Erick Vargas

Club algoritmia (ESCOM)

Contents

1 Búsqueda binaria

1.1 Planetas

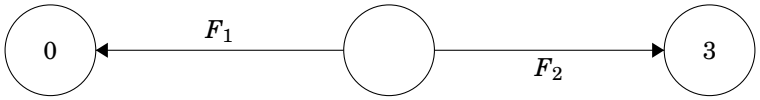
Tienes un conjunto de planetas y te dan las posiciones de estos planetas, quieres colocar un meteorito en cualquier lugar entre los planetas, el meteorito debe quedarse en su lugar y no ser atraido por la fuerza gravitacional de los planetas. Sabemos de física básica que la sumatoria de todas las fuerzas debe ser igual con cero. Sabemos que:

- $F_1 = F_2$ esta en equilibrio el sitema
- $F_1 > F_2$
- $F_1 < F_2$

Tenemos como información la siguiente fórmula:

$$\frac{1}{|X_i - M|}$$

Ejemplo:



Si utilizamos las fórmulas calculamos el valor de F_1, F_2

$$F_1 = \frac{1}{|0 - 1.5|} = \frac{1}{1.5}$$

y

$$F_2 = \frac{1}{|3 - 1.5|} = \frac{1}{1.5}$$

Podemos ver que si restamos ambas fuerzas efectivamente el sistema está en equilibrio.

Pero ¿Qué pasa si tenemos más de dos planetas? Tenemos n - 1 soluciones ya que debemos hacer binaria entre cada pareja de planetas, si tomamos solo el planeta origen y el más alejado vamos a perder muchas soluciones.

Recordemos que el meteorito está en una posición válida la suma de las fuerzas debe ser igual con cero:

$$F_1 + F_2 + F_3 + F_3 = 0$$

¿Que ocurre si la suma de fuerzas no es igual con cero? Debemos de mover nuestra binaria. Si la suma es mayor que cero nos movemos a la izquierda, si es menor que cero nos movemos a la derecha.

Tomar en cuenta que si eliminas de la fórmula el valor absoluto no necesitaremos comprobar la posición del planeta en la que estamos. Si es negativo el resultado de el elemento de la fórmula que estamos haciendo entonces la fuerza que tenemos esta a la derecha en otro caso está a la izquierda.

```cpp
#include <bits/stdc++.h>

using namespace std;
vector< double > planets;
int n;
//\sum^{ n }_{ 0 }{ \frac{ 1 }{ X_{i} - M } }
double SumOfForces( double middle ){
    double sum = 0.0;
    for( int i = 0; i < n; i++ ){
        sum += 1 / ( planets[ i ] - middle );
    }
    return sum;
}

int main(){
    ios::sync_with_stdio( false );
    cout.tie( nullptr );
    cin.tie( nullptr );
    cin >> n;
    planets.resize( n );
    for( int i = 0; i < n; i++)
        cin >> planets[ i ];
    sort( planets.begin(), planets.end() );
    cout << n - 1 << endl;
    for( int i = 0; i < n - 1; i++ ){
        double begin = planets[ i ];
        double end = planets[ i + 1 ];
        double middle;
        for( int i = 0; i < 25; i++ ){
            middle = ( begin + end ) / 2;
```

```
                if( SumOfForces( middle ) < 0 ){
                    begin = middle;
                } else {
                    end = middle;
                }
            }
            cout << fixed;
            cout << setprecision( 3 );
            cout << middle << " ";
        }
        cout << endl;
        return 0;
    }
```

2 Programación dinámica

Partimos de una solución recursiva bruta y podemos hacer uso de una función de memorización. El ejemplo más clásico es el de Fibonacci

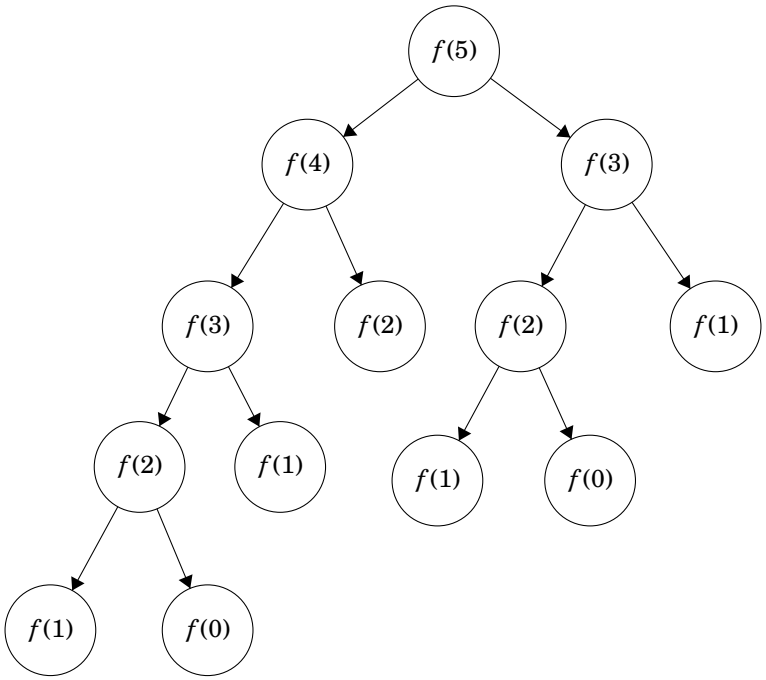
2.1 Fibonacci

$$fibonacci(n) = \begin{cases} 0 & , n = 0 \\ 1 & , n = 1 \\ fibonacci(n-1) + fibonacci(n-2) & , n \geq 2 \end{cases}$$

En código

```
int fibonacci( int n ){
    if( n == 0 )
        return 0;
    if( n == 1 )
        return 1;
    return fibonacci( n - 1 ) * fibonacci( n - 2 );
}
```

Si dibujamos las llamadas recursivas como un árbol tenemos lo siguiente



Como podemos observar hay llamadas recursivas que se repiten varias veces como ejemplo tenemos la llamada recursiva con un valor de 3, o 2. ¿Podemos evitar esto?

```
int memoria[ 100000 ];
.
.
.
int fibonacci( int n ){
    if( n == 0 )
        return 0;
    if( n == 1 )
        return 1;
    if( memoria[ n ] != -1 )
        return memoria[ n ];
    return memoria[ n ] = ( fibonacci( n - 1 ) * fibonacci(
        n - 2 ) );
}

int main(){
    memset( memoria, -1, sizeof(memoria) );
    .
    .
    .
}
```

La complejidad la podemos calcular viendo los estados, en nuestro caso es el tamaño de la memoria, por ejemplo si llamamos a fibonacci de 10 siempre ten-

emos el mismo resultado, por tanto multiplicamos los estados por la complejidad de la función en nuestro caso es $10^5 * constante$

2.2 Coeficiente binomial

$$Coef\_Bin(n,k) = \begin{cases} 1 & , n = k \\ 1 & , k = 0 \\ Coef\_Bin(n-1,k-1) + Coef\_Bin(n-1,k) & , k \leq n \end{cases}$$

Si programamos esta función tenemos lo siguiente:

```
int coef_bin( int n, int k ){
    if( n == k || k == 0 )
        return 1;
    return ( coef_bin( n - 1, k - 1 ) + coef_bin( n - 1, k )
         );
}
```

Si aplicamos DP

```
memoria[ 1000 ][ 1000 ];
.
.
.
int coef_bin( int n, int k ){
    if( n == k || k == 0 )
        return 1;
    if( mem[ n ][ k ] != -1 )
        return memoria[ n ][ k ];
    return memoria[ n ][ k ] = ( coef_bin( n - 1, k - 1 ) +
        coef_bin( n - 1, k ) );
}
```

En complejidad tenemos $O(n*k)$ y sin la función de memorización tenemos $O(2^n)$

2.3 Problema

Dado un grid de nxm, cada casilla tiene un número. Obtener un camino de la fila 1 a la fila n con suma máxima, ejemplo:

3	5	10
6	4	3
2	1	0

Como restricciones tenemos que $n, m \leq 10^3$ y además $A_{i,j} \leq 10^6$ Además solo nos podemos mover de la siguiente manera. Supongamos que estamos en el recuadro con el color gris más oscuro, estando en esa posición podemos movernos a cualquiera de los tres rectangulos con color gris más claro, es decir, podemos movernos hacía $(f+1,c),(f+1,c-1),(f+1,c+1)$

2.3.1 Solución

```
int max_sum( int f, int c ){
    //Primero comprobamos si es un caso válido
    if( c < 0 || c >= m )
        return MIN_INT //MIN_INT es como un menos infinito
    //Caso base
    if( f == n )
        return grid[ f ][ c ];
    int A = max_sum( f + 1, c - 1);
    int B = max_sum( f + 1, c );
    int C = max_sum( f + 1, c + 1 );
    //No exite una función max para 3 elementos, en dado
        caso podemos usar vector es decir ponemos max({A, B,
         C})
    return max( A, max(B, C) ) + grid[ f ][ c ];
}
```

Ahora bien ¿Cómo reducimos la complejidad de el código anterior? Debemos aplicar programación dinámica, primeramente ¿Cuál es el tamaño máximo de nuestra función de memoria? en nuestro caso es 1005 * 1005

```
int memoria[ 1005 ][ 1005 ];
.
.
.
int max_sum( int f, int c ){
    //Primero comprobamos si es un caso válido (que no esté
        fuera de rango)
    if( c < 0 || c >= m )
        return -1 //-1 es un valor bandera que identifica si
             ya visitamos o no la casilla
    //Caso base
    if( f == n )
        return grid[ f ][ c ];

    if( memoria[ f ][ c ] != -1 )
        return memoria[ f ][ c ]

    int A = max_sum( f + 1, c - 1);
    int B = max_sum( f + 1, c );
    int C = max_sum( f + 1, c + 1 );
    //No exite una función max para 3 elementos, en dado
        caso podemos usar vector es decir ponemos max({A, B,
         C})
    return memoria[ f ][ c ] = max( A, max(B, C) ) + grid[ f
         ][ c ];
}
```

2.4 Problema

Del ejercicio anterior que sucede si podemos añadir ¿números negativos? Podemos hacer uso de otro arreglo auxiliar de booleanos, en el que marcamos si ya visitamos o no una casilla

3

Problemas clásicos de DP

3.1 Mochilas

Vas a una tienda y quieres comprar algunos productos, digamos que hay n productos y cada producto tiene un precio y tenemos M pesos para gastar. Queremos comprar la mayor cantidad de productos gastando lo más posible.
Restricciones:

- N productos $N \leq 1000$
- M Pesos $M \leq 1000$
- $A_i \leq 1000$

Ejemplo:
N = 3, M = 8, y tenemos los productos A = 3, 4, 6 nuestro solución es comprar 3, 4 el precio es 7 y nos llevamos 4 productos.

Para poder solucionar este problema debemos de calcular el caso de probar y no probar el i-ésimo elemento. Es decir generamos todos los posibles subconjuntos del arreglo:

- ()
- (3)
- (3, 4)
- (3, 6)
- (4)
- (4, 6)

- (6)
- (3, 4, 6)

```
int max_sum( int indice, int pesos_restantes ){
    //Validación, n es el número de elementos del arreglo A
    if( indice >= n )
        return 0;
    //Como no es una respuesta válida para descartarla
        regresamos un menos infinito
    if( pesos_restantes < 0)
        return MIN_INT;
    //Recursivamente puedo tomar o no tomar un elemento
    int tomar = max_sum( indice + 1, pesos - A[ indice ]) +
        A[ indice ];
    int no_tomar = max_sum( indice + 1, pesos);
    return max( tomar, no_tomar );
}
```

Esta solución es la más bruta si queremos aplicar programación dinámica ¿Cuáles son los estados que tenemos? En este caso tendremos los índices y la cantidad de pesos restantes. Es decir:

```
.
.
.
int memoria[ 1005 ][ 1005 ];
bool visitado[ 1005 ][ 1005 ];
.
.
.
int max_sum( int indice, int pesos_restantes ){
    //Validación, n es el número de elementos del arreglo A
    if( indice >= n )
        return 0;
    //Como no es una respuesta válida para descartarla
        regresamos un menos infinito
    if( pesos_restantes < 0)
        return MIN_INT;

    if( visitado[ indice ][ pesos_restantes ] )
        return memoria[ indice ][ pesos_restantes ];

    //Recursivamente puedo tomar o no tomar un elemento
    int tomar = max_sum( indice + 1, pesos - A[ indice ]) +
        A[ indice ];
    int no_tomar = max_sum( indice + 1, pesos);
    //Como no se ha visitado el cálculo actual lo marcamos
        como visitado
    visitado[ indice ][ pesos_restantes ] = true;
    //Almacenamos el valor calculado en nuestra memoria
    return memoria[ indice ][ pesos_restantes ] = max( tomar
        , no_tomar );
}
```

Con esta solución tenemos una complejidad de $O(n^2)$ mientras que la solución bruta tiene una complejidad de $O(2^n)$

3.2 Mochila clásica

Queremos meter N piedras a una mochila cuya capacidad es de M y cada piedra tiene un valor V, queremos meter la mayor cantidad de piedras respetando que el peso límite soportado por la mochila no se exceda y se maximice el valor de las piedras dentro de la mochila
Restricciones:

- N piedras $N \leq 1000$
- M capacidad de la mochila $M \leq 1000$
- V_i valor de la piedra $V_i \leq 10^9$
- W_i peso de la piedra $W_i \leq 1000$

Ejemplo:
N = 3, M = 50;
V = [60, 100, 120]
W = [10, 20, 30]
Para este caso tenemos como solución tomar la segunda y tercer piedra teniendo un valor de 220 y un peso de 50, el cual puede soportar perfectamente nuestra mochila.

Podemos encontrar una solución siguiendo más o menos lo mismo que el ejercico anterior, vamos a tratar de maximizar el valor tomando o no tomando el i-ésimo peso

```
int sum_max( int indice, int capacidad ){
    if( indice >= n )
        return 0;
    if( capacidad < 0 )
        return MIN_INT;
    //Si lo tomo cuál es el valor máximo que podemos tener
        con la respuesta actual?
    int tomar = sum_max( indice + 1, capacidad - W[ indice ]
         ) + V[ indice ];
    int no_tomar = sum_max( indice + 1, capacidad );
    return max(tomar, no_tomar);
}
```

Si aplicamos programación dinámica

```
.
.
.
int memoria[ 1005 ][ 1005 ];
.
.
.
int sum_max( int indice, int capacidad ){
    if( indice >= n )
        return 0;
    if( capacidad < 0 )
```

```
            return MIN_INT;
        if( memoria[ indice ][ capacidad ] != MIN_INT )
            return memoria[ indice ][ capacidad ];
        //Si lo tomo cuál es el valor máximo que podemos tener
            con la respuesta actual?
        int tomar = sum_max( indice + 1, capacidad - W[ indice ]
             ) + V[ indice ];
        int no_tomar = sum_max( indice + 1, capacidad );
        return memoria[ indice ][ capacidad ] = max(tomar,
            no_tomar);
    }
```

Ahora bien debemos responder ¿dónde está la respuesta a nuestro problema?

```
    .
    .
    .
    int main(){
        .
        .
        .
        //Si indexamos desde cero aquí estará la solución
        cout << sum_max( 0, M );
        .
        .
        .
        return 0;
    }
```

3.3 Problema

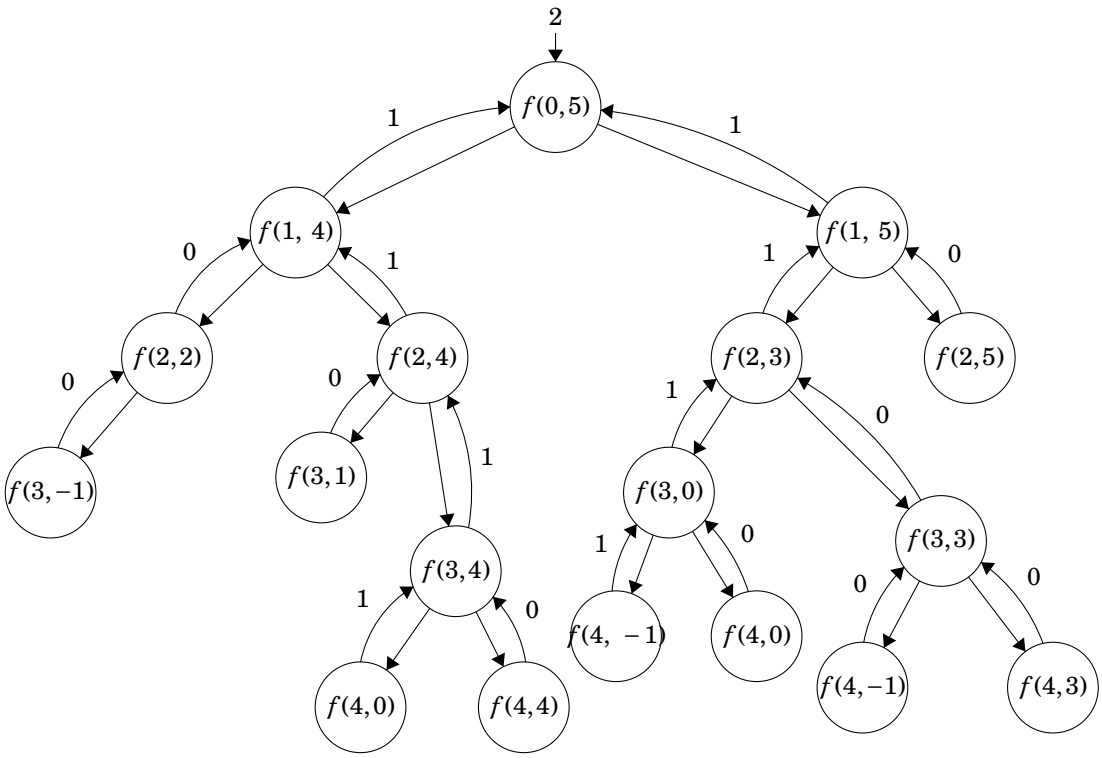
Imagina que compraste algo en algúna tienda, supermercado, etc, tienes que pagar con N monedas, pero quieres saber ¿De cuantas formas utilizando cualquiera de esas N monedas puedo dar el cambio (M)?
Restricciones:

- N monedas $N \leq 1000$
- M cambio $N \leq 1000$
- C_i Monedas $1leqC_i \leq 1000$

Ejemplo: N = 4, M = 5
C = [1, 2, 3, 4] //Indexado desde cero

- (0, 3) = 1 + 4 = 5
- (1, 2) = 2 + 3 = 5

Es decir nuevamente debemos de considerar el problema como tomar o no tomar



Como solución bruta ¿Qué podemos programar?

```
.
.
.
int cuenta( int indice, int cambio ){
    //Verificamos que no nos salgamos del rango
    if( indice == n ){
        //Si llegamos al límite verificamos si lo que
            tenemos es una solución válida
        if( cambio == 0 )
            //Como encontramos una solución la contamos
            return 1;
        //Como no hay solución retornamos cero
        return 0;
    }
    //Si el cambio es negativo no es una combinación válida
    if( cambio < 0 )
        return 0;
    //Intentamos tomando el elemento actual
    int tomar = cuenta( indice + 1, cambio - C[ indice ] );
    int no_tomar = cuenta( indice + 1, cambio );
    return tomar + no_tomar;
}
```

Si añadimos programación dinámica tenemos lo siguiente

```
.
.
.
int memoria[ 1005 ][ 1005 ];
bool visitado[ 1005 ][ 1005 ];
.
.
.
int cuenta( int indice, int cambio ){
    //Verificamos que no nos salgamos del rango
    if( indice == n ){
        //Si llegamos al límite verificamos si lo que
            tenemos es una solución válida
        if( cambio == 0 )
            //Como encontramos una solución la contamos
            return 1;
        //Como no hay solución retornamos cero
        return 0;
    }
    //Si el cambio es negativo no es una combinación válida
    if( cambio < 0 )
        return 0;

    if( visitado[ indice ][ cambio ] )
        return memoria[ indice ][ cambio ];
    //Intentamos tomando el elemento actual
    int tomar = cuenta( indice + 1, cambio - C[ indice ] );
    int no_tomar = cuenta( indice + 1, cambio );
    visitado[ indice ][ cambio ] = true;
    return memoria[ indice ][ cambio ] = (tomar + no_tomar);
}
```

3.4 Boredrom

Vamos a tomar el a_k elemento, vamos remover de nuestro arrego el a_k+1 y a_k-1
Restricciones: 1 1 2 2 2 2 2 3 3
En este caso elegimos el 2 y removemos los unos y los tres nos queda
2 2 2 2 2
Como no tenemos más elementos para elegir nuestra respuesta es la suma de los elementos restantes es decir 5
¿Cómo podemos programar eso?

```
#include <bits/stdc++.h>

using namespace std;
typedef long long int lli;
lli bucket[ 100005 ];
lli DP[ 100005 ];
bool visited[ 100005 ];
lli n;
int MAX = -1;

lli max_sum( int index ){
```

```
        if( index >= MAX + 1)
            return 0;

        if( visited[ index ] )
            return DP[ index ];
        visited[ index ] = true;
        //Tomamos el elemento actual
        lli take = max_sum( index + 2 ) + ( bucket[ index ] *
            index );
        //Nos saltamos el elemento actual
        lli not_take = max_sum( index + 1 );

        return DP[ index ] = max( take, not_take );
    }

    int main(){
        cin >> n;
        for( int i = 0, v; i < n; i++){
            cin >> v;
            MAX = max( MAX, v );
            //Generamos nuestra cubeta
            bucket[ v ]++;
        }

        cout << max_sum( 1 ) << endl;
        return 0;
    }
```

3.5 LIS - Longest Increasing Subsequence

Nos sirve para encontrar la subsecuencia más larga que sea creciente. Por ejemplo:

[1, 5, 3, 7, 8, 4]

Tenemos algunas posibilidades

5, 7, 4 consideremos que el conjunto vacío también es válido. Pero esta subsecuencia no es creciente.

Pero si las siguientes 1, 5, 7, 8 y 1, 3, 7, 8. dónde la resuesta es 4.

¿Cómo podemos solucionar esto? Podemos calcular todos los pusibles subconjuntos haciendo algo similar al problema de la mochila, es decir, tomar o no

tomar. También requerimos conocer el número anterior ya que este nos ayudará a decidir si el número actual podemos tomarlo o no. **Restricciones:**

- $0 < n \leq 1000$
- $0 < A_i \leq 1000$

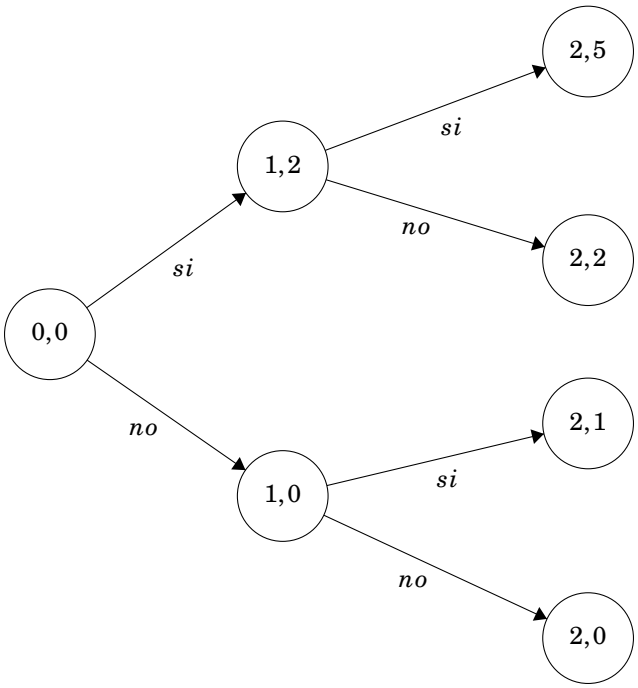
```
int LIS( int index, int last ){
    //Caso base
    if( indice == n )
        return 0;

    //Vamos a tomar como válido el valor si y solo si si el
        i-ésimo elemento es mayor que el i-ésimo - 1
    if( A[ index ] > last ){
        int take = LIS( index + 1, A[ index ] ) + 1;
        int not_take = LIS( index + 1, last );
        return max( take, not_take );
    }
    return LIS( index + 1, last );
}
.
.
.
int main(){
    .
    .
    .
        cout << LIS( 0, INT_MIN ) << endl;
    .
    .
    .
}
```

Esta solución obtiene todos los subconjuntos pero la complejidad es de $O(2^n)$. Si queremos optimizar usando memoria debemos ver los estados de nuestra DP, en este caso tenemos dos, el primero el índice y el segundo el valor más grande que puede valer last, es decir, 1000 y 1000 respectivamente. Por tanto obtenemos cono resultado

```
.
.
.
int DP[ 1005 ][ 1005 ] = { -1 };
.
.
.
int LIS( int index, int last ){
    //Caso base
    if( indice == n )
        return 0;
    if( DP[ index ][ last ] != -1 )
        return DP[ index ][ last ];
    //Vamos a tomar como válido el valor si y solo si si el
        i-ésimo elemento es mayor que el i-ésimo - 1
    if( A[ index ] > last ){
```

```
            int take = LIS( index + 1, A[ index ] ) + 1;
            int not_take = LIS( index + 1, last );
            return DP[ index ][ last ] = max( take, not_take );
        }
        return DP[ index ][ last ] = LIS( index + 1, last );
    }
    .
    .
    .
    int main(){
        .
        .
        .
            cout << LIS( 0, INT_MIN ) << endl;
        .
        .
        .
    }
```

Con esta solución la complejidad que obtenemos es de $O(n * max(A_i))$
Gráficamente tenemos algo como esto.



Si queremos imprimir la secuencia que tenemos ¿Que podemos hacer? Podríamos usar un string o vector o también recorrer la memoria ¿Hay alguna otra solución?

Podemos reutilizar nuestra función LIS, para reconstruir la solución. Dónde vamos a añadir el valor que nos convino más como solución a nuestra respuesta.

Es decir podemos tener un vector de enteros que obtenga la reconstrucción.

```
.
.
.
vector< int > reaconstruct;
void reconstructLIS( int index, int last ){

    if( index >= n )
        return;

    if( A[index] > last ){
        //Como ya se han memorizado los valores de todos los
             subconjuntos válidos obtendremos en constante
            cual camino me es más conveniente tomar
        int take = LIS( index + 1, A[ index ]) + 1;
        int not_take = LIS( index + 1, last );
        //Aquí podemos decidir que camino nos conviene más
            si el de tomar o no tomar
        if( take > not_take ) {
            //Añadimos a nuestra respuesta
            reconstruct.push_back( A[ index ] );
            reconstructLIS( index + 1, A[ index ] );
        } else
            reconstructLIS( index + 1, last );
    }
    reconstructLIS( index + 1, last );
}
.
.
.
```

¿Qué complejidad tiene reconstruir la respuesta? $O(n)$ ya que solamente vamos ha realizar una sola llamada recursiva

3.6 Longest Common Subsequence

Dadas dos cadenas, hallar la subsecuencia común más larga.

Restricciones:

Ejemplo:

- S = AGPGTAB
- T = GXTXAYB

A	G	P	G	T	A	B
G	X	T	X	A	Y	B

¿Podemos atacar este problema utilizando la misma lógica que en knapsack?

Si intentamos hacer esto tendremos varios problemas. Podemos remover un caracter de alguna de las dos cadenas en algún punto en el que la cadena no coincida. Si ámbos caracteres coinciden ambos los "quitamos" y aumentamos en 1 la respuesta actual.

```
int LCS( string S, string T ){

    if( S.empty() || T.empty() )
        return 0;

    string S2 = S;
    string T2 = T;
    //erase elimina el caracter (es un iterador) en nuestro
        caso el primero de la cadena S2
    S2.erase( S2.begin() );
    T2.erase( T2.begin() );

    //Si ámbos coinciden aumentamos en uno la respuesta
    if( S[ 0 ] == T[ 0 ] )
        //Como el caracter es el mismo en ámbos la respuesta
            aumenta en uno
        return LCA( S2, T2 ) + 1;

    int delete_first = LCS( S2, T );
    int delete_second = LCS( S, T2 );

    return max( delete_first, delete_second );
}
```

¿Como podemos asegurar que nos conviene realizar lo anterior? o bien ¿Hay algún caso que no se ha cubierto? y ¿Cuál es la complejidad de este algoritmo?

Supongamos que tenemos una cadena donde un caracter de la respuesta aparece más de una vez, si consideramos el segundo y no el primero es posible que nuestra respuesta o no mejore o empeore.

La complejidad de realizar esto es $O(2^n * n)$ (El n esta ahí por la función erase). ¿Cómo podemos mejorar esta solución?

Nótese que solo estamos utilizando lo caracteres del final, es decir solo utilizamos el sufijo (caracteres del final), así pues en lugar de utilizar una subcadena podemos utilizar un apuntador.

```
int LCS( int index_s, int index_t ){
    //Si ya no hay caracteres que analizar terminamos las
        recursiones
    if( index_s == S.size() || index_t == S.size() )
        return 0;

    if( S[index_s] == T[index_t] )
        return LCA( index_s + 1, index_t + 1) + 1;
```

```
        int move_S = LCS( index_s + 1, index_t );
        int move_T = LCS( index_s, index_t + 1 );

        return max( move_S, move_T );
    }
```

¿Ahora que complejidad obtenemos? $O(2^n)$ ya que hemos eliminado el uso de la función erase. Finalmente ¿Como implementamos programación dinámica? En este caso tenemos dos estados index_s e index_t.

```
    .
    .
    .
    int DP[ 1005 ][ 1005 ];
    .
    .
    .
    int LCS( int index_s, int index_t ){
        //Si ya no hay caracteres que analizar terminamos las
            recursiones
        if( index_s == S.size() || index_t == S.size() )
            return 0;
        //Nuestra ''bandera'' es el valor de -1 ya que nunca nos
             devolverá ese valor la función
        if( DP[ index_s ][ index_t ] != -1 )
            return DP[ index_s ][ index_t ]

        if( S[index_s] == T[index_t] )
            return DP[ index_s ][ index_t ] = LCA( index_s + 1,
                index_t + 1) + 1;

        int move_S = LCS( index_s + 1, index_t );
        int move_T = LCS( index_s, index_t + 1 );

        return DP[ index_s ][ index_t ] = max( move_S, move_T );
    }
    .
    .
    .
    int main(){
        .
        .
        .
        cout << LCS( 0, 0 ) << endl;
        .
        .
        .
        return 0;
    }
```

Y finalmente hemos optimizado sustancialmente ese problema, hemos pasado de tener una complejidad de $O(2^n * n)$ a $O(2^n)$ a finalmente $O(|S| * |T|)$ o bien en $O(n^2)$