



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
7                  >= k; ++j){
8                  num.append(to_string(v[j]));
9              }
10             if(stoi(num) < smallestNumber)
11                 smallestNumber = stoi(num);
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 ordenar 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 análisis 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 algoritmo 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 algoritmo 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
7         - p2.y) );
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         }
19     }
```

```

19     }
20     return minDist;
21 }

```

Brute Force de ^{E1} 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 (Pablo Indio complétalo).

El código es el siguiente:

```

1     void swap(Point* a, Point* b) {
2         Point t = *a;
3         *a = *b;
4         *b = t;
5     }
6
7     int partition(vector<Point>& arr, int low, int high) {
8         double pivot = arr[high].x;
9         int i = (low - 1);
10
11         for (int j = low; j <= high - 1; j++) {
12             if (arr[j].x < pivot) {
13                 i++;
14                 swap(&arr[i], &arr[j]);
15             }
16         }
17         swap(&arr[i + 1], &arr[high]);
18         return (i + 1);
19     }
20
21     void quickSort(vector<Point>& arr, int low, int high) {
22         if (low < high) {
23             int pi = partition(arr, low, high);
24
25             quickSort(arr, low, pi - 1);
26             quickSort(arr, pi + 1, high);
27         }
28     }
29
30     void sort_by_x(vector<Point>& points) {
31         quickSort(points, 0, points.size() - 1);
32     }
33
34     double bruteForce(const vector<Point>& P, int l, int r) {
35         double minDist = DBL_MAX;
36         for (int i = l; i <= r; i++) {

```

```

37         for (int j = i+1; j <= r; j++) {
38             double d = dist(P[i], P[j]);
39             if (d < minDist) {
40                 minDist = d;
41             }
42         }
43     }
44     return minDist;
45 }

46
47
48 double closestUtil(vector<Point> &PX, int l, int r) {
49     // Si el subrango es peque, fuerza bruta
50     if (r - l <= 3) {
51         return bruteForce(PX, l, r);
52     }
53
54     int mid = (l + r) / 2;
55     double midX = PX[mid].x;
56
57     // bestPair se va actualizando;
58     // para tener la minima distancia, llamo recursivamente
59     // a izq y der
60     double dl = closestUtil(PX, l, mid);
61     double dr = closestUtil(PX, mid+1, r);
62
63     double d = dl < dr ? dl : dr;
64     // Construir el strip (puntos que quedan en [midX - d,
65     // midX + d])
66     vector<Point> strip;
67     strip.reserve(r - l + 1);
68     for (int i = l; i <= r; i++) {
69         if (fabs(PX[i].x - midX) < d) {
70             strip.push_back(PX[i]);
71         }
72     }
73     // Ordenar la franja por Y
74     sort(strip.begin(), strip.end(), [](auto &a, auto &b){
75         return a.y < b.y;
76     });
77
78     // Revisar cada punto del strip con sus siguientes
79     // mientras la diferencia en Y sea < d
80     for (int i = 0; i < (int)strip.size(); i++) {
81         for (int j = i+1; j < (int)strip.size() && (
82             strip[j].y - strip[i].y) < d; j++) {
83             double distStrip = dist(strip[i], strip
84                 [j]);
85             if (distStrip < d) {

```

```

82                                     d = distStrip;
83                                 }
84                            }
85                        }
86
87                    return d;
88                }

```

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

Tras el análisis teórico comprobamos que este código tiene una eficiencia de (PABLO INDIO HAZLO)

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

- 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 sabes 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      ) {
11          int n = points.size();
12          set<ParPuntos> hull;
13
14          // Recorremos cada par de puntos
15          for (int i = 0; i < n; i++) {
16              for (int j = i + 1; j < n; j++) {
17                  int left = 0, right = 0;
18
19                  // Verificamos la posicion de cada otro
20                  // punto respecto a la linea formada
21                  // por points[i] y points[j]
22                  for (int k = 0; k < n; k++) {
23                      if (k == i || k == j)
24                          continue;
25
26                      float position = (points[j].x -
27                                      points[i].x) * (points[k].y
28                                      - points[i].y) -
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```



```

25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
-
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 (pablo)

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

- 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 **??**. Aún así en la gran mayoría de casos aplicar Divide y Vencerás supone una mejoría en la eficiencia del código.