Relatório Projeto 1 AED 2023-2024

Nome: João António Faustino Vaz Nº Estudante: 2022231087
PL (inscrição): 8

Registar os tempos computacionais das 3 soluções. Os tamanhos das arrays (N) devem ser: 20000, 40000, 60000, 80000, 100000. Só deve ser contabilizado o tempo do algoritmo. Exclui-se o tempo de leitura do input e de impressão dos resultados. Devem apresentar e discutir as regressões para as 3 soluções, incluindo também o coeficiente de determinação/regressão (r quadrado).

Tabela para as 3 soluções

Tamanho do array x	Tempo (ms) Algoritmo 1 y	Tempo (ms) Algoritmo 2 y2	Tempo (ms) Algoritmo 3 y3
20000	145.979851	4.5953	0.09807529999
40000	498.6517	8.6034	0.19654279999
60000	1106.0459500002	13.764382	0.259536999
80000	2012.647908	17.438	0.3539195
100000	3180.882058	23.514008	0.4142035

Gráfico para a solução A

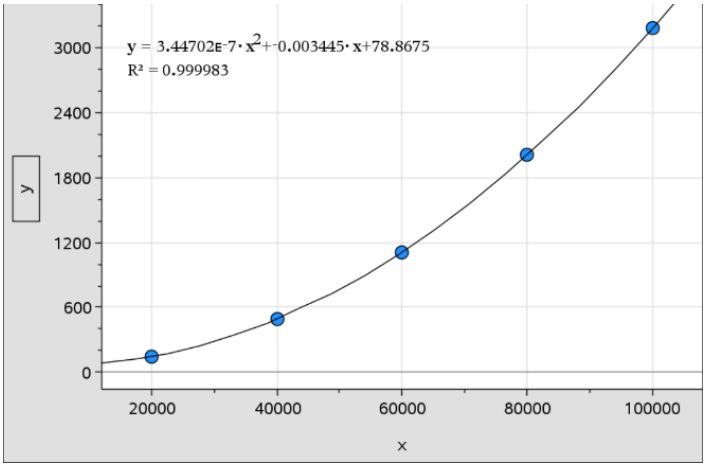


Gráfico para a solução B

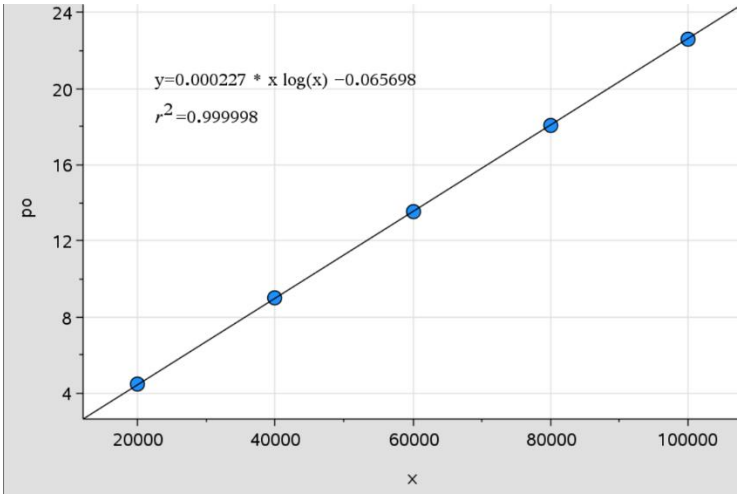
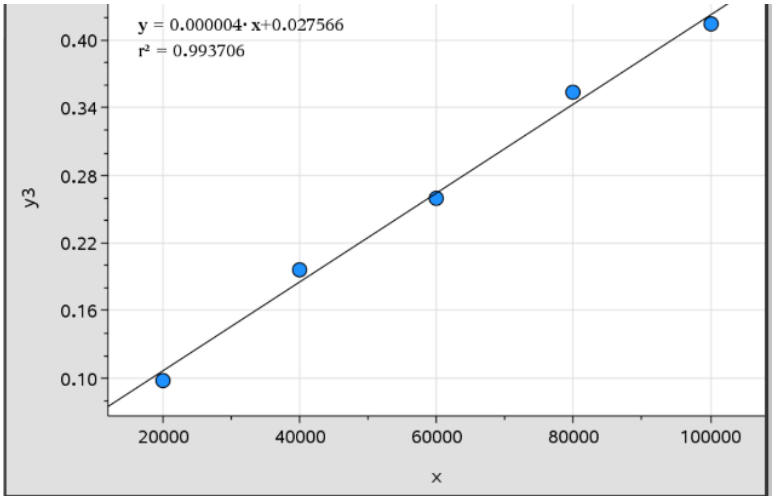


Gráfico para a solução C



Análise dos resultados tendo em conta as regressões obtidas e como estas se comparam com as complexidades teóricas:

Na solução A usei uma solução exaustiva, usando dois for('s). Para cada elemento do ArrayList, percorremos os outros todos e verificamos se algum satisfaz a condição(pares de números cuja soma seja igual a k), logo a complexidade deste algoritmo é $O(n^2)$ e, portanto, observa-se um crescimento polinomial de grau 2 (quadrático) dos tempos em função do tamanho do array.

Na solução B tirei partido do ordenamento dos elementos do ArrayList para encontrar rapidamente pares de números cuja soma seja igual a k, logo a complexidade deste algoritmo é O (n log(n)) e que provém da função Collections.sort do Java e, portanto, observa-se um crescimento semelhante ao linear dos tempos em função do tamanho do array.

Na solução C usei um array auxiliar com zeros de modo a registar os valores únicos e percorrer o ArrayList uma vez, mantendo um registo dos números já vistos no array auxiliar e tenta encontrar um par de números que satisfaça a condição, logo a complexidade deste algoritmo é O (n) e, portanto, observa-se um crescimento linear dos tempos em função do tamanho do array.

Notas sobre o trabalho realizado:

A linguagem escolhida para este projeto foi Java.

O tempo de execução foi calculado em nanossegundos(ns) usando a função System.nanoTime() e posteriormente convertido para milissegundos(ms).

Para todos os tamanhos de array a testar foram feitas 1000 repetições, sendo o tempo registado na tabela a média do tempo de execução dessas mesmas repetições.

Foi usado nos três algoritmos implementados ArrayLists de modo a conseguirmos usar a função Collections.sort() do Java.

Na solução A sendo uma solução exaustiva, mesmo que a função encontre um par que satisfaz a condição a função não dá return e, portanto, continua a percorrer o ArrayList mesmo que já se tenha encontrado a solução.

Note-se que nem sempre o uso de dois for's implica uma complexidade $O(n^2)$, mas no caso da solução A é mesmo $O(n^2)$.

Na solução C recorri à função fill() para preencher o array auxiliar com zeros.

O código utilizado para o input foi utilizado nos três algoritmos.

Código dos algoritmos

Input

```java
public class algoritmo1 {
    public static void main(String[] args) {
        double total=0.0;
        int num=1000;
        for (int z = 0; z < num; z++) {
            int size = 100000;
            Random random = new Random();
            int k;
            k = random.nextInt( bound: 2 * size) + 1;
            ArrayList<Integer> arrayList = new ArrayList<>();
            for (int i = 0; i < size; i++) {
                arrayList.add(random.nextInt(size) + 1);
            }
            long tempoInicial = System.nanoTime();
            algoritmo1Funcao(arrayList, k);
            long tempoFinal = System.nanoTime();
            long tempoExecucao = tempoFinal - tempoInicial;
            double tempoExecucaoMilisegundos = (double) tempoExecucao / 1000000.0;
            total+=tempoExecucaoMilisegundos;
        }
        System.out.println("O tempo medio foi: " + (total/num) + " milisegundos");
    }
```

Algoritmo 1

```java
    public static boolean algoritmo1Funcao(ArrayList<Integer> arrayList, int k) {
        boolean resultado=false;
        for (int i = 0; i < arrayList.size(); i++) {
            for (int j = i; j < arrayList.size(); j++) {
                if (((arrayList.get(i) + arrayList.get(j)) == k) && (arrayList.get(i) != arrayList.get(j))) {
                    resultado=true;
                }
            }
        }
        return resultado;
    }
}
```

Algoritmo 2

```java
public static boolean algoritmo2Funcao(ArrayList<Integer> arrayList, int k, int size) {
    Collections.sort(arrayList);
    int i = 0, j = size-1;
    while (i < j){
        int sum = arrayList.get(i) + arrayList.get(j);
        if ((sum == k) && (arrayList.get(i) != arrayList.get(j))) {
            return true;
        } else if (sum > k) {
            j--;
        } else {
            i++;
        }
    }
    return false;
}
}
```

Algoritmo 3

```java
public static boolean algoritmo3Funcao(ArrayList<Integer> arrayList, int k,int size) {
    int[] c = new int[size*2];
    Arrays.fill(c, val: 0);
    int aux;
    for(Integer num:arrayList){
        if(num<k && c[num]!=1){
            c[num]=1;
            aux=k-num;
            if(arrayList.contains(aux) && aux!=num){
                return true;
            }
        }
    }
    return false;
}
}
```