Problema do Máximo Subarray
Uma Análise de sua Complexidade

Matheus Garay Trindade[1], **Guilherme de Freitas Gaiardo**[1]

[1]Departamento de Eletrônica e Computação – Universidade Federal de Santa Maria (UFSM)
97.105-900 – Santa Maria – RS – Brazil

{mtrindade,ggaiardo}@inf.ufsm.br

Abstract. *This paper analises the classic computing problem of the maximum subarray. Three solutions with different complexities will be presented. This complexities will then be proved. For better understanding of what these complexities actually mean, various simulations were conducted with inputs of different sizes. With so, it is possible to perceive how the algorithm complexity impacts on its performance.*

Resumo. *Este artigo faz uma análise do clássico problema do subarray máximo. Serão apresentadas três soluções com diferentes complexidades. Essas complexidades serão então demonstradas. Para melhor entendimento do que essas complexidades significam, diversas simulações foram feitas com entradas de diferentes tamanhos. Com isso, é possível perceber o quanto a forma de crescimento do algoritmo em função da entrada têm um impacto direto na performance.*

1. Introdução

Em computação, é recorrente encontrar problemas que, devido a limitações dos computares, não se é possível resolver em tempo hábil para certas entradas. Muitas vezes, algoritmos completamente funcionais para entradas pequenas se tornam inutilizáveis para entradas grandes. No intuito de prever o desempenho de algoritmos, podemos usar técnicas para estimar o crescimento da quantidade de instruções necessárias para resolver um dado problema em função do tamanho da entrada. Vale ressaltar que essas técnicas são independentes de arquitetura e simplesmente mostram como o custo computacional se comporta.

Para exemplificar esse conceito, esse artigo faz uma análise de complexidade do problema clássico do máximo subarray. Nessa análise, serão apresentadas três algoritmos que resolvem esse problema. Cada um desses algoritmos possui uma complexidade, i.e., um crescimento do custo computacional, diferente. Essas complexidades serão demonstradas usando diversas técnicas, dependendo da estrutura do algoritmo. As soluções apresentadas são baseadas em [Cormen et al. 2001]

2. Problema do Subarray Máximo

O clássico problema do subarray máximo consiste do seguinte: dado um array de valores contendo valores negativos, encontrar o subarray que, dentro do conjunto de subarrays do array original, se somarmos os valores de seus elementos, possui a maior soma. Por exemplo, podemos tomar o array A apresentado na sequência:

$$A = [13, -3, -25, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4, 7]$$

Observando o array, percebemos que o subarray com a soma máxima consiste em:

$$[18, 20, -7, 12]$$

A soma desse subarray corresponde a 43. Nenhum outro subarray possui uma soma superior a esta. O problema então consiste em escrever um algoritmo para analisar o conjunto de subarrays e achar o que maximiza a soma de seus elementos.

3. Soluções para o Problema

A seguir, serão discutidas diversas formas de resolver o problema. Além disso, uma extensiva análise da solução será feita para demonstrar a complexidade em cada caso.

3.1. Força Bruta

A solução talvez mais óbvia e simples consiste em, exaustivamente, calcular todos os subarrays e compará-los entre si. O algoritmo consiste em dois laços aninhados para montar todos os subarrays. A soma é computada e salva. Se a soma for maior que a maior soma corrente (ou não existir soma corrente), ela passa ser a maior soma e o subarray máximo passa a ser o subarray que gerou esta soma. A solução é melhor apresentada no seguinte pseudocódigo:

Algorithm 1 Força Bruta

```
1:  function MAXSUBARRAY(A)
2:      maxSoma ← −∞                                   ▷ 1
3:      n ← len(A)                                     ▷ 1
4:      i0 ← −1                                        ▷ 1
5:      j0 ← −1                                        ▷ 1
6:      for i ∈ [1 : n] do                             ▷ n + 1
7:          soma ← 0                                   ▷ n
8:          for j ∈ [i : n] do                         ▷ Σ_{i=1}^{n+1} i = (n²+3n+2)/2
9:              soma ← soma + A[j]                     ▷ Σ_{i=1}^{n} i = (n²+n)/2
10:             if soma > maxSoma then                 ▷ Σ_{i=1}^{n} i = (n²+n)/2
11:                 maxSoma ← soma                     ▷ Σ_{i=1}^{n} t_j
12:                 i0 ← i                             ▷ Σ_{i=1}^{n} t_j
13:                 j0 ← j                             ▷ Σ_{i=1}^{n} t_j
14:             end if
15:         end for
16:     end for
17:     return A[i0 : j0]                              ▷ 1
18: end function
```

Ao lado de cada linha, temos a quantidade de vezes que a mesma é executada em função de n, o tamanho da entrada. Os somatórios surgem para representar a quantidade de vezes que a linha é executada levando em conta que ela está dentro de um laço. t_j foi inserido pois as linhas executadas dentro da condicional dependem da entrada, mas no pior caso, temos o esses somatórios igual a:

$$\sum_{i=1}^{n} t_j = \sum_{i=1}^{n} i = \frac{n^2+n}{2}$$

Logo, temos que:

$$0 \leq \sum_{i=1}^{n} t_j \leq \frac{n^2+n}{2}$$

Assim, se somarmos o número de vezes que cada linha é executada, temos que a quantidade de instruções pode ser ser expressa pelo polinômio $T(n)$:

$$T(n) = an^2 + bn + c \mid a, b, c \in \mathbf{R}$$

Assim, desconsiderando os termos de baixa ordem, temos que, para essa solução:

$$f(n) = \theta(n^2)$$

3.2. Divisão e Conquista

Uma solução mais eficiente e elaborada é utilizar a técnica de divisão e conquista. A solução é definida recursivamente, dividindo o array original em arrays menores, até chegar ao caso básico onde existe somente um elemento. A ideia geral do algoritmo é dividir em três possibilidades. Ou o subarray máximo está à esquerda do meio do array, ou está à direita ou então está passando pelo meio do array. O algoritmo irá realizar a chamada recursiva para os dois primeiros casos, e para o caso de estar no meio, é utilizado uma chamada especial onde é calculado o subarray máximo de forma linear. Ao fim das chamadas recursivas, é verificado qual foi o retorno com a maior soma (maior subarray). A solução é descrita no Algoritmo 2. A sua complexidade será analisada usando o Teorema Mestre, descrito na Seção 3.2.1.

3.2.1. Teorema Mestre

O Teorema Mestre [Cormen et al. 2001] é uma solução para análise de recorrências. Ele não resolve qualquer recorrência, no entanto, resolve as da forma:

$$T(n) = aT(n/b) + f(n)$$

onde $a >= 1$ e $b > 1$. Sendo n o tamanho da entrada, a o número de divisões por recursão, n/b é o tamanho do subproblema da próxima recursão e $f(n)$ é o custo envolvido em computações fora das chamadas recursivas e o custo de junção das soluções. Existem três casos onde é possível determinar o crescimento do custo:

Se $f(n) = O(n^{log_b a - \epsilon})$, para algum $\epsilon > 0$, então a solução é $T(n) \in \theta(n^{log_b a})$.

O segundo caso é, se $f(n) = \theta(n^{log_b a})$, então a solução é $T(n) = \theta(n^{log_b a} lg\ n)$.

O terceiro e último caso válido é, se $f(n) = \Omega(n^{log_b a + \epsilon})$ para algum $\epsilon > 0$, e $af(n/b) <= cf(n)$, para uma constante $c < 1$ e todos n's suficientemente grandes, então a solução é $T(n) = \theta(f(n))$

Algorithm 2 Divisão e Conquista

```
1: function FINDMAXCROSSINGSUBARRAY(A, low, mid, high)
2:     left_sum = −∞
3:     soma = 0
4:     max_left = mid
5:     for i = mid ; i > low − 1 ; i = i − 1 do
6:         soma = soma + a[i]
7:         if soma > left_sum then
8:             left_sum = soma
9:             max_left = i
10:        end if
11:    end for
12:
13:    right_sum = −∞
14:    soma = 0
15:    max_right = mid
16:    for i = mid + 1; i < high + 1; i = i + 1) : do
17:        soma = som + a[i]
18:        if soma > right_sum then
19:            right_sum = soma
20:            max_right = i
21:        end if
22:    end for
23:    return(max_left, max_right, right_sum + left_sum)
24: end function
25:
26:
27: function SUBARRAYMAXIMODIVECONQ(A, low, high)
28:    if low = high then
29:        return (low, high, A[low])
30:    end if
31:
32:    mid ← ⌊(low+high)/2⌋
33:
34:    (l_low, l_high, l_sum) = SubarrayMaximoDivEConq(a, low, mid)
35:    (r_low, r_high, r_sum) = SubarrayMaximoDivEConq(a, mid + 1, high)
36:    (c_low, c_high, c_sum) = findMaxCrossingSubarray(a, low, mid, high)
37:
38:    if l_sum >= r_sumandl_sum >= c_sum then
39:        return (l_low, l_high, l_sum)
40:    else if r_sum >= l_sumandr_sum >= c_sum then
41:        return (r_low, r_high, r_sum)
42:    else
43:        return (c_low, c_high, c_sum)
44:    end if
45: end function
```

Resolvendo com a estratégia de divisão e conquista o problema do sub array máximo, podemos verificar que:

$$T(n) = 2T(n/2) + n$$

Onde $a = 2$ (as duas chamadas por nível de recursão), $b = 2$ (o problema é dividido ao meio), e $f(n) = n$ (tempo linear da chamada da função que verifica o meio do array). Temos que:

$$f(n) = \theta(n^{log_2 2}) = \theta(n)$$

Assim, a condição do segundo caso está satisfeitas. Aplicando o teorema, temos:

$$T(n) = \theta(n^{log_2 2} lg\ n) = \theta(nlg\ n)$$

4. Solução Linear

Existe ainda outra forma de resolver o problema do Subarray Máximo. Essa forma é descrita pelo Algoritmo 3. Esse algoritmo é baseado no famoso algoritmo de Kadane.

Algorithm 3 Recursivo

```
1: function _SUBARRAYMAXIMORECURSIVO(A, i, j, max_ending_here)
2:     if max_ending_here > 0 then
3:         (i, j, max_ending_here) = (i, j, max_ending_here + A[j)
4:     else
5:         (i, j, max_ending_here) = (j, j, A[j)
6:     end if
7:     (i1, j1, max_so_far) ← (0, 0, 0)
8:     if j + 1 < len(A) then
9:         (i1, j1, max_so_far) ← _SubarrayMaximoRecursivo(A, i, j + 1, max_ending_here)
10:    else
11:        (i1, j1, max_so_far) ← (i, j, max_ending_here)
12:    end if
13:    if max_ending_here > max_so_far then
14:        return (i, j, max_ending_here)
15:    else
16:        return (i1, j1, max_so_far)
17:    end if
18: end function
19: function SUBARRAYMAXIMORECURSIVO(A, i, j, max_ending_here)
20:    return _SubarrayMaximoRecursivo(A, 0, 0, −∞)
21: end function
```

A ideia se baseia no fato de que, se temos uma soma acumulada negativa, podemos iniciar uma nova contagem no elemento corrente. Isso se deve ao fato de que, se temos uma soma acumulada menor que zero, o maior subarray acumulado passa a ser apenas o elemento. Assim, começando no primeiro termo, vamos montando subarrays usando a ideia descrita anteriormente. Chamando recursivamente para o próximo elemento, calculamos o maior subarray terminando no próximo elemento. Avaliamos o retorno com o calculado e retornamos o maior.

Para a análise de complexidade, podemos descrever o algoritmo como a seguinte recorrência:

$$T(n) = T(n-1) + \theta(1)$$

$\theta(n)$ representa a complexidade de cada passo. Como é sempre constante, é evidente que temos $\theta(1)$. Para resolver a recorrência, podemos fazer o seguinte "chute":

$$T(n) = \theta(n)$$

Essa hipótese justifica o caso básico, já que:

$$T(1) = \theta(1)$$

E para o passo indutivo temos:

$$T(n) = T(n) = T(n-1) + \theta(1) = \theta(n) + \theta(1) = \theta(n)$$

Assim, temos que a solução exata da recorrência é $\theta(n)$. Portanto, o algoritmo tem complexidade linear.

5. Considerações Finais

Ao implementar-mos o algoritmo na linguagem Python, verificamos a eficiência dos três algoritmos para diferentes tamanhos e conteúdos de entradas. As entradas são geradas pseudo-aleatoriamente com a função $random.seed()$ e selecionam-se n números, tal que $n(i) \in [-1000, 1000]$. O tempo de duração do algoritmo é medido utilizando a função $time.process\_time()$ que contabiliza apenas o tempo total de CPU que o processo recebeu.

Os tempos, notação $t(a)$ (tempo do algoritmo a), em relação aos tamanhos de entrada estão expostos na tabela a seguir:

Tamanho	Resultados
5	$t(FB) \approx t(Lin) < t(DeC)$
10	$t(Lin) \approx t(FB) < t(DeC)$
15	$t(Lin) < t(FB) < t(DeC)$
50	$t(Lin) < t(FB) < t(DeC)$
57	$t(Lin) < t(FB) \approx t(DeC)$
65	$t(Lin) < t(DeC) < t(FB)$

Tabela 1: Mostra os resultados obtidos com os algoritmos. FB: Algoritmo Força Bruta; DeC: Divisão e Conquista; Lin: Algoritmo Linear.

Como podemos verificar, com n suficientemente pequeno (ie. menor que 10), os algoritmos de Força Bruta e Linear se comportam de forma parecida, variando pouco, conforme o conteúdo da entrada. O algoritmo Linear desponta como mais rápido para $n >= 15$. Já o Divisão e Conquista, para n pequeno (ie. menor que 65), é pior que o Força Bruta. Com $n > 65$ os resultados apontam o Força Bruta como pior caso.

Referências

Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. (2001). *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition.