UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES
PROGRAMA DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

BRENO DA SILVA NOGUEIRA

# Problema da Seleção

IAA - Introdução a Análise de Algoritmos

São Paulo
2021

# INTRODUÇÃO

Na procura de algoritmos cada vez mais eficientes, procuramos sempre novos métodos para diminuir tempo e recursos na esperança de sempre chegarmos no famigerado O(log n). Uma das técnicas desenvolvidas nessa jornada é o conceito de Divisão e Conquista, que como o nome pode sugerir, divide o problema em problemas menores na esperança de poder resolvê-lo com mais eficiência. Nesta atividade trabalharemos com esse conceito de duas maneiras diferentes, a fim de determinar o quão eficiente as duas implementações são em resolver o problema da seleção.

Antes de qualquer coisa, o problema em si.

O problema da seleção se dá pelo seguinte enunciado: Dada uma sequência de inteiros $a_1 \ldots a_n$ e um inteiro $i$ tal que $1 \leq i \leq n$, o programa deve retornar o $i$-ésimo menor número da sequência.

As aplicações desse problema são muitas, além do já citado uso para encontrar medianas, pode ser usado para algoritmos de busca de menor preço, por exemplo, existem seus usos em estáticas, catalogação, entre outros.

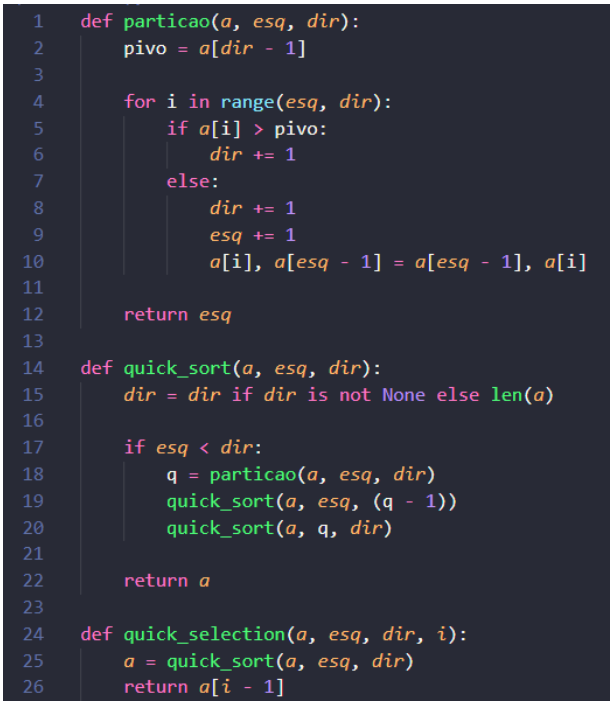
O problema em si é simples o bastante, até mesmo uma solução simples com um loop *for,* e uma sentinela seria o suficiente para resolver, contudo, buscamos uma forma eficiente e rápida de fazê-lo, e neste momento é onde o conceito de Divisão e Conquista entra.

Para testar a eficiência do método de DeC, como chamaremos daqui para frente, iremos testar a seleção por meio de partições, e colocar contra um algoritmo de ordenação, que, após ordenar, retorna o $i$-ésimo menor número. Sendo a única limitação usarmos um algoritmo de ordenação com o caso médio de O(n log n).

Apesar de não especificar o uso de um algoritmo de ordenação que use DeC, para a minha iteração do teste achei interessante usar QuickSort, que se utiliza desse método, para comparar a performance de partições em dois âmbitos diferentes. Então, iremos usar DeC para ordenar e para efetuar uma busca, e tiramos nossas conclusões ao final.

(Os algoritmos usados, os testes e os resultados retirados estão em anexo deste documento)

***Esboço da correção e tempo de processamento***



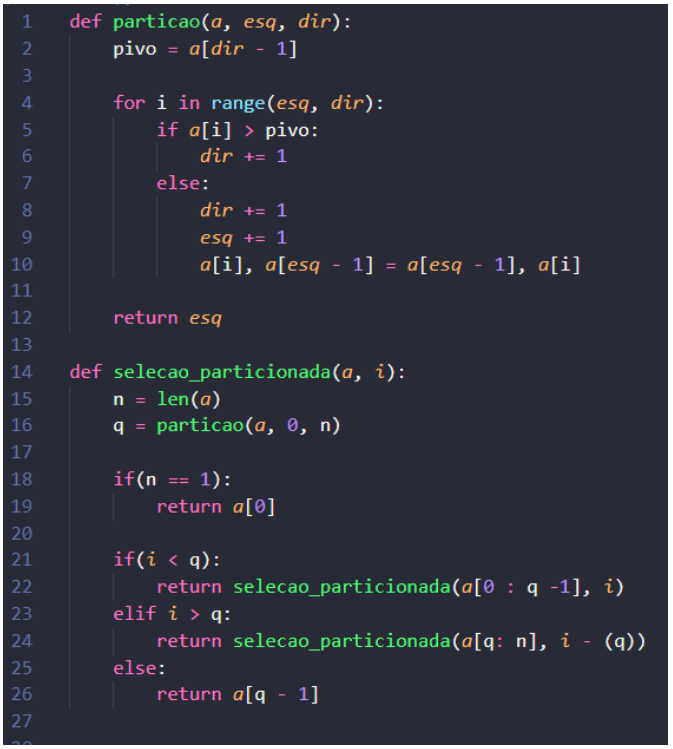
```python
def particao(a, esq, dir):
    pivo = a[dir - 1]

    for i in range(esq, dir):
        if a[i] > pivo:
            dir += 1
        else:
            dir += 1
            esq += 1
            a[i], a[esq - 1] = a[esq - 1], a[i]

    return esq

def quick_sort(a, esq, dir):
    dir = dir if dir is not None else len(a)

    if esq < dir:
        q = particao(a, esq, dir)
        quick_sort(a, esq, (q - 1))
        quick_sort(a, q, dir)

    return a

def quick_selection(a, esq, dir, i):
    a = quick_sort(a, esq, dir)
    return a[i - 1]
```

**Figura 1**: Algoritmo de Quick Selection (Quick Sort + Seleção de *i*-ésimo menor elemento)

O Quick Selection, como o próprio nome pode sugerir, se utiliza do algoritmo de ordenação Quick Sort, que por si só utiliza DeC. Seu funcionamento é bem simples, o array é particionado de acordo com um "pivô", então é iterado para que todos os elementos a sua esquerda sejam menores do que ele, e que todos os elementos a sua esquerda sejam maiores do que ele, ou seja, estará ordenado. Retornando para a função principal, o array será dividido em dois: um será da posição inicial até uma posição antes do pivô no array, e outro uma posição depois do pivô no array até o final, ambos neste momento estão desordenados. Esse processo é repetido recursivamente, até que todos os pivôs estejam ordenados, em outras palavras, o array inteiro esteja ordenado.

Aqui podemos ver o DeC em ação, diminuindo o tamanho do array exponencialmente e organizando um elemento por vez, criando uma boa estratégia para um algoritmo de ordenação.

O tempo de processamento deste algoritmo, como já visto em aula, no seu tempo médio, é de (n log n).



```python
def particao(a, esq, dir):
    pivo = a[dir - 1]

    for i in range(esq, dir):
        if a[i] > pivo:
            dir += 1
        else:
            dir += 1
            esq += 1
            a[i], a[esq - 1] = a[esq - 1], a[i]

    return esq

def selecao_particionada(a, i):
    n = len(a)
    q = particao(a, 0, n)

    if(n == 1):
        return a[0]

    if(i < q):
        return selecao_particionada(a[0 : q -1], i)
    elif i > q:
        return selecao_particionada(a[q: n], i - (q))
    else:
        return a[q - 1]
```

**Figura 2**: Algoritmo de Seleção Particionada

A Seleção Particionada, assim como o Quick Sort usa o particionamento como seu método de procura, contudo, mesmo se utilizando de particionamento e recursão, esses recursos são utilizados de uma maneira um pouco diferente. A primeira parte do algoritmo, é idêntica ao do Quick Sort, um pivô é eleito, todos menores para esquerda, maiores para a direita, pivô é ordenado. A diferença é no que é feito com o índice do pivô, caso o índice dele seja maior do o *i* que procuramos, iremos procurar o i-ésimo menor do lado direito, onde todos os números menores do que eles estão, e vice versa. Caso seja igual, achamos o *i*-ésimo menor, e assim devolvemos ele, alternativamente, se chegar ao ponto em que o array só tem um elemento, ele tem que ser o menor do array, e o retornamos (este caso só é possível ao procurar o primeiro menor número do array).

Para o tempo de processamento, no caso médio, temos que as partições são feitas mais ou menos na metade, então temos que o for de comparação será rodado (n + n/2 + n/4 + …) e que será menor que 2n, portanto temos a grandeza de O(n).

## OBJETIVO

Para a análise dos dois códigos, após sua implementação, foram criados uma série de testes para verificar o tempo de execução de ambos os códigos de acordo com a mesma entrada. Para que não houvesse nenhum desvio muito grande de resultados, os testes foram repetidos cinco vezes e iremos analisar somente a média dos tempos de execução.

Para o algoritmo de Quick Selection, temos a seguinte tabela de tempo por entrada:

| Quick Select | | | | | | |
|---|---|---|---|---|---|---|
| | Tempo de processamento (em segundos) | | | | | |
| Entrada (em milhões) | Rodada 1 | Rodada 2 | Rodada 3 | Rodada 4 | Rodada 5 | Média |
| 1 | 4,680 | 4,236 | 4,207 | 4,189 | 4,240 | 4,310 |
| 5 | 26,825 | 25,183 | 25,098 | 25,340 | 25,301 | 25,549 |
| 10 | 56,767 | 55,556 | 55,563 | 55,948 | 55,850 | 55,937 |
| 15 | 86,472 | 86,186 | 85,945 | 85,729 | 86,390 | 86,144 |
| 20 | 120,214 | 120,511 | 119,638 | 119,615 | 120,238 | 120,043 |
| 25 | 157,944 | 151,043 | 149,315 | 149,488 | 150,579 | 151,674 |
| 30 | 188,490 | 185,102 | 185,197 | 185,350 | 185,289 | 185,886 |
| 35 | 222,743 | 221,127 | 221,091 | 221,253 | 220,741 | 221,391 |
| 40 | 255,236 | 256,073 | 253,990 | 256,128 | 255,822 | 255,450 |
| 45 | 285,406 | 284,712 | 283,397 | 285,484 | 283,445 | 284,489 |
| 50 | 340,220 | 338,799 | 337,016 | 339,129 | 339,740 | 338,981 |
| 55 | 365,860 | 363,943 | 361,947 | 363,296 | 362,190 | 363,447 |
| 60 | 404,900 | 399,775 | 396,972 | 398,752 | 397,034 | 399,487 |
| 65 | 436,143 | 433,722 | 434,005 | 435,638 | 435,417 | 434,985 |
| 70 | 474,528 | 475,561 | 472,244 | 475,162 | 475,530 | 474,605 |
| 75 | 534,327 | 532,511 | 529,663 | 529,330 | 531,795 | 531,525 |
| 80 | 548,519 | 547,888 | 547,646 | 546,048 | 549,742 | 547,969 |
| 85 | 590,173 | 592,458 | 591,971 | 589,120 | 592,976 | 591,339 |
| 90 | 614,233 | 615,990 | 612,919 | 615,252 | 616,149 | 614,909 |
| 95 | 666,478 | 666,539 | 663,546 | 669,230 | 665,150 | 666,189 |
| 100 | 714,501 | 715,284 | 712,944 | 713,527 | 713,563 | 713,964 |

**Tabela 1**: Algoritmo de Quick Selection

Como podemos perceber, os tempos não se desviam muito entre si, e como um algoritmo de (n log n) o aumento gradativo de tempo de execução em razão do aumento da entrada não saiu muito do esperado. Podemos perceber que é um algoritmo ainda custoso, se perceber que com cem milhões de entradas ele pode demorar mais de dez minutos em tempo de execução.

Para o algoritmo da Seleção Particionada, temos a seguinte tabela:

| Seleção Particionada | | | | | | |
|---|---|---|---|---|---|---|
| | Tempo de processamento (em segundos) | | | | | |
| Entrada (em milhões) | Rodada 1 | Rodada 2 | Rodada 3 | Rodada 4 | Rodada 5 | Média |
| 1 | 0,102 | 0,111 | 0,095 | 0,098 | 0,098 | 0,101 |
| 5 | 2,187 | 2,185 | 2,192 | 2,204 | 2,197 | 2,193 |
| 10 | 12,315 | 12,220 | 12,224 | 12,164 | 12,374 | 12,260 |
| 15 | 51,527 | 51,482 | 51,342 | 51,313 | 51,768 | 51,486 |
| 20 | 39,884 | 39,824 | 39,807 | 39,707 | 40,036 | 39,852 |
| 25 | 49,536 | 49,425 | 49,336 | 49,253 | 49,579 | 49,426 |
| 30 | 38,018 | 37,969 | 37,935 | 37,915 | 38,325 | 38,032 |
| 35 | 69,695 | 69,764 | 69,663 | 69,565 | 70,306 | 69,799 |
| 40 | 64,528 | 64,447 | 64,323 | 64,098 | 64,797 | 64,438 |
| 45 | 33,213 | 33,386 | 33,331 | 33,219 | 33,562 | 33,342 |
| 50 | 50,061 | 50,199 | 49,772 | 50,069 | 50,555 | 50,131 |
| 55 | 46,577 | 46,668 | 46,710 | 46,572 | 46,892 | 46,684 |
| 60 | 68,113 | 68,389 | 68,298 | 68,206 | 68,727 | 68,347 |
| 65 | 66,128 | 66,299 | 66,241 | 66,329 | 66,784 | 66,356 |
| 70 | 77,850 | 77,538 | 77,628 | 77,518 | 78,206 | 77,748 |
| 75 | 60,874 | 60,683 | 60,694 | 60,512 | 61,232 | 60,799 |
| 80 | 75,692 | 75,332 | 75,621 | 75,857 | 76,122 | 75,725 |
| 85 | 65,477 | 65,223 | 65,467 | 65,361 | 65,831 | 65,472 |
| 90 | 122,115 | 122,202 | 122,182 | 122,057 | 123,054 | 122,322 |
| 95 | 75,019 | 75,055 | 75,169 | 75,534 | 75,549 | 75,265 |
| 100 | 60,588 | 60,333 | 60,369 | 60,919 | 60,779 | 60,598 |

**Tabela 2**: Algoritmo de Seleção Particionada

Com a Seleção Particionada, podemos perceber uma generosa diminuição no tempo de execução, onde a iteração que mais demorou (122 segundos) se aproxima de um sétimo da iteração mais custosa do Quick Selection (713 segundos).

Podemos também perceber que os tempos parecem dar "pulos" indo de tempos maiores para menores, mesmo com o aumento da entrada. É fácil entender como algum erro do código, ou até mesmo de medida, contudo, analisando o algoritmo é compreensível que haja essas discrepâncias de tempo entre números maiores de entradas, já que a escolha de um pivô para a busca é importante para que o resultado seja encontrado mais facilmente. Como na nossa implementação o pivô foi escolhido de uma maneira estática (sempre o último elemento do array), ele está sujeito a acelerar ou retardar o processo de encontrar o *i*-ésimo menor elemento, dependendo do quão próximo ele está do número desejado, já que é o pivô que termina o número de recursões que o algoritmo irá executar.

## RESULTADOS

Com todos os dados das médias apresentados na parte anterior, podemos montar o seguinte gráfico:

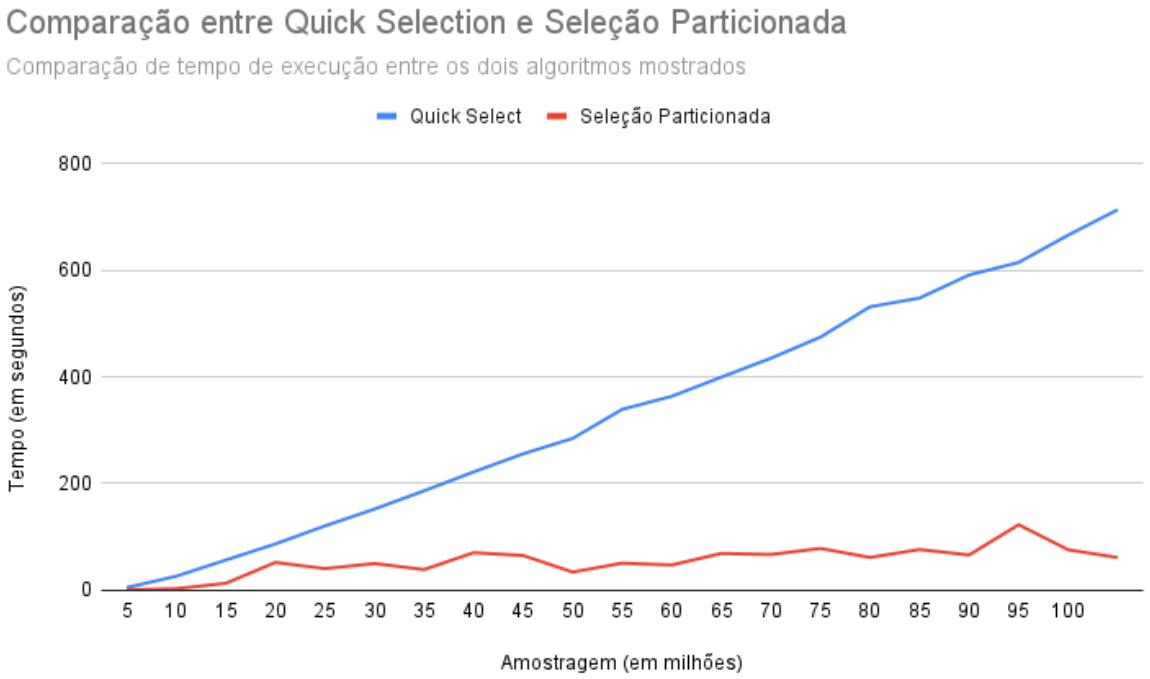



**Figura 3**: Gráfico de tempo por amostragem dos dois algoritmos.

Visando essa representação gráfica, podemos perceber o quão grandiosa é a discrepância entre os dois algoritmos, onde a Seleção Particionada é claramente mais eficiente em retornar a saída esperada. Conseguimos perceber também a instabilidade da Seleção Particionada, onde

Podemos culpar essa discrepância de eficiência principalmente no fato de que a ordenação do Quick Selection é uma operação custosa, e apesar de ser a mais segura e mais estável para resolver o problema, o preço é pago pelo alto consumo de tempo.

Mas é interessante perceber que mesmo ambos os algoritmos se utilizando de DeC e de particionamento, eles possuem eficiências bem diferentes. Portanto, serve de lição que mesmo um método sendo eficiente em uma aplicação, não necessariamente a mesma aplicação pode resolver outros problemas. Nesse caso, se utilizar de um método eficiente de ordenação, não necessariamente quer dizer que ele será eficiente em encontrar um certo número *n*.

## CONCLUSÃO

Após as análises feitas, chegamos à parte em que teremos que escolher entre as duas soluções apresentadas. A decisão aqui é óbvia, o algoritmo mais eficiente entre os dois é claramente a Seleção Particionada.

Apesar de seu caráter instável, um tanto randômico até podemos dizer, este algoritmo ainda é mais eficiente do que o Quick Selection, isso é, mesmo variando o tempo de execução com um mesmo número de entradas (pode ser até a mesma entrada, mas com a disposição dos elementos diferente), é improvável que a Seleção Particionada demore tanto quanto o Quick Selection. Claro, já que o outro deve ordenar o array, ele percorre por muito mais do array do que o SP.

Mesmo sendo o mais eficiente entre os dois, não necessariamente é a solução perfeita. Se algum algoritmo conseguir estabilizar ou achar um “pivô perfeito” para uma dada sequência para que a seleção fosse mais confiável em seu tempo de execução, com certeza seria um algoritmo mais indicado, não excluindo também o uso de outros métodos para chegar a um resultado melhor.