p

Relatório Projeto 1 AED 2023-2024

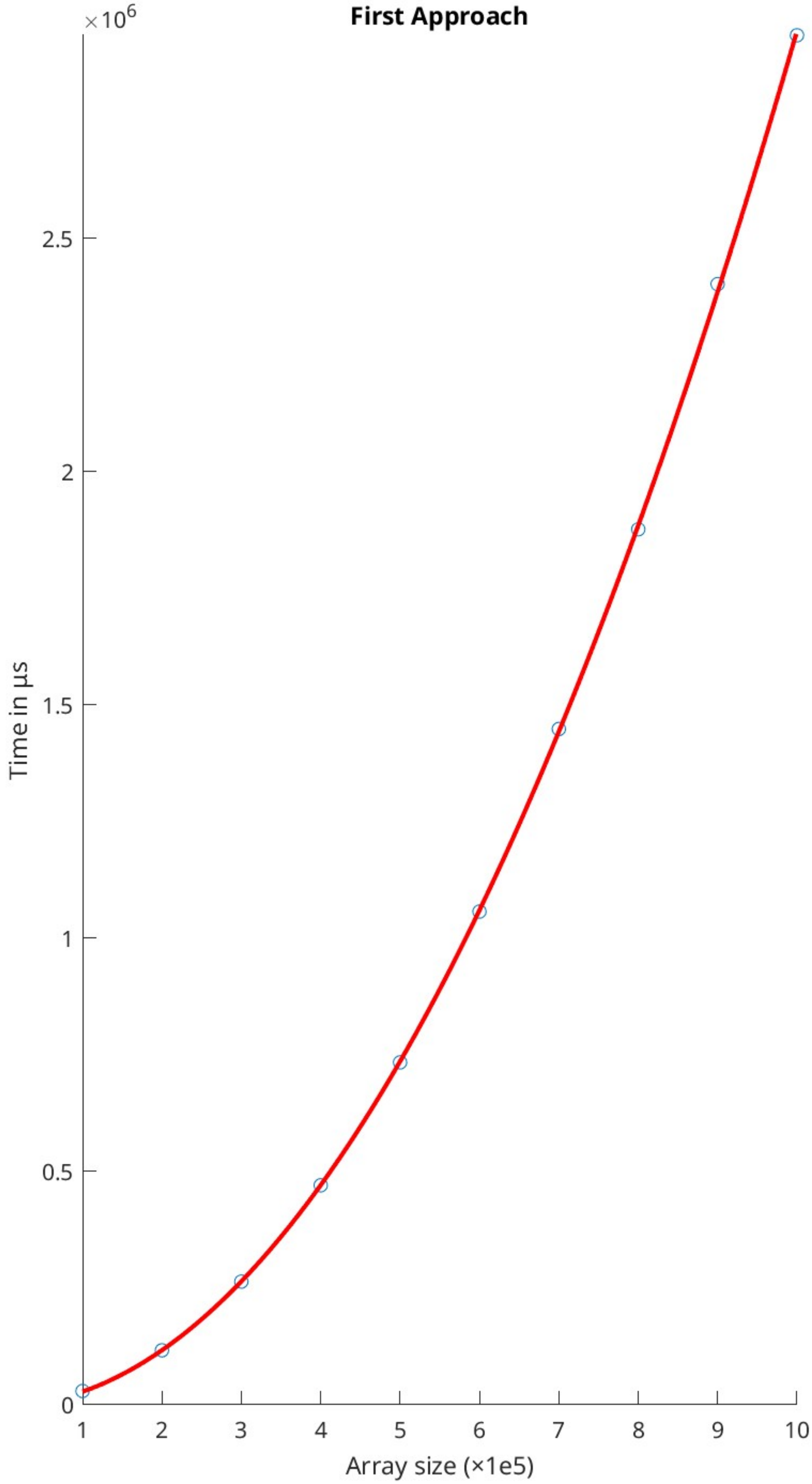
Nome: Nuno Batista Nº Estudante:2022216127
PL (inscrição): PL8

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).

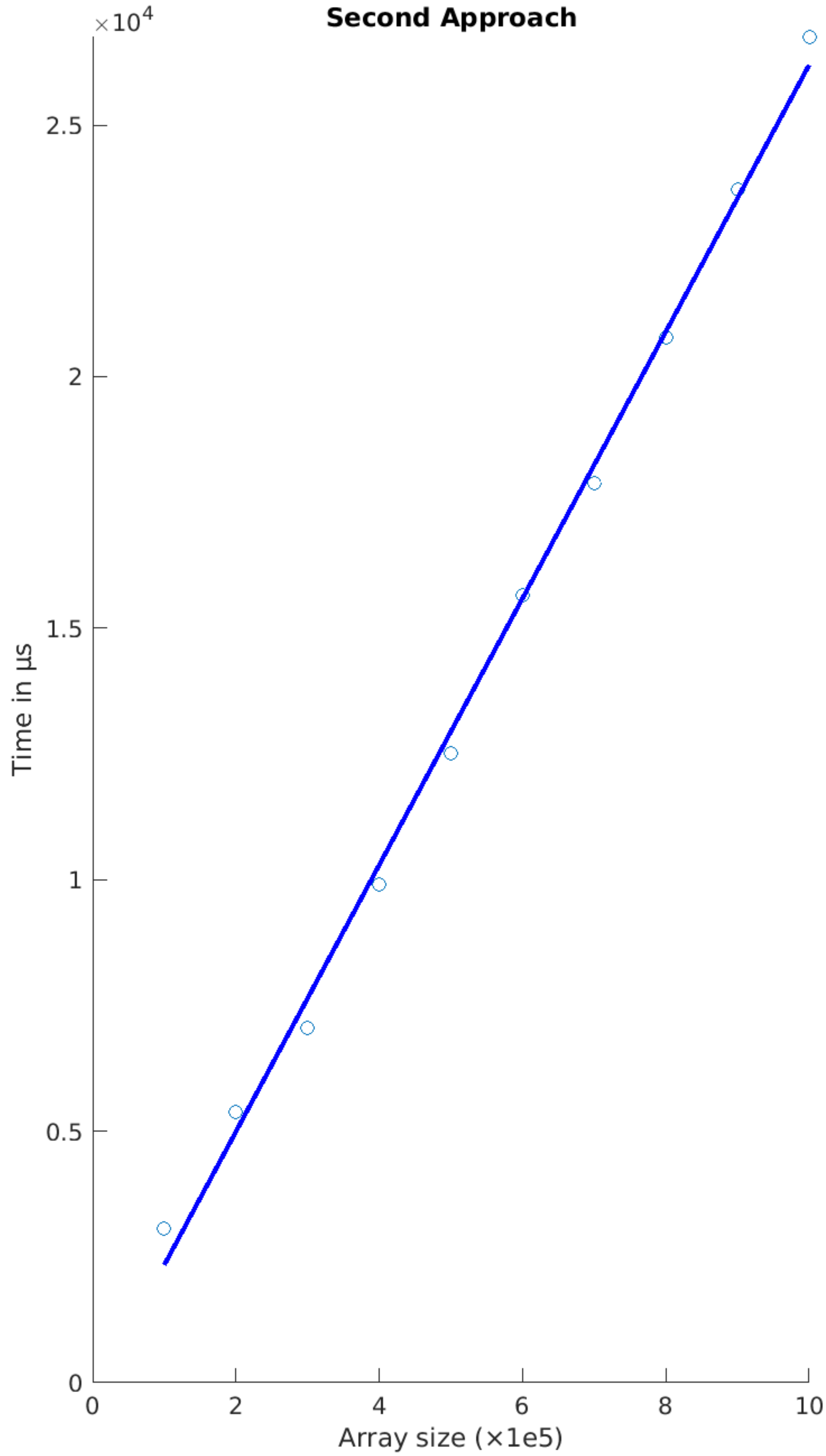
Tabela para as 3 soluções

	First Approach	Second Approach	Third Approach
N = 10000	29639	3057	42
N = 20000	116791	5378	74
N = 30000	264081	7045	220
N = 40000	470469	9906	307
N = 50000	733966	12521	367
N = 60000	1057253	15651	448
N = 70000	1448531	17873	516
N = 80000	1876498	20771	653
N = 90000	2402131	23716	779
N = 100000	2935325	26762	834

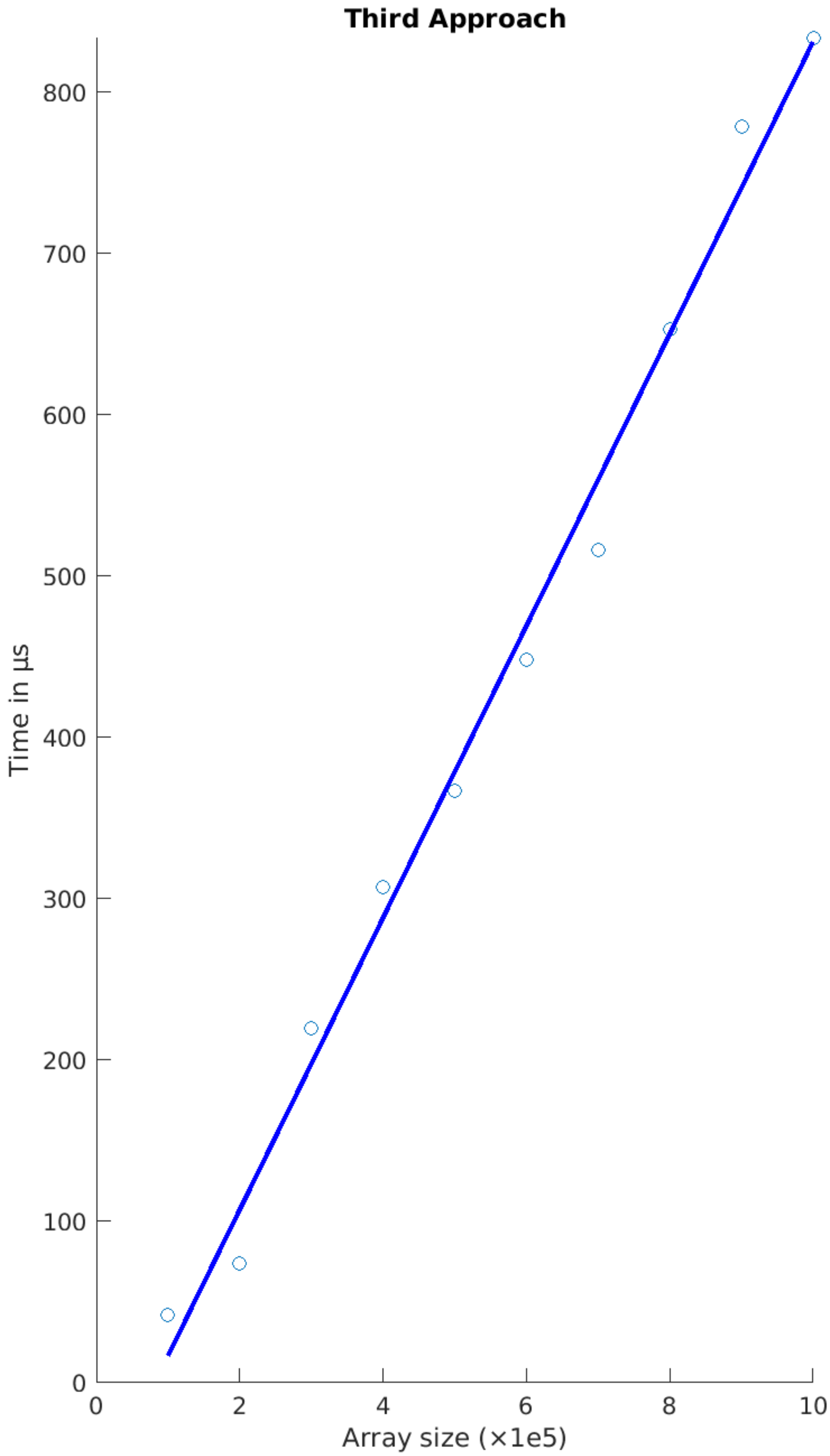
Algoritmo 1 ($R^2 = 0.9999$)



Algoritmo 2 (R^2 = 0.9968)



Algoritmo 3 (R^2 = 0.9904)



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

1 – A complexidade teórica do primeiro algoritmo é *O(N²)*, pois para cada iteração do ciclo *for* que está a percorrer os *N* elementos do array, há outro ciclo *for* que também percorre os *N* elementos.

A regressão quadrática, com $R^2 = 0.9999$, indica que a complexidade empirica está de acordo com a teórica.

2 – A complexidade teórica do segundo algoritmo é *O(NlogN)*, pois o algoritmo de sorting utilizado tem complexidade *O(NlogN)*, apesar da complexidade da pesquisa com 2 ponteiros ser *O(N)* uma vez que se trata de um ciclo while que, no pior caso possível, acaba por percorrer os N elementos do array, analisando assintoticamente, a complexidade é governada por *O(NlogN)*.

A regressão NlogN, com $R^2 = 0.9968$, também indica que os resultados empíricos vão de encontro aos resultados esperados.

3 – A complexidade teórica é *O(N)*, pois uma vez que os elementos já vistos vão sendo guardados num set, a única coisa que é preciso verificar para cada elemento *i* é se *K-i* já foi visto até ao momento. Então, basta percorrer os *N* elementos uma única vez e a complexidade temporal de procura num set é O(1).

A regressão linear, com $R^2 = 0.9904$, mais uma vez, indica que a complexidade teórica é refletida na empírica.

```
/*

Algorithm 1
Time Complexity: O(N^2)
*/
bool bruteForceSolution(std::vector<ll> arr, int k){
	int arrLen = arr.size();

	for(int i = 0; i < arrLen; ++i){
		for(int j = i; j < arrLen; ++j){
			if(arr[j] == arr[i]) continue;
		if(arr[i] + arr[j] == k)
			return true;
		}
	}
	return false;
}
```

```cpp
/*
Algorithm 2
Time Complexity: O(Nlog(N))
*/
bool twoPointersSolution(std::vector<ll> arr, int k){
	sort(arr.begin(), arr.end());
	int low = 0, high = arr.size()-1;

	while(low < high){
		if(arr[low] + arr[high] < k){
			low++;
		}
		else if(arr[low] + arr[high] > k){
			high--;
		}
		else if(arr[low] == arr[high]){
			break;
		}
		else{
			return true;
		}
	}
	return false;
}


/*
Algorithm 3
Time Complexity: O(N)
*/
bool cacheSolution(std::vector<ll> arr, int k, int sizeLimit){
	int arrLen = arr.size();
	std::unordered_set<int> dp;

	for(int i = 0; i < arrLen; ++i){
		if((k - arr[i] >= sizeLimit) || (arr[i] * 2 == k))
			continue;

		//Check if k-arr[i] is already in the dp set
		if(dp.find(k-arr[i]) != dp.end())
			return true;

		dp.insert(arr[i]);
	}
	return false;
}
```

```
//Generate random vector
auto seed = std::chrono::system_clock::now().time_since_epoch().count();
std::default_random_engine dre(seed);
std::uniform_int_distribution<int> di(0, sizeLimit);
int kLimit = 2 * sizeLimit;
std::vector<ll> arr(sizeLimit);
std::generate(arr.begin(), arr.end(), [&]{
	return di(dre);
});
```