Estrutura de Dados II (ED2)

Aula 08 – Recursão e árvores

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (UFES)

(Material baseado nos slides do Professor Eduardo Zambon)

Temas da aula

- Recursão
  - Maior enfoque
- Árvores
  - Já vimos vários algoritmos no Laboratório 2
  - É importante entender algumas propriedades teóricas
- Um pouco sobre grafos
  - Comparação com o conceito de árvores
  - Algoritmos de Busca em Largura e Busca em Profundidade (Importantes para encontrar Componentes Conexas...)

O estudo da parte sobre **Árvores** e **Grafos** ficará como tarefa de casa.

Referências

Chapter 5 – Recursion and Trees
R. Sedgewick

Padrões de Projeto de Algoritmos

- Força Bruta
- Dividir para Conquistar
- Guloso
- Programação Dinâmica

Introdução

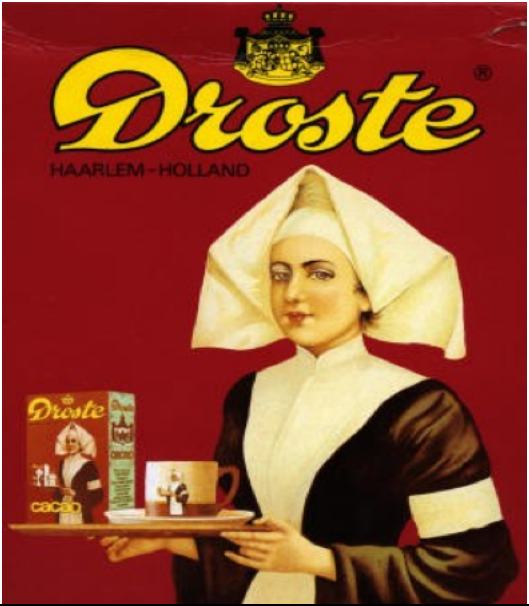
- O conceito de recursão é fundamental na matemática e na ciência da computação.

O que é recursão?

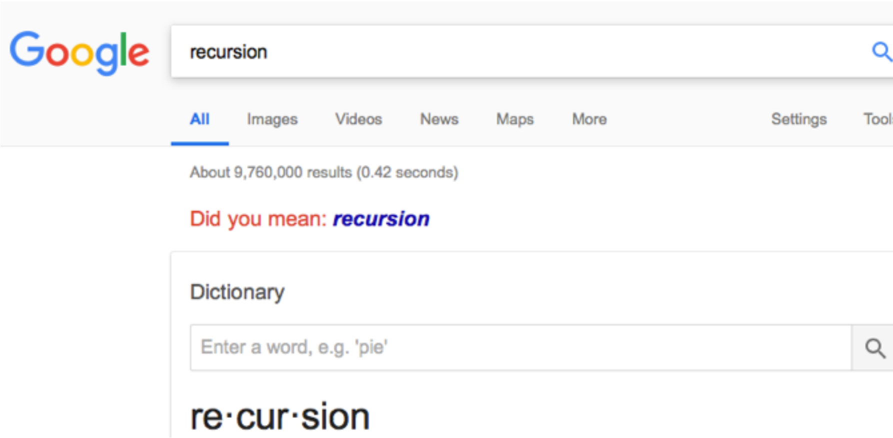
Definição recursiva

Para aprender recursão, primeiro você tem que aprender recursão!

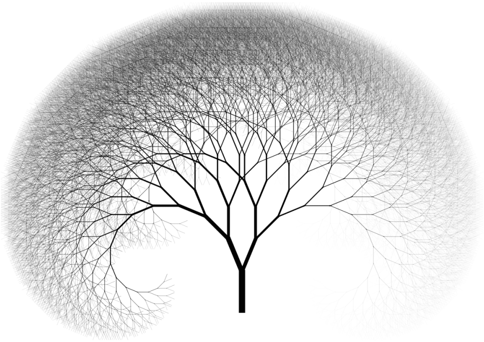
O que é recursão?



O que é recursão?



O que é recursão?



Algoritmos recursivos

- Um algoritmo recursivo resolve um problema a partir de soluções para instâncias menores do mesmo problema.
- Implementação em C: funções recursivas.

Algoritmos recursivos

- Exemplo clássico é a função fatorial, definida pela relação de recorrência:

$$N! = N \cdot (N-1)!, \quad \text{para } N \geq 1 \text{ com } 0! = 1 \quad .$$

- Implementável e utilizável para N pequeno (< 13).

```
int rec_factorial(int N) {
  if (N == 0) return 1;
  return N * rec_factorial(N-1);
}
```

- Relações de recorrência são funções definidas recursivamente.

Algoritmos recursivos

- A função recursiva do slide anterior é equivalente a um único *loop*.

```
int it_factorial(int N) {
  int t = 1;
  for (int i = 1; i <= N; i++) t *= i;
  return t;
}
```

- Sempre é possível transformar um programa recursivo em um não-recursivo que realiza a mesma computação.
- A transformação inversa também é sempre possível.

- Vamos usar constantemente recursão: permite expressar algoritmos complexos de forma compacta...
- ...sem sacrificar eficiência. (Veja os resultados do Lab. 02.)

Algoritmos recursivos – Correção

- Q: Como provar que a função fatorial recursiva está correta?
- A: Usando indução matemática:
  - Base: A função computa 0!.
  - Passo indutivo: Assumindo que a função computa $k!$ para $k < N$ (hipótese indutiva), ela computa $N!$.
- Não vamos nos preocupar com provas de correção de programas, mas buscamos ao menos uma intuição de que as tarefas serão executadas corretamente.
- Programas recursivos devem possuir duas características:
  - Resolver um caso base explicitamente.
  - Cada chamada recursiva deve envolver argumentos com valores menores.
- As condições acima garantem que uma prova indutiva pode ser construída.

Um programa recursivo questionável

O programa recursivo abaixo viola as condições anteriores.

```
int puzzle(int N) {
  printf("puzzle(%d)\n", N);
  if (N == 1) return 1;
  if (N % 2 == 0) return puzzle(N/2);
  else            return puzzle(3*N+1);
}
```

- Como há casos aonde o argumento da chamada recursiva cresce, não é possível usar indução na análise.
- Conjectura de Collatz: não se sabe se esse programa termina para todas as possíveis entradas.

```
puzzle(3)
  puzzle(10)
    puzzle(5)
      puzzle(16)
        puzzle(8)
          puzzle(4)
            puzzle(2)
              puzzle(1)
```

Exemplo das chamadas recursivas realizadas para N = 3.

Um outro programa recursivo questionável

O programa recursivo abaixo também viola as condições anteriores.

```
int dumb_factorial(int N) {
  return dumb_factorial(N + 1) / (N + 1);
}
```

- $N! = \frac{(N+1)!}{N+1}$

```
10!
    11!
        12!
            13!
                14!
                    ...
```

Problema: Máximo divisor coumum

Dados dois números inteiros não negativos a e b, qual o maior inteiro, c, que divide a e b?

- MDC - Máximo Divisor Comum; ou GCD - Greatest Common Divisor
- $\text{GCD}(10, 5) = 5$
- $\text{GCD}(7, 10) = 1$

A seguinte recorrência foi descoberta pelo matemático grego Euclides há mais de dois mil anos. Suponha $a \neq 0$

$$gcd(a, b) = \begin{cases} a, & \text{se } b = 0 \\ gcd(b, a \bmod b), & \text{se } b > 0 \end{cases}$$

Algoritmo de Euclides

```c
int gcd(int m, int n) {
  printf("gcd(%d, %d)\n", m, n);
  if (n == 0) return m;
  return gcd(n, m % n);
}
```

```
gcd(314159, 271828)
  gcd(271828, 42331)
    gcd(42331, 17842)
      gcd(17842, 6647)
        gcd(6647, 4548)
          gcd(4548, 2099)
            gcd(2099, 350)
              gcd(350, 349)
                gcd(349, 1)
                  gcd(1, 0)
```

Exemplo de uma sequência de chamadas recursivas que indica que 314159 e 271828 são relativamente primos.

A profundidade da recursão deste algoritmo é logarítmica.

Divisão e conquista

- Vários programas que vamos ver utilizam duas chamadas recursivas, cada uma operando sobre cerca de metade da entrada.
- Essa é a instância mais importante do famoso padrão de projeto de algoritmos conhecido como divisão e conquista (*divide and conquer*).
- Exemplo: para encontrar o valor máximo em um *array* de *N* inteiros, podemos usar o programa abaixo.

```
int it_max(int *a, int lo, int hi) {
  int t = a[lo];
  for (int i = lo+1; i < hi; i++)
    if (a[i] > t) t = a[i];
  return t;
}
```

- Vamos usar uma versão recursiva do programa acima para ilustrar o conceito de divisão e conquista.

Divisão e conquista

Função recursiva para encontrar o máximo em um *array*.

```
int rec_max(int *a, int lo, int hi) {
  int m = (lo + hi) / 2; //Danger of overflow if array is large.
  if (lo == hi) return a[lo];
  int u = rec_max(a, lo, m);
  int v = rec_max(a, m+1, hi);
  if (u > v) return u; else return v;
}
```

- Q: Por que sabemos que essa função está correta?
- A: Porque as chamadas recursivas satisfazem os requisitos necessários (caso base, e argumentos recursivos menores), logo é possível usar indução para provar o seu funcionamento.

- Q: Como analisar o desempenho dessa função?
- A: Usando uma relação de recorrência.

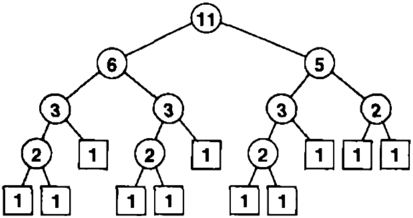
Encontrar o máximo – Relação de recorrência

A quantidade de chamadas recursivas A_N da função `rec_max` para uma entrada de tamanho N é dada pela relação de recorrência:

$$A_N = A_k + A_{N-k} + 1, \quad \text{para } N \geq 1 \text{ com } A_1 = 0,$$

onde k é o tamanho de um dos lados da divisão.

- A solução da relação acima é $A_N = N - 1$.
- $\Rightarrow$ A profundidade da recursão é sempre menor que N.
- A árvore abaixo indica como a função recursiva divide uma entrada de tamanho 11.



Busca binária

A busca binária é um algoritmo de divisão e conquista que quebra o problema ao meio, e trabalha somente sobre uma das metades.
Abaixo, a versão iterativa do algoritmo.

```
int bin_search(int *a, int sz, int key) {
  int lo = 0, hi = sz-1;
  while (lo <= hi) {
    int mid = lo + ((hi - lo) / 2);
    if      (key < a[mid]) hi = mid - 1;
    else if (key > a[mid]) lo = mid + 1;
    else return mid;
  }
  return -1;
}
```

Busca binária

A busca binária também pode ser implementar de forma recursiva.

```
int rec_bin_serach(int *a, int lo, int hi, int key) {
   if (hi >= lo) {
        int mid = lo + (hi - lo) / 2;
        if (a[mid] == key)
            return mid;
        if (a[mid] > key)
            return rec_bin_search(a, lo, mid - 1, key);
        return rec_bin_search(a, mid + 1, hi, key);
   }
   return -1;
}
```

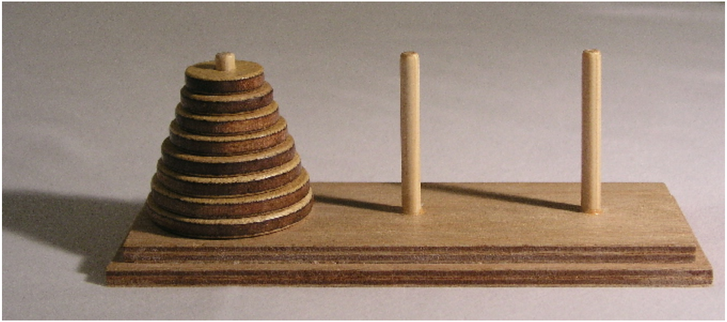
Busca binária – Relação de recorrência

A quantidade de comparações C_N da função `bin_search` para uma entrada de tamanho N é dada pela relação de recorrência:

$$C_N = C_{N/2} + 1,$$

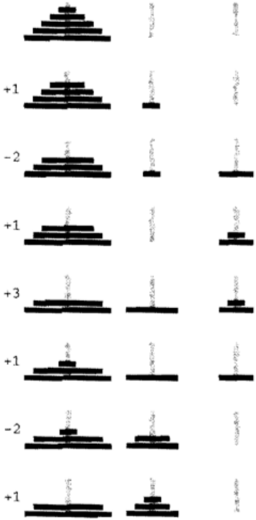
cuja solução aproximada é $C_N \sim \lg N$.

Torre de Hanoi



- Do menor para o maior, discos estão numerados de 1 a N;
- Queremos mover os N discos para o pino do meio;
- Só podemos mover um disco por vez e NUNCA colocar um disco maior sobre um menor.

Torre de Hanoi – Exemplo



Torre de Hanoi – Solução

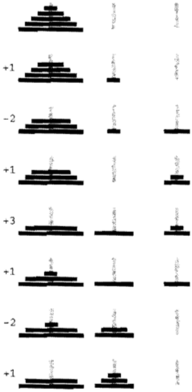
O problema clássico das Torres de Hanoi pode ser resolvido facilmente por um programa recursivo de divisão e conquista.

```
void hanoi(int N, int d) {
  if (N == 0) return;
  hanoi(N-1, -d);
  shift(N, d);
  hanoi(N-1, -d);
}
```

- `hanoi(N, 1)`: mover *N* discos do pino 1 para o pino do meio;
- `shift(K, d)`: move disco *K* uma posição para direita se $d > 0$ ou uma posição para a esquerda se $d < 0$.

Torre de Hanoi – Solução + Exemplo

```
void hanoi(int N, int d) {
  if (N == 0) return;
  hanoi(N-1, -d);
  shift(N, d);
  hanoi(N-1, -d);
}
```




```
hanoi(3, +1)
  hanoi(2, -1)
    hanoi(1, +1)
      hanoi(0, -1)
      shift(1, +1)
      hanoi(0, -1)
    shift(2, -1)
    hanoi(1, +1)
      hanoi(0, -1)
      shift(1, +1)
      hanoi(0, -1)
  shift(3, +1)
  hanoi(2, -1)
    hanoi(1, +1)
      hanoi(0, -1)
      shift(1, +1)
      hanoi(0, -1)
    shift(2, -1)
    hanoi(1, +1)
      hanoi(0, -1)
      shift(1, +1)
      hanoi(0, -1)
```

Torres de Hanoi – Relação de recorrência

O problema clássico das Torres de Hanoi pode ser resolvido facilmente por um programa recursivo de divisão e conquista.

```
void hanoi(int N, int d) {
  if (N == 0) return;
  hanoi(N-1, -d);
  shift(N, d);
  hanoi(N-1, -d);
}
```

A quantidade movimentos T_N para uma entrada com N discos é dada pela relação de recorrência:

$$T_N = 2T_{N-1} + 1 \quad \text{para } N \geq 2 \text{ com } T_1 = 1,$$

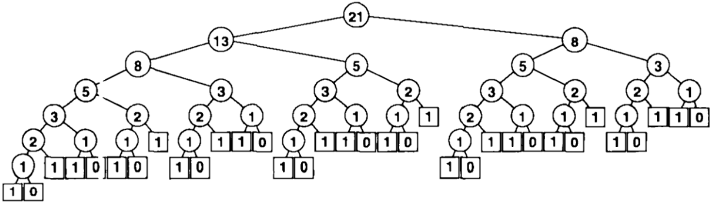
cuja solução é $T_N = 2^N - 1$.

Partindo da lenda aonde $N = 64$ e assumindo que os monges levam 1 segundo por movimento, a execução total levará aproximadamente 585 bilhões de anos.

Programação dinâmica – Fibonacci

- Característica essencial até agora: particionar o problema em subproblemas independentes.
- Quando isso não acontece mesmo os algoritmos mais simples podem ficar incrivelmente ineficientes!
- O programa abaixo tem complexidade ϕ^N, onde $\phi \approx 1.618$ é a proporção áurea. A seguir: árvore para $F_8 = 21$.

```
int rec_fib(int N) {
  if (N == 0) return 0;
  if (N == 1) return 1;
  return rec_fib(N-1) + rec_fib(N-2);
}
```



Programação dinâmica – Fibonacci

- Por outro lado, o programa equivalente abaixo tem complexidade linear em relação à entrada N.

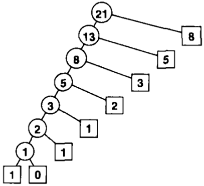
```
int it_fib(int N) {
  int F[47]; // Max Fib num as a 32-bit int.
  F[0] = 0; F[1] = 1;
  for (int i = 2; i <= N; i++)
    F[i] = F[i-1] + F[i-2];
  return F[N];
}
```

- Os números de Fibonacci crescem exponencialmente.
- O *array* precisa de no máximo 47 posições, já que $F_{46} = 1836311903$ é o maior número que pode ser armazenado em um inteiro de 32-bits (com sinal).
- Método exemplificado acima chama-se programação dinâmica (*dynamic programming*).
- Em particular: *bottom-up dynamic programming*. *Loop* que usa valores menores para calcular valores maiores.

Programação dinâmica – Fibonacci

- *Top-down dynamic programming*: técnica que permite executar funções recursivas com um custo igual ou melhor que na versão iterativa *bottom-up*.
- Também chamado de memoização (*memoization*).
- Último passo armazena o valor computado pela função em um vetor externo.

```
int knownF[] = {[0 ... 46] = -1};
int dyn_fib(int N) {
  if (knownF[N] != -1) return knownF[N];
  int t;
  if (N == 0) t = 0;
  if (N == 1) t = 1;
  if (N > 1) t = dyn_fib(N-1) + dyn_fib(N-2);
  return knownF[N] = t;
}
```



Veja também no livro: problema da mochila (*knapsack*).