Algoritmos e Estruturas de Dados 1 (AED1)

Recursão, exponencial, Fibonacci, análise de desempenho

Estrutura geral de um programa recursivo

se a instância em questão é pequena,
    resolva-a diretamente;
senão
    reduza-a a instâncias menores do mesmo problema,
    aplique o método a essas
    e use suas soluções para resolver a instância original.

Exponencial

Queremos projetar um algoritmo que recebe inteiros positivos k e n e devolve k^n.

Como k^n = k * k * k * … * k (n vezes)

- podemos projetar o seguinte algoritmo iterativo:

```
int expI(int k, int n)
{
   int i, exp = 1;
   for (i = 0; i < n; i++)
       exp *= k; // exp = exp * k
   return exp;
}
```

Invariante e corretude:

- A corretude do algoritmo iterativo expI depende
  - da correta identificação de seu invariante de laço.
- O invariante principal,
  - que vale no início de cada iteração do laço,
    - é exp = k^i.
- Para verificar que o invariante está correto,
  - observe que ele vale antes da primeira iteração
  - e continua valendo de uma iteração para outra.
- Demonstração por indução:
  - Primeiro verificamos que o invariante vale
    - no início da primeira iteração (caso base).
      - Antes da primeira iteração temos exp = 1 e i = 0.
        - Como exp = 1 = k^0 = k^i, o invariante vale.
  - Então, supomos que o invariante vale
    - no início de uma iteração qualquer, ou seja, exp = k^i,
      - e mostramos que no início da iteração seguinte
        - ele continua valendo.

- Chamamos exp' e i' as variáveis no início da iteração seguinte.
  - Pelo comportamento do algoritmo dentro do laço, temos
    - exp' = exp * k = k^i * k = k^(i + 1)
    - i' = i + 1
- Portanto, exp' = k^(i + 1) = k^i', e o invariante vale.

- Para verificar que o algoritmo está correto,
  - note que ao final do laço i = n,
    - caso em que o invariante garante que exp = k^n.

Eficiência de tempo:

- Número de operações proporcional a n, i.e., O(n),
  - já que o laço tem n iterações
  - e em cada iteração é realizado um número constante de operações.

Eficiência de espaço:

- Memória auxiliar utilizada é constante, i.e., O(1),
  - pois o número de variáveis locais independe de n.

Também podemos descrever k^n usando a seguinte regra recursiva:
k^n = { k * k^(n - 1), se n > 0,
1, se n = 0. }

Essa regra induz o seguinte algoritmo recursivo:

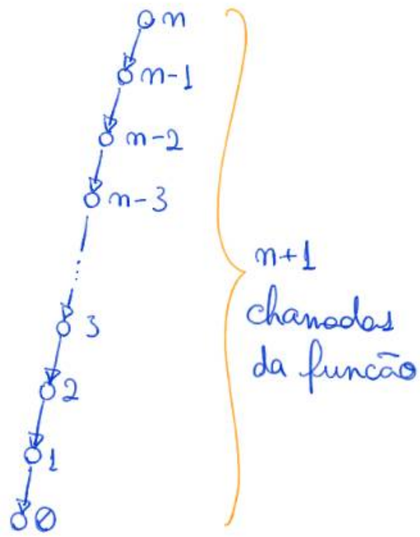
```
int expR(int k, int n)
{
   if (n == 0)
       return 1;
   return k * expR(k, n - 1);
}
```

Corretude:

- No caso de expR, como é comum com algoritmos recursivos,
  - sua corretude deriva diretamente da implementação
    - da regra em que este se baseia,
  - podendo ser verificada usando uma prova por indução.

Eficiência de tempo:

- O cálculo da eficiência de um algoritmo recursivo depende
  - da resolução de uma função de recorrência que captura
    - a ordem do número de operações que tal algoritmo realiza.
- No caso do algoritmo expR temos a recorrência
  - T(n) = T(n - 1) + 1, já que cada chamada da função
    - desencadeia apenas uma outra chamada,
      - com n decrementado de 1,
    - e realiza uma quantidade constante de operações localmente.
  - T(0) = 1, pois o caso base da função recursiva ocorre quando n = 0.



- Identificada a recorrência, podemos resolvê-la por substituição:
    - $T(n) = T(n - 1) + 1$
    - $T(n - 1) = T(n - 2) + 1$
    - $T(n - 2) = T(n - 3) + 1$
    - $T(n - 3) = T(n - 4) + 1$
    - …
- Substituindo
    - $T(n) = T(n - 1) + 1$
    - $T(n) = (T(n - 2) + 1) + 1 = T(n - 2) + 2$
    - $T(n) = (T(n - 3) + 2) + 1 = T(n - 3) + 3$
    - $T(n) = (T(n - 4) + 3) + 1 = T(n - 4) + 4$
    - …
- Generalizando
    - $T(n) = T(n - i) + i$
- No final (caso base da recursão) temos
    - $n - i = 0 \Rightarrow i = n$
- Portanto,
    - $T(n) = T(n - n) + n = T(0) + n = n + 1$.
- Ou seja, o número total de operações é da ordem de n,
    - i.e., O(n).

Eficiência de espaço:
- A quantidade de memória auxiliar utilizada é igual à eficiência de tempo,
    - i.e., O(n).
- Isso acontece porque cada nova chamada recursiva
    - utiliza algumas variáveis auxiliares locais,
        - que só começam a ser liberadas
            - depois que a última chamada é resolvida.

Nossos dois algoritmos levam tempo proporcional a n.
- **Será que conseguimos fazer melhor?**

Pensando no caso especial em que n é múltiplo de 2,
- podemos calcular k^n com a seguinte regra:
    - k^n = k^(n / 2) * k^(n / 2)

Note que, os dois termos multiplicados são iguais,

- o que sugere um algoritmo recursivo em que cada chamada da função
    - realiza apenas uma chamada recursiva, com n dividido por 2,
        - e depois multiplica o resultado desta chamada por ele próprio.
- Essa ideia parece interessante, pois n está diminuindo mais rapidamente
    - do que nos nossos algoritmos anteriores.

Claro que, o algoritmo tem que tomar algum cuidado quando n não é par.

- Lembrando que a operação n / 2 corresponde ao piso da divisão,
    - para n ímpar temos a regra:
        - k^n = k^(n / 2) * k^(n / 2) * k

Por fim, o caso base continua ocorrendo quando n = 0.

- Assim, temos o algoritmo:

```
int expR2(int k, int n) {
   int exp;
   if (n == 0)
       return 1;
   if (n % 2 == 0) {
       exp = expR2(k, n / 2);
       exp *= exp;
   }
   else { // n % 2 == 1
       exp = expR2(k, n / 2);
       exp *= exp;
       exp *= k;
   }
   return exp;
}
```

Como a parte inicial das regras é comum,

- podemos implementar a seguinte versão mais curta do algoritmo:

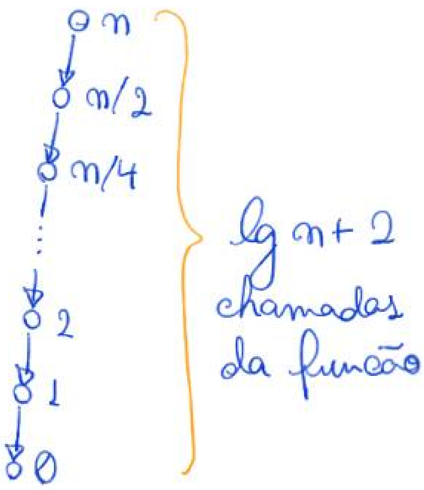
```
int expR3(int k, int n) {
   int exp;
   if (n == 0)
       return 1;
   exp = expR3(k, n / 2);
   exp *= exp;
   if (n % 2 == 0)
       return exp;
   return k * exp;
}
```

Corretude:

- A corretude deste algoritmo deriva diretamente
    - da corretude da regra que o motivou.

Eficiência de tempo:

- A eficiência de tempo deste algoritmo guarda uma grata surpresa,
    - que deriva do fato dele dividir n por 2 a cada chamada recursiva.



- Para calcular a ordem do número de operações que ele realiza
    - vamos usar a recorrência:
        - T(n) = T(n / 2) + 1,
    - já que cada chamada da função desencadeia apenas
        - uma outra chamada, com n dividido por 2.
    - T(0) = 1, pois o caso base da função recursiva ocorre quando n = 0.
- Resolução da recorrência por substituição:
    - T(n) = T(n / 2) + 1
    - T(n / 2) = T(n / 4) + 1
    - T(n / 4) = T(n / 8) + 1
    - T(n / 8) = T(n / 16) + 1
    - T(n / 16) = T(n / 32) + 1
    - …
- Substituindo
    - T(n) = T(n / 2) + 1
    - T(n) = (T(n / 4) + 1) + 1 = T(n / 4) + 2
    - T(n) = (T(n / 8) + 1) + 2 = T(n / 8) + 3
    - T(n) = (T(n / 16) + 1) + 3 = T(n / 16) + 4
    - T(n) = (T(n / 32) + 1) + 4 = T(n / 32) + 5
    - ...
- Sucessivas divisões por 2 nos sugerem ficar atentos
    - ao aparecimento de funções logarítmicas e exponenciais.
- Observe que
    - T(n) = T(n / 2) + 1 = T(n / 2^1) + 1
    - T(n) = T(n / 4) + 2 = T(n / 2^2) + 2
    - T(n) = T(n / 8) + 3 = T(n / 2^3) + 3
    - T(n) = T(n / 16) + 4 = T(n / 2^4) + 4
    - T(n) = T(n / 32) + 5 = T(n/2^5) + 5
    - …

- Generalizando
  - T(n) = T(n / 2^i) + i
- No final (caso base da recursão) temos n / 2^i = 0,
  - o que pode causar estranheza, já que
    - o resultado de uma divisão não deveria ser 0.
  - Isso se explica por se tratar de uma divisão inteira,
    - que devolve o piso do resultado da divisão.
  - Assim, conseguimos inferir que se n / 2^i = 0,
    - então n / 2^(i - 1) = 1,
      - já que a divisão inteira de 1 por 2 é 0.
  - Resolvendo n / 2^(i - 1) = 1 ⇒ i = lg n + 1.
- Portanto,
  - T(n) = T(n / 2^(lg n + 1)) + (lg n + 1)
  - T(n) = T(n / 2n) + lg n + 1
  - T(n) = T(0) + lg n + 1
  - T(n) = 1 + lg n + 1
  - T(n) = lg n + 2.
- Ou seja, o número de operações realizadas por expR3(n)
  - é da ordem de lg n, i.e., O(log n).
- Note que, este algoritmo é muito mais eficiente
  - que os anteriores para valores grandes de n.

Eficiência de espaço:
- A quantidade de memória auxiliar utilizada é igual à eficiência de tempo,
  - i.e., O(log n).
- Isso acontece porque cada nova chamada recursiva
  - utiliza algumas variáveis auxiliares locais,
    - que só começam a ser liberadas
      - depois que a última chamada é resolvida.

Vale destacar que esse algoritmo é um exemplo de uso
- da **técnica de divisão e conquista**, que é largamente usada
  - para obter algoritmos mais eficientes para diversos problemas.

Desafio/Quiz: Por fim, fica o seguinte exercício:
- projetar um algoritmo iterativo
  - que seja igualmente eficiente para o cálculo da exponencial.

Comentários sobre corretude e eficiência de algoritmos

Corretude de algoritmos:

- Em algoritmos recursivos costuma ser uma análise direta, bastando:
  - Mostrar que o algoritmo devolve o valor correto no caso base e,
  - Supondo que o algoritmo encontra o valor correto quando a entrada é menor que n, mostrar que ele devolve o valor correto para n.
- Em algoritmos iterativos é necessário identificar propriedades invariantes:
  - Um invariante é uma relação que depende dos valores das variáveis do algoritmo e se mantém válida ao longo das iterações de um laço.
  - Encontrando os invariantes corretos, a análise de corretude é semelhante a de algoritmos recursivos.

Eficiência de algoritmos:

- Em algoritmos iterativos costuma ser bem direta,
  - bastando somar operações realizadas dentro de um laço e
    - multiplicar o resultado pelo número de iterações do laço.
- Em algoritmos recursivos é necessário usar uma fórmula de recorrência:
  - Por exemplo, a recorrência T(n) = T(n - 1) + 1 e T(0) = 1, captura
    - a ordem de operações realizada por um algoritmo que, em cada chamada recursiva, realiza um número constante de operações e faz uma chamada recursiva com a entrada reduzida de um,
    - e que faz um número constante de operações no caso base.
  - Outro exemplo, a recorrência T(n) = 2 T(n / 2) + 1 e T(1) = 1, captura
    - a eficiência de um algoritmo que, em cada chamada recursiva, realiza um número constante de operações e faz duas chamadas recursivas com a entrada reduzida pela metade,
    - e que faz um número constante de operações no caso base.
- De posse da recorrência, pode-se simular seu comportamento, para
  - encontrar uma função (n, n^2, log n, 2^n, etc) que descreva o comportamento da mesma conforme o tamanho da entrada n varia.
  - Esta função descreve a ordem do número de operações realizada pelo algoritmo recursivo.

(Bônus) Soluções para o exercício/quiz deixado no final:

```
int expI2(int k, int n)
{
   if (n == 0)
       return 1;
   // invariante: exp = k^i
   int i = 1, exp = k;
   while (i <= n / 2)
   {
       exp *= exp; // exp = exp * exp
       i = 2 * i;
   }
   if (i < n)
       exp *= expI2(k, n - i);
   return exp;
}

int expI3(int k, int n)
{
   int exp_acum = 1, n_falt = n;
   while (n_falt > 0)
   {
       // invariante: exp = k^i
       int i = 1, exp = k;
       while (i <= n_falt / 2)
       {
           exp *= exp; // exp = exp * exp
           i = 2 * i;
       }

       exp_acum *= exp; // exp_acum = exp_acum * exp;
       n_falt -= i;     // n_falt = n_falt - i;
   }
   return exp_acum;
}
```

Fibonacci

Números de Fibonacci:

- F0 = 0
- F1 = 1
- Fn = Fn-1 + Fn-2

Sequência:

- n   0 1 2 3 4 5 6 7 8 9
- Fn   0 1 1 2 3 5 8 13 21 34

Algoritmo recursivo para Fn

```
long long int fibonacciR(int n) {
   if (n == 0) return 0;
   if (n == 1) return 1;
   return fibonacciR(n - 1) + fibonacciR(n - 2);
}
```

Algoritmo iterativo para Fn

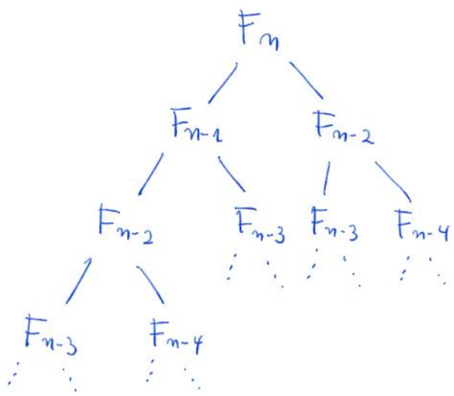
```
long long int fibonacciI(int n) {
   int i;
   long long int proximo, anterior, atual;
   if (n == 0) return 0;
   if (n == 1) return 1;
   anterior = 0; // Fi-1
   atual = 1;    // Fi
   for (i = 1; i < n; i++) {
       proximo = anterior + atual;
       anterior = atual;
       atual = proximo;
   }
   return atual;
}
```

Corretude:

- Como de costume, a corretude do algoritmo recursivo deriva
    - diretamente da corretude da relação de recorrência que o inspirou.
- Sobre o algoritmo iterativo, qual o invariante principal
    - para demonstrar que ele obteve o resultado correto?
- Ou seja, que propriedade/relação se mantém verdadeira
    - ao longo de todas as suas iterações?
- Resp: no início de cada iteração i as variáveis atual e anterior
    - possuem, respectivamente, os valores Fi e Fi-1.

Eficiência:

- Como de costume, a eficiência do algoritmo iterativo
  - deriva do número de vezes que o laço é executado,
    - e neste caso é da ordem de n.
- Já o algoritmo recursivo depende do número total de chamadas recursivas.
  - Vamos obter intuição deste número usando uma árvore de recorrência.



- Note que, ela lembra a árvore com crescimento exponencial base 2,
  - mas desbalanceada para um lado.
- Observe também o grande número de subproblemas recalculados,
  - sugerindo ineficiência.
- Para obter a ordem do número de chamadas recursivas,
  - vamos analisar a seguinte recorrência, que descreve tal número?
    - T(n) = T(n - 1) + T(n - 2) + 2 para n > 1, T(0) = T(1) = 0.
  - Um limitante inferior para T(n)
    - T(n) = T(n - 1) + T(n - 2) + 2 >= 2 T(n - 2) + 2.
    - Assim,

      T(n) = 2 T(n - 2) + 2
      T(n - 2) = 2 T(n - 4) + 2
      T(n - 4) = 2 T(n - 6) + 2
      T(n - 6) = 2 T(n - 8) + 2
    - Logo,

      T(n) = 2 T(n - 2) + 2
      = 2 (2 T(n - 4) + 2) + 2 = 4 T(n - 4) + 6
      = 4 (2 T(n - 6) + 2) + 6 = 8 T(n - 6) + 14
      = 8 (2 T(n - 6) + 2) + 14 = 16 T(n - 8) + 30
    - Observando o padrão,

      T(n) = 2^1 T(n - 2) + 2^2 - 2
      = 2^2 T(n - 4) + 2^3 - 2
      = 2^3 T(n - 6) + 2^4 - 2
      = 2^4 T(n - 8) + 2^5 - 2
    - Chegamos a,

      T(n) = 2^i T(n - 2i) + 2^(i + 1) - 2
    - Escolhendo i = n / 2,

      T(n) = 2^(n / 2) T(n - n) + 2^((n / 2)+1) - 2
      = 2^(n / 2 + 1) - 2
  - Portanto, o número de chamadas recursivas cresce
    - pelo menos como uma exponencial de base 2 e expoente n/2.