Trabalho 1
Experimentos em Complexidade

Disciplina: 4645G-04 - Algoritmos e Estruturas de Dados I
Escola Politécnica — PUCRS
Elaborado pelo Prof. João Batista Oliveira

12 de março de 2023

Resumo

Este trabalho tem o objetivo de analisar o desempenho de algoritmos usando técnicas experimentais. Esta **não** é a técnica mais eficiente para medir complexidade, mas vale a pena conhecer antes de outras técnicas mais sofisticadas.

Introdução

Muitas vezes você vai desejar saber se o algoritmo que você desenvolveu é eficiente (ou seja, se é um jeito bom de resolver o problema) e esta é uma pergunta que tem uma resposta bastante difícil. Ela depende do problema que está sendo resolvido, da abordagem usada e de vários outros fatores. Cuidado, pode incluir teoria.

Por outro lado, verificar se o seu algoritmo é mais eficiente do que o algoritmo do vizinho é relativamente simples e existem técnicas descomplicadas para fazer isto. Este trabalho mostra uma destas técnicas. Ela funciona, mas tem alguns defeitos:

1. Antes de tudo, ela só pode ser usada quando o programa já existe. Ou seja, você já pensou no que fazer, já programou, já tirou os bugs do programa e está com ele prontinho. Depois disso tudo, se descobrir que ele não é bom vai ter que jogar fora todo esse esforço. Isso claramente é uma notícia ruim.[1]

2. Ela é baseada em executar o programa e obter dados da execução. Isto significa que se você escolher dados de entrada muito "bons", "ruins" ou viciados de alguma forma, você não terá resultados realistas.

Os fundamentos

A técnica fundamental para obter a complexidade de algoritmos de maneira experimental é bem simples. Imagine que o seu algoritmo é parecido com este aqui de baixo:

```
int f(n)
  if n <= 1 return 1
  return f(n-1) + f(n-2)
```

[1] E se você está querendo saber se dá pra medir a eficiência **sem** programar nada, é possível sim. Técnicas mais avançadas fazem isso.

Sim, este é o seu velho amigo Fibonacci recursivo. Agora que ele está programado, queremos saber como ele se comporta à medida que o valor de `n` varia. Podemos adaptar o algoritmo para ser executado com vários valores de `n` e depois analisar de duas formas diferentes:

1. Medindo tempos

   Neste caso, podemos medir o tempo de execução para cada um dos valores de `n` e depois analisar. Para fazer isso precisamos colocar medidas de tempo dentro do programa antes e depois de começar nosso algoritmo e medir a diferença entre elas. Este método, claro, está sujeito a flutuações do sistema operacional pois ele também tem outras coisas pra fazer.

2. Medindo operações

   Aqui é diferente: precisamos identificar as operações mais "significativas" do algoritmo e depois alterar o programa colocando uma variável para contar essas operações, imprimindo a quantidade delas no fim da execução.

Para o programa que foi desenvolvido, vamos adicionar uma variável `cont_op` que é incrementada no início da função e com isso saberemos quantas chamadas de função serão realizadas. Vamos usar essa medida de operações fazendo um segundo programa mais ou menos assim:

```
int f(n)
  cont_op++;
  if n <= 1 return 1
  return f(n-1) + f(n-2)

main()
  for(i=0; i<=30; i++)
    cont_op = 0
    f(i)          // altera cont_op
    print i, cont_op
```

0	1
1	1
2	3
3	5
4	9
5	15
6	25
7	41
8	67
9	109

Agora é só executar o programa para obter os dados, que iniciam como a tabela ao lado está mostrando. Perceba que **não** são os números de Fibonacci, mas são os números de quantas vezes a função é chamada.

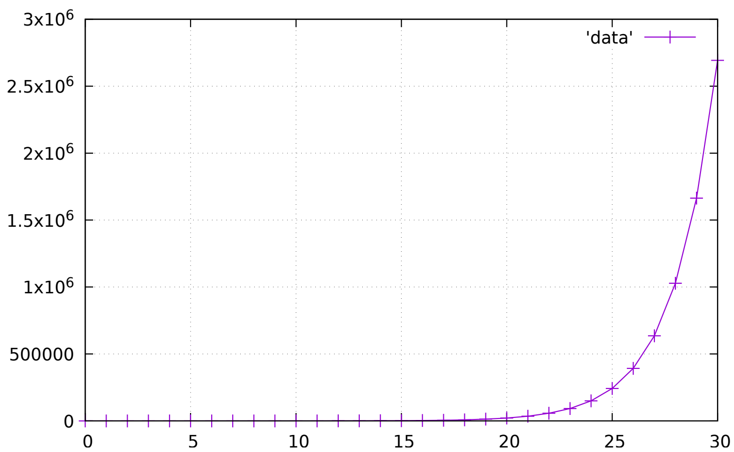

Figura 1: Plotagem do número de chamadas da função `f(n)`.

Veja na figura 1 que os números crescem depressa e é aí que está o problema: crescem com que velocidade? Como n^2? Como n^5? Como alguma outra função que não conhecemos? Esta função desconhecida informaria exatamente com que velocidade o algoritmo consome operações e permitiria prever quantas operações ele precisaria consumir para resolver o problema para outros valores de n.[2]

Durante a história da informática descobrimos que as funções típicas para o gasto de operações de um algoritmo separam-se em dois grandes grupos:

Funções polinomiais Nestas funções a variável (ou variáveis) representando o tamanho do problema é usada em um polinômio, como por exemplo

$$f(n) = n^3 + 4n + 16.$$

Perceba que os expoentes não mudam quando n for alterado. Por outro lado, não há limitação para os expoentes, e a função

$$f(n) = n^{200} + n^{199} + n^{198} + \cdots + n + 1$$

é um polinômio perfeitamente válido. Porém, se você pensar em algoritmos, é sempre bom que o maior expoente seja tão baixo quanto possível, certo?

Funções exponenciais Nestas funções as variáveis representando o tamanho do problema são usadas **como expoentes**, como por exemplo

$$f(n) = 3 * 2^n + 11.$$

No caso dos polinômios você tinha uma base que crescia mas agora é o expoente que cresce cada vez mais rápido. A consequência é que estas funções costumam cresce muito mais depressa do que as polinomiais, e se o seu algoritmo consumir operações (ou tempo) de uma forma exponencial você tem um problema sério nas mãos.

Faça um teste com a função acima para alguns valores de n (digamos $n = 2, 3, 4, 5, 6$) e percebe que com o aumento de 1 unidade a função praticamente duplica de valor. Agora imagine um algoritmo que faz algo interessante com um vetor, mas colocando um elemento a mais vai levar o dobro do tempo. Com 15 elementos ele leva umas mil vezes o tempo que precisa para apenas 5 elementos e se forem 25 elementos o tempo aumenta um milhão de vezes. É assim.

Descobrindo a função

A esta altura já sabemos o suficiente para coletar os dados sobre o número de operações e podemos explorá-los para descobrir que tipo de função é a "função característica" do algoritmo. As funções mais simples de serem investigadas são as exponenciais (sim, logo o pior tipo), as que tem a forma parecida com

$$f(n) = a * b^n.$$

Primeiro a notícia ruim que você já sabia: simplesmente plotar os dados não serve pra nada por que é impossível olhar para uma curva como a da Figura 1 e identificar quem são a e b. Isso simplesmente não existe. Temos que tentar algo mais sério do que adivinhação.

[2] Pergunta para o futuro: se soubermos quantas oeprações precisa para calcular Fibonacci com $n = 10$ poderíamos prever quantas operações precisamos para $n = 80$? A resposta é sim e talvez não seja como você está pensando. Continue lendo até o fim.

Felizmente existe uma técnica para isso, que usa o fato de que exponenciais e logaritmos são funções inversas uma da outra. A pergunta que leva para uma solução é: "O que acontece se usarmos o logaritmo da função $f(n)$?" Fazendo isso, temos

$$\log(f(n)) = \log(a * b^n) = \log(a) + \log(b^n) = n * \log(b) + \log(a)$$

e você deve cumprir as tarefas abaixo:

1. Convencer-se de que está tudo certo;
2. Unir as duas pontas para obter

$$\log(f(n)) = n * \log(b) + \log(a)$$

3. Perceber que o lado direito dessa equação representa **uma linha reta**

$$\log(f(n)) = r * n + s$$

onde $r = \log(b)$ e $s = \log(a)$.

Talvez não pareça muita coisa, mas isso é tudo o que precisamos saber. Se extrairmos o logaritmo de nossos números de operações e aparecer uma linha reta, eles certamente vieram de uma função exponencial. Melhor ainda, podemos analisar a reta e descobrir quem são r e s e com eles achar a e b. Testando essa ideia, produzimos a Figura 2 com as operações do algoritmo para Fibonacci. Perceba que a escala em y é logarítmica, e ela realmente mostra uma linha (quase) reta.

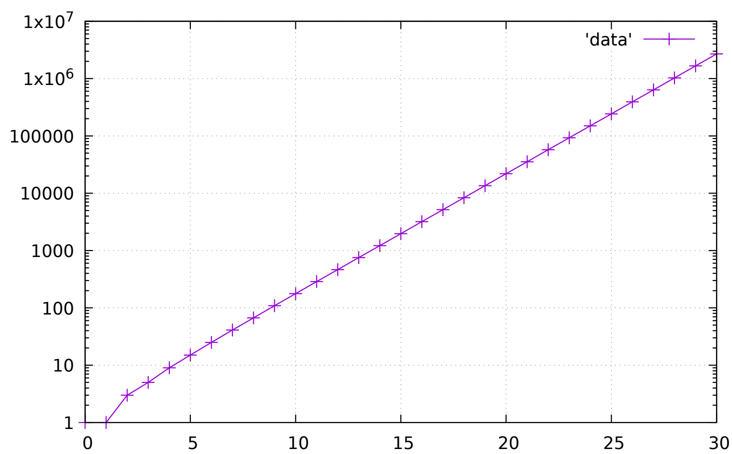

Figura 2: Plotagem do logaritmo do número de chamadas.

Para achar um valor aproximado para r, que é a inclinação da reta, usamos dois valores que obtivemos com nosso algoritmo:

$$\begin{aligned} f(2) &= 3 \\ f(30) &= 2692537 \end{aligned}$$

Neste caso usamos os logaritmos de f e temos

$$r \approx \frac{\log(2692537) - \log(3)}{30 - 2} = 0.48954936218\ldots$$

E como sabemos a relação entre r e b, temos imediatamente

$$b = e^r \approx 1.6315808\ldots$$

Ou seja, a função $f(n)$ cresce mesmo exponencialmente e tem uma base b que pode ser calculada aproximadamente. E é aproximadamente mesmo, pois outras técnicas mais avançadas comprovam que

$$b = \frac{\sqrt{5}+1}{2} \approx 1.618033988\ldots$$

Para algo que foi feito de maneira experimental, até que não está mau.

O caso polinomial

Para fazer um exemplo com o caso polinomial, vamos analisar outro algoritmo:

```
int f(n)
  local i, j, r
  r = 0
  for(i=0; i<=n; i++)
    for(j=i+2; j<=2*n; j++)
      r = r + 1
  return r
```

Vamos coletar informações sobre o número de operações feitas pelo algoritmo. Desta vez iniciamos com uma boa notícia: como a operação mais importante é o incremento da variável `r`, o próprio resultado da chamada da função é também a contagem de operações. Vamos usar o próprio programa para coletar os dados:

```
for(i=0; i<=100; i++)
  print i, f(i)
```

Agora é só executar o programa para obter os dados, que iniciam como a tabela ao lado está mostrando. Fazendo isso e plotando os dados obtemos a Figura 3.

0	0
1	1
2	6
3	14
4	25
5	39
6	56
7	76
8	99
9	125
10	154

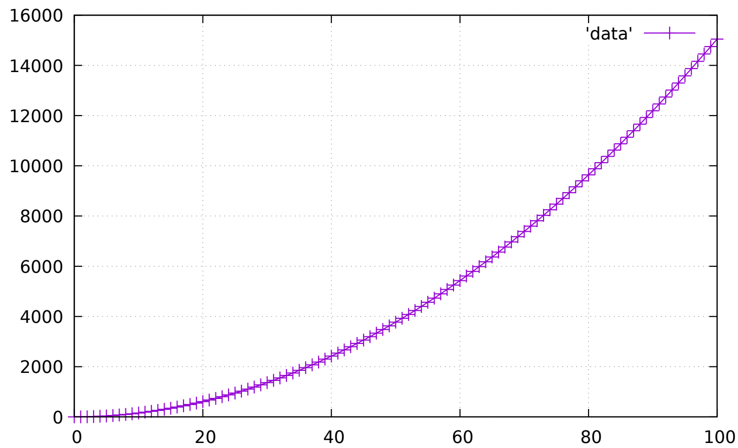

Figura 3: Plotagem do número de chamadas da função.

Até este ponto já podemos perceber que o algoritmo também consome operações rapidamente, mas não temos como saber a velocidade. Tentando usar a técnica do logaritmo mais uma vez, temos a Figura 4, que definitivamente não representa uma linha reta. Para avançar, precisaremos de mais análise.

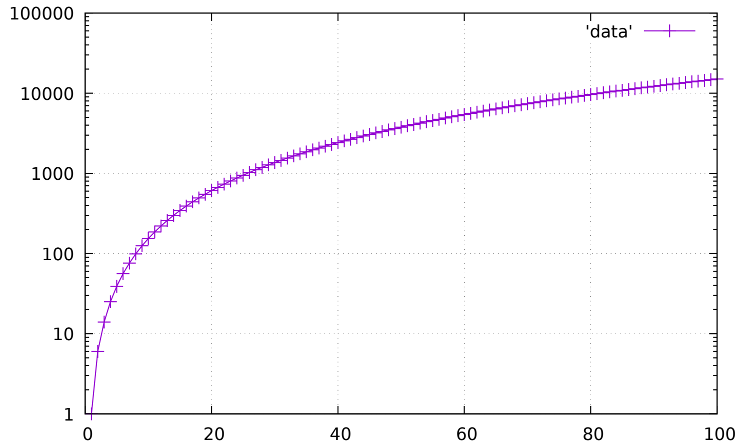

Figura 4: Plotagem do número de chamadas da função.

Se a função característica do algoritmo não é exponencial, a próxima possibilidade para ser examinada é um polinômio da forma

$$f(n) = n^b$$

onde b é um valor fixo e ainda desconhecido. Se tentarmos usar o logaritmo como usamos para as funções polinomiais, teremos

$$\log(f(n)) = \log(n^b) = b * \log(n)$$

e isso representa alguma curva logarítmica que deve ser parecida com a da Figura 4. Pelo jeito estamos no caminho certo e gostaríamos de fazer sumir o $\log(n)$ que está na expressão. No caso das exponenciais queríamos fazer um exp() sumir e usamos log() para isso, então agora podemos pensar em fazer o contrário! A maneira é calcular $f(e^n)$ em vez de $f(n)$, pois assim teremos

$$\log(f(e^n)) = \log((e^n)^b) = b * \log(e^n) = n * b * \log(e) = n * b * 1 = n * b$$

e ao fim obteremos **outra** reta. Nesta reta, o coeficiente angular b dá o expoente usado na função $f(n)$. Provavelmente você está perguntando como vai fazer para calcular $f(e^n)$, mas existe uma maneira mais simples de obter a mesma informação: se nos gráficos para as funções exponenciais "espremamos" o eixo y com um logaritmo, se fizermos isso também com o eixo x teremos o mesmo efeito. O resultado de aplicar logaritmos aos dois eixos está mostrado na Figura 5.

Para achar um valor aproximado para b usamos outra vez dois valores que podemos calcular com nosso algoritmo:

$$\begin{aligned} f(1) &= 1 \\ f(100) &= 15049 \end{aligned}$$

Agora usamos os logaritmos de f e também dos valores usados para x e temos

$$b \approx \frac{\log(15049) - \log(1)}{\log(100) - \log(1)} = 2.088753821\ldots$$

Ou seja, isto sugere que a função $f(n)$ cresce como uma função de segundo grau:

$$f(n) \approx n^{2.09}.$$

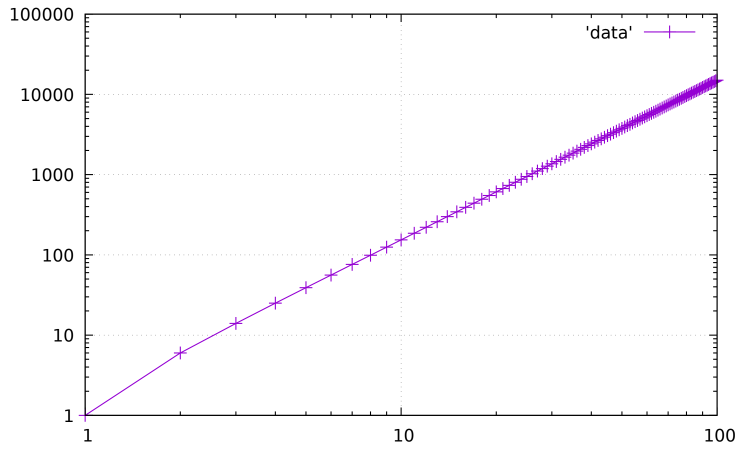

Figura 5: Plotagem do logaritmo do número de operações.

Isto tudo ainda é um resultado experimental e sujeito a algum erro (por exemplo, se fizermos a comparação entre $f(3)$ e $f(100)$ acharemos $b \approx 1.99055874\ldots$ o que reforça a hipótese de que o expoente seja 2 ou um valor muito próximo. Outras técnicas mais avançadas podem comprovar que o expoente é exatamente 2.

Agora a pergunta final: se o programa está correto e a contagem de operações está correta, não deveríamos ter achado um valor 2 exato? Por que não?

O software

O software usado para gerar estes gráficos foi o `gnuplot` na plataforma Linux (no Windows, o Excel é uma opção similar). Se armazenarmos os dados em arquivos com o mesmo formato usado neste texto (duas colunas, uma com n e outra com a contagem de operações) em um arquivo chamado `dados.txt` podemos plotar facilmente com os comandos

```
set grid
set logscale y
plot "dados.txt" w lp
```

Além destes dois comandos simples existe uma variedade de comandos para gerar figuras em formatos como `pdf` e outros, como por exemplo

```
set term pdf
set output "out.pdf"
plot "dados.txt" w lp
```

Investigue entrando no `gnuplot` e usando o comando `help`. E se você achar que `gnuplot` é complicado e tem muitas opções, você tem razão.

A missão

Sua missão é entregar um relatório fazendo esta mesma análise para os algoritmos a seguir, achando todos detalhes possíveis sobre eles: implementar, medir o número de operações, obter dados experimentais, construir os gráficos, determinar as funções características justificando sua resposta. Não esqueça de apresentar as suas conclusões. Sua missão é descobrir o quanto os algoritmos custam, **não é** descobrir o que os algoritmos fazem nem usar notação O() na análise por que ela será vista em sala de aula. Este relatório é **individual** e consiste **no Trabalho 1** da disciplina.

```
// Algoritmo 1
int f( int n ) {
  int i, j, k, res = 0
  int cont_op = 0
  for( i = 1; i <= n+1; i += 1 )
    for( j = 1; j <= i*i; j += i+1 )
      for( k = i/2; k <= n+j; k += 2 ) {
        res = res + n-1
        cont_op++
      }
  return cont_op
}
```

```
// Algoritmo 2
int f( int n ) {
  int i, j, k, res = 0
  int cont_op = 0
  for( i = n; i <= n; i += i/2+1 )
    for( j = i/2; j <= i*i; j += i+1 )
      for( k = n; k <= 2*n; k += i+1 ) {
        res = res + n
        cont_op++
      }
  return cont_op
}
```

```
// Algoritmo 3
int f( int n ) {
  int i, j, k, res = 0
  int cont_op = 0
  for( i = 1; i <= n*n; i += 2 )
    for( j = i/2; j <= 2*i; j += i/2+1 )
      for( k = j+1; k <= n+j; k += k/2+1 ) {
        res = res + abs(j-i)
        cont_op++
      }
  return cont_op
}
```

```
// Algoritmo 4
int f( int n ) {
  int i, j, k, res = 0
  int cont_op = 0
  for( i = n; i <= n*n; i += 2 )
    for( j = n+1; j <= n*n; j += 2 )
      for( k = j; k <= 2*j; k += 2 ) {
        res = res + 1
        cont_op++
```

```
    }
  return cont_op
}


// Algoritmo 5
int f( int n ) {
  int i, j, k, res = 0
  int cont_op = 0
  for( i = 1; i <= n*n; i += 1 )
    for( j = 1; j <= i; j += 2 )
      for( k = n+1; k <= 2*i; k += i*j ) {
        res = res + k+1
        cont_op++
      }
  return cont_op
}
```

Por que fazemos tudo isso?

Esta é uma pergunta muito justa e se você ainda não a fez, deveria ter feito. Ainda bem que estamos aqui para resolver o problema. Estudamos a maneira como os algoritmos "consomem" tempo ou operações por mais de um motivo:

1. Para poder comparar um algoritmo com outro, preferindo aqueles que realizam uma tarefa consumindo um número menor de operações (ou um número que cresce com uma velocidade menor).

2. Para poder prever quanto tempo um algoritmo vai levar para cumprir uma tarefa. Aqui está um exemplo: imagine um algoritmo que consome operações com uma velocidade $f(n) = n^3$. Neste caso, o que acontece quando dermos a ele um problema com o dobro do tamanho, ou seja, $2n$? A primeira ideia é prever o número de operações usando a função:

$$f(2n) = (2n)^3 = 2^3n^3 = 8n^3,$$

ou seja, resolver um problema com o dobro do tamanho leva 8 vezes mais tempo (ou operações) do que o tamanho original n. Perceba que este raciocínio não depende do valor de n e é verdadeiro sempre. Mudando de $n = 10$ para $n = 20$ aumenta 8 vezes, de $n = 100$ para $n = 200$ também. E faça o teste: para o triplo do problema (ou seja, $3n$) vai custar 27 vezes mais caro. Esta é uma péssima notícia e justifica que procuremos um algoritmo que consome operações de forma mais lenta. Para confirmar, procure um algoritmo dado nos exercícios que tenha uma função parecida com esta e faça um teste.

Pode ser que você ainda não tenha percebido ou ainda não tenham dito pra você, mas a triste verdade é que a maior parte dos algoritmos que fazem alguma coisa interessante tem uma função típica $f(x) = n^k$ com o expoente k maior do que 1. Ou seja, ao dobrar o tamanho do problema geralmente o preço da solução aumenta mais do que o dobro. Existem algumas exceções com certeza, mas quase sempre é assim. Vá acostumando, pois é raro que resolver o dobro de problema custe o dobro do tempo.

Entendendo tudo isso podemos fazer algumas previsões bastante avançadas. Por exemplo, se o algoritmo que consome $f(n) = n^3$ operações for usado para $n = 5$ e isso leva 1 segundo, podemos facilmente prever quanto tempo levaria para $n = 1000$. Vai ser um problema 200 vezes maior, o equivalente a perguntar por

$$f(200n) = (200n)^3 = 200^3 n^3 = 8000000n^3.$$

Então o tempo para um problema 200 vezes maior será 8000000 vezes o tempo do original, ou seja, 8000000 segundos. Se for assim mesmo, então teremos cerca de 92 dias de processamento. Enquanto espera três meses pelo resultado você pode continuar pensando e talvez você ache um outro algoritmo que tem a função $f(n) = n^2$. Ele levaria menos de 12 horas.