Relatório do Software Anti-plágio CopySpider
Para mais detalhes sobre o CopySpider, acesse: https://copyspider.com.br

Instruções
Este relatório apresenta na próxima página uma tabela na qual cada linha associa o conteúdo do arquivo de entrada com um documento encontrado na internet (para "Busca em arquivos da internet") ou do arquivo de entrada com outro arquivo em seu computador (para "Pesquisa em arquivos locais"). A quantidade de termos comuns representa um fator utilizado no cálculo de Similaridade dos arquivos sendo comparados. Quanto maior a quantidade de termos comuns, maior a similaridade entre os arquivos. É importante destacar que o limite de 3% representa uma estatística de semelhança e não um "índice de plágio". Por exemplo, documentos que citam de forma direta (transcrição) outros documentos, podem ter uma similaridade maior do que 3% e ainda assim não podem ser caracterizados como plágio. Há sempre a necessidade do avaliador fazer uma análise para decidir se as semelhanças encontradas caracterizam ou não o problema de plágio ou mesmo de erro de formatação ou adequação às normas de referências bibliográficas. Para cada par de arquivos, apresenta-se uma comparação dos termos semelhantes, os quais aparecem em vermelho.
Veja também:
Analisando o resultado do CopySpider
Qual o percentual aceitável para ser considerado plágio?

Versão do CopySpider: 2.3.1
Relatório gerado por: humbertojoji@gmail.com
Modo: web / normal

Arquivos	Termos comuns	Similaridade
Trabalho_aps.docx X https://joaoarthurbm.github.io/eda/posts/quick-sort	85	1,26
Trabalho_aps.docx X https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao	67	1,12
Trabalho_aps.docx X https://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html	43	0,72
Trabalho_aps.docx X https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261	55	0,64
Trabalho_aps.docx X https://br.usembassy.gov/pt/visas-pt/vistos-de-nao-imigrantes	14	0,19
Trabalho_aps.docx X https://lista.mercadolivre.com.br/espelho-de-seguranca-convexo	2	0,03
Trabalho_aps.docx X https://www.zeusdobrasil.com.br/produto/espelho-de-seguranca-convexo-com-suporte-borda-borracha-329	1	0,02
Trabalho_aps.docx X http://www.w3.org/2000/svg	0	0,00
Arquivos com problema de download		
https://www2.unifap.br/furtado/files/2016/11/Aula4.pdf	Não foi possível baixar o arquivo. É recomendável baixar o arquivo manualmente e realizar a análise em conluio (Um contra todos). - PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target	
https://www.youtube.com/watch%3Fv%3DwU7Q8Z51MUI	Não foi possível baixar o arquivo. É recomendável baixar o arquivo manualmente e realizar a análise em conluio (Um contra todos). - Erro: Parece que o documento não existe ou não pode ser acessado. HTTP response code: 429 - Server returned HTTP response code: 429 for URL: https://www.youtube.com/watch%3Fv%3DwU7Q8Z51MUI	

==========

Arquivo 1: Trabalho_aps.docx (4117 termos)
Arquivo 2: https://joaoarthurbm.github.io/eda/posts/quick-sort (2671 termos)
Termos comuns: 85
Similaridade: 1,26%
O texto abaixo é o conteúdo do documento Trabalho_aps.docx (4117 termos)
Os termos em vermelho foram encontrados no documento
https://joaoarthurbm.github.io/eda/posts/quick-sort (2671 termos)

==========

UNIVERSIDADE PAULISTA

G85JFG6 ? Eric Mitchell Barbosa Durham
N078146 ? Rafael Hiro Portilho
R0238H0 ? Matheus Reis Oliveira
N4031A9 ? Humberto Joji Fukui

Ordenação de dados amazônicos
Análise de algoritmos de ordenação de dados

São Paulo
2024

Neste trabalho tem o intuito de apresentar e comparar sobre os algoritmos Merge Sort, Quicksort e Heap Sort o seu desenvolvimento, eficiencia e complexidade. Avaliaremos os resultados adquiridos por cada algoritmo e discutir a aplicação desses agoritmos em alguns tipos de situações e uma analise do tempo que consiguram resolver cada situação. Os algoritmos de ordenações desenpenham papeis fundamentais na organização e otmização de dados, tornando-se fundamental para a eficiência para operações de busca e armazenamento em sistemas em sistemas de bancos de dados e outras aplicações computacionais.

Os algoritmos Merge sort, Heap sort e Quicksort conhecidos por sua eficiência e são amplamente utilizados em diversas linguagens de programação e bibliotecas. Cada um deles possui suas próprias características e vantagens, que serão exploradas em detalhes neste trabalho.

Quick sort se destaca por sua escolha programada de pivôs e rearranjo recursivo dos elementos, apresenta uma complexidade média de O(n log n), por outro lado o Merge sort conhecido por sua eficiencia na ordenação por divisão e conquista, destaca-se pela sua eficiência e estabilidade, dando uma ordenação eficaz dos elementos com complexidade O(n log n).O Heap sort, oferece um ordenação eficiente com complexidade O(n log n), destacando-se pela sua robustez.

Ao estudar esses algoritimos, temos como objetivo mostrar uma visão abrangente de suas características, desempenho e enetendimento a diferentes contextos, auxiliando na seleção do algoritmo mais adequado de acordo com a necessidade especifica de cada situação aplicada a eles.

2 REFERENCIAL TEÓRICO

2.1 QUICKSORT

Esse algoritmo usa a técnica da "divisão e conquista", que consiste na divisão recursiva de um problema em vários menores. No caso, o QuickSort escolhe um elemento do conjunto como pivô, sendo esse geralmente o primeiro elemento. Daí, reorganiza-se os demais elementos de forma que apenas os elementos menores que o pivô se situam à esquerda deste, assim formando dois subconjuntos em cada lado do pivô: um com elementos menores e outro com elementos maiores.

Na configuração resultante, o pivô passa a estar necessariamente na posição correta, e é então fixado. O processo descrito acima é repetido nos subconjuntos formados, definindo novo pivôs em cada. Isso continua recursivamente, formando conjuntos cada vez menores até o surgimento de conjuntos unitários, conforme esquematizado na figura 1.

O QuickSort é considerado relativamente eficiente dentre os algoritmos de ordenação mais usados. Entretanto, há algumas desvantagens principais como o uso de memória necessário, que aumenta consideravelmente com o número de elementos do vetor de interesse devido ao caráter recursivo do algoritmo.
Além disso, o QuickSort pode tornar-se bem ineficiente quando apresentado com vetores em certos arranjos. Por exemplo, podemos citar o caso quando a lista já está maiormente ordenada em ordem crescente ou decrescente, na qual o algoritmo não poderá fazer rearranjos significativos.

2.2 MERGESORT

Também se enquadrando na categoria de ?divisão e conquista?, o MergeSort divide o vetor em duas partes aproximadamente iguais, até a formação de conjuntos unitários similarmente ao QuickSort, antes de intercalar-se de volta na ordem reversa, até retornar ao tamanho inicial do vetor de interesse.
O processo de intercalação entre duas listas é feito inserindo os elementos de cada num vetor auxiliar, que é subsequentemente sobrescrito nos dois conjuntos iniciais. Compara-se um elemento de cada conjunto, inserindo sempre o menor (caso não haja valores iguais), começando com os respectivos primeiros elementos e avançando conforme a inserção, de forma independente.
Realizando-se essa junção a partir dos conjuntos unitários formados, os subconjuntos a serem intercalados sempre estão organizados internamente, facilitando a organização final como um todo.

Similarmente ao QuickSort, esse algoritmo exige uso de memória além do próprio vetor de interesse por ser recursivo.

2.3 HEAPSORT

É um algoritmo baseado na árvore binária máxima. Nesse tipo de sequência, cada elemento pai tem no máximo dois elementos filhos associados de valores igual ao menor que o pai. Os elementos filhos, por sua vez, podem ter filhos próprios, assim formando uma estrutura similar a uma árvore, onde o primeiro elemento é considerado a raiz da estrutura como um todo.
Aqui, utiliza-se repetidamente um método chamado de Heapify, descrito como segue: um elemento parente X é comparado com seus filhos (ou filho único). Se não existe filho superior ao X, o método é finalizado. Caso contrário, trocam-se as posições do X e do filho com valor superior, e esse processo continua na nova posição de X. Note que caso haja dois filhos com valor distintos maior que X, o maior será sempre escolhido.
No processo do HeapSort, incialmente o vetor é reorganizado até que se constitua numa árvore binária máxima. Isso é feito aplicando Heapify em todos os elementos parentes (com pelo menos um filho), em ordem decrescente. Ao final, isso é suficiente em tornar o vetor completo numa árvore máxima válida.

Em seguida, trocamos a raiz com o último elemento da árvore, fixando a raiz original na posição nova e desconsiderando-o do resto da árvore. Daí Heapify é chamado na nova raiz, e a troca é realizada novamente até todos os elementos serem eliminados da árvore, momento na qual o vetor passa a ser necessariamente ordenado.

Como exemplo, considere o vetor na esquerda da figura 3, que não constitui numa árvore válida ainda. Aplicando o HeapSort, chama-se o Heapify nos seguintes elementos, conforme a sequência:
Elemento 8: Como esse já é maior que os filhos, nenhuma troca é realizada;
Elemento 1: Existe filho com valor superior. Assim, troca-se 1 com 9, o filho de maior valor;
Elemento 7: Conforme a lógica do passo anterior, 7 é trocado com 8. Os novos filhos de 7, 4 e 6, são inferiores;
Elemento 3: Nesse ponto, os filhos passam a ser 8 e 9. Pela mesma lógica, troca-se 3 com 9, tornando 5 e 1 os novos filhos de 3. Assim, troca-se 3 com 5.
Assim, obtemos a árvore da direita da figura 3. Pelo outro lado, a figura 4 apresenta o passo seguinte, que envolve a eliminação do 9 da árvore.

Aqui, 9 tem sua posição trocada com 6, o elemento final da árvore. Em seguida, 9 é retirado da árvore e fixada na posição correta do vetor, fazendo a árvore resultante inválida mais uma vez. Daí, chama-se o Heapify no novo nó 6, que resultará na troca entre 6 e 8, e entre 6 e 7, para restaurar a propriedade da árvore. Com isso, 6 é trocado com 4, a nova posição final da árvore, e assim por diante.
O HeapSort não necessita de memória adicional além da própria função e do vetor de interesse, pois todos os rearranjos dos elementos são realizados dentro do vetor.

Este trabalho tem como objetivo desenvolver um sistema computacional para avaliar o desempenho de diversos algoritmos de ordenação de dados. Considerando a importância de algoritmos eficientes na análise e manipulação de grandes volumes de dados, o sistema proposto visa fornecer uma análise comparativa de desempenho utilizando dados internos e externos. Através do sistema, será possível mensurar o tempo de execução de cada algoritmo em diferentes condições, permitindo identificar suas vantagens e limitações.
Geração e/ou Obtenção de Dados para Ordenação:
Os dados são elementos essenciais para a análise deste trabalho, uma vez que eles permitem verificar a eficácia dos algoritmos estudados. Assim, foram desenvolvidas duas versões para cada programa de ordenação: uma para obter os dados a partir de arquivos de texto e outra para gerar números aleatórios. Dessa forma, busca-se proporcionar maior embasamento e rigor à análise, aproximando-se de resultados mais precisos e consistentes.
1.1 Obtenção de Dados:
Para a obtenção dos dados, foram utilizadas informações fornecidas pelo professor, organizadas em uma pasta principal que contém subpastas com conjuntos de dados de diferentes tamanhos (dez, cem, mil, dez

mil, cinquenta mil, cem mil, quinhentos mil e cinco mil). Cada subpasta possui versões com dados duplicados e sem duplicados, permitindo uma análise mais abrangente. Dentro delas, possuíam mais outras cinco opções, aleatório, côncavo-decresce cresce, convexo-cresce decresce, crescente e decrescente, todos, continham dez arquivos de dados, cada uma tendo a característica descrita pela pasta , como pode ser visto na imagem abaixo.

Figura 1 ? Pasta com dados e caminho mostrando algumas das pastas

Fonte: De autoria própria.

No entanto, era necessário fazer o sistema ter a capacidade de leitura desses arquivos que até então ainda era impossível, portanto, foi usado a estrutura FILE, contida dentro da biblioteca de entradas e saídas padrão da linguagem de programação C (stdio.h). Por haver certas dificuldades para fazer com que o sistema percorre o caminho até onde era encontrado o arquivo, todo o processo de testes foi feito com os arquivos dentro da mesma pasta do programa.

A leitura do acervo de informações, é feita da seguinte maneira, cria-se uma variável de ponteiro, capaz de armazenar elementos de maneira dinâmica, ela faz a leitura dos arquivos, verificando também se é possível ou não a alocação de memória, caso tudo ocorra da maneira correta, uma função de loop, irá transferir todos os números guardados no arquivo para o vetor criado pelo ponteiro, até ele não ser capaz de encontrar mais dados, a figura abaixo mostra o trecho do código que faz todo esse processo.

Figura 2 ? Trecho do sistema de obtenção de dados.

Fonte: De autoria própria.

1.2 Geração dos dados

Em contrapartida, na geração de dados, foi utilizado a função rand(), da qual retornará um número-pseudo-aleatório, é dito como pseudo, porque ele não é completamente aleatório, já que é feito um cálculo especial usando em base um número padrão que é tomado como semente para gerar os seguintes, tendo como padrão o um. Há uma outra forma de se gerar esses dados, no entanto, seria utilizado o comando srand(), este teria uma semente diferente fazendo com que os números pudessem ser mais diferentes entre si, porém ele não foi utilizado no programa.

Conforme visto anteriormente, fora utilizado a função rand, para a geração de dados, porém é necessário um loop que possa ser capaz de criar esses números até o tamanho limite indicado pelo desenvolvedor. Consequentemente, foi usado um loop for, que fará a tarefa repetidas vezes até alcançar o tamanho limite indicado, a figura abaixo mostra o trecho do programa descrito acima:

Figura 3 ? Trecho do sistema de geração de dados

Fonte: De autoria própria

Processo de Ordenação de Dados:

Por outro lado, há o processo de ordenação, neste trabalho foram citadas e utilizadas três das principais, geralmente, as mais usadas, são elas, Quick Sort, Heap Sort, e Merge Sort, cada uma possuindo seu método específico de organizar os dados, mas todas as escolhidas, são consideradas as mais rápidas dentre as mais populares.

Cada algoritmo possui uma complexidade de tempo e de espaço, que permite prever, respectivamente, o tempo necessário para a execução do algoritmo e a quantidade de memória exigida para armazenar as estruturas necessárias durante o processamento.

O uso desses cálculos é extremamente importante para ver se é possível ou não o uso desse programa na máquina que será testada, caso contrário, diversos tipos de erros podem acabar ocorrendo, afetando diretamente o resultado.

2.1 Quick Sort:

O algoritmo Quick Sort, traduzido como ?ordenação rápida?, é conhecido por sua eficiência e velocidade na ordenação de dados. Sua eficiência se deve à estratégia de ?Divisão e Conquista?, que divide o vetor em partes menores, organiza cada parte individualmente e, em seguida, combina as segmentações ordenadas para obter o resultado.
Segundo Cormen et al. (2012), essa abordagem é especialmente eficiente em algoritmos de ordenação, pois permite reduzir o tempo de execução para uma complexidade de O(n log n), na média dos casos.
No entanto, para poder dividir, ele precisa encontrar um ponto para dividir, essa necessidade é resolvida após a escolha de um elemento na lista como um pivô, este número será o valor de referência usado para dividir a lista em duas partes, elementos maiores, e menores.

Figura 4 ? Trecho do algoritmo Quick Sort realizando o loop de partição do vetor
Fonte: De própria autoria
Assim como na figura acima, o algoritmo então particiona o vetor, fazendo com que os elementos menores que o pivô se vão para a esquerda e os maiores fiquem à sua direita, esse processo é feito recursivamente para os subvetores decorrentes. Este processo continua a ocorrer até que a lista possua no máximo um elemento, pois assim, ela já estaria ordenada.
A figura abaixo mostra o todo processo descrito anteriormente, dentro do sistema computacional.

Figura 5 ? Algoritmo de ordenação QuickSort

Fonte: De própria autoria
Já a complexidade de tempo média dele é representada pela fórmula:
O(n log n) (1)
Isto significa, que, ele realiza um número de comparações e movimentações que equivalem à fórmula (1), onde n é o tamanho da lista a ser ordenada. Portanto, este é um dos motivos que faz este algoritmo ser

tão eficiente, o seu tempo relativo já é suficientemente bom.
E por último, ele não possui cálculo de armazenamento, porque o Quick Sort é um algoritmo ?in-place?, ou seja, não requer memória adicional para armazenar as estruturas temporárias durante a ordenação. Tornando-o extremamente eficiente em termos de uso de memória.
2.2 Heap Sort:
O algoritmo Heap Sort funciona como uma árvore binária especial, na qual o valor de cada pai é maior ou igual ao valor de seus filhos (max-heap) ou menor ou igual (min-heap). No contexto do heapsort, é usado o max-folhas (max-heap), ou seja, ele irá tentar buscar o maior valor e colocá-lo no topo da árvore, e sempre seguindo dessa maneira até conseguir ordenar corretamente os elementos guardados.
Primeiramente, o algoritmo começa a iniciar o processo de transformação do vetor em uma árvore, sempre buscando o maior número e colocando-o ao topo da árvore, o funcionamento é parecido com o de uma pirâmide, esta figura abaixo representa visualmente como é o funcionamento.
Após o maior número atingir o topo da árvore, ele é movido para a posição final da lista e removido da árvore, então ocorre uma reorganização entre todas as folhas para levar o maior número para o topo novamente, para isso que ocorra até todos os elementos serem levados para o vetor ao ponto de ele estar completo, com todos os seus dados de volta.
Em suma, o algoritmo Heap, cria uma árvore binária e realiza uma constante busca para levar todos os números ao topo para que assim consigam ser organizados do maior para o menor. Em relação à complexidade de tempo médio, o cálculo do algoritmo Heap Sort é o mesmo que do Quick Sort (1), portanto possui as mesmas qualidades dele, sendo tão rápido quanto ele.
Assim como o tempo, sua complexidade de espaço também é a mesma do Quick, os dois são considerados in-place, ele usa o mesmo vetor para armazenar os elementos e para o processo de ordenação, sendo extremamente eficiente em situações em que se há pouca memória no computador. A figura abaixo mostra o algoritmo do Heap Sort por completo.
Figura 7 ? Algoritmo de ordenação Heap Sort

Fonte: De própria autoria
2.3 Merge Sort:
O Merge Sort se assemelha, muito com o algoritmo Quick Sort, seja por conta da sua estratégia, mas também por conta de sua versatilidade, eficiência e velocidade em organizar um vetor desorganizado.
Em comparação, os dois utilizam a estratégia, divisão e conquista, porém, uma grande diferença acaba os diferenciando e mudando radicalmente, isto é, a aplicação dessa estratégia é diferente para cada um deles .
Enquanto o algoritmo Quick Sort, acaba necessitando de um pivô, o Merge não se prende a isso, pois ele divide a lista em parte iguais e em seguida, combina essas partes ordenadas, completando a ordenação por completo. Em suma, um deles encontra o meio e divide e o outro escolhe um pivô, tornando o que separa no meio, o melhor, principalmente nos piores casos, pois, o pivô escolhido pode acabar não sendo o do meio, dificultando bastante e aumentando a complexidade de tempo para a seguinte fórmula:
$O(n^2)(2)$
Por outro lado, o Merge Sort continua constante mesmo em situações em que existem muitos elementos a serem ordenados, também não dependendo da sorte, facilitando ainda mais a organização, a figura 9 mostra uma representação de como é realizada a separação:
Figura 8 - Algoritmo de ordenação Merge Sort

Fonte: De autoria própria

A figura acima mostra como é feita a divisão recursiva do Merge Sort, no programa usado neste trabalho. Como citado, a complexidade de tempo do algoritmo é sempre constante, em todos os casos, por conta de como ele aplica a estratégia, então ele mantém a fórmula (1) sempre.

No entanto, a complexidade de espaço, acaba sendo levemente diferente, pois ele possui um auxiliar que acaba consumindo memória, fazendo-o não se tornar in-place. Na figura 8, podemos ver a variável ?t? sendo usada como vetor auxiliar. Assim a fórmula de espaço acaba sendo a seguinte, onde n é o número de elementos:

O(n)(3)

Listagem dos Valores Antes e Depois da Ordenação:

A análise será feita lendo o mesmo arquivo de texto, um de dez dados duplicados, para ser possível diferenciar a forma que cada um ordena, explicando seus passos e métodos.

3.1 Quick Sort:

Figura 9 ? Reposta dada pelo algoritmo de ordenação Quick Sort

Fonte: De autoria própria

Na figura acima, é possível ver como funciona o processo de ordenação do algoritmo Quick Sort lendo o arquivo, no entanto, o código acaba tendo certos problemas em mostrar o elemento, como é possível ver na terceira linha de passos imprimida, o último elemento mostra um endereço de memória, por conta de uma das recursividades acabar utilizando o endereço, fazendo a impressão se tornar confusa.

Contudo, analisando algumas partes compreensíveis da imagem, pode-se concluir que os passos da ordenação prosseguem dividindo, utilizando o pivô escolhido que acaba mudando após um certo período de impressões, pois ele vai ordenando os subvetores restantes, utilizando recursividade e economizando na alocação de memória.

3.2 Heap Sort:

Figura 10 - Resposta dada pelo algoritmo de ordenação Heap Sort

Fonte: De própria autoria

Assim como explicado, o maior valor é sempre levado para o fim do vetor, na figura acima, é mais facilmente visível como é feita essa ação. À medida que se progride, é visível a sucessão do processo, com o algoritmo mostrando a movimentação dos maiores elementos para a parte mais à extremidade esquerda, então, após percorrer por todas as folhas, move-as para o fim do vetor.

Por conta de ele ser uma árvore, as impressões acabam sendo um pouco maior, porém, é visível, em comparação com o Quick que por sua vez, acaba mostrando os endereços de memória.

3.3 Merge Sort:

Figura 11 - Resposta dada pelo algoritmo de ordenação Merge Sort

Fonte: De autoria própria

É imprimido de maneira bem similar, como o Quick Sort, porém de maneira visível mostrando o número invés do endereço, isso ocorre, pois, a impressão está ocorrendo em um momento diferente do Quick, neste caso, ela está após a transferência do vetor temporário, o que faz com que não seja imprimida o endereço de memória.

Portanto, é mais perceptível como funciona o procedimento de divisão e conquista, vendo como o 958 é lentamente passado para o meio, até aquele momento ele é considerado o maior número da primeira metade fazendo-o fica naquela posição e então cada metade vai se organizando.
Apresentação dos Resultados Comparativos de Performance:
Esta seção apresenta uma análise de resultados entre os algoritmos utilizando quatro tipos de cenários diferentes entre eles, podendo ser duplicados ou não. Cada um dos programas foi avaliado quanto ao tempo médio de execução, visando descobrir o melhor, dependendo em cada ocasião. Os testes foram feitos com arquivos de texto com 50 mil dados e os gerados tiveram 100 mil amostras.
4.1 Medição:
Para obter os resultados, foi necessário o uso da função clock() da biblioteca da linguagem de programação C, da qual pode medir, os milissegundos, com mais precisão, o momento exato que é executado o algoritmo de ordenação, sem contar o tempo de criação de variáveis, alocação de memória e obtenção ou geração de dados.
Desta maneira, foram obtidos os dados com o tempo médio em (ms) apresentados na seguinte tabela:
Tabela 1 ? Comparação do Tempo Médio de Execução dos Algoritmos com Dados Duplicados
Fonte: De autoria própria
Tabela 2 - Comparação do Tempo Médio de Execução dos Algoritmos sem Dados Duplicados
Fonte: De autoria própria
Com estas tabelas, é possível observar que, em situações de dados espalhados de forma aleatória, o algoritmo Heap Sort é um dos piores no quesito tempo, perdendo por poucos milissegundos.
No entanto, em situações em que os elementos são espalhados de forma côncava, caso os dados sejam duplicados, o Quick acaba perdendo, mas em situações sem duplicidade, o Heap acaba tendo uma desvantagem de 515 milissegundos, uma diferença que quando comparada em situações de mais dados, faz com que ele não seja tão eficiente.
Toda essa diferença entre os dois, ocorre devido a diferença da estratégia entre eles, enquanto um necessita fazer uma árvore e deve percorrê-la, o outro apenas se divide e compara até conseguir se ordenar.
Apesar do Quick conseguir ser mais veloz que o Heap nesses cenários, existem muitos casos em que a sua estratégia não funciona tão bem, por exemplo, com grandes quantidades de números, esse cenário faz com que o Quick se torne extremamente instável nessas ocasiões. Pois, a forma em que ele exerce seu plano, depende muito de o pivô estar mais perto do centro ou não, podendo facilitar e muito em arquivos de dados pequenos.
Algo que não ocorre com o Merge Sort, que por sua vez, acaba brilhando nessas situações já que a aplicação da estratégia Divisão e Conquista é diferente, fazendo uma divisão por 2, não precisando ser tão dependente de achar ou não um pivô mais próximo do centro do vetor, já que quando dividido, sempre cairá ao meio.
No entanto, o algoritmo Merge Sort acaba tendo uma grande desvantagem em comparação aos outros dois algoritmos, o armazenamento de espaço, o que torna ele estável na maior parte das situações, tende a deixá-lo lento, pois ocorre a transferência de números do vetor temporário ao original.

Ref Biblio:
CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: Teoria e Prática. 3. ed. Rio de Janeiro: Elsevier, 2012.
SEDGEWICK, Robert; WAYNE, Kevin. Algorithms. 4th ed. Boston: Addison-Wesley, 2011.
MANZANO, José Augusto N. G.; OLIVEIRA, Jayr Figueiredo de. Estruturas de Dados e Algoritmos. 3. ed. São Paulo: Érica, 2011.
SIPSER, Michael. Introduction to the Theory of Computation. 3rd ed. Boston: Cengage Learning, 2012.
KNUTH, Donald E. The Art of Computer Programming: Sorting and Searching. 2nd ed. Massachusetts: Addison-Wesley, 1998.
DEITEL, H.; DEITEL, P. C: Como Programar. 7. ed. São Paulo: Pearson, 2010.
Resultados e Discussão

Como visto previamente neste documento, cada tipo de método de ordenação tem características próprias que o tornam altamente eficiente em algumas ocasiões e altamente ineficiente em outras. No entanto torna-se necessário uma análise mais profunda destes dados para um melhor entendimento dos mesmos. Para tal feito, foram coletadas mais de 800 amostras visando um melhor entendimento de cada método de ordenação.

Todas as amostras foram coletadas tendo como princípio a meta de 400 dados para cada tipo de método de ordenação, separados em dois subgrupos de 200 dados, respectivamente com e sem duplicidade de dados. Cada um destes subgrupos foi dividido em 5 subgrupos de 40 dados cada (aleatório, côncavo, convexo, crescente, decrescente) e um subgrupo com 100 dados(gerado pelo nosso grupo). Cada dado era composto por 50.000 números(como demonstrado abaixo em nossa imagem de autoria própria).

Ao serem analisados foi possível se tirar a seguinte tabela (autoria própria) como conclusão:

Os números com código de cores ao lado na parte inferior dos gráficos acima representam o tempo (em segundos) que cada algoritmo demorou para ser executado, já os números ao lado da palavra ?Frequência? representam a frequência de cada um dos números com código de cores ao lado.
Os gráficos acima permitem concluir diversos fatores. Por exemplo, os métodos de ordenação heap e merge têm uma variação menor do que o quick sort em termos de análise de dados. No entanto, o método merge possui uma demora muito maior para processar os dados que o quick e o heap.
Com tais dados foi possível montar uma curva normal extremamente detalhada para cada tipo de algoritmo de ordenação, com e sem duplicidade de números, e para os números também localmente gerados a partir de métodos de ordenação. Abaixos estão respectivamente as curvas normais (autoria própria) dos dados listados acima :

Os grupos de curvas normais apresentados acima são respectivamente pertencentes aos métodos de ordenação hippie, quick e merge. Com os códigos de cores preto para ordenação com duplicidade, vermelho para ordenação se duplicidade e azul para os dados coletados a partir de dados gerados localmente.
É possível notar que em cada grupo é apresentado vários símbolos com significados diversos. O símbolo ? denota a média de todos os dados coletados. O símbolo ? descreve o desvio padrão do objeto em estudo.
O número indicado no balão é o mesmo que o do símbolo X. A porcentagem que se segue no meio da curva de sino traz qual a probabilidade de haver números abaixo de X o que, por inferência, nos traz uma taxa de confiança de 95% para x ? X. Portanto pode-se afirmar que nenhum número recairá abaixo de X a não ser por acaso, e que X deve ser tomado como nosso objeto de estudo atual ignorando-se todos os tempos acima do mesmo.
Nota-se também que o método de ordenação quick está com menos dados na parte de com e sem duplicidade. Isso se deve ao fato do erro Stack overflow ter ocorrido durante a execução de ambos os dados decrescente e crescente, então se foi executado apenas uma vez cada e posto o tempo de demora para o código sair.
Com isso em mente é possível concluir que os métodos de ordenação quick e heap, são realmente mais rápidos em algumas ocasiões. No entanto, os métodos que apresentam mais estabilidade durante a execução são os métodos heap e merge. Há também a probabilidade de o método quick dar o erro Stack overflow devido a quantidade de memória ram que ocupa durante o processo de execução. Por fim, é possível concluir que o método de ordenação mais estável e o mais rápido é o heap.
Concluimos que os algoritmos de ordenação Quick Sort, Merge Sort e Heap Sort, mesmo que apresentem complexidades de tempo semelhantes , demonstram comportamentos distintos em diferentes cenários.

Algoritmos de ordenação estáveis, como o Merge Sort , são úteis em situações em que a preservação da ordem relativa é importante, como ordenação de registros em bancos de dados ou classificação por múltiplos critérios. Por outro lado, algoritmos de ordenação instáveis, como o Quicksort, podem alterar a ordem de elementos com chaves iguais durante o processo de classificação.
O Quick Sort e Heap Sort tiveram um desempenho melhor que o Merge sort sem duplicidade e com duplicidade, onde o Quick Sort se sobrepõe ao Heap Sort tendo menos frequencia, com uma diferença de 25 no tem 0,006 sem duplicidade e com duplicidade um diferença de 42.
Portanto a escolha do algoritmmo de ordenação ideal depende de diversas situações, como o tamho do conjunto de dados, os requisitos da memória ea estabilidade. No geral o Quick Sort será uma excelente opção para a maioria dos casos devido ao seu desempenho .No entanto o Heap Sort pode ser uma boa opção para grandes conjutos de dados. O Merge Sort pode ser uma boa opção quando o espaço de memória for limitado.

AlgoritmoAleatório (seg.)Côncavo (seg.)Convexo (seg.)Gerado (seg.)
Quick Sort0.005550.006450.00540,00527
Heap Sort0.0077750.0058750.00580,01695
Merge Sort0.011150.008150.0082750,011

AlgoritmoAleatório (seg.)Côncavo (seg.)Convexo (seg.)
Quick Sort0.0056750.006450.005875
Heap Sort0.00750.01160.0065
Merge Sort0.01150.00850.0085

==========

Arquivo 1: Trabalho_aps.docx (4117 termos)
Arquivo 2: https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao (1881 termos)
Termos comuns: 67
Similaridade: 1,12%
O texto abaixo é o conteúdo do documento Trabalho_aps.docx (4117 termos)
Os termos em vermelho foram encontrados no documento
https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao (1881 termos)

==========

UNIVERSIDADE PAULISTA

G85JFG6 ? Eric Mitchell Barbosa Durham
N078146 ? Rafael Hiro Portilho
R0238H0 ? Matheus Reis Oliveira
N4031A9 ? Humberto Joji Fukui

Ordenação de dados amazônicos
Análise de algoritmos de ordenação de dados

São Paulo
2024

Neste trabalho tem o intuito de apresentar e comparar sobre os algoritmos Merge Sort, Quicksort e Heap Sort o seu desenvolvimento, eficiencia e complexidade. Avaliaremos os resultados adquiridos por cada algoritmo e discutir a aplicação desses agoritmos em alguns tipos de situações e uma analise do tempo que consiguram resolver cada situação. Os algoritmos de ordenações desenpenham papeis fundamentais na organização e otmização de dados, tornando-se fundamental para a eficiência para operações de busca e armazenamento em sistemas em sistemas de bancos de dados e outras aplicações computacionais.

Os algoritmos Merge sort, Heap sort e Quicksort conhecidos por sua eficiência e são amplamente utilizados em diversas linguagens de programação e bibliotecas. Cada um deles possui suas próprias características e vantagens, que serão exploradas em detalhes neste trabalho.

Quick sort se destaca por sua escolha programada de pivôs e rearranjo recursivo dos elementos, apresenta uma complexidade média de O(n log n), por outro lado o Merge sort conhecido por sua eficiencia na ordenação por divisão e conquista, destaca-se pela sua eficiência e estabilidade, dando uma ordenação eficaz dos elementos com complexidade O(n log n).O Heap sort, oferece um ordenação eficiente com complexidade O(n log n), destacando-se pela sua robustez.

Ao estudar esses algoritimos, temos como objetivo mostrar uma visão abrangente de suas características, desempenho e enetendimento a diferentes contextos, auxiliando na seleção do algoritmo mais adequado de acordo com a necessidade especifica de cada situação aplicada a eles.

2 REFERENCIAL TEÓRICO

2.1 QUICKSORT

Esse algoritmo usa a técnica da "divisão e conquista", que consiste na divisão recursiva de um problema em vários menores. No caso, o QuickSort escolhe um elemento do conjunto como pivô, sendo esse geralmente o primeiro elemento. Daí, reorganiza-se os demais elementos de forma que apenas os elementos menores que o pivô se situam à esquerda deste, assim formando dois subconjuntos em cada lado do pivô: um com elementos menores e outro com elementos maiores.

Na configuração resultante, o pivô passa a estar necessariamente na posição correta, e é então fixado. O processo descrito acima é repetido nos subconjuntos formados, definindo novo pivôs em cada. Isso continua recursivamente, formando conjuntos cada vez menores até o surgimento de conjuntos unitários,

conforme esquematizado na figura 1.

O QuickSort é considerado relativamente eficiente dentre os algoritmos de ordenação mais usados. Entretanto, há algumas desvantagens principais como o uso de memória necessário, que aumenta consideravelmente com o número de elementos do vetor de interesse devido ao caráter recursivo do algoritmo.
Além disso, o QuickSort pode tornar-se bem ineficiente quando apresentado com vetores em certos arranjos. Por exemplo, podemos citar o caso quando a lista já está maiormente ordenada em ordem crescente ou decrescente, na qual o algoritmo não poderá fazer rearranjos significativos.

2.2 MERGESORT

Também se enquadrando na categoria de ?divisão e conquista?, o MergeSort divide o vetor em duas partes aproximadamente iguais, até a formação de conjuntos unitários similarmente ao QuickSort, antes de intercalar-se de volta na ordem reversa, até retornar ao tamanho inicial do vetor de interesse.
O processo de intercalação entre duas listas é feito inserindo os elementos de cada num vetor auxiliar, que é subsequentemente sobrescrito nos dois conjuntos iniciais. Compara-se um elemento de cada conjunto, inserindo sempre o menor (caso não haja valores iguais), começando com os respectivos primeiros elementos e avançando conforme a inserção, de forma independente.
Realizando-se essa junção a partir dos conjuntos unitários formados, os subconjuntos a serem intercalados sempre estão organizados internamente, facilitando a organização final como um todo.

Similarmente ao QuickSort, esse algoritmo exige uso de memória além do próprio vetor de interesse por ser recursivo.

2.3 HEAPSORT

É um algoritmo baseado na árvore binária máxima. Nesse tipo de sequência, cada elemento pai tem no máximo dois elementos filhos associados de valores igual ao menor que o pai. Os elementos filhos, por sua vez, podem ter filhos próprios, assim formando uma estrutura similar a uma árvore, onde o primeiro elemento é considerado a raiz da estrutura como um todo.
Aqui, utiliza-se repetidamente um método chamado de Heapify, descrito como segue: um elemento parente X é comparado com seus filhos (ou filho único). Se não existe filho superior ao X, o método é finalizado. Caso contrário, trocam-se as posições do X e do filho com valor superior, e esse processo continua na nova posição de X. Note que caso haja dois filhos com valor distintos maior que X, o maior será sempre escolhido.
No processo do HeapSort, incialmente o vetor é reorganizado até que se constitua numa árvore binária máxima. Isso é feito aplicando Heapify em todos os elementos parentes (com pelo menos um filho), em

ordem decrescente. Ao final, isso é suficiente em tornar o vetor completo numa árvore máxima válida. Em seguida, trocamos a raiz com o último elemento da árvore, fixando a raiz original na posição nova e desconsiderando-o do resto da árvore. Daí Heapify é chamado na nova raiz, e a troca é realizada novamente até todos os elementos serem eliminados da árvore, momento na qual o vetor passa a ser necessariamente ordenado.

Como exemplo, considere o vetor na esquerda da figura 3, que não constitui numa árvore válida ainda. Aplicando o HeapSort, chama-se o Heapify nos seguintes elementos, conforme a sequência:
Elemento 8: Como esse já é maior que os filhos, nenhuma troca é realizada;
Elemento 1: Existe filho com valor superior. Assim, troca-se 1 com 9, o filho de maior valor;
Elemento 7: Conforme a lógica do passo anterior, 7 é trocado com 8. Os novos filhos de 7, 4 e 6, são inferiores;
Elemento 3: Nesse ponto, os filhos passam a ser 8 e 9. Pela mesma lógica, troca-se 3 com 9, tornando 5 e 1 os novos filhos de 3. Assim, troca-se 3 com 5.
Assim, obtemos a árvore da direita da figura 3. Pelo outro lado, a figura 4 apresenta o passo seguinte, que envolve a eliminação do 9 da árvore.

Aqui, 9 tem sua posição trocada com 6, o elemento final da árvore. Em seguida, 9 é retirado da árvore e fixada na posição correta do vetor, fazendo a árvore resultante inválida mais uma vez. Daí, chama-se o Heapify no novo nó 6, que resultará na troca entre 6 e 8, e entre 6 e 7, para restaurar a propriedade da árvore. Com isso, 6 é trocado com 4, a nova posição final da árvore, e assim por diante.
O HeapSort não necessita de memória adicional além da própria função e do vetor de interesse, pois todos os rearranjos dos elementos são realizados dentro do vetor.

Este trabalho tem como objetivo desenvolver um sistema computacional para avaliar o desempenho de diversos algoritmos de ordenação de dados. Considerando a importância de algoritmos eficientes na análise e manipulação de grandes volumes de dados, o sistema proposto visa fornecer uma análise comparativa de desempenho utilizando dados internos e externos. Através do sistema, será possível mensurar o tempo de execução de cada algoritmo em diferentes condições, permitindo identificar suas vantagens e limitações.
Geração e/ou Obtenção de Dados para Ordenação:
Os dados são elementos essenciais para a análise deste trabalho, uma vez que eles permitem verificar a eficácia dos algoritmos estudados. Assim, foram desenvolvidas duas versões para cada programa de ordenação: uma para obter os dados a partir de arquivos de texto e outra para gerar números aleatórios. Dessa forma, busca-se proporcionar maior embasamento e rigor à análise, aproximando-se de resultados mais precisos e consistentes.
1.1 Obtenção de Dados:
Para a obtenção dos dados, foram utilizadas informações fornecidas pelo professor, organizadas em uma

pasta principal que contém subpastas com conjuntos de dados de diferentes tamanhos (dez, cem, mil, dez mil, cinquenta mil, cem mil, quinhentos mil e cinco mil). Cada subpasta possui versões com dados duplicados e sem duplicados, permitindo uma análise mais abrangente. Dentro delas, possuíam mais outras cinco opções, aleatório, côncavo-decresce cresce, convexo-cresce decresce, crescente e decrescente, todos, continham dez arquivos de dados, cada uma tendo a característica descrita pela pasta , como pode ser visto na imagem abaixo.

Figura 1 ? Pasta com dados e caminho mostrando algumas das pastas

Fonte: De autoria própria.

No entanto, era necessário fazer o sistema ter a capacidade de leitura desses arquivos que até então ainda era impossível, portanto, foi usado a estrutura FILE, contida dentro da biblioteca de entradas e saídas padrão da linguagem de programação C (stdio.h). Por haver certas dificuldades para fazer com que o sistema percorre o caminho até onde era encontrado o arquivo, todo o processo de testes foi feito com os arquivos dentro da mesma pasta do programa.

A leitura do acervo de informações, é feita da seguinte maneira, cria-se uma variável de ponteiro, capaz de armazenar elementos de maneira dinâmica, ela faz a leitura dos arquivos, verificando também se é possível ou não a alocação de memória, caso tudo ocorra da maneira correta, uma função de loop, irá transferir todos os números guardados no arquivo para o vetor criado pelo ponteiro, até ele não ser capaz de encontrar mais dados, a figura abaixo mostra o trecho do código que faz todo esse processo.

Figura 2 ? Trecho do sistema de obtenção de dados.

Fonte: De autoria própria.

1.2 Geração dos dados

Em contrapartida, na geração de dados, foi utilizado a função rand(), da qual retornará um número-pseudo-aleatório, é dito como pseudo, porque ele não é completamente aleatório, já que é feito um cálculo especial usando em base um número padrão que é tomado como semente para gerar os seguintes, tendo como padrão o um. Há uma outra forma de se gerar esses dados, no entanto, seria utilizado o comando srand(), este teria uma semente diferente fazendo com que os números pudessem ser mais diferentes entre si, porém ele não foi utilizado no programa.

Conforme visto anteriormente, fora utilizado a função rand, para a geração de dados, porém é necessário um loop que possa ser capaz de criar esses números até o tamanho limite indicado pelo desenvolvedor. Consequentemente, foi usado um loop for, que fará a tarefa repetidas vezes até alcançar o tamanho limite indicado, a figura abaixo mostra o trecho do programa descrito acima:

Figura 3 ? Trecho do sistema de geração de dados

Fonte: De autoria própria

Processo de Ordenação de Dados:

Por outro lado, há o processo de ordenação, neste trabalho foram citadas e utilizadas três das principais, geralmente, as mais usadas, são elas, Quick Sort, Heap Sort, e Merge Sort, cada uma possuindo seu método específico de organizar os dados, mas todas as escolhidas, são consideradas as mais rápidas dentre as mais populares.

Cada algoritmo possui uma complexidade de tempo e de espaço, que permite prever, respectivamente, o tempo necessário para a execução do algoritmo e a quantidade de memória exigida para armazenar as estruturas necessárias durante o processamento.

O uso desses cálculos é extremamente importante para ver se é possível ou não o uso desse programa na máquina que será testada, caso contrário, diversos tipos de erros podem acabar ocorrendo, afetando diretamente o resultado.

2.1 Quick Sort:

O algoritmo Quick Sort, traduzido como ?ordenação rápida?, é conhecido por sua eficiência e velocidade na ordenação de dados. Sua eficiência se deve à estratégia de ?Divisão e Conquista?, que divide o vetor em partes menores, organiza cada parte individualmente e, em seguida, combina as segmentações ordenadas para obter o resultado.

Segundo Cormen et al. (2012), essa abordagem é especialmente eficiente em algoritmos de ordenação, pois permite reduzir o tempo de execução para uma complexidade de O(n log n), na média dos casos.

No entanto, para poder dividir, ele precisa encontrar um ponto para dividir, essa necessidade é resolvida após a escolha de um elemento na lista como um pivô, este número será o valor de referência usado para dividir a lista em duas partes, elementos maiores, e menores.

Figura 4 ? Trecho do algoritmo Quick Sort realizando o loop de partição do vetor

Fonte: De própria autoria

Assim como na figura acima, o algoritmo então particiona o vetor, fazendo com que os elementos menores que o pivô se vão para a esquerda e os maiores fiquem à sua direita, esse processo é feito recursivamente para os subvetores decorrentes. Este processo continua a ocorrer até que a lista possua no máximo um elemento, pois assim, ela já estaria ordenada.

A figura abaixo mostra o todo processo descrito anteriormente, dentro do sistema computacional.

Figura 5 ? Algoritmo de ordenação QuickSort

Fonte: De própria autoria

Já a complexidade de tempo média dele é representada pela fórmula:

O(n log n) (1)

Isto significa, que, ele realiza um número de comparações e movimentações que equivalem à fórmula (1),

onde n é o tamanho da lista a ser ordenada. Portanto, este é um dos motivos que faz este algoritmo ser tão eficiente, o seu tempo relativo já é suficientemente bom.
E por último, ele não possui cálculo de armazenamento, porque o Quick Sort é um algoritmo ?in-place?, ou seja, não requer memória adicional para armazenar as estruturas temporárias durante a ordenação. Tornando-o extremamente eficiente em termos de uso de memória.
2.2 Heap Sort:
O algoritmo Heap Sort funciona como uma árvore binária especial, na qual o valor de cada pai é maior ou igual ao valor de seus filhos (max-heap) ou menor ou igual (min-heap). No contexto do heapsort, é usado o max-folhas (max-heap), ou seja, ele irá tentar buscar o maior valor e colocá-lo no topo da árvore, e sempre seguindo dessa maneira até conseguir ordenar corretamente os elementos guardados.
Primeiramente, o algoritmo começa a iniciar o processo de transformação do vetor em uma árvore, sempre buscando o maior número e colocando-o ao topo da árvore, o funcionamento é parecido com o de uma pirâmide, esta figura abaixo representa visualmente como é o funcionamento.
Após o maior número atingir o topo da árvore, ele é movido para a posição final da lista e removido da árvore, então ocorre uma reorganização entre todas as folhas para levar o maior número para o topo novamente, para isso que ocorra até todos os elementos serem levados para o vetor ao ponto de ele estar completo, com todos os seus dados de volta.
Em suma, o algoritmo Heap, cria uma árvore binária e realiza uma constante busca para levar todos os números ao topo para que assim consigam ser organizados do maior para o menor. Em relação à complexidade de tempo médio, o cálculo do algoritmo Heap Sort é o mesmo que do Quick Sort (1), portanto possui as mesmas qualidades dele, sendo tão rápido quanto ele.
Assim como o tempo, sua complexidade de espaço também é a mesma do Quick, os dois são considerados in-place, ele usa o mesmo vetor para armazenar os elementos e para o processo de ordenação, sendo extremamente eficiente em situações em que se há pouca memória no computador. A figura abaixo mostra o algoritmo do Heap Sort por completo.
Figura 7 ? Algoritmo de ordenação Heap Sort

Fonte: De própria autoria
2.3 Merge Sort:
O Merge Sort se assemelha, muito com o algoritmo Quick Sort, seja por conta da sua estratégia, mas também por conta de sua versatilidade, eficiência e velocidade em organizar um vetor desorganizado.
Em comparação, os dois utilizam a estratégia, divisão e conquista, porém, uma grande diferença acaba os diferenciando e mudando radicalmente, isto é, a aplicação dessa estratégia é diferente para cada um deles .
Enquanto o algoritmo Quick Sort, acaba necessitando de um pivô, o Merge não se prende a isso, pois ele divide a lista em parte iguais e em seguida, combina essas partes ordenadas, completando a ordenação por completo. Em suma, um deles encontra o meio e divide e o outro escolhe um pivô, tornando o que separa no meio, o melhor, principalmente nos piores casos, pois, o pivô escolhido pode acabar não sendo o do meio, dificultando bastante e aumentando a complexidade de tempo para a seguinte fórmula:
$O(n^2)(2)$
Por outro lado, o Merge Sort continua constante mesmo em situações em que existem muitos elementos a serem ordenados, também não dependendo da sorte, facilitando ainda mais a organização, a figura 9 mostra uma representação de como é realizada a separação:
Figura 8 - Algoritmo de ordenação Merge Sort

Fonte: De autoria própria

A figura acima mostra como é feita a divisão recursiva do Merge Sort, no programa usado neste trabalho. Como citado, a complexidade de tempo do algoritmo é sempre constante, em todos os casos, por conta de como ele aplica a estratégia, então ele mantém a fórmula (1) sempre.

No entanto, a complexidade de espaço, acaba sendo levemente diferente, pois ele possui um auxiliar que acaba consumindo memória, fazendo-o não se tornar in-place. Na figura 8, podemos ver a variável ?t? sendo usada como vetor auxiliar. Assim a fórmula de espaço acaba sendo a seguinte, onde n é o número de elementos:

O(n)(3)

Listagem dos Valores Antes e Depois da Ordenação:

A análise será feita lendo o mesmo arquivo de texto, um de dez dados duplicados, para ser possível diferenciar a forma que cada um ordena, explicando seus passos e métodos.

3.1 Quick Sort:

Figura 9 ? Reposta dada pelo algoritmo de ordenação Quick Sort

Fonte: De autoria própria

Na figura acima, é possível ver como funciona o processo de ordenação do algoritmo Quick Sort lendo o arquivo, no entanto, o código acaba tendo certos problemas em mostrar o elemento, como é possível ver na terceira linha de passos imprimida, o último elemento mostra um endereço de memória, por conta de uma das recursividades acabar utilizando o endereço, fazendo a impressão se tornar confusa.

Contudo, analisando algumas partes compreensíveis da imagem, pode-se concluir que os passos da ordenação prosseguem dividindo, utilizando o pivô escolhido que acaba mudando após um certo período de impressões, pois ele vai ordenando os subvetores restantes, utilizando recursividade e economizando na alocação de memória.

3.2 Heap Sort:

Figura 10 - Resposta dada pelo algoritmo de ordenação Heap Sort

Fonte: De própria autoria

Assim como explicado, o maior valor é sempre levado para o fim do vetor, na figura acima, é mais facilmente visível como é feita essa ação. À medida que se progride, é visível a sucessão do processo, com o algoritmo mostrando a movimentação dos maiores elementos para a parte mais à extremidade esquerda, então, após percorrer por todas as folhas, move-as para o fim do vetor.

Por conta de ele ser uma árvore, as impressões acabam sendo um pouco maior, porém, é visível, em comparação com o Quick que por sua vez, acaba mostrando os endereços de memória.

3.3 Merge Sort:

Figura 11 - Resposta dada pelo algoritmo de ordenação Merge Sort

Fonte: De autoria própria

É imprimido de maneira bem similar, como o Quick Sort, porém de maneira visível mostrando o número invés do endereço, isso ocorre, pois, a impressão está ocorrendo em um momento diferente do Quick, neste caso, ela está após a transferência do vetor temporário, o que faz com que não seja imprimida o

endereço de memória.
Portanto, é mais perceptível como funciona o procedimento de divisão e conquista, vendo como o 958 é lentamente passado para o meio, até aquele momento ele é considerado o maior número da primeira metade fazendo-o fica naquela posição e então cada metade vai se organizando.
Apresentação dos Resultados Comparativos de Performance:
Esta seção apresenta uma análise de resultados entre os algoritmos utilizando quatro tipos de cenários diferentes entre eles, podendo ser duplicados ou não. Cada um dos programas foi avaliado quanto ao tempo médio de execução, visando descobrir o melhor, dependendo em cada ocasião. Os testes foram feitos com arquivos de texto com 50 mil dados e os gerados tiveram 100 mil amostras.
4.1 Medição:
Para obter os resultados, foi necessário o uso da função clock() da biblioteca da linguagem de programação C, da qual pode medir, os milissegundos, com mais precisão, o momento exato que é executado o algoritmo de ordenação, sem contar o tempo de criação de variáveis, alocação de memória e obtenção ou geração de dados.
Desta maneira, foram obtidos os dados com o tempo médio em (ms) apresentados na seguinte tabela:
Tabela 1 ? Comparação do Tempo Médio de Execução dos Algoritmos com Dados Duplicados
Fonte: De autoria própria
Tabela 2 - Comparação do Tempo Médio de Execução dos Algoritmos sem Dados Duplicados
Fonte: De autoria própria
Com estas tabelas, é possível observar que, em situações de dados espalhados de forma aleatória, o algoritmo Heap Sort é um dos piores no quesito tempo, perdendo por poucos milissegundos.
No entanto, em situações em que os elementos são espalhados de forma côncava, caso os dados sejam duplicados, o Quick acaba perdendo, mas em situações sem duplicidade, o Heap acaba tendo uma desvantagem de 515 milissegundos, uma diferença que quando comparada em situações de mais dados, faz com que ele não seja tão eficiente.
Toda essa diferença entre os dois, ocorre devido a diferença da estratégia entre eles, enquanto um necessita fazer uma árvore e deve percorrê-la, o outro apenas se divide e compara até conseguir se ordenar.
Apesar do Quick conseguir ser mais veloz que o Heap nesses cenários, existem muitos casos em que a sua estratégia não funciona tão bem, por exemplo, com grandes quantidades de números, esse cenário faz com que o Quick se torne extremamente instável nessas ocasiões. Pois, a forma em que ele exerce seu plano, depende muito de o pivô estar mais perto do centro ou não, podendo facilitar e muito em arquivos de dados pequenos.
Algo que não ocorre com o Merge Sort, que por sua vez, acaba brilhando nessas situações já que a aplicação da estratégia Divisão e Conquista é diferente, fazendo uma divisão por 2, não precisando ser tão dependente de achar ou não um pivô mais próximo do centro do vetor, já que quando dividido, sempre cairá ao meio.
No entanto, o algoritmo Merge Sort acaba tendo uma grande desvantagem em comparação aos outros dois algoritmos, o armazenamento de espaço, o que torna ele estável na maior parte das situações, tende a deixá-lo lento, pois ocorre a transferência de números do vetor temporário ao original.

Ref Biblio:
CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: Teoria e Prática. 3. ed. Rio de Janeiro: Elsevier, 2012.
SEDGEWICK, Robert; WAYNE, Kevin. Algorithms. 4th ed. Boston: Addison-Wesley, 2011.
MANZANO, José Augusto N. G.; OLIVEIRA, Jayr Figueiredo de. Estruturas de Dados e Algoritmos. 3. ed. São Paulo: Érica, 2011.
SIPSER, Michael. Introduction to the Theory of Computation. 3rd ed. Boston: Cengage Learning, 2012.
KNUTH, Donald E. The Art of Computer Programming: Sorting and Searching. 2nd ed. Massachusetts: Addison-Wesley, 1998.
DEITEL, H.; DEITEL, P. C: Como Programar. 7. ed. São Paulo: Pearson, 2010.
Resultados e Discussão

Como visto previamente neste documento, cada tipo de método de ordenação tem características próprias que o tornam altamente eficiente em algumas ocasiões e altamente ineficiente em outras. No entanto torna-se necessário uma análise mais profunda destes dados para um melhor entendimento dos mesmos. Para tal feito, foram coletadas mais de 800 amostras visando um melhor entendimento de cada método de ordenação.
Todas as amostras foram coletadas tendo como princípio a meta de 400 dados para cada tipo de método de ordenação, separados em dois subgrupos de 200 dados, respectivamente com e sem duplicidade de dados. Cada um destes subgrupos foi dividido em 5 subgrupos de 40 dados cada (aleatório, côncavo, convexo, crescente, decrescente) e um subgrupo com 100 dados(gerado pelo nosso grupo). Cada dado era composto por 50.000 números(como demonstrado abaixo em nossa imagem de autoria própria).

Ao serem analisados foi possível se tirar a seguinte tabela (autoria própria) como conclusão:

Os números com código de cores ao lado na parte inferior dos gráficos acima representam o tempo (em segundos) que cada algoritmo demorou para ser executado, já os números ao lado da palavra ?Frequência? representam a frequência de cada um dos números com código de cores ao lado.
Os gráficos acima permitem concluir diversos fatores. Por exemplo, os métodos de ordenação heap e merge têm uma variação menor do que o quick sort em termos de análise de dados. No entanto, o método merge possui uma demora muito maior para processar os dados que o quick e o heap.
Com tais dados foi possível montar uma curva normal extremamente detalhada para cada tipo de algoritmo de ordenação, com e sem duplicidade de números, e para os números também localmente gerados a partir de métodos de ordenação. Abaixos estão respectivamente as curvas normais (autoria própria) dos dados listados acima :

Os grupos de curvas normais apresentados acima são respectivamente pertencentes aos métodos de ordenação hippie, quick e merge. Com os códigos de cores preto para ordenação com duplicidade, vermelho para ordenação se duplicidade e azul para os dados coletados a partir de dados gerados localmente.
É possível notar que em cada grupo é apresentado vários símbolos com significados diversos. O símbolo ? denota a média de todos os dados coletados. O símbolo ? descreve o desvio padrão do objeto em estudo.
O número indicado no balão é o mesmo que o do símbolo X. A porcentagem que se segue no meio da curva de sino traz qual a probabilidade de haver números abaixo de X o que, por inferência, nos traz uma taxa de confiança de 95% para x ? X. Portanto pode-se afirmar que nenhum número recairá abaixo de X a não ser por acaso, e que X deve ser tomado como nosso objeto de estudo atual ignorando-se todos os tempos acima do mesmo.
Nota-se também que o método de ordenação quick está com menos dados na parte de com e sem duplicidade. Isso se deve ao fato do erro Stack overflow ter ocorrido durante a execução de ambos os dados decrescente e crescente, então se foi executado apenas uma vez cada e posto o tempo de demora para o código sair.
Com isso em mente é possível concluir que os métodos de ordenação quick e heap, são realmente mais rápidos em algumas ocasiões. No entanto, os métodos que apresentam mais estabilidade durante a execução são os métodos heap e merge. Há também a probabilidade de o método quick dar o erro Stack overflow devido a quantidade de memória ram que ocupa durante o processo de execução. Por fim, é possível concluir que o método de ordenação mais estável e o mais rápido é o heap.
Concluimos que os algoritmos de ordenação Quick Sort, Merge Sort e Heap Sort, mesmo que apresentem

complexidades de tempo semelhantes , demonstram comportamentos distintos em diferentes cenários. Algoritmos de ordenação estáveis, como o Merge Sort , são úteis em situações em que a preservação da ordem relativa é importante, como ordenação de registros em bancos de dados ou classificação por múltiplos critérios. Por outro lado, algoritmos de ordenação instáveis, como o Quicksort, podem alterar a ordem de elementos com chaves iguais durante o processo de classificação.
O Quick Sort e Heap Sort tiveram um desempenho melhor que o Merge sort sem duplicidade e com duplicidade, onde o Quick Sort se sobrepõe ao Heap Sort tendo menos frequencia, com uma diferença de 25 no tem 0,006 sem duplicidade e com duplicidade um diferença de 42.
Portanto a escolha do algoritmmo de ordenação ideal depende de diversas situações, como o tamho do conjunto de dados, os requisitos da memória ea estabilidade. No geral o Quick Sort será uma excelente opção para a maioria dos casos devido ao seu desempenho .No entanto o Heap Sort pode ser uma boa opção para grandes conjutos de dados. O Merge Sort pode ser uma boa opção quando o espaço de memória for limitado.

AlgoritmoAleatório (seg.)Côncavo (seg.)Convexo (seg.)Gerado (seg.)
Quick Sort0.005550.006450.00540,00527
Heap Sort0.0077750.0058750.00580,01695
Merge Sort0.011150.008150.0082750,011

AlgoritmoAleatório (seg.)Côncavo (seg.)Convexo (seg.)
Quick Sort0.0056750.006450.005875
Heap Sort0.00750.01160.0065
Merge Sort0.01150.00850.0085

==========

Arquivo 1: Trabalho_aps.docx (4117 termos)
Arquivo 2: https://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html (1884 termos)
Termos comuns: 43
Similaridade: 0,72%
O texto abaixo é o conteúdo do documento Trabalho_aps.docx (4117 termos)
Os termos em vermelho foram encontrados no documento
https://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html (1884 termos)

==========

UNIVERSIDADE PAULISTA

G85JFG6 ? Eric Mitchell Barbosa Durham
N078146 ? Rafael Hiro Portilho
R0238H0 ? Matheus Reis Oliveira
N4031A9 ? Humberto Joji Fukui

Ordenação de dados amazônicos
Análise de algoritmos de ordenação de dados

São Paulo
2024

Neste trabalho tem o intuito de apresentar e comparar sobre os algoritmos Merge Sort, Quicksort e Heap Sort o seu desenvolvimento, eficiencia e complexidade. Avaliaremos os resultados adquiridos por cada algoritmo e discutir a aplicação desses agoritmos em alguns tipos de situações e uma analise do tempo que consiguram resolver cada situação. Os algoritmos de ordenações desenpenham papeis fundamentais na organização e otmização de dados, tornando-se fundamental para a eficiência para operações de busca e armazenamento em sistemas em sistemas de bancos de dados e outras aplicações computacionais.

Os algoritmos Merge sort, Heap sort e Quicksort conhecidos por sua eficiência e são amplamente utilizados em diversas linguagens de programação e bibliotecas. Cada um deles possui suas próprias características e vantagens, que serão exploradas em detalhes neste trabalho.

Quick sort se destaca por sua escolha programada de pivôs e rearranjo recursivo dos elementos, apresenta uma complexidade média de O(n log n), por outro lado o Merge sort conhecido por sua eficiencia na ordenação por divisão e conquista, destaca-se pela sua eficiência e estabilidade, dando uma ordenação eficaz dos elementos com complexidade O(n log n).O Heap sort, oferece um ordenação eficiente com complexidade O(n log n), destacando-se pela sua robustez.

Ao estudar esses algoritimos, temos como objetivo mostrar uma visão abrangente de suas características, desempenho e enetendimento a diferentes contextos, auxiliando na seleção do algoritmo mais adequado de acordo com a necessidade especifica de cada situação aplicada a eles.

2 REFERENCIAL TEÓRICO

2.1 QUICKSORT

Esse algoritmo usa a técnica da "divisão e conquista", que consiste na divisão recursiva de um problema em vários menores. No caso, o QuickSort escolhe um elemento do conjunto como pivô, sendo esse geralmente o primeiro elemento. Daí, reorganiza-se os demais elementos de forma que apenas os elementos menores que o pivô se situam à esquerda deste, assim formando dois subconjuntos em cada lado do pivô: um com elementos menores e outro com elementos maiores.

Na configuração resultante, o pivô passa a estar necessariamente na posição correta, e é então fixado. O processo descrito acima é repetido nos subconjuntos formados, definindo novo pivôs em cada. Isso continua recursivamente, formando conjuntos cada vez menores até o surgimento de conjuntos unitários, conforme esquematizado na figura 1.

O QuickSort é considerado relativamente eficiente dentre os algoritmos de ordenação mais usados. Entretanto, há algumas desvantagens principais como o uso de memória necessário, que aumenta consideravelmente com o número de elementos do vetor de interesse devido ao caráter recursivo do algoritmo.
Além disso, o QuickSort pode tornar-se bem ineficiente quando apresentado com vetores em certos arranjos. Por exemplo, podemos citar o caso quando a lista já está maiormente ordenada em ordem crescente ou decrescente, na qual o algoritmo não poderá fazer rearranjos significativos.

2.2 MERGESORT

Também se enquadrando na categoria de ?divisão e conquista?, o MergeSort divide o vetor em duas partes aproximadamente iguais, até a formação de conjuntos unitários similarmente ao QuickSort, antes de intercalar-se de volta na ordem reversa, até retornar ao tamanho inicial do vetor de interesse.
O processo de intercalação entre duas listas é feito inserindo os elementos de cada num vetor auxiliar, que é subsequentemente sobrescrito nos dois conjuntos iniciais. Compara-se um elemento de cada conjunto, inserindo sempre o menor (caso não haja valores iguais), começando com os respectivos primeiros elementos e avançando conforme a inserção, de forma independente.
Realizando-se essa junção a partir dos conjuntos unitários formados, os subconjuntos a serem intercalados sempre estão organizados internamente, facilitando a organização final como um todo.

Similarmente ao QuickSort, esse algoritmo exige uso de memória além do próprio vetor de interesse por ser recursivo.

2.3 HEAPSORT

É um algoritmo baseado na árvore binária máxima. Nesse tipo de sequência, cada elemento pai tem no máximo dois elementos filhos associados de valores igual ao menor que o pai. Os elementos filhos, por sua vez, podem ter filhos próprios, assim formando uma estrutura similar a uma árvore, onde o primeiro elemento é considerado a raiz da estrutura como um todo.
Aqui, utiliza-se repetidamente um método chamado de Heapify, descrito como segue: um elemento parente X é comparado com seus filhos (ou filho único). Se não existe filho superior ao X, o método é finalizado. Caso contrário, trocam-se as posições do X e do filho com valor superior, e esse processo continua na nova posição de X. Note que caso haja dois filhos com valor distintos maior que X, o maior será sempre escolhido.
No processo do HeapSort, incialmente o vetor é reorganizado até que se constitua numa árvore binária máxima. Isso é feito aplicando Heapify em todos os elementos parentes (com pelo menos um filho), em ordem decrescente. Ao final, isso é suficiente em tornar o vetor completo numa árvore máxima válida.

Em seguida, trocamos a raiz com o último elemento da árvore, fixando a raiz original na posição nova e desconsiderando-o do resto da árvore. Daí Heapify é chamado na nova raiz, e a troca é realizada novamente até todos os elementos serem eliminados da árvore, momento na qual o vetor passa a ser necessariamente ordenado.

Como exemplo, considere o vetor na esquerda da figura 3, que não constitui numa árvore válida ainda. Aplicando o HeapSort, chama-se o Heapify nos seguintes elementos, conforme a sequência:
Elemento 8: Como esse já é maior que os filhos, nenhuma troca é realizada;
Elemento 1: Existe filho com valor superior. Assim, troca-se 1 com 9, o filho de maior valor;
Elemento 7: Conforme a lógica do passo anterior, 7 é trocado com 8. Os novos filhos de 7, 4 e 6, são inferiores;
Elemento 3: Nesse ponto, os filhos passam a ser 8 e 9. Pela mesma lógica, troca-se 3 com 9, tornando 5 e 1 os novos filhos de 3. Assim, troca-se 3 com 5.
Assim, obtemos a árvore da direita da figura 3. Pelo outro lado, a figura 4 apresenta o passo seguinte, que envolve a eliminação do 9 da árvore.

Aqui, 9 tem sua posição trocada com 6, o elemento final da árvore. Em seguida, 9 é retirado da árvore e fixada na posição correta do vetor, fazendo a árvore resultante inválida mais uma vez. Daí, chama-se o Heapify no novo nó 6, que resultará na troca entre 6 e 8, e entre 6 e 7, para restaurar a propriedade da árvore. Com isso, 6 é trocado com 4, a nova posição final da árvore, e assim por diante.
O HeapSort não necessita de memória adicional além da própria função e do vetor de interesse, pois todos os rearranjos dos elementos são realizados dentro do vetor.

Este trabalho tem como objetivo desenvolver um sistema computacional para avaliar o desempenho de diversos algoritmos de ordenação de dados. Considerando a importância de algoritmos eficientes na análise e manipulação de grandes volumes de dados, o sistema proposto visa fornecer uma análise comparativa de desempenho utilizando dados internos e externos. Através do sistema, será possível mensurar o tempo de execução de cada algoritmo em diferentes condições, permitindo identificar suas vantagens e limitações.
Geração e/ou Obtenção de Dados para Ordenação:
Os dados são elementos essenciais para a análise deste trabalho, uma vez que eles permitem verificar a eficácia dos algoritmos estudados. Assim, foram desenvolvidas duas versões para cada programa de ordenação: uma para obter os dados a partir de arquivos de texto e outra para gerar números aleatórios. Dessa forma, busca-se proporcionar maior embasamento e rigor à análise, aproximando-se de resultados mais precisos e consistentes.
1.1 Obtenção de Dados:
Para a obtenção dos dados, foram utilizadas informações fornecidas pelo professor, organizadas em uma pasta principal que contém subpastas com conjuntos de dados de diferentes tamanhos (dez, cem, mil, dez

mil, cinquenta mil, cem mil, quinhentos mil e cinco mil). Cada subpasta possui versões com dados duplicados e sem duplicados, permitindo uma análise mais abrangente. Dentro delas, possuíam mais outras cinco opções, aleatório, côncavo-decresce cresce, convexo-cresce decresce, crescente e decrescente, todos, continham dez arquivos de dados, cada uma tendo a característica descrita pela pasta , como pode ser visto na imagem abaixo.

Figura 1 ? Pasta com dados e caminho mostrando algumas das pastas

Fonte: De autoria própria.

No entanto, era necessário fazer o sistema ter a capacidade de leitura desses arquivos que até então ainda era impossível, portanto, foi usado a estrutura FILE, contida dentro da biblioteca de entradas e saídas padrão da linguagem de programação C (stdio.h). Por haver certas dificuldades para fazer com que o sistema percorre o caminho até onde era encontrado o arquivo, todo o processo de testes foi feito com os arquivos dentro da mesma pasta do programa.

A leitura do acervo de informações, é feita da seguinte maneira, cria-se uma variável de ponteiro, capaz de armazenar elementos de maneira dinâmica, ela faz a leitura dos arquivos, verificando também se é possível ou não a alocação de memória, caso tudo ocorra da maneira correta, uma função de loop, irá transferir todos os números guardados no arquivo para o vetor criado pelo ponteiro, até ele não ser capaz de encontrar mais dados, a figura abaixo mostra o trecho do código que faz todo esse processo.

Figura 2 ? Trecho do sistema de obtenção de dados.

Fonte: De autoria própria.

1.2 Geração dos dados

Em contrapartida, na geração de dados, foi utilizado a função rand(), da qual retornará um número-pseudo-aleatório, é dito como pseudo, porque ele não é completamente aleatório, já que é feito um cálculo especial usando em base um número padrão que é tomado como semente para gerar os seguintes, tendo como padrão o um. Há uma outra forma de se gerar esses dados, no entanto, seria utilizado o comando srand(), este teria uma semente diferente fazendo com que os números pudessem ser mais diferentes entre si, porém ele não foi utilizado no programa.

Conforme visto anteriormente, fora utilizado a função rand, para a geração de dados, porém é necessário um loop que possa ser capaz de criar esses números até o tamanho limite indicado pelo desenvolvedor. Consequentemente, foi usado um loop for, que fará a tarefa repetidas vezes até alcançar o tamanho limite indicado, a figura abaixo mostra o trecho do programa descrito acima:

Figura 3 ? Trecho do sistema de geração de dados

Fonte: De autoria própria

Processo de Ordenação de Dados:

Por outro lado, há o processo de ordenação, neste trabalho foram citadas e utilizadas três das principais, geralmente, as mais usadas, são elas, Quick Sort, Heap Sort, e Merge Sort, cada uma possuindo seu método específico de organizar os dados, mas todas as escolhidas, são consideradas as mais rápidas dentre as mais populares.

Cada algoritmo possui uma complexidade de tempo e de espaço, que permite prever, respectivamente, o tempo necessário para a execução do algoritmo e a quantidade de memória exigida para armazenar as estruturas necessárias durante o processamento.

O uso desses cálculos é extremamente importante para ver se é possível ou não o uso desse programa na máquina que será testada, caso contrário, diversos tipos de erros podem acabar ocorrendo, afetando diretamente o resultado.

2.1 Quick Sort:

O algoritmo Quick Sort, traduzido como ?ordenação rápida?, é conhecido por sua eficiência e velocidade na ordenação de dados. Sua eficiência se deve à estratégia de ?Divisão e Conquista?, que divide o vetor em partes menores, organiza cada parte individualmente e, em seguida, combina as segmentações ordenadas para obter o resultado.
Segundo Cormen et al. (2012), essa abordagem é especialmente eficiente em algoritmos de ordenação, pois permite reduzir o tempo de execução para uma complexidade de O(n log n), na média dos casos.
No entanto, para poder dividir, ele precisa encontrar um ponto para dividir, essa necessidade é resolvida após a escolha de um elemento na lista como um pivô, este número será o valor de referência usado para dividir a lista em duas partes, elementos maiores, e menores.

Figura 4 ? Trecho do algoritmo Quick Sort realizando o loop de partição do vetor
Fonte: De própria autoria
Assim como na figura acima, o algoritmo então particiona o vetor, fazendo com que os elementos menores que o pivô se vão para a esquerda e os maiores fiquem à sua direita, esse processo é feito recursivamente para os subvetores decorrentes. Este processo continua a ocorrer até que a lista possua no máximo um elemento, pois assim, ela já estaria ordenada.
A figura abaixo mostra o todo processo descrito anteriormente, dentro do sistema computacional.

Figura 5 ? Algoritmo de ordenação QuickSort

Fonte: De própria autoria
Já a complexidade de tempo média dele é representada pela fórmula:
O(n log n) (1)
Isto significa, que, ele realiza um número de comparações e movimentações que equivalem à fórmula (1), onde n é o tamanho da lista a ser ordenada. Portanto, este é um dos motivos que faz este algoritmo ser

tão eficiente, o seu tempo relativo já é suficientemente bom.
E por último, ele não possui cálculo de armazenamento, porque o Quick Sort é um algoritmo ?in-place?, ou seja, não requer memória adicional para armazenar as estruturas temporárias durante a ordenação. Tornando-o extremamente eficiente em termos de uso de memória.
2.2 Heap Sort:
O algoritmo Heap Sort funciona como uma árvore binária especial, na qual o valor de cada pai é maior ou igual ao valor de seus filhos (max-heap) ou menor ou igual (min-heap). No contexto do heapsort, é usado o max-folhas (max-heap), ou seja, ele irá tentar buscar o maior valor e colocá-lo no topo da árvore, e sempre seguindo dessa maneira até conseguir ordenar corretamente os elementos guardados.
Primeiramente, o algoritmo começa a iniciar o processo de transformação do vetor em uma árvore, sempre buscando o maior número e colocando-o ao topo da árvore, o funcionamento é parecido com o de uma pirâmide, esta figura abaixo representa visualmente como é o funcionamento.
Após o maior número atingir o topo da árvore, ele é movido para a posição final da lista e removido da árvore, então ocorre uma reorganização entre todas as folhas para levar o maior número para o topo novamente, para isso que ocorra até todos os elementos serem levados para o vetor ao ponto de ele estar completo, com todos os seus dados de volta.
Em suma, o algoritmo Heap, cria uma árvore binária e realiza uma constante busca para levar todos os números ao topo para que assim consigam ser organizados do maior para o menor. Em relação à complexidade de tempo médio, o cálculo do algoritmo Heap Sort é o mesmo que do Quick Sort (1), portanto possui as mesmas qualidades dele, sendo tão rápido quanto ele.
Assim como o tempo, sua complexidade de espaço também é a mesma do Quick, os dois são considerados in-place, ele usa o mesmo vetor para armazenar os elementos e para o processo de ordenação, sendo extremamente eficiente em situações em que se há pouca memória no computador. A figura abaixo mostra o algoritmo do Heap Sort por completo.
Figura 7 ? Algoritmo de ordenação Heap Sort

Fonte: De própria autoria
2.3 Merge Sort:
O Merge Sort se assemelha, muito com o algoritmo Quick Sort, seja por conta da sua estratégia, mas também por conta de sua versatilidade, eficiência e velocidade em organizar um vetor desorganizado.
Em comparação, os dois utilizam a estratégia, divisão e conquista, porém, uma grande diferença acaba os diferenciando e mudando radicalmente, isto é, a aplicação dessa estratégia é diferente para cada um deles .
Enquanto o algoritmo Quick Sort, acaba necessitando de um pivô, o Merge não se prende a isso, pois ele divide a lista em parte iguais e em seguida, combina essas partes ordenadas, completando a ordenação por completo. Em suma, um deles encontra o meio e divide e o outro escolhe um pivô, tornando o que separa no meio, o melhor, principalmente nos piores casos, pois, o pivô escolhido pode acabar não sendo o do meio, dificultando bastante e aumentando a complexidade de tempo para a seguinte fórmula:
$O(n^2)(2)$
Por outro lado, o Merge Sort continua constante mesmo em situações em que existem muitos elementos a serem ordenados, também não dependendo da sorte, facilitando ainda mais a organização, a figura 9 mostra uma representação de como é realizada a separação:
Figura 8 - Algoritmo de ordenação Merge Sort

Fonte: De autoria própria
A figura acima mostra como é feita a divisão recursiva do Merge Sort, no programa usado neste trabalho. Como citado, a complexidade de tempo do algoritmo é sempre constante, em todos os casos, por conta de como ele aplica a estratégia, então ele mantém a fórmula (1) sempre.
No entanto, a complexidade de espaço, acaba sendo levemente diferente, pois ele possui um auxiliar que acaba consumindo memória, fazendo-o não se tornar in-place. Na figura 8, podemos ver a variável ?t? sendo usada como vetor auxiliar. Assim a fórmula de espaço acaba sendo a seguinte, onde n é o número de elementos:
O(n)(3)
Listagem dos Valores Antes e Depois da Ordenação:
A análise será feita lendo o mesmo arquivo de texto, um de dez dados duplicados, para ser possível diferenciar a forma que cada um ordena, explicando seus passos e métodos.
3.1 Quick Sort:
Figura 9 ? Reposta dada pelo algoritmo de ordenação Quick Sort

Fonte: De autoria própria
Na figura acima, é possível ver como funciona o processo de ordenação do algoritmo Quick Sort lendo o arquivo, no entanto, o código acaba tendo certos problemas em mostrar o elemento, como é possível ver na terceira linha de passos imprimida, o último elemento mostra um endereço de memória, por conta de uma das recursividades acabar utilizando o endereço, fazendo a impressão se tornar confusa.
Contudo, analisando algumas partes compreensíveis da imagem, pode-se concluir que os passos da ordenação prosseguem dividindo, utilizando o pivô escolhido que acaba mudando após um certo período de impressões, pois ele vai ordenando os subvetores restantes, utilizando recursividade e economizando na alocação de memória.
3.2 Heap Sort:
Figura 10 - Resposta dada pelo algoritmo de ordenação Heap Sort

Fonte: De própria autoria
Assim como explicado, o maior valor é sempre levado para o fim do vetor, na figura acima, é mais facilmente visível como é feita essa ação. À medida que se progride, é visível a sucessão do processo, com o algoritmo mostrando a movimentação dos maiores elementos para a parte mais à extremidade esquerda, então, após percorrer por todas as folhas, move-as para o fim do vetor.
Por conta de ele ser uma árvore, as impressões acabam sendo um pouco maior, porém, é visível, em comparação com o Quick que por sua vez, acaba mostrando os endereços de memória.
3.3 Merge Sort:

Figura 11 - Resposta dada pelo algoritmo de ordenação Merge Sort

Fonte: De autoria própria
É imprimido de maneira bem similar, como o Quick Sort, porém de maneira visível mostrando o número invés do endereço, isso ocorre, pois, a impressão está ocorrendo em um momento diferente do Quick, neste caso, ela está após a transferência do vetor temporário, o que faz com que não seja imprimida o endereço de memória.

Portanto, é mais perceptível como funciona o procedimento de divisão e conquista, vendo como o 958 é lentamente passado para o meio, até aquele momento ele é considerado o maior número da primeira metade fazendo-o fica naquela posição e então cada metade vai se organizando.
Apresentação dos Resultados Comparativos de Performance:
Esta seção apresenta uma análise de resultados entre os algoritmos utilizando quatro tipos de cenários diferentes entre eles, podendo ser duplicados ou não. Cada um dos programas foi avaliado quanto ao tempo médio de execução, visando descobrir o melhor, dependendo em cada ocasião. Os testes foram feitos com arquivos de texto com 50 mil dados e os gerados tiveram 100 mil amostras.
4.1 Medição:
Para obter os resultados, foi necessário o uso da função clock() da biblioteca da linguagem de programação C, da qual pode medir, os milissegundos, com mais precisão, o momento exato que é executado o algoritmo de ordenação, sem contar o tempo de criação de variáveis, alocação de memória e obtenção ou geração de dados.
Desta maneira, foram obtidos os dados com o tempo médio em (ms) apresentados na seguinte tabela:
Tabela 1 ? Comparação do Tempo Médio de Execução dos Algoritmos com Dados Duplicados
Fonte: De autoria própria
Tabela 2 - Comparação do Tempo Médio de Execução dos Algoritmos sem Dados Duplicados
Fonte: De autoria própria
Com estas tabelas, é possível observar que, em situações de dados espalhados de forma aleatória, o algoritmo Heap Sort é um dos piores no quesito tempo, perdendo por poucos milissegundos.
No entanto, em situações em que os elementos são espalhados de forma côncava, caso os dados sejam duplicados, o Quick acaba perdendo, mas em situações sem duplicidade, o Heap acaba tendo uma desvantagem de 515 milissegundos, uma diferença que quando comparada em situações de mais dados, faz com que ele não seja tão eficiente.
Toda essa diferença entre os dois, ocorre devido a diferença da estratégia entre eles, enquanto um necessita fazer uma árvore e deve percorrê-la, o outro apenas se divide e compara até conseguir se ordenar.
Apesar do Quick conseguir ser mais veloz que o Heap nesses cenários, existem muitos casos em que a sua estratégia não funciona tão bem, por exemplo, com grandes quantidades de números, esse cenário faz com que o Quick se torne extremamente instável nessas ocasiões. Pois, a forma em que ele exerce seu plano, depende muito de o pivô estar mais perto do centro ou não, podendo facilitar e muito em arquivos de dados pequenos.
Algo que não ocorre com o Merge Sort, que por sua vez, acaba brilhando nessas situações já que a aplicação da estratégia Divisão e Conquista é diferente, fazendo uma divisão por 2, não precisando ser tão dependente de achar ou não um pivô mais próximo do centro do vetor, já que quando dividido, sempre cairá ao meio.
No entanto, o algoritmo Merge Sort acaba tendo uma grande desvantagem em comparação aos outros dois algoritmos, o armazenamento de espaço, o que torna ele estável na maior parte das situações, tende a deixá-lo lento, pois ocorre a transferência de números do vetor temporário ao original.

Ref Biblio:
CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: Teoria e Prática. 3. ed. Rio de Janeiro: Elsevier, 2012.
SEDGEWICK, Robert; WAYNE, Kevin. Algorithms. 4th ed. Boston: Addison-Wesley, 2011.
MANZANO, José Augusto N. G.; OLIVEIRA, Jayr Figueiredo de. Estruturas de Dados e Algoritmos. 3. ed. São Paulo: Érica, 2011.
SIPSER, Michael. Introduction to the Theory of Computation. 3rd ed. Boston: Cengage Learning, 2012.
KNUTH, Donald E. The Art of Computer Programming: Sorting and Searching. 2nd ed. Massachusetts: Addison-Wesley, 1998.
DEITEL, H.; DEITEL, P. C: Como Programar. 7. ed. São Paulo: Pearson, 2010.
Resultados e Discussão

Como visto previamente neste documento, cada tipo de método de ordenação tem características próprias que o tornam altamente eficiente em algumas ocasiões e altamente ineficiente em outras. No entanto torna-se necessário uma análise mais profunda destes dados para um melhor entendimento dos mesmos. Para tal feito, foram coletadas mais de 800 amostras visando um melhor entendimento de cada método de ordenação.

Todas as amostras foram coletadas tendo como princípio a meta de 400 dados para cada tipo de método de ordenação, separados em dois subgrupos de 200 dados, respectivamente com e sem duplicidade de dados. Cada um destes subgrupos foi dividido em 5 subgrupos de 40 dados cada (aleatório, côncavo, convexo, crescente, decrescente) e um subgrupo com 100 dados(gerado pelo nosso grupo). Cada dado era composto por 50.000 números(como demonstrado abaixo em nossa imagem de autoria própria).

Ao serem analisados foi possível se tirar a seguinte tabela (autoria própria) como conclusão:

Os números com código de cores ao lado na parte inferior dos gráficos acima representam o tempo (em segundos) que cada algoritmo demorou para ser executado, já os números ao lado da palavra ?Frequência? representam a frequência de cada um dos números com código de cores ao lado.
Os gráficos acima permitem concluir diversos fatores. Por exemplo, os métodos de ordenação heap e merge têm uma variação menor do que o quick sort em termos de análise de dados. No entanto, o método merge possui uma demora muito maior para processar os dados que o quick e o heap.
Com tais dados foi possível montar uma curva normal extremamente detalhada para cada tipo de algoritmo de ordenação, com e sem duplicidade de números, e para os números também localmente gerados a partir de métodos de ordenação. Abaixos estão respectivamente as curvas normais (autoria própria) dos dados listados acima :

Os grupos de curvas normais apresentados acima são respectivamente pertencentes aos métodos de ordenação hippie, quick e merge. Com os códigos de cores preto para ordenação com duplicidade, vermelho para ordenação se duplicidade e azul para os dados coletados a partir de dados gerados localmente.
É possível notar que em cada grupo é apresentado vários símbolos com significados diversos. O símbolo ? denota a média de todos os dados coletados. O símbolo ? descreve o desvio padrão do objeto em estudo.
O número indicado no balão é o mesmo que o do símbolo X. A porcentagem que se segue no meio da curva de sino traz qual a probabilidade de haver números abaixo de X o que, por inferência, nos traz uma taxa de confiança de 95% para x ? X. Portanto pode-se afirmar que nenhum número recairá abaixo de X a não ser por acaso, e que X deve ser tomado como nosso objeto de estudo atual ignorando-se todos os tempos acima do mesmo.
Nota-se também que o método de ordenação quick está com menos dados na parte de com e sem duplicidade. Isso se deve ao fato do erro Stack overflow ter ocorrido durante a execução de ambos os dados decrescente e crescente, então se foi executado apenas uma vez cada e posto o tempo de demora para o código sair.
Com isso em mente é possível concluir que os métodos de ordenação quick e heap, são realmente mais rápidos em algumas ocasiões. No entanto, os métodos que apresentam mais estabilidade durante a execução são os métodos heap e merge. Há também a probabilidade de o método quick dar o erro Stack overflow devido a quantidade de memória ram que ocupa durante o processo de execução. Por fim, é possível concluir que o método de ordenação mais estável e o mais rápido é o heap.
Concluimos que os algoritmos de ordenação Quick Sort, Merge Sort e Heap Sort, mesmo que apresentem complexidades de tempo semelhantes , demonstram comportamentos distintos em diferentes cenários.

Algoritmos de ordenação estáveis, como o Merge Sort , são úteis em situações em que a preservação da ordem relativa é importante, como ordenação de registros em bancos de dados ou classificação por múltiplos critérios. Por outro lado, algoritmos de ordenação instáveis, como o Quicksort, podem alterar a ordem de elementos com chaves iguais durante o processo de classificação.
O Quick Sort e Heap Sort tiveram um desempenho melhor que o Merge sort sem duplicidade e com duplicidade, onde o Quick Sort se sobrepõe ao Heap Sort tendo menos frequencia, com uma diferença de 25 no tem 0,006 sem duplicidade e com duplicidade um diferença de 42.
Portanto a escolha do algoritmmo de ordenação ideal depende de diversas situações, como o tamho do conjunto de dados, os requisitos da memória ea estabilidade. No geral o Quick Sort será uma excelente opção para a maioria dos casos devido ao seu desempenho .No entanto o Heap Sort pode ser uma boa opção para grandes conjutos de dados. O Merge Sort pode ser uma boa opção quando o espaço de memória for limitado.

AlgoritmoAleatório (seg.)Côncavo (seg.)Convexo (seg.)Gerado (seg.)
Quick Sort0.005550.006450.00540,00527
Heap Sort0.0077750.0058750.00580,01695
Merge Sort0.011150.008150.0082750,011

AlgoritmoAleatório (seg.)Côncavo (seg.)Convexo (seg.)
Quick Sort0.0056750.006450.005875
Heap Sort0.00750.01160.0065
Merge Sort0.01150.00850.0085

==========

Arquivo 1: Trabalho_aps.docx (4117 termos)
Arquivo 2: https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261 (4411 termos)
Termos comuns: 55
Similaridade: 0,64%
O texto abaixo é o conteúdo do documento Trabalho_aps.docx (4117 termos)
Os termos em vermelho foram encontrados no documento https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261 (4411 termos)

==========

UNIVERSIDADE PAULISTA

G85JFG6 ? Eric Mitchell Barbosa Durham
N078146 ? Rafael Hiro Portilho
R0238H0 ? Matheus Reis Oliveira
N4031A9 ? Humberto Joji Fukui

Ordenação de dados amazônicos
Análise de algoritmos de ordenação de dados

São Paulo
2024
Neste trabalho tem o intuito de apresentar e comparar sobre os algoritmos Merge Sort, Quicksort e Heap Sort o seu desenvolvimento, eficiencia e complexidade. Avaliaremos os resultados adquiridos por cada algoritmo e discutir a aplicação desses agoritmos em alguns tipos de situações e uma analise do tempo que consiguram resolver cada situação. Os algoritmos de ordenações desenpenham papeis fundamentais na organização e otmização de dados, tornando-se fundamental para a eficiência para operações de busca e armazenamento em sistemas em sistemas de bancos de dados e outras aplicações computacionais.
Os algoritmos Merge sort, Heap sort e Quicksort conhecidos por sua eficiência e são amplamente utilizados em diversas linguagens de programação e bibliotecas. Cada um deles possui suas próprias características e vantagens, que serão exploradas em detalhes neste trabalho.
Quick sort se destaca por sua escolha programada de pivôs e rearranjo recursivo dos elementos, apresenta uma complexidade média de O(n log n), por outro lado o Merge sort conhecido por sua eficiencia na ordenação por divisão e conquista, destaca-se pela sua eficiência e estabilidade, dando uma ordenação eficaz dos elementos com complexidade O(n log n).O Heap sort, oferece um ordenação eficiente com complexidade O(n log n), destacando-se pela sua robustez.
Ao estudar esses algoritimos, temos como objetivo mostrar uma visão abrangente de suas características, desempenho e enetendimento a diferentes contextos, auxiliando na seleção do algoritmo mais adequado de acordo com a necessidade especifica de cada situação aplicada a eles.

2 REFERENCIAL TEÓRICO

2.1 QUICKSORT

Esse algoritmo usa a técnica da "divisão e conquista", que consiste na divisão recursiva de um problema em vários menores. No caso, o QuickSort escolhe um elemento do conjunto como pivô, sendo esse geralmente o primeiro elemento. Daí, reorganiza-se os demais elementos de forma que apenas os elementos menores que o pivô se situam à esquerda deste, assim formando dois subconjuntos em cada lado do pivô: um com elementos menores e outro com elementos maiores.
Na configuração resultante, o pivô passa a estar necessariamente na posição correta, e é então fixado. O processo descrito acima é repetido nos subconjuntos formados, definindo novo pivôs em cada. Isso continua recursivamente, formando conjuntos cada vez menores até o surgimento de conjuntos unitários,

conforme esquematizado na figura 1.

O QuickSort é considerado relativamente eficiente dentre os algoritmos de ordenação mais usados. Entretanto, há algumas desvantagens principais como o uso de memória necessário, que aumenta consideravelmente com o número de elementos do vetor de interesse devido ao caráter recursivo do algoritmo.
Além disso, o QuickSort pode tornar-se bem ineficiente quando apresentado com vetores em certos arranjos. Por exemplo, podemos citar o caso quando a lista já está maiormente ordenada em ordem crescente ou decrescente, na qual o algoritmo não poderá fazer rearranjos significativos.

2.2 MERGESORT

Também se enquadrando na categoria de ?divisão e conquista?, o MergeSort divide o vetor em duas partes aproximadamente iguais, até a formação de conjuntos unitários similarmente ao QuickSort, antes de intercalar-se de volta na ordem reversa, até retornar ao tamanho inicial do vetor de interesse.
O processo de intercalação entre duas listas é feito inserindo os elementos de cada num vetor auxiliar, que é subsequentemente sobrescrito nos dois conjuntos iniciais. Compara-se um elemento de cada conjunto, inserindo sempre o menor (caso não haja valores iguais), começando com os respectivos primeiros elementos e avançando conforme a inserção, de forma independente.
Realizando-se essa junção a partir dos conjuntos unitários formados, os subconjuntos a serem intercalados sempre estão organizados internamente, facilitando a organização final como um todo.

Similarmente ao QuickSort, esse algoritmo exige uso de memória além do próprio vetor de interesse por ser recursivo.

2.3 HEAPSORT

É um algoritmo baseado na árvore binária máxima. Nesse tipo de sequência, cada elemento pai tem no máximo dois elementos filhos associados de valores igual ao menor que o pai. Os elementos filhos, por sua vez, podem ter filhos próprios, assim formando uma estrutura similar a uma árvore, onde o primeiro elemento é considerado a raiz da estrutura como um todo.
Aqui, utiliza-se repetidamente um método chamado de Heapify, descrito como segue: um elemento parente X é comparado com seus filhos (ou filho único). Se não existe filho superior ao X, o método é finalizado. Caso contrário, trocam-se as posições do X e do filho com valor superior, e esse processo continua na nova posição de X. Note que caso haja dois filhos com valor distintos maior que X, o maior será sempre escolhido.
No processo do HeapSort, incialmente o vetor é reorganizado até que se constitua numa árvore binária máxima. Isso é feito aplicando Heapify em todos os elementos parentes (com pelo menos um filho), em

ordem decrescente. Ao final, isso é suficiente em tornar o vetor completo numa árvore máxima válida. Em seguida, trocamos a raiz com o último elemento da árvore, fixando a raiz original na posição nova e desconsiderando-o do resto da árvore. Daí Heapify é chamado na nova raiz, e a troca é realizada novamente até todos os elementos serem eliminados da árvore, momento na qual o vetor passa a ser necessariamente ordenado.

Como exemplo, considere o vetor na esquerda da figura 3, que não constitui numa árvore válida ainda. Aplicando o HeapSort, chama-se o Heapify nos seguintes elementos, conforme a sequência:
Elemento 8: Como esse já é maior que os filhos, nenhuma troca é realizada;
Elemento 1: Existe filho com valor superior. Assim, troca-se 1 com 9, o filho de maior valor;
Elemento 7: Conforme a lógica do passo anterior, 7 é trocado com 8. Os novos filhos de 7, 4 e 6, são inferiores;
Elemento 3: Nesse ponto, os filhos passam a ser 8 e 9. Pela mesma lógica, troca-se 3 com 9, tornando 5 e 1 os novos filhos de 3. Assim, troca-se 3 com 5.
Assim, obtemos a árvore da direita da figura 3. Pelo outro lado, a figura 4 apresenta o passo seguinte, que envolve a eliminação do 9 da árvore.

Aqui, 9 tem sua posição trocada com 6, o elemento final da árvore. Em seguida, 9 é retirado da árvore e fixada na posição correta do vetor, fazendo a árvore resultante inválida mais uma vez. Daí, chama-se o Heapify no novo nó 6, que resultará na troca entre 6 e 8, e entre 6 e 7, para restaurar a propriedade da árvore. Com isso, 6 é trocado com 4, a nova posição final da árvore, e assim por diante.
O HeapSort não necessita de memória adicional além da própria função e do vetor de interesse, pois todos os rearranjos dos elementos são realizados dentro do vetor.

Este trabalho tem como objetivo desenvolver um sistema computacional para avaliar o desempenho de diversos algoritmos de ordenação de dados. Considerando a importância de algoritmos eficientes na análise e manipulação de grandes volumes de dados, o sistema proposto visa fornecer uma análise comparativa de desempenho utilizando dados internos e externos. Através do sistema, será possível mensurar o tempo de execução de cada algoritmo em diferentes condições, permitindo identificar suas vantagens e limitações.
Geração e/ou Obtenção de Dados para Ordenação:
Os dados são elementos essenciais para a análise deste trabalho, uma vez que eles permitem verificar a eficácia dos algoritmos estudados. Assim, foram desenvolvidas duas versões para cada programa de ordenação: uma para obter os dados a partir de arquivos de texto e outra para gerar números aleatórios. Dessa forma, busca-se proporcionar maior embasamento e rigor à análise, aproximando-se de resultados mais precisos e consistentes.
1.1 Obtenção de Dados:
Para a obtenção dos dados, foram utilizadas informações fornecidas pelo professor, organizadas em uma

pasta principal que contém subpastas com conjuntos de dados de diferentes tamanhos (dez, cem, mil, dez mil, cinquenta mil, cem mil, quinhentos mil e cinco mil). Cada subpasta possui versões com dados duplicados e sem duplicados, permitindo uma análise mais abrangente. Dentro delas, possuíam mais outras cinco opções, aleatório, côncavo-decresce cresce, convexo-cresce decresce, crescente e decrescente, todos, continham dez arquivos de dados, cada uma tendo a característica descrita pela pasta , como pode ser visto na imagem abaixo.

Figura 1 ? Pasta com dados e caminho mostrando algumas das pastas

Fonte: De autoria própria.

No entanto, era necessário fazer o sistema ter a capacidade de leitura desses arquivos que até então ainda era impossível, portanto, foi usado a estrutura FILE, contida dentro da biblioteca de entradas e saídas padrão da linguagem de programação C (stdio.h). Por haver certas dificuldades para fazer com que o sistema percorre o caminho até onde era encontrado o arquivo, todo o processo de testes foi feito com os arquivos dentro da mesma pasta do programa.

A leitura do acervo de informações, é feita da seguinte maneira, cria-se uma variável de ponteiro, capaz de armazenar elementos de maneira dinâmica, ela faz a leitura dos arquivos, verificando também se é possível ou não a alocação de memória, caso tudo ocorra da maneira correta, uma função de loop, irá transferir todos os números guardados no arquivo para o vetor criado pelo ponteiro, até ele não ser capaz de encontrar mais dados, a figura abaixo mostra o trecho do código que faz todo esse processo.

Figura 2 ? Trecho do sistema de obtenção de dados.

Fonte: De autoria própria.

1.2 Geração dos dados

Em contrapartida, na geração de dados, foi utilizado a função rand(), da qual retornará um número-pseudo-aleatório, é dito como pseudo, porque ele não é completamente aleatório, já que é feito um cálculo especial usando em base um número padrão que é tomado como semente para gerar os seguintes, tendo como padrão o um. Há uma outra forma de se gerar esses dados, no entanto, seria utilizado o comando srand(), este teria uma semente diferente fazendo com que os números pudessem ser mais diferentes entre si, porém ele não foi utilizado no programa.

Conforme visto anteriormente, fora utilizado a função rand, para a geração de dados, porém é necessário um loop que possa ser capaz de criar esses números até o tamanho limite indicado pelo desenvolvedor. Consequentemente, foi usado um loop for, que fará a tarefa repetidas vezes até alcançar o tamanho limite indicado, a figura abaixo mostra o trecho do programa descrito acima:

Figura 3 ? Trecho do sistema de geração de dados

Fonte: De autoria própria

Processo de Ordenação de Dados:

Por outro lado, há o processo de ordenação, neste trabalho foram citadas e utilizadas três das principais, geralmente, as mais usadas, são elas, Quick Sort, Heap Sort, e Merge Sort, cada uma possuindo seu método específico de organizar os dados, mas todas as escolhidas, são consideradas as mais rápidas dentre as mais populares.

Cada algoritmo possui uma complexidade de tempo e de espaço, que permite prever, respectivamente, o tempo necessário para a execução do algoritmo e a quantidade de memória exigida para armazenar as estruturas necessárias durante o processamento.

O uso desses cálculos é extremamente importante para ver se é possível ou não o uso desse programa na máquina que será testada, caso contrário, diversos tipos de erros podem acabar ocorrendo, afetando diretamente o resultado.

2.1 Quick Sort:

O algoritmo Quick Sort, traduzido como ?ordenação rápida?, é conhecido por sua eficiência e velocidade na ordenação de dados. Sua eficiência se deve à estratégia de ?Divisão e Conquista?, que divide o vetor em partes menores, organiza cada parte individualmente e, em seguida, combina as segmentações ordenadas para obter o resultado.

Segundo Cormen et al. (2012), essa abordagem é especialmente eficiente em algoritmos de ordenação, pois permite reduzir o tempo de execução para uma complexidade de O(n log n), na média dos casos.

No entanto, para poder dividir, ele precisa encontrar um ponto para dividir, essa necessidade é resolvida após a escolha de um elemento na lista como um pivô, este número será o valor de referência usado para dividir a lista em duas partes, elementos maiores, e menores.

Figura 4 ? Trecho do algoritmo Quick Sort realizando o loop de partição do vetor

Fonte: De própria autoria

Assim como na figura acima, o algoritmo então particiona o vetor, fazendo com que os elementos menores que o pivô se vão para a esquerda e os maiores fiquem à sua direita, esse processo é feito recursivamente para os subvetores decorrentes. Este processo continua a ocorrer até que a lista possua no máximo um elemento, pois assim, ela já estaria ordenada.

A figura abaixo mostra o todo processo descrito anteriormente, dentro do sistema computacional.

Figura 5 ? Algoritmo de ordenação QuickSort

Fonte: De própria autoria

Já a complexidade de tempo média dele é representada pela fórmula:

O(n log n) (1)

Isto significa, que, ele realiza um número de comparações e movimentações que equivalem à fórmula (1),

onde n é o tamanho da lista a ser ordenada. Portanto, este é um dos motivos que faz este algoritmo ser tão eficiente, o seu tempo relativo já é suficientemente bom.

E por último, ele não possui cálculo de armazenamento, porque o Quick Sort é um algoritmo ?in-place?, ou seja, não requer memória adicional para armazenar as estruturas temporárias durante a ordenação. Tornando-o extremamente eficiente em termos de uso de memória.

2.2 Heap Sort:

O algoritmo Heap Sort funciona como uma árvore binária especial, na qual o valor de cada pai é maior ou igual ao valor de seus filhos (max-heap) ou menor ou igual (min-heap). No contexto do heapsort, é usado o max-folhas (max-heap), ou seja, ele irá tentar buscar o maior valor e colocá-lo no topo da árvore, e sempre seguindo dessa maneira até conseguir ordenar corretamente os elementos guardados.

Primeiramente, o algoritmo começa a iniciar o processo de transformação do vetor em uma árvore, sempre buscando o maior número e colocando-o ao topo da árvore, o funcionamento é parecido com o de uma pirâmide, esta figura abaixo representa visualmente como é o funcionamento.

Após o maior número atingir o topo da árvore, ele é movido para a posição final da lista e removido da árvore, então ocorre uma reorganização entre todas as folhas para levar o maior número para o topo novamente, para isso que ocorra até todos os elementos serem levados para o vetor ao ponto de ele estar completo, com todos os seus dados de volta.

Em suma, o algoritmo Heap, cria uma árvore binária e realiza uma constante busca para levar todos os números ao topo para que assim consigam ser organizados do maior para o menor. Em relação à complexidade de tempo médio, o cálculo do algoritmo Heap Sort é o mesmo que do Quick Sort (1), portanto possui as mesmas qualidades dele, sendo tão rápido quanto ele.

Assim como o tempo, sua complexidade de espaço também é a mesma do Quick, os dois são considerados in-place, ele usa o mesmo vetor para armazenar os elementos e para o processo de ordenação, sendo extremamente eficiente em situações em que se há pouca memória no computador. A figura abaixo mostra o algoritmo do Heap Sort por completo.

Figura 7 ? Algoritmo de ordenação Heap Sort

Fonte: De própria autoria

2.3 Merge Sort:

O Merge Sort se assemelha, muito com o algoritmo Quick Sort, seja por conta da sua estratégia, mas também por conta de sua versatilidade, eficiência e velocidade em organizar um vetor desorganizado.

Em comparação, os dois utilizam a estratégia, divisão e conquista, porém, uma grande diferença acaba os diferenciando e mudando radicalmente, isto é, a aplicação dessa estratégia é diferente para cada um deles .

Enquanto o algoritmo Quick Sort, acaba necessitando de um pivô, o Merge não se prende a isso, pois ele divide a lista em parte iguais e em seguida, combina essas partes ordenadas, completando a ordenação por completo. Em suma, um deles encontra o meio e divide e o outro escolhe um pivô, tornando o que separa no meio, o melhor, principalmente nos piores casos, pois, o pivô escolhido pode acabar não sendo o do meio, dificultando bastante e aumentando a complexidade de tempo para a seguinte fórmula:

$O(n^2)(2)$

Por outro lado, o Merge Sort continua constante mesmo em situações em que existem muitos elementos a serem ordenados, também não dependendo da sorte, facilitando ainda mais a organização, a figura 9 mostra uma representação de como é realizada a separação:

Figura 8 - Algoritmo de ordenação Merge Sort

Fonte: De autoria própria
A figura acima mostra como é feita a divisão recursiva do Merge Sort, no programa usado neste trabalho. Como citado, a complexidade de tempo do algoritmo é sempre constante, em todos os casos, por conta de como ele aplica a estratégia, então ele mantém a fórmula (1) sempre.
No entanto, a complexidade de espaço, acaba sendo levemente diferente, pois ele possui um auxiliar que acaba consumindo memória, fazendo-o não se tornar in-place. Na figura 8, podemos ver a variável ?t? sendo usada como vetor auxiliar. Assim a fórmula de espaço acaba sendo a seguinte, onde n é o número de elementos:
O(n)(3)
Listagem dos Valores Antes e Depois da Ordenação:
A análise será feita lendo o mesmo arquivo de texto, um de dez dados duplicados, para ser possível diferenciar a forma que cada um ordena, explicando seus passos e métodos.
3.1 Quick Sort:
Figura 9 ? Reposta dada pelo algoritmo de ordenação Quick Sort

Fonte: De autoria própria
Na figura acima, é possível ver como funciona o processo de ordenação do algoritmo Quick Sort lendo o arquivo, no entanto, o código acaba tendo certos problemas em mostrar o elemento, como é possível ver na terceira linha de passos imprimida, o último elemento mostra um endereço de memória, por conta de uma das recursividades acabar utilizando o endereço, fazendo a impressão se tornar confusa.
Contudo, analisando algumas partes compreensíveis da imagem, pode-se concluir que os passos da ordenação prosseguem dividindo, utilizando o pivô escolhido que acaba mudando após um certo período de impressões, pois ele vai ordenando os subvetores restantes, utilizando recursividade e economizando na alocação de memória.
3.2 Heap Sort:
Figura 10 - Resposta dada pelo algoritmo de ordenação Heap Sort

Fonte: De própria autoria
Assim como explicado, o maior valor é sempre levado para o fim do vetor, na figura acima, é mais facilmente visível como é feita essa ação. À medida que se progride, é visível a sucessão do processo, com o algoritmo mostrando a movimentação dos maiores elementos para a parte mais à extremidade esquerda, então, após percorrer por todas as folhas, move-as para o fim do vetor.
Por conta de ele ser uma árvore, as impressões acabam sendo um pouco maior, porém, é visível, em comparação com o Quick que por sua vez, acaba mostrando os endereços de memória.
3.3 Merge Sort:

Figura 11 - Resposta dada pelo algoritmo de ordenação Merge Sort

Fonte: De autoria própria
É imprimido de maneira bem similar, como o Quick Sort, porém de maneira visível mostrando o número invés do endereço, isso ocorre, pois, a impressão está ocorrendo em um momento diferente do Quick, neste caso, ela está após a transferência do vetor temporário, o que faz com que não seja imprimida o

endereço de memória.
Portanto, é mais perceptível como funciona o procedimento de divisão e conquista, vendo como o 958 é lentamente passado para o meio, até aquele momento ele é considerado o maior número da primeira metade fazendo-o fica naquela posição e então cada metade vai se organizando.
Apresentação dos Resultados Comparativos de Performance:
Esta seção apresenta uma análise de resultados entre os algoritmos utilizando quatro tipos de cenários diferentes entre eles, podendo ser duplicados ou não. Cada um dos programas foi avaliado quanto ao tempo médio de execução, visando descobrir o melhor, dependendo em cada ocasião. Os testes foram feitos com arquivos de texto com 50 mil dados e os gerados tiveram 100 mil amostras.
4.1 Medição:
Para obter os resultados, foi necessário o uso da função clock() da biblioteca da linguagem de programação C, da qual pode medir, os milissegundos, com mais precisão, o momento exato que é executado o algoritmo de ordenação, sem contar o tempo de criação de variáveis, alocação de memória e obtenção ou geração de dados.
Desta maneira, foram obtidos os dados com o tempo médio em (ms) apresentados na seguinte tabela:
Tabela 1 ? Comparação do Tempo Médio de Execução dos Algoritmos com Dados Duplicados
Fonte: De autoria própria
Tabela 2 - Comparação do Tempo Médio de Execução dos Algoritmos sem Dados Duplicados
Fonte: De autoria própria
Com estas tabelas, é possível observar que, em situações de dados espalhados de forma aleatória, o algoritmo Heap Sort é um dos piores no quesito tempo, perdendo por poucos milissegundos.
No entanto, em situações em que os elementos são espalhados de forma côncava, caso os dados sejam duplicados, o Quick acaba perdendo, mas em situações sem duplicidade, o Heap acaba tendo uma desvantagem de 515 milissegundos, uma diferença que quando comparada em situações de mais dados, faz com que ele não seja tão eficiente.
Toda essa diferença entre os dois, ocorre devido a diferença da estratégia entre eles, enquanto um necessita fazer uma árvore e deve percorrê-la, o outro apenas se divide e compara até conseguir se ordenar.
Apesar do Quick conseguir ser mais veloz que o Heap nesses cenários, existem muitos casos em que a sua estratégia não funciona tão bem, por exemplo, com grandes quantidades de números, esse cenário faz com que o Quick se torne extremamente instável nessas ocasiões. Pois, a forma em que ele exerce seu plano, depende muito de o pivô estar mais perto do centro ou não, podendo facilitar e muito em arquivos de dados pequenos.
Algo que não ocorre com o Merge Sort, que por sua vez, acaba brilhando nessas situações já que a aplicação da estratégia Divisão e Conquista é diferente, fazendo uma divisão por 2, não precisando ser tão dependente de achar ou não um pivô mais próximo do centro do vetor, já que quando dividido, sempre cairá ao meio.
No entanto, o algoritmo Merge Sort acaba tendo uma grande desvantagem em comparação aos outros dois algoritmos, o armazenamento de espaço, o que torna ele estável na maior parte das situações, tende a deixá-lo lento, pois ocorre a transferência de números do vetor temporário ao original.

Ref Biblio:
CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: Teoria e Prática. 3. ed. Rio de Janeiro: Elsevier, 2012.
SEDGEWICK, Robert; WAYNE, Kevin. Algorithms. 4th ed. Boston: Addison-Wesley, 2011.
MANZANO, José Augusto N. G.; OLIVEIRA, Jayr Figueiredo de. Estruturas de Dados e Algoritmos. 3. ed. São Paulo: Érica, 2011.
SIPSER, Michael. Introduction to the Theory of Computation. 3rd ed. Boston: Cengage Learning, 2012.
KNUTH, Donald E. The Art of Computer Programming: Sorting and Searching. 2nd ed. Massachusetts: Addison-Wesley, 1998.
DEITEL, H.; DEITEL, P. C: Como Programar. 7. ed. São Paulo: Pearson, 2010.
Resultados e Discussão

Como visto previamente neste documento, cada tipo de método de ordenação tem características próprias que o tornam altamente eficiente em algumas ocasiões e altamente ineficiente em outras. No entanto torna-se necessário uma análise mais profunda destes dados para um melhor entendimento dos mesmos. Para tal feito, foram coletadas mais de 800 amostras visando um melhor entendimento de cada método de ordenação.
Todas as amostras foram coletadas tendo como princípio a meta de 400 dados para cada tipo de método de ordenação, separados em dois subgrupos de 200 dados, respectivamente com e sem duplicidade de dados. Cada um destes subgrupos foi dividido em 5 subgrupos de 40 dados cada (aleatório, côncavo, convexo, crescente, decrescente) e um subgrupo com 100 dados(gerado pelo nosso grupo). Cada dado era composto por 50.000 números(como demonstrado abaixo em nossa imagem de autoria própria).

Ao serem analisados foi possível se tirar a seguinte tabela (autoria própria) como conclusão:

Os números com código de cores ao lado na parte inferior dos gráficos acima representam o tempo (em segundos) que cada algoritmo demorou para ser executado, já os números ao lado da palavra ?Frequência? representam a frequência de cada um dos números com código de cores ao lado.
Os gráficos acima permitem concluir diversos fatores. Por exemplo, os métodos de ordenação heap e merge têm uma variação menor do que o quick sort em termos de análise de dados. No entanto, o método merge possui uma demora muito maior para processar os dados que o quick e o heap.
Com tais dados foi possível montar uma curva normal extremamente detalhada para cada tipo de algoritmo de ordenação, com e sem duplicidade de números, e para os números também localmente gerados a partir de métodos de ordenação. Abaixos estão respectivamente as curvas normais (autoria própria) dos dados listados acima :

Os grupos de curvas normais apresentados acima são respectivamente pertencentes aos métodos de ordenação hippie, quick e merge. Com os códigos de cores preto para ordenação com duplicidade, vermelho para ordenação se duplicidade e azul para os dados coletados a partir de dados gerados localmente.
É possível notar que em cada grupo é apresentado vários símbolos com significados diversos. O símbolo ? denota a média de todos os dados coletados. O símbolo ? descreve o desvio padrão do objeto em estudo.
O número indicado no balão é o mesmo que o do símbolo X. A porcentagem que se segue no meio da curva de sino traz qual a probabilidade de haver números abaixo de X o que, por inferência, nos traz uma taxa de confiança de 95% para x ? X. Portanto pode-se afirmar que nenhum número recairá abaixo de X a não ser por acaso, e que X deve ser tomado como nosso objeto de estudo atual ignorando-se todos os tempos acima do mesmo.
Nota-se também que o método de ordenação quick está com menos dados na parte de com e sem duplicidade. Isso se deve ao fato do erro Stack overflow ter ocorrido durante a execução de ambos os dados decrescente e crescente, então se foi executado apenas uma vez cada e posto o tempo de demora para o código sair.
Com isso em mente é possível concluir que os métodos de ordenação quick e heap, são realmente mais rápidos em algumas ocasiões. No entanto, os métodos que apresentam mais estabilidade durante a execução são os métodos heap e merge. Há também a probabilidade de o método quick dar o erro Stack overflow devido a quantidade de memória ram que ocupa durante o processo de execução. Por fim, é possível concluir que o método de ordenação mais estável e o mais rápido é o heap.
Concluimos que os algoritmos de ordenação Quick Sort, Merge Sort e Heap Sort, mesmo que apresentem

complexidades de tempo semelhantes , demonstram comportamentos distintos em diferentes cenários. Algoritmos de ordenação estáveis, como o Merge Sort , são úteis em situações em que a preservação da ordem relativa é importante, como ordenação de registros em bancos de dados ou classificação por múltiplos critérios. Por outro lado, algoritmos de ordenação instáveis, como o Quicksort, podem alterar a ordem de elementos com chaves iguais durante o processo de classificação.
O Quick Sort e Heap Sort tiveram um desempenho melhor que o Merge sort sem duplicidade e com duplicidade, onde o Quick Sort se sobrepõe ao Heap Sort tendo menos frequencia, com uma diferença de 25 no tem 0,006 sem duplicidade e com duplicidade um diferença de 42.
Portanto a escolha do algoritmmo de ordenação ideal depende de diversas situações, como o tamho do conjunto de dados, os requisitos da memória ea estabilidade. No geral o Quick Sort será uma excelente opção para a maioria dos casos devido ao seu desempenho .No entanto o Heap Sort pode ser uma boa opção para grandes conjutos de dados. O Merge Sort pode ser uma boa opção quando o espaço de memória for limitado.

AlgoritmoAleatório (seg.)Côncavo (seg.)Convexo (seg.)Gerado (seg.)
Quick Sort0.005550.006450.00540,00527
Heap Sort0.0077750.0058750.00580,01695
Merge Sort0.011150.008150.0082750,011

AlgoritmoAleatório (seg.)Côncavo (seg.)Convexo (seg.)
Quick Sort0.0056750.006450.005875
Heap Sort0.00750.01160.0065
Merge Sort0.01150.00850.0085

==========

Arquivo 1: Trabalho_aps.docx (4117 termos)
Arquivo 2: https://br.usembassy.gov/pt/visas-pt/vistos-de-nao-imigrantes (2946 termos)
Termos comuns: 14
Similaridade: 0,19%
O texto abaixo é o conteúdo do documento Trabalho_aps.docx (4117 termos)
Os termos em vermelho foram encontrados no documento https://br.usembassy.gov/pt/visas-pt/vistos-de-nao-imigrantes (2946 termos)

==========

UNIVERSIDADE PAULISTA

G85JFG6 ? Eric Mitchell Barbosa Durham
N078146 ? Rafael Hiro Portilho
R0238H0 ? Matheus Reis Oliveira
N4031A9 ? Humberto Joji Fukui

Ordenação de dados amazônicos
Análise de algoritmos de ordenação de dados

São Paulo
2024

Neste trabalho tem o intuito de apresentar e comparar sobre os algoritmos Merge Sort, Quicksort e Heap Sort o seu desenvolvimento, eficiencia e complexidade. Avaliaremos os resultados adquiridos por cada algoritmo e discutir a aplicação desses agoritmos em alguns tipos de situações e uma analise do tempo que consiguram resolver cada situação. Os algoritmos de ordenações desenpenham papeis fundamentais na organização e otmização de dados, tornando-se fundamental para a eficiência para operações de busca e armazenamento em sistemas em sistemas de bancos de dados e outras aplicações computacionais.

Os algoritmos Merge sort, Heap sort e Quicksort conhecidos por sua eficiência e são amplamente utilizados em diversas linguagens de programação e bibliotecas. Cada um deles possui suas próprias características e vantagens, que serão exploradas em detalhes neste trabalho.

Quick sort se destaca por sua escolha programada de pivôs e rearranjo recursivo dos elementos, apresenta uma complexidade média de O(n log n), por outro lado o Merge sort conhecido por sua eficiencia na ordenação por divisão e conquista, destaca-se pela sua eficiência e estabilidade, dando uma ordenação eficaz dos elementos com complexidade O(n log n).O Heap sort, oferece um ordenação eficiente com complexidade O(n log n), destacando-se pela sua robustez.

Ao estudar esses algoritimos, temos como objetivo mostrar uma visão abrangente de suas características, desempenho e enetendimento a diferentes contextos, auxiliando na seleção do algoritmo mais adequado de acordo com a necessidade especifica de cada situação aplicada a eles.

2 REFERENCIAL TEÓRICO

2.1 QUICKSORT

Esse algoritmo usa a técnica da "divisão e conquista", que consiste na divisão recursiva de um problema em vários menores. No caso, o QuickSort escolhe um elemento do conjunto como pivô, sendo esse geralmente o primeiro elemento. Daí, reorganiza-se os demais elementos de forma que apenas os elementos menores que o pivô se situam à esquerda deste, assim formando dois subconjuntos em cada lado do pivô: um com elementos menores e outro com elementos maiores.

Na configuração resultante, o pivô passa a estar necessariamente na posição correta, e é então fixado. O processo descrito acima é repetido nos subconjuntos formados, definindo novo pivôs em cada. Isso continua recursivamente, formando conjuntos cada vez menores até o surgimento de conjuntos unitários, conforme esquematizado na figura 1.

O QuickSort é considerado relativamente eficiente dentre os algoritmos de ordenação mais usados. Entretanto, há algumas desvantagens principais como o uso de memória necessário, que aumenta consideravelmente com o número de elementos do vetor de interesse devido ao caráter recursivo do algoritmo.
Além disso, o QuickSort pode tornar-se bem ineficiente quando apresentado com vetores em certos arranjos. Por exemplo, podemos citar o caso quando a lista já está maiormente ordenada em ordem crescente ou decrescente, na qual o algoritmo não poderá fazer rearranjos significativos.

2.2 MERGESORT

Também se enquadrando na categoria de ?divisão e conquista?, o MergeSort divide o vetor em duas partes aproximadamente iguais, até a formação de conjuntos unitários similarmente ao QuickSort, antes de intercalar-se de volta na ordem reversa, até retornar ao tamanho inicial do vetor de interesse.
O processo de intercalação entre duas listas é feito inserindo os elementos de cada num vetor auxiliar, que é subsequentemente sobrescrito nos dois conjuntos iniciais. Compara-se um elemento de cada conjunto, inserindo sempre o menor (caso não haja valores iguais), começando com os respectivos primeiros elementos e avançando conforme a inserção, de forma independente.
Realizando-se essa junção a partir dos conjuntos unitários formados, os subconjuntos a serem intercalados sempre estão organizados internamente, facilitando a organização final como um todo.

Similarmente ao QuickSort, esse algoritmo exige uso de memória além do próprio vetor de interesse por ser recursivo.

2.3 HEAPSORT

É um algoritmo baseado na árvore binária máxima. Nesse tipo de sequência, cada elemento pai tem no máximo dois elementos filhos associados de valores igual ao menor que o pai. Os elementos filhos, por sua vez, podem ter filhos próprios, assim formando uma estrutura similar a uma árvore, onde o primeiro elemento é considerado a raiz da estrutura como um todo.
Aqui, utiliza-se repetidamente um método chamado de Heapify, descrito como segue: um elemento parente X é comparado com seus filhos (ou filho único). Se não existe filho superior ao X, o método é finalizado. Caso contrário, trocam-se as posições do X e do filho com valor superior, e esse processo continua na nova posição de X. Note que caso haja dois filhos com valor distintos maior que X, o maior será sempre escolhido.
No processo do HeapSort, incialmente o vetor é reorganizado até que se constitua numa árvore binária máxima. Isso é feito aplicando Heapify em todos os elementos parentes (com pelo menos um filho), em ordem decrescente. Ao final, isso é suficiente em tornar o vetor completo numa árvore máxima válida.

Em seguida, trocamos a raiz com o último elemento da árvore, fixando a raiz original na posição nova e desconsiderando-o do resto da árvore. Daí Heapify é chamado na nova raiz, e a troca é realizada novamente até todos os elementos serem eliminados da árvore, momento na qual o vetor passa a ser necessariamente ordenado.

Como exemplo, considere o vetor na esquerda da figura 3, que não constitui numa árvore válida ainda. Aplicando o HeapSort, chama-se o Heapify nos seguintes elementos, conforme a sequência:
Elemento 8: Como esse já é maior que os filhos, nenhuma troca é realizada;
Elemento 1: Existe filho com valor superior. Assim, troca-se 1 com 9, o filho de maior valor;
Elemento 7: Conforme a lógica do passo anterior, 7 é trocado com 8. Os novos filhos de 7, 4 e 6, são inferiores;
Elemento 3: Nesse ponto, os filhos passam a ser 8 e 9. Pela mesma lógica, troca-se 3 com 9, tornando 5 e 1 os novos filhos de 3. Assim, troca-se 3 com 5.
Assim, obtemos a árvore da direita da figura 3. Pelo outro lado, a figura 4 apresenta o passo seguinte, que envolve a eliminação do 9 da árvore.

Aqui, 9 tem sua posição trocada com 6, o elemento final da árvore. Em seguida, 9 é retirado da árvore e fixada na posição correta do vetor, fazendo a árvore resultante inválida mais uma vez. Daí, chama-se o Heapify no novo nó 6, que resultará na troca entre 6 e 8, e entre 6 e 7, para restaurar a propriedade da árvore. Com isso, 6 é trocado com 4, a nova posição final da árvore, e assim por diante.
O HeapSort não necessita de memória adicional além da própria função e do vetor de interesse, pois todos os rearranjos dos elementos são realizados dentro do vetor.

Este trabalho tem como objetivo desenvolver um sistema computacional para avaliar o desempenho de diversos algoritmos de ordenação de dados. Considerando a importância de algoritmos eficientes na análise e manipulação de grandes volumes de dados, o sistema proposto visa fornecer uma análise comparativa de desempenho utilizando dados internos e externos. Através do sistema, será possível mensurar o tempo de execução de cada algoritmo em diferentes condições, permitindo identificar suas vantagens e limitações.
Geração e/ou Obtenção de Dados para Ordenação:
Os dados são elementos essenciais para a análise deste trabalho, uma vez que eles permitem verificar a eficácia dos algoritmos estudados. Assim, foram desenvolvidas duas versões para cada programa de ordenação: uma para obter os dados a partir de arquivos de texto e outra para gerar números aleatórios. Dessa forma, busca-se proporcionar maior embasamento e rigor à análise, aproximando-se de resultados mais precisos e consistentes.
1.1 Obtenção de Dados:
Para a obtenção dos dados, foram utilizadas informações fornecidas pelo professor, organizadas em uma pasta principal que contém subpastas com conjuntos de dados de diferentes tamanhos (dez, cem, mil, dez

mil, cinquenta mil, cem mil, quinhentos mil e cinco mil). Cada subpasta possui versões com dados duplicados e sem duplicados, permitindo uma análise mais abrangente. Dentro delas, possuíam mais outras cinco opções, aleatório, côncavo-decresce cresce, convexo-cresce decresce, crescente e decrescente, todos, continham dez arquivos de dados, cada uma tendo a característica descrita pela pasta , como pode ser visto na imagem abaixo.

Figura 1 ? Pasta com dados e caminho mostrando algumas das pastas

Fonte: De autoria própria.

No entanto, era necessário fazer o sistema ter a capacidade de leitura desses arquivos que até então ainda era impossível, portanto, foi usado a estrutura FILE, contida dentro da biblioteca de entradas e saídas padrão da linguagem de programação C (stdio.h). Por haver certas dificuldades para fazer com que o sistema percorre o caminho até onde era encontrado o arquivo, todo o processo de testes foi feito com os arquivos dentro da mesma pasta do programa.

A leitura do acervo de informações, é feita da seguinte maneira, cria-se uma variável de ponteiro, capaz de armazenar elementos de maneira dinâmica, ela faz a leitura dos arquivos, verificando também se é possível ou não a alocação de memória, caso tudo ocorra da maneira correta, uma função de loop, irá transferir todos os números guardados no arquivo para o vetor criado pelo ponteiro, até ele não ser capaz de encontrar mais dados, a figura abaixo mostra o trecho do código que faz todo esse processo.

Figura 2 ? Trecho do sistema de obtenção de dados.

Fonte: De autoria própria.

1.2 Geração dos dados

Em contrapartida, na geração de dados, foi utilizado a função rand(), da qual retornará um número-pseudo-aleatório, é dito como pseudo, porque ele não é completamente aleatório, já que é feito um cálculo especial usando em base um número padrão que é tomado como semente para gerar os seguintes, tendo como padrão o um. Há uma outra forma de se gerar esses dados, no entanto, seria utilizado o comando srand(), este teria uma semente diferente fazendo com que os números pudessem ser mais diferentes entre si, porém ele não foi utilizado no programa.

Conforme visto anteriormente, fora utilizado a função rand, para a geração de dados, porém é necessário um loop que possa ser capaz de criar esses números até o tamanho limite indicado pelo desenvolvedor. Consequentemente, foi usado um loop for, que fará a tarefa repetidas vezes até alcançar o tamanho limite indicado, a figura abaixo mostra o trecho do programa descrito acima:

Figura 3 ? Trecho do sistema de geração de dados

Fonte: De autoria própria

Processo de Ordenação de Dados:

Por outro lado, há o processo de ordenação, neste trabalho foram citadas e utilizadas três das principais, geralmente, as mais usadas, são elas, Quick Sort, Heap Sort, e Merge Sort, cada uma possuindo seu método específico de organizar os dados, mas todas as escolhidas, são consideradas as mais rápidas dentre as mais populares.

Cada algoritmo possui uma complexidade de tempo e de espaço, que permite prever, respectivamente, o tempo necessário para a execução do algoritmo e a quantidade de memória exigida para armazenar as estruturas necessárias durante o processamento.

O uso desses cálculos é extremamente importante para ver se é possível ou não o uso desse programa na máquina que será testada, caso contrário, diversos tipos de erros podem acabar ocorrendo, afetando diretamente o resultado.

2.1 Quick Sort:

O algoritmo Quick Sort, traduzido como ?ordenação rápida?, é conhecido por sua eficiência e velocidade na ordenação de dados. Sua eficiência se deve à estratégia de ?Divisão e Conquista?, que divide o vetor em partes menores, organiza cada parte individualmente e, em seguida, combina as segmentações ordenadas para obter o resultado.
Segundo Cormen et al. (2012), essa abordagem é especialmente eficiente em algoritmos de ordenação, pois permite reduzir o tempo de execução para uma complexidade de O(n log n), na média dos casos.
No entanto, para poder dividir, ele precisa encontrar um ponto para dividir, essa necessidade é resolvida após a escolha de um elemento na lista como um pivô, este número será o valor de referência usado para dividir a lista em duas partes, elementos maiores, e menores.

Figura 4 ? Trecho do algoritmo Quick Sort realizando o loop de partição do vetor
Fonte: De própria autoria
Assim como na figura acima, o algoritmo então particiona o vetor, fazendo com que os elementos menores que o pivô se vão para a esquerda e os maiores fiquem à sua direita, esse processo é feito recursivamente para os subvetores decorrentes. Este processo continua a ocorrer até que a lista possua no máximo um elemento, pois assim, ela já estaria ordenada.
A figura abaixo mostra o todo processo descrito anteriormente, dentro do sistema computacional.

Figura 5 ? Algoritmo de ordenação QuickSort

Fonte: De própria autoria
Já a complexidade de tempo média dele é representada pela fórmula:
O(n log n) (1)
Isto significa, que, ele realiza um número de comparações e movimentações que equivalem à fórmula (1), onde n é o tamanho da lista a ser ordenada. Portanto, este é um dos motivos que faz este algoritmo ser

tão eficiente, o seu tempo relativo já é suficientemente bom.
E por último, ele não possui cálculo de armazenamento, porque o Quick Sort é um algoritmo ?in-place?, ou seja, não requer memória adicional para armazenar as estruturas temporárias durante a ordenação. Tornando-o extremamente eficiente em termos de uso de memória.
2.2 Heap Sort:
O algoritmo Heap Sort funciona como uma árvore binária especial, na qual o valor de cada pai é maior ou igual ao valor de seus filhos (max-heap) ou menor ou igual (min-heap). No contexto do heapsort, é usado o max-folhas (max-heap), ou seja, ele irá tentar buscar o maior valor e colocá-lo no topo da árvore, e sempre seguindo dessa maneira até conseguir ordenar corretamente os elementos guardados.
Primeiramente, o algoritmo começa a iniciar o processo de transformação do vetor em uma árvore, sempre buscando o maior número e colocando-o ao topo da árvore, o funcionamento é parecido com o de uma pirâmide, esta figura abaixo representa visualmente como é o funcionamento.
Após o maior número atingir o topo da árvore, ele é movido para a posição final da lista e removido da árvore, então ocorre uma reorganização entre todas as folhas para levar o maior número para o topo novamente, para isso que ocorra até todos os elementos serem levados para o vetor ao ponto de ele estar completo, com todos os seus dados de volta.
Em suma, o algoritmo Heap, cria uma árvore binária e realiza uma constante busca para levar todos os números ao topo para que assim consigam ser organizados do maior para o menor. Em relação à complexidade de tempo médio, o cálculo do algoritmo Heap Sort é o mesmo que do Quick Sort (1), portanto possui as mesmas qualidades dele, sendo tão rápido quanto ele.
Assim como o tempo, sua complexidade de espaço também é a mesma do Quick, os dois são considerados in-place, ele usa o mesmo vetor para armazenar os elementos e para o processo de ordenação, sendo extremamente eficiente em situações em que se há pouca memória no computador. A figura abaixo mostra o algoritmo do Heap Sort por completo.
Figura 7 ? Algoritmo de ordenação Heap Sort

Fonte: De própria autoria
2.3 Merge Sort:
O Merge Sort se assemelha, muito com o algoritmo Quick Sort, seja por conta da sua estratégia, mas também por conta de sua versatilidade, eficiência e velocidade em organizar um vetor desorganizado.
Em comparação, os dois utilizam a estratégia, divisão e conquista, porém, uma grande diferença acaba os diferenciando e mudando radicalmente, isto é, a aplicação dessa estratégia é diferente para cada um deles .
Enquanto o algoritmo Quick Sort, acaba necessitando de um pivô, o Merge não se prende a isso, pois ele divide a lista em parte iguais e em seguida, combina essas partes ordenadas, completando a ordenação por completo. Em suma, um deles encontra o meio e divide e o outro escolhe um pivô, tornando o que separa no meio, o melhor, principalmente nos piores casos, pois, o pivô escolhido pode acabar não sendo o do meio, dificultando bastante e aumentando a complexidade de tempo para a seguinte fórmula:
$O(n^2)(2)$
Por outro lado, o Merge Sort continua constante mesmo em situações em que existem muitos elementos a serem ordenados, também não dependendo da sorte, facilitando ainda mais a organização, a figura 9 mostra uma representação de como é realizada a separação:
Figura 8 - Algoritmo de ordenação Merge Sort

Fonte: De autoria própria
A figura acima mostra como é feita a divisão recursiva do Merge Sort, no programa usado neste trabalho. Como citado, a complexidade de tempo do algoritmo é sempre constante, em todos os casos, por conta de como ele aplica a estratégia, então ele mantém a fórmula (1) sempre.
No entanto, a complexidade de espaço, acaba sendo levemente diferente, pois ele possui um auxiliar que acaba consumindo memória, fazendo-o não se tornar in-place. Na figura 8, podemos ver a variável ?t? sendo usada como vetor auxiliar. Assim a fórmula de espaço acaba sendo a seguinte, onde n é o número de elementos:
O(n)(3)
Listagem dos Valores Antes e Depois da Ordenação:
A análise será feita lendo o mesmo arquivo de texto, um de dez dados duplicados, para ser possível diferenciar a forma que cada um ordena, explicando seus passos e métodos.
3.1 Quick Sort:
Figura 9 ? Reposta dada pelo algoritmo de ordenação Quick Sort

Fonte: De autoria própria
Na figura acima, é possível ver como funciona o processo de ordenação do algoritmo Quick Sort lendo o arquivo, no entanto, o código acaba tendo certos problemas em mostrar o elemento, como é possível ver na terceira linha de passos imprimida, o último elemento mostra um endereço de memória, por conta de uma das recursividades acabar utilizando o endereço, fazendo a impressão se tornar confusa.
Contudo, analisando algumas partes compreensíveis da imagem, pode-se concluir que os passos da ordenação prosseguem dividindo, utilizando o pivô escolhido que acaba mudando após um certo período de impressões, pois ele vai ordenando os subvetores restantes, utilizando recursividade e economizando na alocação de memória.
3.2 Heap Sort:
Figura 10 - Resposta dada pelo algoritmo de ordenação Heap Sort

Fonte: De própria autoria
Assim como explicado, o maior valor é sempre levado para o fim do vetor, na figura acima, é mais facilmente visível como é feita essa ação. À medida que se progride, é visível a sucessão do processo, com o algoritmo mostrando a movimentação dos maiores elementos para a parte mais à extremidade esquerda, então, após percorrer por todas as folhas, move-as para o fim do vetor.
Por conta de ele ser uma árvore, as impressões acabam sendo um pouco maior, porém, é visível, em comparação com o Quick que por sua vez, acaba mostrando os endereços de memória.
3.3 Merge Sort:

Figura 11 - Resposta dada pelo algoritmo de ordenação Merge Sort

Fonte: De autoria própria
É imprimido de maneira bem similar, como o Quick Sort, porém de maneira visível mostrando o número invés do endereço, isso ocorre, pois, a impressão está ocorrendo em um momento diferente do Quick, neste caso, ela está após a transferência do vetor temporário, o que faz com que não seja imprimida o endereço de memória.

Portanto, é mais perceptível como funciona o procedimento de divisão e conquista, vendo como o 958 é lentamente passado para o meio, até aquele momento ele é considerado o maior número da primeira metade fazendo-o fica naquela posição e então cada metade vai se organizando.
Apresentação dos Resultados Comparativos de Performance:
Esta seção apresenta uma análise de resultados entre os algoritmos utilizando quatro tipos de cenários diferentes entre eles, podendo ser duplicados ou não. Cada um dos programas foi avaliado quanto ao tempo médio de execução, visando descobrir o melhor, dependendo em cada ocasião. Os testes foram feitos com arquivos de texto com 50 mil dados e os gerados tiveram 100 mil amostras.
4.1 Medição:
Para obter os resultados, foi necessário o uso da função clock() da biblioteca da linguagem de programação C, da qual pode medir, os milissegundos, com mais precisão, o momento exato que é executado o algoritmo de ordenação, sem contar o tempo de criação de variáveis, alocação de memória e obtenção ou geração de dados.
Desta maneira, foram obtidos os dados com o tempo médio em (ms) apresentados na seguinte tabela:
Tabela 1 ? Comparação do Tempo Médio de Execução dos Algoritmos com Dados Duplicados
Fonte: De autoria própria
Tabela 2 - Comparação do Tempo Médio de Execução dos Algoritmos sem Dados Duplicados
Fonte: De autoria própria
Com estas tabelas, é possível observar que, em situações de dados espalhados de forma aleatória, o algoritmo Heap Sort é um dos piores no quesito tempo, perdendo por poucos milissegundos.
No entanto, em situações em que os elementos são espalhados de forma côncava, caso os dados sejam duplicados, o Quick acaba perdendo, mas em situações sem duplicidade, o Heap acaba tendo uma desvantagem de 515 milissegundos, uma diferença que quando comparada em situações de mais dados, faz com que ele não seja tão eficiente.
Toda essa diferença entre os dois, ocorre devido a diferença da estratégia entre eles, enquanto um necessita fazer uma árvore e deve percorrê-la, o outro apenas se divide e compara até conseguir se ordenar.
Apesar do Quick conseguir ser mais veloz que o Heap nesses cenários, existem muitos casos em que a sua estratégia não funciona tão bem, por exemplo, com grandes quantidades de números, esse cenário faz com que o Quick se torne extremamente instável nessas ocasiões. Pois, a forma em que ele exerce seu plano, depende muito de o pivô estar mais perto do centro ou não, podendo facilitar e muito em arquivos de dados pequenos.
Algo que não ocorre com o Merge Sort, que por sua vez, acaba brilhando nessas situações já que a aplicação da estratégia Divisão e Conquista é diferente, fazendo uma divisão por 2, não precisando ser tão dependente de achar ou não um pivô mais próximo do centro do vetor, já que quando dividido, sempre cairá ao meio.
No entanto, o algoritmo Merge Sort acaba tendo uma grande desvantagem em comparação aos outros dois algoritmos, o armazenamento de espaço, o que torna ele estável na maior parte das situações, tende a deixá-lo lento, pois ocorre a transferência de números do vetor temporário ao original.

Ref Biblio:
CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: Teoria e Prática. 3. ed. Rio de Janeiro: Elsevier, 2012.
SEDGEWICK, Robert; WAYNE, Kevin. Algorithms. 4th ed. Boston: Addison-Wesley, 2011.
MANZANO, José Augusto N. G.; OLIVEIRA, Jayr Figueiredo de. Estruturas de Dados e Algoritmos. 3. ed. São Paulo: Érica, 2011.
SIPSER, Michael. Introduction to the Theory of Computation. 3rd ed. Boston: Cengage Learning, 2012.
KNUTH, Donald E. The Art of Computer Programming: Sorting and Searching. 2nd ed. Massachusetts: Addison-Wesley, 1998.
DEITEL, H.; DEITEL, P. C: Como Programar. 7. ed. São Paulo: Pearson, 2010.
Resultados e Discussão

Como visto previamente neste documento, cada tipo de método de ordenação tem características próprias que o tornam altamente eficiente em algumas ocasiões e altamente ineficiente em outras. No entanto torna-se necessário uma análise mais profunda destes dados para um melhor entendimento dos mesmos. Para tal feito, foram coletadas mais de 800 amostras visando um melhor entendimento de cada método de ordenação.
Todas as amostras foram coletadas tendo como princípio a meta de 400 dados para cada tipo de método de ordenação, separados em dois subgrupos de 200 dados, respectivamente com e sem duplicidade de dados. Cada um destes subgrupos foi dividido em 5 subgrupos de 40 dados cada (aleatório, côncavo, convexo, crescente, decrescente) e um subgrupo com 100 dados(gerado pelo nosso grupo). Cada dado era composto por 50.000 números(como demonstrado abaixo em nossa imagem de autoria própria).

Ao serem analisados foi possível se tirar a seguinte tabela (autoria própria) como conclusão:

Os números com código de cores ao lado na parte inferior dos gráficos acima representam o tempo (em segundos) que cada algoritmo demorou para ser executado, já os números ao lado da palavra ?Frequência? representam a frequência de cada um dos números com código de cores ao lado.
Os gráficos acima permitem concluir diversos fatores. Por exemplo, os métodos de ordenação heap e merge têm uma variação menor do que o quick sort em termos de análise de dados. No entanto, o método merge possui uma demora muito maior para processar os dados que o quick e o heap.
Com tais dados foi possível montar uma curva normal extremamente detalhada para cada tipo de algoritmo de ordenação, com e sem duplicidade de números, e para os números também localmente gerados a partir de métodos de ordenação. Abaixos estão respectivamente as curvas normais (autoria própria) dos dados listados acima :

Os grupos de curvas normais apresentados acima são respectivamente pertencentes aos métodos de ordenação hippie, quick e merge. Com os códigos de cores preto para ordenação com duplicidade, vermelho para ordenação se duplicidade e azul para os dados coletados a partir de dados gerados localmente.
É possível notar que em cada grupo é apresentado vários símbolos com significados diversos. O símbolo ? denota a média de todos os dados coletados. O símbolo ? descreve o desvio padrão do objeto em estudo.
O número indicado no balão é o mesmo que o do símbolo X. A porcentagem que se segue no meio da curva de sino traz qual a probabilidade de haver números abaixo de X o que, por inferência, nos traz uma taxa de confiança de 95% para x ? X. Portanto pode-se afirmar que nenhum número recairá abaixo de X a não ser por acaso, e que X deve ser tomado como nosso objeto de estudo atual ignorando-se todos os tempos acima do mesmo.
Nota-se também que o método de ordenação quick está com menos dados na parte de com e sem duplicidade. Isso se deve ao fato do erro Stack overflow ter ocorrido durante a execução de ambos os dados decrescente e crescente, então se foi executado apenas uma vez cada e posto o tempo de demora para o código sair.
Com isso em mente é possível concluir que os métodos de ordenação quick e heap, são realmente mais rápidos em algumas ocasiões. No entanto, os métodos que apresentam mais estabilidade durante a execução são os métodos heap e merge. Há também a probabilidade de o método quick dar o erro Stack overflow devido a quantidade de memória ram que ocupa durante o processo de execução. Por fim, é possível concluir que o método de ordenação mais estável e o mais rápido é o heap.
Concluimos que os algoritmos de ordenação Quick Sort, Merge Sort e Heap Sort, mesmo que apresentem complexidades de tempo semelhantes , demonstram comportamentos distintos em diferentes cenários.

Algoritmos de ordenação estáveis, como o Merge Sort , são úteis em situações em que a preservação da ordem relativa é importante, como ordenação de registros em bancos de dados ou classificação por múltiplos critérios. Por outro lado, algoritmos de ordenação instáveis, como o Quicksort, podem alterar a ordem de elementos com chaves iguais durante o processo de classificação.
O Quick Sort e Heap Sort tiveram um desempenho melhor que o Merge sort sem duplicidade e com duplicidade, onde o Quick Sort se sobrepõe ao Heap Sort tendo menos frequencia, com uma diferença de 25 no tem 0,006 sem duplicidade e com duplicidade um diferença de 42.
Portanto a escolha do algoritmmo de ordenação ideal depende de diversas situações, como o tamho do conjunto de dados, os requisitos da memória ea estabilidade. No geral o Quick Sort será uma excelente opção para a maioria dos casos devido ao seu desempenho .No entanto o Heap Sort pode ser uma boa opção para grandes conjutos de dados. O Merge Sort pode ser uma boa opção quando o espaço de memória for limitado.

AlgoritmoAleatório (seg.)Côncavo (seg.)Convexo (seg.)Gerado (seg.)
Quick Sort0.005550.006450.00540,00527
Heap Sort0.0077750.0058750.00580,01695
Merge Sort0.011150.008150.0082750,011

AlgoritmoAleatório (seg.)Côncavo (seg.)Convexo (seg.)
Quick Sort0.0056750.006450.005875
Heap Sort0.00750.01160.0065
Merge Sort0.01150.00850.0085

==========

Arquivo 1: Trabalho_aps.docx (4117 termos)
Arquivo 2: https://lista.mercadolivre.com.br/espelho-de-seguranca-convexo (1019 termos)
Termos comuns: 2
Similaridade: 0,03%
O texto abaixo é o conteúdo do documento Trabalho_aps.docx (4117 termos)
Os termos em vermelho foram encontrados no documento https://lista.mercadolivre.com.br/espelho-de-seguranca-convexo (1019 termos)

==========

UNIVERSIDADE PAULISTA

G85JFG6 ? Eric Mitchell Barbosa Durham
N078146 ? Rafael Hiro Portilho
R0238H0 ? Matheus Reis Oliveira
N4031A9 ? Humberto Joji Fukui

Ordenação de dados amazônicos
Análise de algoritmos de ordenação de dados

São Paulo
2024

Neste trabalho tem o intuito de apresentar e comparar sobre os algoritmos Merge Sort, Quicksort e Heap Sort o seu desenvolvimento, eficiencia e complexidade. Avaliaremos os resultados adquiridos por cada algoritmo e discutir a aplicação desses agoritmos em alguns tipos de situações e uma analise do tempo que consiguram resolver cada situação. Os algoritmos de ordenações desenpenham papeis fundamentais na organização e otmização de dados, tornando-se fundamental para a eficiência para operações de busca e armazenamento em sistemas em sistemas de bancos de dados e outras aplicações computacionais.

Os algoritmos Merge sort, Heap sort e Quicksort conhecidos por sua eficiência e são amplamente utilizados em diversas linguagens de programação e bibliotecas. Cada um deles possui suas próprias características e vantagens, que serão exploradas em detalhes neste trabalho.

Quick sort se destaca por sua escolha programada de pivôs e rearranjo recursivo dos elementos, apresenta uma complexidade média de O(n log n), por outro lado o Merge sort conhecido por sua eficiencia na ordenação por divisão e conquista, destaca-se pela sua eficiência e estabilidade, dando uma ordenação eficaz dos elementos com complexidade O(n log n).O Heap sort, oferece um ordenação eficiente com complexidade O(n log n), destacando-se pela sua robustez.

Ao estudar esses algoritimos, temos como objetivo mostrar uma visão abrangente de suas características, desempenho e enetendimento a diferentes contextos, auxiliando na seleção do algoritmo mais adequado de acordo com a necessidade especifica de cada situação aplicada a eles.

2 REFERENCIAL TEÓRICO

2.1 QUICKSORT

Esse algoritmo usa a técnica da "divisão e conquista", que consiste na divisão recursiva de um problema em vários menores. No caso, o QuickSort escolhe um elemento do conjunto como pivô, sendo esse geralmente o primeiro elemento. Daí, reorganiza-se os demais elementos de forma que apenas os elementos menores que o pivô se situam à esquerda deste, assim formando dois subconjuntos em cada lado do pivô: um com elementos menores e outro com elementos maiores.

Na configuração resultante, o pivô passa a estar necessariamente na posição correta, e é então fixado. O processo descrito acima é repetido nos subconjuntos formados, definindo novo pivôs em cada. Isso continua recursivamente, formando conjuntos cada vez menores até o surgimento de conjuntos unitários, conforme esquematizado na figura 1.

O QuickSort é considerado relativamente eficiente dentre os algoritmos de ordenação mais usados. Entretanto, há algumas desvantagens principais como o uso de memória necessário, que aumenta consideravelmente com o número de elementos do vetor de interesse devido ao caráter recursivo do algoritmo.
Além disso, o QuickSort pode tornar-se bem ineficiente quando apresentado com vetores em certos arranjos. Por exemplo, podemos citar o caso quando a lista já está maiormente ordenada em ordem crescente ou decrescente, na qual o algoritmo não poderá fazer rearranjos significativos.

2.2 MERGESORT

Também se enquadrando na categoria de ?divisão e conquista?, o MergeSort divide o vetor em duas partes aproximadamente iguais, até a formação de conjuntos unitários similarmente ao QuickSort, antes de intercalar-se de volta na ordem reversa, até retornar ao tamanho inicial do vetor de interesse.
O processo de intercalação entre duas listas é feito inserindo os elementos de cada num vetor auxiliar, que é subsequentemente sobrescrito nos dois conjuntos iniciais. Compara-se um elemento de cada conjunto, inserindo sempre o menor (caso não haja valores iguais), começando com os respectivos primeiros elementos e avançando conforme a inserção, de forma independente.
Realizando-se essa junção a partir dos conjuntos unitários formados, os subconjuntos a serem intercalados sempre estão organizados internamente, facilitando a organização final como um todo.

Similarmente ao QuickSort, esse algoritmo exige uso de memória além do próprio vetor de interesse por ser recursivo.

2.3 HEAPSORT

É um algoritmo baseado na árvore binária máxima. Nesse tipo de sequência, cada elemento pai tem no máximo dois elementos filhos associados de valores igual ao menor que o pai. Os elementos filhos, por sua vez, podem ter filhos próprios, assim formando uma estrutura similar a uma árvore, onde o primeiro elemento é considerado a raiz da estrutura como um todo.
Aqui, utiliza-se repetidamente um método chamado de Heapify, descrito como segue: um elemento parente X é comparado com seus filhos (ou filho único). Se não existe filho superior ao X, o método é finalizado. Caso contrário, trocam-se as posições do X e do filho com valor superior, e esse processo continua na nova posição de X. Note que caso haja dois filhos com valor distintos maior que X, o maior será sempre escolhido.
No processo do HeapSort, incialmente o vetor é reorganizado até que se constitua numa árvore binária máxima. Isso é feito aplicando Heapify em todos os elementos parentes (com pelo menos um filho), em ordem decrescente. Ao final, isso é suficiente em tornar o vetor completo numa árvore máxima válida.

Em seguida, trocamos a raiz com o último elemento da árvore, fixando a raiz original na posição nova e desconsiderando-o do resto da árvore. Daí Heapify é chamado na nova raiz, e a troca é realizada novamente até todos os elementos serem eliminados da árvore, momento na qual o vetor passa a ser necessariamente ordenado.

Como exemplo, considere o vetor na esquerda da figura 3, que não constitui numa árvore válida ainda. Aplicando o HeapSort, chama-se o Heapify nos seguintes elementos, conforme a sequência:
Elemento 8: Como esse já é maior que os filhos, nenhuma troca é realizada;
Elemento 1: Existe filho com valor superior. Assim, troca-se 1 com 9, o filho de maior valor;
Elemento 7: Conforme a lógica do passo anterior, 7 é trocado com 8. Os novos filhos de 7, 4 e 6, são inferiores;
Elemento 3: Nesse ponto, os filhos passam a ser 8 e 9. Pela mesma lógica, troca-se 3 com 9, tornando 5 e 1 os novos filhos de 3. Assim, troca-se 3 com 5.
Assim, obtemos a árvore da direita da figura 3. Pelo outro lado, a figura 4 apresenta o passo seguinte, que envolve a eliminação do 9 da árvore.

Aqui, 9 tem sua posição trocada com 6, o elemento final da árvore. Em seguida, 9 é retirado da árvore e fixada na posição correta do vetor, fazendo a árvore resultante inválida mais uma vez. Daí, chama-se o Heapify no novo nó 6, que resultará na troca entre 6 e 8, e entre 6 e 7, para restaurar a propriedade da árvore. Com isso, 6 é trocado com 4, a nova posição final da árvore, e assim por diante.
O HeapSort não necessita de memória adicional além da própria função e do vetor de interesse, pois todos os rearranjos dos elementos são realizados dentro do vetor.

Este trabalho tem como objetivo desenvolver um sistema computacional para avaliar o desempenho de diversos algoritmos de ordenação de dados. Considerando a importância de algoritmos eficientes na análise e manipulação de grandes volumes de dados, o sistema proposto visa fornecer uma análise comparativa de desempenho utilizando dados internos e externos. Através do sistema, será possível mensurar o tempo de execução de cada algoritmo em diferentes condições, permitindo identificar suas vantagens e limitações.
Geração e/ou Obtenção de Dados para Ordenação:
Os dados são elementos essenciais para a análise deste trabalho, uma vez que eles permitem verificar a eficácia dos algoritmos estudados. Assim, foram desenvolvidas duas versões para cada programa de ordenação: uma para obter os dados a partir de arquivos de texto e outra para gerar números aleatórios. Dessa forma, busca-se proporcionar maior embasamento e rigor à análise, aproximando-se de resultados mais precisos e consistentes.
1.1 Obtenção de Dados:
Para a obtenção dos dados, foram utilizadas informações fornecidas pelo professor, organizadas em uma pasta principal que contém subpastas com conjuntos de dados de diferentes tamanhos (dez, cem, mil, dez

mil, cinquenta mil, cem mil, quinhentos mil e cinco mil). Cada subpasta possui versões com dados duplicados e sem duplicados, permitindo uma análise mais abrangente. Dentro delas, possuíam mais outras cinco opções, aleatório, côncavo-decresce cresce, convexo-cresce decresce, crescente e decrescente, todos, continham dez arquivos de dados, cada uma tendo a característica descrita pela pasta , como pode ser visto na imagem abaixo.

Figura 1 ? Pasta com dados e caminho mostrando algumas das pastas

Fonte: De autoria própria.

No entanto, era necessário fazer o sistema ter a capacidade de leitura desses arquivos que até então ainda era impossível, portanto, foi usado a estrutura FILE, contida dentro da biblioteca de entradas e saídas padrão da linguagem de programação C (stdio.h). Por haver certas dificuldades para fazer com que o sistema percorre o caminho até onde era encontrado o arquivo, todo o processo de testes foi feito com os arquivos dentro da mesma pasta do programa.

A leitura do acervo de informações, é feita da seguinte maneira, cria-se uma variável de ponteiro, capaz de armazenar elementos de maneira dinâmica, ela faz a leitura dos arquivos, verificando também se é possível ou não a alocação de memória, caso tudo ocorra da maneira correta, uma função de loop, irá transferir todos os números guardados no arquivo para o vetor criado pelo ponteiro, até ele não ser capaz de encontrar mais dados, a figura abaixo mostra o trecho do código que faz todo esse processo.

Figura 2 ? Trecho do sistema de obtenção de dados.

Fonte: De autoria própria.

1.2 Geração dos dados

Em contrapartida, na geração de dados, foi utilizado a função rand(), da qual retornará um número-pseudo-aleatório, é dito como pseudo, porque ele não é completamente aleatório, já que é feito um cálculo especial usando em base um número padrão que é tomado como semente para gerar os seguintes, tendo como padrão o um. Há uma outra forma de se gerar esses dados, no entanto, seria utilizado o comando srand(), este teria uma semente diferente fazendo com que os números pudessem ser mais diferentes entre si, porém ele não foi utilizado no programa.

Conforme visto anteriormente, fora utilizado a função rand, para a geração de dados, porém é necessário um loop que possa ser capaz de criar esses números até o tamanho limite indicado pelo desenvolvedor. Consequentemente, foi usado um loop for, que fará a tarefa repetidas vezes até alcançar o tamanho limite indicado, a figura abaixo mostra o trecho do programa descrito acima:

Figura 3 ? Trecho do sistema de geração de dados

Fonte: De autoria própria

Processo de Ordenação de Dados:

Por outro lado, há o processo de ordenação, neste trabalho foram citadas e utilizadas três das principais, geralmente, as mais usadas, são elas, Quick Sort, Heap Sort, e Merge Sort, cada uma possuindo seu método específico de organizar os dados, mas todas as escolhidas, são consideradas as mais rápidas dentre as mais populares.

Cada algoritmo possui uma complexidade de tempo e de espaço, que permite prever, respectivamente, o tempo necessário para a execução do algoritmo e a quantidade de memória exigida para armazenar as estruturas necessárias durante o processamento.

O uso desses cálculos é extremamente importante para ver se é possível ou não o uso desse programa na máquina que será testada, caso contrário, diversos tipos de erros podem acabar ocorrendo, afetando diretamente o resultado.

2.1 Quick Sort:

O algoritmo Quick Sort, traduzido como ?ordenação rápida?, é conhecido por sua eficiência e velocidade na ordenação de dados. Sua eficiência se deve à estratégia de ?Divisão e Conquista?, que divide o vetor em partes menores, organiza cada parte individualmente e, em seguida, combina as segmentações ordenadas para obter o resultado.
Segundo Cormen et al. (2012), essa abordagem é especialmente eficiente em algoritmos de ordenação, pois permite reduzir o tempo de execução para uma complexidade de O(n log n), na média dos casos.
No entanto, para poder dividir, ele precisa encontrar um ponto para dividir, essa necessidade é resolvida após a escolha de um elemento na lista como um pivô, este número será o valor de referência usado para dividir a lista em duas partes, elementos maiores, e menores.

Figura 4 ? Trecho do algoritmo Quick Sort realizando o loop de partição do vetor
Fonte: De própria autoria
Assim como na figura acima, o algoritmo então particiona o vetor, fazendo com que os elementos menores que o pivô se vão para a esquerda e os maiores fiquem à sua direita, esse processo é feito recursivamente para os subvetores decorrentes. Este processo continua a ocorrer até que a lista possua no máximo um elemento, pois assim, ela já estaria ordenada.
A figura abaixo mostra o todo processo descrito anteriormente, dentro do sistema computacional.

Figura 5 ? Algoritmo de ordenação QuickSort

Fonte: De própria autoria
Já a complexidade de tempo média dele é representada pela fórmula:
O(n log n) (1)
Isto significa, que, ele realiza um número de comparações e movimentações que equivalem à fórmula (1), onde n é o tamanho da lista a ser ordenada. Portanto, este é um dos motivos que faz este algoritmo ser

tão eficiente, o seu tempo relativo já é suficientemente bom.
E por último, ele não possui cálculo de armazenamento, porque o Quick Sort é um algoritmo ?in-place?, ou seja, não requer memória adicional para armazenar as estruturas temporárias durante a ordenação. Tornando-o extremamente eficiente em termos de uso de memória.
2.2 Heap Sort:
O algoritmo Heap Sort funciona como uma árvore binária especial, na qual o valor de cada pai é maior ou igual ao valor de seus filhos (max-heap) ou menor ou igual (min-heap). No contexto do heapsort, é usado o max-folhas (max-heap), ou seja, ele irá tentar buscar o maior valor e colocá-lo no topo da árvore, e sempre seguindo dessa maneira até conseguir ordenar corretamente os elementos guardados.
Primeiramente, o algoritmo começa a iniciar o processo de transformação do vetor em uma árvore, sempre buscando o maior número e colocando-o ao topo da árvore, o funcionamento é parecido com o de uma pirâmide, esta figura abaixo representa visualmente como é o funcionamento.
Após o maior número atingir o topo da árvore, ele é movido para a posição final da lista e removido da árvore, então ocorre uma reorganização entre todas as folhas para levar o maior número para o topo novamente, para isso que ocorra até todos os elementos serem levados para o vetor ao ponto de ele estar completo, com todos os seus dados de volta.
Em suma, o algoritmo Heap, cria uma árvore binária e realiza uma constante busca para levar todos os números ao topo para que assim consigam ser organizados do maior para o menor. Em relação à complexidade de tempo médio, o cálculo do algoritmo Heap Sort é o mesmo que do Quick Sort (1), portanto possui as mesmas qualidades dele, sendo tão rápido quanto ele.
Assim como o tempo, sua complexidade de espaço também é a mesma do Quick, os dois são considerados in-place, ele usa o mesmo vetor para armazenar os elementos e para o processo de ordenação, sendo extremamente eficiente em situações em que se há pouca memória no computador. A figura abaixo mostra o algoritmo do Heap Sort por completo.
Figura 7 ? Algoritmo de ordenação Heap Sort

Fonte: De própria autoria
2.3 Merge Sort:
O Merge Sort se assemelha, muito com o algoritmo Quick Sort, seja por conta da sua estratégia, mas também por conta de sua versatilidade, eficiência e velocidade em organizar um vetor desorganizado.
Em comparação, os dois utilizam a estratégia, divisão e conquista, porém, uma grande diferença acaba os diferenciando e mudando radicalmente, isto é, a aplicação dessa estratégia é diferente para cada um deles .
Enquanto o algoritmo Quick Sort, acaba necessitando de um pivô, o Merge não se prende a isso, pois ele divide a lista em parte iguais e em seguida, combina essas partes ordenadas, completando a ordenação por completo. Em suma, um deles encontra o meio e divide e o outro escolhe um pivô, tornando o que separa no meio, o melhor, principalmente nos piores casos, pois, o pivô escolhido pode acabar não sendo o do meio, dificultando bastante e aumentando a complexidade de tempo para a seguinte fórmula:
$O(n^2)(2)$
Por outro lado, o Merge Sort continua constante mesmo em situações em que existem muitos elementos a serem ordenados, também não dependendo da sorte, facilitando ainda mais a organização, a figura 9 mostra uma representação de como é realizada a separação:
Figura 8 - Algoritmo de ordenação Merge Sort

Fonte: De autoria própria

A figura acima mostra como é feita a divisão recursiva do Merge Sort, no programa usado neste trabalho. Como citado, a complexidade de tempo do algoritmo é sempre constante, em todos os casos, por conta de como ele aplica a estratégia, então ele mantém a fórmula (1) sempre.

No entanto, a complexidade de espaço, acaba sendo levemente diferente, pois ele possui um auxiliar que acaba consumindo memória, fazendo-o não se tornar in-place. Na figura 8, podemos ver a variável ?t? sendo usada como vetor auxiliar. Assim a fórmula de espaço acaba sendo a seguinte, onde n é o número de elementos:

O(n)(3)

Listagem dos Valores Antes e Depois da Ordenação:

A análise será feita lendo o mesmo arquivo de texto, um de dez dados duplicados, para ser possível diferenciar a forma que cada um ordena, explicando seus passos e métodos.

3.1 Quick Sort:

Figura 9 ? Reposta dada pelo algoritmo de ordenação Quick Sort

Fonte: De autoria própria

Na figura acima, é possível ver como funciona o processo de ordenação do algoritmo Quick Sort lendo o arquivo, no entanto, o código acaba tendo certos problemas em mostrar o elemento, como é possível ver na terceira linha de passos imprimida, o último elemento mostra um endereço de memória, por conta de uma das recursividades acabar utilizando o endereço, fazendo a impressão se tornar confusa.

Contudo, analisando algumas partes compreensíveis da imagem, pode-se concluir que os passos da ordenação prosseguem dividindo, utilizando o pivô escolhido que acaba mudando após um certo período de impressões, pois ele vai ordenando os subvetores restantes, utilizando recursividade e economizando na alocação de memória.

3.2 Heap Sort:

Figura 10 - Resposta dada pelo algoritmo de ordenação Heap Sort

Fonte: De própria autoria

Assim como explicado, o maior valor é sempre levado para o fim do vetor, na figura acima, é mais facilmente visível como é feita essa ação. À medida que se progride, é visível a sucessão do processo, com o algoritmo mostrando a movimentação dos maiores elementos para a parte mais à extremidade esquerda, então, após percorrer por todas as folhas, move-as para o fim do vetor.

Por conta de ele ser uma árvore, as impressões acabam sendo um pouco maior, porém, é visível, em comparação com o Quick que por sua vez, acaba mostrando os endereços de memória.

3.3 Merge Sort:

Figura 11 - Resposta dada pelo algoritmo de ordenação Merge Sort

Fonte: De autoria própria

É imprimido de maneira bem similar, como o Quick Sort, porém de maneira visível mostrando o número invés do endereço, isso ocorre, pois, a impressão está ocorrendo em um momento diferente do Quick, neste caso, ela está após a transferência do vetor temporário, o que faz com que não seja imprimida o endereço de memória.

Portanto, é mais perceptível como funciona o procedimento de divisão e conquista, vendo como o 958 é lentamente passado para o meio, até aquele momento ele é considerado o maior número da primeira metade fazendo-o fica naquela posição e então cada metade vai se organizando.
Apresentação dos Resultados Comparativos de Performance:
Esta seção apresenta uma análise de resultados entre os algoritmos utilizando quatro tipos de cenários diferentes entre eles, podendo ser duplicados ou não. Cada um dos programas foi avaliado quanto ao tempo médio de execução, visando descobrir o melhor, dependendo em cada ocasião. Os testes foram feitos com arquivos de texto com 50 mil dados e os gerados tiveram 100 mil amostras.
4.1 Medição:
Para obter os resultados, foi necessário o uso da função clock() da biblioteca da linguagem de programação C, da qual pode medir, os milissegundos, com mais precisão, o momento exato que é executado o algoritmo de ordenação, sem contar o tempo de criação de variáveis, alocação de memória e obtenção ou geração de dados.
Desta maneira, foram obtidos os dados com o tempo médio em (ms) apresentados na seguinte tabela:
Tabela 1 ? Comparação do Tempo Médio de Execução dos Algoritmos com Dados Duplicados
Fonte: De autoria própria
Tabela 2 - Comparação do Tempo Médio de Execução dos Algoritmos sem Dados Duplicados
Fonte: De autoria própria
Com estas tabelas, é possível observar que, em situações de dados espalhados de forma aleatória, o algoritmo Heap Sort é um dos piores no quesito tempo, perdendo por poucos milissegundos.
No entanto, em situações em que os elementos são espalhados de forma côncava, caso os dados sejam duplicados, o Quick acaba perdendo, mas em situações sem duplicidade, o Heap acaba tendo uma desvantagem de 515 milissegundos, uma diferença que quando comparada em situações de mais dados, faz com que ele não seja tão eficiente.
Toda essa diferença entre os dois, ocorre devido a diferença da estratégia entre eles, enquanto um necessita fazer uma árvore e deve percorrê-la, o outro apenas se divide e compara até conseguir se ordenar.
Apesar do Quick conseguir ser mais veloz que o Heap nesses cenários, existem muitos casos em que a sua estratégia não funciona tão bem, por exemplo, com grandes quantidades de números, esse cenário faz com que o Quick se torne extremamente instável nessas ocasiões. Pois, a forma em que ele exerce seu plano, depende muito de o pivô estar mais perto do centro ou não, podendo facilitar e muito em arquivos de dados pequenos.
Algo que não ocorre com o Merge Sort, que por sua vez, acaba brilhando nessas situações já que a aplicação da estratégia Divisão e Conquista é diferente, fazendo uma divisão por 2, não precisando ser tão dependente de achar ou não um pivô mais próximo do centro do vetor, já que quando dividido, sempre cairá ao meio.
No entanto, o algoritmo Merge Sort acaba tendo uma grande desvantagem em comparação aos outros dois algoritmos, o armazenamento de espaço, o que torna ele estável na maior parte das situações, tende a deixá-lo lento, pois ocorre a transferência de números do vetor temporário ao original.

Ref Biblio:
CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: Teoria e Prática. 3. ed. Rio de Janeiro: Elsevier, 2012.
SEDGEWICK, Robert; WAYNE, Kevin. Algorithms. 4th ed. Boston: Addison-Wesley, 2011.
MANZANO, José Augusto N. G.; OLIVEIRA, Jayr Figueiredo de. Estruturas de Dados e Algoritmos. 3. ed. São Paulo: Érica, 2011.
SIPSER, Michael. Introduction to the Theory of Computation. 3rd ed. Boston: Cengage Learning, 2012.
KNUTH, Donald E. The Art of Computer Programming: Sorting and Searching. 2nd ed. Massachusetts: Addison-Wesley, 1998.
DEITEL, H.; DEITEL, P. C: Como Programar. 7. ed. São Paulo: Pearson, 2010.
Resultados e Discussão

Como visto previamente neste documento, cada tipo de método de ordenação tem características próprias que o tornam altamente eficiente em algumas ocasiões e altamente ineficiente em outras. No entanto torna-se necessário uma análise mais profunda destes dados para um melhor entendimento dos mesmos. Para tal feito, foram coletadas mais de 800 amostras visando um melhor entendimento de cada método de ordenação.
Todas as amostras foram coletadas tendo como princípio a meta de 400 dados para cada tipo de método de ordenação, separados em dois subgrupos de 200 dados, respectivamente com e sem duplicidade de dados. Cada um destes subgrupos foi dividido em 5 subgrupos de 40 dados cada (aleatório, côncavo, convexo, crescente, decrescente) e um subgrupo com 100 dados(gerado pelo nosso grupo). Cada dado era composto por 50.000 números(como demonstrado abaixo em nossa imagem de autoria própria).

Ao serem analisados foi possível se tirar a seguinte tabela (autoria própria) como conclusão:

Os números com código de cores ao lado na parte inferior dos gráficos acima representam o tempo (em segundos) que cada algoritmo demorou para ser executado, já os números ao lado da palavra ?Frequência? representam a frequência de cada um dos números com código de cores ao lado.
Os gráficos acima permitem concluir diversos fatores. Por exemplo, os métodos de ordenação heap e merge têm uma variação menor do que o quick sort em termos de análise de dados. No entanto, o método merge possui uma demora muito maior para processar os dados que o quick e o heap.
Com tais dados foi possível montar uma curva normal extremamente detalhada para cada tipo de algoritmo de ordenação, com e sem duplicidade de números, e para os números também localmente gerados a partir de métodos de ordenação. Abaixos estão respectivamente as curvas normais (autoria própria) dos dados listados acima :

Os grupos de curvas normais apresentados acima são respectivamente pertencentes aos métodos de ordenação hippie, quick e merge. Com os códigos de cores preto para ordenação com duplicidade, vermelho para ordenação se duplicidade e azul para os dados coletados a partir de dados gerados localmente.
É possível notar que em cada grupo é apresentado vários símbolos com significados diversos. O símbolo ? denota a média de todos os dados coletados. O símbolo ? descreve o desvio padrão do objeto em estudo.
O número indicado no balão é o mesmo que o do símbolo X. A porcentagem que se segue no meio da curva de sino traz qual a probabilidade de haver números abaixo de X o que, por inferência, nos traz uma taxa de confiança de 95% para x ? X. Portanto pode-se afirmar que nenhum número recairá abaixo de X a não ser por acaso, e que X deve ser tomado como nosso objeto de estudo atual ignorando-se todos os tempos acima do mesmo.
Nota-se também que o método de ordenação quick está com menos dados na parte de com e sem duplicidade. Isso se deve ao fato do erro Stack overflow ter ocorrido durante a execução de ambos os dados decrescente e crescente, então se foi executado apenas uma vez cada e posto o tempo de demora para o código sair.
Com isso em mente é possível concluir que os métodos de ordenação quick e heap, são realmente mais rápidos em algumas ocasiões. No entanto, os métodos que apresentam mais estabilidade durante a execução são os métodos heap e merge. Há também a probabilidade de o método quick dar o erro Stack overflow devido a quantidade de memória ram que ocupa durante o processo de execução. Por fim, é possível concluir que o método de ordenação mais estável e o mais rápido é o heap.
Concluimos que os algoritmos de ordenação Quick Sort, Merge Sort e Heap Sort, mesmo que apresentem complexidades de tempo semelhantes , demonstram comportamentos distintos em diferentes cenários.

Algoritmos de ordenação estáveis, como o Merge Sort , são úteis em situações em que a preservação da ordem relativa é importante, como ordenação de registros em bancos de dados ou classificação por múltiplos critérios. Por outro lado, algoritmos de ordenação instáveis, como o Quicksort, podem alterar a ordem de elementos com chaves iguais durante o processo de classificação.
O Quick Sort e Heap Sort tiveram um desempenho melhor que o Merge sort sem duplicidade e com duplicidade, onde o Quick Sort se sobrepõe ao Heap Sort tendo menos frequencia, com uma diferença de 25 no tem 0,006 sem duplicidade e com duplicidade um diferença de 42.
Portanto a escolha do algoritmmo de ordenação ideal depende de diversas situações, como o tamho do conjunto de dados, os requisitos da memória ea estabilidade. No geral o Quick Sort será uma excelente opção para a maioria dos casos devido ao seu desempenho .No entanto o Heap Sort pode ser uma boa opção para grandes conjutos de dados. O Merge Sort pode ser uma boa opção quando o espaço de memória for limitado.

AlgoritmoAleatório (seg.)Côncavo (seg.)Convexo (seg.)Gerado (seg.)
Quick Sort0.005550.006450.00540,00527
Heap Sort0.0077750.0058750.00580,01695
Merge Sort0.011150.008150.0082750,011

AlgoritmoAleatório (seg.)Côncavo (seg.)Convexo (seg.)
Quick Sort0.0056750.006450.005875
Heap Sort0.00750.01160.0065
Merge Sort0.01150.00850.0085

==========

Arquivo 1: Trabalho_aps.docx (4117 termos)
Arquivo 2: https://www.zeusdobrasil.com.br/produto/espelho-de-seguranca-convexo-com-suporte-borda-borracha-329 (771 termos)
Termos comuns: 1
Similaridade: 0,02%
O texto abaixo é o conteúdo do documento Trabalho_aps.docx (4117 termos)
Os termos em vermelho foram encontrados no documento https://www.zeusdobrasil.com.br/produto/espelho-de-seguranca-convexo-com-suporte-borda-borracha-329 (771 termos)

==========

UNIVERSIDADE PAULISTA

G85JFG6 ? Eric Mitchell Barbosa Durham
N078146 ? Rafael Hiro Portilho
R0238H0 ? Matheus Reis Oliveira
N4031A9 ? Humberto Joji Fukui

Ordenação de dados amazônicos
Análise de algoritmos de ordenação de dados

São Paulo
2024

Neste trabalho tem o intuito de apresentar e comparar sobre os algoritmos Merge Sort, Quicksort e Heap Sort o seu desenvolvimento, eficiencia e complexidade. Avaliaremos os resultados adquiridos por cada algoritmo e discutir a aplicação desses agoritmos em alguns tipos de situações e uma analise do tempo que consiguram resolver cada situação. Os algoritmos de ordenações desenpenham papeis fundamentais na organização e otmização de dados, tornando-se fundamental para a eficiência para operações de busca e armazenamento em sistemas em sistemas de bancos de dados e outras aplicações computacionais.

Os algoritmos Merge sort, Heap sort e Quicksort conhecidos por sua eficiência e são amplamente utilizados em diversas linguagens de programação e bibliotecas. Cada um deles possui suas próprias características e vantagens, que serão exploradas em detalhes neste trabalho.

Quick sort se destaca por sua escolha programada de pivôs e rearranjo recursivo dos elementos, apresenta uma complexidade média de O(n log n), por outro lado o Merge sort conhecido por sua eficiencia na ordenação por divisão e conquista, destaca-se pela sua eficiência e estabilidade, dando uma ordenação eficaz dos elementos com complexidade O(n log n).O Heap sort, oferece um ordenação eficiente com complexidade O(n log n), destacando-se pela sua robustez.

Ao estudar esses algoritimos, temos como objetivo mostrar uma visão abrangente de suas características, desempenho e enetendimento a diferentes contextos, auxiliando na seleção do algoritmo mais adequado de acordo com a necessidade especifica de cada situação aplicada a eles.

2 REFERENCIAL TEÓRICO

2.1 QUICKSORT

Esse algoritmo usa a técnica da "divisão e conquista", que consiste na divisão recursiva de um problema em vários menores. No caso, o QuickSort escolhe um elemento do conjunto como pivô, sendo esse geralmente o primeiro elemento. Daí, reorganiza-se os demais elementos de forma que apenas os elementos menores que o pivô se situam à esquerda deste, assim formando dois subconjuntos em cada lado do pivô: um com elementos menores e outro com elementos maiores.

Na configuração resultante, o pivô passa a estar necessariamente na posição correta, e é então fixado. O processo descrito acima é repetido nos subconjuntos formados, definindo novo pivôs em cada. Isso

continua recursivamente, formando conjuntos cada vez menores até o surgimento de conjuntos unitários, conforme esquematizado na figura 1.

O QuickSort é considerado relativamente eficiente dentre os algoritmos de ordenação mais usados. Entretanto, há algumas desvantagens principais como o uso de memória necessário, que aumenta consideravelmente com o número de elementos do vetor de interesse devido ao caráter recursivo do algoritmo.
Além disso, o QuickSort pode tornar-se bem ineficiente quando apresentado com vetores em certos arranjos. Por exemplo, podemos citar o caso quando a lista já está maiormente ordenada em ordem crescente ou decrescente, na qual o algoritmo não poderá fazer rearranjos significativos.

2.2 MERGESORT

Também se enquadrando na categoria de ?divisão e conquista?, o MergeSort divide o vetor em duas partes aproximadamente iguais, até a formação de conjuntos unitários similarmente ao QuickSort, antes de intercalar-se de volta na ordem reversa, até retornar ao tamanho inicial do vetor de interesse.
O processo de intercalação entre duas listas é feito inserindo os elementos de cada num vetor auxiliar, que é subsequentemente sobrescrito nos dois conjuntos iniciais. Compara-se um elemento de cada conjunto, inserindo sempre o menor (caso não haja valores iguais), começando com os respectivos primeiros elementos e avançando conforme a inserção, de forma independente.
Realizando-se essa junção a partir dos conjuntos unitários formados, os subconjuntos a serem intercalados sempre estão organizados internamente, facilitando a organização final como um todo.

Similarmente ao QuickSort, esse algoritmo exige uso de memória além do próprio vetor de interesse por ser recursivo.

2.3 HEAPSORT

É um algoritmo baseado na árvore binária máxima. Nesse tipo de sequência, cada elemento pai tem no máximo dois elementos filhos associados de valores igual ao menor que o pai. Os elementos filhos, por sua vez, podem ter filhos próprios, assim formando uma estrutura similar a uma árvore, onde o primeiro elemento é considerado a raiz da estrutura como um todo.
Aqui, utiliza-se repetidamente um método chamado de Heapify, descrito como segue: um elemento parente X é comparado com seus filhos (ou filho único). Se não existe filho superior ao X, o método é finalizado. Caso contrário, trocam-se as posições do X e do filho com valor superior, e esse processo continua na nova posição de X. Note que caso haja dois filhos com valor distintos maior que X, o maior será sempre escolhido.
No processo do HeapSort, incialmente o vetor é reorganizado até que se constitua numa árvore binária

máxima. Isso é feito aplicando Heapify em todos os elementos parentes (com pelo menos um filho), em ordem decrescente. Ao final, isso é suficiente em tornar o vetor completo numa árvore máxima válida. Em seguida, trocamos a raiz com o último elemento da árvore, fixando a raiz original na posição nova e desconsiderando-o do resto da árvore. Daí Heapify é chamado na nova raiz, e a troca é realizada novamente até todos os elementos serem eliminados da árvore, momento na qual o vetor passa a ser necessariamente ordenado.

Como exemplo, considere o vetor na esquerda da figura 3, que não constitui numa árvore válida ainda. Aplicando o HeapSort, chama-se o Heapify nos seguintes elementos, conforme a sequência:
Elemento 8: Como esse já é maior que os filhos, nenhuma troca é realizada;
Elemento 1: Existe filho com valor superior. Assim, troca-se 1 com 9, o filho de maior valor;
Elemento 7: Conforme a lógica do passo anterior, 7 é trocado com 8. Os novos filhos de 7, 4 e 6, são inferiores;
Elemento 3: Nesse ponto, os filhos passam a ser 8 e 9. Pela mesma lógica, troca-se 3 com 9, tornando 5 e 1 os novos filhos de 3. Assim, troca-se 3 com 5.
Assim, obtemos a árvore da direita da figura 3. Pelo outro lado, a figura 4 apresenta o passo seguinte, que envolve a eliminação do 9 da árvore.

Aqui, 9 tem sua posição trocada com 6, o elemento final da árvore. Em seguida, 9 é retirado da árvore e fixada na posição correta do vetor, fazendo a árvore resultante inválida mais uma vez. Daí, chama-se o Heapify no novo nó 6, que resultará na troca entre 6 e 8, e entre 6 e 7, para restaurar a propriedade da árvore. Com isso, 6 é trocado com 4, a nova posição final da árvore, e assim por diante.
O HeapSort não necessita de memória adicional além da própria função e do vetor de interesse, pois todos os rearranjos dos elementos são realizados dentro do vetor.

Este trabalho tem como objetivo desenvolver um sistema computacional para avaliar o desempenho de diversos algoritmos de ordenação de dados. Considerando a importância de algoritmos eficientes na análise e manipulação de grandes volumes de dados, o sistema proposto visa fornecer uma análise comparativa de desempenho utilizando dados internos e externos. Através do sistema, será possível mensurar o tempo de execução de cada algoritmo em diferentes condições, permitindo identificar suas vantagens e limitações.
Geração e/ou Obtenção de Dados para Ordenação:
Os dados são elementos essenciais para a análise deste trabalho, uma vez que eles permitem verificar a eficácia dos algoritmos estudados. Assim, foram desenvolvidas duas versões para cada programa de ordenação: uma para obter os dados a partir de arquivos de texto e outra para gerar números aleatórios. Dessa forma, busca-se proporcionar maior embasamento e rigor à análise, aproximando-se de resultados mais precisos e consistentes.
1.1 Obtenção de Dados:

Para a obtenção dos dados, foram utilizadas informações fornecidas pelo professor, organizadas em uma pasta principal que contém subpastas com conjuntos de dados de diferentes tamanhos (dez, cem, mil, dez mil, cinquenta mil, cem mil, quinhentos mil e cinco mil). Cada subpasta possui versões com dados duplicados e sem duplicados, permitindo uma análise mais abrangente. Dentro delas, possuíam mais outras cinco opções, aleatório, côncavo-decresce cresce, convexo-cresce decresce, crescente e decrescente, todos, continham dez arquivos de dados, cada uma tendo a característica descrita pela pasta , como pode ser visto na imagem abaixo.

Figura 1 ? Pasta com dados e caminho mostrando algumas das pastas

Fonte: De autoria própria.

No entanto, era necessário fazer o sistema ter a capacidade de leitura desses arquivos que até então ainda era impossível, portanto, foi usado a estrutura FILE, contida dentro da biblioteca de entradas e saídas padrão da linguagem de programação C (stdio.h). Por haver certas dificuldades para fazer com que o sistema percorre o caminho até onde era encontrado o arquivo, todo o processo de testes foi feito com os arquivos dentro da mesma pasta do programa.

A leitura do acervo de informações, é feita da seguinte maneira, cria-se uma variável de ponteiro, capaz de armazenar elementos de maneira dinâmica, ela faz a leitura dos arquivos, verificando também se é possível ou não a alocação de memória, caso tudo ocorra da maneira correta, uma função de loop, irá transferir todos os números guardados no arquivo para o vetor criado pelo ponteiro, até ele não ser capaz de encontrar mais dados, a figura abaixo mostra o trecho do código que faz todo esse processo.

Figura 2 ? Trecho do sistema de obtenção de dados.

Fonte: De autoria própria.

1.2 Geração dos dados

Em contrapartida, na geração de dados, foi utilizado a função rand(), da qual retornará um número-pseudo-aleatório, é dito como pseudo, porque ele não é completamente aleatório, já que é feito um cálculo especial usando em base um número padrão que é tomado como semente para gerar os seguintes, tendo como padrão o um. Há uma outra forma de se gerar esses dados, no entanto, seria utilizado o comando srand(), este teria uma semente diferente fazendo com que os números pudessem ser mais diferentes entre si, porém ele não foi utilizado no programa.

Conforme visto anteriormente, fora utilizado a função rand, para a geração de dados, porém é necessário um loop que possa ser capaz de criar esses números até o tamanho limite indicado pelo desenvolvedor. Consequentemente, foi usado um loop for, que fará a tarefa repetidas vezes até alcançar o tamanho limite indicado, a figura abaixo mostra o trecho do programa descrito acima:

Figura 3 ? Trecho do sistema de geração de dados

Fonte: De autoria própria

Processo de Ordenação de Dados:

Por outro lado, há o processo de ordenação, neste trabalho foram citadas e utilizadas três das principais, geralmente, as mais usadas, são elas, Quick Sort, Heap Sort, e Merge Sort, cada uma possuindo seu método específico de organizar os dados, mas todas as escolhidas, são consideradas as mais rápidas dentre as mais populares.

Cada algoritmo possui uma complexidade de tempo e de espaço, que permite prever, respectivamente, o tempo necessário para a execução do algoritmo e a quantidade de memória exigida para armazenar as estruturas necessárias durante o processamento.

O uso desses cálculos é extremamente importante para ver se é possível ou não o uso desse programa na máquina que será testada, caso contrário, diversos tipos de erros podem acabar ocorrendo, afetando

diretamente o resultado.

2.1 Quick Sort:

O algoritmo Quick Sort, traduzido como ?ordenação rápida?, é conhecido por sua eficiência e velocidade na ordenação de dados. Sua eficiência se deve à estratégia de ?Divisão e Conquista?, que divide o vetor em partes menores, organiza cada parte individualmente e, em seguida, combina as segmentações ordenadas para obter o resultado.

Segundo Cormen et al. (2012), essa abordagem é especialmente eficiente em algoritmos de ordenação, pois permite reduzir o tempo de execução para uma complexidade de O(n log n), na média dos casos.

No entanto, para poder dividir, ele precisa encontrar um ponto para dividir, essa necessidade é resolvida após a escolha de um elemento na lista como um pivô, este número será o valor de referência usado para dividir a lista em duas partes, elementos maiores, e menores.

Figura 4 ? Trecho do algoritmo Quick Sort realizando o loop de partição do vetor

Fonte: De própria autoria

Assim como na figura acima, o algoritmo então particiona o vetor, fazendo com que os elementos menores que o pivô se vão para a esquerda e os maiores fiquem à sua direita, esse processo é feito recursivamente para os subvetores decorrentes. Este processo continua a ocorrer até que a lista possua no máximo um elemento, pois assim, ela já estaria ordenada.

A figura abaixo mostra o todo processo descrito anteriormente, dentro do sistema computacional.

Figura 5 ? Algoritmo de ordenação QuickSort

Fonte: De própria autoria

Já a complexidade de tempo média dele é representada pela fórmula:

O(n log n) (1)

Isto significa, que, ele realiza um número de comparações e movimentações que equivalem à fórmula (1), onde n é o tamanho da lista a ser ordenada. Portanto, este é um dos motivos que faz este algoritmo ser tão eficiente, o seu tempo relativo já é suficientemente bom.
E por último, ele não possui cálculo de armazenamento, porque o Quick Sort é um algoritmo ?in-place?, ou seja, não requer memória adicional para armazenar as estruturas temporárias durante a ordenação. Tornando-o extremamente eficiente em termos de uso de memória.
2.2 Heap Sort:
O algoritmo Heap Sort funciona como uma árvore binária especial, na qual o valor de cada pai é maior ou igual ao valor de seus filhos (max-heap) ou menor ou igual (min-heap). No contexto do heapsort, é usado o max-folhas (max-heap), ou seja, ele irá tentar buscar o maior valor e colocá-lo no topo da árvore, e sempre seguindo dessa maneira até conseguir ordenar corretamente os elementos guardados.
Primeiramente, o algoritmo começa a iniciar o processo de transformação do vetor em uma árvore, sempre buscando o maior número e colocando-o ao topo da árvore, o funcionamento é parecido com o de uma pirâmide, esta figura abaixo representa visualmente como é o funcionamento.
Após o maior número atingir o topo da árvore, ele é movido para a posição final da lista e removido da árvore, então ocorre uma reorganização entre todas as folhas para levar o maior número para o topo novamente, para isso que ocorra até todos os elementos serem levados para o vetor ao ponto de ele estar completo, com todos os seus dados de volta.
Em suma, o algoritmo Heap, cria uma árvore binária e realiza uma constante busca para levar todos os números ao topo para que assim consigam ser organizados do maior para o menor. Em relação à complexidade de tempo médio, o cálculo do algoritmo Heap Sort é o mesmo que do Quick Sort (1), portanto possui as mesmas qualidades dele, sendo tão rápido quanto ele.
Assim como o tempo, sua complexidade de espaço também é a mesma do Quick, os dois são considerados in-place, ele usa o mesmo vetor para armazenar os elementos e para o processo de ordenação, sendo extremamente eficiente em situações em que se há pouca memória no computador. A figura abaixo mostra o algoritmo do Heap Sort por completo.
Figura 7 ? Algoritmo de ordenação Heap Sort

Fonte: De própria autoria
2.3 Merge Sort:
O Merge Sort se assemelha, muito com o algoritmo Quick Sort, seja por conta da sua estratégia, mas também por conta de sua versatilidade, eficiência e velocidade em organizar um vetor desorganizado.
Em comparação, os dois utilizam a estratégia, divisão e conquista, porém, uma grande diferença acaba os diferenciando e mudando radicalmente, isto é, a aplicação dessa estratégia é diferente para cada um deles .
Enquanto o algoritmo Quick Sort, acaba necessitando de um pivô, o Merge não se prende a isso, pois ele divide a lista em parte iguais e em seguida, combina essas partes ordenadas, completando a ordenação por completo. Em suma, um deles encontra o meio e divide e o outro escolhe um pivô, tornando o que separa no meio, o melhor, principalmente nos piores casos, pois, o pivô escolhido pode acabar não sendo o do meio, dificultando bastante e aumentando a complexidade de tempo para a seguinte fórmula:
$O(n^2)(2)$
Por outro lado, o Merge Sort continua constante mesmo em situações em que existem muitos elementos a serem ordenados, também não dependendo da sorte, facilitando ainda mais a organização, a figura 9 mostra uma representação de como é realizada a separação:

Figura 8 - Algoritmo de ordenação Merge Sort

Fonte: De autoria própria

A figura acima mostra como é feita a divisão recursiva do Merge Sort, no programa usado neste trabalho. Como citado, a complexidade de tempo do algoritmo é sempre constante, em todos os casos, por conta de como ele aplica a estratégia, então ele mantém a fórmula (1) sempre.

No entanto, a complexidade de espaço, acaba sendo levemente diferente, pois ele possui um auxiliar que acaba consumindo memória, fazendo-o não se tornar in-place. Na figura 8, podemos ver a variável ?t? sendo usada como vetor auxiliar. Assim a fórmula de espaço acaba sendo a seguinte, onde n é o número de elementos:

O(n)(3)

Listagem dos Valores Antes e Depois da Ordenação:

A análise será feita lendo o mesmo arquivo de texto, um de dez dados duplicados, para ser possível diferenciar a forma que cada um ordena, explicando seus passos e métodos.

3.1 Quick Sort:

Figura 9 ? Reposta dada pelo algoritmo de ordenação Quick Sort

Fonte: De autoria própria

Na figura acima, é possível ver como funciona o processo de ordenação do algoritmo Quick Sort lendo o arquivo, no entanto, o código acaba tendo certos problemas em mostrar o elemento, como é possível ver na terceira linha de passos imprimida, o último elemento mostra um endereço de memória, por conta de uma das recursividades acabar utilizando o endereço, fazendo a impressão se tornar confusa.

Contudo, analisando algumas partes compreensíveis da imagem, pode-se concluir que os passos da ordenação prosseguem dividindo, utilizando o pivô escolhido que acaba mudando após um certo período de impressões, pois ele vai ordenando os subvetores restantes, utilizando recursividade e economizando na alocação de memória.

3.2 Heap Sort:

Figura 10 - Resposta dada pelo algoritmo de ordenação Heap Sort

Fonte: De própria autoria

Assim como explicado, o maior valor é sempre levado para o fim do vetor, na figura acima, é mais facilmente visível como é feita essa ação. À medida que se progride, é visível a sucessão do processo, com o algoritmo mostrando a movimentação dos maiores elementos para a parte mais à extremidade esquerda, então, após percorrer por todas as folhas, move-as para o fim do vetor.

Por conta de ele ser uma árvore, as impressões acabam sendo um pouco maior, porém, é visível, em comparação com o Quick que por sua vez, acaba mostrando os endereços de memória.

3.3 Merge Sort:

Figura 11 - Resposta dada pelo algoritmo de ordenação Merge Sort

Fonte: De autoria própria

É imprimido de maneira bem similar, como o Quick Sort, porém de maneira visível mostrando o número invés do endereço, isso ocorre, pois, a impressão está ocorrendo em um momento diferente do Quick,

neste caso, ela está após a transferência do vetor temporário, o que faz com que não seja imprimida o endereço de memória.
Portanto, é mais perceptível como funciona o procedimento de divisão e conquista, vendo como o 958 é lentamente passado para o meio, até aquele momento ele é considerado o maior número da primeira metade fazendo-o fica naquela posição e então cada metade vai se organizando.
Apresentação dos Resultados Comparativos de Performance:
Esta seção apresenta uma análise de resultados entre os algoritmos utilizando quatro tipos de cenários diferentes entre eles, podendo ser duplicados ou não. Cada um dos programas foi avaliado quanto ao tempo médio de execução, visando descobrir o melhor, dependendo em cada ocasião. Os testes foram feitos com arquivos de texto com 50 mil dados e os gerados tiveram 100 mil amostras.
4.1 Medição:
Para obter os resultados, foi necessário o uso da função clock() da biblioteca da linguagem de programação C, da qual pode medir, os milissegundos, com mais precisão, o momento exato que é executado o algoritmo de ordenação, sem contar o tempo de criação de variáveis, alocação de memória e obtenção ou geração de dados.
Desta maneira, foram obtidos os dados com o tempo médio em (ms) apresentados na seguinte tabela:
Tabela 1 ? Comparação do Tempo Médio de Execução dos Algoritmos com Dados Duplicados
Fonte: De autoria própria
Tabela 2 - Comparação do Tempo Médio de Execução dos Algoritmos sem Dados Duplicados
Fonte: De autoria própria
Com estas tabelas, é possível observar que, em situações de dados espalhados de forma aleatória, o algoritmo Heap Sort é um dos piores no quesito tempo, perdendo por poucos milissegundos.
No entanto, em situações em que os elementos são espalhados de forma côncava, caso os dados sejam duplicados, o Quick acaba perdendo, mas em situações sem duplicidade, o Heap acaba tendo uma desvantagem de 515 milissegundos, uma diferença que quando comparada em situações de mais dados, faz com que ele não seja tão eficiente.
Toda essa diferença entre os dois, ocorre devido a diferença da estratégia entre eles, enquanto um necessita fazer uma árvore e deve percorrê-la, o outro apenas se divide e compara até conseguir se ordenar.
Apesar do Quick conseguir ser mais veloz que o Heap nesses cenários, existem muitos casos em que a sua estratégia não funciona tão bem, por exemplo, com grandes quantidades de números, esse cenário faz com que o Quick se torne extremamente instável nessas ocasiões. Pois, a forma em que ele exerce seu plano, depende muito de o pivô estar mais perto do centro ou não, podendo facilitar e muito em arquivos de dados pequenos.
Algo que não ocorre com o Merge Sort, que por sua vez, acaba brilhando nessas situações já que a aplicação da estratégia Divisão e Conquista é diferente, fazendo uma divisão por 2, não precisando ser tão dependente de achar ou não um pivô mais próximo do centro do vetor, já que quando dividido, sempre cairá ao meio.
No entanto, o algoritmo Merge Sort acaba tendo uma grande desvantagem em comparação aos outros dois algoritmos, o armazenamento de espaço, o que torna ele estável na maior parte das situações, tende a deixá-lo lento, pois ocorre a transferência de números do vetor temporário ao original.

Ref Biblio:
CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: Teoria e Prática. 3. ed. Rio de Janeiro: Elsevier, 2012.
SEDGEWICK, Robert; WAYNE, Kevin. Algorithms. 4th ed. Boston: Addison-Wesley, 2011.
MANZANO, José Augusto N. G.; OLIVEIRA, Jayr Figueiredo de. Estruturas de Dados e Algoritmos. 3. ed. São Paulo: Érica, 2011.
SIPSER, Michael. Introduction to the Theory of Computation. 3rd ed. Boston: Cengage Learning, 2012.
KNUTH, Donald E. The Art of Computer Programming: Sorting and Searching. 2nd ed. Massachusetts: Addison-Wesley, 1998.
DEITEL, H.; DEITEL, P. C: Como Programar. 7. ed. São Paulo: Pearson, 2010.
Resultados e Discussão

Como visto previamente neste documento, cada tipo de método de ordenação tem características próprias que o tornam altamente eficiente em algumas ocasiões e altamente ineficiente em outras. No entanto torna-se necessário uma análise mais profunda destes dados para um melhor entendimento dos mesmos. Para tal feito, foram coletadas mais de 800 amostras visando um melhor entendimento de cada método de ordenação.
Todas as amostras foram coletadas tendo como princípio a meta de 400 dados para cada tipo de método de ordenação, separados em dois subgrupos de 200 dados, respectivamente com e sem duplicidade de dados. Cada um destes subgrupos foi dividido em 5 subgrupos de 40 dados cada (aleatório, côncavo, convexo, crescente, decrescente) e um subgrupo com 100 dados(gerado pelo nosso grupo). Cada dado

era composto por 50.000 números(como demonstrado abaixo em nossa imagem de autoria própria).

Ao serem analisados foi possível se tirar a seguinte tabela (autoria própria) como conclusão:

Os números com código de cores ao lado na parte inferior dos gráficos acima representam o tempo (em segundos) que cada algoritmo demorou para ser executado, já os números ao lado da palavra ?Frequência? representam a frequência de cada um dos números com código de cores ao lado.
Os gráficos acima permitem concluir diversos fatores. Por exemplo, os métodos de ordenação heap e merge têm uma variação menor do que o quick sort em termos de análise de dados. No entanto, o método merge possui uma demora muito maior para processar os dados que o quick e o heap.
Com tais dados foi possível montar uma curva normal extremamente detalhada para cada tipo de algoritmo de ordenação, com e sem duplicidade de números, e para os números também localmente gerados a partir de métodos de ordenação. Abaixos estão respectivamente as curvas normais (autoria própria) dos dados listados acima :

Os grupos de curvas normais apresentados acima são respectivamente pertencentes aos métodos de ordenação hippie, quick e merge. Com os códigos de cores preto para ordenação com duplicidade, vermelho para ordenação se duplicidade e azul para os dados coletados a partir de dados gerados localmente.
É possível notar que em cada grupo é apresentado vários símbolos com significados diversos. O símbolo ? denota a média de todos os dados coletados. O símbolo ? descreve o desvio padrão do objeto em estudo.
O número indicado no balão é o mesmo que o do símbolo X. A porcentagem que se segue no meio da curva de sino traz qual a probabilidade de haver números abaixo de X o que, por inferência, nos traz uma taxa de confiança de 95% para x ? X. Portanto pode-se afirmar que nenhum número recairá abaixo de X a não ser por acaso, e que X deve ser tomado como nosso objeto de estudo atual ignorando-se todos os tempos acima do mesmo.
Nota-se também que o método de ordenação quick está com menos dados na parte de com e sem duplicidade. Isso se deve ao fato do erro Stack overflow ter ocorrido durante a execução de ambos os dados decrescente e crescente, então se foi executado apenas uma vez cada e posto o tempo de demora para o código sair.
Com isso em mente é possível concluir que os métodos de ordenação quick e heap, são realmente mais rápidos em algumas ocasiões. No entanto, os métodos que apresentam mais estabilidade durante a execução são os métodos heap e merge. Há também a probabilidade de o método quick dar o erro Stack overflow devido a quantidade de memória ram que ocupa durante o processo de execução. Por fim, é possível concluir que o método de ordenação mais estável e o mais rápido é o heap.

Concluimos que os algoritmos de ordenação Quick Sort, Merge Sort e Heap Sort, mesmo que apresentem complexidades de tempo semelhantes , demonstram comportamentos distintos em diferentes cenários. Algoritmos de ordenação estáveis, como o Merge Sort , são úteis em situações em que a preservação da ordem relativa é importante, como ordenação de registros em bancos de dados ou classificação por múltiplos critérios. Por outro lado, algoritmos de ordenação instáveis, como o Quicksort, podem alterar a ordem de elementos com chaves iguais durante o processo de classificação.
O Quick Sort e Heap Sort tiveram um desempenho melhor que o Merge sort sem duplicidade e com duplicidade, onde o Quick Sort se sobrepõe ao Heap Sort tendo menos frequencia, com uma diferença de 25 no tem 0,006 sem duplicidade e com duplicidade um diferença de 42.
Portanto a escolha do algoritmmo de ordenação ideal depende de diversas situações, como o tamho do conjunto de dados, os requisitos da memória ea estabilidade. No geral o Quick Sort será uma excelente opção para a maioria dos casos devido ao seu desempenho .No entanto o Heap Sort pode ser uma boa opção para grandes conjutos de dados. O Merge Sort pode ser uma boa opção quando o espaço de memória for limitado.

AlgoritmoAleatório (seg.)Côncavo (seg.)Convexo (seg.)Gerado (seg.)
Quick Sort0.005550.006450.00540,00527
Heap Sort0.0077750.0058750.00580,01695
Merge Sort0.011150.008150.0082750,011

AlgoritmoAleatório (seg.)Côncavo (seg.)Convexo (seg.)
Quick Sort0.0056750.006450.005875
Heap Sort0.00750.01160.0065
Merge Sort0.01150.00850.0085

==========

Arquivo 1: Trabalho_aps.docx (4117 termos)
Arquivo 2: http://www.w3.org/2000/svg (93 termos)
Termos comuns: 0
Similaridade: 0,00%
O texto abaixo é o conteúdo do documento Trabalho_aps.docx (4117 termos)
Os termos em vermelho foram encontrados no documento http://www.w3.org/2000/svg (93 termos)

==========

UNIVERSIDADE PAULISTA

G85JFG6 ? Eric Mitchell Barbosa Durham
N078146 ? Rafael Hiro Portilho
R0238H0 ? Matheus Reis Oliveira
N4031A9 ? Humberto Joji Fukui

Ordenação de dados amazônicos
Análise de algoritmos de ordenação de dados

São Paulo
2024

Neste trabalho tem o intuito de apresentar e comparar sobre os algoritmos Merge Sort, Quicksort e Heap Sort o seu desenvolvimento, eficiencia e complexidade. Avaliaremos os resultados adquiridos por cada algoritmo e discutir a aplicação desses agoritmos em alguns tipos de situações e uma analise do tempo que consiguram resolver cada situação. Os algoritmos de ordenações desenpenham papeis fundamentais na organização e otmização de dados, tornando-se fundamental para a eficiência para operações de busca e armazenamento em sistemas em sistemas de bancos de dados e outras aplicações computacionais.

Os algoritmos Merge sort, Heap sort e Quicksort conhecidos por sua eficiência e são amplamente utilizados em diversas linguagens de programação e bibliotecas. Cada um deles possui suas próprias características e vantagens, que serão exploradas em detalhes neste trabalho.

Quick sort se destaca por sua escolha programada de pivôs e rearranjo recursivo dos elementos, apresenta uma complexidade média de O(n log n), por outro lado o Merge sort conhecido por sua eficiencia na ordenação por divisão e conquista, destaca-se pela sua eficiência e estabilidade, dando uma ordenação eficaz dos elementos com complexidade O(n log n).O Heap sort, oferece um ordenação eficiente com complexidade O(n log n), destacando-se pela sua robustez.

Ao estudar esses algoritimos, temos como objetivo mostrar uma visão abrangente de suas características, desempenho e enetendimento a diferentes contextos, auxiliando na seleção do algoritmo mais adequado de acordo com a necessidade especifica de cada situação aplicada a eles.

2 REFERENCIAL TEÓRICO

2.1 QUICKSORT

Esse algoritmo usa a técnica da "divisão e conquista", que consiste na divisão recursiva de um problema em vários menores. No caso, o QuickSort escolhe um elemento do conjunto como pivô, sendo esse geralmente o primeiro elemento. Daí, reorganiza-se os demais elementos de forma que apenas os elementos menores que o pivô se situam à esquerda deste, assim formando dois subconjuntos em cada lado do pivô: um com elementos menores e outro com elementos maiores.

Na configuração resultante, o pivô passa a estar necessariamente na posição correta, e é então fixado. O processo descrito acima é repetido nos subconjuntos formados, definindo novo pivôs em cada. Isso continua recursivamente, formando conjuntos cada vez menores até o surgimento de conjuntos unitários, conforme esquematizado na figura 1.

O QuickSort é considerado relativamente eficiente dentre os algoritmos de ordenação mais usados. Entretanto, há algumas desvantagens principais como o uso de memória necessário, que aumenta consideravelmente com o número de elementos do vetor de interesse devido ao caráter recursivo do algoritmo.
Além disso, o QuickSort pode tornar-se bem ineficiente quando apresentado com vetores em certos arranjos. Por exemplo, podemos citar o caso quando a lista já está maiormente ordenada em ordem crescente ou decrescente, na qual o algoritmo não poderá fazer rearranjos significativos.

2.2 MERGESORT

Também se enquadrando na categoria de ?divisão e conquista?, o MergeSort divide o vetor em duas partes aproximadamente iguais, até a formação de conjuntos unitários similarmente ao QuickSort, antes de intercalar-se de volta na ordem reversa, até retornar ao tamanho inicial do vetor de interesse.
O processo de intercalação entre duas listas é feito inserindo os elementos de cada num vetor auxiliar, que é subsequentemente sobrescrito nos dois conjuntos iniciais. Compara-se um elemento de cada conjunto, inserindo sempre o menor (caso não haja valores iguais), começando com os respectivos primeiros elementos e avançando conforme a inserção, de forma independente.
Realizando-se essa junção a partir dos conjuntos unitários formados, os subconjuntos a serem intercalados sempre estão organizados internamente, facilitando a organização final como um todo.

Similarmente ao QuickSort, esse algoritmo exige uso de memória além do próprio vetor de interesse por ser recursivo.

2.3 HEAPSORT

É um algoritmo baseado na árvore binária máxima. Nesse tipo de sequência, cada elemento pai tem no máximo dois elementos filhos associados de valores igual ao menor que o pai. Os elementos filhos, por sua vez, podem ter filhos próprios, assim formando uma estrutura similar a uma árvore, onde o primeiro elemento é considerado a raiz da estrutura como um todo.
Aqui, utiliza-se repetidamente um método chamado de Heapify, descrito como segue: um elemento parente X é comparado com seus filhos (ou filho único). Se não existe filho superior ao X, o método é finalizado. Caso contrário, trocam-se as posições do X e do filho com valor superior, e esse processo continua na nova posição de X. Note que caso haja dois filhos com valor distintos maior que X, o maior será sempre escolhido.
No processo do HeapSort, incialmente o vetor é reorganizado até que se constitua numa árvore binária máxima. Isso é feito aplicando Heapify em todos os elementos parentes (com pelo menos um filho), em ordem decrescente. Ao final, isso é suficiente em tornar o vetor completo numa árvore máxima válida.
Em seguida, trocamos a raiz com o último elemento da árvore, fixando a raiz original na posição nova e

desconsiderando-o do resto da árvore. Daí Heapify é chamado na nova raiz, e a troca é realizada novamente até todos os elementos serem eliminados da árvore, momento na qual o vetor passa a ser necessariamente ordenado.

Como exemplo, considere o vetor na esquerda da figura 3, que não constitui numa árvore válida ainda. Aplicando o HeapSort, chama-se o Heapify nos seguintes elementos, conforme a sequência:
Elemento 8: Como esse já é maior que os filhos, nenhuma troca é realizada;
Elemento 1: Existe filho com valor superior. Assim, troca-se 1 com 9, o filho de maior valor;
Elemento 7: Conforme a lógica do passo anterior, 7 é trocado com 8. Os novos filhos de 7, 4 e 6, são inferiores;
Elemento 3: Nesse ponto, os filhos passam a ser 8 e 9. Pela mesma lógica, troca-se 3 com 9, tornando 5 e 1 os novos filhos de 3. Assim, troca-se 3 com 5.
Assim, obtemos a árvore da direita da figura 3. Pelo outro lado, a figura 4 apresenta o passo seguinte, que envolve a eliminação do 9 da árvore.

Aqui, 9 tem sua posição trocada com 6, o elemento final da árvore. Em seguida, 9 é retirado da árvore e fixada na posição correta do vetor, fazendo a árvore resultante inválida mais uma vez. Daí, chama-se o Heapify no novo nó 6, que resultará na troca entre 6 e 8, e entre 6 e 7, para restaurar a propriedade da árvore. Com isso, 6 é trocado com 4, a nova posição final da árvore, e assim por diante.
O HeapSort não necessita de memória adicional além da própria função e do vetor de interesse, pois todos os rearranjos dos elementos são realizados dentro do vetor.

Este trabalho tem como objetivo desenvolver um sistema computacional para avaliar o desempenho de diversos algoritmos de ordenação de dados. Considerando a importância de algoritmos eficientes na análise e manipulação de grandes volumes de dados, o sistema proposto visa fornecer uma análise comparativa de desempenho utilizando dados internos e externos. Através do sistema, será possível mensurar o tempo de execução de cada algoritmo em diferentes condições, permitindo identificar suas vantagens e limitações.
Geração e/ou Obtenção de Dados para Ordenação:
Os dados são elementos essenciais para a análise deste trabalho, uma vez que eles permitem verificar a eficácia dos algoritmos estudados. Assim, foram desenvolvidas duas versões para cada programa de ordenação: uma para obter os dados a partir de arquivos de texto e outra para gerar números aleatórios. Dessa forma, busca-se proporcionar maior embasamento e rigor à análise, aproximando-se de resultados mais precisos e consistentes.
1.1 Obtenção de Dados:
Para a obtenção dos dados, foram utilizadas informações fornecidas pelo professor, organizadas em uma pasta principal que contém subpastas com conjuntos de dados de diferentes tamanhos (dez, cem, mil, dez mil, cinquenta mil, cem mil, quinhentos mil e cinco mil). Cada subpasta possui versões com dados

duplicados e sem duplicados, permitindo uma análise mais abrangente. Dentro delas, possuíam mais outras cinco opções, aleatório, côncavo-decresce cresce, convexo-cresce decresce, crescente e decrescente, todos, continham dez arquivos de dados, cada uma tendo a característica descrita pela pasta , como pode ser visto na imagem abaixo.

Figura 1 ? Pasta com dados e caminho mostrando algumas das pastas

Fonte: De autoria própria.

No entanto, era necessário fazer o sistema ter a capacidade de leitura desses arquivos que até então ainda era impossível, portanto, foi usado a estrutura FILE, contida dentro da biblioteca de entradas e saídas padrão da linguagem de programação C (stdio.h). Por haver certas dificuldades para fazer com que o sistema percorre o caminho até onde era encontrado o arquivo, todo o processo de testes foi feito com os arquivos dentro da mesma pasta do programa.

A leitura do acervo de informações, é feita da seguinte maneira, cria-se uma variável de ponteiro, capaz de armazenar elementos de maneira dinâmica, ela faz a leitura dos arquivos, verificando também se é possível ou não a alocação de memória, caso tudo ocorra da maneira correta, uma função de loop, irá transferir todos os números guardados no arquivo para o vetor criado pelo ponteiro, até ele não ser capaz de encontrar mais dados, a figura abaixo mostra o trecho do código que faz todo esse processo.

Figura 2 ? Trecho do sistema de obtenção de dados.

Fonte: De autoria própria.

1.2 Geração dos dados

Em contrapartida, na geração de dados, foi utilizado a função rand(), da qual retornará um número-pseudo-aleatório, é dito como pseudo, porque ele não é completamente aleatório, já que é feito um cálculo especial usando em base um número padrão que é tomado como semente para gerar os seguintes, tendo como padrão o um. Há uma outra forma de se gerar esses dados, no entanto, seria utilizado o comando srand(), este teria uma semente diferente fazendo com que os números pudessem ser mais diferentes entre si, porém ele não foi utilizado no programa.

Conforme visto anteriormente, fora utilizado a função rand, para a geração de dados, porém é necessário um loop que possa ser capaz de criar esses números até o tamanho limite indicado pelo desenvolvedor. Consequentemente, foi usado um loop for, que fará a tarefa repetidas vezes até alcançar o tamanho limite indicado, a figura abaixo mostra o trecho do programa descrito acima:

Figura 3 ? Trecho do sistema de geração de dados

Fonte: De autoria própria

Processo de Ordenação de Dados:

Por outro lado, há o processo de ordenação, neste trabalho foram citadas e utilizadas três das principais, geralmente, as mais usadas, são elas, Quick Sort, Heap Sort, e Merge Sort, cada uma possuindo seu método específico de organizar os dados, mas todas as escolhidas, são consideradas as mais rápidas dentre as mais populares.

Cada algoritmo possui uma complexidade de tempo e de espaço, que permite prever, respectivamente, o tempo necessário para a execução do algoritmo e a quantidade de memória exigida para armazenar as estruturas necessárias durante o processamento.

O uso desses cálculos é extremamente importante para ver se é possível ou não o uso desse programa na máquina que será testada, caso contrário, diversos tipos de erros podem acabar ocorrendo, afetando diretamente o resultado.

2.1 Quick Sort:

O algoritmo Quick Sort, traduzido como ?ordenação rápida?, é conhecido por sua eficiência e velocidade

na ordenação de dados. Sua eficiência se deve à estratégia de ?Divisão e Conquista?, que divide o vetor em partes menores, organiza cada parte individualmente e, em seguida, combina as segmentações ordenadas para obter o resultado.
Segundo Cormen et al. (2012), essa abordagem é especialmente eficiente em algoritmos de ordenação, pois permite reduzir o tempo de execução para uma complexidade de O(n log n), na média dos casos.
No entanto, para poder dividir, ele precisa encontrar um ponto para dividir, essa necessidade é resolvida após a escolha de um elemento na lista como um pivô, este número será o valor de referência usado para dividir a lista em duas partes, elementos maiores, e menores.

Figura 4 ? Trecho do algoritmo Quick Sort realizando o loop de partição do vetor
Fonte: De própria autoria
Assim como na figura acima, o algoritmo então particiona o vetor, fazendo com que os elementos menores que o pivô se vão para a esquerda e os maiores fiquem à sua direita, esse processo é feito recursivamente para os subvetores decorrentes. Este processo continua a ocorrer até que a lista possua no máximo um elemento, pois assim, ela já estaria ordenada.
A figura abaixo mostra o todo processo descrito anteriormente, dentro do sistema computacional.

Figura 5 ? Algoritmo de ordenação QuickSort

Fonte: De própria autoria
Já a complexidade de tempo média dele é representada pela fórmula:
O(n log n) (1)
Isto significa, que, ele realiza um número de comparações e movimentações que equivalem à fórmula (1), onde n é o tamanho da lista a ser ordenada. Portanto, este é um dos motivos que faz este algoritmo ser tão eficiente, o seu tempo relativo já é suficientemente bom.

E por último, ele não possui cálculo de armazenamento, porque o Quick Sort é um algoritmo ?in-place?, ou seja, não requer memória adicional para armazenar as estruturas temporárias durante a ordenação. Tornando-o extremamente eficiente em termos de uso de memória.

2.2 Heap Sort:

O algoritmo Heap Sort funciona como uma árvore binária especial, na qual o valor de cada pai é maior ou igual ao valor de seus filhos (max-heap) ou menor ou igual (min-heap). No contexto do heapsort, é usado o max-folhas (max-heap), ou seja, ele irá tentar buscar o maior valor e colocá-lo no topo da árvore, e sempre seguindo dessa maneira até conseguir ordenar corretamente os elementos guardados.

Primeiramente, o algoritmo começa a iniciar o processo de transformação do vetor em uma árvore, sempre buscando o maior número e colocando-o ao topo da árvore, o funcionamento é parecido com o de uma pirâmide, esta figura abaixo representa visualmente como é o funcionamento.

Após o maior número atingir o topo da árvore, ele é movido para a posição final da lista e removido da árvore, então ocorre uma reorganização entre todas as folhas para levar o maior número para o topo novamente, para isso que ocorra até todos os elementos serem levados para o vetor ao ponto de ele estar completo, com todos os seus dados de volta.

Em suma, o algoritmo Heap, cria uma árvore binária e realiza uma constante busca para levar todos os números ao topo para que assim consigam ser organizados do maior para o menor. Em relação à complexidade de tempo médio, o cálculo do algoritmo Heap Sort é o mesmo que do Quick Sort (1), portanto possui as mesmas qualidades dele, sendo tão rápido quanto ele.

Assim como o tempo, sua complexidade de espaço também é a mesma do Quick, os dois são considerados in-place, ele usa o mesmo vetor para armazenar os elementos e para o processo de ordenação, sendo extremamente eficiente em situações em que se há pouca memória no computador. A figura abaixo mostra o algoritmo do Heap Sort por completo.

Figura 7 ? Algoritmo de ordenação Heap Sort

Fonte: De própria autoria

2.3 Merge Sort:

O Merge Sort se assemelha, muito com o algoritmo Quick Sort, seja por conta da sua estratégia, mas também por conta de sua versatilidade, eficiência e velocidade em organizar um vetor desorganizado.

Em comparação, os dois utilizam a estratégia, divisão e conquista, porém, uma grande diferença acaba os diferenciando e mudando radicalmente, isto é, a aplicação dessa estratégia é diferente para cada um deles .

Enquanto o algoritmo Quick Sort, acaba necessitando de um pivô, o Merge não se prende a isso, pois ele divide a lista em parte iguais e em seguida, combina essas partes ordenadas, completando a ordenação por completo. Em suma, um deles encontra o meio e divide e o outro escolhe um pivô, tornando o que separa no meio, o melhor, principalmente nos piores casos, pois, o pivô escolhido pode acabar não sendo o do meio, dificultando bastante e aumentando a complexidade de tempo para a seguinte fórmula:

$O(n^2)(2)$

Por outro lado, o Merge Sort continua constante mesmo em situações em que existem muitos elementos a serem ordenados, também não dependendo da sorte, facilitando ainda mais a organização, a figura 9 mostra uma representação de como é realizada a separação:

Figura 8 - Algoritmo de ordenação Merge Sort

Fonte: De autoria própria

A figura acima mostra como é feita a divisão recursiva do Merge Sort, no programa usado neste trabalho. Como citado, a complexidade de tempo do algoritmo é sempre constante, em todos os casos, por conta de como ele aplica a estratégia, então ele mantém a fórmula (1) sempre.
No entanto, a complexidade de espaço, acaba sendo levemente diferente, pois ele possui um auxiliar que acaba consumindo memória, fazendo-o não se tornar in-place. Na figura 8, podemos ver a variável ?t? sendo usada como vetor auxiliar. Assim a fórmula de espaço acaba sendo a seguinte, onde n é o número de elementos:
O(n)(3)
Listagem dos Valores Antes e Depois da Ordenação:
A análise será feita lendo o mesmo arquivo de texto, um de dez dados duplicados, para ser possível diferenciar a forma que cada um ordena, explicando seus passos e métodos.
3.1 Quick Sort:
Figura 9 ? Reposta dada pelo algoritmo de ordenação Quick Sort

Fonte: De autoria própria
Na figura acima, é possível ver como funciona o processo de ordenação do algoritmo Quick Sort lendo o arquivo, no entanto, o código acaba tendo certos problemas em mostrar o elemento, como é possível ver na terceira linha de passos imprimida, o último elemento mostra um endereço de memória, por conta de uma das recursividades acabar utilizando o endereço, fazendo a impressão se tornar confusa.
Contudo, analisando algumas partes compreensíveis da imagem, pode-se concluir que os passos da ordenação prosseguem dividindo, utilizando o pivô escolhido que acaba mudando após um certo período de impressões, pois ele vai ordenando os subvetores restantes, utilizando recursividade e economizando na alocação de memória.
3.2 Heap Sort:
Figura 10 - Resposta dada pelo algoritmo de ordenação Heap Sort

Fonte: De própria autoria
Assim como explicado, o maior valor é sempre levado para o fim do vetor, na figura acima, é mais facilmente visível como é feita essa ação. À medida que se progride, é visível a sucessão do processo, com o algoritmo mostrando a movimentação dos maiores elementos para a parte mais à extremidade esquerda, então, após percorrer por todas as folhas, move-as para o fim do vetor.
Por conta de ele ser uma árvore, as impressões acabam sendo um pouco maior, porém, é visível, em comparação com o Quick que por sua vez, acaba mostrando os endereços de memória.
3.3 Merge Sort:

Figura 11 - Resposta dada pelo algoritmo de ordenação Merge Sort

Fonte: De autoria própria
É imprimido de maneira bem similar, como o Quick Sort, porém de maneira visível mostrando o número invés do endereço, isso ocorre, pois, a impressão está ocorrendo em um momento diferente do Quick, neste caso, ela está após a transferência do vetor temporário, o que faz com que não seja imprimida o endereço de memória.
Portanto, é mais perceptível como funciona o procedimento de divisão e conquista, vendo como o 958 é

lentamente passado para o meio, até aquele momento ele é considerado o maior número da primeira metade fazendo-o fica naquela posição e então cada metade vai se organizando.

Apresentação dos Resultados Comparativos de Performance:

Esta seção apresenta uma análise de resultados entre os algoritmos utilizando quatro tipos de cenários diferentes entre eles, podendo ser duplicados ou não. Cada um dos programas foi avaliado quanto ao tempo médio de execução, visando descobrir o melhor, dependendo em cada ocasião. Os testes foram feitos com arquivos de texto com 50 mil dados e os gerados tiveram 100 mil amostras.

4.1 Medição:

Para obter os resultados, foi necessário o uso da função clock() da biblioteca da linguagem de programação C, da qual pode medir, os milissegundos, com mais precisão, o momento exato que é executado o algoritmo de ordenação, sem contar o tempo de criação de variáveis, alocação de memória e obtenção ou geração de dados.

Desta maneira, foram obtidos os dados com o tempo médio em (ms) apresentados na seguinte tabela:

Tabela 1 ? Comparação do Tempo Médio de Execução dos Algoritmos com Dados Duplicados

Fonte: De autoria própria

Tabela 2 - Comparação do Tempo Médio de Execução dos Algoritmos sem Dados Duplicados

Fonte: De autoria própria

Com estas tabelas, é possível observar que, em situações de dados espalhados de forma aleatória, o algoritmo Heap Sort é um dos piores no quesito tempo, perdendo por poucos milissegundos.

No entanto, em situações em que os elementos são espalhados de forma côncava, caso os dados sejam duplicados, o Quick acaba perdendo, mas em situações sem duplicidade, o Heap acaba tendo uma desvantagem de 515 milissegundos, uma diferença que quando comparada em situações de mais dados, faz com que ele não seja tão eficiente.

Toda essa diferença entre os dois, ocorre devido a diferença da estratégia entre eles, enquanto um necessita fazer uma árvore e deve percorrê-la, o outro apenas se divide e compara até conseguir se ordenar.

Apesar do Quick conseguir ser mais veloz que o Heap nesses cenários, existem muitos casos em que a sua estratégia não funciona tão bem, por exemplo, com grandes quantidades de números, esse cenário faz com que o Quick se torne extremamente instável nessas ocasiões. Pois, a forma em que ele exerce seu plano, depende muito de o pivô estar mais perto do centro ou não, podendo facilitar e muito em arquivos de dados pequenos.

Algo que não ocorre com o Merge Sort, que por sua vez, acaba brilhando nessas situações já que a aplicação da estratégia Divisão e Conquista é diferente, fazendo uma divisão por 2, não precisando ser tão dependente de achar ou não um pivô mais próximo do centro do vetor, já que quando dividido, sempre cairá ao meio.

No entanto, o algoritmo Merge Sort acaba tendo uma grande desvantagem em comparação aos outros dois algoritmos, o armazenamento de espaço, o que torna ele estável na maior parte das situações, tende a deixá-lo lento, pois ocorre a transferência de números do vetor temporário ao original.

Ref Biblio:
CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: Teoria e Prática. 3. ed. Rio de Janeiro: Elsevier, 2012.
SEDGEWICK, Robert; WAYNE, Kevin. Algorithms. 4th ed. Boston: Addison-Wesley, 2011.
MANZANO, José Augusto N. G.; OLIVEIRA, Jayr Figueiredo de. Estruturas de Dados e Algoritmos. 3. ed. São Paulo: Érica, 2011.
SIPSER, Michael. Introduction to the Theory of Computation. 3rd ed. Boston: Cengage Learning, 2012.
KNUTH, Donald E. The Art of Computer Programming: Sorting and Searching. 2nd ed. Massachusetts: Addison-Wesley, 1998.
DEITEL, H.; DEITEL, P. C: Como Programar. 7. ed. São Paulo: Pearson, 2010.
Resultados e Discussão

Como visto previamente neste documento, cada tipo de método de ordenação tem características próprias que o tornam altamente eficiente em algumas ocasiões e altamente ineficiente em outras. No entanto torna-se necessário uma análise mais profunda destes dados para um melhor entendimento dos mesmos. Para tal feito, foram coletadas mais de 800 amostras visando um melhor entendimento de cada método de ordenação.
Todas as amostras foram coletadas tendo como princípio a meta de 400 dados para cada tipo de método de ordenação, separados em dois subgrupos de 200 dados, respectivamente com e sem duplicidade de dados. Cada um destes subgrupos foi dividido em 5 subgrupos de 40 dados cada (aleatório, côncavo, convexo, crescente, decrescente) e um subgrupo com 100 dados(gerado pelo nosso grupo). Cada dado era composto por 50.000 números(como demonstrado abaixo em nossa imagem de autoria própria).

Ao serem analisados foi possível se tirar a seguinte tabela (autoria própria) como conclusão:

Os números com código de cores ao lado na parte inferior dos gráficos acima representam o tempo (em segundos) que cada algoritmo demorou para ser executado, já os números ao lado da palavra ?Frequência? representam a frequência de cada um dos números com código de cores ao lado.
Os gráficos acima permitem concluir diversos fatores. Por exemplo, os métodos de ordenação heap e merge têm uma variação menor do que o quick sort em termos de análise de dados. No entanto, o método merge possui uma demora muito maior para processar os dados que o quick e o heap.
Com tais dados foi possível montar uma curva normal extremamente detalhada para cada tipo de algoritmo de ordenação, com e sem duplicidade de números, e para os números também localmente gerados a partir de métodos de ordenação. Abaixos estão respectivamente as curvas normais (autoria própria) dos dados listados acima :

Os grupos de curvas normais apresentados acima são respectivamente pertencentes aos métodos de ordenação hippie, quick e merge. Com os códigos de cores preto para ordenação com duplicidade, vermelho para ordenação se duplicidade e azul para os dados coletados a partir de dados gerados localmente.
É possível notar que em cada grupo é apresentado vários símbolos com significados diversos. O símbolo ? denota a média de todos os dados coletados. O símbolo ? descreve o desvio padrão do objeto em estudo.
O número indicado no balão é o mesmo que o do símbolo X. A porcentagem que se segue no meio da curva de sino traz qual a probabilidade de haver números abaixo de X o que, por inferência, nos traz uma taxa de confiança de 95% para x ? X. Portanto pode-se afirmar que nenhum número recairá abaixo de X a não ser por acaso, e que X deve ser tomado como nosso objeto de estudo atual ignorando-se todos os tempos acima do mesmo.
Nota-se também que o método de ordenação quick está com menos dados na parte de com e sem duplicidade. Isso se deve ao fato do erro Stack overflow ter ocorrido durante a execução de ambos os dados decrescente e crescente, então se foi executado apenas uma vez cada e posto o tempo de demora para o código sair.
Com isso em mente é possível concluir que os métodos de ordenação quick e heap, são realmente mais rápidos em algumas ocasiões. No entanto, os métodos que apresentam mais estabilidade durante a execução são os métodos heap e merge. Há também a probabilidade de o método quick dar o erro Stack overflow devido a quantidade de memória ram que ocupa durante o processo de execução. Por fim, é possível concluir que o método de ordenação mais estável e o mais rápido é o heap.
Concluimos que os algoritmos de ordenação Quick Sort, Merge Sort e Heap Sort, mesmo que apresentem complexidades de tempo semelhantes , demonstram comportamentos distintos em diferentes cenários. Algoritmos de ordenação estáveis, como o Merge Sort , são úteis em situações em que a preservação da

ordem relativa é importante, como ordenação de registros em bancos de dados ou classificação por múltiplos critérios. Por outro lado, algoritmos de ordenação instáveis, como o Quicksort, podem alterar a ordem de elementos com chaves iguais durante o processo de classificação.
O Quick Sort e Heap Sort tiveram um desempenho melhor que o Merge sort sem duplicidade e com duplicidade, onde o Quick Sort se sobrepõe ao Heap Sort tendo menos frequencia, com uma diferença de 25 no tem 0,006 sem duplicidade e com duplicidade um diferença de 42.
Portanto a escolha do algoritmmo de ordenação ideal depende de diversas situações, como o tamho do conjunto de dados, os requisitos da memória ea estabilidade. No geral o Quick Sort será uma excelente opção para a maioria dos casos devido ao seu desempenho .No entanto o Heap Sort pode ser uma boa opção para grandes conjutos de dados. O Merge Sort pode ser uma boa opção quando o espaço de memória for limitado.

AlgoritmoAleatório (seg.)Côncavo (seg.)Convexo (seg.)Gerado (seg.)
Quick Sort0.005550.006450.00540,00527
Heap Sort0.0077750.0058750.00580,01695
Merge Sort0.011150.008150.0082750,011

AlgoritmoAleatório (seg.)Côncavo (seg.)Convexo (seg.)
Quick Sort0.0056750.006450.005875
Heap Sort0.00750.01160.0065
Merge Sort0.01150.00850.0085