Speed run

Licenciatura em Engenharia Informática

Algoritmos e Estruturas de Dados

Docente:

Professor Tomás Oliveira e Silva

Alunos:

Bárbara Nóbrega Galiza – 105937 (50%)

Tomás António de Oliveira Victal - 109018 (50%)

Novembro de 2022

Índice

Introdução

O problema consiste em uma “corrida”, na qual um carro tenta chegar ao fim de uma rodovia dividida em n *segmentos*, $n \in \{10,20,50,100,200,400,800\}$ com o menor número de *movimentos*. Um *movimento* é caracterizado pelos seguintes passos:

1) Escolher dentre as opções:
    i) Diminuir sua velocidade em 1 (travar)
    ii) Manter sua velocidade atual
    iii) Aumentar sua velocidade em 1 (acelerar)
2) Avançar para a nova posição;

A *velocidade* do carro corresponde ao número de segmentos que ele avança em cada movimento. Cada segmento possui uma *velocidade máxima permitida*, e o carro deve chegar ao último com velocidade = *1*. O carro parte do primeiro segmento com velocidade = *0*.

Há várias formas de alcançar o final para o mesmo percurso, e por isso o objetivo do programa é otimizar o número de movimentos necessários para tal, *i.e,* encontrar a solução com o menor número de movimentos.

Métodos

Plain Recursion

Método dado pelo professor, implementa uma função recursiva, que testa para cada movimento as três possibilidades: travar, manter a velocidade e acelerar. A função recebe como argumentos o número do movimento, a posição e a velocidade atuais além da posição final pretendida. Cada chamada de função representa um movimento, e em cada movimento é gravada a posição atual em um *array*. Quando se atinge a posição final num determinado movimento com a velocidade 1, tem-se uma "solução", que corresponde a uma *struct* com dois campos: número de movimentos e *array* de posições. A fim de comparar as diversas soluções que são geradas, duas instâncias dessa estrutura são declaradas antes da função como variáveis globais, uma para caracterizar as soluções de forma geral (*solution_1*) e outra para caracterizar a melhor solução *(solution_1_best*).

Portanto, quando se chega a uma solução, é feito um teste para verificar se esta solução é melhor que a melhor solução já encontrada até agora (testado com base no número de movimentos). Se for o caso, a solução atual é gravada como melhor solução (*solution_1 = solution_1_best*).

Assim, após correr todas as possibilidades, o programa retorna à sua primeira chamada, e basta consultar os campos de *solution_1_best* para obter o número de movimentos e as posições do trajeto mais eficiente para a posição final passada como argumento.

Esse método usa a estratégia de testar todas as possibilidades, e por isso, não é eficiente. Sua menor eficácia em relação às próximas soluções implementadas ocorre devido ao teste de possibilidades que deveriam ser descartadas logo no início, por já mostrarem pelo seu número de movimentos que são soluções piores.

Smart Recursion

Segundo método implementado, utiliza como base o método anterior, *Plain Recursion*, com uma pequena mudança: uma instrução *if,* introduzida antes do ciclo que percorre as três velocidades, que testa se a melhor solução já encontrada chegou à posição atual em um menor número de movimentos que o número de movimentos atual. Caso isso seja verdade, a função retorna, ou seja, o algoritmo desiste dessa tentativa. Assim, não é necessário chegar até a posição final para saber que a solução a ser testada não é a melhor solução, o que proporciona um enorme ganho de tempo.

Em comparação com os outros métodos, esse fica em 3° lugar, pois apesar de ser melhor que o anterior, é pior que os apresentados a seguir.

Smarter Recursion

A terceira implementação consiste em uma melhoria do método *Smart Recursion*. No código foi alterado o ciclo *for,* que percorre entre as 3 novas velocidades possíveis, de modo a começar pela velocidade mais alta em vez da mais baixa. Desse modo, são testadas as soluções mais eficientes (com um menor número de movimentos) primeiro, o que faz com que o código tenha de testar menos soluções futuramente, pois quando testá-las o *if* herdado da solução anterior vai, na maioria dos casos, retornar.

Assim esta solução torna-se uma das soluções mais rápidas e eficientes implementadas, juntamente com a solução apresentada a seguir.

Check Speed Limit

Ao contrário das implementações antes apresentadas, nesta não se utiliza recursão, e por isso obtém-se uma solução única, a melhor solução possível. Para chegar a essa solução este método usa duas funções, uma principal e uma auxiliar, *solution_4_checkSpeedLimit* e *checkSpeedLimit,* respetivamente.

A função principal recebe a posição final desejada como único argumento, e em seu corpo é usado um ciclo *while* em que cada iteração equivale a um movimento do carro, e que acaba quando a posição atual for igual a posição final. Em cada iteração são testadas as três ações possíveis pela respetiva ordem: acelerar, manter a velocidade ou (na pior das hipóteses) travar, o que vai determinar a nova velocidade. Quando o teste de uma das possibilidades for positivo, essa nova velocidade é usada para calcular o próximo movimento e o ciclo passa para a próxima iteração. Os testes da nova velocidade são feitos pela função auxiliar, a qual recebe como argumentos uma velocidade, a quantidade de posições que serão testadas, a posição atual, e a posição final. Esta função tem como objetivo testar se é possível, numa situação limite, avançar para a próxima posição na velocidade que foi passada no argumento. Ou seja, a função testa se é possível, da posição atual para frente, passar por todas as posições (número de posições que foi passado nos argumentos para testar), sempre a travar, sem exceder a velocidade máxima de cada uma delas.

Exemplo de como funciona a função *checkSpeedLimit:*

5	5	6	6	8	7	8	9	8	9	8	9	9	8	9	7	7	6	6	6	5	6	5	4	4	3	5	3	3	3	3
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]	[17]	[18]	[19]	[20]	[21]	[22]	[23]	[24]	[25]	[26]	[27]	[28]	[29]	[30]

Se o programa estiver na casa 10 por exemplo, será passado para a função:

1° caso: velocidade = 5 *check_size* = 15 (valor tirado do *array check_size*).

No 1° caso a função tem de ver se nos próximos 6 (5+1 do atual) blocos pode andar à velocidade 6, depois nos próximos 4 blocos se pode andar a uma velocidade de 4, nos próximos 3 blocos andar à velocidade de 3, até chegar ao valor de *check_size*.

Caso check_size seja maior que a quantidade de casas até ao final, retorna 0.

Quando o ciclo da função principal acabar, o número de movimentos é guardado no melhor número de movimentos da solução 4 (*solution_4_best.n_moves*).

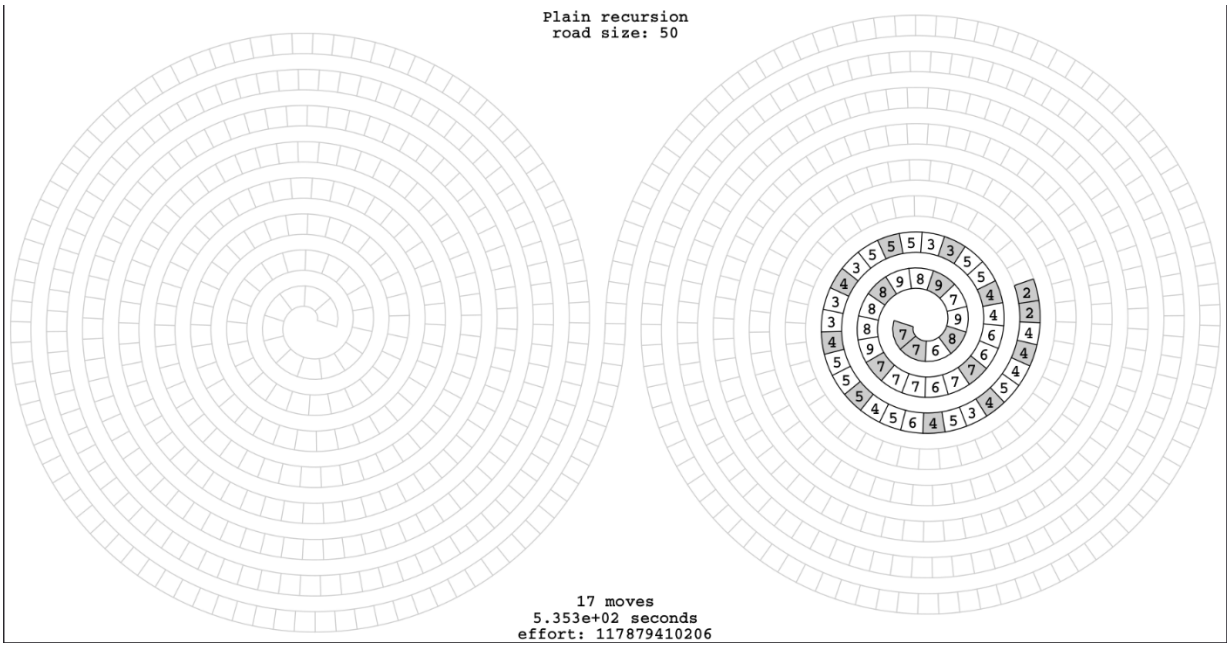
Como é calculada uma única solução, essa implementação é, junto com solução anterior, a mais rápida e eficiente entre as apresentadas. A diferença entre ambas é mínima, e o título de “solução mais rápida” varia para cada número mecanográfico testado, como é mostrado na secção de resultados.

Resultados

Plain Recursion

Como referido, esta é a solução mais lenta. Com o *time_limit = 3600* (1 hora), consegue chegar apenas à posição final de 55, para qual leva mais que 1 hora (o programa só interrompe após obter a primeira solução com o tempo superior ao limite).

Figura 1: PDF gerado pelo método *Plain Recursion* para a posição final 50.



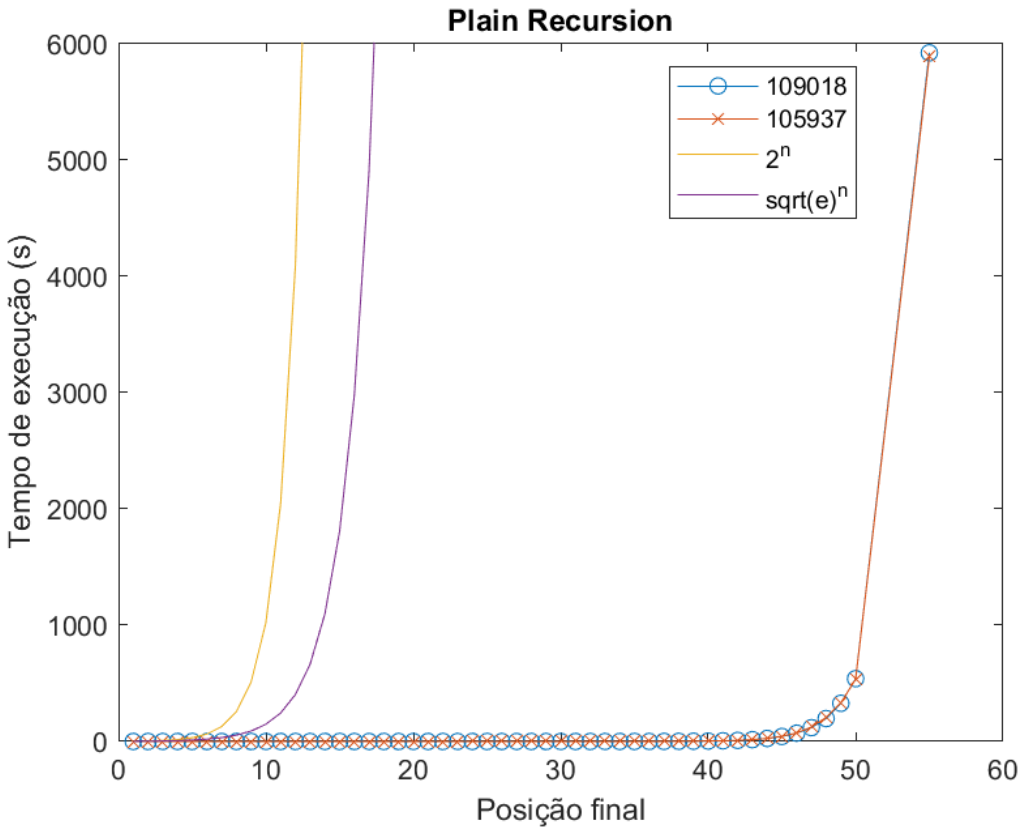
(Fonte: autoria própria do grupo, com o uso do código *make_custom_pdf.c* disponibilizado pelo professor Tomás Oliveira Silva, 2022)

Seu algoritmo é de complexidade computacional exponencial, $O(2^n)$. Na verdade, um *upperbound* menor pode ser definido: $O(\sqrt{e}^n)$. Essa complexidade se dá pelo facto

de que em cada movimento a função recursiva é chamada para todas as novas velocidades válidas possíveis (que nem sempre serão as três). Em média, como pode ser confirmado com a técnica de *least squares fit* que será descrita a seguir, cada chamada à função faz $\sqrt{e} \cong 1{,}64872$ novas chamadas.

O gráfico abaixo mostra o crescimento exponencial do tempo de execução em função da posição final, para os números mecanográficos de cada elemento do grupo. Além disso, traça as *upperbounds* mencionadas.

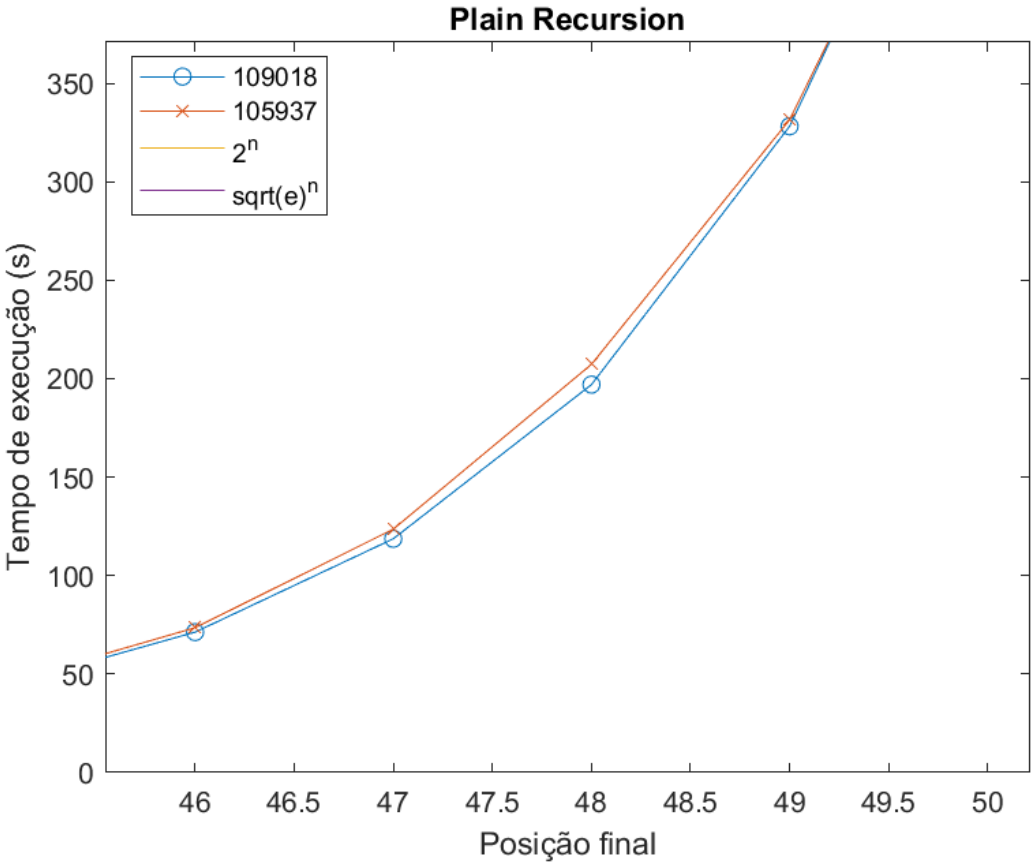
Figura 2: Gráfico do tempo de execução do método *Plain Recursion*.



(Fonte: autoria própria do grupo, com o uso do MATLAB (código presente no apêndice), 2022)

Ao aplicar um *zoom,* vê-se que os tempos de execução variam muito pouco de acordo com o número mecanográfico utilizado:

Figura 3: Gráfico do tempo de execução do método *Plain Recursion* ampliado.



(Fonte: autoria própria do grupo, com o uso do MATLAB (código presente no apêndice), 2022)

Plain Recursion: Least Squares Fit

A fim de obter uma melhor aproximação da função que gera esses dois resultados, aplicamos a técnica dos menores quadrados (*least squares fit*) aos dados obtidos. Para o efeito, desenvolvemos um código MATLAB baseado no código apresentado no guião da disciplina (presente no apêndice).

A partir dos gráficos anteriores, sabíamos que a função seria da forma $y = a * e^{x*b}$. Para aplicar ao código que nos foi dado, precisamos aplicar o logaritmo natural à função, que, consequentemente, tomou a forma de uma função afim: $\ln y = \ln a + x * b$.

Dessa forma, para cada número mecanográfico obtivemos os valores das constantes e completamos a fórmula:

- 105937: $y = 7 * 10^{-8} * e^{x*0.4382}$
- 109018: $y = 4.32 * 10^{-8} * e^{x*0.4542}$

Portanto, a fórmula pode ser descrita de forma geral por uma média entre ambas:

$$y = 5.66 * 10^{-8} * e^{x*0.4467}$$

Esse resultado confirma o referido acima sobre a complexidade, visto que a função y encontrada é da ordem $e^{x*0.4467} \cong \sqrt{e}^{x}$.

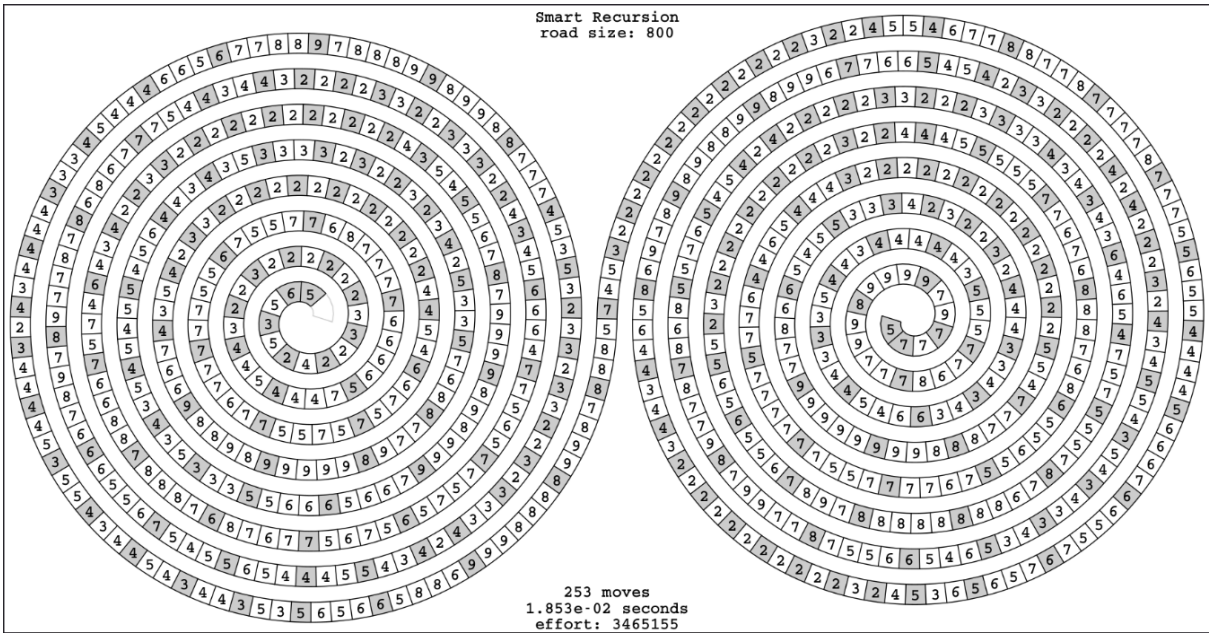
Plain Recursion: Estimativa para a posição final de 800

A partir dessa fórmula, foi possível fazer uma estimativa de quanto tempo demoraria para essa solução encontrar uma resposta para a posição final de 800. Ao substituir o x por 800, obtemos o tempo de $\mathbf{8.95967969603990 * 10^{147}}$ **segundos**, o que corresponde a $\mathbf{2.839151169936851 * 10^{140}}$ **anos** (ao usar a conversão 1 segundo = $3.1688087814029 * 10^{8}$ anos, retirada de https://www.unitconverters.net/time/second-to-year.htm, Acesso em 04/12/22). A critério de comparação, o número estimado de átomos presentes no universo é igual a $1.201 * 10^{79}$!! (retirado de https://corujasabia.com/quantos-atomos-existem-no-universo/, Acesso em 04/12/22). Isso mostra que nunca chegaríamos a esse número, e prova a ineficiência do algoritmo.

Smart Recursion

Este algoritmo contém a primeira optimização feita ao código original, o que já o torna bastante mais eficiente, e o permite chegar à posição final de 800, como mostra a figura a seguir.

Figura 4: PDF gerado pelo método *Smart Recursion* para a posição final 800.

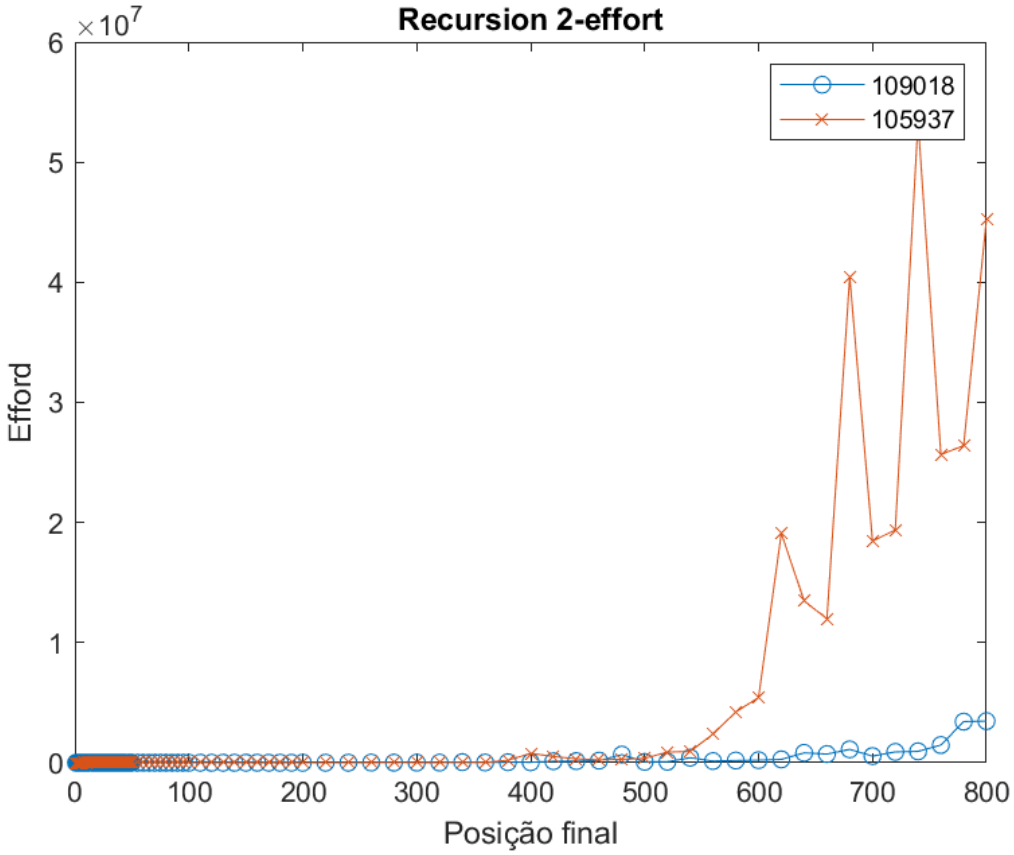


(Fonte: autoria própria do grupo, com o uso do código *make_custom_pdf.c* disponibilizado pelo professor Tomás Oliveira Silva, 2022)

Em seu gráfico, nota-se que o tempo de execução varia muito entre números mecanográficos, *i.e.*, trajetos diferentes. Essa variação acontece porque o código começa por tentar primeiro as velocidades mais lentas (*new_speed = speed - 1*), o que implica a condição do *if* ser executada apenas em situações específicas em que uma velocidade mais lenta gerou uma solução com número de movimentos inferior a uma velocidade mais rápida, o que é causado exclusivamente pela disposição de velocidades máximas limites, própria de cada trajeto. A variação também é perceptível no gráfico a

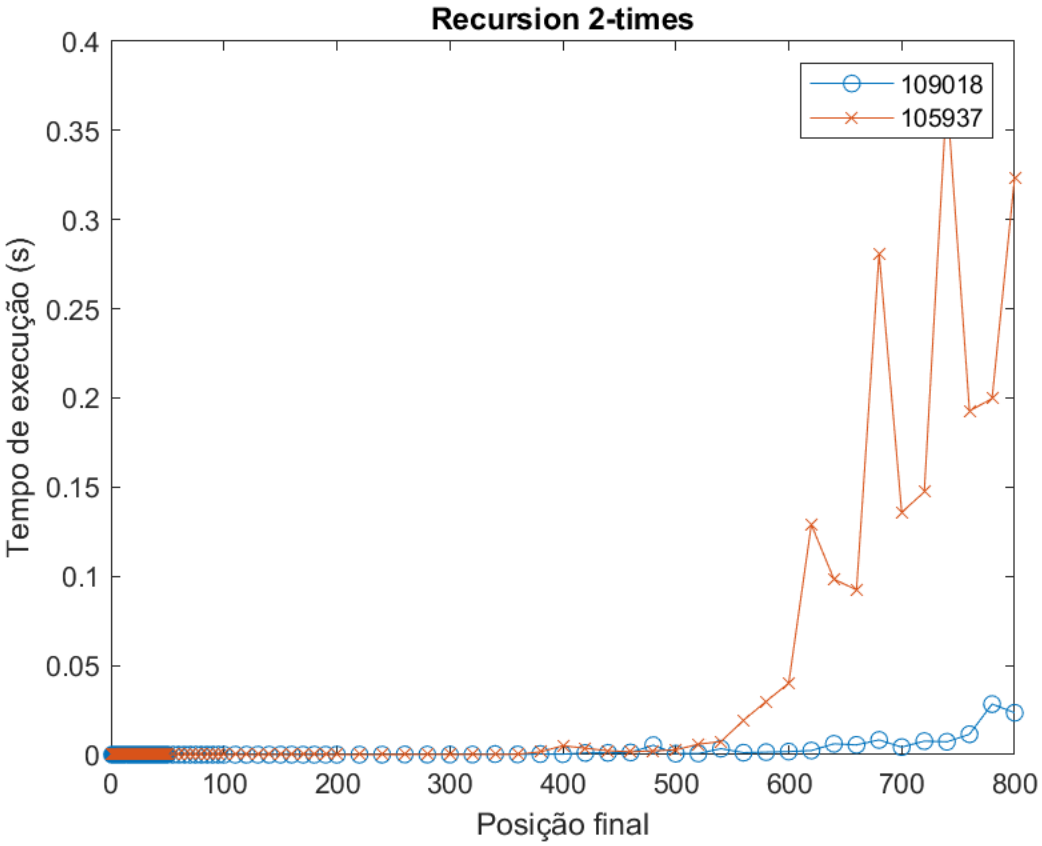
seguir, em que os valores do eixo y correspondem ao número de vezes que a função recursiva foi executada.

Figura 5: Gráfico do esforço despendido pelo método *Smart Recursion.*



(Fonte: autoria própria do grupo, com o uso do MATLAB (código presente no apêndice), 2022)

Figura 6: Gráfico do tempo de execução do método *Smart Recursion.*



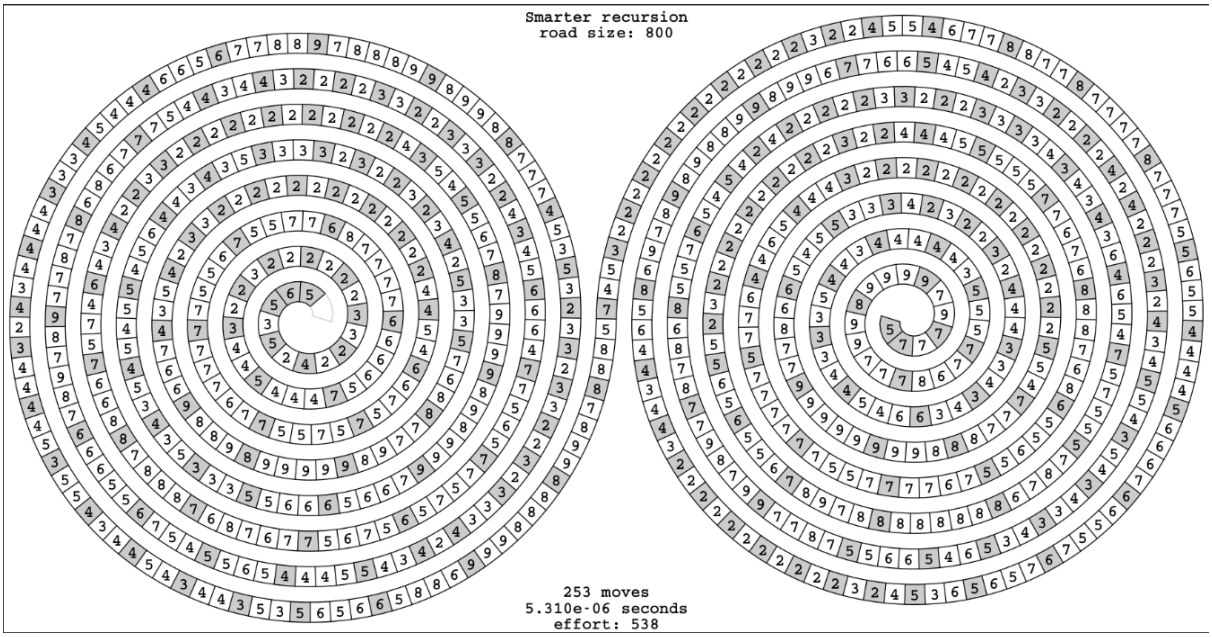
(Fonte: autoria própria do grupo, com o uso do MATLAB (código presente no apêndice), 2022)

A partir do gráfico, tira-se que a complexidade computacional desse algoritmo com certeza não é linear, e sabe-se que também não é exponencial pois chega até 800. Porém, não é possível estabelecer uma complexidade exata, visto a variação do gráfico e a imprevisibilidade da quantidade de chamadas à função.

Smarter recursion

Este algoritmo contém a segunda optimização feita ao código original, o que faz ele ser um dos dois melhores algoritmos implementados, e o permite chegar à posição final de 800 em microssegundos, como mostra a figura a seguir.

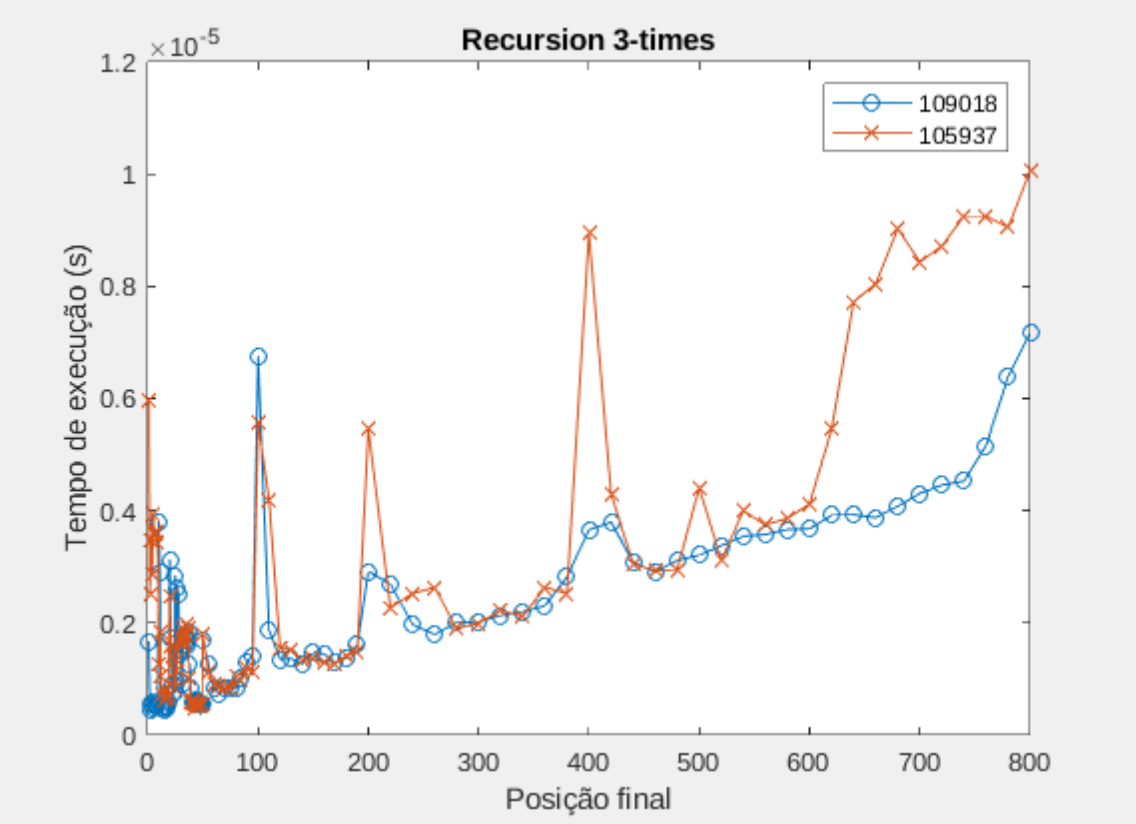
Figura 7: PDF gerado pelo método *Smarter Recursion* para a posição final 800.



(Fonte: autoria própria do grupo, com o uso do código *make_custom_pdf.c* disponibilizado pelo professor Tomás Oliveira Silva, 2022)

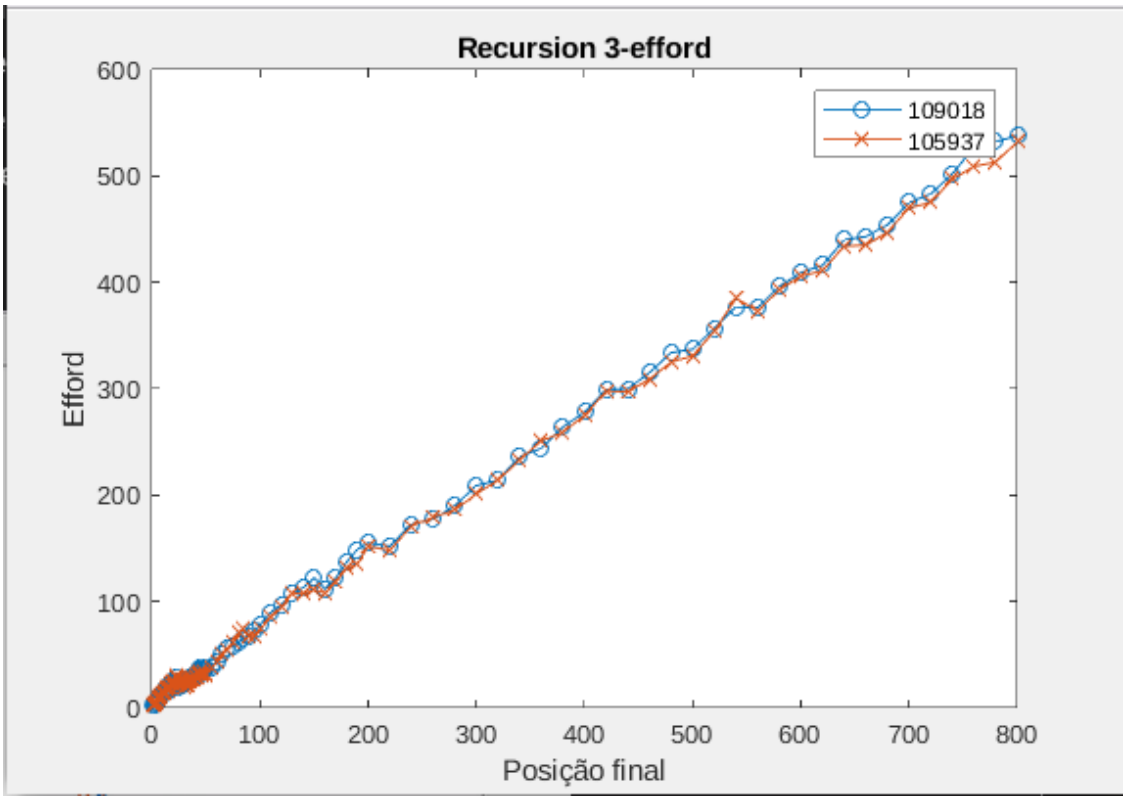
Ao analisar o gráfico abaixo, conclui-se que o algoritmo tem uma complexidade O(n), pois tem um aspecto linear, apesar dos picos. Estes podem ser explicados pelo funcionamento do sistema operativo (mudanças de contexto), e não aparecem no gráfico do esforço. Para além disso, ao contrário da implementação anterior, o esforço para os dois números mecanográficos tem uma variação muito pequena, o que mostra que esse algoritmo é determinístico e possui uma complexidade linear bem definida.

Figura 8: Gráfico do tempo de execução do método *Smarter Recursion.*



(Fonte: autoria própria do grupo, com o uso do MATLAB (código presente no apêndice), 2022)

Figura 9: Gráfico do esforço despendido pelo método *Smarter Recursion.*

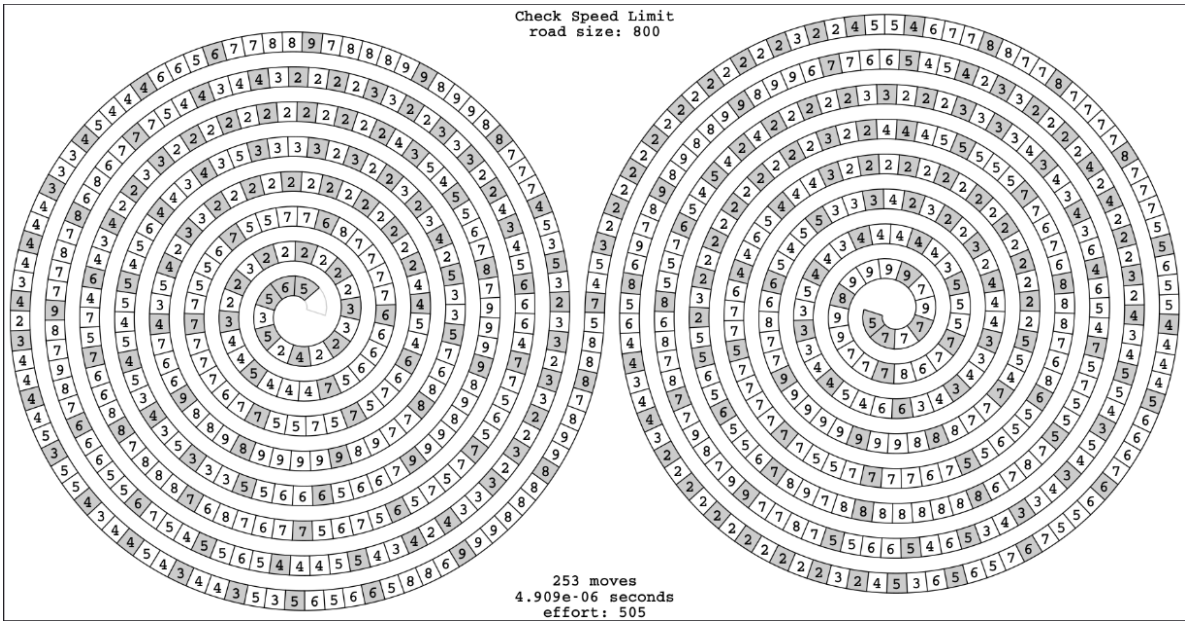


(Fonte: autoria própria do grupo, com o uso do MATLAB (código presente no apêndice), 2022)

Check Speed Limit

Como referido anteriormente, esta solução é uma das duas mais rápidas. Esta solução consegue chegar à posição 800 também em 10^{-6} segundos, como é mostrado na imagem seguinte.

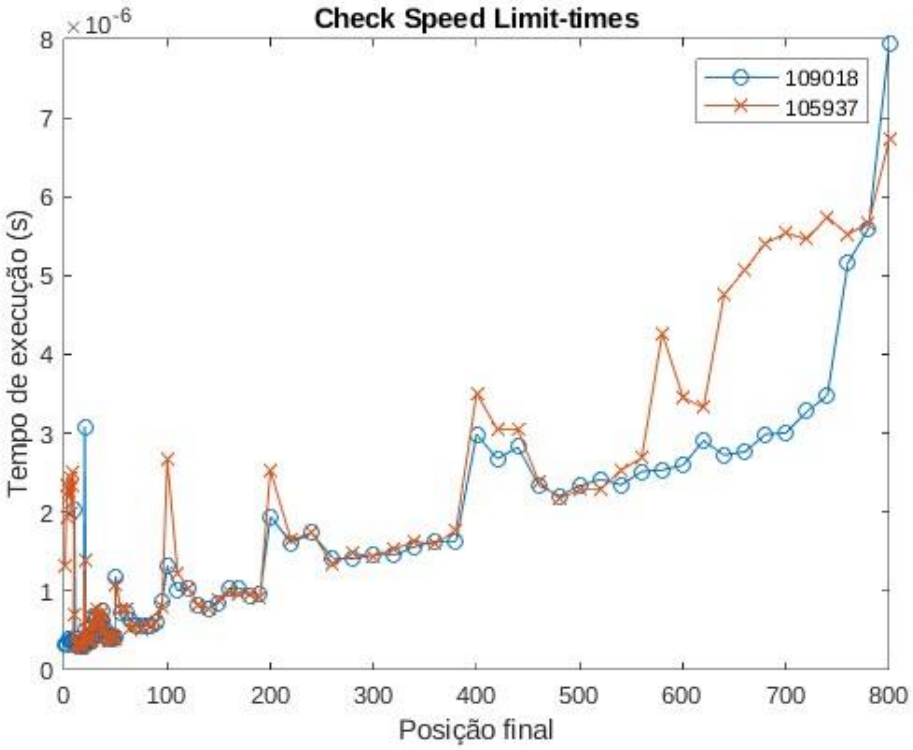
Figura 10: PDF gerado pelo método *Check Speed Limit* para a posição final 800.



(Fonte: autoria própria do grupo, com o uso do código *make_custom_pdf.c* disponibilizado pelo professor Tomás Oliveira Silva, 2022)

Assim como o anterior, esse algoritmo tem uma complexidade O(n), o que pode ser verificado no gráfico abaixo e através da análise do código, em que o ciclo *for* é executado 3n vezes, sendo n o número de movimentos feitos até ao final.

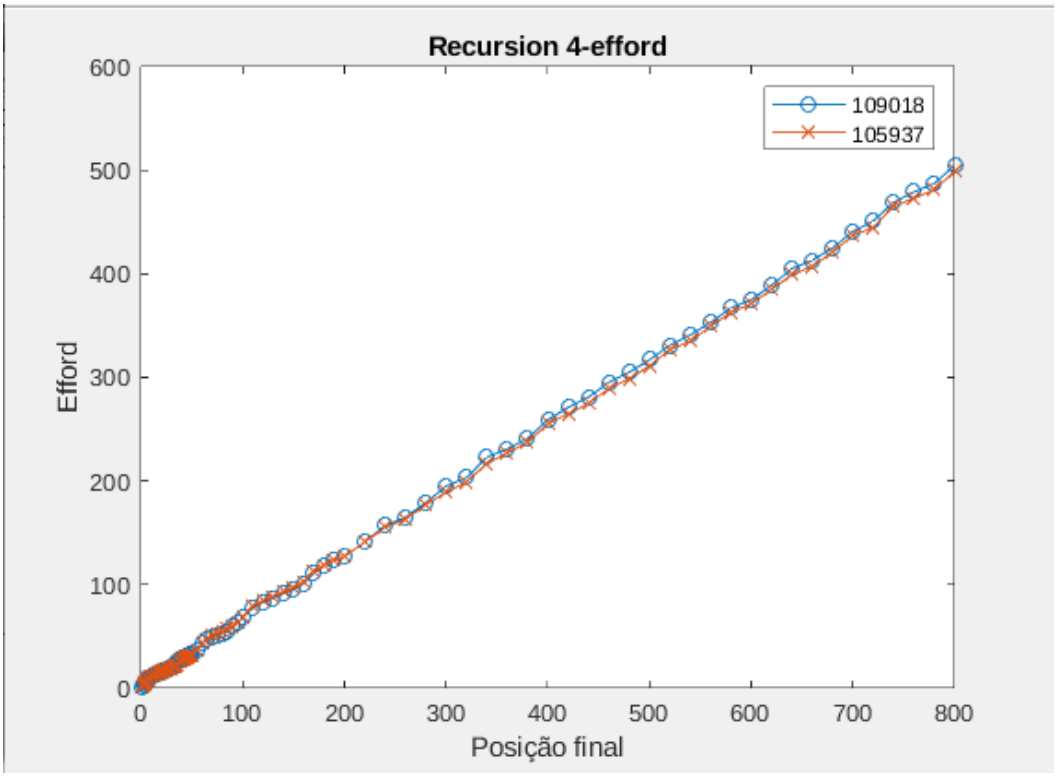
Figura 11: Gráfico do tempo de execução do método *Check Speed Limit.*



(Fonte: autoria própria do grupo, com o uso do MATLAB (código presente no apêndice), 2022)

Pela análise do gráfico do esforço, gráfico este que, para essa solução, mede quantas vezes a função auxiliar foi chamada, é possível concluir que a variação é também muito baixa entre as soluções geradas para os dois números mecanográficos.

Figura 12: Gráfico do esforço despendido pelo método *Check Speed Limit.*

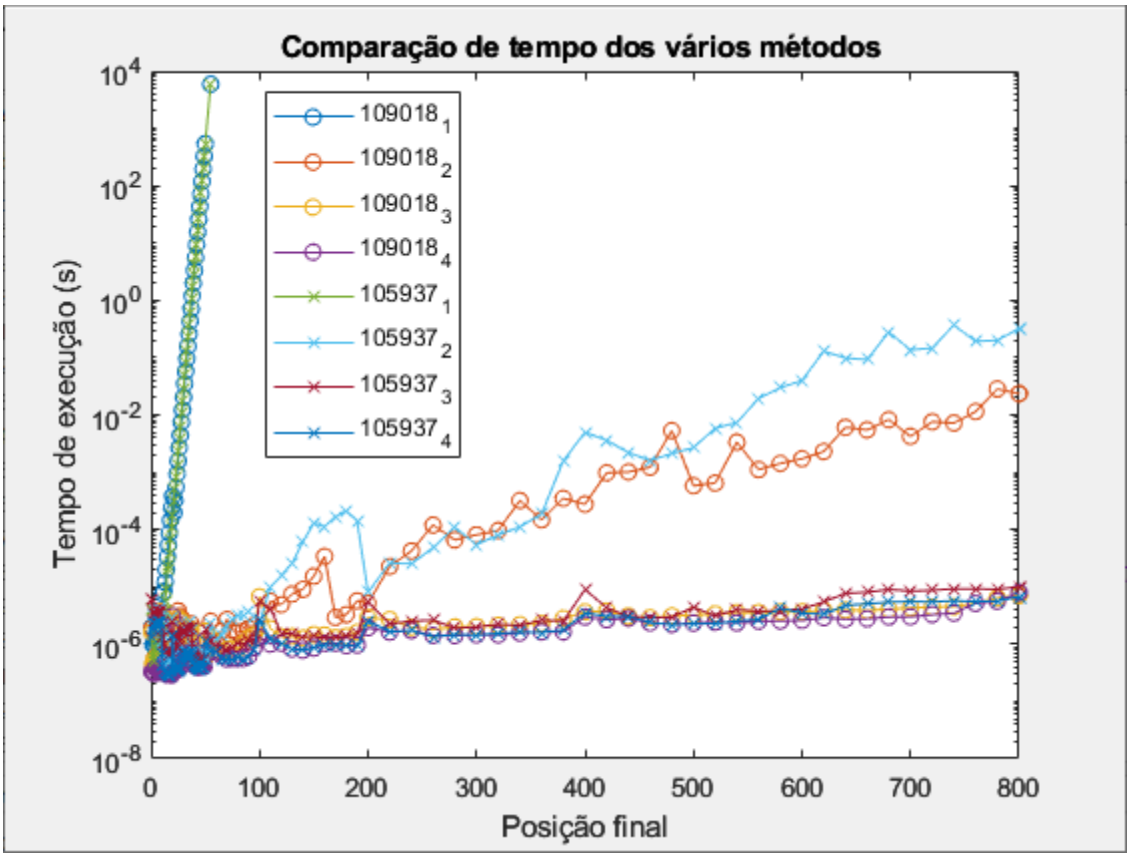


(Fonte: autoria própria do grupo, com o uso do MATLAB (código presente no apêndice), 2022)

Comparação das várias implementações

Portanto, conclui-se que as soluções *Smarter Recursion* (3) e *Check Speed Limit* (4) são as soluções mais eficientes. Em terceiro lugar, está a solução *Smart Recursion* (2), e em último, *Plain Recursion* (1). O gráfico a seguir evidencia esse cenário:

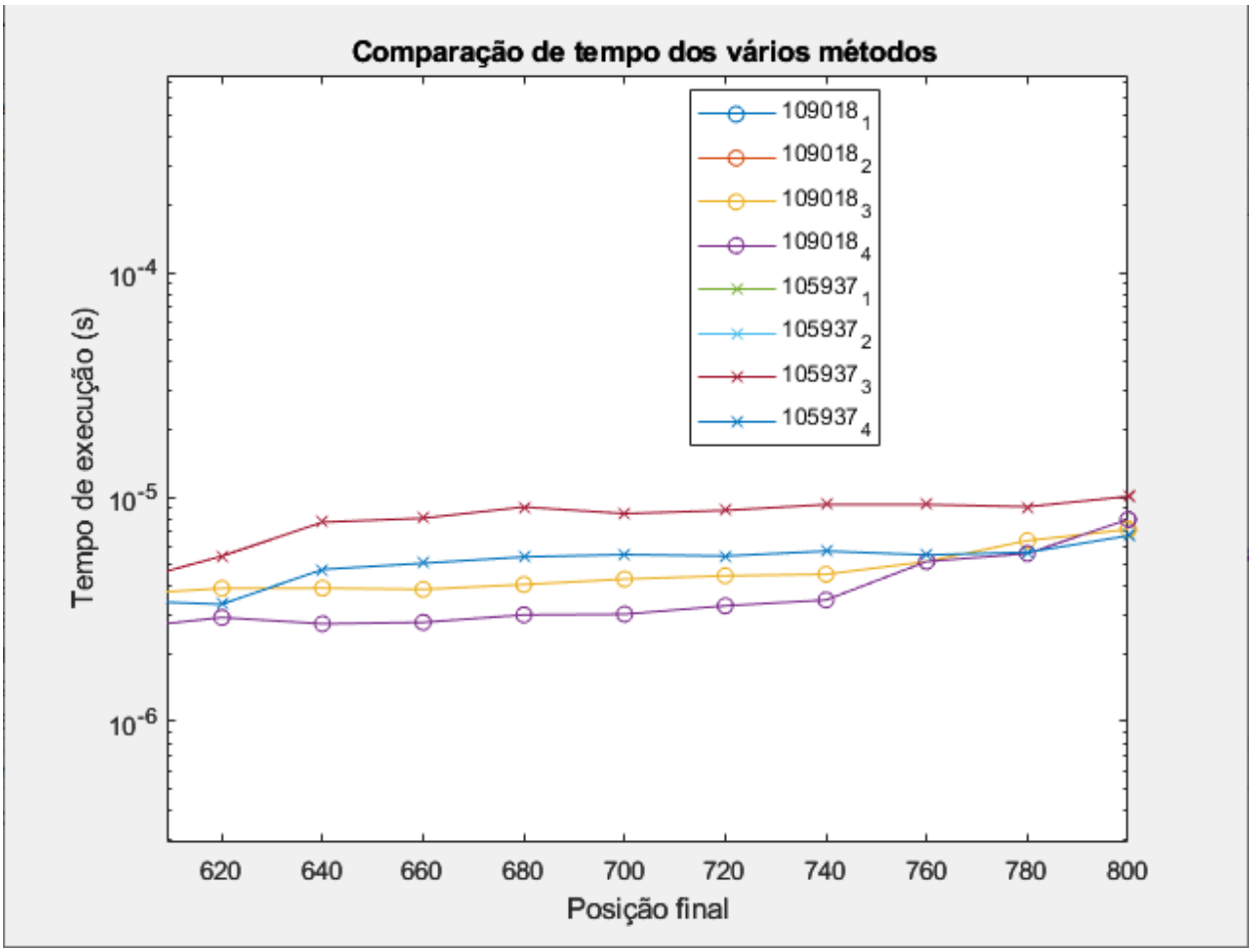
Figura 13: Gráfico semi-log com os tempos de execução de cada método descrito.



(Fonte: autoria própria do grupo, com o uso do MATLAB (código presente no apêndice), 2022)

Nota-se que as soluções 3 e 4 são muito próximas em termos de eficiência, apesar de uma ser recursiva e a outra não. No gráfico ampliado a seguir, nota-se ainda que a solução mais rápida entre as duas varia para o trajeto criado (número mecanográfico).

Figura 14: Gráfico semi-log com os tempos de execução de cada método descrito ampliado.



Webgrafia

Site CorujaSabia.com (2019). Disponível em:

https://corujasabia.com/quantos-atomos-existem-no-universo/

Acesso em 01/12/2022

Site UnitConverters.net (2022). Disponível em:

https://www.unitconverters.net/time/second-to-year.htm

Acesso em 01/12/2022

Apêndice

Código C

```c
//
// 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. 105937  Name: Bárbara Nóbrega Galiza
//   N.Mec. 109018  Name: Tomás António de Oliveira Victal


//
// static configuration
//

#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 "../P02/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,solution_2,solution_2_best,
solution_3,solution_3_best,solution_4_best;
```

```
static double solution_1_elapsed_time, solution_2_elapsed_time,
solution_3_elapsed_time, solution_4_elapsed_time; // time it took to solve the
problem
static unsigned long solution_1_count, solution_2_count , solution_3_count,
solution_4_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;
}

//
// solution 2
//
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;
  }
  // was the best solution ahead of me in the same ammount of moves?
  if(solution_2_best.positions[move_number] > solution_2.positions[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_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;
}

//
// solution 3
//
static void 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;
  }

  // was the best solution ahead of me in the same ammount of moves?
  if(solution_3_best.positions[move_number] > solution_3.positions[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_3_recursion(move_number + 1,position +
new_speed,new_speed,final_position);
    }
}

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;
}

//
// solution 4
//
static int checkSpeedLimit(int speed,int check_size,int current_pos,int
road_size){
  int check_speed = speed;
  int counter = speed+1;
  if(check_size+current_pos > road_size) return 0;
  for(int i= 0; i<check_size+1;i++){
    if(max_road_speed[current_pos+i] < check_speed){
      return 0;
    }
    counter--;
    if (counter <= 0){
      check_speed = check_speed - 1;
      counter = check_speed;
    }
  }
  return 1;
}
static void solution_4_checkSpeedLimit(int final_position)
{
  int position = 0;
```

```
  int check_quantaty[] = {0,1,3,6,10,15,21,28,36,45};
  int new_speed;
  int speed = 0;
  int move_number = 0;
  solution_4_best.positions[move_number] = position;
  while(position < final_position){
    move_number++;
    for(new_speed = speed+1;new_speed >= speed-1;new_speed--){
        solution_4_count++;
      if(checkSpeedLimit(new_speed,check_quantaty[new_speed],position,final_posit
ion)){
        position = position + new_speed;
        speed = new_speed;
        break;
      }
    }
    solution_4_best.positions[move_number] = position;
  }
  solution_4_best.n_moves = move_number;
}

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);
  }
  solution_4_elapsed_time = cpu_time();
  solution_4_count = 0ul;
  solution_4_best.n_moves = final_position + 100;
  solution_4_checkSpeedLimit(final_position);
  solution_4_elapsed_time = cpu_time() - solution_4_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_co
unt,"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_  5.0
  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);

    // first solution method (very bad)
    if(solution_1_elapsed_time < _time_limit_)
    {
      solve_1(final_position);
      if(print_this_one != 0)
      {
        sprintf(file_name,"%03d_1.pdf",final_position);
        make_custom_pdf_file(file_name,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(" %3d %16lu %9.3e
|",solution_1_best.n_moves,solution_1_count,solution_1_elapsed_time);
    }
    else
    {
      solution_1_best.n_moves = -1;
      printf("                                |");
    }

    // second solution method (less bad)
    if(solution_2_elapsed_time < _time_limit_)
    {
      solve_2(final_position);
      if(print_this_one != 0)
      {
        sprintf(file_name,"%03d_2.pdf",final_position);
        make_custom_pdf_file(file_name,final_position,&max_road_speed[0],solution
_2_best.n_moves,&solution_2_best.positions[0],solution_2_elapsed_time,solution_2_
count,"Smart Recursion");
      }
      printf(" %3d %16lu %9.3e
|",solution_2_best.n_moves,solution_2_count,solution_2_elapsed_time);
    }
    else
    {
      solution_2_best.n_moves = -1;
      printf("                                |");
    }
```

```
    // third solution method
    if(solution_3_elapsed_time < _time_limit_)
    {
      solve_3(final_position);
      if(print_this_one != 0)
      {
        sprintf(file_name,"%03d_3.pdf",final_position);
        make_custom_pdf_file(file_name,final_position,&max_road_speed[0],solution
_3_best.n_moves,&solution_3_best.positions[0],solution_3_elapsed_time,solution_3_
count,"Smarter recursion");
      }
      printf(" %3d %16lu %9.3e
|",solution_3_best.n_moves,solution_3_count,solution_3_elapsed_time);
    }
    else
    {
      solution_3_best.n_moves = -1;
      printf("                                |");
    }

    // fourth solution method
    if(solution_4_elapsed_time < _time_limit_)
    {
      solve_4(final_position);
      if(print_this_one != 0)
      {
        sprintf(file_name,"%03d_4.pdf",final_position);
        make_custom_pdf_file(file_name,final_position,&max_road_speed[0],solution
_4_best.n_moves,&solution_4_best.positions[0],solution_4_elapsed_time,solution_4_
count,"Check Speed Limit");
      }
      printf(" %3d %16lu %9.3e
|",solution_4_best.n_moves,solution_4_count,solution_4_elapsed_time);
    }
    else
    {
      solution_4_best.n_moves = -1;
      printf("                                |");
    }
    // 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_
}
```

Código MATLAB: *Get_graphs*

```
%% load data
load("table_values.mat")
sol1_109018_t = readtable("sol1_55_109018.txt");
sol1_105937_t = readtable("sol1_55_105937.txt");
n1 = [1:50, 55];
n2 = table2array(testes109018(:,1));
%% arrays sol_1
n_moves_109018_1 = table2array(sol1_109018_t([1:51],3));
efford_109018_1 = table2array(sol1_109018_t([1:51],4));
times_109018_1 = table2array(sol1_109018_t([1:51],5));
n_moves_105937_1 = table2array(sol1_105937_t([1:51],3));
efford_105937_1 = table2array(sol1_105937_t([1:51],4));
times_105937_1 = table2array(sol1_105937_t([1:51],5));
%% arrays sol_2
n_moves_109018_2 = table2array(testes109018(:,7));
efford_109018_2 = table2array(testes109018(:,8));
times_109018_2 = table2array(testes109018(:,9));
n_moves_105937_2 = table2array(testes105937(:,7));
efford_105937_2 = table2array(testes105937(:,8));
```

```
times_105937_2 = table2array(testes105937(:,9));
%% arrays sol_3
n_moves_109018_3 = table2array(testes109018(:,11));
efford_109018_3 = table2array(testes109018(:,12));
times_109018_3 = table2array(testes109018(:,13));
n_moves_105937_3 = table2array(testes105937(:,11));
efford_105937_3 = table2array(testes105937(:,12));
times_105937_3 = table2array(testes105937(:,13));
%% arrays sol_4
n_moves_109018_4 = table2array(testes109018(:,15));
efford_109018_4 = table2array(testes109018(:,16));
times_109018_4 = table2array(testes109018(:,17));
n_moves_105937_4 = table2array(testes105937(:,15));
efford_105937_4 = table2array(testes105937(:,16));
times_105937_4 = table2array(testes105937(:,17));
%% sol_1
figure(1)
plot(n1,efford_109018_1,"-o");
hold on
plot(n1,efford_105937_1,"-x")
legend("109018","105937")
title("Recursion 1-efford")
xlabel("Posição final")
ylabel("Efford")
figure(12)
plot(n1,times_109018_1,"-o");
hold on
plot(n1,times_105937_1,"-x")
plot(n1,2.^n1)
plot(n1,exp(n1/2))
legend("109018","105937","2^n","sqrt(e)^n")
```

```
title("Plain Recursion")
xlabel("Posição final")
ylabel("Tempo de execução (s)")
axis([0 60 0 6010])

%% sol_2
figure(2)
plot(n2,efford_109018_2,"-o");
hold on
plot(n2,efford_105937_2,"-x")
legend("109018","105937")
title("Recursion 2-efford")
xlabel("Posição final")
ylabel("Efford")
figure(22)
plot(n2,times_109018_2,"-o");
hold on
plot(n2,times_105937_2,"-x")
legend("109018","105937")
title("Recursion 2-times")
xlabel("Posição final")
ylabel("Tempo de execução (s)")
%% sol_3
figure(3)
plot(n2,efford_109018_3,"-o");
hold on
plot(n2,efford_105937_3,"-x")
legend("109018","105937")
title("Recursion 3-efford")
xlabel("Posição final")
ylabel("Efford")
```

```
figure(32)
plot(n2,times_109018_3,"-o");
hold on
plot(n2,times_105937_3,"-x")
legend("109018","105937")
title("Recursion 3-times")
xlabel("Posição final")
ylabel("Tempo de execução (s)")
%% sol_4
figure(4)
plot(n2,efford_109018_4,"-o");
hold on
plot(n2,efford_105937_4,"-x")
legend("109018","105937")
title("Recursion 4-efford")
xlabel("Posição final")
ylabel("Efford")
figure(42)
plot(n2,times_109018_4,"-o");
hold on
plot(n2,times_105937_4,"-x")
legend("109018","105937")
title("Check Speed Limit-times")
xlabel("Posição final")
ylabel("Tempo de execução (s)")
%% all times
figure(5)
semilogy(n1,times_109018_1,"-o");
hold on;
semilogy(n2,times_109018_2,"-o");
semilogy(n2,times_109018_3,"-o");
```

```
semilogy(n2,times_109018_4,"-o");

semilogy(n1,times_105937_1,"-x")

semilogy(n2,times_105937_2,"-x")

semilogy(n2,times_105937_3,"-x")

semilogy(n2,times_105937_4,"-x")

legend("109018_1","109018_2","109018_3","109018_4","105937_1","105937_2","105937_3","
105937_4")

title("Comparação de tempo dos vários métodos ")

xlabel("Posição final")

ylabel("Tempo de execução (s)")
```

Código MATLAB: *LeastSquaresFit*

```
%% Student Number 109018

load("table_values.mat")

sol1_t = readtable("sol1_55_109018.txt");


%% Student Number 105937

load("table_values.mat")

%sol1_t = readtable("sol1_55.txt");


%% Least Squares Fit

x = table2array(sol1_t([1:51],1)); % extract the first column

y = table2array(sol1_t([1:51],5)); % extract the 5th column

y2 = log(y);

X = [ 0*x+1, x ]; % build the X matrix

w = pinv(X)*y2; % optimal solution (could also be written as w = X \ y;)

e = y2-X*w; % optional: compute the errors vector

format long

w % print w --- A = w(1), B = w(2), and C = w(3)
```

```
norm(e) % optional: print the norm of the error vector (square root of the sum of squares)

plot(x,y2,'.r',x,X*w,'og'); % plot the original data and its best least squares approximation


a = exp(w(1));

b = w(2);

seconds = a*exp(800*b)

years = seconds*3.1688087814029e-8
```