Análise dos Problemas da 2ª Seletiva 14/07/17

Maratona de Programação Unioeste

E - Frota de Táxi

- **Problema**

  Dados preço por litro e rendimento de álcool/gasolina, dizer qual é mais econômico.

- **Solução**

  Dividindo preço por rendimento, temos o preço por km:

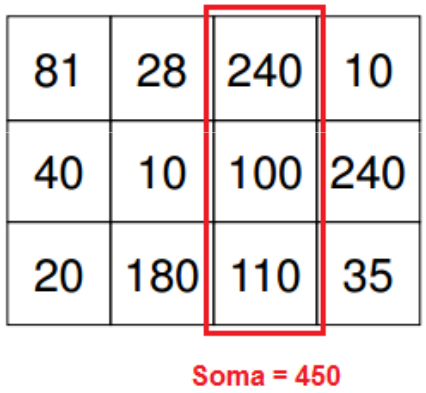
$$\frac{R\$}{L} * \frac{L}{km} = \frac{R\$}{km}$$

  Assim, basta imprimir aquele com menor preço por km.

G – Campo de Minhocas

- **Problema**

  Dado uma matriz, imprimir a maior soma dos elementos de uma linha ou coluna da matriz.



- **Solução**

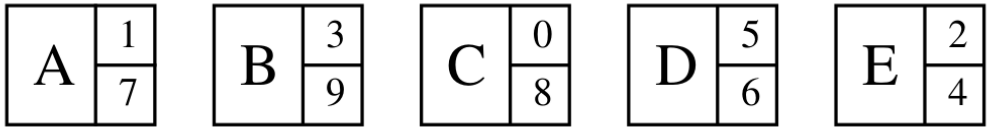
  Implementar o que é pedido.

```
for (i = 0; i < n; i++)
{
    soma = 0;

    for(j = 0; j < m; j++)
        soma += matriz[i][j];

    if(soma > maior)
        maior = soma;
}
```

A – Proteja sua Senha

- **Problema**

  Dadas N senhas digitadas pelo cliente (letras) e as associações de cada letra com os dígitos, determinar qual a senha do cliente. O problema garante que as entradas sempre determinam univocamente a senha.



- **Solução**

  Inicializamos uma matriz 6x10 indicando, para cada posição da senha, quais dígitos são possíveis naquela posição.

  mat[i][j] => 1 se o dígito ‘j’ pode ser o i-ésimo dígito da senha

  Inicialmente a matriz é preenchida com 1.

A – Proteja sua Senha

- Ao ler cada senha digitada, obtemos dois dígitos possíveis naquela posição:

  1 7 **3 9** 0 8 5 6 2 4 **B** C E A E B
- Ao ler B, na primeira posição são possíveis os dígitos 3 e 9.
- Zeramos a matriz nesta linha para todos os dígitos exceto 3 e 9.

```
a = digitos[2 * (c – ‘A’)];
b = digitos[2 * (c – ‘A’) + 1];

for (i = 0; i < 10; i++)
    if(i != a && i != b)
        possivel[lin][i] = 0;
```

- Ao final, cada linha terá apenas 1 valor ‘1’. Encontramos qual dígito que tem valor 1 e imprimimos.

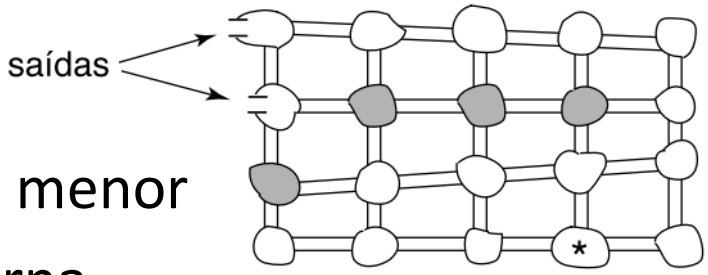
A – Proteja sua Senha

- Outra solução:
- Como são 6 dígitos, há 10^6 possibilidades de senha.
- Podemos testar todas as possibilidades de senha e ver se ela se encaixa no que foi digitado.

```
int senha[6];

void gera(int i)
{
    if(i == 6) {
        testa_senha();
        return;
    }

    for(int k = 0; k < 10; k++)
    {
        senha[i] = k;
        gera(i+1);
    }
}
```

- Obviamente é bem mais lenta, mas o código pode ser mais simples.
- Neste problema, extrair a solução das restrições é simples; nem sempre é o caso.
- 5552 - Codebreakers

I – Duende Perdido



- **Problema**

  Dado o mapa da caverna, encontrar a menor distância do duende para sair da caverna.

- **Solução**

  Fazemos um Flood Fill (BFS) a partir da posição inicial do duende, explorando todas os elementos adjacentes com valor != 2.

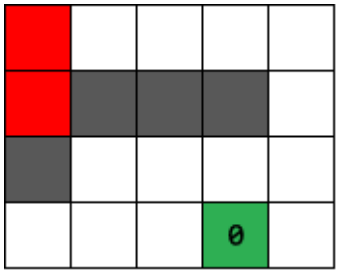
  Ao encontrar uma célula não visitada, marcamos sua distância como dist[atual] + 1.

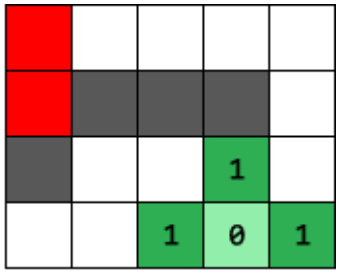
  Se usarmos fila (e não a versão recursiva), o Flood Fill explorará todas as células a 1 de distância; depois todas a 2 de distância; depois todas a 3 de distância, etc.

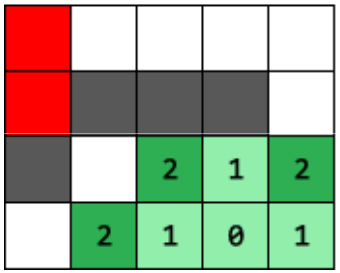
I – Duende Perdido

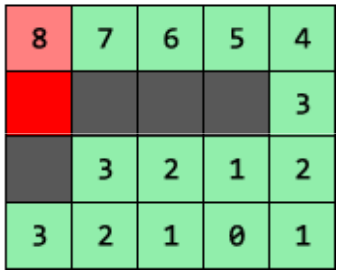
- FloodFill(ini):

```
fila.push(ini);
visitado[ini] = 1;
dist[ini] = 0;

enquanto fila não vazia:
    atual = fila.cabeca();
    fila.remove();

    para cada adjacente do atual:
      se(!visitado[adj] e mapa[adj] != parede)
        visitado[adj] = 1;
        dist[adj] = dist[atual] + 1
        fila.push(adj);
```









- Quando encontrarmos uma saída (mapa[atual] == 0), retornamos dist[atual].

C - Orkut

- **Problema**

  Dados os amigos que Larissa quer adicionar e as restrições que eles têm para adicioná-la, imprimir a ordem em que eles devem ser adicionados (ou imprimir que não há solução).

- **Solução**

  Primeiro, para simplificar a implementação, transformamos todos os nomes em números:

```
int id(char *nome) {
    for(int i = 0; i < n; i++)
        if(strcmp(nome, nomes[i]) == 0)
            return i;
    return -1;
}
```

Em C++, podemos usar map<string,int> (mais simples e eficiente).

C - Orkut

- Como os limites são baixos (N <= 30), podemos ficar simulando repetidamente o processo:
  - Percorremos a lista de amigos:
    - Caso o amigo ainda não esteja adicionado, percorremos sua lista de restrições. Se todos os amigos da lista tiverem sido adicionados, adicionamos este amigo também.
- O laço é interrompido quando percorrermos toda a lista sem adicionar ninguém (ou então fixamos N iterações, pois após N iterações todos os possíveis terão sido adicionados).

C - Orkut

- Outra abordagem mais eficiente seria inverter as listas: para cada amigo, armazenamos os amigos que têm aquele amigo na sua lista de restrições.
- Inicializamos um vetor com o número de amigos que faltam ser adicionados para adicionar cada amigo.
- Repetimos o processo de simulação, mas agora, ao invés de percorrer toda a lista de restrições, simplesmente conferimos se faltam[i] == 0.
- Se faltam[i] == 0 e i ainda não foi adicionado, adicionamos o amigo i e percorremos sua lista decrementado o vetor ‘faltam’ para quem está em sua lista.
- Podemos ainda utilizar uma fila de prioridade para encontrar o menor valor faltam[i].

C - Orkut

- O problema ainda pode ser resolvido utilizando grafos.
- Criamos o grafo inverso (cada amigo tem arestas para os amigos que o têm em sua lista).
- Se houver ciclo, não há solução;
- Senão, fazemos uma ordenação topológica e imprimimos esta ordenação.

D - Tempo

- **Problema**

  Dadas duas datas e um período de tempo, calcular quantas vezes o período de tempo ‘cabe’ entre as duas datas.

- **Solução**

  Fazemos uma função que recebe uma data e retorna o número de segundos desde 01/01/1970 00:00:00.

  Calculamos a diferença entre o número de segundos das duas datas.

  Transformamos o período de tempo em segundos e imprimimos a diferença dividido pelo período de tempo.

D - Tempo

- Implementando a função para calcular o número de segundos:

```
int dias_mes[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

int segundos_desde_1970(int ano, int mes, int dia, int hora, int min, int sec)
{
    int dias = (ano - 1970) * 365 + (ano - 1969) / 4;

    for(int i = 1; i < mes; i++)
        dias += dias_mes[i];
    if(ano % 4 == 0 && mes > 2) dias++;

    dias += dia;

    int segundos_hoje = hora * 3600 + min * 60 + sec;

    return dias * 24 * 3600 + segundos_hoje;
}
```

- Como o ano varia apenas entre 1970 e 2030, não é necessário se preocupar com anos divisíveis por 4 mas não bissextos.
- URI 1374, 1619

H - Append

- **Problema**

  Considere o esquema de codificação composto por uma sequência de pares (p_i, r_i). Decodificamos uma palavra da seguinte forma:

  Se $p_i = 0$, r_i é um caractere, que é adicionado ao fim da string decodificada;

  Senão, r_i é um inteiro, e adicionamos ao fim da string decodificada r_i caracteres a partir do caractere p_i antes do fim.

H - Append

Par	A adicionar	Nova string
(0,a)	a	a
(1,1)	a	aa
(0,b)	b	aab
(3,3)	aab	aabaab
(3,3)	aabaab	aabaabaab
(3,2)	aabaabaab	aabaabaabaa
(0,c)	c	aabaabaabaac

Dado o esquema de codificação Cw que produz a palavra w, quantas possibilidades existem para expressar Cw como CuCv não-vazios tal que w = uv?

H - Append

- **Solução**
- Entendido o problema, temos N-1 divisões possíveis:
  - [0, 1) e [1, N)
  - [0, 2) e [2, N)
  - ....
  - [0, N-1) e [N-1, N)
- Primeiro observamos que Cu sempre forma uma palavra u que é prefixo de w, uma vez que em ambos tipos de operação sempre adicionamos caracteres ao final da string.
- Então o problema passa a ser em quais dessas divisões a sequência Cv formará uma sequência que é sufixo de w de forma que uv = w.

H - Append

- Poderíamos criar uma função que recebe o intervalo [l, r] de pares a ser considerado e retorna a string correspondente, e assim testar se u + v == w, mas isso é muito lento (o número de pares, não especificado na entrada, vai até 50.000).
- Veja que nem todas subsequências são válidas (podem fazer referência a caracteres inexistentes na string decodificada); por exemplo subsequências que não começam com (0,_) não são válidas. Repare que todas as sequências Cu são válidas.
- Na verdade, ocorre que todas as sequências Cv válidas formam uma string v tal que uv = w.

H - Append

- Consideremos uma sequência Cv válida cujo primeiro par é (0,X). Digamos que na string w, esse caractere X ocorre na posição m.
- String: _ _ _ ... _ _ X
  0 1 2 m
- Como a sequência é válida, nenhum dos pares a partir daí requer a cópia de um caractere antes de m (pois nesse caso o caractere não existiria na string da sequência Cv e a sequência seria inválida).
- Logo, todas as operações a partir de (0,X) na sequência original Cw adicionarão os mesmos caracteres que as operações na sequência Cv, de forma que v é sufixo de w.

H - Append

- Além disso, o tamanho da string decodificada depende apenas dos pares envolvidos: o tamanho é a soma de todos os r_i mais um para cada $p_i = 0$.
- Então $|u| + |v| = |w|$, e como u é prefixo de w e v é sufixo de w, então uv = w.
- Assim, ao invés de gerar as strings, podemos simplesmente testar para cada i se a sequência [i, N) é válida. Para cada sequência válida incrementamos a resposta.
- Se testarmos isso de forma ingênua, ainda temos N^2 operações.

H - Append

- A solução linear é obtida usando uma pilha.

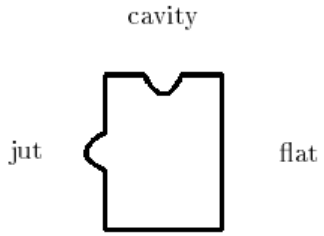
```
int tam = 0;                    // tam mantém o tamanho atual da string w
para cada par(p, r) {
    se(p == 0) pilha.push(tam++);
    senao
    {
        enquanto(tam – pilha.top() < p) pilha.pop();
        tam += r;
    }
}
                                // pilha.size()-1 pois desconsideramos a primeira
print(pilha.size() – 1);        // sequência quando Cv = Cw e Cu = vazio
```

- Toda vez que encontramos um par (0,_) adicionamos na pilha a posição na string em que aquela sequência começa.
- Quando encontramos um par (p,r) com p > 0, testamos se a sequência no topo da pilha é inválida. tam – pilha.top() nos dá o tamanho da sequência no topo da pilha. Retiramos todas as sequências com tamanho menor que p (ou seja, sequências inválidas). Ao final, as sequências que restarem são válidas.

F – Quebra-cabeça

- **Problema**

  Dadas as peças de um quebra cabeça, dizer se é possível formar um retângulo n x m com as peças.



- **Solução**

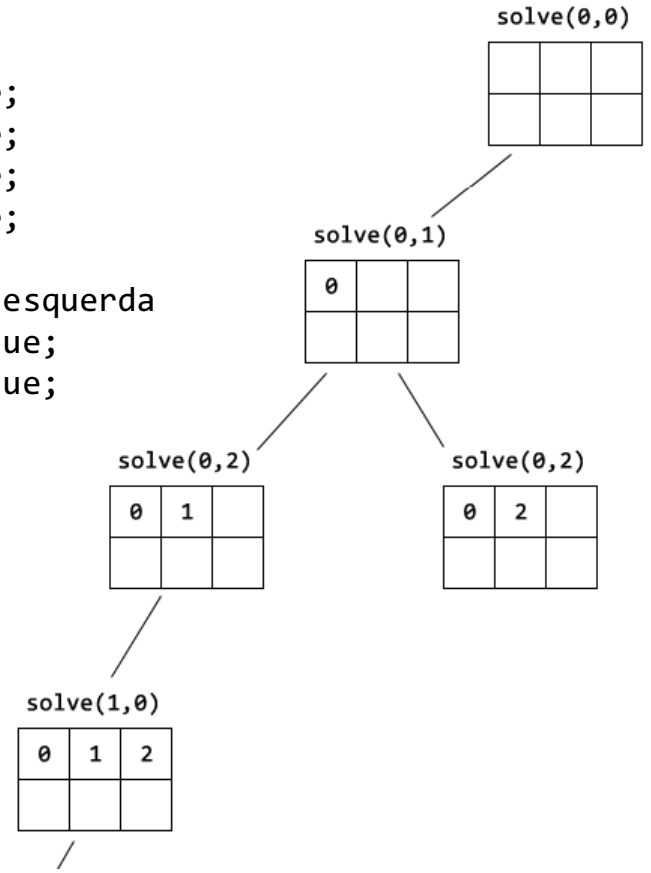
  n, m <= 6 sugerem uma solução com busca exaustiva.

  Podemos resolver o problema com Backtracking com algumas otimizações.

  Testamos posição por posição do retângulo n x m colocar as peças que ainda não foram utilizadas. Elas devem encaixar com as peças anteriores e também com a borda do tabuleiro se for o caso.

F – Quebra-cabeça

```
bool solve(int i, int j)
{
    if(fim(i,j)) return 1;

    for(int k = 0; k < numpecas; k++)
    {
        if(used[k]) continue;      // used e tab são variáveis globais

        // confere se a peça encaixa nas bordas
        if(i == 0   && pecas[k][SUPERIOR] != 'F') continue;
        if(i == n-1 && pecas[k][INFERIOR] != 'F') continue;
        if(j == 0   && pecas[k][ESQUERDA] != 'F') continue;
        if(j == m-1 && pecas[k][DIREITA]  != 'F') continue;

        // confere se a peça encaixa com a peça acima e à esquerda
        if(i > 0 && !encaixa(CIMA, k, tab[i-1][j])) continue;
        if(j > 0 && !encaixa(LADO, k, tab[i][j-1])) continue;

        used[k] = 1;
        tab[i][j] = k;

        if(solve(prox(i,j))
            return 1;

        used[k] = 0;
    }
    return 0;
}
```



F – Quebra-cabeça

- Essa versão ainda é muita lenta => TLE.
- Podemos aplicar duas otimizações:

  1. Não colocar peças com lado ‘F’ no meio do tabuleiro (na versão anterior conferimos “se é borda, tem que ser F”; passamos também a conferir “se não é borda, tem que ser != F”).

  2. Redundância de peças: embora existam 81 peças diferentes possíveis (3^4), algumas peças devem ser repetidas.

  Por exemplo, para as peças do meio do tabuleiro, cada lado deve ser ‘O’ ou ‘I’ = 2^4 = 16 peças possíveis. No caso 6x6, há 5x5 = 25 peças do meio, ou seja algumas peças se repetem.

  Assim, cada recursão marca em um vetor quais peças já foram testadas e não testa peças iguais mais de uma vez.

F – Quebra-cabeça

```
bool solve(int i, int j)
{
    if(fim(i,j)) return 1;

    bool visto[81] = {};

    for(int k = 0; k < numpecas; k++)
    {
        if(used[k] || visto[peca[k]]) continue;
        visto[peca[k]] = 1;

        if((i == 0)   ^ (pecas[k][SUPERIOR] == 'F')) continue; // uso do XOR
        if((i == n-1) ^ (pecas[k][INFERIOR] == 'F')) continue;
        if((j == 0)   ^ (pecas[k][ESQUERDA] == 'F')) continue;
        if((j == m-1) ^ (pecas[k][DIREITA]  == 'F')) continue;

        if(i > 0 && !encaixa(CIMA, k, tab[i-1][j])) continue;
        if(j > 0 && !encaixa(LADO, k, tab[i][j-1])) continue;

        used[k] = 1;
        tab[i][j] = k;

        if(solve(prox(i,j))
            return 1;

        used[k] = 0;
    }
    return 0;
}
```

B - Palavra

- **Problema**

  Dada uma palavra de tamanho até 16, e 8 regras descrevendo uma transformação de um caractere i da palavra em função dos caracteres i-2, i e i+1, escrever a palavra após *s* transformações.

- **Solução**

  Para simplificar a implementação e tornar o algoritmo mais eficiente, como usamos apenas os caracteres a e b, podemos considerá-los como bits: a = 0 e b = 1 (ou vice-versa).

  Assim, aab = 001 = 1, bab = 101 = 5, etc.

  Desse modo, podemos armazenar as regras em um vetor de 8 posições, de acordo com o número binário formado pelos 3 caracteres que determinam a regra.

B - Palavra

- Para realizar uma transformação, temos:

```
for(int i = 0; i < n; i++)
{
    int r = bin(s[i-2+n) % n],
                s[i],
                s[(i+1) % n]);

    novaS[i] = regras[r];
}
strcpy(s, novaS);
```

- O número de transformações é muito alto (n <= 2*10^9) para realizarmos as transformações uma a uma.
- Como o tamanho da palavra vai até 16, há apenas 2^16 palavras possíveis. A próxima palavra a ser gerada depende unicamente da palavra atual.
- Assim, em no máximo 2^16 = 65.536 transformações, veremos uma palavra que já foi vista = ciclo de transformações.

B - Palavra

- Para cada palavra, associamos a ela o número binário onde a = bit 0 e b = bit 1.

```
int palavra_bin()
{
    int h = 0;
    for(int i = 0; i < n; i++)
        h = (h << 1) | (pal[i] == ‘b’);
    return h;
}
```

- Em um vetor de 65.536 posições, guardamos a iteração em que cada palavra foi vista.
- Quando encontrarmos uma palavra de novo, a diferença entre a iteração atual e a iteração em que foi vista pela primeira vez é o tamanho do ciclo.
- Agora recalculamos s: s = (s – iteracoes) % ciclo.

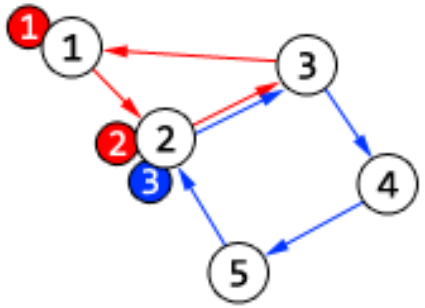
B - Palavra

- Até achar o ciclo são necessárias no máximo 65.536 iterações.
- Após, o novo s terá tamanho máximo de 65.536 (pois pulamos todos os ciclos completos até o valor de s).
- Então teremos que realizar no máximo cerca de 130 mil transformações, cada qual transformando uma palavra de 16 caracteres, o que é rápido o suficiente.
- Não esquecer de, na hora de imprimir a saída, imprimir a menor rotação cíclica da palavra lexicograficamente.

J - Fofoca

- **Problema**

  São dadas N linhas de ônibus, descritas como sequências cíclicas de pontos de ônibus, e D motoristas, cada qual percorrendo uma linha iniciando em um dado ponto de ônibus. Em um instante de tempo, todos os motoristas avançam para o próximo ponto em sua linha. Ao se encontrar em um mesmo ponto, dois motoristas trocam suas novidades. Passado tempo suficiente, todos os motoristas saberão as novidades de todos os outros?



J - Fofoca

- **Solução**
- Podemos modelar os motoristas em um grafo não-direcionado, onde existe aresta de A para B se os motoristas A e B se encontram em algum momento.
- Para determinar se os motoristas A e B se encontram em algum momento, simulamos seus caminhos: começamos com (p_a, p_b), onde p_a e p_b são os pontos iniciais de A e B. Agora avançamos para $(prox(p_a), prox(p_b))$, e assim por diante. Se em algum momento os dois estiverem no mesmo ponto, eles se encontram. Caso visitemos um par já visitado, então encontramos um ciclo e eles não se encontram.

J - Fofoca

- Como há no máximo 50 pontos de ônibus, teremos no máximo 50*50 = 2500 iterações do laço.
- Fazemos isso para todo par de motoristas = $O(D^2S^2)$ = 30*30*50*50 = 2,25 * 10^6.

```
bool se_encontram(int a, int b)
{
    bool visto[50][50] = {};
    int pa = inicial[a], pb = inicial[b];

    while(!visto[pa][pb])
    {
         if(pa == pb) return 1;

         visto[pa][pb] = 1;
         pa = prox(a, pa);
         pb = prox(b, pb);
    }

    return 0;
}
```

J - Fofoca

- Com o grafo montado, se existe um caminho entre dois motoristas então um saberá das novidades do outro.
- Para que todos saibam as novidades de todos, deve haver caminho de todos para todos, isto é, o grafo deve ter apenas 1 componente conexo, o que pode ser conferido com BFS/DFS/DSU.