UNIVERSIDADE DE SÃO PAULO

ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES

BACHARELADO EM SISTEMAS DE INFORMAÇÃO

ANÁLISE DE ALGORITMOS DE ORDENAÇÃO E BUSCA

SÃO PAULO

2014

ADRIANO AUGUSTO SATTI ORTEGA BOSCHI
CAIO VINICIUS MARQUES TEIXEIRA
FELIPE GRECOV SANTANA
JOÃO PEDRO NARDARI DOS SANTOS

ANÁLISE DE ALGORITMOS DE ORDENAÇÃO E BUSCA

SÃO PAULO
2014

Introdução

Este relatório visa apresentar o conteúdo dos algoritmos de busca (Sequencial, Binária e por Tabela de Espalhamento) e ordenação (Selection Sort e Heap Sort). Sendo este conteúdo: Uma breve descrição do seus funcionamentos, a análise dos algoritmos e as apresentações de experimentos realizados pelo grupo com entradas de dados de n, 2n e 4n com determinado algoritmo.

Algoritmos de Busca

Busca Sequencial

Seleciona-se uma palavra da lista de palavras a serem buscadas. Após obtida, ela é comparada com a lista de palavras disponíveis, dando um aviso se foi encontrada ou não. Feito essa verificação, troca-se a palavra para a próxima da lista usando um laço.

Análise

A análise assintótica demonstra que o algoritmo de Busca Sequencial é, no pior caso, O(n) . Isso devido ao fato que ele terá que percorrer a lista até achar a palavra, gerando resultados maiores quanto mais longe o elemento estiver na lista. Caso não esteja nela, percorre-se todas as palavras, sendo esse o pior caso.

Busca Binária

Da mesma maneira que a sequencial, seleciona-se uma palavra a ser buscada. Contudo, cria-se a posição inicial, mediana e final. Depois é verificado se a palavra é “maior” ou “menor” que a palavra posicionada no meio. Caso ela seja a palavra, avisa-se que foi encontrada e troca-se a palavra. Se não for, troca-se a posição inicial dependendo se for maior ou menor. Isso se repete até que a palavra seja encontrada ou a posição inicial seja maior que final.

Análise

Na busca binária, a análise assintótica demonstra que, no pior caso e no caso médio, o algoritmo possui fórmula O(log n). Isso demonstra que o algoritmo possui meio que um padrão pro pior e melhor caso. O melhor caso acontece quando a palavra buscada encontra-se no meio da lista.

Busca por tabela de espalhamento (Hash 1 e Hash 2)

Escolhendo uma das formulas de hash, hash1 fornecida no EP e hash2 criada pelo grupo. O algoritmo faz a leitura do arquivo de palavras e realiza a inserção das palavras na hashTable (um vetor de listas de strings com n/10 slots) conforme o calculo hash escolhido. Após isso seleciona-se uma palavra ser buscada, calcula-se o hash (conforme fórmula escolhida) e caso tenha mais de uma palavra no slot calculado realiza-se a busca sequencial até encontrar a palavra ou varrer toda linha.

Justificativa da escolha da função hash2

Conforme observado através de analise e de experimentos com a função fornecida junto ao enunciado do exercício programa 2 (função hash1), foi constatado que por utilizar-se de apenas 3 caracteres iniciais da palavra e seus respectivos números da tabela ASCII para calcular qual slot seria inserida determinada palavra aconteciam muitos conflitos em uma massa de palavras grande (no caso foi utilizado uma massa de palavras n = 80000 palavras). Então como ideia inicial da função hash2 aumentamos o número de caracteres escolhidos para 5 caracteres **((p[0] * 2^16 + p[1] * 2^12 + p[2] * 2^8 + p[3] * 2^4 + p[4]) % n/10)**, realizamos novamente os experimentos e comparamos os resultados obtidos com a fórmula hash1 com os resultados utilizando-se a fórmula criada pelo grupo. Como resultado esperado, o numero de conflitos por linha foi menor quando utilizada a função hash2, logo houve um maior espalhamento das palavras no vetor de n/10 slots, tornando a função hash2 melhor que a hash1 até em questão do tempo de busca das palavras. Segue resultado com numero de comparações e tempo para busca do teste realizado com n = 80000:

Palavra	Numero de Comparações		Tempo de Execução	
	Hash 1	Hash 2	Hash 1	Hash 2
tangenciaremo	1	1	0.094746	0.088035
corporifiques	331	24	0.251077	0.024476
cronometrasse	41	28	0.038293	0.028029
emparelhasseis	653	81	0.545973	0.063559
trago	916	2	0.438988	0.003158
tragoeis	917	3	0.195808	0.003553

Análise:

No algoritmo de hash table com n/10, independente da fórmula, conforme a análise aponta, no melhor caso (em que é calculado o hash e não há conflito) a complexidade fica como O(1) já que já será retornada a palavra buscada e realizada apenas uma comparação. O caso médio seria um slot onde dentre os n conflitos a palavra buscada (sequencialmente) seria a do meio, logo uma complexidade de O(n/2) e no pior caso a complexidade O(n), visto que seria necessário comparar todos os conflitos existentes (com busca sequencial) no slot (calculado pela fórmula hash) para encontrar ou não a palavra que foi selecionada para busca. Visto que a fórmula hash 2 espalhou melhor as palavras do que a fórmula hash1, é constatado que a mesma é mais eficiente na realização das buscas para uma massa de dados grande.

Algoritmos de Ordenação

Selection Sort

O Selection Sort é um algoritmo de ordenação que consiste em encontrar o menor valor por pesquisa sequencial. Quando o menor valor é encontrado, este é trocado de posição com o primeiro valor do arranjo. O processo é repetido para o segundo menor valor, que é encontrado e trocado de posição com o segundo e assim sucessivamente. Logo, o algoritmo basicamente divide o arranjo em duas partes, uma parte ordenada e outra desordenada, quando encontra um menor elemento na parte desordenada o move para a parte ordenada, até que não existam mais elementos desordenados.

Exemplo (passo a passo):

5 – 3 – 1 – 4 – 2	Arranjo inicial
1 – 3 – 5 – 4 – 2	Procura o menor e troca com a primeira posição.
1 – 2 – 5 – 4 – 3	Procura o segundo menor e troca com o segundo valor.
...	Segue sucessivamente.
1 – 2 – 3 – 4 – 5	Estado final do vetor.

Análise

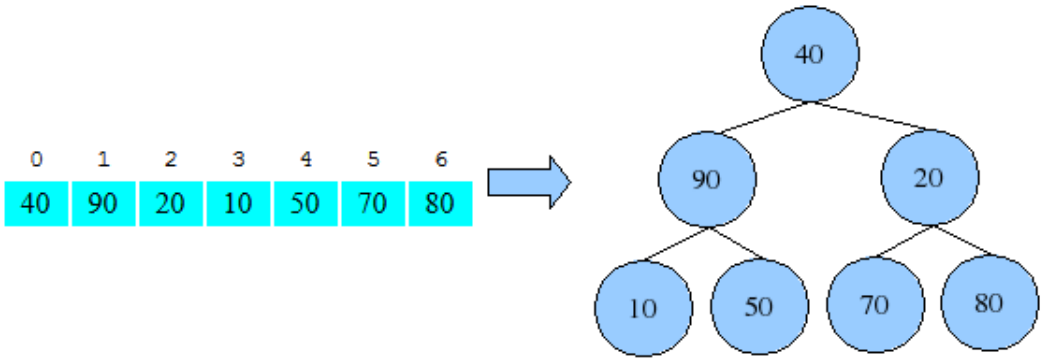
Para analisar a complexidade assintótica do Selection Sort, devemos contar o número de comparações e trocas que o algoritmo faz no pior caso. Isso se dá pela fórmula de recorrência abaixo:

$$T(n) = \ 0, se\ n = 1; T(n-1) + n - 1\ caso\ contrário.$$

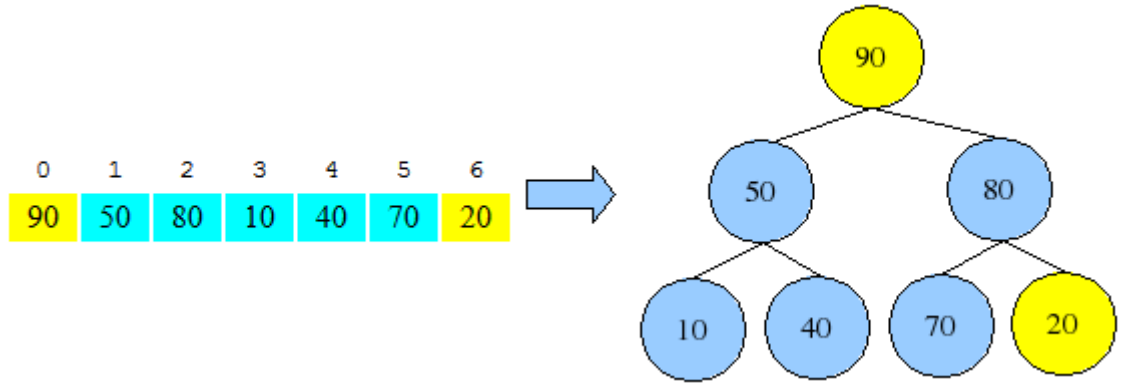
Portanto, $\theta(n^2)$ comparações são executadas no pior caso.

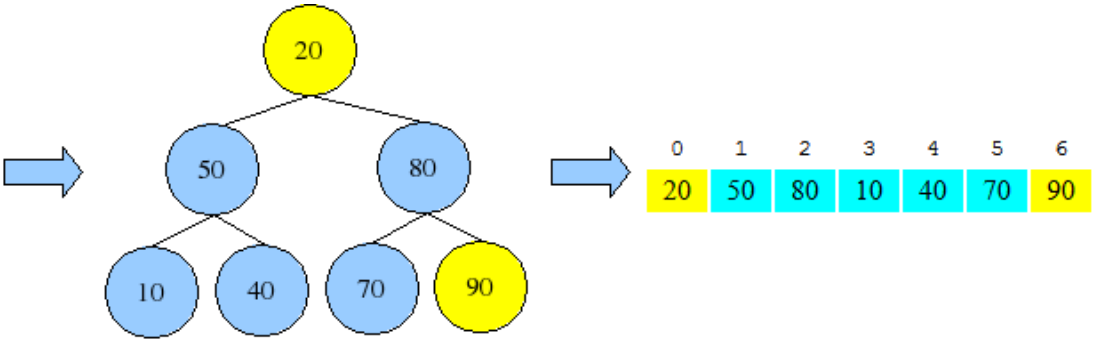
Heap Sort

O algoritmo de ordenação heapsort funciona com uma estrutura de dados chamada heap binário. Um heap binário é uma arvore binária construída na forma de um vetor.



O heap é gerado e mantido no próprio vetor a ser ordenado. Para uma ordenação crescente, o heap deve se tornar mínimo, ou seja, o menor elemento está na raiz da arvore (e de suas subarvores). Logo, a ideia é procurar o menor elemento entre as subarvores filhas e trocar a posição do menor elemento com a raiz e assim sucessivamente.





Análise

Para analisar a complexidade assintótica do Heap Sort, devemos notar alguns pontos: O método ConstroiHeap() possui custo de $O(n)$, O método heapfica() possui custo de $\log(n)$ e é chamado$(n-1)$ vezes. Logo o custo total do HeapSort é definido por:

$$T(n) = O(n) + (n-1)O(\log n)$$

Portanto o custo do HeapSort é $O(n \log n)$.

Experimentos

Nesta sessão, apresentamos os resultados com os experimentos com os algoritmos de ordenação apresentados anteriormente. Em ambos os experimentos, foram utilizadas listas de palavras extraídas de um dicionário online, estas listas possuem determinadas quantidades de palavras aleatórias organizadas de forma complemente randômica.

Algoritmos de Busca

Utilizando listas de tamanhos 10.000, 20.000 e 40.000. Foram feitas 5 buscas utilizando cada tamanho e cada algoritmos.
Abaixo há um formulário que contêm as comparações de cada algoritmo em cada lista mostrando seus Números de Comparações e seu Tempo de Execução no caso de mínimo, caso médio e máximo:

10.000 palavras	C	Min T.E	C	Med T.E	C	Max T.E
Busca Sequencial	10	0.010263	5191	0.607762	10000	1.364564
Busca Binária	12	0.006693	13	0.004908	13	0.006693
Hash 1	1	0.017402	7	0.006248	12	0.008925
Hash 2	1	0.075412	3	0.006693	26	0.052655
20.000 palavras	**C**	**Min T.E**	**C**	**Med T.E**	**C**	**Max T.E**
Busca Sequencial	23	0.010263	10120	1.798296	20000	3.664866
Busca Binária	13	0.003123	14	0.002677	15	0.004016
Hash 1	6	0.049978	12	0.011155	21	0.012941
Hash 2	1	0.049977	13	0.009817	52	0.030344
40.000 palavras	**C**	**Min T.E**	**C**	**Med T.E**	**C**	**Max T.E**
Busca Sequencial	411	0.069165	25158	3.744294	40000	2.342694
Busca Binária	14	0.003123	15	0.004908	16	0.003123
Hash 1	3	0.049085	33	0.017403	132	0.070504
Hash 2	1	0.049977	13	0.009817	52	0.030344

Sendo “C” o Número de Comparações e T.E o Tempo de Execução.

Algoritmos de Ordenação

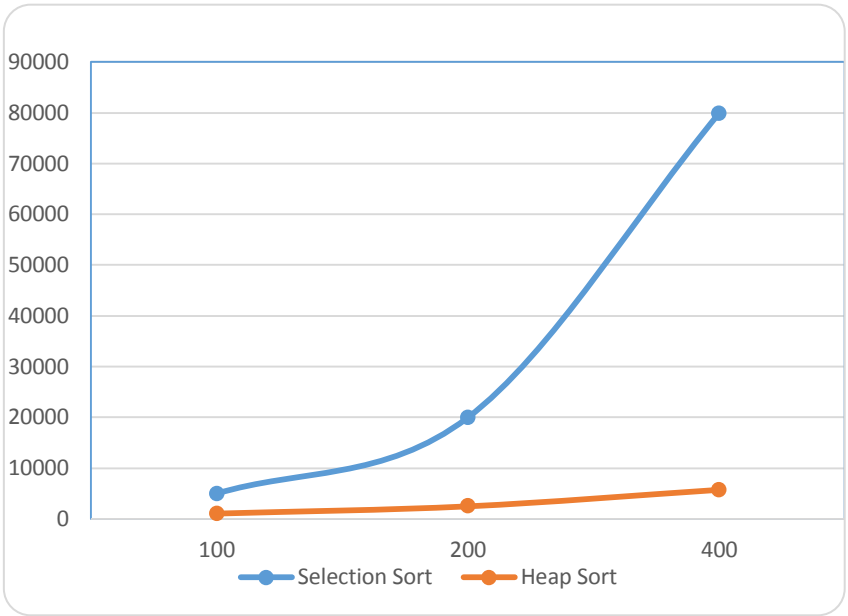
Número de comparações

O objetivo deste experimento é testar o crescimento do número de comparações de cada algoritmo de acordo com o tamanho da entrada. Para este experimento, utilizamos entradas com 100, 200 e 400 palavras desordenadas. Foram utilizados ambos os algoritmos para ordenar cada conjunto de palavras.

A tabela abaixo exibe a quantidade de comparações que cada algoritmo fez para ordenar a lista de palavras. O gráfico nos permite observar o crescimento do número de comparações, de acordo com o tamanho da lista de palavras.

	Selection Sort	Heap Sort
100	4950	1034
200	19990	2458
400	79800	5714

Como resultado, podemos notar que o crescimento do número de comparações de acordo com o tamanho da entrada para o algoritmo Selection Sort



realmente se comporta de forma quadrática. Assim, podemos concluir que o

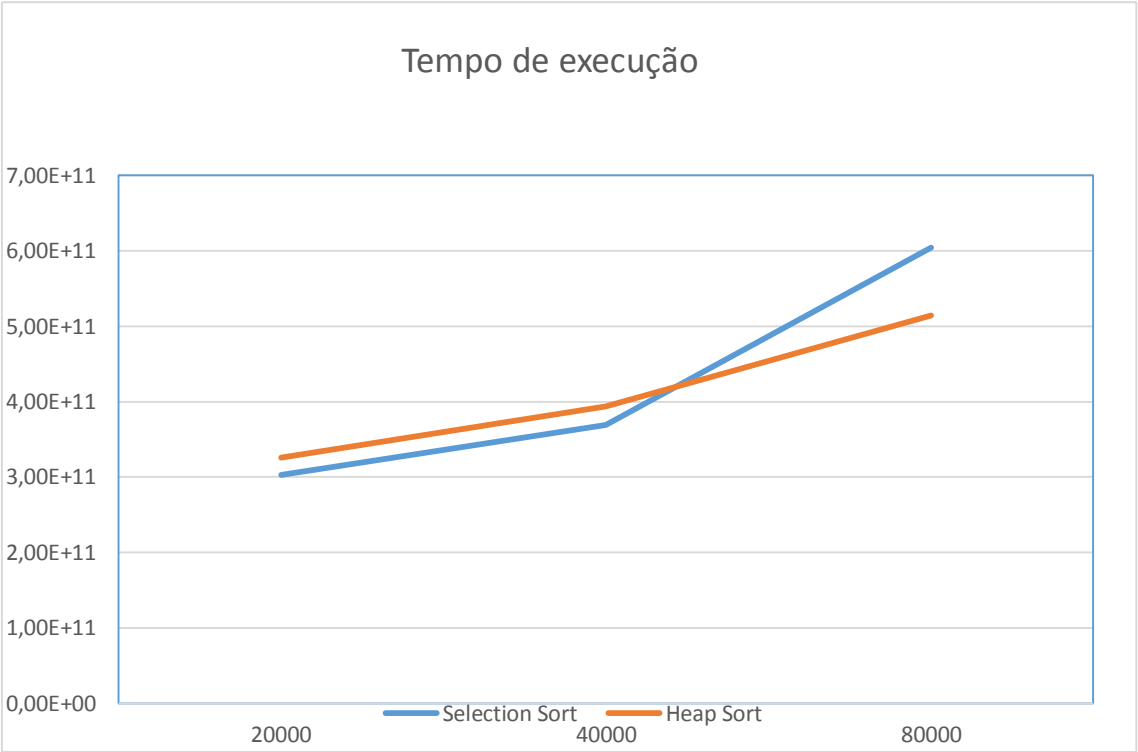
algoritmo Heap Sort é mais eficiente, ordenando com um número menor de comparações.

Tempo de Execução

O objetivo deste experimento é comparar o tempo de execução de ambos os algoritmos. Os testes foram realizados contando a quantidade de milissegundos que o algoritmo levou para ordenar a lista de palavras, vale salientar, que os resultados podem variar de acordo com o computador, o estado da JVM, memória, etc. Neste caso, utilizamos um computador com processador Core i5 2450M rodando o sistema operacional Ubuntu 13.04.

Foram utilizadas listas de 20 mil, 40 mil e 80 mil palavras para realizar esse teste. A razão para números tão grandes é que notamos que apesar de o Heap Sort ser um algoritmo mais eficiente, a inicialização de variáveis pela JVM e outros processos do sistema operacional o fazem levar mais tempo para executar, logo decidimos escolher listas que nos permitam ver quando seu tempo de execução se torna efetivamente menor que o do Selection Sort. Vale salientar que isto não significa que o algoritmo seja pior para listas menores, esta redução de desempenho está na casa dos nano segundos e pode não acontecer de acordo com a plataforma que o algoritmo está sendo executado.

O gráfico e a tabela abaixo mostram o tempo (em milissegundos) que cada algoritmo levou para ordenar cada lista de palavras.



	Selection Sort	Heap Sort
20000	302932640280 ms	325573488836 ms
40000	369058664084 ms	394032614795 ms
80000	604099013414 ms	514600368370 ms

Podemos observar com esses resultados, que o algoritmo HeapSort se torna mais rápido que o Selection Sort em listas grandes (nesse caso, acima de 40 mil elementos). Apesar disso, a diferença entre ambos em listas pequenas é ínfima (na casa dos nano segundos), portanto é mais interessante usar o Heap Sort que se comporta bem tanto no pior caso, como no melhor caso.

Conclusão

As considerações finais obtidas pelo grupo são que, após implementarmos e analisarmos os algoritmos de busca e de ordenação. Realizar a análise de algoritmos realmente nos orienta sobre como implementar um algoritmo eficiente. Observar como o algoritmo se comporta ao se procurar as palavras em uma lista, realmente expõe as principais diferenças entre eles, isto tornou-se notório com o calculo de comparações e tempo de execução sendo uma aplicação teórico-prática das aulas de introdução à análise de algoritmos, mostrando através dos experimentos realizados em qual situação um possui vantagem sobre o outro em termos de eficiência.