Instituto Nacional de Matemática Pura e Aplicada

Aluno: Hallison da Paz

Curso de Algoritmos
Trabalho 1 - Ordenação

Rio de Janeiro, setembro de 2015.

1 Implemente os algoritmos estudados em aula (seleção, inserção, mergesort, quicksort, heapsort).

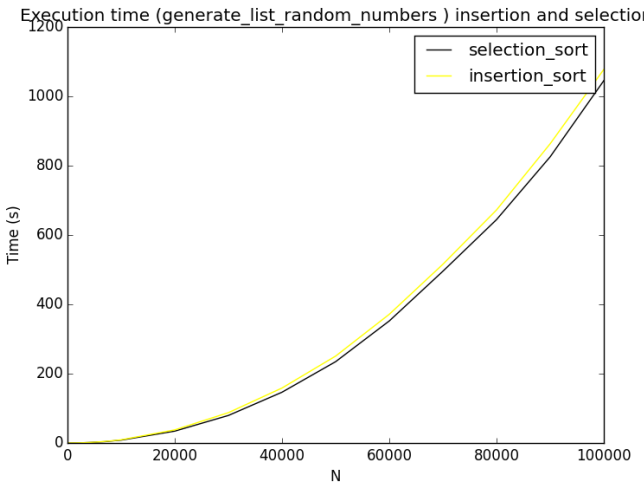
Algoritmos estão implementados no arquivo ***Sorting.py***. Foi utilizada a linguagem de programação Python 3.4 [1]. Por utilizar alguns recursos específicos das versões 3.x da linguagem Python, este programa não é compatível (não executará) com as versões 2.x desta linguagem. Todo o código assim como os resultados completos obtidos podem ser encontrados em [2].

2 Compare o esforço necessário para a implementação correta de cada algoritmo.

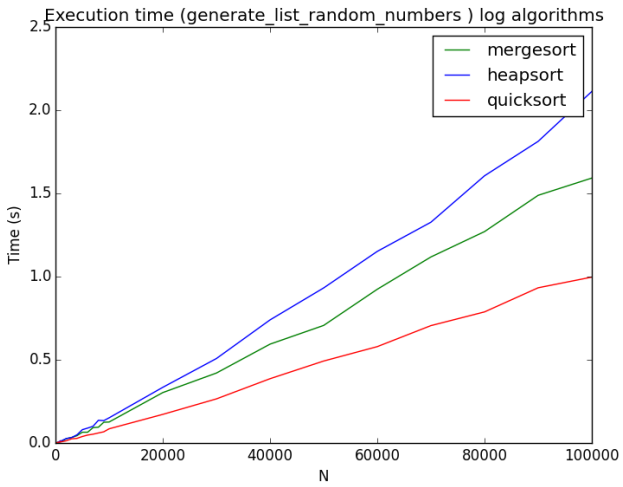
Os algoritmos *selection sort* e *insertion sort* demandaram um baixo esforço de implementação, visto que a ideia base por trás desses dois algoritmos é bem intuitiva para o raciocínio humano e eles não requerem manipulações complicadas de índices. A ideia do *merge sort* é mais elaborada que a dos dois algoritmos anteriores, requerindo um esforço um pouco maior. Embora a estratégia dividir para conquistar não seja tão imediata quanto as abordagens diretas dos outros algoritmos, o procedimento de *merge* não é tão complicado de realizar, visto que conseguimos pensar nele com uma analogia à junção ordenada de dois blocos de cartas ordenadas. Entre *quicksort* e o *heapsort*, considero o primeiro aquele que demanda maior esforço para a implementação correta. Embora o heapsort tenha como base uma outra estrutura de dados (heap), uma vez entendida sua propriedade (relação entre os elementos pai e filhos), não é tão difícil codificar o procedimento que atua sobre os elementos para manter esta propriedade. Com este procedimento implementado corretamente, o algoritmo heapsort em si fica bastante simples. Quanto ao quicksort, a própria ideia de separar os elementos menores que e maiores que um elemento pivô em tempo linear já é um procedimento de maior esforço.

3 Compare o desempenho dos algoritmos em sequências de números inteiros geradas aleatoriamente. Use sequências de 100 até 100000 números. (E mais se for possível.) Meça e compare os tempos de cada algoritmo.

As figuras 1a e 1b ilustram as curvas obtidas separando os algoritmos de seleção e inserção, que executam em tempo $O(n^2)$ dos demais, que executam em $O(nlogn)$ (desconsiderando o pior caso do quicksort, que não ocorreu). Percebe-se que os algoritmos *selection sort* e *insertion sort* demandam muito mais tempo que os demais, o que distorceria a escala de comparação se fossem colocados junto a eles. Estes algoritmos levaram mais de 1000 segundos para executar sobre uma entrada de tamanho 100000, enquanto o mergesort, heapsort e quicksort levaram algo em torno de 2 segundos.



(a) Algoritmos de seleção e inserção



(b) *Quicksort, mergesort e heapsort*

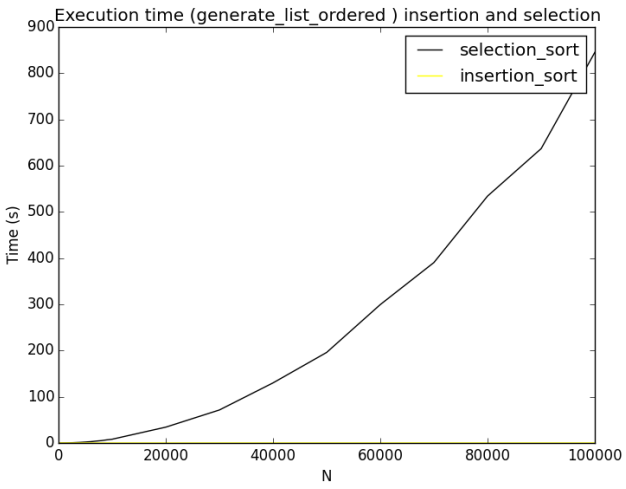
Figure 1: Desempenho dos algorimos por classes em sequências aleatórias

Embora o algoritmo de inserção, ao contrário do de seleção, leve em conta a forma da entrada, podendo realizar menos operações, verificou-se nos testes com sequências aleatórias que o *insertion sort* levou mais tempo para terminar do que o *selection sort*. Uma possível explicação para este fato pode ser a maior quantidade de operações de escrita em variáveis realizadas pelo *insertion sort,* que são mais custosas em tempo que as comparações.

4 Repita os testes com sequências especiais: ordenadas em ordem crescente, ordenadas em ordem descrescente, com muitas repetições, com poucas repetições.

A figura 2 ilustra o desempenho dos algoritmos sobre sequências ordenadas de forma crescente. Como esperado, o *insertion sort* apresentou o melhor desempenho sobre esta entrada particular, com eficiência O(n), enquanto os demais algoritmos não apresentaram diferenças expressivas de desempenho (pequenas flutuações de tempo são normais devido à gerência dos recursos computacionais pelo sistema operacional).

Para o algoritmo *quicksort*, se escolhermos como pivô sempre o último ou primeiro elemento dos subvetores de cada chamada recursiva, teremos um *overflow* da pilha de execução, pois cada nova chamada recursiva atuará sobre um subvetor de tamanho 1 unidade menor que o da chamada anterior, gerando um consumo de espaço O(n). Por conta disso, a cada passo, o pivô foi escolhido aleatoriamente por meio de um procedimento auxiliar que realiza a troca entre o último elemento do subvetor e um outro elemento em uma posição determinada aleatoriamente neste subvetor. Assim, o caso ordenado deixa de ser o pior caso de execução do quicksort.

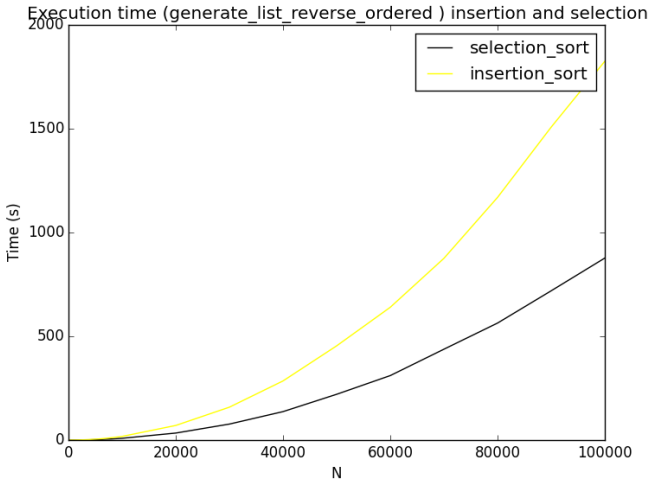


(a) Selection sort e Insertion Sort

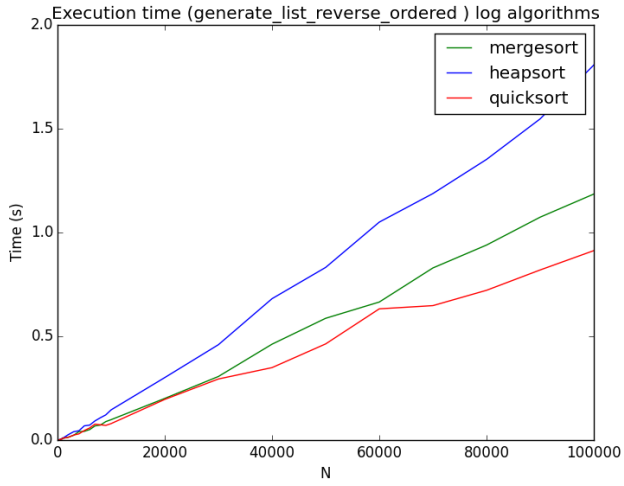
(b) Quicksort, mergesort e heapsort

Figure 2: Desempenho dos algoritmos em sequências ordenadas em ordem crescente

A figura 3 ilustra o desempenho dos algoritmos sobre sequências ordenadas de forma decrescente. Na comparação entre *insertion sort* e *selection sort*, vemos que o desempenho do *insertion sort* piorou consideravelmente, demandando muito mais tempo que o selection sort realizar a ordenação. Quanto aos demais algoritmos, não houve mudanças significativas.



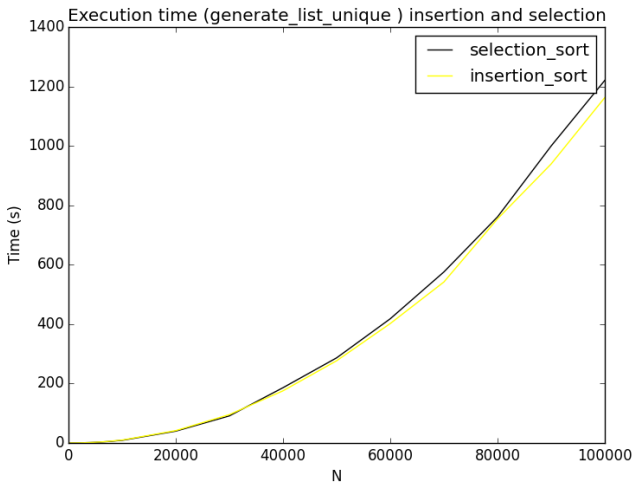
(a) Selection sort e Insertion Sort



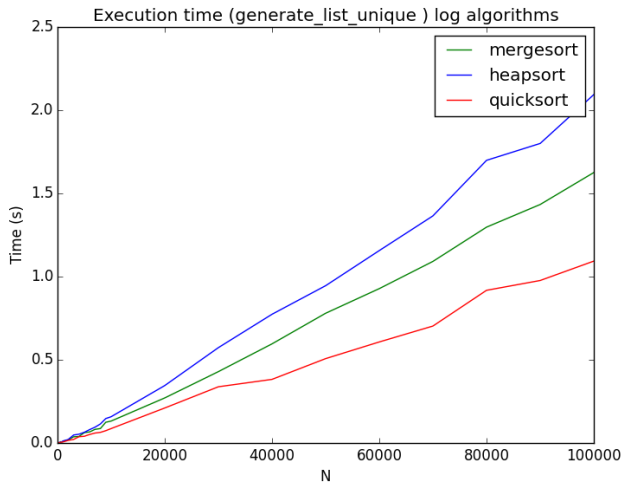
(b) Quicksort, mergesort e heapsort

Figure 3: Desempenho dos algoritmos em sequências ordenadas em ordem decrescente

A figura 4 apresenta as curvas obtidas para sequências com todos os elementos únicos. Desta vez, o insertion sort apresenta um ligeiro ganho de desempenho em relação ao selection sort.



(a) Selection sort e insertion sort



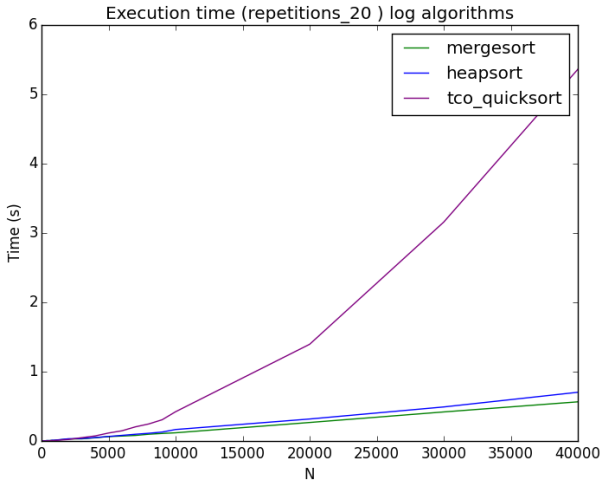
(b) Merge, Heap e Quick sorts

Figure 4: Sequências com todos os elementos únicos

A figura 5 ilustra o caso em que a entrada apresenta poucas repetições (20%), enquanto a figura 6 ilustra o caso com muitas repetições. Percebe-se que com o aumento do número de elementos repetidos, o desempenho do *insertion sort* varia sensivelmente, terminando cada vez mais rápido.

Infelizmente, esta entrada especial é um caso patológico para o algoritmo quicksort como o implementamos. Mesmo que o elemento pivô não esteja mais fixo, evitando overflow em sequências ordenadas (crescentes ou decrescentes), a inserção de muitos elementos repetidos ocasiona um número de chamadas recursivas alto, que no pior caso pode ser $O(n)$. Para resolver este problema, implementou-se uma versão do quicksort em que chama-se apenas uma recursão no menor subvetor e utiliza-se de uma iteração para reaproveitar o mesmo ponto da pilha, assim o consumo de memória do quicksort fica em $O(logn)$. Percebe-se a natureza quadrática do pior caso do quicksort, que com cada vez mais repetições na entrada, fica mais próximo dos algoritmos de seleção e inserção do que do mergesort e heapsort.

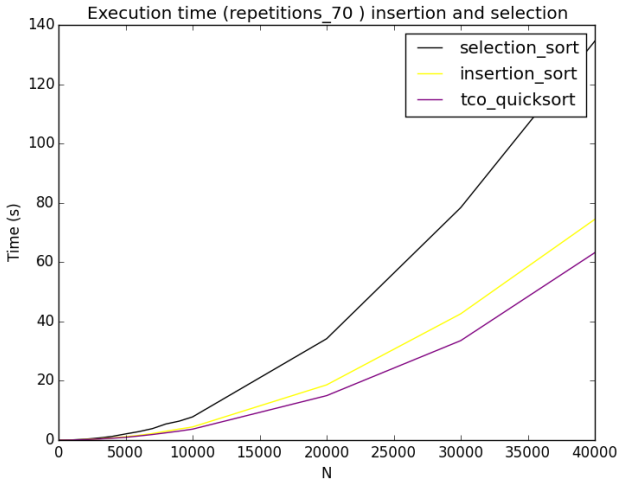
(a) Selection e insertion sort



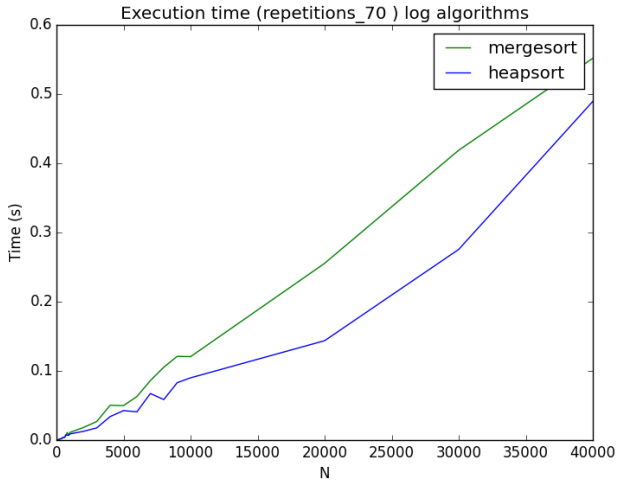
(b) Mergesort, heapsort e quicksort

Figure 5: Desempenho dos algoritmos em sequências com 20% de repetições

Quanto aos demais algoritmos, vemos que o desempenho do heapsort supera o do mergesort para o caso em que há muitos elementos repetidos (com 50% de repetição as curvas já são bastante próximas e com 60%, o heapsort já supera o mergesort). De fato, o mergesort sempre realiza a divisão em subproblemas até o último nível, independente da entrada, enquanto a flutuação de elementos no heap pode ser menos custosa.



(a) Insertion, selection e quick sort



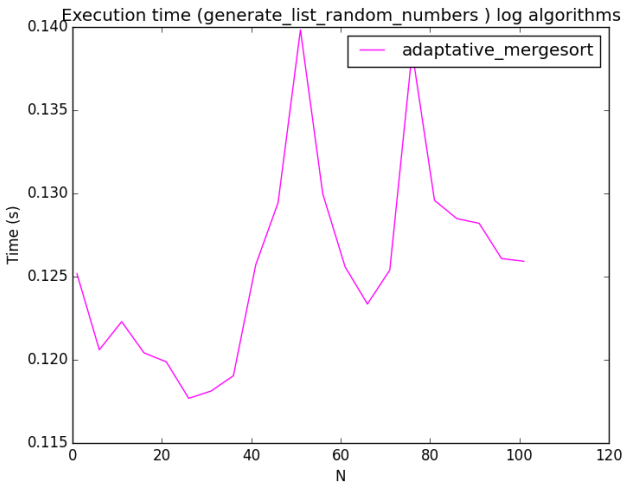
(b) Mergesort e heapsort

Figure 6: Desempenho dos algoritmos em sequências com 70% de elementos repetidos

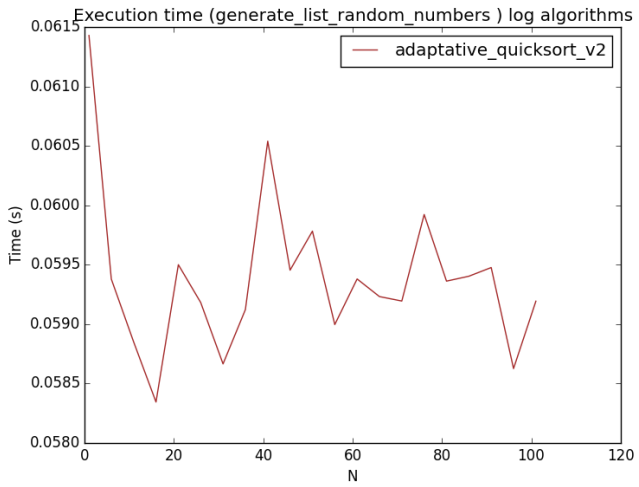
5 No caso de mergesort e quicksort, interrompa a recursão quando o subproblema for pequeno e execute ordenação por inserção em cada subproblema. No caso de quicksort, faça o mesmo executando ordenação por inserção uma única vez, ao final do processo. Vale a pena fazer essas modificações? A partir de quando? Para qual definição de pequeno?

A figura 7 ilustra o desempenho dos algoritmos adaptativos quando a recursão é interrompida em diversos pontos entre 1 e 101. Para a comparação entre os algoritmos, utilizou-se uma mesma sequência com 10000 elementos gerada aleatoriamente.

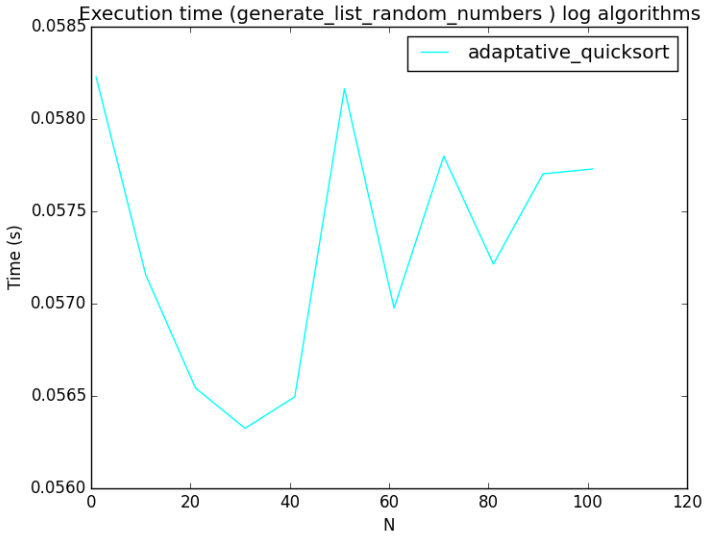
Observa-se que em alguns casos vale a pena realizar estas modificações, pois o overhead embutido pela estratégia dividir para conquistar dos algoritmos *merge sort* e *quicksort* fazem com que a complexidade destes algoritmos esconda uma constante que os torna menos eficientes que o *insertion sort* para valores pequenos. Particularmente, observou-se que para valores menores que 40 elementos, o desempenho do mergesort adaptativo foi superior ao original. Já no caso do quicksort, na implementação com chamadas ao insertion sort em cada subvetor, o desempenho é superior mesmo para subvetores de tamanho pouco superior a 100 elementos. A implementação adaptativa que realiza uma única chamada ao *insertion sort* obteve ganhos de desempenho até uma faixa pouco superior aos 40 elementos.



(a) Mergesort adaptativo



(b) Quicksort adaptativo (duas chamadas ao insertion sort)

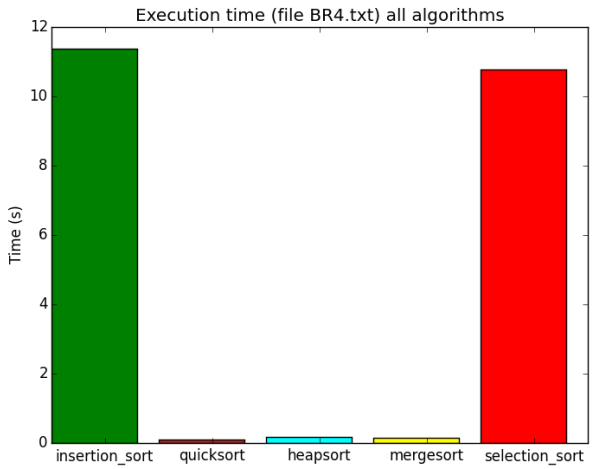


(c) Quicksort adaptativo (uma chamada ao insertion sort)

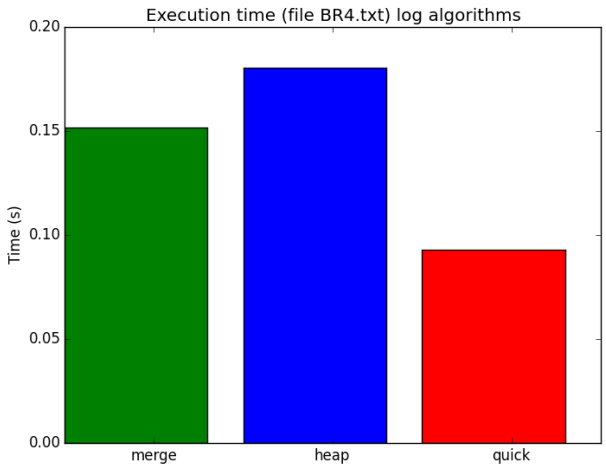
Figure 7: Desempenho dos algoritmos adaptativos

6 Adapte os seus programas para ordenar palavras. Teste o desempenho de cada algoritmo no arquivo BR4.txt contendo 10000 palavras e no arquivo BR5.txt contendo 100000 palavras. Houve mudança do desempenho relativo dos algoritmos agora que comparar valores é mais complicado?

As figuras 8 e 9 ilustram o resultado obtido para o desempenho desses algoritmos sobre as palavras contidas nos arquivos de teste.

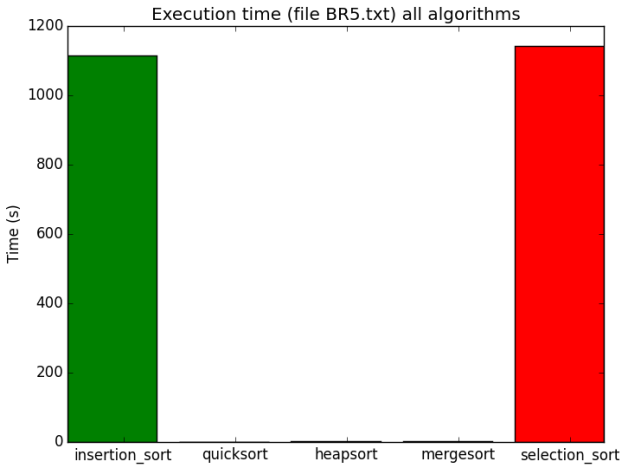


(a) Todos os algoritmos

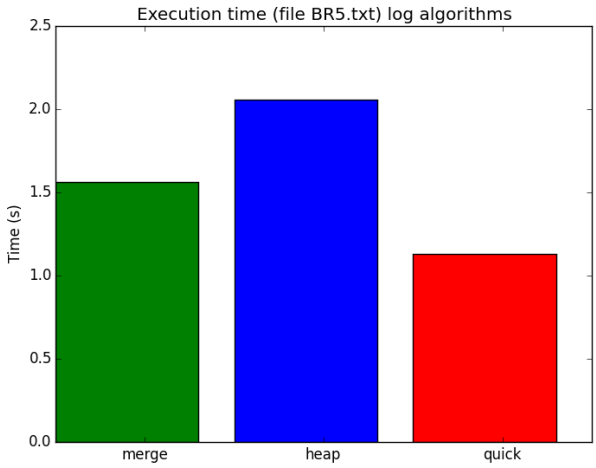


(b) Apenas merge, heap e quicksort

Figure 8: Desempenho na ordenação com palavras BR4.txt



(a) Todos os algoritmos



(b) Apenas merge, heap e quicksort

Figure 9: Desempenho na ordenação com palavras BR4.txt

Pode-se observar que não houve mudança no desempenho relativo dos algoritmos. O fato de que é mais complicado comparar palavras do que números inteiros tem influência apenas sobre o tempo total de execução dos algoritmos, que pode aumentar um pouco; no entanto, relativamente um ao outro temos uma ordem de comparação similar.

References

[1] Python; [internet] Disponível em: <https://www.python.org/> [acesso em 12 de setembro de 2015]

[2] hallpaz Github; [internet] Disponível em <https://github.com/hallpaz/Algorithms-Course-IMPA-2015/tree/master/assignments/sorting> [acesso em 22 de setembro de 2015]