

- ① Now we know that total no. of games = n  
 A → needs has played i games  
 B → has played j games

So let  $P(i, j)$  be the probability that A wins in i games and B wins in j games  
 We come up with a set of equations as follows:-

$$\begin{aligned} P(i, j) &= 1 \quad \text{if } i=0 \text{ and } j>0 \quad \dots \quad \text{A wins} \\ &= 0 \quad \text{if } i>0 \text{ and } j=0 \quad \dots \quad \text{B wins} \\ &= [P(i-1, j) + P(i, j-1)]/2 \quad \dots \quad \text{equal probability.} \end{aligned}$$

Now to compute the time period/complexity of the above equation we get  $\geq O(2^i \cdot 2^j) = O(2^{i+j})$   
 where  $i+j=n$

∴ The total time complexity  $\geq O(2^n)$

Since this is done by a Recursive algorithm the disadvantage of this is that all the previous values of the probability need to be re-computed for any given  $i, j$

The solution to this is dynamic programming where we start filling the values diagonally from the bottom.

Time complexity for the proposed solution  $\rightarrow O(n^2)$

define calculate\_probability (int i, int j)

{  
 int a, b; int n = i+j;  
 for (int x=1; x≤n; x++)

$$P[0, n] = 1$$

$$P[n, 0] = 0$$

for (int y=1; y≤x-1; y++)

$$P[y, n-y] = P[y-1, n-y] + P[y, n-y-1]$$

$$P[y, n-y] = P[y, n-y]/2;$$

} return P[i, j];

This is an efficient solution as even the lower bound limit for recursive algorithm is  $O(2^n/\sqrt{n})$   
This grows much faster than  $O(n^2)$  of -  
dynamic programming

② The algorithm goes as follows:-

- if  $1 \leq n \leq n$  and  $0 \leq y \leq t$  then we can say that  $T(n, y)$  exist if and only if the input adds up to  ~~$\neq y$~~  which is eventually less than equal to the integer 't'.
- if  $a_j \leq y$  and we include that in our subset there needs to be another elements in the input which adds up to  $y - a_j$ .
- if  $a_j \leq y$  and we do not include that in our subset then the input should consist of elements that add up to  ~~$\neq y$~~ .

∴ Subset  $T = T(n-1, y) \cup (a_j \leq y) \text{ iff } T(n-1, \frac{y}{a_j} - a_j)$

Algorithm:-

define my-function (int t, int n):

for ( $y=1$ ;  $y \leq n$ ;  $y++$ )

for ( $x=0$ ;  $x \leq t$ ;  $x++$ )

{ if ( $y == 1$ )

subset T = { include  $x=0$  OR  $x=a_1$  }

} else if ( $a_j \leq x$ ) where  $j=y$ .

subset T = { include ~~T<sub>j</sub>~~  $\oplus^v T(y_1, n-a_j)$  }

} else

{ subset T = { include  $T[y-1, n]$  } }

}

} return T[n, t] —— return the subset.

### ③ Divide and Conquer Method

Algorithm:-

\* define My-function(int []A, int start, int end)

{ if (start >= end) return null;

{

int Mid = start + (end - start) / 2;

int left\_half = My-function(A, start, Mid);

int right\_half = My-function(A, Mid+1, end);

int min = A[start]

for (int n = <sup>start+1</sup>; n ≤ mid; n++)

{

if (A[n] < min)

{

min = A[n];

}

}

int left\_min = min;

int <sup>max</sup>Min = A[Mid];

for (int n = Mid+1; n ≤ end; n++)

{

if (A[n] > <sup>max</sup>Min)

{

<sup>max</sup>Min = A[n];

,

}

int right\_max = min;

return Math.Max((right\_max - left\_min), left\_half, right\_half),  
(right\_max, left\_min))

}

diff, A[j], A[:] ~~difference~~ = My-function(A[], start, end);

## Dynamic Programming Algorithm

```
def my_function(int [] A)
{
    int diff = -1
    int Max = A[A.length - 1]
    int a, b
    for (int i = A.length - 2; i >= 0; i--)
    {
        if (Max < A[i])
            Max = A[i]
        else if (Max > A[i])
        {
            diff = Max(Max, Max - A[i])
            if (diff == Max - A[i])
                a = Max
                b = A[i]
        }
    }
    return diff, a, b;
}
```

- difference, A[j], A[i] = My\_function (A[ ])

References:- (for Divide & Conquer and Dynamic programming)

<https://algorithms.tutorialhorizon.com/maximum-difference-between-two-elements-where-larger-element-appears-after-the-smaller-element/>

④ For this problem we follow the following steps:-

- Create a subset  $S$  of all the potential guest  
ie.  $S \subseteq \{1, \dots, n\}$
- Create another subset known ' $k$ ' where  $k \in \{\text{the number of people a person } i \text{ knows}\}$
- Create another subset Don't-know  $DK \subseteq \{\text{the number of people a person } i \text{ dont know}\}$

Hence we come up with a simple Algorithm.

while  $i \in S$

```
{  
    if  $k_i < 5$  and  $DK_i < 5$   
    {  
        // remove that person from list  $S \rightarrow del(i, S)$   
    }  
    else  
    {  
        // the person stays in the guest-list  $S$   
    }  
}  
return  $S$ .
```

→ the run time of this algorithm  $\rightarrow O(n^2)$

Since there are  $O(n)$  iterations and in each we will need to scan through the remaining possible invitees.

(5)

First we sort the array of start and end-times using quick sort

```
def quicksort (arr [], start, end):  
    if (start < end)  
    {  
        part = partition (arr [], start, end)  
        quicksort (arr [], start, part - 1)  
        quicksort (arr [], part + 1, end)  
    }
```

```
def partition (arr [], start, end):  
    pivot = arr [end]  
    temp = start
```

```
    for i = start to end - 1:  
        if arr [i] < pivot:  
            swap (arr [temp], arr [i])  
            temp = temp + 1  
    swap (arr [temp], arr [end])  
    return (temp).
```

// Now the problem is a circular queue of all the time intervals sorted together. We want to cut all the jobs that occur at midnight and gets past it so that it does not violate the 24 hr window period. So we call this point ~~as~~ <sup>as</sup> j. Choose a point 'x' on j

// Now Remove all the overlapping intervals that contain the point j

```
def (& Remove_overlaps (Ij, n)  
{  
    → Removing all intervals containing the point x  
}
```

// Now we simply use greedy algorithm to solve the  
// remaining jobs using interval scheduling

```
def interval_scheduling(start_time, end_time)
```

```
{  
    temp = 0, list1 = []  
    for i in end_time:  
        if (start_time[i] > end_time[i + temp]):  
            list1.add(i)  
        temp = end_time[i]  
    return list1  
}
```

// Now we will get a maximum set of subset of  
// activities

// However this does not guarantee an optimal solution  
// so we do this for  $n \in [1, n]$  where n is  
// the total number of intervals and select the  
// maximum of these solutions  
~~Hence this last step is~~

```
def repetition()
```

```
{  
    for time_interval = [...] // sorted time-intervals  
    set = []
```

```
    for n in time_interval
```

```
    O(n^2) {  
        set.append(interval)  
        remove_overlaps()  
        set.append(interval_scheduling(start, end))  
    }  
    → select the Max (set)
```

d. // This guarantees optimal solution and the total  
// time complexity now is  $O(n^2)$ .