# Soluções da maratona de programação da SBC 2013

Autor: Gabriel Dalalio

## Introdução

Neste texto serão apresentadas dicas para as soluções dos problemas da prova da sub-regional brasileira da maratona de programação de 2013. Além da discussão dos algoritmos esperados, o texto contém dicas de implementação em C++, links úteis para estudo e referências a problemas de outras competições que são relacionados aos temas das questões da prova.

O enunciado das questões da prova pode ser obtido no seguinte link:

http://maratona.ime.usp.br/prim-fase2013/maratona.pdf

# Problema A
# Zerinho ou Um

## Solução

Esse era o problema mais fácil da prova. Ele pode ser resolvido com a implementação de uma lógica simples.

A seguir, um exemplo de uma implementação curta para o problema:

```
scanf("%d %d %d",&a,&b,&c);
printf("%c\n",(a+b+c)%3==0?'*':'A'+(a==c)+2*(a==b));
```

## Links para estudo

http://www.cplusplus.com/doc/tutorial/control/

## Problemas relacionados

http://br.spoj.com/problems/PAR/

# Problema B
# Balão

## Solução

O problema nos dá $N$ segmentos e também $C$ pontos de partida para balões que iram subir e deslizar nos segmentos. Para cada balão temos de dizer qual é o seu destino.

Os valores de $N$ e $C$ podem ir até $10^5$, então já sabemos que teremos de ser bem eficientes para descobrir as respostas das consultas, o programa não passará no tempo se tentarmos simular o caminho do balão para cada consulta.

Para a resolução do problema é crucial descobrir para cada consulta qual é o primeiro segmento que o balão encosta (ou se ele escapa direto). Também é importante saber para cada segmento o que acontece com o balão depois de tocar esse segmento, para qual segmento o balão vai (ou se ele escapa). Nessa hora vamos precisar de um algoritmo baseado em *line sweep*.

Imagine uma reta vertical passando no plano do problema, vindo desde o menor valor de $x$ e indo até o maior valor. Acontecem 3 tipos de eventos interessantes:

- A reta vertical encosta em um segmento.
- A reta vertical desencosta de um segmento.
- A reta vertical chega em um valor de $x$ de onde um balão é solto.

Se conseguirmos manter o conjunto de segmentos que toca essa reta em um determinado instante durante esse processo, isso pode nos ajudar a descobrir o destino do balão após ser solto e após tocar em um segmento. Além de simplesmente manter esse conjunto, será interessante manter esse conjunto de segmentos ordenado, do segmento que está mais baixo até o segmento mais alto.

A implementação disso não é tão simples, por isso esse texto tentará mostrar alguns detalhes de como fazer isso utilizando C++.

Iremos precisar de estruturas para ponto e segmento. A estrutura de ponto deve possuir dois inteiros, $x$ e $y$. A estrutura de segmento deve possuir dois pontos, $p1$ e $p2$ tal que $p1.x < p2.x$.

Para manter o conjunto de segmentos ordenados de baixo para cima, usaremos o set da STL do C++. Iremos criar nossa própria função de ordenação para o set receber os identificadores dos segmentos e ordená-los para que os segmentos fiquem de baixo para cima no conjunto. Veja uma possível implementação desse set:

```
seg v[MAXN]; // vetor com os segmentos do problema

// produto vetorial
// retorna negativo quando b está abaixo da reta ac
long long cross( ponto a, ponto b, ponto c ){
    long long dx1 = a.x - b.x, dy1 = a.y - b.y;
    long long dx2 = c.x - b.x, dy2 = c.y - b.y;
    return dx1*dy2 - dx2*dy1;
}

// comparacao entre segmentos - segmento mais embaixo vem antes
// só faz sentido quando uma reta vertical pode tocar os dois
// segmentos ao mesmo tempo
bool comp( int a, int b ){
    seg sa=v[a], sb=v[b];
    if( sa.p1.x >= sb.p1.x ){
        return cross(sb.p1,sa.p1,sb.p2)<0;
    }
    else{
        return cross(sa.p1,sb.p1,sa.p2)>0;
    }
}

// set que mantém ordenados os segmentos presentes em uma
// mesma coordenada x durante o line sweep
set<int, bool(*)(int,int)> lines (comp);
```

Com essa estrutura de set, pode-se então jogar os identificadores dos segmentos no conjunto e eles ficarão ordenados de acordo com a altura dos segmentos. Agora vamos ver como lidar com os três tipos de evento:

- 1) Reta vertical encosta em um segmento de número $k$:
    - Insira $k$ no conjunto.
    - Se $v[k].p1.y > v[k].p2.y$, após um balão tocar o segmento $k$, ele irá escapar para o segmento logo acima de $k$ no conjunto.

- 2) Reta vertical desencosta de um segmento de número $k$:
  - Se $v[k].p2.y > v[k].p1.y$, após um balão tocar o segmento $k$, ele irá escapar para o segmento logo acima de $k$ no conjunto.
  - Remover $k$ do conjunto.
- 3) Reta vertical chega ao $x$ de uma consulta:
  - Se o conjunto de segmentos estiver vazio, o balão escapa direto nessa consulta.
  - Caso contrário, o primeiro segmento que o balão encostará é o primeiro segmento presente no conjunto, $*lines.begin()$.

Note que para isso funcionar, se houver empate de coordenada $x$ entre os eventos, o evento tipo 1 deve preceder o evento tipo 3, e o evento tipo 3 deve preceder o tipo 2. O enunciado deixa explícito que não há $x$ em comum entre dois segmentos, então não há empate entre eventos do tipo 1 e 2. Se houvesse $x$ em comum seria necessária uma travessia um pouco mais complicada nos eventos.

Outro detalhe é como pegar um segmento que está no conjunto e descobrir o segmento acima dele. Isso pode ser feito da seguinte maneira:

```
int segmento_acima( int id ){
    set<int, bool(*)(int,int)>::iterator it;
    it = lines.find(id);
    it++;
    if( it == lines.end() ){
        //não há segmento acima
        return -1;
    }
    else{
        return *it;
    }
}
```

Agora que temos as informações sobre os destinos dos balões após baterem nos segmentos e serem soltos nas consultas, podemos utilizar uma programação dinâmica que dado um segmento inclinado, responde qual seria o destino final do balão. A parte mais difícil já foi, o leitor fica encarregado dos detalhes de implementação dessa parte final da resolução.

## Links para estudo

http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lineSweep

## Problemas relacionados

http://www.spoj.com/problems/RAIN1/

# Problema C
# Chefe

## Solução

É dado um grafo direcionado acíclico e pede-se um programa que realize instruções de trocas de vértices e perguntas sobre antecessores diretos ou indiretos de um vértice.

Vamos discutir primeiro a instrução de troca de vértices. Quando se é realizado uma troca de vértices no grafo, a estrutura do grafo não muda. Durante a execução do programa, só precisamos saber onde está cada pessoa no grafo e qual é a idade de cada vértice. Podemos manter dois vetores, $pos[i]$ igual ao vértice atual da pessoa $i$ e $idade[i]$ sendo a idade da pessoa no vértice $i$. Dessa maneira, a troca de duas pessoas $a$ e $b$ pode ser feito em tempo $O(1)$ como mostrado abaixo:

```
swap(idade[pos[a]],idade[pos[b]]);
swap(pos[a],pos[b]);
```

Inicialmente, pode-se fazer $pos[i] = i$ para todo $i$.

Agora serão discutidas duas soluções para a outra instrução.

## Solução 1

Para implementar a instrução de pergunta, pode-se realizar uma busca para cada instrução. Isso pode ser feito usando busca em largura (BFS) ou busca em profundidade (DFS) atravessando as arestas no sentido contrário. A complexidade para cada query será de $O(N+M)$.

## Solução 2

Como já foi dito, as operações de troca de vértices não alteram a estrutura do grafo. Com isso, podemos pré-calcular todos os antecessores dos vértices do grafo.

A relação de antecessor é transitiva, se A é antecessor de B e B é antecessor de C, então A é antecessor de C. Para achar todos os

antecessores de um vértice, precisamos calcular o fecho transitivo da relação (ver *transitive closure* nos links para estudo). Para calcular o fecho transitivo pode-se utilizar o algoritmo de Floyd-Warshall.

Inicialmente, caso exista aresta de $A$ para $B$, faz-se $antecessor[A][B]=1$, caso contrário $antecessor[A][B]=0$. Após isso, utiliza-se o Floyd-Warshall como mostrado abaixo:

```
for( k=1 ; k<=n ; k++ ){
    for( i=1 ; i<=n ; i++ ){
        if( antecessor[i][k] ){
            for( j=1 ; j<=n ; j++ ){
                antecessor[i][j] |= antecessor[k][j];
            }
        }
    }
}
```

Após executar esse trecho de código, $antecessor[A][B]$ é igual a 1 se e somente se $A$ é antecessor de $B$. Possuindo essa tabela, cada instrução de pergunta pode ser feita em $O(N)$ iterando-se por todos os vértices e verificando quais deles são antecessores do vértice perguntado.

## Links para estudo

http://en.wikipedia.org/wiki/Depth-first_search
http://en.wikipedia.org/wiki/Breadth-first_search
http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=graphsDataStrucs1
http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
http://en.wikipedia.org/wiki/Transitive_closure

## Problemas relacionados

http://br.spoj.com/problems/DEPENDEN/

# Problema D
# Máquina dobradora

## Solução

É dado uma fita de $N$ números em um estado inicial. O problema nos pede para dizer se após algumas dobras é possível se chegar em um estado final.

Deve-se atentar ao fato que o limite para $N$ é igual a 15. Esse valor baixo nos sugere que esse problema pode ser resolvido com força bruta, testando todas as formas de dobrar recursivamente. É possível se calcular com uma função recursiva que o número de maneiras de dobrar uma fita de tamanho 15 é na ordem de um milhão.

Dependendo da eficiência do programa para calcular todas essas dobras, talvez seja necessário algum corte na força bruta, como por exemplo:

- Se a soma dos números da fita inicial for diferente da soma da fita final, a resposta é *N*.
- Não adianta dobrar mais vezes a fita se ela já está menor que o estado final.
- Após uma dobra, os números só podem aumentar. Se o elemento máximo da fita passar do elemento máximo do estado final, também se pode realizar um corte na busca.

## Links para estudo

http://en.wikipedia.org/wiki/Brute-force_search

## Problemas relacionados

http://www.urionlinejudge.com.br/judge/problems/view/1395

# Problema E
# Mergulho

## Solução

O problema pedia para achar quais números inteiros de 1 a $N$ não apareciam na entrada. O problema poderia ser resolvido utilizando um vetor com $N$ posições para marcar os números que apareceram na entrada. Se o número $i$ aparece na entrada, pode-se fazer $v[i]=1$. Dessa maneira é possível obter uma solução com complexidade de tempo $O(N)$.

É possível também utilizar a estrutura do *set* da STL do C++ para obter uma solução na complexidade $O(N.\log N)$, pois o *set* possui funções de inserção e busca em tempo $O(\log N)$.

## Links para estudo

http://www.cplusplus.com/doc/tutorial/arrays/
http://www.cplusplus.com/reference/set/set/

## Problemas relacionados

Peça perdida – OBI 2007 Programação nível 1 fase 1
http://olimpiada.ic.unicamp.br/passadas/OBI2007/res_fase1_prog/programacao_n1/tarefas_solucoes

# Problema F
# Triângulos

## Solução 1

São dados $N \leq 10^5$ pontos em uma circunferência e pede-se quantos triângulos equiláteros é possível formar com esses pontos. Primeiro fato que temos de notar é o limite para o $N$. Não podemos, por exemplo, testar todos os trios de pontos ou todos os pares de pontos, nesse caso o programa excederá o tempo limite.

Uma informação valiosa nesse problema é que o triângulo equilátero determina 3 arcos iguais no círculo, ou seja, podemos tomar a soma de todos os arcos e dividir por 3 para achar o tamanho do arco que os pontos de triângulo equilátero determinam. Se a soma não for divisível por 3, a resposta para o problema é nula.

Seja o vetor $soma[]$ definido por $soma[i] = \sum_{k=1}^{i} X_i$, ou seja, o vetor soma representa a soma acumulada dos arcos da entrada. Para encontrar um triângulo equilátero, devemos encontrar os trios $i$, $j$, $k$ tais que:

$$soma[j] = soma[i] + (soma[N]/3)$$
$$soma[k] = soma[i] + 2*(soma[N]/3)$$

Podemos testar todos os valores possíveis $i$, são $N$ possibilidades. Para achar $j$ e $k$ podemos utilizar busca binária ou a estrutura de *set* da STL, resultando em uma solução com complexidade de tempo $O(N.\log N)$.

## Solução 2

Podemos melhorar um pouco o último passo da solução 1 e conseguir um algoritmo que roda em tempo linear. A medida que tentamos as possibilidades para o $i$ em ordem crescente, é possível fazer a busca por $j$ e $k$ apenas incrementando seus valores.

Veja a seguir um trecho de um código que mostra como implementar isso:

```
len=soma[n]/3;
j=k=0;
for( i=0 ; soma[i]+2*len <= soma[n-1] ; i++ ){
    while( soma[j] < soma[i]+len ){
        j++;
    }
    while( soma[k] < soma[i]+2*len ){
        k++;
    }
    if( soma[j]==soma[i]+len && soma[k] == soma[i]+2*len ){
        resp++; //achou um triangulo
    }
}
```

Perceba que apesar dos loops aninhados, esse algoritmo é linear no tempo, pois as variáveis $i$, $j$, $k$ são incrementadas no máximo $N$ vezes.

## Links para estudo

http://www.cplusplus.com/reference/set/set/
http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binarySearch

## Problemas relacionados

http://br.spoj.com/problems/POLIGONO/

# Problema G
# Linhas de contêineres

## Solução 1

É dada uma matriz $LxC$ com elementos embaralhados e pede-se o número mínimo de movimentos de troca de linhas ou colunas para desembaralhar a matriz.

A primeira solução que será apresentada é construtiva, descreve como realizar vários movimentos necessários até que se descubra que a matriz está desembaralhada ou que a tarefa é impossível.

Primeira coisa é que podemos achar o elemento igual a 1 e trocar sua coluna com a primeira coluna e sua linha pela primeira linha. A partir disso, não será mais possível mexer na primeira linha e nem na primeira coluna. A primeira linha deve conter os números de 1 a $C$ e a primeira coluna deve conter todos os números que possuem resto 1 por $C$. Caso contrário, não haverá solução para o teste.

Agora vamos arrumar a primeira linha e a primeira coluna. Enquanto houver elementos em posições erradas na primeira linha, troque duas colunas para colocar pelo menos um elemento a mais em sua posição correta. Faça o mesmo com a primeira coluna, enquanto houver elementos errados troque linhas.

Após isso, se a matriz toda ficar na posição correta, imprima o número de passos que foram feitos. Caso contrário não há solução, pois qualquer troca de coluna irá desarrumar a primeira linha e qualquer troca de linha irá desarrumar a primeira coluna.

O número máximo para a resposta é $L+C-2$ e cada operação de troca de fileira pode ser feita em tempo linear no número de elementos movidos. Dessa maneira, essa solução possui complexidade $O(LC)$.

## Solução 2

A segunda solução que será apresentada não faz nenhuma alteração na matriz para descobrir a resposta. Analisaremos as permutações presentes na matriz e a solução será encontrada a partir da contagem de ciclos de permutação (ver links para estudo).

Se dois elementos estão na mesma coluna, eles sempre ficarão na mesma coluna após as operações de troca de fileiras. O mesmo acontece com dois elementos que estão na mesma linhaa, eles sempre estarão na mesma linha. Se verificarmos que todos os elementos $X[i][j]$ pertencem à mesma linha do elemento $X[i][j-1]$ e à mesma coluna do elemento $X[i-1][j]$, então há resposta para o problema.

O valor da resposta será dado pela soma $L+C$ menos o número de ciclos de permutação de uma linha e de uma coluna (todas as linhas equivalem à mesma permutação, o mesmo ocorre para as colunas). A solução também pode ser implementada na mesma complexidade da solução 1.

## Links para estudo

http://mathworld.wolfram.com/PermutationCycle.html

## Problemas relacionados

http://br.spoj.com/problems/CADEIR09/

# Problema H
# Ônibus

## Solução

O problema pede a contagem de maneiras que se pode formar uma fila com dois tipos de ônibus. Existem $K$ ônibus de tamanho 5 e $L$ ônibus de tamanho 10. Como $N$ é múltiplo de 5, podemos começar o problema dividindo $N$ por 5 e considerar que os ônibus tem tamanho 1 e 2.

O próximo passo da solução é obter uma formula recursiva para o problema. Seja $f(n)$ o número de maneiras de formar uma fila de tamanho $n$. Tem-se $f(0)=1$ (uma maneira de formar uma fila vazia) e $f(1)=K$ (só dá para usar um dos $K$ ônibus de tamanho 1). Para $n \geq 2$, pode-se começar a fila por um ônibus de tamanho 1 ou 2. Se escolhermos o começo pelo ônibus de tamanho 1, há $K$ possibilidades para escolher o tipo do ônibus e há $f(n-1)$ possibilidades para escolher o resto da fila, resultando em $K.f(n-1)$ maneiras. De modo similar, para o começo com ônibus de tamanho 2 haverá $L.f(n-2)$ maneiras. Com isso, $f(n)=K.f(n-1)+L.f(n-2)$ para $n \geq 2$.

Descoberta a recorrência, poderíamos utilizar programação dinâmica para obter uma solução na complexidade de tempo $O(N)$. Porém o valor de $N$ pode ser muito alto, isso não seria suficiente para passar no tempo.

A recorrência encontrada é linear, assim como a sequência de Fibonacci. Vamos aprender agora como calcular $f(n)$ rapidamente utilizando exponenciação de matrizes. Repare que a seguinte identidade é válida:

$$\begin{bmatrix} f(n-2) & f(n-1) \end{bmatrix} . \begin{bmatrix} 0 & L \\ 1 & K \end{bmatrix} = \begin{bmatrix} f(n-1) & f(n) \end{bmatrix}$$

Com essa fórmula, pode-se chegar no valor de $f(n)$ a partir dos valores de $f(0)$ e $f(1)$ da seguinte maneira:

$$[f(0) \quad f(1)].\begin{bmatrix}0 & L\\ 1 & K\end{bmatrix} = [f(1) \quad f(2)]$$

$$[f(0) \quad f(1)].\begin{bmatrix}0 & L\\ 1 & K\end{bmatrix}^2 = [f(1) \quad f(2)].\begin{bmatrix}0 & L\\ 1 & K\end{bmatrix} = [f(2) \quad f(3)]$$

$$\vdots$$

$$[f(0) \quad f(1)].\begin{bmatrix}0 & L\\ 1 & K\end{bmatrix}^n = [f(n) \quad f(n+1)]$$

Para obter a resposta do problema, basta agora que tenhamos um método rápido para fazer exponenciação de matrizes. Existe um método rápido para isso, veja o link para estudo sobre e*xponentiation by squaring.* Segue abaixo um pseudo código que mostra como fazer a exponenciação com $O(\log N)$ multiplicações.

```
matriz exponencial( matriz m, long long expoente ){
    if( expoente==0 ){
        return identidade;
    }
    else{
        matriz ret = exponencial(m,expoente/2);
        ret = ret*ret;
        if( expoente%2==1 ){
            ret = ret*m;
        }
        return ret;
    }
}
```

Não se esqueçam de sempre tirar resto por $10^6$ nas operações!

## Links para estudo

http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=dynProg
http://en.wikipedia.org/wiki/Exponentiation_by_squaring

## Problemas relacionados

http://www.urionlinejudge.com.br/judge/problems/view/1422
https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8&category=328&page=show_problem&problem=2304

# Problema I
# Remendo

## Solução

São dadas $N$ posições de furos em um pneu de bicicleta. Com remendos de tamanho $T_1$ e $T_2$, quer se cobrir todos os furos utilizando o mínimo total de remendo.

Um fato interessante para a busca da solução ótima é que toda solução pode ser transformada em um uma solução onde todos os remendos cobrem um furo em uma de suas extremidades. Isso ocorre porque podemos transladar os remendos até que isso aconteça e sem descobrir nenhum furo.

Uma das complicações do problema é o fato do pneu ser circular. Vamos primeiro esquecer esse fato e fingir que o pneu é uma tira esticada. Imagine o primeiro furo $F_1$ dessa tira. Esse furo deverá ser coberto por um remendo de tamanho $T_1$ ou $T_2$. Além disso, podemos colocar os remendos com uma das pontas exatamente em $F_1$. Após colocar um remendo, podemos esquecer todos os pontos que ficaram cobertos. Dessa maneira, a solução recai sobre o mesmo problema inicial, mas com um tira com menos furos. Quando isso acontece, pode-se tentar solucionar o problema com programação dinâmica.

Seja a função $cobertura(i)$ que retorna qual é o mínimo de remendo para se cobrir os furos de 1 a $i$. Pode-se obter o seguinte pseudocódigo para essa função:

```
int cobertura( int i ){
    int resp1, resp2;
    if( f[i]-f[1] <= t1 ){
        resp1 = t1;
    }
    else{
        int j = maior j tal que f[i]-f[j] > t1;
        resp1 = cobertura(j) + t1;
    }
    if( f[i]-f[1] <= t2 ){
        resp2 = t2;
    }
    else{
        int j = maior j tal que f[i]-f[j] > t2;
        resp2 = cobertura(j) + t2;
    }
    return min(resp1,resp2);
}
```

Esse pseudocódigo não mostra o processo de memoização, mas ele deve ser feito, é necessário salvar as respostas de $cobertura(i)$ para que a função não calcule a mesma coisa várias vezes.

Com isso resolvemos o problema para uma tira, não para o pneu. Mas o pneu tem $N$ possibilidades para ser transformado em uma tira, pode-se cortar o pneu entre os $N$ furos. Deve-se testar todas as tiras e pegar a menor resposta. Como a função cobertura tem complexidade $O(N)$, pode-se obter a resposta para todas as tira em $O(N^2)$. Outro detalhe é que os valores de $j$ que a função cobertura usa podem ser pré-calculados em $O(N^2)$, $O(N.\log N)$ ou $O(N)$ utilizando loops ou usando técnicas que foram usadas nas soluções do problema "F – Triângulos".

## Links para estudo

http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=dynProg

## Problemas relacionados

https://icpcarchive.ecs.baylor.edu/index.php?option=onlinejudge&page=show_problem&problem=1152

# Problema J
# Caminhão

## Solução 1

O problema nos dá um grafo de $N$ vértices e $M$ arestas. Nesse grafo é necessário obter a resposta de $S$ consultas. Escolhidas duas cidades, deve-se imprimir o valor máximo que pode ter a menor aresta de um caminho entre as duas cidades.

É possível modificar a função de distância dos algoritmos de Dijkstra ou Floyd-Warshall para eles obterem as respostas das consultas do problema. Utilizando o algoritmo de Dijkstra, seria necessário executá-lo para cada consulta, obtendo solução na complexidade $O(S.(N+M).\log N)$ implementando o algoritmo com fila de prioridade. Usando o Floyd-Warshall, é possível pré-processar todas as consultas, a complexidade seria $O(N^3+S)$. Nenhuma dessas tentativas passa no tempo.

Uma alternativa seria tentar resolver o problema por uma busca linear na resposta. Para cada consulta entre as cidades $A$ e $B$, começasse um grafo sem arestas. A cada passo, adiciona-se ao grafo a maior aresta que ainda não foi colocada. Quando existir caminho entre $A$ e $B$, a resposta para a consulta é dada pelo valor da última aresta adicionada. A existência do caminho poderia ser verificada utilizando algoritmo de *union-find*. Isso possibilita a resolução em complexidade $O(S.M)$, que ainda não é suficiente para passar.

Analisando melhor essa última resolução, não precisamos adicionar as arestas ao grafo que não liguem vértices em componentes conexas diferentes. Dessa maneira, o algoritmo executado seria idêntico ao algoritmo de Kruskal para achar a árvore geradora máxima do grafo. Isso nos leva a um fato muito importante para a resolução do problema: as respostas de todas as consultas permanecem as mesmas se analisarmos apenas a árvore geradora máxima do grafo. Ou seja, inicialmente podemos obter essa árvore e pensar agora como responder as consultas para a árvore.

A árvore geradora máxima normalmente é obtida pelo algoritmo de Kruskal ou pelo algoritmo de Prim. No caso da solução que será discutida agora, o uso do algoritmo de Prim será mais prático, pois ele encontra a árvore geradora já determinando um pai para cada nó. Com o Kruskal, é necessário utilizar uma busca como a DFS para a obtenção dos pais.

Falta pensar como proceder as consultas na árvore. A grande vantagem da árvore é que só há um caminho entre um par de vértices. Para cada consulta é necessário achar esse caminho e dizer a menor aresta. Isso pode ser feito com uma busca para cada consulta, o que leva a uma complexidade de $O(S.N)$, que ainda não passa no tempo.

Para encontrar rapidamente o caminho entre dois vértices numa árvore pode-se utilizar a técnica de encontrar o primeiro ancestral comum entre dois vértices em tempo logarítmico. Essa técnica é explicada no artigo do site do *Topcoder* apresentado na seção de links para estudo. Mesmo assim, esse texto explicará a implementação da aplicação dessa técnica especificamente para encontrar a menor aresta em um caminho na árvore.

Após encontrar a árvore geradora máxima, deve-se encontrar a distância de cada vértice até a raiz, ou seja, a altura de cada vértice na árvore. Pode-se escolher qualquer vértice como raiz, sua altura será igual 0. Deve-se calcular também uma tabela $t[i][j]$ com dois valores. Seja o caminho formado por $2^j$ arestas que começa no vértice $i$ e vai em direção à raiz. O vértice final do caminho será igual a $t[i][j].pai$ e a menor aresta do caminho será $t[i][j].val$.

Os valores de $t[i][0]$ são calculados facilmente a partir da construção da árvore. O código a seguir mostra como calcular o resto da tabela com complexidade de tempo constante para cada elemento.

```
void precomputa(){
    int i, j, aux;
    for( j=1 ; (1<<j)<n ; j++ ){
        for( i=1 ; i<=n ; i++ ){
            if( (1<<j) <= altura[i] ){
                aux=t[i][j-1].pai;
                t[i][j].pai = t[aux][j-1].pai;
                t[i][j].val = min(t[i][j-1].val,t[aux][j-1].val);
            }
        }
    }
}
```

Com a tabela pronta, é possível utilizá-la nas consultas para obter uma complexidade logarítmica para achar a menor aresta entre dois vértices. Sendo $\log n$ o maior número tal que $(1 << \log n) < n$, veja o código a seguir que encontra a menor aresta entre dois vértices $A$ e $B$.

```
int menor_aresta( int a, int b ){
    int i, ret = INF;
    if( altura[a] != altura[b] ){
        if( altura[a] < altura[b] ){
            swap(a,b);
        }
        for( i=logn ; i>=0 ; i-- ){
            if( altura[b] <= altura[a] - (1<<i) ){
                ret=min(ret,t[a][i].val);
                a=t[a][i].pai;
            }
        }
    }
    if( a!=b ){
        for( i=logn ; i>=0 ; i-- ){
            if( (1<<i) <= altura[a] && t[a][i].pai!=t[b][i].pai ){
                ret=min(ret,min(t[a][i].val,t[b][i].val));
                a=t[a][i].pai;
                b=t[b][i].pai;
            }
        }
        ret=min(ret,min(t[a][i].val,t[b][i].val));
        a=b=t[a][0].pai;
    }
    return ret;
}
```

Ao final da função, as variáveis $a$ e $b$ sempre acabam iguais ao primeiro ancestral comum dos dois vértices passados para a função, mas nessa questão a identidade do ancestral não precisou ser utilizada. Ao final do cálculo da arvore geradora máxima, do cálculo da tabela e do cálculo das

respostas das consultas, obtemos um algoritmo com complexidade de tempo igual a $O(M.\log M + N.\log N + S.\log N)$, que é suficiente para passar no tempo limite do problema.

## Solução 2

A segunda solução que será discutida agora leva em conta o fato de que não precisamos responder cada consulta no mesmo momento em que fazemos leitura da mesma na entrada. Por exemplo, pode-se fazer a leitura de todas as consultas e só após isso responder todas as consultas na ordem dada na entrada.

Será proposta aqui uma maneira de se modificar o algoritmo de *union-find* durante a execução do algoritmo de Kruskal para que ao final da execução todas as consultas já estejam respondidas.

Durante a execução do algoritmo de Kruskal, manteremos para cada componente conexa um conjunto de identificadores de consultas. Inicialmente, o grafo começa sem arestas, cada vértice representa uma componente diferente e sendo assim cada um possui um conjunto de identificadores.

Após a leitura de todas as consultas, se a consulta de número $i$ é dada pelos vértices $a$ e $b$, adiciona-se o inteiro $i$ aos conjuntos dos vértices $a$ e $b$. É dessa maneira então que iremos começar o nosso algoritmo, vamos analisar agora como fazer a junção das componentes e responder as consultas.

O algoritmo de Kruskal irá adicionando ordenadamente as arestas no grafo, da maior para a menor. As arestas que realmente serão inseridas juntam duas componentes conexas diferentes. Quando essa junção ocorrer, iremos juntar também o conjunto de identificadores dessas componentes. Se uma aresta juntar duas componentes que contém um mesmo identificador de consulta, a resposta dessa consulta será igual ao valor da aresta. Com isso em mente, veja a seguir o pseudocódigo da modificação do algoritmo de *union-find* para resolver o problema.

```
int pai[MAXN]; //inicialmente pai[i]=i
int resp[MAXS]; //resposta das consultas
par query[MAXS]; //cada consulta possui a e b

//encontra o representante da componente do vértice a
int find( int a ){
    if( pai[a]==a ){
        return a;
    }
    return pai[a]=find(pai[a]);
}

//funcao para adicionar a aresta a<->b com custo val
void join( int a, int b, int val ){
    int i=find(a);
    int j=find(b);
    if( i!=j ){
        if( conjunto de i é menor que o conjunto de j ){
            swap(i,j);
        }
        pai[j]=i;
        Para todo elemento k no conjunto de j{
            if( consulta k ainda não foi respondida ){
                if( find(query[k].a)==find(query[k].b) ){
                    resp[k] = val;
                }
            }
            inserir k no conjunto de i;
        }
        limpar conjunto de j;
    }
}
```

Após usar a função *join* para todas as arestas do grafo, indo da maior para a menor, todas as consultas estarão respondidas no vetor *resp*. Agora falta analisar a complexidade do algoritmo.

Os conjuntos de identificadores só precisam permitir as operações de inserção, travessia por todo o conjunto e deleção de todo o conjunto. Usando a estrutura de *vector* da STL ou usando uma lista ligada, a inserção pode ser feita em tempo constante e as outras duas operações podem ser feitas em tempo linear no número de elementos do conjunto.

Em uma análise ingênua, poder-se-ia dizer que a complexidade da função $join$ é de $O(S)$, levando o algoritmo ter complexidade $O(N.S)$, porém temos de analisar melhor quantas operações pode-se ter no total.

Toda vez que um identificador muda de conjunto, ele vai para um conjunto maior ou de mesmo tamanho que o conjunto em que ele estava. Após a passagem de todos os elementos do conjunto de $j$ para o conjunto de $i$, os elementos que estavam no conjunto de $j$ passam a estar num conjunto que é pelo menos duas vezes maior que o conjunto onde eles estavam. Isso significa que cada identificador de consulta é movido no máximo $\log_2(2S)$ vezes, pois nenhum conjunto pode ter tamanho maior que $2.S$.

Após essa análise, descobrimos que o algoritmo pode ser executado em complexidade de tempo igual a $O(M.\log M + S.\log S)$, o que inclui a ordenação das arestas e a execução do algoritmo de Kruskal com o *union-find* modificado.

A vantagem da solução 2 é que sua implementação pode ficar bem menor que a implementação da solução 1, podendo ter por volta de 80 linhas de código em C++. Um detalhe de implementação que pode ser destacado é que durante a execução da função $join$, pode-se descartar identificadores de consultas já respondidas para diminuir o tempo médio de execução.

## Links para estudo

http://pt.wikipedia.org/wiki/Algoritmo_de_Prim
http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Lowest Common Ancestor (LCA)
http://pt.wikipedia.org/wiki/Algoritmo_de_Kruskal
http://en.wikipedia.org/wiki/Disjoint-set_data_structure

## Problemas relacionados

https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2297
http://br.spoj.com/problems/ANTS10/
http://www.spoj.com/problems/KOICOST/