ED2 - Aula 3.pdf - Rafael Manfrim

Considere que temos o seguinte problema: queremos contar o número de pares em um vetor que somam a um valor específico. Modelando computacionalmente, nossa entrada é um vetor de n elementos e nossa saída é um número inteiro representando o número de pares que somam a um valor v.

Por exemplo, suponha que o vetor seja [1, 4, 3, 2] e o valor para a soma seja 5. Vamos encontrar todos os pares de elementos que somam 5:

- O par (1, 4) soma 5.
- O par (3, 2) soma 5.

Então, neste exemplo, há 2 pares de elementos que somam 5. Vamos criar um algoritmo que resolve esse problema.

```
#include<iostream>
int conta_pares_sol_um(int *vetor, int n, int soma) {
    int pares = 0;
    for(int i = 0; i < n - 1; i++) {
        for(int j = i + 1; j < n; j++) {
            if(vetor[i] + vetor[j] == soma)
                pares++;
        }
    }
    return pares;
}
int main() {
    int vetor[] = {1, 5, 7, 4, 2, 5};
    int n = sizeof(vetor) / sizeof(vetor[0]);
    int soma = 6;
    int pares = conta_pares_sol_um(vetor, n, soma);
    std::cout << "Número de pares:" << pares;
    return 0;
}
```

Determine a sua complexidade de tempo no pior caso utilizando a notação-O.

```
  Temos dois laços de repetição, o primeiro executará n − 1 vezes e o segundo
também n - 1 vezes para cada execução do primeiro, contudo, no segundo laço, a
```

quantidade de execuções diminui um para cada execução do primeiro.

O número de comparações executado por esses dois laços é $(n - 1) + (n - 2) + ... + (n - (n - 2)) + (n - (n - 1))$.

Isso é uma P.A. então podemos aplicar a fórmula $(n \times (a1 + an)) / 2$:

$((n - 1) \times (1 + n - 1)) / 2$

$(n^2 - n) / 2$

E agora podemos excluir os termos de menor ordem: $O(n^2)$

Aqui está outra solução

```
#include<iostream>
int conta_pares_sol_dois(int *vetor, int n, int soma) {
    int ocorrencias[1000] = {0}; //Assumindo que os valores estão entre 0 e 999
    int pares = 0;
    for(int i = 0; i < n; i++) {
        int complemento = soma - vetor[i];
        if(complemento >= 0 && ocorrencias[complemento])
            pares += ocorrencias[complemento];
        ocorrencias[vetor[i]]++;
    }
    return pares;
}
```

Como ela funciona? Determine a sua complexidade de tempo no pior caso utilizando a notação-O.

A função percorre todos os elementos de um vetor e verifica quantos pares de elementos nele somam um determinado valor.

Para isso ela salva todos os números anteriores e usa o complemento de um número para verificar se ele número já possui um par adequado.

Dado o vetor [2, 6, 7, 1] e soma = 8:

Iteração 1 (vetor[0] = 2):

- complemento = 8 - 2 = 6: Não há ocorrência de 6 ainda.
- ocorrencias[2]++

Iteração 2 (vetor[1] = 6):

- complemento = 8 - 2 = 2: Uma ocorrência de 2 foi encontrada.
- pares += ocorrencias[complemento];
- ocorrencias[6]++

E assim por diante.

A complexidade de tempo no pior caso é $O(n)$, pois será necessário percorrer o vetor inteiro.

Encontrar a soma de subvetores de tamanho fixo em um vetor. Consiste em calcular a soma de todos os subvetores de um determinado tamanho fixo k em um vetor dado. Um subvetor é uma sequência contígua de elementos dentro de um vetor. Por exemplo, vamos considerar o vetor V = [1, 2, 3, 4, 5] e k = 3.

- Subvetor 1: [1, 2, 3] Soma = 1 + 2 + 3 = 6
- Subvetor 2: [2, 3, 4] Soma = 2 + 3 + 4 = 9
- Subvetor 3: [3, 4, 5] Soma = 3 + 4 + 5 = 12

Portanto, para o vetor V , com k = 3, o resultado é 6, 9 e 12. Criem um algoritmo que resolva este problema.

```
#include <iostream>

using namespace std;

int main() {
    int vetor[] = {1, 2, 3, 4, 5};
    int n = 5;

    int k = 3;
    int tamanho_resposta = n - (k - 1);
    int resposta[tamanho_resposta];

    for (int i = 0; i < tamanho_resposta; i++) {
        int final = i + k;

        int soma = 0;

        for(int j = i; j < final; j++) {
            soma += vetor[j];
        }

        resposta[i] = soma;
    }

    for (int i = 0; i < tamanho_resposta; i++) {
        cout << resposta[i] << " ";
    }

    return 0;
}
```

Determine a sua complexidade de tempo no pior caso utilizando a notação-O.

```
  O loop de fora executa para a quantidade de valores contidos na resposta, o que pode ser de 1 até n.
  O loop de dentro, executa pela quantidade de k.
  Com isso, temos que a quantidade de execuções será: (n - (k - 1)) x k
  Em notação O podemos dizer que as subtrações são a parte menos significativa, então pode ficar O(n x k)
  Ou até mesmo podemos dizer que é O(n)
  Esse código tem o melhor caso quando k = 1 ou k = n, ambos rodarão exatamente em O(n). Os outros casos, o código executará um pouco mais, mas ainda assim muito próximo de O(n).
```

Caso crie um algoritmo de complexidade quadrática ou similar, ponderem: é possível criar um algoritmo mais eficiente? Caso não seja possível, argumente a razão dessa impossibilidade. Caso seja possível, escreva o algoritmo e determine a sua complexidade utilizando a notação-O.

```
  O algoritmo acima já é provavelmente um dos mais eficientes possíveis para essa solucão, pois não seria possível encontrar todas as possibilidades de soma sem necessitar percorrer o vetor em pelo menos n execuções.
```