UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

ANTÔNIO ANDSON DA SILVA

AVALIAÇÃO EXPERIMENTAL DE ALGORITMOS

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 DESCRIÇÃO

É apresentado um problema e dois algoritmos para este problema. Os algoritmos têm complexidades de tempo de pior caso distintos. Contém a descrição do problema, a descrição dos algoritmos, a análise de complexidade de pior caso dos algoritmos, e a avaliação experimental de caso médio dos algoritmos (descrita abaixo).

Avaliação experimental:

- São produzidas entradas aleatórias (análise de caso médio).
- Para cada algoritmo, é feito um gráfico do tempo de execução em função do log do tamanho da entrada. Cada tempo de execução é dividido pela complexidade de pior caso (explicado abaixo).
- Começa com um tamanho de entrada n que produz tempo de pelo menos alguns milissegundos (ou seja, se o tempo é truncado em zero não serve).
- Dividido cada tempo de execução pela valor da função de complexidade de pior caso. Por exemplo, se na análise teórica de pior caso determinar que o algoritmo é $\Theta(n \log n)$, é dividido o tempo de execução por $\Theta(n \log n)$.
- Para algoritmos polinomiais, é dobrado o tamanho da entrada até perceber uma convergência. Para algoritmos exponenciais, a entrada é aumentada em uma unidade em cada iteração, até atingir o tempo limite.
- Quando o gráfico converge para uma constante positiva, então a complexidade de caso médio tem o mesmo crescimento da complexidade de pior caso. Por outro lado, se convergir para zero (ou seja, o valor diminui com o aumento da tamanho da entrada, mesmo para entradas grandes) então o crescimento do caso médio é menor que o crescimento de pior caso. É discuto isso nos resultados.
- Como exemplo interessante temos o Quicksort, cuja complexidade de pior caso é $\Theta(n^2)$, mas no caso médio é $\Theta(n \log n)$.
- Se o gráfico não convergir, então (i) se não tem pontos suficientes; ou (ii) sua implementação está errada; ou (iii) sua análise teórica de pior caso está errada.

Na avaliação experimental é importante discutir se os resultados (caso médio) estão compatíveis com as complexidades teóricas de pior caso obtidas, e também deve-se informar o ambiente para realização dos experimentos (sistema operacional e configuração do hardware, como clock e memória RAM).

2 PROBLEMA

O **problema** é o seguinte: Dada uma haste de n polegadas de comprimento e uma tabela de preços p_i para $(i = 1,2,...,n)$, determine a receita máxima c que se pode obter cortando a haste e vendendo os pedaços. Observe que, se o preço p_n para uma haste de comprimento n for suficientemente grande, uma solução ótima pode exigir que ela não seja cortada (CORMEN *et al.*, 2009).

Vamos de exemplo: para $n = 4$

Tabela 1 – Exemplo de entrada dos algoritmos

tamanho i	1	2	3	4
preço p_i:	2	7	8	10

Temos 8 possibilidades:

- Estratégia ideal é cortar em 4 partes (tamanhos 2 e 2) retornando $c = 7+7 = 14$
- Cortando em 2 partes (tamanhos 1 e 3) retorna $c = 2+8 = 10$
- Cortando em 2 partes (tamanhos 2 e 2) retorna $c = 7+7 = 14$
- Cortando em 2 partes (tamanhos 3 e 1) retorna $c = 8+2 = 10$
- Cortando em 3 partes (tamanhos 1, 1 e 2) retorna $c = 2+2+7 = 9$
- Cortando em 3 partes (tamanhos 1, 2 e 1) retorna $c = 2+7+2 = 9$
- Cortando em 3 partes (tamanhos 2, 1 e 1) retorna $c = 7+2+2 = 9$
- Cortando em 4 partes (tamanhos 1, 1, 1 e 1) retorna $c = 2+2+2+2 = 8$
- Não cortando (tamanho 4) retorna $c = 10$

<PRE-COND>: ambos algoritmos têm como entrada um vetor $p[1..n]$ representando a tabela de preços e um inteiro positivo n que representa o tamanho da haste.

<POS-COND>: ambos retornam c que é a máxima receita possível para uma haste de comprimento n.

3 ALGORITMOS

3.1 Algoritmo Recursivo

O problema é particionado em subproblemas desconexos, sendo resolvidos por recursividade, e no final as soluções são combinadas para retornar a solução do problema original.

- Corta-se um primeiro pedaço de tamanho i e o pedaço restante é cortado da melhor maneira possível. Como o primeiro pedaço tem tamanho i, o segundo pedaço tem tamanho $n-i$.
- O valor do primeiro corte, custa $p_i + c_k$ $(k = n-i)$, onde c representa o custo. Então, para encontrar c, temos:
  - $c_n = max(p_i + c_k)$.
  - $c_0 = 0$, como caso base, visto que com esse tamanho, haste não existe.

Algoritmo 1: Solução Algoritmo Recursivo - CORTE-HASTE-REC

Entrada: (p,n)

Saida: (c)

início

$n == 0$ retorna 0 ;

$c = -\infty$ (problema de maximização);

para i = *1 até n* **faça**

$c = max_i(c, p[i]$+CORTE-HASTE-REC$(p, n-i))$ (comparação dos custos dos cortes);

fim

retorna c;

fim

3.1.1 Execução

A execução desse algoritmo funciona da seguinte forma: $CHR(n) = max(p[i] + CHR(n-k)), (0 <= i < n)$.

- P/ n = 4
  - CHR(4) = max(p[0]+CHR(3),p[1]+CHR(2),p[2]+CHR(1),p[3]+CHR(0))
  - CHR(3) = max(p[0]+CHR(2),p[1]+CHR(1),p[2]+CHR(0))

- CHR(2) = max(p[0]+CHR(1),p[1]+CHR(0))
- CHR(1) = max(p[0]+CHR(0))
- CHR(0) = 0

3.1.2 Complexidade

A complexidade da função será dada pela quantidade de chamadas recursivas:

- Para n = 3 temos 8 chamadas recursivas.
- Para n = 4 temos 16 chamadas recursivas.
- Para n = 5 temos 32 chamadas recursivas.
- Analisando para outros n, conseguimos visualizar um padrão, onde a quantidade de chamadas recursivas é igual 2^n.
- Logo, a complexidade é $\Theta(2^n)$.
- Essa complexidade é a por conta do método de força bruta, visto que essa solução avalia todos os casos possíveis e escolhe o melhor deles.

Outra forma de resolver a complexidade, é pelo método indutivo:

- $T(n) = 1 + \sum_{i=0}^{n-1} T(i)$.
- Hipótese: $T(n) = 2^n$.
- Caso base: $T(0) = 2^0 = 1$.
- Passo Indutivo: $T(k) = 2^k$ para todo $0 <= k < n$, logo, $T(n) = 1 + \sum_{i=0}^{n-1} 2^i = 1 + (2^n - 1)/(2-1) = 2^n$.

3.2 Algoritmo Programação Dinâmica

A solução por recursão é ineficiente, visto que o mesmo subproblema é resolvido várias vezes. Nessa solução, a ideia de programação dinâmica é utilizada, que é resolver por recursão um subproblema apenas um vez, guardando a mesma em uma tabela/memória. Ao invés de resolver um subproblema várias vezes, uma consulta é realizada, substituindo esse procedimento. Vamos também abordar o conceito ascendente ou Bottom UP. Bottom UP usa a ordem natural dos subproblemas (começa utilizando o caso base), onde um problema de tamanho i que é inferior do que um problema j, onde $i < j$, é resolvido antes. Sabendo disso:

- Seguindo o conceito ascendente, comentado acima, toda vez que um subproblema for resolvido, sua solução é armazenada em uma memória/tabela.

- A cada tamanho da haste, todos os casos de cortes são comparados, e em seguida o c mais lucrativo é armazenado (reforçando o que foi dito acima).
- Após armazenar essa solução ideal, os novos pedaços de tamanhos superiores, irão construir sua solução com base nesse armazenamento.

Algoritmo 2: Solução Programação Dinâmica - CH-BOTTOM-UP

Entrada: (p,n)

Saida: (c)

início

```
int memoria[n+1];
memoria[0] = 0;
para j=1 até n faça
    c = −∞ (problema de maximização);
    para i=0 até j faça
        c = max_i(c, p[i] + memoria[j − k]);
    fim
    memoria[j] = c;
fim
retorna memoria[n];
```

fim

3.2.1 Execução

Exemplo da execução do algoritmo para uma haste de 4 polegadas (segue Tabela 2):

Tabela 2 – Execução PD 01

tamanhos	1	2	3	4
preço \$ $p[i]$	1	5	8	9

- O caso base é tamanho 0 que retorna 0. O próximo valor a definir é v[1] que vai ser igual a 1. (segue Tabela 3)

Tabela 3 – Execução PD 02

tamanhos	1	2	3	4
v[j]	1			

- O próximo valor a definir é o v[2] que vai ser o maior valor entre (Segue Tabela 04):

  - $p[1] + v[21] = 1 + 1 = 2$.
  - $p[2] = 5$.

Tabela 4 – Execução PD 03

tamanhos	1	2	3	4
v[j]	1	5		

- O próximo valor a definir é o v[3] que vai ser o maior valor entre (Segue Tabela 05):
  - $p[1] + v[31] = 1 + 5 = 6$.
  - $p[2] + v[32] = 5 + 1 = 6$.
  - $p[3] = 8$.

Tabela 5 – Execução PD 04

tamanhos	1	2	3	4
v[j]	1	5	8	

- O próximo valor a definir é o v[4] que vai ser o maior valor entre (Segue Tabela 06):
  - $p[1] + v[41] = 1 + 5 = 9$.
  - $p[2] + v[42] = 5 + 5 = 10$.
  - $p[3] + v[43] = 8 + 1 = 9$.
  - $p[4] = 9$.

Tabela 6 – Execução PD 05

tamanhos	1	2	3	4
v[j]	1	5	8	10

- Quando os laços são finalizados retornamos o valor de v[n] que vai ser igual a 10.

3.2.2 Complexidade

A complexidade da função será dada pelos laços:

- O primeiro laço, vai de $j = 1$ até n representando $\Theta(n)$.
- O segundo laço, vai de $i = 0$ até j, onde $j + 1$ vezes é atuado para cada j, logo:
  - $\sum_{j=1}^{n}(j+1) = \Theta(n^2)$ (somas aritméticas polinomial)
- Portanto, a complexidade é $\Theta(n^2)$.

4 AVALIAÇÃO EXPERIMENTAL

Para realização dessa etapa, os códigos foram feitos em C++ e você pode acompanhar neste repositório https://github.com/andsonsilv/avaliacao-experimental-de-algoritmos . Para cada algoritmo foram utilizados entradas aleatórias e o tempo de execução está detalhado abaixo. O sorteio dos preços foram feitos utilizando o rand() do C++, onde em cada iteração pode ser sorteado de 0 até n_i, n_i = tamanho da entrada . Por exemplo: para $n_i = 50$, o preço escolhido pode ser 0 <= preço < 50.

4.1 Tempo de Execução

4.1.1 Recursivo

Dividindo em cada iteração, o tempo de execução pelo valor da função de complexidade de pior caso, nota-se que essas divisões convergem para **0,0024**.

TAMANHO DA ENTRADA: 11 - TEMPO GASTO: 1 ms
TAMANHO DA ENTRADA: 12 - TEMPO GASTO: 1 ms
TAMANHO DA ENTRADA: 13 - TEMPO GASTO: 2 ms
TAMANHO DA ENTRADA: 14 - TEMPO GASTO: 4 ms
TAMANHO DA ENTRADA: 15 - TEMPO GASTO: 7 ms
TAMANHO DA ENTRADA: 16 - TEMPO GASTO: 15 ms
TAMANHO DA ENTRADA: 17 - TEMPO GASTO: 31 ms
TAMANHO DA ENTRADA: 18 - TEMPO GASTO: 64 ms
TAMANHO DA ENTRADA: 19 - TEMPO GASTO: 127 ms
TAMANHO DA ENTRADA: 20 - TEMPO GASTO: 254 ms
TAMANHO DA ENTRADA: 21 - TEMPO GASTO: 514 ms
TAMANHO DA ENTRADA: 22 - TEMPO GASTO: 1018 ms
TAMANHO DA ENTRADA: 23 - TEMPO GASTO: 2042 ms
TAMANHO DA ENTRADA: 24 - TEMPO GASTO: 4078 ms
TAMANHO DA ENTRADA: 25 - TEMPO GASTO: 8145 ms
TAMANHO DA ENTRADA: 26 - TEMPO GASTO: 16305 ms
TAMANHO DA ENTRADA: 27 - TEMPO GASTO: 32600 ms
TAMANHO DA ENTRADA: 28 - TEMPO GASTO: 65108 ms

TAMANHO DA ENTRADA: 29 - TEMPO GASTO: 130253 ms
TAMANHO DA ENTRADA: 30 - TEMPO GASTO: 260390 ms
TAMANHO DA ENTRADA: 31 - TEMPO GASTO: 522141 ms

4.1.2 Programação Dinâmica

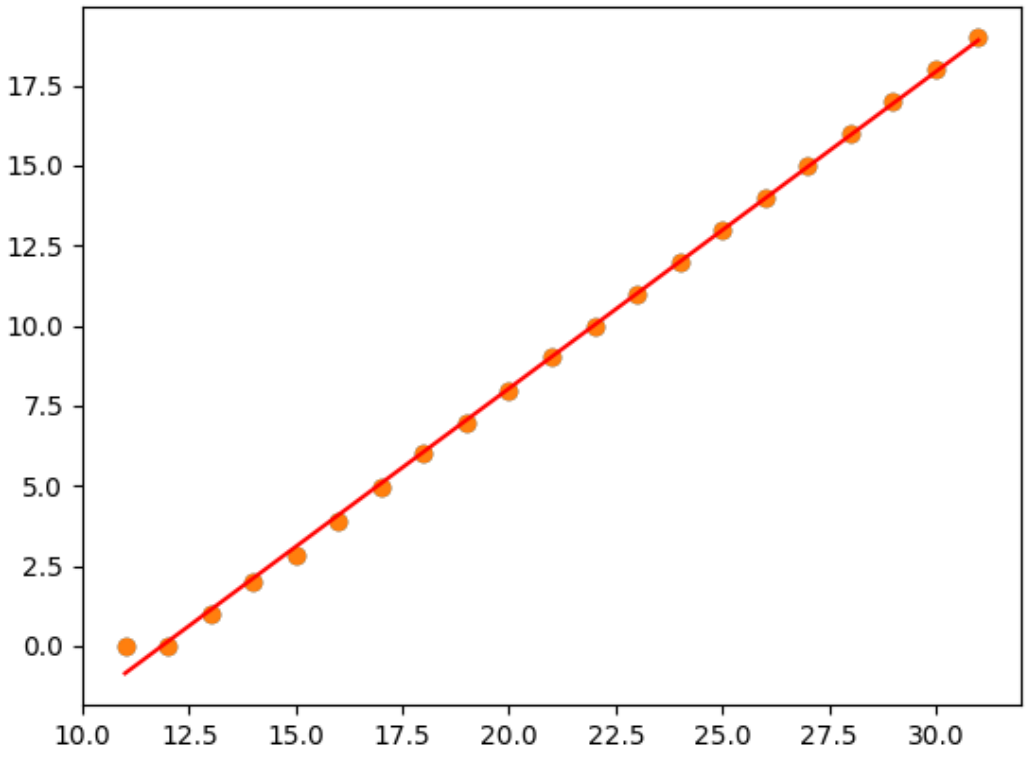
Dividindo em cada iteração, o tempo de execução pelo valor da função de complexidade de pior caso, nota-se que essas divisões convergem para uma constante positiva **4,10**.
TAMANHO DA ENTRADA: 400 - TEMPO GASTO: 1 ms
TAMANHO DA ENTRADA: 800 - TEMPO GASTO: 2 ms
TAMANHO DA ENTRADA: 1600 - TEMPO GASTO: 10 ms
TAMANHO DA ENTRADA: 3200 - TEMPO GASTO: 41 ms
TAMANHO DA ENTRADA: 6400 - TEMPO GASTO: 167 ms
TAMANHO DA ENTRADA: 12800 - TEMPO GASTO: 672 ms
TAMANHO DA ENTRADA: 25600 - TEMPO GASTO: 2683 ms
TAMANHO DA ENTRADA: 51200 - TEMPO GASTO: 10745 ms
TAMANHO DA ENTRADA: 102400 - TEMPO GASTO: 43017 ms
TAMANHO DA ENTRADA: 204800 - TEMPO GASTO: 172129 ms
TAMANHO DA ENTRADA: 409600 - TEMPO GASTO: 688239 ms

4.2 Gráficos do tempo de execução em função do LOG do tamanho da entrada de cada a algoritmo

Esta etapa foi feita em Python onde foram utilizadas as bibliotecas numpy, matplotlib.pyplot e LinearRegression. Códigos Python pode ser encontradas também no repositório GitHub já citado acima. avaliacao1.py é referente ao algoritmo recursivo e avaliacao2.py refere-se ao algoritmo de programação dinâmica. Segue abaixo os gráficos que apresentam uma reta, reta obtida através do método regressão linear. Também é mostrado o coeficiente da inclinação de cada reta. Por fim, mostramos uma constante c para cada algoritmo, onde se alcança através do coeficiente citado anteriormente juntamente com dois pontos da função. Com a constante c mais o coeficiente e uma entrada k, conseguimos encontrar um estimado tempo de execução. Equação trabalhada sempre na forma básica: $T(n) = c.b^{a.n}.n^d.\log^e.n$.

4.2.1 Recursivo

Figura 1 – Reta 01



Fonte: elaborado pelo autor

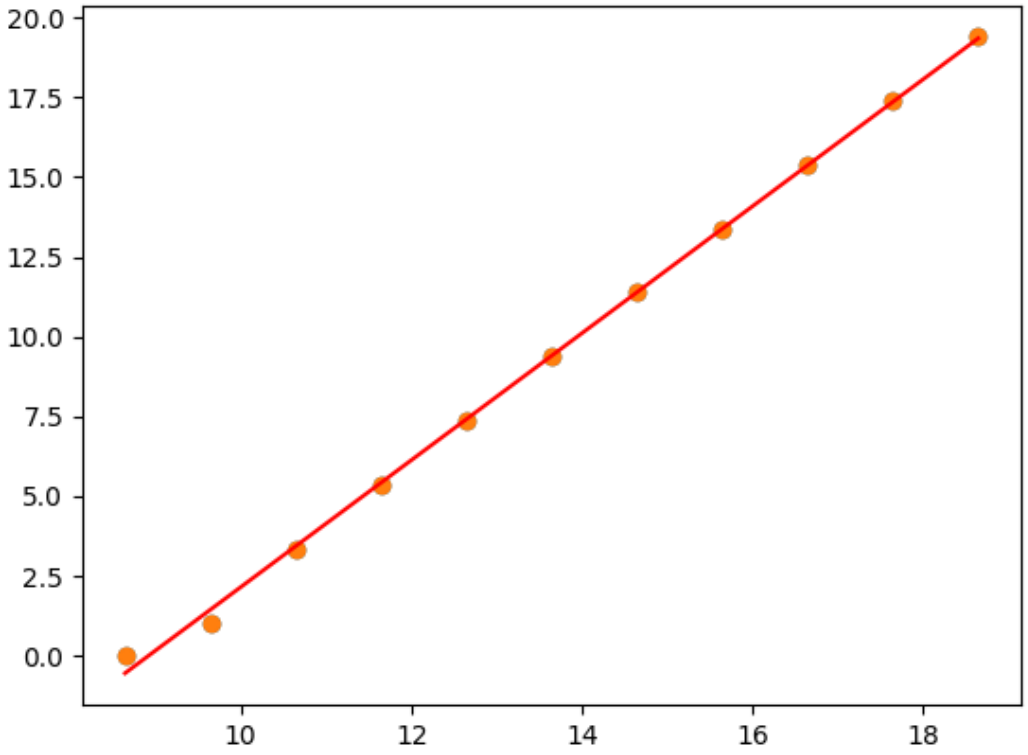
Como esperado, a reta ficou nitidamente alinhada aos pontos. O eixo x é representado por n e o eixo y por (n). O código Python também calculou o coeficiente da inclinação que é aproximadamente **SLOPE: [0,989]** e o intervalo de confiança, com confiança 95% que é SLOPE CONF INTERVAL: [0,97208705 1,0055886]. Calculando a constante c e pegando o último par de pontos do tempo de execução, temos:

- $T(n) = c.b^n$.
- Como o coeficiente da reta é igual $\log b$ (a constante).
- $b \approx 2^{0,989} \approx 1,984$
- $522141 \approx c.1,984^{31}$.
- $c \approx 3,07971.10^{-4}$

Podemos estimar para qualquer entrada um tempo de execução, como por exemplo: $T(50) \approx$ 232055865613 ms.

4.2.2 Programação Dinâmica

Figura 2 – Reta 02



Fonte: elaborado pelo autor

Como esperado, a reta ficou nitidamente alinhada aos pontos. O eixo x é representado por logn e o eixo y por $logT(n)$. O código Python também calculou o coeficiente da inclinação que é aproximadamente **SLOPE: [1,989]** e o intervalo de confiança, com confiança 95% que é SLOPE CONF INTERVAL: [1,93764082 2,04092146]. Calculando a constante c e pegando o último par de pontos do tempo de execução, temos:

- $T(n) = c.n^d$
- $688239 = c.409600^{1,989}$
- $c \approx 4,72884.10^{-6}$

Podemos estimar para qualquer entrada um tempo de execução, como por exemplo: $T(2048000) \approx 16904016$ ms.

4.3 Ambiente dos Experimentos

Sistema operacional e configuração do hardware, como clock e memória RAM.

- dispositivo: Nitro-AN515-52.
- memória: 7,6 GiB.
- processador: Intel® Core™ i5-8300H CPU @ 2.30GHz × 8.
- gráficos: NV137 / Mesa Intel® UHD Graphics 630 (CFL GT2).
- sistema operacional: Ubuntu 20.04.2 LTS 64 bits.
- versão do GNOME: 3.36.8.
- sistema de janelas: X11.

5 DISCUSSÃO DOS RESULTADOS

Desde o momento da escolha dos algoritmos, tentamos escolher dois algoritmos com graus de complexidades diferentes, no caso, um exponencial e um polinomial. O critério de escolha foi que um algoritmos teria que testar todos os casos resolvendo o mesmo subproblema várias vezes e o outro não, resolvendo um subproblema apenas uma vez. Os escolhidos como visto, foram por recursão e por programação dinâmica, com complexidade 2^n e n^2 respectivamente. Os dois algoritmos foram testados com entradas aleatórias e inicialmente, tentamos incluir entradas iguais, porém, com n pequenos, não conseguimos obter o tempo de execução em *ms* no algoritmo de programação dinâmica, e com n maiores, ficou inviável esperar a execução do algoritmo recursivo. Ficou viável esperar a execução do algoritmo recursivo com entrada de tamanho 31 e do algoritmo de programação dinâmica com tamanho 409600. A quantidade de entradas testadas foram satisfatória visto que a regressão linear de cada algoritmo retornou uma reta muito próxima dos pontos, como visto nos gráficos. Por fim, dividindo o tempo de execução pelo valor da função de complexidade de pior caso de cada algoritmo, concluímos que ambos tem o crescimento do caso médio igual ao crescimento da complexidade de pior caso, ambos convergiram para uma constante positiva.

REFERÊNCIAS

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to algorithms**. [S.l.]: MIT press, 2009.