SPEED RUN

ALGORITMOS E ESTRUTURAS DE DADOS

Trabalho realizado por:

→ Pedro Ramos > 34%
  > NºMec: 107348

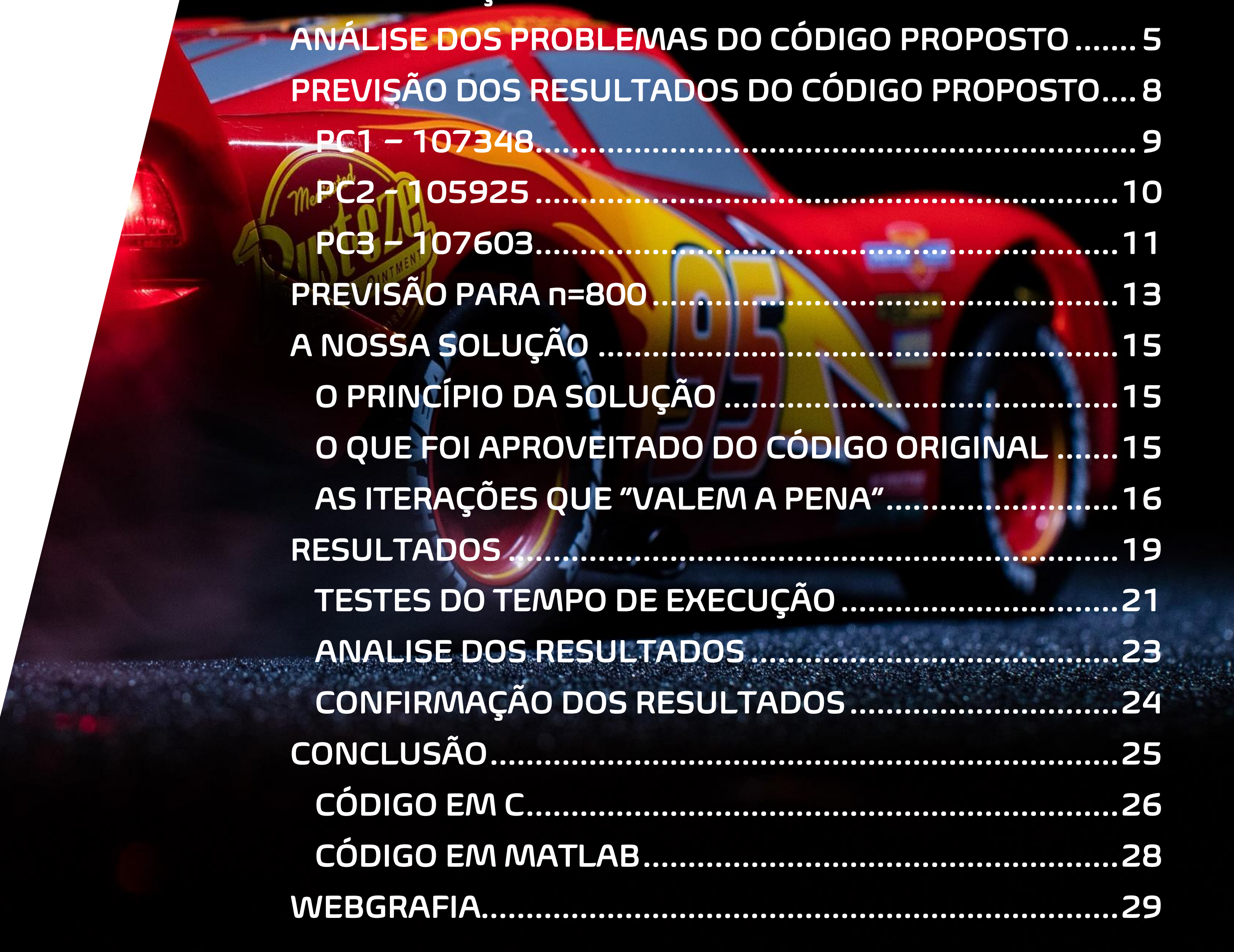
→ Daniel Madureira > 33%
  > NºMec: 107603

→ Rafael Kauati > 33%
  > NºMec: 105925

universidade de aveiro

deti
universidade de aveiro

ÍNDICE



SPEED RUN

Uma estrada está dividida por quilómetros;

Em cada quilómetro há um limite de velocidade;

A velocidade é dada pelo número de quilómetros que o carro pode fazer por movimento;

Em cada quilómetro o carro pode travar, acelerar ou manter a velocidade;

O carro começa parado e termina com velocidade 1;

Qual é o número mínimo de movimentos para o carro acabar a estrada?

INTRODUÇÃO

PROPOSIÇÃO DO PROBLEMA

O propósito deste trabalho prático é criar um **algoritmo** que consiga identificar o **número mínimo de movimentos entre dois pontos** sabendo que:

→ A distância entre estes dois pontos é subdividida em **segmentos de igual tamanho**;

→ A velocidade é o **número de segmentos** que se pode atravessar num **único movimento**;

→ **Cada segmento tem o seu limite máximo de velocidade**, isto é, ao passar por um segmento a **velocidade do carro nunca poderá ser maior que o limite máximo** de velocidade do segmento;

→ Em cada movimento podemos apenas alterar a velocidade:

> **Acelerar** : speed + 1;

> **Travar** : speed − 1;

> **Manter** : speed;

PROPOSIÇÃO DO TRABALHO

Visto que já nos foi providenciado um código para a resolução deste problema, tivemos como objetivo principal **alterar este código de forma a ser possível usá-lo para "estradas" com mais de 800 segmentos num tempo na ordem de alguns milissegundos**.
Apesar de ser possível utilizar o código fornecido para "estradas" com 800 ou mais segmentos, este é **muitíssimo ineficiente**, o suficiente para que um supercomputador como o *FUGAKU* (com incríveis 442PFLOPS) demorasse cerca de 10^{130} vezes a idade do nosso universo (1.38×10^{10} anos) para calcular 800 segmentos.

ANÁLISE DOS PROBLEMAS DO CÓDIGO PROPOSTO

O código base que nos foi fornecido utiliza um **algoritmo recursivo**, isto é, um algoritmo que se invoca para calcular o seu próprio valor. Esta é uma ótima forma de **dividir problemas complexos numa série de problemas mais simples** e semelhantes entre si.

Este código em específico percorre os segmentos da estrada até passar ou chegar ao fim da mesma.

Cada vez que o algoritmo corre, um **contador é incrementado**. Este contador (`solution_1_count`) indica-nos aproximadamente a **complexidade computacional da nossa solução** (idealmente será o menor possível).

É gravada também a posição relativa ao nosso número de movimentos.

```

// record move
solution_1_count++;
solution_1.positions[move_number] = position;

```

O algoritmo verifica posteriormente se a **posição onde se encontra é a final**, e se a sua velocidade é 1.

Caso isto seja verdade, sabemos que o algoritmo **chegou a uma solução final**, logo podemos **testá-la** de modo a verificar se é a **melhor solução** encontrada até então.

Se for, guardamo-la numa variável dedicada à melhor solução (`solution_1_best`).

```c
// 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;
}

```

Após cada **solução** ou **"beco sem saída"** – não consegue travar a tempo de passar pelo fim com velocidade **1** e passa pelo fim com velocidade **> 1** – o algoritmo **regressa um segmento e prossegue**.

Chegamos agora à parte mais complexa do algoritmo original:
Aqui, o algoritmo testa se pode **acelerar, travar ou manter a sua velocidade**. Caso possa, o algoritmo vai tentar **mover-se com cada novo parâmetro**, dividindo-se como se fosse uma **árvore**.

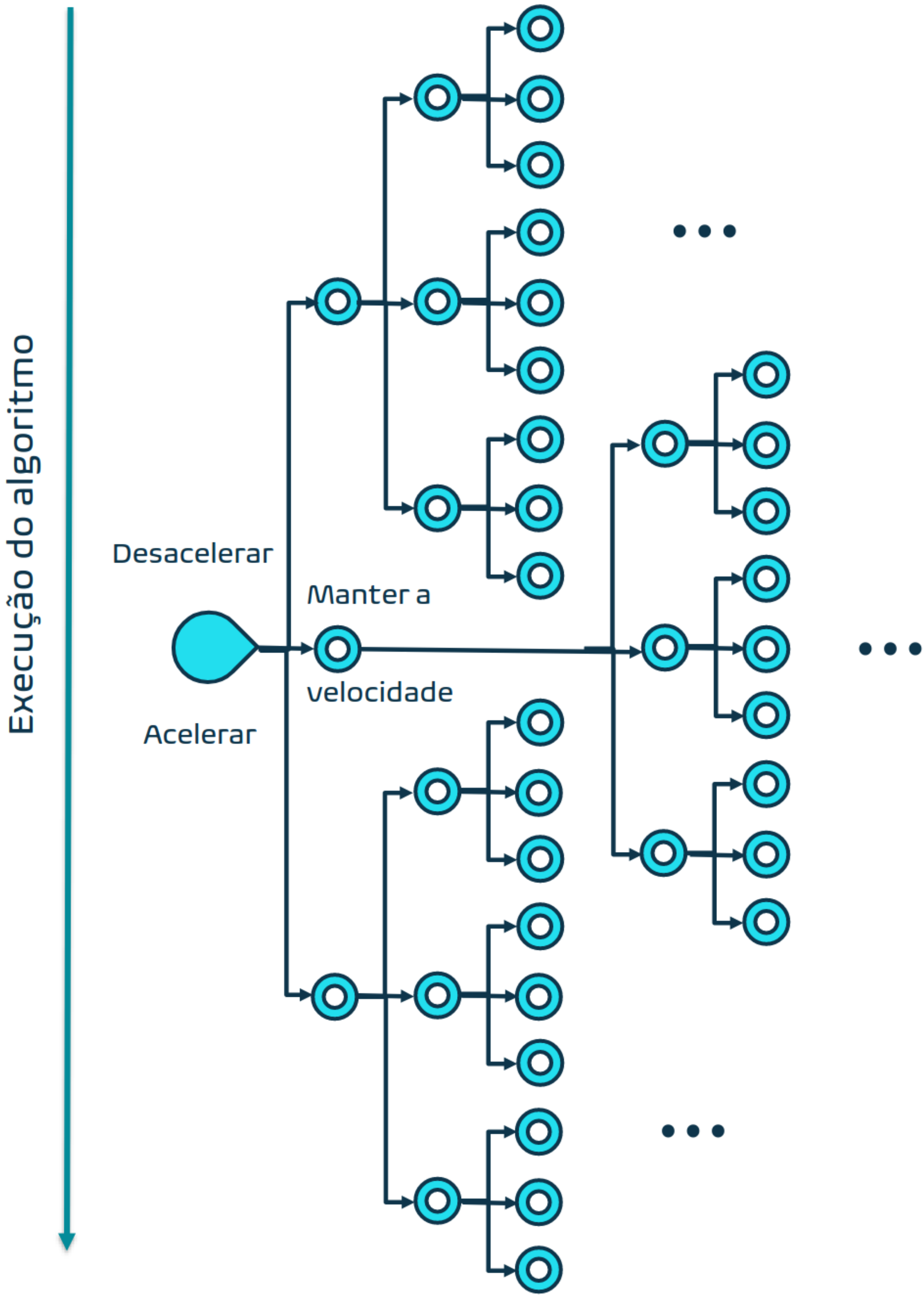
Esta divisão irá acontecer até que **todas as possibilidades de movimentos sejam calculadas** (mantendo em conta o limite máximo de velociade de cada segmento).

```c

  // 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);
    }
  }
}

```

Após todas as possibilidades serem analisadas, a variável `solution_1_best` irá guardar **a solução que chegou ao fim com o menor número de movimentos**.

*É de notar que, neste algoritmo, as primeiras iterações calculadas são as travagens, depois as que mantem a velocidade e só no fim são calculadas as iterações com aceleração.

→ Demonstração do algoritmo:



PREVISÃO DOS RESULTADOS DO CÓDIGO PROPOSTO

Ao executar o código original, notamos de imediato que a **complexidade** inerte ao código é **bastante elevada**, visto que para um caminho com cerca de 45 posições a solução já começa a demorar mais de um minuto a ser calculada.

Conseguimos também observar que a complexidade parece seguir um **aumento semelhante ao dos números de fibonacci**, mas iremos analisar isto melhor posteriormente.

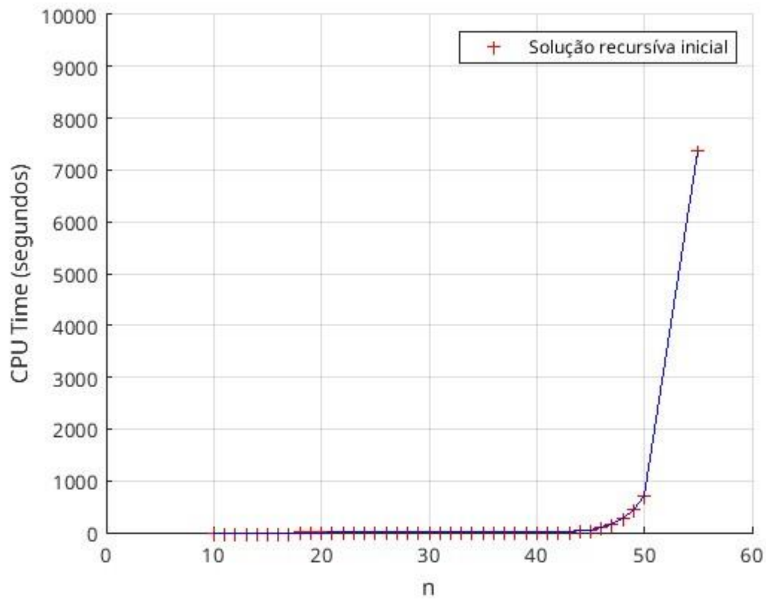
É importante notar que o código gera as **velocidades máximas para cada segmento de forma pseudoaleatória**, sendo possível controlar estas velocidades com a introdução de um número mecanográfico de um aluno como argumento ao correr o programa.

Para garantir que a nossa solução funciona para **qualquer vetor de velocidades máximas**, corremos o código em **três máquinas** diferentes com três números mecanográficos diferentes.

O código foi então corrido **várias vezes aumentando o tamanho** do problema (ou seja, aumentado a estrada) até que chegue a um tamanho de **800** ou que a iteração demore **mais que 1h** (inclusive) **a ser calculado**.
Os testes foram corridos nas seguintes máquinas:

	Processador	Sistema Operativo	Memória	Nº Mec
PC1	AMD Ryzen7 7500U (16 threads) @ 4.3 GHz	Manjaro (Arch), Linux kernel 5.15	16GB RAM 3.20 GHz	107348
PC2	Intel Core i7-1085G7 (8 threads) @ 4.8 GHz	EndeavourOS (Arch), Linux Kernel 6.0.10	16GB RAM 3.20 GHz	105925
PC3	Intel Core i5-1035G7 (8 threads) @ 4.2 GHz	Ubuntu Linux Kernel 5.13.0	8GB RAM 3.20 GHz	107603

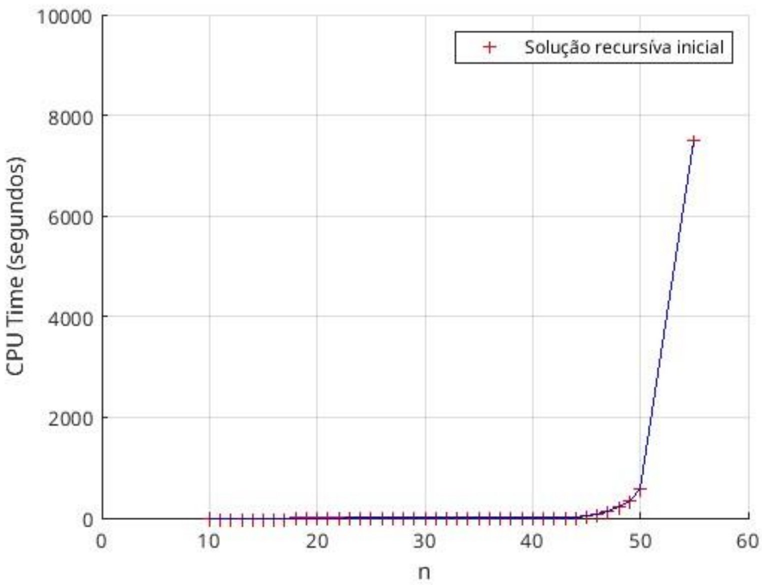
PC1 – 107348

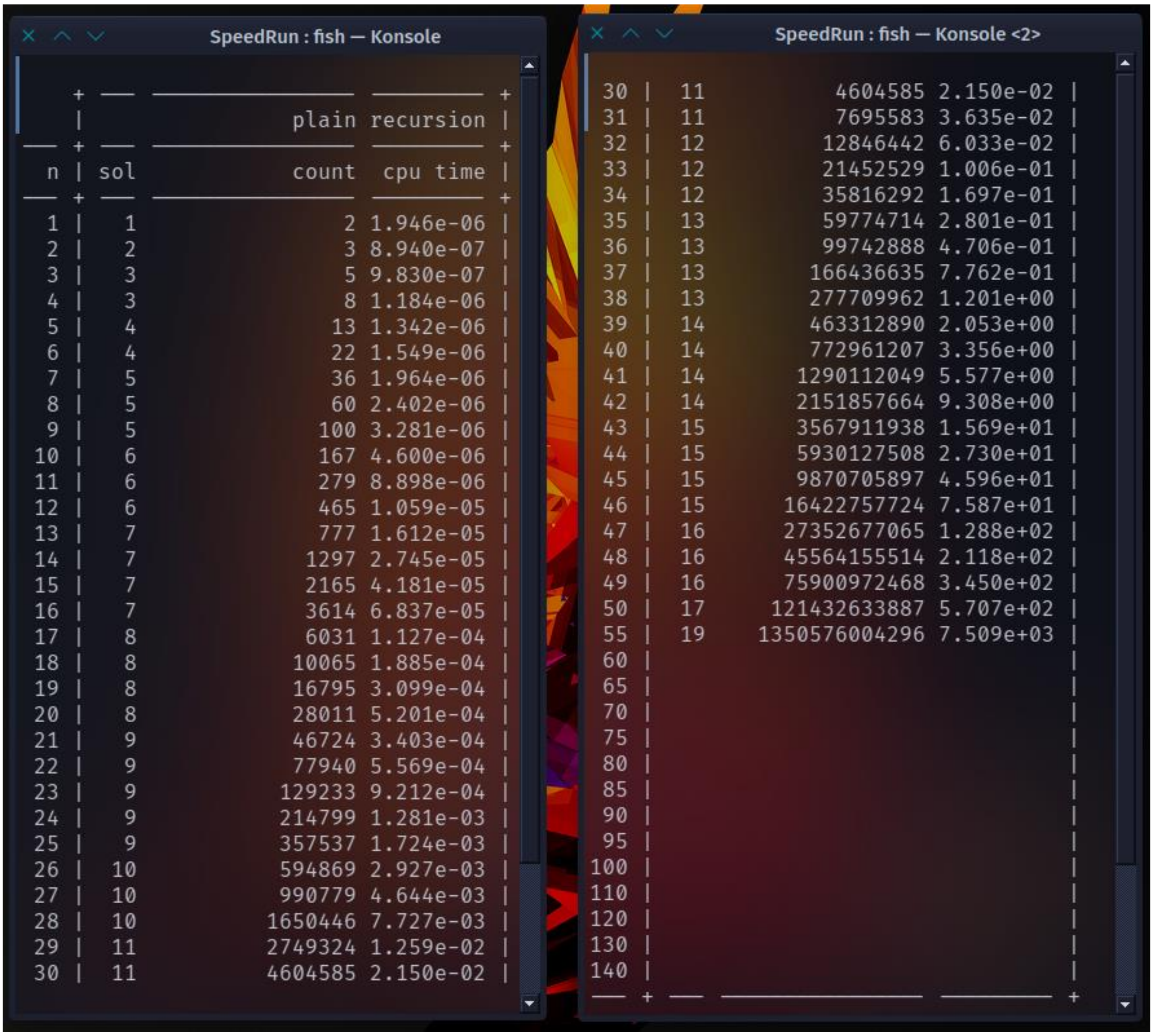


		plain recursion	
n	sol	count	cpu time
1	1	2	2.495e-06
2	2	3	8.910e-07
3	3	5	9.720e-07
4	3	8	1.163e-06
5	4	13	1.222e-06
6	4	22	1.553e-06
7	5	36	1.804e-06
8	5	60	2.234e-06
9	5	100	3.035e-06
10	6	167	4.017e-06
11	6	279	7.874e-06
12	6	465	9.067e-06
13	7	777	1.357e-05
14	7	1297	2.157e-05
15	7	2165	3.475e-05
16	7	3614	5.686e-05
17	8	6031	9.362e-05
18	8	10065	1.564e-04
19	8	16795	2.574e-04
20	8	28011	4.281e-04
21	9	46724	7.580e-04
22	9	77940	1.257e-03
23	9	130018	2.149e-03
24	9	216884	3.484e-03
25	9	361795	5.901e-03
26	10	599912	9.677e-03
27	10	997122	1.679e-02
28	10	1659747	2.720e-02
29	10	2761495	4.241e-02
30	11	4599395	7.476e-02

30	11	4599395	7.476e-02
31	11	7661720	1.242e-01
32	11	12762951	2.094e-01
33	12	21265449	3.499e-01
34	12	35430978	4.994e-01
35	12	59036163	3.967e-01
36	13	98370244	6.514e-01
37	13	163910216	1.107e+00
38	13	273122694	1.850e+00
39	14	455101542	2.777e+00
40	14	758333230	4.401e+00
41	14	1263611087	8.449e+00
42	15	2116400584	1.327e+01
43	15	3537407204	2.313e+01
44	15	5875115288	3.572e+01
45	16	9781288723	5.652e+01
46	16	16290155677	9.449e+01
47	16	27116619673	1.644e+02
48	16	45178474140	2.640e+02
49	17	72266166660	4.323e+02
50	17	117415713647	6.971e+02
55	19	1300309416640	7.358e+03
60			
65			
70			
75			
80			
85			
90			
95			
100			
110			
120			
130			
140			

PC2 - 105925



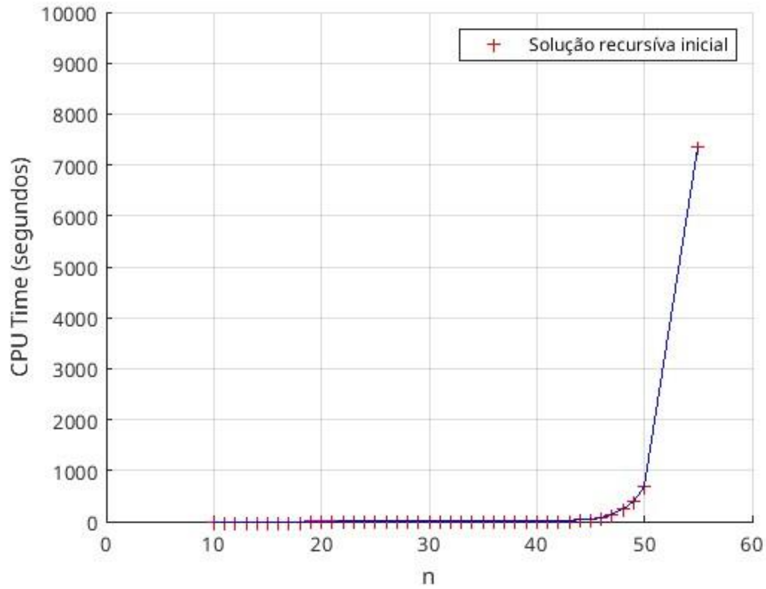

SpeedRun : fish — Konsole

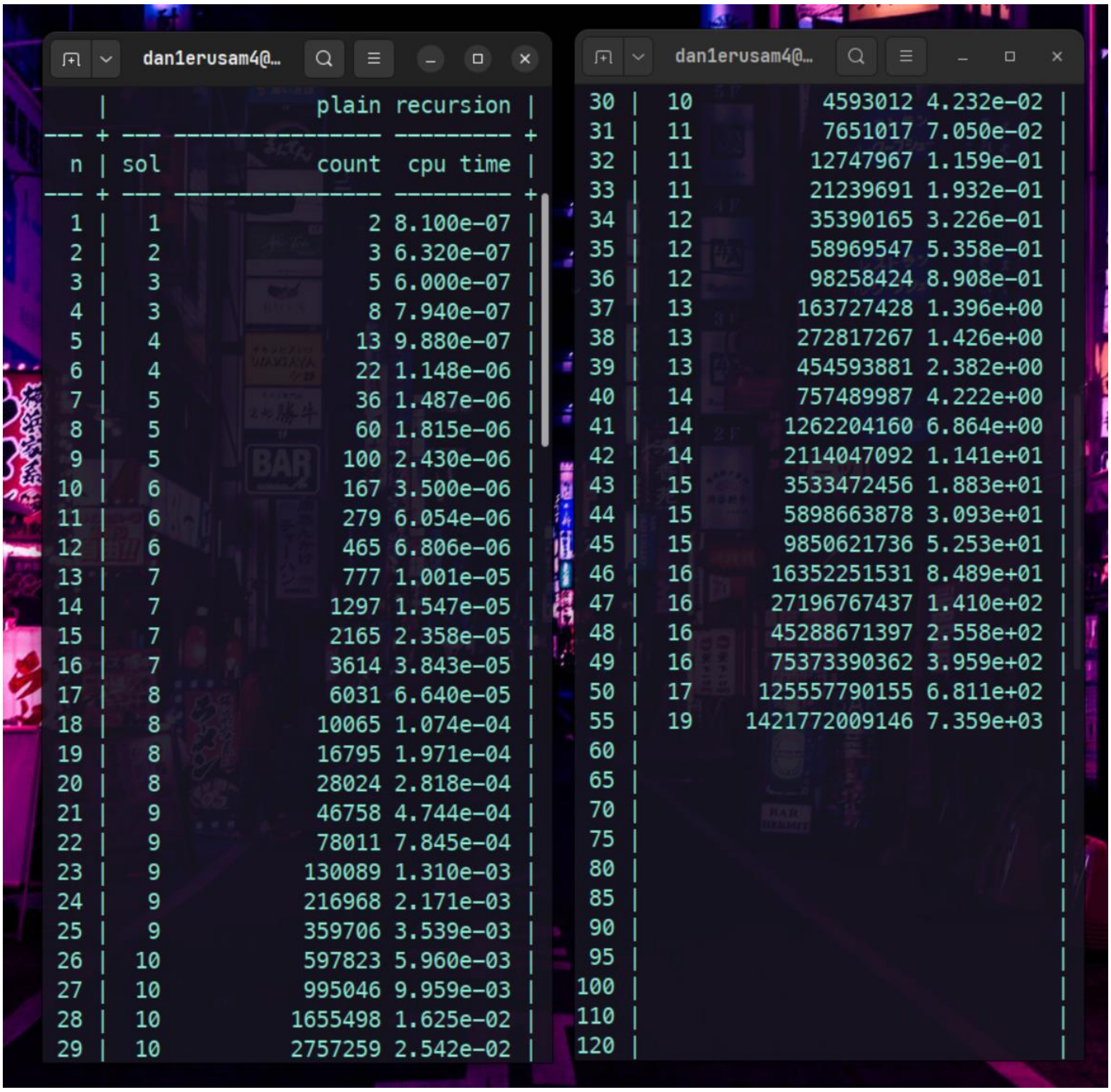
n	sol	plain recursion count	cpu time
1	1	2	1.946e-06
2	2	3	8.940e-07
3	3	5	9.830e-07
4	3	8	1.184e-06
5	4	13	1.342e-06
6	4	22	1.549e-06
7	5	36	1.964e-06
8	5	60	2.402e-06
9	5	100	3.281e-06
10	6	167	4.600e-06
11	6	279	8.898e-06
12	6	465	1.059e-05
13	7	777	1.612e-05
14	7	1297	2.745e-05
15	7	2165	4.181e-05
16	7	3614	6.837e-05
17	8	6031	1.127e-04
18	8	10065	1.885e-04
19	8	16795	3.099e-04
20	8	28011	5.201e-04
21	9	46724	3.403e-04
22	9	77940	5.569e-04
23	9	129233	9.212e-04
24	9	214799	1.281e-03
25	9	357537	1.724e-03
26	10	594869	2.927e-03
27	10	990779	4.644e-03
28	10	1650446	7.727e-03
29	11	2749324	1.259e-02
30	11	4604585	2.150e-02

SpeedRun : fish — Konsole <2>

30	11	4604585	2.150e-02
31	11	7695583	3.635e-02
32	12	12846442	6.033e-02
33	12	21452529	1.006e-01
34	12	35816292	1.697e-01
35	13	59774714	2.801e-01
36	13	99742888	4.706e-01
37	13	166436635	7.762e-01
38	13	277709962	1.201e+00
39	14	463312890	2.053e+00
40	14	772961207	3.356e+00
41	14	1290112049	5.577e+00
42	14	2151857664	9.308e+00
43	15	3567911938	1.569e+01
44	15	5930127508	2.730e+01
45	15	9870705897	4.596e+01
46	15	16422757724	7.587e+01
47	16	27352677065	1.288e+02
48	16	45564155514	2.118e+02
49	16	75900972468	3.450e+02
50	17	121432633887	5.707e+02
55	19	1350576004296	7.509e+03
60			
65			
70			
75			
80			
85			
90			
95			
100			
110			
120			
130			
140			

PC3 – 107603

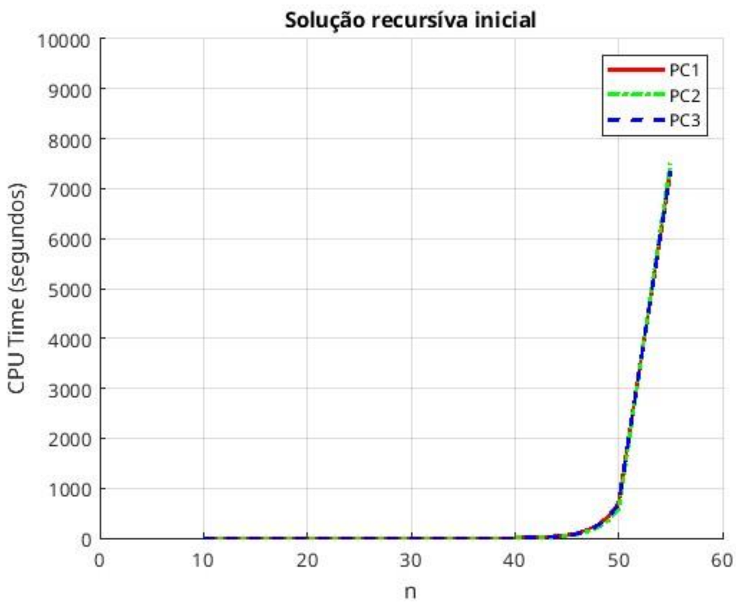



dan1erusam4@...

dan1erusam4@...

		plain recursion	
n	sol	count	cpu time
1	1	2	8.100e-07
2	2	3	6.320e-07
3	3	5	6.000e-07
4	3	8	7.940e-07
5	4	13	9.880e-07
6	4	22	1.148e-06
7	5	36	1.487e-06
8	5	60	1.815e-06
9	5	100	2.430e-06
10	6	167	3.500e-06
11	6	279	6.054e-06
12	6	465	6.806e-06
13	7	777	1.001e-05
14	7	1297	1.547e-05
15	7	2165	2.358e-05
16	7	3614	3.843e-05
17	8	6031	6.640e-05
18	8	10065	1.074e-04
19	8	16795	1.971e-04
20	8	28024	2.818e-04
21	9	46758	4.744e-04
22	9	78011	7.845e-04
23	9	130089	1.310e-03
24	9	216968	2.171e-03
25	9	359706	3.539e-03
26	10	597823	5.960e-03
27	10	995046	9.959e-03
28	10	1655498	1.625e-02
29	10	2757259	2.542e-02
30	10	4593012	4.232e-02
31	11	7651017	7.050e-02
32	11	12747967	1.159e-01
33	11	21239691	1.932e-01
34	12	35390165	3.226e-01
35	12	58969547	5.358e-01
36	12	98258424	8.908e-01
37	13	163727428	1.396e+00
38	13	272817267	1.426e+00
39	13	454593881	2.382e+00
40	14	757489987	4.222e+00
41	14	1262204160	6.864e+00
42	14	2114047092	1.141e+01
43	15	3533472456	1.883e+01
44	15	5898663878	3.093e+01
45	15	9850621736	5.253e+01
46	16	16352251531	8.489e+01
47	16	27196767437	1.410e+02
48	16	45288671397	2.558e+02
49	16	75373390362	3.959e+02
50	17	125557790155	6.811e+02
55	19	1421772009146	7.359e+03
60			
65			
70			
75			
80			
85			
90			
95			
100			
110			
120			

Comparação dos resultados:



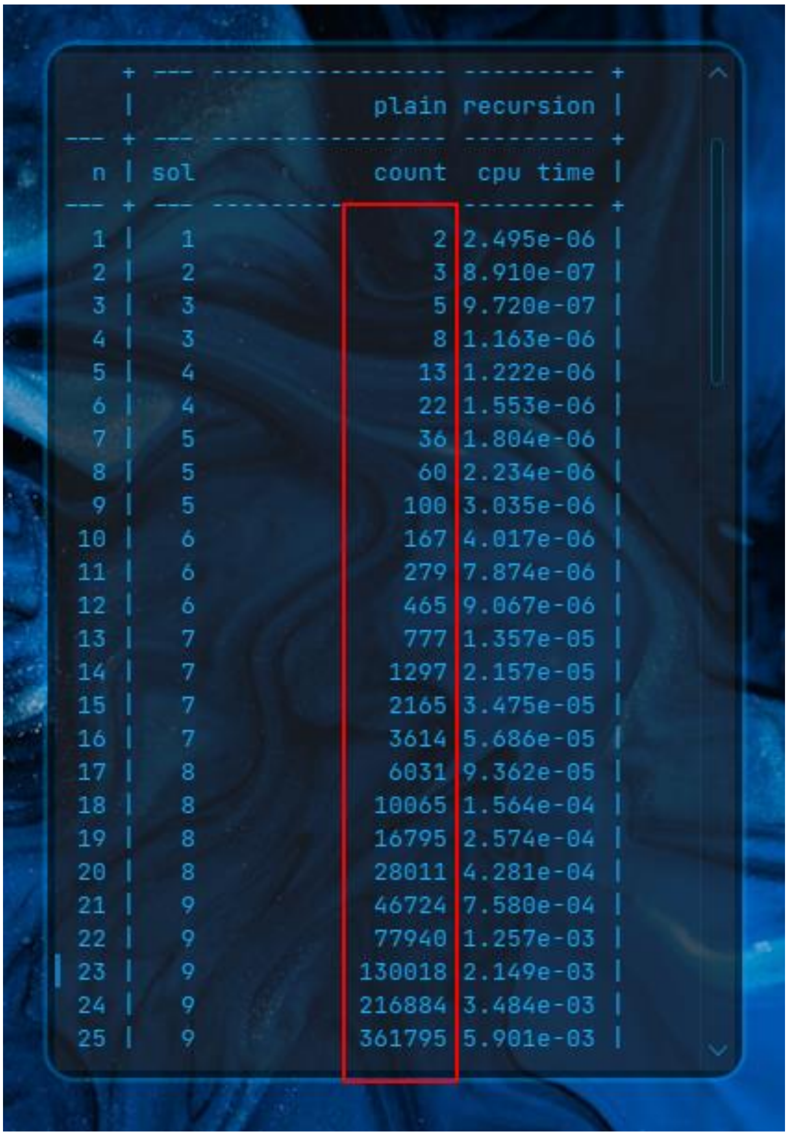
É de notar que foram removidos os valores para problemas com menos de 10 segmentos (n < 10) pois para um *n* muito pequeno os tempos de execução do código são muito irregulares devido ao processo ser bastante afetado por *interrupts* do sistema, gerando valores de tempo mais altos do que o esperado e incondizentes com o tempo real de cálculo do problema.

PREVISÃO PARA N=800

O objetivo deste trabalho é obter um tempo de execução razoável no cálculo do problema proposto para 800 segmentos (*n=800*).

Com os gráficos anteriores, é possível deduzir que o **algoritmo inicial** tem uma **complexidade exponencial**.

Ao **analisar o número de ramos** realizados para cada tamanho de problema, podemos também deduzir que o seu crescimento segue uma regra parecida com a **lei dos números de Fibonacci**, como podemos notar aqui (mas não está 100% correta):



		plain recursion	
n	sol	count	cpu time
1	1	2	2.495e-06
2	2	3	8.910e-07
3	3	5	9.720e-07
4	3	8	1.163e-06
5	4	13	1.222e-06
6	4	22	1.553e-06
7	5	36	1.804e-06
8	5	60	2.234e-06
9	5	100	3.035e-06
10	6	167	4.017e-06
11	6	279	7.874e-06
12	6	465	9.067e-06
13	7	777	1.357e-05
14	7	1297	2.157e-05
15	7	2165	3.475e-05
16	7	3614	5.686e-05
17	8	6031	9.362e-05
18	8	10065	1.564e-04
19	8	16795	2.574e-04
20	8	28011	4.281e-04
21	9	46724	7.580e-04
22	9	77940	1.257e-03
23	9	130018	2.149e-03
24	9	216884	3.484e-03
25	9	361795	5.901e-03

Para calcular o **tempo total** (T) que esta solução demora a resolver um problema com n = 800 segmentos temos primeiro de descobrir:

→ O **número médio de intruções por segundo**, I_S;

→ O **número de intruções** para n = 800, n_i .

Como vimos anteriormente, este algoritmo cresce de forma muito parecida aos números de Fibonacci. Assim podemos usar uma aproximação de **n=803 na sequência de números de Fibonacci** (803 pois para um n=1 já se começa com 2 ramos).

Logo:

$$fibonacci(803) = 16976270961326926046761105785534532532240518889429148378684513162106221074178195667852362249470316325577 \; instruções$$

Falta agora calcular o número médio de intruções por segundo, I_n .

Usando n = 55 temos:

$$I_n = \frac{1\,300\,309\,416\,640\ instruções}{7.358 \times 10^3\ segundos} \approx 176\,720\,497\ intruções/segundo$$

Assim, o tempo total de execução T , em anos, para n = 800 segmentos é:

$$T\ =\ \frac{n_i}{31\,556\,926\ \times \mathrm{I}_n} \approx 5.226 \times 10^{150}\ anos$$

Palavras não conseguem descrever o quão massivo este número é. Para referência, estima-se que a idade do nosso universo seja aproximadamente 1.38×10^{10} anos.

Este número lê-se como sendo:

5,266 novemquadragintilliões*

e será (caso não seja óbvio) certamente impossível de alcançar sem conseguir inventar um computador que consiga sobreviver a quase 2×10^{130} (dois duoquadragintilliões) colapsos e criações diferentes do nosso universo (assumindo que este vive os estimados 3.5×10^{10} anos).

Então como será possível o nosso código alcançar o **mesmo resultado** em apenas alguns **microssegundos** (3.1×10^{-14} anos)?

* Referência para os nomes dos números:
  > https://lcn2.github.io/mersenne-english-name/tenpower/tenpower.html

A NOSSA SOLUÇÃO

O PRINCÍPIO DA SOLUÇÃO

Como vimos durante a análise do código providenciado, este é altamente ineficiente.
A sua principal falha é que o algoritmo **não é capaz de decidir se deve continuar ou não um ramo**, logo este algoritmo faz com que **todos os ramos sejam completamente calculados até chegarem a uma solução**, o que se torna extremamente ineficiente.

Um exemplo seria um ramo que só se mexe com velocidade 1, que será sempre uma das piores soluções, mas que ainda assim é calculada pelo programa.

Só a implementação da capacidade de **ignorar ramos "desnecessários"** diminuirá drasticamente o tempo de execução do código, tornando-o significativamente mais rápido, uma vez que ao **invés de correr todos os milhares de ramos** até ao fim agora **corre apenas os melhores** e facilmente encontra a melhor solução.

O QUE FOI APROVEITADO DO CÓDIGO ORIGINAL

Apesar do grave problema apresentado anteriormente, a solução que nos foi dada utiliza um algoritmo recursivo muito bom para calcular o resultado final, visto que este **segue os princípios da programação dinâmica**, dividindo o problema final em problemas sucessivamente menores.

Portanto, o nosso algoritmo segue a mesma filosofia, baseando-se no código original, mas implementando a **capacidade de seleção dos ramos** que realmente "valem a pena".

Outro tipo de solução seria um **algoritmo interativo**.
Este correria **apenas uma única vez** por todos os segmentos, calculando **quantos movimentos são precisos para chegar a cada um deles**. Este código seria mais complicado de implementar, apesar de apresentar menor **complexidade computacional**.

Acabamos por não seguir esta abordagem pois a nossa **solução recursiva chegou a uma complexidade computacional parecida à esperada do algoritmo iterativo, mas com menos código**.

AS ITERAÇÕES QUE "VALEM A PENA"

Na nossa implementação, o algoritmo decide se deve cortar o ramo em que está com base em **2 critérios**:

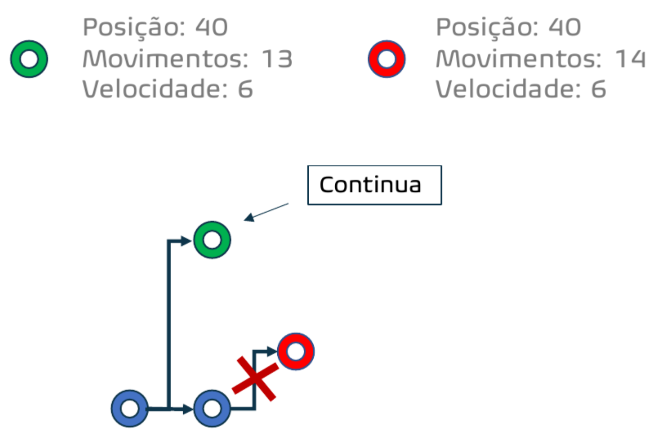
→ Se já chegou à posição atual noutro ramo com **menor número de movimentos**;

→ Se já chegou à posição atual noutro ramo com **maior velocidade.**

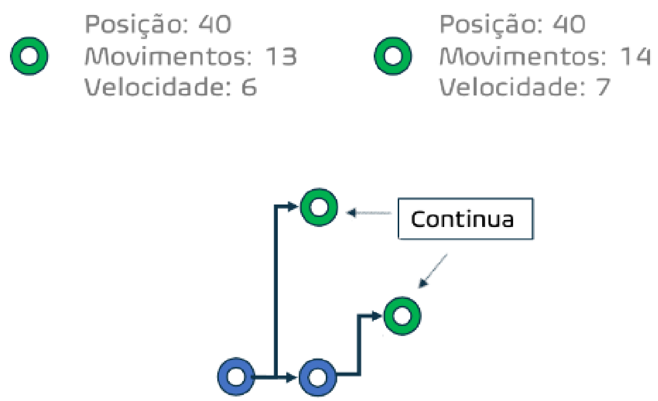
```
if (solution_2.positions[solution_2_best.n_moves] != 0){
  if (new_speed <= maxVelocidade[position+new_speed] &&
      move_number+1 >= minSaltos[position+new_speed]) {
    continue;
  }
}
```

Consideramos também que que **só é possivel cortar um ramo se já existir uma solução final**. Assim garantimos que temos sempre pelo menos um resultado final (útil em casos com muito poucos segmentos).

Por exemplo:
Se num ramo anterior o algoritmo chegou à **posição 40 em 13 movimentos**, mas no meu ramo atual chego à **posição 40 em 14 movimentos**, o algoritmo conclui que **não vale a pena continuar no meu ramo** uma vez que o outro é mais rápido.



Mas em casos extremamente raros (Ex: milhares de segmentos) pode acontecer que uma solução tenha **menos movimentos finais** do que outra apesar de ter chegado a uma **determinada posição com mais movimentos**, pois a sua **velocidade ao chegar a essa posição é maior**.



Logo, temos de ter em conta a **velocidade** com que chega ao ramo.

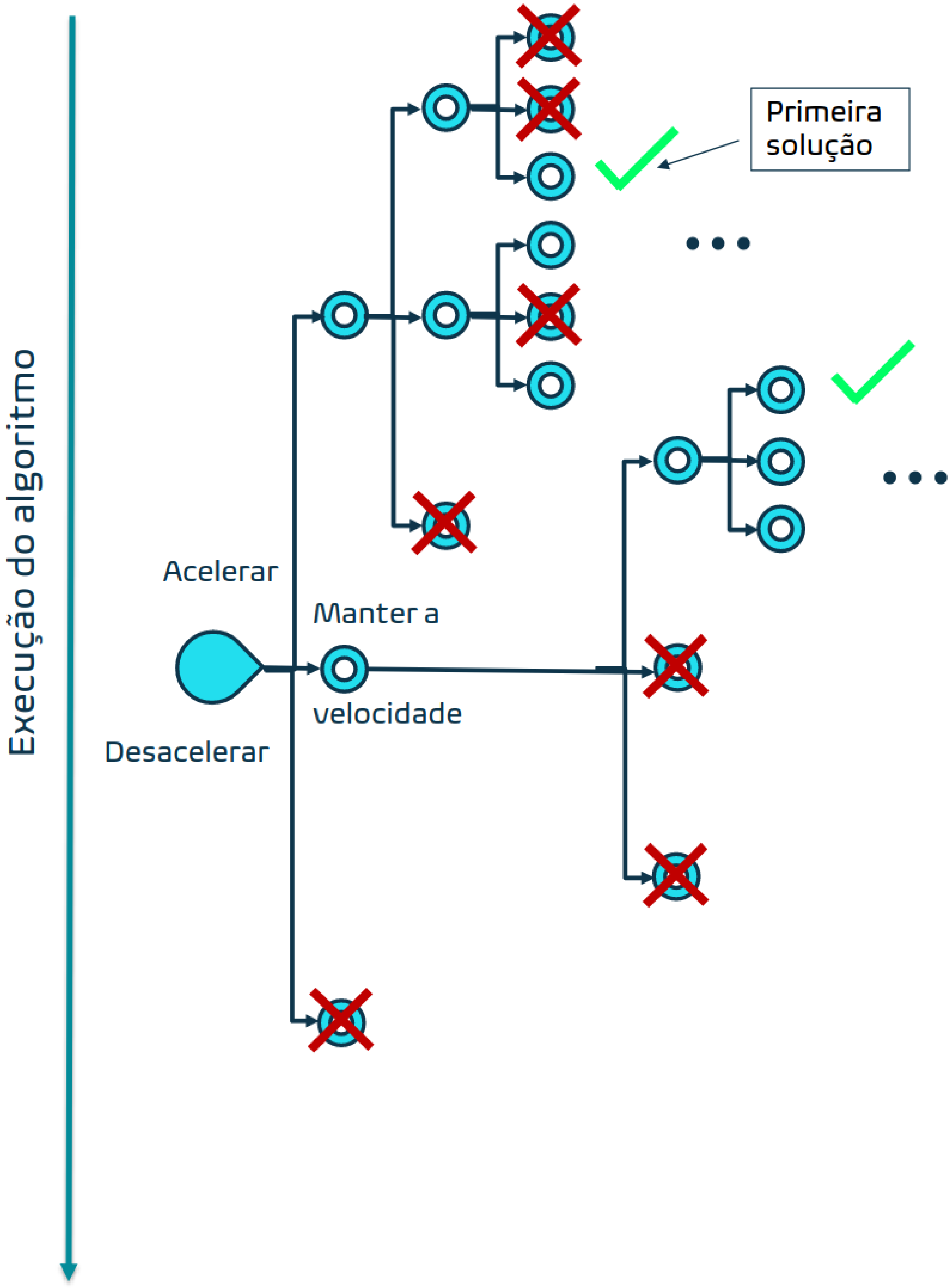
Caso o ramo continue, podemos **atualizar os vetores** onde guardamos o número de **movimentos mínimo** e as **velocidades máximas** da posição que estamos, visto que serão obrigatóriamente as melhores possíveis.

```
// Posição atual
minSaltos[position] = move_number+1;
maxVelocidade[position] = speed;

// Posições intermédias
for (int n = 1; n < new_speed; n++) {
  minSaltos[position+n] = move_number+1;
  maxVelocidade[position+n] = new_speed;
}
```

Também temos de preencher as **posições intermédias**.
Caso contrário, o **código vai deixar alguns ramos continuarem** de forma a tentar **preencher o array** de saltos mínimos e de velocidades máximas, o que não nos interessa visto que **maior parte das posições intermédias são irrelevantes para nós**.

Finalmente, **sabendo que estamos garantidamente num ramo que vale a pena** continuar, podemos **avançar para o próximo segmento** a ser calculado, voltando a chamar a função e repetindo o código para os novos parâmetros.

→ Demonstração do nosso algoritmo:

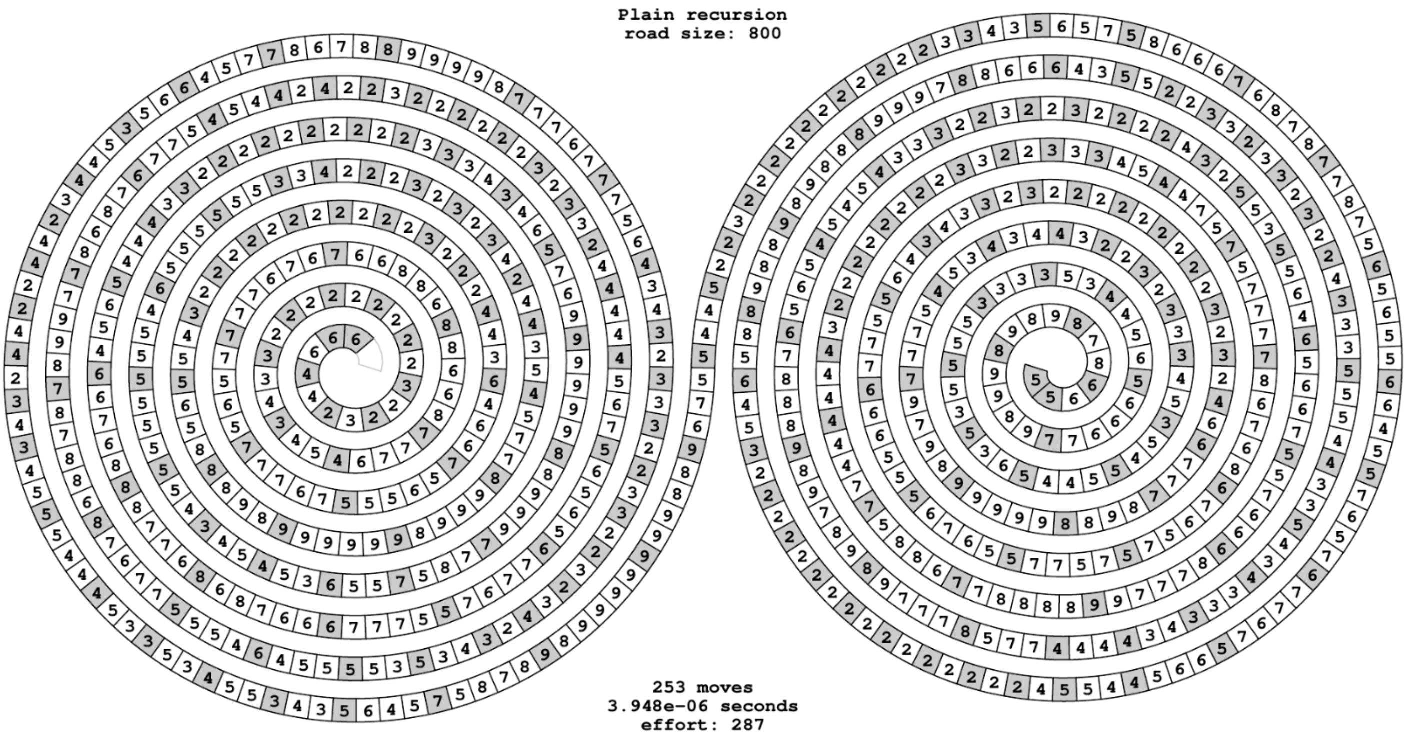


RESULTADOS

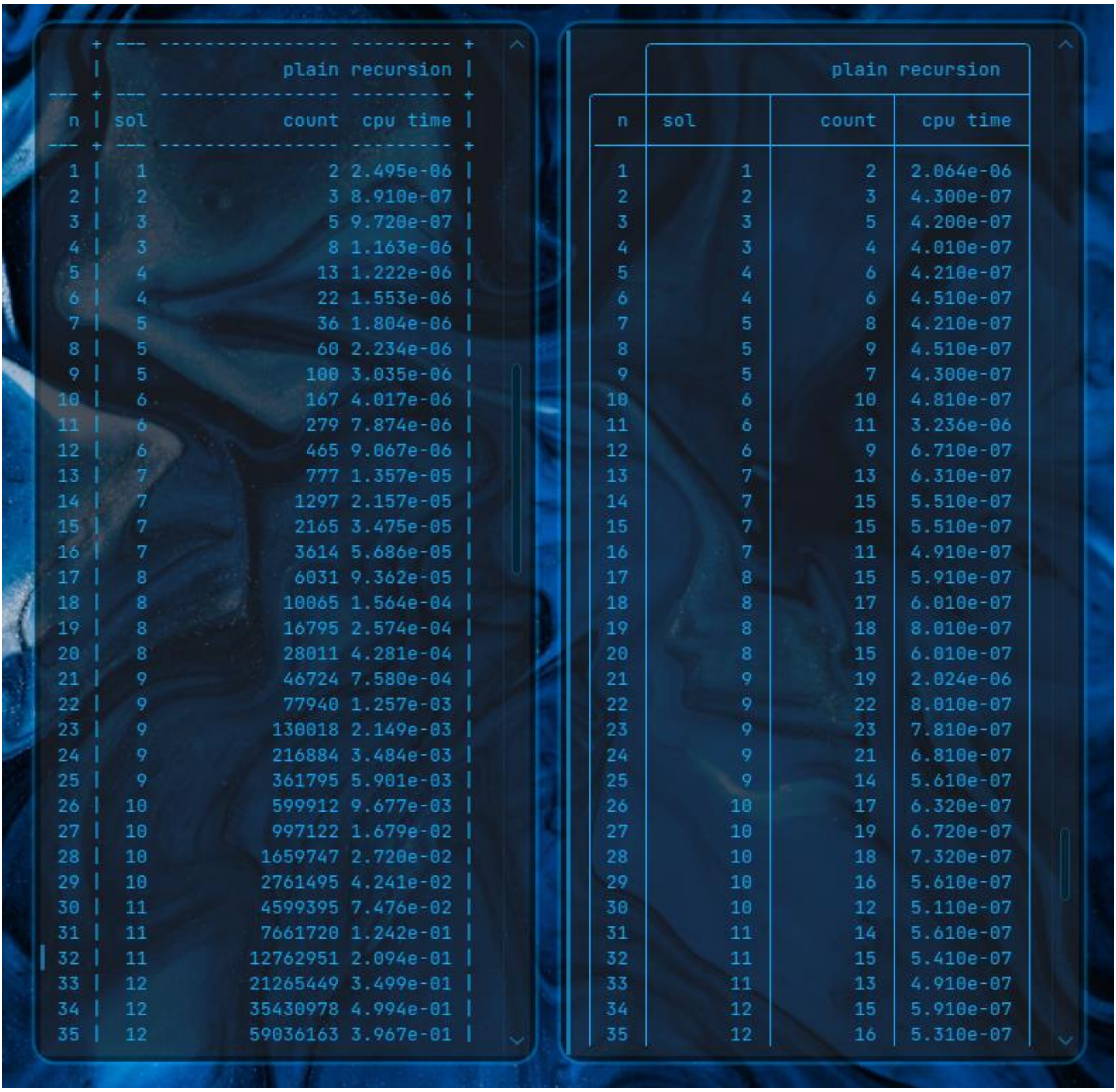
Finalmente, resta mostrar os resultados que obtivemos ao correr o nosso algoritmo.

Para um problema de tamanho n = 800, e sem número mecanográfico inserido, obtivemos um número de movimentos finais de **253 movimentos**, realizando **287 iterações** da função recursiva (ramos).
Isto demonstra uma **complexidade computacional bastante próxima de** $\boldsymbol{\theta(n)}$.

Nas mesmas condições, o algoritmo demora entre **4x10^{-6} e 6x10^{-6} segundos** a correr (depende da máquina onde o código está a ser executado e dos *interrupts* que este sofre), alcançando o tempo de execução pretendido para este projeto.



→ Resultados de execução da solução inicial contra a solução nova. É de notar que os resultados *"sol "* **são os mesmos** enquanto o número de ramos (*"count "*) e de *"cpu time "* são **enormemente inferiores**.

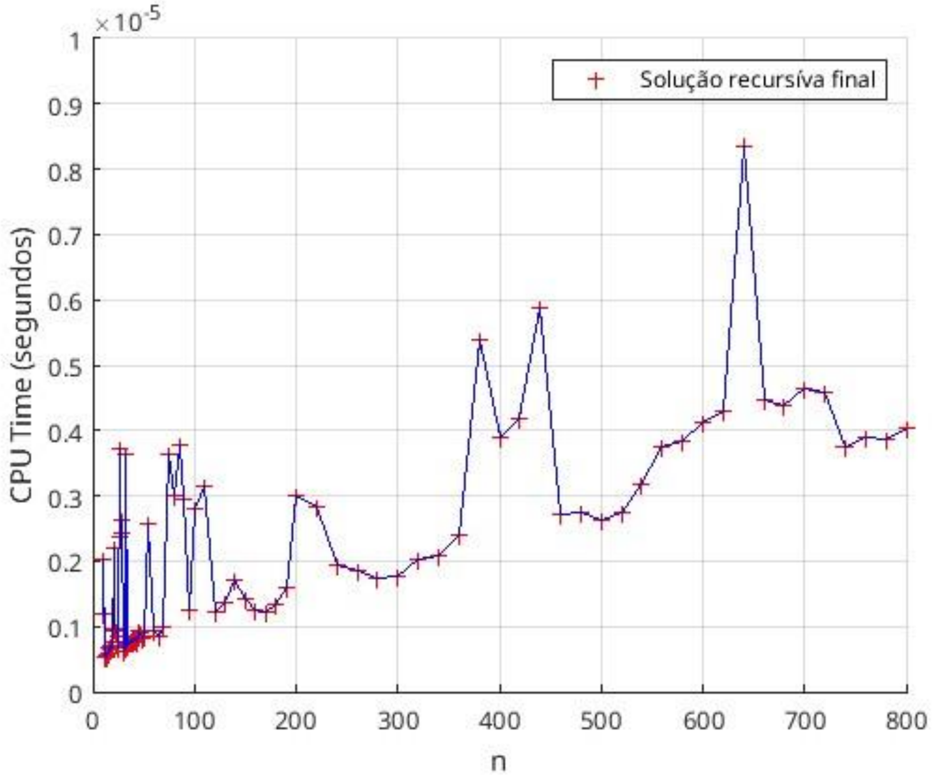


n	sol	plain recursion count	plain recursion cpu time
1	1	2	2.495e-06
2	2	3	8.910e-07
3	3	5	9.720e-07
4	3	8	1.163e-06
5	4	13	1.222e-06
6	4	22	1.553e-06
7	5	36	1.804e-06
8	5	60	2.234e-06
9	5	100	3.035e-06
10	6	167	4.017e-06
11	6	279	7.874e-06
12	6	465	9.067e-06
13	7	777	1.357e-05
14	7	1297	2.157e-05
15	7	2165	3.475e-05
16	7	3614	5.686e-05
17	8	6031	9.362e-05
18	8	10065	1.564e-04
19	8	16795	2.574e-04
20	8	28011	4.281e-04
21	9	46724	7.580e-04
22	9	77940	1.257e-03
23	9	130018	2.149e-03
24	9	216884	3.484e-03
25	9	361795	5.901e-03
26	10	599912	9.677e-03
27	10	997122	1.679e-02
28	10	1659747	2.720e-02
29	10	2761495	4.241e-02
30	11	4599395	7.476e-02
31	11	7661720	1.242e-01
32	11	12762951	2.094e-01
33	12	21265449	3.499e-01
34	12	35430978	4.994e-01
35	12	59036163	3.967e-01

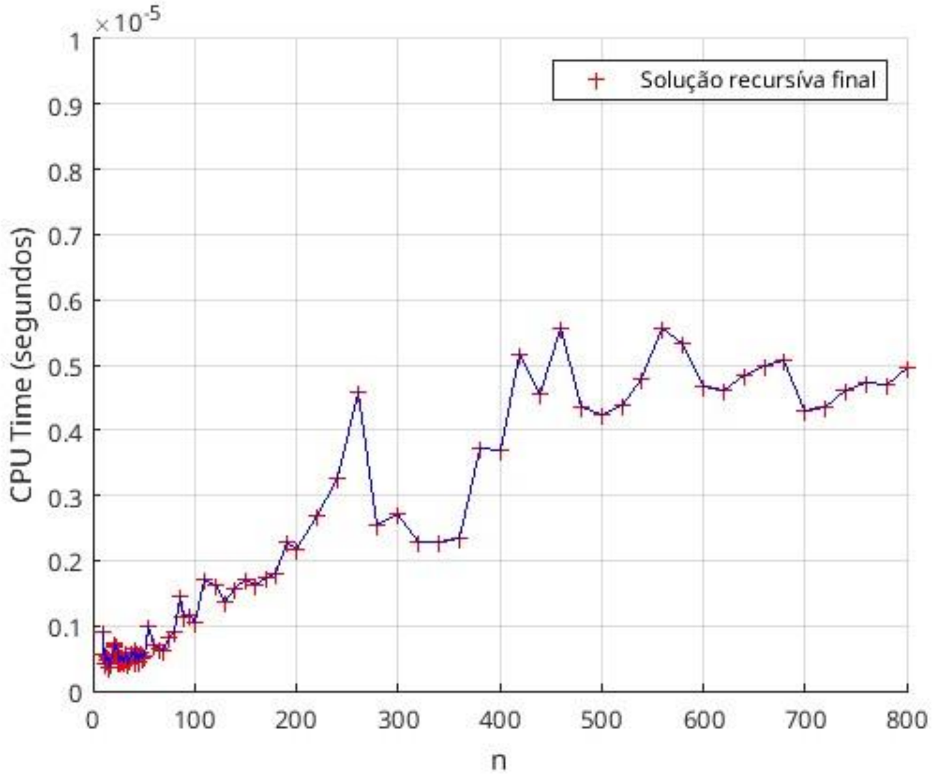
n	sol	plain recursion count	plain recursion cpu time
1	1	2	2.064e-06
2	2	3	4.300e-07
3	3	5	4.200e-07
4	3	4	4.010e-07
5	4	6	4.210e-07
6	4	6	4.510e-07
7	5	8	4.210e-07
8	5	9	4.510e-07
9	5	7	4.300e-07
10	6	10	4.810e-07
11	6	11	3.236e-06
12	6	9	6.710e-07
13	7	13	6.310e-07
14	7	15	5.510e-07
15	7	15	5.510e-07
16	7	11	4.910e-07
17	8	15	5.910e-07
18	8	17	6.010e-07
19	8	18	8.010e-07
20	8	15	6.010e-07
21	9	19	2.024e-06
22	9	22	8.010e-07
23	9	23	7.810e-07
24	9	21	6.810e-07
25	9	14	5.610e-07
26	10	17	6.320e-07
27	10	19	6.720e-07
28	10	18	7.320e-07
29	10	16	5.610e-07
30	10	12	5.110e-07
31	11	14	5.610e-07
32	11	15	5.410e-07
33	11	13	4.910e-07
34	12	15	5.910e-07
35	12	16	5.310e-07

TESTES DO TEMPO DE EXECUÇÃO

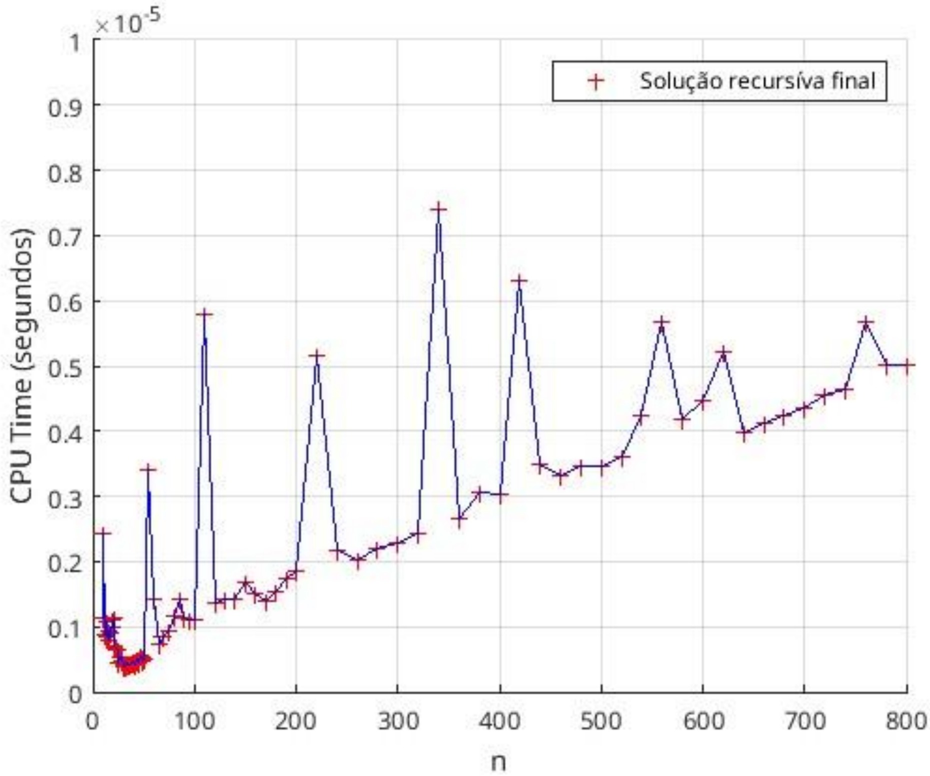
PC1



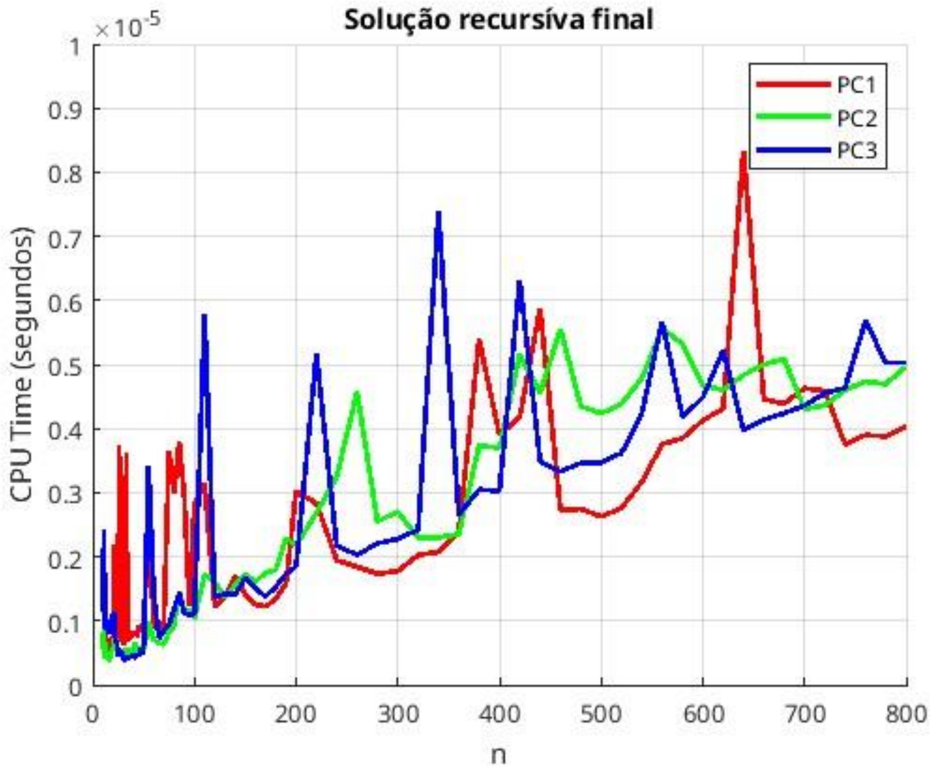
PC2



PC3



Comparação dos resultados dos 3 PCS



ANÁLISE DOS RESULTADOS

Analisando os resultados obtidos e a sua representação gráfica vemos que todos os tempos de execução estão na **ordem de 10^{-6} segundos**.

Podemos verificar também que a nossa implementação segue uma **relação praticamente linear** entre o tamanho do problema e o tempo que este demora a ser calculado, relevando uma **complexidade computacional $\boldsymbol{\theta(n)}$**.
Podemos relacionar os **picos observados** nos gráficos com ***interrupts* do sistema**, isto é, o tempo de execução aumenta devido a **outros programas que estão a correr simultaneamente no sistema**.
Num sistema "perfeito" estes picos **não existiriam** e a solução revelar-se-ia mais linear, e consequentemente com complexidade computacional $\theta(n)$.

Tambem testamos a nossa implementação em **máquinas mais antigas** (Intel Core i3-4010H e i3-350M) e num servidor AWS da Amazon (com dois CPUs Intel Xeon E5-2676) para verificar se os tempos de execução seriam diferentes, uma vez que a performance destes é menor dada a sua idade (mais de 12 anos no caso do i3-350M) ou às suas características originais (as cpu Xeon são específicamente desenhadas para performance multi-core, visto que a sua aplicação majoritária é em servidores).

No entanto, os tempos obtidos nestas máquinas **foram semelhantes aos obtidos nas máquinas mais recentes** (abaixo de 8×10^{-6} s para n = 800)*.

*Observação: Como os tempos de execução do nosso programa foram parecidos entre as mais diversas *CPUs* (gerações diferentes, fabricantes diferentes, *CPUs* de servidores, *CPUs* com mais de 12 anos), e tendo em conta que a nossa implementação é singlecore, podemos concluir que as *CPUs* atuais têm evoluído mais em termos de multicore performance e eficiência, e não em termos de single core performance.

CONFIRMAÇÃO DOS RESULTADOS

O código original, apesar de ter uma complexidade computacional incrivelmente alta, chega **sempre à melhor solução possível para um dado *n***, uma vez que esta averigua **todas as posições e ramos possíveis** para esse *n*.

Sabendo que o nosso algoritmo é baseado no código original, podemos assumir que os nossos resultados finais para cada *n* estão corretos.

Conseguimos confirmar esta afirmação comparando os resultados para os valores iniciais de *n* :

30	11	14	1.663e-06
31	11	15	1.673e-06
32	11	13	1.563e-06
33	12	15	1.874e-06
34	12	16	1.773e-06
35	12	14	1.623e-06
36	13	16	2.104e-06
37	13	17	1.803e-06
38	13	15	1.634e-06
39	14	17	2.064e-06
40	14	18	1.843e-06
41	14	16	1.633e-06
42	15	19	2.023e-06
43	15	20	1.763e-06
44	15	18	1.673e-06
45	16	21	1.894e-06
46	16	22	2.144e-06
47	16	21	2.094e-06
48	16	18	1.844e-06
49	17	20	1.943e-06
50	17	20	2.104e-06
55	19	20	4.369e-06

30	11	4599395	7.476e-02
31	11	7661720	1.242e-01
32	11	12762951	2.094e-01
33	12	21265449	3.499e-01
34	12	35430978	4.994e-01
35	12	59036163	3.967e-01
36	13	98370244	6.514e-01
37	13	163910216	1.107e+00
38	13	273122694	1.850e+00
39	14	455101542	2.777e+00
40	14	758333230	4.401e+00
41	14	1263611087	8.449e+00
42	15	2116400584	1.327e+01
43	15	3537407204	2.313e+01
44	15	5875115288	3.572e+01
45	16	9781288723	5.652e+01
46	16	16290155677	9.449e+01
47	16	27116619673	1.644e+02
48	16	45178474140	2.640e+02
49	17	72266166660	4.323e+02
50	17	117415713647	6.971e+02
55	19	1300309416640	7.358e+03

A nossa implementação (à esquerda) e o código original (à direita).

CONCLUSÃO

Através deste projeto pudemos desenvolver e aprimorar as nossas capacidades relacionadas à **criação, implementação e teste de algoritmos**, tal como a verificação e manipulação de dados.

Este projeto serviu também como consolidação de alguns conteúdos da própria cadeira de Algoritmos e Estrutura de Dados, nomeadamente a utilização de **técnicas de programação dinâmica** para resolver problemas através da **combinação de sub-soluções memorizadas**, permitindo **otimizar a verificação** de várias etapas em diversos algoritmos, bem como as próprias técnicas de otimização de verificação presentes em algoritmos de pequena e grande complexidade.

O grupo reconhece que o nosso algoritmo, por mais que já se encontre bastante eficiente e otimizado, sempre carecerá de algumas adaptações que o tornem ainda mais **eficiente e dinâmico**.

Futuramente, com mais experiência em desenvolvimento de algoritmos, poderá ser possivel aperfeiçoar este código de modo a ser ainda mais rápido e/ou desenvolver um novo algoritmo baseado em novos conceitos e ideias.

CÓDIGO

CÓDIGO EM C

```c
//
//  FUNC FINAL
//
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 int minSaltos[_max_road_size_];           // Array com o numero mínimo de passos
precisos para chegar a cada posição
                                                 // Ex: se chegar à posição 14 em 5 saltos,
minSaltos[14] = 5;
                                                 // Se nouta iteração chegar em 7 saltos, o
programa acaba com o ramo;
                                                 // Se chegar com 4 saltos o programa começa
a usar esse ramo com o principal;

static int maxVelocidade[_max_road_size_];       // Mesma coisa do que o Array de cima, mas
desta vez conta a velocidade a que se chega
                                                 // a cada posição. Só serve para
"desempatar" ramos que podem ter chegado com o mesmo
                                                 // número de saltos mas diferentes
velocidades.
                                                 // Ex: Se 2 ramos chegarem com 6 saltos à
posição 16, o que tiver mais velocidade contínua,
                                                 // caso sejam a mesma, ambos ramos
continuam até se desempatarem noutra posição


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;

  //  Ver se é solução. Neste código chegar a uma solução implica que é
  // sempre a melhor, pois náo foi cortada antes de lá chegar
  if(position == final_position && speed == 1) {
    // this solution is always the best
    solution_2_best = solution_2;
    solution_2_best.n_moves = move_number;
    return;
  }
```

```
  // Como não é solução, podemos continuar o código
  for(new_speed = speed + 1;new_speed >= speed - 1;new_speed--) {

    if(new_speed > 0 && new_speed <= max_road_speed[position + new_speed] && position +
new_speed <= final_position) {

      for(i = 0;i <= new_speed && new_speed <= max_road_speed[position + i];i++);

      if(i > new_speed) {

        //  Este é o código simples que verifica se vale a pena continuar o ramo ou não
        //  Apenas começa a cortar ramos se já houver pelo menos uma solução
        // (assim garantimos sempre uma solução, sem isto pode haver problemas
        // para alguns final_positions)

        //  Como o primeiro ramo vai sempre pelas velocidades mais altas possíveis (menos
no fim),
        // há uma grande chance de ser o melhor, mas não podemos assumir isso.

        //  O ramo é cortado caso já se tenha passado pela mesma posição com um número
menor de
        // movimentos ou maior velocidade num ramo anterior, logo qualquer solução nova é
        // intrínsecamente melhor que a anterior, pois não foi cortada até acabar.

        if (solution_2.positions[solution_2_best.n_moves] != 0){
          if (new_speed <= maxVelocidade[position+new_speed] &&
              move_number+1 >= minSaltos[position+new_speed]) {
            continue;
          }
        }

        //  Como o ramo continuou, podemos afirmar que o numero de passos utilizados
        // para chegar a esta posição é o menor até agora e que a velocidade com que
        // lá chegamos é a menor de todas as iterações anteriores, logo podemos atualizar
        // os valores do número de saltos minimos e velocidade máxima desta posição e de
        // todas as posições intermédias por que passa-mos.

        // Posição atual
        minSaltos[position] = move_number+1;
        maxVelocidade[position] = speed;

        // Posições intermédias
        for (int n = 1; n < new_speed; n++) {
          minSaltos[position+n] = move_number+1;
          maxVelocidade[position+n] = new_speed;
        }

        // próximos ramos
        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_1: bad final_position\n");
    exit(1);
  }
  memset( minSaltos, _max_road_size_, final_position*sizeof(minSaltos[0]));
  memset( maxVelocidade, 0, final_position*sizeof(maxVelocidade[0]) );
  memset( solution_2.positions, 0, final_position*sizeof(solution_2.positions[0]));
  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;
}
```

CÓDIGO EM MATLAB

```
clear all;

%% Declaração dos valores
nVector = [10:1:50 55:5:100 110:10:200 220:20:800];
resultVector = [1.133e-06 2.428e-06 8.700e-07 1.075e-06 ...];

%% Plot dos valores
figure(1);
hold on;
plot(nVector, resultVector, "+r");
plot(nVector, resultVector, "b");
legend("Solução recursíva final");
axis([0 800 0 10e-06]);
xlabel("n");
ylabel("CPU Time (segundos)");
grid on;
hold off;
```

WEBGRAFIA

> https://unix.stackexchange.com/
> https://superuser.com/
> https://stackoverflow.com/
> https://linuxhint.com/
> https://en.wikipedia.org/
> https://dev.to/
> ttps://lcn2.github.io/mersenne-english-name/tenpower/tenpower.html