Algoritmos e Estruturas de Dados 1 (AED1)
Recursão, binomial, 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.

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,
  - cada interruptor acende uma lâmpada a mais,
- e queremos saber de quantas maneiras diferentes podemos 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
    - temos 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
    - temos 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),
- enquanto o expoente k do termo ?(a^k)(b^n-k) diz quantas lâmpadas ligadas.

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):									Tabela preenchida com (n escolhe k):								
n/k	0	1	2	3	4	5	6	7	n/k	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	2	1	2	1	0	0	0	0	0
3	0	2	2	0	0	0	0	0	3	1	3	3	1	0	0	0	0
4	0	3	5	3	0	0	0	0	4	1	4	6	4	1	0	0	0
5	0	4	9	9	4	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.

```
long long int binomialR2(int n, int k)
{
    if (k == 1)
        return n;
    return binomialR2(n - 1, k - 1) * n / 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 recursivo pode sofrer

  - com erros de precisão numérica em função da divisão n / k.
- De fato, se fizermos a divisão real n / k e multiplicarmos a razão obtida
  - pelo resultado da chamada recursiva, teremos erros de precisão.
- No entanto, se formos cuidadosos com a ordem dos cálculos
  - na última linha do algoritmo, podemos trabalhar com divisões inteiras
    - e ter certeza de que não ocorrerá imprecisão.
- Quiz1: Diga qual a ordem correta dos termos multiplicados na última linha
    - e explique porque nenhuma divisão deixará resto.
  - Dica: compare cada termo no denominador com o total no numerador.
- Quiz2: Faça uma versão iterativa do último algoritmo recursivo,
  - tomando cuidado com a ordem em que o laço deve ser percorrido.