UNIVERSIDAD
DE GRANADA

Universidad de Granada

Facultad de Ingeniería Informática y telecomunicaciones

Práctica 2: Divide y vencerás

Doble Grado Ingeniería Informática y Matemáticas

Autores:
Adolfo Martínez Olmedo, Pablo Delgado Galera, Marcos Baena Solar

Marzo 2025

1. Introducción

En esta práctica se desarrolla la resolución de tres problemas diferentes:

- Menor número
- Menor distancia
- Menor envolvente convexa

Lo haremos de dos maneras distintas, primero haremos una aproximación mediante algoritmos de fuerza bruta y después aplicaremos lo aprendido en clase, desarrollando soluciones basadas en divide y vencerás.

2. Objetivos

Los objetivos de esta práctica son los siguientes:

- Comprender la técnica de Divide y Vencerás y sus ventajas.
- Comparar con la estrategia de fuerza bruta y analizar la complejidad.
- Implementar ambos enfoques (fuerza bruta y Divide y Vencerás) para cada problema.
- Experimentar con el umbral de la técnica de Divide y Vencerás.

3. El número más pequeño

3.1. Análisis de Fuerza Bruta

Nuestra idea para desarrollar el algoritmo de fuerza bruta se basa en ir recorriendo el vector de puntos tantas veces como cifras tenga el número que busquemos, y en cada recorrido se busca el número más pequeño, se elimina y se procede a hacer lo mismo en la siguiente iteración.

El código es el siguiente:

```
1   int getSmallestNumber(vector<int> v, int k){
2           int smallestNumber = std::numeric_limits<int>::max();
3           for (int i = 0; i < v.size()-1; ++i){
4                   string num;
5                   num.append(to_string(v[i]));
6                   for (int j = i+1; j-i <= k  && v.size()-1 - j
                        >= k; ++j){
7                           num.append(to_string(v[j]));
8                   }
9                   if(stoi(num) < smallestNumber)
10                          smallestNumber = stoi(num);
11          }
12
13          return smallestNumber;
```

```
14          }
```

Brute Force de **."El número más pequeño"**

Tras el análisis teórico comprobamos que este código tiene una eficiencia de $O(k \cdot n)$.

3.2. Análisis de Divide y Vencerás

Nuestra idea para desarrollar el algoritmo de Divide y Vencerás se basa en orderar el vector utilizando quickSort (para dividir y vencer) y una vez lo tenemos ordenado simplemente cogemos los k primeros números.

El código es el siguiente:

```
1  void quicksort(vector<int> &v, int left, int right)
2  {
3          if (left >= right)
4                  return;
5          int pivot = v[right];
6          int partitionIndex = left;
7          for (int i = left; i < right; ++i)
8          {
9                  if (v[i] < pivot)
10                 {
11                         swap(v[i], v[partitionIndex]);
12                         ++partitionIndex;
13                 }
14         }
15         swap(v[partitionIndex], v[right]);
16         quicksort(v, left, partitionIndex - 1);
17         quicksort(v, partitionIndex + 1, right);
18 }
19
20 int getSmallestNumber(vector<int> v, int k)
21 {
22         quicksort(v, 0, v.size() - 1);
23         string result = "";
24         for (int i = 0; i < k; ++i)
25         {
26                 result += to_string(v[i]);
27         }
28         return stoi(result);
29 }
```

Divide y vencerás de **."El número más pequeño"**

Tras el análisis teórico comprobamos que este código tiene una eficiencia de $O(n \cdot \log(n))$

3.3. Configuración de las Pruebas

- Descripción del entorno de ejecución (CPU, memoria, compilador, etc.).
- Conjunto de datos utilizados para las pruebas (tamaño, forma de generarlos).

3.4. Tablas y Gráficos de Rendimiento

Insertar aquí las tablas y/o gráficos que muestren los tiempos de ejecución, uso de memoria, etc.

3.5. Análisis de los Resultados

Tras el analisis teórico y visualizar la gráfica comparativa, vemos que el algoritmo de Fuerza Bruta consigue resolver el problema de forma más eficiente que el algoritmo Divide y Vencerás. Esto se debe a que el primer algortimo depende de una constante k que consideramos siempre menor que n, y por tanto no afecta a la eficiencia. Mientras que en el segundo algoritmo podemos apreciar que quicksort tiene la eficiencia del peor caso, ya que al solo usar dígitos del 1 al 9, encontramos en el vector una gran cantidad de números repetidos lo cual afecta gravemente a la eficiencia de este algortimo de ordenación.

- Comparación cualitativa (efectividad, facilidad de implementación).
- Comparación cuantitativa (tiempos de ejecución, consumo de memoria).
- Conclusiones sobre el umbral experimental.

4. El par de puntos más cercano

4.1. Análisis de Fuerza Bruta

Nuestra idea para desarrollar el algoritmo de fuerza bruta se basa en un doble bucle for el cual hace todas las distancias de todos los puntos con todos, hasta encontrar la mínima distancia.

El código es el siguiente:

```
1  struct Point {
2          double x, y;
3  };
4
5  double dist(Point p1, Point p2) {
6          return sqrt( (p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y
               - p2.y) );
7  }
8
9
10 double bruteForce(const vector<Point>& P, int l, int r) {
11         double minDist = DBL_MAX;
12         for (int i = l; i <= r; i++) {
13                 for (int j = i+1; j <= r; j++) {
14                         double d = dist(P[i], P[j]);
15                         if (d < minDist) {
16                                 minDist = d;
17                         }
18                 }
```

```
        }
        return minDist;
}
```

Brute Force de **."El par de puntos más cercano"**

Tras el análisis teórico comprobamos que este código tiene una eficiencia de $O(n^2)$.

4.2. Análisis de Divide y Vencerás

Nuestra idea para desarrollar el algoritmo de Divide y Vencerás se basa en ir haciendo divisiones de un array de puntos previamente ordenados por la coordenada x de los puntos (de derecha a izquierda). Tras esto, vamos haciendo divisiones partiendo el vector por la mitad recursivamente hasta quedarnos con 3 puntos o menos en una franja, donde se resolverá el problema usando fuerza bruta. Así obtenemos la mínima distancia en una franja, tras esto nos quedamos con la mínima distancia entre la franja de la izquierda y la derecha, y la usaremos para estudiar la mínima distancia entre una franja intermedia de tamaño la mínima distancia, creamos esta franja con los puntos que estan a una distancia menor de la menor distancia previamente calculada respecto del punto central. Una vez construido el strip comparamos los puntos con respecto a su coordenada y (esta comparación se hace con un número comnstante de puntos al reducirlos con la distancia mínima), buscando que sea menor que la menor distancia previa, obtenemos así la mínima distancia entre la franja izquierda, derecha y central. Resolviendo recursivamente obtenemos la mínima distancia en un conjunto de puntos.

El código es el siguiente:

```
        void swap(Point* a, Point* b) {
                Point t = *a;
                *a = *b;
                *b = t;
        }

        int partition(vector<Point>& arr, int low, int high) {
                double pivot = arr[high].x;
                int i = (low - 1);

                for (int j = low; j <= high - 1; j++) {
                        if (arr[j].x < pivot) {
                                i++;
                                swap(&arr[i], &arr[j]);
                        }
                }
                swap(&arr[i + 1], &arr[high]);
                return (i + 1);
        }

        void quickSort(vector<Point>& arr, int low, int high) {
                if (low < high) {
                        int pi = partition(arr, low, high);

                        quickSort(arr, low, pi - 1);
```

```
                        quickSort(arr, pi + 1, high);
                }
        }

        void sort_by_x(vector<Point>& points) {
                quickSort(points, 0, points.size() - 1);
        }

        double bruteForce(const vector<Point>& P, int l, int r) {
                double minDist = DBL_MAX;
                for (int i = l; i <= r; i++) {
                        for (int j = i+1; j <= r; j++) {
                                double d = dist(P[i], P[j]);
                                if (d < minDist) {
                                        minDist = d;
                                }
                        }
                }
                return minDist;
        }


        double closestUtil(vector<Point> &PX, int l, int r) {
                // Si el subrango es peque, fuerza bruta
                if (r - l <= 3) {
                        return bruteForce(PX, l, r);
                }

                int mid = (l + r) / 2;
                double midX = PX[mid].x;

                // bestPair se va actualizando;
                // para tener la minima distancia, llamo recursivamente
                    a izq y der
                double dl = closestUtil(PX, l, mid);
                double dr = closestUtil(PX, mid+1, r);

                double d = dl < dr ? dl : dr;
                // Construir el strip (puntos que quedan en [midX - d,
                   midX + d])
                vector<Point> strip;
                strip.reserve(r - l + 1);
                for (int i = l; i <= r; i++) {
                        if (fabs(PX[i].x - midX) < d) {
                                strip.push_back(PX[i]);
                        }
                }
                // Ordenar la franja por Y
                sort(strip.begin(), strip.end(), [](auto &a, auto &b){
```

```
73                          return a.y < b.y;
74                  });
75
76                  // Revisar cada punto del strip con sus siguientes
77                  // mientras la diferencia en Y sea < d
78                  for (int i = 0; i < (int)strip.size(); i++) {
79                          for (int j = i+1; j < (int)strip.size() && (
                               strip[j].y - strip[i].y) < d; j++) {
80                                  double distStrip = dist(strip[i], strip
                                       [j]);
81                                  if (distStrip < d) {
82                                          d = distStrip;
83                                  }
84                          }
85                  }
86
87                  return d;
88          }
```

Divide y vencerás de **."El par de puntos más cercano"**

Para el análisis teórico de este algoritmo debemos estudiar la ecuación de recurrencia: $T(n) = 2T(\frac{n}{2}) + O(n)$. Esta ecuación se obtiene de que una llamada a la función de tamaño n hará dos llamadas recursivas con cada mitad de los puntos (izquierda y derecha), O(n) se obtiene de calcular los puntos en el strip central, que auqnue pueda parecer $O(n)$, cada punto del strip solo se compara con un número constante de puntos, al habernos quedado previamnete con una cantidad de puntos pequeñas dentro de un rango. Cuando el tamaño es menor o igual que 3 (caso base), donde aplicamos fuerza bruta, al ser un tamaño tan pequeño, no afectará a la eficiencia.

Tras resolver la recurrencia comprobamos que este código tiene una eficiencia de $O(n \cdot \log(n))$.

4.3. Detalles de Implementación

4.4. Configuración de las Pruebas

- Descripción del entorno de ejecución (CPU, memoria, compilador, etc.).
- Conjunto de datos utilizados para las pruebas (tamaño, forma de generarlos).

4.5. Tablas y Gráficos de Rendimiento

Insertar aquí las tablas y/o gráficos que muestren los tiempos de ejecución, uso de memoria, etc.

4.6. Análisis de los Resultados

Como se esperaba tras el análisis teórico, en las gráficas podemos observar que el algoritmo Divide y Vencerás tiene una mayor eficiencia que el de Fuerza Bruta, también podemos observar

la clara forma parábolica del algoritmo Fuerza Bruta como habiamos previsto. También se puede observar la región a partir del cual el primer algortimo, comienza a tener tiempos de ejecución más bajos que el segundo.

- Comparación cualitativa (efectividad, facilidad de implementación).
- Comparación cuantitativa (tiempos de ejecución, consumo de memoria).
- Conclusiones sobre el umbral experimental.

5. La envolvente convexa

5.1. Análisis de Fuerza Bruta

Nuestra idea para desarrollar el algoritmo de fuerza bruta se basa en empezar seleccionando el punto más a la izquierda, el cual sabemos que siempre va a estar en la envolvente convexa, seguidamente seleccionas otro punto y compruebas si la recta que une los dos puntos deja el resto de puntos a la izquierda, una vez encuentras ese punto, es nuevo punto actúa como el primero que seleccionaste, y haces esto hasta que encuentres todos los puntos pertenecientes a la envolvente convexa.

El código es el siguiente:

```
1   // Estructura para representar un punto
2   struct Point {
3       float x, y;
4   };
5
6   typedef pair<int, int> ParPuntos;
7
8   // Funcion que calcula el convex hull por fuerza bruta
9   set<ParPuntos> convexHullBruteForce(const vector<Point>& points
        ) {
10      int n = points.size();
11      set<ParPuntos> hull;
12
13      // Recorremos cada par de puntos
14      for (int i = 0; i < n; i++) {
15          for (int j = i + 1; j < n; j++) {
16              int left = 0, right = 0;
17
18              // Verificamos la posicion de cada otro
                    punto respecto a la linea formada
                   por points[i] y points[j]
19              for (int k = 0; k < n; k++) {
20                  if (k == i || k == j)
21                      continue;
22
23                  float position = (points[j].x -
                        points[i].x) * (points[k].y
                        - points[i].y) -
```

```
                                                (
                                                    points[j].y - points[i].y) * (points[k].x - points[i].x);

                if (position > 0)
                        left++;
                else if (position < 0)
                        right++;
        }

        // Si todos los puntos se encuentran de un mismo lado (o son colineales), se adjuntan los puntos de la arista
        if (left == 0 || right == 0) {
                hull.insert({points[i].x, points[i].y});
                hull.insert({points[j].x, points[j].y});
```

```
                                    }
                                }
                            }

                            return hull;
                        }
```

Brute Force de **"La envolvente convexa"**

Tras el análisis teórico comprobamos que este código tiene una eficiencia de $O(n^3)$

5.2. Análisis de Divide y Vencerás

Nuestra idea para desarrollar el algoritmo de Divide y Vencerás se basa en ir diviendo los puntos del plano, previamente ordenados de derecha a izquierda, en dos partes (izquierda y derecha) hasta llegar a un tamaño pequeño donde se calculen las envolventes convexas, tanto de izquierda como de derecha que se juntarán usando la función merge. Esta función comienza buscando de entre los puntos de la envolvente de la derecha el punto más a la izquierda, actuando analogamente con los de la izquierda, tras esto buscamos la tangente superior que cuyos puntos formarán parte de la envolvente. Para ello, vamos buscando con el punto más a la derecha de la izquierda fijo, y rotando el punto más a la izquierda de los derecha, ahcia la derecha hasta encontrar uno que deje el resto de puntos a la derecha, una vez lo encontramos fijamos ese punto y rotamos hacia la izquierda los puntos de la izquierda hasta encontrar uno que deje todos los puntos de la izquierda a la izquierda. Repetimos para la tangente inferior rotando en sentido contrario, finalizamos guardando todoos los puntos resultantes en el vector a devolver.s

código

Para el análisis teórico de este algoritmo debemos estudiar la ecuación de recurrencia: $T(n) = 2T(\frac{n}{2}) + O(n)$. Esta ecuación se obtiene de que una llamada a la función de tamaño n hará dos llamadas recursivas con cada mitad de los puntos (izquierda y derecha), O(n) se obtiene de la llamada a la función merger que junta las dos envolventes con una eficiencia de O(n), ya que hay se recorren varios bucles de la mitad del tamaño en cada iteración ($\frac{n}{2}$) obviando la constante obtenemos O(n). El caso base usa la fuerza bruta en un conjunto de 5 puntos, su eficiencia es presecindible.

Tras resolver la recurrencia comprobamos que este código tiene una eficiencia de $O(n \cdot \log(n))$.

5.3. Detalles de Implementación

5.4. Configuración de las Pruebas

- Descripción del entorno de ejecución (CPU, memoria, compilador, etc.).
- Conjunto de datos utilizados para las pruebas (tamaño, forma de generarlos).

5.5. Tablas y Gráficos de Rendimiento

Insertar aquí las tablas y/o gráficos que muestren los tiempos de ejecución, uso de memoria, etc.

5.6. Análisis de los Resultados

Como se esperaba tras el análisis teórico, en las gráficas podemos observar que el algoritmo Divide y Vencerás tiene una mayor eficiencia que el de Fuerza Bruta, también podemos observar la clara forma cúbica del algoritmo Fuerza Bruta como habiamos previsto. También se puede observar la región a partir del cual el primer algortimo, comienza a tener tiempos de ejecución más bajos que el segundo.

- Comparación cualitativa (efectividad, facilidad de implementación).
- Comparación cuantitativa (tiempos de ejecución, consumo de memoria).
- Conclusiones sobre el umbral experimental.

En esta sección se describen los diferentes problemas que se abordarán.

6. Conclusiones

Tras esta práctica hemos descubierto que la estrategia de desarrollo de algoritmos Divide y Vencerás es útil en un amplio repertorio de casos pero esto no quiere decir que siempre sea mejor que un algoritmo fuerza bruta, esto lo podemos comprobar en 3. Aún así en la gran mayoría de casos aplicar Divide y Vencerás supone una mejoría en la eficiencia del código.