




# Algoritmos BackTracking y Branch and Bound

Suma mínima de cuadrados sumando N  
Problema de los Recipientes





# PROBLEMA DE LA SUMA MÍNIMA DE CUADRADOS SUMANDO $N$



# Representación vector Solución

El vector solución tiene un tamaño fijo  $N$ , siendo  $N$  el número que tiene que sumar en total el conjunto representado por el vector.

Cada  $x[i]$  representa al número  $i+1$  y toma un valor del conjunto  $\{2, 1, 0, -1\}$ .

- ★  $2 \rightarrow$  NULO
- ★  $1 \rightarrow$  El valor  $i+1$  se toma como sumando
- ★  $0 \rightarrow$  El valor  $i+1$  no se toma como sumando
- ★  $-1 \rightarrow$  END

# Restricciones implícitas y explícitas

Restricciones explícitas:

- ★ Cada  $x[i]$  toma un valor del conjunto  $\{2, 1, 0, -1\}$ .
- ★ Los números a los que representa cada posición están ordenados de menor a mayor.
- ★ El valor de la posición  $i$  del vector  $x$  se corresponde con el número  $i+1$ .

Restricciones implícitas:

- ★ La suma de los números que representa el vector debe ser  $N$ .
- ★ Para saber si es factible un vector solución que se está recorriendo aún y cuya suma aún no sea  $N$  se deben cumplir dos condiciones, siendo  $k$  la posición del vector en la que nos encontramos:
  - ☆ La suma de los números hasta la posición  $k$  + la suma de los números que quedan por sumar (desde  $k+1$  hasta  $n$ ) debe ser mayor o igual que  $N$
  - ☆ La suma de los números hasta la posición  $k$  más el siguiente elemento no debe de ser mayor que  $N$ , ya que como los elementos están ordenados, si seguimos recorriendo más allá del siguiente elemento no se podrá encontrar una solución, ya que todos son mayores a éste.

# Suma de cuadrados - Clase Solución

```
class Solucion {
private:
    vector<int> x;
    int sumaCuadrados;
    pair< int, vector<int> > mejorSol;

public:
    Solucion(const int N);
    int size() const;
    void iniciaComp(int k);
    void sigValComp(int k);
    bool todosGenerados(int k);
    int decision(int k) const;
    void procesaSolucion();
    bool factible(int k);
    void mostrarMejorSol();
    ~Solucion() = default;
};
```

# Función recursiva

```
void back_recursivo(Solucion &Sol, int k){  
    if (k == Sol.size())  
        Sol.procesaSolucion();  
  
    else{  
        Sol.iniciaComp(k);  
        Sol.sigValComp(k);  
        while (!Sol.todosGenerados(k)){  
            if (Sol.factible(k))  
                back_recursivo(Sol, k + 1);  
  
            Sol.sigValComp(k);  
        }  
    }  
}
```

# Función procesaSolucion()

```
void Solucion::procesaSolucion(){  
    if(sumaCuadrados < mejorSol.first){  
        mejorSol.first = sumaCuadrados;  
        mejorSol.second = x;  
    }  
}
```

# BackTracking sin cotas

```
bool Solucion::factible(int k){
    assert(k >= 0 && k < x.size());
    int suma = 0, sumaRestantes = 0;
    for (int i = 0; i <= k; i++){
        suma += x[i] * (i+1);
        sumaCuadrados += (x[i] * (i+1)) * (x[i] * (i+1));
    }

    for (int i = k+1; i < x.size(); i++)
        sumaRestantes -= i+1;

    return ( ((suma + sumaRestantes >= x.size()) &&
            (suma + k + 2) <= x.size() ) || suma == x.size());
}
```



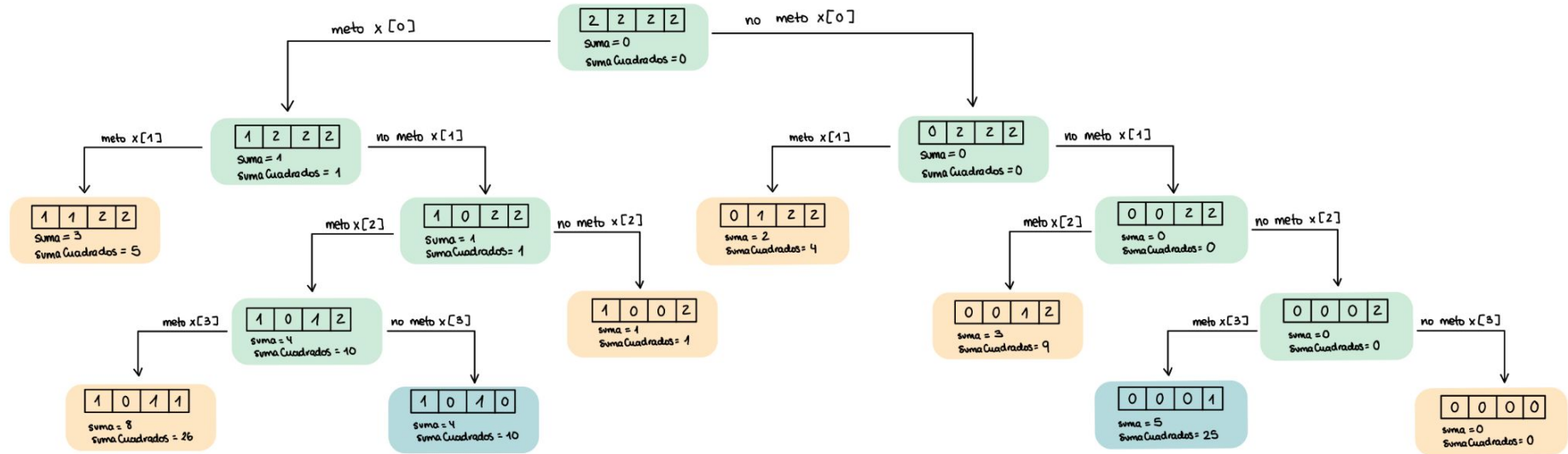
# BackTracking con cotas

```
bool Solucion::factible(int k){
    assert(k >= 0 && k < x.size());
    int suma = 0, sumaRestantes = 0;
    for (int i = 0; i <= k; i++){
        suma += x[i] * (i+1);
        sumaCuadrados += (x[i] * (i+1)) * (x[i] * (i+1));
    }

    for (int i = k+1; i < x.size(); i++)
        sumaRestantes -= i+1;

    return ( ((suma + sumaRestantes >= x.size()) &&
        (suma + k + 2) <= x.size() ) || suma == x.size() &&
        sumaCuadrados < mejorSol.first );
}
```

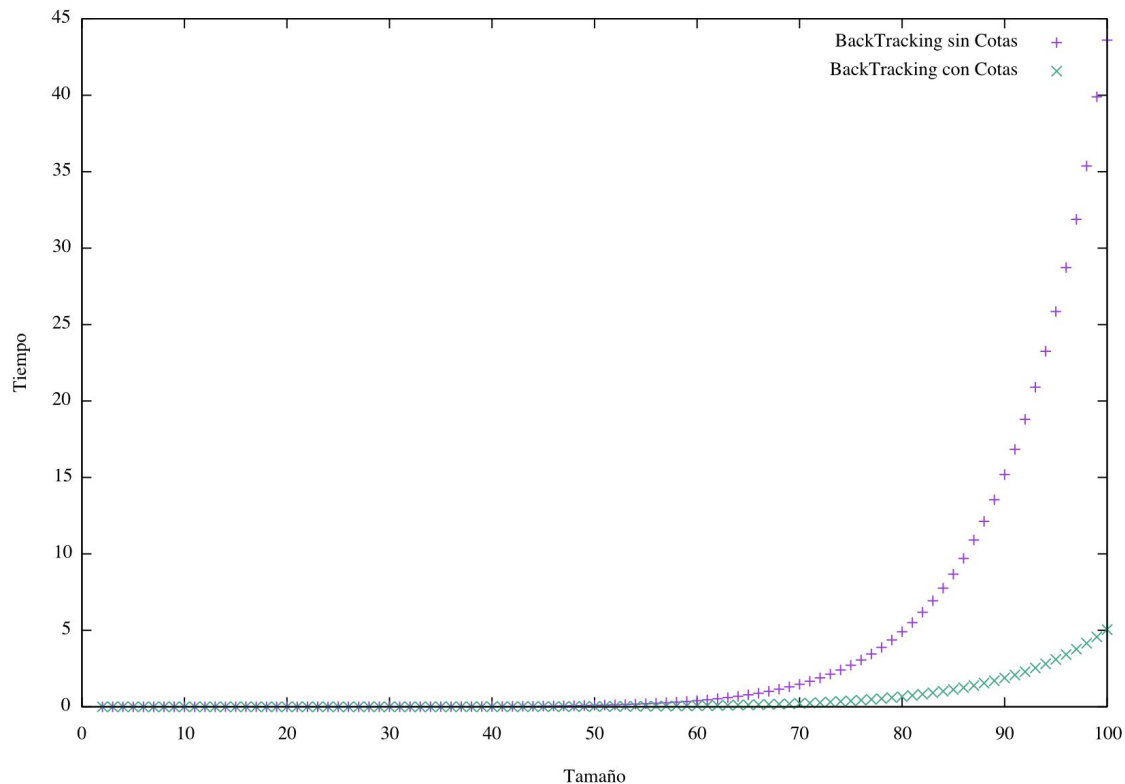
# Ejemplo Árbol de estados para $N = 4$



# Suma de cuadrados - Funciones de factibilidad

Hemos comparado la eficiencia empírica de el algoritmo backtracking sin cotas y las del algoritmo con cotas. Hemos comprobado que al añadirle cotas al algoritmo de BackTracking, el tiempo de ejecución se reduce considerablemente puesto que en la mayoría de casos se marca un hijo como no factible mucho antes, ya que en el momento que la suma de cuadrados que llevamos calculada hasta el momento es mayor que la mejor solución calculada hasta el momento se poda.

# Suma de cuadrados - Eficiencia empírica



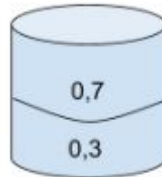


# PROBLEMA DE LOS RECIPIENTES



# Representación Gráfica del problema de los Recipientes

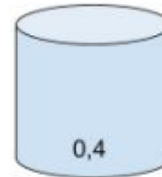
Vector objetos	0,3	0,2	0,7	0,5	0,2	0,1	0,4
Vector solución	1	2	1	2	2	2	3



Recipiente 1



Recipiente 2



Recipiente 3

# Problema de los Recipientes - Clase Solución

```
class Solucion{
private:
    vector<int> x;
    vector<double> objetos;
    int pos_e;
    int numRecipientes;
    double peso;
    int estimador;

public:
    Solucion(const int tam_max, vector<double> obj);
    bool Factible() const;
    void PrimerValorComp(int k);
    void sigValComp(int k);
    int comp(int k) const;
    int cotaLocal() const;
    bool hayMasValores(int k) const;
    int evalua();
    int numComponentes() const;
    bool esSolucion() const;
    int compActual();
    void actualizaEstimador(int k);
    void actualizarNumRecipientes();
    bool operator<(const Solucion &s) const;
    friend ostream &operator<<(ostream &os, const Solucion &Solucion);
};
```

# Recipientes - funciones clave

```
bool Solucion::Factible() const{
    //Si peso acumulado es mayor que numero de
    recipientes no ejecuta el bucle
    if ( peso > numRecipientes ){
        return false;
    }
    double suma = 0.0;
    for (int i = 1; i <= numRecipientes; i++){
        suma = 0.0;
        for(int j = 0; j < x.size(); j++){
            if(x[j] == i){
                suma += objetos[j];
                if(suma > 1){
                    return false;
                }
            }
        }
    }
    return true;
}
```

```
void Solucion::actualizaEstimador(int k){
    double num;
    for (int i = k+1; i < objetos.size();
    i++)
        num += objetos[i];

    int recisAprox = num;
    estimador = recisAprox + numRecipientes;
}
```



# Recipientes - funciones clave

```
bool Solucion::esSolucion() const{

    //comprueba que todos los elementos != NULO y ningún
    recipiente se desborda
    bool sol = true;

    for (int i = 0; i < x.size() && sol; i++)
        if (x[i] == 0)
            sol = false;

    double suma = 0.0;
    for (int i = 1; i <= numRecipientes && sol; i++){
        suma = 0.0;
        for(int j = 0; j < x.size() && sol; j++)
            if(x[j] == i){
                suma += objetos[j];
                if(suma > 1)
                    sol = false;
            }
        }
    return sol;
}
```

```
void Solucion::actualizarNumRecipientes(){
    for (int i = 0; i < objetos.size(); i++){
        if (numRecipientes < x[i])
            numRecipientes = x[i];
    }
}
```

```

Solucion Algoritmo_BB(vector<double> pesos)
{
    priority_queue<Solucion> Q;
    Solucion n_e(pesos.size(), pesos), mejor_solucion(pesos.size(), pesos);
    //nodoE en expansion
    int k;
    float CG = INT_MAX; // Cota Global
    float ganancia_actual;

    Q.push(n_e);
    while(!Q.empty() && (Q.top().cotaLocal() < CG)) {
        n_e = Q.top();

        Q.pop();
        k = n_e.compActual();

        for (n_e.PrimerValorComp(k + 1); n_e.hayMasValores(k + 1)
            && (k+1) < n_e.numComponentes(); n_e.sigValComp(k + 1)){
            if (n_e.esSolucion()){
                ganancia_actual = n_e.evalua();
                if (ganancia_actual < CG){
                    CG = ganancia_actual;
                    mejor_solucion = n_e;
                }
            } else{
                if (n_e.Factible() && n_e.cotaLocal() < CG){
                    Q.push(n_e);
                }
            }
        }
    }
    return mejor_solucion;
}

```

# Comparación B&B con sol. FuerzaBruta

