UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
ESTRUTURA DE DADOS BÁSICA I

IANCO SOARES OLIVEIRA
LUCAS VINÍCIUS GÓIS NOGUEIRA

ANÁLISE EMPÍRICA DE ALGORITMOS

NATAL/RN
2022

IANCO SOARES OLIVEIRA
LUCAS VINÍCIUS GÓIS NOGUEIRA

ANÁLISE EMPÍRICA DE ALGORITMOS

Relatório técnico apresentado à disciplina Estrutura de Dados Básica I, como requisito parcial para obtenção da nota referente à Unidade I.

NATAL/RN
2022

SUMÁRIO

1. INTRODUÇÃO

Um problema computacional pode ser definido como um problema que possui entradas e retorna uma solução a partir de uma instância deste problema. É possível solucionar este problema a partir de algoritmos, que nada mais são do que uma sequência de passos bem definidos para resolvê-lo. Existem diversas maneiras de escrever um algoritmo que resolva o mesmo problema. Por isso, existe uma forma de avaliar diferentes algoritmos de modo que determine qual o mais eficiente, visto que a execução desses algoritmos demandam recursos do computador (processador, memória RAM, etc). Este método de análise de eficiência é a análise de complexidade.

Este relatório objetiva realizar análises empíricas de complexidade temporal e a comparação de algoritmos que resolvem o problema de ordenação de um arranjo Foram implementados sete algoritmos (São eles: o Selection Sort, Bubble Sort, Insertion Sort, Shell Sort, Merge Sort e Quick Sort) e as análises de tempo foram executadas em um mesmo computador. Todos os valores foram registrados em tabelas e plotados em gráficos, permitindo fácil comparação. Nas seções seguintes, é apresentado o método seguido, abrangendo os materiais e ferramentas utilizadas, bem como o pseudocódigo dos algoritmos implementados; depois, são mostrados os resultados obtidos e, por fim, as discussões geradas a partir deles.

A análise empírica de complexidade temporal tem uma ideia bastante simples: mede-se o tempo de execução de um determinado algoritmo para tamanhos de inputs crescentes e, ao analisar o crescimento desses tempos de resposta pode-se avaliar a função de complexidade temporal do algoritmo.

Nota-se que o Merge Sort e o Radix Sort são algoritmos que têm uma complexidade espacial linear. Ou seja, eles usam uma quantidade de memória RAM proporcional ao tamanho da entrada para resolver o problema de ordenação. Essa é uma desvantagem que pode afetar a eficiência desses algoritmos em relação a outros algoritmos. Nós não iremos calcular a complexidade espacial desses algoritmos neste trabalho, mas esse fato será considerado para a discussão dos resultados.

2. METODOLOGIA

2.1. MATERIAIS UTILIZADOS

2.1.1. COMPUTADOR

A tabela 1 apresenta as principais especificações do computador utilizado nos testes:

Computador	Especificações
Acer Aspire A315-23G	Processador: AMD Ryzen 7 3700U - 2.30 GHz RAM: 8,00 GB DDR4 SSD: 256GB SSD PCIe NVMe 3x2 M2 2.280 GPU: Radeon™ 625 com memória dedicada VRAM de 2GB GDDR5

Tabela 1: Principais especificações técnicas do computador utilizado.

Nesse mesmo computador o sistema operacional GNU/Linux foi utilizado por meio do Windows Subsystem for Linux 2 (WSL-2) como ambiente para os testes. A tabela 2 abaixo traz os detalhes técnicos.

Sistema	Especificações
Windows	Windows Windows 11 Home 64bits
GNU/Linux 5.10.102.1-microsoft-standard-WSL2	Ubuntu 20.04.4 LTS 64bits

Tabela 2: Detalhes técnicos do sistema operacional utilizado.

2.1.2. FERRAMENTAS DE PROGRAMAÇÃO

Os sete algoritmos foram implementados na linguagem C++, padrão ISO/IEC 2017, ou simplesmente C++17.

No ambiente Ubuntu com o WSL-2, os códigos foram compilados pelo CMake[1], na versão 3.16.3, através das linhas:

```
cmake -S source/ -B build -D CMAKE_BUILD_TYPE=Release
cmake –build build
```

E os gráficos foram gerados usando o gnuplot na sua versão 5.2 através do comando

```
gnuplot plot-data.gp
```

[1] https://cmake.org/

2.1.3. ALGORITMOS

2.1.3.1 PROBLEMA DA ORDENAÇÃO DE UM ARRANJO SEQUENCIAL

Neste trabalho, o problema da ordenação de uma arranjo sequencial foi definido como segue:

> *Dada uma sequência $<a_1, \ldots, a_n>$ de n elementos, com $n \in \mathbb{Z}$ e $n > 0$, tal que os elementos da sequência pertencem a um conjunto totalmente ordenável por '$\leqslant$', encontrar uma permutação $<a_{\pi 1}, \ldots, a_{\pi n}>$ da sequência de entrada tal que $a_{\pi 1} \leqslant a_{\pi 2} \leqslant a_{\pi n}$.*

Uma possível solução consiste em, receber o arranjo como entrada e dividi-lo em dois subconjuntos: um ordenado e um não-ordenado, movendo um elemento por vez do sub-conjunto não-ordenado para o ordenado. Essa possibilidade está descrita no Algoritmo 1, também conhecido como Insertion Sort.

Algoritmo 1 (Insertion Sort): Primeira solução para o problema da ordenação de um arranjo sequencial.

```
Input: Um arranjo não ordenado A de tamanho n
Result: O arranjo A em ordem não decrescente
begin
InsertionSort (A)
    for i <- 1 to n:
        value <- A[i];
        hole <- i;
        while hole > 0 and A[hole-1] > value:
            A[i] <- A[hole-1];
            hole < hole-1;
        A[hole] <-  value;
end
```

Outra solução seria receber o arranjo como entrada e também dividi-lo em dois subconjuntos. Porém, diferente do algoritmo 1, nesse caso vamos selecionar os menores elementos e colocá-los nas suas posições (o menor na posição 0, o segundo menor na 1 e assim por diante) até que o arranjo esteja completamente ordenado.

Algoritmo 2 (Selection Sort): Segunda solução para o problema da ordenação de um arranjo sequencial.

Input: Um arranjo A = {a_1, a_2, …, a_n}
Result: Um arranjo não ordenado **A** de tamanho n
begin
SelectionSort(A):

```
for i <- 0 to n:
        index_min <- i;
        for j <- i+1 to n:
                if A[j] < A[index_min]:
                        index_min <- j;
        swap(A[index_min], A[i]);
```

end

Uma terceira solução seria receber um arranjo e percorrê-lo de modo a comparar sempre um elemento com o seu adjacente, trocando-os de posição se estiverem na ordem errada. O algoritmo 3 deve repetir esse procedimento de comparação em pares até todos os elementos estarem ordenados. Esse algoritmo é conhecido como o Bubble Sort.

Algoritmo 3 (Bubble Sort): Terceira solução para o problema da ordenação de um arranjo sequencial.

Input: Um arranjo não ordenado **A** de tamanho n,
Result: O arranjo **A** ordenado,.
begin
BubbleSort (A):

```
    for i <- 1 to n :
        for j <- 0 to n-i-1:
            flag <- false;
            if A[j] > A[j+1]:
                swap(A[j], A[j+1]);
                flag <- true;
        if flag = false: return;
end
```

Uma quarta opção de algoritmo é um pouco mais complexa. Ela utiliza a estratégia de dividir e conquistar, tornando o problema de ordenação em problemas menores. Primeiramente escolhe-se um elemento qualquer do arranjo, em seguida “divide” o arranjo em três partes, a parte a direita do pivô que será composta por elementos menores do que ele mesmo, o pivô em si, que está na está na posição correta e os elementos a direita do pivô que serão maiores que o pivô. O terceiro passo do algoritmo é se chamar, recursivamente, passando dessa vez o arranho à esquerda e à direita do pivô para serem reordenados.

Algoritmo 4 (Quick Sort): Quarta solução para o problema da ordenação de um arranjo sequencial.

Input: Um arranjo não ordenado **A**, um índice **L** do começo do arranjo e **R** do fim do arranjo.

Result: O arranjo **A** ordenado,.

```
begin
QuickSort (A, L, R)
    if ( L >= R): return;
    random <- an number between 0 and n.
    pivot <- A[random];
    swap(A[i], A[random]
    pivot_index <- L;
    for i <- L to R:
        if A[i] <= pivot;
```

```
                    swap(A[i], A[pivot_Index]);
                    pivot_index <- pivot_index+1;
        swap(A[pivot_index], A[R]);
        QuickSort (A, L, pivot_index-1);
        QuickSort (A, pivot_index+1, R);
end
```

Uma outra possível solução para a ordenação de um arranjo seria o algoritmo Merge Sort. Esse algoritmo também usa da mesma estratégia do dividir e conquistar do Quick Sort e também executa várias vezes a sua rotina fundamental (merge) para ordenar um arranjo. A implementação consiste em ordenar dois subconjuntos dentro do arranjo e juntá-los, ou seja, mesclá-los (em inglês, merge) em um novo subconjunto ordenado. Repete-se o processo sucessivamente até o arranjo estar ordenado.

Algoritmo 5 (Merge Sort): Quinta solução para o problema da ordenação de um arranjo sequencial.

```
Input: Um arranjo não ordenado A de tamanho n,
Result: O arranjo A ordenado,.
begin
MergeSort(A):
        if n < 2: return;
        mid <- n/2;
        L <- new array of size mid;
        R <- new array of size mid;
        for i <- 0 to mid-1: L[i] <- A[i];
        for i <- mid  to n: R[i-mid] <- A[i];
        MergeSort(L);
        MergeSort(R);
        nL <- length(L);
        nR <- length(R);
        i <- j <- k <- 0;
```

```
        while i < nL and j < nR:
                if L[i] <= R[j]: A[k] <- L[i];    i++;
                else:   A[k] <- R[j];   j++;
                k++;
        while i < nL:  A[k] <- L[i];    i++;     k++;
        while i < RL: A[k] <- L[i];    j++;     k++;
end
```

Uma variação do insertion sort mais eficiente para a maioria dos casos é o shell sort. No shell sort se escolhe um “salto” e compara os números que estão a esse salto de distância, e então coloca-os ordenados um em relação ao outro. Em seguida diminui esse salto pela metade e refaz as ordenações, quando esse salto ficar menor que um, o arranjo estará ordenado.

Algoritmo 6 (Shell Sort): Sexta solução para o problema da ordenação de um arranjo sequencial.

```
Input: Um arranjo não ordenado A de tamanho n,
Result: O arranjo A ordenado,.
begin
ShellSort(A):
        for gap <- n/2 to gap > 0:
                for i <- gap to i < n:
                        temp <- A[i];
                        j <- 0;
                        for j <- i to j >= gap and A[j-gap] > tamp:
                                A[j] <- A[j-gap];
                                j <- j -gap;
                        A[j] <- temp;
                gap <- gap/2;
end
```

O Radix Sort é diferente dos outros algoritmos de ordenação apresentados até aqui, ele não usa a comparação como base para a resolução

do problema. Invés disso usa-se um contador de algarismos. Imagine que tenha um arranjo A com uma quantidade *n* muito grande de números de 0 a 9. Esse arranjo poderia ser totalmente representado por um outro array de 10 elementos onde cada casa representa um contador do total de vezes que cada digito aparece nesse array. E para ordenar o array A basta inserir, em ordem, o total de vezes que cada digito estava no arranjo original. Usando esse princípio sucessivamente o Radix Sort é capaz de ordenar um arranjo que contenha números com qualquer número de dígitos.

Algoritmo 7 (Radix Sort): Sétima solução para o problema da ordenação de um arranjo sequencial.

```
Input: Um arranjo não ordenado A de tamanho n,
Result: O arranjo A ordenado,.
begin
RadixSort (A):
    max <- max value of A;
    for dPlace <- 1 to max/dPlace > 0:
        output <- new array of size n;
        base <- base of the elements of A;
        count <- new array of size base;
        for j <- 1 to i < n: count[A[j]/dPlace %10]++;
        for j <- 1 to j < base: count[j] = count[j] + count[j -1];
        for j <- n-1 to j >- 0:
            output[count[A[j]/dPlace % 10]-1] <- A[j];
            count[A[j]/dPlace % 10]--;
            j--;
        for j <- 1 to j < n: count[j] = A[j] <- output[j]
end
```

2.2 MÉTODO DE COMPARAÇÕES

Os algoritmos foram comparados segundo os critérios de tempo de execução.

Nos testes, utilizaram-se 25 instâncias do problema, com o tamanho da entrada $n \in [100, 100.000]$ e incrementando em intervalos crescentes. Para cada valor de n, os algoritmos foram executados 5 vezes sobre uma instância e foi calculada a média dessas cinco execuções para diminuir os impactos das flutuações de desempenho da máquina que está rodando os algoritmos.

3. RESULTADOS

Para cada algoritmo, são exibidos os resultados dos testes de tempo em tabelas e gráficos. As tabelas usadas para gerar os gráficos são extensas e não serão necessárias para as análises. Mas, estão disponíveis no apêndice .”A”. Vamos apresentar aqui os gráficos gerados a partir dessas tabelas e na seção 4 discutiremos de forma mais profunda esses resultados.

Os algoritmos foram analisados em seis cenários diferentes, sendo o primeiro cenário onde os arranjos já estavam em ordem não decrescente, o segundo cenário onde os arranjos estavam em ordem não crescente, o terceiro onde 25% dos elementos do arranjo estavam ordenados aleatoriamente, o quarto onde 50% dos elementos estavam fora da posição, o quinto cenário onde 75% dos elementos estavam em ordem aleatória e último cenário onde todos os elementos estavam em ordem aleatória. Segue os gráficos:

Gráfico 1A: Tempo de execução para arranjos em ordem não decrescente

Gráfico 1B: Tempo de execução para arranjos em ordem não decrescente com eixo y em escala decimal.

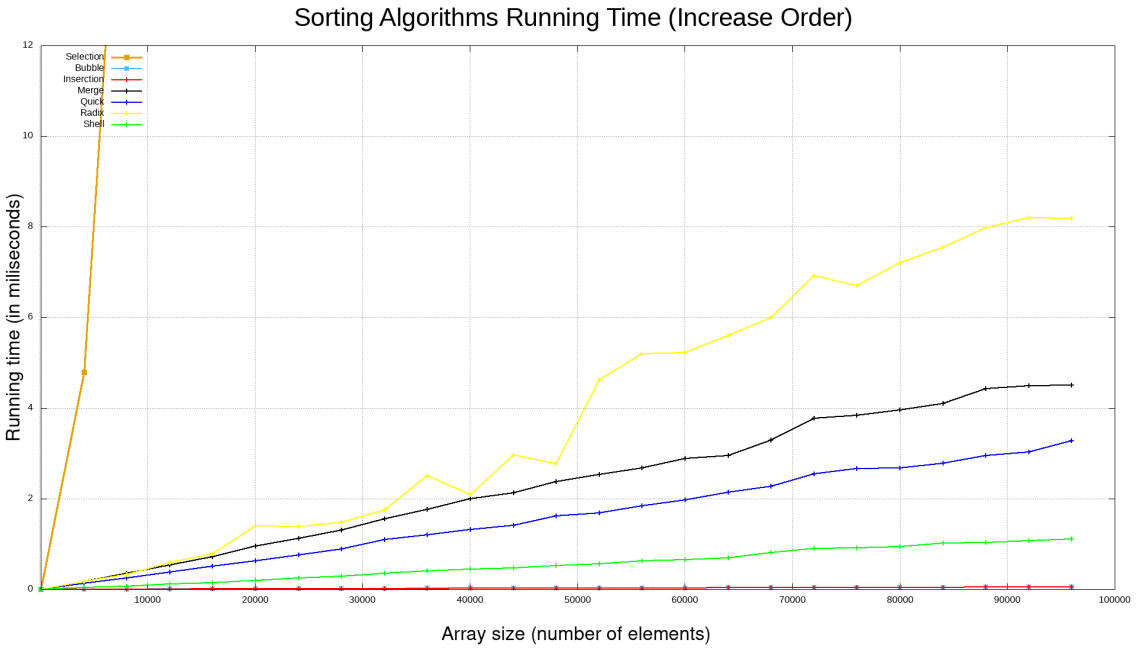


Gráfico 2: Tempo de execução para arranjos em ordem não crescente

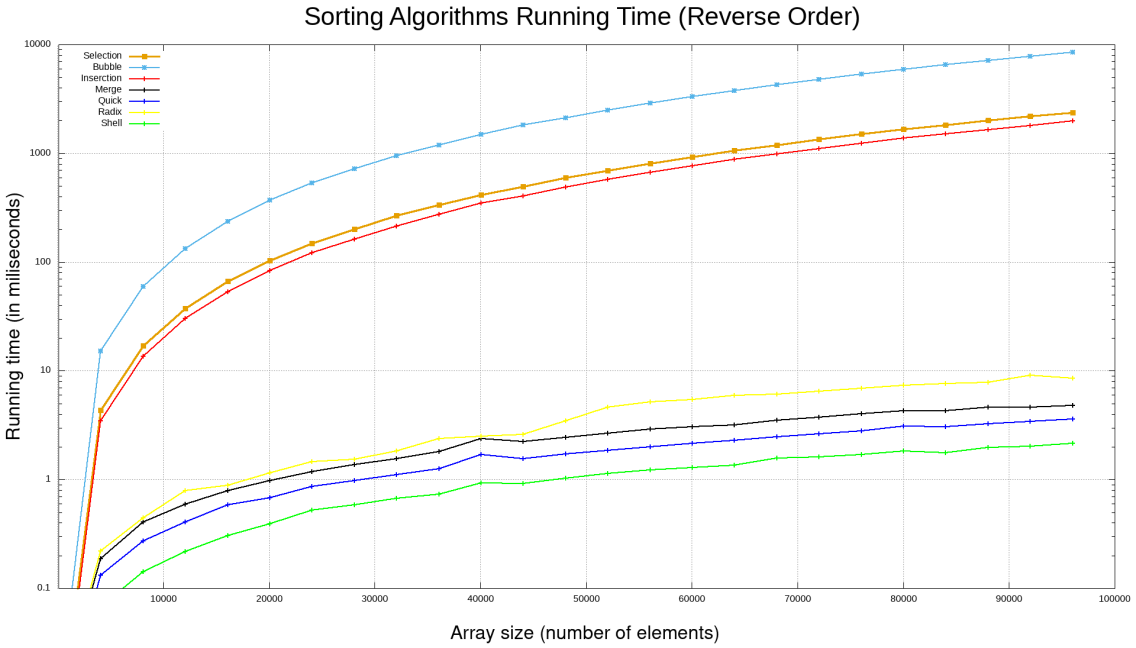


Gráfico 3: Arranjo com 25% dos elementos fora de ordem

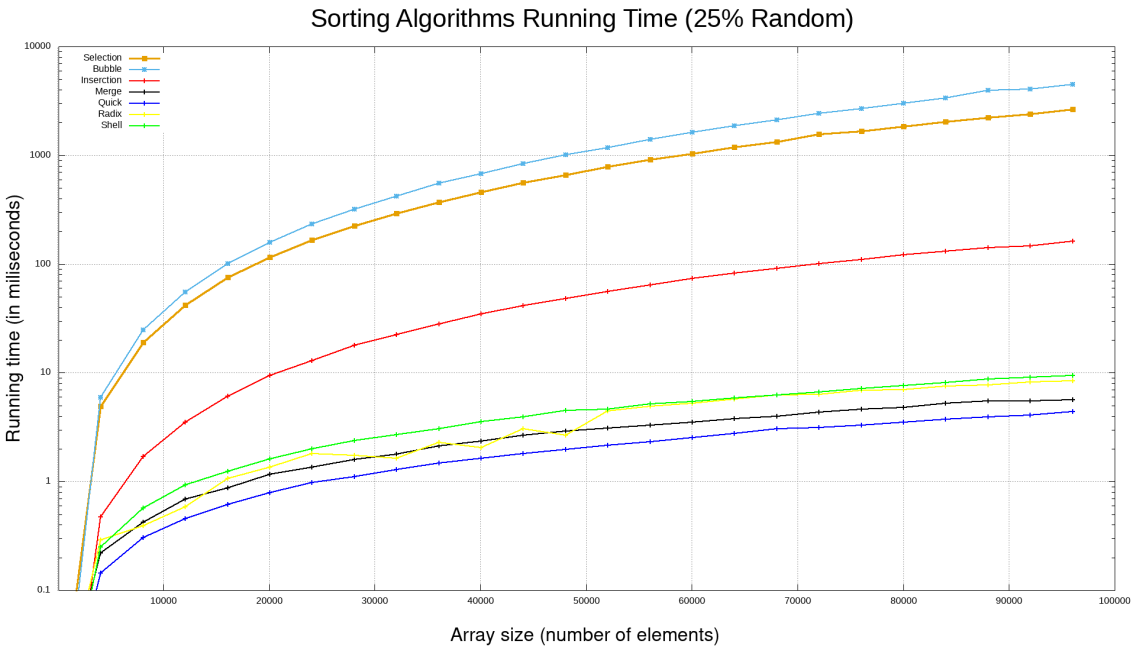


Gráfico 4: Arranjo com 50% dos elementos fora de ordem.

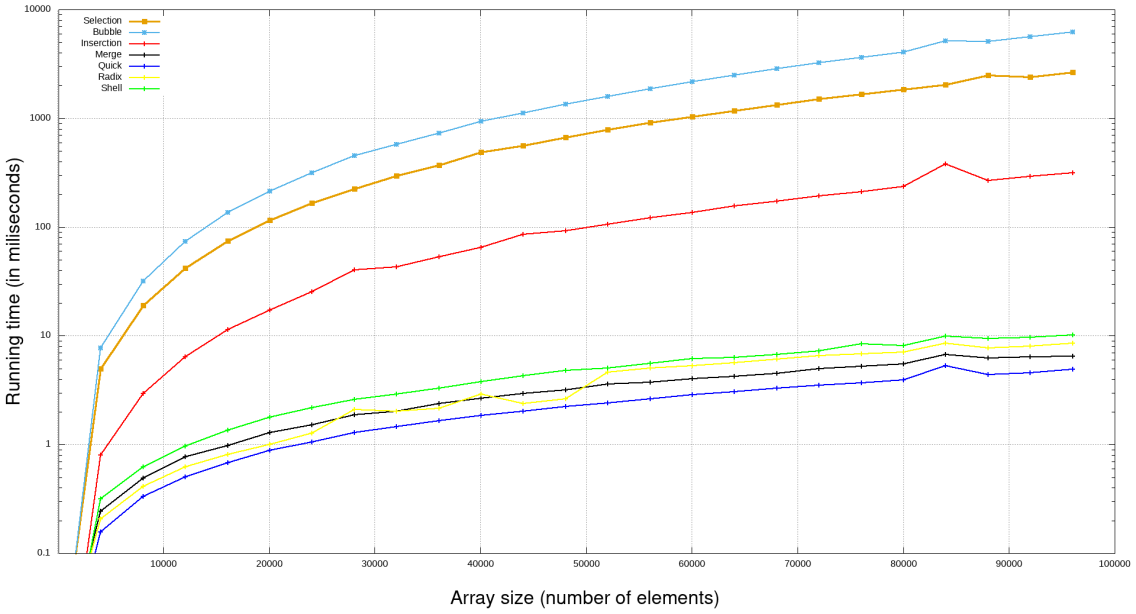


Gráfico 5: Arranjo com 75% dos elementos fora de ordem.

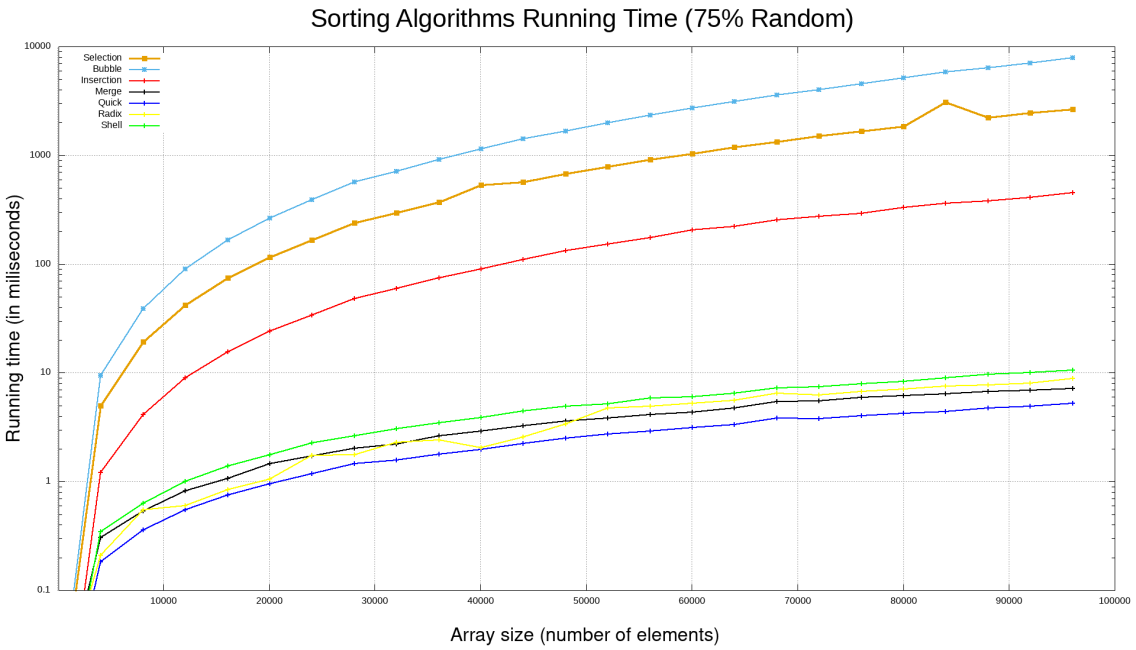
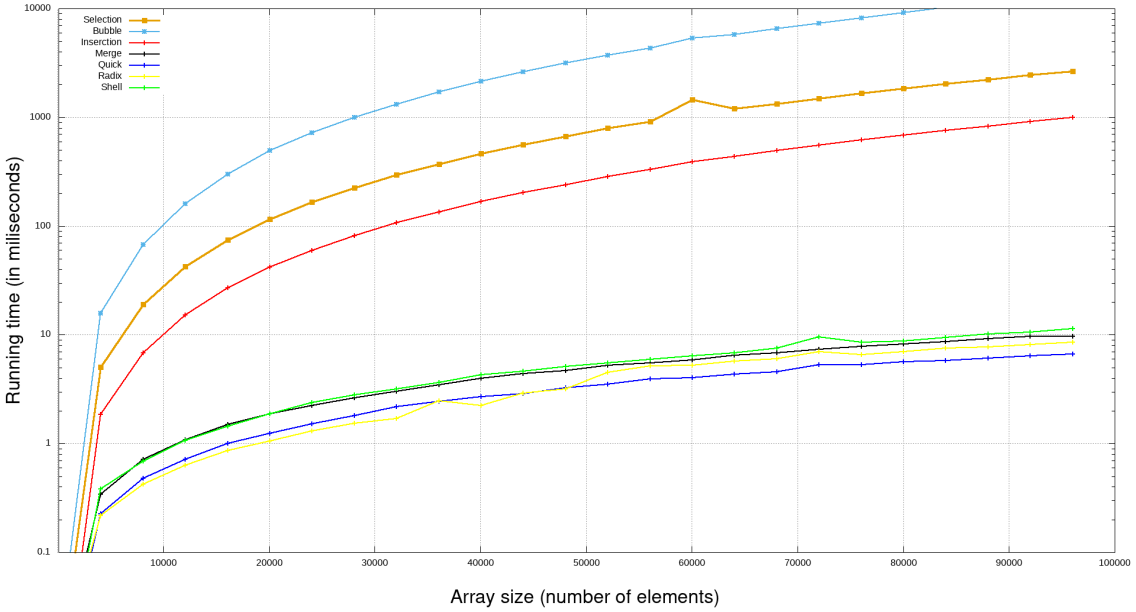


Gráfico 6: Arranjo com elementos totalmente aleatórios



4. DISCUSSÃO:

Os gráficos estão na escala com o eixo Y na escala logarítmica, isso é importante porque, a diferença de magnitude do tempo de execução dos diferentes algoritmos para arranjos maiores pode ser tão grande que torna a análise do gráfico impossível. Usando escala logarítmica, na maioria dos casos, todos dados podem ser desenhados sem uma grande perda de informação.

4.1. ANÁLISE DOS GRÁFICOS

Começando pela análise dos dados para o cenário onde os arranjos já estavam em ordem não decrescente. Uma característica notável é verificar o desempenho do Bubble Sort nesse cenário, como o Bubble só faz operações se encontrar elementos sucessivos que não estejam em ordem não decrescente, ele acaba não fazendo nenhuma operação de troca em um arranjo já ordenado e faz apenas uma verificação linear da ordenação.

Ainda no cenário 1, no Gráfico 1B é possível verificar o bom desempenho do Insertion Sort, isso porque, assim como o Bubble Sort, e Insertion não faz nenhuma operação de inserção caso o arranjo já esteja ordenado, entretanto faz uma quantidade muito maior de verificações que o Bubble. É difícil recomendar o uso de qualquer um dos dois algoritmos (o Bubble ou o Insertion Sort) visto que eles tem desempenhos ruins para a maior parte do caso, mas no caso de uma lista onde relativamente poucos elementos estejam fora de ordem, o Insertion Sort pode ter um bom desempenho.

Já no cenário 2, onde os elementos estão em ordem não crescente, o Bubble e o Insertion Sort têm seus piores desempenhos. Entretanto o Shell Sort, que é uma variação do Insertion Sort, apresenta seu melhor desempenho nesse cenário. Inclusive superando o Quicksort que é amplamente considerado o mais rápido algoritmo de ordenação. O Shell Sort é indicado para esse cenário.

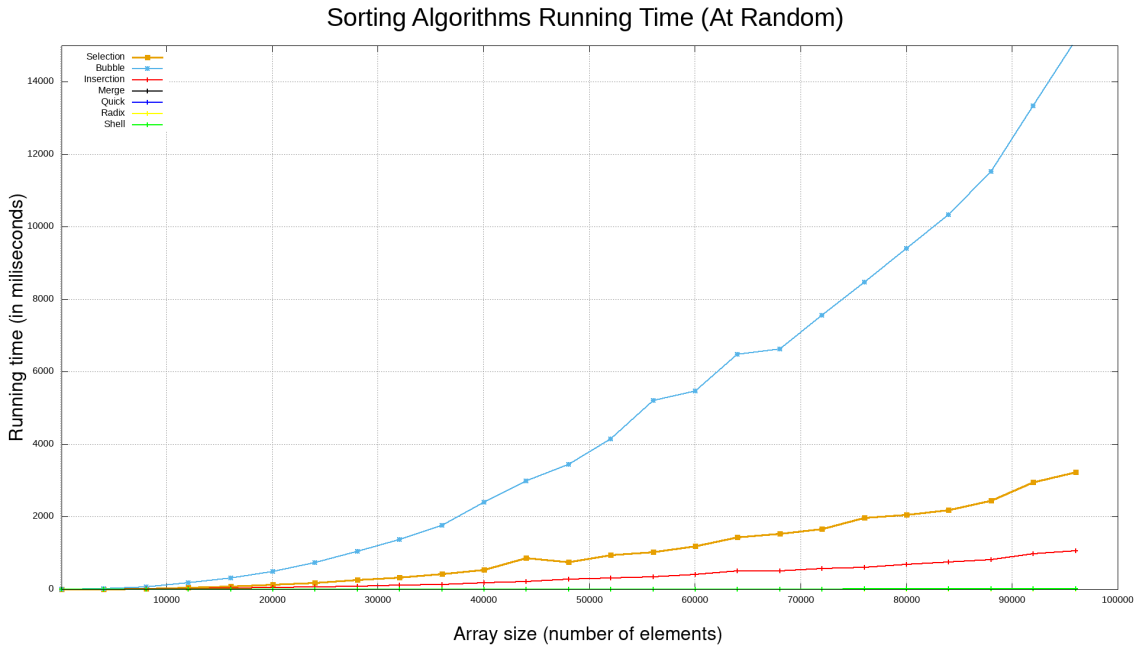
Para qualquer cenário onde existem uma grande quantidade de elementos em ordem aleatória o Quick Sort foi o algoritmo que representou o melhor desempenho. Nota-se que o Quick Sort tem, em seu pior caso, uma complexidade de $O(n^2)$, entretanto esses cenários podem ser facilmente evitáveis adotando um pivô aleatório Dessa maneira ele tem um desempenho

médio muito melhor que o Merge Sort que, em seu pior caso, tem a complexidade de O(nlogn).

Outros dois algoritmos notáveis são o Shell Sort e o Radix Short, o Radix tem um desempenho muito semelhante ao Quick Sort, mas como ele tem uma complexidade linear para uso de memória enquanto o Quick Sort tem uma complexidade constante, o Quick é mais indicado para a maior parte dos casos. Já o Shell Sort chama atenção por ser uma variação do Insertion Sort mas com um desempenho muito melhor para os casos em que os elementos são dispostos aleatoriamente

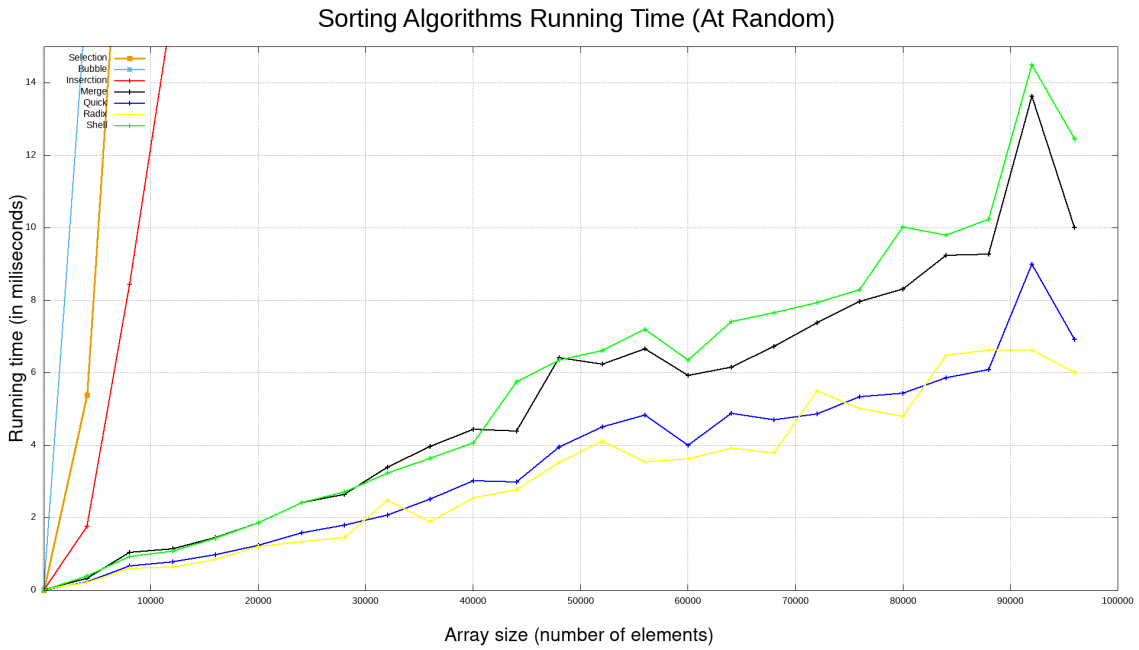
4.1,1, ALGORITMOS O(n²)

Uma última análise pode ser feita usando os apenas os algoritmos de complexidade O(n²). Plotando-os com o eixo Y em escala logarítmica é difícil ver os seus desempenhos relativos. Usando os dados da tabela 6 (ver no apêndice) que foi usada para fazer o gráfico 6, podemos fazer novos gráficos focando apenas no Shell Sort, Bubble Sort, Insertion Sort e Selection Sort.



Dessa maneira dá para notar a diferença de escala de valores para algoritmos de complexidade quadrática para O(nlogn).

4.1,2, ALGORITMOS O(nlogn) E RADIX SORT



Esse é o desempenho dos algoritmos O(nlogn). note que o Shell Sort tem uma complexidade quadrática entretanto ele tem um desempenho, para arranjos de até 100000 elementos, não é suficiente para tirá-lo de escala.

E também é possível notar que o Radix Sort têm um desempenho muito muito próximo ao Quick Sort no cenário onde os elementos são 100% aleatórios, mas como o Radix tem uma complexidade de espaço linear enquanto o Quick tem uma complexidade constante, ainda é aconselhável o uso do Quick Sort nesses casos.

5. APÊNDICE:

Tabela 1: Ordem Não Decrescente							
N	Inserction	Bubble	Selection	Merge	Quick Sort	Radix	Shell
100	0.0057106	0.0002522	0.000302	0.0076964	0.0038342	0.0057402	0.000784
4096	502.846	0.0031382	0.002362	0.1724	0.12078	0.164949	0.0478336
8092	198.523	0.004354	0.004666	0.354293	0.252444	0.338685	0.0699202
12088	449.546	0.0070002	0.0069882	0.62469	0.400175	0.636152	0.121142
16084	838.925	0.0128286	0.0126866	0.74413	0.515417	112.878	0.154145
20080	130.013	0.0104264	0.010722	0.939502	0.635476	100.305	0.201044
24076	184.719	0.0127648	0.0132102	111.853	0.757454	121.859	0.32124
28072	316.518	0.0160642	0.0166448	173.844	115.132	153.417	0.542219
32068	428.028	0.0176608	0.0167046	145.112	102.338	164.937	0.329818
36064	409.928	0.0291568	0.027521	190.095	118.811	181.539	0.392333
40060	529.592	0.0331534	0.0208346	23.116	137.316	28.805	0.469351

44056	659.116	0.0414212	0.0399452	259.916	183.661	325.519	0.582273
48052	742.112	0.0370372	0.0370212	243.841	15.948	245.925	0.69785
52048	922.104	0.0429694	0.0557874	282.655	168.793	290.346	0.760694
56044	1084.69	0.029427	0.029117	262.589	181.789	285.826	0.606411
60040	1263.8	0.0394692	0.0368052	31.802	205.429	331.224	0.650385
64036	1439.83	0.0367052	0.0443396	289.682	207.348	388.412	0.758365
68032	1666.83	0.0473634	0.0445914	395.129	292.043	559.742	0.975842
72028	1672.05	0.0383592	0.0373532	348.312	254.393	444.803	100.401
76024	1826.75	0.0399254	0.0396174	396.832	256.272	434.931	0.877568
80020	2046.47	0.0424276	0.0415312	394.042	275.519	549.089	118.949
84016	2354.02	0.0447216	0.0436498	417.025	276.794	480.441	130.457
88012	2439.22	0.0678502	0.067454	451.128	288.243	575.774	102.768
92008	2678.52	0.0722886	0.0721906	46.818	312.364	641.052	12.337
96004	3475.07	0.056186	0.0786824	820.501	42.361	616.616	122.228

Tabela 2: Ordem Não Crescente							
N	Inserction	Bubble	Selection	Merge	Quick	Radix	Shell Sort
100	0.00464	0.0086686	0.0027202	0.0063162	0.0034024	0.0117264	0.001386
4096	553.994	156.173	335.451	0.211285	0.139765	0.226939	0.0776102
8092	208.517	603.504	133.961	0.387786	0.271967	0.448653	0.145867
12088	428.709	137.68	303.382	0.607065	0.427688	0.643493	0.224845
16084	720.144	246.708	591.549	0.755385	0.565952	0.87377	0.313696
20080	110.75	387.35	895.171	0.969375	0.698142	119.888	0.417073
24076	179.368	537.929	127.564	11.954	0.853467	183.424	0.485431
28072	273.558	800.291	164.619	139.359	0.970359	15.338	0.583325
32068	287.861	970.81	217.789	159.716	110.527	172.335	0.748383
36064	362.63	1232.02	272.667	206.512	130.327	217.393	0.720969
40060	480.386	1596.41	401.265	214.533	148.522	243.695	0.814962
44056	592.882	1942.03	468.495	316.057	187.109	306.613	126.403
48052	711.775	2368.69	553.79	241.151	169.606	257.407	0.997312
52048	838.963	2713.7	646.474	369.894	246.847	357.753	147.138
56044	1002.8	3318.45	749.17	349.767	328.707	454.651	203.124
60040	1045.46	3704.1	839.965	300.846	213.305	362.034	12.763
64036	1220.07	4154.66	986.357	326.386	227.127	344.841	134.573
68032	1417.86	4706.13	976.731	369.247	260.044	42.355	15.115
72028	1481.11	4842.97	1204.11	393.372	28.041	486.557	1.838
76024	1637.09	5393.93	1225.65	415.499	27.575	433.139	163.014
80020	1825.65	6070.62	1373.45	425.315	293.257	567.964	195.021
84016	2014.09	6506.97	1568.71	431.244	299.884	600.328	18.599
88012	2200.71	7245.43	1675.66	482.369	362.893	495.108	223.309
92008	2606.38	8782.31	2015.1	535.091	357.392	513.069	258.675
96004	3109.05	10215.2	2361.84	588.875	376.721	688.131	331.086

	Tabela 3: 25% Aleatório						
N	Inserction	Bubble	Selection	Merg	Quick	Radix	Shell
100	0.006484	0.0035002	0.000802	0.0057564	0.0037282	0.00682	0.002228
4096	588.861	629.148	0.426149	0.222423	0.175164	0.205347	0.256224
8092	357.651	272.699	189.438	0.475111	0.335867	0.434408	0.6159
12088	564.888	599.871	336.268	0.710759	0.477531	0.590495	0.894477
16084	832.339	108.12	613.192	0.925726	0.669087	0.787361	131.014
20080	131.648	171.444	915.315	115.138	0.785423	0.982161	171.001
24076	184.508	250.173	136.676	135.906	0.951244	160.879	208.969
28072	272.433	388.789	183.439	165.425	112.415	145.976	245.612
32068	318.023	469.837	22.516	17.795	12.925	206.721	274.705
36064	434.629	580.073	282.202	218.648	158.026	187.881	317.985
40060	529.877	784.294	416.217	286.957	199.056	284.397	433.327
44056	692.248	1016.15	40.356	279.607	1.806	231.943	416.005
48052	841.091	1153.59	586.294	342.157	240.525	306.591	567.182
52048	956.955	1361.79	544.802	358.668	230.139	277.283	474.212
56044	1144.44	1650.34	677.066	377.324	233.587	381.657	516.299
60040	1203.9	1815.4	821.751	510.944	376.643	342.956	769.966
64036	1455.4	2187	964.134	443.051	328.603	410.713	766.881
68032	1499.83	2292.64	940.155	410.723	307.293	395.175	806.006
72028	1650.83	2487.3	101.138	445.822	340.887	371.274	712.184
76024	1896.24	2825.91	107.786	547.196	349.647	397.227	720.863
80020	2020.04	3182.11	119.348	540.123	348.906	645.524	766.019
84016	2311.52	3529.21	127.392	555.215	392.179	610.181	816.426
88012	2512.41	3884.37	136.903	536.819	384.605	525.266	888.067
92008	3114	6986.32	186.04	687.647	483.208	548.351	105.375
96004	3289.84	5205.22	172.572	700.548	431.385	706.255	107.659

Tabela 4: 50% Aleatório							
N	Inserction	Bubble	Selection	Merge	Quick	Radix	Shell
100	0.005878	0.0040482	0.00111	0.006158	0.0039224	0.0088944	0.003048
4096	50.232	771.982	0.911549	0.259955	0.158607	0.210507	0.30857
8092	246.576	410.957	32.394	0.536683	0.348162	0.435256	0.633952
12088	502.201	791.753	713.186	0.785177	0.527337	0.617392	100.241
16084	867.705	146.382	113.041	0.973247	0.686164	0.789137	132.377
20080	131.73	225.397	173.125	128.884	0.86656	102.896	171.191
24076	196.905	329.103	24.28	152.254	105.401	130.837	216.277
28072	259.05	459.924	332.117	184.017	125.214	139.882	25.316
32068	327.647	585.419	437.284	206.741	144.908	169.164	306.897
36064	437.624	866.907	542.688	242.675	165.959	181.572	329.778
40060	530.134	967.289	64.207	279.378	186.938	224.235	393.676
44056	684.915	1316.32	867.854	356.278	24.769	358.492	549.337

48052	791.205	1522.27	108.221	413.795	269.194	298.993	574.926
52048	925.836	1827.92	133.345	424.237	292.521	350.963	608.409
56044	1075.69	2138.53	150.355	368.382	281.669	317.067	555.017
60040	1256.22	2479.12	165.63	475.955	360.041	38.419	776.156
64036	1473.06	2799.4	151.044	433.156	300.711	43.491	719.949
68032	1467.03	3027.75	171.44	451.713	340.344	396.252	694.363
72028	1643.51	3351.25	207.443	726.828	382.327	456.152	741.477
76024	1824.41	3753.38	211.039	540.612	368.462	435.919	863.933
80020	1998.73	4215.98	232.188	552.361	395.988	509.514	800.544
84016	2265.63	4874.67	248.853	637.495	450.938	493.418	859.849
88012	2437.13	5216.69	267.6	60.535	466.439	482.099	972.168
92008	3097.18	6642.04	308.544	667.887	525.571	745.058	113.545
96004	3357.32	7551.26	321.711	753.298	537.743	761.179	128.671

Tabela 5: 75% Aleatório							
N	Inserction Sort	Bubble Sort	Selection Sort	Merge Sort	Quick Sort	Radix Sort	Shell Sort
100	0.007614	0.0085104	0.001482	0.0075962	0.0043262	0.0114704	0.003402
4096	509.276	942.915	120.883	0.267972	0.177064	0.204603	0.346753
8092	20.485	41.046	431.166	0.541303	0.384264	0.424913	0.817697
12088	43.006	101.76	932.964	0.851477	0.647997	0.824482	110.553
16084	811.445	176.449	155.875	106.638	0.74328	0.833335	141.115
20080	119.591	275.586	248.663	144.373	0.95661	103.576	178.821
24076	188.958	521.469	547.897	242.771	140.806	152.005	267.398
28072	246.16	589.328	514.685	247.388	177.493	15.721	31.868
32068	328.871	771.192	596.127	234.435	15.917	162.511	308.902
36064	405.043	959.075	755.397	261.027	177.583	18.399	371.503
40060	526.871	1266.92	111.124	369.873	276.457	300.647	472.906
44056	674.527	1616.53	117.581	327.558	230.755	224.503	434.119
48052	825.079	1980.6	150.472	349.237	242.739	259.978	487.278
52048	948.376	2278.76	172.284	4.707	321.688	417.688	739.086
56044	1021.01	2685.31	200.945	432.244	30.086	420.043	599.504
60040	1234.9	3145.73	221.657	555.595	39.638	474.356	73.913
64036	1464.38	3601.72	259.3	573.115	424.891	404.784	793.164
68032	1551.94	3726.98	242.701	510.751	369.171	45.032	69.591
72028	1643.47	4193.52	272.4	54.964	377.155	407.805	871.223
76024	1824.91	4687.85	309.16	576.088	402.645	475.535	790.069
80020	2030.29	5273.13	337.809	624.942	432.564	456.205	865.883
84016	2267.69	5870.89	358.324	633.019	457.185	615.661	898.497
88012	2464.05	6571.07	397.284	675.023	472.571	642.342	960.921
92008	3093.13	8558.76	506.224	99.885	662.011	782.472	145.482
96004	2984.59	9301.42	467.263	730.965	518.839	678.806	107.124

Tabela 6: Totalmente Aleatório							
N	Inserction	Bubble	Selection	Merge	Quick	Radix Sort	Shell
100	0.0069982	0.0126028	0.001876	0.01001	0.0070582	0.0091362	0.004138
4096	538.799	165.706	176.316	0.334012	0.226161	0.218305	0.392494
8092	228.388	717.325	844.507	104.158	0.67121	0.611332	0.9254
12088	481.275	177.766	160.437	114.469	0.775749	0.637328	107.681
16084	792.271	309.489	275.926	145.294	0.981363	0.851696	143.143
20080	126.349	495.335	419.567	18.596	12.456	121.302	18.655
24076	178.902	734.445	596.294	242.138	158.157	133.293	241.317
28072	254.251	1040.65	822.535	263.813	179.492	145.402	270.414
32068	329.73	1364.86	107.143	339.702	206.779	248.534	322.834
36064	416.768	1757.71	138.705	39.738	252.003	189.457	364.382
40060	538.959	2391.54	175.776	443.331	301.759	254.544	407.185
44056	870.918	2983.76	215.538	438.408	297.971	277.422	574.154
48052	751.516	3445.02	269.981	640.923	395.397	352.987	634.715
52048	952.353	4152.25	302.226	623.695	450.374	412.028	661.138
56044	1029.45	5201	348.458	666.632	482.879	354.326	719.574
60040	1196.01	5474.75	405.139	592.648	399.644	362.494	634.387
64036	1433.19	6475.19	511.755	615.934	487.869	391.157	741.423
68032	1537.31	6632.04	508.826	67.261	470.657	378.972	765.436
72028	1660.79	7556.14	568.618	737.825	486.639	549.404	793.457
76024	1983.03	8472.42	607.752	797.083	533.232	50.072	829.083
80020	2061.56	9398.92	690.648	831.364	543.122	479.677	10.027
84016	2194.46	10326.7	756.423	923.756	585.721	648.188	979.692
88012	2446.41	11528.7	823.426	926.578	609.349	662.393	102.395
92008	2955.87	13341	978.956	136.246	900.149	663.397	144.881
96004	3236	15123.4	1063.3	10.013	691.323	600.777	124.497