



**DEPARTAMENTO
DE COMPUTACION**

Facultad de Ciencias Exactas y Naturales - UBA

TP2

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Ezequiel Aguerre	246/07	ezeaguerre@gmail.com
Juan Vanecek	169//10	juann.vanecek@gmail.com
Santiago Camacho	110/09	santicamacho90@gmail.com
Tomas Rodriguez	527/10	tomirodriguez.89@gmail.com



Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

<http://www.fcen.uba.ar>

Índice

1. Problema 1: Robanúmeros	3
1.1. Presentación del problema	3
1.2. Resolución	3
1.3. Pseudocódigo	4
1.4. Casos de Test	5
1.4.1. Elige soluciones intermedias no “óptimas”	5
1.4.2. Elige cartas negativas para aumentar la diferencia de puntos	6
1.4.3. Pierde por lo menos posible	6
1.5. Análisis de complejidad	6
1.6. Test de complejidad	6
1.7. Compilar y ejecutar	7
1.8. Código Fuente	8
2. Problema 2: La centralita (de gas)	9
2.1. Presentación del problema	9
2.2. Resolución	9
2.3. Demostración	9
2.4. Análisis de complejidad	9
2.5. Test de complejidad	9
2.6. Compilar y ejecutar	9
3. Problema 3: Saltos en <i>La Matrix</i>	10
3.1. Presentación del problema	10
3.2. Resolución	10
3.2.1. Algoritmo	10
3.3. Demostración	11
3.4. Análisis de complejidad	11

1. Problema 1: Robanúmeros

1.1. Presentación del problema

El Robanúmeros es un juego de cartas para dos jugadores. Al iniciar se dispone sobre la mesa una secuencia de cartas boca arriba. Cada carta tiene dibujado un número entero (no necesariamente positivo). El Robanúmeros se juega por turnos, alternando un turno para cada jugador. En su turno el jugador debe elegir uno de los dos extremos (izquierdo o derecho) de la secuencia que actualmente están en la mesa y robar cartas comenzando por dicho extremo. Puede robar la cantidad de cartas que quiera pero tiene que robar cartas adyacentes a partir del extremo elegido. Por ejemplo, si las cartas en la mesa son

$$2, -4, 6, 1, 9, -10$$

Entonces el jugador podría robar las cartas 2, -4, 6 empezando por la izquierda por ejemplo o bien las cartas -10, 9, 1, 6 empezando desde la derecha, pero no podría robar las cartas 2, 6, 1 ya que se estaría salteando la carta -4. Tampoco sería posible robar las cartas 6, 1, 9 ya que si bien son adyacentes, la secuencia no se inicia en alguno de los extremos.

En su turno, el jugador debe robar al menos una carta y el juego termina cuando ya no quedan cartas en la mesa. En este momento, cada jugador suma los valores de las cartas que robó y el que obtenga una suma mayor gana el juego.

Mingo y Aníbal son dos jugadores expertos de Robanúmeros y nos retaron a escribir un algoritmo que juegue tan bien como ellos. Es decir, tenemos que escribir un algoritmo que juegue en forma óptima a este juego suponiendo que su contrincante es tan inteligente como él. El programa a implementar debe tomar una secuencia inicial de cartas y debe indicar qué cartas robaría Mingo y Aníbal en cada turno, asumiendo que ambos juegan de manera óptima a este juego. En este contexto, decimos que un jugador juega de manera óptima si la diferencia de puntos obtenida a su favor es la mayor diferencia que se puede obtener frente a un oponente que juega también de manera óptima cada situación que se le deje. En caso de haber más de una solución óptima, el algoritmo puede devolver cualquiera de ellas. Se pide que el algoritmo desarrollado tenga una complejidad temporal de peor caso de $O(n^3)$, donde n es la cantidad de cartas en la secuencia inicial.

1.2. Resolución

Defino la función $f(i, j)$ como la solución óptima usando de las cartas i a j . Esto es, lo máximo que puedo agarrar con las cartas de la izquierda (1*) o de la derecha (2*).

- (1*) Supongamos que por la izquierda lo mejor que puedo hacer es usando las primeras k cartas. Significa que el valor que puedo tomar es $\sum_{t=i}^k carta[t]$ más lo que me deja tomar el otro jugador (que va a jugar de manera óptima) en la mitad $[k+1, j]$. Esto es el total que suma las cartas en dicha mitad, menos $f(k+1, j)$, ya que es el valor óptimo y el puntaje que va a juntar el otro jugador.
- (2*) Asimismo, si lo mejor por derecha es usando k cartas, entonces el puntaje óptimo es la suma del valor de esas cartas ($\sum_{t=k}^n carta[t]$) más lo que puedo tomar del lado $[i, k-1]$ después de que haya jugado el otro jugador. Y como este lo va a hacer óptimamente, el valor que a mí me queda por elegir es $\sum_{t=i}^{k-1} carta[t] - f(i, k-1)$.

Es decir, que f va a buscar el k (entre i y j) tal que maximiza el puntaje por izquierda (1^*), y el de la derecha (2^*), y se va a quedar con el máximo de estos dos.

Matemáticamente, la función queda definida de la siguiente manera (para cuando $i \leq j$):

$$f(i, j) = \max \left(\max_{i \leq k \leq j} \left\{ \sum_{t=i}^k v[t] + \left(\sum_{t=k+1}^j v[t] - f(k+1, j) \right) \right\}, \max_{i \leq k \leq j} \left\{ \sum_{t=k}^j v[t] + \left(\sum_{t=i}^{k-1} v[t] - f(i, k-1) \right) \right\} \right)$$

Para $i > j$ $f(i, j) = 0$. v el vector con los valores de las cartas. Si desarrollamos esta función nos queda la siguiente expresión.

$$\begin{aligned} f(i, j) &= \max \left(\max_{i \leq k \leq j} \left\{ \sum_{t=i}^k v[t] + \left(\sum_{t=k+1}^j v[t] - f(k+1, j) \right) \right\}, \max_{i \leq k \leq j} \left\{ \sum_{t=k}^j v[t] + \left(\sum_{t=i}^{k-1} v[t] - f(i, k-1) \right) \right\} \right) \\ &= \max \left(\max_{i \leq k \leq j} \left\{ \sum_{t=i}^j v[t] - f(k+1, j) \right\}, \max_{i \leq k \leq j} \left\{ \sum_{t=i}^j v[t] - f(i, k-1) \right\} \right) \\ &= \sum_{t=i}^j v[t] + \max \left(\max_{i \leq k \leq j} \{-f(k+1, j)\}, \max_{i \leq k \leq j} \{-f(i, k-1)\} \right) \\ &= \sum_{t=i}^j v[t] - \min \left(\min_{i \leq k \leq j} \{f(k+1, j)\}, \min_{i \leq k \leq j} \{f(i, k-1)\} \right) \\ &= \sum_{t=i}^j v[t] - \min_{i \leq k \leq j} \{\min(f(k+1, j), f(i, k-1))\} \end{aligned}$$

Veamos un ejemplo con dos cartas

$$\begin{aligned} f(1, 2) &= v[1] + v[2] - \min(\min(f(2, 2), f(1, 0)), \min(f(3, 2), f(1, 1))) \\ &= v[1] + v[2] - \min(\min(f(2, 2), 0), \min(0, f(1, 1))) \\ &= v[1] + v[2] - \min\{f(2, 2), f(1, 1), 0\} = \\ &= v[1] + v[2] - \min\{v[2], v[1], 0\} \end{aligned}$$

Si $v[1] \geq 0$ y $v[2] \geq 0$ entonces $v[1] + v[2] - \min\{v[2], v[1], 0\} = v[1] + v[2]$ que es el máximo que se puede obtener. Si $v[1] < 0$ entonces $v[1] + v[2] - \min\{v[2], v[1], 0\} = v[1] + v[2] - v[1] = v[2]$, que es positivo y es el óptimo. Lo mismo para $v[2] < 0$. Si $v[1] < 0$ y $v[2] < 0$ entonces $v[1] + v[2] - \min\{v[2], v[1], 0\} = v[1]$ o $v[2]$, dependiendo cuál sea el máximo, que es, claramente la solución óptima, ya que obtenemos el mejor puntaje posible (agarrar la carta más grande).

1.3. Pseudocódigo

Para resolverlo de manera iterativa y construyendo la solución desde “abajo” se recorre una matriz en diagonal desde la diagonal principal, donde cada elemento (i, j) representa el puntaje obtenido al tomar desde la carta i hasta la j de manera óptima. Al finalizar de construir la matriz el puntaje óptimo se encuentra en la celda $(0, n-1)$. Para construir cada celda (i, j) necesitamos los elementos:

- $(k+1, j) \forall i \leq k < j$
- $(i, k-1) \forall i < k \leq j$

Esto es así porque necesitamos saber qué puntaje obtendría el otro jugador si le dejamos esas cartas, de manera de poder dejarle el menor puntaje posible.

```

calcular( cartas : array )
  n := cantidad de elementos en el arreglo
  cache := array(n*n)
  jugadas := array(n*n)
  para todo 0 <= i < n
    cache[i][i] := cartas[i]
    jugadas[i][i] := Jugada<izquierda,1 carta>

  para 1 <= columna < n
    para 0 <= diagonal < n - columna
      i := diagonal
      j := columna + diagonal
      min_ij := +Infinito
      puntos := 0
      para i <= k <= j
        a := cache[k+1][j] o 0 si k+1 > j
        b := cache[i][k-1] o 0 si k-1 < j
        si b < min_ij o a < min_ij
          si a < b
            jugadas[i][j] = Jugada<izquierda,k-1+1 cartas>
            min_ij = a
          sino
            jugadas[i][j] = Jugada<derecha,j-k+1 cartas>
            min_ij = b
        cache[i][j] = puntos - min_ij
  devolver optimo puntaje := cache[0][n-1]
  devolver lista de jugadas := reconstruir jugadas a partir de jugadas[0][n-1]

```

1.4. Casos de Test

1.4.1. Elige soluciones intermedias no “óptimas”

Probamos con las cartas 1, -10, 2 y 3. Para poder ganar se debe sólo tomar el 3, en lugar del 2 y 3 que nos daría un mayor puntaje parcial.

- Si tomara el 1 el otro elegiría el 2 y el 3 y nos quedaría el -10 y perderíamos 5 a -9.
- Si tomara el 2 y 3 el otro elegiría el 1 y tendríamos luego que tomar el -10 y perderíamos 1 a -5.
- Si toma sólo el 3 el otro puede elegir tanto el 1 como el 2, pues nosotros elegiríamos la otra carta y al oponente sólo le quedaría el -10 y ganaríamos. Por lo tanto el oponente tiene que elegir las 3 cartas que quedan, para maximizar su puntaje.

1.4.2. Elige cartas negativas para aumentar la diferencia de puntos

Probamos con las cartas 10, -10, 2 y 3. La idea es simplemente ver que toma cartas negativas si eso lleva a una solución óptima. En este caso tiene que tomar todas las cartas, si tomara sólo el 3 el oponente podría sumar 2 puntos y ganar sólo por un punto de diferencia, al tomar todas las cartas se asegura ganar por 5 puntos.

- Si tomara el 10 el otro elegiría 2 y 3 y nos quedaría el -10, perdiendo 5 a 0.
- Si tomara el 2 y el 3 el otro elegiría el 10 y nosotros el -10, perdiendo 10 a -5.
- Si tomara 10 y -10 el otro elegiría 2 y 3 y perderíamos 5 a 0.
- Si tomara 2, 3 y -10 perderíamos 10 a -5.
- Si tomara sólo el 3 el otro elegiría lo que queda y llegaría a 2, ganando 3 a 2.
- Si toma todas las cartas se gana 5 a 0.

1.4.3. Pierde por lo menos posible

Probamos con las cartas 1, -10 y 1. Si no se puede ganar, por lo menos se pierda por la menor cantidad de puntos.

- Si tomara sólo cualquiera de los unos, el oponente elegiría el 1 restante y a nosotros nos quedaría el -10, perdiendo 1 a -9 (10 puntos de diferencia)
- Si tomara el 1 y el -10 perdería 1 a -9 (10 puntos de diferencia).
- Si toma todas las cartas pierde 0 a -8 (8 puntos de diferencia).

1.5. Análisis de complejidad

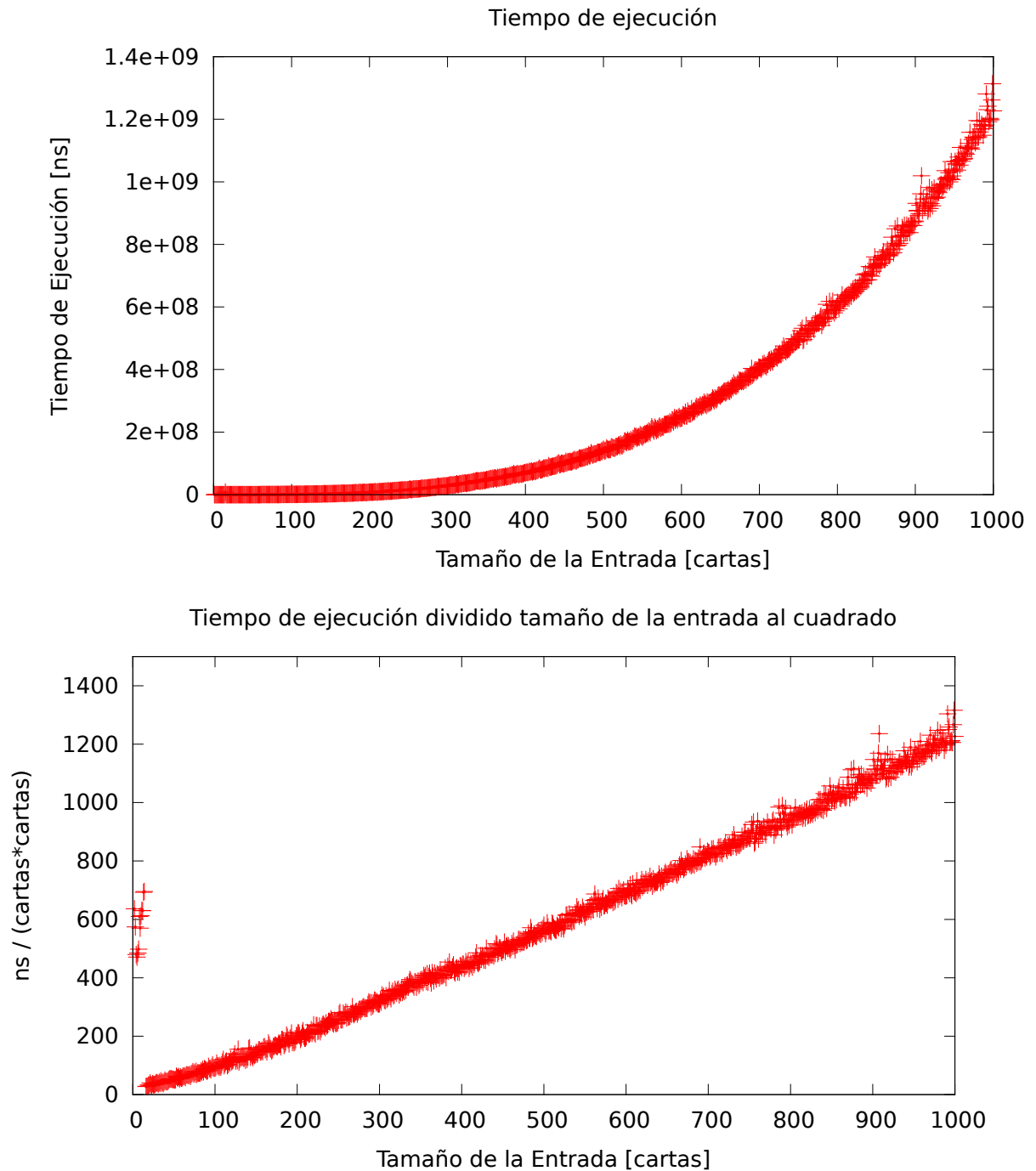
Son simplemente 3 ciclos anidados que como mucho recorren n elementos, el resto de las operaciones es $O(1)$, por ende la complejidad es $O(n^3)$.

1.6. Test de complejidad

Para convencernos que la complejidad es $O(n^3)$ se graficó el tiempo que tarda el algoritmo en resolver distintos problemas.

El tamaño de la entrada es simplemente la cantidad de cartas, ya que es de lo único que depende el algoritmo, pues para una cantidad de cartas determinada siempre va a realizar la misma cantidad de iteraciones.

También graficamos el tiempo de ejecución dividido el tamaño de la entrada al cuadrado, para comprobar que obtenemos una recta.



1.7. Compilar y ejecutar

Desde el directorio src/Ejercicio1:

- **Compilar:** ./ej1 make
- **Ejecutar:** ./ej1
- **Correr casos de test:** ./ej1 tests
- **Benchmarks:** ./ej1 bench

1.8. Código Fuente

```

public void calcularSolucion() {
    int n = cartas.length;
    for ( int i = 0; i < n; i++ ) {
        cache[i][i] = cartas[i];
        jugadas[i][i].desdeIzquierda( 1 );
    }

    for ( int columna = 1; columna < n; columna++ ) {
        for ( int diagonal = 0; diagonal < n - columna; diagonal++ ) {
            int i = diagonal;
            int j = columna + diagonal;
            int min_ij = Integer.MAX_VALUE;
            int puntos = 0;
            int x = i, y = j;
            for ( int k = i; k <= j; k++ ) {
                int a = 0, b = 0;
                puntos += cartas[k];
                if ( k + 1 <= j ) a = cache[k + 1][j];
                if ( k - 1 >= i ) b = cache[i][k - 1];
                if ( min_ij < a && min_ij < b ) continue;
                if ( a < b ) {
                    jugadas[i][j].desdeIzquierda( k - i + 1 );
                    min_ij = a;
                } else {
                    jugadas[i][j].desdeDerecha( j - k + 1 );
                    min_ij = b;
                }
            }
            cache[i][j] = puntos - min_ij;
        }
    }
}

```


2. Problema 2: La centralita (de gas)

2.1. Presentación del problema

2.2. Resolución

2.3. Demostración

2.4. Análisis de complejidad

2.5. Test de complejidad

2.6. Compilar y ejecutar

3. Problema 3: Saltos en *La Matrix*

3.1. Presentación del problema

La Matrix es un juego que consiste de participantes en un tablero cuadrado. Cada casillero contiene un resorte que le permite al jugador saltar a otro casillero en dirección vertical u horizontal dependiendo del valor de salto del resorte.

El problema está en buscar la cantidad mínima de saltos que puede hacer un participante desde un casillero origen a otro destino. Además cada participante tiene una cantidad de potencias extra que puede usar para llegar a casilleros más alejados.

3.2. Resolución

Para resolver este problema se planteó hacer un grafo *tridimensional* (explicado más adelante).

Al plantear el grafo (dirigido) de casilleros como nodos conectados a sus posibles casilleros de saltos hacía fácil la búsqueda de camino mínimo entre origen y destino, pero se complicaba el cálculo del uso de las potencias, ya que era muy costoso saber cuántas potencias se habían usado.

Es por eso que se planteó finalmente uno tridimensional. Este incluye información de las potencias en sí mismo. El grafo dirigido tridimensional consiste en repetir el grafo de casilleros como nodos y conexiones como aristas k cantidad de veces, donde k = número de potencias. Es decir, cuando se toma una decisión que no requiere potencias, me quedo en la misma matriz, pero cuando uso alguna potencia, salto a otra matriz dependiendo de la cantidad de potencias utilizadas. No puedo saltar de una matriz con mayor k a una de menor. Entonces obtengo un grafo dirigido de $(n^2) * k$ nodos, donde una misma posición está repetida k veces.

3.2.1. Algoritmo

Como cada arista representa un salto, todas tienen el mismo costo. Por eso, adaptamos el problema a buscar el camino mínimo con el algoritmo de BFS en un grafo dirigido, empezando desde el casillero origen.

```
LaMatrix(Tablero inicio fin potencia)
  niveles = { inicio : 0 } //diccionario con la clave inicio y 0 como significado
  anterior = { inicio : NULL } //diccionario, clave inicio y NULL como significado
  i = 1
  frontera = [inicio]
  while(frontera){
    siguiente = []
    for x in frontera {
      for y in vecinos(x){ //vecinos es un arreglo del alcance que tiene x
        if !nivel.estaDefinido(y){
          nivel[y] = i
          anterior[y] = x
        }
      }
    }
    frontera = siguiente
    i++
  }
```

```

        siguiente.agregar(y)
    }
}
frontera = siguiente
i++
}

saltosAfin = niveles.significado(fin) //cantidad de saltos
secuenciaDeSaltos = anterior.dameSecuencia(fin) //devuelve el camino desde el inicio
return (saltosAfin, secuenciaDeSaltos)

```

3.3. Demostración

3.4. Análisis de complejidad

El algoritmo es principalmente una búsqueda en anchura, y esta tiene complejidad temporal de $O(V + E)$ donde V son los nodos y E las aristas. En nuestro caso tenemos $(n^2) * k$ nodos y en caso máximo en que todos los nodos esten conectados a sus posibles lugares de salto serian $(n^2) * 2(n - 1) * k$ ya que hay n^2 nodos con $n - 1$ conexiones máximas en cada dirección (vertical u horizontal) por cada nivel k . Esto daría acotado superiormente $(n^3) * k$. Es decir que el peor caso sería de $O((n^2) * k + (n^3) * k)$ que es lo mismo que $O((n^3) * k)$.