

Escola de Artes, Ciências e Humanidades - EACH-USP

Introdução à Análise de Algoritmos

Exercício de Programação 1

Kennedy Rohab Menezes da Silva, nº USP: 12683395, Turma 04

19 de novembro de 2021

1 Introdução

1.1 Algoritmos

Antes de embarcarmos neste relatório e demonstrar a eficiência dos algoritmos aqui propostos é válido entendermos a importância do porque existem algoritmos e, também, porque da existência de trabalhos como este. Os algoritmos, como parte da computação, servem para resolver problemas em um tempo satisfatório, pois uma resolução que leve mais que o tempo de uma vida não pode ser tratado como um algoritmo eficiente; ao mesmo tempo pretende-se que seja correto, pois de nada adianta um algoritmo rápido que não entregue o desejado, mas nem só de algoritmos rápidos é feita a computação (apesar de que correto, sempre), assim esse relatório tem essa tarefa: mostrar, com uma saída correta, o algoritmo mais eficiente para um determinado problema, levando em conta a *hardware* disponível.

1.2 Problema de Seleção

O presente trabalho pretende apresentar duas soluções para um problema, que é a busca por um *i-ésimo* número numa lista de números e suas respectivas soluções com dois algoritmos de seleção em que lhes serão passados uma sequência de números e o índice de um dos elementos a ser retornado desta sequência. As soluções são:

Selecao1(A, i)

- 1: *Ordene(A)*
- 2: **return** a_i

Selecao2(A, i)

- 1: $q \leftarrow \text{Particao}(A)$
- 2: **if** $n = 1$ **return** a_i
- 3: **if** $i < q$ **return** *Selecao2*($A[1 : q - 1], i$)
- 4: **else if** $i > q$ **return** *Selecao2*($A[q + 1 : n], i - (q + 1)$)
- 5: **else return** a_q

1.3 Exemplos de aplicação

Essas soluções podem ter algumas aplicações úteis na área da Estatística em que encontrar um determinado valor pode ser útil, como por exemplo: **1)** encontrar a mediana da idade de pessoas de um determinado grupo (mediana da idade de pessoas que possuem casa própria) e **2)** até mesmo por qualquer quartil que se queira trabalhar, contanto que o *i-ésimo* número buscado corresponda a esses *quartis*. Vale notar que algoritmos são pensados para ser eficientes dado uma grande quantidade de dados, pois é assim que pode-se testar sua eficiência. Por isso os exemplos devem considerar essa abordagem quantitativa.

1.4 Esboço da correção

Como esboço da correção elucidaremos a eficiência, *a priori*, de cada algoritmo, reservando para mais tarde nesse relatório a demonstração empírica da eficiência deles.

1.4.1 Esboço da *Selecao1*

Para o algoritmo *Selecao1*(A, i) temos que se o tamanho da lista que passamos para o algoritmo for maior que 1 então esse algoritmo será ordenado uma vez que seu tamanho é reduzido à metade a cada chamada recursiva do algoritmo *mergeSort*, por fim quando o tamanho dessas listas se tornarem iguais a 1, a próxima etapa (*merge*) juntará essas listas em ordem. Assim, após passar por todos os elementos da lista, retornará ela ordenada e o *Selecao1*(A, i) devolverá o *i-ésimo* número passado como parâmetro. Podemos notar que esse algoritmo **passará** por todos os elementos a fim de ordená-los.

1.4.2 Esboço da *Selecao2*

Vejam agora o algoritmo *Selecao2*(A, i). Esse algoritmo recebe uma lista e a partir de *Partition*($A, size$) ele escolhe um pivô e vai trocando a posição dos elementos (sem passar por todos eles), afim de que no final, todos os elementos maiores que esse pivô estarão de uma lado da lista, e os menores que ele do outro lado. Assim quando esse pivô for igual ao *i-ésimo* número passado como parâmetro, o algoritmo terminará e retornará esse valor. Note que o algoritmo **não precisou** passar por toda a extensão da lista e nem ordená-la por completo, o que torna esse algoritmo mais eficiente.

1.5 Tempo de Execução

1.5.1 Tempo de Execução - Algoritmo *Selecao1*

O algoritmo *Selecao1*(A, i) tem tempo de execução $\Theta(n \log(n))$ [1], onde n é o tamanho da entrada, podemos escrever como $T(n) = \Theta(n \log(n))$, vejamos o porque:

- $\Theta(1)$ se $n = 1$;
- $2T(\frac{n}{2})$ - As duas divisões recursivas do arranjo pela metade do tamanho;
- Para a etapa de *merge* temos que: $\Theta(n)$, já que percorrerá toda a lista;
- Logo, $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ - Estamos assumindo aqui que a lista é um múltiplo de 2.

Porém pelo teorema mestre podemos escrever da seguinte maneira, se $\Theta(1) = c$ e considerano toda a sub-árvore que decorre do algoritmo a parte de recursão temos que essa árvore irá se dividir $c(\frac{n}{2^i})$ em que i é a quantidade de vezes de subdivisão da árvore. Assim o *i-ésimo* nível abaixo da árvore tem custo $2^i c(\frac{n}{2^i}) = cn$. O total de níveis será $\lg n + 1$. Assim, para calcular o custo total, somamos os custos de todos os níveis. Com a árvore de recursão $\lg n + 1$ e cada nível com custo cn temos:

- $cn(\lg n + 1) = cn \lg n + cn$, ignorando termo de ordem baixa e constante c , obtemos:
- $\Theta(n \log(n))$.

1.5.2 Tempo de Execução - Algoritmo *Selecao2*

Para analisarmos o tempo de execução do *Selecao2* elucidaremos o pior e melhor caso.

- **O melhor caso** acontece quando o algoritmo executa a menor quantidade de vezes possíveis. Isso ocorre quando o pivô é igual ao elemento que buscamos, ou seja, o último elemento da lista. Assim o tempo que terá destaque é o $\Theta(n)$ parte do algoritmo da partição que será executado uma vez. A comparação $q = i$ toma tempo constante.
- **O pior caso** acontece quando o algoritmo executa a maior quantidade de vezes possíveis. Isso ocorre quando o elemento que estamos buscando é encontrado só na última execução do algoritmo, isso significa que escolhemos um pivô (na nossa implementação sendo o último), mas o número que queremos está na outra extremidade da lista, ou seja, o primeiro - aqui temos que considerar

que o arranjo estará ordenado de forma crescente, o que é extremamente raro de acontecer dado que a entrada, pelo menos para esse relatório, está ocorrendo de forma aleatória. Assim o algoritmo escolherá sempre o elemento maior como pivô e deverá passar um a um até chegar no menor elemento da lista. Desse modo o algoritmo no pior caso é $\Theta(n^2)$, pois $T(n) = n + (n-1) + (n-2) + \dots + 1 \rightarrow n(n+1)/2 = \Theta(n^2)$

2 Objetivo

2.1 Configuração de testes

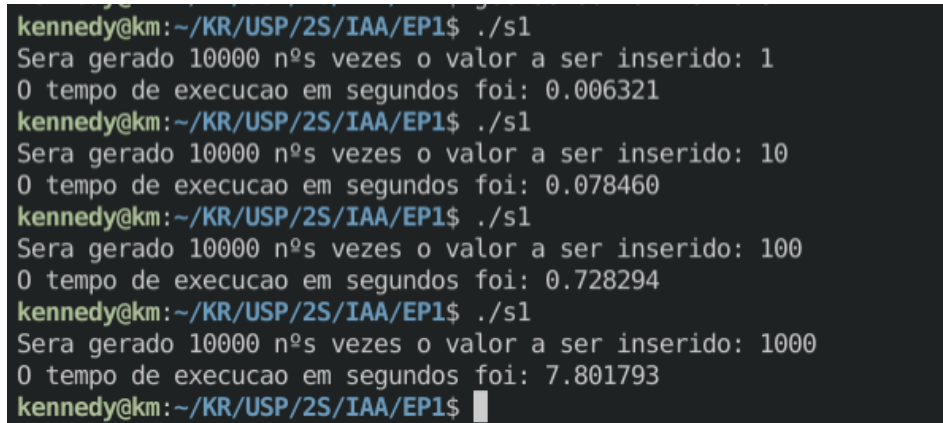
Este relatório tem como objetivo analisar e comparar os algoritmos já citados. Como os programas recebem entradas de tamanhos diferentes cabe, por meio de análise do tempo de processamento, verificar qual deles possui uma eficácia melhor. Sendo assim, comparemos o tempo de processamento de cada um deles.

Para isso utilizaremos entradas de tamanhos 1.10^4 , 1.10^5 , 1.10^6 , 1.10^7 , 1.10^8 e 1.10^9 , ou seja, entradas que variam de dez mil a um bilhão.

O número a ser buscado será de maneira aleatória, ou seja, um *i-ésimo* número que pode ser de 1, ... , a_i , em que i é o tamanho da sequência.

2.2 Comparação do tempo de processamento

Os valores obtidos levando em conta a seção 2.1 são:



```
kennedy@km:~/KR/USP/2S/IAA/EP1$ ./s1
Sera gerado 10000 n°s vezes o valor a ser inserido: 1
O tempo de execucao em segundos foi: 0.006321
kennedy@km:~/KR/USP/2S/IAA/EP1$ ./s1
Sera gerado 10000 n°s vezes o valor a ser inserido: 10
O tempo de execucao em segundos foi: 0.078460
kennedy@km:~/KR/USP/2S/IAA/EP1$ ./s1
Sera gerado 10000 n°s vezes o valor a ser inserido: 100
O tempo de execucao em segundos foi: 0.728294
kennedy@km:~/KR/USP/2S/IAA/EP1$ ./s1
Sera gerado 10000 n°s vezes o valor a ser inserido: 1000
O tempo de execucao em segundos foi: 7.801793
kennedy@km:~/KR/USP/2S/IAA/EP1$
```

Figura 1: Teste para *Selecao1* com os número de entrada 1.10^4 - 1.10^7 .

Vale notar (Na Figura 1) que essa foi a segunda execução do código do algoritmo e a imagem não contempla uma lista maior que 1.10^7 , pois para valores a partir de 1.10^8 a máquina usada para teste trava.

Já para *Selecao2* (Figura 2) a máquina não conseguiu chegar aos valores de 1.10^9 . Uma curiosidade também é que para os três testes de tamanho 1.10^8 um deles pode ser visto como um *outsider*, pois enquanto os outros dois ficaram por volta de 75/80 segundos aquele ficou próximo de 10 segundos. O que pode significar menos posições da lista não-afetadas pelo algoritmo e o retorno de *i-ésimo* número mais rápido. Vale a suposição também de que o elemento buscado estava próximo do pivô.

```

kennedy@km:~/KR/USP/2S/IAA/EP1$ ./s2
Sera gerado 10000 n°s vezes o valor a ser inserido: 1
0 tempo de execucao em segundos foi: 0.000394
kennedy@km:~/KR/USP/2S/IAA/EP1$ ./s2
Sera gerado 10000 n°s vezes o valor a ser inserido: 10
0 tempo de execucao em segundos foi: 0.008542
kennedy@km:~/KR/USP/2S/IAA/EP1$ ./s2
Sera gerado 10000 n°s vezes o valor a ser inserido: 100
0 tempo de execucao em segundos foi: 0.070761
kennedy@km:~/KR/USP/2S/IAA/EP1$ ./s2
Sera gerado 10000 n°s vezes o valor a ser inserido: 1000
0 tempo de execucao em segundos foi: 1.408750
kennedy@km:~/KR/USP/2S/IAA/EP1$ ./s2
Sera gerado 10000 n°s vezes o valor a ser inserido: 10000
0 tempo de execucao em segundos foi: 76.294101
kennedy@km:~/KR/USP/2S/IAA/EP1$ ./s2
Sera gerado 10000 n°s vezes o valor a ser inserido: 10000
0 tempo de execucao em segundos foi: 9.741274
kennedy@km:~/KR/USP/2S/IAA/EP1$ ./s2
Sera gerado 10000 n°s vezes o valor a ser inserido: 10000
0 tempo de execucao em segundos foi: 80.010291
kennedy@km:~/KR/USP/2S/IAA/EP1$ ./s2
Sera gerado 10000 n°s vezes o valor a ser inserido: 100000
Segmentation fault (core dumped)

```

Figura 2: Teste para *Selecao2* com os número de entrada 1.10^4 - 1.10^7 .

3 Resultados

3.1 Tempo de processamento dos algoritmos

Entrada/Algoritmo	<i>Selecao1</i>	<i>Selecao2</i>
1.10^4	0.006321	0.000394
1.10^5	0.078460	0.008542
1.10^6	0.728294	0.070761
1.10^7	7.801793	1.408750
1.10^8	-	80.010291
1.10^9	-	Seg. Fault

Tabela 1: Entrada e tempo de execução de cada algoritmo em segundos.

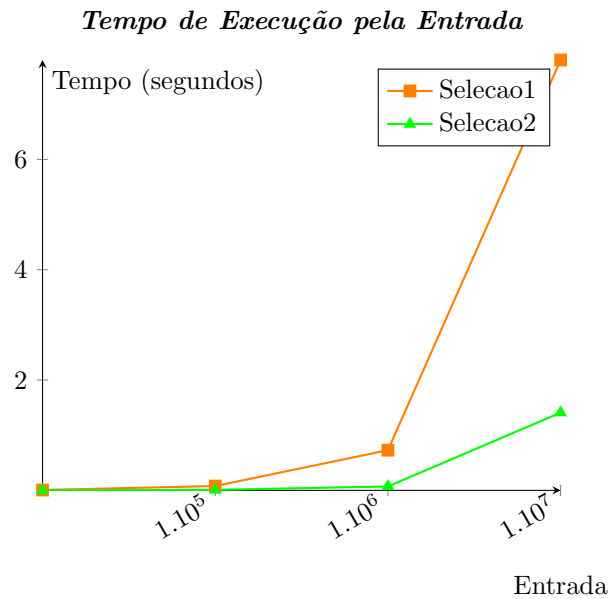


Figura 3: Gráfico da entrada e tempo de execução de cada algoritmo em segundos

Fica claro, então, que dado a máquina onde foi feito os testes não é possível trabalhar com entradas da ordem de 10^9 .

4 Conclusão

Portanto fica claro e evidente (Pela Figura 3¹ e pela Tabela 1) que o algoritmo Selecao2 é mais eficiente, ele consegue trabalhar com valores de ordens mais elevadas superando bastante o algoritmo Selecao1. É claro que como pode-se verificar, a máquina cujo o programa for ser compilado e executado precisa concomitantemente ser eficiente para que o algoritmo possa trabalhar com números extensos e superiores aos que trabalhamos aqui.

Referências

[1] T. H. Cormen, “Algoritmos, teoria e prática,” p. 40, 2009.

Os códigos usados para este relatório encontram-se zipados, entregues junto a este PDF, caso haja algum imprevisto eles estarão disponíveis neste repositório: [sourcecode](#)

¹Omitiu-se da Figura 3 a ordem de 10^8 para que a escala ficasse mais apropriada e a leitura pudesse se tornar eficiente.