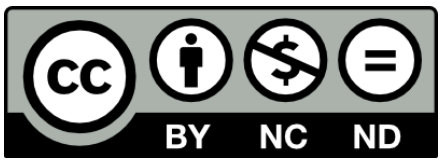
Algorítmica
Examen III

Los Del DGIIM, `losdeldgiim.github.io`

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



Algorítmica
Examen III

Los Del DGIIM, `losdeldgiim.github.io`

Laura Mandow Fuentes

Granada, 2023-2024

Asignatura Algorítmica.

Curso Académico 2018/2019.

Grado Doble Grado en Ingienería Informática y Matemáticas.

Grupo Único.

Profesor Juan Francisco Huete Guadix[1].

Descripción Convocatoria Extraordinaria.

Fecha 28 de junio de 2019.

[1]El examen lo pone el departamento.

Ejercicio 1. (2 puntos) Calcula el orden de eficiencia de la siguiente función recursiva:

```
1  int calcula(int ini, int fin) {
       if (ini < fin)
           return calcula(ini,fin-1) +
                  calcula(ini+1,fin)+
5                 calcula(ini+1,fin-1);
       else return ini-fin;
   }
```

Notemos por n a `fin - ini`. Tenemos que la recursión nos queda:

$$T(n) = \begin{cases} 1 & n = 0 \\ 2 \cdot T(n-1) + T(n-2) + 1 & n > 0 \end{cases}$$

Aplicando el cambio de variable $T(n) = x^n$, tenemos que la parte homogénea nos queda:

$$x^n - 2x^{n-1} - x^{n-2} = 0 \implies x^{n-2}(x^2 - 2x - 1) = 0$$

Como $x^{n-1} > 0$ para todo $n \in \mathbb{N}$, se tiene que las únicas soluciones del polinomio característico de la parte homogénea son las de la ecuación de segundo grado:

$$x^2 - 2x - 1 = 0 \implies x = 1 \pm \sqrt{2}$$

Añadiendo la parte no homogénea, se tiene que $1 = 1^n n^0$, por lo que el polinomio caractertístico de la ecuación en recurrente es:

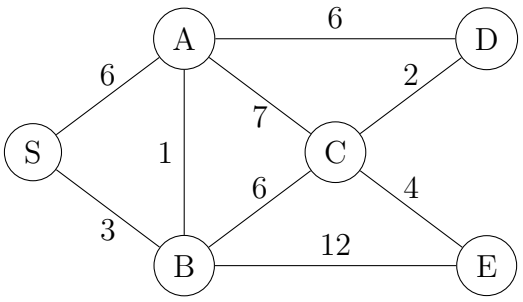
$$p(x) = (x-1)(x-1-\sqrt{2})(x-1+\sqrt{2})$$

Por tanto, la solución general de la ecuación recurrente es:

$$T(x) = x^n = c_1 1^n + c_2(1+\sqrt{2})^n + c_3(1-\sqrt{2})^n \in O((1+\sqrt{2})^n)$$

Por tanto, tenemos que el código dado es de orden **exponencial**.

Ejercicio 2. (2 puntos) Dado el grafo de la figura, donde aparecen las distancias entre los vértices adyacentes, se desea calcular cuáles son los caminos mínimos entre el vértice S y cada uno de los demás vértices. Describid detalladamente un algoritmo para resolver ese problema en general y aplicadlo paso a paso para ese grafo en concreto.



El ejercicio nos pide un algoritmo para calcular los caminos mínimos de **UN nodo a TODOS** para un grafo cuyos pesos son todos **positivos**. Por tanto, la solución al problema se trata de aplicar algoritmo de *Dijkstra*.

El algoritmo de Dijkstra calcula la distancia mínima desde un nodo (S en nuestro caso) a todos los demás. El algoritmo funciona de la siguiente forma:

1. De todos nodos a los que todavía no les hemos calulado su distancia mínima a S (no han sido seleccionados), seleccionamos aquel cuya distancia a S es mínima.

2. Actualizamos todos sus vecinos, de forma que si se puede llegar a uno de sus vecinos pasando por el nodo seleccionado por un camino más corto que el que ya tiene actualizamos su distancia y almacenamos que el nodo que va antes que el vecino es el nodo seleccionado.

3. Repetir hasta que todos los nodos hayan sido seleccionados.

Dado que el grafo de ejemplo tiene muchos arcos respecto al número de nodos, optaremos por la implementación para grafos densos.

```
const int INF = 1e9; // Infinito
int n; // Numero de nodos
vector<vector<pair<int, int>>> adjacency_list; // Lista de adyacencia
void dijkstra(int s){
    // d[i] = distancia (minima) de s a i
    vector<int> d(n,INF);
    d[s] = 0;
    // p[i] va antes que i en el camino de s a i
    vector<int> p(n,-1);
    // selected[i] si ya hemos seleccionado (calculado la distancia minima) de i
    vector<bool> selected(n,false);
    for(int i=0; i<n; ++i){
        int selected_node = -1;
        for (int j = 0; j < n; ++j){
            if(!selected[j] && (selected_node == -1 || d[j] < d[selected_node]))
                selected_node = j;
        }

        // Marcamos como seleccionado al nuevo nodo seleccionado
        selected[selected_node] = true;

        // Actualizamos las distancias de sus vecinos en caso de haga falta
        for(auto edge : adjacency_list[selected_node]){
            int neighbour = edge.first;
            int cost = edge.second;

            // Si podemos mejorar la distancia al vecino pasando
            // por el nuevo nodo seleccionado
            if(d[selected_node] + cost < d[neighbour]){
                d[neighbour] = d[selected_node] + cost;
                p[neighbour] = selected_node;
            }
```

```
33            }
34        }
35    }
```

Apliquemos ahora el algoritmo al grafo de ejemplo.

1. Inicialmente el vector de distancias comienza así:

S	A	B	C	D	E
0	∞	∞	∞	∞	∞

   Ya que lo único que sabemos es que la distancia de S a si mismo es 0. Procedemos entonces a seleccionar S por ser el nodo no seleccionado cuya a distancia a S es menor. Noteremos el nuevo nodo seleccionado en azul y los ya seleccionados en **negrita**.

2. Actualizamos los vecinos de S (A y B) y seleccionamos el nodo cuya distancia a S es menor de los no seleccionados.

S	A	B	C	D	E
0	6	3	∞	∞	∞

   En este caso es B. Ambos vecinos han sido actualizados ya que previamente su distancia a S era infinito.

3. Actualizamos los vecinos de B (A,C, E (S ya fue seleccionado por tanto ya se calculo su distancia mínima y por tanto esta no será actualizada nunca más)) y seleccionamos el nodo cuya distancia a S es menor de los no seleccionados.

S	A	**B**	C	D	E
0	4	**3**	9	∞	15

   En este caso es A. Todos los vecinos han sido actualizados (incluyendo A) salvo los ya seleccionados.

4. Actualizamos los vecinos (no seleccionados) de A (C y D) y seleccionamos el nodo cuya distancia a S es menor de los no seleccionados.

S	**A**	**B**	C	D	E
0	**4**	**3**	9	10	15

   En este caso es C. En este caso D fue actualizado al ser infinito pero C no por ser mejor el camino ya calculado.

5. Actualizamos los vecinos (no seleccionados) de C (E y D) y seleccionamos el nodo cuya distancia a S es menor de los no seleccionados.

S	**A**	**B**	**C**	D	E
0	**4**	**3**	**9**	10	13

En este caso es D. En este caso E fue actualizado al ser mejor el camino que pasa por C pero E no por ser mejor el camino ya calculado.

6. Finalmente tenemos que D no tiene vecinos no seleccionados, y ya no quedan más nodos aparte de E por actualizar, por tanto hemos acabado y tenenos que el vector de distancias a S que nos queda es:

S	A	B	C	D	E
0	4	3	9	10	13

Ejercicio 3. (2 puntos) Diseña un algoritmo para calcular $x^n, n \in \mathbb{N}$, con un coste $O(\log n)$ en términos del número de multiplicaciones.

Claramente nos están pidiento que implementemos la función `potencia` para exponentes **naturales** mediante el método de exponenciación rápida. Este método consiste en abusar de la propiedad matématica:

$$x^{2n} = (x^n)^2 = x^n \cdot x^n$$

Y también en aplicar

$$x^{n+1} = x^n \cdot x$$

De esta forma, podemos definir la función potencia para exponentes naturales recursivamente:

$$x^n = \begin{cases} 1 & n = 0 \\ x^{n/2} \cdot x^{n/2} & n > 0 \wedge n\ par \\ x^{n/2} \cdot x^{n/2} \cdot x & n > 0 \wedge n\ impar \end{cases}$$

Donde notamos por $n/2$ al máximo número entero k tal que $2k \leqslant n$ (dicho de otro modo, entedemos la división de enteros igual que c++). Por ejemplo $\frac{1}{2} = 0$ y $\frac{3}{2} = 1$. Equivalentemente, si n es par la división es la que conocemos de toda la vida, y si n es impar se corresponde con $\frac{n-1}{2}$, ya que sabemos que $n - 1$ es par.
Definimos la siguiente función recursiva:

$$potencia(x, n) = \begin{cases} 1 & n = 0 \\ potencia(x, n/2) \cdot potencia(x, n/2) & n > 0 \wedge n\ par \\ potencia(x, n/2) \cdot potencia(x, n/2) \cdot x & n > 0 \wedge n\ impar \end{cases}$$

Nos fijamos en que $potencia(x, n/2)$ hace falta dos veces en cada paso, motivo por el cual hará falta almacenar este resultado en alguna variable para evitar tener que recalcularlo.
La implementación en c++ sería:

```
1 double potencia(double x, int n){
2     if(n == 0) return 1;
3     int y = potencia(x,n/2);
4     y = y * y;
5     if(n % 2) // n impar
6         y *= x;
7     return y;
8 }
```

La función de eficiencia sería:

$$T(n) = T(n/2) + 1$$

Por tanto es claramente de orden **logarítmico**.

Ejercicio 4. (2 puntos) Consideremos un mapa formado por N países. Queremos viajar entre países. Cada vez que atravesemos una frontera tenemos que pagar una tasa. Todas las tasas son conocidas. Construid un algoritmo de programación dinámica que determine el coste del viaje más barato entre dos países dados.

Tenemos que se trata de un problema de caminos mínimos. En este caso el coste de ir de un nodo (país) a otro es la tasa que nos imponen en la frontera. Entendemos que todas las tasas son positivas (no tendría mucho sentido que nos dieran dinero por viajar a otro país) y que el enunciado desea saber el viaje más barato entre dos países **cualesquiera**. Por tanto, se trata de calcular el camino mínimo de **TODOS a TODOS** los nodos y que los pesos de los arcos son **positivos**. Claramente debemos usar el algoritmo de *Floyd-Warshall*. Este algoritmo emplea la técnica de programación dinámica, tal y como se pide en el enunciado.

La implementación de este algoritmo en C++ es la siguiente:

```
int n; // Num nodes
int dist[n][n]; // dp (min distance to go from i to j)

for(int k = 0; k < n; k++){
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            if(dist[i][j] > dist[i][k] + dist[k][j]){
                dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
}
```

Ejercicio 5. (2 puntos) Diseñad un algoritmo de exploración de grafos que, dado un número natural n, devuelva todas las formas posibles en que un conjunto ascendente de números positivos sume exactamente n. Por ejemplo, si $n = 10$, la salida debería ser:

```
1+2+3+4
1+2+7
1+3+6
1+4+5
1+9
2+3+5
2+8
3+7
4+6
10
```

Especificad claramente, además del algoritmo, la representación de la solución, las restricciones explícitas e implícitas, así como las posibles cotas a utilizar.

Tenemos pues que se trata de un problema de exploración de grafos, por tanto debemos aplicar o *backtracking* o *branch and bound*.

Dado que el problema nos pide que imprimamos **todas** las soluciones, vamos a optar por aplicar *backtracking* al ser más sencillo, ya que no nos podemos beneficiar de "descartar" soluciones "peores" mediante la cola de prioridad de *branch and bound*, y por tanto nos supondría una complejidad añadida.

La solucion consistirá en un conjunto de conjuntos s, donde cada conjunto s es una k-tupla de numeros naturales (el cero no es natural en este caso) ordenada ascendentemente sin numeros repetidos cuya suma en n. Las escribiremos como $(x_1, x_2, ..., x_k)$. Notamos distintas tuplas pueden tener tamaños distintos.

Representaremos los tuplas de números con `set` ya que no podemos repetir números y vamos a realizar muchas inserciones y borrados por lo que nos conviene que sean operaciones eficientes.
Por comodidad emplearemos la siguiente estructura:

```
struct tupla {
    set<int> tupla;
    int suma;

    tupla(){
        tupla = {};
        suma = 0;
    }

    void add(int x){
        suma += x;
        tupla.insert(x);
    }

    void erase(int x){
        suma -= x;
        tupla.erase(x);
    }
};
```

Notamos que las estructuras incluyen el conjunto en cuestión y un campo donde iremos almacenando la suma del conjunto para poder acceder a ella en tiempo **constante**. Además se incluyen funciones para añadir y eliminar numeros del conjunto.

Las **restricciones explícitas** son que para toda tupla $(x_1, x_2, ..., x_k)$ se tiene que $x_i \in \mathbb{N}$ para todo i tal que $0 < i \leqslant k$.

Las **restricciones implícitas** son que para toda tupla $(x_1, x_2, ..., x_k)$ se tiene que $x_i < x_{i+1}$ para todo i tal que $0 < i \leqslant k$ (debe estar ordenada) y además $\sum_{i=1}^{k} x_i = n$.

Procedamos ahora a implementar el algoritmo mediante un TDA (clase) `Solucion`. Tendremos las funciones `esSolucion` para ver si ya hemos alcanzado una solucion, `procesaSolucion` para cuando alcancemos una imprimirla y guardarla, `factible` para ver si un conjunto puede llegar a ser solucion o todos sus caminos acabaran en fracaso. También tenemos `f_cota` que representa la función de cota usada para por `factible`.

En este caso al tener que sacar todas las solucione posibles y deber alcanzar

una suma, tenemos cota inferior (para comprobar si nos vamos a pasar de la suma deseada) como cota local (no tiene sentido una cota superior ya que siempre podemos seguir sumando). La cota global o superior en este caso sería el valor a alcanzar sumando.

```
            class Solucion{
                // Almacenamos las soluciones aqui
                set<tupla> soluciones;
                // Numero cuyas sumas debemos alcanzar
                int n;

                void backtracking(int k,tupla & s){
                    if(esSolucion(s))
                        procesaSolucion(s);

                    // Seleccionamos el siguiente numero para el conjunto
                    // Como los conjuntos deben estar ordenados, 
                    // el siguiente numero debe ser mayor que el anterior
                    // Ademas no puede ser mayor que el n-s.suma
                    // ya que entonces al agregarlo al conjunto
                    // la suma seria mayor que n
                    int tope = n - s.suma;
                    for(int i=k+1; i⩽tope; ++i){
                        // Si la solucion (junto con el nuevo numero) es factible
                        // proseguimos este camino
                        if(factible(i,s)){
                            s.insert(i);
                            backtracking(i,s);
                            s.erase(i);
                        }
                    }
                }

                bool esSolucion(tupla & s){
                    // Si la suma de todos los numeros del conjunto
                    // es igual n tenemos una solucion valida
                    return s.suma == n;
                }

                bool factible(int k,tupla & s){
                    // Sacamos la cota inferior de cual puede ser la suma
                    // y comprobamos que es menor igual que la suma
                    // que queremos alcanzar
                    int cota_inf = f_cota(k,s);
                    return cota_inf ⩽ n;
                }

                int f_cota(int k, tupla & s){
                    // Tenemos que para que una solucion sea factible
                    // al agregarle el valor k la suma del conjunto
                    // debe ser menor o igual que n
                    return k + s.suma;
```

```
48                  }
49
50                  void procesaSolucion(tupla & s){
51                      // Imprimos la solucion y la guardamos
52                      for(auto it = s.begin(); it != s.end(); ++it)
53                          cout << *it << "+"
54                      cout << endl;
55                      soluciones.insert(s);
56                  }
57              };
58
```