Algorítmica

Grado en Ingeniería Informática

Práctica 3

Formas de sumar n

Autores

María Jesús López Salmerón

Nazaret Román Guerrero

Laura Hernández Muñoz

José Baena Cobos

Carlos Sánchez Páez

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Curso 2017-2018

Índice

1. Descripción de la práctica **1**

2. Algoritmos implementados **1**
2.1. Algoritmo de fuerza bruta 1
2.2. Algoritmos backtracking 1
2.2.1. Algoritmo backtracking sin información 1
2.2.2. Algoritmo backtracking con información 1

3. Análisis de eficiencia **2**

4. Conclusiones **5**

5. Anexo: código fuente **5**

Índice de figuras

1. Tamaños utilizados para la ejecución 2
2. Fuerza bruta 2
3. Backtracking sin información 3
4. Backtracking con información 3
5. Comparativas entre algoritmos 4
6. Comparativa entre algoritmos backtracking 4
7. Fuerza bruta 8
8. Backtracking sin información 11
9. Backtracking con información 15
10. Ejecución de la práctica 15
11. Ejecución individual 15
12. Generación de gráficas 16

1. Descripción de la práctica

El objetivo de esta práctica es calcular todas las maneras posibles de sumar un número n utilizando los elementos pertenecientes a un conjunto $C = \{1, ..., n\}$.

2. Algoritmos implementados

Hemos implementado tres algortimos diferentes: uno con fuerza bruta y dos con bactracking. Estos dos últimos se diferencian en la manera de almacenar la información referente a los cálculos llevados a cabo.

2.1. Algoritmo de fuerza bruta

En este algoritmo evaluamos todas las posibles combinaciones de números, comenzando por el primer elemento del conjunto; este primer elemento se va sumando con los elementos sucesivos, comprobando si dicha suma alcanza nuestro objetivo n.

2.2. Algoritmos backtracking

- **Solución parcial**: tupla de tamaño fijo que contiene el valor 1 en el caso de que el elemento correspondiente a dicha posición se encuentre dentro de la solución y 0 en otro caso.
- **Restricciones explícitas**: el conjunto debe estar ordenado en orden no decreciente.
- **Restricciones implícitas**: la suma resultante de cada tupla debe ser igual a n y no debe haber dos elementos repetidos.

2.2.1. Algoritmo backtracking sin información

- **Función de factibilidad**: se comprueba si al añadir el siguente elemento a la suma no sobrepasamos n, si al sumar los elementos que ya tenemos y los restantes somos capaces de llegar a n y por último si la solución parcial es, en efecto, una solución.

2.2.2. Algoritmo backtracking con información

- **Función de factibilidad**: es la misma que la del caso anterior con una diferencia: en esta versión la suma actual y la suma de los elementos restantes se almacenan en variables de forma que no haya que calcularlas en cada iteración. No obstante, hay que tener una precaución: debemos resetear ambas variables cada vez que encontramos una solución.

3. Análisis de eficiencia

	Tamaño inicial	Tamaño final	Número de ejecuciones
Fuerza bruta Backtracking sin información Backtracking con información	1	25	24

Figura 1: Tamaños utilizados para la ejecución

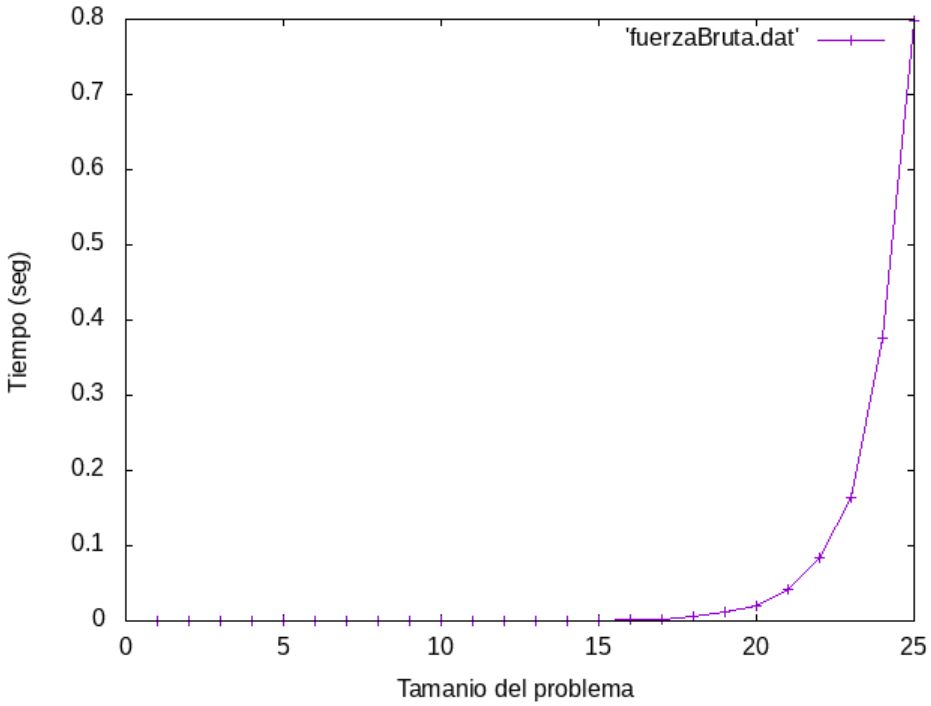

Figura 2: Fuerza bruta

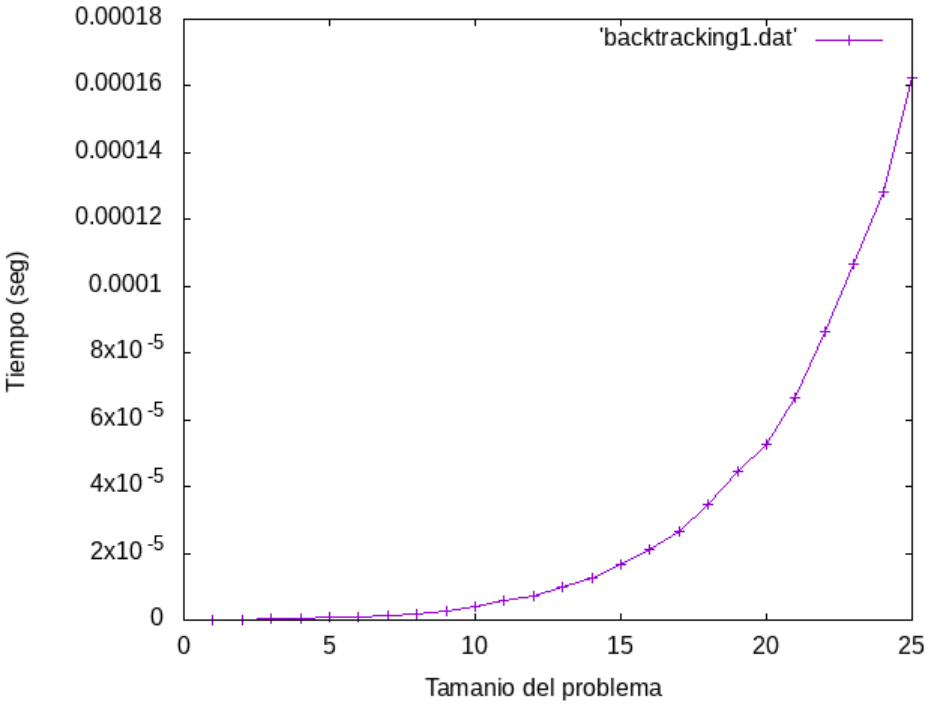

Figura 3: Backtracking sin información

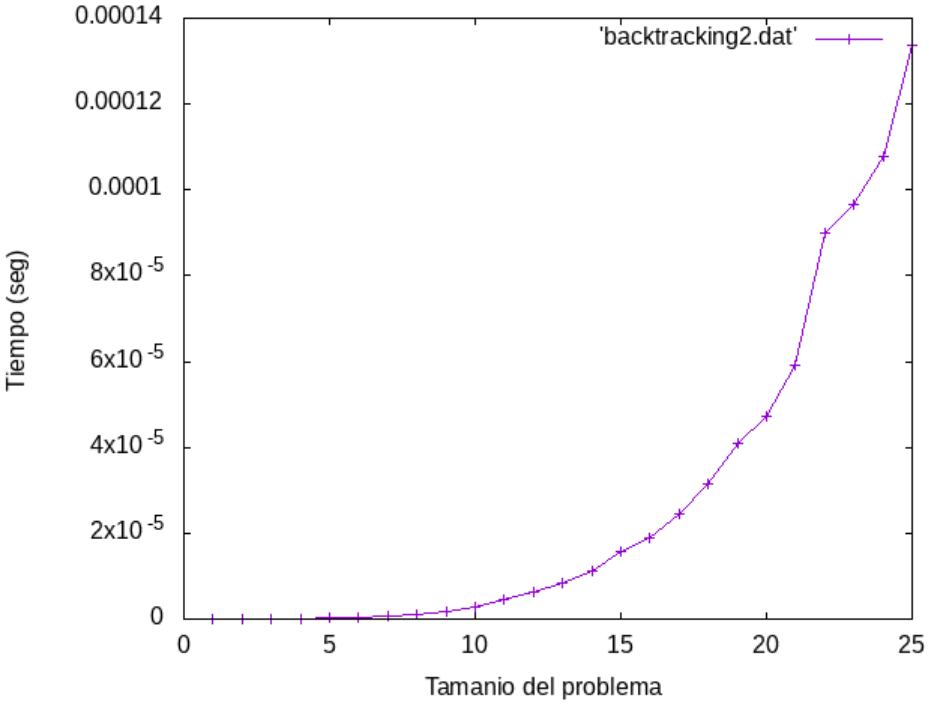

Figura 4: Backtracking con información

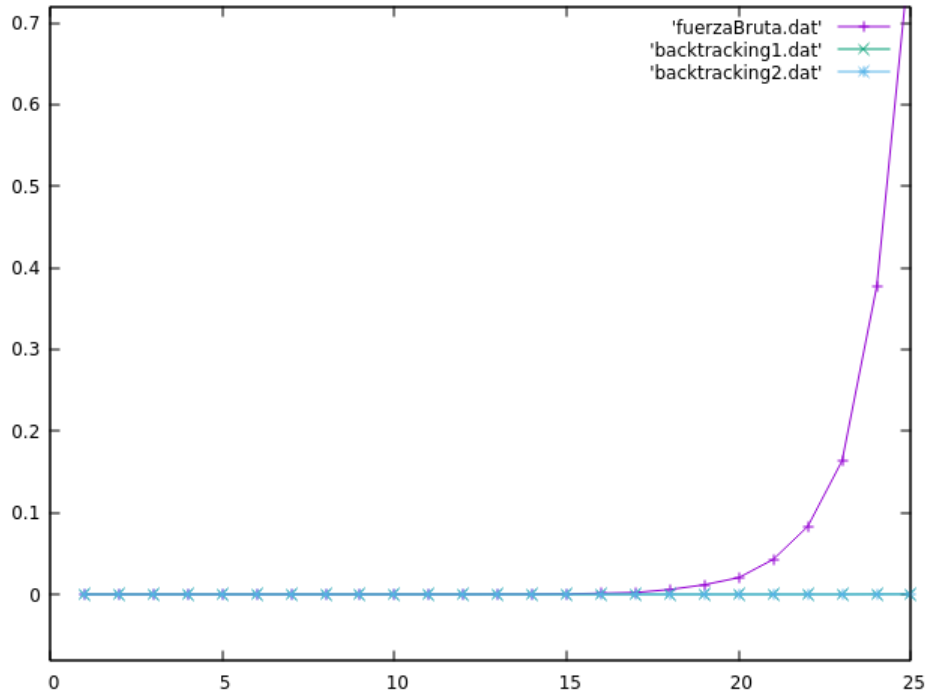

Figura 5: Comparativas entre algoritmos

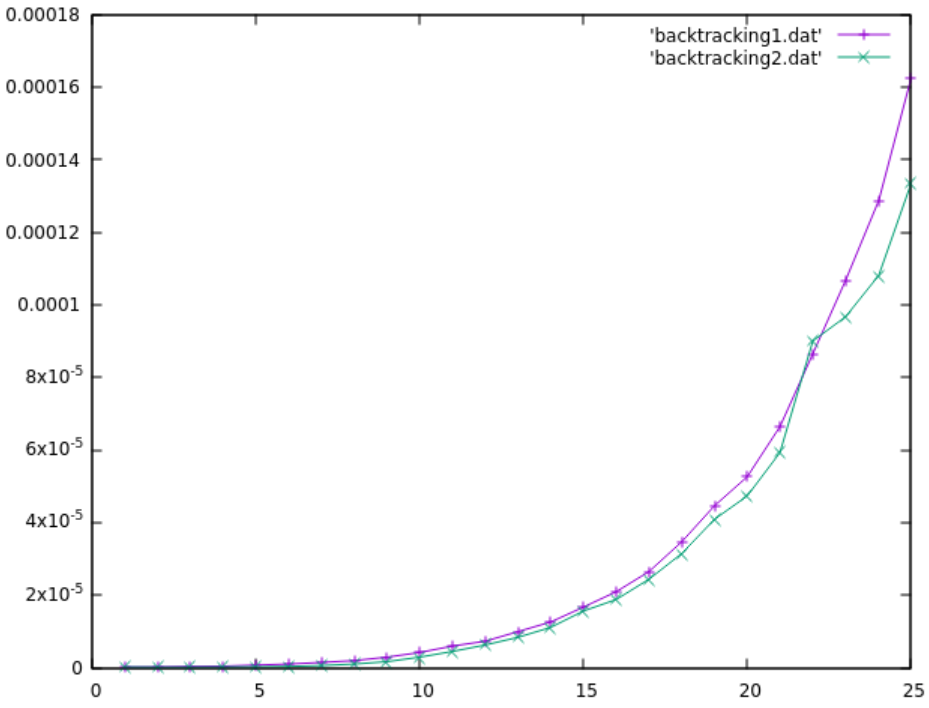

Figura 6: Comparativa entre algoritmos backtracking

4. Conclusiones

Como hemos podido observar en las figuras anteriores, el algortimo fuerza bruta es nefasto. La eficiencia de este algoritmo es factorial lo cual da lugar a tiempos muy elevados.

Sin embargo con backtracking, conseguimos órdenes de eficiencia mucho menores por el simple hecho de realizar una comprobación de factibilidad.

Pero podemos afinar aún más; si almacenamos los datos que se evalúan en la función de factibilidad evitamos tener que calcularlos en cada iteración, transformando una función de orden de eficiencia lineal en una función de orden constante.

5. Anexo: código fuente

```
#include <chrono>
#include <iostream>
#include <list>
#include <stdlib.h>
#include <vector>

using namespace std;
using namespace chrono;

#define NULO 2
#define END -1
#define DEBUG 0

list<vector<int>> lista;

class Solucion {
private:
  vector<int> tuplas;
  vector<int> v;
  int M;
  int GetSuma() const;

public:
  Solucion(int tam, int num_m);
  ~Solucion();
  int size() const;
  int getM() const;
  int getTupla(int i) const;
  bool TodosGenerados(int i) const;
  void InicializaElemento(int i);
  void DecrementaElemento(int i);
  void Aniadir();
  bool SolucionEncontrada();
};

```

```
36 Solucion::Solucion(int tam, int num_m) {
37   tuplas.resize(tam);
38   v.resize(tam);
39   M = num_m;
40 
41   for (int i = 0; i < tam; i++)
42     v[i] = i + 1;
43 }
44 
45 Solucion::~Solucion(){};
46 
47 int Solucion::size() const { return tuplas.size(); }
48 
49 int Solucion::getM() const { return M; }
50 
51 int Solucion::getTupla(int i) const { return tuplas[i]; }
52 
53 bool Solucion::TodosGenerados(int i) const { return tuplas[i] == END; }
54 
55 void Solucion::InicializaElemento(int i) { tuplas[i] = NULO; }
56 
57 void Solucion::DecrementaElemento(int i) { tuplas[i]--; }
58 
59 bool Solucion::SolucionEncontrada() { return GetSuma() == M; }
60 int Solucion::GetSuma() const {
61   int sum_aux = 0, i = 0;
62   while (i < tuplas.size()) {
63     sum_aux += tuplas[i] * v[i];
64     i++;
65   }
66   return sum_aux;
67 }
68 
69 void Solucion::Aniadir() { lista.push_back(tuplas); }
70 
71 void FuerzaBruta(Solucion &sol, int i) {
72   if (i == sol.size()) {          // Si he llegado al final
73     if (sol.SolucionEncontrada()) // Si es solución
74       sol.Aniadir();
75   }
76 
77   else {                        // No he llegado al final
78     sol.InicializaElemento(i); // Pongo tuplas[i]=NULO
79     sol.DecrementaElemento(i); // tuplas[i]--
80 
81     while (!sol.TodosGenerados(
82         i)) { // Mientas no llegue al final (no haya generado todo)
```

```
83        FuerzaBruta(sol, i + 1); // Llamo recursiruamente con el próximo
          ↪ elemento
84        sol.DecrementaElemento(i); // tuplas[i]-- (END)
85      }
86    }
87  }
88
89  ostream &operator<<(ostream &f, const vector<int> &v) {
90    for (auto it = v.begin(); it != v.end(); ++it)
91      f << *it << " ";
92    return f;
93  }
94
95  ostream &operator<<(ostream &f, const list<vector<int>> &lista) {
96    for (auto it = lista.begin(); it != lista.end(); ++it)
97      f << *it << endl;
98    return f;
99  }
100 /*****************************************************************************/
101
102 int main(int argc, char **argv) {
103   lista.clear();
104
105   if (argc != 2) {
106     cerr << "Uso: " << argv[0] << " <número>" << endl << endl;
107     exit(1);
108   }
109
110   int num = atoi(argv[1]);
111
112   Solucion sol(num, num);
113
114   high_resolution_clock::time_point tantes =
      ↪ high_resolution_clock::now();
115
116   FuerzaBruta(sol, 0);
117
118   high_resolution_clock::time_point tdespues =
      ↪ high_resolution_clock::now();
119   duration<double> total = duration_cast<duration<double>>(tdespues -
      ↪ tantes);
120
121   cout << num << "\t" << total.count() << endl;
122
123 #if DEBUG
124   cout << "Soluciones:" << endl;
125   cout << lista << endl;
126 #endif
```

```
127
128   return 0;
129 }
```

Figura 7: Fuerza bruta

```
1  #include <chrono>
2  #include <iostream>
3  #include <list>
4  #include <stdlib.h>
5  #include <vector>
6
7  using namespace std;
8  using namespace chrono;
9
10 ostream &operator<<(ostream &f, const vector<int> &v) {
11   for (auto it = v.begin(); it != v.end(); ++it)
12     f << *it << " ";
13   return f;
14 }
15
16 ostream &operator<<(ostream &f, const list<vector<int>> &lista) {
17   for (auto it = lista.begin(); it != lista.end(); ++it)
18     f << *it << endl;
19   return f;
20 }
21
22 #define NULO 2
23 #define END -1
24 #define DEBUG 0
25
26 list<vector<int>> lista;
27
28 class Solucion {
29 private:
30   vector<int> tuplas;
31   vector<int> v;
32   int M;
33   int GetSuma() const;
34
35 public:
36   Solucion(int tam, int num_m);
37   ~Solucion();
38   int size() const;
39   int getM() const;
40   int getTupla(int i) const;
41   bool TodosGenerados(int i) const;
42   void InicializaElemento(int i);
```

```
43    void DecrementaElemento(int i);
44    void Aniadir();
45    bool SolucionEncontrada();
46    bool Factibilidad(int indice);
47  };
48
49  Solucion::Solucion(int tam, int num_m) {
50    tuplas.resize(tam);
51    v.resize(tam);
52    M = num_m;
53
54    for (int i = 0; i < tam; i++)
55      v[i] = i + 1;
56  }
57
58  Solucion::~Solucion(){};
59
60  int Solucion::size() const { return tuplas.size(); }
61
62  int Solucion::getM() const { return M; }
63
64  int Solucion::getTupla(int i) const { return tuplas[i]; }
65
66  bool Solucion::TodosGenerados(int i) const { return tuplas[i] == END; }
67
68  void Solucion::InicializaElemento(int i) { tuplas[i] = NULO; }
69
70  void Solucion::DecrementaElemento(int i) { tuplas[i]--; }
71
72  bool Solucion::SolucionEncontrada() { return GetSuma() == M; }
73  int Solucion::GetSuma() const {
74    int sum_aux = 0;
75    for (int i = 0; i < tuplas.size(); i++)
76      sum_aux += tuplas[i] * v[i];
77
78    return sum_aux;
79  }
80
81  bool Solucion::Factibilidad(int indice) {
82    int suma_restante = 0;
83    int suma_parcial = 0;
84
85    for (int i = 0; i <= indice; i++)
86      suma_parcial += tuplas[i] * v[i];
87
88    for (int i = indice + 1; i < v.size(); i++)
89      suma_restante += v[i];
90
```

```
91    return ((suma_parcial + v[indice + 1] <= M) &&
92            (suma_parcial + suma_restante >= M)) ||
93           (suma_parcial == M);
94  }
95
96
97  void Solucion::Aniadir() { lista.push_back(tuplas); }
98
99
100 void Backtracking_sin_info(Solucion &sol, int i) {
101   if (i != sol.size()) {       // Si no nos hemos pasado
102     sol.InicializaElemento(i); // Pongo tuplas[i]=NULO
103     sol.DecrementaElemento(i); // tuplas[i]--
104
105     while (!sol.TodosGenerados(i)) { // Mientas no llegue al final y sea
        ↪ factible
106       if (sol.Factibilidad(i))
107         Backtracking_sin_info(
108             sol, i + 1); // Llamo recursirvamente con el próximo
              ↪ elemento
109       sol.DecrementaElemento(i); // tuplas[i]-- (END)
110     }
111   } else {
112     sol.Aniadir();
113   }
114 }
115
116 /*****************************************************************************/
117
118 int main(int argc, char **argv) {
119   lista.clear();
120
121   if (argc != 2) {
122     cerr << "Uso: " << argv[0] << " <número>" << endl << endl;
123     exit(1);
124   }
125
126   int num = atoi(argv[1]);
127
128   Solucion sol(num, num);
129   int veces = 100;
130   double media = 0.0;
131
132   for (int i = 0; i < veces; i++) {
133     high_resolution_clock::time_point tantes =
        ↪ high_resolution_clock::now();
134
135     Backtracking_sin_info(sol, 0);
```

```
136
137     high_resolution_clock::time_point tdespues =
        ↪  high_resolution_clock::now();
138     duration<double> total = duration_cast<duration<double>>(tdespues -
        ↪  tantes);
139     media += total.count();
140   }
141   media /= veces;
142
143   cout << num << "\t" << media << endl;
144
145 #if DEBUG
146   cout << "Soluciones:" << endl;
147   cout << lista << endl;
148 #endif
149
150   return 0;
151 }
```

Figura 8: Backtracking sin información

```
1  #include <chrono>
2  #include <iostream>
3  #include <list>
4  #include <stdlib.h>
5  #include <vector>
6
7  using namespace std;
8  using namespace chrono;
9
10 #define NULO 2
11 #define END -1
12 #define DEBUG 0
13
14 list<vector<int>> lista;
15
16 class Solucion {
17 private:
18   vector<int> tuplas;
19   vector<int> v;
20   int M;
21   int suma_parcial;
22   int suma_restantes;
23   int GetSuma() const;
24
25 public:
26   Solucion(int tam, int num_m);
27   ~Solucion();
```

```
28    int size() const;
29    int getM() const;
30    int getTupla(int i) const;
31    bool TodosGenerados(int i) const;
32    void InicializaElemento(int i);
33    void DecrementaElemento(int i);
34    void Aniadir();
35    bool SolucionEncontrada();
36    bool Factibilidad(int i);
37    void reset(int i);
38  };
39
40  Solucion::Solucion(int tam, int num_m) {
41    tuplas.resize(tam);
42    v.resize(tam);
43    M = num_m;
44    suma_parcial = 0;
45    suma_restantes = 0;
46
47    for (int i = 0; i < tam; i++) {
48      v[i] = i + 1;
49      suma_restantes += v[i];
50    }
51  }
52
53  Solucion::~Solucion(){};
54
55  int Solucion::size() const { return tuplas.size(); }
56
57  int Solucion::getM() const { return M; }
58
59  int Solucion::getTupla(int i) const { return tuplas[i]; }
60
61  bool Solucion::TodosGenerados(int i) const { return tuplas[i] == END; }
62
63  void Solucion::InicializaElemento(int i) { tuplas[i] = NULO; }
64
65  void Solucion::DecrementaElemento(int i) {
66    tuplas[i]--;
67    if (tuplas[i] == 1) { // Si el elemento vale 1 (aún aparece)
68      suma_parcial += v[i];
69      suma_restantes -= v[i];
70    } else if (tuplas[i] == 0) // Si vale 0 (no aparece)
71      suma_parcial -= v[i];
72  }
73
74  bool Solucion::SolucionEncontrada() { return GetSuma() == M; }
75  int Solucion::GetSuma() const {
```

```
76    int sum_aux = 0, i = 0;
77    while (i < tuplas.size()) {
78      sum_aux += tuplas[i] * v[i];
79      i++;
80    }
81    return sum_aux;
82  }
83
84  bool Solucion::Factibilidad(int i) {
85    return ((suma_parcial + v[i + 1] <= M) &&
86            (suma_parcial + suma_restantes >= M)) ||
87           (suma_parcial == M);
88  }
89
90  void Solucion::Aniadir() { lista.push_back(tuplas); }
91
92  void Solucion::reset(int i) {
93    if (i != tuplas.size()) {
94      suma_restantes += v[i];
95      tuplas[i] = NULO;
96    }
97  }
98
99
100 void Backtracking_info(Solucion &sol, int i) {
101   if (i != sol.size()) {       // Si no nos hemos pasado
102     sol.InicializaElemento(i); // Pongo tuplas[i]=NULO
103     sol.DecrementaElemento(i); // tuplas[i]--
104     while (!sol.TodosGenerados(i)) {
105       if (sol.Factibilidad(i)) {
106         Backtracking_info(
107             sol, i + 1); // Llamo recursivamente con el próximo
                ↪  elemento
108         sol.reset(i + 1);
109       }
110       sol.DecrementaElemento(i); // tuplas[i]-- (END)
111     }
112   } else
113     sol.Aniadir();
114 }
115
116 ostream &operator<<(ostream &f, const vector<int> &v) {
117   for (auto it = v.begin(); it != v.end(); ++it)
118     f << *it << " ";
119   return f;
120 }
121
122 ostream &operator<<(ostream &f, const list<vector<int>> &lista) {
```

```
123   for (auto it = lista.begin(); it != lista.end(); ++it)
124     f << *it << endl;
125   return f;
126 }
127 /*****************************************************************************/
128
129 int main(int argc, char **argv) {
130
131   if (argc != 2) {
132     cerr << "Uso: " << argv[0] << " <número>" << endl << endl;
133     exit(1);
134   }
135
136   int num = atoi(argv[1]);
137
138   Solucion sol(num, num);
139   int veces = 50;
140   double media = 0.0;
141
142   for (int i = 0; i < 50; i++) {
143
144     high_resolution_clock::time_point tantes =
      ↪  high_resolution_clock::now();
145
146     Backtracking_info(sol, 0);
147
148     high_resolution_clock::time_point tdespues =
      ↪  high_resolution_clock::now();
149     duration<double> total = duration_cast<duration<double>>(tdespues -
      ↪  tantes);
150     media += total.count();
151   }
152   media /= veces;
153
154   cout << num << "\t" << media << endl;
155
156 #if DEBUG
157   cout << "Soluciones:" << endl;
158   cout << lista << endl;
159 #endif
160
161   return 0;
162 }
```

Figura 9: Backtracking con información

```
1 #!/bin/bash
2 echo "Compilando..."
```

```
g++ -o fuerzaBruta fuerzaBruta.cpp -O3
g++ -o backtracking1 backtracking1.cpp -O3
g++ -o backtracking2 backtracking2.cpp -O3
echo "Ejecutando fuerza bruta..."
./individual.sh fuerzaBruta 1 1
echo "Ejecutando backtracking sin información..."
./individual.sh backtracking1 1 1
echo "Ejecutando backtracking con información..."
./individual.sh backtracking2 1 1
echo "Generando gráficas..."
./gnuplot.sh
```

Figura 10: Ejecución de la práctica

```
#!/bin/bash
if [ $# -eq 3 ]
then
i="0"
tam=$2
#Primer argumento: programa a ejecutar
#Segundo argumento: tamaño inicial
#Tercer argumento : incremento
echo "" > $1.dat
while [ $i -lt 25 ]
do
  ./$1 $tam >> ./$1.dat
  i=$[$i+1]
  tam=$[$tam+$3]
done
else
echo "Error de argumentos"
fi
```

Figura 11: Ejecución individual

```
#!/usr/bin/gnuplot

set xlabel "Tamanio del problema"
set ylabel "Tiempo (seg)"
set terminal png size 640,480
set output 'fuerzaBruta.png'
plot 'fuerzaBruta.dat' with linespoints
set output 'backtracking1.png'
plot 'backtracking1.dat' with linespoints
set output 'backtracking2.png'
plot 'backtracking2.dat' with linespoints
```

Figura 12: Generación de gráficas