1. Lucas, como sempre muito observador, percebeu que alguns funcionários estavam esbarrando demais no corredor da empresa.

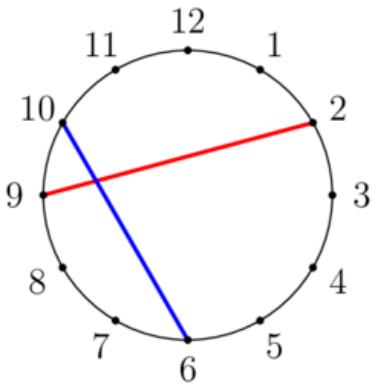
   Pensando nisso, ele teve uma ideia: sugeriu que você, um(a) excelente programador(a), desenvolvesse um sistema com um TAD para representar os funcionários e verificar se os turnos de trabalho deles se sobrepõem.

   Para facilitar as coisas, a empresa adotou um relógio especial, que vai de 1 a 12 (não usa o formato 24 horas).

   Sua missão é criar um TAD Funcionario, contendo:

   - Nome
   - Id
   - Horário de início do expediente (número inteiro de 1 a 12)
   - Horário de fim do expediente (número inteiro de 1 a 12)

   Após cadastrar alguns funcionários, será possível consultar, dado o **ID de dois funcionários**, se **os horários de trabalho deles se sobrepõem,** ou seja, se há algum intervalo de tempo em que **ambos estejam trabalhando ao mesmo tempo**



   - Funcionário de ID **1** (Lucas): início às **2**, fim às **9**
   - Funcionário de ID **2** (iShowSpeed): início às **6**, fim às **10**
   - Como ambos estão na empresa das **6 às 9**, **há sobreposição** de horário! (como mostra o relógio da imagem acima)

Possíveis protótipos:

- **criar_funcionario(nome, idade, inicio, fim): Funcionario**
- **verificar_sobreposicao(id1, id2): booleano**

2. Com base na estrutura desenvolvida no **Problema Final (Da lista de alocação dinâmica) – Implementação de Vetor Dinâmico estilo c++**, construa agora um TAD que irá **simular os ataques do seu personagem em uma batalha contra o chefe final** de um videogame.

   O chefe possui **H** de vida. Seu personagem possui **N** ataques. Cada ataque possui:

   - **dano** que causa ao chefe (dano);
   - **tempo de recarga** (cooldown), ou seja, após ser usado, ele só poderá ser usado novamente após cooldown turnos.

   Inicialmente, **todos os ataques estão disponíveis**. A cada turno:

   - Você pode usar **todos os ataques disponíveis**.
   - Depois de usados, esses ataques entram em cooldown.
   - Se **nenhum ataque estiver disponível**, o turno apenas avança.
   - O chefe é derrotado quando sua vida for **menor ou igual a zero**

   Seu objetivo:

   Com base no TAD Vector da lista de alocação dinâmica, implemente o TAD abaixo em conjunto com o Vector, para resolver o problema descrito:

```
typedef struct {
   int dano;
   int cooldown;
   int prontoNoTurno; // turno em que o ataque poderá ser usado novamente
} Ataque;
```

3. Você deve implementar um TAD que represente uma **avenida de comprimento x**. Inicialmente, não há sinais de trânsito. Um a um, sinais são adicionados em posições específicas. Sua tarefa é **calcular o maior trecho contínuo sem sinal de trânsito após cada inserção**.

   Avenida *criaAvenida(int comprimento, int capacidade);
   void adicionaSinal(Avenida *a, int posicao);
   int maiorTrechoSemSinal(Avenida *a);
   void liberaAvenida(Avenida *a);

   struct Avenida {
   int comprimento; // Comprimento da avenida
   int *sinais; // Lista com as posições onde os semáforos **serão postos**
   int total; // Total de semáforos **até o momento**
   int capacidade; // Capacidade maxima de semáforos
   };

4. Ao contrário dos cavaleiros da famosa mesa redonda nobres, justos e meio entediantes, os **Cavaleiros da Mesa Poligonal** vivem em um cenário bem mais caótico. Eles não se importam muito com honra, **adoram ostentar ouro** e, se for preciso, passam a perna uns nos outros com um sorriso no rosto.

   Cada cavaleiro possui:

- Um **nível de poder** (quanto maior, mais perigoso),
- Uma certa quantidade de **ouro** (que ele carrega por aí, exibindo para impressionar dragões e rivais).

   Mas como eles são espertos (e um pouco aproveitadores), ao se apresentar à **Mesa TADangular**, cada cavaleiro pode "**herdar"** (pegar emprestado, se é que você me entende..) o ouro de até **K** cavaleiros **mais fracos**.

   Seu objetivo:

   struct Cavaleiro {

   int poder;

int ouro;

int id_original; // para manter a ordem de entrada

};

Você deve:

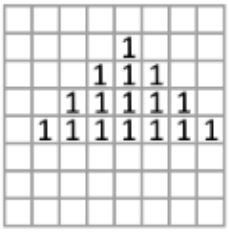
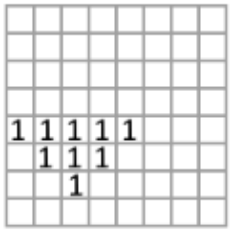
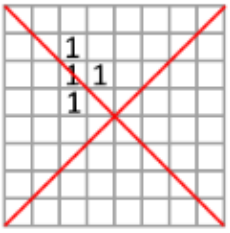
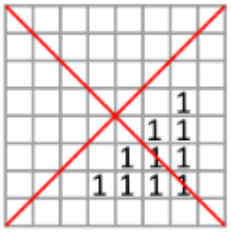
1. **Cadastrar os cavaleiros.**
2. **Ordená-los por poder.**
3. **Calcular o total de ouro que cada cavaleiro pode somar ao seu (considerando o ouro de até K cavaleiros mais fracos).**
4. **Exibir o valor final de ouro de cada um, na ordem em que foram inseridos.**

5. Após se aposentar da maratona de programação, Lucas desenvolveu um forte trauma por questões de geometria computacional. Bastava ouvir a palavra *"retângulo"* que ele já começava a tremer.

   Porém, como todo bom programador aposentado, ele não conseguiu ficar longe da lógica por muito tempo...

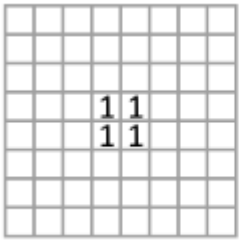
   Agora, ele quer sua ajuda: dado uma **malha binária** representando uma figura composta apenas por '0' e '1', identifique se a forma desenhada é um **quadrado** ou um **triângulo**.

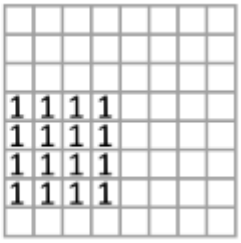
   Para facilitar sua vida (e evitar gatilhos no Lucas), você deve implementar um **TAD**, que esconda toda a dor da geometria por trás de uma bela abstração.

Triângulo:

- **Ele cresce linha por linha, como uma pirâmide.**
- **O inverso também será valido**
- **Rotações do triangulo não deveram ser tratadas (90º)**





Quadrado:

- **É um quadrado..**

Observação:

Este problema não é necessariamente voltado apenas à implementação de TADs, mas o uso de uma estrutura adequada pode **facilitar o raciocínio** e **modelar o problema** de forma mais clara.
O intuito é justamente esse: exercitar a modelagem e a lógica, **sem precisar de fórmulas geométricas.**