i++){
        for (int j=0; j<s; j++){
            if (grid[i+rs][j+cs] == n)
                return false;
    }}
}
bool solve(){
    int i, j;
    for (int i→n)
        for (j→n)
            if (grid[i][j]==0) break;
```

```
if (i == n && j == n)
        return true;

for (int num = 1; num <= n; num++) {
    if (isSafe(i, j, num)) {
        grid[i][j] = num;
        if (solve()) return true;
        grid[i][j] = 0;
    }
}
return false;
}.
```