universidade de aveiro

deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

Algoritmos e Estruturas de Dados

Speed Run

Universidade de Aveiro

Licenciatura em Engenharia de Computadores e Informática

Francisco Ribeiro, Nº 107993, 33%

Guilherme Vieira, Nº 108671, 33%

Miguel Vila, Nº 107276, 33%

8 de dezembro de 2022, Aveiro

Índice

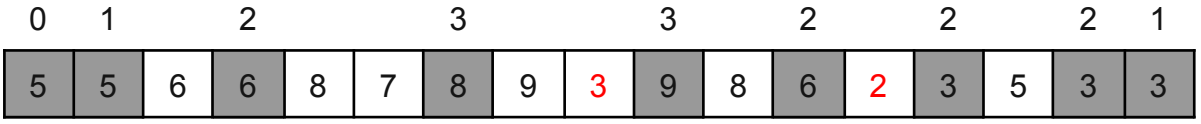
Introdução

O objetivo deste trabalho prático foi implementar uma função capaz de determinar a melhor velocidade a que um veículo deve circular, respeitando a velocidade máxima nos segmentos de uma estrada, fazendo com que este chegue ao final no menor número de movimentos possíveis.

De uma forma mais pormenorizada, este desafio consiste numa estrada, dividida em segmentos, com uma velocidade máxima permitida em cada um deles [0,9]. Esta velocidade consiste no número de segmentos que o carro percorre num único movimento.

Um movimento é definido como o ato de mover o carro para um segmento à frente do atual, a uma velocidade inferior (-1), igual (0) ou superior (+1) à velocidade a que atualmente se encontra. Um movimento é válido se a velocidade a que o carro se move respeitar velocidade máxima permitida no segmento para o qual se move mas também respeitar as velocidades máximas permitidas nos segmentos entre a posição atual e a posição para a qual se move.

À medida que o veículo se aproxima do final da rua, a sua velocidade terá de ir diminuindo gradualmente de modo a que a velocidade no último segmento seja 1.



Também é de notar que, ainda que o carro inicie a sua jornada com velocidade 0, durante o percurso terá que estar sempre em movimento, ou seja, a velocidade nunca pode ser menor ou igual a 0.

Abordagens

Todas as nossas soluções foram implementadas em *C* utilizando o método *Brute Force*. A única diferença entre as soluções são pequenas otimizações que melhoraram de forma significativa o tempo de execução e, em alguns casos, diminuíram o *effort* necessário para encontrar a solução.

O método de *Brute Force* é dos mais utilizados na área da computação, tendo em conta que é uma técnica que soluciona problemas mais simples do ponto de vista do programador, enumerando todas as soluções possíveis e verificando se cada uma satisfaz as condições definidas pelo problema.

Brute Force - Solução 1

Este código, fornecido pelos professores, é bastante ineficiente. A forma como procura soluções é a seguinte:

- Começa no primeiro espaço com velocidade 0;
- Para cada espaço, vai calcular a velocidade seguinte, começando por tentar diminuir a velocidade, depois manter a velocidade e finalmente aumentar a velocidade.

Esta solução é ineficiente porque, à medida que vai encontrando possíveis soluções, continua à procura de outras possíveis soluções, mesmo que nesse ramo o número de movimentos dados seja superior ao número de movimentos de outra solução que já tenha encontrado. Isto torna o código ineficiente visto que o que se pretende é procurar a solução com o menor número de movimentos.

Outro fator que contribui para a ineficiência desta abordagem é o facto de que se pretende encontrar a solução que utiliza menos movimentos. Logo, em vez de começar a procurar soluções nas velocidades mais baixas (isto é, em vez de começar a diminuir a velocidade), deveria começar por procurar possíveis soluções nas velocidades mais altas. Por outras palavras, o algoritmo deveria começar por tentar aumentar a velocidade, depois manter a velocidade e por fim diminuí-la.

Brute Force - Solução 2

Nesta primeira tentativa de otimização do código fornecido no enunciado, tentou-se otimizar o primeiro problema referido anteriormente: caso o algoritmo esteja a procurar a solução num ramo e esse mesmo ramo exceda o número de movimentos de uma solução encontrada previamente, vai descartar esse ramo de procura.

Desta forma, conseguiu-se evitar que o código demore tanto tempo a procurar soluções que à partida já não são as melhores, tendo em conta que o principal objetivo é encontrar a solução com o menor número de movimentos possível.

Mais uma vez, como foi dito anteriormente, esta solução continua a ter um problema que a torna ineficiente: para encontrar a solução que utiliza menos movimentos com menor esforço, deve-se começar por procurar soluções nas velocidades mais altas, em seguida na mesma velocidade e, por fim, nas velocidades mais baixas.

Brute Force - Solução 3

Esta tentativa de otimização pretende corrigir o erro encontrado nas duas soluções anteriores: começar a procurar possíveis soluções nas velocidades mais altas, em vez de começar pelas velocidades mais baixas. Outra diferença desta solução para as anteriores é que mal seja encontrada uma solução válida naquele ramo, todos os outros ramos são descartados.

Por mais que esta solução seja mais eficiente em termos de tempo comparando com as outras soluções, um dos problemas desta solução é que não conseguimos garantir que a solução encontrada seja a que tenha um menor número de movimentos, tendo em conta que só consideramos a primeira solução encontrada.

Brute Force - Solução 4

Ao contrário das soluções anteriores, esta solução não é uma otimização do código já fornecido. Esta solução explora um método *brute force* com a utilização de dívidas e de uma lista de velocidades proibidas.

De modo a ser mais percetível, vai-se enumerar as operações mais relevantes que são executadas no código:

1. Na função principal *solution_4_recursion*, são passados 2 argumentos adicionais, estes sendo o *debt_pos* e o *debt*, ou seja, a posição em que a dívida aconteceu e a dívida atual.
    1.1. Este sistema de dívidas serve para controlar a velocidade a ser testada. Se a *debt* for 0, a *speed* é incrementada; se *debt* for -1, a *speed* é mantida; se debt for -2, a *speed* é decrementada.

    ```
    new_speed = speed + 1 + (position == debt_pos ? debt : 0);
    ```

    Quando o algoritmo não consegue progredir, a variável *debt* atinge valores inferiores a -2, assim, este sistema de dívidas tem ainda outra função: recuar para a posição anterior para testar novas velocidades.
2. A cada chamada da função recursiva, a prioridade tentar será aumentar a velocidade em cada movimento. Não sendo possível, tentará mantê-la. Caso ainda não seja possível, tentará diminuí-la.
3. Após definir a velocidade a testar, será feita uma validação do limite de velocidade para as posições a seguir à posição atual. Por exemplo, caso a nova velocidade seja 3, vai verificar as próximas 3 posições. Caso o limite de velocidade seja excedido em alguma destas posições, ou a velocidade já foi previamente testada nessa posição, a função recursiva será chamada novamente, mas desta vez com um valor de *debt* decrementado (para, dependendo do valor manter ou decrementar a velocidade ou recuar uma posição) e o valor de *debt_pos* definido para a posição atual onde estava já a testar.
4. Caso a velocidade seja válida e passe pelas verificações anteriores, irá salvar o valor da velocidade atual, e qual velocidade foi utilizada

(aumentar, manter ou diminuir) na *solution_4_info*, e depois irá chamar a *solution_4_recursion* com os novos valores de velocidade e posição, mas com a dívida e a posição da dívida a 0.

Dynamic Programming - Solução 5

Tal como a anterior, esta solução não se trata de uma otimização do código já fornecido, mas sim de um repensar da forma como o problema poderia ser resolvido.

Enquanto as soluções anteriores optam por uma abordagem recursiva, esta resolve o problema recorrendo a uma função iterativa com capacidade de memorizar segmentos já calculados em chamadas anteriores, fazendo assim uso de uma técnica de programação dinâmica, diminuindo o *effort* e o tempo de execução.

Por forma a entender melhor a forma como esta abordagem funciona, vamos enumerar em geral as operações que esta função realiza para resolver o problema:

1. Após validar a posição final, a função *solve_5* obtém os valores da solução anterior (mesmo não havendo solução anterior) e chama a função principal *solution_5_dynamic* responsável por gerar e validar novas soluções.
2. Esta função começa a gerar mais soluções a partir da última posição válida salva. Em seguida tenta as soluções possíveis pela seguinte ordem: velocidade atual + 1, velocidade atual + 0 e velocidade - 1.
3. Para cada uma das velocidades, valida o valor da velocidade para a posição atual e para as próximas posições. Caso a solução com a velocidade incrementada não seja válida, repete este ponto mantendo a velocidade. Se ainda assim esta é inválida, decrementa a velocidade e repete este ponto.
4. No caso de uma tentativa de velocidade ser válida (respeitar a velocidade das posições) e levar a posição atual além da posição final ou ser válida fazendo o carro chegar à posição final com velocidade 1, a solução é salva de modo a poder ser reutilizada numa futura chamada.

Resultados

Para cada uma das soluções, registámos os tempos de execução para tamanhos diferentes do problema e usámos o MATLAB para criar gráficos desses tempos de execução.

Todas as soluções foram executadas no mesmo computador, com as seguintes especificações:

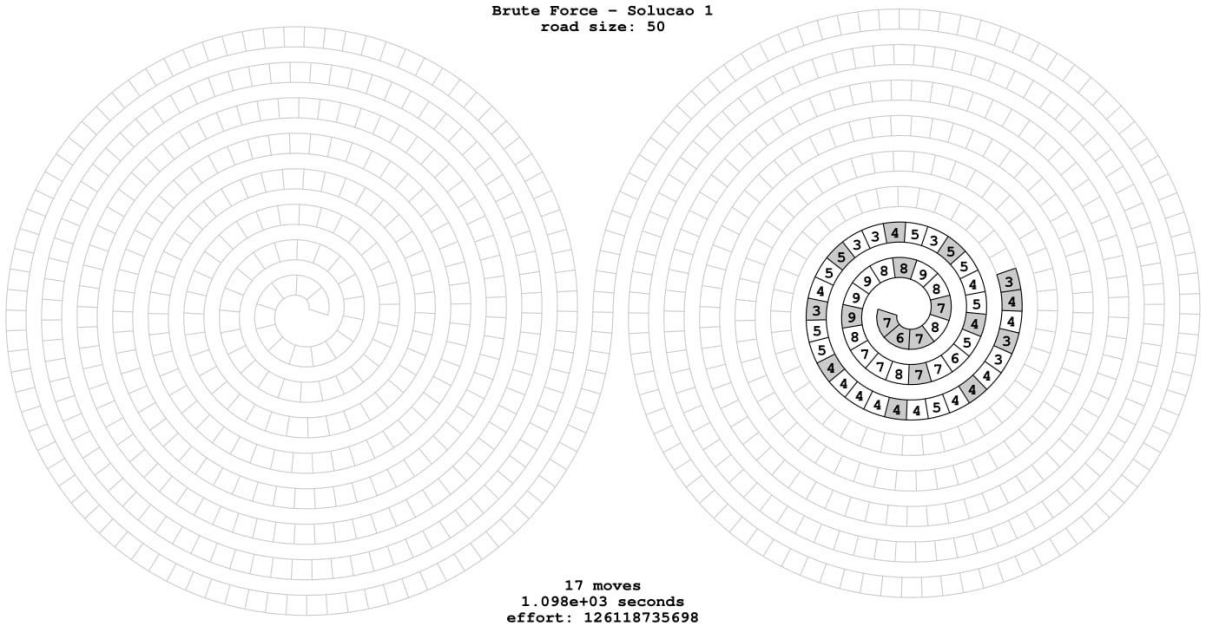
- Processador: Intel® Core™ i7-7700HQ @ 2.80GHz
- RAM: 16GB @ 2400MHz

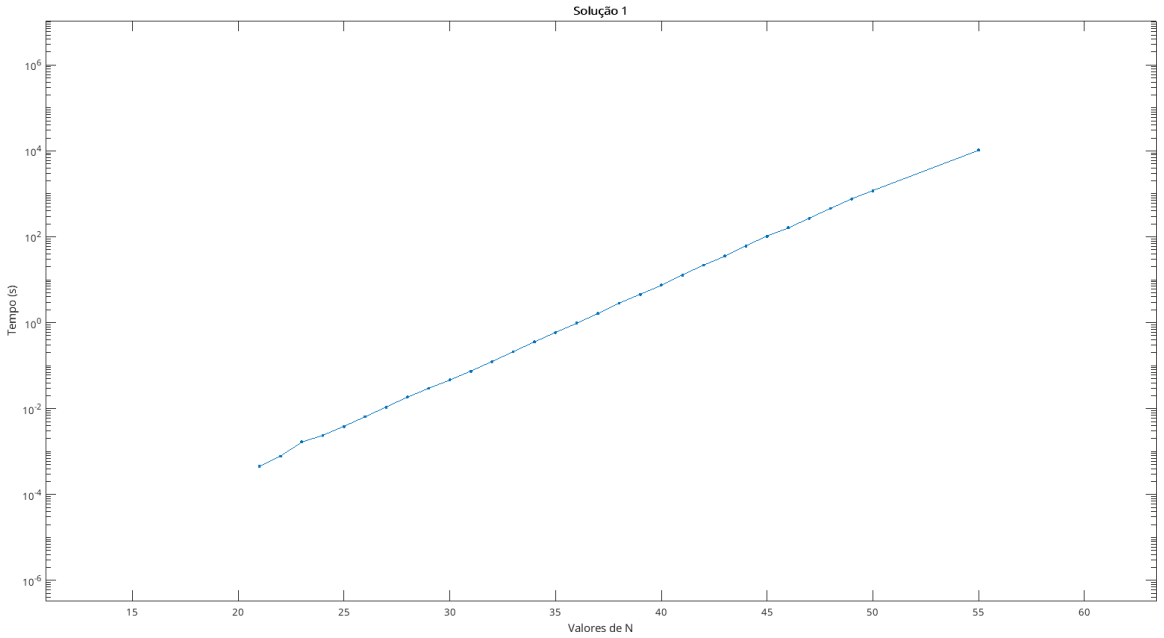
A seguir vamos mostrar os resultados obtidos por cada solução, assim como um gráfico dos tempos que cada solução demorou a ser executada para os diferentes tamanhos.

Todas as soluções foram executadas com o mesmo número mecanográfico a servir como semente (Nº 107276), por motivos de consistência.

Como o código tem um limite de tempo, para as soluções 1 e 2 apenas vamos colocar os resultados até um tamanho limite de 50, tendo em conta que estas soluções são as mais lentas, e iriam demorar bastante tempo a ser executadas por completo. Para cada uma destas soluções, vamos apresentar uma reta de ajuste, assim como os cálculos efetuados para chegar a uma estimativa do tempo de execução para o tamanho 800.

Solução 1



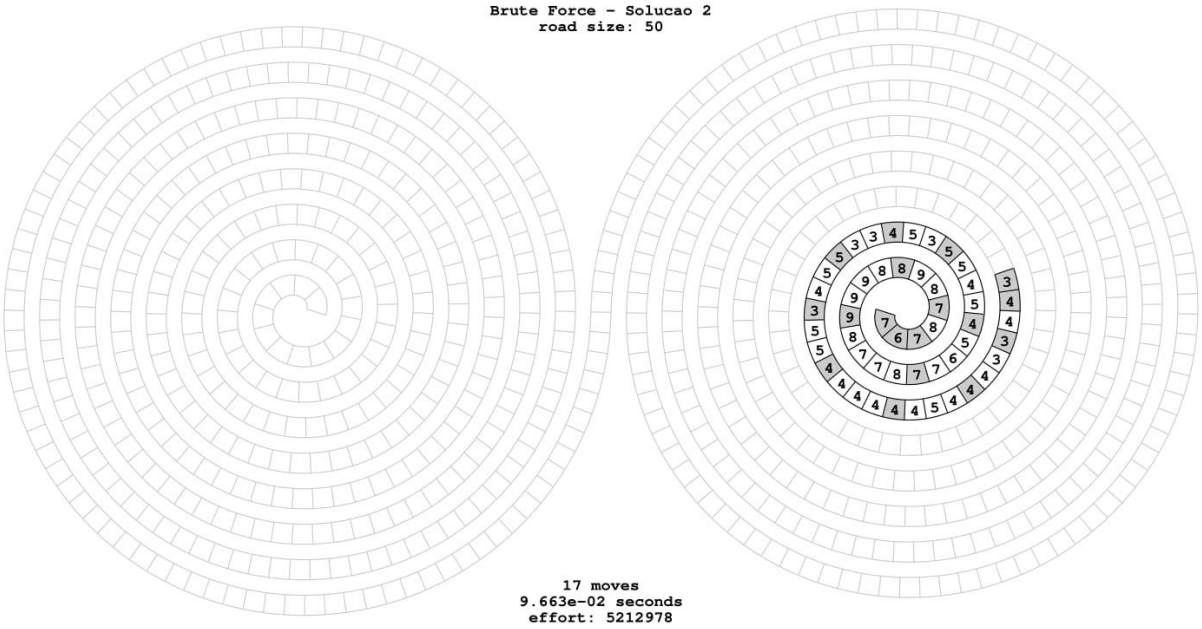


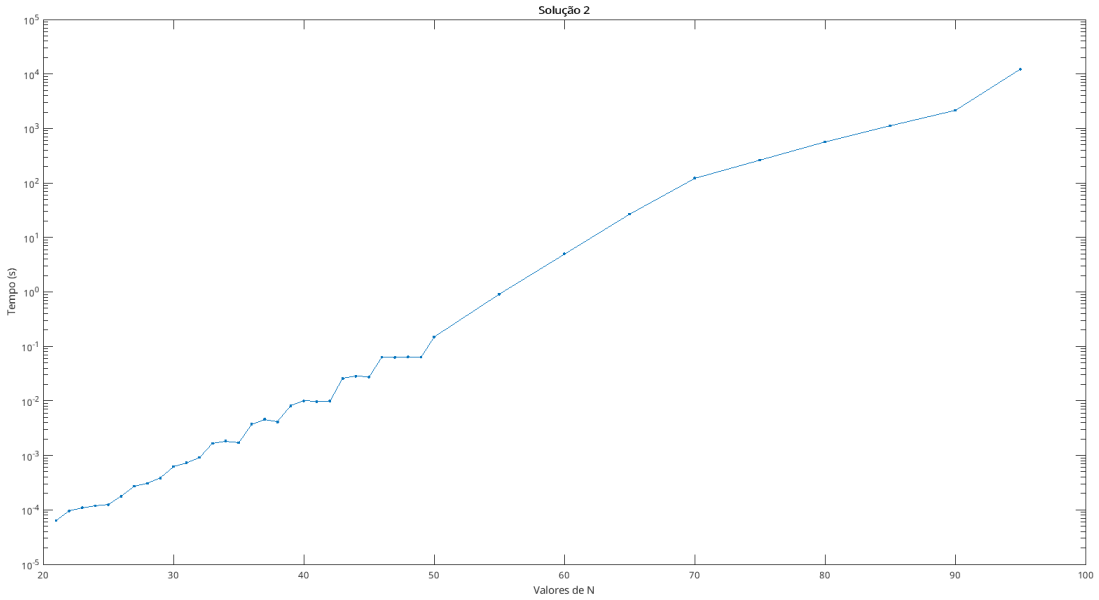
Reta de ajuste (valores de N como x, log10 dos valores de tempo como y):

y = 0.21918x + -7.8907

Estimativa de tempo para n = 800: 2.8413e+167 segundos

Solução 2



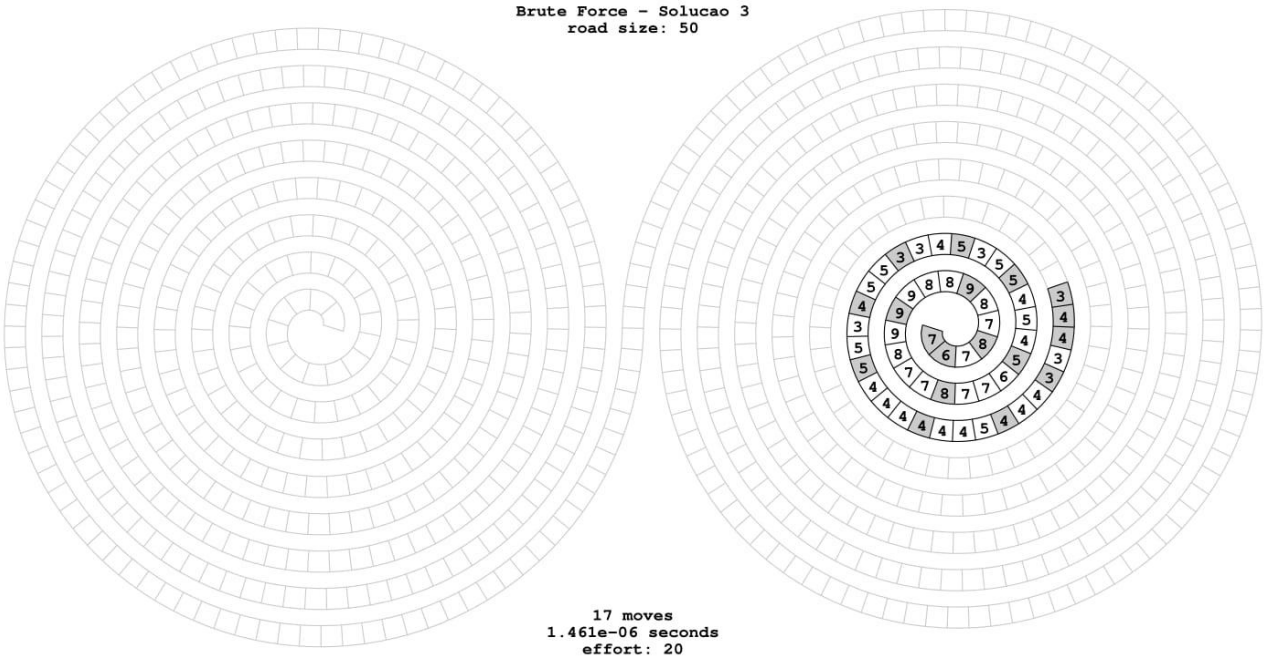


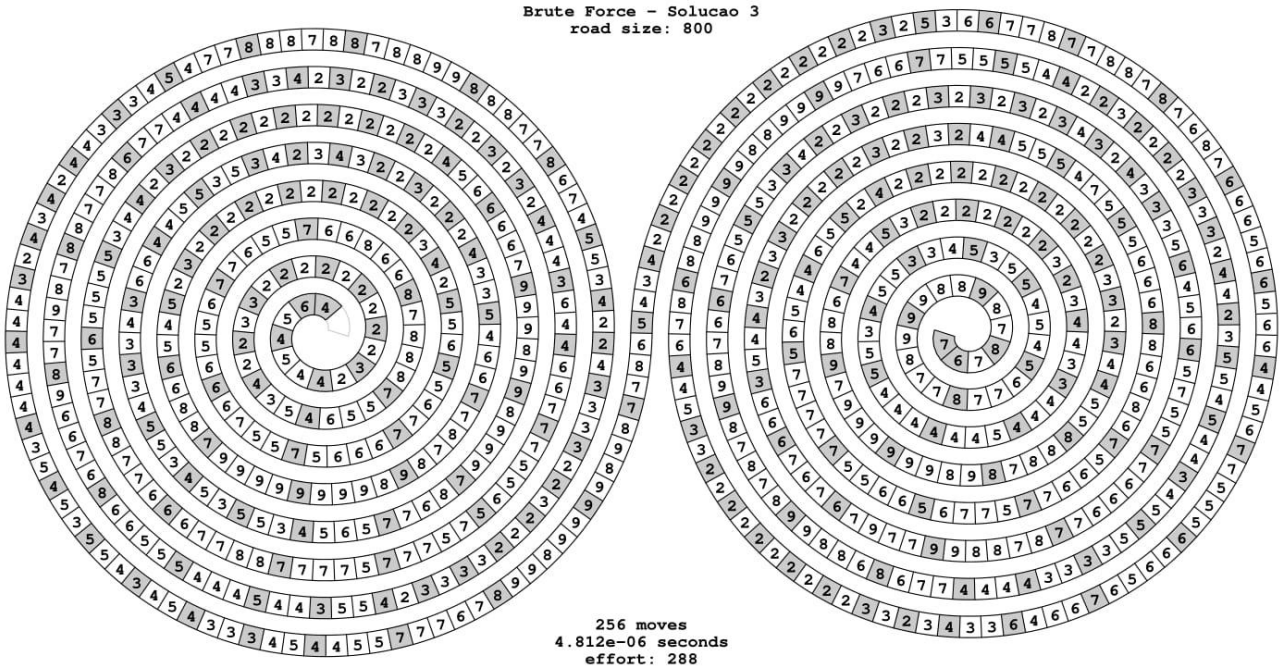
Reta de ajuste (valores de N como x, log10 dos valores de tempo como y):

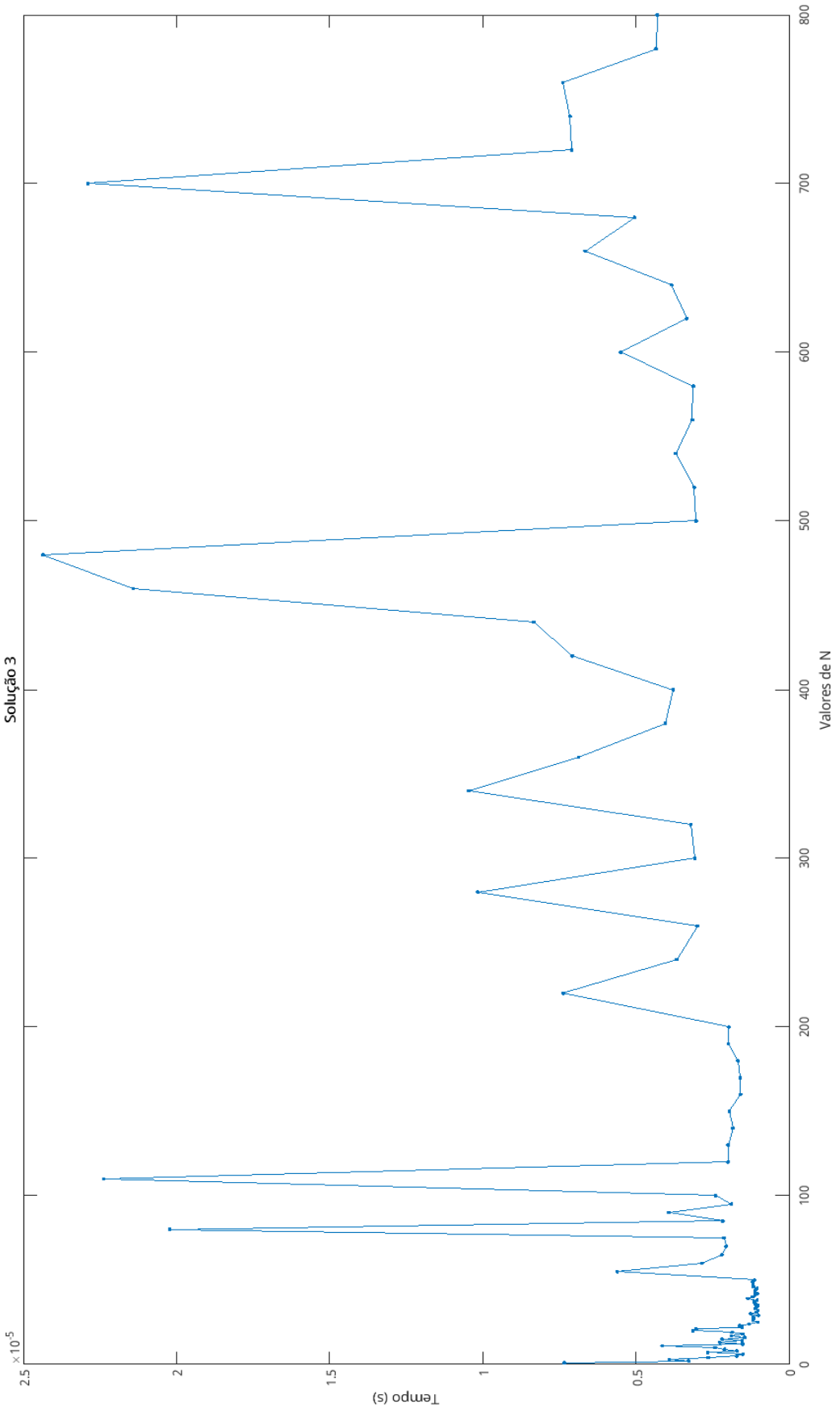
y = 0.11751x + -6.721

Estimativa de tempo para n = 800: 1.9513e+87 segundos

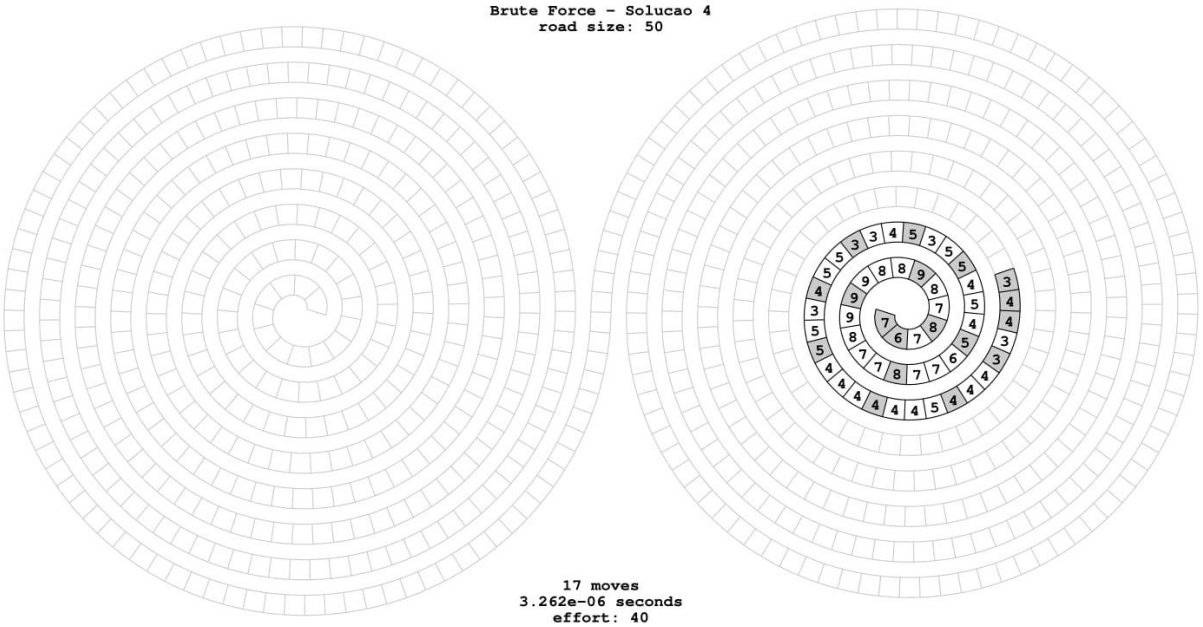
Solução 3

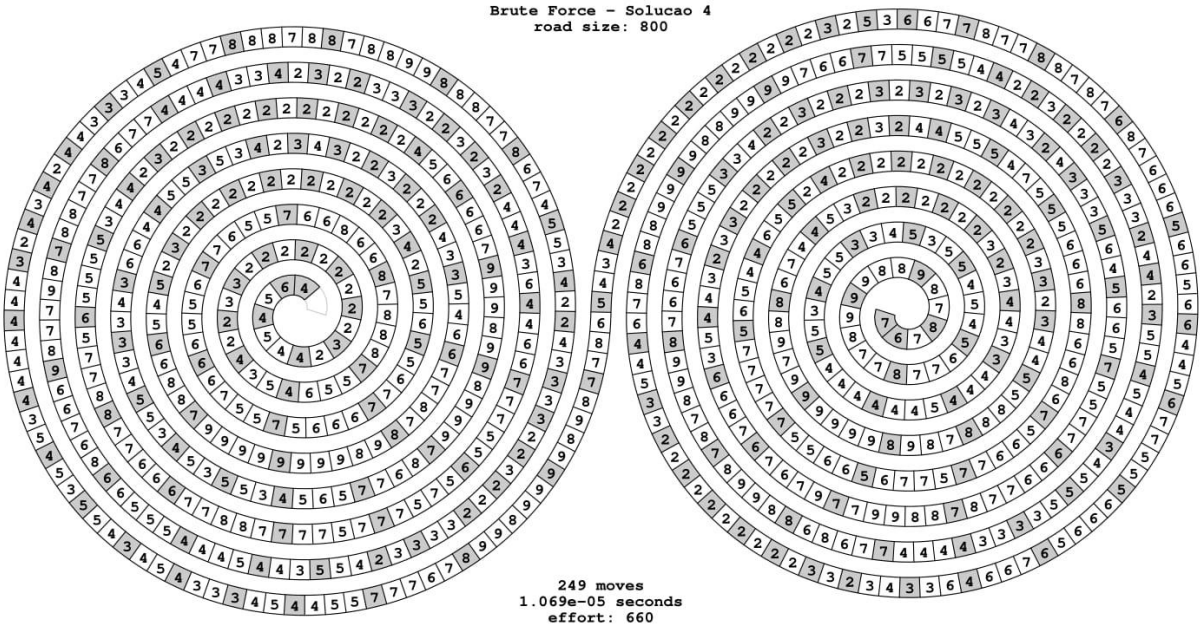

Brute Force - Solucao 3
road size: 50
17 moves
1.461e-06 seconds
effort: 20

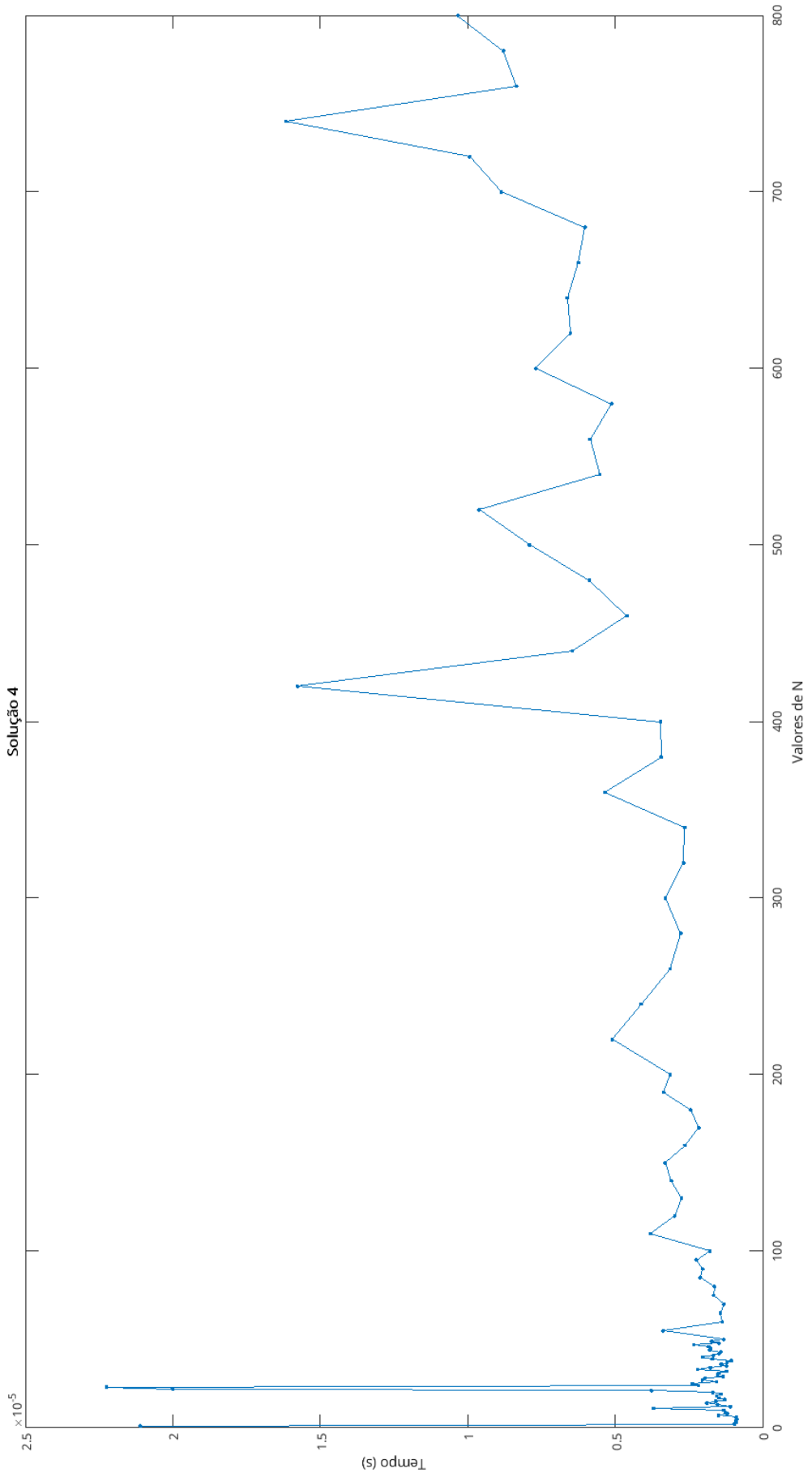

Brute Force - Solucao 3
road size: 800
256 moves
4.812e-06 seconds
effort: 288


Solução 3
Tempo (s)
Valores de N
×10^-5
2.5
2
1.5
1
0.5
0
0
100
200
300
400
500
600
700
800

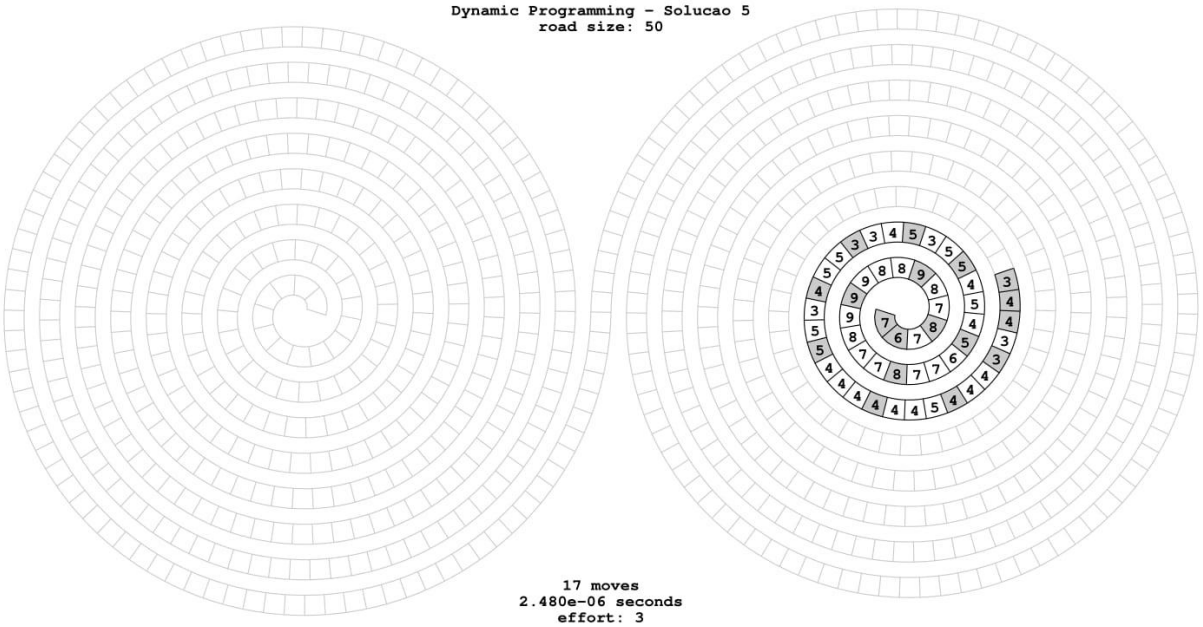
Solução 4

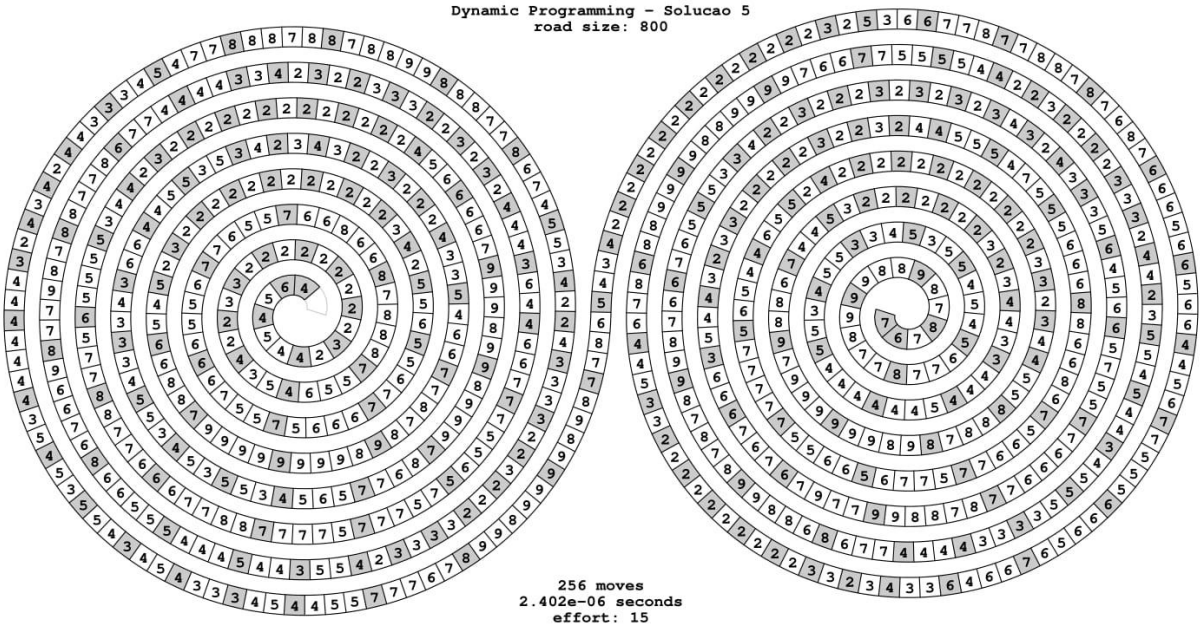

Brute Force - Solucao 4
road size: 50
17 moves
3.262e-06 seconds
effort: 40


Brute Force - Solucao 4
road size: 800
249 moves
1.069e-05 seconds
effort: 660


Solução 4
Tempo (s)
Valores de N
×10^-5
2.5
2
1.5
1
0.5
0
0
100
200
300
400
500
600
700
800

Solução 5


Dynamic Programming - Solucao 5
road size: 50
17 moves
2.480e-06 seconds
effort: 3


Dynamic Programming - Solucao 5
road size: 800
256 moves
2.402e-06 seconds
effort: 15

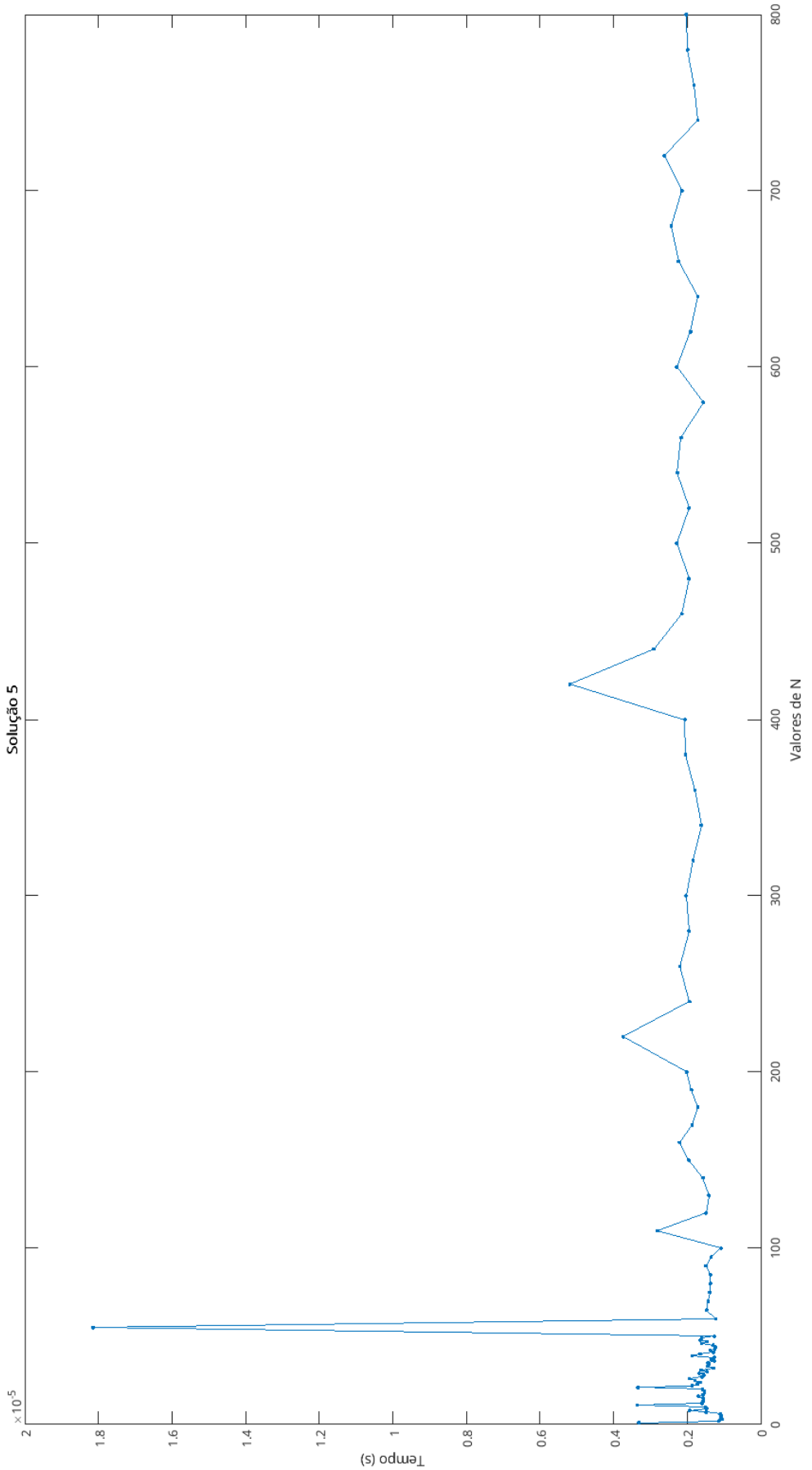
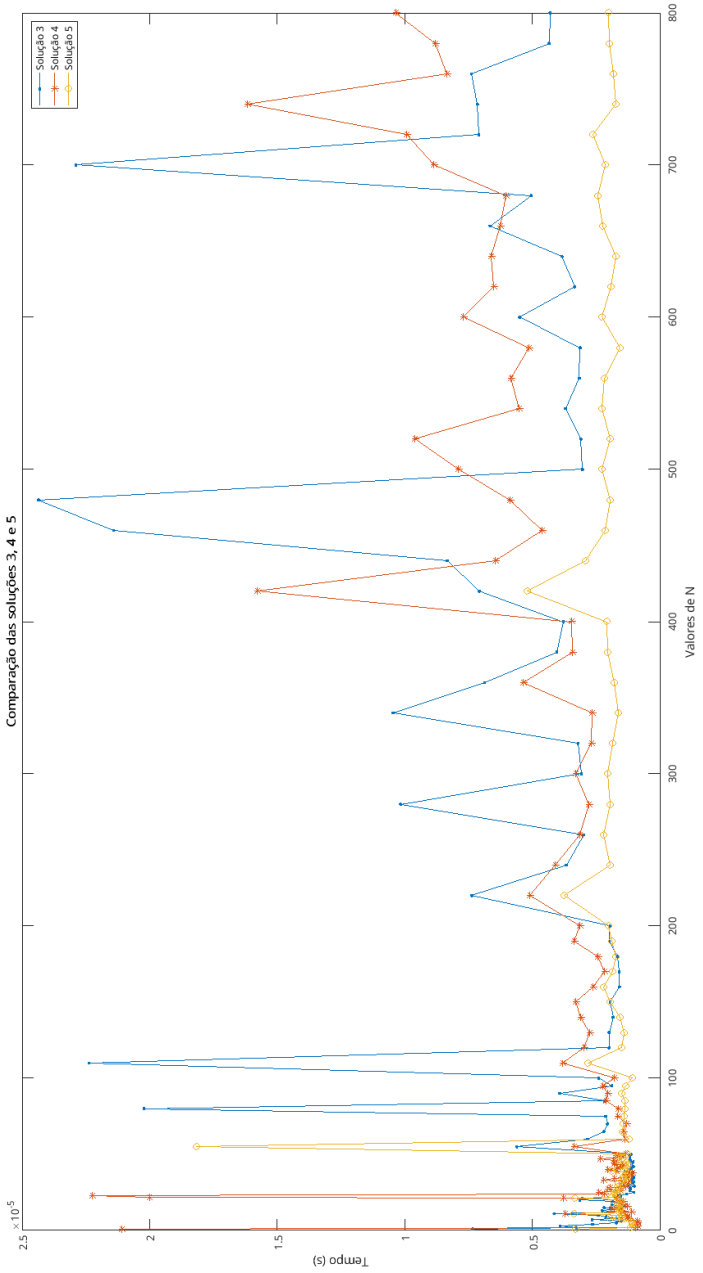

Solução 5
Tempo (s)
Valores de N
×10^-5
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
100
200
300
400
500
600
700
800

Gráfico de Comparação entre S3, S4 e S5



Como podemos observar pelo gráfico, a solução 5 é a mais eficiente de todas as soluções apresentadas.

Podemos assim concluir que, para resolver um dado problema, importa tanto a sua resolução como o método utilizado para o resolver, tendo em conta que, dependendo do método utilizado, os tempos irão variar bastante.

Anexos

Soluções

Solução 1

```
static solution_t solution_1, solution_1_best;
static double solution_1_elapsed_time; // time it took to solve the problem
static unsigned long solution_1_count; // effort dispended solving the problem

static void solution_1_recursion(int move_number, int position, int speed, int
final_position)
{
 int i, new_speed;

 // record move
 solution_1_count++;
 solution_1.positions[move_number] = position;
 // is it a solution?
 if (position == final_position && speed == 1)
 {
   // is it a better solution?
   if (move_number < solution_1_best.n_moves)
   {
     solution_1_best = solution_1;
     solution_1_best.n_moves = move_number;
   }
   return;
 }
 // no, try all legal speeds
 for (new_speed = speed - 1; new_speed <= speed + 1; new_speed++)
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <=
final_position)
   {
     for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
       ;
     if (i > new_speed)
           solution_1_recursion(move_number + 1, position + new_speed, new_speed,
final_position);
   }
}
```

Solução 2

```
static solution_t solution_2, solution_2_best;
static double solution_2_elapsed_time; // time it took to solve the problem
static unsigned long solution_2_count; // effort dispended solving the problem

static void solution_2_recursion(int move_number, int position, int speed, int
final_position)
{
 int i, new_speed;

 // record move
 solution_2_count++;
 solution_2.positions[move_number] = position;
 // is it a solution?
 if (position == final_position && speed == 1)
 {
   // is it a better solution?
   if (move_number < solution_2_best.n_moves)
   {
     solution_2_best = solution_2;
     solution_2_best.n_moves = move_number;
   }
   return;
 }
 // no, try all legal speeds
 for (new_speed = speed - 1; new_speed <= speed + 1; new_speed++)
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <=
final_position)
   {
     for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
       ;
     if (i > new_speed)
     {
       if (move_number + 1 < solution_2_best.n_moves)
            solution_2_recursion(move_number + 1, position + new_speed, new_speed,
final_position);
     }
   }
}
```

Solução 3

```
static solution_t solution_3, solution_3_best;
static double solution_3_elapsed_time; // time it took to solve the problem
static unsigned long solution_3_count; // effort dispended solving the problem

static int solution_3_recursion(int move_number, int position, int speed, int
final_position)
{
 int i, new_speed;

 // record move
 solution_3_count++;
 solution_3.positions[move_number] = position;
 // is it a solution?
 if (position == final_position && speed == 1)
 {
   // is it a better solution?
   if (move_number < solution_3_best.n_moves)
   {
     solution_3_best = solution_3;
     solution_3_best.n_moves = move_number;
   }
   return 1;
 }
 // no, try all legal speeds
 for (new_speed = speed + 1; new_speed >= speed - 1; new_speed--)
 {
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <=
final_position)
   {
      for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
// Confirma se todas as roads têm velocidade aceitável para a new_speed
       ;
     if (i > new_speed)
      { // Este testa, se for verdade (se todas as roads tiverem velocidade máxima
aceitavel), chama outra vez esta função
           int d = solution_3_recursion(move_number + 1, position + new_speed,
new_speed, final_position);
       if (d == 1)
         return 1;
     }
   }
 }
 return 0;
}
```

Solução 4

```
typedef struct
{
 int speed;                 // the current speed
 int blacklisted_speeds[3]; // the speeds that are not allowed at this position
 int i                      // the number of blacklisted speeds
} solution_t_info;

static solution_t solution_4, solution_4_best;
static solution_t_info solution_4_info[_max_road_size_ + 1];
static double solution_4_elapsed_time; // time it took to solve the problem
static unsigned long solution_4_count; // effort dispended solving the problem

static int check_speed(int position, int speed)
{
 for (int i = 0; i < solution_4_info[position].i; i++)
   if (solution_4_info[position].blacklisted_speeds[i] == speed)
     return 1;
 return 0;
}

static void solution_4_recursion(int speed, int position, int debt_pos, int debt,
int move_number, int final_position)
{
 int i, new_speed, new_position, speed_limit;

 // record move
 solution_4_count++;
 solution_4.positions[move_number] = position;

 if (position == final_position && speed == 1)
 {
   // is it a better solution?
   if (move_number < solution_4_best.n_moves)
   {
     solution_4_best = solution_4;
     solution_4_best.n_moves = move_number;
   }
   return;
 }

 if (debt < -2)
 {
   int previous_position = solution_4.positions[move_number - 1];
   int previous_speed = solution_4_info[previous_position].speed;

   // reset current info
   solution_4_info[position].speed = 0;
   solution_4_info[position].i = 0;

    solution_4_recursion(previous_speed, previous_position, 0, 0, move_number - 1,
final_position);
   return;
 }

 new_speed = speed + 1 + (position == debt_pos ? debt : 0);
```

```
 new_position = position + new_speed;

 for (i = 1; i <= new_speed; i++)
 {
   speed_limit = max_road_speed[position + i];

   if (new_position >= final_position)
     speed_limit = 1;

   // fail to run in this road
   if (new_speed > speed_limit || check_speed(position, new_speed - speed))
   {
         solution_4_recursion(speed, position, position, debt - 1, move_number,
final_position);
     return;
   }
 }

 // ok to run in this road
 solution_4_info[position].speed = speed;
     solution_4_info[position].blacklisted_speeds[solution_4_info[position].i]     =
new_speed - speed;
 solution_4_info[position].i++;
   solution_4_recursion(new_speed,   new_position,   0,   0,   move_number   +   1,
final_position);
}

static void solve_4(int final_position)
{
 if (final_position < 1 || final_position > _max_road_size_)
 {
   fprintf(stderr, "solve_4: bad final_position\n");
   exit(1);
 }

 for (int i = 0; i < _max_road_size_ + 1; i++)
 {
   solution_4_info[i].speed = 0;
   solution_4_info[i].i = 0;

   for (int j = 0; j < 3; j++)
     solution_4_info[i].blacklisted_speeds[j] = 100;
 }

 solution_4_elapsed_time = cpu_time();
 solution_4_count = 0ul;
 solution_4_best.n_moves = final_position + 100;
 solution_4_recursion(0, 0, 0, 0, 0, final_position);
 solution_4_elapsed_time = cpu_time() - solution_4_elapsed_time;
}
```

Solução 5

```
typedef struct
{
 int n_moves;                                // the number of moves (the number of positions is one more than the number of moves)
 int positions[1 + _max_road_size_]; // the positions (the first one must be zero)
 int speed;                                  // the speed at the end of this solution
} solution_t_previous;

typedef struct
{
 int save;  // if this is a valid solution
 int saved; // if this solution has been saved
 int valid; // if this solution is valid
} run_status_t;

static solution_t solution_5, solution_5_best;
static solution_t_previous solution_5_previous;
static double solution_5_elapsed_time; // time it took to solve the problem
static unsigned long solution_5_count; // effort dispended solving the problem

// Este array guarda a quantidade de posições necessárias para chegar à velocidade 1.
// Exemplo: sumCache[5] = 15, ou seja, para chegar à velocidade 1, é necessário percorrer 15 posições.
// velocidades |05|--|--|--|04|--|--|--|03|--|--|02|--|01|
// nº de casas |15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1|
static int sumCache[10] = {0, 1, 3, 6, 10, 15, 21, 28, 36, 45};

static void solution_5_dynamic(int move_number, int position, int speed, int final_position)
{
 run_status_t run_status;
 run_status.save = 0;
 run_status.saved = 0;

 while (position != final_position)
 {
   // Vai aumentar o effort
   solution_5_count++;
   // Vai guardar a posição atual onde está a fazer o movimento
   solution_5.positions[move_number] = position;

   // Vai experimentar as velocidades possiveis: aumentar, manter, diminuir
   for (int i = 1; i >= -1; i--)
   {
      // Definição da nova velocidade dependendo da velocidade atual e da velocidade que se quer aumentar, diminuir ou manter
     int new_speed = speed + i;
      // Definição da nova posição dependendo da posição atual e da nova velocidade definida
     int new_position = position + new_speed;
      // À partida a solução é válida, mas se for encontrada alguma situação que a torne inválida, esta variável vai ser alterada para 0
     run_status.valid = 1;
```

```
      // Se a velocidade for menor que 1 ou inválida para a posição atual, a solução
é inválida e passa-se para a próxima velocidade
      if (new_speed < 1 || new_speed > _max_road_speed_ || max_road_speed[position]
< new_speed)
        continue;

      int positions_array[sumCache[new_speed]];

       // Vai preencher o array com as posições que se vão percorrer para chegar à
velocidade 1
      int temp_index = 0;
      for (int j = new_speed; j > 0; j--)
      {
        for (int k = 0; k < j; k++)
        {
          positions_array[temp_index] = j;
          temp_index++;
        }
      }

      // Vai verificar se a nova velocidade é válida para as próximas posições
      for (int j = 1; j <= sumCache[new_speed]; j++)
      {
           // Se a velocidade for inválida para a posição atual + j, a solução é
inválida
        if (max_road_speed[position + j] < positions_array[j - 1])
        {
          run_status.valid = 0;
          break;
        }

           // Se a posição atual + j for maior que a posição final, a solução é
inválida
         // e a solução é guardada para ser usada na proxima chamada quando a posição
atual for menor que a posição final
        if (position + j > final_position)
        {
          run_status.save = 1;
          run_status.valid = 0;
          break;
        }
      }

       // Se a solução for válida, a solução é guardada para ser usada na proxima
chamada quando a posição atual for menor que a posição final
      if (run_status.save && !run_status.saved)
      {
        // Guarda as posições que já foram percorridas
        for (int i = 0; i < _max_road_size_ + 1; i++)
          solution_5_previous.positions[i] = solution_5.positions[i];
        // Guarda o número de movimentos que já foram feitos
        solution_5_previous.n_moves = move_number;
        // Guarda a velocidade atual
        solution_5_previous.speed = speed;
        // Indica que a solução foi guardada
        run_status.saved = 1;
      }
```

```
      // Se a solução for válida a posição atual é atualizada para a nova posição e
a velocidade atual é atualizada para a nova velocidade
      // e o número de movimentos é incrementado
      if (run_status.valid)
      {
        position = new_position;
        speed = new_speed;
        move_number++;
          // Visto que a solução é válida, não se precisa de ir verificar as outras
velocidades
        break;
      }
    }
  }

  solution_5.positions[move_number] = position;
  solution_5_best = solution_5;
  solution_5_best.n_moves = move_number;
}

static void solve_5(int final_position)
{
  if (final_position < 1 || final_position > _max_road_size_)
  {
    fprintf(stderr, "solve_5: bad final_position\n");
    exit(1);
  }
  solution_5_elapsed_time = cpu_time();
  solution_5_count = 0ul;
  solution_5_best.n_moves = final_position + 100;

  // Get the values from the previous run
  solution_5.n_moves = solution_5_previous.n_moves;
  for (int i = 0; i < _max_road_size_ + 1; i++)
    solution_5.positions[i] = solution_5_previous.positions[i];

                                      solution_5_dynamic(solution_5_previous.n_moves,
solution_5_previous.positions[solution_5_previous.n_moves],
solution_5_previous.speed, final_position);
  solution_5_elapsed_time = cpu_time() - solution_5_elapsed_time;
}
```

Código Completo

```
//
// AED, August 2022 (Tomás Oliveira e Silva)
//
// First practical assignement (speed run)
//
// Compile using either
//   cc -Wall -O2 -D_use_zlib_=0 solution_speed_run.c -lm
// or
//   cc -Wall -O2 -D_use_zlib_=1 solution_speed_run.c -lm -lz
//
// Place your student numbers and names here
//   N.Mec. 107276  Name: Miguel Vila
//   N.Mec. 107993  Name: Francisco Ribeiro
//   N.Mec. 108671  Name: Guilherme Vieira
//
//
// static configuration
//

#define                             executeSolution(i,                              name)
\
          if          (solution_##i##_elapsed_time          <          _time_limit_)
\
                                                                                    {
\
                                                          solve_##i(final_position);
\
                                  if          (print_this_one          !=          0)
\
                                                                                    {
\
                     sprintf(file_name,    "%03d_%02d.pdf",    final_position,    i);
\
                                  make_custom_pdf_file(file_name,      final_position,
\
                                      &max_road_speed[0], solution_##i##_best.n_moves,
\
                                                  &solution_##i##_best.positions[0],
solution_##i##_elapsed_time,                          \
                                                     solution_##i##_count,  name);
\
                                                                                    }
\
    printf(" %3d %16lu %9.3e |", solution_##i##_best.n_moves, solution_##i##_count,
solution_##i##_elapsed_time); \
                                                                                    }
\
                                                                                 else
\
                                                                                    {
\
                                  solution_##i##_best.n_moves          =          -1;
\
      printf("                                                                      |");
\
```

```
 }

#define _max_road_size_ 800 // the maximum problem size
#define _min_road_speed_ 2  // must not be smaller than 1, shouldnot be smaller
than 2
#define _max_road_speed_ 9  // must not be larger than 9 (only because of the PDF
figure)

//
// include files --- as this is a small project, we include the PDF generation code
directly from make_custom_pdf.c
//

#include <math.h>
#include <stdio.h>
#include "./elapsed_time.h"
#include "make_custom_pdf.c"

//
// road stuff
//

static int max_road_speed[1 + _max_road_size_]; // positions 0.._max_road_size_

static void init_road_speeds(void)
{
 double speed;
 int i;

 for (i = 0; i <= _max_road_size_; i++)
 {
   speed = (double)_max_road_speed_ * (0.55 + 0.30 * sin(0.11 * (double)i) + 0.10 *
sin(0.17 * (double)i + 1.0) + 0.15 * sin(0.19 * (double)i));
   max_road_speed[i] = (int)floor(0.5 + speed) + (int)((unsigned int)random() % 3u)
- 1;
   if (max_road_speed[i] < _min_road_speed_)
     max_road_speed[i] = _min_road_speed_;
   if (max_road_speed[i] > _max_road_speed_)
     max_road_speed[i] = _max_road_speed_;
 }
}

//
// description of a solution
//

typedef struct
{
 int n_moves;                                // the number of moves (the number of
positions is one more than the number of moves)
 int positions[1 + _max_road_size_]; // the positions (the first one must be zero)
} solution_t;

//
// the (very inefficient) recursive solution given to the students
//
```

```
static solution_t solution_1, solution_1_best;
static double solution_1_elapsed_time; // time it took to solve the problem
static unsigned long solution_1_count; // effort dispended solving the problem

static void solution_1_recursion(int move_number, int position, int speed, int
final_position)
{
 int i, new_speed;

 // record move
 solution_1_count++;
 solution_1.positions[move_number] = position;
 // is it a solution?
 if (position == final_position && speed == 1)
 {
   // is it a better solution?
   if (move_number < solution_1_best.n_moves)
   {
     solution_1_best = solution_1;
     solution_1_best.n_moves = move_number;
   }
   return;
 }
 // no, try all legal speeds
 for (new_speed = speed - 1; new_speed <= speed + 1; new_speed++)
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <=
final_position)
   {
     for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
       ;
     if (i > new_speed)
           solution_1_recursion(move_number + 1, position + new_speed, new_speed,
final_position);
   }
}

static void solve_1(int final_position)
{
 if (final_position < 1 || final_position > _max_road_size_)
 {
   fprintf(stderr, "solve_1: bad final_position\n");
   exit(1);
 }
 solution_1_elapsed_time = cpu_time();
 solution_1_count = 0ul;
 solution_1_best.n_moves = final_position + 100;
 solution_1_recursion(0, 0, 0, final_position);
 solution_1_elapsed_time = cpu_time() - solution_1_elapsed_time;
}

//
// Our solution n. 1 using optimized brute force
//

static solution_t solution_2, solution_2_best;
static double solution_2_elapsed_time; // time it took to solve the problem
static unsigned long solution_2_count; // effort dispended solving the problem
```

```
static void solution_2_recursion(int move_number, int position, int speed, int
final_position)
{
 int i, new_speed;

 // record move
 solution_2_count++;
 solution_2.positions[move_number] = position;
 // is it a solution?
 if (position == final_position && speed == 1)
 {
   // is it a better solution?
   if (move_number < solution_2_best.n_moves)
   {
     solution_2_best = solution_2;
     solution_2_best.n_moves = move_number;
   }
   return;
 }
 // no, try all legal speeds
 for (new_speed = speed - 1; new_speed <= speed + 1; new_speed++)
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <=
final_position)
   {
     for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
       ;
     if (i > new_speed)
     {
       if (move_number + 1 < solution_2_best.n_moves)
            solution_2_recursion(move_number + 1, position + new_speed, new_speed,
final_position);
     }
   }
}

static void solve_2(int final_position)
{
 if (final_position < 1 || final_position > _max_road_size_)
 {
   fprintf(stderr, "solve_2: bad final_position\n");
   exit(1);
 }
 solution_2_elapsed_time = cpu_time();
 solution_2_count = 0ul;
 solution_2_best.n_moves = final_position + 100;
 solution_2_recursion(0, 0, 0, final_position);
 solution_2_elapsed_time = cpu_time() - solution_2_elapsed_time;
}

//
// Our solution n. 2 using optimized brute force
//

static solution_t solution_3, solution_3_best;
static double solution_3_elapsed_time; // time it took to solve the problem
static unsigned long solution_3_count; // effort dispended solving the problem
```

```
static int solution_3_recursion(int move_number, int position, int speed, int
final_position)
{
 int i, new_speed;

 // record move
 solution_3_count++;
 solution_3.positions[move_number] = position;
 // is it a solution?
 if (position == final_position && speed == 1)
 {
   // is it a better solution?
   if (move_number < solution_3_best.n_moves)
   {
     solution_3_best = solution_3;
     solution_3_best.n_moves = move_number;
   }
   return 1;
 }
 // no, try all legal speeds
 for (new_speed = speed + 1; new_speed >= speed - 1; new_speed--)
 {
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <=
final_position)
   {
      for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
// Confirma se todas as roads têm velocidade aceitável para a new_speed
       ;
     if (i > new_speed)
      { // Este testa, se for verdade (se todas as roads tiverem velocidade máxima
aceitavel), chama outra vez esta função
           int d = solution_3_recursion(move_number + 1, position + new_speed,
new_speed, final_position);
       if (d == 1)
         return 1;
     }
   }
 }
 return 0;
}

static void solve_3(int final_position)
{
 if (final_position < 1 || final_position > _max_road_size_)
 {
   fprintf(stderr, "solve_3: bad final_position\n");
   exit(1);
 }
 solution_3_elapsed_time = cpu_time();
 solution_3_count = 0ul;
 solution_3_best.n_moves = final_position + 100;
 solution_3_recursion(0, 0, 0, final_position);
 solution_3_elapsed_time = cpu_time() - solution_3_elapsed_time;
}

//
```

```
// Our solution n. 3 using optimized brute force
//
typedef struct
{
 int speed;                 // the current speed
 int blacklisted_speeds[3]; // the speeds that are not allowed at this position
 int i                      // the number of blacklisted speeds
} solution_t_info;

static solution_t solution_4, solution_4_best;
static solution_t_info solution_4_info[_max_road_size_ + 1];
static double solution_4_elapsed_time; // time it took to solve the problem
static unsigned long solution_4_count; // effort dispended solving the problem

static int check_speed(int position, int speed)
{
 for (int i = 0; i < solution_4_info[position].i; i++)
   if (solution_4_info[position].blacklisted_speeds[i] == speed)
     return 1;
 return 0;
}

static void solution_4_recursion(int speed, int position, int debt_pos, int debt,
int move_number, int final_position)
{
 int i, new_speed, new_position, speed_limit;

 // record move
 solution_4_count++;
 solution_4.positions[move_number] = position;

 if (position == final_position && speed == 1)
 {
   // is it a better solution?
   if (move_number < solution_4_best.n_moves)
   {
     solution_4_best = solution_4;
     solution_4_best.n_moves = move_number;
   }
   return;
 }

 if (debt < -2)
 {
   int previous_position = solution_4.positions[move_number - 1];
   int previous_speed = solution_4_info[previous_position].speed;

   // reset current info
   solution_4_info[position].speed = 0;
   solution_4_info[position].i = 0;

    solution_4_recursion(previous_speed, previous_position, 0, 0, move_number - 1,
final_position);
   return;
 }

 new_speed = speed + 1 + (position == debt_pos ? debt : 0);
```

```
 new_position = position + new_speed;

 for (i = 1; i <= new_speed; i++)
 {
   speed_limit = max_road_speed[position + i];

   if (new_position >= final_position)
     speed_limit = 1;

   // fail to run in this road
   if (new_speed > speed_limit || check_speed(position, new_speed - speed))
   {
         solution_4_recursion(speed, position, position, debt - 1, move_number,
final_position);
     return;
   }
 }

 // ok to run in this road
 solution_4_info[position].speed = speed;
     solution_4_info[position].blacklisted_speeds[solution_4_info[position].i]     =
new_speed - speed;
 solution_4_info[position].i++;
   solution_4_recursion(new_speed,   new_position,   0,   0,   move_number   +   1,
final_position);
}

static void solve_4(int final_position)
{
 if (final_position < 1 || final_position > _max_road_size_)
 {
   fprintf(stderr, "solve_4: bad final_position\n");
   exit(1);
 }

 for (int i = 0; i < _max_road_size_ + 1; i++)
 {
   solution_4_info[i].speed = 0;
   solution_4_info[i].i = 0;

   for (int j = 0; j < 3; j++)
     solution_4_info[i].blacklisted_speeds[j] = 100;
 }

 solution_4_elapsed_time = cpu_time();
 solution_4_count = 0ul;
 solution_4_best.n_moves = final_position + 100;
 solution_4_recursion(0, 0, 0, 0, 0, final_position);
 solution_4_elapsed_time = cpu_time() - solution_4_elapsed_time;
}

//
// Our solution n. 4 using dynamic programming
//

typedef struct
{
```

```
 int n_moves;                                // the number of moves (the number of
positions is one more than the number of moves)
 int positions[1 + _max_road_size_]; // the positions (the first one must be zero)
 int speed;                          // the speed at the end of this solution
} solution_t_previous;

typedef struct
{
 int save;  // if this is a valid solution
 int saved; // if this solution has been saved
 int valid; // if this solution is valid
} run_status_t;

static solution_t solution_5, solution_5_best;
static solution_t_previous solution_5_previous;
static double solution_5_elapsed_time; // time it took to solve the problem
static unsigned long solution_5_count; // effort dispended solving the problem

// Este array guarda a quantidade de posições necessárias para chegar à velocidade
1.
// Exemplo: sumCache[5] = 15, ou seja, para chegar à velocidade 1, é necessário
percorrer 15 posições.
// velocidades |05|--|--|--|04|--|--|--|03|--|--|02|--|01|
// nº de casas |15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1|
static int sumCache[10] = {0, 1, 3, 6, 10, 15, 21, 28, 36, 45};

static  void  solution_5_dynamic(int  move_number,  int  position,  int  speed,  int
final_position)
{
 run_status_t run_status;
 run_status.save = 0;
 run_status.saved = 0;

 while (position != final_position)
 {
   // Vai aumentar o effort
   solution_5_count++;
   // Vai guardar a posição atual onde está a fazer o movimento
   solution_5.positions[move_number] = position;

   // Vai experimentar as velocidades possiveis: aumentar, manter, diminuir
   for (int i = 1; i >= -1; i--)
   {
     // Definição da nova velocidade dependendo da velocidade atual e da velocidade
que se quer aumentar, diminuir ou manter
     int new_speed = speed + i;
      // Definição da nova posição dependendo da posição atual e da nova velocidade
definida
     int new_position = position + new_speed;
      // À partida a solução é válida, mas se for encontrada alguma situação que a
torne inválida, esta variável vai ser alterada para 0
     run_status.valid = 1;

      // Se a velocidade for menor que 1 ou inválida para a posição atual, a solução
é inválida e passa-se para a próxima velocidade
      if (new_speed < 1 || new_speed > _max_road_speed_ || max_road_speed[position]
< new_speed)
```

```
        continue;

      int positions_array[sumCache[new_speed]];

       // Vai preencher o array com as posições que se vão percorrer para chegar à
velocidade 1
      int temp_index = 0;
      for (int j = new_speed; j > 0; j--)
      {
        for (int k = 0; k < j; k++)
        {
          positions_array[temp_index] = j;
          temp_index++;
        }
      }

      // Vai verificar se a nova velocidade é válida para as próximas posições
      for (int j = 1; j <= sumCache[new_speed]; j++)
      {
           // Se a velocidade for inválida para a posição atual + j, a solução é
inválida
        if (max_road_speed[position + j] < positions_array[j - 1])
        {
          run_status.valid = 0;
          break;
        }

           // Se a posição atual + j for maior que a posição final, a solução é
inválida
         // e a solução é guardada para ser usada na proxima chamada quando a posição
atual for menor que a posição final
        if (position + j > final_position)
        {
          run_status.save = 1;
          run_status.valid = 0;
          break;
        }
      }

       // Se a solução for válida, a solução é guardada para ser usada na proxima
chamada quando a posição atual for menor que a posição final
      if (run_status.save && !run_status.saved)
      {
        // Guarda as posições que já foram percorridas
        for (int i = 0; i < _max_road_size_ + 1; i++)
          solution_5_previous.positions[i] = solution_5.positions[i];
        // Guarda o número de movimentos que já foram feitos
        solution_5_previous.n_moves = move_number;
        // Guarda a velocidade atual
        solution_5_previous.speed = speed;
        // Indica que a solução foi guardada
        run_status.saved = 1;
      }

       // Se a solução for válida a posição atual é atualizada para a nova posição e
a velocidade atual é atualizada para a nova velocidade
      // e o número de movimentos é incrementado
```

```
      if (run_status.valid)
      {
        position = new_position;
        speed = new_speed;
        move_number++;
          // Visto que a solução é válida, não se precisa de ir verificar as outras
velocidades
        break;
      }
    }
  }

  solution_5.positions[move_number] = position;
  solution_5_best = solution_5;
  solution_5_best.n_moves = move_number;
}

static void solve_5(int final_position)
{
  if (final_position < 1 || final_position > _max_road_size_)
  {
    fprintf(stderr, "solve_5: bad final_position\n");
    exit(1);
  }
  solution_5_elapsed_time = cpu_time();
  solution_5_count = 0ul;
  solution_5_best.n_moves = final_position + 100;

  // Get the values from the previous run
  solution_5.n_moves = solution_5_previous.n_moves;
  for (int i = 0; i < _max_road_size_ + 1; i++)
    solution_5.positions[i] = solution_5_previous.positions[i];

                                      solution_5_dynamic(solution_5_previous.n_moves,
solution_5_previous.positions[solution_5_previous.n_moves],
solution_5_previous.speed, final_position);
  solution_5_elapsed_time = cpu_time() - solution_5_elapsed_time;
}

//
// example of the slides
//
static void example(void)
{
  int i, final_position;

  srandom(0xAED2022);
  init_road_speeds();
  final_position = 30;
  solve_1(final_position);
      make_custom_pdf_file("example.pdf",     final_position,     &max_road_speed[0],
solution_1_best.n_moves,   &solution_1_best.positions[0],   solution_1_elapsed_time,
solution_1_count, "Plain recursion");
  printf("mad road speeds:");
  for (i = 0; i <= final_position; i++)
    printf(" %d", max_road_speed[i]);
  printf("\n");
```

```
 printf("positions:");
 for (i = 0; i <= solution_1_best.n_moves; i++)
   printf(" %d", solution_1_best.positions[i]);
 printf("\n");
}

//
// main program
//
int main(int argc, char *argv[argc + 1])
{
#define _time_limit_ 3600.0
 int n_mec, final_position, print_this_one;
 char file_name[64];
 // generate the example data
 if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 'e' && argv[1][2] == 'x')
 {
   example();
   return 0;
 }
 // initialization
 n_mec = (argc < 2) ? 0xAED2022 : atoi(argv[1]);
 srandom((unsigned int)n_mec);
 init_road_speeds();
 // run all solution methods for all interesting sizes of the problem
 final_position = 1;
 solution_1_elapsed_time = 0.0;
 printf("    + --- ---------------- --------- +\n");
 printf("    |                plain recursion |\n");
 printf("--- + --- ---------------- --------- +\n");
 printf("  n | sol            count  cpu time |\n");
 printf("--- + --- ---------------- --------- +\n");
 while (final_position <= _max_road_size_ /* && final_position <= 20*/)
 {
    print_this_one = (final_position == 10 || final_position == 20 || final_position
== 50 || final_position == 100 || final_position == 200 || final_position == 400 ||
final_position == 800) ? 1 : 0;
   printf("%3d |", final_position);

   executeSolution(1, “Brute Force - Solucao 1”);
   executeSolution(2, “Brute Force - Solucao 2”);
   executeSolution(3, “Brute Force - Solucao 3”);
   executeSolution(4, “Brute Force - Solucao 4”);
   executeSolution(5, “Dynamic Programming - Solucao 5”);
   // done
   printf("\n");
   fflush(stdout);
   // new final_position
   if (final_position < 50)
     final_position += 1;
   else if (final_position < 100)
     final_position += 5;
   else if (final_position < 200)
     final_position += 10;
   else
     final_position += 20;
 }
```

```
 printf("--- + --- ---------------- --------- +\n");
 return 0;
#undef _time_limit_
}
```

Geração de Gráficos

Solução 1

```
% Load Values
valores = load("solution_1_times.txt");
n1 = valores(1:end, 1);
t = valores(1:end, 2);

% Graph
figure(5);
semilogy(n1, t, '.-');
ylabel("Tempo (s)");
xlabel("Valores de N");
title("Solução 1");
```

Solução 2

```
% Load Values
valores = load("solution_2_times.txt");
n1 = valores(1:end, 1);
t = valores(1:end, 2);

% Graph
figure(5);
semilogy(n1, t, '.-');
ylabel("Tempo (s)");
xlabel("Valores de N");
title("Solução 2");
```

Solução 3

```
% Load Values
valores = load("solution_3_times.txt");
n1 = valores(1:end, 1);
t = valores(1:end, 2);

% Graph
figure(5);
plot(n1, t, '.-');
ylabel("Tempo (s)");
xlabel("Valores de N");
title("Solução 3");
```

Solução 4

```
% Load Values
valores = load("solution_4_times.txt");
n1 = valores(1:end, 1);
t = valores(1:end, 2);

% Graph
figure(5);
plot(n1, t, '.-');
ylabel("Tempo (s)");
xlabel("Valores de N");
title("Solução 4");
```

Solução 5

```
% Load Values
valores = load("solution_5_times.txt");
n1 = valores(1:end, 1);
t = valores(1:end, 2);

% Graph
figure(5);
plot(n1, t, '.-');
ylabel("Tempo (s)");
xlabel("Valores de N");
title("Solução 5");
```

Gráfico de Comparação entre S3, S4 e S5

```
% Load Values
valores3 = load("solution_3_times.txt");
valores4 = load("solution_4_times.txt");
valores5 = load("solution_5_times.txt");
n = valores3(1:end, 1);
t1 = valores3(1:end, 2);
t2 = valores4(1:end, 2);
t3 = valores5(1:end, 2);

% Graph
figure(5);
plot(n, t1, '.-');
hold on;
plot(n, t2, '*-');
hold on;
plot(n, t3, 'o-');
hold on;
legend('Solução 3', 'Solução 4', 'Solução 5')
ylabel("Tempo (s)");
xlabel("Valores de N");
title("Comparação das soluções 3, 4 e 5");
```

Estimativas de Tempo de Execução

Solução 1

```
% Clear Command Window
clc

% Load Values
valores = load("solution_1_times.txt");
n1 = valores(1:end, 1);
t = valores(1:end, 2);

% Calculate Linear Regression, getting the log10 for
P = polyfit(n1, log10(t), 1);
m = P(1);
b = P(2);
disp(['Linear Regression equation: y = ' num2str(m) 'x + ' num2str(b) ])

% Calculate y with x = 800, and then do 10^y, to get the time
y = m*800 + b;
final = 10^y;
disp(['Tempo final = ' num2str(final, 5) ' segundos'])
```

Solução 2

```
% Clear Command Window
clc

% Load Values
valores = load("solution_2_times.txt");
n1 = valores(1:end, 1);
t = valores(1:end, 2);

% Calculate Linear Regression, getting the log10 for
P = polyfit(n1, log10(t), 1);
m = P(1);
b = P(2);
disp(['Linear Regression equation: y = ' num2str(m) 'x + ' num2str(b) ])

% Calculate y with x = 800, and then do 10^y, to get the time
y = m*800 + b;
final = 10^y;
disp(['Tempo final = ' num2str(final, 5) ' segundos'])
```