BRUTE-FORCE $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p[1], p[2], \ldots, p[n]$ via binary search.

RETURN COMPUTE-OPT(n).

In [349]:
```
n=8
#El formato que siguen las tripletas es [S_j,F_j,W_j], es decir [tiempo inicio, tiempo final, ganancia]

tasks=[[0,6,3],[1,4,5],[3,5,1],[3,8,8],[4,7,4],[5,9,4],[6,10,2],[4,11,3]]

tasks
```

Out[349]:
```
[[0, 6, 3],
 [1, 4, 5],
 [3, 5, 1],
 [3, 8, 8],
 [4, 7, 4],
 [5, 9, 4],
 [6, 10, 2],
 [4, 11, 3]]
```

In [350]:
```
from operator import itemgetter
#Ordena los trabajos por tiempo de finalización
tasks=sorted(tasks,key=itemgetter(1))
for i,t in enumerate(tasks):
    print(f'Trabajo {i+1} => finaliza en {t[1]}')

tasks
```

```
Trabajo 1 => finaliza en 4
Trabajo 2 => finaliza en 5
Trabajo 3 => finaliza en 6
Trabajo 4 => finaliza en 7
```

In [352]:
```
#La búsqueda binaria va buscar un elemento en un arreglo ordenado. Lo compara contra el
#elemento de la mitad del arreglo, si es menor, descarta la segunda mitad del arreglo y
#vuelve a comparar el elemento buscado contra el elemento de la mitad del primer arreglo
def binary_search(A,index,start,end):
    mid=int((start+end)/2)
    if index==A[mid]:
        return mid
    elif start==end:
        return -1
    elif index<A[mid]:
        return binary_search(A,index,0,mid-1)
    else:
        return binary_search(A,index,mid+1,end)
```

In [353]:
```
#probamos que la busqueda binaria funcione
f=[4,5,6,7,8,9,10,11]
#binary_search(vector,index,start,end)
#En este ejemplo estamos buscando el número 6 en el arreglo, el cual está en la posición 2
binary_search(f,32,0,7)
```

Out[353]: -1

In [354]:
```
#Extraemos la lista de los tiempos finales de la lista de trabajos (tasks) y también las ganancias de cada trabajo
t_f=[]
weigths=list()
for i in tasks:
    t_f.append(i[1])
    weigths.append(i[2])
print(t_f)
weigths
```

```
[4, 5, 6, 7, 8, 9, 10, 11]
```

Out[354]:
```
[5, 1, 3, 4, 8, 4, 2, 3]
```

In [355]:
```
#ahora vamos calcular los p()'s de todos los procesos
w=[]
print(tasks)
for index,trabajo in enumerate(tasks):
    print(f'trabajo {index+1}, inicia {trabajo[0]}, lista de tiempos finales {t_f}')
    w.append(binary_search(t_f,trabajo[0],0,7))
```

```
[[1, 4, 5], [3, 5, 1], [0, 6, 3], [4, 7, 4], [3, 8, 8], [5, 9, 4], [6, 10, 2], [4, 11, 3]]
trabajo 1, inicia 1, lista de tiempos finales [4, 5, 6, 7, 8, 9, 10, 11]
trabajo 2, inicia 3, lista de tiempos finales [4, 5, 6, 7, 8, 9, 10, 11]
trabajo 3, inicia 0, lista de tiempos finales [4, 5, 6, 7, 8, 9, 10, 11]
trabajo 4, inicia 4, lista de tiempos finales [4, 5, 6, 7, 8, 9, 10, 11]
trabajo 5, inicia 3, lista de tiempos finales [4, 5, 6, 7, 8, 9, 10, 11]
trabajo 6, inicia 5, lista de tiempos finales [4, 5, 6, 7, 8, 9, 10, 11]
trabajo 7, inicia 6, lista de tiempos finales [4, 5, 6, 7, 8, 9, 10, 11]
trabajo 8, inicia 4, lista de tiempos finales [4, 5, 6, 7, 8, 9, 10, 11]
```

In [356]:
```
#Estos son los valores p()'s de los trabajos *** List comprenhension
p=[i+1 for i in w]
print(p)
len(p)
```

```
[0, 0, 0, 1, 0, 2, 3, 1]
```

Out[356]:
```
8
```

In [357]:
```
list(map(lambda i: i+1,w)) #***** funciones lambda y map
```

Out[357]:
```
[0, 0, 0, 1, 0, 2, 3, 1]
```

Ahora tenemos que programar la función COMPUTE-OPT(n)

COMPUTE-OPT(j)

IF ($j = 0$)

  RETURN 0.

ELSE

  RETURN max {COMPUTE-OPT($j-1$), w_j + COMPUTE-OPT($p[j]$) }.

In [358]:
```
#Acuerdense que los vectores empiezan desde el índice 0, entones la posición del trabajo 8, sería p[7], pero para evita
#esta confusión, mejor insertamos un cero en la posición 0 del vector, y ya la p de 8 si sería la p[8] ;)
#  M[j]=max(McomputeOpt(j-1,p,w,M),w[j]+McomputeOpt(p[j],p,w,M))

p.insert(0,0)
weigths.insert(0,0)

def computeOpt(j,p,w):
    if j==0:
        return 0
    else:
        return max(computeOpt(j-1,p,w),w[j]+computeOpt(p[j],p,w))
```

```
In [360]: print(computeOpt(8,p,weigths))
```

```
9
```

{ Memoization: The **Ultimate** Guide }

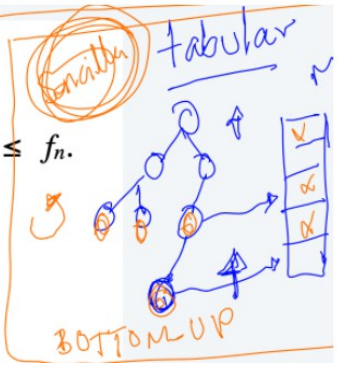
TOP-DOWN($n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n$)

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p[1], p[2], \ldots, p[n]$ via binary search.

$M[0] \leftarrow 0$. ⟵ global array

RETURN M-COMPUTE-OPT(n).



M-COMPUTE-OPT(j)

IF ($M[j]$ is uninitialized)

$M[j] \leftarrow \max \{$ M-COMPUTE-OPT$(j-1)$, w_j + M-COMPUTE-OPT$(p[j])$ $\}$.

RETURN $M[j]$.

n [361]:
```
#Inicializamos el vector M
M=[]
for i in range(9):
    M.append(-1)
M
#print(p)
```

ut[361]:
```
[-1, -1, -1, -1, -1, -1, -1, -1, -1]
```

n [362]:
```
def McomputeOpt(j,p,w):
    if j>0:
        if M[j]==-1:
            ##En esta parte iría 1 línea de código, que es la que M[j]=max..... (así como se muestra arriba)
            M[j]=max(McomputeOpt(j-1,p,w),w[j]+McomputeOpt(p[j],p,w))
        return M[j]
    else:
        return 0
```

n [363]:
```
print(McomputeOpt(8,p,weigths))
```

```
9
```