Análise de algoritmos para a solução do Problema da Seleção

> Por Guilherme de Abreu Barreto[1]

Resumo

É o objetivo deste relatório definir o *Problema da Seleção* e apresentar dois algoritmos passíveis de solucioná-lo. Compara-se as estratégias adotadas em cada algoritmo, assim como o tempo de execução previsto e experimental de cada qual. Com base nessas observações infere-se a eficiência relativa dos algoritmos.

Introdução

O *Problema da Seleção* no contexto desta análise refere-se a necessidade de, para uma sequência de elementos $x_a, \dots, x_i, \dots, x_b$ onde $a \leq i \leq b$, sendo a e b sendo os índices inicial e final respectivamente, acessar o i-esimo elemento x_i de acordo com um dado parâmetro. Trata-se de um problema comumente observado nas situações aquelas em que se busca obter medidas de posição tais quais a mediana ou os quartis, para citar uma aplicação no âmbito da estatística. Para demonstrar a resolução deste, iremos aqui admitir que $\forall x \in \mathbb{Z}$ e utilizaremos como critério o valor de x de maneira a selecionar o i-esimo elemento de menor valor. Vamos abordar este problema de duas formas:

Na primeira solução (solução `mergeSelect`) ordenaremos o conjunto de elementos em ordem crescente para, em seguida, acessar a_i diretamente.

Na segunda solução (solução `quickSelect`) continuamente particionaremos a sequência à partir de comparações com o valor de um elemento desta, o pivô. Isso até que, ou o pivô ao final do processo de partição corresponda ao índice i, ou o conjunto particionado avaliado tenha tamanho 1 (portanto, contendo somente o elemento de índice i).

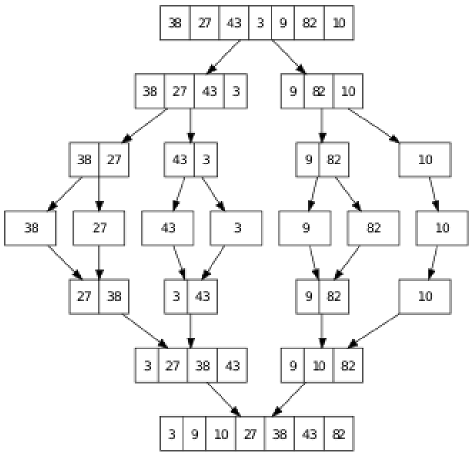
Solução *mergeSelect*

Definição

O algoritmo aqui implementado para a ordenação do arranjo de valores inteiros é o *Merge Sort*, que dá nome a solução. Trata-se de um algoritmo de ordenação eficiente e estável pautado pela comparação de valores e baseado no paradigma da *Divisão e Conquista*.

Conceitualmente, seu funcionamento se dá da seguinte maneira:

1. Divide-se o arranjo em n sub-arranjos de tamanho 1, sendo n o tamanho do conjunto original a ser ordenado, pois uma lista de um único elemento trivialmente já se encontra ordenada;
2. continuamente integra-se e ordena-se os sub-arranjos pré-ordenados para a par; até que
3. resta apenas a lista original ordenada. A partir de então, para encontrar o i-ésimo menor elemento, basta referenciar o arranjo pelo índice i: `A[i]`.



> Diagrama ilustrando os passos para execução do *Merge Sort* em uma sequência de 7 números inteiros.[2]

Implementação

Minha implementação deste algoritmo para o presente problema é dado pelas seguintes funções:

```
#include <stdlib.h>
#define array int*

void merge (array A, int pivot, int size) {
    int i, k, j = pivot, tmp[size];
    array tmp = malloc(size * sizeof(int));

    for (i = k = 0; k < size; k++)
        tmp[k] = ((A[i] <= A[j] && i < pivot) || j == size) ?
        A[i++] : A[j++];
    for (k = 0; k < size; k++)
        A[k] = tmp[k];
    free(tmp);
}

```

```
16 | void mergeSort (array A, int size) {
17 |     int pivot;
18 |
19 |     if (size <= 1)
20 |         return;
21 |     pivot = size / 2;
22 |     mergeSort(A, pivot);
23 |     mergeSort(A + pivot, size - pivot);
24 |     merge(A, pivot, size);
25 | }
```

Correção

Podemos aferir que o algoritmo acima é correto (isto é, adequado a produzir a solução) pois

1. Nas linhas 22 e 23 subdivide-se a sequência em pares recursivamente, procedimento este que se repetirá até que restem apenas subconjuntos de 1 elemento e a função retorne na linha 20.

2. Cada recursão anterior então acessa a função `merge` na linha 24. Esta integra pares de conjuntos adjacentes ao índice `pivot` nas linhas 8 à 10, percorrendo-os cada qual desde seus respectivos índices iniciais `i` e `j` e posicionando o elemento de menor valor no índice `k`. Isso, seguro de que estes já estarão ordenados em ordem crescente dada a condição anterior. Este então retornará os elementos de ambos os conjuntos em ordem crescente na linha 11 e 12.

3. O procedimento anterior se repete $(n - 1)$ vezes até que todos os pares ordenados combinados produzem a sequência original ordenada.

Assim, o algoritmo corresponde a definição dada anteriormente.

Teste empírico

A correção do algoritmo foi testada manualmente para pequenos arranjos utilizando o programa `manual_test.c`[4].

Tempo de Execução

É possível afirmar que o tempo de execução $T(n)$ do *Merge Sort* para quaisquer entradas de mesmo tamanho n é assintoticamente equivalente pois independentemente dos valores contidos na entrada, o mesmo número de operações de comparação (na linha 9) e atribuição (na linha 12) são executadas. Isto é, diferentemente do que é visto noutros algoritmos de ordenação tais quais o *Quick Sort* ou o *Insertion Sort*. Para estimar este tempo de execução, podemos recorrer ao *Teorema Mestre para recorrências de Divisão e Conquista*, por este ser um algoritmo do tipo *Divisão e Conquista*. Segundo Márcio Ribeiro (2021) o teorema mestre pode ser descrito nos seguintes termos:

> Sejam $a \geq 1, b > 1$ e $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, então:
>
> 1. se $f(n) \in O\left(n^{\log_b (a-\varepsilon)}\right)$ para algum $\varepsilon > 0$ então $T(n) \in \Theta(n^{\log_b a})$;
> 2. se $f(n) \in \Theta(n^{log_b a})$ então $T(n) \in \Theta(n^{\log_b a} \cdot \lg(n))$;
> 3. se $f(n) \in \Omega\left(n^{\log_b (a+\varepsilon)}\right)$ para algum $\varepsilon > 0$ e se $af\left(\frac{n}{b}\right) \leq cf(n)$ para algum $c < 1$ e todo n suficientemente grande então $T(n) \in \Theta(f(n))$

Temos que o tempo de execução $T(n)$ em função do tamanho n da entrada do algoritmo *Merge Sort* é constante das linhas 17 à 21 mas varia nas linhas 22 e 23 ($T\left(\frac{n}{2}\right)$ cada), e 24 (cn, para uma constante $c > 0$). Assim temos que a forma geral do tempo de execução é:

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

Ou seja, conforme a definição $aT\left(\frac{n}{b}\right) + f(n)$ tem-se que

- $a = b = 2$;
- $n^{\log_b a} = n^{log_2 2} = n$;
- $f(n) = cn$, sendo $\Theta(cn) \equiv \Theta(n)$;
- e portanto $f(n) \in \Theta(n^{\log_b a})$ pois $\Theta(n^{\log_b a}) = \Theta(n)$.

Assim, temos pela aplicação do teorema que:

$$T(n) \in \Theta(n^{\log_b a} \cdot \lg(n)) \implies T(n) \in \Theta(n \cdot \lg(n))$$

Solução *quickSelect*

Definição

O algoritmo `quickSelect`, tal qual o algoritmo *Quick Sort*, foi desenvolvido por Tony Hoare e utiliza a mesma estratégia do último para comparar valores.

Conceitualmente, seu funcionamento se dá da seguinte maneira:

- Particiona-se a sequência à partir do valor de um elemento qualquer contido nessa denominado o pivô, separando-a em dois subconjuntos contendo respectivamente valores maiores e menores que este. Assim, o pivô passa a ocupar uma posição intermediária às partições e é considerado ordenado;

- Se o pivô assume a posição do índice buscado ou resta apenas um único índice no conjunto particionado, a busca se encerra; senão repete-se o procedimento na partição aquela que contiver o índice i (sendo este menor ou maior que o pivô).

Implementação

```c
#define array int*

void swap (int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int partition (array A, int pivot, int size) {
    int i, j, lastIndex = size - 1;

    swap(&A[pivot], &A[lastIndex]);
    for (i = j = 0; i < lastIndex; i++)
        if (A[i] <= A[lastIndex])
            swap(&A[i], &A[j++]);
    swap(&A[i], &A[j]);
    return j;
}

int * quickSelect (array A, int size, int i) {
    int pivot;

    if (size <= 1)
        return A;
    pivot = partition(A, i, size);
    if (A + pivot == A + i)
        return A + i;
    if (i < pivot)
        return quickSelect(A, pivot, i);
    pivot++;
    return quickSelect(A + pivot, size - pivot, i - pivot);
}
```

Correção

Podemos aferir que o algoritmo acima é correto pois

1. Se a a entrada contiver um único elemento, este é o elemento devolvido — Este é o caso base.

2. Senão, o conjunto é particionado na linha 25 e comparações são feitas na linha 26 para aferir se o índice foi encontrado no pivô ou, senão, na linha 27 para determinar em qual partição o procedimento de busca deve ser repetido até que as condições anteriores sejam satisfeitas.

3. Como toda partição é menor que o conjunto que lhe deu origem, no pior caso a partição avaliada eventualmente será pequena o suficiente para corresponder ao caso base.

Teste empírico

A correção do algoritmo anterior foi testada manualmente para pequenos arranjos utilizando o programa `manual_test.c`[5].

Tempo de Execução

O tempo de execução deste algoritmo pode variar bastante em função do posicionamento do pivô ao final de cada procedimento de partição. Analisemos, portanto, aqueles que podem ser considerados o melhor, o pior, e o caso médio.

Análise do melhor caso

O melhor caso para a execução deste algoritmo é aquele em que o i-ésimo menor elemento do arranjo já se encontra na i-ésima posição ao iniciarmos a busca. Assim, a partição será feita com este elemento enquanto pivô na linha 13, retornando o mesmo a sua posição inicial na linha 16, e a busca estará findada tendo percorrido o arranjo uma única vez na linha 27. Desta forma, o tempo de busca em função do tamanho da entrada escalaria de forma estritamente linear. Ou seja, $T(n) \in \Theta(n)$.

Análise do pior caso

O pior caso para a execução deste algoritmo seria aquele onde o usuário busca o menor valor de um arranjo que encontra-se perfeitamente em ordem crescente **à partir do segundo elemento**. Assim, o primeiro elemento tem o maior valor e cada processo de partição se dá de forma em que o pivô é o $(n-(i-1))$ maior elemento, produzindo assim partições de tamanho menor que a anterior em apenas uma unidade. Assim o arranjo é percorrido de maneira a realizar um número de S_n de comparações na linha 14 equivalente à:

$$S_n = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i = n(n-1) - \frac{(n-1)[(n-1)+1]}{2}$$

$$= (n-1)\left(n - \frac{n}{2}\right) = \frac{n(n-1)}{2} = \frac{n^2 - n}{2}$$

Ou seja, o tempo de execução do algoritmo restaria em ordem quadrática: $T(n) \in \Theta(n^2)$.

Análise do caso médio

Aplicando-se o Teorema Mestre conseguimos avaliar o **caso médio**, pois este admite que recursões são feitas em conjuntos menores de igual tamanho entre si. Fosse sempre este o caso com este algoritmo o índice i ser menor ou maior que aquele do pivô não afetaria o tempo de execução da recursão seguinte, o qual também não seria nem muito pequeno (com o tamanho da partição seguinte abaixo da média) ou muito grande (com o tamanho da partição seguinte acima da média).

Temos que o tempo de execução $T(n)$ em função do tamanho n da entrada do algoritmo *quickSelect* é constante senão nas linhas 25 (cn, para uma constante $c > 0$), 29 e 31 ($T\left(\frac{n}{2}\right)$ cada), das quais apenas uma delas é executada condicionalmente. Assim temos que a forma geral do tempo de execução é:

$$T(n) = T\left(\frac{n}{2}\right) + cn$$

Ou seja, conforme a definição $aT\left(\frac{n}{b}\right) + f(n)$ tem-se que

- $a = 1$ e $b = 2$;
- $n^{\log_b a} = n^{log_2 1} = n^0 = 1$;
- $f(n) = cn$, sendo $\Theta(cn) \equiv \Theta(n)$;
- e portanto
    - $f(n) \in \Omega\left(n^{\log_b (a+\varepsilon)}\right)$ pois $\Omega\left(n^{\log_2 (1+1)}\right) = \Omega(n)$;
    - $f\left(\frac{n}{2}\right) < f(n) \implies af\left(\frac{n}{b}\right) \leq \varepsilon f(n)$ para algum $\varepsilon < 1$;

Assim, temos pela aplicação do teorema que:

$$T(n) \in \Theta(cn) \implies T(n) \in \Theta(n)$$

Vemos então que teoricamente o caso médio aproxima-se mais da situação observada no melhor caso que do pior caso e, para valores de n suficientemente grandes, oferece melhor desempenho com relação a solução `mergeSelect`.

Objetivo

Iremos comparar o desempenho de ambas as soluções para uma mesma entrada de tamanho n, para valores de n cada vez maiores. Para tal utilizaremos o seguinte equipamento:

```
System:
  Host: manjaro Kernel: 5.10.70-1-MANJARO x86_64 bits: 64
  Desktop: GNOME 40.5 Distro: Manjaro Linux
Machine:
  Type: Portable System: Dell product: Inspiron 5548 v: A10
  serial: <superuser required>
  Mobo: Dell model: 0YDTG3 v: A02 serial: <superuser required>
  UEFI-[Legacy]: Dell v: A10 date: 05/28/2019
Battery:
  ID-1: BAT1 charge: 39.3 Wh (100.0%) condition: 39.3/42.2 Wh (93.3%)
CPU:
  Info: Dual Core Intel Core i7-5500U [MT MCP] speed: 2395 MHz
  min/max: 1500/3000 MHz
Graphics:
  Device-1: Intel HD Graphics 5500 driver: i915 v: kernel
  Device-2: AMD Topaz XT [Radeon R7 M260/M265 / M340/M360 / M440/M445
  / 530/535 / 620/625 Mobile]
  driver: amdgpu v: kernel
  driver: uvcvideo
  Display: x11 server: X.Org 1.20.13 driver:
  loaded: amdgpu,ati,modesetting resolution: 1920x1080~60Hz
  OpenGL: renderer: Mesa Intel HD Graphics 5500 (BDW GT2)
  v: 4.6 Mesa 21.2.3
Drives:
  Local Storage: total: 931.51 GiB used: 916.26 GiB (98.4%)
Info:
  Processes: 288 Uptime: 1h 59m Memory: 15.55 GiB
  used: 3.66 GiB (23.6%) Shell: fish inxi: 3.3.08
```

> *Output* do comando `inxi -b`, algumas informações foram omitidas.

Método

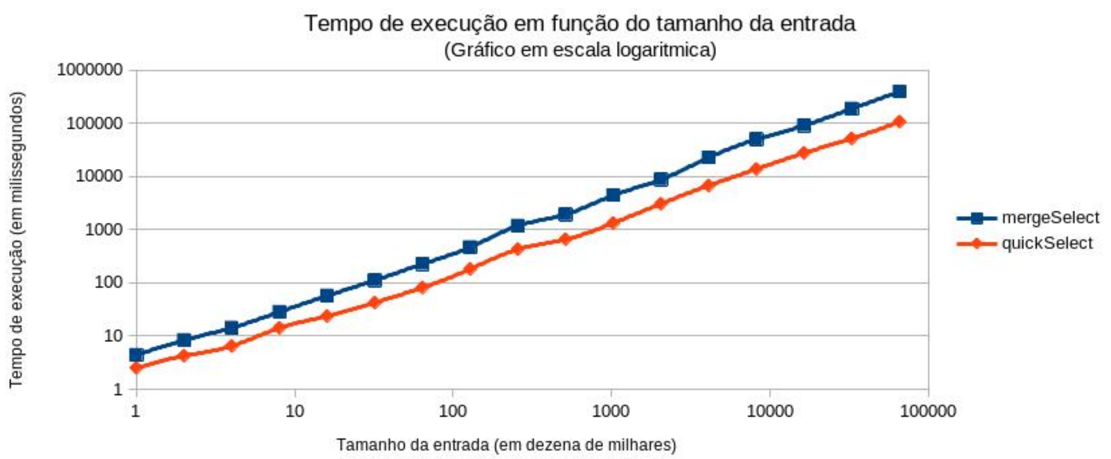
Os valores a serem avaliados foram sorteados fazendo uso do programa `gerador` [6] e depositados em um arquivo. Então, utilizando o comando `random`, uma função do shell `fish` em sua versão 3.3.1, sorteou-se um índice a ser buscado, inserindo-o ao final do arquivo. Finalmente, tais aquivos foram providos enquanto argumentos para os programas de teste[7] [8], e o desempenho destes foi aferido fazendo uso do comando `time`, também função do shell `fish`. À seguir vemos os comandos passados e um exemplo de saída:

```
> for n in 1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384
      wrn "Teste para |n = "(math "$n * 10000")"|"
      ./gerador.out $n > $n.txt
      random 0 (math "$n * 10000") >> $n.txt
      reg "|Merge Select|"
      time mergeSelect/automatic_test.out < $n.txt
      reg "|Quick Select|"
      time quickSelect/automatic_test.out < $n.txt
      rm $n.txt
  end

! Teste para n = 10000
Merge Select
O 331º elemento de menor valor: 70930890


________________________________________________________
Executed in    4,39 millis    fish           external
   usr time    2,16 millis  264,00 micros    1,89 millis
   sys time    1,98 millis   80,00 micros    1,90 millis

Quick Select
O 331º elemento de menor valor: 70930890


________________________________________________________
Executed in    2,51 millis    fish           external
   usr time    2,48 millis  300,00 micros    2,18 millis
   sys time    0,09 millis   90,00 micros    0,00 millis
```

O tempo descrito em `Executed in` será aquele que iremos avaliar.

Resultado

Os seguintes gráfico e tabela relatam o resultado da experimentação:



Tamanho da entrada	Tempo de execução — *mergeSelect*	Tempo de execução — *quickSelect*
1	4,39	2,51
2	8,35	4,24
4	14,27	6,47
8	28,30	14,20
16	57,38	23,56
32	111,11	42,34
64	225,69	81,37
128	465,20	181,65
256	$1,18 \times 10^3$	428,07
512	$1,93 \times 10^3$	658,30
1024	$4,43 \times 10^3$	$1,33 \times 10^3$
2048	$8,75 \times 10^3$	$3,06 \times 10^3$
4096	$22,62 \times 10^3$	$6,73 \times 10^3$
8192	$49,62 \times 10^3$	$13,74 \times 10^3$
16384	$90,54 \times 10^3$	$27,65 \times 10^3$
32768	$187,31 \times 10^3$	$51,45 \times 10^3$
65736	$390,29 \times 10^3$	$106,61 \times 10^3$

> Tamanho da entrada dado em dezena de milhares (10^4) e tempo de execução em milissegundos (ms).

Conclusão

A análise preliminar dos algoritmos de busca elaborou quantitativamente minha expectativa para crescimentos significativamente distintos do tempo de execução em função do tamanho da entrada, a qual foi seguidamente demonstrada em um experimento empírico. É possível afirmar que dentre os algoritmos apresentados o *quickSelect* é aquele mais eficiente na solução do problema apresentado dada uma entrada com valores aleatórios e com pouca ou nenhuma repetição, situação esta que corresponde de maneira mais próxima a seu caso médio. Podemos atribuir esta maior eficiência ao seu procedimento de particionamento que apenas parcialmente ordena o arranjo de valores, ainda que suficientemente para encontrar o i -ésimo maior valor, reduzindo desta forma o número de comparações feitas.

Por fim, fica demonstrada a importância e utilidade de métodos de análise de algoritmos para verificar a correção e desempenho destes à partir da avaliação de seu código, em particular na identificação de padrões de recorrência como a *Divisão e Conquista* e na aplicação do *Teorema Mestre* tal qual foi aqui realizado.

---

1. nUSP: 12543033; Turma 04. ↩
2. **Merge sort**. Disponível em: https://en.wikipedia.org/w/index.php?title=Merge_sort&oldid=1050948230. Acesso em: 8 nov. 2021. ↩

3. RIBEIRO, M. **Introdução à Análise de Algoritmos**. Disponível em: https://github.com/marciomr/apostila-iaa/blob/master/apostila-iaa.pdf. Acesso em: 13 out. 2021. ↩
4. BARRETO, G. **mergeSelect/manual_test.c**. Disponível em: https://git.disroot.org/SI/semestre_2/src/branch/master/Introdu%c3%a7%c3%a3o%20%c3%a0%20An%c3%a1lise%20de%20Algoritmos/EP%201/mergeSelect/manual_test.c. Acesso em: 13 out. 2021 ↩
5. BARRETO, G. **quickSelect/manual_test.c**. Disponível em: https://git.disroot.org/SI/semestre_2/src/branch/master/Introdu%c3%a7%c3%a3o%20%c3%a0%20An%c3%a1lise%20de%20Algoritmos/EP%201/quickSelect/manual_test.c. Acesso em 13 out. 2021 ↩
6. RIBEIRO, M. **gerador.c**. Disponível em: https://github.com/marciomr/IAA/blob/main/gerador.c. Acesso em: 12 nov. 2021 ↩
7. BARRETO, G. **mergeSelect/automatic_test.c**. Disponível em: https://git.disroot.org/SI/semestre_2/src/branch/master/Introdu%c3%a7%c3%a3o%20%c3%a0%20An%c3%a1lise%20de%20Algoritmos/EP%201/mergeSelect/automatic_test.c. Acesso em 13 nov. 2021 ↩
8. BARRETO, G. **quickSelect/automatic_test.c**. Disponível em: https://git.disroot.org/SI/semestre_2/src/branch/master/Introdu%c3%a7%c3%a3o%20%c3%a0%20An%c3%a1lise%20de%20Algoritmos/EP%201/quickSelect/automatic_test.c. Acesso em 13 nov. 2021 ↩