UnB Departamento de
Ciência da Computação

2016/1

Subalgoritmos

Prof. Guilherme N. Ramos

1 Introdução

Algoritmos iterativos permitem que se compute coisas mais complexas que simples aritmética, talvez algo "útil" como $\sqrt{n}$. Por exemplo, a implementação do método babilônico que aplica uma estratégia de tentativa e erro:

12-raiz2-3.c

```c
int  n, r = 1, tentativas = 0;

printf("Qual o valor de n?\n");
scanf("%d", &n);

if(n < 0)
  printf("Não sei calcular a raiz quadrada de número negativo.\n");
else {
  while(abs(r*r - n) > r) {
    r = (r+(n/r))/2;
    ++tentativas;
  }

  printf("Depois de %d tentativas, a aproximação da raiz de %d é %d.\n",
 tentativas, n, r);
}
```

Depois de tantas versões de instruções para obter o valor aproximado, está bem claro *como realizar* as computações para calcular r. Entretando, raramente se quer conhecer estes detalhes, o interessante é obter o resultado para utilizá-lo (por exemplo, para calcular a distância entre personagens em um plano cartesiano). Assim, é de interesse que se tenha uma forma mais abstrata de representar estes cálculos de modo que possa lidar apenas com o resultado, algo simples como $r = \sqrt{n}$.

A abstração $\sqrt{\ }$, permite separar os detalhes da implementação dos da utilização da computação (seu comportamento) [1]. Pode-se considerar a implementação como uma *caixa preta*, e que, conhecendo seu comportamento esperado, basta saber interagir com ela (como lidar com as entradas/saídas).

2 Subalgoritmos

A resolução de um problema torna-se mais fácil se é possível dividí-lo nos subproblemas que o compõem. Um *subalgoritmo* é o algoritmo que define uma solução para um subproblema específico. Esta possibilidade de modularizar o algoritmo facilita planejamento/implementação da solução, também a composição/compreensão do código, e permite que um mesmo módulo seja reaproveitado em diversas aplicações. Pode-se controlar a complexidade do programa usando abstrações que escondem os detalhes quando apropriado [2]. Por exemplo:

Implementação

```
if(x < y)
    z = x;
else
    z = y;

while(abs(r*r - n) > r)
    r = (r+(n/r))/2;
```

Abstração

```
z = min(x, y);

r = raiz2(n);
```

Na verdade este tipo de abstração não é novidade, `printf` têm sido usado já há algum tempo pra abstrair o complicado processo de mostrar texto formatado na saída padrão. O mesmo se aplica a `scanf`.

Por exemplo, suponha que a tarefa em questão é determinar as raízes de uma equação de segundo grau utilizando a fórmula de Bhaskara:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Fica claro que calcular a raiz quadrada é um subproblema que precisa ser resolvido para encontrar as raízes da equação, e também é evidente que a forma como computar este valor não é o objetivo da tarefa. Felizmente é possível abstrair esta utilidade (`raiz2`) e propor a seguinte solução (dados os coeficientes a, b, e c):

```
float delta = b*b - 4*a*c;
float x1 = (-b + raiz2(delta))/(2*a);
float x2 = (-b - raiz2(delta))/(2*a);
```

`raiz2` permite isolar a implementação da aplicação, e reutilizar as mesmas instruções diversas vezes, inclusive para tarefas diferentes. `raiz2` poderia ser usada para dizer a distância entre dois pontos no plano cartesiano.

3 Funções

Funções são o primeiro passo na organização do programa, dividindo um algoritmo em subalgoritmos menores e, portanto, mais fáceis de resolver. Além disso, cada subproblema é independente do outro, então as funções podem ser implementadas por programadores diferentes. A função é chamada pelo `identificador`, recebendo `argumentos` para processar (ou não), e retornando um `resultado` (ou não).

Por exemplo:

```
desligue_o_computador()
data ← que_dia_e_hoje()
resultado ← eleva_ao_cubo(2)
```

O reuso de funções implica que você só precisa definir as instruções uma vez para executar quantas quiser. Esta centralização facilita a manutenção, já que qualquer problema só precisa ser resolvido uma vez. O mesmo se aplica a inclusão de instruções, basta acrescentar uma vez que todos os algoritmos que utilizam a função serão afetados. Por exemplo, um comportamento de muita utilidade nesta disciplina é a leitura de números:

```c
int leia_inteiro() {
    int num;
    printf("\nDigite o número: ");
    scanf("%d", &num);
    return num;
}
```

A instrução `return` interrompe a execução da função imediatamente, devolvendo o valor especificado - que deve ser compatível com o tipo de retorno definido na função. Como exercício, tente gerar uma abstração para a ler um número positivo, e outra para calcular a raiz cúbica de um número real[1].

Outra aplicação muito difundida é a análise numérica, o ramo da matemática que estuda algoritmos que convergem para resultados [matematicamente válidos] de problemas [matemáticos]. Como o método babilônico para computar a raiz quadrada.

Nesta abordagem, o ideal é chegar ao valor mais próximo viável com o mínimo de iterações (de modo a minimizar o custo computacional). Uma das técnicas mais interessantes é o Método de Newton-Raphson, um algoritmo para aproximar os valores das raízes de uma função.

Dado um polinômio qualquer:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} \cdots a_1 x + a_0$$

deseja-se a raiz r tal que $f(r) = 0$. Newton provou que uma boa aproximação da raiz é a divisão do polinômio por sua derivada ${}^{f}/_{f'}$. Assim, partindo de um valor r inicial, pode-se realizar sucessivas iterações para atualizar o valor de r, aproximando-se do valor real da raiz do polinômio.

O problema torna-se então definir "quantas serão estas iterações?" Deseja-se que a resposta seja correta, mas por ser um valor aproximado - não se sabe o valor exato - é preciso definir uma margem de erro aceitável de modo que o processo possa ser interrompido se a resposta for boa o suficiente, evitando esforço desnecessário. Contudo, uma precisão excessiva pode exigir muitas iterações, portanto também é interessante limitar a quantidade de passos realizadas. O processo de calcular a raiz naturalmente se divide em subproblemas [2].

`0-Newton-Raphson.c`

```c
float Newton_Raphson(float n, int iteracoes, float precisao) {
    float r = valor_inicial(n);

    for(; iteracoes > 0; --iteracoes) {
        r = aproxima(r, n);

        if(precisao >= erro(r, n))
            break;
    }

    return r;
}
```

A aproximação é feita pela fórmula proposta, uma possível implementação é:

$$r_i = r_{i-1} - \frac{f(r)}{f'(r)}$$

Facilmente implementado por:

```c
float aproxima(float r, float n) {
    return r - f(r, n)/fp(r, n);
}
```

O erro associado a aproximação também é trivialmente obtido pela diferença entre a estimativa (função de r) e o valor correto n. Basta apenas definir o polinômio (e sua derivada) para calcular

[1] Lembre-se que, neste caso, a raiz pode ser negativa e estar no intervalo $[0, 1]$.

qualquer raiz. Por exemplo, a raiz quadrada de n seria a raiz do polinômio $f(r) = r^2 - n$ (cuja derivada é $2r$).

É interessante notar que o método é independente do polinômio, os passos descritos para inicializar, aproximar e avaliar r serão sempre os mesmos. Portanto a implementação de `Newton-Raphson` funciona para qualquer polinômio (tente implementar as funções `f` e `fp` para $\sqrt[3]{n}$ ou outro polinômio qualquer no `4-Newton-Raphson/README.md`). O mesmo pode ser dito para a definição do valor inicial de r, que pode ser lido do usuário, estimado a partir de n, ou obtido de outra forma.

Esta separação da implementação possibilita que se considere apenas como utilizar as soluções dos subproblemas. As entradas e saídas das funções podem (e devem) ser concatenadas entre si (como na filosofia Unix). Mas para uso correto, é preciso saber como se comunicar com elas (E/S) e o que elas fazem - mas não [necessariamente] como elas o fazem. Por exemplo:

`main` é a função de entrada dos programas em C e retorna uma valor inteiro que é lido pelo sistema operacional, indicando a ocorrência de erros em sua execução. Por questões históricas, valor de saída é *igual a* 0 se não houve erro (`EXIT_SUCCESS`), ou *diferente de* 0 se houve erro (geralmente o valor indica *qual* erro.)

`printf` retorna um inteiro com o número de caracteres impressos na tela.

`scanf` retorna um inteiro com o número de elementos lidos.

4 Escopos

O *escopo* é um formalimo que associa o par $< escopo, identificador >$ ao valor armazenado em memória. No *escopo local* o identificador tem significado apenas no bloco em que foi declarado, e sobrepõe-se a outro identificador igual (se houver). Já no *escopo global* o identificador tem significado em qualquer escopo (a menos que sobreposto por um identificador idêntico em um escopo local). O significado de um identificador está confinado ao escopo em que é declarado [3] e. embora muito úteis em certos casos, nesta disciplina *não usaremos escopos globais*.

`3-escopo.c`

```c
/* Troca os valores de x e y no escopo local. */
void troca(int x, int y) {
  int aux = x;
  x = y;
  y = aux;

  printf("troca: (x,y) = (%d,%d)\n", x, y);
}

int main() {
    int x = 1;
    int y = 2;

    printf("(x,y) = (%d,%d)\n", x, y);
    printf("Trocando... ");
    troca(x, y);
    printf("Trocou!\n");
    printf("(x,y) = (%d,%d)\n", x, y);

  return 0;
}
```

Cada chamada de função cria seu próprio escopo na memória, independente dos demais. Considerando o exemplo acima, há duas variáveis x, mas uma está no escopo da função `main`, e a outra no da função `troca`. Portanto o computador considera os pares $< main, x >$ e $< troca, x >$.

Considere:

4-escopo.c

```
int var_global;

/* Incrementa as variáveis "local" e "var_global". */
int incrementa(int local) {
  ++local;
  ++var_global;
  printf("incrementa: local = %d, var_global = %d\n", local, var_global);

  return local;
}

/* Duplica as variáveis "local" e "var_global". */
int duplica(int local) {
  local *= 2;
  var_global *= 2;
  printf("duplica: local = %d, var_global = %d\n", local, var_global);

  return local;
}
```

A variável *var_global* tem escopo $< \mathit{global} >$, e é acessível por todos os escopos, e há duas variáveis *local*, $< \mathit{main}, \mathit{local} >$ e $< \mathit{incrementa}, \mathit{local} >$. Tente verificar os valores mostrados a cada `printf`.

Considerando a função abaixo:

5-escopo.c

```
1  /* Limita o valor de x ao intervalo entre inf e sup.
2  Supõe inf <= sup */
3  int limita(int x, int inf, int sup) {
4    /* Retorna o maior valor entre x e y. */
5    int maior(int x, int y) {
6      return (x > y ? x : y);
7    }
8
9    /* Retorna o menor valor entre x e y. */
10   int menor(int x, int y) {
11     return (x < y ? x : y);
12   }
13
14   return maior(inf, menor(x, sup));
15 }
```

Há três variáveis x, cada uma identificada em seu próprio escopo. A $< \mathit{main}, x >$ é acessível no escopo da função `main`, definido pelos caracteres { e } (limitado as linhas 15 a 23). A $< \mathit{limita}, x >$ é acessível no escopo da função `limita` (limitado as linhas 2 a 11). A $< \mathit{limita.maior}, x >$ é acessível no escopo da função `maior` (limitado as linhas 3 a 4), e se sobrepõe a $< \mathit{limita}, x >$. O mesmo ocorre com $< \mathit{limita.menor}, x >$ (limitado as linhas 7 a 8).

A vantagem deste tipo de estruturação é que as funções auxiliares (`maior` e `menor`) ficam escondidas dentro do escopo de `limita`, e portanto não "poluem" o ambiente. O mesmo poderia ser feito com as funções auxiliares do método `Newton-Raphson`.

5 Recursividade

Suponha que o computador não tenha a primitiva $\times$ (multiplicação). Como é algo extremamente útil, seria interessante que houvesse uma função que a implementasse, já que este comportamento é facilmente obtido com laços de repetição e a primitiva +. Desta forma, pode-se definir a função iterativa que computa a multiplicação $a \times b$ por meio de adições:

7-multiplica.c

```c
/* Retorna a multiplicação
 * de a, b vezes. Assume
 * que b > 0. */
int mult(int a,int b) {
    int resultado = 0;
    while(b--)
         resultado += a;
    return resultado;
}
```

Uma análise um pouco mais detalhada evidencia que o cálculo realizado é $a \times b = a + a \times (b - 1)$, sendo que a parte $a \times (b - 1)$ é uma versão menor (e mais fácil) do mesmo problema. Mas como realizar esta nova "multiplicação"? Bom, sendo o mesmo problema, a resposta é: *da mesma forma* (chamando a função `mult` com uma pequena alteração de argumento).

O cálculo realizado neste caso é $a \times (b - 1) = a + a \times (b - 2)$, que novamente leva a versão menor (e mais fácil) do mesmo problema (percebeu uma tendência?). Para se obter o valor correto, este cálculo deve ser repetido até um ponto em que o problema se torne tão pequeno que não seja necessário calcular mais, é possivel oferecer uma resposta diretamente (o *caso base*).

Percebe-se b, cujo valor é positivo, diminui a cada passo, ou seja, vai se aproximando do valor zero. Um número a qualquer multiplicado por 0 é um caso fácil de se resolver, pois indepente do valor de a e a resposta é conhecida, portanto os cálculos podem ser interrompidos. Pode-se, enfim, reformular a abstração da seguinte forma:

8-multiplica.c

```c
/* Retorna a multiplicação
 * de a, b vezes. Assume
 * que b > 0. */
int mult(int a, int b) {
  if(b == 0)
    return 0;
  return a + mult(a, b-1);
}
```

"*Para entender recursão, você precisa entender recursão.*"

David Hunter

Recursão é o termo usado de maneira mais geral para descrever o processo de repetição de um objeto de um jeito similar ao que já fora mostrad, e muitos algoritmos têm uma estrutura recursiva [4]. Em matemática/programação, uma *função recursiva* é aquela que chama a si mesma [3].

Dada uma função de natureza recursiva, sua implementação tem dois aspectos igualmente importantes: a formulação da função [recursivamente], e seu *critério de parada*. Se a formulação for incorreta, o algoritmo perde utilidade por não produzir o resultado desejado. Se o critério de parada não for adequado, o algoritmo pode tornar-se incorreto (em termos do resultado produzido) ou mesmo nunca terminar.

Funções recursivas são ideais para certos problemas, e possibilitam soluções ditas *elegantes - programas simples que geralmente são mais confiáveis, seguros, robustos e eficientes que seus primos complexos, e muito mais fáceis de se manter* [5]. Compare as implementações de uma função que computa o fatorial de um número inteiro:

9-fatorial.c

```c
int fatorial_i(int n) {
    int fat = 1;

    while(n > 1) {
        fat = n*fat;
        n = n - 1;
    }

    return fat;
}
```

9-fatorial.c

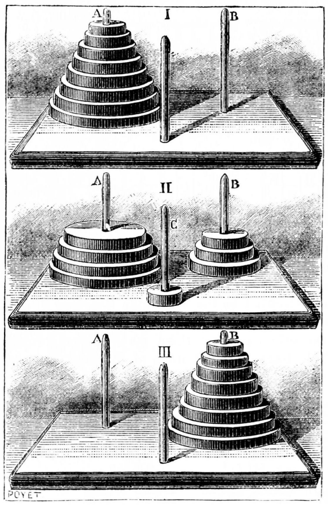
```c
int fatorial_r(int n) {
    if(n < 2)
        return 1;

    return n*fatorial_r(n-1);
```

Torres de Hanoi O deus Brahma supostamente criou uma torre com 64 discos de ouro de tamanhos distintos e outras duas estacas equilibradas sobre uma plataforma, e ordenou que movessem todos os discos da primeira estaca para a terceira. Mas:

1. apenas um disco poderia ser movido por vez;
2. cada disco só pode ser movido de uma estaca para outra;
3. nunca um disco maior deveria ficar por cima de um disco menor.

Segundo a lenda, quando todos os discos fossem transferidos, o mundo desapareceria...



Fonte: The Popular science monthly (1884)[2]

Este problema é muito interessante por dois motivos, o primeiro é sua solução extremamente elegante, o segundo são as noções envolvidas em uma análise dos custos computacionais. A solução é simples, pense bastante e tente resolvê-lo.

"*Simplicidade é a maior sofisticação.*"

Leonardo da Vinci

Qual o esforço necessário para resolver este problema? O custo de um algoritmo recursivo, como um iterativo, depende da quantidade de instruções envolvidas em cada chamada e na quantidade de chamadas recursivas. Seja T^n o número de movimentos necessários para levar n discos de uma estaca para outra, e M_{AB} o movimento de um disco da estaca A para a B.

No caso de T^0, há nenhum disco e, portanto, nenhum custo. No caso de T^1 há um disco e, portanto, um movimento (M_{AC}). No caso de T^2, é preciso tirar um disco (M_{AB}), posicionar o maior disco na estaca correta (M_{AC}), e depois mover o outro disco para esta estaca (M_{BC}) - 3 movimentos. No caso T^3, como não se pode ter um disco maior sobre um menor, é preciso tirar um disco (M_{AC}), depois o outro (M_{AB}) e posicionar o primeiro sobre o segundo (M_{CB}); então mover o maior disco para a estaca correta (M_{AC}), e depois posicionar os outros sobre ele (M_{BA}, M_{BC}, M_{AC}), resultando em 7 movimentos.

A mesma ideia se repete para $T^4, T^5, \cdots$. O importante é perceber que um mesmo procedimento simples é realizado diversas vezes. Fica claro qual o caso base aqui (T^0), e para qualquer $n > 0$, o processo para mover os discos de A para C é mover $n-1$ discos para B, mover o disco restante para C, e então novamente os $n-1$ discos de B para C. De outra forma:

$$T^0 = 0$$

$$T^1 = 1$$

[2] https://archive.org/details/popularsciencemon26newy

$$
\begin{aligned}
T^2 &= T^1 + T^1 + T^1 = 2 * T^1 + 1 = 3 \\
T^3 &= T^2 + T^1 + T^2 = 2 * T^2 + 1 = 7 \\
\ldots \\
T^n &= T^{n-1} + T^1 + T^{n-1} = 2 * T^{n-1} + 1 = 2^n - 1
\end{aligned}
$$

Assim pode-se estimar o tempo necessário para terminar a tarefa do deus Brahma com $n = 64$ (suponha que o monge nunca descansa e consegue transferir 2 discos por segundo).

6 Módulos

Praticamente todas as linguagens de programação possuem uma forma de incluir (ou importar) o conteúdo de outro(s) arquivo(s) em programa, de modo a permitir a modularização e reutilização de código. Isso possibilita a criação de bibliotecas de código, que centralizam as instruções para uma série de vantagens: simplificam a referência e manutenção do código, enquanto garantem que todos usam as mesmas instruções. Por exemplo, a biblioteca de matemática (math.h/math.py) que oferece funções matemáticas básicas como valor absoluto, seno, etc.

Em linguagem `C`, os módulos são inseridos (recursivamente) com a diretiva `#include` (indicando o nome do arquivo entre aspas), e o arquivo geralmente são nomeados `modulo.h` (claro, "modulo" é um identificador adequado para as funcionalidades oferecidas). Por exemplo, o arquivo stdio.h (*standard buffered input/output*) oferece as funcionalidades padrões de E/S de dados como `printf/scanf`.

É preciso ficar atento com a duplicação de identificadores, por exemplo se dois (ou mais) arquivos incluirem um outro mesmo arquivo. É possível evitar isso evitando o uso funcionalidades externas (por exemplo copiando os trechos desejados), mas a solução adequada é definir proteções que evitem a repetição. Isso é facilmente implementado com um teste condicional que verifica se seu módulo já foi inserido ou não. Veja `apc_subalgoritmos.c`.

Boas Práticas na codificação de funções ajudam a manter o código legível a facilitam a utilização e manutenção das funções. Esta sugestões não se aplicam a todos os sistemas (ou pessoas), mas servem de diretrizes.

Ao codificar funções, faça com que cada função realize uma tarefa [corretamente] (resolva um problema), esta modularização facilita o entendimento e a manutenção. Ao identificá-las, com nomes adequadamente descritivos (inferência direta do comportamento oferecido). Tente mantê-las *curtas*, de modo que o programador consiga visualizar todas as instruções ao mesmo tempo. Atente a indentação, funções organizadas são mais legíveis. Evite efeitos colaterais de sua utilização.

Referências

[1] John Guttag. *Introduction to computation and programming using Python.* The MIT Press, Cambridge, Massachusetts, 2013.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs.* MIT Press ; McGraw-Hill, Cambridge, Mass.; New York, 1996.

[3] Luis Joyanes Aguilar. *Fundamentos de Programação: Algoritmos, estruturas de dados e objetos.* McGraw-Hill, 3a edition, 2008.

[4] Thomas H. Cormen. *Algorithms unlocked.* The MIT Press, Cambridge, Massachusetts, 2013.

[5] Jon Louis Bentley. *Programming pearls.* Addison-Wesley, Reading, Mass, 2nd ed edition, 2000.