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

Versão recursiva que reduz o vetor pelo fim.

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

Versão recursiva que reduz o vetor pelo início.

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

int maximoRbegin(int v[], int i, int n)
{
   int x;
   if (i == n - 1)
       return v[i];
   x = maximoRbegin(v, i + 1, n);
   if (x > v[i])
       return x;
   return v[i];
}
```

Versão iterativa.

```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;
}
```

Qual o invariante mais importante do algoritmo iterativo?

Comparação da ordem do número de operações entre as versões.

Versão ineficiente por fazer duas chamadas recursivas (quiz).

```
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];
}
```

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:

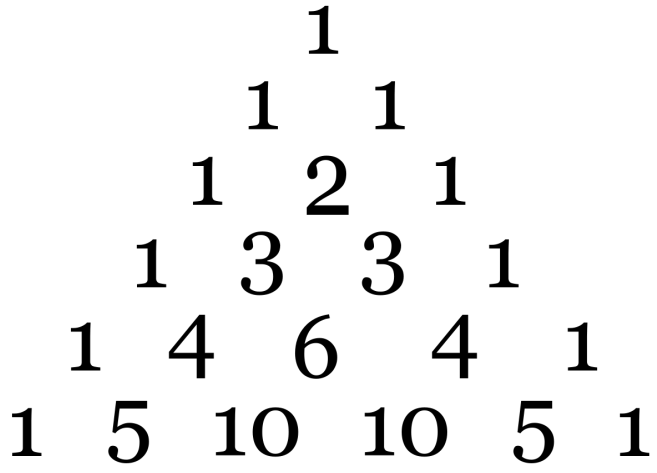
- se n = 0 e k > 0 temos que acender mais lâmpadas do que temos interruptores, então não existe qualquer maneira de acendê-las

- 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 então teremos de acender apenas k-1 lâmpadas restantes com n-1 interruptores restantes, e existem (n-1 k-1) maneiras de fazer isso.
    - 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):



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)
- sendo que cada (a + b) corresponde a um interruptor que pode ficar ligado (escolher a) ou desligado (escolher b).

Apresentação do algoritmo recursivo que implementa a regra de Pascal.

```
long 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);
}
```

Apresentação do algoritmo iterativo que preenche a tabela (triângulo de Pascal).

```
long binomialI(int n, int k)
{
   int i, j, 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 as versões:

- versão iterativa realiza da ordem de n * k operações.
- versão recursiva 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
- 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. }

Apresentação do algoritmo recursivo melhorado.

```
long 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 ainda resolver várias vezes o mesmo subproblema?
    - 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 (mostrar usando árvore de recursão)
    - 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
- Resolvendo a recorrência usando uma tabela:

Preenchendo a seguinte tabela com os valores de 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

podemos observar que seu valor corresponde a (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, então o resultado 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. }

```
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
    - T(n, 1) = 1
- Resolução da recorrência por substituição:
  T(n, k) = T(n-1, k-1) + 1

  (cálculos auxiliares)
  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

  (desenvolvendo)
  T(n, k) = T(n-1, k-1) + 1
  = (T(n-2, k-2) + 1) + 1 = T(n-2, k-2) + 2
  = (T(n-3, k-3) + 1) + 2 = T(n-3, k-3) + 3
  = (T(n-4, k-4) + 1) + 3 = T(n-4, k-4) + 4
  …
  (generalizando a regra)
  T(n, k) = T(n-i, k-i) + i

  (como T(n, 1) = 1, para fazer k-i = 1 escolhemos i = k-1)
  T(n, k) = T(n-(k-1), k-(k-1)) + (k-1)
  = T(n-k+1, 1) + k-1
  = 1 + k-1
  = k

  (portanto, o número de chamadas recursivas e o número de operações realizadas por binomialR2(n, k) é da ordem de k).