Projeto e Análise de Algoritmos
Relatório DataGrid

Autores: Cristiano Larréa, Felipe Lamarca, e Paloma Borges
Docente: Thiago Pinheiro de Araújo

Escola de Matemática Aplicada
FGV EMAp

Rio de Janeiro
2023.2

Sumário

1 Introdução

O objetivo deste trabalho é escrever um módulo que implemente a lógica de negócio de um datagrid utilizando algoritmos estudados em aula.

2 Decisões de projeto

2.1 Estrutura de dados base

A modelagem geral do projeto consiste na implementação de um datagrid, utilizando a estrutura de HashTable de endereçamento aberto, para armazenar e acessar eficientemente os eventos. Cada evento é representado por uma instância da classe `Event`, e o datagrid é representado pela classe `DataGrid`, que contém métodos para inserção, exclusão, busca, ordenação e seleção de eventos.

2.2 Tabelas Hash

A escolha das HashTables se baseia na facilidade em acessar dados por meio de uma função de espalhamento, que associa uma determinada chave a um índice da tabela hash. Além de a implementação de uma tabela hash de endereçamento aberto[1] ser relativamente simples, o fato de que cada `Event` é identificado por um `id` único, conforme a descrição do trabalho, sugeriu que essa seria uma implementação razoável, permitindo que a inserção, busca e deleção ocorressem de maneira eficiente, principalmente no caso do campo `id`. Como veremos nos resultados, mesmo com uma função de espalhamento simples `key % M`, conseguimos alcançar eficiência nas operações.

A tabela hash implementada não inova em nenhum aspecto que mereça destaque, exceto no caso das diferentes operações de busca/deleção, que serão comentadas mais adiante. Trata-se de uma implementação clássica, que conta com métodos de inserção, deleção e busca, além de funções auxiliares para verificação da necessidade de resize e rehash. O fator de carga utilizado na implementação é de 75%, e a função de espalhamento é `key % M`, onde `M` é o tamanho da tabela hash. A cada inserção, checamos a necessidade de aplicar resize; em caso positivo, triplicamos a capacidade da tabela. A tabela é inicializada com valores `None` e, para identificar se um elemento foi deletado, checamos o atributo `Event.deleted`, inicializado como `False`.

2.3 Operações

read_csv(file, sep=‘,’,enconding=‘utf-8’) método que popula a tabela hash a partir dos dados de um arquivo CSV. Ela percorre o arquivo CSV fazendo leitura linha a

[1] As implementações clássicas de tabela hash são endereçamento aberto e lista encadeada. Como o trabalho foi desenvolvido na linguagem Python, a opção mais razoável foi a de endereçamento aberto.

linha, criando instâncias da classe `Event` e inserindo-os na tabela hash utilizando o método `insert()` da própria tabela hash.

show(start=0, end=100, returns=True, prints=False) método responsável por exibir as entradas do datagrid, limitando a exibição ao intervalo definido pelos parâmetros. Se a tabela já estiver ordenada, ela usará a ordem atual para exibir os dados. Caso contrário, ela percorrerá a tabela hash original. Além dos parâmetros solicitados na descrição do trabalho, foram adicionados outros dois, com valores default definidos, para que o usuário informe se deseja que a função printe, retorne, realize as duas operações ou nenhuma delas. `returns=True` retorna a lista de objetos da classe `Event` entre `start` e `end`, e o `prints=True` faz o display do conteúdo desses objetos.

insert_row(row) método que insere novos eventos no datagrid. Ela recebe um dicionário contendo os dados do evento a ser inserido e cria uma instância de `Event` a partir desses dados. Em seguida, insere o evento na tabela hash, associando-o à sua chave única.

delete_row(column, value) método que remove entradas da tabela que correspondem a um valor especificado em uma coluna específica. Ela percorre a tabela hash em busca de eventos que correspondam ao critério especificado e os marca como excluídos (definindo o atributo `deleted` como `True`), mantendo a integridade da tabela. Caso o método seja chamado com `column = 'position', value`, com `value` podendo ser uma tupla `(start, end)` de inteiros positivos ou um número inteiro positivo, será deletado o range de linhas ou a linha `value` na ordenação atual da tabela.

search(column, value) método permite encontrar e exibir as entradas da tabela que correspondem a um valor especificado em uma coluna. Dependendo do tipo de busca definido para a coluna, a método verificará a correspondência e exibirá as entradas encontradas. O método retorna uma lista de objetos da classe `Event`.

sort(column, direction='asc') método que permite ordenar a tabela com base nos valores de uma coluna especificada. O algoritmo de ordenação utilizado varia dependendo do tipo de dado na coluna, como radix sort para strings, merge sort para outros tipos. A método também pode inverter a ordem se `direction` for definida como descendente.

select_count(i, j, how='median-of-medians') método para extrair e exibir entradas dentro do intervalo especificado com base na coluna `count` ordenada de forma crescente, usando algoritmos de seleção como o Median of Medians, Quickselect ou Heapsort. Utiliza o Median of Medians por padrão, por ser o mais eficiente. O método retorna uma lista de objetos da classe `Event`.

2.4 Algoritmos de Busca/Deleção

Para a operação de busca e deleção, o código implementa a pesquisa em campos específicos da entidade `Event` usando diferentes algoritmos com base no tipo de busca definido para cada campo. A operação de deleção é essencialmente a mesma que a busca, exceto pelo fato de que na deleção, encontrado o elemento, alteramos seu atributo `deleted` para `True`. Portanto, as especificações que seguem são válidas tanto para os algoritmos de busca quanto de deleção.

No caso do campo `id`, que a busca é realizada diretamente na tabela hash usando a chave como identificador, a complexidade é $O(1)$, pois a estrutura da tabela hash proporciona acesso direto aos elementos considerando uma função de espalhamento que minimiza colisões. Para os campos `owner_id` e de busca por intervalo (`creation_date` e `count`), é necessário percorrer todos os eventos da tabela realizando comparações, resultando em uma complexidade de $O(n)$, onde n é o número de eventos na tabela. No caso de `creation_date`, a verificação do intervalo é realizada através das funções `is_posterior_to` e `is_prior_to`, que comparam as datas — a função `is_posterior_to` verifica se a data de um evento é posterior à data inicial do intervalo, enquanto `is_prior_to` verifica se ela é anterior à data final do intervalo. Combinando essas duas verificações, é possível determinar se o evento está dentro do intervalo desejado ou não. Para `count`, é feita uma verificação direta se o valor do campo `count` do evento está dentro do intervalo especificado.

Já para a busca por conteúdo parcial (do tipo "contém"), implementada nos campos `name` e `content`, foi utilizado o algoritmo de pseudo-código abaixo, que procura se existe um parágrafo em um texto.

```
1    matches = 0
2    Para cada caracter no texto
3        Se caracter[matches] == caracter_paragraph[matches]
4            matches++
5        Senao
6            matches=0
7        Se matches==quantidade de caracteres do paragrafo
8            Retorna True
9    Retorna Falso
```

Esse algoritmo verifica se o padrão de pesquisa (texto que estamos procurando) está contido no texto do evento, percorrendo-o da direita para a esquerda e fazendo comparações caracter a caracter. Quando são iguais, um contador é incrementado, quando não, é reinicializado como 0. Quando o contador atinge o tamanho do padrão de pesquisa, então está contido. Perceba que a complexidade no pior caso é quando busca pelo texto inteiro e não acha o padrão. Dessa forma, a complexidade desse algoritmo é $O(k)$, sendo k o tamanho do texto. Como rodamos para cada entrada da HashTable, então a complexidade total será $O(n * k)$.

A tabela a seguir resume as decisões de busca para cada campo, além de seus tipos de dado, tipos de busca, algoritmos associados e complexidades.

Nome	Tipo do dado	Tipo de Busca	Algoritmo	Complexidade
id	integer	exato	Busca em HT (no index)	$O(1)$
owner_id	string	exato	Busca em HT	$O(n)$
creation_date	string	intervalo	Busca em HT	$O(n)$
count	integer	intervalo	Busca em HT	$O(n)$
name	string	contém	B. em HT + B. no texto	$O(n * k)$
content	string	contém	B. em HT + B. no texto	$O(n * k)$

2.5 Algoritmos de Ordenação

Para a ordenação, foi utilizada a função `_extractArray(df, column)` que extrai a tupla `(Event, value)` da HashTable inicial, onde `value` é o valor do `Event` na coluna `column`. Essa operação itera todos os elementos da tabela, de modo que tem complexidade $O(n)$; e, por ser utilizada em todos os campos, é conhecido que a complexidade mínima buscada na ordenação seria $O(n)$. Além disso, isso define que essa lista de tuplas será a estrutura de dados utilizada na ordenação. Nesse ponto, é importante salientar que a ordenação é feita somente com base no segundo elemento da tupla (`value`).

A tabela abaixo sintetiza as decisões tomadas pelo grupo para cada operação em cada campo, além de suas análises de complexidade.

Nome	Tipo do dado	Algoritmo	Complexidade	Espaço adicional
id	integer	MergeSort	$O(nlogn)$	$O(n)$
owner_id	string	Radix Sort	$O(5 * (256 + n))$	$O(256 + n)$
creation_date	string	Radix Sort	$O(15 * (256 + n))$	$O(256 + n)$
count	integer	MergeSort	$O(nlogn)$	$O(n)$
name	string	Radix Sort	$O(20 * (256 + n))$	$O(256 + n)$
content	string	MergeSort	$O(nlogn)$	$O(n)$

2.5.1 Radix Sort

Para os campos `owner_id`, `creation_date` e `name` foi escolhido o algoritmo Radix Sort para ordenação. Os dois primeiros campos possuem uma quantidade sempre igual de caracteres - o que nos permite garantir uma ordenação em $O(n)$. Já o terceiro possui elementos de até 20 caracteres. Entretanto, computacionalmente, podemos ajustar para preencher com espaços nas palavras com menos de 20 caracteres, o que também irá garantir todos elementos com o mesmo tamanho. Além disso, a quantidade de caracteres desses campos não é alta, o que também não fará a constante ser alta a ponto de mudar a

complexidade. O tamanho do universo é relativamente considerável, sendo 256 o número de caracteres ASCII, já que todos os campos são strings.

2.5.2 Heapsort vs. MergeSort

Para os demais campos (`id, count` e `content`), foram considerados como campos do tipo `integer` sem restrições (o campo `content` é uma string, mas pode ser transformado em um inteiro ao utilizar o valor ASCII das letras). Para o campo `id`, foi definido que é único, isto é, que o tamanho U do universo será n, garantidamente. Para os outros, isso não foi garantido, mas também não podemos assumir que essa situação não ocorrerá. Por isso, nenhum algoritmo que exigisse espaço computacional alto (consideramos alto como maior que $O(n)$) seria eficiente de ser utilizado. Assim, Counting Sort e Radix Sort já foram descartados. Além disso, há uma grande probabilidade de nossos dados encontrarem-se em ordem descrescente para serem ordenados à ordem ascedente (já que a função ''ort' possui esse parâmetro). Trata-se do pior caso do QuickSort, que rodaria em $O(n^2)$ - situação que nos fez descartar esse algoritmo. Por isso, a escolha ficou entre HeapSort e MergeSort. Sabe-se das aulas que ambos possuem a mesma complexidade $O(n \log n)$, mas que o HeapSort é um pouco mais lento. Assim, foi realizado um experimento com os 3 algoritmos clássicos de complexidade $O(n \log n)$ para entradas aleatórias de números inteiros, conforme discutido na próxima seção.

2.6 Algoritmos de Seleção

Para implementação do método `select_count()`, que extrai as entradas no intervalo $[i, j]$ considerando a coluna `count` ordenada de forma crescente, foram considerados três algoritmos diferentes para teste, sendo dois de seleção e um de ordenação: Mediana das Medianas, Quickselect e Heapsort. Veja o pseudocódigo da implementação:

```
def select_count(i, j, how='median-of-medians'):
  # usa _extractArray(df, 'count')
  arr = extrai lista de events com valores de count

  se how == 'median-of-medians'
    aplica selectMOM em arr para achar i_MOM
    aplica selectMOM em arr lista para achar j_MOM
    instancia count=0
    instancia uma lista entries

  para cada event em arr
      se i_MOM <= event[1] <= j_MOM
        insere event em entries
      se count == j-i-1  # todos os elementos no range especificado
        aplicamos heapsort em entries
        sai do loop
```

```
17
18    instanciamos uma lista objects
19    para cada event em entries
20      inserimos event[0] em objects
21    return objects
22
23  se how == 'quickselect'
24    aplica quickselect em arr para achar i_quick
25    aplica quickselect em arr lista para achar j_quick
26    instancia count=0
27    instancia uma lista entries
28
29  para cada event em arr
30      se i_quick <= event[1] <= j_quick
31        insere event em entries
32      se count == j-i-1  # todos os elementos no range especificado
33        aplicamos heapsort em entries
34        sai do loop
35
36    instanciamos uma lista objects
37    para cada event em entries
38      inserimos event[0] em objects
39    return objects
40
41  se how == 'heapsort'
42    aplica heapsort em arr
43    instanciamos a lista objects
44    para cada event em arr
45      inserimos event[0] em objects
46    return objects[:j-i+1]
```

A implementação funciona de maneira semelhante no caso do Quickselect e Median of Medians: aplicamos os respectivos algoritmos à lista de tuplas para encontrar a i-ésima e j-ésima entradas. No caso da Mediana das Medianas sabemos que esse processo ocorre em $O(n)$. Para o Quickselect o $O(n)$ não é garantido, dado que o pior caso é $O(n^2)$, embora nossa implementação utilize pivôs aleatórios para tentar obter algum ganho de eficiência.

Tendo os elementos i e j, iteramos a HashTable em $O(n)$ para obter a lista de elementos no range especificado. Como precisamos retornar as entradas ordenadas em ordem crescente, aplicamos Heapsort aos $j-i$ elementos da lista. Para retornar, iteramos a lista ordenada para obter somente os objetos da classe `Event` das tuplas. No caso da implementação que utiliza somente o Heapsort, ordenamos toda a lista em $O(n \log n)$ e indexamos os elementos até $j-i$, considerando que a indexação de listas em Python inicia em 0.

A implementação da Mediana das Medianas se justifica pelo fato de que, ao obter-

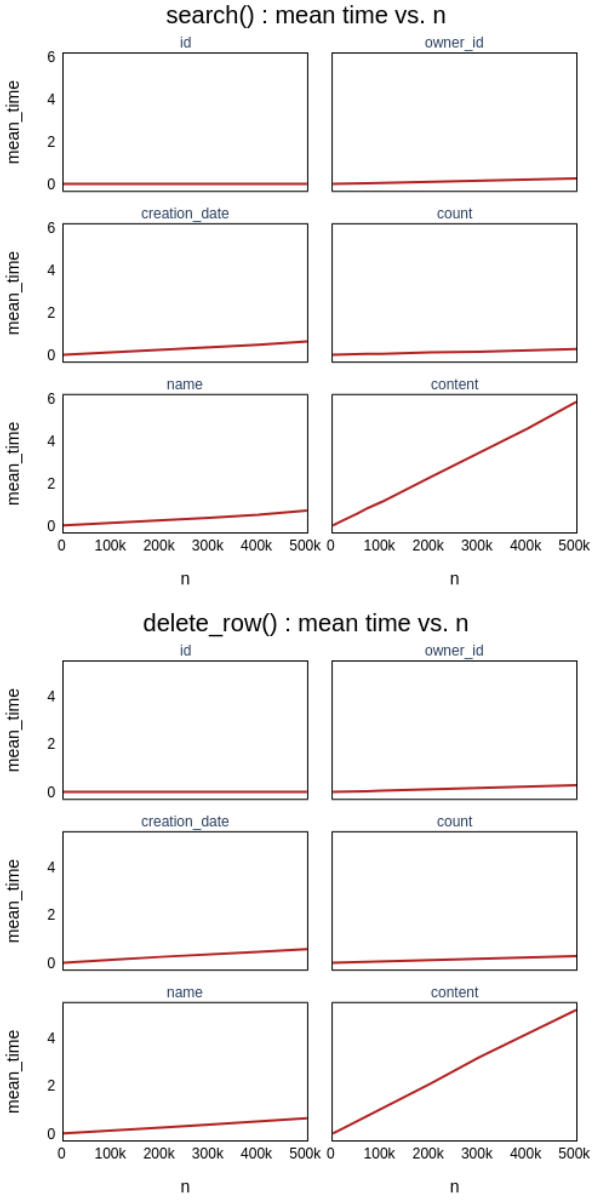
mos aproximadamente o elemento na metade do conjunto como pivô, chegamos à complexidade $O(n)$ em qualquer caso. O Quickselect utilizando pivô aleatório possui vantagens, mas não impede que chegue ao seu pior caso, em que possui complexidade $O(n^2)$. Além disso, a aplicação do Heapsort para ordenação acontece, nesses casos, em uma lista de tamanho $i - j$. A não ser que o usuário insira $i = 0$ e $j =$ tamanho da HashTable, esse valor será menor que o número total de elementos da tabela, levando a uma complexidade de execução menor que $O(n \log n)$ para ordenar. Desse modo, a Mediana das Medianas, mesmo combinada com o Heapsort, parece uma abordagem mais eficiente do que aplicar sua variação com o Quickselect ou mesmo o Heapsort sozinho.

3 Resultados e discussões

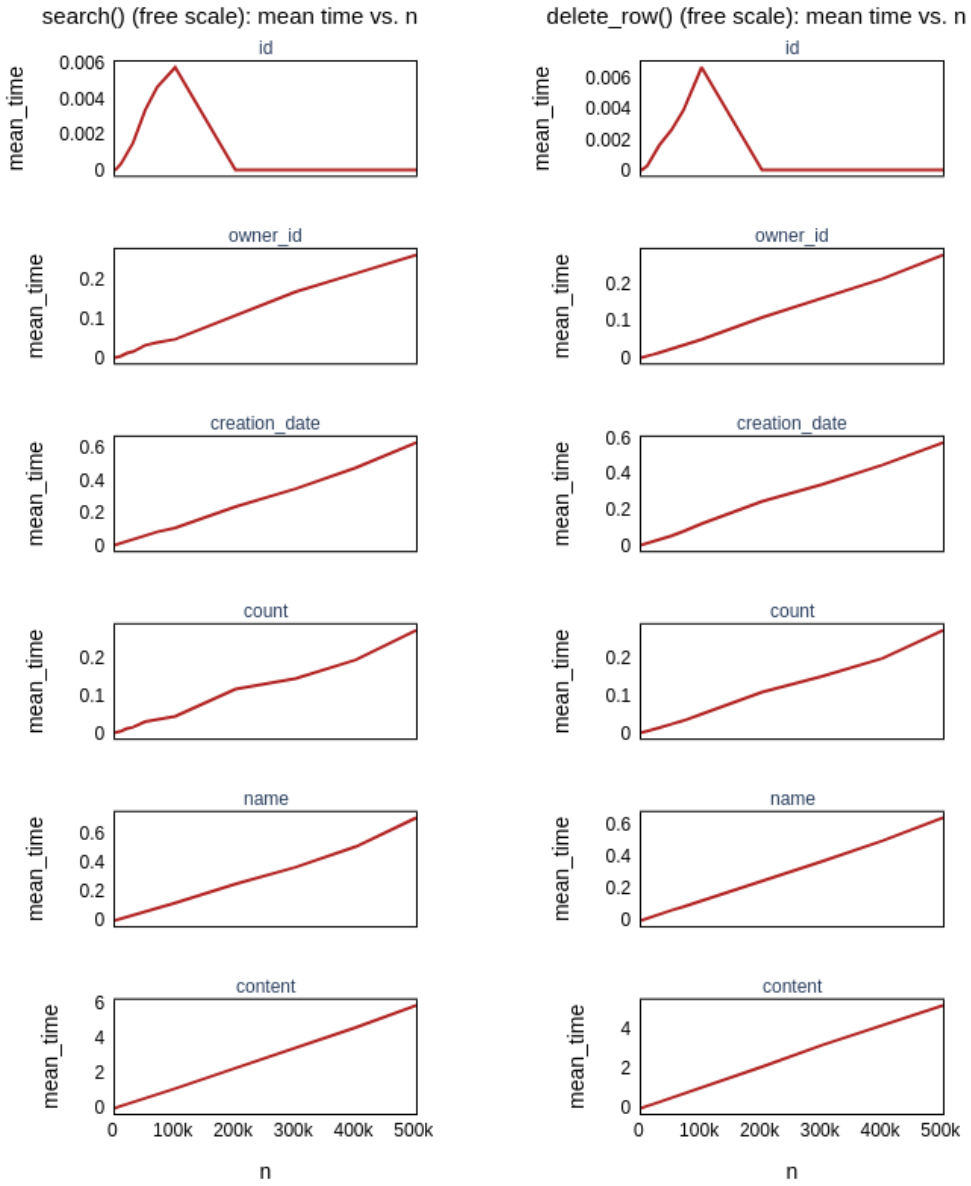
Para avaliar se as implementações funcionam de acordo com o esperado, executamos as operações repetidamente. Foi criado um arquivo `dataGenerator.py`, que gera dados aleatórios no formato especificado para o trabalho, e que permitiu a testagem para vários tamanhos de entrada diferentes. Veja abaixo, inicialmente, o resultado a operação de `insert` para tamanhos crescentes de entrada.

Observe que o gráfico é marcado por uma série de picos e quedas no tempo de inserção. Trata-se de um resultado razoável que se explica por conta da realização frequente de operações de resize e re-hash, que são operações $O(n)$, conforme são inseridos mais dados na tabela.

No caso das operações de `search` e `deleção` para as diferentes colunas, esperamos observar algo semelhante a $O(1)$ no caso do `id` e $O(n)$, com diferenças entre as constantes, para as outras colunas. Os gráficos apresentados nesses casos são apresentados em média: para diferentes tamanhos de entrada, definimos uma lista de buscas/deleções para serem realizadas, medimos o tempo de execução de cada uma delas e calculamos uma média. O resultado é o que segue:

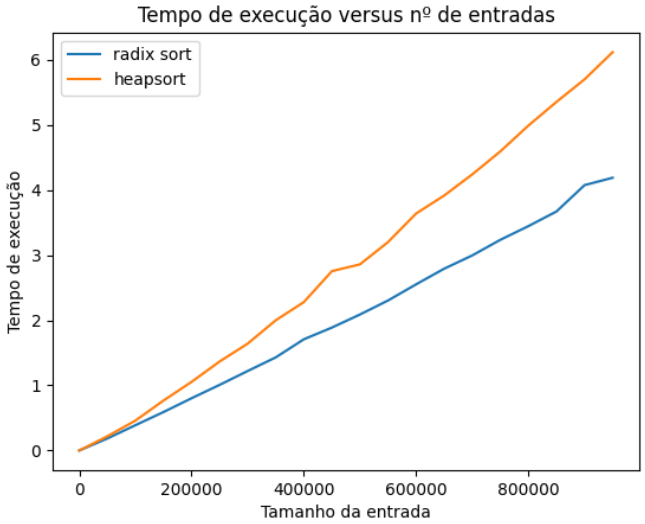


De fato, o resultado obtido condiz com o resultado esperado. Como a função de espalhamento se baseia no `id`, as buscas e deleções são mais eficientes e realizadas em $O(1)$. No caso das outras colunas, conseguimos observar $O(n)$, com diferenças relativas à constante. Conseguimos observar de forma mais específica os comportamento plotando gráficos com escalas independentes:

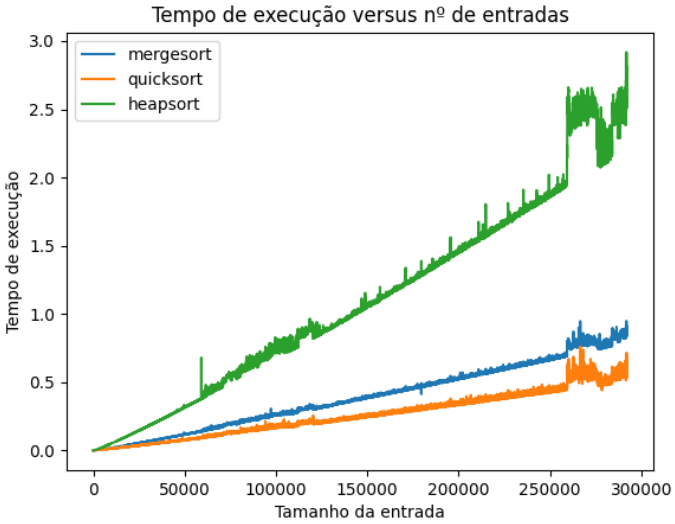


Esse é mais um resultado que confirma nossas hipóteses anteriores. Observamos que a complexidade de busca e deleção dos valores `id` são constantes, enquanto para as outras colunas a complexidade segue proporcional ao tamanho da tabela.

Quanto aos algoritmos de ordenação, foram executados experimentos para analisar se, a partir de algum n, seria mais vantajoso usar um algoritmo de complexidade $O(nlogn)$. O resultado encontra-se na figura abaixo, onde é possível observar que é vantagem usar o Radix Sort para todo n. Uma observação importante é que o experimento foi rodado para quando a constante $w = 20$, isto é, a maior constante dentre todos os Radix Sort implementados. Além disso, foi analisado apenas com o Heapsort, pois é o algoritmo que não exige espaço adicional para armazenamento.

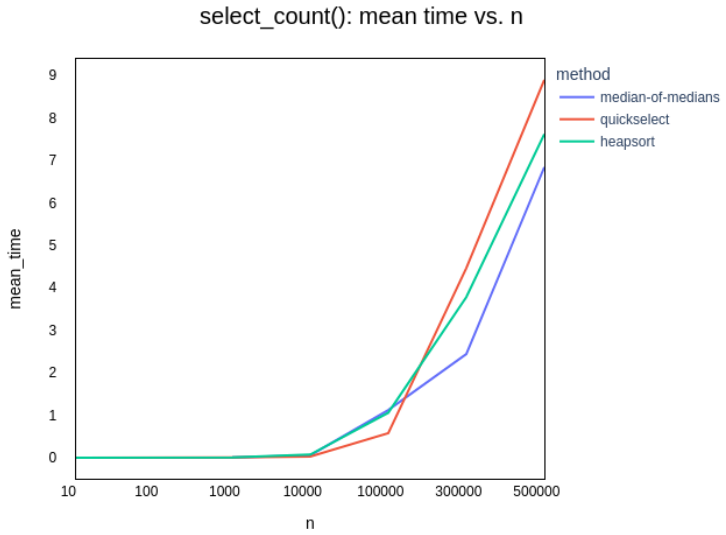


Para as demais entradas, foi rodado um experimento com os 3 algoritmos clássicos de complexidade $O(n \log n)$ para entradas aleatórias de números inteiros, conforme figura abaixo. Perceba que aqui fica evidente que o Heapsort possui um tempo mais elevado que os outros; em contrapartida, sua grande vantagem é realizar a ordenação *inplace*, de forma que praticamente não ocupa memória. Entretanto, como em nossa modelagem já estamos passando cópias da HashTable como parâmetro para as funções de ordenação, então essa vantagem do HeapSort acaba não existindo. Assim, o grupo optou por utilizar o algoritmo de MergeSort.



Os últimos resultados a serem discutidos dizem respeito ao desempenho do `select_count()`. Como já discutimos em outra oportunidade, esperamos que a implementação que utiliza Mediana das Medianas ($O(n)$) seja mais eficiente em relação às outras, já que o Quick-select no pior caso tem complexidade $O(n^2)$ e o Heapsort tem complexidade $O(n \log n)$. É importante ressaltar que, para a realização dos testes, o range de seleção do método

partiu sempre do 0 até um número aleatório entre 0 e o tamanho da entrada. Vejamos os resultados:



Pelo gráfico, conseguimos observar que, pelo menos até 10.000 entradas, não existe vantagem em utilizar um ou outro algoritmo. Por outro lado, para entradas maiores, podemos observar diferentes comportamentos. É interessante notar, por exemplo, que embora o Quickselect tenha apresentado um desempenho médio melhor que os outros para 100.000 entradas, seu tempo de execução rapidamente cresce desse ponto em diante, reforçando sua característica no pior caso — com complexidade maior, inclusive, que o Heapsort, conforme o gráfico reflete. O algoritmo que utililiza Mediana das Medianas, de fato, foi o que apresentou melhor desempenho para entradas maiores.