Uma imagem com texto

Descrição gerada automaticamente

**Speed Run**

Algoritmos e Estruturas de Dados 2022

Professor Tomás Silva e Professor Pedro Lavrador

Trabalho realizado por:

João Nuno da Silva Luís (107403) | 50%

Diana Raquel Rodrigues Miranda (107457) | 50%

**Índice**

[Índice de figuras 3](#_Toc121395696)

[Introdução 4](#_Toc121395697)

[Algoritmo fornecido 5](#_Toc121395698)

[Segundo Algoritmo 12](#_Toc121395699)

[Terceiro Algoritmo – Programação Dinâmica 14](#_Toc121395700)

[Resultados 18](#_Toc121395701)

[ **Tempos de execução finais** 20](#_Toc121395702)

[Conclusões finais 21](#_Toc121395703)

[Web grafia 22](#_Toc121395704)

[Apêndice 23](#_Toc121395705)

[**Código C** 23](#_Toc121395706)

[ Speed\_run.c 23](#_Toc121395707)

[**Código MATLAB** 33](#_Toc121395708)

[ Execution\_time.m 33](#_Toc121395709)

# **Índice de figuras**

[Figura 1 – Exemplo gráfico da árvore percorrida pela função fornecida. 5](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337282)

[Figura 2- Tempo de execução da solução fornecida 5](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337283)

[Figura 3- Reta de ajuste aos dados da solução fornecida 6](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337284)

[Figura 4- Comparação da reta de ajuste até à posição n=800 8](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337285)

[Figura 5- Reta de ajuste aos dados da 2ª melhoria 9](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337286)

[Figura 6- Comparação das diferentes retas de ajuste. 9](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337287)

[Figura 7 - Reta de ajuste aos dados da 3ª melhoria 10](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337288)

[Figura 8- Comparação das diferentes retas de ajuste. 11](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337289)

[Figura 9 - Função da Solução 2 12](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337290)

[Figura 10 - Função usada na solução 2 para verificar a velocidade 13](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337291)

[Figura 11 - Reta de ajuste aos dados do 2º algoritmo 13](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337292)

[Figura 12 - 10 segmentos 14](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337293)

[Figura 13 - 20 segmentos 14](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337294)

[Figura 14 - 30 segmentos 14](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337295)

[Figura 15 - Estrutura criada para a solução 3 15](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337296)

[Figura 16 - Função da Solução 3 15](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337297)

[Figura 17 - Função usada na solução 3 para verificar a velocidade 16](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337298)

[Figura 18 - Função que chama a função da solução 3 16](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337299)

[Figura 19- Dados do 3º algoritmo 17](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337300)

[Figura 20 - Resultados com solução do professor otimizada 18](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337301)

[Figura 21 - Resultados com a segunda solução 18](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337302)

[Figura 22 - Resultados com a terceira solução (Programação dinâmica) 19](file:///C:\LEI\2Ano\AED\Project_AED\Report.docx#_Toc121337303)

# **Introdução**

Este trabalho foi realizado no âmbito da disciplina de Algoritmos e Estruturas de Dados do 2º ano da Licenciatura em Engenharia Informática.

Foi-nos proposto desenvolver um algoritmo em C que determinasse o número mínimo de movimentos necessários para alcançar a posição final. No entanto, têm de ser respeitadas certas regras, que são as seguintes:

- O carro só pode aumentar 1 velocidade, reduzir 1 velocidade ou mantê-la;

- O carro começa no primeiro segmento da estrada com uma velocidade 0 e tem de atingir o último segmento de estrada com uma velocidade de um;

- O carro não pode em nenhum momento passar num segmento de estrada com uma velocidade superior à nela permitida.

Com isto em mente, temos que a finalidade principal do nosso trabalho é conseguir otimizar o algoritmo fornecido pelo professor de modo a tornar possível atingir a posição 800. Para além disto, se for possível, desenvolver um novo algoritmo que resolva o problema com o menor tempo de execução possível.

# **Algoritmo fornecido**

O algoritmo fornecido segue o conceito de depth first search, que é um algoritmo utilizado para realizar uma procura em árvore, estrutura de árvore ou grafo. Intuitivamente, o algoritmo começa num nó raiz e explora tanto quanto possível cada um dos seus ramos, antes de retroceder.

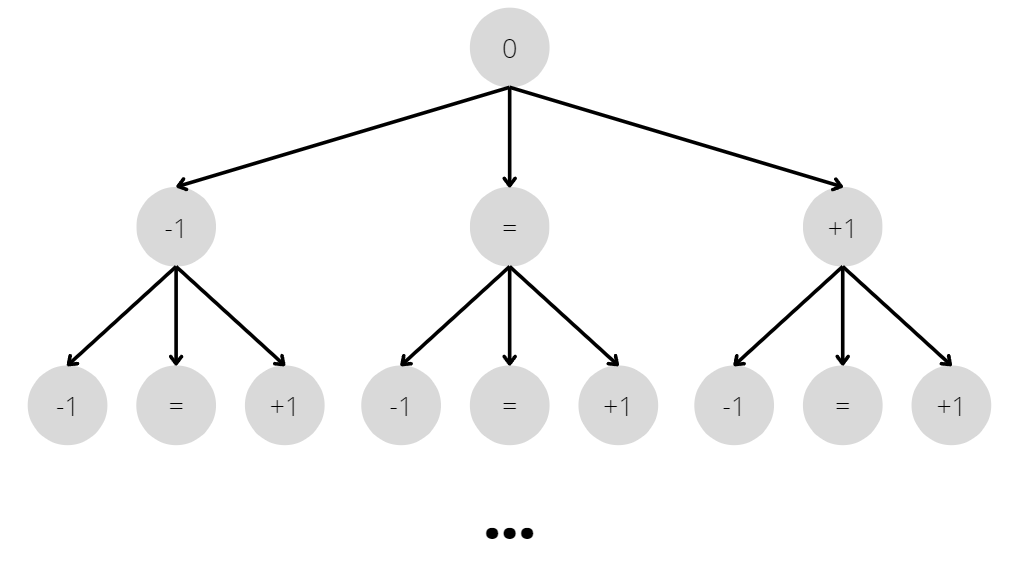
Posto isto, temos no nosso caso de estudo (Speed run) o nó raiz como a primeira posição de onde o carro irá arrancar e desse nó irão sair três novos nós, um com a opção de aumentar a velocidade, um com a opção de a manter e outro com a opção de a diminuir, e só depois de todas as possibilidades terem sido percorridas, é que o algoritmo vai retroceder e escolher o melhor caminho.

Figura 1 – Exemplo gráfico da árvore percorrida pela função fornecida.

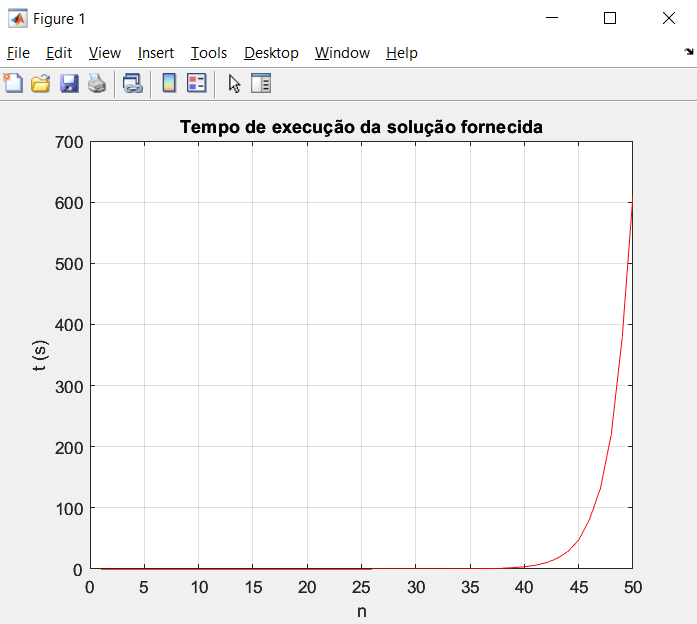
Fazendo o gráfico do tempo de execução do algoritmo fornecido, antes de fazer qualquer otimização, obtemos um gráfico exponencial a partir da posição 40.

Figura 2- Tempo de execução da solução fornecida

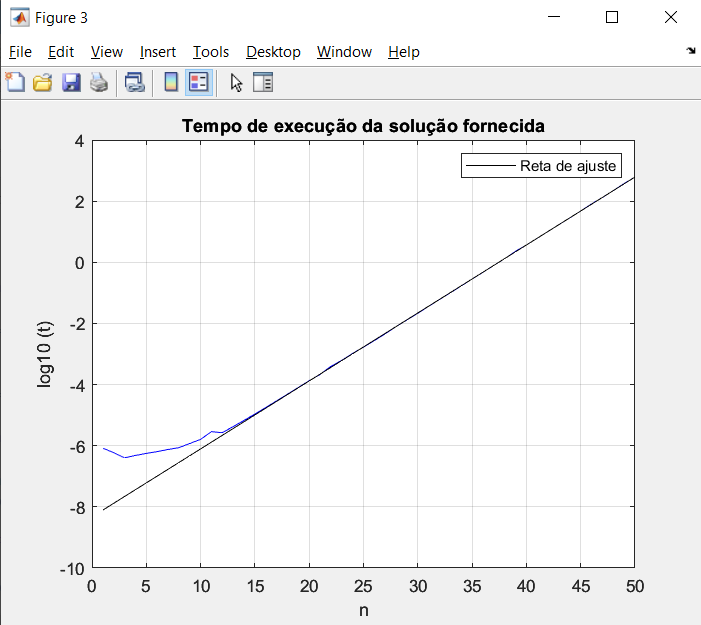
Este tipo de gráfico não é o melhor para visualizarmos os nossos dados, nem calcular a reta de ajuste, pelo que podemos aplicar um logaritmo na base dez ao eixo dos yy (eixo dos tempos), e ao mesmo tempo calcular a reta de ajuste aos dados obtidos, que nos permite fazer uma previsão do tempo de execução deste algoritmo até à posição final.

Figura 3- Reta de ajuste aos dados da solução fornecida

Através desta resta de ajuste, calculamos que a solução fornecida iria demorar 1.114e+162 segundos a chegar à posição 800.

Estes gráficos provam que apesar do algoritmo ser capaz de chegar à solução correta, o mesmo irá demorar muitíssimo tempo a resolver o problema para as 800 posições.

O nosso método para tornar este algoritmo mais eficiente foi pensar numa maneira de otimizar a pesquisa em árvore e diminuir o número de ramos visitados.

**1ª melhoria** – Tentar acelerar primeiro.

Como o nosso objetivo é chegar à posição final o mais rápido possível, vamos visitar primeiro o nó em que se verifica o aumento da velocidade e depois visitamos os outros.

Uma imagem com texto

Descrição gerada automaticamentePara isso, alterámos o seguinte pedaço de código:

Uma imagem com texto

Descrição gerada automaticamentePara:

Assim, vai começar a sua pesquisa sempre pelo nó que acelera.

Com o código fornecido, sem nenhuma alteração, e em execução durante uma hora (com o número mecanográfico 107457) é possível chegar à posição 50 em 6.121e+02 segundos. Com esta alteração foi possível, durante o mesmo tempo de execução, chegar à posição 50 em 5.935e+02 segundos. É uma melhoria mínima, pois mesmo a começar a pesquisa pelo nó que acelera o algoritmo vai verificar todos os ramos da árvore possíveis, o que não melhora muito o tempo de execução.

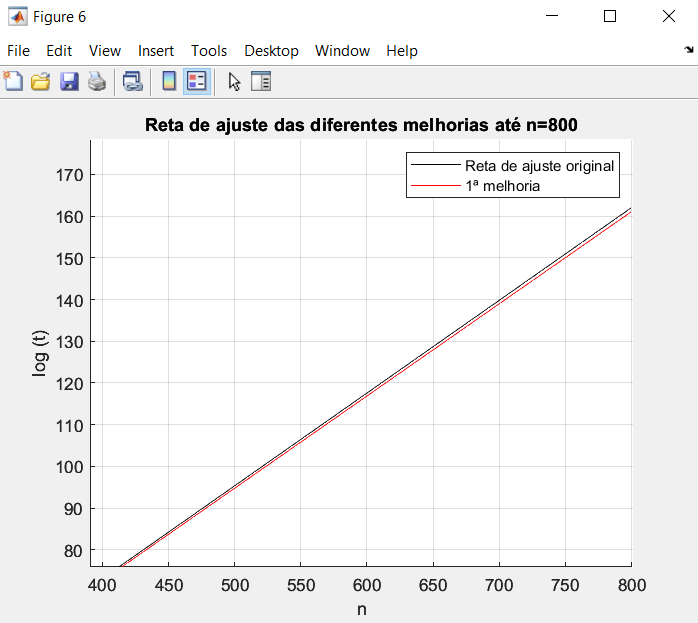
****

Figura 4- Comparação da reta de ajuste até à posição n=800

**2ª melhoria** – Acrescentar um if no código da função fornecida.

Uma imagem com texto

Descrição gerada automaticamente Este if verifica se o número de movimentos da solução que está a ser vista nesse momento já é maior do que o número de movimentos total da “melhor” solução anteriormente guardada. Se for maior, então o algoritmo pode parar essa procura, pois já não nos interessa uma vez que já temos uma solução melhor, o que torna possível cortar alguns ramos da árvore.

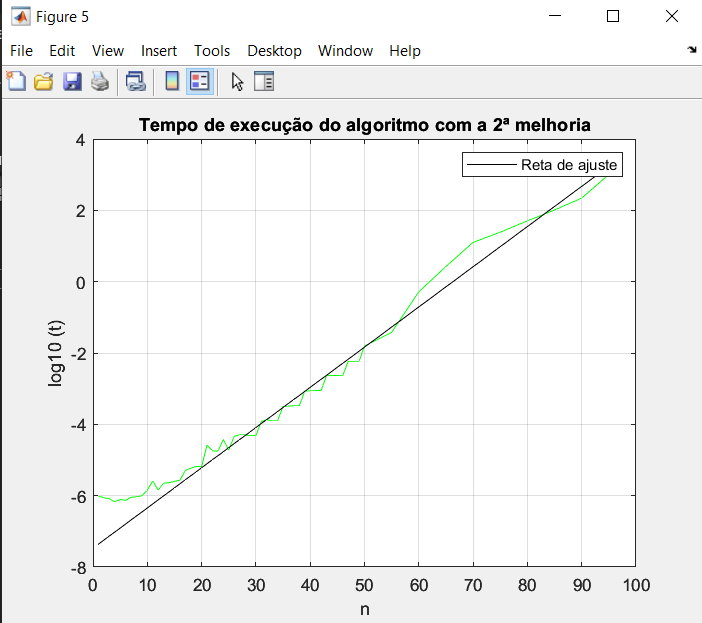
 Apesar de esta melhoria ser bastante simples, pois só acrescentámos duas linhas de código, é uma melhoria que nos permite reduzir o tempo de execução para metade. Com isto, durante uma hora de execução (com o número mecanográfico 107457), já conseguimos chegar à posição 95 em 1.009e+03 segundos.

Figura 5- Reta de ajuste aos dados da 2ª melhoria

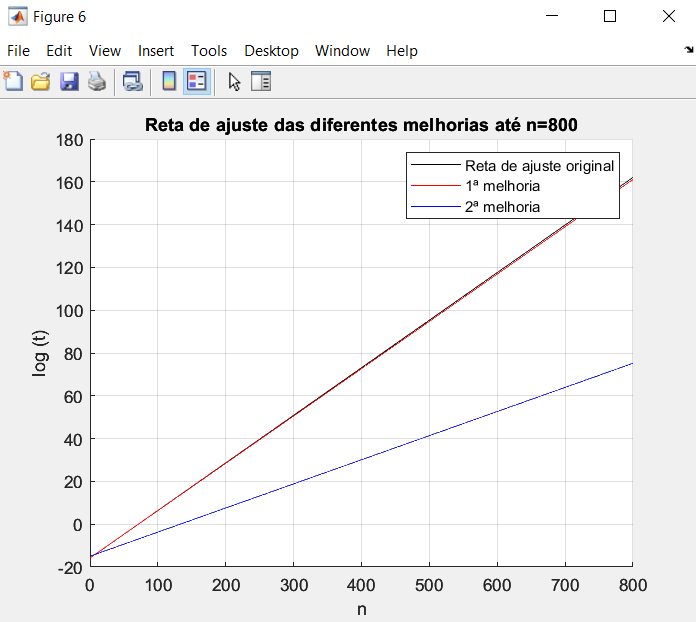


Figura 6- Comparação das diferentes retas de ajuste.

A 2º melhoria permite reduzir o tempo de execução para metade.

**3ª melhoria** – Acrescentar um outro if no código da função fornecida.

Uma imagem com texto

Descrição gerada automaticamente

Neste segundo if, vai ser verificado se numa determinada posição foi possível, na solução já guardada, passar com uma velocidade maior do a que está a ser vista naquele momento. Se tiver sido possível então podemos abandonar a pesquisa desse ramo, pois interessa-nos andar sempre com a velocidade máxima possível.

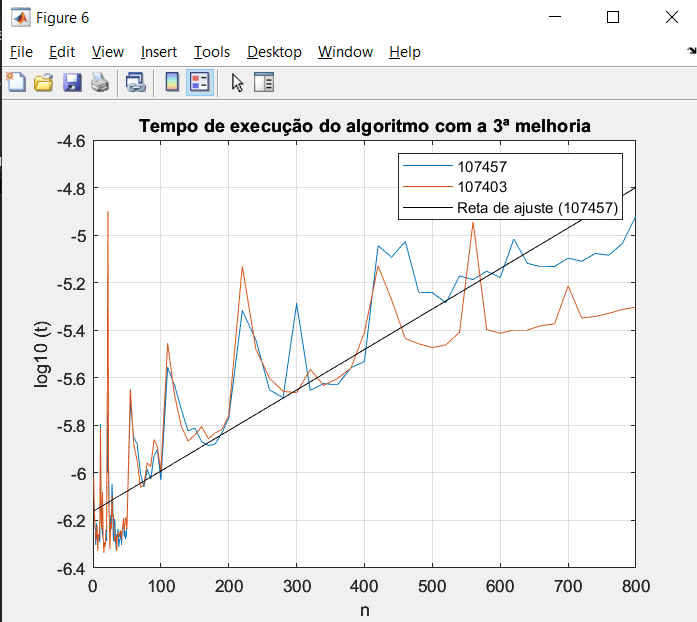
Esta melhoria, também sendo simples, torna possível, juntamente com as outras melhorias resolver o problema para as 800 posições, sendo possível agora chegar à posição 800 em 1.203e-05 segundos (com o número mecanográfico 107457).

Figura 7 - Reta de ajuste aos dados da 3ª melhoria para o Nmec 107457

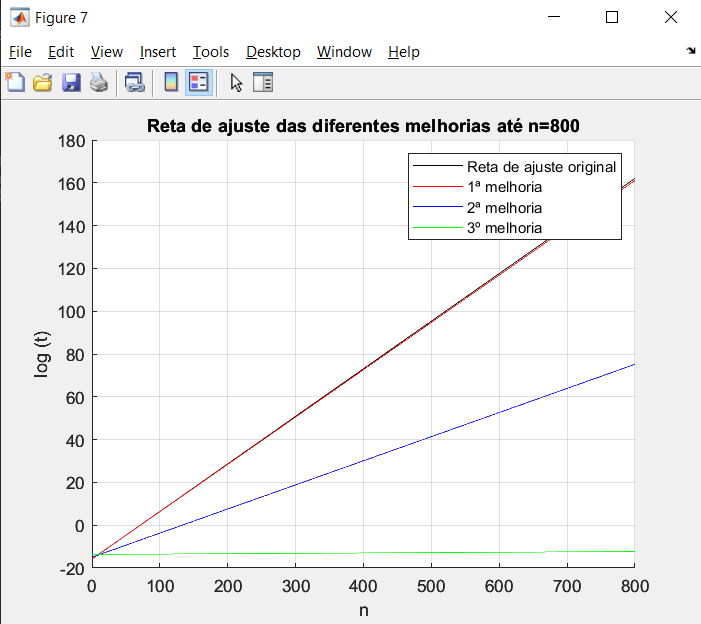


Figura 8- Comparação das diferentes retas de ajuste.

A 2ª e a 3ª melhoria foram baseadas no algoritmo de Branch and Bound, uma vez que a sua função é descartar logo um ramo se essa solução for pior que a “melhor” solução anteriormente encontrada e guardada.

Posto isto, com apenas três simples otimizações conseguimos melhorar a solução já fornecida a ponto de ser possível chegar à posição 800 em microssegundos.

# **Segundo** **Algoritmo**

Este algoritmo foi criado pensando numa maneira de descobrir a melhor solução para o problema percorrendo a estrada toda uma única vez.

Uma imagem com texto

Descrição gerada automaticamentePosto isto, começamos por criar um ciclo while que permitisse percorrer toda a estrada. Dentro desse ciclo temos um if que vai verificar se é possível aumentar, diminuir ou manter a velocidade.

Figura 9 - Função da Solução 2

Para conseguir fazer uma verificação que garantisse que em nenhum momento se iria desrespeitar as regras da estrada criámos uma função (respect\_limits) que tem como parâmetros de entrada a posição onde se encontra, a velocidade a que está a tentar seguir e a posição final da estrada. O objetivo desta é verificar se a velocidade que está a tentar seguir é válida, e para essa avaliação, o algoritmo verifica, se com essa velocidade teria tempo de travar se já se encontrasse perto do fim da estrada. Verifica, também, se essa velocidade respeita a velocidade de todos os segmentos por onde vai passar até chegar à posição seguinte.

Uma imagem com texto

Descrição gerada automaticamenteSe a velocidade cumprir estes dois requisitos a função vai retornar o valor 1, caso contrário retorna o valor 0.

Figura 10 - Função usada na solução 2 para verificar a velocidade

Com esta solução temos que o tempo de execução para a posição 800 é 2.815e-06 segundos (com o número mecanográfico 107457).

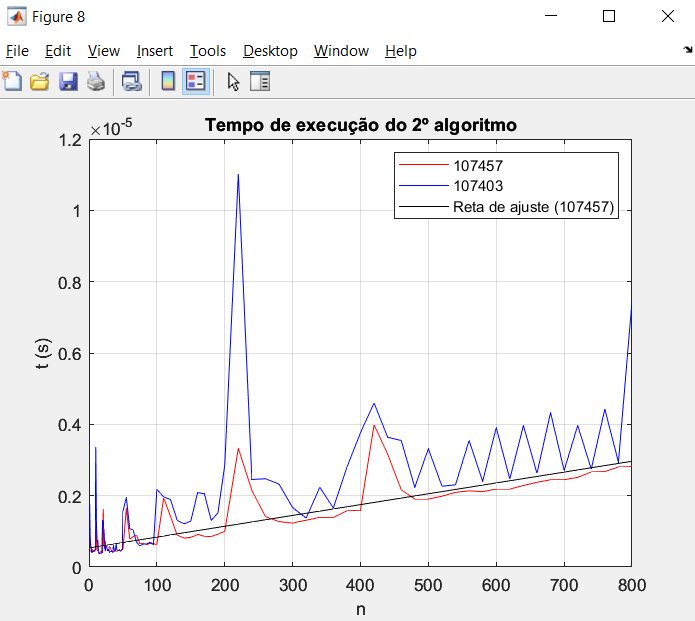


Figura 11 - Reta de ajuste aos dados do 2º algoritmo para o Nmec 107457

De notar que na Figura 11 não foi necessário aplicar logaritmos à construção da reta de ajuste, pois a mesma não é exponencial. Assim sendo, temos um gráfico do tempo (t(s)) de execução em função do número de segmentos da estrada (n).

# **Terceiro Algoritmo – Programação Dinâmica**

Para este algoritmo utilizámos a programação dinâmica, que consiste em dividir um problema de otimização em subproblemas mais simples e guardar a solução para cada, de modo que cada subproblema seja resolvido só uma vez.

No contexto do problema em estudo, o que começámos por pensar foi, por exemplo, numa estrada com 10 segmentos os saltos que o carro vai dar até começar a travar, por já estar perto do fim da estrada, vão ser os mesmo que numa estrada com 20 segmentos até essa posição, e assim em diante.

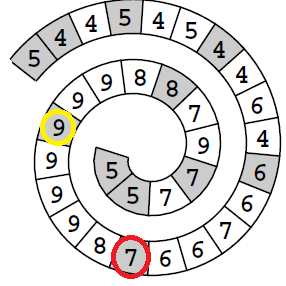
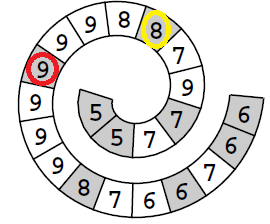
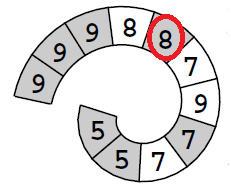


Figura 12 - 10 segmentos

Figura 13 - 20 segmentos

Figura 14 - 30 segmentos

As figuras acima demonstram o que foi descrito anteriormente, a posição marcada a vermelho é a posição onde o carro começa a travar, por já se encontrar perto do fim, e a amarelo está marcado a posição onde o algoritmo vai começar a nova procura, pois, as posições que estão para trás são iguais às já pesquisadas anteriormente.

Com este pensamento, criámos um algoritmo que segue a mesma ideia do segundo algoritmo, mas neste é criado um array com as posições onde o carro passou até à posição onde começa a travar, por se encontrar perto do fim da estrada, e é também guardado a velocidade com que ia nessa posição. Assim, na próxima pesquisa é reaproveitado esse array, e em vez da pesquisa começar no inico da estrada com velocidade 0, a pesquisa é iniciada na última posição desse array e com a velocidade guardada, uma vez que até aí as posições de paragem vão ser sempre as mesmas, o que torna a procura mais rápida e eficiente, evitando assim que faça duas vezes a mesma pesquisa.

Para esta solução foi criada uma estrutura nova (solution3\_t) para conseguir guardar o número de saltos, a velocidade, a posição em que ficou e o array das posições onde passou, e poder assim usar esses valores quando necessário.

Uma imagem com texto

Descrição gerada automaticamente

Figura 15 - Estrutura criada para a solução 3

Foi também reaproveitado o código da solução 2, fazendo só algumas alterações, que são:

Uma imagem com texto

Descrição gerada automaticamente- Dentro do while foi acrescentado um if que só é executado na primeira vez em que o carro chega a uma posição com uma velocidade onde tem de começar a reduzir para respeitar os limites até ao fim da estrada.

Figura 16 - Função da Solução 3

Uma imagem com texto

Descrição gerada automaticamente- Na função que verifica os limites de velocidade foram alterados os valores de retorno. Como para esse algoritmo é necessário saber exatamente quando é que a verificação falha por se ultrapassar a posição final da estrada é retornado 1 quando isso acontece. De seguida é verificado se não é ultrapassado o limite de velocidade de nenhum segmento de estrada por onde passa, se for retorna 2. Por fim, se a velocidade passar estas duas verificações o valor de retorno é 0.

Figura 17 - Função usada na solução 3 para verificar a velocidade

Alterámos também a função que executa o código do terceiro algoritmo, para chamar a solução com o número de saltos, a posição e a velocidade guardadas na estrutura da solução 3.

Uma imagem com texto

Descrição gerada automaticamente

Figura 18 - Função que chama a função da solução 3

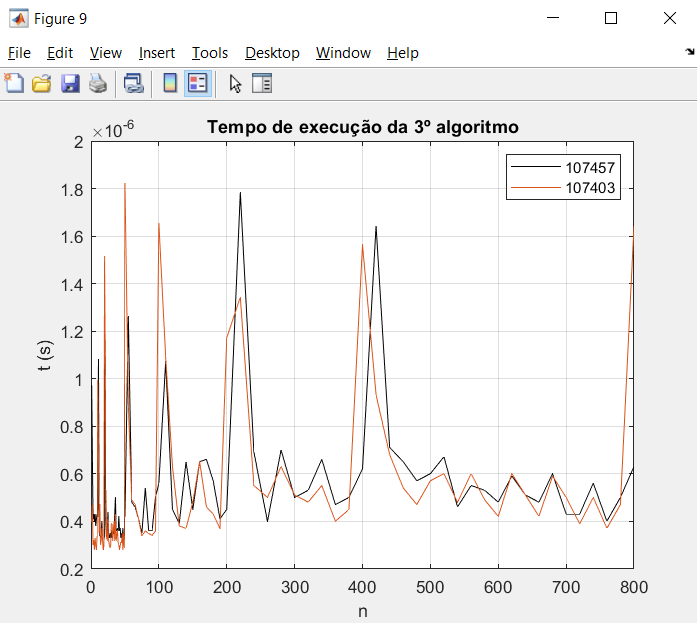
Com este algoritmo temos que o tempo de execução para a posição 800 é 6.310e-07 segundos (com o número mecanográfico 107457).

Figura 19- Dados do 3º algoritmo para os Nmec’s 107403 e 107457

Para este algoritmo, não fizemos a resta de ajuste aos dados visto que o gráfico é inconstante e tem bastante ruído. O gráfico tem aspeto de ser decrescente pois sempre que se inicia uma nova procura com um novo número de posição final o tempo é começado a zero, no entanto as primeiras posições possíveis já foram calculadas anteriormente e guardadas num array logo o tempo que demorou a fazer essa pesquisa antes não vai ser voltar a ser contabilizado. Isto provoca estas inconsistências no gráfico.

# **Resultados**

Figura 20 - Resultados com solução do professor otimizada

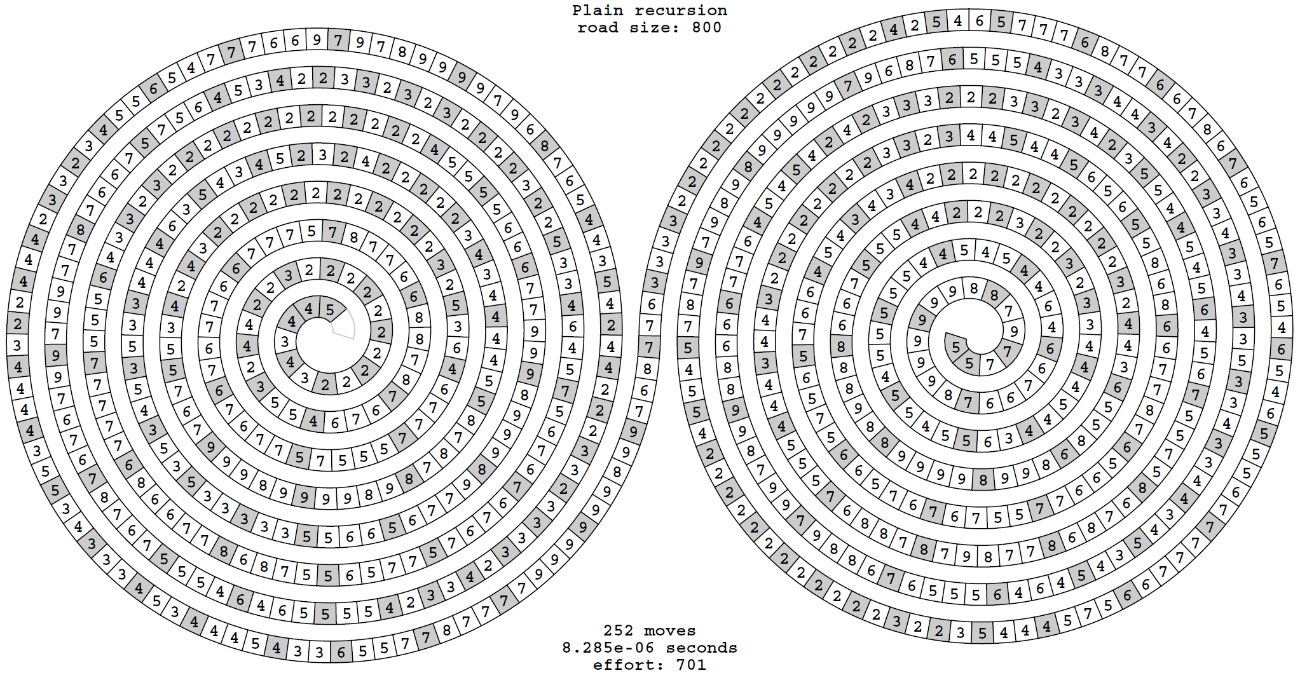


Figura 21 - Resultados com a segunda solução

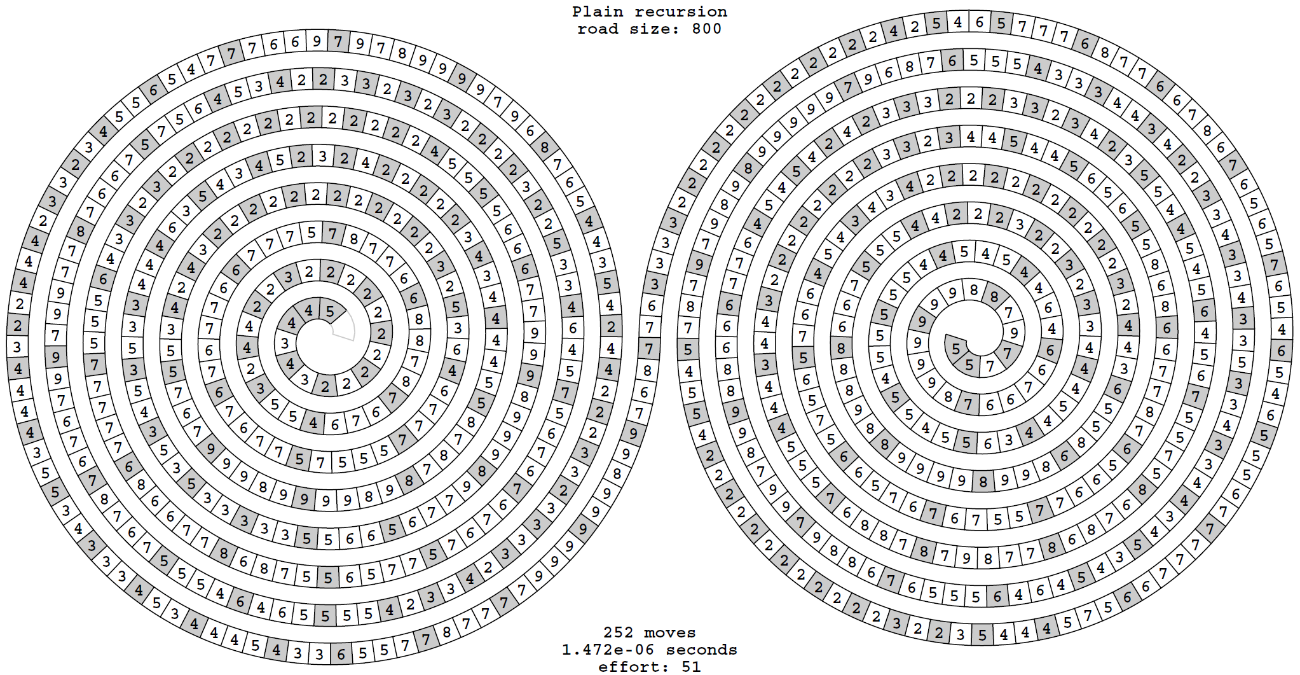
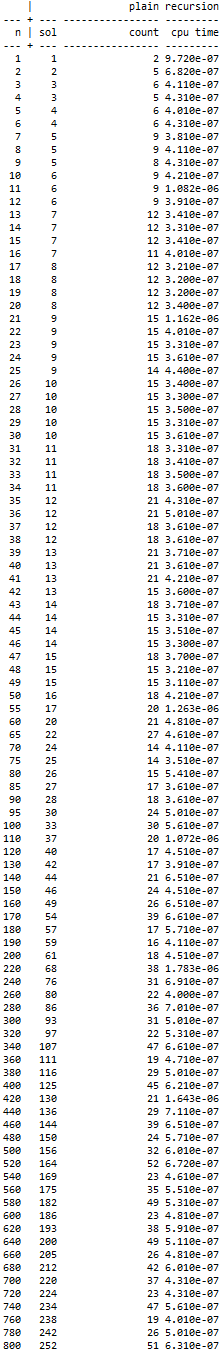


Figura 22 - Resultados com a terceira solução (Programação dinâmica)

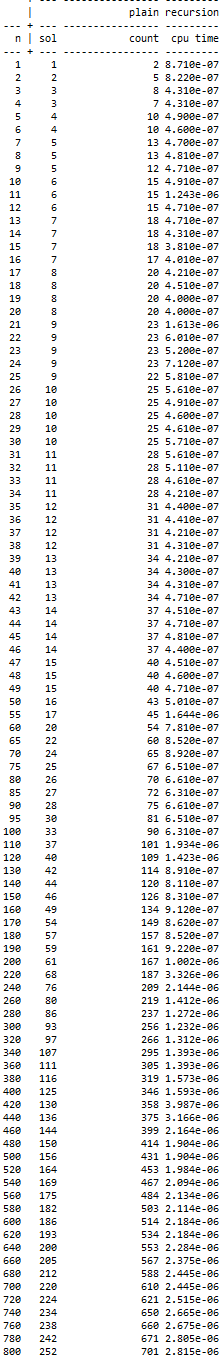
Através destas imagens vemos que os resultados das 3 solução são iguais, ou seja, todos dão 252 saltos (com o número mecanográfico 107457). No entanto, os tempos de execução variam, o que nos permite concluir que a terceira solução é a melhor, uma vez que tem um tempo de execução menor.

Estes resultados vão de encontro ao que era suposto, pois esta solução usa programação dinâmica, que permite combinar um array, já guardado, com as posições vistas anteriormente com as posições da solução que se está a calcular no momento, evitando assim que se volte a calcular uma coisa já antes vista, enquanto os outros dois algoritmos vão sempre procura uma solução partindo do zero.

## **Tempos de execução finais**

Uma imagem com texto

Descrição gerada automaticamente1º Algoritmo: 2º Algoritmo: 3º Algoritmo



# **Conclusões finais**

Com a realização deste trabalho, conseguimos aprofundar bastante os nossos conhecimentos em linguagem C e também adquirir novos conhecimentos sobre alguns algoritmos como o depth first search, o branch and bound e programação dinâmica. Antes do início da execução do projeto tínhamos pouco conhecimento sobre a otimização de algoritmos e como é que uma simples alteração num código consegue fazer tanta diferença na complexidade computacional e no tempo de execução.

Sentimos alguma dificuldade a entender a solução já fornecida devido à função recursiva e à falta de experiência de trabalho na linguagem em C. No entanto essas dificuldades foram diminuindo à medida que tínhamos mais aulas e que fazíamos mais pesquisas sobre a matéria, conseguindo assim ultrapassá-las.

Por fim, podemos afirmar que os objetivos propostos foram alcançados com sucesso, visto que conseguimos implementar 2 soluções de forma eficiente e otimizar a solução já fornecida com sucesso, chegando assim a ter 3 algoritmos que atingem a posição final com o menor número de saltos no tempo de execução de microssegundos.

# **Web grafia**

* <https://brilliant.org/wiki/depth-first-search-dfs/> consultado em 10/11/2022.
* <https://www.baeldung.com/cs/branch-and-bound> consultado em 10/11/2022.
* [https://web.stanford.edu/class/cs106b-8/lectures/backtracking-optimization/Lecture13.pdf consultado em 22/11/2022](https://web.stanford.edu/class/cs106b-8/lectures/backtracking-optimization/Lecture13.pdf%20consultado%20em%2022/11/2022).
* <https://www.freecodecamp.org/news/demystifying-dynamic-programming-3efafb8d4296/#:~:text=Dynamic%20Programming%20Defined,example%20of%20a%20sub-problem> consultado em 23/11/2022.

# **Apêndice**

## **Código C**

### Speed\_run.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. 107457  Name: Diana Raquel Rodrigues Miranda

//   N.Mec. 107403 Name: João NUno da Silva Luís

//

// 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;

typedef struct

{

  int move\_number;                        // the number of moves (the number of positions is one more than the number of moves)

  int speed;

  int position; // the positions (the first one must be zero)

  int positions[1 + \_max\_road\_size\_]; // the positions (the first one must be zero)

} solution3\_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

//Solution 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

//Solution 3

static solution3\_t solution3, 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 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 solution\_1\_otimized\_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)

      {

        if(move\_number >= solution\_1\_best.n\_moves){

          return;

        }

        if(solution\_1.positions[move\_number] < solution\_1\_best.positions[move\_number]){

          return;

        }

        solution\_1\_otimized\_recursion(move\_number + 1, position + new\_speed, new\_speed, final\_position);

      }

    }

}

static int respect\_limits(int position, int speed,int final\_position) //verificar se não execede a velocidade em nenhuma estrada por onde passa

{

  solution\_2\_count++;

  for(int s = speed; s >= 1; s--){

    for(int i = 0;i<=s;i++){

      if(((position + i) > final\_position) || max\_road\_speed[position + i] < s ){

        return 0;

      }

    }

    position += s;

  }

  return 1;

}

static void solution2(int move\_number, int position, int speed, int final\_position)

{

  while ((position != final\_position))

  {

    solution\_2\_count++;

    solution\_2.positions[move\_number] = position;

    if (respect\_limits(position, speed + 1, final\_position) == 1) //verificar se pode subir a velocidade

    {

      speed++;

      move\_number++;

      position += speed;

    }else if(respect\_limits(position, speed, final\_position) == 1) //verificar se pode manter a velocidade

    {

      move\_number++;

      position += speed;

    }else //pode diminuir a velocidade

    {

      speed--;

      move\_number++;

      position += speed;

    }

  }

  solution\_2.positions[move\_number] = position;

  //para  pintar a casa final

  solution\_2\_best = solution\_2;

  solution\_2\_best.n\_moves = move\_number;

  return;

}

static int respect\_limitsV2(int position, int speed,int final\_position) //verificar se não execede a velocidade em nenhuma estrada por onde passa

{

  solution\_3\_count++;

  //verifica se a soma de as posições ao desacelerar não ultrapassa a posição final

  if(((position + (speed\*(speed+1))/2) > final\_position)){

    return 1;

  }

  //Verifica se a velocidade máxima de cada segmento de estrada por onde passa é respeitada

  for(int s = speed; s >= 1; s--){

    for(int i = 0;i<=s;i++){

      if (max\_road\_speed[position + i] < s ){

        return 2;

      }

    }

    position += s;

  }

  return 0;

}

static void solution\_3(int move\_number, int position, int speed, int final\_position)

{

  int a = 0;

  while ((position != final\_position))

  {

    solution\_3\_count++;

    solution3.positions[move\_number] = position;

    int res = respect\_limitsV2(position, speed + 1, final\_position); //Iniciar a variável res verificando se pode acelerar

    if (res == 1 && a == 0){

      solution3.move\_number = move\_number;

      solution3.position = position;

      solution3.speed = speed;

      a = 1;  //Flag para só executar este if a primeira vez que ele tiver de travar por já se encontrar perto da posição final

    }

    if (res == 0)

    {

      speed++;

      move\_number++;

      position += speed;

    }else if(respect\_limitsV2(position, speed, final\_position) == 0) //verificar se pode manter a velocidade

    {

      move\_number++;

      position += speed;

    }else //pode diminuir a velocidade

    {

      speed--;

      move\_number++;

      position += speed;

    }

  }

  solution3.positions[move\_number] = position;

  solution\_3\_best = solution3;

  solution\_3\_best.move\_number = move\_number;

  return;

}

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\_otimized\_recursion(0,0,0,final\_position);

  solution\_1\_elapsed\_time = cpu\_time() - solution\_1\_elapsed\_time;

}

static void solve\_2(int final\_position)

{

  if (final\_position < 1 || final\_position > \_max\_road\_size\_)

  {

    fprintf(stderr, "solve\_1: bad final\_position\n");

    exit(1);

  }

  solution\_2\_elapsed\_time = cpu\_time();

  solution\_2\_count = 0ul;

  solution\_2\_best.n\_moves = final\_position + 100;

  solution2(0, 0, 0, final\_position);

  solution\_2\_elapsed\_time = cpu\_time() - solution\_2\_elapsed\_time;

}

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.move\_number = final\_position + 100;

  solution\_3(solution3.move\_number, solution3.position, solution3.speed, final\_position);

  solution\_3\_elapsed\_time = cpu\_time() - solution\_3\_elapsed\_time;

}

//

// example of the slides

//

static void example(void)

{

  int i, final\_position;

  srandom(0xAED2022);

  init\_road\_speeds();

  final\_position = 30;

  //solve\_3(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;

  solution\_2\_elapsed\_time = 0.0;

  solution\_3\_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);

    //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("                                |");

    }

    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);

    //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,"Plain 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("                                |");

    }

    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);

    //third solution method (less bad)

    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.move\_number,&solution\_3\_best.positions[0],solution\_3\_elapsed\_time,solution\_3\_count,"Plain recursion");

      }

      printf(" %3d %16lu %9.3e |",solution\_3\_best.move\_number,solution\_3\_count,solution\_3\_elapsed\_time);

      //printf("%3d %16lu %9.3e",solution\_3\_best.move\_number,solution\_3\_count,solution\_3\_elapsed\_time);

    }

    else

    {

      solution\_3\_best.move\_number = -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**

### Execution\_time.m

clear;clc;

DATA = load("SolucaoProf1hour.txt");

SoFor = load("SolProfOtimizadaFor.txt");

ForE1If= load("SolProfOtimizadaForE1If.txt");

ForE2If= load("SolProfOtimizadaForE2If.txt");

Sol2 = load("Solution2\_107457.txt");

Sol3 = load("Solution3\_107457.txt");

ForE2If\_107403= load("Solution1\_107403.txt");

Sol2\_107403 = load("Solution2\_107403.txt");

Sol3\_107403 = load("Solution3\_107403.txt");

n = DATA(:,1); % selecionar dos dados do .txt a primeira coluna com os valores de n

t = DATA(:,4); % selecionar dos dados do .txt a quarta coluna com os valores de n

%% Gráficos dos algoritmos originais

figure(1)

plot(n,t,"r") % gráfico super exponencial a partir do x=40

title("Tempo de execução da solução fornecida");

xlabel("n");

ylabel("t (s)")

grid on

figure(2)

semilogy(n,t,"g")

title("Tempo de execução do algoritmo");

xlabel("n");

ylabel("semilogy")

grid on

figure(3)

plot(n,log10(t),"b") % quase o mesmo que o plot do semilogy, mudando os valores do eixo y

title("Tempo de execução da solução fornecida");

xlabel("n");

ylabel("log10 (t)")

grid on

t\_log =log10(t);

N = [n(20:end) 1+0\*n(20:end)]; % começa no 20, porque é a partir desse n que a reta fica mais estável,

% para a reta de ajuste apanhar a maior parte dos dados

Coefs = pinv(N)\*t\_log(20:end); % matriz de regressão

hold on

Ntotal = [n n\*0+1];

% regra de ajuste aos dados

P2= plot(n, Ntotal\*Coefs, "k");

legend(P2,"Reta de ajuste")

hold off

t800\_log = [800 1]\* Coefs;

% gráfico para as 800 posições

% temos de calcular os t's ate a essa posição e não só o t=800

n= 1:800;

for i=n

t(i)= [i 1]\*Coefs;

t(i)= 10.^t(i) / 3600 / 24 /365;

end

t\_log =log10(t);

%% construir o grafico para a modificação do FOR

n\_for = SoFor(:,1);

t\_for = SoFor(:,4);

figure(4)

plot(n\_for,log10(t\_for),"b")

t\_log\_for =log10(t\_for);

N = [n\_for(20:end) 1+0\*n\_for(20:end)];

Coefs = pinv(N)\*t\_log\_for(20:end); % matriz de regressão

hold on

Ntotal = [n\_for n\_for\*0+1];

% regra de ajuste aos dados

P2= plot(n\_for, Ntotal\*Coefs, "k");

title("Tempo de execução do algoritmo com a 1ª melhoria");

xlabel("n");

ylabel("log10 (t)")

legend(P2,"Reta de ajuste")

grid on

hold off

t800\_log\_for = [800 1]\* Coefs;

n= 1:800;

for i=n

t\_for(i)= [i 1]\*Coefs;

t\_for(i)= 10.^t\_for(i) / 3600 / 24 /365;

end

t\_log\_for =log10(t\_for);

%% construir o grafico para a 2º melhoria: FOR mais 1 IF

n\_F1if = ForE1If(:,1);

t\_F1if = ForE1If(:,4);

figure(5)

plot(n\_F1if,log10(t\_F1if),"g")

t\_log\_F1if =log10(t\_F1if);

N = [n\_F1if(20:end) 1+0\*n\_F1if(20:end)];

Coefs = pinv(N)\*t\_log\_F1if(20:end); % matriz de regressão

hold on

Ntotal = [n\_F1if n\_F1if\*0+1];

% regra de ajuste aos dados

P2= plot(n\_F1if, Ntotal\*Coefs, "k");

title("Tempo de execução do algoritmo com a 2ª melhoria");

xlabel("n");

ylabel("log10 (t)")

legend(P2,"Reta de ajuste")

grid on

hold off

t800\_log\_F1if = [800 1]\* Coefs;

n= 1:800;

for i=n

t\_F1if(i)= [i 1]\*Coefs;

t\_F1if(i)= 10.^t\_F1if(i) / 3600 / 24 /365;

end

t\_log\_F1if =log10(t\_F1if);

%% construir o grafico para a 2º melhoria: FOR mais 2 IF

n\_F2if = ForE2If(:,1);

t\_F2if = ForE2If(:,4);

n\_F2if\_107403 = ForE2If\_107403(:,1);

t\_F2if\_107403 = ForE2If\_107403(:,4);

figure(6)

plot(n\_F2if,log10(t\_F2if))

TempoRealPos800\_log = log10(t\_F2if(100,1)); % ir buscar o valor real do tempo demorado na posição 800 (mas em log)

hold on

t\_log\_F2if =log10(t\_F2if);

N = [n\_F2if(20:end) 1+0\*n\_F2if(20:end)];

Coefs = pinv(N)\*t\_log\_F2if(20:end); % matriz de regressão

plot(n\_F2if\_107403,log10(t\_F2if\_107403))

Ntotal = [n\_F2if n\_F2if\*0+1];

% regra de ajuste aos dados

plot(n\_F2if, Ntotal\*Coefs, "k");

title("Tempo de execução do algoritmo com a 3ª melhoria");

xlabel("n");

ylabel("log10 (t)")

legend("107457","Reta de ajuste")

grid on

hold off

t800\_log\_F2if = [800 1]\* Coefs;

n= 1:800;

for i=n

t\_F2if(i)= [i 1]\*Coefs;

t\_F2if(i)= 10.^t\_F2if(i) / 3600 / 24 /365;

end

t\_log\_F2if =log10(t\_F2if);

%% Todos os gráficos dos diferentes algoritmos para 800 n's da solução 1

figure(7)

hold on

plot(n,t\_log, "k")

plot(n,t\_log\_for, "r")

plot(n,t\_log\_F1if, "b")

plot(n,t\_log\_F2if, "g")

xlabel("n");

ylabel("log (t)");

title("Reta de ajuste das diferentes melhorias até n=800")

legend("Reta de ajuste original","1ª melhoria","2ª melhoria","3º melhoria");

grid on

hold off

%% construir o grafico para a 2º solução

n\_Sol2 = Sol2(:,1);

t\_Sol2 = Sol2(:,4);

figure(8)

plot(n\_Sol2,t\_Sol2,"r")

title("Tempo de execução do 2º algoritmo");

xlabel("n");

ylabel("t (s)")

grid on

hold on

plot(n\_Sol2\_107403,t\_Sol2\_107403,"b")

N = [n\_Sol2(20:end) 1+0\*n\_Sol2(20:end)];

Coefs = pinv(N)\*t\_Sol2(20:end); % matriz de regressão

Ntotal = [n\_Sol2 n\_Sol2\*0+1];

% regra de ajuste aos dados

plot(n\_Sol2, Ntotal\*Coefs, "k");

legend("107457","107403","Reta de ajuste (107457)")

%% construir o grafico para a 3º solução

n\_Sol3 = Sol3(:,1);

t\_Sol3 = Sol3(:,4);

n\_Sol3\_107403 = Sol3\_107403(:,1);

t\_Sol3\_107403 = Sol3\_107403(:,4);

figure(9)

plot(n\_Sol3,t\_Sol3,"k")

hold on

plot(n\_Sol3\_107403,t\_Sol3\_107403)

title("Tempo de execução da 3º algoritmo");

xlabel("n");

ylabel("t (s)")

grid on

legend("107457","107403")