João Guilherme Lopes Alves da Costa
Paulo Vitor Fernandes Andrade

Análise empírica de algoritmos de ordenação

Brasil
2021

João Guilherme Lopes Alves da Costa
Paulo Vitor Fernandes Andrade

Análise empírica de algoritmos de ordenação

Relatório técnico apresentado à disciplina de Estrutura de Dados Básicas I, como requisito parcial para obtenção de nota referente à unidade I. Orientado pelo docente Selan Rodrigues dos Santos.

Universidade Federal do Rio Grande do Norte - UFRN

Instituto Metrópole Digital - IMD

Bacharelado em Tecnologia da Informação

Brasil

2021

Sumário

Lista de Tabelas

Lista de Figuras

1 Introdução

Este trabalho visou conhecer a implementação dos mais famosos algoritmos de ordenação, além de fazer a análise de cada algoritmo quanto a sua eficiência em relação a complexidade temporal. A necessidade de se ordenar conjuntos de elementos facilita muito a vida do programador, até porque é algo que é possível fazer em vários contextos. Quase tudo na computação envolve matemática e consequentemente números, um bom exemplo são os caracteres (char) que em mais "baixo nível"são representados por números, permitindo uma ordenação até mesmo das letras.

A ordenação é útil, por exemplo, para fazermos uma busca em um array. Os algoritmos de busca mais famosos são o *binary search* e o *linear search*, porém, a busca binária só é possível ao termos um vetor ordenado. E como a busca binária é muito mais eficiente do que a busca linear, se conseguirmos ordenar esse vetor, teremos um programa mais eficiente. Dessa forma, foi estudado sete algoritmos de ordenação: *insertion sort*, *selection sort*, *bubble sort*, *shell sort*, *quick sort*, *merge sort* e *radix sort*.

Inicialmente, foi feito um estudo da lógica dos diferentes algoritmos e logo depois a tentativa de implementá-los. Após isso, foi avaliado qual dos algoritmos obtinha um melhor tempo de execução (*runtime*), de acordo com o aumento no número de elementos e com o cenário. É importante salientar que apesar de uns serem melhor em execução do que outros, tem outros fatores importantes, tais como complexidade espacial (uso de memória), que irão definir qual o algoritmo mais adequado a se utilizar em um determinado problema.

Para a realização deste trabalho foi utilizado a linguagem de programação C++. Com esta linguagem foi feito a implementação dos algoritmos de ordenação, e junto da biblioteca Chronos pode-se calcular o tempo de execução de cada algoritmo. Tendo o tempo de cada algoritmo, pode-se fazer a análise empírica, que é a comparação de cada um observando quem obteve um melhor desempenho. Para isso, foi utilizado a construção de gráficos através da ferramente Gnuplot.

2 Metodologia

2.1 Materiais utilizados

2.1.1 Computador

Para a realização da análise empírica (geração dos tempos de execução de cada algoritmo) foi utilizado um computador com as seguintes especificações:

- Processador: AMD Ryzen 5 3600 6-Core 3.60 GHz
- Memória Ram: Corsair DDR4 3000mhz 16gb
- Placa de Vídeo: Gigabyte AMD Radeon RX 5600 XT Gaming OC, 6GB, GDDR6
- Placa-Mãe: ASRock B450M Steel Legend

A geração dos tempos foi realizado através do sistema operacional Windows com utilização do Subsistema Windows para Linux para a compilação e execução:

- Sistema operacional: Windows 10 Pro 64bits
- Windows Subsystem for Linux 1 (WSL 1)

No entanto, a geração dos gráficos foi feita no Gnuplot meio do sistema operacional GNU/Linux (distribuição Kubuntu):

- Kubuntu 20.04 (Ubuntu 9.3.0-17ubuntu1)

2.1.2 Ferramentas de programação

Para a geração dos códigos a linguagem de programação utilizada foi o C++, mais especificamente o C++11, e o editor de texto Microsoft Visual Studio Code.

No Windows os códigos foram compilados pelo G++ 9.3.0 (2020), através do WSL1. A execução também foi feita pelo WSL1.

```
//Compilacao
g++ -Wall std=c+11 <arquivos.cpp> -o <executavel>
```

Para realizar a medição dos tempos, foi utilizado a biblioteca *chrono* do STL.

2.1.3 Algoritmos

1. Insertion sort

   Para implementar o merge basta seguir o passo a passo:

   - Nós iremos comparando um elemento com o seu próximo da esquerda;
   - Caso seja menor que o elemento a esquerda, trocam de lugar;
   - Irá comparar com todos os elementos a sua esquerda enquanto não tiver mais elementos a esquerda ou quando tiver um maior que ele;
   - Ao acabar isso, passará para o próximo elemento a direita e fará o mesmo passo a passo anterior;
   - Acabaremos o algoritmo quando fizermos o passo a passo para o último elemento.

2. Selection sort

   A ideia do selection é baseada em encontrar o menor número entre os ainda não ordenados e colocar ele na posição correta. Para isso, utilizamos um for que vai do primeiro elemento ao ultimo, dentro dele iniciamos colocando o primeiro elemento da parte não ordenada como menor valor e vamos comparando com todos os valores posteriores a ele, verificando se ele continua sendo o menor valor. Ao final, fazemos uma troca dentro do array para colocar o menor valor na posição certa.

3. Bubble sort

   Para implementar o bubble em sua versão otimizada, basta seguir o passo a passo:

   - Compara cada elemento do vetor com seu adjacente;
   - Caso o elemento de menor índice for maior que o elemento de maior índice, então há troca de posição entre eles;
   - Senão, não faz nada;
   - Enquanto percorrer o vetor e houver troca de elementos o algoritmo fica rodando;
   - Se percorrer o vetor inteiro e não houver nenhuma troca, encerramos o algoritmo.

4. Shell sort

   A ideia do shell basicamente é de um insertion "otimizado"de forma que permita a troca de elementos distantes. Essa distância irá diminuindo até que o algoritmo fará basicamente um insertion sort.

5. Merge sort

   Para implementar o merge basta seguir o passo a passo:

   - Será dividido em vetor em dois novos vetores. Pega o elemento do meio, todos os elementos antes do elemento do meio irão compor o vetor Left e os elementos a direita do elemento do meio (junto com o elemento do meio) irão compor o vetor Right;
   - Chamaremos recursivamente o merge com o vetor Left e chamaremos recursivamente o merge com o vetor Right;
   - Depois dessa chamada cada vetor estará ordenado. Chamaremos uma função auxiliar para ir comparando cada elemento dos dois vetores para decidir quem entrará no vetor completo, indíce por índice.

6. Quick sort

   Para implementar o quick basta seguir o passo a passo:

   - Decidiremos um valor de pivot do vetor (no nosso caso foi o do meio);
   - Colocaremos todos os valores menores que o pivot a esquerda dele;
   - Após isso chamaremos recursivamente a função do quick com o vetor a esquerda do pivot e depois recursivamente com o vetor a direita do pivot junto com o pivot;

7. Radix sort

   Para implementar o radix basta seguir o passo a passo:

   - Precisaremos de um vetor de vector de 10 posições (nesse caso será de 10 posições pela nosso sistema numérico ser decimal, ou seja, 10 posições) para representar os *buckets*;
   - Contaremos o número máximo de dígitos;
   - Para cada loop analisaremos do dígito menos significativo para o mais significativo. Colocaremos o elemento no bucket correspondente ao valor do dígito analisado naquele loop.
   - Ao terminar um loop pegaremos os valores na ordem colocada nos buckets;
   - Repetiremos o mesmo número de vezes do número máximo de dígitos.

2.2 Métodos de Comparação

Os algoritmos foram comparados segundo o critério de tempo de execução (runtime). Foi simulado seis cenários diferentes para testar cada algoritmo:

1. arranjos com elementos em ordem não decrescente,
2. arranjos com elemento em ordem não crescente,
3. arranjos com elementos 100% aleatórios,
4. arranjos com 75% de seus elementos em sua posição definitiva,
5. arranjos com 50% de seus elementos em sua posição definitiva, e
6. arranjos com 25% de seus elementos em sua posição definitiva.

Para testar um algoritmo x em um cenário y foi utilizado 25 tamanhos diferentes de amostras, igualmente espaçados entre a menor amostra e a maior amostra para cada algoritmo em cada cenário. A faixa de tamanhos de amostras a serem gerados é $[400, 10^4]$, aumentando em 400 o tamanho para cada nova amostra. Para cada tamanho de amostra, foi testado 5 vezes cada algoritmo e feito a média aritmética. Os números preenchidos no vetor foram iguais aos índices do vetor, por exemplo, um vetor de 400 elementos foi preenchido com os seguintes valores: $0, 1, 2, 3, \ldots, 398, 399$.

3 Resultados

Os resultados serão apresentados de acordo com o cenário. Em cada seção de cenário serão apresentados os gráficos de cada algoritmo separadamente e por último um gráfico com todos os algoritmos.

3.1 Cenário 1

A tabela seguinte apresenta o desempenho dos 7 algoritmos no cenário em que os elementos do vetor estão em **ordem não decrescente**.

Tamanho da entrada	Tempo (ms)						
	Insertion	Selection	Bubble	Shell	Merge	Quick	Radix
400	0.00734	0.56404	0.00722	0.04352	0.18044	0.62154	0.05512
800	0.02138	2.46888	0.01092	0.07988	0.37178	2.3771	0.11748
1200	0.02086	5.0613	0.0196	0.19178	0.66438	5.4491	0.20492
1600	0.02772	9.19402	0.02158	0.18158	0.77486	9.37276	0.29952
2000	0.03642	13.9599	0.02696	0.23036	0.98726	14.6188	0.35028
2400	0.06618	20.3997	0.03736	0.2851	1.19742	20.9606	0.42474
2800	0.0504	27.3018	0.04288	0.33682	1.40402	28.7362	0.5132
3200	0.05606	36.2434	0.04392	0.38616	1.64046	37.1895	0.60066
3600	0.062	45.3333	0.04924	0.4507	1.85296	46.9865	0.63684
4000	0.07228	55.6516	0.0531	0.50376	2.04832	58.1749	0.68402
4400	0.08832	67.5724	0.06488	0.5745	2.22384	70.0355	0.78058
4800	0.09416	80.6256	0.08142	0.632	2.42454	83.1757	0.84938
5200	0.09192	94.0544	0.07756	0.69324	2.6493	98.4022	0.91512
5600	0.10804	109.484	0.07426	0.76102	2.84498	113.413	0.99618
6000	0.1193	125.721	0.10072	0.81718	3.12164	130.684	1.07498
6400	0.13206	141.645	0.08504	0.85412	3.32948	147.605	1.11834
6800	0.15142	159.959	0.08968	0.9184	3.52194	167.249	1.21466
7200	0.18812	179.099	0.11664	0.98162	3.7745	186.766	1.28114
7600	0.1702	200.629	0.10246	1.0394	3.88628	208.149	1.3653
8000	0.18848	222.009	0.1054	1.1006	4.17146	229.489	1.41614
8400	0.21952	245.449	0.11372	1.16184	4.35554	253.912	1.5418
8800	0.2423	268.56	0.12706	1.22282	4.6496	278.764	1.55114
9200	0.24414	294.801	0.12496	1.26982	4.76906	306.268	1.64982
9600	0.25794	319.778	0.13744	1.3349	5.01084	331.986	1.71706
10000	0.17674	347.349	0.13216	1.3891	5.1913	360.593	1.80962

Tabela 1: Tempo dos algoritmos em função do tamanho da entrada quando o vetor está em ordem não decrescente

Figura 1: Análise de tempo para o insertion sort, ordem não decrescente

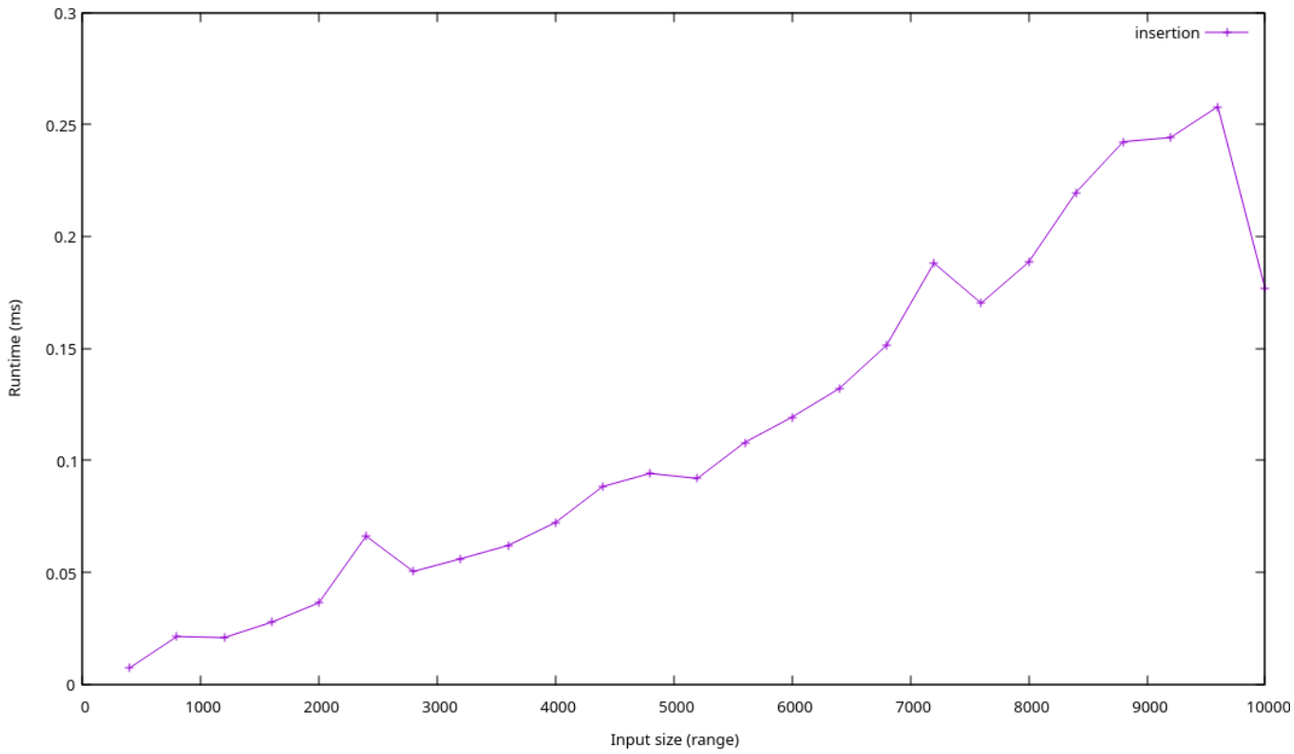


Figura 2: Análise de tempo para o selection sort, ordem não decrescente

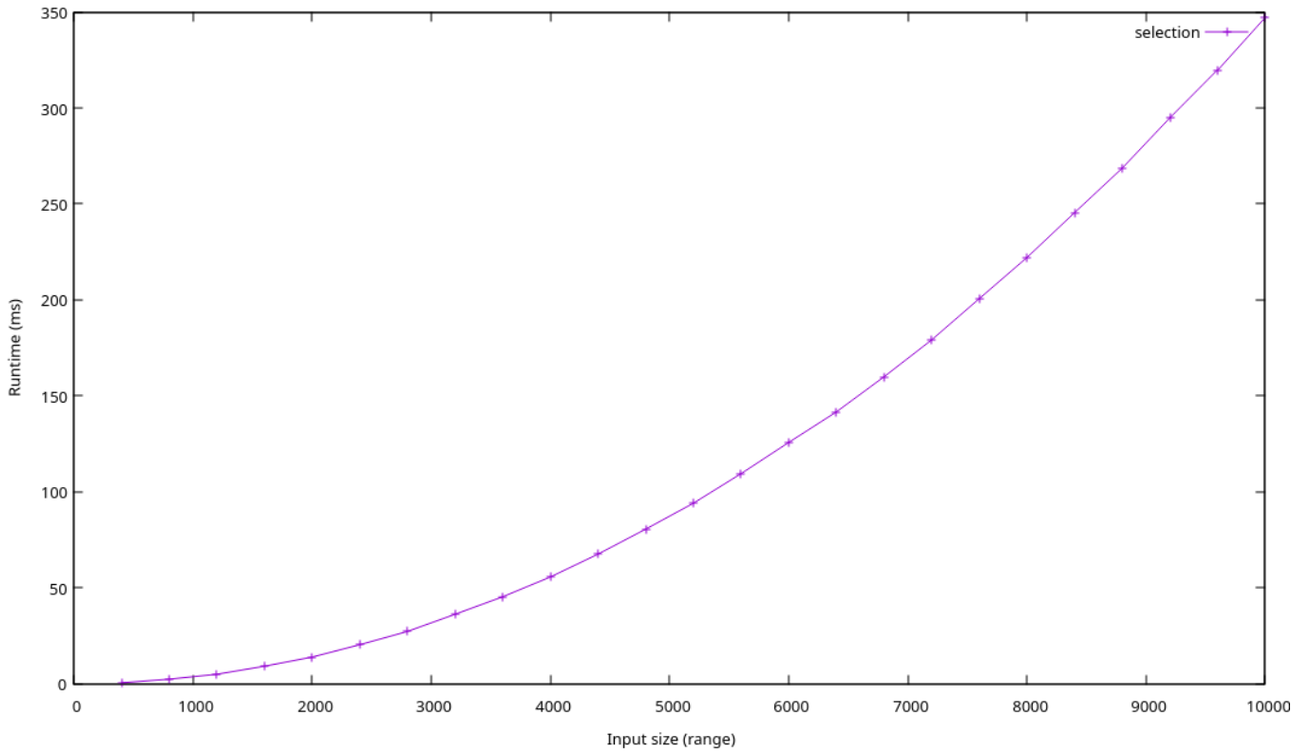


Figura 3: Análise de tempo para o bubble sort, ordem não decrescente

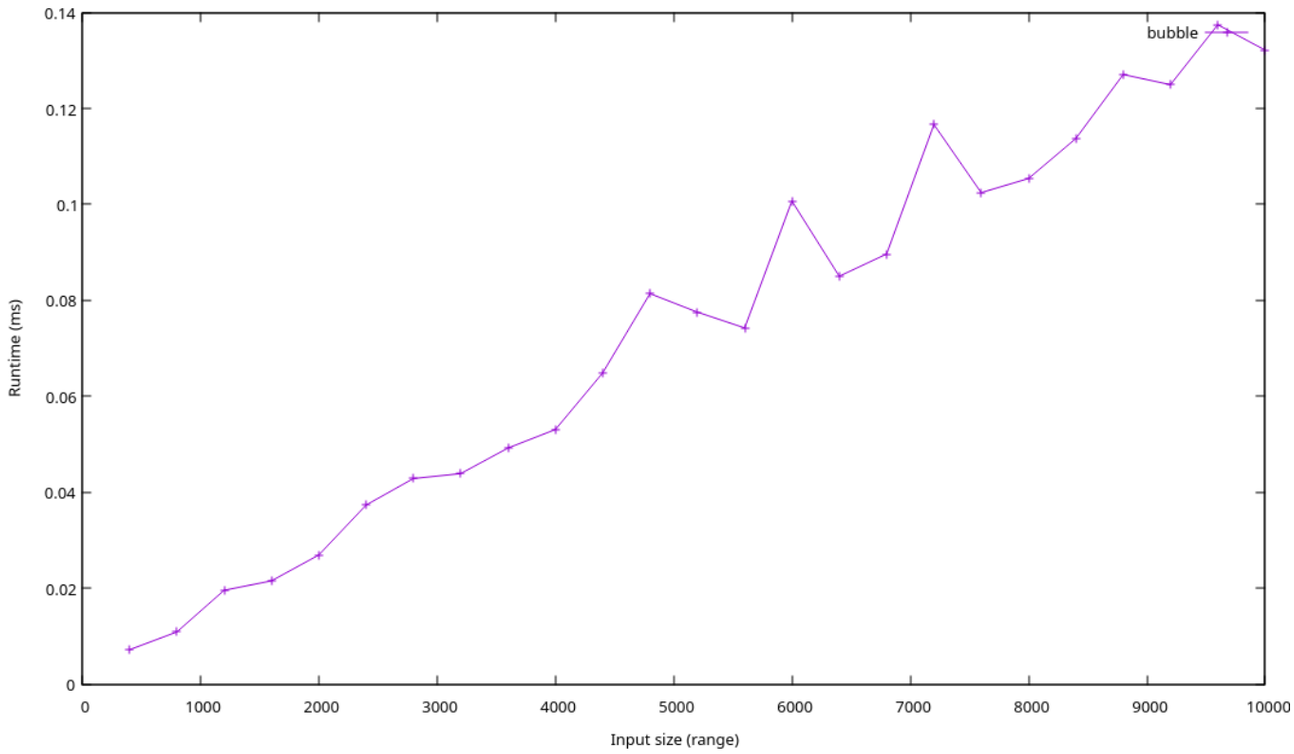


Figura 4: Análise de tempo para o shell sort, ordem não decrescente

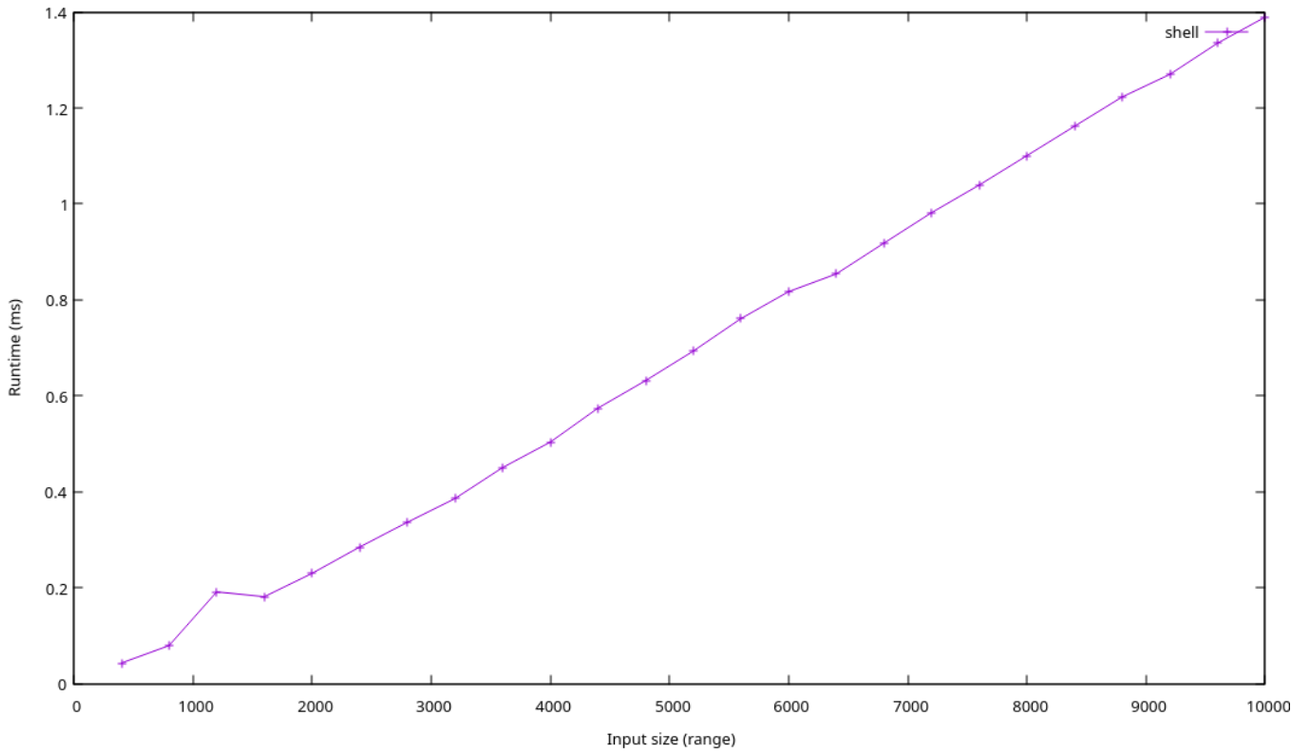


Figura 5: Análise de tempo para o merge sort, ordem não decrescente

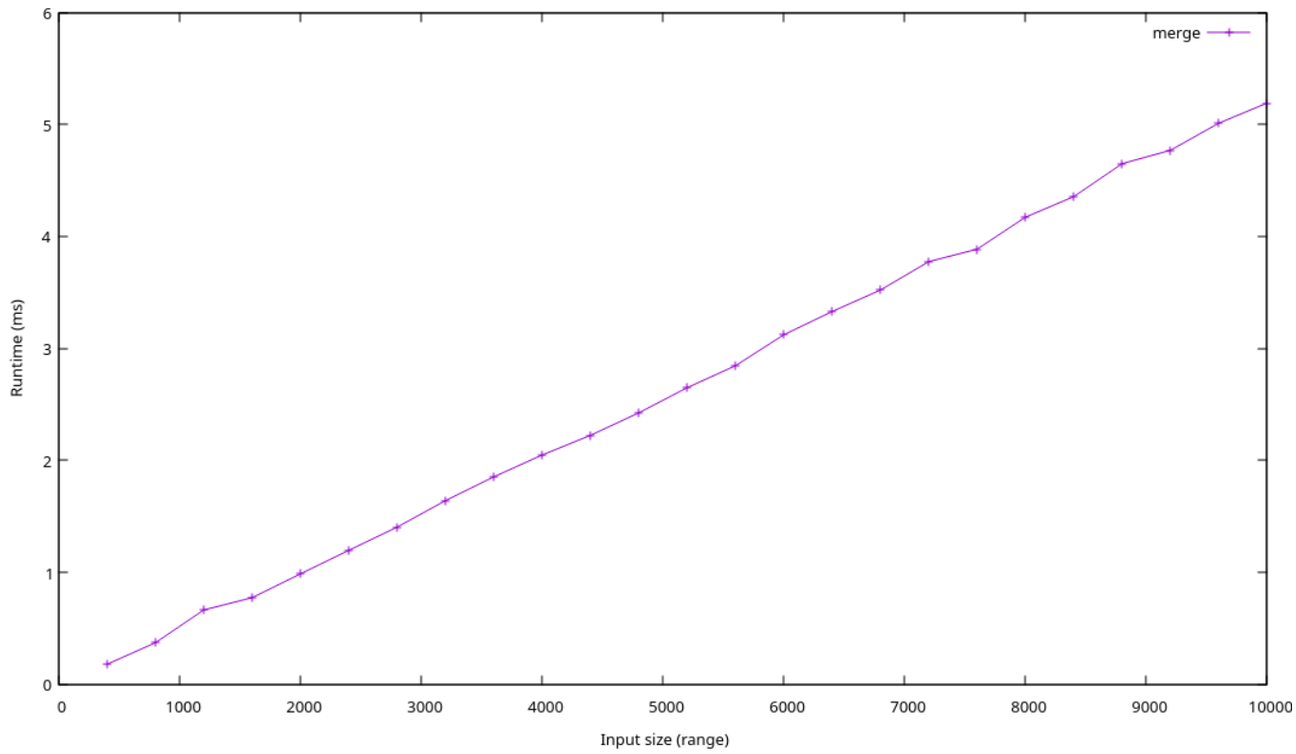


Figura 6: Análise de tempo para o quick sort, ordem não decrescente

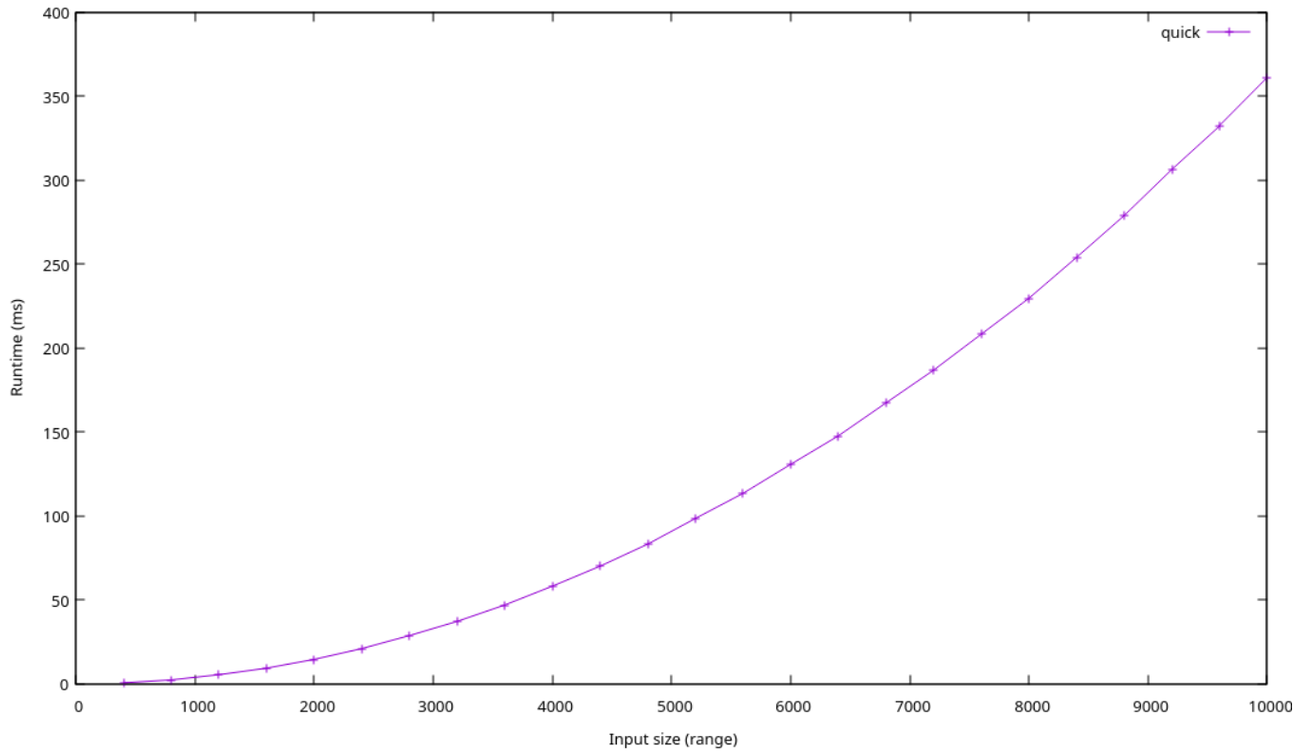


Figura 7: Análise de tempo para o radix sort, ordem não decrescente

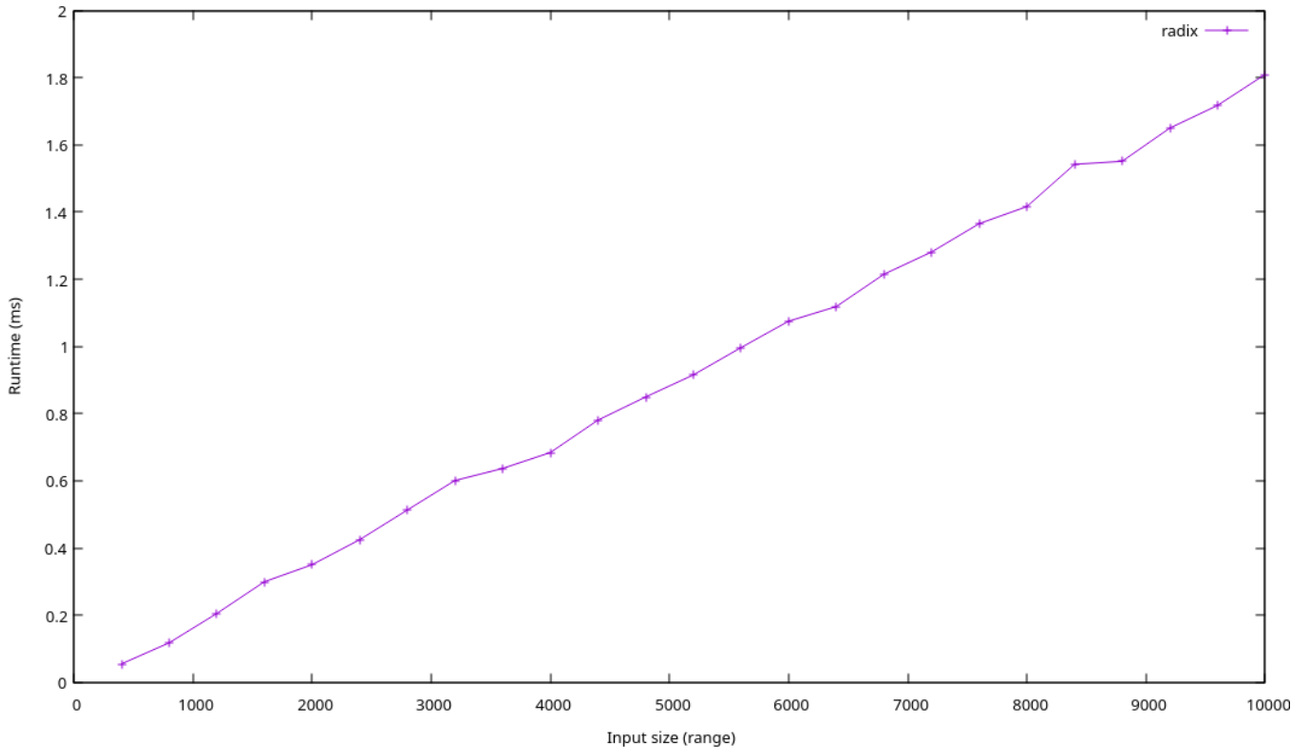
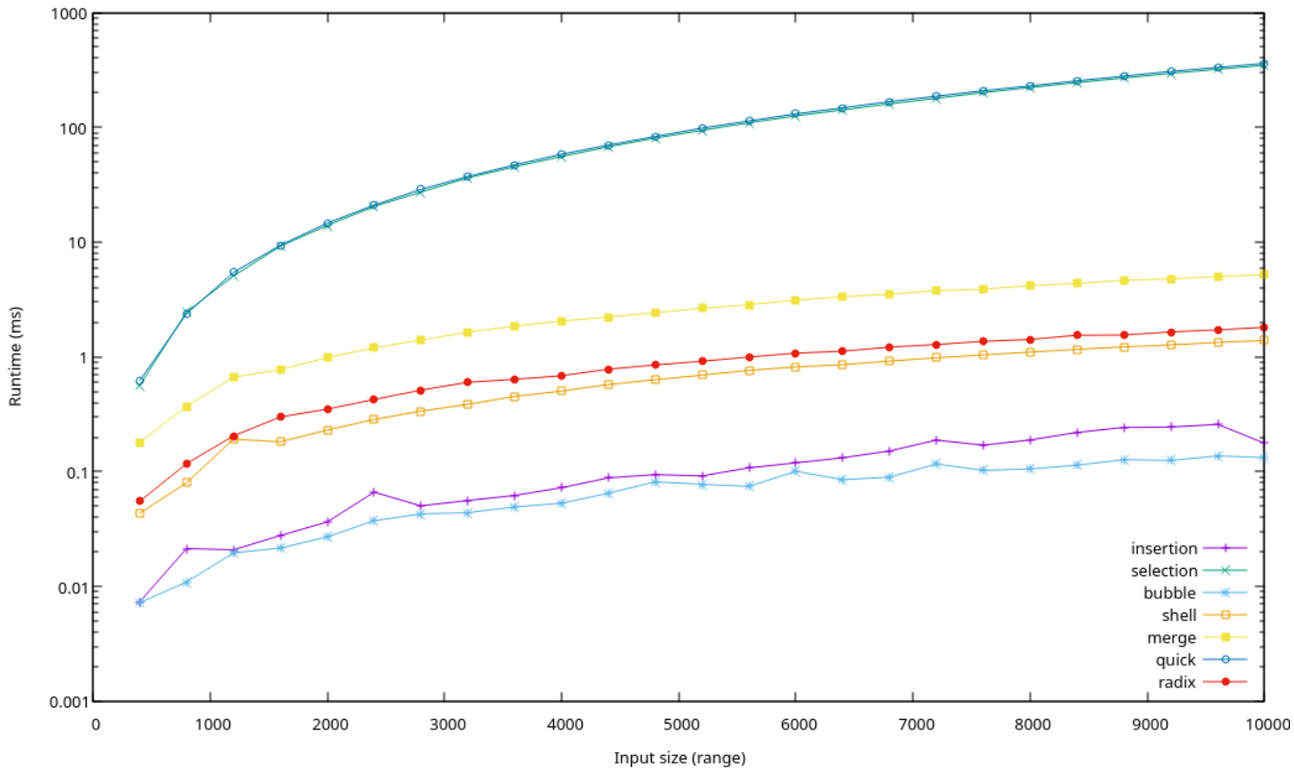


Figura 8: Análise de tempo de todos os algoritmos, ordem não decrescente



3.2 Cenário 2

A tabela seguinte apresenta o desempenho dos 7 algoritmos no cenário em que os elementos do vetor estão em **ordem não crescente**.

Tamanho da entrada	Tempo (ms)						
	Insertion	Selection	Bubble	Shell	Merge	Quick	Radix
400	3.56532	0.68188	4.22026	0.06724	0.1996	0.07722	0.05164
800	13.2171	2.7176	16.8128	0.14494	0.3954	0.17234	0.10646
1200	29.385	6.05626	37.8608	0.26008	0.63572	0.27032	0.19996
1600	54.3051	11.0018	69.9845	0.30546	0.8327	0.3811	0.27694
2000	87.8661	16.7557	106.28	0.4261	1.02866	0.48882	0.35972
2400	127.163	24.4533	152.412	0.51608	1.32202	0.58812	0.40564
2800	176.725	32.7781	208.311	0.66072	1.51688	0.7333	0.47658
3200	210.427	42.5734	268.26	0.75786	1.76208	0.8286	0.53986
3600	289.204	54.0769	340.258	0.87524	2.04694	0.97832	0.62306
4000	353.459	66.4148	439.566	0.95128	2.1193	1.05692	0.68944
4400	439.837	83.142	514.548	1.18112	2.61628	1.23634	0.8009
4800	493.525	96.3983	616.755	1.04692	2.74744	1.3016	0.86704
5200	593.131	113.306	779.566	1.3234	3.16928	1.43756	0.90642
5600	705.69	133.456	838.616	1.36808	3.27804	1.55032	1.0213
6000	795.301	151.269	957.7	1.4124	3.45524	1.67944	1.04186
6400	910.388	176.484	1095.45	1.5916	3.6392	1.83344	1.10286
6800	1012.91	194.485	1230.87	1.80728	3.96562	1.97446	1.18942
7200	1151.35	218.284	1382.56	1.77338	4.3452	2.10184	1.29724
7600	1274.78	242.873	1538.35	1.90616	4.4829	2.20706	1.32576
8000	1416.5	271.225	1718.57	2.04764	4.5157	2.33328	1.38526
8400	1578.99	305.825	1893.95	2.51638	4.88774	2.47948	1.48006
8800	1715.46	329.751	2090.56	2.25246	5.03788	2.59226	1.52562
9200	1876.52	349.858	2230.72	2.39582	5.46016	2.70572	1.5881
9600	2042.29	386.103	2463.88	2.68704	5.50126	2.80452	1.69104
10000	2201.99	418.257	2667	2.6173	5.84404	2.9934	2.13566

Tabela 2: Tempo dos algoritmos em função do tamanho da entrada quando o vetor está em ordem não crescente

Figura 9: Análise de tempo para o insertion sort, ordem não crescente

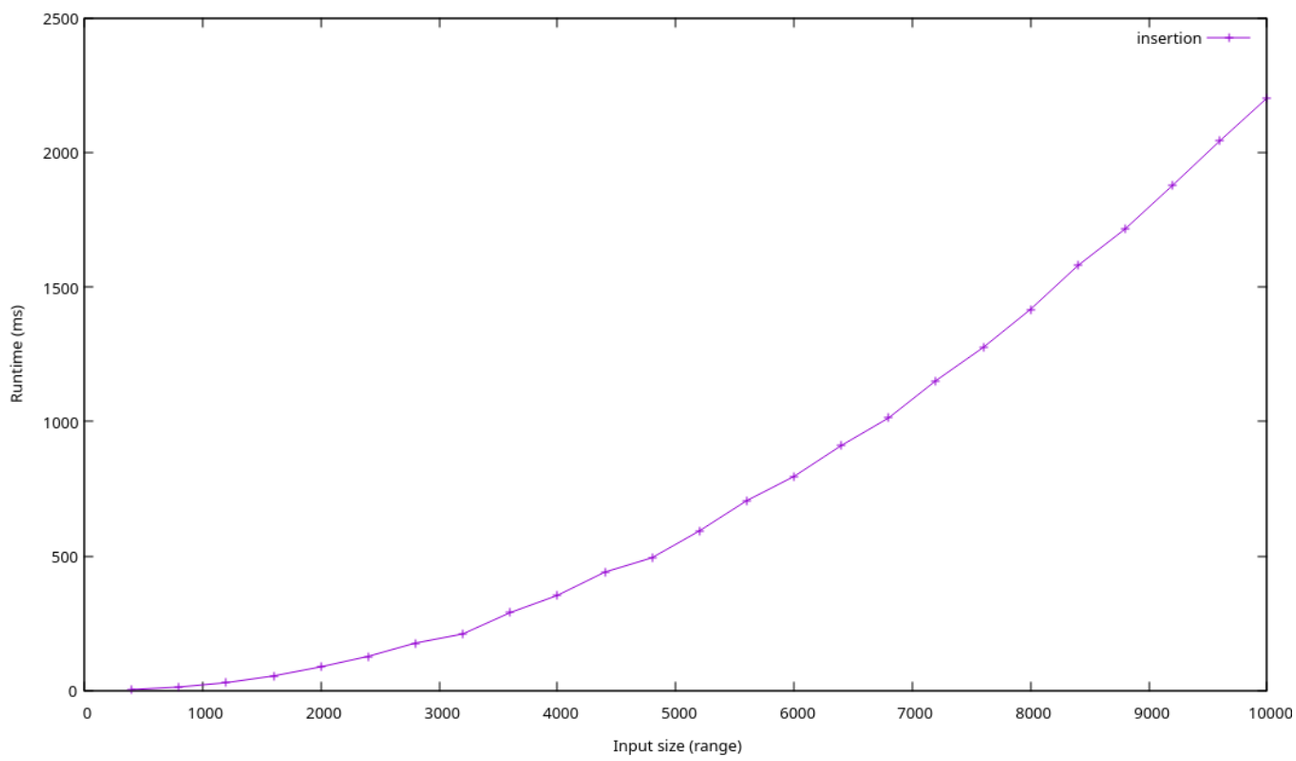


Figura 10: Análise de tempo para o selection sort, ordem não crescente

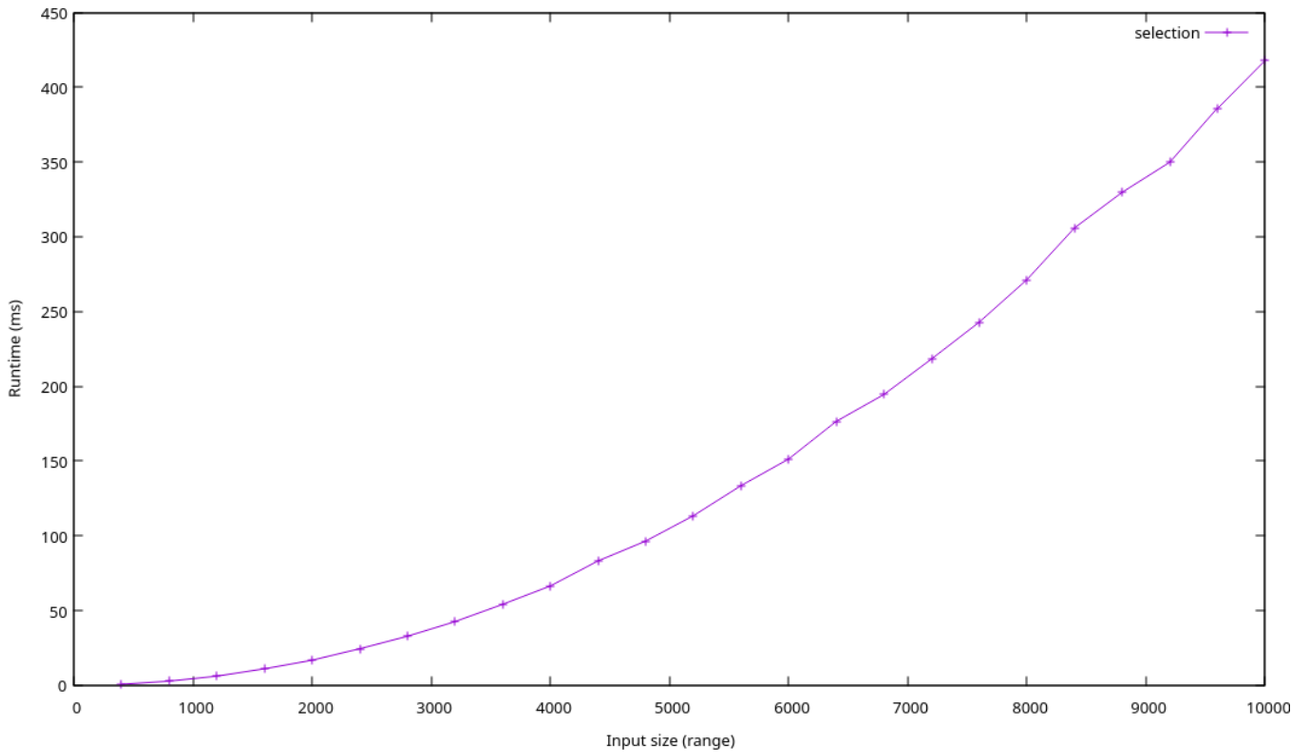


Figura 11: Análise de tempo para o bubble sort, ordem não crescente

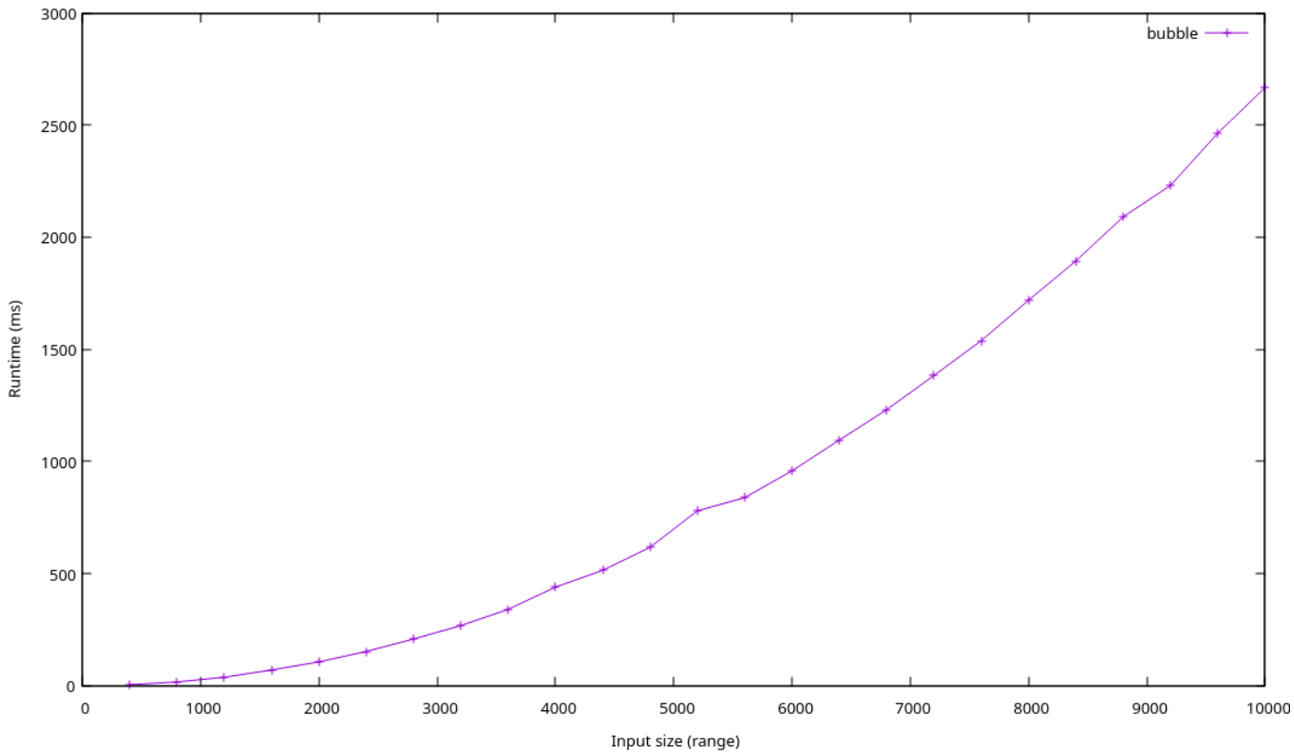


Figura 12: Análise de tempo para o shell sort, ordem não crescente

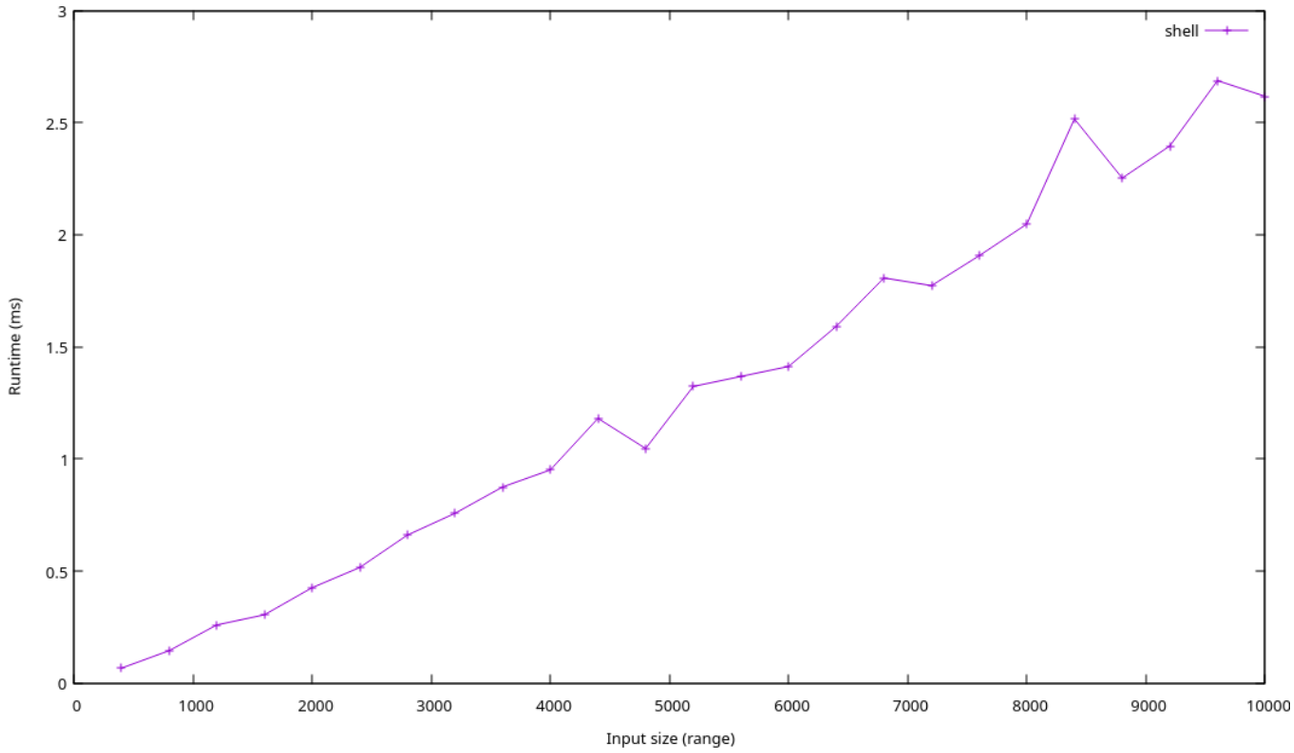


Figura 13: Análise de tempo para o merge sort, ordem não crescente

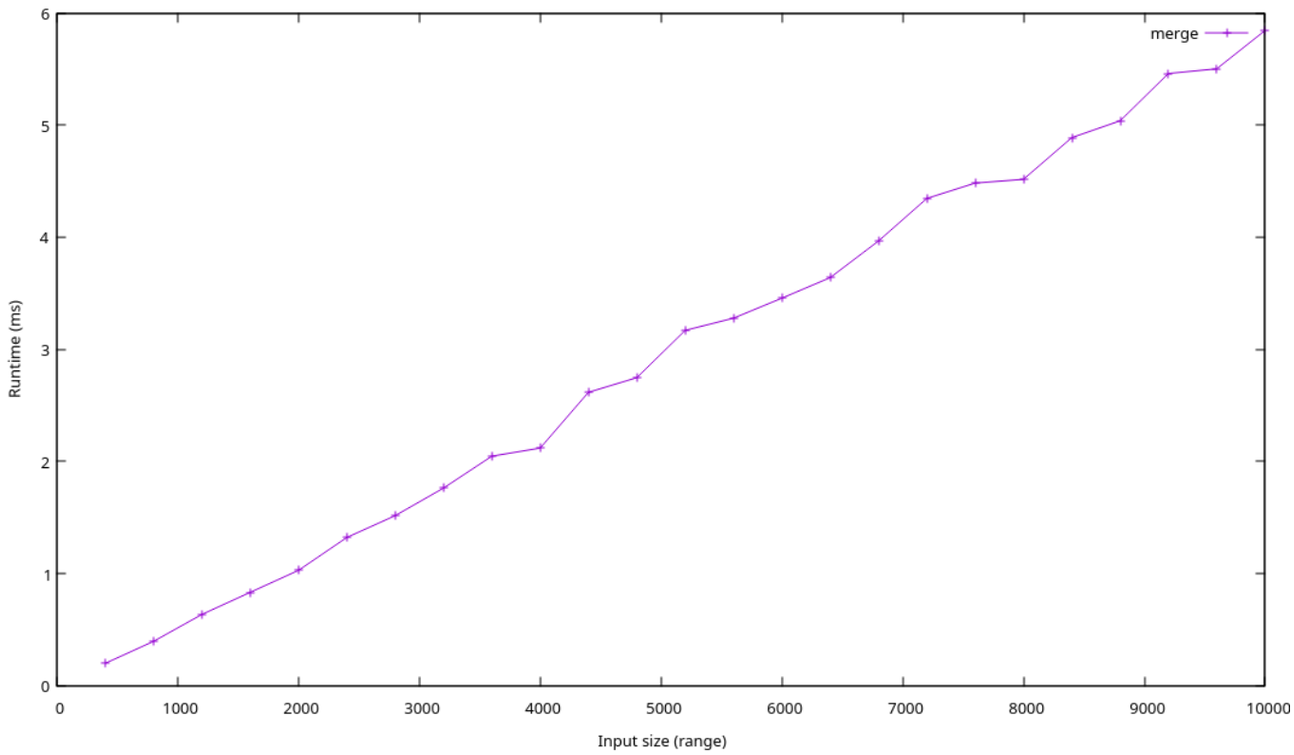


Figura 14: Análise de tempo para o quick sort, ordem não crescente

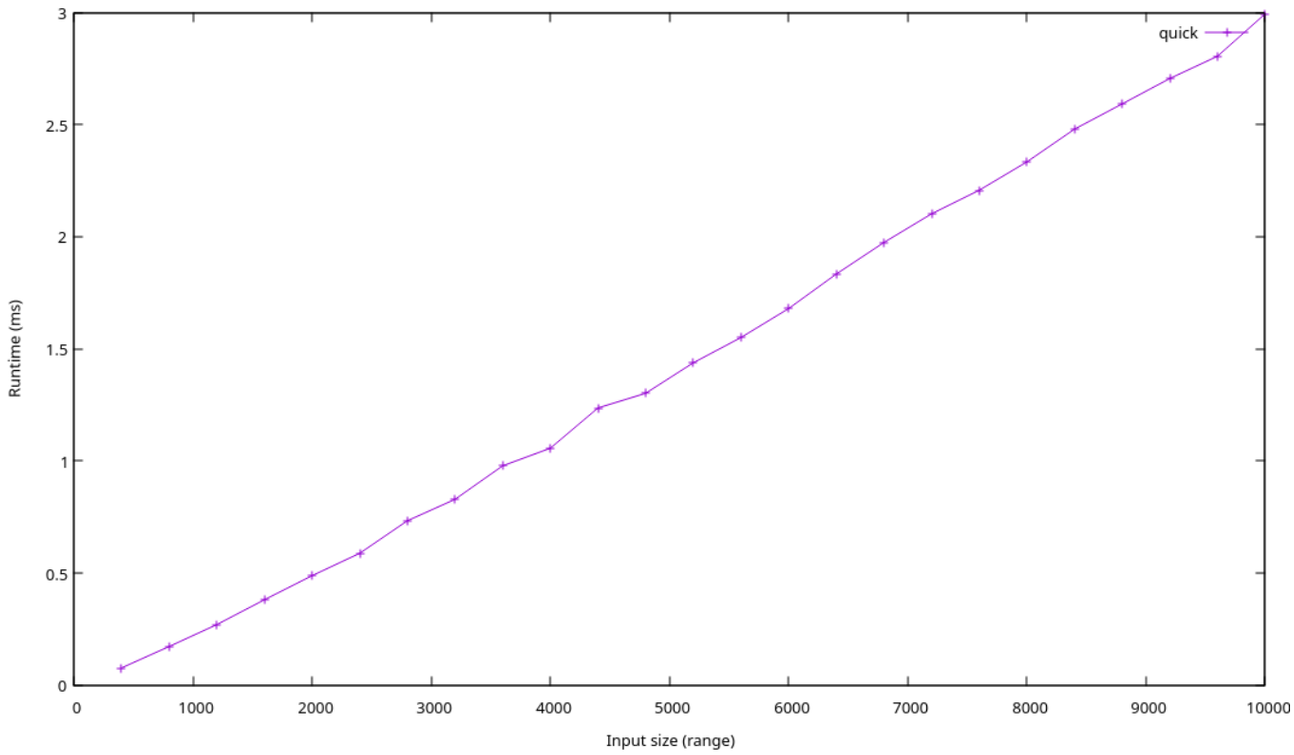


Figura 15: Análise de tempo para o radix sort, ordem não crescente

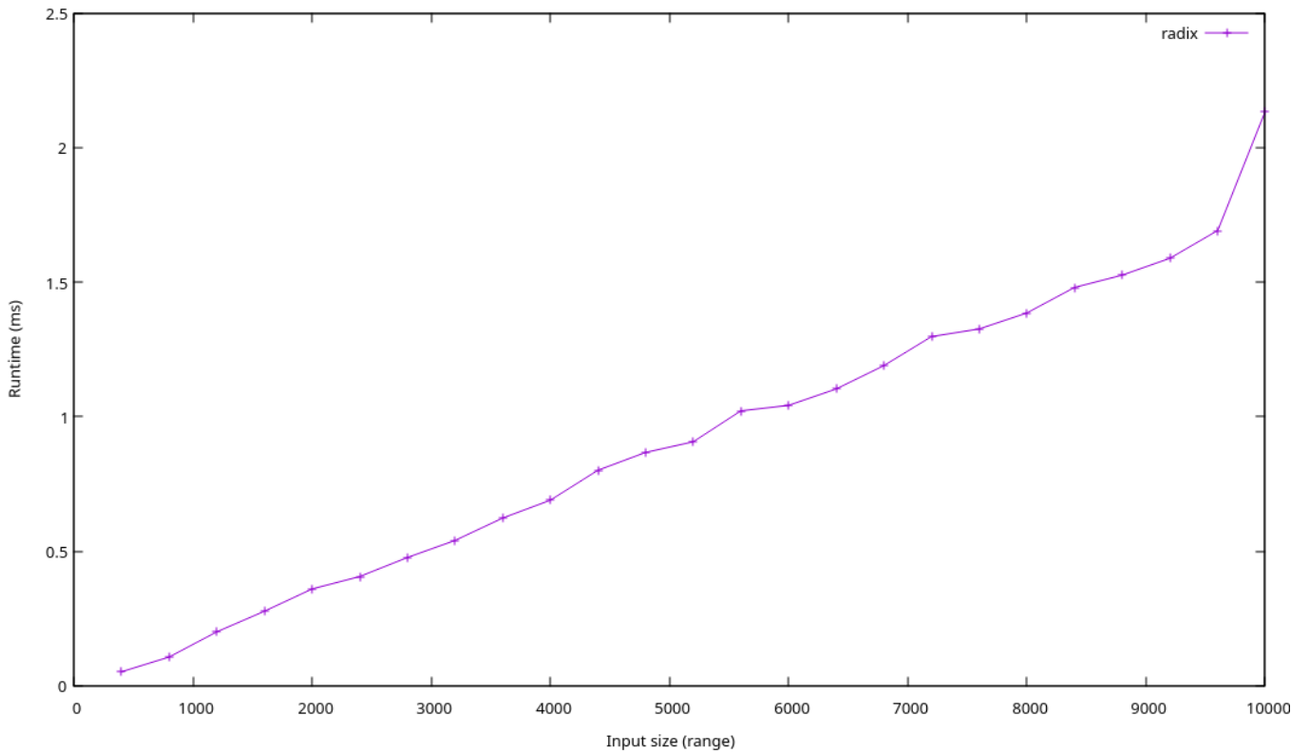
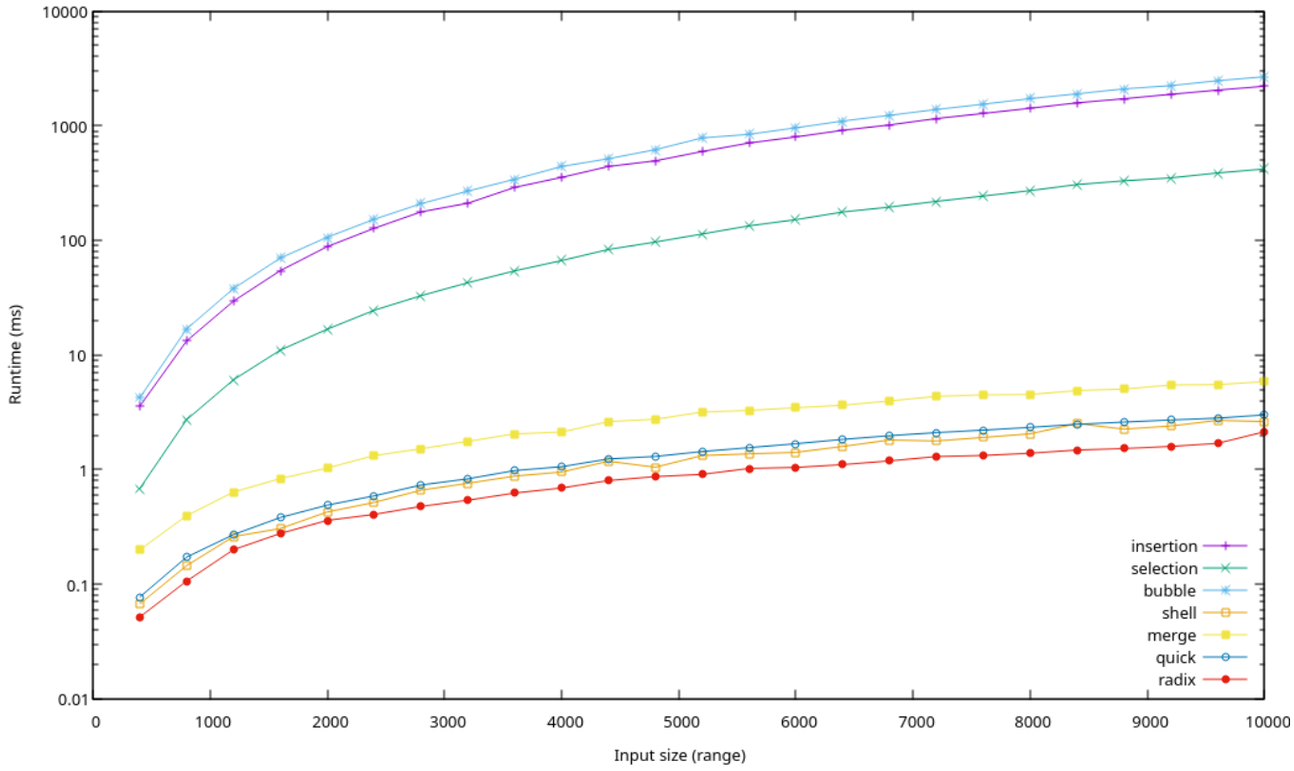


Figura 16: Análise de tempo de todos os algoritmos, ordem não crescente



3.3 Cenário 3

A tabela seguinte apresenta o desempenho dos 7 algoritmos no cenário em que os elementos do vetor estão em **ordem 100% aleatórios**.

Tamanho da entrada	Tempo (ms)						
	Insertion	Selection	Bubble	Shell	Merge	Quick	Radix
400	1.63946	0.58884	3.45854	0.14412	0.2416	0.07484	0.05106
800	7.0407	2.29062	13.5802	0.28544	0.47286	0.17128	0.1077
1200	15.2591	5.10458	30.552	0.6456	0.76612	0.27174	0.2234
1600	27.981	9.92008	54.3096	0.66684	1.00016	0.4002	0.2725
2000	42.5287	14.2429	86.2086	0.85934	1.24248	0.48332	0.34436
2400	68.8095	20.8522	125.925	1.0368	1.50036	0.56736	0.41342
2800	94.2063	27.9416	163.1	1.2927	1.79604	0.66652	0.48116
3200	115.385	36.4132	220.351	1.53018	2.09712	0.78018	0.71126
3600	134.753	45.5187	277.12	1.71008	2.35454	0.9261	0.62424
4000	166.729	56.186	343.996	1.8792	2.69306	1.0209	0.68534
4400	213.123	70.4321	425.918	2.30762	3.05436	1.26244	0.74306
4800	255.11	81.074	499.988	2.4885	3.21886	1.29276	0.83148
5200	301.79	95.639	583.086	2.68174	3.52652	1.4007	0.90006
5600	340.188	110.053	679.851	2.96256	3.72156	1.48804	0.9626
6000	378.719	125.503	765.054	3.15878	4.03724	1.60684	1.04732
6400	464.302	143.518	881.616	3.77614	4.48738	1.73502	1.13674
6800	515.305	161.623	988.126	3.93248	4.5748	1.81994	1.17588
7200	563.235	182.186	1131.48	4.32798	5.4615	1.96592	1.25926
7600	642.32	203.876	1246.95	4.11686	5.21766	2.3376	1.35186
8000	712.627	228.796	1367	4.40454	5.47958	2.20966	1.39572
8400	742.266	248.69	1516.91	4.71856	5.77036	2.41538	1.48906
8800	853.308	273.149	1631.8	4.80192	5.9904	2.47308	1.56446
9200	929.442	296.541	1837.52	5.26454	6.40492	2.65626	1.58406
9600	1006.09	322.654	1994.16	5.49856	6.72526	2.78612	1.69692
10000	1090.32	350.689	2154.95	5.78234	6.9488	2.82868	1.7479

Tabela 3: Tempo dos algoritmos em função do tamanho da entrada quando o vetor está em ordem dos elementos 100% aleatória

Figura 17: Análise de tempo para o insertion sort, ordem 100% aleatória

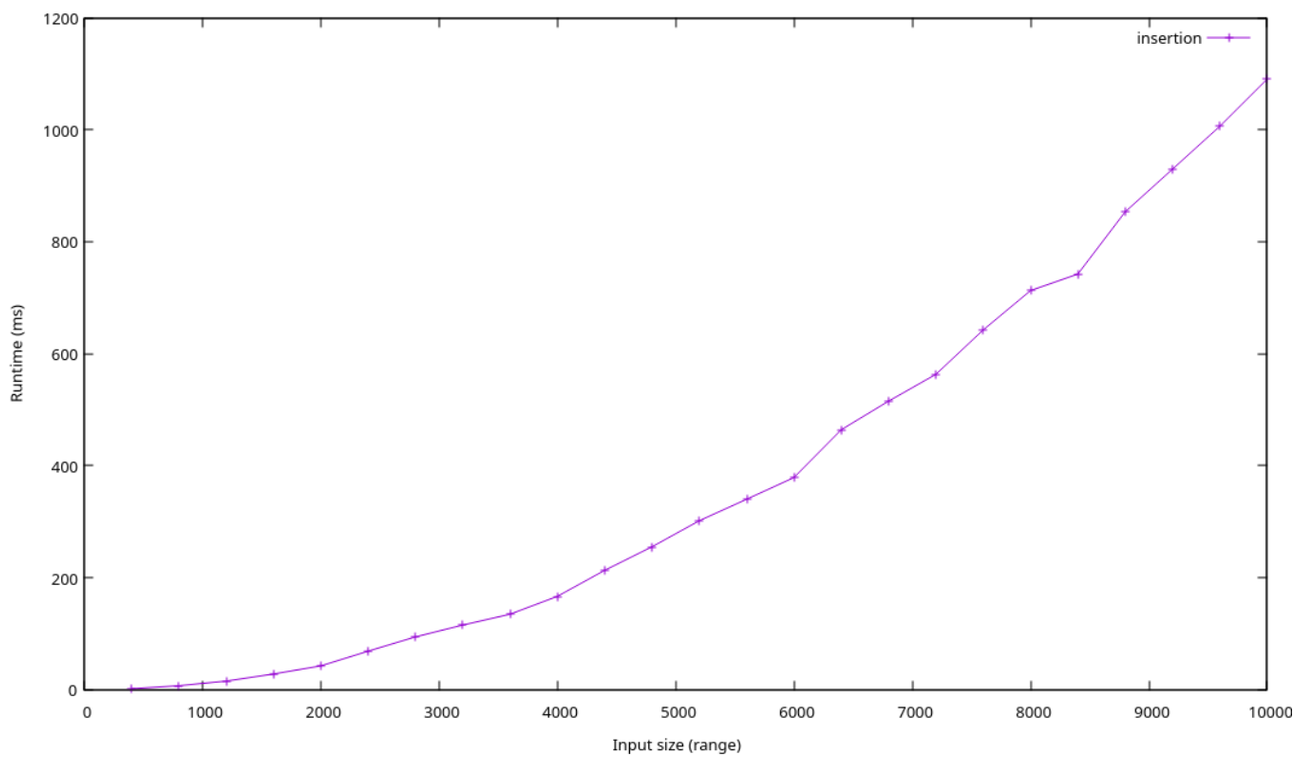


Figura 18: Análise de tempo para o selection sort, ordem 100% aleatória

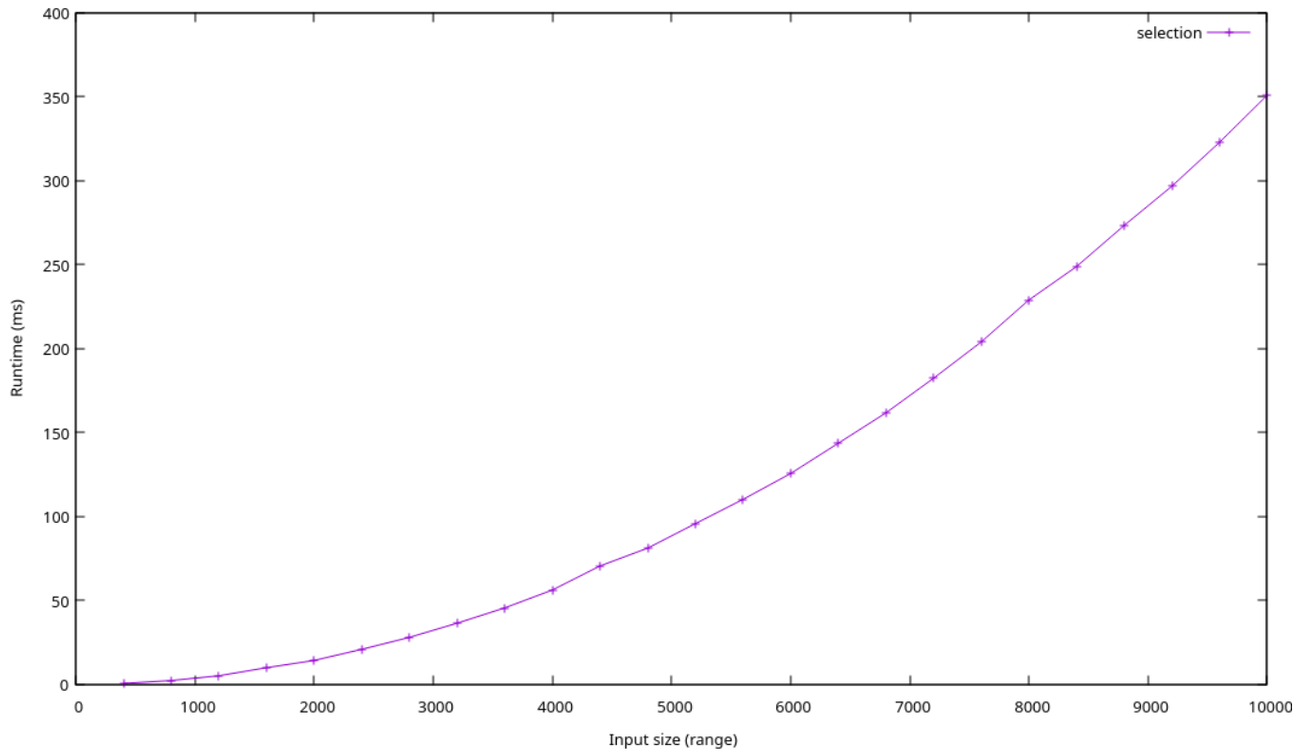


Figura 19: Análise de tempo para o bubble sort, ordem 100% aleatória

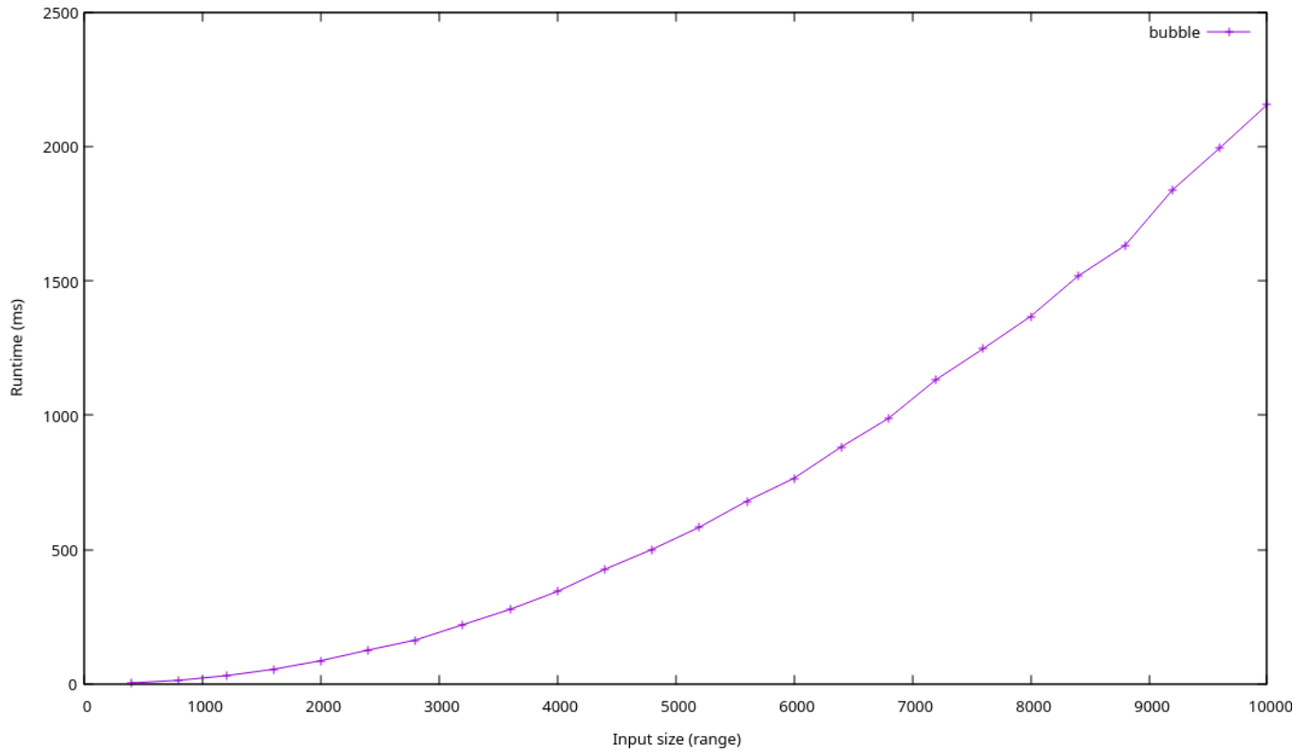


Figura 20: Análise de tempo para o shell sort, ordem 100% aleatória

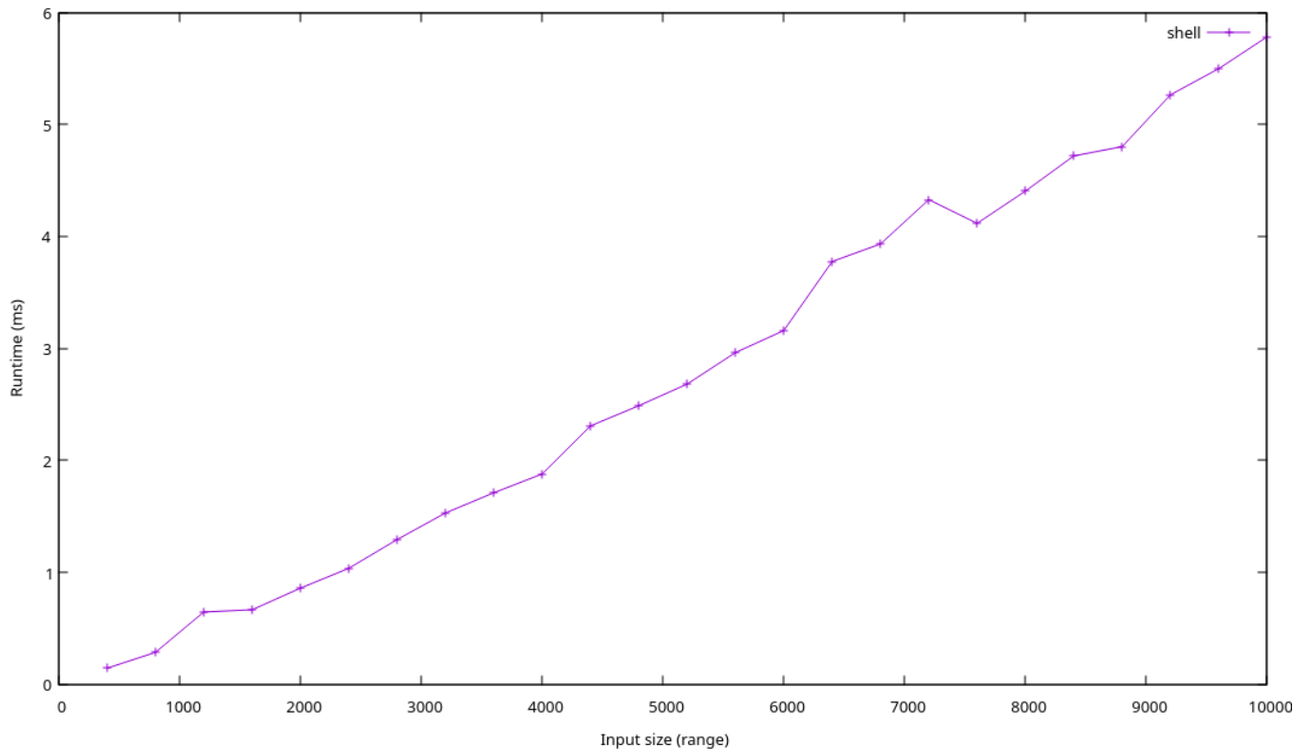


Figura 21: Análise de tempo para o merge sort, ordem 100% aleatória

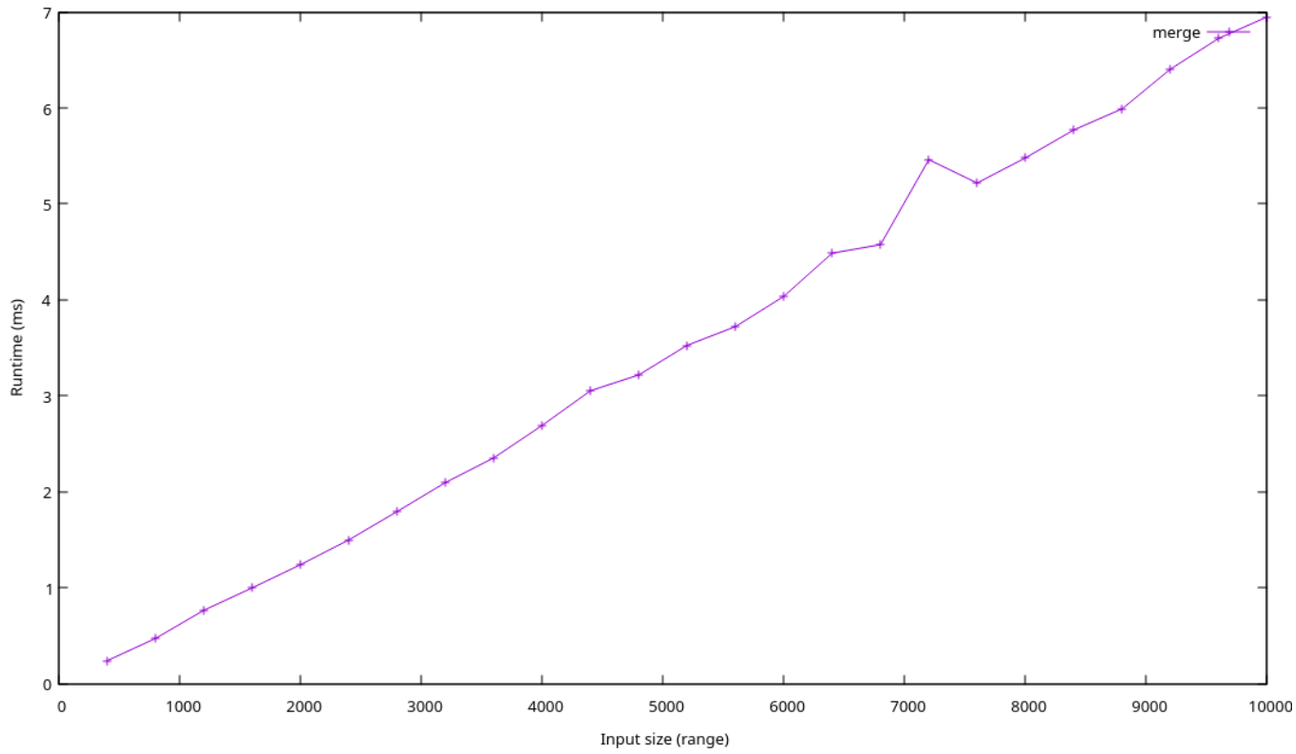


Figura 22: Análise de tempo para o quick sort, ordem 100% aleatória

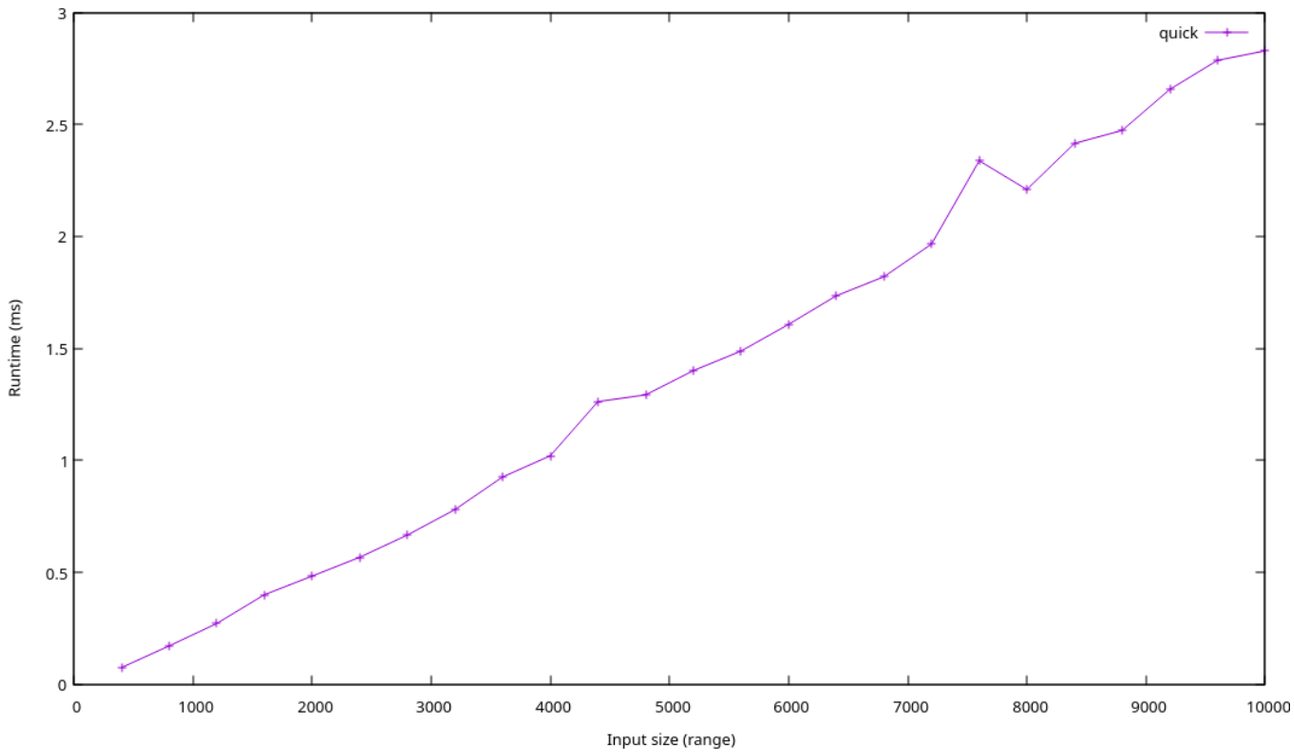


Figura 23: Análise de tempo para o radix sort, ordem 100% aleatória

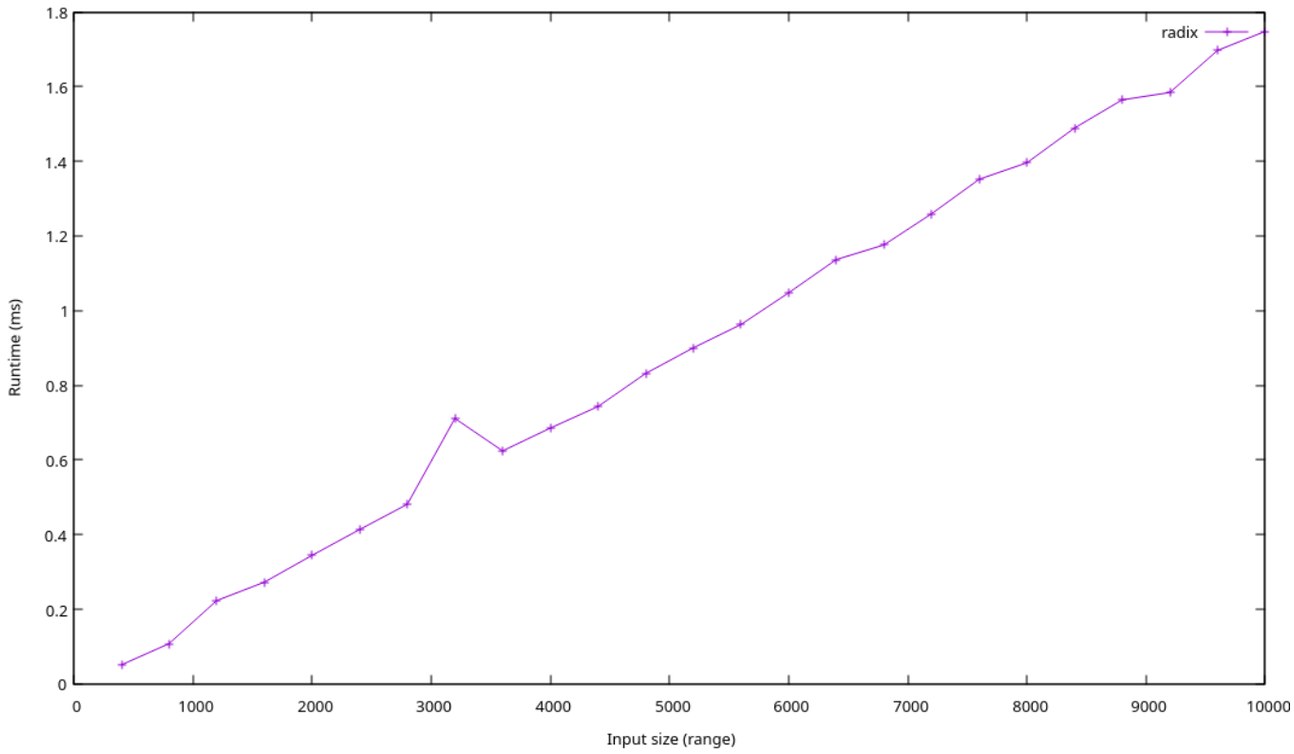
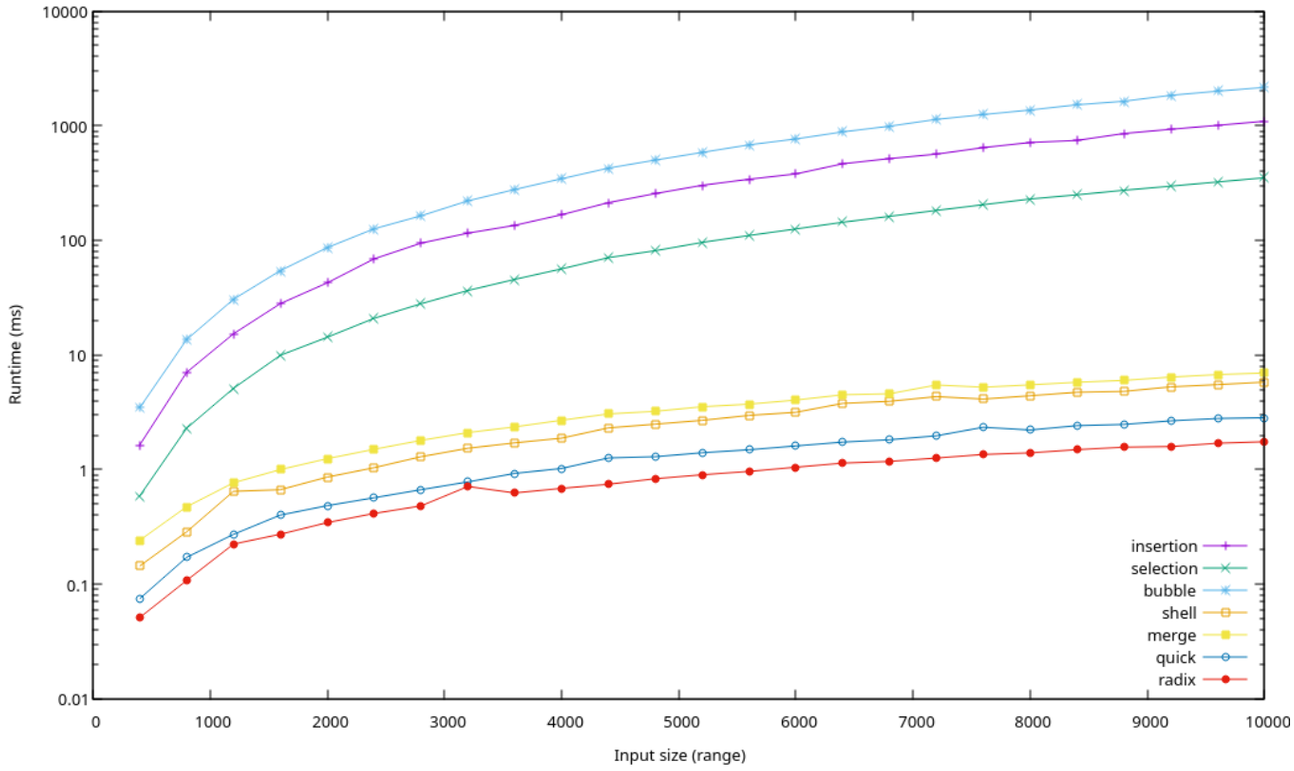


Figura 24: Análise de tempo de todos os algoritmos, ordem 100% aleatória



3.4 Cenário 4

A tabela seguinte apresenta o desempenho dos 7 algoritmos no cenário em que os elementos do vetor estão em **ordenados 75% dos elementos**.

Tamanho da entrada	Tempo (ms)						
	Insertion	Selection	Bubble	Shell	Merge	Quick	Radix
400	0.49806	0.57198	2.07278	0.09876	0.21414	0.09032	0.05554
800	2.06518	2.25388	9.5279	0.23446	0.43468	0.29594	0.10554
1200	4.92102	5.11928	21.3638	0.38412	0.672	0.54116	0.2711
1600	9.3416	8.97836	38.2563	0.5806	0.92798	0.47662	0.28218
2000	13.2949	14.1444	60.5371	0.77004	1.15276	0.8596	0.33586
2400	19.32	20.1967	87.7126	0.89638	1.3872	0.9026	0.41524
2800	27.864	27.3202	125.389	1.13194	1.67092	1.40168	0.49026
3200	36.2812	37.9667	164.665	1.33932	2.02442	1.16038	0.60356
3600	43.9198	46.8466	202.443	1.49684	2.18422	1.32942	0.6212
4000	56.6481	55.8576	252.447	1.71684	2.46702	1.39094	0.69976
4400	65.5226	67.6695	304.294	1.86252	2.69924	1.76028	0.76618
4800	80.8068	80.2869	374.295	2.1524	2.9192	1.97358	0.82042
5200	92.5833	94.4053	427.742	2.30818	3.19894	2.67074	0.89758
5600	108.821	109.58	507.942	2.6212	3.4862	2.35638	0.96604
6000	125.794	125.434	556.54	2.90124	3.6381	3.06548	1.03862
6400	133.778	144.005	625.177	3.03318	4.043	2.32168	1.13518
6800	163.77	162.287	724.222	3.30132	4.20618	2.99794	1.17534
7200	180.981	180.678	796.899	3.45844	4.56036	3.1825	1.24888
7600	200.705	200.919	887.702	3.60148	4.66276	3.16836	1.33184
8000	225.6	223.239	1035.75	3.93152	5.04858	3.4995	1.43568
8400	229.132	247.243	1098.2	4.07604	5.34738	3.90536	1.4739
8800	272.245	270.721	1193.92	4.32198	5.60604	3.24092	1.55354
9200	300.916	296.901	1340.47	4.72592	5.8145	3.8668	1.59256
9600	297.681	322.164	1447.95	5.50164	6.76648	3.69018	1.6896
10000	359	349.978	1572.17	5.18816	6.28972	4.08512	1.75942

Tabela 4: Tempo dos algoritmos em função do tamanho da entrada quando o vetor tem 75% dos elementos ordenados

Figura 25: Análise de tempo para o insertion sort, 75% ordenado

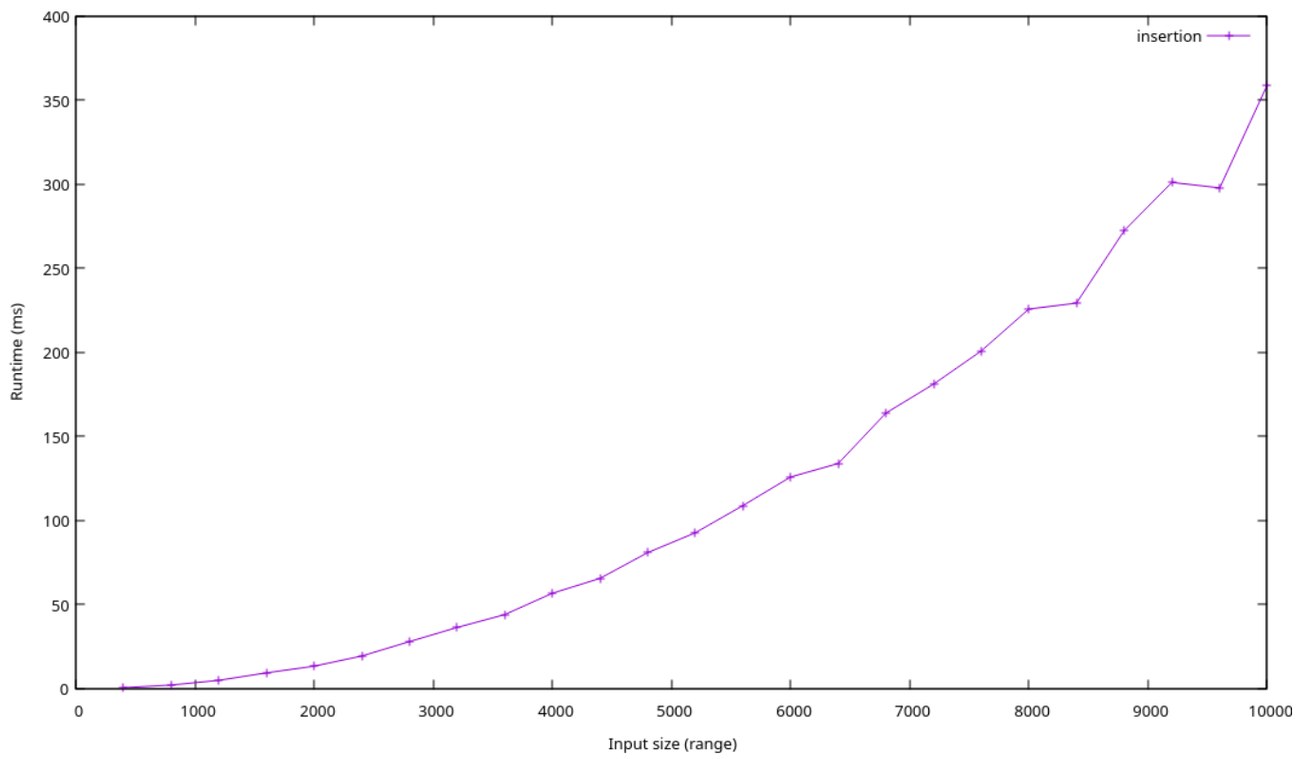


Figura 26: Análise de tempo para o selection sort, 75% ordenado

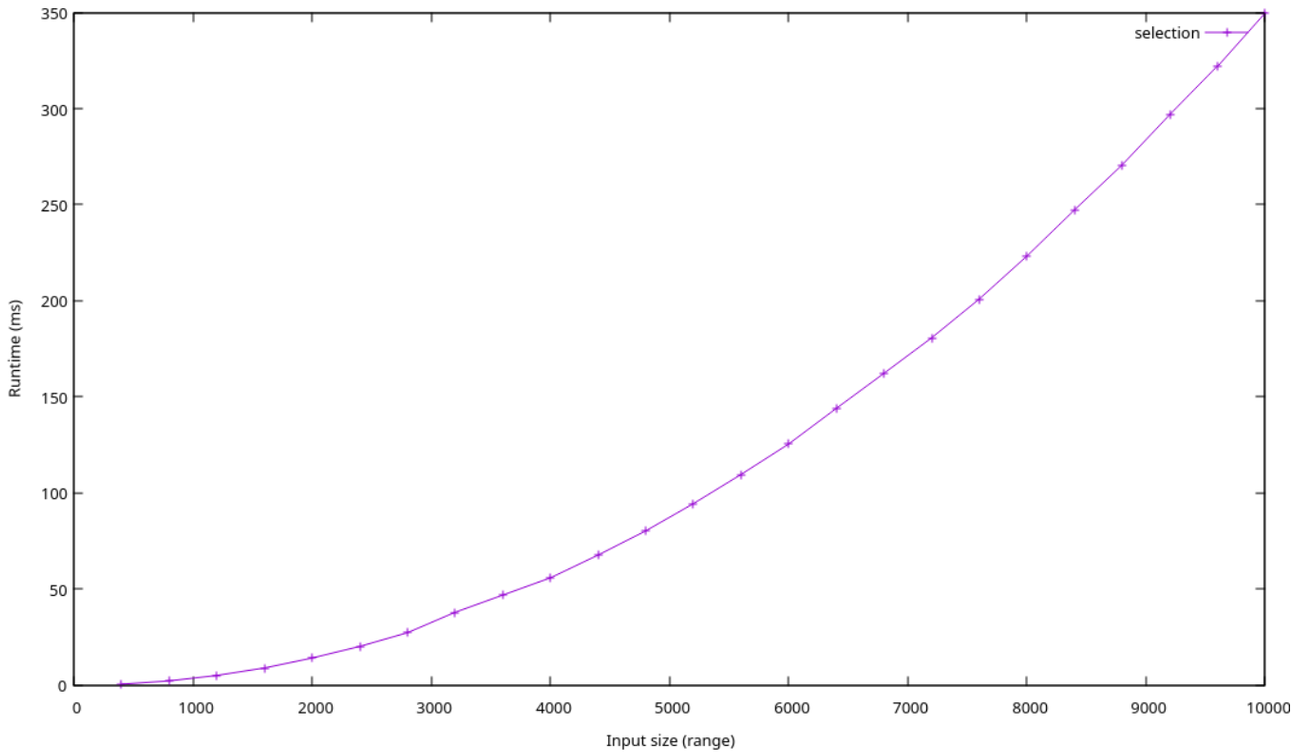


Figura 27: Análise de tempo para o bubble sort, 75% ordenado

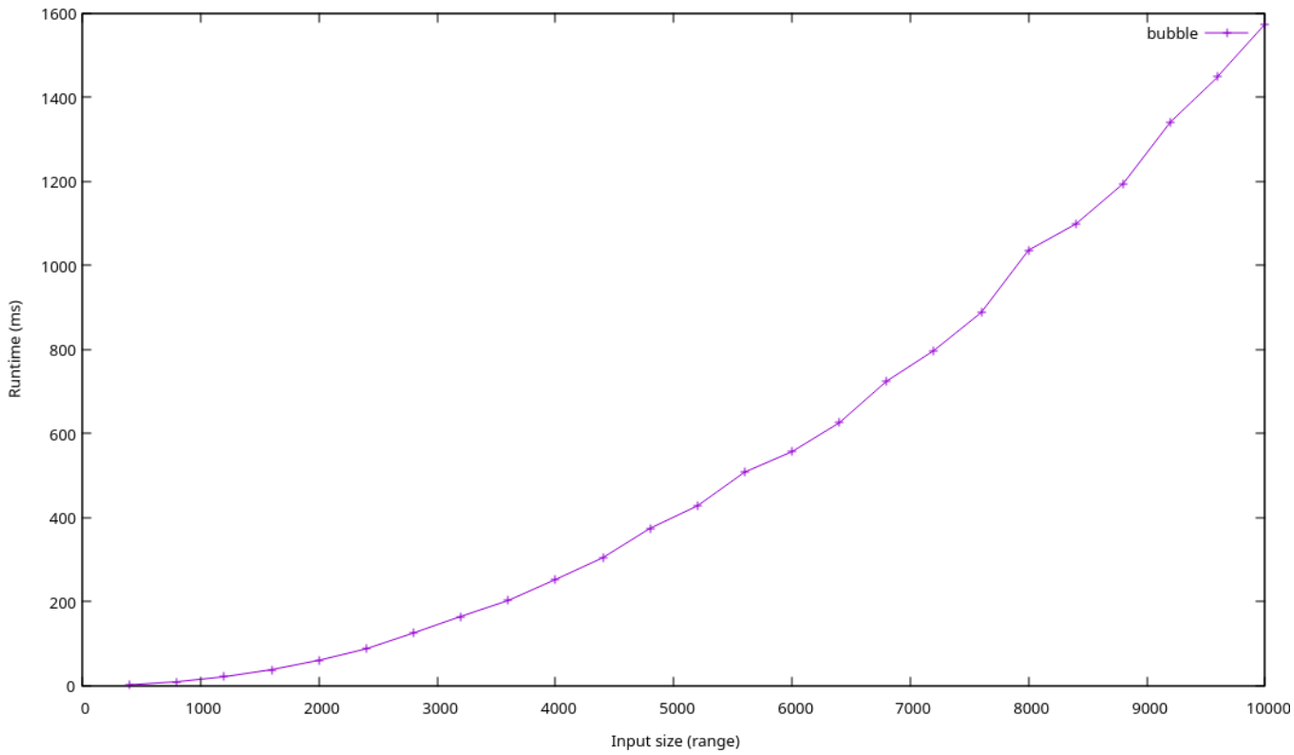


Figura 28: Análise de tempo para o shell sort, 75% ordenado

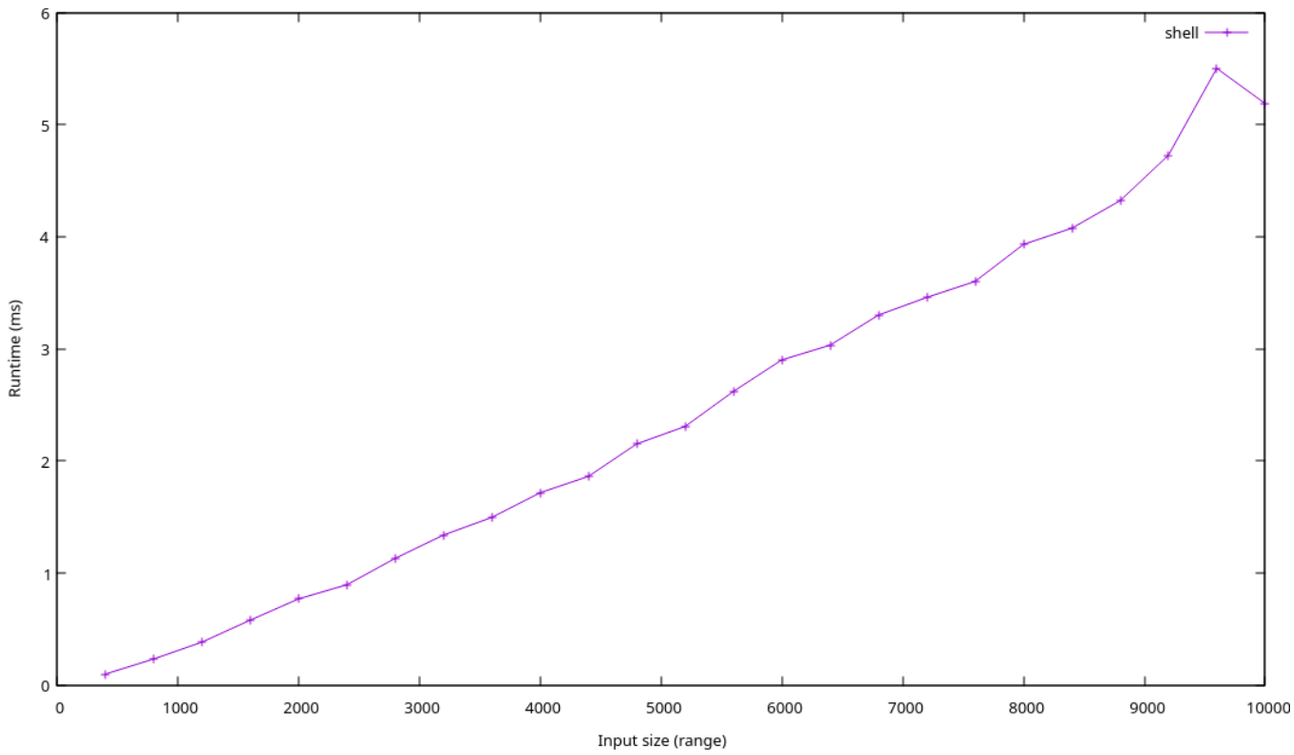


Figura 29: Análise de tempo para o merge sort, 75% ordenado

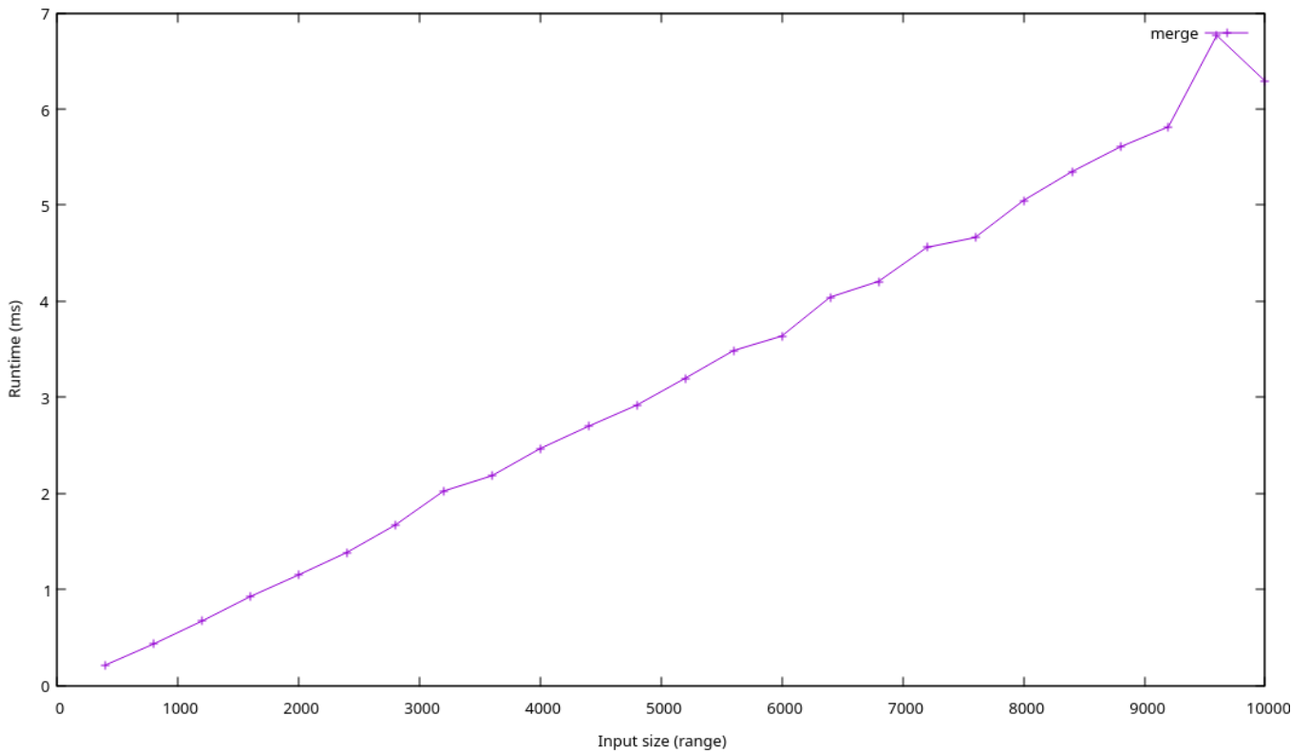


Figura 30: Análise de tempo para o quick sort, 75% ordenado

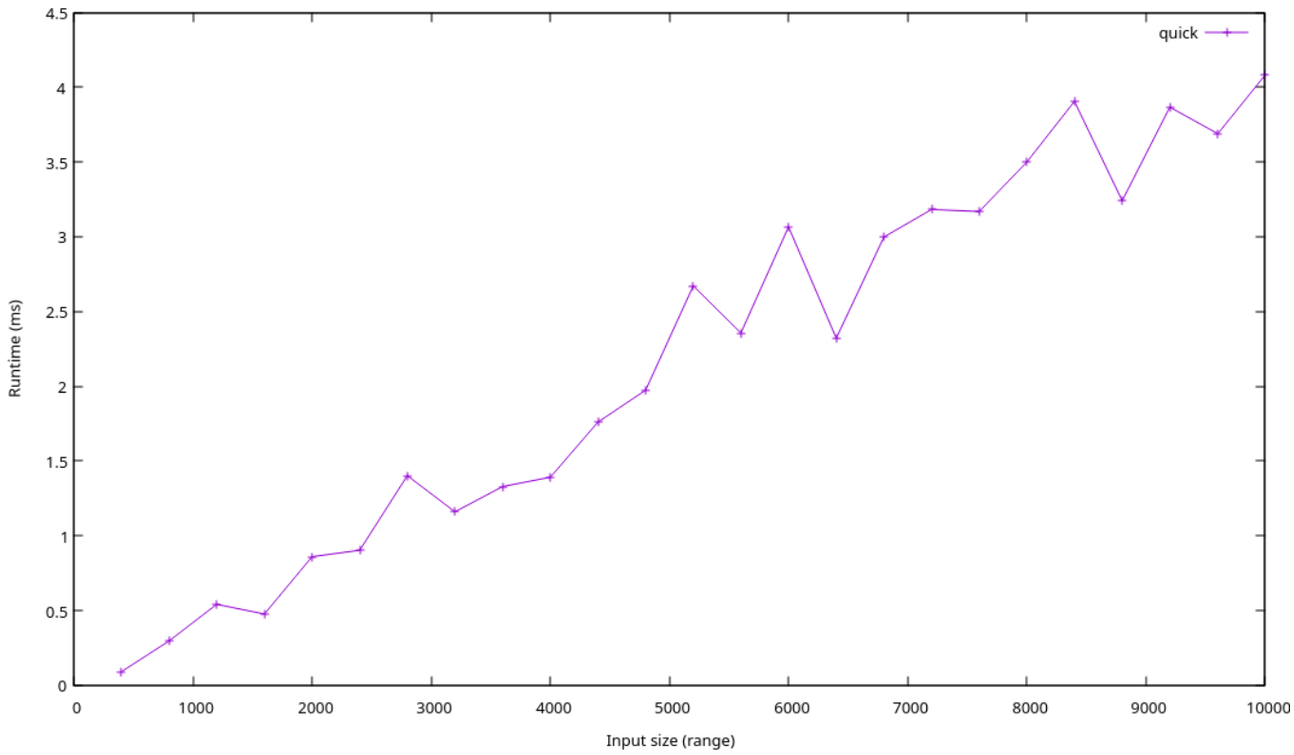


Figura 31: Análise de tempo para o radix sort, 75% ordenado

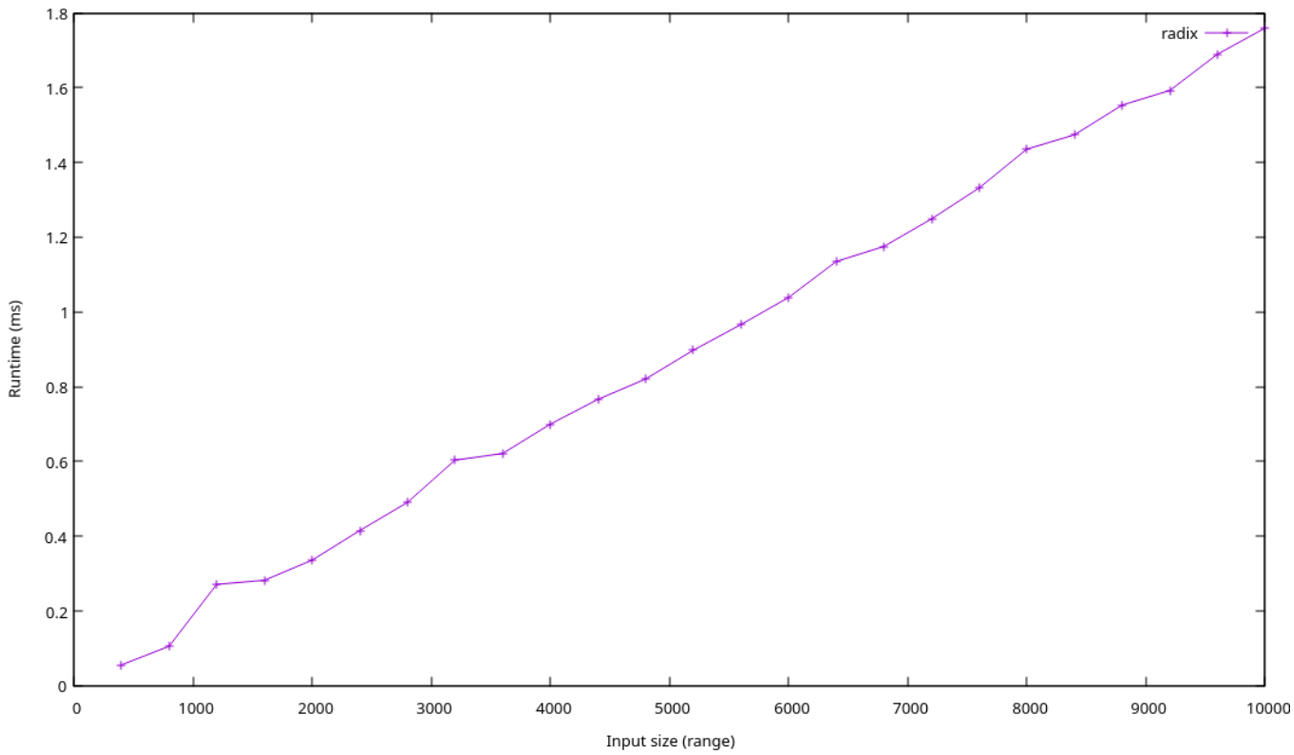
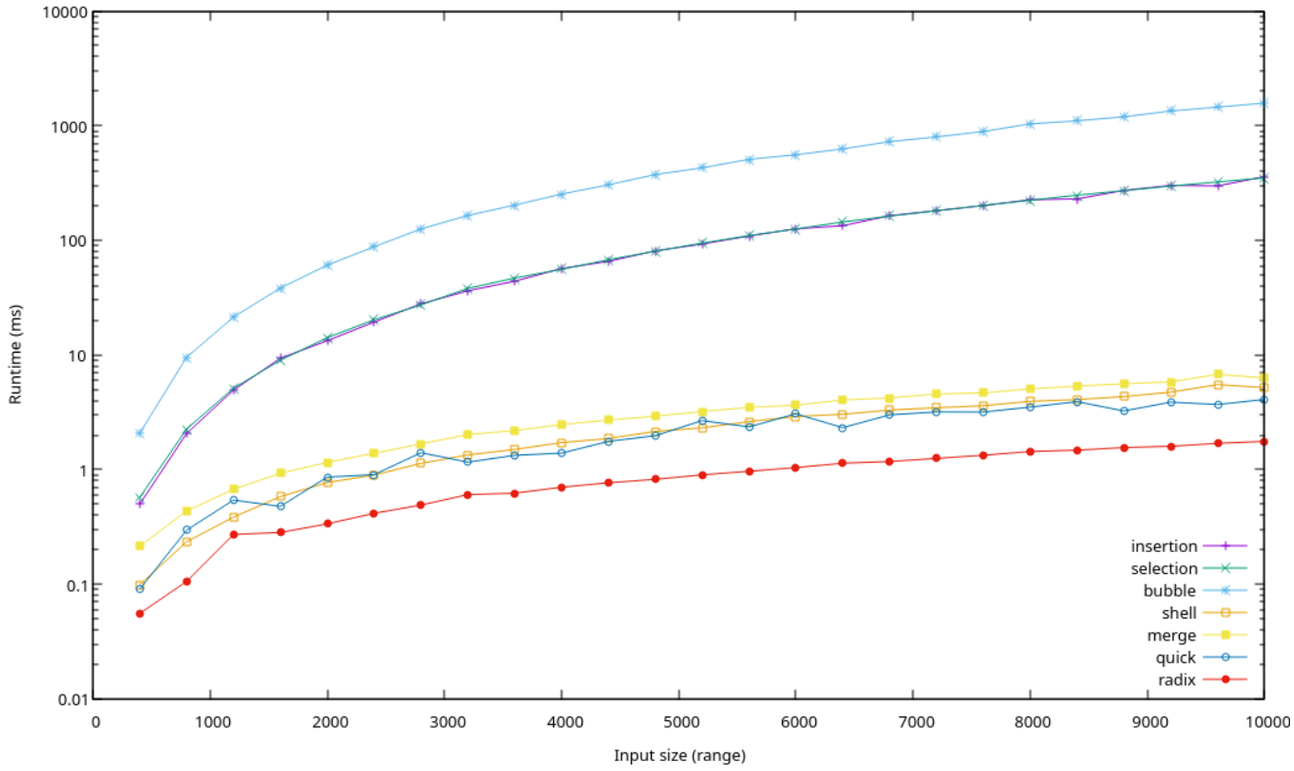


Figura 32: Análise de tempo de todos os algoritmos, 75% ordenado



3.5 Cenário 5

A tabela seguinte apresenta o desempenho dos 7 algoritmos no cenário em que os elementos do vetor estão em **ordenados 50% dos elementos**.

Tamanho da entrada	Tempo (ms)						
	Insertion	Selection	Bubble	Shell	Merge	Quick	Radix
400	1.09468	0.59212	2.78078	0.11042	0.24196	0.09406	0.05174
800	3.77418	2.30826	11.36	0.2581	0.45304	0.20208	0.1187
1200	8.91044	5.13442	26.2005	0.44516	0.78492	0.3039	0.20106
1600	16.0264	9.16388	45.2937	0.5858	0.95594	0.49936	0.28076
2000	29.1293	13.9392	72.4632	0.89794	1.24622	0.5847	0.35432
2400	35.5002	20.0619	103.997	0.98992	1.61294	0.75276	0.40884
2800	50.5922	27.4391	145.514	1.22066	1.73102	0.80732	0.48272
3200	65.1743	36.8555	187.724	1.39454	1.97906	0.90832	0.54622
3600	82.963	46.7302	239.182	1.62672	2.21308	1.25282	0.61758
4000	99.6351	55.6497	299.292	1.9083	2.52156	1.21066	0.71876
4400	128.294	69.7013	356.524	2.10652	2.77798	1.3998	0.75704
4800	144.082	80.0983	402.766	2.3163	3.0625	1.52114	0.84252
5200	173.019	94.1047	486.956	2.62138	3.384	1.68608	0.89466
5600	189.018	109.066	553.317	2.79746	3.7635	1.79198	0.9815
6000	237.641	125.522	667.796	2.96256	3.83552	1.97954	1.1052
6400	256.973	152.514	741.177	3.25812	4.14508	2.08548	1.11988
6800	298.336	160.709	821.762	3.4645	4.47184	2.09914	1.1733
7200	329.887	179.847	919.676	3.66098	4.73896	2.37008	1.32866
7600	379.229	212.942	1049.23	4.20214	4.98994	2.16888	1.31964
8000	403.898	233.592	1177.75	4.29936	5.2418	2.75972	1.42022
8400	461.245	245.087	1275.87	4.52608	5.55186	2.56604	1.48028
8800	492.397	270.694	1411.38	4.9358	5.89104	3.0495	1.54726
9200	542.369	294.698	1546.82	4.87876	6.20328	3.10342	1.58738
9600	588.047	321.731	1656.14	5.53244	6.40596	3.17824	1.78188
10000	634.562	348.852	1785.18	5.66072	7.50132	3.26712	1.8505

Tabela 5: Tempo dos algoritmos em função do tamanho da entrada quando o vetor tem 50% dos elementos ordenados

Figura 33: Análise de tempo para o insertion sort, 50% ordenado

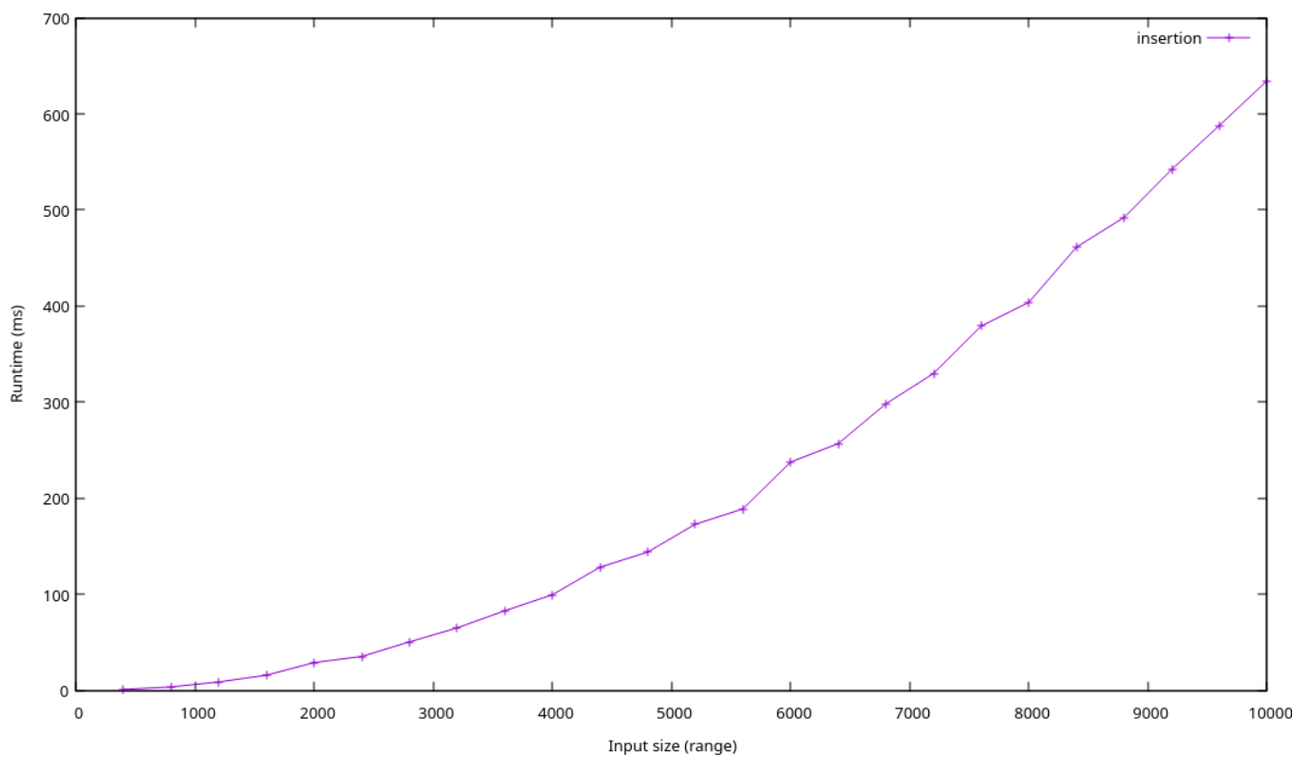


Figura 34: Análise de tempo para o selection sort, 50% ordenado

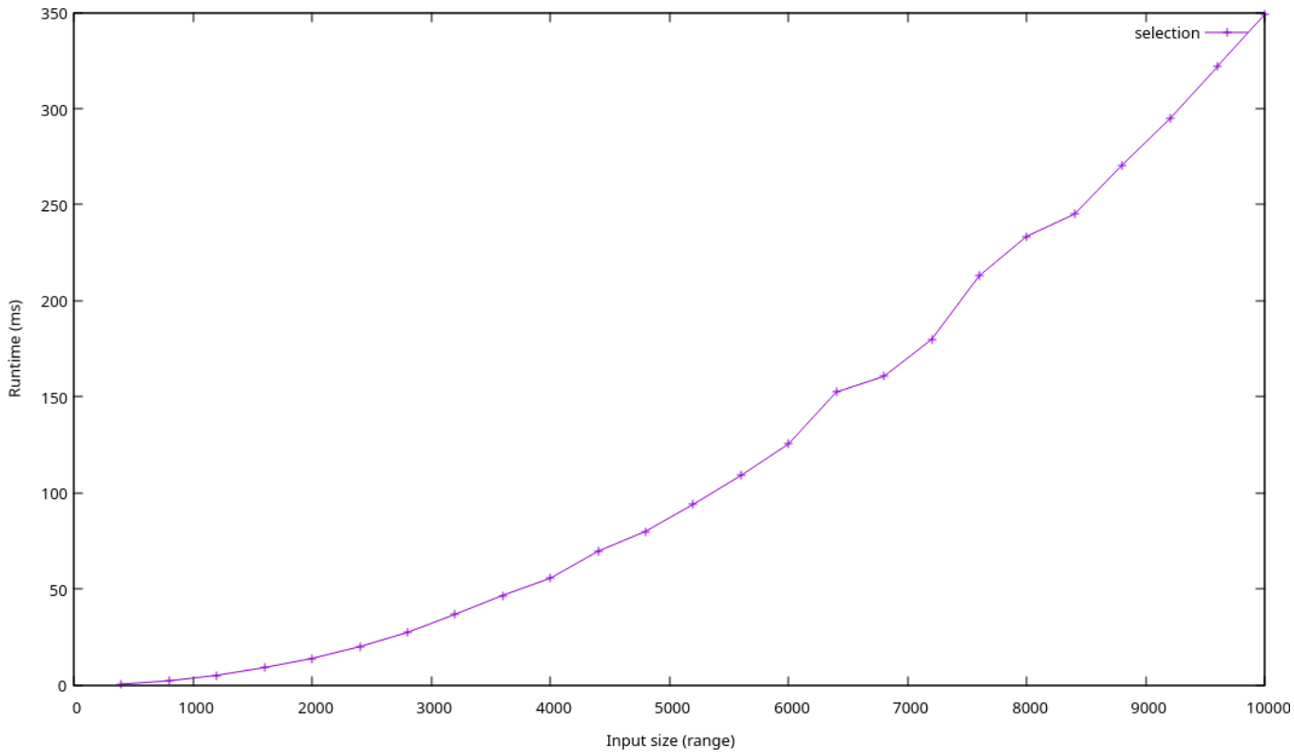


Figura 35: Análise de tempo para o bubble sort, 50% ordenado

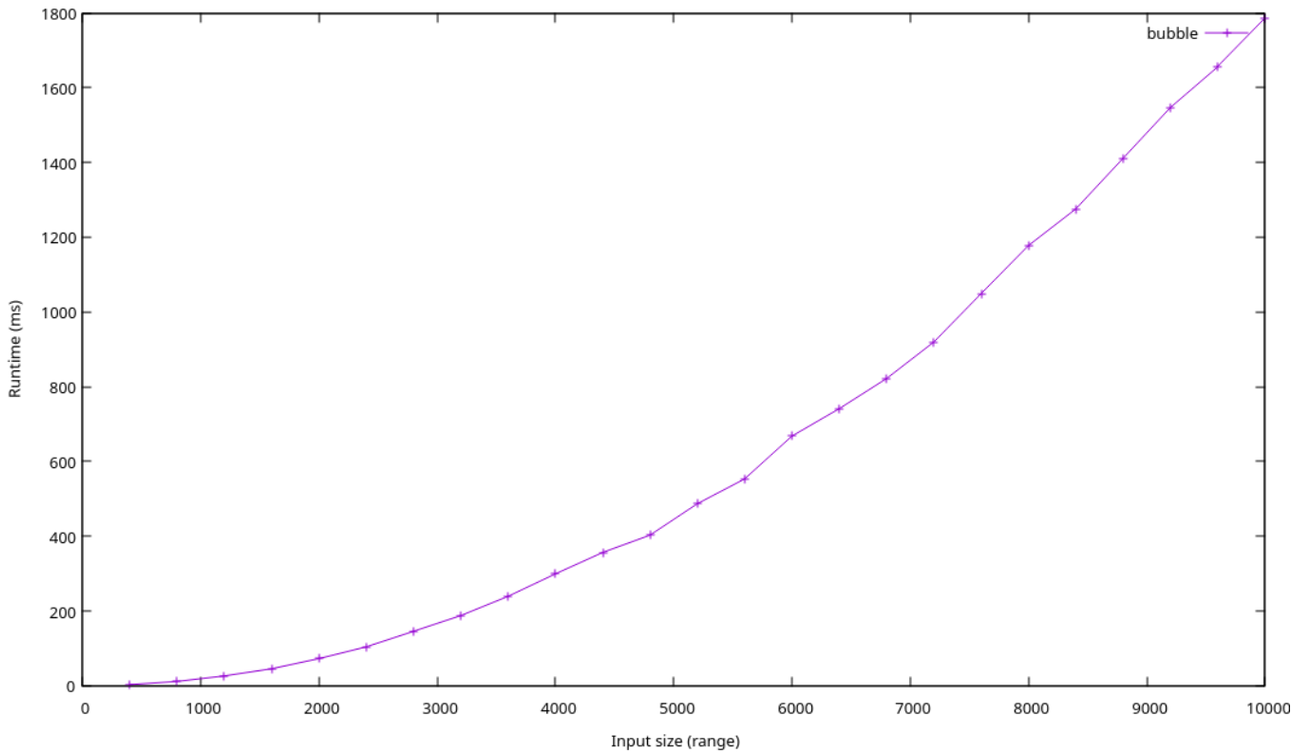


Figura 36: Análise de tempo para o shell sort, 50% ordenado

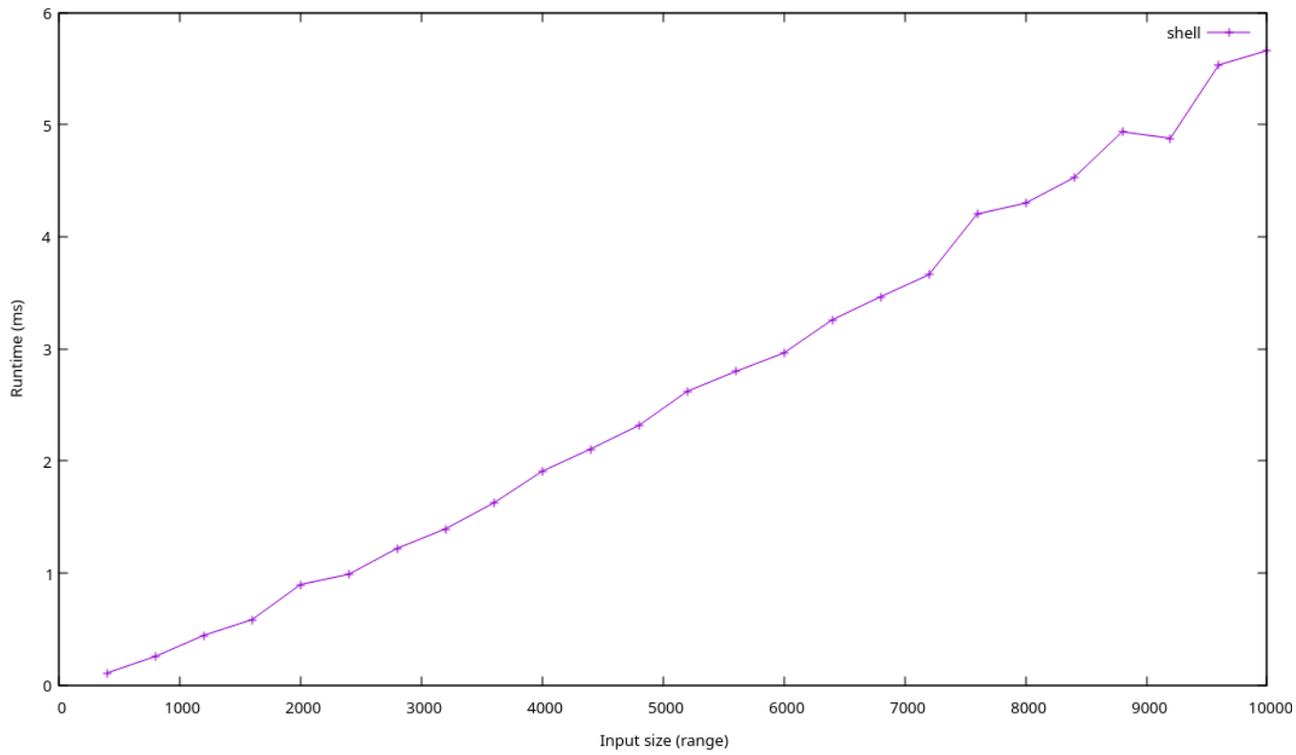


Figura 37: Análise de tempo para o merge sort, 50% ordenado

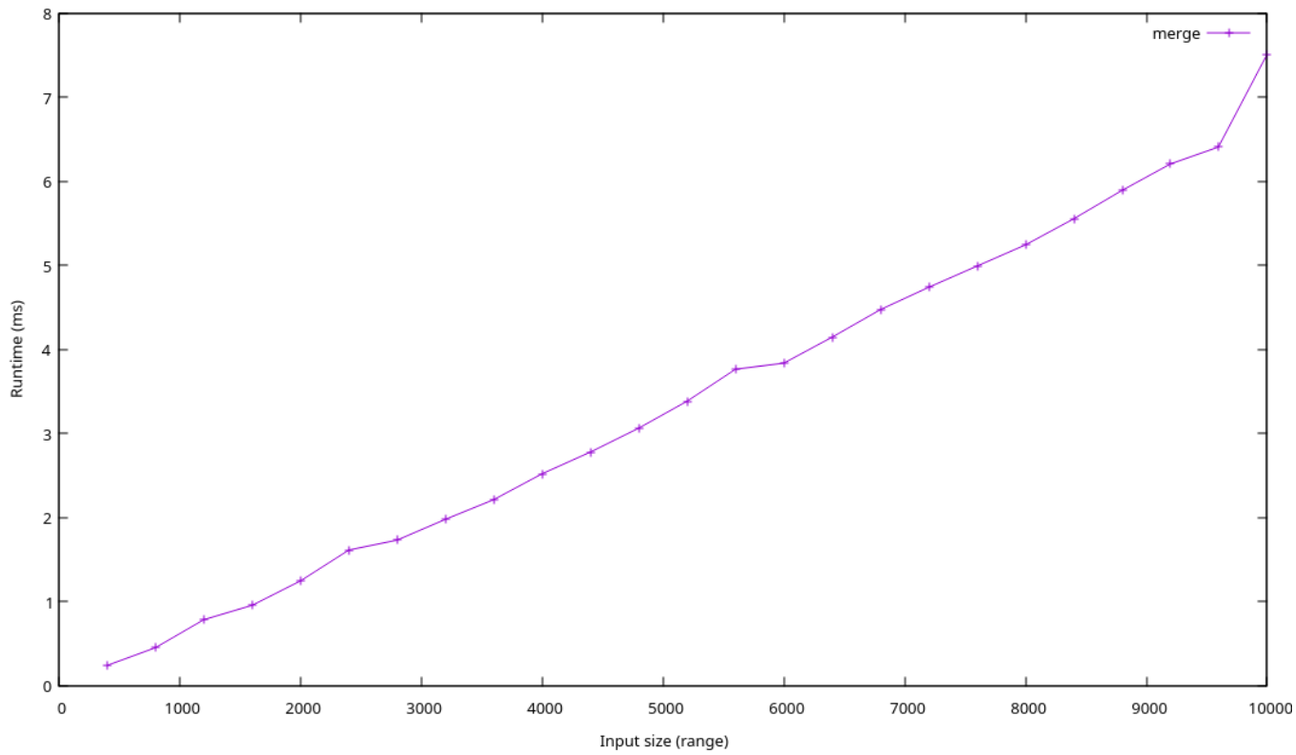


Figura 38: Análise de tempo para o quick sort, 50% ordenado

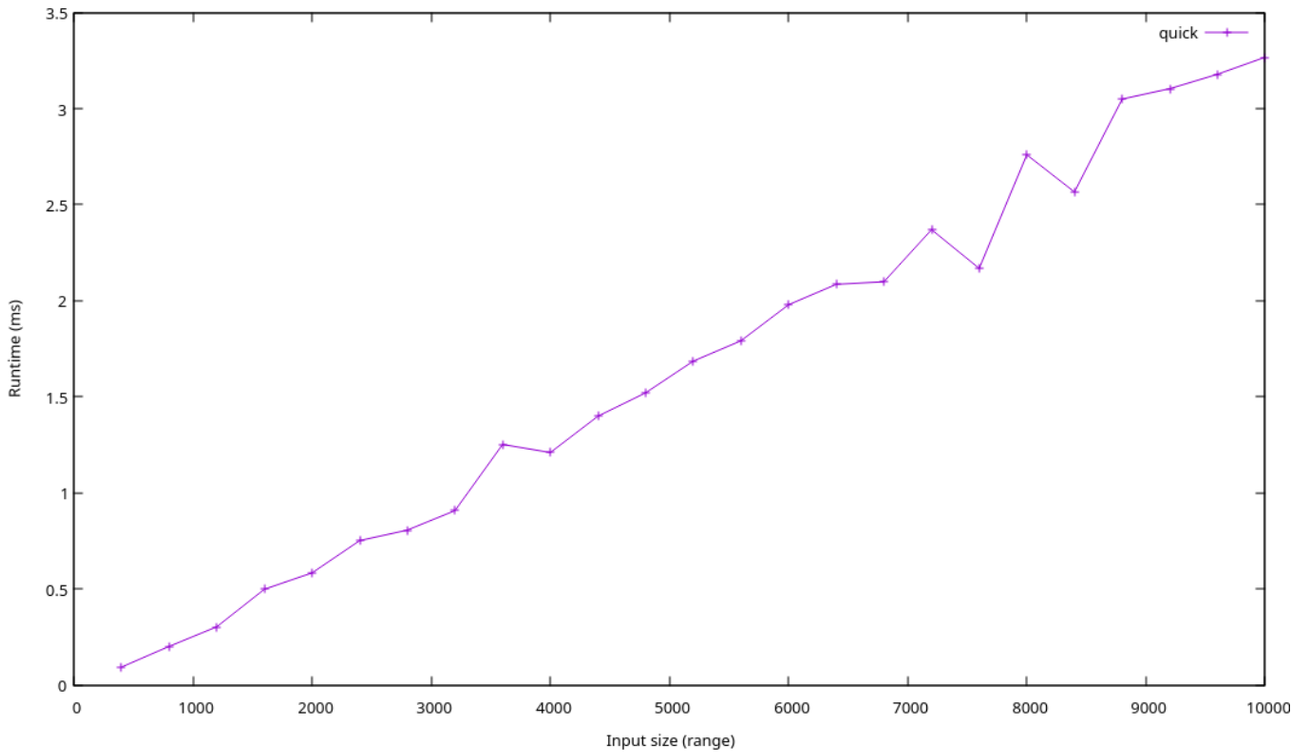


Figura 39: Análise de tempo para o radix sort, 50% ordenado

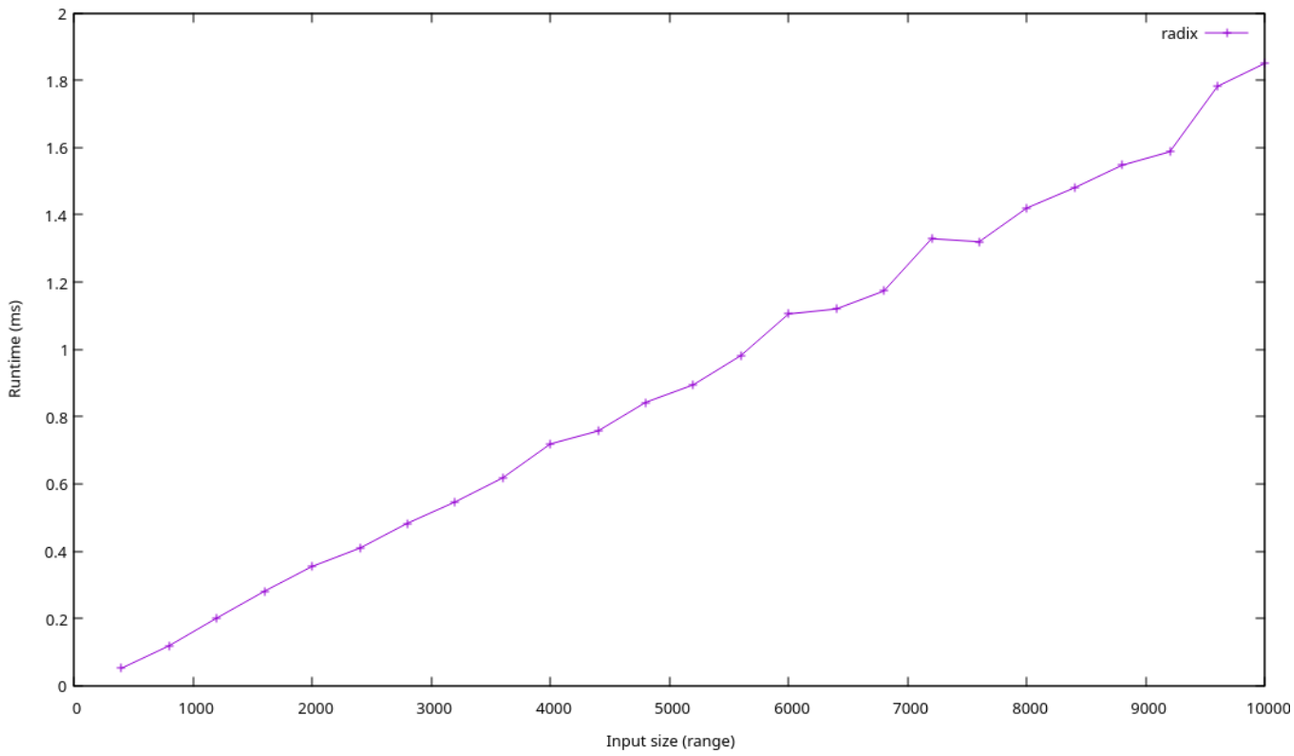
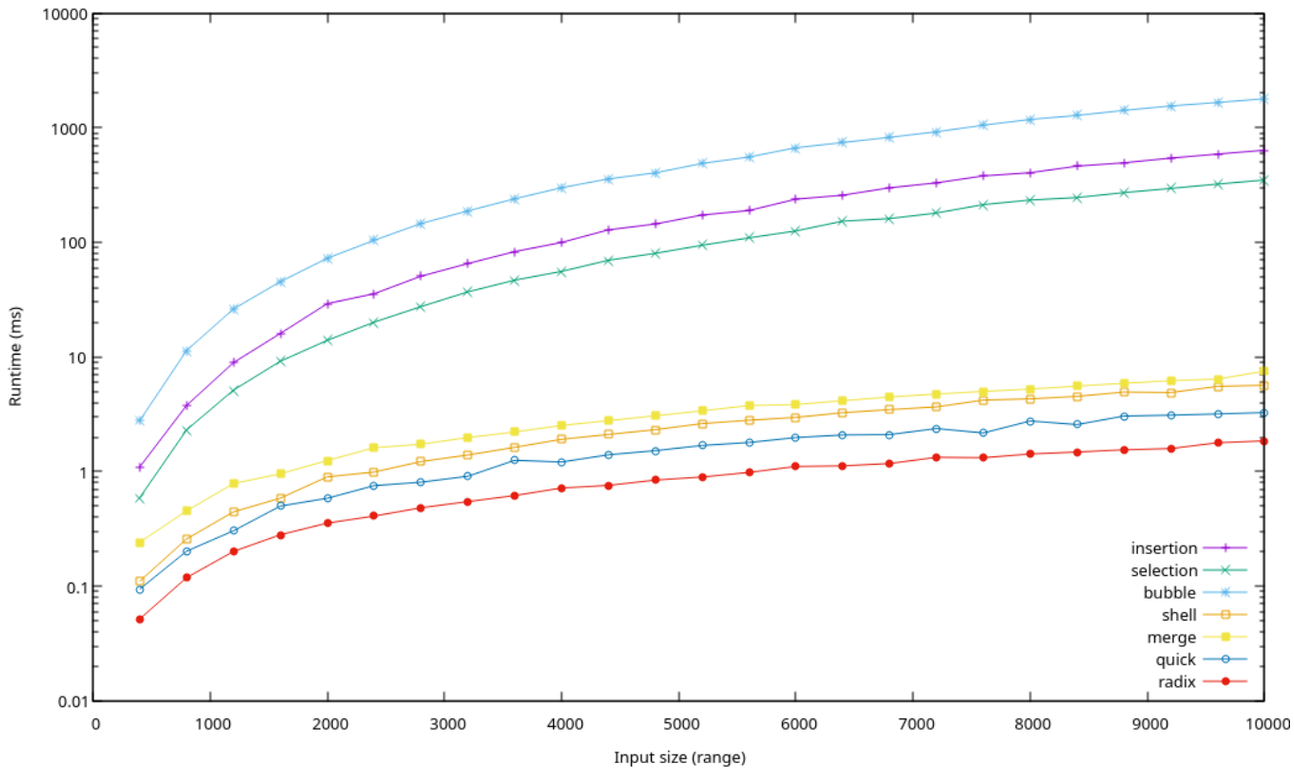


Figura 40: Análise de tempo de todos os algoritmos, 50% ordenado



3.6 Cenário 6

A tabela seguinte apresenta o desempenho dos 7 algoritmos no cenário em que os elementos do vetor estão em **ordenados 25% dos elementos**.

Tamanho da entrada	Tempo (ms)						
	Insertion	Selection	Bubble	Shell	Merge	Quick	Radix
400	1.4784	0.58376	3.24444	0.1498	0.24782	0.07596	0.05522
800	5.5331	2.34534	12.7329	0.29448	0.46062	0.18258	0.10562
1200	13.285	5.1972	28.7808	0.43376	0.72288	0.27924	0.20474
1600	23.3658	8.94498	52.098	0.65184	1.00246	0.40038	0.27498
2000	35.7381	13.9495	77.4253	0.82768	1.26058	0.47714	0.37432
2400	51.0427	20.2973	115.648	1.035	1.49274	0.6227	0.40938
2800	68.1735	27.4721	156.902	1.21436	1.77354	0.73936	0.5205
3200	89.6724	38.7944	198.354	1.52076	2.03552	0.86388	0.55228
3600	125.982	45.3308	261.75	1.69286	2.29458	0.93926	0.61544
4000	148.747	56.5284	319.053	1.94714	2.59022	1.12522	0.7138
4400	180.362	70.9485	410.712	2.19938	3.37294	1.29488	0.76688
4800	202.327	84.4147	464.152	2.34988	3.14928	1.36292	0.83046
5200	246.885	99.5552	543.109	2.69064	3.45922	1.55578	0.90354
5600	280.031	109.083	643.109	2.92636	3.70474	1.67686	0.96842
6000	326.19	125.69	712.404	3.10172	4.00982	1.68266	1.04392
6400	377.412	143.707	824.626	3.35088	4.25558	1.87378	1.11884
6800	427.354	161.161	935.148	3.60422	4.56814	2.0784	1.18466
7200	473.1	180.539	1038.22	3.92972	4.89432	2.10064	1.25148
7600	529.767	202.577	1160.15	4.24148	5.09992	2.21322	1.31998
8000	562.435	224.188	1253.23	4.56548	5.41814	2.30346	1.39366
8400	641.541	245.899	1408.58	4.60426	5.59958	2.66206	1.8800
8800	688.548	268.66	1563.99	4.97394	6.07336	2.63408	1.6638
9200	792.632	312.74	1728.53	5.33126	6.29396	2.68486	1.58148
9600	831.19	321.562	1872.26	5.28804	6.56652	2.82052	1.76284
10000	927.084	364.09	2042.94	5.9233	6.82856	3.03978	1.74984

Tabela 6: Tempo dos algoritmos em função do tamanho da entrada quando o vetor tem 25% dos elementos ordenados

Figura 41: Análise de tempo para o insertion sort, 25% ordenado

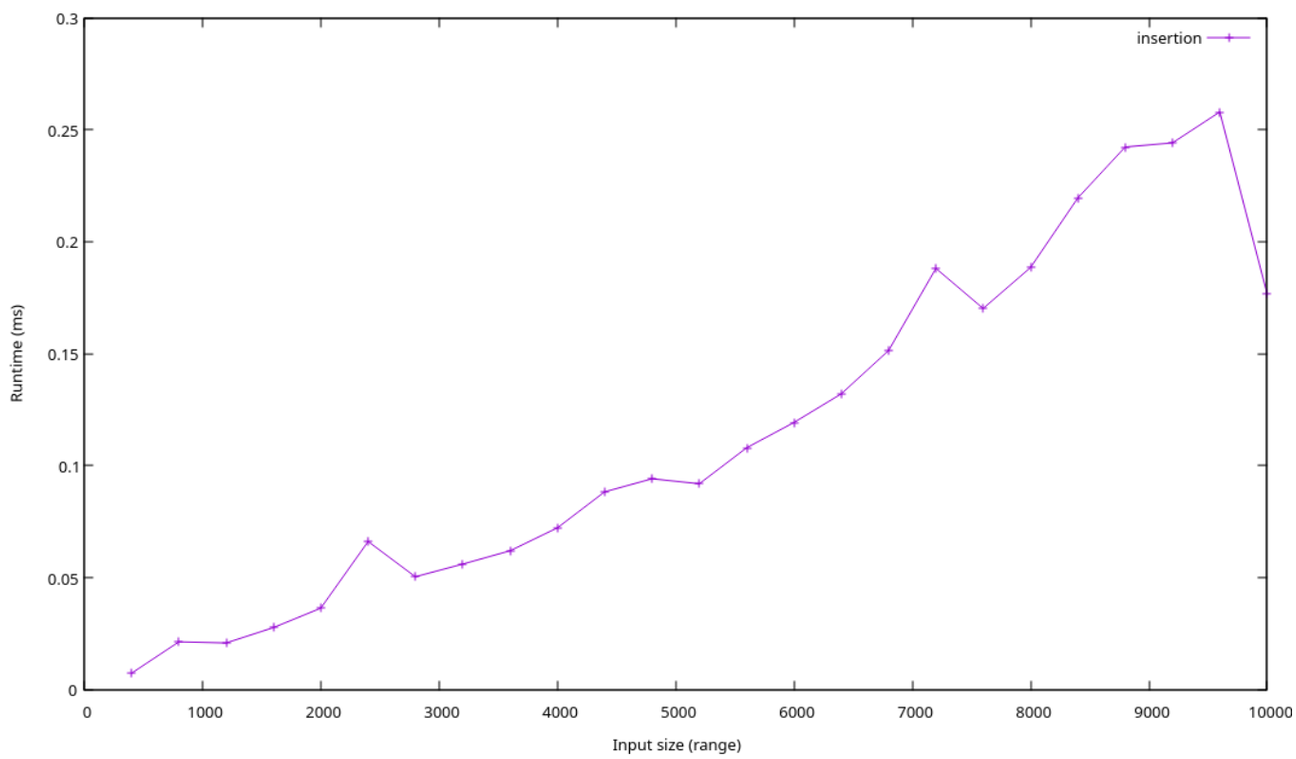


Figura 42: Análise de tempo para o selection sort, 25% ordenado

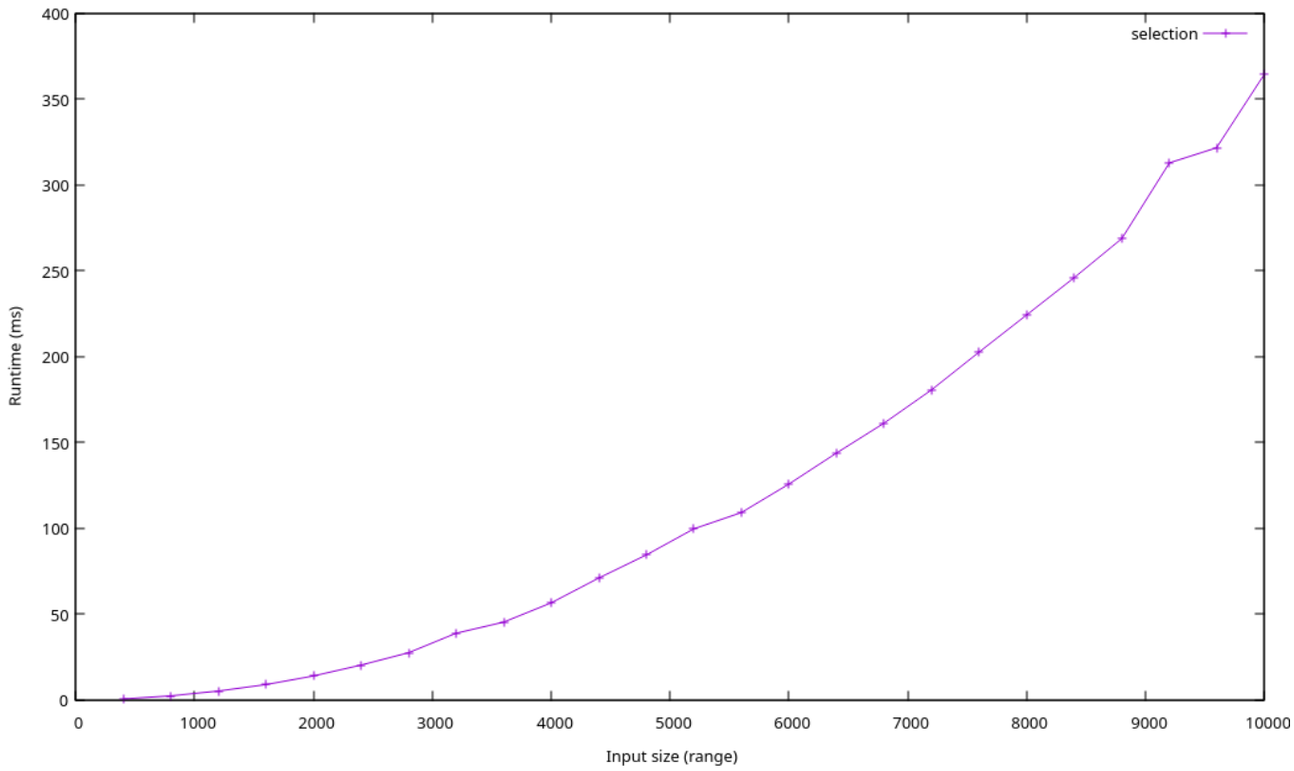


Figura 43: Análise de tempo para o bubble sort, 25% ordenado

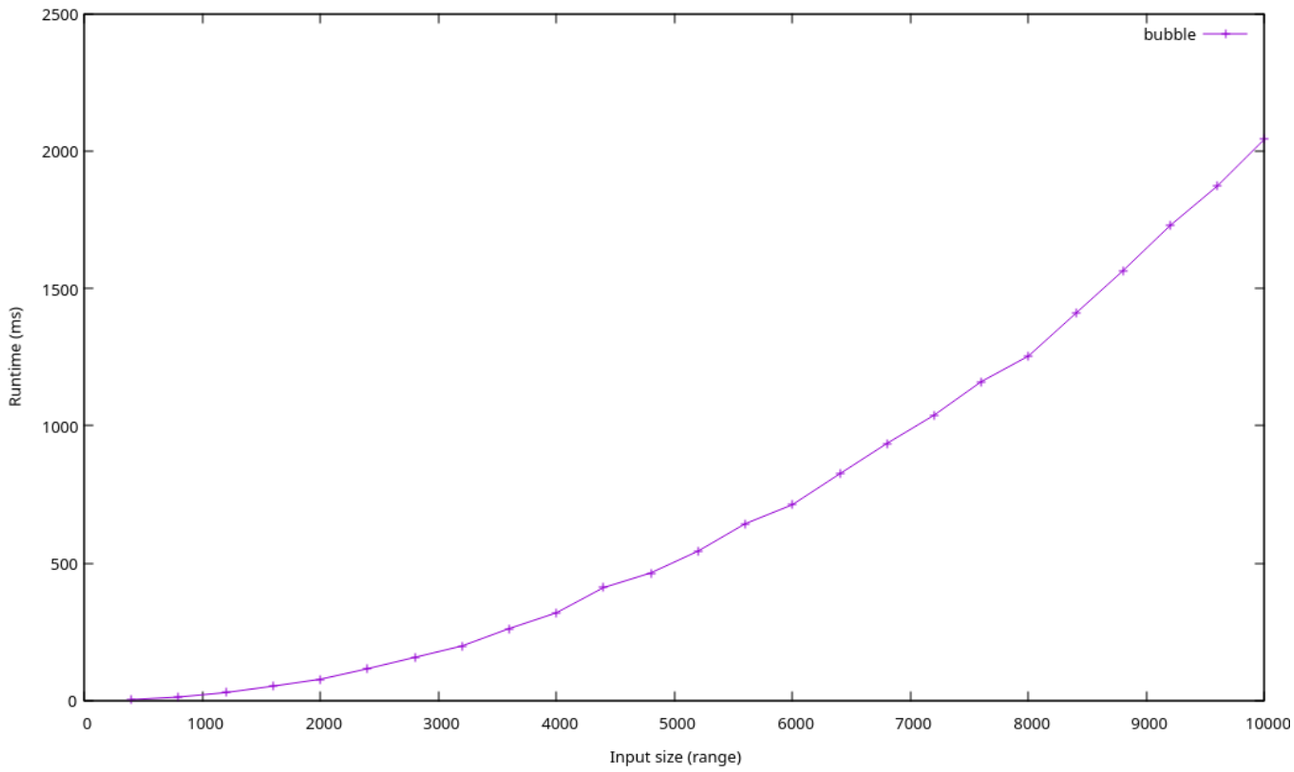


Figura 44: Análise de tempo para o shell sort, 25% ordenado

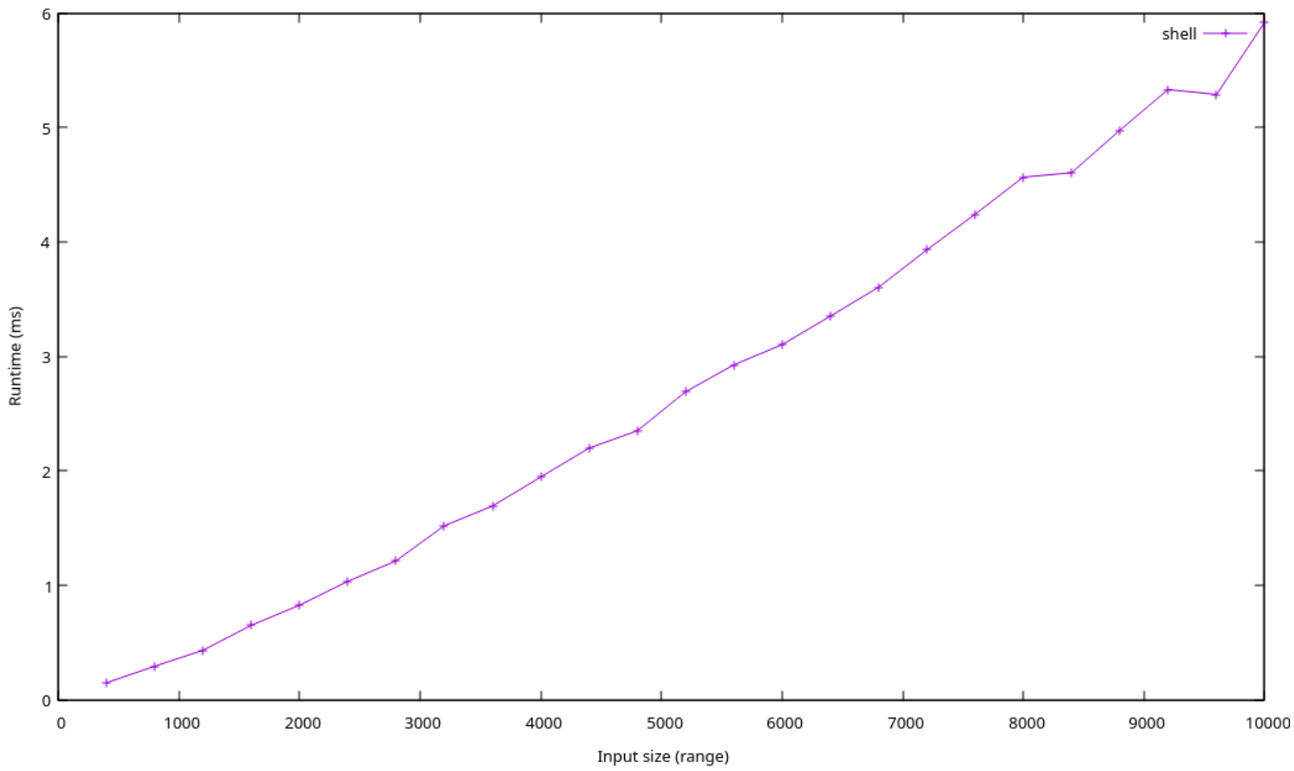


Figura 45: Análise de tempo para o merge sort, 25% ordenado

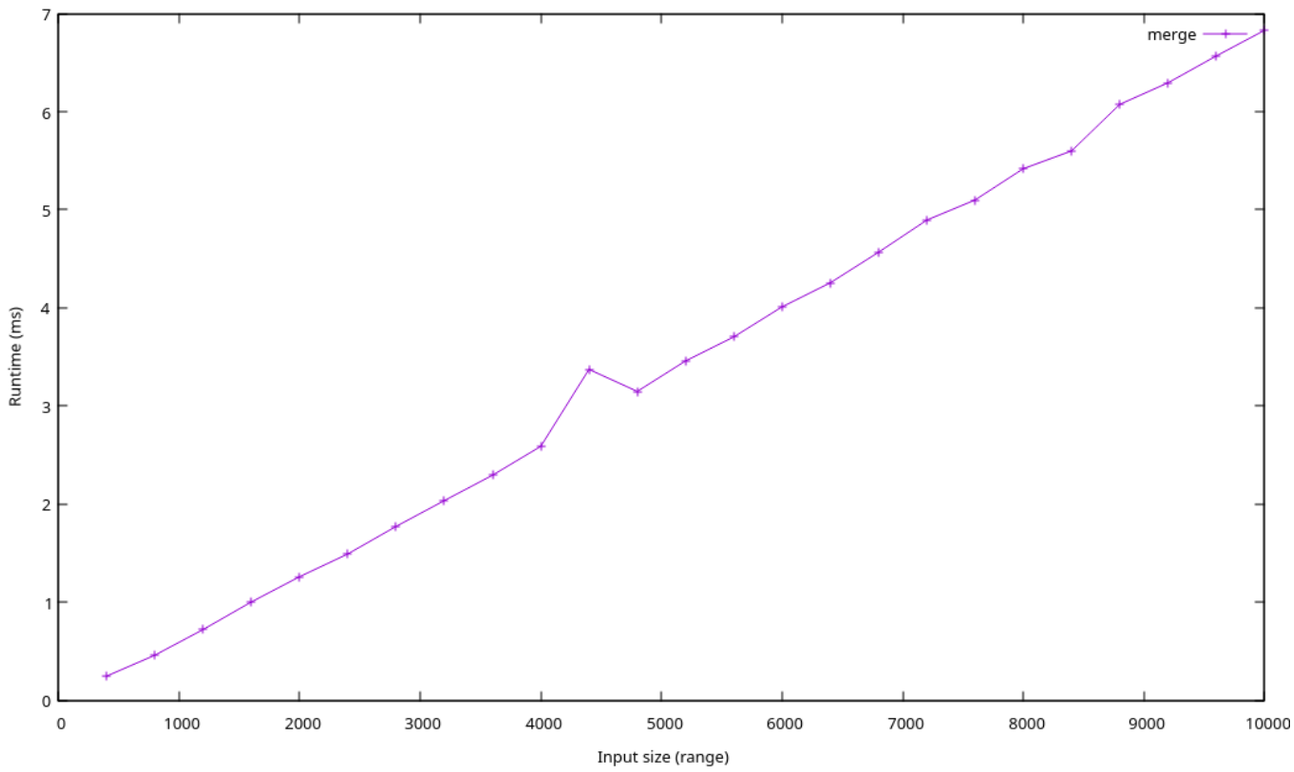


Figura 46: Análise de tempo para o quick sort, 25% ordenado

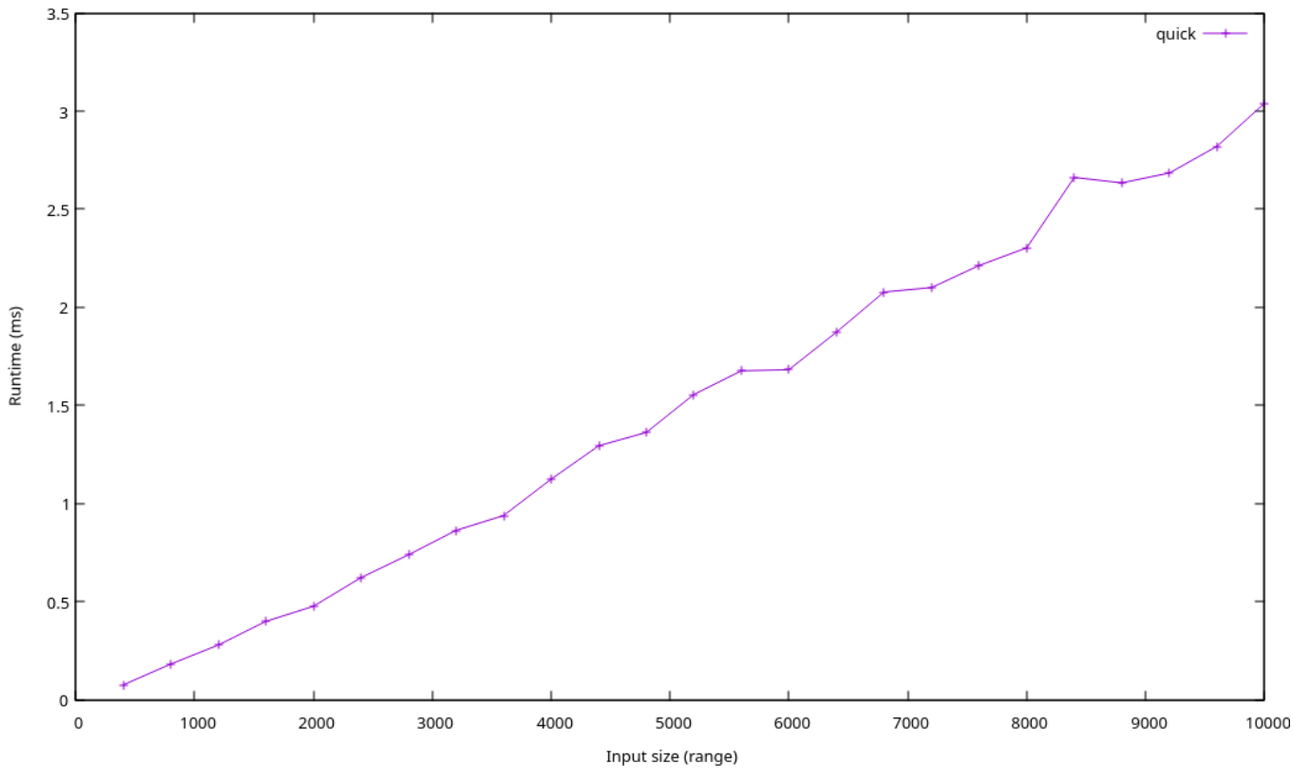


Figura 47: Análise de tempo para o radix sort, 25% ordenado

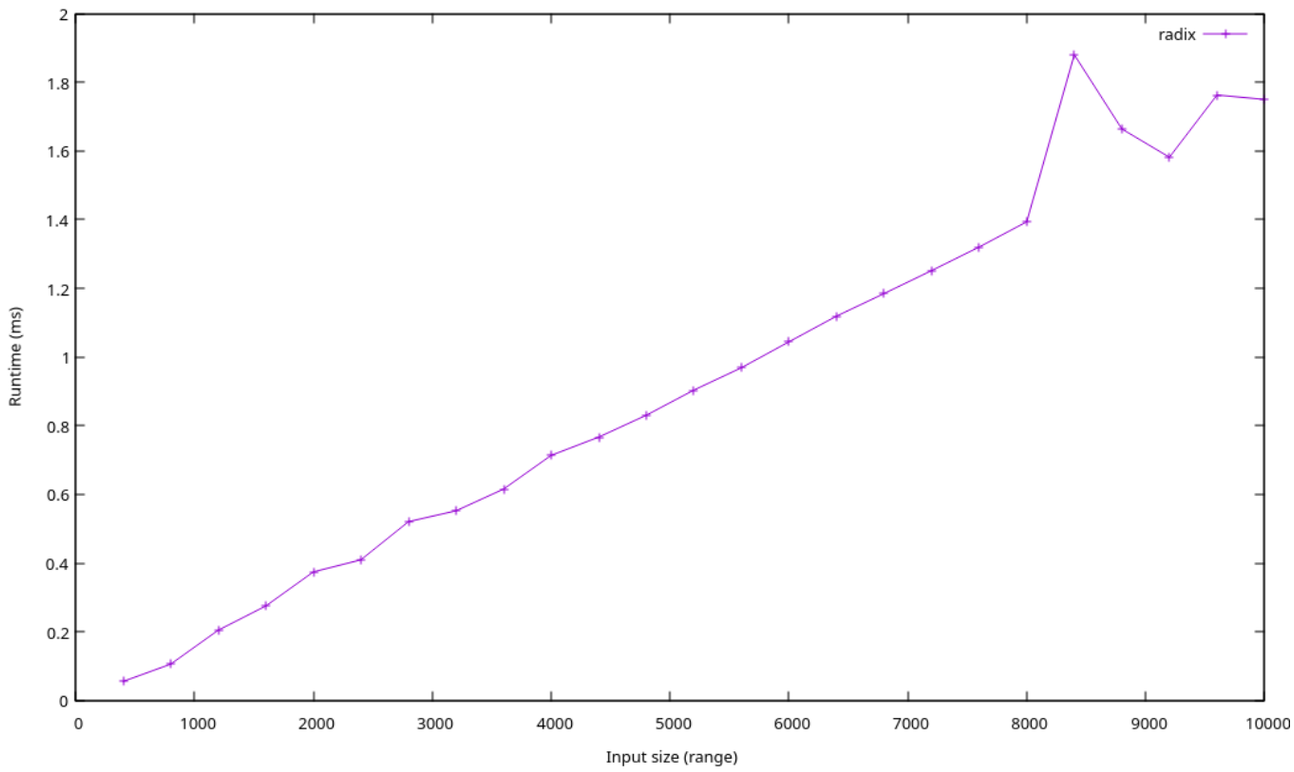
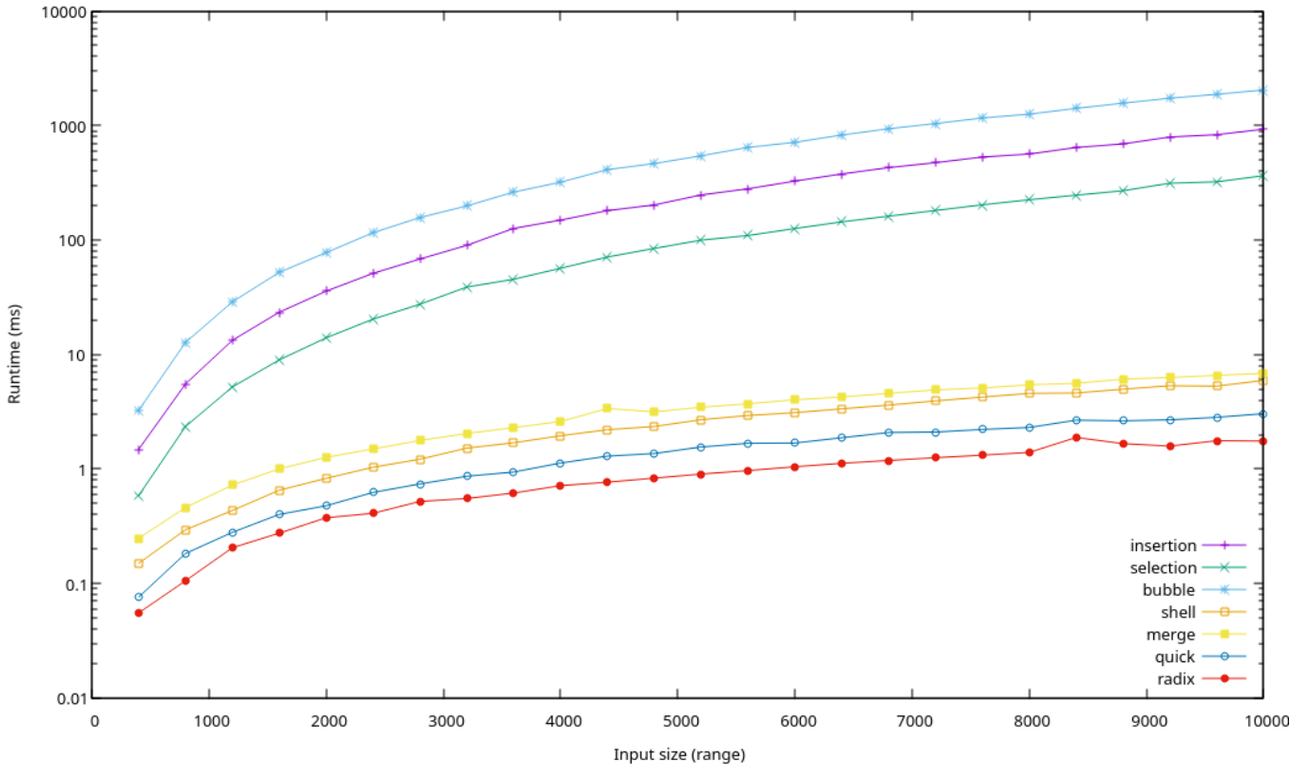


Figura 48: Análise de tempo de todos os algoritmos, 25% ordenado



4 Discussão

4.1 O que você descobriu de maneira geral?

Pudemos observar com essa análise que o Bubble Sort é sempre o pior dos algoritmos de ordenação. Por mais que implementemos sua versão otimizada, ela não terá um desempenho tão bom quanto os outros. Sua versão otimizada se destaca aos demais somente no caso em que o vetor está em ordem não decrescente, ou seja, ordenado. O insertion sort e o selection sort apresentaram um desempenho melhor que o bubble, porém, bem inferior aos demais.

Em relação aos melhores algoritmos, pôde-se observar que o radix foi superior aos demais em praticamente quase todos os cenários. No entanto, vale destacar o contexto em que foi realizado o teste: pelo fato dos elementos dos vetores serem os números dos índices, não tivemos números maiores que 10000, e com isso, o radix foi favorecido. O desempenho do radix piora quanto maior for o número máximo de dígitos, e como no teste realizado o limite para o número máximo de dígitos era 5, isso favoreceu com que o radix tivesse um desempenho superior. Mas, isso não tira o mérito de o radix ser um algoritmo bastante rápido.

Tendo o bubble como o pior algoritmo e o radix como o melhor (em relação ao teste discutido neste relatório), é interessante observar o desempenho do merge, quick e shell sort. O shell sort quando implementado com seu *gap* clássico, ele tende a apresentar complexidade temporal pior do que o merge e o quick. Porém, como foi usado o *gap* da implementação do shell sort de Donald Ervin Knuth, 1973, baseado em Vaughan Pratt, 1971, a eficiência acabou sendo bem melhor. Diante disso, tivemos o quick sort e o shell sort tendo desempenhos próximos, mas com uma leve vantagem do quick e o merge vindo logo atrás do shell.

4.2 Quais algoritmos são recomendados para quais cenários?

Algoritmos que necessitaram de um tempo maior para realizarem a ordenação, tais como bubble sort, insertion sort ou selection sort, são úteis para cenários que não é preciso ordenar muitos elementos ou que o tempo não é problema. Ademais, esses três algoritmos não necessitam de alocação dinâmica, sendo um ponto interessante caso a complexidade espacial seja algo a se preocupar.

Se complexidade espacial é um problema, shell sort deve ser a melhor solução. Isso porque é um dos que apresentaram melhor desempenho e sua implementação não realiza alocação dinâmica. O radix, merge e quick sort, todos utilizam mais da memória do que os demais. O radix é o que menos usa memória na maioria dos casos, já que ele irá usar memória a mais somente para alocar os *buckets* no ínicio do algoritmo. Desse modo, ele é recomendado caso os números contidos no vetor não sejam muito grandes (ou seja, não tenham muitos dígitos).

Quick sort e merge sort apresentaram grande eficiência nos testes, sendo um dos melhores. No entanto, ambos trabalham com mais memória do que os demais algoritmos. No caso do quick sort, por ser um algoritmo recursivo, ele irá necessitar de uma considerável pilha de execução, mas nem sempre isso é possível. Por isso, o quick sort por vezes é tido como instável em algumas implementações. O merge sort também tem problema com memória devido às constantes alocações dinâmicas que ele gera.

4.3 Como o algoritmo de decomposição de chave (radix) se compara com o melhor algoritmo baseado em comparação de chaves?

O radix apresenta melhor desempenho que o melhor algoritmo baseado em comparação de chaves (quick sort) no teste realizado. O radix não foi infinitamente superior, mas teve boa vantagem em relação ao quick sort.

A comparação que podemos fazer é que o quick sort obteve complexidade temporal e

espacial pior que o radix. No entanto, vale salientar o contexto do teste, o número máximo de dígitos testado foi baixo (5), então se colocássemos um limite maior, ele teria uma complexidade temporal pior, mas ainda assim obteria uma boa eficiência. Também vale destacar que o pior caso do quick sort (vetor já ordenado) foi bem pior que o pior caso do radix.

4.4 É verdade que o quick sort, na prática, é mesmo mais rápido que o merge sort?

Sim, de fato o quick sort obteve um desempenho melhor do que o merge sort em relação ao tempo.

4.5 Aconteceu algo inesperado nas medições? (por exemplo, picos ou vales nos gráficos). Se sim, por que?

Sim, em alguns testes ouve picos e vales nos gráficos. É algo inesperado porque tempo é diretamente proporcional a quantidade de elementos, ou seja, quanto mais elementos maior o tempo que se leva para executar o algoritmo.

O motivo da presença de picos e vales se deve provavelmente pela atuação do sistema operacional Windows 10 durante o teste. Observa-se que esse comportamento inesperado foi mais presente quando o número de elementos foi ficando maior, podendo-se conjecturar que essa variação foi devido a maior instabilidade do consumo de memória do SO naquele momento.

4.6 A análise empírica é compatível com a análise matemática?

Sim, note que os algoritmos *insertion sort*, *selection sort* e *bubble sort* todos são $O\left(n^2\right)$ no meio e pior caso, e isso se reflete no comportamento do gráfico, que é uma parábola. Vale salientar que no caso em que o vetor já está ordenado, o bubble sort não apresenta um comportamento $O\left(n^2\right)$ devido a sua implementação otimizada.

Algoritmos como o *quick sort* e *merge sort* de acordo com a análise matemática são $O\left(n \log n\right)$ em seu caso médio. Ao analisar o gráfico podemos perceber que o crescimento desses algoritmos são bastante devagar, e apresentam uma curva logarítmica. O quick sort no caso em que o vetor já está ordenado ele apresenta um comportamento pior, pois ele fará todas as chamadas recursivas possíveis, apresentando comportamento $O\left(n^2\right)$. Ao analisar o gráfico é possível perceber um comportamento de parábola no quick sort. O merge sort apresenta um desempeno pior nesse cenário também, porém, ainda preserva seu comportamento $O\left(n \log n\right)$.

O *shell sort* é outro algoritmo que apresenta tempo de complexidade $O\left(n \log n\right)$ pela análise matemática no caso médio e no pior caso apresenta $O\left(n^2\right)$. No entanto, isso vale para a implementação do *shell sort* clássico, em que o *gap* é a metade dos elementos, e implementamos usando o *gap* em que o pior caso é $O\left(n^{\frac{3}{2}}\right)$ (o *gap* sendo um terço dos elementos). Por isso, o *shell sort* acabou superando em alguns casos algoritmos como o merge sort e quick sort.

O *radix sort* acabou sendo o com melhor desempenho, sua complexidade é $O\left((n+b) \cdot D\right)$, onde b é a base do sistema que estamos lidando (decimal, binário, ...) e $D = \log_b(k)$, em que k é o número máximo de dígitos que estamos tratando. Dessa forma, como o teste foi realizado com o número máximo de dígitos baixa, o radix acabou tendo um desempenho superior aos decimais, mas como dito antes, se esse limite fosse maior ele acabaria perdendo para os outros.

Referências

- Playlist sorting algorithms mycodeschool
- Vídeo com explicação visual para o Radix Sort
- Merge sort no site geeksforgeeks
- Radix sort no site geeksforgeeks
- Shell sort na Wikipédia em inglês