AED1 - Aula 03
Recursão, máximo, binomial, análise de desempenho

Estrutura geral de um programa recursivo

se a instância em questão é pequena,
    resolva-a diretamente;
senão
    reduza-a a uma instância menor do mesmo problema,
    aplique o método à instância menor
    e volte à instância original.

Problema do máximo

Definição:
- Dado um vetor v de inteiros com tamanho n,
  - devolva o valor do maior elemento deste vetor.

Ideia de uma abordagem recursiva:
- Tome um elemento arbitrário do vetor.
- Encontre recursivamente o máximo do subproblema
  - que contém os demais elementos.
- Compare o elemento tomado com o máximo do subproblema
  - e devolva o maior deles.
- Observe que se o subproblema tem apenas um elemento,
  - então ele pode ser resolvido diretamente.

Algoritmo recursivo que reduz o vetor pelo fim.

```
int maximoRend(int v[], int n)
{
   int x; // auxiliar que guarda o máximo do subproblema
   if (n == 1)
       return v[0];
   x = maximoRend(v, n - 1);
   if (x > v[n - 1])
       return x;
   return v[n - 1];
}
```

Algoritmo recursivo que reduz o vetor pelo início.

```
int maximoRbegin(int v[], int inicio, int n)
{
   int x; // auxiliar que guarda o máximo do subproblema
   if (inicio == n - 1)
       return v[inicio];
   x = maximoRbegin(v, inicio + 1, n);
   if (x > v[inicio])
       return x;
   return v[inicio];
}

int maximo(int v[], int n)
{
   return maximoRbegin(v, 0, n);
}
```

Eficiência de tempo:

- Qual a ordem do número de operações dos algoritmos recursivos?
    - Da ordem de n, i.e., O(n), pois cada chamada da função
        - realiza um número constante de operações locais
        - e desencadeia no máximo uma chamada recursiva
            - na qual o tamanho do subvetor
                - é reduzido em uma unidade.

Eficiência de espaço:

- Qual a quantidade de memória auxiliar utilizada?
    - Nesse caso 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.

Ideia de uma abordagem iterativa:

- Percorra o vetor da esquerda para a direita,
    - mantendo numa variável auxiliar
        - o maior valor do subvetor já percorrido.

Algoritmo iterativo.

```c
int maximoI(int v[], int n)
{
   int max, i;
   max = v[0];
   for (i = 1; i < n; i++)
      if (max < v[i])
         max = v[i];
   return max;
}
```

Invariante e corretude:

- Qual o invariante principal do algoritmo iterativo?
  - No início de cada iteração temos que
    - max armazena o maior valor do subvetor v[0 .. i - 1].
- Observe que, ao final do laço, quando i = n,
  - o invariante garante que max é o maior elemento do vetor.

Eficiência de tempo:

- Qual o número de iterações em função de n?
  - É igual a n, pois o laço começa com i = 1 e vai até i = n - 1.
- Como o número de operações realizadas em cada iteração é constante,
  - o número total de operações é proporcional a n, i.e., O(n).

Eficiência de espaço:

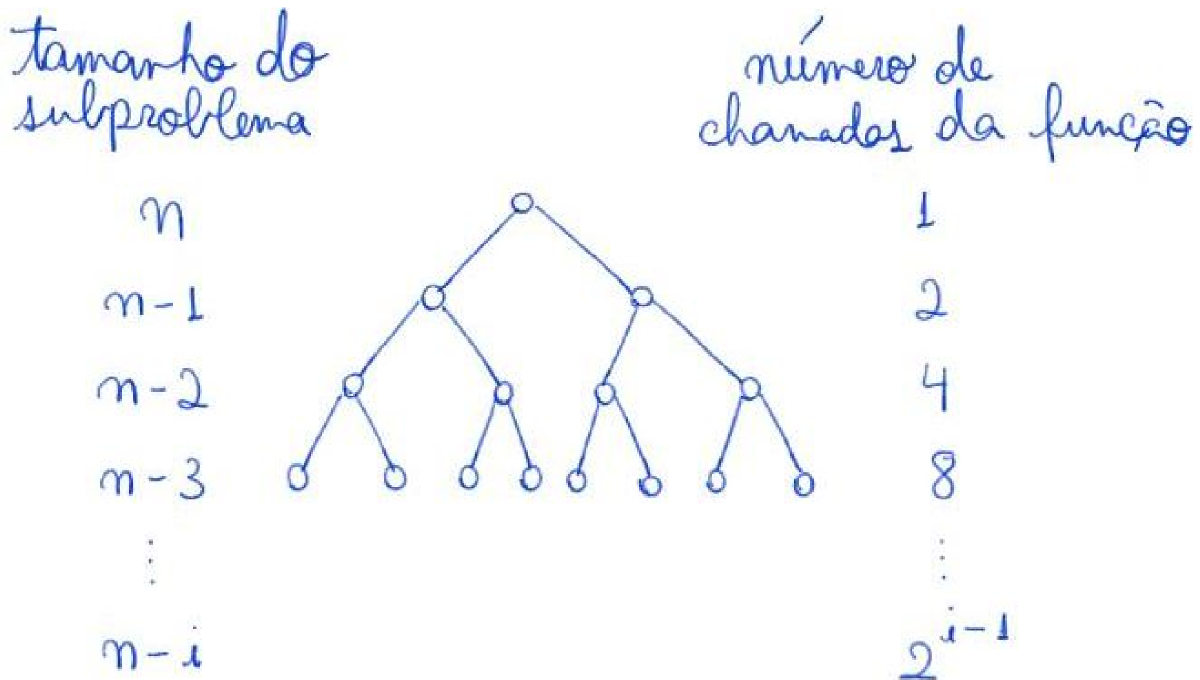
- Qual a quantidade de memória auxiliar usada?
  - O(1), pois a função possui um pequeno número de variáveis simples.

Quiz: qual a eficiência de pior caso do seguinte algoritmo recursivo?

```c
int maximoR(int v[], int n)
{
   int x;
   if (n == 1)
      return v[0];
   if (maximoR(v, n - 1) > v[n - 1])
      return maximoR(v, n - 1);
   return v[n - 1];
}
```

- No pior caso ele leva tempo O(2^n), pois cada chamada da função

    - pode dar origem a duas chamadas recursivas.



- Isso torna esse algoritmo muito ineficiente, embora
    - ele esteja correto e, conceitualmente, seja muito parecido
        - com o primeiro algoritmo recursivo que estudamos.

Coeficientes binomiais e o triângulo de Pascal

Coeficientes binomiais são definidos como

- (n k) = (n escolhe k) = n! / (n - k)! k!

e nos ajudam a responder à pergunta:

- de quantas maneiras podemos escolher k itens dentre n?

Relação de coeficientes binomiais com contagem de combinações:

- pense que tem n interruptores para acender k lâmpadas
- e quer saber de quantas maneiras diferentes você pode acendê-las,
    - ou seja, quantos subconjuntos distintos de tamanho k existem?

Regra de Pascal:
(n k) = { 0, se n = 0 e k > 0,
1, se n >= 0 e k = 0,
(n-1 k) + (n-1 k-1), se n > 0 e k > 0. }

Interpretação da regra de Pascal:

- se n = 0 e k > 0 temos que acender mais lâmpadas
    - do que temos interruptores,

            - então não existe qualquer maneira de acendê-las
- se k = 0 não queremos qualquer lâmpada acesa.
    - Assim, todos os interruptores devem estar desligados,
        - qualquer que seja seu número.
- se n > 0 e k > 0 então podemos considerar o último interruptor.
    - Se escolhermos deixá-lo desligado então
        - teremos de acender todas as k lâmpadas usando os n - 1 interruptores restantes, e existem (n-1 k) maneiras de fazer isso.
    - Se escolhermos ligar o último interruptor então
        - teremos de acender apenas k - 1 lâmpadas restantes com n - 1 interruptores restantes, e existem (n-1 k-1) maneiras de fazê-lo.
    - Como queremos o total de possibilidades,
        - somamos as opções com o último interruptor desligado e ligado.

Preenchimento do triângulo de Pascal numa tabela:

n/k	0	1	2	3	4	5	6	7
0	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0
2	1	2	1	0	0	0	0	0
3	1	3	3	1	0	0	0	0
4	1	4	6	4	1	0	0	0
5	1	5	10	10	5	1	0	0

Triângulo de Pascal (na forma tradicional):

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1

Relação da contagem de combinações com as binomiais,
- que deram nome aos coeficientes:
    - (a + b)^n = 1(a^n) + ?(a^n-1)(b^1) + ... + ?(a^1)(b^n-1) + 1(b^n).
- Expandindo (a + b)^n temos
    - (a + b) * (a + b) * (a + b) * … * (a + b)
- Pense que cada (a + b) corresponde a um interruptor
    - que pode ficar ligado (escolher a) ou desligado (escolher b).

Algoritmo recursivo que implementa a regra de Pascal.

```
long long int binomialR0(int n, int k)
{
   if (n == 0 && k > 0)
       return 0;
   if (n >= 0 && k == 0)
       return 1;
   return binomialR0(n - 1, k) + binomialR0(n - 1, k - 1);
}
```

Algoritmo iterativo que preenche a tabela (triângulo de Pascal).

```
long long int binomialI(int n, int k)
{
   int i, j;
   long long int bin[100][100];
   for (j = 1; j <= k; j++)
       bin[0][j] = 0;
   for (i = 0; i <= n; i++)
       bin[i][0] = 1;
   for (i = 1; i <= n; i++)
       for (j = 1; j <= k; j++)
           bin[i][j] = bin[i - 1][j] + bin[i - 1][j - 1];
   return bin[n][k];
}
```

Comparação da ordem do número de operações entre os algoritmos:

Algoritmo iterativo realiza da ordem de n * k operações, i.e., O(nk),

- por conta dos dois laços aninhados,
  - um variando i de 1 até n
  - e o outro variando j de 1 até k.

Algoritmo recursivo é mais difícil de analisar,

- pois seu tempo segue uma função de recorrência do tipo
  - $T(n, k) = T(n - 1, k) + T(n - 1, k - 1) + 1$, para $n > 0$ e $k > 0$,
    - ou seja, o tempo para calcular (n escolhe k)
      - depende do tempo para calcular (n-1 k) e (n-1 k-1),
        - mais algum trabalho local.

- Voltaremos para essa análise, mas por hora vale notar que
    - a função recursiva recalcula várias vezes os mesmos subproblemas
        - binomialR0(3, 2)
            - binomialR0(2, 2)
                - binomialR0(1, 2)
                    - binomialR0(0, 2)
                    - binomialR0(0, 1)
                - binomialR0(1, 1)
                    - binomialR0(0, 1)
                    - binomialR0(0, 0)
            - binomialR0(2, 1)
                - binomialR0(1, 1)
                    - binomialR0(0, 1)
                    - binomialR0(0, 0)
                - binomialR0(1, 0)

Regra de Pascal com melhores condições de contorno ($n < k$, $n = k$ ou $k = 0$).

(n k) = { 0, se $n < k$,
1, se $n = k$ ou $k = 0$,
(n-1 k) + (n-1 k-1), se $n > k > 0$. }

Interpretação das mudanças na regra de Pascal:

- se $n < k$ temos que acender mais lâmpadas
    - do que temos interruptores,
        - então não existe qualquer maneira de acendê-las
- se $n = k$ queremos todas as lâmpadas acesas.
    - Assim, todos os interruptores devem estar ligados,
        - qualquer que seja seu número.

Algoritmo recursivo melhorado.

```
long long int binomialR1(int n, int k)
{
   if (n < k)
       return 0;
   if (n == k || k == 0)
       return 1;
   return binomialR1(n - 1, k) + binomialR1(n - 1, k - 1);
}
```

Nova comparação de ordem do número de operações:

- Será que o novo algoritmo ainda resolve várias vezes o mesmo problema?
    - binomialR1(3, 2)
        - binomialR1(2, 2)
        - binomialR1(2, 1)
            - binomialR1(1, 1)
            - binomialR1(1, 0)
- A princípio pode parecer que não, mas de fato ainda o faz.
    - binomialR1(4, 2)
        - binomialR1(3, 2)
            - binomialR1(2, 2)
            - binomialR1(2, 1)
                - binomialR1(1, 1)
                - binomialR1(1, 0)
        - binomialR1(3, 1)
            - binomialR1(2, 1)
                - binomialR1(1, 1)
                - binomialR1(1, 0)
            - binomialR1(2, 0)
- Então vamos analisar a recorrência T(n, k),
    - que descreve o número de adições realizadas
        - ao longo das chamadas de binomialR1.
            - $T(n, k) = T(n - 1, k) + T(n - 1, k - 1) + 1$, para $n > k > 0$
            - $T(n, k) = 0$, para $n < k$
            - $T(n, k) = 0$, para $n = k$ ou $k = 0$
- Note que, o número de chamadas da função binomialR1
    - é igual ao dobro do número de adições,
        - e que o trabalho local realizado por cada chamada
            - à binomialR1 é constante.
- Assim, T(n, k) nos dá a ordem do número de operações total.
    - Vamos resolver a recorrência usando uma tabela.

Tabela preenchida com T(n, k):

n/k	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0
3	0	2	2	0	0	0	0	0
4	0	3	5	3	0	0	0	0
5	0	4	9	9	4	0	0	0

Tabela preenchida com (n escolhe k):

n/k	0	1	2	3	4	5	6	7
0	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0
2	1	2	1	0	0	0	0	0
3	1	3	3	1	0	0	0	0
4	1	4	6	4	1	0	0	0
5	1	5	10	10	5	1	0	0

- podemos observar que seu valor corresponde a (n escolhe k) - 1 = (n k) - 1.

Podemos demonstrar que (n k) >= 2^n/2

- quando k é próximo de n / 2, fazendo:
    - (n n/2) = n! / [(n / 2)! (n / 2)!]
        - >= [n * (n - 1) * … * (n / 2 + 1)] / [(n / 2) * (n / 2 - 1) * … * 1]
        - >= 2^(n/2)

Como o número de chamadas recursivas feitas por binomialR1 é:

- 2 * (n 2) - 2 >= 2 * 2^(n/2) - 2

temos que a mesma leva tempo exponencial.

Note que, binomialR0 faz mais chamadas recursivas que binomialR1.

- Por isso, o resultado anterior também é um limitante inferior
    - para o número de chamadas que esta realiza.

Binomal mais eficiente:

(n k) = n! / [(n - k)! k!]
= [n * (n - 1)!] / [(n - k)! * k * (k - 1)!]
= (n / k) * (n - 1)! / [(n - k)! (k - 1)!]
= (n / k) * (n - 1)! / [((n - 1) - (k - 1))! (k - 1)!]
= (n / k) * (n-1 k-1)

Regra mais eficiente:

(n k) = { n, se k = 1,
(n / k) * (n - 1 k - 1), se k > 1. }

Algoritmo recursivo baseado na nova regra.

```
double binomialR2(long n, long k)
{
   if (k == 1)
       return (double)n;
   return binomialR2(n - 1, k - 1) * (double)n / (double)k;
}
```

Note a diferença na cadeia de chamadas recursivas:

- binomialR2(10, 6)
    - binomialR2(9, 5)
        - binomialR2(8, 4)
            - binomialR2(7, 3)
                - binomialR2(6, 2)
                    - binomialR2(5, 1)

Qual a ordem do número de operações deste último algoritmo?

- É da ordem de k, pois segue a recorrência:
  - $T(n, k) = T(n - 1, k - 1) + 1$, para $n > k > 1$
  - $T(n, k) = 1$, para $k = 1$
- Resolvendo a recorrência por substituição
  - $T(n, k) = T(n - 1, k - 1) + 1$
  - $T(n - 1, k - 1) = T(n - 2, k - 2) + 1$
  - $T(n - 2, k - 2) = T(n - 3, k - 3) + 1$
  - $T(n - 3, k - 3) = T(n - 4, k - 4) + 1$
  - …
- Substituindo
  - $T(n, k) = T(n - 1, k - 1) + 1$
  - $T(n, k) = (T(n - 2, k - 2) + 1) + 1 = T(n - 2, k - 2) + 2$
  - $T(n, k) = (T(n - 3, k - 3) + 1) + 2 = T(n - 3, k - 3) + 3$
  - $T(n, k) = (T(n - 4, k - 4) + 1) + 3 = T(n - 4, k - 4) + 4$
  - …
- Generalizando
  - $T(n, k) = T(n - i, k - i) + i$
- No final (caso base da recursão) temos
  - $k - i = 1 \Rightarrow i = k - 1$
- Portanto,
  - $T(n, k) = T(n - (k - 1), k - (k - 1)) + (k - 1)$
  - $T(n, k) = T(n - k + 1, 1) + (k - 1) = 1 + k - 1 = k$
- Ou seja, o número de operações realizadas por binomialR2(n, k)
  - é da ordem de k.

Vale notar que este último algoritmo pode sofrer com erros de precisão numérica,

- já que utiliza divisões reais.