Algoritmos para programación competitiva

Wilmer Emiro Castrillón Calderón

17 de septiembre de 2022

Índice general

Prefacio

Este documento esta escrito con el objetivo de recopilar los algoritmos mas utilizados en programación competitiva en un documento en español, exponer como funcionan, como utilizarlos y su implementación en C++, con explicaciones claras y precisas.

En este documento se asume que el lector ya tiene un conocimiento general del lenguaje C++, la librería estándar de C++, complejidades algorítmicas (notación O grande) y una experiencia por lo menos básica en competencias de tipo ICPC.

Este documento contiene una lista de temas que personalmente trabaje a lo largo de varios años como competidor en maratones de ICPC y espero que sea de ayuda para los nuevos competidores.

Capítulo 1

Estructuras de datos

1.1. Tablas aditivas

Son estructuras de datos utilizadas para realizar operaciones acumuladas sobre un conjunto de datos estáticos en un rango específico, es decir, ejecutar una misma operación(como por ejemplo la suma) sobre un intervalo de datos, se asume que los datos en la estructura no van a cambiar. Estas estructuras también son conocidas como *Summed-area table* para el procesamiento de imágenes, a pesar de su nombre no necesariamente son exclusivas para operaciones de suma, pues la idea general es aplicable a otras operaciones.

Durante el cálculo de una misma operación sobre diferentes rangos se presenta superposición de problemas, las tablas aditivas son utilizadas para reducir la complejidad computacional aprovechando estas superposiciones utilizando programación dinámica.

Ejemplo inicial.

Dado un vector V = {5,2,8,2,4,3,1} encontrar para múltiples consultas la suma de todos los elementos en un rango [i,j], indexando desde 1, por ejemplo con el rango [1, 3] la suma es [5+2+8] = 15.

La solución trivial es hacer un ciclo recorriendo el vector entre el intervalo [i,j], en el peor de los casos se debe recorrer todo el vector, esto tiene una complejidad $O(n)$ puede que para una consulta sea aceptable, pero en casos grandes como por ejemplo un vector de tamaño 10^5 y una cantidad igualmente grande de consultas el tiempo de ejecución se hace muy alto, por lo tanto se hace necesario encontrar una mejor solución.

La operación suma tiene propiedades que nos pueden ayudar a resolver este problema de una forma mas eficiente:

1. La suma es asociativa es decir, se cumple: $a + (b + c) = (a + b) + c$, esto indica que sin importar la agrupación que se realice el resultado sera igual (no confundir con propiedad conmutativa).

2. La suma posee elemento neutro, es decir existe un β tal que $a + \beta = a$, en la suma $\beta = 0$.

3. La suma tiene operación inversa, es decir existe una operación que puede revertir la suma, la cual es la resta: si $a + b = c$ entonces $c - a = b$.

Considerando las anteriores propiedades el problema se puede trabajar desde otro enfoque, primero se puede definir $suma(x) = \sum_{k=1}^{x} V_k$, ahora tomando como ejemplo una consulta en el rango [3,5] del vector V, se puede utilizar $suma(2) = V_1 + V_2$ y $suma(5) = V_1 + V_2 + V_3 + V_4 + V_5$ para encontrar $V_3 + V_4 + V_5$, utilizando la propiedad asociativa se obtiene: $(V_1 + V_2) + (V_3 + V_4 + V_5) = (V_1 + V_2 + V_3 + V_4 + V_5)$ y usando la propiedad inversa se llega a: $(V_3 + V_4 + V_5) = (V_1 + V_2 + V_3 + V_4 + V_5) - (V_1 + V_2)$ por lo tanto $(V_3 + V_4 + V_5) = suma(5) - suma(2)$. De esta manera el problema se puede generalizar como $\sum_{k=i}^{j} V_k = suma(j) - suma(i-1)$ cuando $i \neq 1$ y $suma(j)$ cuando $i = 1$.

Ahora pre-calculando $suma(x)$ se puede dar una solución inmediata a cada consulta, esto se puede resolver utilizando un enfoque básico de programación dinámica. Para encontrar $suma(x)$ se puede reescribir como: $suma(x) = V_x + suma(x-1)$ con caso base $suma(1) = V_1$ y por definición la operación acumulada sobre un conjunto vacío es igual al elemento neutro (esto significa que el indice cero tendrá el valor del elemento neutro), a partir de esto se puede obtener la siguiente solución en C++ con consultas indexando desde 1.

```
1 int V[] = {5,2,8,2,4,3,1}, memo[8];
2
3 void precalcular(){
4     memo[0] = 0;
5     for(int i = 0; i < 7; i++)
6         memo[i+1] = V[i] + memo[i];
7 }
8
9 int consulta(int i, int j){ return memo[j] - memo[i-1]; }
```

De manera general las tablas aditivas son aplicables a cualquier operación que posea las tres propiedades descritas anteriormente: ser asociativa, tener elemento neutro y operación inversa, por ejemplo la suma, multiplicación o el operador de bits XOR.

Tablas aditivas en 2D

Las operaciones acumuladas no solo se pueden usar sobre una dimension, sino también sobre n-dimensiones, en estos casos se debe trabajar con el principio de inclusión-exclusión pues se debe considerar mejor las operaciones entre intervalos, si no tiene en cuenta este principio las soluciones contendrían elementos duplicados o faltantes, lo que produciría soluciones incorrectas.

Ejemplo: dada la matriz M encontrar para múltiples consultas la suma de todos los elemento en una submatriz $Q_{(i1,j1),(i2,j2)}$:

$$M = \begin{array}{|c|c|c|c|c|} \hline 1 & 9 & 6 & 3 & 7 \\ \hline 7 & 5 & 3 & 0 & 5 \\ \hline 0 & 7 & 6 & 5 & 3 \\ \hline 7 & 8 & 9 & 5 & 0 \\ \hline 9 & 5 & 3 & 7 & 8 \\ \hline \end{array}$$

En el intervalo $Q_{(2,2),(3,4)}$ el resultado es $5 + 3 + 0 + 7 + 6 + 5 = 26$.

En el caso de 1D se definió $suma(x) = \sum_{k=1}^{x} V_k$, ahora esta debe tener dos dimensiones, es decir, $suma(i, j)$ debe contener la suma de los elementos en la submatriz $Q_{(1,1),(i,j)}$, entonces ahora se definirá: $suma(i, j) = \sum_{k=1}^{i} \sum_{w=1}^{j} M_{k,w}$, mas sin embargo pre-calcular $suma(i, j)$ de forma eficiente requiere de usar el principio de inclusión-exclusión, de manera trivial se puede calcular la primera fila como $suma(1, j) = \sum_{w=1}^{j} M_{1,w}$ y la primera columna como $suma(i, 1) = \sum_{k=1}^{i} M_{k,1}$, en base a esto se puede calcular el resto de la matriz pero se debe tener algo de cuidado, por ejemplo si se toma $suma(2, 2) = suma(1, 2) + suma(2, 1) + M_{2,2}$ se obtendría un resultado incorrecto pues se estaría realizando la siguiente operación: $(M_{1,1} + M_{1,2}) + (M_{1,1}, M_{2,1}) + M_{2,2}$, se puede observar que el elemento $M_{1,1}$ se esta sumando dos veces, acá se aplica el principio de inclusión-exclusión: $|A| \cup |B| = |A| + |B| - |A \cap B|$ lo que significa que hace falta quitar la intersección, esta es $suma(1, 1)$ entonce se puede generalizar como: $suma(i, j) = M_{i,j} + suma(i - 1, j) + suma(i, j - 1) - suma(i - 1, j - 1)$ cuando $i, j \neq 1$.

Una vez construido el pre-calculo es necesario realizar las consultas, se utilizara como ejemplo la consulta en el rango $Q_{(2,2),(3,4)}$, de igual manera se debe tener cuidado de no sumar un mismo intervalo mas de una vez, entonces para encontrar la suma en este intervalo se tomaría $suma(i2, j2)$ esta contendría $\sum_{k=1}^{i2} \sum_{w=1}^{j2} M_{k,w}$, esta tiene elemento adicionales como lo muestra la figura 1.1, al restarle $suma(i1 - 1, j2)$ se quitan algunos elementos (figura 1.2), al restarle $suma(i2, j1 - 1)$ pasaríamos a restar dos veces el intervalo $M_{(1,1),(i1-1,j1-1)}$ (figura 1.3), por lo tanto es necesario reponer lo faltante agregando $suma(i1 - 1, j1 - 1)$ (figura 1.4), de esta manera se puede generalizar la formula: $\sum_{k=i1}^{i2} \sum_{w=j1}^{j2} M_{k,w} = suma(i2, j2) - suma(i1 - 1, j2) - suma(i2, j1 - 1) + suma(i1 - 1, j1 - 1)$.

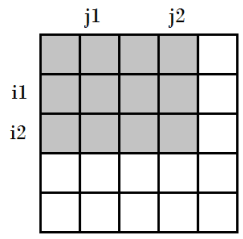


Figura 1.1

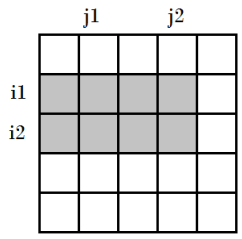


Figura 1.2

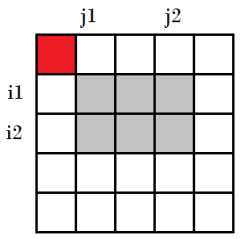


Figura 1.3

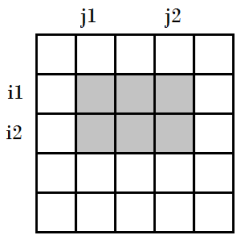


Figura 1.4

Para la solución en C++ la fila y columna cero de la tabla donde se guardara el pre-calculo se deberá llenar con el elemento neutro, el cero, luego se debe llenar el resto de la tabla siguiendo las formulas antes establecidas, este ejemplo es para consultas indexando desde 1.

```
1 void precalcular(){
2     memset(memo, 0, sizeof(memo));
3     for (int i = 1; i <= fila; i++)
4     for (int j = 1; j <= col; j++)
5         memo[i][j] = memo[i][j - 1] + memo[i - 1][j] +
6                         M[i - 1][j - 1] - memo[i - 1][j - 1];
7 }
8
```

```
int consulta(int i1, int j1, int i2, int j2){
    return memo[i2][j2] - memo[i1-1][j2] -  memo[i2][j1-1] + memo[i1-1][j1
        -1];
}
```

Tablas aditivas en 3D

Las tablas aditivas se pueden generalizar para trabajar en n-dimensiones, mas sin embargo la dificultad de hacer los pre-cálculos y las consultas aumenta bastante, pues crece considerablemente la cantidad de operaciones a realizar. En el caso 3D igualmente se debe tener cuidado con el principio de inclusión-exclusión, el pre-calculo se realizaría de la siguiente manera: sea $suma(i,j,k) = \sum_{x=1}^{i}\sum_{y=1}^{j}\sum_{z=1}^{k} V_{x,y,z}$, entonces $suma(i,j,k) = V_{i,j,k} + suma(i,j,k-1) + suma(i-1,j,k) + suma(i,j-1,k) - suma(i-1,j-1,k) - suma(i-1,j,k-1) - suma(i,j-1,k-1) + suma(i-1,j-1,k-1)$, y para las consultas: $\sum_{x=i1}^{i2}\sum_{y=j1}^{j2}\sum_{z=k1}^{k2} V_{x,y,z} = suma(i2,j2,k2) - suma(i2,j2,k1-1) - suma(i1-1,j2,k2) + suma(i1-1,j2,k1-1) - suma(i2,j1-1,k2) + suma(i2,j1-1,k1-1) + suma(i1-1,j1-1,k2) - suma(i1-1,j1-1,k1-1)$.

Conclusiones

Esta estructura de datos facilita encontrar el valor acumulado de una operación sobre un rango de valores estáticos, puede ser aplicada para operaciones que cumplan con las tres propiedades descritas anteriormente, la complejidad computacional de hacer el pre-calculo es lineal a la cantidad de elementos en la estructura, y las consultas se realizan en complejidad constante al realizar solamente operaciones directas sobre elementos en la tabla del pre-calculo. Si se requiere actualizar los valores de la tabla esto haría necesario actualizar el pre-calculo, esto representa que la operación de actualizar es lineal a la cantidad de elementos, pero esa complejidad no suele ser favorable, por lo tanto se recomienda utilizar solo en arreglos estáticos, y recurrir a otras estructuras como las Sparse table en el caso de arreglos dinámicos.

1.2. Disjoint set

Es una estructura de datos que permite agrupar elementos en conjuntos y consultar si dos elementos pertenecen a un mismo conjunto, contiene una estructura en forma de árbol pero con una implementacion poco compleja. La estructura consta de dos operaciones principales, $union(u,v)$ para unir los conjuntos que incluyen a los nodos u y v, y la operación $find(v)$ para indicar a cual conjunto pertenece un nodo v.

Cada nodo tendrá la información de su nodo padre, y cada conjunto tiene un nodo raíz. Entonces se tiene un arreglo lineal p donde $p[v]$ tiene el padre del nodo v, para los nodos raíz se tendrá que $p[v] = v$, como se muestra en la imagen siguiente, inicialmente cada nodo es un conjunto diferente entonces todo el arreglo debe iniciar con $p[v] = v$.

Operación *Find*

La operación consiste en devolver el nodo raíz del conjunto al cual pertenece un nodo v, como cada nodo guarda el indice de su nodo padre, entonces solo se debe subir sobre el árbol hasta llegar al nodo raíz el cual se puede detectar con la condición $p[x] = x$.

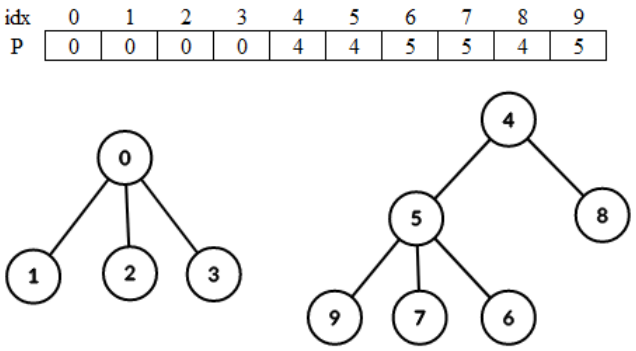


Figura 1.5: Disjoint set

Operación *Union*

Consiste en unir dos elementos u y v en un mismo conjunto, básicamente consiste en hacer que el nodo raíz del conjunto de u sea padre del nodo raíz del conjunto de v, de esta forma ambos nodos quedan dentro de un mismo árbol, si u y v ya se encuentran dentro en un mismo conjunto entonces no se hace nada.

```
struct union_find{
    int padre[100];

    void build(int n){
        for(int i = 0; i < n; i++) padre[i] = i;
    }

    int buscar(int v){//find
        if(v == padre[v]) return v;
        else return buscar(padre[v]);
    }

    void unir(int u, int v){//union
        u = buscar(u);  v = buscar(v);
        if(u != v) padre[u] = v;
    }

    bool MismoGrupo(int u, int v){
        return buscar(u) == buscar(v);
    }
};
```

Unión por rangos

Los disjoint set son estructuras flexibles que nos permiten realizar diferentes variantes para distintas tareas, una de ellas es la unión por rangos, la cual consiste en tener guardado en un arreglo la máxima profundidad que contiene cada nodo y nos permite durante la operación *Union* unir el árbol de menor rango a otro de mayor de rango. En la siguiente figura se observa un ejemplo de unir dos conjuntos con distinto rango, los conjuntos del nodo 9 y 8, sus raíces son 5 y 4 respectivamente, entonces como el nodo 4 tiene un rango mayor, se volverá el padre del nodo 5.

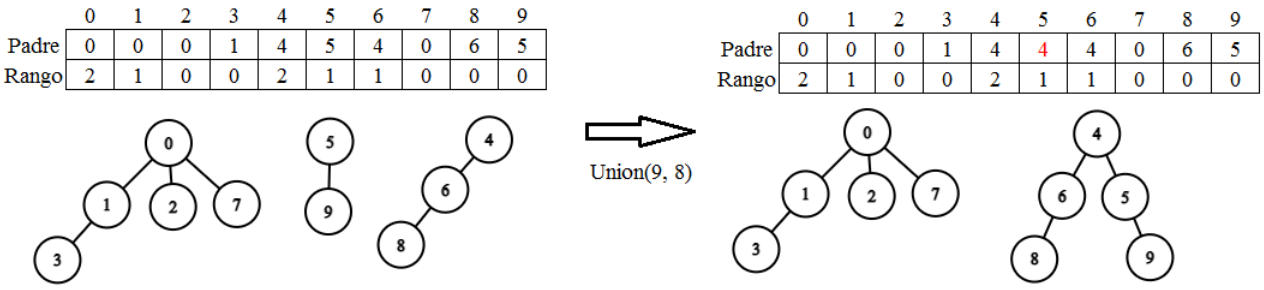


Figura 1.6: Unión por rangos

```
int padre[100], rango[100];

void build(int n){
    for (int i = 0; i < n; i++){
        padre[i] = i;  rango[i] = 0;
    }
}

void unir(int u, int v){
    u = buscar(u);  v = buscar(v);
    if(u == v) return;
    if(rango[u] > rango[v]){
        padre[v] = u;
        return;
    }
    padre[u] = v;
    if(rango[v] == rango[u]) rango[v]++;
}
```

Compresión de caminos

Cuando se utiliza gran cantidad de nodos el disjoint set tradicional es ineficiente, debido a la posibilidad de formar arboles muy profundos, llevando a la operación $find(v)$ a tener complejidad $O(n)$ para cada búsqueda, existe una optimización la cual permite reducir la profundidad de los arboles, de tal forma que la máxima longitud de rangos sera 1, de esta forma se evita tener arboles profundos. La implementacion es sencilla y consiste que en cada búsqueda actualizar los nodos visitados como hijos directos del nodo raíz. La siguiente figura muestra como se vería el anterior ejemplo con compresión de caminos.

Conclusiones

Esta estructura permite asociar elementos en conjuntos de forma eficiente, tiene múltiples aplicaciones como por ejemplo almacenar componentes conexas de grafos, es utilizada por otros algoritmos como el algoritmo de kruskal para la búsqueda del árbol de expansión mínima en grafos, también se puede almacenar información adicional sobre los conjuntos como su cantidad de nodos o almacenar para cada nodo la lista de sus descendientes, entre otras aplicaciones.

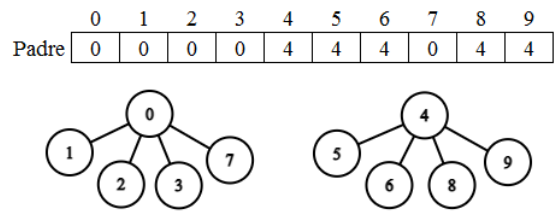


Figura 1.7

```
1  int buscar(int v){
2      if(v == padre[v]) return v;
3      else return padre[v] = buscar(
           padre[v]);
4  }
5
6  void unir(int u, int v){ padre[
       buscar(u)] = buscar(v); }
```

1.3. Sparse table

Es una estructura de datos para realizar operaciones sobre rangos en un conjunto datos, esta estructura utiliza un precálculo sobre los valores iniciales para poder realizar consultas en complejidad logarítmica, permite realizar operaciones que cumplen con la propiedad asociativa como la suma o la función mínimo(el valor mínimo entre dos valores), una de sus ventajas es su implementación corta, sin embargo esta estructura no permite actualizaciones, por lo que debe ser usada con datos estáticos.

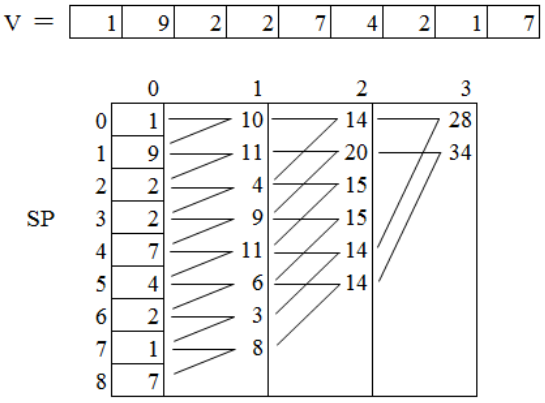
Esta estructura aprovecha la propiedad de que todo número entero se puede representar de forma única como sumas de potencias de dos, consiste en operar rangos del arreglo que tengan longitudes de potencia de dos y guardarlos en una tabla de precálculo, se utiliza una matriz de n filas y $log_2(n)$ columnas, la primera columna contendrá el arreglo inicial, las demás columnas tendrán la operación acumulada en el rango $[i, i + 2^j)$ del arreglo inicial, para cada consulta se operan los valores de sólo algunos rangos, evitando que se operen todos los valores en el intervalo. Para la explicación de construcción y consulta se usará un vector V para los valores iniciales, una matriz SP para guardar el sparse table y la operación suma.

La construcción es corta y consiste de inicialmente llenar la primera columna con los valores iniciales $SP_{i,0} = V_i$, y para las demás columnas se aplica la siguiente fórmula $SP_{i,j} = SP_{i,j-1} + SP_{i+2^{j-1},j-1}$, con $1 \leq j \leq log_2(n)$ y $1 \leq i + 2^j - 1 < n$, al final cada posición de la matriz tendrá los siguientes valores $SP_{i,j} = \sum_{x=i}^{i+2^j-1} V_x$ con $0 \leq i < n$ y $0 \leq i + 2^j - 1 < n$. La construcción tiene una complejidad $O(n * log_2(n))$.

Para las consultas se debe tener en cuenta que $SP_{i,j}$ guarda la operación acumulada en el intervalo $[i, i + 2^j - 1]$, Entonces para obtener la operación acumulada en un rango $[L, R]$ se debe tomar la fila L y buscar el máximo j tal que $L + 2^j - 1 \leq R$ es decir, tomar el intervalo mas grande que no se salga del rango $[L, R]$, con esto ya se tiene el acumulado de una parte del intervalo, de tal forma que se puede actualizar el rango a $[L + 2^j, R]$ y repetir el proceso hasta encontrar el valor de todo el intervalo $[L, R]$; la complejidad es $O(log_2(n))$. Por ejemplo para calcular suma en el rango $[1, 6]$ se puede obtener con solo sumar dos valores de la tabla, el primero $SP_{1,2}$ el cual contiene la suma en el rango [1,4] y el segundo $SP_{5,1}$ el cual contiene la suma en el rango $[5, 6]$, entonces resultado es 26, como se muestra en la figura 1.9.

Optimización para operaciones idempotentes

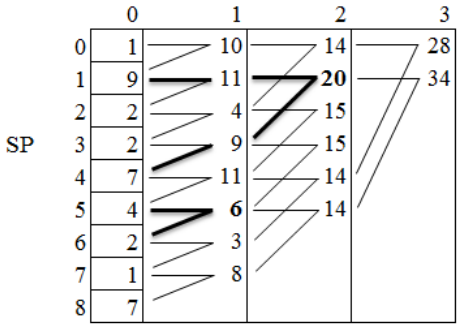
Las operaciones idempotentes son aquellas que se pueden aplicar múltiples veces y el resultado será el mismo que al aplicarse la primera vez, es decir $f(f(x)) = f(x)$, por




```
void construir(){
    for(int i = 0; i < n; ++i)
        SP[i][0] = V[i];

    int x = log2(n);
    for(int j = 1; j <= x; ++j)
        for(int i = 0; i+(1<<j)-1 <
             n; ++i)
            SP[i][j] = min(SP[i][j
                -1], SP[i+(1<<(j-1))
                ][j-1]);
}
```

Figura 1.8




```
int consulta(int L, int R){
    int res = 0;
    while(L <= R){
        int j = log2(R-L+1);
        res += SP[L][j];
        L += 1<<j;
    }
    return res;
}
```

Figura 1.9

ejemplo las funciones mínimo y máximo, para este tipo de casos se puede trabajar con intervalos superpuestos y la respuesta seguirá siendo igual, por ejemplo para encontrar el RMQ en un intervalo $[L, R]$ con $L < R$ se cumple que $RMQ(L, R) = min(RMQ(L, R-1), RMQ(L+1, R))$, entonces la consulta sobre el sparse table se puede simplificar para este tipo de funciones, de tal forma que solo dos intervalos son suficientes para encontrar la respuesta acumulada en cualquier rango, estos intervalos serán $[L, L+2^j]$ y $[R-2^j+1, R]$ con $j = floor(log_2(R-L+1))$ de esta forma la complejidad es $O(1)$.

```
int consulta_idempotentes(int L, int R){
    int j = log2(R-L+1);
    return min(SP[L][j], SP[R-(1<<j)+1][j]);
}
```

1.4. Fewnwick tree

También conocido como árbol binario indexado, es una estructura de datos no lineal la cual permite calcular operaciones acumuladas sobre rangos permitiendo actualizar el conjunto de datos, se basa en dos operaciones básicas, calcular la operación acumulada sobre los primeros i elementos, y actualizar un elemento del conjunto de datos, ambas se realizan en $O(log_2(n))$, el árbol requiere $O(n)$ de espacio, según la implementación utilizada se puede empezar a indexar desde 0 o 1, sin embargo ambas son equivalentes

en cuanto a complejidad algorítmica.

La estructura consiste en almacenar en un arreglo lineal ft la operación acumulada sobre un intervalo del arreglo de datos A, de tal forma que al operar uno o múltiples elementos en ft se pueda obtener la operación acumulada del intervalo $[1, i]$, indexando desde 1, los intervalos del precalculo se toman basados en la representación binaria del indice, en la figura 1.10 se muestra un ejemplo utilizando la operación suma, para entender mejor se recomienda escribir los indices i en binario y comparar con los rangos sumados en ft_i.

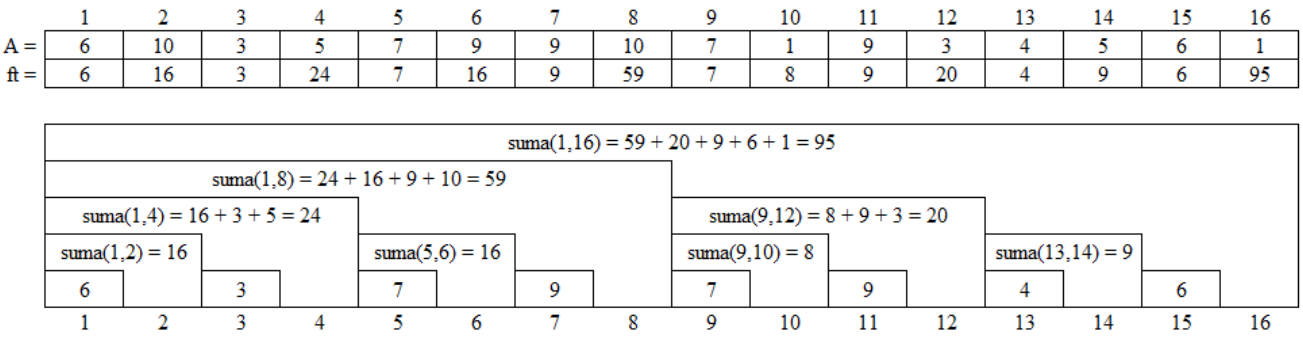


Figura 1.10: Fenwick tree

La construcción del árbol consiste en empezar con ft vacío, es decir $\forall i \in [0, n] ft_i = 0$, posteriormente realizar n actualizaciones, una por cada elemento en A, entonces la construcción inicial tiene complejidad $O(n * log_2(n))$, la actualización consiste en agregar la diferencia entre el valor nuevo y el anterior, cada actualización consiste en tomar el indice i (indexando desde cero) e ir recorriendo sus bits comenzando desde el lsb (lowest significat bit), si el k-ésimo bit es cero significa que ft_{i+2^k} también debe ser actualizado (puede ser algo confuso al comienzo, se recomienda hacer el proceso a mano para que se pueda entender mejor), una forma mas rápida de recorrer los bits que necesitamos es utilizando operaciones de bits, si en lugar de usar i utilizamos a $\sim i$, los bits que requerimos serán los que están en 1, comenzando desde el lsb, entonces se puede tomar el ultimo dígito en 1 usando la operación $i\& - i$ y posteriormente apagarlo para tomar el siguiente encendido, de esta forma se puede ir directamente a los bits que se necesitan, y por ultimo en lugar de actualizar ft_i se puede actualizar ft_{i+1} para que los indices empiecen en 1.

La operación de consulta, consiste en obtener la sumatoria hasta un indice i, esta operación tiene una idea similar a la de actualización, se empieza con el valor ft_i, posteriormente se actualiza i apagando su ultimo bit encendido y al resultado se le suma el nuevo ft_i, este proceso se repite mientras i sea mayor a 0.

```
1 int ft[MAX], n = 16;
2
3 struct fenwick_tree {//indexando desde 1
4
5   fenwick_tree() {
6     for(int i = 0; i < n; ++i) ft[i] = 0;
7   }
```

```
8
9   void update(int idx, int valor) {
10    int x = ~idx, lsb = 0;
11
12    while(lsb < n) {
13      ft[idx + lsb + 1] += valor;
14      lsb ^= x & -x;
15      x = x ^ (x & -x);
16    }
17  }
18
19  void construir(vector<int> &v) {
20    for(int i = 0; i < v.size(); ++i) {
21      update(i, v[i]);
22    }
23  }
24
25  int query(int i) {
26    int res = 0, lsb = 0;
27
28    while(i > 0) {
29      res += ft[i];
30      lsb = i & -i;
31      i = i ^ lsb;
32    }
33    return res;
34  }
35
36  int query(int i, int j) {
37    return query(j) - query(i - 1);
38  }
39 };
```

Capítulo 2

Grafos

2.1. Teoría de grafos

Los grafos son una estructura de datos donde se pueden relacionar distintos objetos entre si, los grafos se pueden definir como un conjunto de nodos(objetos) unidos por aristas(relaciones), estos son estudiados por la matemática y las ciencias de la computación, esta rama se conoce como teoría de grafos, ademas tienen muchas aplicaciones, por ejemplo permiten modelar redes informáticas, sistemas de carreteras, redes sociales, etc.

Terminología general

Los grafos se pueden clasificar de distintas formas, las mas utilizadas son:

1. **Grafos ponderados y no ponderados** Si las aristas de un grafo tienen un peso, es decir si para atravesar una arista esta tiene un costo asociado, entonces el grafo se clasifica como ponderado, pero si ninguna arista tiene algún costo entonces el grafo se clasifica como no ponderado.

2. **Grafos dirigidos y no dirigidos:** Si un grafo posee por lo menos una arista dirigida entonces se clasifica como un grafo dirigido, una arista (A, B) es dirigida si esta permite el paso de A hacia B, pero no permite ir de B hacia A. Si todas las aristas son bidireccionales(no dirigidas) entonces el grafo se clasifica como no dirigido.

3. **Grafos cíclicos y acíclicos** Se considera un grafo como acíclico cuando este no tiene ciclos, es decir para cada pareja de nodos (A, B) si existe un camino para ir de A hacia B entonces no existe otro camino para ir de B hacia A. Si un grafo no es acíclico entonces este es cíclico.

4. **Grafos conexos y no conexos** Si para cada pareja de nodos (A, B) existe un camino para ir de A hacia B y también existe camino para ir de B hacia A entonces el grafo es conexo, en caso contrario es un grafo no conexo.

Representación.

Los grafos se pueden representar de múltiples maneras cada una permite realizar distintos algoritmos, cada forma de representar los grafos se puede ajustar según su tipo, un aspecto importante a considerar es que cada una arista bidireccional (A, B) se puede interpretar como dos aristas dirigidas (A, B) y (B, A). Existen principalmente de tres formas distintas de representarlos:

1. **Matriz de adyacencia:** Es una matriz M en la cual cada fila representa un nodo de inicio y cada columna un nodo destino(a cada nodo se le asignara un numero representando su posición en fila y columna), si existe una arista (A, B) entonces se marca la casilla $M_{A,B}$, en el caso de grafos ponderados se debe llenar $M_{A,B}$ con el costo de la arista (A, B), para los todos los pares de nodos que no tengan aristas entre ellos el costo es infinito. Si el grafo es no ponderado entonces en la matriz simplemente se marca si existe o no la arista (A, B). Por definición cada nodo tiene conexión con sí mismo con costo cero.

2. **Lista de adyacencia:** Consiste en guardar para cada nodo una lista con las conexiones que posee, es decir, para cada arista (A, B) se pondrá en la lista de conexiones de A el nodo B. Si el grafo es ponderado entonces se deberá guardar el nodo destino junto con su costo. Esta es la manera mas utilizada para representar grafos, pues el espacio de memoria que ocupa es menor que el utilizado por una matriz de adyacencia y ademas facilita recorrer el grafo de manera sencilla.

3. **Lista de aristas:** Consiste en una lista en la cual se guardara todas las aristas de un grafo, para cada una se guarda el nodo de inicio, nodo de destino y el costo en caso de tener. Esta es la menos utilizada de las tres, aunque facilita ordenar las aristas según su costo.

A continuación se muestra un pequeño ejemplo para cada una de las tres formas:

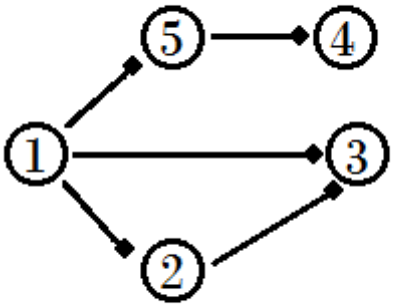


Figura 2.1

```
1  bool grafo[10][10];
2
3  void construir_grafo(){
4      memset(grafo, false, sizeof(grafo))
           ;
5      grafo[1][5] = true;
6      grafo[1][3] = true;
7      grafo[1][2] = true;
8      grafo[5][4] = true;
9      grafo[2][3] = true;
10 }
```

Matriz de adyacencia

2.2. DFS y BFS

Los grafos son estructuras no lineales, estos no tienen un nodo inicio o un orden especifico para recorrerlos, existen principalmente dos algoritmos que permiten recorrer un grafo DFS y BFS, estos no son algoritmos muy estrictos, es decir, se pueden modificar de múltiples maneras para realizar diferentes tareas, sin embargo la idea básica sigue siendo la misma, para la implementación de ambos se utiliza una lista de adyacencia.

DFS.

```cpp
vector<vector<int>> grafo(10);

void construir_grafo(){
    grafo[1].push_back(5);
    grafo[1].push_back(3);
    grafo[1].push_back(2);
    grafo[5].push_back(4);
    grafo[2].push_back(3);
}
```

Lista de adyacencia

```cpp
typedef pair<int, int> ii;
vector<ii> grafo;

void construir_grafo(){
    grafo.push_back(ii(1, 5));
    grafo.push_back(ii(1, 3));
    grafo.push_back(ii(1, 2));
    grafo.push_back(ii(5, 4));
    grafo.push_back(ii(2, 3));
}
```

Lista de aristas

El DFS (deep first search) o búsqueda en profundidad, es un algoritmo que permite recorrer un grafo de forma recursiva, de manera general consiste en tomar un nodo, marcarlo como visitado y para cada arista hacer un llamado recursivo al nodo destino, solo si ese nodo no esta visitado, y repetir el proceso hasta que no queden mas nodos por visitar. De esta forma se van marcando los nodos mientras se visitan y en caso de llegar a un nodo sin aristas o un nodo cuyos vecinos se encuentran visitados entonces se devuelve al nodo anterior.

En la siguientes figuras se observa el orden en el cual los nodos son visitados empezando por el nodo 1, entonces comienza con el 1, después avanza al 5 y luego al 4, como no hay mas destinos se devuelve al 5, pero su único vecino ya esta visitado, se devuelve al 1, desde ahí pasa al 3, este no tiene vecinos y se devuelve otra vez al 1, y finalmente al 2, como su vecino ya esta visitado regresa al 1, y como este no tiene mas vecinos finaliza el algoritmo.

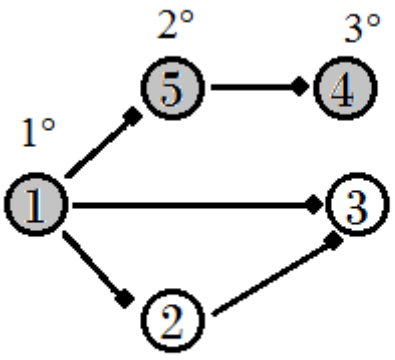


Figura 2.2

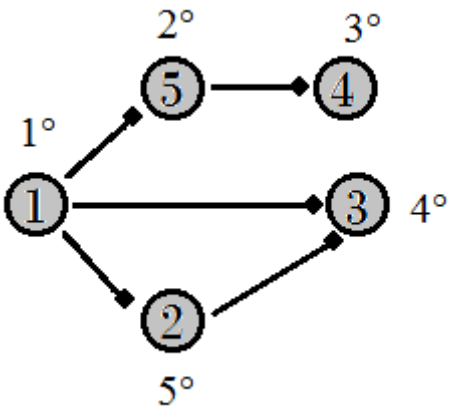


Figura 2.3

La implementación del DFS es corta y consiste básicamente en un algoritmo de backtraking, tiene una complejidad algorítmica $O(V + E)$ con V como el numero de nodos y E como el numero de aristas.

```cpp
vector<vector<int>> grafo(10);
bool visitado[10];
```

```
3
4  void dfs(int nodo){
5      visitado[nodo] = true;
6      for(int i = 0; i < grafo[nodo].size(); i++){
7          if(!visitado[grafo[nodo][i]]){
8              dfs(grafo[nodo][i]);
9          }
10     }
11 }
```

BFS.

El BFS(Breadth First Search) o búsqueda en anchura es un algoritmo que permite recorrer un grafo de forma iterativa, la idea consiste en tomar un nodo inicial y recorrer todos los nodos que están a un "salto"de distancia(es decir, sus vecinos), después los nodos que están a dos "saltos"de distancia y así sucesivamente hasta recorrer todos los nodos.

La implementación consiste de utilizar una cola(queue) inicialmente con un nodo, este nodo se debe marcar como visitado, después se empieza a iterar los elementos en la cola, en cada iteración se desencola el siguiente nodo, después se agrega todos sus nodos vecinos no visitados a la cola y se marcan como visitados, el algoritmo termina cuando la cola queda vacía. En la siguiente figura se observa el orden en el cual se visitan los nodos en el grafo de ejemplo anterior.

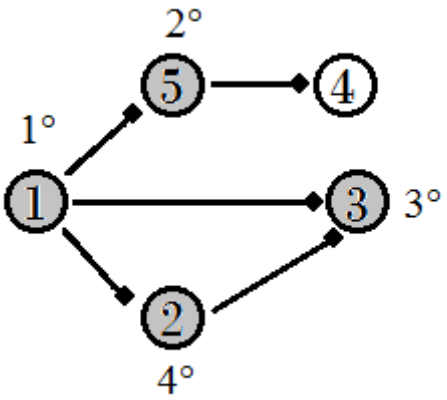


Figura 2.4

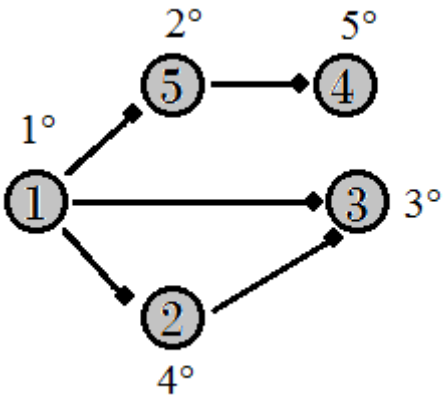


Figura 2.5

La implementación del BFS es sencilla, tiene una complejidad algorítmica $O(V + E)$ al igual que el DFS, ademas al ser iterativo evita los problemas de stack overflow que puede presentar el DFS al trabajar con grafos muy extensos.

```
1  vector<vector<int>> grafo(10);
2  bool visitado[10];
3
4  void bfs(int nodo){
5      queue<int> cola;
```

```
6       cola.push(nodo);  visitado[nodo] = true;
7
8       while(cola.size()){
9           nodo = cola.front();  cola.pop();
10
11          for(int i = 0; i < grafo[nodo].size(); i++){
12              if(!visitado[grafo[nodo][i]]){
13                  cola.push(grafo[nodo][i]);
14                  visitado[grafo[nodo][i]] = true;
15              }
16          }
17      }
18  }
```

Observaciones.

Ambos algoritmos son igual de buenos para recorrer grafos de forma general, dependiendo del problema a resolver uno de los dos algoritmos puede resultar mejor que el otro, esto se debe a las distintas formas de recorrer los grafos, el DFS al ser un algoritmo recursivo permite recorrer los nodos de una forma ordenada, siguiendo un camino en el cual cada nodo es vecino del anterior, en cambio el BFS permite recorrer los grafos por niveles, comenzando con los nodos mas cercanos al nodo inicial, de esta forma se puede encontrar mas fácil distancias entre los distintos nodos. Al utilizar DFS se debe tener cuidado con la pila de memoria(call stack) la cual se puede llenar rápidamente con grafo extensos, produciendo errores de ejecución.

2.3. Grafos bipartitos

Un grafo bipartito es un grafo el cual se puede dividir sus vértices en dos conjuntos X, Y en los cuales todos los nodos de un mismo conjunto no están conectados entre si, es decir no existe ninguna arista la cual conecte dos nodos de un mismo conjunto, los conjuntos X, Y pueden estar vacíos.

Para verificar si un grafo es bipartito se puede utilizar un algoritmo sencillo de DFS o BFS, la implementación consiste en tomar un nodo inicial y marcarlo como nodo perteneciente al conjunto X después se visitan y marcan sus vecinos como nodos pertenecientes al conjunto Y, este proceso se repite para cada nodo visitando y marcando sus vecinos como nodos del conjunto opuesto, en caso de encontrar en algún momento a dos nodos vecinos del mismo conjunto entonces significa que el grafo no es bipartito, pero si no se encuentran contradicciones significa que si es bipartito.

```
1   typedef pair<int, int> ii;
2   vector<vector<int>> grafo;
3   bool color[200], used[200];
4
5   bool BipartiteCheck(int nodo){
6       queue<ii> cola;
7       cola.push(ii(nodo, 0));
```

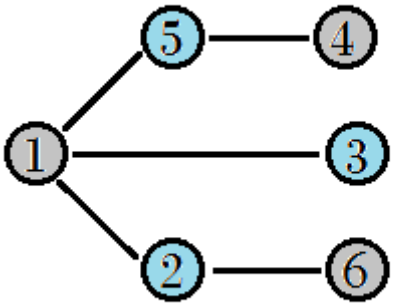


Figura 2.6: Grafo bipartito

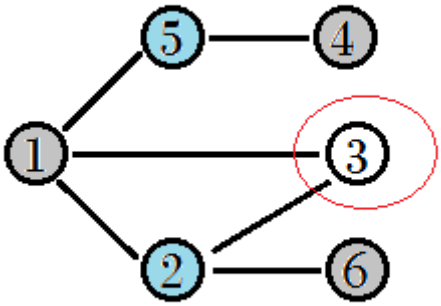


Figura 2.7: Grafo no bipartito

```
8       memset(used, false, sizeof(used));
9       color[nodo] = 0;  used[nodo] = true;
10      int auxnodo;
11
12      while(cola.size()){
13          ii x = cola.front();  cola.pop();
14          nodo = x.first;
15          bool newcolor = 1 - x.second;
16
17          for(int i = 0; i < grafo[nodo].size(); ++i){
18              auxnodo = grafo[nodo][i];
19              if(used[auxnodo] && newcolor != color[auxnodo])
20                  return false;
21              if(used[auxnodo]) continue;
22
23              cola.push(ii(auxnodo, newcolor));
24              color[auxnodo] = newcolor;
25              used[auxnodo] = true;
26          }
27      }
28      return true;
29  }
```

2.4. Topological sort

El ordenamiento topológico o toposort de un grafo G acíclico y dirigido es un ordenamiento lineal de los vértices, de tal forma que para cada arista de u a v se cumple que el nodo u aparece antes que el nodo v, este ordenamiento suele ser utilizado para representar tareas que dependen de otras, es decir si una tarea x depende de otra tarea y, es decir existe una arista $y \to x$, entonces x no puede iniciar hasta que y este terminada, entonces en el ordenamiento topológico primero debe aparecer y y después x, este ordenamiento

puede dar un posible orden para realizar tareas. En la figura 2.8 se puede observar un sencillo ejemplo.

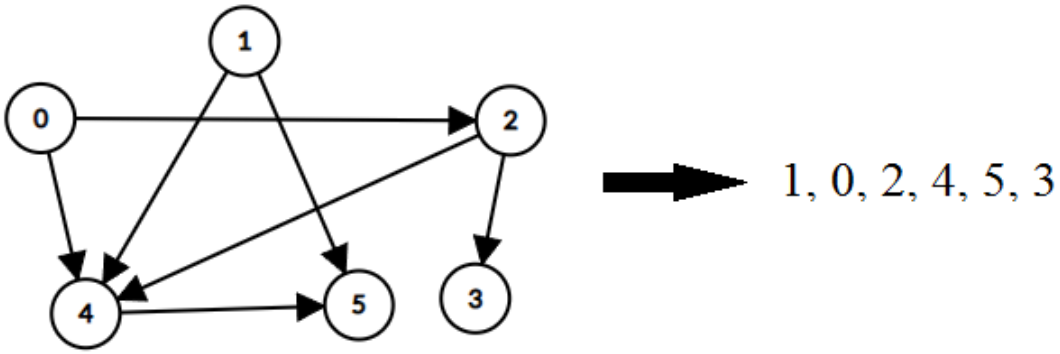


Figura 2.8

2.4.1. Topological sort con DFS

Una manera sencilla de encontrar el toposort es utilizando DFS, la solución se construye agregando los nodos en una lista de enteros, sin embargo el objetivo sera agregar primero los nodos destino antes que los nodos de origen, es decir, construir un toposort invertido de tal forma que final solo se debe invertir la lista para obtener el resultado.

Para realizar esto se necesita agregar el nodo v a la lista justo antes de terminar el llamado del DFS, debido a que en ese punto ya se recorrieron y agregaron todos los nodos que van después de v, se recomienda hacer el proceso manualmente para que se pueda entender mejor. La complejidad final seguirá siendo la misma de un DFS normal.

```
vector<vector<int>> grafo(MAX);
bool visitado[MAX];

void dfs(int nodo, vector<int> &toposort){
    visitado[nodo] = true;
    for(int i = 0; i < grafo[nodo].size(); i++){
        if(!visitado[grafo[nodo][i]])
            dfs(grafo[nodo][i], toposort);
    }
    toposort.push_back(nodo);
}

vector<int> toposort_dfs(int n){
    vector<int> toposort;
    memset(visitado, false, sizeof(visitado));

    for(int i = 0; i < n; ++i) {
        if(!visitado[i]) dfs(i, toposort);
    }
    reverse(toposort.begin(), toposort.end());
    return toposort;
```

```
}
```

2.5. Minimum spanning tree

También conocido como árbol recubridor mínimo o abreviado en inglés MST, es un problema clásico en la teoría de grafos, consiste en lo siguiente, Sea $G = (V, E)$ un grafo conexo, ponderado y no dirigido, se denomina árbol recubridor a cualquier subgrafo T tal que T sea un árbol e incluya todos los vértices de G, entonces el árbol recubridor mínimo es un subgrafo $T*$ tal que la suma de los pesos de las aristas de $T*$ es menor o igual al de todos los demás árboles recubridores T, esto significa que pueden haber varios posibles árboles recubridores mínimos. Existen varios algoritmos que pueden resolver este problema, dos de ellos son el algoritmo de Kruskal y el algoritmo de Prim.

2.5.1. Algoritmo de Kruskal

Es un algoritmo inventado por el matemático Joseph Kruskal, permite hallar el MST utilizando un enfoque greddy y funciona en complejidad $O(E * log(V))$ con E como el número de aristas y V el número de nodos, consiste en representar el grafo como una lista de aristas, ordenar la lista según el peso de las aristas e ir seleccionando aristas para formar un árbol, entonces para cada arista se verifica que no forme un ciclo con las demás aristas seleccionadas, de esta forma se priorizan las aristas de menor peso y el árbol formado al final es un MST.

la implementación consiste en representar el grafo como una lista de aristas, posteriormente ordenarlas de forma ascendente según el peso de la arista, después se empieza a construir un árbol, se van recorriendo las aristas y para cada una se verifica si al agregarla al árbol se forma un ciclo, de ser así la arista se descarta, en caso contrario la arista se agrega al árbol, para realizar esa verificación se utiliza un disjoint set, el cual resulta mucho mejor computacionalmente que realizar un DFS, cada vez que se agrega una arista al árbol se actualiza el disjoint set uniendo los dos nodos de la arista en un mismo conjunto, entonces si en algún momento los dos nodos de una arista se encuentran en un mismo conjunto se puede concluir que esos dos nodos ya están dentro de una misma componente conexa y por lo tanto, al agregar esa arista se estaría formando un ciclo, de esta forma se valida cada arista rápidamente. El algoritmo termina cuando se han agregado $n - 1$ aristas al árbol(la cantidad necesaria para que el árbol conecte todos los N nodos) o cuando se acaben las aristas, lo que significa que el árbol no es conexo.

```
#define mpiii(peso, inicio, destino) iii(peso, ii(inicio, destino))
typedef pair<int, int> ii;
typedef pair<int, ii> iii;

int kruskal(vector<iii>& grafo, int n) {
  sort(grafo.begin(), grafo.end());
  union_find uf(n+1);
  int peso = 0, cont = 0;

  for(int i = 0; i < grafo.size(); i++) {
    ii arista = grafo[i].second;
```

```
12      if(!uf.mismo_grupo(arista.first, arista.second)) {
13        uf.unir(arista.first, arista.second);
14        peso += grafo[i].first;
15        if(++cont == n - 1)
16          return peso;
17      }
18    }
19    return -1;
20  }
```

Capítulo 3

Programación dinámica

3.1. Introducción a la programación dinámica

La programación dinámica es una metodología utilizada para reducir la complejidad computacional a un algoritmo, es usada principalmente para resolver problemas de optimización y conteo, se basa en la estrategia *divide y vencerás*, consiste en tomar un problema complejo y dividirlo sucesivamente en subproblemas mas pequeños hasta llegar a un caso base, y partir de ahí empezar a construir la solución de cada subproblema, hasta llegar a una solución global. Durante la búsqueda de soluciones se utiliza tablas de memorización, en la cuales se guarda la solución óptima de cada subproblema. Como abreviatura a programación dinámica se suele utilizar *dp*, pues son sus siglas en ingles.

Los problemas de programación dinámica presentan tres condiciones básicas:

1. El problema se puede dividir en subproblemas, y estos a su vez en más subproblemas, y así hasta llegar a uno o múltiples casos base.

2. La solución óptima de cada subproblema depende de la solución óptima de cada uno de sus propios subproblemas, entonces cumple con el principio de optimalidad de Bellman.

3. Se presenta superposición de problemas, es decir, existen sub-problemas que aparecen múltiples veces a lo largo de la búsqueda de la solución general. Esta es la razón principal que le permite tener menor complejidad computacional.

Ejemplo inicial.

Como un ejemplo básico se puede utilizar la sucesión de Fibonacci, esta comienza con los números 0 y 1, y a partir de estos dos números iniciales los siguientes son la suma de los dos anteriores, entonces los primeros números de Fibonacci son los siguientes:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...$$

Ahora se buscará resolver el siguiente problema: dado un numero i calcular el i-esimo numero de Fibonacci. Para dar una solución con dp se deben identificar los subproblemas, y la solución a cada subproblema se debe guardar en una tabla de memorización. En este caso la tabla de memorización será un vector v en el cual la posición i-esima del vector corresponderá al número i-esimo de la sucesión, y para la división en subproblemas se puede escribir la siguiente formula recursiva:

$$fibo(n) = \begin{cases} n & \text{if } n \leq 1 \\ fibo(n-1) + fibo(n-2) & \text{if } n \geq 2 \end{cases}$$

Para implementar dp surgen dos enfoques: Top-Down y Buttom-Up.

Buttom-Up

En este enfoque se irá recorriendo la tabla de memorización mientras se llena, se comienza con los casos base de la formula recursiva y a partir de ahí se deberá calcular y guardar los demás resultados. En este enfoque primero se calcula el resultado de todos los subproblemas antes de dar alguna solución global. Aunque este enfoque sea iterativo puede resultar mas difícil de implementar, pues por lo general es menos intuitivo.

Para este ejercicio comenzamos llenando en un vector v los casos base: $v[0] = 0$ y $v[1] = 1$, después se llena el resto del vector según se indica en la formula recursiva: $v[i] = v[i-1] + v[i-2]$, de tal forma que $v[i-1]$ y $v[i-2]$ son valores que se han calculado antes y ademas $v[x] = fibo(x)$. A continuación se muestra el ejemplo en C++ calculando los primero 45 números de la sucesión:

```
void dp(){
    v[0] = 0;
    v[1] = 1;
    for(int i = 2; i < 45; i++)
        v[i] = v[i-1] + v[i-2];
}
```

Top-Down

A diferencia del anterior enfoque en este se utilizan llamados recursivos, en los cuales se irá guardando los resultados en la tabla de memorización mientras se calculan, y así evitar que se realice dos veces el mismo llamado recursivo. En este enfoque se hace mas notorio la importancia de la tabla de memorización, si se escribiera únicamente la formula recursiva, como se muestra en el código de abajo, se obtendría una solución ineficiente, pues para $fibo(6)$ se obtendría el árbol de recursión de la figura 3.1.

```
int fibo(int x){
    if(x <= 1)
        return x;
    else
        return fibo(x-1) + fibo(x-2);
}
```

Se puede observar en la figura 3.1 que se realizan múltiples llamados repetidos, $fibo(4)$ se repite 2 veces, $fibo(3)$ 3 veces, $fibo(2)$ 5 veces, $fibo(1)$ 8 veces y $fibo(0)$ 5 veces, resultan bastantes llamados recursivos solamente para encontrar $fibo(6)$, esta solución tiene una complejidad exponencial, en números pequeños no es muy notorio pero cuando intentamos encontrar $fibo(45)$ el tiempo de ejecución se hace muy alto, entonces para mejorar el tiempo se debe implementar una tabla de memorización, en la cual se guarde el resultado de cada llamado recursivo, ademas se debe agregar un condicional al inicio del método para validar si la solución de ese subproblema ya fue encontrada, en tal caso se devuelve el valor guardado, sino se calcula y se guarda, en este ejemplo la tabla comienza llena con -1, un valor que nunca es usado en la solución, de tal forma que si una posición es igual a -1 entonces se debe calcular.

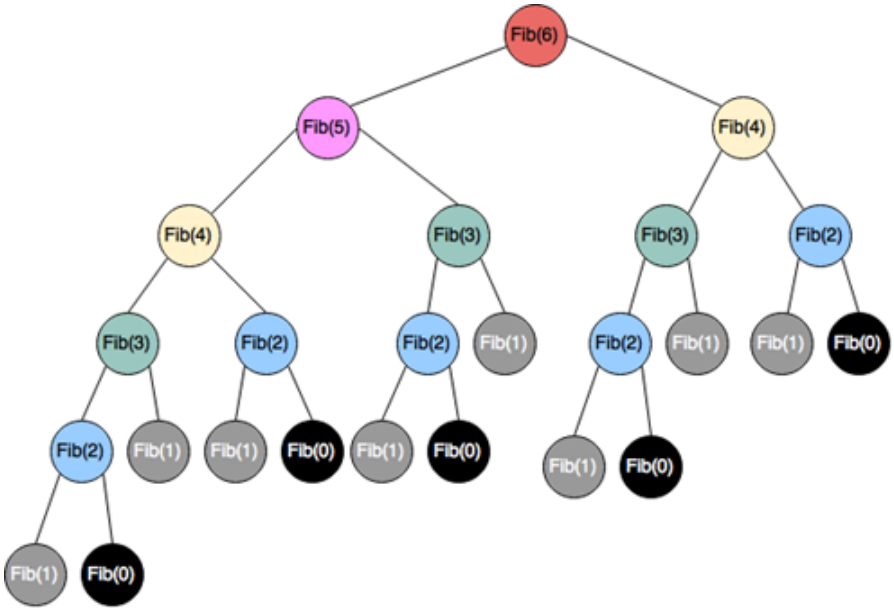


Figura 3.1

```
1 int fibo(int x){
2     if(v[x] != -1) return v[x];
3
4     if(x <= 1)
5         return v[x] = x;
6     else
7         return v[x] = fibo(x-1) + fibo(x-2);
8 }
```

Este ejercicio resulta muy trivial, se puede resolver fácilmente sin pensar en dp, ahora se mostrara otro problema en el cual se hace necesario utilizar dp para dar una solución eficiente.

Recorrido óptimo en una matriz.

Teniendo una matriz de n x m con números enteros, se quiere conocer la suma máxima que se puede obtener recorriendo la matriz, comenzando desde la esquina superior-izquierda y terminando en la esquina inferior-derecha, realizando únicamente pasos hacia la derecha y abajo, por ejemplo:

5	6	4
3	8	5
2	11	15
5	2	17

Cuadro 3.1

El camino óptimo es 5-6-8-11-15-17 y la suma es 62. Este es un problema de optimización, a simple vista la solución consistiría de siempre ir a la casilla adyacente con mayor valor, pero no siempre es el mejor camino, por ejemplo con la siguiente matriz:

1	12	4	9
6	5	21	15
35	18	8	10
12	2	4	15

Cuadro 3.2

No es una solución óptima ir a la siguiente casilla con mayor valor, pues haciendo esto se tendría el siguiente camino 1-12-5-21-15-10-15 con un acumulado de 79, en cambio el camino 1-6-35-18-8-10-15 tiene un acumulado de 93, entonces con la anterior estrategia no siempre se obtiene el camino óptimo. En este problema es necesario realizar una toma de decisiones, pues existen múltiples opciones de caminos, y se necesita encontrar el conjunto de decisiones que permita llegar a un resultado optimo, una solución inicial puede ser con BackTraking pero esta tendría complejidad exponencial O(2^n), por lo tanto se necesita utilizar otro enfoque.

Este problema se puede resolver con programación dinámica, pues se puede encontrar la solución general a partir de la solución de sub-problemas más pequeños, supongamos que tenemos una matriz T de tamaño n x m, para encontrar el camino óptimo hasta $T[n-1][m-1]$(ultima casilla) es necesario primero encontrar el óptimo de $T[n-2][m-1]$ y $T[n-1][m-2]$, pues para llegar a $T[n-1][m-1]$ hay dos opciones, llegar por arriba (equivalente a tomar la decisión ir-abajo desde $T[n-2][m-1]$) o llegar por la izquierda (equivalente a tomar la decisión ir-derecha desde $T[n-1][m-2]$), entonces la solución general sera *T[n-1][m-1] + máximo(camino óptimo hasta T[n-2][m-1], camino óptimo hasta T[n-1][m-2])*, y a su vez el óptimo para $T[n-2][m-1]$ y $T[n-1][m-2]$ se calcula de manera similar, de esta forma se puede dividir el problema general en subproblemas.

La solución general se puede escribir como una formula recursiva $f(i,j)$ la cual devolverá el valor del recorrido optimo desde $T[0][0]$ hasta $T[i][j]$, donde i es la posición de fila y j la posición de columna, si i y j son iguales a 0 entonces el resultado es $T[0][0]$(se encuentra en el punto inicial), si solo i es 0 entonces el resultado es $T[0][j]+f(0,j-1)$, solo se puede llegar a $T[0][j]$ desde su izquierda, pues desde arriba estaría afuera de la matriz, si solo j es 0 entonces de manera similar se tiene como resultado $T[i][0]+f(i-1,0)$, con base en esto se puede construir la siguiente formula recursiva:

$$f(i,j) = \begin{cases} T[0][0] & \text{if } i,j = 0 \\ T[0][j] + f(0,j-1) & \text{if } i = 0 \\ T[i][0] + f(i-1,0) & \text{if } j = 0 \\ T[i][j] + max(f(i-1,j), f(i,j-1)) & \text{if } i,j \neq 1 \end{cases}$$

Realizando manualmente la función recursiva se puede tener mas claridad. El caso $f(0,0)$ indica que la solución es el elemento en $T[0][0]$, la solución para la primera fila sera la sumatoria de los elementos desde la primera casilla hasta la columna correspondiente, ese es el único camino posible, de similar manera ocurre con la primera columna (figura 3.2), para los demás casos se abren dos opciones, llegar desde arriba o llegar desde la izquierda (figura 3.3), la formula indica tomar el de mayor valor, de esta manera mientras se avanza se ira seleccionando el mejor camino hasta llegar a la ultima casilla (figura 3.4).

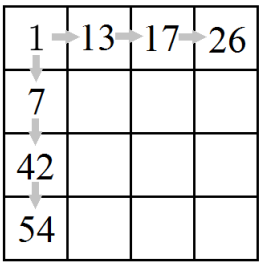


Figura 3.2

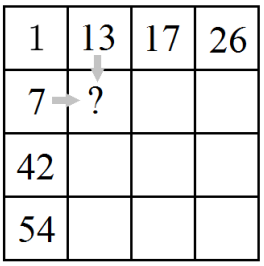


Figura 3.3

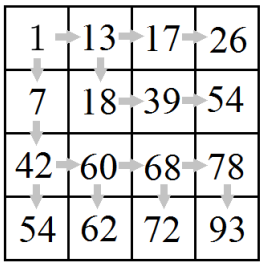


Figura 3.4

Para terminar el dp es necesario agregar la tabla de memorización, pues aplicar la formula directamente resultaría en un algoritmo de complejidad exponencial, como tabla de memorización se usara otra matriz de igual tamaño, en este ejemplo se llamara *memo* y guardara las soluciones, es decir, que en $memo[i][j]$ se guarda el acumulado óptimo para ir desde $T[0][0]$ hasta $T[i][j]$, ahora se puede solucionar el problema usando cualquiera de los dos enfoques de dp. La complejidad algorítmica resultante para ambos enfoques es $O(n * m)$, resultando mucho mejor que un algoritmo de complejidad exponencial utilizando fuerza bruta.

Para usar Top-Down solamente se agrega la tabla de memorización a la formula recursiva, y se verifica si la solución al subproblema ya fue encontrada. Para obtener el resultado se debe invocar $f(n-1, m-1)$. La implementación es la siguiente:

```
int f(int i, int j){
    if(i==0 && j==0) return T[0][0];
    if(memo[i][j] != -1) return memo[i][j];

    if(i == 0) return memo[0][j] = T[0][j] + f(0, j-1);
    if(j == 0) return memo[i][0] = T[i][0] + f(i-1, 0);

    return memo[i][j]= T[i][j] + max(f(i-1, j), f(i, j-1));
}
```

Para la solución con Buttom-Up se debe recorrer la matriz y llenarla según la formula recursiva, se debe comenzar con los casos base antes de calcular lo demás. De esta forma para conocer la solución se debe imprimir $memo[n-1][m-1]$. La implementación es la siguiente:

```
void dp(){
    memo[0][0] = T[0][0];//caso base

    for(int i = 1; i < n; i++)//llenar filas
        memo[i][0] += memo[i-1][0] + T[i][0];

    for(int i = 1; i < m; i++)//llenar columnas
        memo[0][i] += memo[0][i-1] + T[0][i];

    for(int i = 1; i < n; i++)
        for(int j = 1; j < m; j++)
```

```
12              memo[i][j] += max(memo[i - 1][j], memo[i][j - 1]) + T[i][j];
13  }
```

3.2. Sub-SetSum

También conocido como suma de subconjuntos, el problema consiste en que dado un conjunto de números enteros V, encontrar si es posible que un número n sea el resultado de la suma de los elementos de algún subconjunto de V, por ejemplo con el conjunto V=1, 2, 5 es posible sumar 7 usando el subconjunto {2, 5}, con este conjunto V seria posible sumar {1, 2, 3, 5, 6, 7, 8}.

Este es un problema de decisión, para cada elemento en V existen dos posibilidades, tomarlo o no, y validar si el subconjunto de elementos seleccionados suman n, un algoritmo de BackTraking tiene una complejidad O(2^n) para cada consulta, es decir, si se necesita validar Q números distintos el proceso se repite Q veces, una mejor opción es calcular todos los posibles subconjuntos y guardar todas las posibles soluciones, esto implica un precalculo con complejidad O(2^n) que es muy alto pero permite responder cada consulta en complejidad constante.

Este problema puede ser resuelto de manera mas eficiente usando dp, se puede reescribir el numero n como: $n = V[k_1] + x_1$, con $V[k_1]$ como algún numero en V, de manera similar se puede reescribir x_1 como $x_1 = v[k_2] + x_2$ y así sucesivamente hasta encontrar $x_n = v[k_n] + 0$, de esta manera si se toma cada elemento de V que fue utilizado, se podría encontrar el subconjunto $\{v[k_1], v[k_2], ..., v[k_n]\}$ el cual sumando todos sus elementos obtenemos n.

En este problema el resultado deberá ser 'verdadero' o 'falso', mostrando si es posible sumar n con algún subconjunto, con los subproblemas establecidos se puede ahora definir los casos base, si en algún momento se llega a $x_n = 0$ entonces la solución global sera 'verdadero'(se encontró una solución, entonces es posible sumar n), pero si en algún momento usamos todos los elementos en V y aun $x_n > 0$, o en algún momento se llega a $x_n < 0$ entonces la solución en ese subproblema sera 'falso', entonces se tiene la siguiente función recursiva:

$$f(n,k) = \begin{cases} 1 & \text{if n} = 0 \\ 0 & \text{if n} < 0, \text{ k}> \mathit{longitud}(\text{V}) \\ 1 & \text{if } f(n - V[k], k+1) = 1 \text{ o } f(n, k+1) = 1 \\ 0 & \text{if } \mathit{otro} \end{cases}$$

Con esta formula la solución sera igual a $f(n, 0)$, para implementarla se debe hacer una tabla de memorización, la cantidad de columnas de la tabla debe ser por lo menos igual a la longitud del vector, y la cantidad de filas por lo menos igual al resultado más grande que se puede obtener, y como valor inicial se puede usar -1.

```
1  int memo[10][5];
2  int dp(int n, int k, vector<int> &V){
3      if(memo[n][k] != -1) return memo[n][k];
4      if(n == 0) return 1;
```

```
5       if(n < 0 || k >= V.size()) return 0;
6
7       if(dp(n-V[k],k+1,V)==1 || dp(n,k+1,V)==1)
8           return memo[n][k] = 1;
9       else return memo[n][k] = 0;
10  }
```

Otra posible solución al problema es utilizando el enfoque Buttom-Up, los subproblemas se podrían ver de la siguiente manera, tomando a m con el tamaño del vector V, la solución general incluye sumar el elemento $V[m-1]$ a todas las posibles sumas encontradas hasta $V[m-2]$, y este a su vez depende de encontrar las sumas hasta $V[m-3]$ y así sucesivamente hasta llegar a $V[0]$ que es el caso base, mas detalladamente la solución hasta $V[0]$ es solamente él mismo, las posibles sumas hasta $V[1]$ es el conjunto $\{V[0], V[1], V[0]+V[1]\}$, ahora las posibles sumas hasta $V[2]$ son $\{V[0], V[1], V[2], V[0]+V[1], V[0]+V[2], V[1]+V[2], V[0]+V[1]+V[2]\}$, y así sucesivamente hasta llegar a $V[m-1]$, descartando números repetidos.

Por ejemplo sea V=$\{3, 5$ y $8\}$, comenzando con $V[0]$ el conjunto de soluciones es $\{3\}$, tomando el elemento $V[1]$ el conjunto de soluciones pasa a ser a $\{3, 5, 8\}$, y por ultimo incluyendo $V[2]$ el conjunto resultante es: $\{3, 5, 8, 11, 13, 16\}$. Una posible implementación es la siguiente:

```
1   bool memo[10][5];
2   void dp(vector<int> &V){
3       memset(memo, false, sizeof(memo));
4       for(int i = 0; i < V.size(); i++){
5           if(i){
6               for(int j = 1; j < 10; j++)
7                   if(memo[j][i - 1]){
8                       memo[j][i] = true;
9                       memo[j + V[i]][i] = true;
10                  }
11          }
12          memo[V[i]][i] = true;
13      }
14  }
```

En el anterior código se tienen dos ciclos, el primero recorriendo el vector V y el segundo buscando las soluciones anteriores para sumarles $V[i]$, como el caso base es cuando $i = 0$ entonces se debe omitir el segundo ciclo cuando i tiene este valor, ejecutando este código se obtendría todas las soluciones en la última columna, entonces si $memo[n][V.size()-1]$ es verdadero entonce si existe un sub-conjunto que sume n, en caso contrario no es posible.

3.3. Problema de la mochila(Knapsack problem)

Este es un problema clásico de programación dinámica, consiste en lo siguiente, hay una mochila que tiene una capacidad limitada de peso que puede contener, y hay un

grupo de objetos, cada uno tiene un valor y peso, el objetivo consiste en llenar la mochila con los objetos de tal forma que no se exceda su capacidad de peso y que el valor de los objetos que hay dentro de la mochila sea lo más alto posible. Por ejemplo:

i	0	1	2	3	4	5
W_i	1	2	4	5	4	2
V_i	2	7	5	9	5	3

Sea $N = 6$ la cantidad de objetos, $C = 10$ la capacidad de la mochila, W_i el peso del i-ésimo objeto y V_i el valor del i-ésimo objeto, entonces las respuesta es 21, usando los objetos 0, 1, 3, 5 obtenemos un peso 10 con valor 21.

Para cada objeto existen dos opciones, tomarlo o dejarlo, entonces para n objetos existen 2^n posibles opciones, de todas ellas la solución será una configuración que no exceda el peso de la mochila y la suma de los objetos seleccionados sea el más alto, una solución probando todos los posibles casos tendrá complejidad $O(2^n)$, la cual es bastante alta, mientras que una solución greddy no siempre va a funcionar.

Sin embargo es posible reducir bastante la complejidad trabajando con programación dinámica, sea x_i la capacidad usada al estar en el i-ésimo objeto, iniciando con $i = 0$ y $x_i = 0$, se repite el siguiente proceso: para cada objeto existe la decisión de no tomarlo y pasar al objeto $i + 1$ con $x_{i+1} = x_i$ o tomarlo y pasar al objeto $i + 1$ con $x_{i+1} = x_i + W_i$ siempre y cuando $x_i + W_i \leq C$, es decir que no se exceda la capacidad, entonces se puede escribir la siguiente función recursiva:

$$f(i,x) = \begin{cases} 0 & \text{if } i = N \\ max(f(i+1,x), f(i+1,x+W_i)+V_i) & \text{if } x + W_i \leq C \\ f(i+1,x) & \text{if } otro \end{cases}$$

Al programar esta función recursiva se obtendrá una complejidad de $O(2^n)$, pero al agregar la tabla de memorización la complejidad se reduce al tamaño de la tabla, quedado en $O(N * C)$, y finalmente se puede obtener la siguiente implementación.

```
int dp(int i, int x){
    if(i == N) return 0;
    if(memo[i][x] != -1) return memo[i][x];

    if(x+W[i] <= C)
        return memo[i][x] = max(dp(i+1,x), dp(i+1,x+W[i])+V[i]);
    else
        return memo[i][x] = dp(i+1,x);
}
```

3.4. Traveling salesman problem

El problema del vendedor viajero o *Traveling salesman problem* en ingles, es un problema clásico el cual consiste en lo siguiente: Un vendedor quiere visitar n ciudades comenzando desde una ciudad cualquiera y al finalizar regresar a la ciudad inicial, ¿cual es la ruta mas corta posible?. Este problema consiste en encontrar el ciclo hamiltoniano

mas corto en un grafo, la principal dificultad es la cantidad de posibles rutas, pues la cantidad de posibles caminos puede llegar hasta $n!$. Este es un problema clásico que ha sido estudiado por décadas y una de sus mejores soluciones es utilizando programación dinámica. Por ejemplo en la figura 3.5 la respuesta optima es 35 siguiendo el camino $2, 4, 3, 1, 5, 0, 2$.

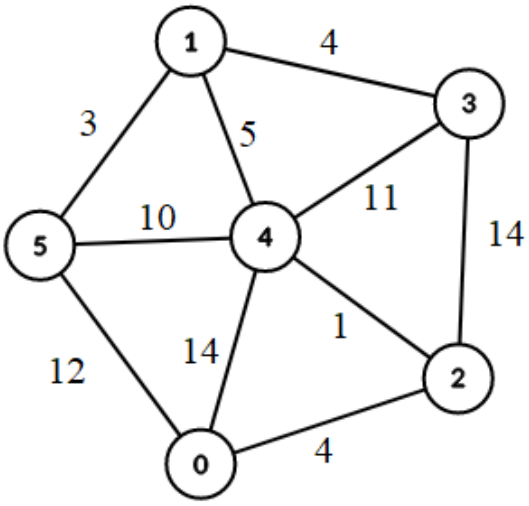


Figura 3.5

En grafos completos(grafos en los cuales existen aristas conectando todos los posibles pares de aristas) la cantidad total de posibles rutas es $n!$ lo cual hace que un algoritmo de backtraking solo se pueda aplicar para grafos muy pequeños, de menos de 12 aristas. Una mejor solución es utilizando programación dinámica con mascara de bits, en esta mascara se van a marcar como visitados los nodos de tal forma que si el i-esimo bit de la mascara es 1 entonces el nodo i ya esta visitado, si es 0 entonces esta disponible, la idea general consiste en ir tomando nodos disponibles e ir sumando la distancias entre el ultimo par de nodos seleccionados, como se puede observar en la siguiente formula recursiva.

$$f(m, v) = \begin{cases} dist(v, 0) & \text{if } m = 2^n - 1 \\ min(f(m|(2^i), i) + dist(v, i)) & \text{if } otro \end{cases}$$

Para la implementación se utiliza una tabla de memorización de tamaño 2^n x n, también se necesita una matriz de adyacencia para obtener rápidamente la distancia entre cualquier par de nodos, como la ruta es un ciclo el primer nodo también debe ser el ultimo, entonces se puede iniciar en cualquier nodo y al recorrer todos los demás se debe volver al nodo inicial, lo mas sencillo es dejar el primer nodo como el primero y ultimo.

En cada llamado recursivo se deben recorrer los n bits de la mascara buscando los nodos disponibles, para cada uno de ellos se debe hacer un llamado recursivo tomando ese nodo como el siguiente de la ruta, al final se debe tomar el valor que minimice el acumulado total, el caso base ocurre cuando $m = 2^n - 1$, es decir todos los n bits de la mascara están en 1 indicando que todos los nodos han sido visitados, entonces se debe devolver la distancia entre el nodo actual y el primer nodo para que la ruta sea un ciclo. Finalmente la respuesta se obtiene llamando la función $f(1, 0)$.

```
int n, target, grafo[MAX][MAX], memo[1 << MAX][MAX];

int dp(int mask, int v) {
  if(mask == target)
    return grafo[v][0];
  if(memo[mask][v] != -1)
    return memo[mask][v];

  int ans = inf;
  for(int i = 0; i < n; i++) {
    if(!(mask & (1<<i) )) {
      ans = min(ans, dp(mask | (1 << i), i) + grafo[v][i]);
    }
  }

  return memo[mask][v] = ans;
}
```

En cuanto a la complejidad algorítmica, existen $2^n * n$ posibles estados y para cada uno de ellos se realiza un ciclo de longitud n validando nodos disponibles, Esto hace que el algoritmo resultante tenga complejidad $O(2^n * n^2)$, a pesar de ser exponencial resulta mejor que $n!$ pues se puede trabajar aproximadamente hasta con 18 nodos.

Capítulo 4

Matematicas

4.1. MCD y MCM

MCD (Máximo común divisor)

El máximo común divisor de un conjunto de dos o mas números enteros es el máximo número entero el cual divide a todos los números del conjunto sin dejar residuo, por ejemplo el máximo común divisor de 35 y 15 es 5 porque es el máximo entero el cual los puede dividir a ambos. Como abreviatura se utiliza MCD o en ingles GCD (greatest common divisor).

El calculo del MCD de dos números se puede hacer de manera eficiente utilizando el algoritmo de Euclides el cual se puede realizar en $O(log(n))$ pasos, antes de explicarlo es necesario tener en cuenta las dos siguientes propiedades de la divisibilidad ($x|y$ significa que x divide a y con residuo 0).

$$x|y \rightarrow x|\alpha y \quad \forall \alpha \in Z \tag{4.1}$$

$$x|y \wedge x|y \pm z \rightarrow x|z \tag{4.2}$$

El algoritmo consiste en lo siguiente: dado dos números a y b se realiza la división entre ambos, obteniendo un cociente q y un residuo r, entonces $a = b * q_1 + r_1$ con $r_1 < b$, teniendo en cuenta que el MCD divide a a y b entonces también divide $b * q_1$ (propiedad 4.1), y también divide a r_1 (propiedad 4.2), el proceso se reduce a encontrar el MCD entre b y r_1 entonces se repite el proceso, $r_1 = b * q_2 + r_2$ con $r_2 < r_1$, y así sucesivamente hasta llegar a $r_n = 0$, lo cual indica que r_{n-1} divide a r_{n-1}, r_{n-2}, ..., b y a, entonces r_{n-1} es el MCD.

$$\begin{aligned} a &= b * q_1 + r_1 & r_1 &< b \\ b &= r_1 * q_2 + r_2 & r_2 &< r_1 \\ r_1 &= r_2 * q_3 + r_3 & r_3 &< r_2 \\ &... & & \\ r_{n-1} &= r_n * q_{n+1} + 0 & 0 &< r_n \end{aligned}$$

Por ejemplo, con $a = 35$ y $b = 15$, $a = b*2+5$, $r_1 = 5$, en el siguiente paso $b = 5*3+0$, $r_2 = 0$, como se ha llegado a un residuo 0, el algoritmo finaliza y la respuesta es el ultimo residuo diferente de 0, entonces $MCD(35, 15) = 5$.

MCM (Mínimo común múltiplo)

El mínimo común múltiplo de un conjunto de números enteros es el numero entero mas pequeño el cual es múltiplo de todos los números del conjunto, por ejemplo el mínimo común múltiplo de 35 y 15 es 105, pues es el menor entero tal que 35|105 y 15|105. Como abreviatura se usa MCM o en ingles LCM (lowest common multiple).

Para calcular el MCM también se puede utilizar el algoritmo de Euclides, pues existe una relación entre el MCD y el MCM. entonces $MCM(a * b) = (a * b)/MCD(a, b)$, esta formula es equivalente a $a * (b/MCD(a, b))$ como $MCD(a, b)|b$ se puede observar que el resultado sera un entero múltiplo de a, de igual es equivalente a $b*(a/MCD(a, b))$ y el resultado es un entero múltiplo de b, entonces $(a*b)/MCD(a, b)$ es múltiplo común de a y b.

Por ejemplo, con $a = 35$ y $b = 15$ el $MCM(a, b) = 35 * 15/5$ (en el ejemplo anterior se calculo el MCD de a y b), entonces el resultado es 105.

Implementación

La implementación del MCD consiste en simplemente aplicar los pasos descritos para el algoritmo de Euclides. La implementación del MCM consiste en aplicar la formula, se recomienda realizar primero la división para evitar un posible overflow al trabajar con números grandes.

```
1 int MCD(int a, int b) {
2   if(b) return MCD(b, a % b);
3   return a;
4 }
```

MCD

```
1 int MCM(int a, int b) {
2   return a*(b/MCD(b, a % b));
3 }
```

MCM

4.2. Criba de Eratóstenes

La búsqueda de números primos es un problema clásico, las soluciones sencillas se basan en tomar un numero x y empezar a validar si algún numero en el intervalo $[2, x-1]$ divide a x, si ninguno lo divide entonces x es un numero primo, esta idea se puede mejorar un poco, se puede validar desde el comienzo si x es un numero par, en ese caso x es primo solamente si es igual a 2, pues el único numero primo par es el 2, si x es impar se hace la validación si algún numero impar en el intervalo $[3, x-1]$ divide a x, existe una propiedad la cual indica que el menor factor primo de un numero x es menor a $\sqrt{x}$, entonces nuevamente es posible reducir el intervalo, finalmente un entero positivo x mayor a 1 es primo si es igual a 2 o si ningún numero impar en el intervalo $[3, \sqrt{x}]$ divide a x. Este método tiene complejidad $O(\sqrt{x})$ para cada numero, si se necesita verificar n números, el proceso se tendría que repetir n veces, por lo tanto si se requiere obtener una lista de

números primos se tiene que buscar una mejor solución

La criba de Eratóstenes es un algoritmo para encontrar todos los números primos hasta un numero n, cosiste de un arreglo de números los cuales se irán marcando si son compuestos, si no están marcados entonces son primos, inicialmente todos son considerados primos (no están marcados), se comienza tomando el 2 y se marcan todos los múltiplos de 2 como números compuestos, después se toma el 3 y se marcan todos los múltiplos de 3 como números compuestos, al llegar al 4 este ya estará marcado entonces el 4 es compuesto, en este caso se omite y se pasa al 5 el cual no esta marcado, y así sucesivamente, los números que no estén marcados son números primos, en la figura 4.1 se puede observar un sencillo ejemplo con $n = 17$.

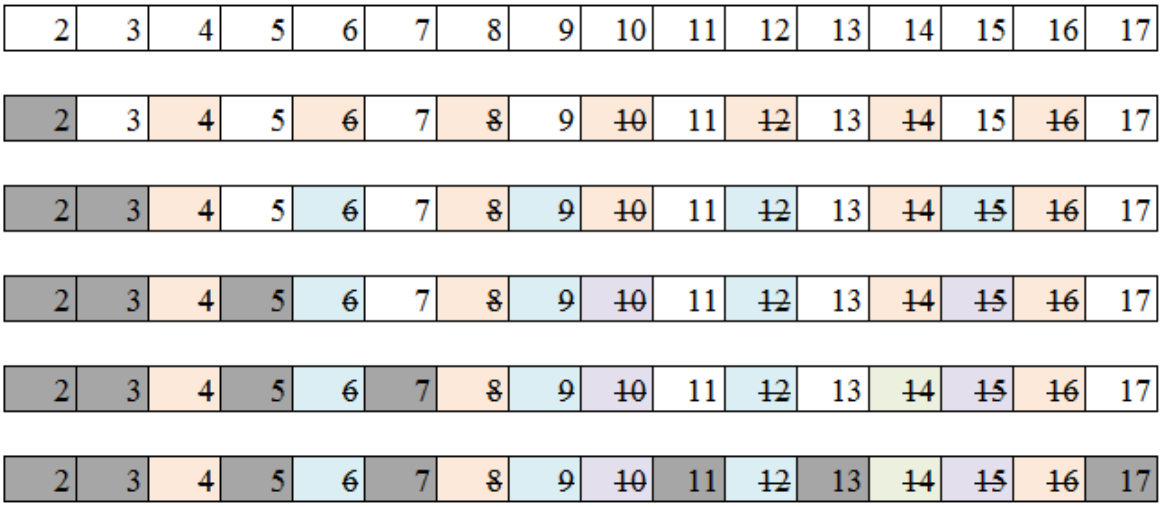


Figura 4.1

La implementación consiste en usar un arreglo de bool inicialmente en true (a excepción del 0 y 1 que no son primos), después se recorre el arreglo verificando los números que no estén marcados, para cada numero i no marcado se empieza a marcar los múltiplos de i, los múltiplos se pueden empezar a marcar desde i^2. Al final se obtiene un arreglo el cual permite validar si un i es primo y permite obtener un vector con la lista de números primos hasta n, el algoritmo tiene complejidad $O(n * log(log(n)))$.

```
1  bool esPrimo[1000005];
2  vector<int> primos;
3
4  void criba(int n) {
5      memset(esPrimo, true, sizeof(esPrimo));
6      esPrimo[0] = esPrimo[1] = false;
7
8      for(int i = 2; i < n; ++i){
9          if(!esPrimo[i]) continue;
10
11         for(int j = i*i; j < n; j += i)
12             esPrimo[j] = false;
```

```
13          primos.push_back(i);
14      }
15  }
```

Capítulo 5

Cadenas

5.1. KMP

El string matching es un problema clásico de las ciencias de la computación, consiste en buscar si una cadena de texto T de longitud n contiene una subcadena P de longitud m, la solución ingenua consiste en deslizar la cadena P sobre toda la cadena T, comparando carácter por carácter si encajan, es decir, se empieza verificando si la subcadena $T[0, n]$ es igual a P, en caso de fallo(caracteres diferentes) se compara $T[1, n+1]$ con P y así sucesivamente, dando como resultado una complejidad de $O(n * m)$, la cual resulta muy alta cuando los cadenas de texto son muy largas.

El algoritmo KMP(Knuth-Morris-Pratt) permite encontrar todas las apariciones de P en T utilizando una tabla de precalculo B sobre la cadena P, el algoritmo inicia con dos punteros, un puntero i sobre T y otro puntero j sobre P, ambos avanzan a la par mientras se comparan $T[i]$ y $P[j]$, en caso de fallo, el cursor j se devuelve a la posición indicada en la tabla de precalculo($B[j]$), de esta forma solo el cursor sobre P se devuelve y se evita devolver el cursor sobre T. El algoritmo KMP se divide en dos partes, una primera para calcular la tabla de precalculo B y una segunda para aplicar la búsqueda de P sobre T.

La tabla de precalculo consiste en buscar la longitud de los "bordes" de P, un borde se define como un substring que es prefijo de P y sufijo de un substring $P[0, k]$, entonces $B[k+1]$ es igual a la longitud del borde del substring $P[0, k]$, por ejemplo en la figura 5.1 se observa el borde para $P =$ "ABRACABRAABRA", se recomienda hacer el proceso a mano para que se pueda entender mejor.

idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	A	B	R	A	C	A	B	R	A	A	B	R	A	
Borde:	-1	0	0	0	1	0	1	2	3	4	1	2	3	4

Figura 5.1

Para el proceso de buscar las repeticiones de P sobre T se necesitan dos punteros como se había mencionado antes, el puntero i sobre T y j sobre P, observemos la figura 5.2, los punteros avanzan hasta llegar a al indice 9, donde hay un carácter de fallo, en este punto se puede observar la mejora que ofrece el KMP, pues solamente se devuelve j

a la posición $B[j]$ para seguir comparando, como se observa en la figura 5.3.

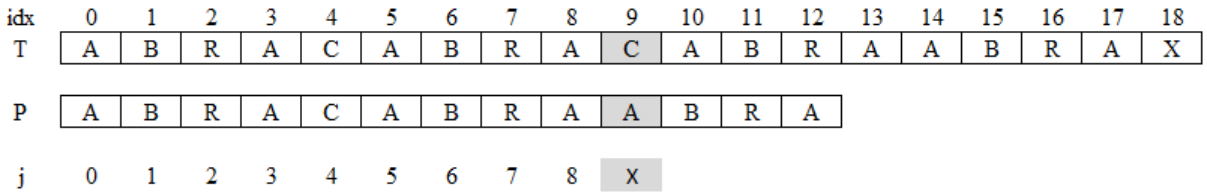


Figura 5.2

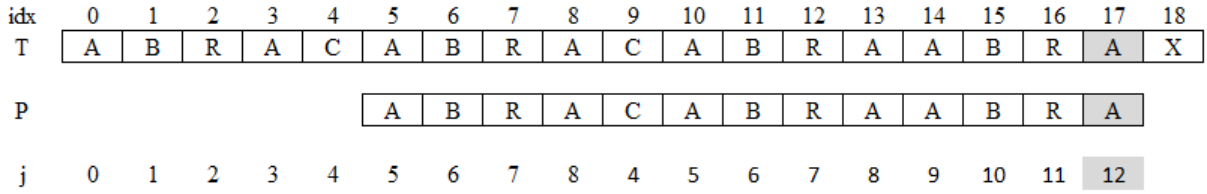


Figura 5.3

La implementacion no es larga y consiste en realizar primero la búsqueda de los bordes de P y posteriormente hacer la búsqueda de P sobre T. El algoritmo resultante tiene complejidad $O(n+m)$.

```
1  int B[MAX] = { -1 };
2
3  void bordes(string p) {
4    B[0] = -1; B[1] = 0;
5    int j = 0;
6    for(int i = 1; i < p.size(); i++) {
7      while(j >= 0 && p[j] != p[i]) j = B[j];
8      B[i + 1] = ++j;
9    }
10 }
11
12 void KMP(string t, string p) {
13   int j = 0;
14   for(int i = 0; i < t.size(); i++) {
15     while(j >= 0 && p[j] != t[i]) j = B[j];
16     j++;
17     if(j == p.size()) {
18       cout << "match␣en␣rango␣(" << (i - j + 1) << "," << i << ")\n";
19       j = B[j];
20     }
21   }
22 }
```

El KMP puede encontrar repeticiones con intervalos cruzados, por ejemplo con $P =$ "ABCABC" y $T =$ "DABCABCABCD", hay 2 repeticiones, en los rangos $(1,6)$ y $(4,9)$, en este caso se cruza el rango $(4,6)$ el cual aparece ambas repeticiones. Si no se requieren

intervalos cruzados se puede simplemente reiniciar el puntero j a 0 al encontrar cada repetición (linea 19 del código), de esta manera se continuará buscando desde el inicio de P.