**Documentação TP3**
AEDS3

**Ícaro Harry**
Matrícula: 2014033050

# 1 Introdução

O trabalho proposto no TP3 foi a simulação de um jogo de Cassino, onde dada uma sequência de números, deveríamos determinar o maior valor possível de se obter apenas somando ou subtraindo os números. Para ganhar o jogo, era necessário que o valor obtido fosse maior do que um limite mínimo. Além disse as operações não poderiam ultrapassar o limite inferior de 0 e o limite superior dado em cada instância do problema.

O principal desafio do TP era desenvolver uma estratégia de força bruta e melhorá-la através de um algoritmo de Programação Dinâmica ou Guloso. Além disso, deveríamos ecolher uma das duas abordagens e desenvolver um algoritmo paralelizado para o problema.

# 2 Solução do Problema

Como abordado na seção anterior, para resolver o problema, primeiro deveríamos desenvolver uma solução de força bruta e depois escolher entre uma abordagem Gulosa ou uma abordagem de Programação Dinâmica para melhorar o problema.

## 2.1 Força bruta

A estratégia de força bruta pode ser facilmente implementada de maneira recursiva, percorrendo todas as possibilidades possíveis de soma ou subtração, como demonstrado na figura abaixo:

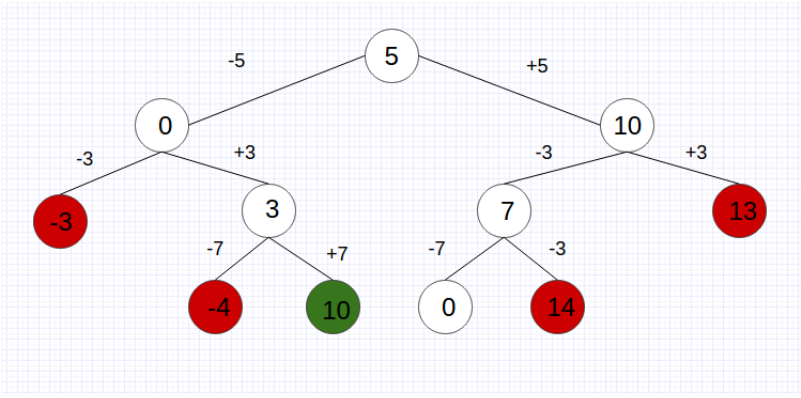


Figure 1: Toy example

O exemplo acima demonstra o Toy example, onde iniciando com o valor 5, devemos utilizar 5, 3 e 7 para obter a maior soma possível, dentro do limite máximo de 10. Então o algoritmo testa todas as possibilidades, podando quando elas ultrapassam o limite de 0 ou 10, o que está destacado em vermelho na figura. No último passo da recursão, o algoritmo busca o maior valor de todos, no caso 10, que está destacado em verde.

O algoritmo implementado de maneira recursiva é o seguinte:

**Algorithm 1** Brute Force

**function** PLAYGAME(*seqSize*, *sum*, *limit*, *sequence*, *element*, *k*, *bigger*)
  **if** (*sum* + *element* minor or equal *limit* and *k* minor than *seqSize*) **then**
    playGame(*seqSize*, *sum* + *element*, *limit*, *sequence*, *sequence*[*k* + 1], *k* + 1, *bigger*);
  **end if**
  **if** (*sum* - *element* bigger or equal 0 and *k* minor than *seqSize*) **then**
    playGame(*seqSize*, *sum* - *elemnt*, *limit*, *sequence*, *sequence*[*k* + 1], *k* + 1, *bigger*);
  **end if**
  **if** *k* == *seqSize* **then**
    *bigger* = *sum* bigger than *bigger* ? *sum* : *bigger*;
  **end if**
**end function**

## 2.2 Programação Dinâmica ou Algoritmo Guloso?

É simples determinar qual abordagem serve para obter uma solução ótima para o problema. Sabemos que um algoritmo guloso caminha para a solução ótima tomando sempre a melhor decisão local. No problema em questão, a escolha local vai ser sempre somar ou subtrair o número X do número Y. Porém é fácil perceber que, tomando uma decisão ótima do maior número obtido válido, entre a soma e a subtração, não nos permite eliminar a outra possibilidade, pois o próximo valor poderá tornar a solução escolhida inválida, já que ele pode tornar a soma maior do que o limite superior ou menor do que 0. É possível enxergar isso no Toy Example desenhado acima. A escolha ótima local levaria para o lado direito da árvore de possibilidades, porém o resultado ótimo está do outro lado.

Sendo assim, para resolver o problema de maneira ótima e, ao mesmo tempo, melhorar a performance do algoritmo de força bruta, escolhi utilizar uma abordagem de Programação Dinâmica.

### 2.2.1 Resolução com Programação Dinâmica

O algoritmo de Programação Dinâmica utiliza uma matriz para marcar os valores das somas obtidas em cada passo. A tabela abaixo ilustra o Toy Example:

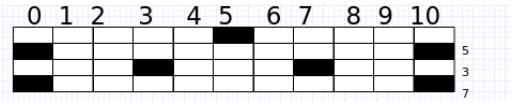


Figure 2: Toy example

Cada linha da matriz corresponde a um estado das somas, sendo o estado inicial fornecido na entrada. As colunas representam os possíveis valores no intervalo entre 0 inclusive e X inclusive. Dessa maneira, a cada passo do algoritmo, marca-se os próximos valores somados. Se a soma ultrapassar os limites, não é feito nada. Ao final do algoritmo, o resultado estará na última linha da matriz e corresponderá ao índice da última posição marcada.

**Algorithm 2** Programação Dinâmica

```
function playGame(seqSize, sum, limit, sequence)
    m = createMatrix(seqSize + 1, limit + 1);
    m[0][sum] = true;
    for i = 0; i minor than seqSize; i++ do
        for j = 0; i minor or equal limit; j++ do
            if m[i][j] then
                if (j + sequence[i] minor or equal limit) then
                    m[i + 1][j + sequence[i]] = true;
                end if
                if (j + sequence[i] bigger or equal 0) then
                    m[i + 1][j − sequence[i]] = true;
                end if
            end if
        end for
    end for
    for i = limit; i bigger or equal 0; i– do
        if m[seqSize][i] then
            return i;
        end if
    end for
    return −1;
end function
```

# 3 Análise Teórica do Custo Assintótico de Tempo

O problema proposto, na abordagem por força bruta, testa todas as possibilidades possíveis de soma e subtração de cada elemento, apenas podando aquelas somas que ultrapassam os limites estabelecidos. Dessa maneira, como podemos observar na figura 1, a cada passo do algoritmo, ou seja, a cada novo número para se somar e subtrair, multiplica-se por dois os números que devem ser testados, o que nos leva a uma complexidade de $O(2^S)$, sendo S o número da sequência dada na entrada.

Podemos observar que o algoritmo de Programação Dinâmica reduz consideravelmente a complexidade em relação ao algoritmo de força bruta. Para chegar ao resultado final, deve-se percorrer a matriz, linha por linha, checando se cada posição está marcada ou não. Ou seja, a complexidade de tempo é dada pelo tamanho da matriz a ser percorrida, no caso é S + 1 por X + 1, sendo S o tamanho da sequência dada e X o limite superior dado. Para encontrar o maior resultado na última linha da matriz, deve-se percorrê-la de trás para frente, o que nos dá uma complexidade de O((S+1) * (X+2)) = O(S*X).

# 4 Análise Teórica do Custo Assintótico de Espaço

O algoritmo de força bruta utiliza apenas um vetor com os números que compõem a sequência dada na entrada, dessa maneira, a complexidade de espaço é O(S).

Já a programação dinâmica, claramente troca computação por espaço, para evitar de se percorrer todas as $2^S$) possibilidades, deve-se criar uma matriz de tamanho S + 1 por X + 1. Sendo assim, a complexidade de espaço é O(S*X).

# 5 Análise Experimental do Custo Assintótico

Para verificar o custo assintótico, realizei testes utilizando entradas que aumentavam de 1 em 1, como demonstrado no algoritmo abaixo. Para medi-los utilizei a função clock da biblioteca time.h.

**Algorithm 3** Testes

```
function TEST
    for i = 0; i minor than limit; i++ do
        for j = 0; j minor than i; j– do
            sequence[j] = 1;
        end for
        begin = clock();
        playGame(i, 0, limit, 30, sequence);
        end = clock();
        timeSpent = (double)(end - begin) / CLOCKS_PER_SEC;
    end for
end function
```

Os resultados obtidos estão descritos nos seguintes gráficos:

# 6 Conclusão

Pude concluir com o presente Trabalho Prático que problemas que aparentemente possuem natureza exponencial, podem ser resolvidos com soluções muito mais eficientes utilizando Programação Dinâmica.

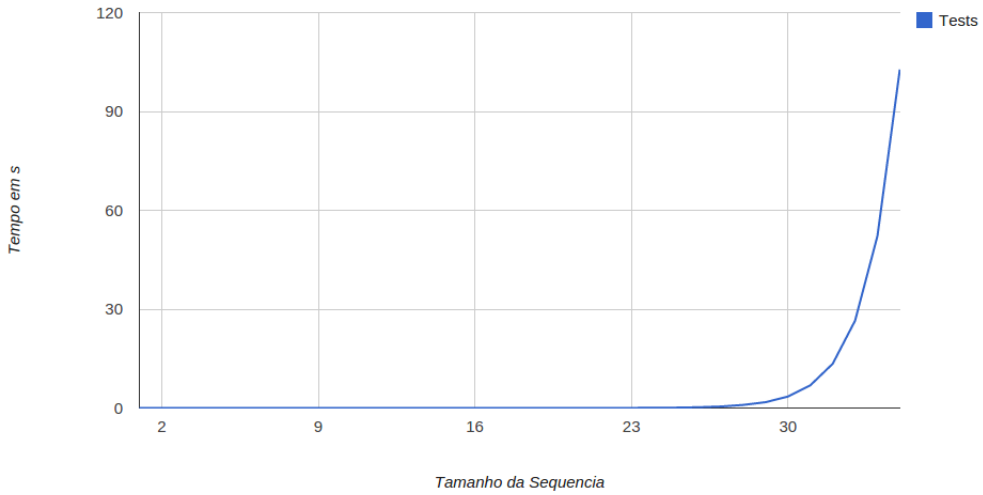


Figure 3: Força bruta com entradas de 1 a 35

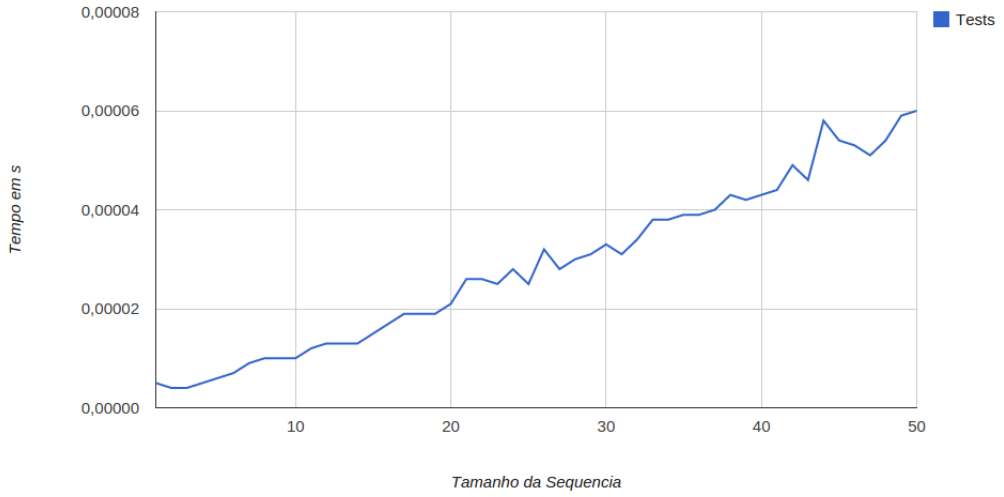


Figure 4: Programação Dinâmica com entradas de 1 a 50