Trabalho da disciplina:
Estrutura de Dados Básicas I

Análise Empírica

Autor: Gabriel Paes Landim de Lucena

Professor: Selan Rodrigues dos Santos

Disciplina: DIM0119 - Estrutura de Dados Básicos I

Turma: T01

Índice

1. Introdução

Este trabalho tem como objetivo apresentar uma análise empírica de vários algoritmos de busca, sendo estes as implementações iterativas e recursivas das buscas: linear, binária e ternária. Será analisado também o desempenho dos algoritmos de jump search e da busca de Fibonacci.

Após isso, serão trazidas algumas discussões sobre os resultados, como por exemplo qual algoritmo linear é mais eficiente no pior caso e se há um pior caso para a busca Fibonacci.

2. Metodologia e materiais utilizados

2.1. Método

De início, é válido informar que a linguagem usada foi C++.

Para esta análise, a metodologia a ser usada será uma comparação dos resultados obtidos com tabelas e gráficos.

O programa recebe como entrada o tamanho do vetor, o elemento a ser buscado e o algoritmo de busca a ser utilizado.

O programa gerará um vetor com o tamanho determinado pelo usuário, e cada elemento do vetor terá um valor igual ao índice. Por exemplo, v[0] = 0; v[1] = 1… v[i] = i.

Imediatamente antes do início da busca, começa-se uma contagem do tempo de execução do algoritmo selecionado com a biblioteca chrono, do C++11. Então, imediatamente após o fim da execução do algoritmo, a contagem termina. Por fim, é apresentado ao usuário o resultado da busca e o tempo de execução da busca em nanosegundos.

Todos os algoritmos retornam um ponteiro de inteiro, que pode ser:

A) O ponteiro de algum elemento do array usado na busca, o que significa que foi encontrado o valor a ser buscado;
B) O mesmo endereço da variável com o valor a ser buscado, indicando que a busca não encontrou nenhum elemento.

Cada algoritmo será testado considerando-se o pior caso. O pior caso de todos os algoritmos é quando o valor a ser buscado não encontra-se no vetor, com exceção do de busca ternária, em que o pio caso é quando o valor está em um dos limites do vetor.

2.2. Materiais

Como dito anteriormente, a linguagem de programação usada foi C++, versão 11. O compilador usado foi o g++.

Foram usados dois computadores nesta análise - um desktop e um notebook. Seguem as especificações dos mesmos.

Computador 1 (Desktop):
Placa mãe ASUS H61M-A/BR
Processador Intel Core i5 3330

8GB RAM DDR3 1600MHz
Placa de vídeo msi GTX 1060 Gaming X
HDD Toshiba 1TB SATA III 7200 RPM
SO Ubuntu 18.04

Computador 2 (Notebook):
Samsung NP270E4E
Processador Intel Core i3 3110m
4GB RAM DDR3 1600MHz
Placa de Vídeo Intel HD Graphis 4000
SO Debian 9 (Stretch)

3. Resultados

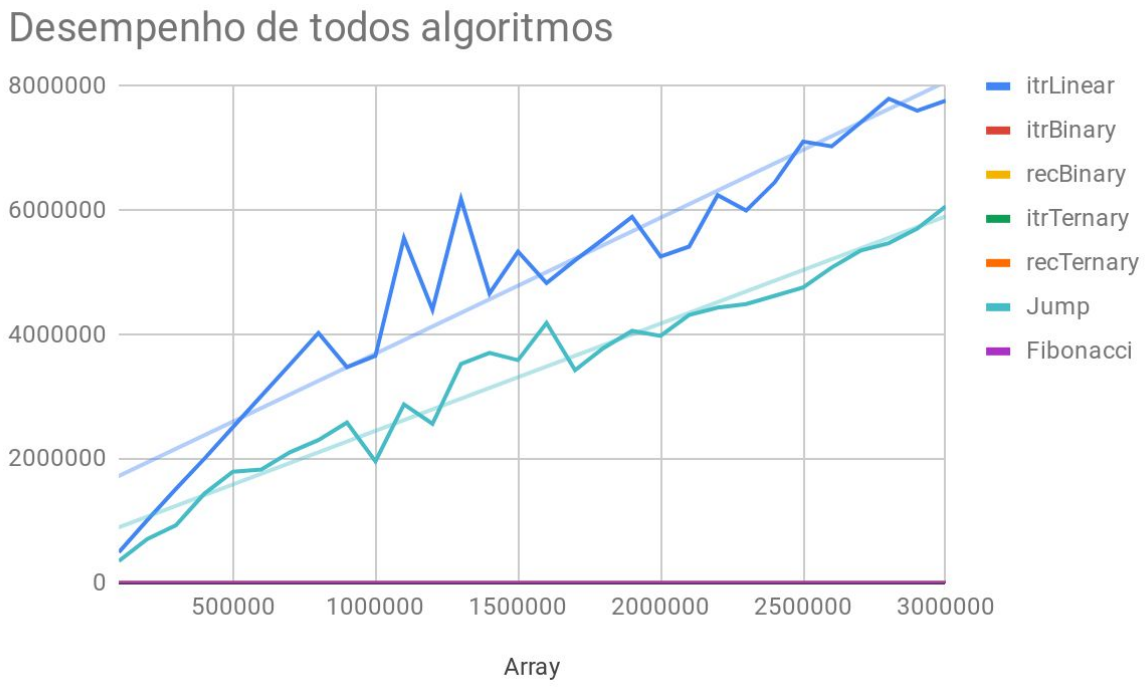
O tempo de execução de cada algoritmo foi contado em nanossegundos. Todos os testes foram feitos no Computador 1 considerando-se o pior caso.

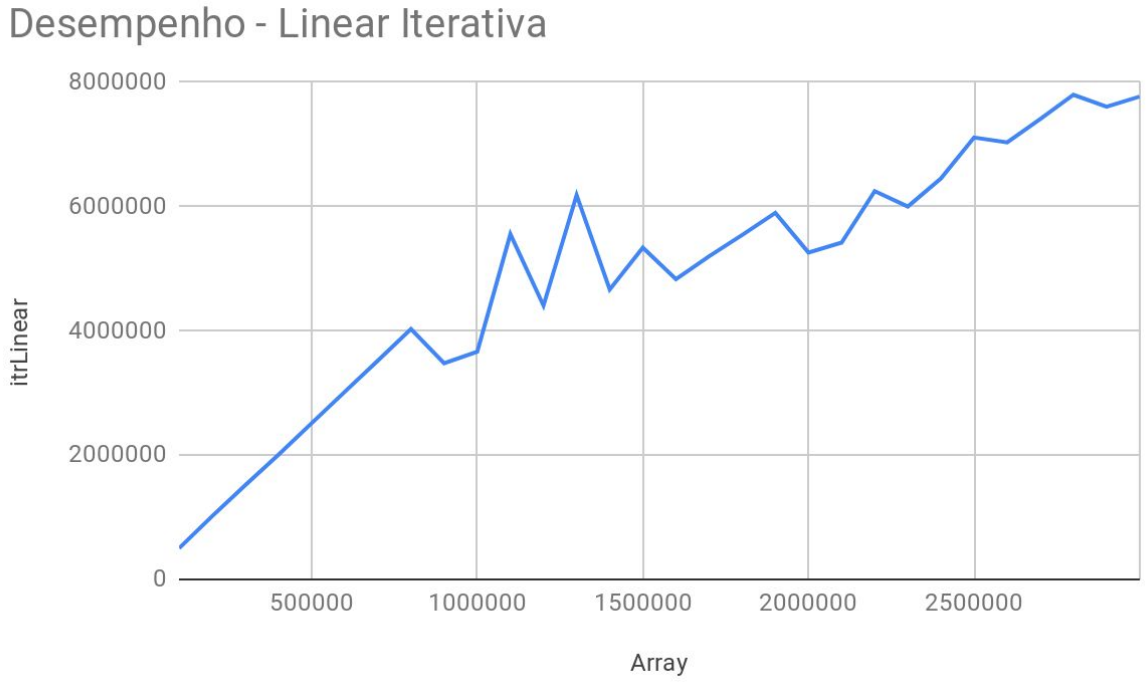
3.1 Resultados Gerais

Array	itrLinear	itrBinary	recBinary	itrTernary	recTernary	Jump	Fibonacci
100000	502083	792	1048	877	959	359971	952
200000	1018339	872	1146	1075	1152	713691	1023
300000	1517846	930	887	1149	1185	934586	1022
400000	2006253	955	938	1022	1160	1443741	1100
500000	2513381	952	1222	1243	1258	1796592	1090
600000	3018012	1042	1275	945	1137	1833604	1068
700000	3521049	977	1315	1090	1162	2110259	1169
800000	4030076	1067	1350	1089	1273	2305809	1231
900000	3479142	1161	1347	1170	1247	2587469	1303
1000000	3663089	1233	1405	1228	1283	1964539	1229
1100000	5555529	1307	1352	1223	1458	2880370	1513
1200000	4406414	1355	1448	1370	1543	2567743	1577
1300000	6184217	1340	1411	1276	1661	3531229	1423
1400000	4664569	1286	1234	1434	1721	3709781	1543
1500000	5339452	1467	1372	1362	1488	3592354	1413
1600000	4832018	1237	1419	1508	1786	4192398	1450
1700000	5199974	1186	1539	1582	1524	3432248	1491
1800000	5543501	1447	1459	1491	1485	3787467	1597
1900000	5897922	1351	1547	1698	1736	4063771	1601
2000000	5260031	1369	1468	1593	1549	3984174	1477
2100000	5419777	1188	1495	1401	1549	4319786	1580
2200000	6247565	1305	1519	1696	1597	4438559	1334
2300000	6000363	1327	1577	1544	1551	4494814	1586
2400000	6453020	1293	1597	1540	1644	4628800	1415
2500000	7111198	1257	1386	1627	1685	4763534	1480
2600000	7032724	1355	1545	1588	1699	5083037	1531
2700000	7410524	1336	1525	1569	1660	5352456	1601
2800000	7799236	1364	1564	1496	1711	5475144	1476
2900000	7608810	1270	1428	1491	1815	5710563	1488
3000000	7770306	1231	1355	1596	1497	6067370	1549

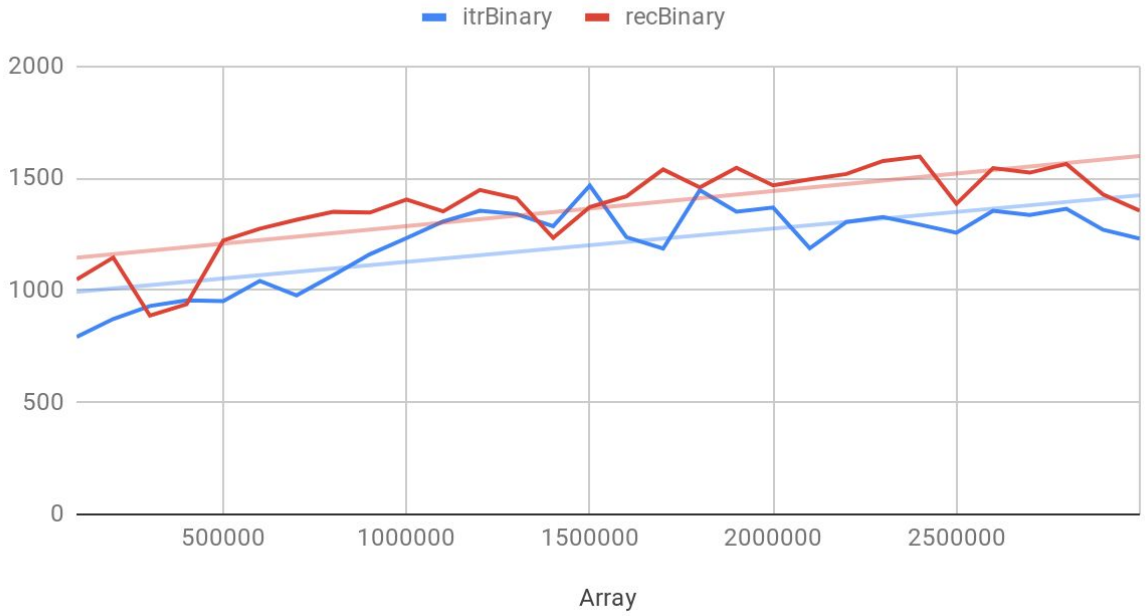
3.2. Gráficos

Para todos os gráficos, o eixo X corresponde ao tamanho do vetor e o eixo Y corresponde ao tempo em nanosegundos em que a busca foi feita.

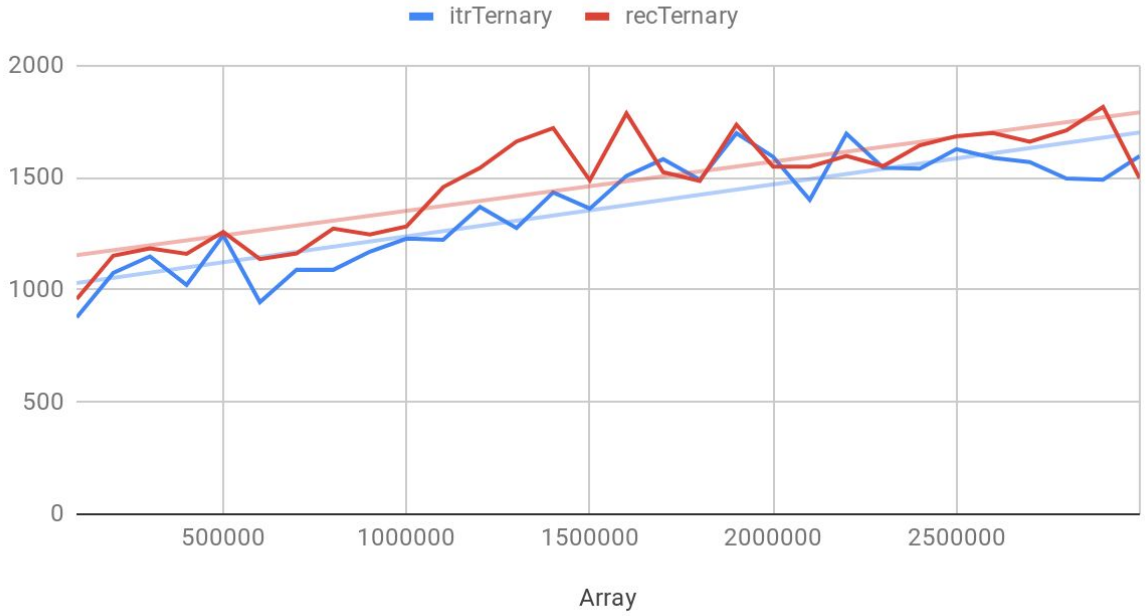


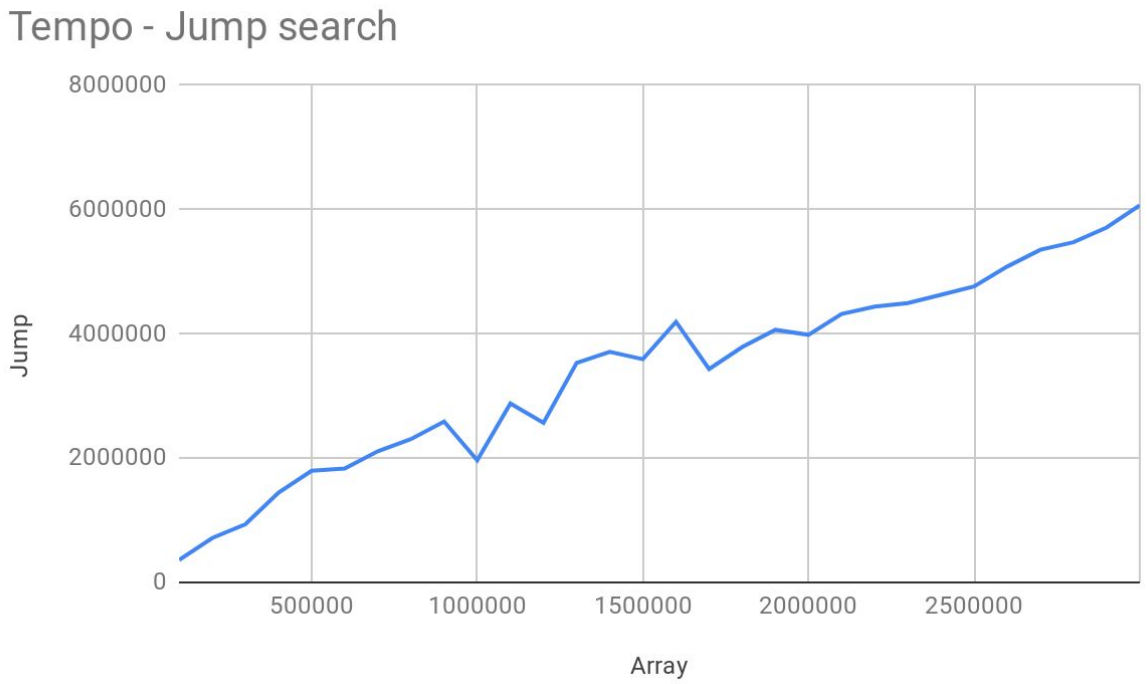


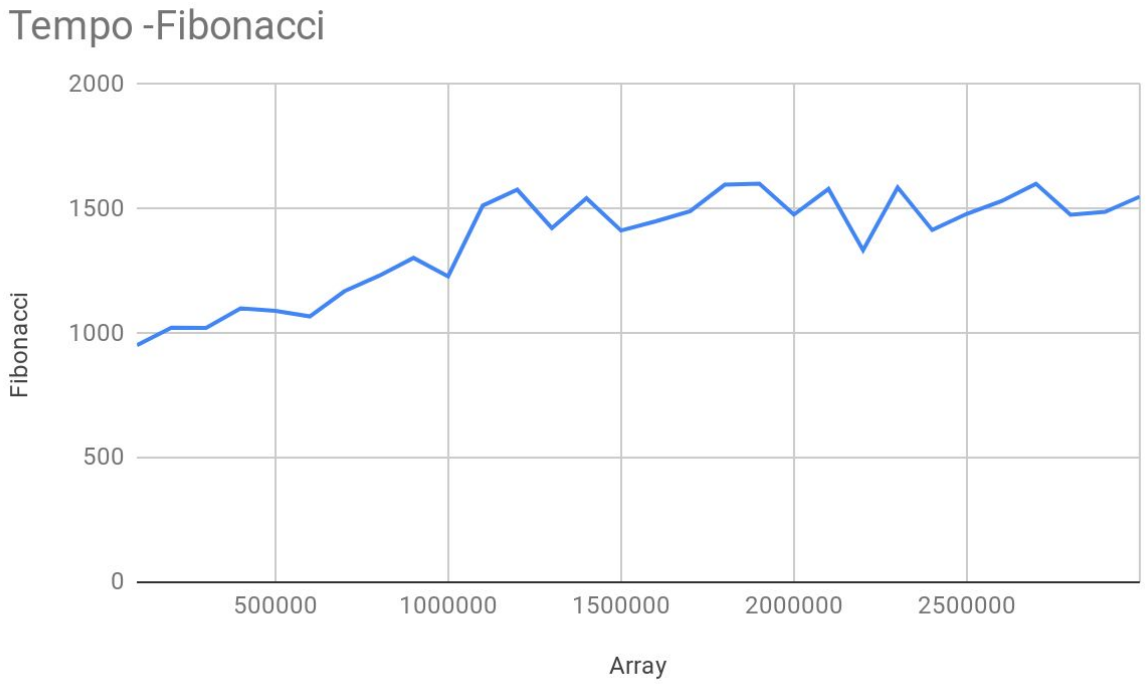
Tempo - Binária iterativa e recursiva



Tempo - Ternária iterativa e recursiva







Caso necessário, a tabela editável pode ser encontrada no arquivo “analysis table. xlsx”.

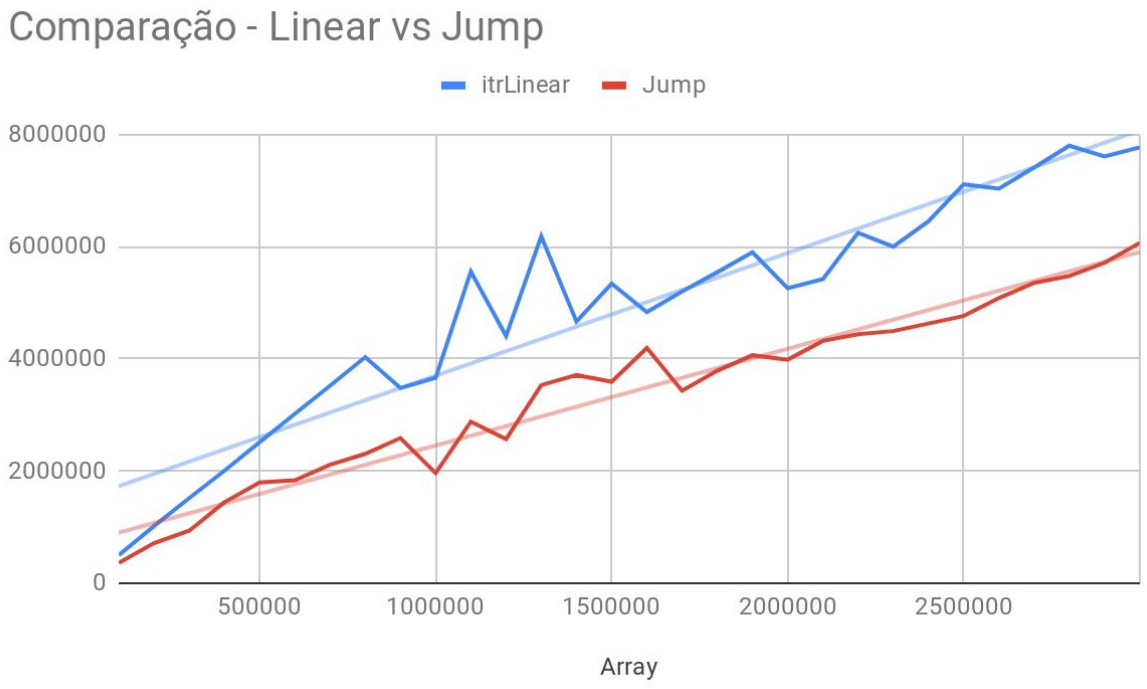
4. Discussão dos resultados

Agora, são trazidos vários questionamentos sobre os resultados obtidos anteriormente.

4.1. Eficiência de algoritmos lineares

"Considerando os dois algoritmos lineares, busca linear e jump search, qual é o mais eficiente no pior caso?"

Comparando os gráficos das duas buscas, fica evidente que o algoritmo da jump search completou a busca em menos tempo no pior caso. Para a elaboração desta pesquisa, o tamanho do salto foi de 10 elementos.



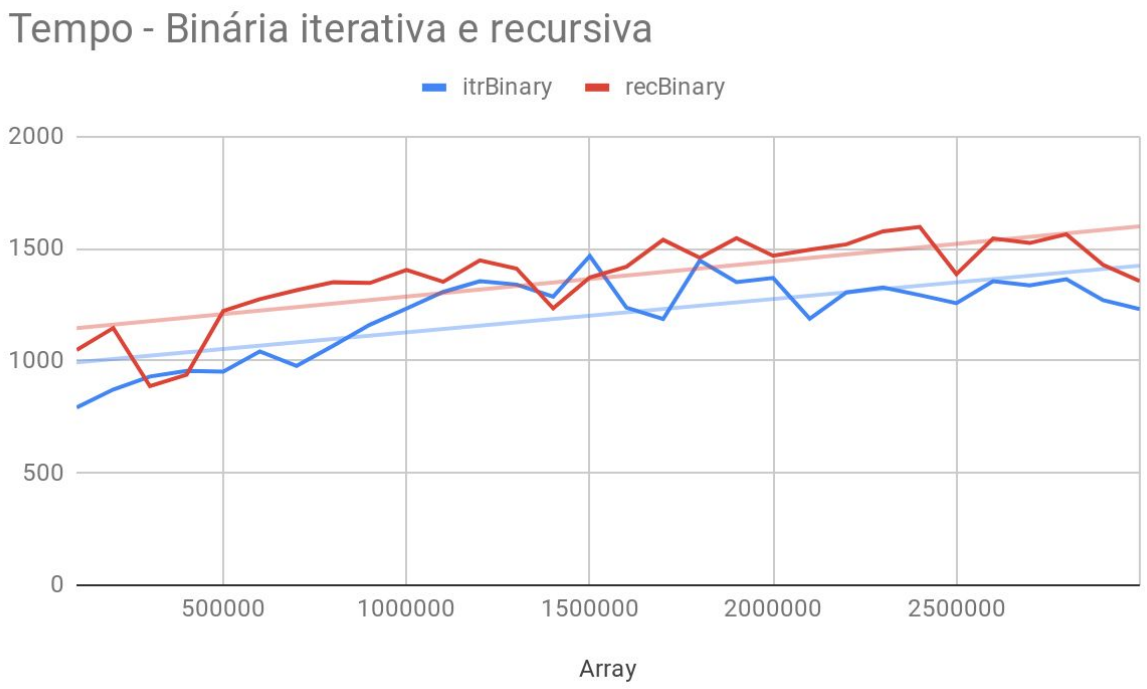
É válido notar que deve-se ficar atento ao tamanho do salto que a jump search dará. Se o tamanho do salto for muito pequeno (1, por exemplo) ou maior que o tamanho do vetor, então o tempo de execução do pior caso será bastante similar ao pior caso da busca linear, pois a quantidade de comparações é muito parecida, senão igual. Entretanto, se o tamanho do salto for um valor muito grande mas menor que o tamanho do vetor, então a busca será completa de forma ainda mais rápida, pois serão ainda menos comparações a serem feitas.

Apesar disso tudo, no fim das contas a jump search é a melhor opção, por fazer uma quantidade menor ou igual de comparações que a busca linear.

4.2 Recursão ou Iteração?

"Qual tipo de estratégia de implementação é mais eficiente, recursão ou iteração?"

Olhemos novamente o gráfico com o tempo de execução das buscas binárias iterativa e recursiva.



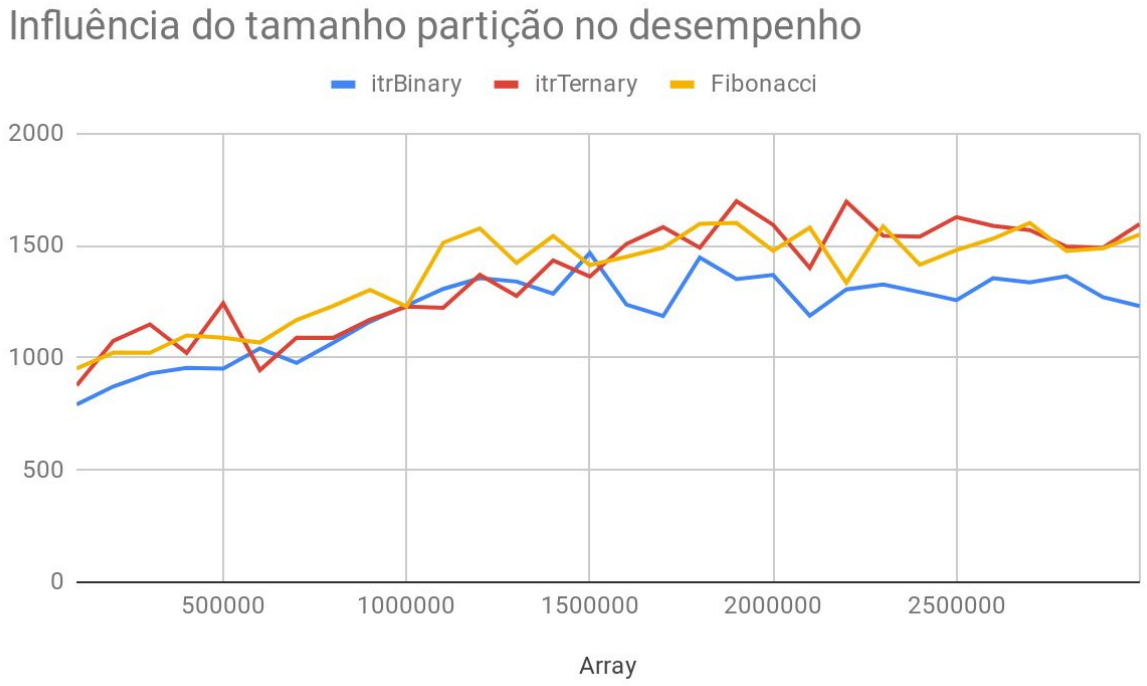
Analisando o algoritmo, podemos ver que a implementação iterativa tem uma quantidade de passos igual. Ambas fazem 4 comparações por laço, ambas calculam o termo do meio em cada laço e ambas têm a mesma quantidade de atribuição de valores a variáveis. Entretanto, a cada novo laço da busca binária recursiva, é alocado mais três endereços de memória com cada chamada da função. Isso pode ter pesado na diferença no tempo de execução presente no gráfico.

Apesar da diferença ser minúscula, pode-se concluir que a implementação iterativa é mais eficiente que a recursiva.

4.3 A influência do tamanho da partição no desempenho

"De que forma o tamanho da partição pode influenciar o desempenho em uma estratégia de busca por divisão e conquista?"

Olhemos o gráfico abaixo para melhor visualizar os resultados:



Aqui, as diferenças no tempo de execução da função não favorecem nenhuma função especificamente. Há momentos que a mais eficiente é a busca binária, outros é a ternária, e há momentos em que as três levam quase que exatamente o mesmo tempo de execução.

Com isso, a diferença no tamanho da partição de memória a ser eliminada não influencia no tempo de execução de forma que apresente um algoritmo que seja mais eficiente que os demais.

4.4 Algoritmos de complexidade diferentes

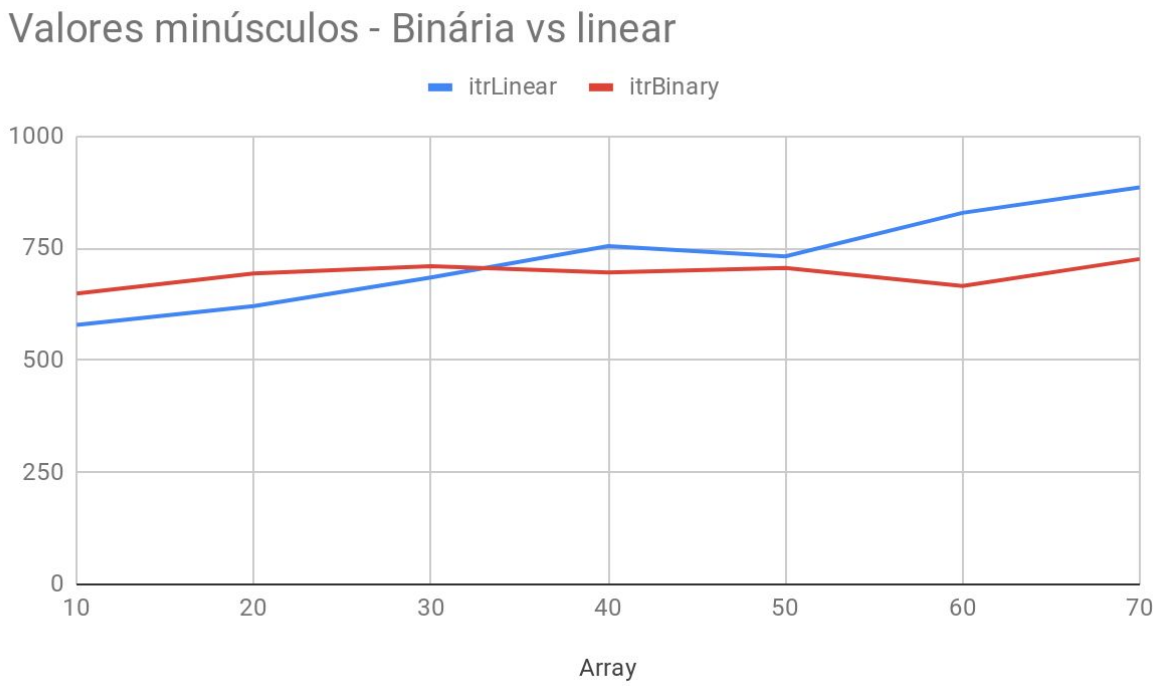
"A partir de que momento algoritmos de classes de complexidade diferentes começam a se diferenciar?"

Olhemos a tabela e gráfico abaixo para melhor visualizarmos nossa resposta:

Array	itrLinear	itrBinary
100	1013	742
300	2032	762
500	3074	789
700	4049	734
900	5096	749

Desde o começo do gráfico, é claro que a busca binária é mais eficiente. Mas e se formos abaixo disso?

Array	itrLinear	itrBinary
10	579	649
20	621	694
30	685	710
40	755	696
50	732	706
60	829	666
70	886	726



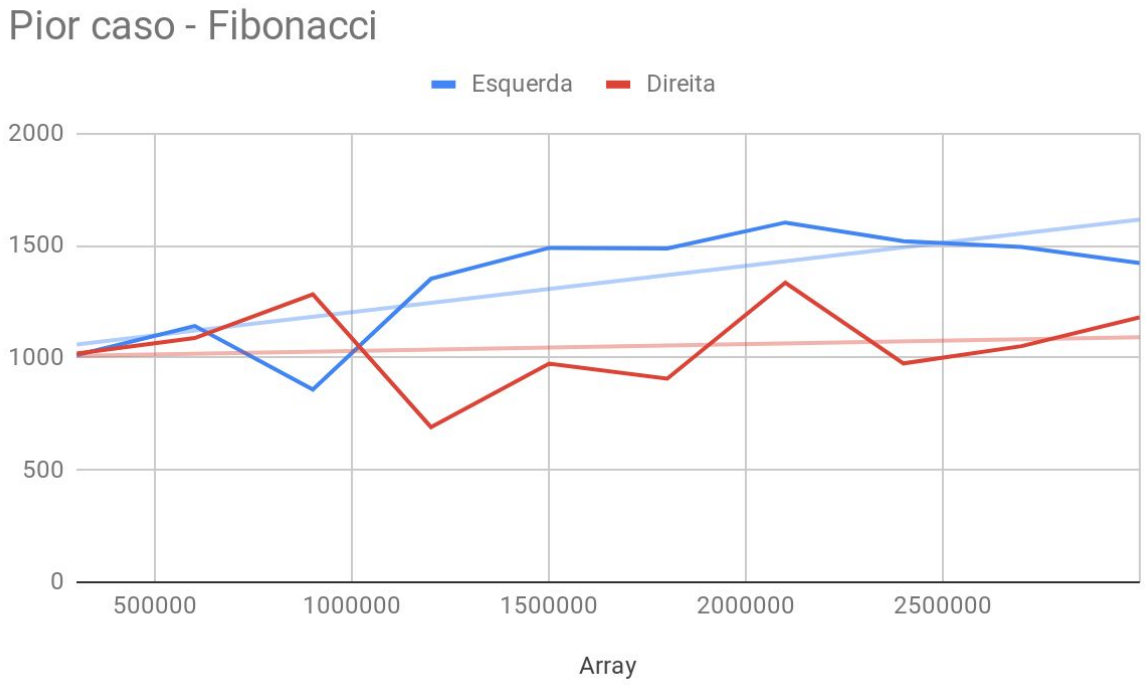
Agora, fica mais claro que a busca linear mostra-se levemente melhor para vetores de tamanhos minúsculos, com 10~30 elementos.

4.5. O pior caso da busca Fibonacci

"Existem categorias diferentes de pior caso para a busca Fibonacci?"

Olhemos o gráfico e a tabela abaixo para melhor vermos nossa resposta:

Array	Valor a esquerda (-1)	Valor a direita (array+1)
300000	1012	1019
600000	1142	1088
900000	859	1283
1200000	1352	691
1500000	1490	974
1800000	1487	907
2100000	1603	1335
2400000	1520	975
2700000	1494	1052
3000000	1422	1180



De fato, se o elemento a ser procurado está a direita/esquerda do vetor pode implicar no pior caso da busca. É claro que no verdadeiro pior caso da busca o elemento está a esquerda do vetor.