Estratégias para a competição

Início da prova

1. Um procura no inicio, outro no meio e um no final
2. Quem achar a questão mais fácil começa no computador
3. É preferível demorar alguns minutos a tomar uma penalidade

Durante a prova

1. Ao ler um problema, **destaque** as partes mais importantes
2. Fique de olho nos *clarifications*
3. Determinar um tempo máx. que uma pessoa pode usar o PC
4. Caso o computador esteja ocupado, escreva no papel

Discussão de problemas

1. Apresentar uma solução para quem não pensou uma solução mata a criatividade
2. Discutir uma solução com alguém que pensou em outra solução faz com que cada um aponte defeitos na solução do colega, de modo que se cheguem em uma solução ótima
3. Pensem no menor caso de teste/menor resposta possível

Testando

1. Gaste algum tempo testando o seu programa
2. Testar os *casos de borda*
3. Definir um limite de tempo para os testes
4. Procure exceções (números negativos, impares, pares, 0, ...)

Submissão

1. **Sempre** faça a impressão do código logo após a submissão, *submit and print*

Wrong Answer

1. **NÃO ENTRE EM PÂNICO**
2. Analise o código impresso no papel
3. Leia o enunciado novamente, preste nos detalhes, limites e *overflows*
4. Utilize o python para gerar entradas aleatórias para o problema
5. Veja na tabela quem já leu o problema e descreva o algoritmo para a pessoa (pato)
    - Obs.: **NÃO** faça isso sem que a pessoa pense em uma solução por si só
6. Use o teste de mesa, ele funciona ;)

Limites Big O

STL

Funções úteis do C++

GCD (Greatest common divisor):

Maior divisor comum

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}

// OR

__gcd(a, b)
```

LCM (Least Common Multiple):

MMC, menor múltiplo comum

```
// Recursive function to return gcd of a and b
long long gcd(long long int a, long long int b)
{
  if (b == 0)
    return a;
  return gcd(b, a % b);
}

// Function to return LCM of two numbers
long long lcm(int a, int b)
{
    return (a / gcd(a, b)) * b;
}
```

Conversão de tipos

1. stoi: **string** to **int**
2. stol: **string** to **long**
3. stoll: **string** to **long long**
4. stod: **string** to **double**
5. to_string: **number** to **string**

Produto dos i-th fatoriais

```
// To compute (a * b) % MOD
long long int mulmod(long long int a, long long int b,
                                    long long int mod)
{
    long long int res = 0; // Initialize result
    a = a % mod;
    while (b > 0) {

        // If b is odd, add 'a' to result
        if (b % 2 == 1)
            res = (res + a) % mod;
```

```
        // Multiply 'a' with 2
        a = (a * 2) % mod;

        // Divide b by 2
        b /= 2;
    }

    // Return result
    return res % mod;
}

// This function computes factorials and
// product by using above function i.e.
// modular multiplication
long long int findProduct(long long int N)
{
    // Initialize product and fact with 1
    long long int product = 1, fact = 1;
    long long int MOD = 1e9 + 7;
    for (int i = 1; i <= N; i++) {

        // ith factorial
        fact = mulmod(fact, i, MOD);

        // product of first i factorials
        product = mulmod(product, fact, MOD);

        // If at any iteration, product becomes
        // divisible by MOD, simply return 0;
        if (product == 0)
            return 0;
    }
    return product;
}

N = 5;
cout << findProduct(N) << endl; // 34560
```

Josephus

There are N people standing in a circle waiting to be executed. The counting out begins at some point in the circle and proceeds around the circle in a fixed direction. In each step, a certain number of people are skipped and the next person is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last person remains, who is given freedom.

Given the total number of persons N and a number k which indicates that k-1 persons are skipped and the kth person is killed in a circle. The task is to choose the person in the initial circle that survives.

```
int Josephus(int N, int k)
{

    // Initialize variables i and ans with 1 and 0
    // respectively.

    int i = 1, ans = 0;

    // Run a while loop till i <= N
```

```
    while (i <= N) {

        // Update the Value of ans and Increment i by 1
        ans = (ans + k) % i;
        i++;
    }

    // Return required answer
    return ans + 1;
}

int N = 14, k = 2;
cout << Josephus(N, k) << endl; // 14
```

Números Primos

Verificar se N é primo

Time complexity: O(sqrt(N))

```
bool is_prime(int n) {
    // Assumes that n is a positive natural number
    // We know 1 is not a prime number
    if (n == 1) {
        return false;
    }

    int i = 2;
    // This will loop from 2 to int(sqrt(x))
    while (i*i <= n) {
        // Check if i divides x without leaving a remainder
        if (n % i == 0) {
            // This means that n has a factor in between 2 and sqrt(n)
            // So it is not a prime number
            return false;
        }
        i += 1;
    }
    // If we did not find any factor in the above loop,
    // then n is a prime number
    return true;
}
```

Sieve of Eratosthenes

Given a number n, print all primes smaller than or equal to n. It is also given that n is a small number.

```
#include <bitset>
#include <iostream>
using namespace std;
bitset<500001> Primes;
void SieveOfEratosthenes(int n)
{
    Primes[0] = 1;
    for (int i = 3; i*i <= n; i += 2) {
```

```
        if (Primes[i / 2] == 0) {
            for (int j = 3 * i; j <= n; j += 2 * i)
                Primes[j / 2] = 1;
        }
    }
}
int main()
{
    int n = 100;
    SieveOfEratosthenes(n);
    for (int i = 1; i <= n; i++) {
        if (i == 2)
            cout << i << ' ';
        else if (i % 2 == 1 && Primes[i / 2] == 0)
            cout << i << ' ';
    }
    // 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
    return 0;
}
```

Binomial Coefficient

A binomial coefficient C(n, k) also gives the number of ways, disregarding order, that k objects can be chosen from among n objects more formally, the number of k-element subsets (or k-combinations) of a n-element set.

O coeficiente binomial, também chamado de número binomial, de um número n, na classe k, consiste no número de combinações de n termos, k a k.

```
int binomialCoeff(int n, int r)
{

    if (r > n)
        return 0;
    long long int m = 1000000007;
    long long int inv[r + 1] = { 0 };
    inv[0] = 1;
    if(r+1>=2)
    inv[1] = 1;

    // Getting the modular inversion
    // for all the numbers
    // from 2 to r with respect to m
    // here m = 1000000007
    for (int i = 2; i <= r; i++) {
        inv[i] = m - (m / i) * inv[m % i] % m;
    }

    int ans = 1;

    // for 1/(r!) part
    for (int i = 2; i <= r; i++) {
        ans = ((ans % m) * (inv[i] % m)) % m;
    }

    // for (n)*(n-1)*(n-2)*...*(n-r+1) part
    for (int i = n; i >= (n - r + 1); i--) {
```

```
        ans = ((ans % m) * (i % m)) % m;
    }
    return ans;
}
int n = 5, r = 2;
cout << "Value of C(" << n << ", " << r << ") is "
<< binomialCoeff(n, r) << endl; // Value of C(5, 2) is 10
```

Conversão de bases numéricas

Qualquer base -> decimal

```
string base2 = "1100";
string base8 = "21";
string base10 = "25";
string base16 = "1E";

cout << (stoi(base2, nullptr, 2)) << endl; // 12
cout << (stoi(base8, nullptr, 8)) << endl; // 17
cout << (stoi(base10, nullptr, 10)) << endl; // 25
cout << (stoi(base16, nullptr, 16)) << endl; // 30
```

Decimal -> Qualquer base

```
// To return char for a value. For example '2'
// is returned for 2. 'A' is returned for 10. 'B'
// for 11
char reVal(int num)
{
    if (num >= 0 && num <= 9)
        return (char)(num + '0');
    else
        return (char)(num - 10 + 'A');
}

// Function to convert a given decimal number
// to a base 'base' and
string fromDeci(string& res, int base, int inputNum)
{
    int index = 0; // Initialize index of result

    // Convert input number is given base by repeatedly
    // dividing it by base and taking remainder
    while (inputNum > 0) {
        res.push_back(reVal(inputNum % base));
        index++;
        inputNum /= base;
    }

    // Reverse the result
    reverse(res.begin(), res.end());

    return res;
}
int inputNum = 282, base = 16; string res;
```

```
cout << "Equivalent of " << inputNum << " in base "
    << base << " is " << fromDeci(res, base, inputNum)
    << endl; //Equivalent of 282 in base 16 is 11A
```

Partição de um número

Given a positive integer n, generate all possible unique ways to represent n as sum of positive integers.

Time Complexity: O(2^n)

```
class Solution {
public:
    vector<int> temp;
    void solve(vector<int> a, vector<vector<int> >& v,
               int idx, int sum, int n)
    {
        // first base case if sum=n we can store vector in a
        // vector
        if (sum == n) {
            v.push_back(temp);
            return;
        }
        // if idx < 0 return
        if (idx < 0) {
            return;
        }
        // not take condition
        solve(a, v, idx - 1, sum, n);
        if (sum < n) {
            temp.push_back(a[idx]);
            // this is main condition where we can take one
            // element many times
            solve(a, v, idx, sum + a[idx], n);
            temp.pop_back();
        }
    }
    vector<vector<int> > UniquePartitions(int n)
    {
        vector<int> a;
        // vector to store elements from 1 to n
        for (int i = 1; i <= n; i++) {
            a.push_back(i);
        }
        vector<vector<int> > v;
        // call solve to get answer
        solve(a, v, n - 1, 0, n);
        reverse(v.begin(), v.end());
        return v;
    }
};
// using
vector<vector<int> > ans = ob.UniquePartitions(4);
cout << "for 4\n";
for (auto i : ans) {
    for (auto j : i) {
        cout << j << " ";
```

```
    }
    cout << "\n";
}
```

Força Bruta e Backtracking

Backtracking é um refinamento do algoritmo de busca por força bruta, no qual boa parte das soluções podem ser eliminadas sem serem explicitamente examinadas. A ideia central é retroceder quando detectar que a solução candidata é inviável

Exemplo, labirinto:

```
int T;
int maze[5][5];
bool vis[5][5];

map<char, pii> movimento = {
  {'D', {-1, 0}},
  {'E', {1, 0}},
  {'B', {0, 1}},
  {'C', {0, -1}},
};
vector<char> movimentos_possiveis = {'B', 'C', 'D', 'E'};
bool ganhou = false;

bool deslocamento_possivel(int x, int y, char caminho) {
  x += movimento[caminho].f;
  y += movimento[caminho].s;

  if(x >= 0 && x < 5 && y >= 0 && y < 5 && maze[x][y] != 1 && vis[x][y] == 0) return true;
  return false;
}

void backtracking(int x, int y) {
  if(ganhou) return;

  vis[x][y] = true;
  if(x == 4 && y == 4) {
    ganhou = true;
    return;
  }

  for(auto c : movimentos_possiveis) {
    if(!deslocamento_possivel(x, y, c)) continue;
    backtracking(x + movimento[c].f, y + movimento[c].s);
  }
}

// ...

backtracking(0, 0);
```

Dado um tabuleiro de xadrez $n \; x \; n$ e uma posição (x, y) do tabuleiro, queremos encontrar um passeio de um cavalo que visite cada casa exatamente uma vez.

- Movimento do cavalo – formato de L:
  - dois quadrados horizontalmente e um verticalmente, ou
  - dois quadrados verticalmente e um horizontalmente.

```
int m[MAX][MAX], n;
vector<pii> movimentos = {{2, -1}, {2, 1}, {-2, 1}, {-2, -1}, {1, 2}, {-1, 2}, {-1, -2}, {1, -2}};

bool posicaoValida(int x, int y){
  return (x >= 0) && (x < n) && (y >= 0) && (y < n) && !m[x][y];
}

int passeioCavalo(int x, int y)
{
  if (m[x][y] == n * n)
    return 1;
  for (auto mov : movimentos)
  {
    int x2 = x + mov.first;
    int y2 = y + mov.second;
    if (posicaoValida(x2, y2))
    {
      m[x2][y2] = m[x][y] + 1;
      if (passeioCavalo(x2, y2))
        return 1;
      m[x2][y2] = 0;
    }
  }
  return 0;
}
```

Busca Binária

Para aplicar um algoritmo de busca binária preciso de:

- Uma estrutura de dados ordenada.
- Acessar qualquer elemento dessa estrutura com a compl exidade contante.

```
#include <bits/stdc++.h>
using namespace std;

// An iterative binary search function.
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // If we reach here, then element was not present
    return -1;
}
```

Funções

- **binary_search(first, last, val)**
  - Retorna um booleano indicando se existe o elemento
- **lower_bound(first, last, val)**
  - Retorna iterator para o **primeiro** valor **não-inferior** a **val**
  - Ou retorna last, caso não encontre **val**
- **upper_bound(first, last, val)**
  - Retorna iterator para o **primeiro** valor **superior** a **val**
  - Ou retorna last, caso não encontre **val**

Método da bissetriz

Exemplo de cálculo de raiz quadrada:

```
double raiz(double x, double eps = 1e-3)
{
  double l = 0, r = x;
  while (r - l > eps) {
    double m = (l + r) / 2.0;
```

```
    cout << m << endl;

    if (m * m < x)
      l = m;
    else
      r = m;
  }

  return (l + r) / 2.0;
}
```

> *sqrt(2) = 1.41421 1 1.5 1.25 1.375 1.4375 1.40625 1.42188 1.41406 1.41797 1.41602 1.41504 1.41455 raiz(2) = 1.41455*

```
ll n;
ll k;
vector<ld> acumuladores(1e4+1);

bool eh_possivel(ld max_energia) {
  ld doadores = 0.0, receptores = 0.0;
  for(ll i = 0; i < n; i++) {
    if(acumuladores[i] > max_energia) doadores += abs(acumuladores[i] - max_energia);

    if(acumuladores[i] < max_energia) receptores += abs(max_energia - acumuladores[i]);
  }

  doadores -= (doadores * k) / 100.0;
  return doadores >= receptores;
}

int main()
{
  speed;
  cin >> n >> k;
  for(ll i = 0; i < n; i++) cin >> acumuladores[i];

  ld l = 0.0, r = 1e12, eps = 1e-6, ans;
  while((r - l) > eps) {
    ld mid = (l + r) / 2.0;

    if(eh_possivel(mid)) {
      ans = mid;
      l = mid;
    } else r = mid;
  }

  cout << fixed << setprecision(9) << (l + r) / 2.0 << endl;

  return 0;
}
```

```
ll N, A;
vll tiras(1e5+1);

ld area_corte(ld altura) {
  ld total = 0.0;
```

```
  for(ll i = 0; i < N; i++) {
    if(tiras[i] <= altura) continue;

    total += ((ld)tiras[i] - altura);
  }

  return total;
}

int main()
{
  speed;
  while(cin >> N >> A) {
    if(N == 0 && A == 0) break;

    for(ll i = 0; i < N; i++) cin >> tiras[i];

    ll max_area = 0;
    for(ll i = 0; i < N; i++) {
      max_area += tiras[i];
    }

    if(max_area < A) {
      cout << "-.-" << endl;
      continue;
    }
    if(max_area == A) {
      cout << ":D" << endl;
      continue;
    }

    ld l = 0, r = (ld)max_area;
    while((r - l) > 1e-6) {
      ld mid = (l + r) / 2.0;

      if(area_corte(mid) > A) {
        l = mid;
      } else {
        r = mid;
      }
    }

    cout << fixed << setprecision(4) << ((l + r) / 2.0) << endl;
  }



  return 0;
}
```

Busca binária na resposta

```
vll pipocas;
ll competidores, tempo, qtd;
```

```
bool eh_possivel(ll chute) {
  ll competidor_atual = 1, resta = chute * tempo;
  for(ll i = 0; i < sz(pipocas); i++) {
    if(resta >= pipocas[i]) resta -= pipocas[i];
    else {
      competidor_atual++;
      resta = chute * tempo;
      i--;
    }
    if(competidor_atual > competidores) return false;
  }

  return true;
}

int main()
{
  speed;
  cin >> qtd >> competidores >> tempo;
  pipocas.assign(qtd, 0);

  for(ll i = 0; i < qtd; i++) cin >> pipocas[i];

  ll l = 0, r = 1e9+1;
  while(l < r) {
    ll m = (l + r) / 2;

    if(!eh_possivel(m)) l = m + 1;
    else r = m;
  }

  cout << l << endl;

  return 0;
}
```

Guloso

Strings

Matemática

Formulas matemáticas, volume, área, perímetro e etc.

Formulas Gerais

Progressão Aritmética

Fórmula do Termo Geral: $an = a1 + (n - 1) \times r$

Soma dos termos da PA: $Sn = n \times (a1 + an) / 2$

Progressão Geométrica

Fórmula do Termo Geral: $an = a1 \times q\text{^}(n-1)$

Soma dos termos da PG: $Sn = a1 \times (q\text{^}n - 1) / (q - 1)$

Número de áreas em um plano divididas por retas e suas intersecções

Fórmula: $A = N + I + 1$

Onde N é o número de retas e I é o número de intersecções. Cada reta horizontal tem uma intersecção com uma reta vertical, então sempre existem pelo menos $v \times h$ intersecções, onde v é o número de retas verticais e h horizontais.

Números Triangulares

Um número triangular é um número natural representado na forma de um triângulo equilátero. O n-ésimo número triangular pode ser visto como o número de pontos de uma forma triangular com lado formado por n pontos, o que equivale à soma dos primeiros n números naturais.

Em geral, o n-ésimo número triangular é dado por: $Tn = \Sigma[k=1 \text{ to } n]\ k = 1 + 2 + 3 + \ldots + (n - 2) + (n - 1) + n = n(n + 1) / 2$

A soma dos primeiros n números triangulares é o n-ésimo número tetraédrico, que tem a fórmula: $n(n + 1)(n + 2) / 6$

Raízes triangulares e teste de identificação (número de linhas triangulares que podem ser formadas com n elementos): $n = \sqrt{}(8x + 1 - 1) / 2$

Múltiplos positivos de k num intervalo

O número de múltiplos positivos m(k) de k no intervalo [1,N] é igual a $m(k) = N / k$.

Número par ou ímpar de divisores

Números que são quadrados perfeitos têm um número ímpar de divisores, enquanto os outros têm um número par.

Número de quadrados perfeitos de A a B

N = floor(sqrt(B)) - ceil(sqrt(A)) + 1

Quadrados e retângulos em um Grid de N lados com K dimensões

Quadrados: $NK + (N - 1)K + (N - 2)K$ até 1

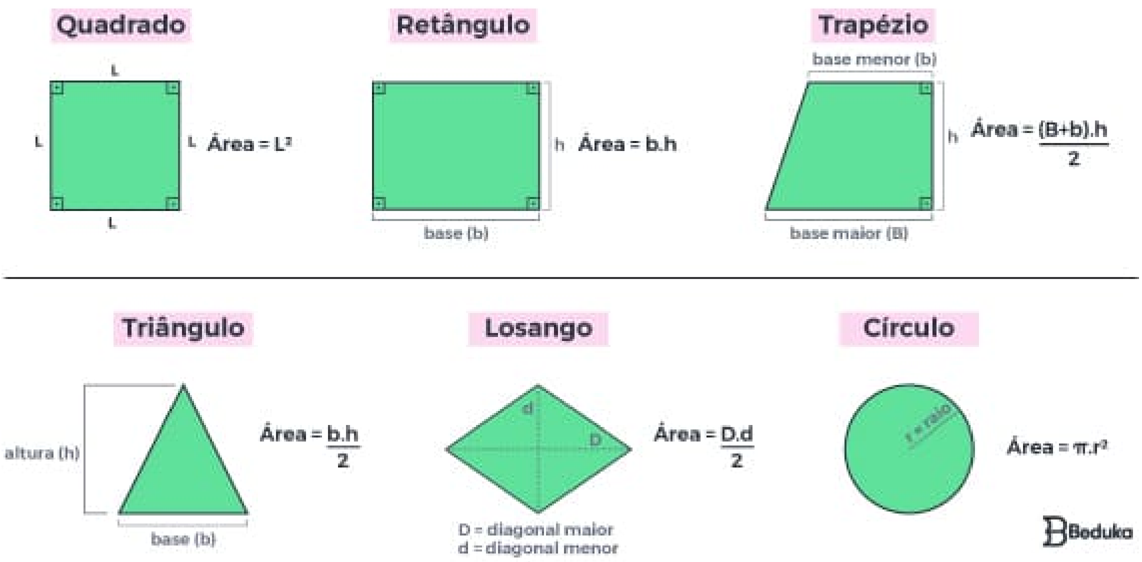
Retângulos: (NK(N+1)K) / 2 - Quadrados*

Número de pares que podem ser formados combinando N elementos

$$P = (n \times (n - 1)) / 2$$

Geometria 2D

Formulas matemáticas de figuras em 2D.



- **Retângulo**:

```
A (retângulo) = b . h
```

- **Quadrado:**

```
A (quadrado) = l²
```

- **Trapézios:** podem ser **divididos em triângulos e retângulos**, então basta guardar essas duas fórmulas, calcular e somar. **Porém, existe uma fórmula própria** dos trapézios que envolve a **base maior (B**) e **base menor (b)**:

```
A (trapézio) = (B + b) h / 2
```

- **Losango:** também **pode ser dividido em triângulos**, então basta calcular eles e somar. **Porém, existe uma fórmula própria** para losangos com base em sua **diagonal maior (D**) e **diagonal menor (d)**:

```
 A = D . d / 2
```

- **Triângulos:** também é dada pela multiplicação de área por altura, mas o valor é dividido na metade **porque o triângulo vai "afunilando"**:

```
 A (triângulo) = b . h / 2** ou **A (t. equilátero) = √3 . l² / 4)
```

- **Circunferência**:

```
A (círculo) = π.r²
```

Figuras

Quadrado

Área de um Quadrado (A): A área de um quadrado pode ser calculada multiplicando o comprimento de um dos lados pelo próprio lado:

```
A = Lado × Lado = Lado²
```

Perímetro de um Quadrado (P): O perímetro de um quadrado é a soma dos comprimentos dos quatro lados:

```
P = 4 × Lado
```

Comprimento da Diagonal (d): A diagonal de um quadrado divide o quadrado em dois triângulos retângulos congruentes. O comprimento da diagonal pode ser calculado usando o teorema de Pitágoras, onde "Lado" é o comprimento dos lados do quadrado:

```
d = Lado × √2
```

Raio da Circunferência Inscrita (r): A circunferência inscrita é uma circunferência que toca os quatro lados do quadrado. O raio dessa circunferência pode ser calculado como metade do lado do quadrado:

```
r = Lado / 2
```

Raio da Circunferência Circunscrita (R): A circunferência circunscrita é uma circunferência que passa pelos quatro vértices do quadrado. O raio dessa circunferência é igual à metade da diagonal do quadrado:

```
R = (Lado × √2) / 2
```

Área do Quadrado em Função da Diagonal (d): A área do quadrado também pode ser expressa em termos do comprimento da diagonal:

```
A = (d²) / 2
```

Triângulo

Claro! Aqui estão as fórmulas relacionadas ao triângulo formatadas em Markdown para você copiar e colar:

Área de um Triângulo (A) usando a base e a altura:

```
A = (Base × Altura) / 2
```

Área de um Triângulo (A) usando os lados (Fórmula de Heron): Onde "s" é o semiperímetro do triângulo.

```
Perímetro (s) = (a + b + c) / 2
A = √(s × (s - a) × (s - b) × (s - c))
```

Teorema de Pitágoras para Triângulos Retângulos:

```
a² + b² = c²
```

Lei dos Senos:

```
a / sen(A) = b / sen(B) = c / sen(C)
```

Lei dos Cossenos:

```
c² = a² + b² - 2ab × cos(C)
```

Altura de um Triângulo:

```
Altura = (2 × Área) / Base
```

Mediana de um Triângulo: A mediana de um triângulo é o segmento que liga um vértice ao ponto médio do lado oposto. As medianas de um triângulo se encontram em um ponto chamado centroide.

Círculo

Claro! Aqui estão algumas fórmulas relacionadas ao círculo formatadas em Markdown para você copiar e colar:

Circunferência de um Círculo (C):

```
C = 2πr
```

Área de um Círculo (A):

```
A = πr²
```

Diâmetro de um Círculo (d):

```
d = 2r
```

Relação entre o Diâmetro e a Circunferência:

```
C = πd
```

Comprimento da Circunferência de um Setor Circular: Se o ângulo central do setor circular é θ (em radianos) e o raio é r:

```
Comprimento = θr
```

Área de um Setor Circular: Se o ângulo central do setor circular é θ (em radianos) e o raio é r:

```
Área = (θ/2) × r²
```

Comprimento do Arco de um Círculo: Se o ângulo central do arco é θ (em radianos) e o raio é r:

```
Comprimento do Arco = θr
```

Fórmula da Área do Círculo em Função do Diâmetro:

```
A = (π/4) × d²
```

Relação entre a Área do Círculo e o Comprimento da Circunferência:

```
A = (C²) / (4π)
```

Comprimento da Corda de um Círculo: Se o ângulo central do setor circular é θ (em radianos) e o raio é r, e a corda é igual ao raio, a fórmula para o comprimento da corda é:

```
Comprimento da Corda = 2r × sen(θ/2)
```

Essas são algumas das fórmulas matemáticas básicas relacionadas ao círculo, cada uma descrevendo diferentes propriedades e relações geométricas do círculo.

Inscrito e circunscrito

Círculo Circunscrito: Um círculo circunscrito é aquele que passa por todos os vértices de uma figura geométrica, geralmente um polígono. No caso de triângulos, por exemplo, um círculo circunscrito passa pelos três vértices do triângulo, tocando cada vértice. A posição do centro do círculo circunscrito é tal que os raios a partir do centro até os vértices do polígono têm o mesmo comprimento, que é o raio da circunferência.

Círculo Inscrito: Um círculo inscrito é aquele que está inteiramente contido dentro de uma figura geométrica, geralmente um polígono. No caso de triângulos, um círculo inscrito está inscrito no interior do triângulo, tangenciando os lados do triângulo em pontos específicos. A posição do centro do círculo inscrito é tal que as linhas que ligam o centro aos pontos de tangência nos lados do polígono são perpendiculares aos lados.

Fórmulas

Triângulo:

Círculo Circunscrito:

- Raio R do círculo circunscrito:

  ```
  R = (a * b * c) / (4 * Área)
  ```

- Verificação: Se $a^2 + b^2 = c^2$, o triângulo é retângulo e está circunscrito a uma circunferência.

Círculo Inscrito:

- Raio r do círculo inscrito:

  ```
  r = Área / s
  ```

- Verificação: Se a + b > c, a + c > b e b + c > a, o triângulo tem uma circunferência inscrita.

Quadrado:

Círculo Circunscrito:

- Raio R do círculo circunscrito:

  ```
  R = lado / 2
  ```

Círculo Inscrito:

- Raio r do círculo inscrito:

  ```
  r = lado / 2
  ```

Hexágono Regular:

Círculo Circunscrito:

- Raio R do círculo circunscrito:

  ```
  R = lado
  ```

Círculo Inscrito:

- Raio r do círculo inscrito:

  ```
  r = (lado * √3) / 2
  ```

Pentágono Regular:

Círculo Circunscrito:

- Raio R do círculo circunscrito:

```
R = (lado / 2) * √(5 + 2√5)
```

Círculo Inscrito:

- Raio r do círculo inscrito:

```
r = (lado / 4) * √(5 - 2√5)
```

Lembrando que nas fórmulas acima, 'lado' representa o comprimento do lado do polígono, 'Área' é a área do polígono e 's' é o semiperímetro do triângulo. Além disso, as verificações mencionadas são critérios para a existência de círculos inscritos ou circunscritos com base nas propriedades dos polígonos.

Geometria 3D

Cubo

Área da superfície do cubo: A área total da superfície de um cubo é dada por:

```
Área = 6 * (lado)^2
```

Volume do cubo: O volume de um cubo é calculado através da fórmula:

```
Volume = (lado)^3
```

Diagonal do cubo: A diagonal de um cubo pode ser encontrada usando o teorema de Pitágoras em três dimensões:

```
Diagonal = √(3) * lado
```

Cilindro

Área da superfície do cilindro: A área total da superfície de um cilindro é a soma da área lateral e das áreas das bases:

```
Área = 2 * π * raio * altura + 2 * π * (raio)^2
```

Volume do cilindro: O volume de um cilindro é dado por:

```
Volume = π * (raio)^2 * altura
```

Diagonal do cilindro: A diagonal de um cilindro retangular pode ser calculada usando o teorema de Pitágoras:

```
Diagonal = √(altura^2 + (2 * raio)^2)
```

Prisma

Área da superfície do prisma: A área total da superfície de um prisma é a soma da área lateral e das áreas das bases:

```
Área = 2 * (área da base) + (perímetro da base) * altura
```

Volume do prisma: O volume de um prisma é dado por:

```
Volume = (área da base) * altura
```

Diagonal do prisma: A diagonal de um prisma retangular pode ser calculada usando o teorema de Pitágoras:

```
Diagonal = √(altura^2 + diagonal da base^2)
```

Pirâmide

Área da superfície da pirâmide: A área total da superfície de uma pirâmide é a soma da área da base e da área lateral:

```
Área = (área da base) + (1/2) * (perímetro da base) * apótema + (área lateral)
```

Volume da pirâmide: O volume de uma pirâmide é dado por:

```
Volume = (1/3) * (área da base) * altura
```

Relação entre altura da pirâmide e altura da pirâmide truncada: Se uma pirâmide é truncada paralelamente à base para formar outra pirâmide, a relação entre as alturas é proporcional à relação das áreas das bases:

```
Altura_truncada = (área da base truncada) / (área da base original) * Altura_original
```

A "apótema" de uma figura geométrica é a distância entre o centro da figura e o ponto médio de um dos lados. Em muitos casos, é usada para representar a distância do centro até o ponto médio de um lado de um polígono regular, como um triângulo, quadrado, pentágono, hexágono, etc.

Cone

Área da superfície do cone: A área total da superfície de um cone é a soma da área lateral e da área da base:

```
Área = π * raio * geratriz + π * (raio)^2
```

onde a "geratriz" é o comprimento da linha reta que liga o vértice do cone até um ponto qualquer na circunferência da base.

Volume do cone: O volume de um cone é dado por:

```
Volume = (1/3) * π * (raio)^2 * altura
```

Relação entre cones semelhantes: Se você tem dois cones com alturas proporcionais, a razão dos volumes é igual ao cubo da razão dos raios:

```
Volume_cone1 / Volume_cone2 = (raio1 / raio2)^3
```

Paralelepípedo

Área da superfície do paralelepípedo: A área total da superfície de um paralelepípedo é a soma das áreas de suas faces:

```
Área = 2 * (comprimento * largura + comprimento * altura + largura * altura)
```

Volume do paralelepípedo: O volume de um paralelepípedo é dado por:

```
Volume = comprimento * largura * altura
```

Diagonais do paralelepípedo: As diagonais de um paralelepípedo podem ser calculadas usando o teorema de Pitágoras:

```
Diagonal1 = √(comprimento^2 + largura^2 + altura^2)
Diagonal2 = √(comprimento^2 + largura^2 + altura^2)
Diagonal3 = √(comprimento^2 + largura^2 + altura^2)
```

Esfera

Área da superfície da esfera: A área total da superfície de uma esfera é dada por:

```
Área = 4 * π * (raio)^2
```

Volume da esfera: O volume de uma esfera é calculado através da fórmula:

```
Volume = (4/3) * π * (raio)^3
```

Diâmetro da esfera: O diâmetro de uma esfera é duas vezes o raio:

```
Diâmetro = 2 * raio
```

Programação dinâmica

Problema do Troco

Given an integer array of coins[ ] of size N representing different types of denominations and an integer sum, the task is to find the number of ways to make sum by using different denominations.

Bottom-Up

Time complexity : O(N*sum)

Auxiliary Space : O(sum)

```
int count(int coins[], int n, int sum)
{
    // table[i] will be storing the number of solutions for
    // value i. We need sum+1 rows as the dp is
    // constructed in bottom up manner using the base case
    // (sum = 0)
    int dp[sum + 1];

    // Initialize all table values as 0
    memset(dp, 0, sizeof(dp));

    // Base case (If given value is 0)
    dp[0] = 1;

    // Pick all coins one by one and update the table[]
    // values after the index greater than or equal to the
    // value of the picked coin
    for (int i = 0; i < n; i++)
        for (int j = coins[i]; j <= sum; j++)
            dp[j] += dp[j - coins[i]];
    return dp[sum];
}


int coins[] = { 1, 2, 3 };
int n = sizeof(coins) / sizeof(coins[0]);
int sum = 5;
cout << count(coins, n, sum); // 5
```

Top-Down

```
int count(vector<int>& coins, int n, int sum,
          vector<vector<int> >& dp)
{
    // Base Case
    if (sum == 0)
        return dp[n][sum] = 1;

    // If number of coins is 0 or sum is less than 0 then
    // there is no way to make the sum.
    if (n == 0 || sum < 0)
        return 0;
```

```
    // If the subproblem is previously calculated then
    // simply return the result
    if (dp[n][sum] != -1)
        return dp[n][sum];

    // Two options for the current coin
    return dp[n][sum]
           = count(coins, n, sum - coins[n - 1], dp)
             + count(coins, n - 1, sum, dp);
}

int n, sum;
n = 3, sum = 5;
vector<int> coins = { 1, 2, 3 };
// 2d dp array to store previously calculated
// results
vector<vector<int> > dp(n + 1,
                        vector<int>(sum + 1, -1));
int res = count(coins, n, sum, dp);
cout << res << endl; // 5
```

Cortando canos

Dada uma relação de comprimentos de cano e seus respectivos valores de venda, determine o maior valor total que possa ser obtido com o corte de um cano de comprimento inicial determinado.

```
ll lucro_maximo(vector<pll>& canos, ll sizeCano) {
    vll memo(1e4 + 10, 0);
    rep(i, sizeCano + 1) {
        foreach (cano, canos) {
            if (i < cano.f) continue;
            memo[i] = max(memo[i], cano.s + memo[i - cano.f]);
        }
    }
    return memo[sizeCano];
}
```

Problema da Mochila

Suponha dado um conjunto de objetos, cada um com um certo peso e um certo valor. Quais dos objetos devo colocar na minha mochila para que o valor total seja o maior possível? Minha mochila tem capacidade para 15 kg apenas.

Bottom-Up

```
// Function to find the maximum profit
int knapSack(int W, int wt[], int val[], int n)
{
    // Making and initializing dp array
    int dp[W + 1];
    memset(dp, 0, sizeof(dp));

    for (int i = 1; i < n + 1; i++) {
        for (int w = W; w >= 0; w--) {
```

```
            if (wt[i - 1] <= w)

                // Finding the maximum value
                dp[w] = max(dp[w],
                            dp[w - wt[i - 1]] + val[i - 1]);
        }
    }
    // Returning the maximum value of knapsack
    return dp[W];
}
int profit[] = { 60, 100, 120 };
int weight[] = { 10, 20, 30 };
int W = 50;
int n = sizeof(profit) / sizeof(profit[0]);
cout << knapSack(W, weight, profit, n);
```

Top-Down

```
class Item {
   public:
    int peso;
    ll valor;
};

vector<vector<ll>> memo(100, vector<ll>(100005, -1));

ll valor_maximo(int item_atual, int capacidade_disponivel, vector<Item>& itens) {
    if (capacidade_disponivel < 0) return -LONG_MAX / 2;
    if (capacidade_disponivel == 0 || item_atual == itens.size()) return 0;

    if (memo[item_atual][capacidade_disponivel] != -1) return memo[item_atual][capacidade_disponivel];

    return memo[item_atual][capacidade_disponivel] = max(
               // Caso a capacidade se torne negativa o retorno será -LONG_MAX, dessa forma assumimos que o
valor
               // retornado é tão pequeno que será desconsiderado pelo MAX()
               itens[item_atual].valor + valor_maximo(item_atual + 1, capacidade_disponivel -
itens[item_atual].peso, itens),
               valor_maximo(item_atual + 1, capacidade_disponivel, itens));
}
cout << valor_maximo(0, capacidade, itens) << "\n";
```

Com tracking de itens

```
void printknapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    int K[n + 1][W + 1];

    // Build table K[][] in bottom up manner
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
```

```
            else if (wt[i - 1] <= w)
                K[i][w] = max(val[i - 1] +
                    K[i - 1][w - wt[i - 1]], K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }

    // stores the result of Knapsack
    int res = K[n][W];
    cout<< res << endl;

    w = W;
    for (i = n; i > 0 && res > 0; i--) {

        // either the result comes from the top
        // (K[i-1][w]) or from (val[i-1] + K[i-1]
        // [w-wt[i-1]]) as in Knapsack table. If
        // it comes from the latter one/ it means
        // the item is included.
        if (res == K[i - 1][w])
            continue;
        else {

            // This item is included.
            cout<<" "<<wt[i - 1] ;

            // Since this weight is included its
            // value is deducted
            res = res - val[i - 1];
            w = w - wt[i - 1];
        }
    }
}

int val[] = { 60, 100, 120 };
int wt[] = { 10, 20, 30 };
int W = 50;
int n = sizeof(val) / sizeof(val[0]);

printknapSack(W, wt, val, n);
```

Com repetição de itens

```
// Returns the maximum value with knapsack of
// W capacity
int unboundedKnapsack(int W, int n,
                       int val[], int wt[])
{
    // dp[i] is going to store maximum value
    // with knapsack capacity i.
    int dp[W+1];
    memset(dp, 0, sizeof(dp));

    // Fill dp[] using above recursive formula
```

```
    for (int i=0; i<=W; i++)
      for (int j=0; j<n; j++)
         if (wt[j] <= i)
            dp[i] = max(dp[i], dp[i-wt[j]] + val[j]);

    return dp[W];
}
int W = 100;
int val[] = {10, 30, 20};
int wt[] = {5, 10, 15};
int n = sizeof(val)/sizeof(val[0]);

cout << unboundedKnapsack(W, n, val, wt); // 300
```

Com repetição e tracking dos itens

```
class UnboundedKnapsack {
   public:
    vector<ll> knapsack;
    vector<vector<ll>> selectedElements;
    ll maximumCapacity;

    vector<ll> knapsack_unbounded(vector<ll>& pesos, vector<ll>& valores, ll num_itens, ll capacidade) {
        // Stores the maximum value which can be reached with a certain capacity
        knapsack.clear();
        knapsack.resize(capacidade + 1);

        maximumCapacity = capacidade + 1;

        // Stores selected elements with a certain capacity
        selectedElements.resize(capacidade + 1);

        // Initializes maximum value vector with zero
        for (ll i = 0; i < capacidade + 1; i++) {
            knapsack[i] = 0;
        }

        // Computes the maximum value that can be reached for each capacity
        for (ll capacity = 0; capacity < capacidade + 1; capacity++) {
            // Goes through all the elements
            for (ll n = 0; n < num_itens; n++) {
                if (pesos[n] <= capacity) {
                    if (knapsack[capacity] <= knapsack[capacity - pesos[n]] + valores[n]) {
                        knapsack[capacity] = knapsack[capacity - pesos[n]] + valores[n];

                        // Stores selected elements
                        selectedElements[capacity].clear();
                        selectedElements[capacity].push_back(n + 1);

                        for (ll elem : selectedElements[capacity - pesos[n]]) {
                            selectedElements[capacity].push_back(elem);
                        }
                    }
                }
            }
```

```
        }

        return this->selectedElements[capacidade];
    }
};

UnboundedKnapsack mochila;
vll index_itens_escolhidos = mochila.knapsack_unbounded(pesos, valores, num_itens, capacidade);

ll sum = 0;
foreach (i, index_itens_escolhidos) {
    sum += valores[i - 1];
}

log(sum);
```

Problema da mochila fracionado

Given the weights and profits of N items, in the form of {profit, weight} put these items in a knapsack of capacity W to get the maximum total profit in the knapsack. In Fractional Knapsack, we can break items for maximizing the total value of the knapsack.

```
struct Item {
    int profit, weight;

    // Constructor
    Item(int profit, int weight)
    {
        this->profit = profit;
        this->weight = weight;
    }
};

// Comparison function to sort Item
// according to profit/weight ratio
static bool cmp(struct Item a, struct Item b)
{
    double r1 = (double)a.profit / (double)a.weight;
    double r2 = (double)b.profit / (double)b.weight;
    return r1 > r2;
}

// Main greedy function to solve problem
double fractionalKnapsack(int W, struct Item arr[], int N)
{
    // Sorting Item on basis of ratio
    sort(arr, arr + N, cmp);

    double finalvalue = 0.0;

    // Looping through all items
    for (int i = 0; i < N; i++) {

        // If adding Item won't overflow,
        // add it completely
        if (arr[i].weight <= W) {
            W -= arr[i].weight;
```

```
            finalvalue += arr[i].profit;
        }

        // If we can't add current Item,
        // add fractional part of it
        else {
            finalvalue
                += arr[i].profit
                   * ((double)W / (double)arr[i].weight);
            break;
        }
    }

    // Returning final value
    return finalvalue;
}

int W = 50;
Item arr[] = { { 60, 10 }, { 100, 20 }, { 120, 30 } };
int N = sizeof(arr) / sizeof(arr[0]);

cout << fractionalKnapsack(W, arr, N); // 240
```

LCS

Given two strings, S1 and S2, the task is to find the length of the Longest Common Subsequence, i.e. longest subsequence present in both of the strings.

A longest common subsequence (LCS) is defined as the longest subsequence which is common in all given input sequences.

Bottom-Up

Time Complexity: O(m * n) which remains the same.

Auxiliary Space: O(m) because the algorithm uses two arrays of size m.

```
int longestCommonSubsequence(string& text1, string& text2)
{
    int n = text1.size();
    int m = text2.size();

    // initializing 2 vectors of size m
    vector<int> prev(m + 1, 0), cur(m + 1, 0);

    for (int idx2 = 0; idx2 < m + 1; idx2++)
        cur[idx2] = 0;

    for (int idx1 = 1; idx1 < n + 1; idx1++) {
        for (int idx2 = 1; idx2 < m + 1; idx2++) {
            // if matching
            if (text1[idx1 - 1] == text2[idx2 - 1])
                cur[idx2] = 1 + prev[idx2 - 1];

            // not matching
            else
                cur[idx2]
                    = 0 + max(cur[idx2 - 1], prev[idx2]);
```

```
        }
        prev = cur;
    }

    return cur[m];
}
longestCommonSubsequence(S1, S2);
```

Top-Dowm

Time Complexity: O(m * n) where m and n are the string lengths.

Auxiliary Space: O(m * n) Here the recursive stack space is ignored.

```
int lcs(string& X, string& Y, int m, int n, vector<vector<int>>& dp) {
    if (m == 0 || n == 0)
        return 0;
    if (X[m - 1] == Y[n - 1])
        return dp[m][n] = 1 + lcs(X, Y, m - 1, n - 1, dp);

    if (dp[m][n] != -1) {
        return dp[m][n];
    }
    return dp[m][n] = max(lcs(X, Y, m, n - 1, dp),
                          lcs(X, Y, m - 1, n, dp));
}

vector<vector<int>> dp(s1.size() + 1, vector<int>(s2.size() + 1, -1));
ll resp = lcs(s1, s2, s1.size(), s2.size(), dp);
```

LIS

Longest Increasing Subsequence

```
int lis(int arr[], int n)
{
    int lis[n];

    lis[0] = 1;

    // Compute optimized LIS values in
    // bottom up manner
    for (int i = 1; i < n; i++) {
        lis[i] = 1;
        for (int j = 0; j < i; j++)
            if (arr[i] > arr[j] && lis[i] < lis[j] + 1)
                lis[i] = lis[j] + 1;
    }

    // Return maximum value in lis[]
    return *max_element(lis, lis + n);
}

int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
```

```
int n = sizeof(arr) / sizeof(arr[0]);
lis(arr, n); // 5
```