Trabalho Prático 3: Beating the house!

1. Introdução

1.1 Introdução ao problema

Nesse trabalho, o problema tratado simula um jogo de cassino. O jogo funciona da seguinte maneira: O jogador recebe um valor inicial e uma sequência de valores, em sequência. O jogador deve somar ou subtrair cada um desses valores em ordem, e o objetivo ao final é obter o maior valor posível, mantendo sempre o valor acumulado entre zero e um limite determinado pelo jogo. Caso esse valor final seja maior que um limite determinado, o jogador vence. O objetivo do nosso programa é encontrar o melhor valor possível ao final do jogo.

Para tratar esse problema, temos duas aproximações: uma utiliza a força bruta, testando todas as possibilidades existentes de soma e subtração de cada valor. É fácil perceber que essa opção executa várias operações, e portanto não é muito eficiente. Dessa forma, gostaríamos de obter um algoritmo menos custoso para resolver o problema. Assim, temos nossa outra aproximação, que se baseia no paradigma de Programação Dinâmica. Ela utiliza a memorização de dados que pela força bruta precisariam ser calculados várias vezes, fazendo com que esses valores possam ser calculados apenas uma vez. Isso gera um ganho considerável de performance.

Ao longo do trabalho, o funcionamento de cada um desses algoritmos será explicado e suas performances serão medidas. Além disso, exibiremos uma solução paralelizada desse problema, ou seja, dividiremos o problema de forma que ele possa ser tratado por vários processadores ao mesmo tempo, o que permite também uma maior rapidez na solução do problema.

1.2 Utilização de Programação Dinâmica

Dois paradigmas de programação muito utilizados são o paradigma guloso e a programação dinâmica. Nesse trabalho, utilizamos a programação dinâmica, pois o paradigma guloso não é aplicável. O paradigma guloso se baseia em, a partir de um ponto inicial, tomar sempre a escolha ótima local para o problema, e através desse processo obter uma solução ótima global. O que acontece é que, nesse problema, obter um ótimo local “prende” a resposta, e pode “podar” soluções que seriam de fato ótimas. Isto é, uma decisão que no momento pode parecer subótima pode levar a um resultado ótimo no final, e vice versa. Exemplo: Temos o valor inicial 5, limite 10, mínimo 10, e sequência 5 3 7. No início do jogo, a decisão ótima local seria somar o valor inicial a 5, obtendo 10, que é o mínimo e o limite. Porém, em seguida, encontramos os valores 3 e 7, que não permitem voltar ao 10 novamente, e portanto perdemos o jogo. Nesse caso, a melhor solução seria subtrair 5 do valor inicial, mesmo que inicialmente pareça uma decisão subótima, e em seguida somar os valores 3 e 7, obtendo 10. Podemos ver então que o paradigma guloso não se aplica ao problema.

2. Solução do problema

2.1. Algoritmo força-bruta

O algoritmo mais simples desse trabalho foi o que resolve o problema através da força bruta, isto é, testando cada possibilidade existente. Seu funcionamento é recursivo, e é dado pelo seguinte algoritmo:

```
bruteForce (sequencia, montante, maximo):
    se montante < 0 ou montante > maximo
        return -1
    soma = montante + sequencia[0]
    subtracao = montante - sequencia[0]
    se |sequencia| = 1:
        return maxValido(soma, subtracao)
    senao:
        proximoSoma = bruteForce(sequencia[1:], soma, maximo)
        proximaSubtracao = bruteForce(sequencia[1:], subtracao, maximo)
        return maxValido(proximoSoma, proximoSubtracao)
```

Sendo que a notação lista[n:] representa a sublista que contém todos os elementos de lista a partir do n-ésimo elemento (inclusive), e a notação |lista| representa o tamanho da lista.

Uma poda aplicada consiste no código não continuar se o montante corrente já saiu dos limites.

2.2. Algoritmo com programação dinâmica

O algoritmo de força bruta calcula várias vezes algumas operações que ocorrem repetidas vezes. Para evitar essa redundância, utilizamos um algoritmo de programação dinâmica. Esse algoritmo funciona da seguinte maneira: inicialmente criamos uma matriz de dimensões S x L, sendo S o tamanho da sequência incluindo o valor inicial, e L é o limite + 1, para englobar também a possibilidade de se obter um montante corrente igual ao limite. Assim, preenchemos toda a matriz com o valor 0. Em seguida, na primeira linha, trocamos para 1 o valor da célula de coluna igual ao valor inicial. Para cada linha i subsequente, olhamos cada coluna j na linha i - 1. Para cada célula encontrada dessa forma que seja igual a 1, preenchemos a célula na linha i e colunas j + sequencia[i] e j – sequencia[i] (desde que essas colunas estejam entre 0 e o limite) com o valor 1. Dessa forma, uma célula c[i][j] na matriz igual a 1 indica que utilizando algum dos possíveis montantes correntes gerados pela carta anterior (i - 1), é possível gerar o valor j a partir de uma soma ou subtração com a carta na posição i (sequencia[i]). Para descobrir o valor desejado, no final da execução do algoritmo o programa checa a última linha da matriz. A célula de valor igual a 1 que estiver na coluna mais à direita possível (maior valor de j, sendo j o número da coluna), representa que o valor a ser retornado é o número de sua coluna. Caso a última linha consista apenas de zeros, a função retorna -1. A seguir mostramos um exemplo para o caso de limite 10, valor inicial 5 e sequência 5 3 7:

	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	0	1	0	0	0	0	0
5	1	0	0	0	0	0	0	0	0	0	1
3	0	0	0	1	0	0	0	1	0	0	0
7	1	0	0	0	0	0	0	0	0	0	1

2.3. Algoritmo paralelo (utilizando programação dinâmica)

Esse último algoritmo funciona da mesma forma que a implementação com programação dinâmica, porém ele utiliza-se de paralelismo, separando o processamento entre threads. Nesse caso, fizemos uma paralelização de dados da seguinte forma: Uma determinada linha da matriz de montantes correntes pode ser calculada toda de uma vez, isto é, se estamos preenchendo uma linha i da matriz, e portanto inspecionando os valores na linha i - 1, o resultado da célula matriz[i][x] não depende do resultado de uma célula matriz[i][y], para quaisquer x e y. Os valores das células nessa linha dependem apenas dos valores na linha anterior. Dessa forma, para cada linha i, estamos sempre lendo da linha i - 1 e escrevendo na linha i, o que não causa conflitos (caso dois threads tentem escrever 1 na mesma posição, eles podem gerar conflitos, mas que no final acabarão por preencher a célula com o valor 1, sem efeitos colaterais). Portanto, separamos uma quantidade de colunas para cada thread. Para cada linha i na tabela (a partir da primeira), cada thread inspeciona os valores na linha i - 1 e em um intervalo de colunas dado e preenche os valores na linha i.

O paralelismo deve ser sincronizado para evitar conflitos. Nesse caso, as threads devem estar sempre na mesma linha, ou o algoritmo gerará resultados incorretos. Para garantir a sincronia, utilizamos o pthread_join da biblioteca pthreads, que pausa o programa principal enquanto uma determinada thread não terminar sua execução. O programa principal, em cada linha da matriz, separa os dados e passa para cada thread uma parcela das colunas que ela deve processar. Em seguida, espera que todas as threads terminem o processamento, utilizando o pthreads_join. Depois, passa para a próxima linha, repetindo o procedimento até que acabem as linhas. O resultado final é obtido da mesma forma que no algoritmo anterior.

3. Análise de complexidade

A análise de complexidade será feita baseada na variável n, que representa a quantidade de elementos na sequência dada, e L, que representa o limite superior do intervalo permitido no jogo. As funções mais importantes são analisadas a seguir:

- **resolveJogoBF**: Executa o algoritmo de força bruta descrito, passando por cada combinação possível. Para cada número na lista, essa função tenta somá-lo e subtraí-lo do montante corrente. Portanto, para cada número, temos duas possibilidades, ou seja, surgem dois novos caminhos que levam a soluções diferentes. Assim, cada novo número na lista multiplica por dois o número de possíveis resultados. Uma sequência de um

elemento tem duas possibilidades: valorInicial + sequencia[0] e valorInicial - sequencia[0]. Inserindo mais um elemento nessa lista, passamos a ter, para cada uma das possibilidades anteriores, duas novas respostas. Adicionando mais um elemento, duplicamos mais uma vez o número de respostas, e assim sucessivamente. Portanto, temos **O(2^n)**.

- **resolveJogoPD**: Executa o algoritmo baseado em programação dinâmica descrito. Esse algoritmo passa por cada célula da matriz descrita seção 2.2, que possui dimensões n + 1 e L. Assim, possui complexidade de **tempo O(n * L)**. Obviamente, a complexidade de **espaço** tem o mesmo valor, dadas as dimensões da matriz: **O(n * L)**.
- **resolveJogoThreads:** Opera da mesma forma que a função **resolveJogoPD**, apenas paralelizando a execução do algoritmo. Ou seja, não possui diferença em sua complexidade. **Tempo: O(n * L)**. **Espaço: O(n * L)**.

Depois dessa análise, podemos ver que há uma grande melhoria de desempenho do algoritmo de programação dinâmica em relação ao de força bruta, pois o força bruta tem complexidade exponencial, o que cresce rapidamente e torna o programa inviável para muitos casos, enquanto o algoritmo que utiliza programação dinâmica executa em tempo polinomial, o que torna o programa viável para muito mais casos.

4. Análise experimental

Para a análise experimental do código, utilizamos arquivos de testes com sequências de tamanhos variados, e limites variados no caso da estratégia que utiliza programação dinâmica.

Para medir o tempo passado, utilizamos a biblioteca <time.h> para contar o número de clocks necessários para executar cada caso de teste, e usamos a variável CLOCKS_PER_SEC para calcular o tempo tomado por cada caso. Os testes foram feitos em uma máquina com processador Core i5 com frequência de 2.6GHz e 4GB de memória RAM.

Medimos o tempo levado para executar o programa com sequências de tamanho variando de 5 em 5, iniciando com uma sequência de tamanho 5 e terminando em uma de tamanho 40, no caso do força bruta, e 50, nos outros casos. Executamos cada caso de teste cinco vezes, e o tempo final foi a média de todos os tempos obtidos.

4.1. Análise utilizando Força bruta

Para o programa de força bruta, obtivemos o seguinte gráfico. O eixo horizontal do gráfico representa o tamanho da sequência recebida, e o eixo vertical representa o número de milissegundos tomados para a execução do programa. Não consideramos a variável L (limite do intervalo) pois ela é irrelevante para esse algoritmo.

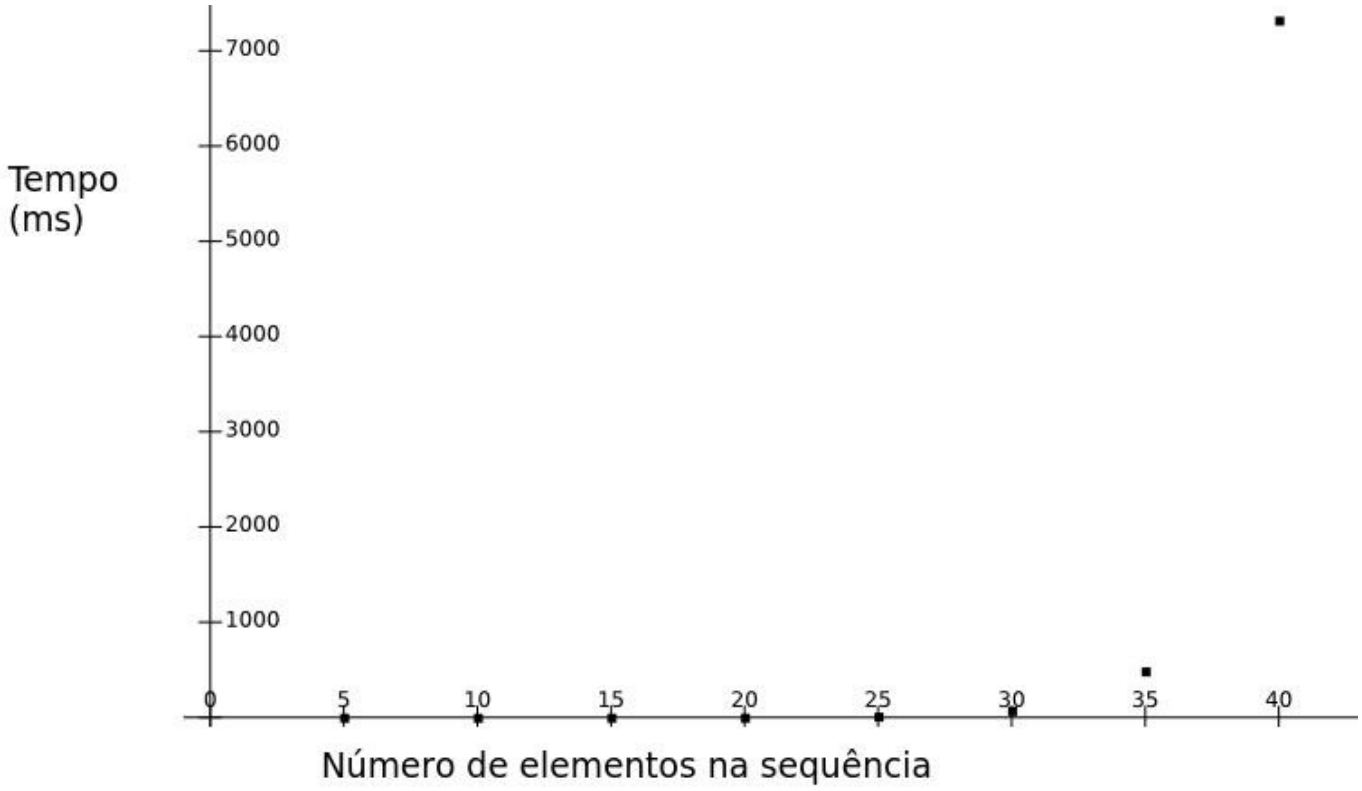

Gráfico 1 – Tempo x Número de elementos na sequência (algoritmo de força bruta)

Podemos ver o comportamento exponencial desse algoritmo, pois o tempo de execução “explode” rapidamente. Caso entradas maiores que 40 fossem utilizadas, a visualização dos pontos seria difícil, pois a maior parte dos pontos pareceria estar sobre o eixo do número de elementos. Isto é: em perspectiva, pareceriam tomar 0 milissegundos para sua execução, tamanho o tempo de execução para as entradas maiores.

4.2. Análise utilizando Programação Dinâmica

Para o algoritmo que utiliza programação dinâmica, temos um fator complicante: existem duas variáveis que impactam no tempo de execução. Para uma melhor visualização, faremos o seguinte: teremos 3 gráficos, cada um com um valor específico para o limite L. Em cada gráfico, teremos vários tamanhos de sequência n. Assim, poderemos comparar a diferença entre esses gráficos e entender o impacto da variável L. O eixo horizontal do gráfico representa o tamanho da sequência recebida, e o eixo vertical representa o número de milissegundos tomados para a execução do programa.

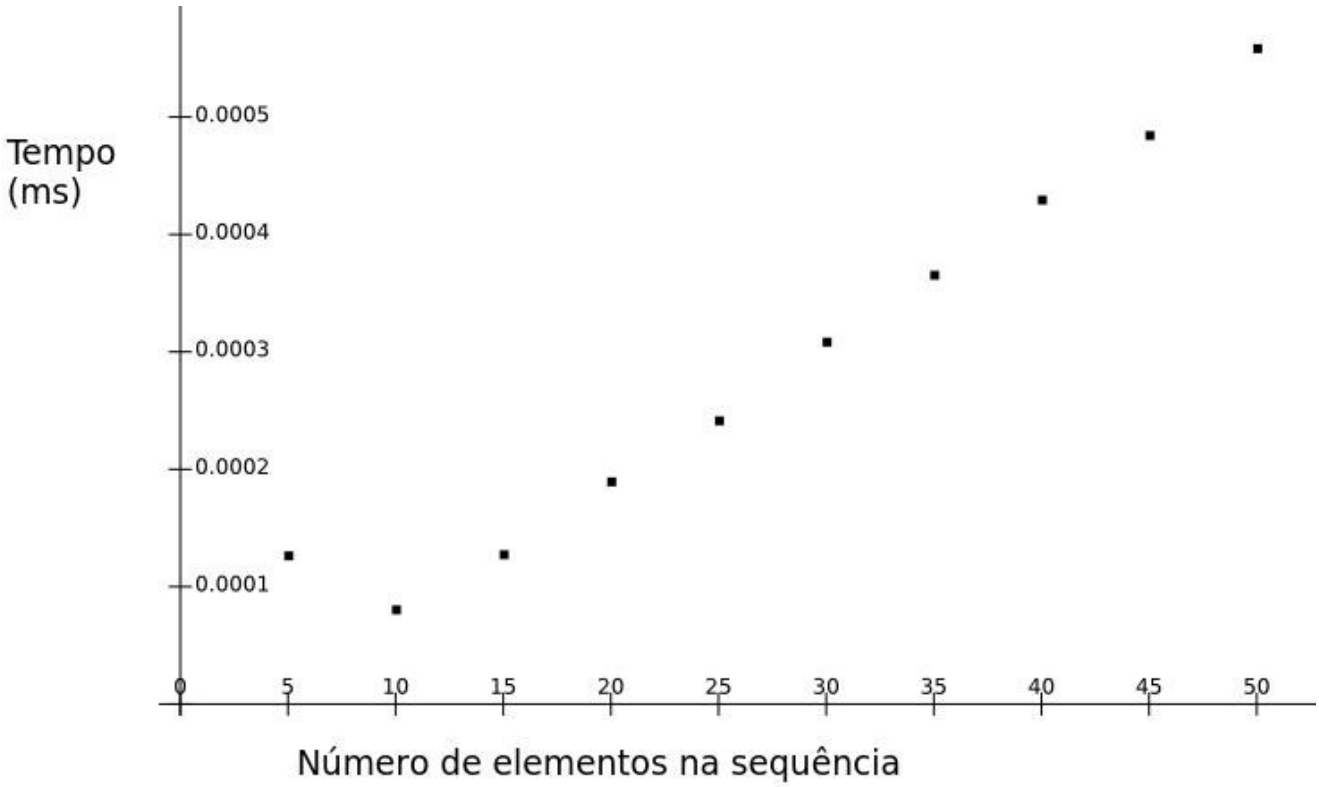

Gráfico 2 – Tempo x Número de elementos na sequência (programação dinâmica com limite igual a 1000)

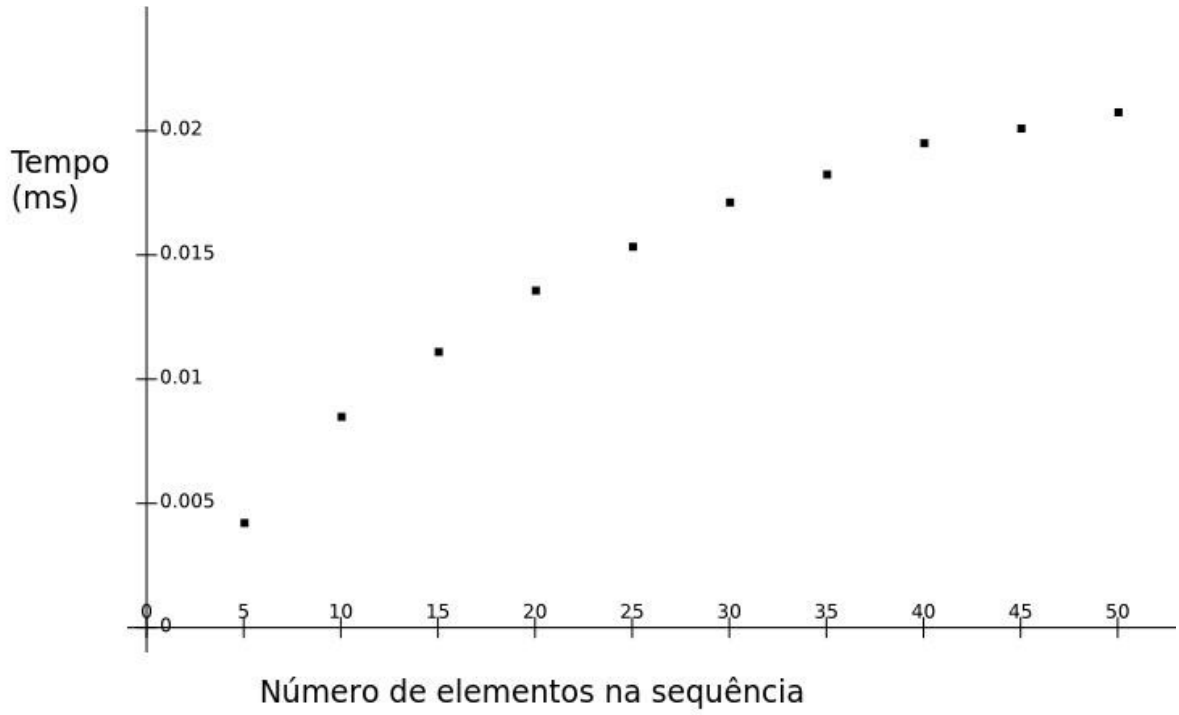

Gráfico 3 – Tempo x Número de elementos na sequência (programação dinâmica com limite igual a 100000)

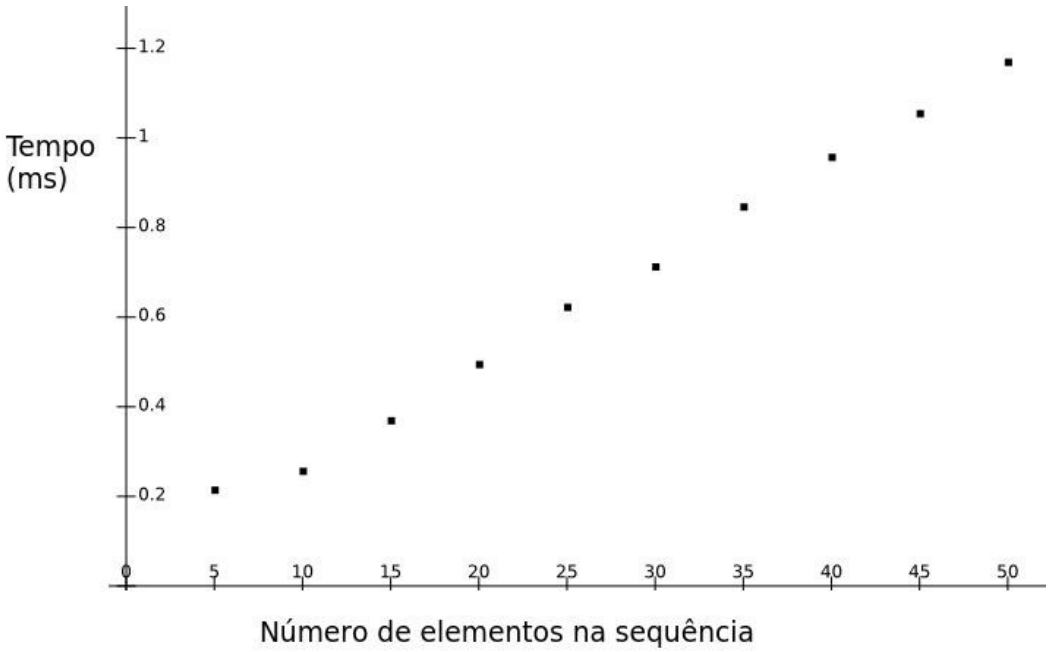

Gráfico 4 – Tempo x Número de elementos na sequência (programação dinâmica com limite igual a 10000000)

Observamos uma relação linear entre a quantidade de elementos na sequência e o tempo decorrido para a execução do programa em todos os gráficos, o que faz sentido considerando nossa análise de complexidade efetuada anteriormente. Vemos também que o aumento do valor do limite L aumenta o tempo tomado para a execução do programa, e observando os valores, é possível perceber uma relação linear entre o valor do limite do intervalo e o tempo decorrido.

Nesses gráficos, podemos ver a melhoria desse algoritmo em relação ao de força bruta. O crescimento do tempo de um ponto para outro é muito menor, e os testes com mais de 40 elementos na sequência, que no algoritmo de força bruta nem foram executados em tempo hábil, são executados rapidamente utilizando programação dinâmica.

4.3. Análise utilizando Paralelização

O algoritmo paralelizado utiliza a mesma técnica de programação dinâmica que acabamos de analisar, portanto não é muito relevante compararmos seu desempenho em relação ao algoritmo de força bruta. Essa comparação já está feita na análise experimental em 4.2. Para esse algoritmo, faz mais sentido compararmos seu desempenho em relação ao de programação dinâmica original. Para isso, utilizaremos um caso de teste que engloba todos os testes colocados em gráfico anteriormente, ou seja, que possui todos os casos de teste com 5, 10, 15, …, 50 elementos. Utilizaremos os casos que possuem limite igual a 10000000. Para essa comparação, o mais relevante é exibirmos o tempo levado para executar o caso de teste com N threads, e variarmos o número de threads. Faremos isso através de uma tabela:

Nº de threads	Não-paralelizado	5	10	15	20	25	30
Tempo (milissegundos)	6742	3794	3559	3532	3636	3625	3753

Pela Lei de Amdahl, temos que:

$$S = D/P$$

Sendo que S é o speedup (melhoria de desempenho) gerado pelo algoritmo paralelo, D é o tempo levado pela execução direta (serial) do programa, e P é o tempo levado pela execução paralela do programa, para determinado número de threads. Assim, queremos maximizar o valor do speedup S. Para isso, devemos escolher o número de threads que leva o menor tempo para executar o programa. Dentre as amostragens feitas, a quantidade de threads que leva menor tempo é 15. Assim, o speedup **S = 6742/3532 ≈ 1.91**. Isto é, o desempenho do programa quase dobra utilizando um algoritmo paralelizado com 15 threads.

5. Conclusão

Ao fim do trabalho, pudemos entender a aplicação de paradigmas de programação especializados e paralelização para melhorar a performance de algoritmos que normalmente levariam muito tempo para ser executados. Pudemos entender o tradeoff Tempo de execução vs Memória proposto pela estratégia de programação dinâmica, pois ela executa em tempo bem menor, mas tem o preço de ocupar mais memória que uma solução "ingênua". Pudemos também entender as dificuldades de implementação de um algoritmo paralelizado, a necessidade de uma sincronização elaborada, e o equilíbrio exigido na escolha da quantidade de threads, para que elas sejam suficientes para agilizar a solução do problema, mas não sejam o suficiente para que o overhead gerado leve a um tempo de execução maior do que o da versão serializada do mesmo algoritmo.

Pudemos observar também que a análise experimental concorda com a análise de complexidade de tempo calculada.