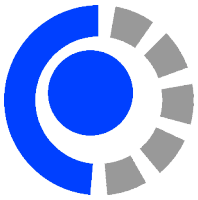
Universidad Nacional del Comahue
Facultad de Informática
Departamento de Programación



Diseño de Algoritmos

Trabajo Práctico 3: técnicas de diseño

Ignacio Alejandro Navarro Oliva

2024

Índice

Introducción **2**

1. Técnicas de diseño **3**
1.1. Algoritmos voraces 3
1.2. Programación dinámica 3
1.3. Divide y vencerás 6

Introducción

En este informe se detallan los ejercicios y su resolución del trabajo práctico 3 de la materia Diseño de Algoritmos, de la Licenciatura en Ciencias de la Computación, carrera dictada en la Universidad del Comahue, Neuquén. Aquellos ejercicios que requieran de código, si bien algunas porciones de este están presentadas en este informe, también se encuentran en un repositorio de GitHub.

1. Técnicas de diseño

1.1. Algoritmos voraces

Considerando el algoritmo de asignaciones estables del trabajo práctico anterior, es posible reconocer componentes que justifican que el algoritmo pertenece a una técnica voraz. Concretamente:

- Conjunto candidatos: N hombres y N mujeres
- Conjunto solución: un conjunto de N asignaciones estables
- Función de factibilidad: asignación de parejas estable
- Función selección: asignar una pareja a un hombre soltero considerando su lista de preferencias, la lista de parejas prohibidas y el caso de que una mujer casada lo prefiera
- Función objetivo: realizar una asignación para todos los individuos (N asignaciones)

1.2. Programación dinámica

La programación dinámica divide un problema en subproblemas que son útiles para la resolución del problema principal. Para el problema del cambio, en el cual dado un billete de valor m se debe devolver m utilizando la menor cantidad de monedas posibles, es útil utilizar un enfoque de programación dinámica.

El siguiente algoritmo emplea programación dinámica para resolver el problema del cambio. Esto es debido a que primero calcula el problema para cada denominación, y luego cada denominación puede utilizar el cálculo de la anterior, para llegar a un resultado óptimo. La resolución de subproblemas se completa en una matriz $N \times m$ en la cual las filas representan cada denominación y las columnas cada costo de 1 a m.

devolverCambio(denominaciones, m)

```java
private int[][] C;

public LinkedList<Integer> devolverCambio(int[] denominaciones, int m) {
    int n = denominaciones.length;
    int INFINITO = Integer.MAX_VALUE / 2;
    C = new int[n][m+1];

    for (int i = 0; i < n; i++) {
        C[i][0] = 0;
    }

    for (int i = 0; i < n; i++) {
        for (int j = 1; j <= m; j++) {
            if (i == 0) {
                if (j < denominaciones[i]) {
                    // no es posible pagar m con monedas
                    C[i][j] = INFINITO;
                } else {
                    // se utiliza al menos 1 moneda
                    C[i][j] = 1 + C[0][j - denominaciones[i]];
                }
            } else {
                if (i - 1 >= 0) {
                    if (j < denominaciones[i]) {
                        // no es posible pagar con esta denominacion
                        // se revisa la denominacion anterior
                        C[i][j] = C[i - 1][j];
                    } else {
                        // se busca el menor de ambos casos
                        // (den anterior o al menos una moneda den[i])
                        C[i][j] =
                        Math.min(C[i - 1][j], 1 + C[i][j - denominaciones[i]]);
                    }
                }
            }
        }
    }

    LinkedList<Integer> S = new LinkedList<>();
    S = buscarSolucion(n-1, m, denominaciones);

    return S;
}
```

Por conveniencia, se utiliza la constante MAX_VALUE (dividida por dos) de la clase Integer, para denotar infinito positivo. Es posible utilizar una clase personalizada para indicar el infinito con algún atributo, o utilizar la clase *Optional* para rellenar con otro elemento que no sea un entero. Para uso real, es improbable que se necesite una aproximación total para billetes con valores grandísimos, por lo que utilizar un número muy grande cumple la tarea.

A través del método *buscarSolucion*, se obtiene una secuencia de monedas cuya suma es igual a m. El algoritmo se observa a continuación:

buscarSolucion(denominaciones, m)

```
public LinkedList<Integer> buscarSolucion(int n, int m, int[] denominaciones) {
    LinkedList<Integer> S = new LinkedList<>();

    while (n >= 0 && m > 0) {
            if (m - denominaciones[n] >= 0
                && C[n][m - denominaciones[n]] != -1
                && (C[n][m] == 1 + C[n][m - denominaciones[n]]
                    || C[n][m - denominaciones[n]] == 0)) {
                // se utilizó una moneda de den[n]
                m = m - denominaciones[n];
                S.add(denominaciones[n]);
            }
            else if (n - 1 >= 0 && C[n][m] == C[n - 1][m]) {
                // se busca den anterior
                n--;
            } else {
                // no hay solucion
                S.clear();
                break;
            }
            if (C[n][m] == C[n - 1][m]
                && C[n][m] == 1 + C[n][m - denominaciones[n]]) {
                // mas de una solucion optima
            }
            if (m == 0) {
                // ya se llego a pagar m
                break;
            }
    }
    return S;
}
```

Se observa que, dependiendo del orden de las estructuras condicionales, podríamos obtener una secuencia distinta (pueden cumplirse los dos primeros casos al mismo tiempo), por lo que es posible adaptar el algoritmo y obtener más de una secuencia posible.

1.3. Divide y vencerás

A través de la partición de un problema en varios subproblemas más pequeño, se pueden crear algoritmos eficientes y comprensibles. El caso más vistoso es la ordenación de arreglos: el método *quickSort* tiene una eficiencia de $O(n\ log\ n)$, así como el método *mergeSort*.

El problema expuesto en este informe es el de construir *convex hulls* (envolventes convexas). Este toma una nube de puntos y retorna una lista de puntos que unidos envuelven a los restantes de la nube.

Tiene diversas soluciones, pero concretamente se observará la solución llamada *quickHull*. Toma un enfoque similar a *quickSort*, de ahí su nombre. Primero toma una recta entre los puntos más lejanos en torno a la coordenada x. Estos puntos son agregados a la lista solución y eliminados de la lista en la que se está trabajando. Luego se separa en segmentos los puntos restantes de acuerdo a su posición respecto de la recta (arriba o debajo). Estos segmentos son procesados recursivamente para calcular el punto más lejano en el segmento a la recta y agregarlo a la lista solución, también quitándolo en la lista que se trabaja. El proceso se repite hasta que el segmento sea vacío. Los segmentos son procesados según si estaban arriba o debajo de la recta (por ejemplo, a partir de un segmento "arriba" solamente se calcularán los segmentos de la misma dirección). Esto es debido a que los demás puntos estarán dentro de la envolvente convexa, por lo que no se deben procesar.

El algoritmo posee el mismo peor caso que *quickSort*: puede ocurrir que la primera segmentación solo produzca un segmento "arriba" o "abajo", por lo que su eficiencia pasa de $O(n\ log\ n)$ a $O(n^2)$.

El algoritmo se define con los siguientes métodos:

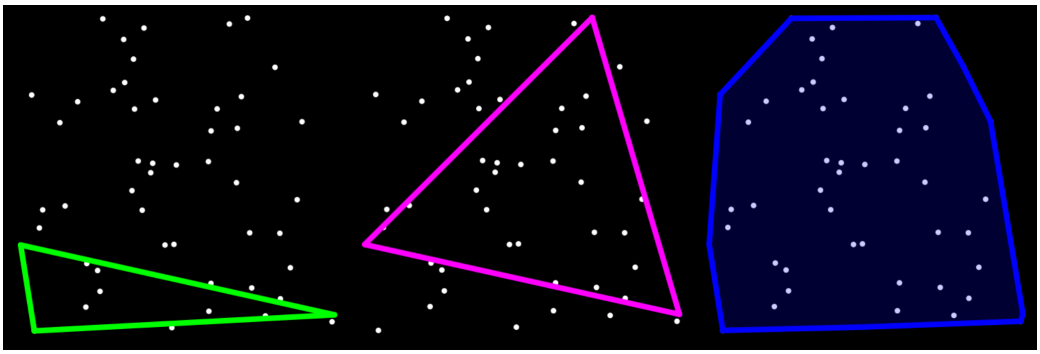
calcularDistancia(P,Q,A)

```
public double calcularDistancia(Point P, Point Q, Point A) {
    // distancia del punto A a la recta formada por PQ
    double rectaX = Q.x - P.x;
    double rectaY = Q.y - P.y;
    double normalX = -rectaY;
    double normalY = rectaX;
    double vectorX = A.x - P.x;
    double vectorY = A.y - P.y;

    return
    Math.abs((vectorX*normalX) + (vectorY*normalY))
    /
    Math.sqrt(normalX*normalX + normalY*normalY);
}
```

encontrarHull(seg,P,Q,tipo)

```
public LinkedList encontrarHull(LinkedList seg, Point P, Point Q, char tipo){
    LinkedList<Point> convexHull = new LinkedList<>();
    if (!seg.isEmpty() && P != null && Q != null) {
        double distanciaMasLejana = -1;
        Point puntoMasLejano = null;
        for (Point point : seg) {
            double distanciaActual = calcularDistancia(P, Q, point);
            if (distanciaActual > distanciaMasLejana) {
                distanciaMasLejana = distanciaActual;
                puntoMasLejano = point;
            }
        }
        convexHull.add(puntoMasLejano);
        seg.remove(puntoMasLejano);
        LinkedList<Point>[] segmentosP = segmentar(seg, P, puntoMasLejano);
        LinkedList<Point>[] segmentosQ = segmentar(seg, puntoMasLejano, Q);
        LinkedList<Point> P_arriba = segmentosP[0];
        LinkedList<Point> P_abajo = segmentosP[1];
        LinkedList<Point> Q_arriba = segmentosQ[0];
        LinkedList<Point> Q_abajo = segmentosQ[1];
        if (tipo == '0') {
            convexHull.addAll(encontrarHull(P_arriba, P, puntoMasLejano, '0'));
            convexHull.addAll(encontrarHull(Q_arriba, puntoMasLejano, Q, '0'));
        } else {
            convexHull.addAll(encontrarHull(P_abajo, P, puntoMasLejano, '1'));
            convexHull.addAll(encontrarHull(Q_abajo, puntoMasLejano, Q, '1'));
        }
    }
    return convexHull;
}
```



(a) Segmento “abajo” (b) Segmento “arriba” (c) Envolvente convexa

Figura 1: Dos primeros segmentos y envolvente resultante

segmentar(S,P,Q)

```
public LinkedList segmentar(LinkedList<Point> S, Point P, Point Q) {
    // calcula segmentos de puntos de S arriba
    // y debajo de la recta formada por PQ
    LinkedList<Point>[] segmentos = new LinkedList[2];
    LinkedList<Point> arriba = new LinkedList<>();
    LinkedList<Point> abajo = new LinkedList<>();
    segmentos[0] = arriba;
    segmentos[1] = abajo;
    if (Q.x - P.x != 0) {
        // no son puntos verticales
        double m = (Q.y - P.y)/(Q.x - P.x);
        double b = -m * P.x + P.y;
        for (Point point : S) {
            if (point.y > m * point.x + b) {
                arriba.add(point);
            } else if (point.y < m * point.x + b) {
                abajo.add(point);
            }
        }
    }
    return segmentos;
}
```

En la figura 1 se observan los dos primeros segmentos formados teniendo en cuenta la primera recta entre los puntos más lejanos entre sí (en la coordenada x), y la envolvente convexa resultante de seguir segmentando. A través del siguiente código en JavaScript se pueden visualizar los segmentos y cómo se construye recursivamente la envolvente convexa.