UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS CHAPECÓ
CURSO DE CIÊNCIA DA COMPUTAÇÃO

EMERSON DALLAGNOL

RESOLUÇÃO DE PROBLEMAS ENVOLVENDO A MARATONA DE PROGRAMAÇÃO DA SBC DE 2013 E 2014

CHAPECÓ
2021

EMERSON DALLAGNOL

RESOLUÇÃO DE PROBLEMAS ENVOLVENDO A MARATONA DE PROGRAMAÇÃO DA SBC DE 2013 E 2014

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.

Orientador: Prof. Dr. Guilherme Dal Bianco

CHAPECÓ

2021

Bibliotecas da Universidade Federal da Fronteira Sul - UFFS

Dallagnol, Emerson
Resolução de problemas envolvendo a maratona de programação da SBC de 2013 e 2014 / Emerson Dallagnol. -- 2022.
34 f.

Orientador: Dr. Guilherme Dal Bianco

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal da Fronteira Sul, Curso de Bacharelado em Ciência da Computação, Chapecó, SC, 2022.

1. Maratona de Programação. I. Bianco, Guilherme Dal, orient. II. Universidade Federal da Fronteira Sul. III. Título.

Elaborada pelo sistema de Geração Automática de Ficha de Identificação da Obra pela UFFS com os dados fornecidos pelo(a) autor(a).

EMERSON DALLAGNOL

RESOLUÇÃO DE PROBLEMAS ENVOLVENDO A MARATONA DE PROGRAMAÇÃO DA SBC DE 2013 E 2014

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.

Orientador: Prof. Dr. Guilherme Dal Bianco

Este trabalho de conclusão de curso foi defendido e aprovado pela banca em: 22/12/2021

BANCA EXAMINADORA:

Dr. Guilherme Dal Bianco - UFFS

Dr. Denio Duarte - UFFS

Dr. Emílio Wuerges - UFFS

RESUMO

As maratonas de programação são excelentes instrumentos para motivar os alunos a estudar tópicos de programação e problemas computacionais, que não são abordados ou aprofundados nos cursos de computação. Os problemas encontrados nas maratonas exigem que o estudante aplique os conhecimentos das diversas áreas da computação e desenvolva soluções eficientes, dadas as limitações do problema. Neste trabalho serão apresentados dois problemas referentes a duas participações na etapa nacional da Maratona de Programação da SBC de 2013 e 2014, juntamente com a análise e a resolução. Um dos problemas envolve conhecimentos em números binários e o outro em grafos, sendo que ambos requerem soluções não triviais para executarem dentro do limite de tempo estabelecido.

Palavras-chave: Maratona de programação. Programação competitiva.

ABSTRACT

Competitive programming is a excellent tool to motivate students to study programming topics and computational problems that are not covered or not covered in depth in computing courses. The problems solved in competitive programming require the student to apply their knowledge from different areas of computing, and develop efficient solutions, given the limitations of the problem. In this work, two problems referring to two participations in the national stage of the 2013 and 2014 SBC Programming Contest will be presented, together with the analysis and resolution. One of the problems involves knowledge of binary numbers and the other requires graphs knowledge, both of them require non-trivial solutions to execute within the established time limit.

Keywords: Competitive Programming. Programming Contest.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ANEXOS

SUMÁRIO

1 INTRODUÇÃO

Maratonas de programação são competições entre equipes de programadores para resolver problemas de diversos níveis. Nestas competições, geralmente cada equipe tem 3 integrantes, que devem desenvolver juntos, um programa que resolve cada problema da prova [1]. As maratonas são excelentes instrumentos para motivar os alunos a estudar fora da sala de aula, assuntos que não são abordados ou aprofundados nos cursos de computação, além de melhorar perspectivas de carreira [4]. Durante os treinos, os alunos são instruídos a estudar algoritmos relacionados a grafos (por exemplo fluxo, caminho mínimo e árvore geradora mínima), geometria (por exemplo fecho convexo e operações com vetores), matemática (exponenciação binária, algoritmos com matrizes, Fibonacci, cálculos com números primos e fatoração), strings, estruturas de dados, entre outros. Os problemas encontrados nas maratonas exigem que o estudante aplique os conhecimentos das diversas áreas da computação e desenvolva soluções eficientes, dadas as limitações do problema.

A maratona de programação da Sociedade Brasileira de Computação (SBC) ocorre anualmente desde 1996, e serve como etapa classificatória para as finais mundiais do International Collegiate Programming Contest (ICPC). A maratona SBC de 2013 teve a participação de 586 times de 182 escolas, e 58 times participaram final nacional. A equipe chamada *The Morgans* da UFFS, em que a participação é descrita neste relatório, ficou na 27ª colocação [2]. A maratona de 2014 teve a participação de 643 times de 199 escolas, e 58 times participaram final nacional, e a equipe *The Morgans*, ficou com a 20ª colocação [3].

Neste trabalho serão descritos os problemas C, da final nacional da maratona da SBC de 2013, e o G, da final nacional de 2014. O problema C envolve conhecimentos em teoria dos números. Já no problema G, são utilizados conhecimentos em grafos. Os dois são de dificuldade intermediária, e foram resolvidos pela equipe durante as respectivas maratonas.

1.1 Maratona Nacional SBC de 2013

A final nacional de 2013 contou com 10 problemas, sendo que a equipe *The Morgans* resolveu os problemas C e F. [2]

O problema C, *Counting ones*, é relacionado a números binários, e foi escolhido para ser abordado neste trabalho pois é de dificuldade intermediária, e apresenta uma solução interessante de ser abordada. O problema foi resolvido por 36 equipes. O problema F, *Footbal*, é um

problema *ad hoc* [1], e dá como entrada os resultados de partidas de um time em um campeonato e um número de gols que podem ser adicionados as partidas, e solicitava que seja calculado o número máximo de pontos que o time pode chegar após adicionar estes gols. O problema foi resolvido por 54, das 58 equipes.

Em relação aos problemas que a equipe não resolveu, o problema A, *Attacking rooks*, é baseado no conhecido problema de xadrez das 8 rainhas, e é resolvido utilizando algoritmos de fluxo em grafos. Este problema foi resolvido por 16 equipes. O problema B, *Blogger language*, é resolvido com árvore de segmentos e é um dos mais difíceis da prova, não foi resolvido por nenhuma equipe. O problema D, *Disjoint water supply*, é um problema de grafos e foi resolvido por 9 equipes. O problema E, *Eleven*, envolve uma solução usando programação dinâmica, e foi resolvido por 7 equipes. O problema G, *Go up the Ultras*, é um problema de grafos, e foi resolvido por 15 equipes. Os problemas H (*Hide and seek*) e I (*Inverting Huffman*) não foram resolvidos por nenhuma equipe. O problema J, *Join two kingdoms*, foi resolvido por 10 equipes e também é um problema de grafos.

A equipe campeã desta edição foi da USP - São Carlos, e resolveu os problemas A, C, D, E, F, G e J.

1.2 Maratona Nacional SBC de 2014

A final nacional de 2014 contou com 11 problemas, sendo que a equipe *The Morgans* resolveu os problemas A, C, G, H e I. [3]

O problema A, *Automated Checking Machine*, era *ad hoc* e todas as 58 equipes resolveram o problema em poucos minutos. A solução era simplesmente comparar as duas *strings* que o problema dava como entrada. O problema C, *Counting substhreengs*, dava uma *string* como entrada, e requisitava a quantidade de *substrings* que eram divisíveis por 3. A solução envolvia ler a string mantendo somente o resto da divisão por 3, pois os números poderiam ficar muito grandes. O problema precisava de conhecimentos básicos em teoria dos números, e foi resolvido por 29 equipes. O problema G, *Galaxy collision*, é resolvido com grafos bipartidos, e foi escolhido para ser abordado neste trabalho pois é de dificuldade intermediária e usa conhecimentos relevantes de grafos e otimização. O problema foi resolvido por 20 equipes. O problema H, *Help cupid*, era *ad hoc*, e requeria apenas calcular a maior diferença entre os

[1] Problemas *ad hoc* são os que possuem solução trivial, e não necessitam de nenhum conhecimento específico para serem resolvidos.

time zones dados na entrada. O problema foi resolvido por 40 equipes. O problema I, *Intrepid climber*, e dava como entrada uma árvore e os custos para percorrer os nós em direção a raiz (na outra direção o custo é zero), a saída deveria ser o custo mínimo para percorrer todos os pontos. A solução envolvia conhecimentos em grafos, usando uma DFS (Depth first search) modificada, e o problema foi resolvido por 26 equipes.

Em relação aos problemas que a equipe não resolveu, o problema B, *Black and white stones*, era *ad hoc*, e foi resolvido por 22 equipes. O problema D, *Dividing the names*, é resolvido usando programação dinâmica, e foi resolvido por apenas uma equipe. O problema E, *Even distribution*, foi resolvido por 8 equipes. O problema F, *Fence the vegetables* foi resolvido por apenas duas equipes. O problema J, *Journey through the kingdom*, foi resolvido por apenas uma equipe. O problema K, *Knights of the Round Table*, também foi resolvido por apenas uma equipe.

A equipe campeã desta edição foi da USP, e resolveu todos os problemas exceto o K.

2 RESOLUÇÃO DOS PROBLEMAS

Neste capítulo, serão apresentados 2 problemas aplicados nas finais nacionais das maratonas de programação SBC de 2013 [2] e 2014 [3], juntamente com as suas soluções e explicações. As descrições originais estão disponíveis nos anexos A e C e as soluções em C++ nos anexos B e D.

2.1 Problema 1: 2013 - C - Counting Ones

Neste problema, são dados dois inteiros A e B como entrada, sendo $1 \leq A \leq B \leq 10^{16}$, e o programa precisa retornar o número de uns existentes na representação binária de todos os inteiros de A até B, inclusive. Por exemplo, para a entrada 1 3, as representações binárias da sequência 1-3 são 1, 10 e 11, logo, o resultado é 4.

Ao analisar o problema, percebe-se que não é possível iterar pelo intervalo A-B, fazendo o cálculo dos uns em cada número, pois isso excederia o limite de tempo. Por exemplo, a iteração no intervalo de 10^{16}, mesmo sem fazer nenhum cálculo, excede o tempo máximo permitido de 1 segundo. Por isso, é preciso desenvolver uma forma de calcular a quantidade de uns sem iterar no intervalo. É importante notar também que o tipo de dado utilizado deve armazenar números de até 10^{19}, neste caso um inteiro de 8 bytes é suficiente.

Primeiramente, para simplificar o problema, será criada uma função $f(N)$ para calcular o número de uns entre 0 e B. Em seguida, será necessário descontar o número de uns entre 0 e $A - 1$, assim, obtendo o resultado final, como mostrado na Equação 2.1.

$$f(B) - f(A - 1) \tag{2.1}$$

Para entender como implementar $f(N)$, será analisada uma sequência, com o objetivo de encontrar um padrão no número de uns, como a demonstrada na Tabela 2.1.

Binário	Decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Tabela 2.1: Sequência de 0 a 15 em representação binária.

O primeiro caso identificado após visualizar a tabela, é o caso de um número binário que é composto de uns (111...), em outras palavras, uma potência de 2 - 1. Aqui será usado o 15 como exemplo. Para este caso em específico, cada coluna tem 8 uns, que é exatamente metade do número atual somado de 1 ($(15 + 1)/2$). Como há $log_2(N + 1)$ colunas, resta multiplicar o número de colunas pelo número de uns em cada uma, como demonstrado na Equação 2.2.

$$log_2(N + 1) * (N + 1)/2 \tag{2.2}$$

Para identificar se o número segue este padrão, será usada a Equação 2.3. Aplicando ao exemplo, obtém-se $15 \wedge (15 + 1)$, que em representação binária resulta em $1111 \wedge (10000)$, totalizando 0.

$$a \wedge (a + 1) = 0 \tag{2.3}$$

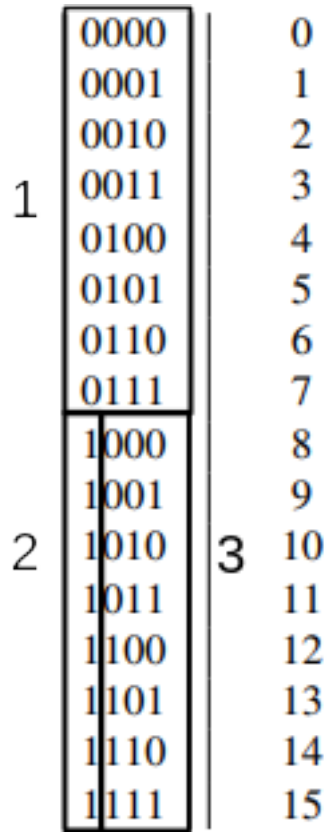


Figura 2.1: Identificando padrões para calcular os outros casos.

Para determinar a fórmula para os casos restantes, a Figura 2.1 separa a tabela em três partes, para facilitar a identificação dos padrões. Como exemplo será usado o número 14 (1110 em representação binária). Primeiramente, como já se sabe calcular um número binário composto de uns, é preciso buscar o número anterior a este que segue este padrão, que no exemplo é o 7 (1110 em representação binária). Pra chegar nele, pode-se fazer um *loop* preenchendo um número com uns, $log2(N)$ vezes, como mostrado no Algoritmo 1. Este número será chamado de b. Assim é possível calcular o número de uns da parte 1, chamando a função f recursivamente para b: $f(b)$. No exemplo, aplicando a Equação 2.2, obtém-se $log_2(7+1) \times (7+1)/2$, que gera o resultado de 12 uns neste intervalo.

Algoritmo 1: Potência de 2 - 1 anterior.

Data: a

$log \leftarrow log2(a)$;
$b \leftarrow 1$;
while $(log \leftarrow log - 1) \neq 0$ **do**
  $b \leftarrow deslocaBitsAEsquerda(b, 1) \vee 1$;
end

A partir do cálculo de b, calcular o número de uns da parte 2 (primeira coluna) é trivial:

$N - b$. Aplicando no exemplo, tem-se $14 - 7 = 7$.

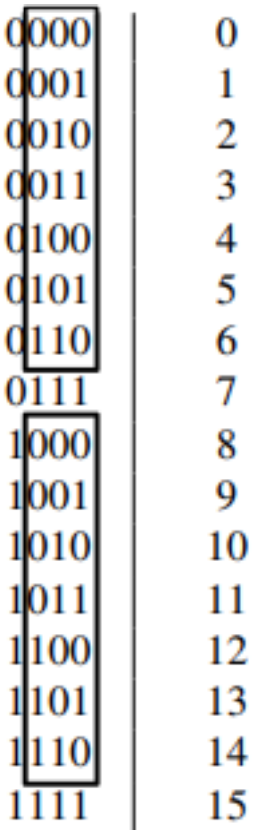


Figura 2.2: Exemplo da parte 3 da Figura 2.1.

Resta calcular a parte 3, para isso será utilizada a recursividade novamente, pois o conteúdo da parte 3 é exatamente igual do correspondente ao número 6, conforme mostrado na Figura 2.2, logo, pode ser usada uma recursão na função f. Para chegar no 6, será usado o resto da divisão com a potência de 2 anterior (que no exemplo é 8, ou $b + 1$): $N \bmod (b + 1)$. Aplicando ao exemplo, $14 \bmod (7 + 1) = 6$, e $f(6)$, por sua vez retornaria 9. Somando o resultado aos outros casos analisados, tem-se $12 + 7 + 9 = 28$. Ficando assim com a Equação 2.4 para os outros casos.

$$f(b) + (N - b) + f(N \bmod (b + 1)) \tag{2.4}$$

Integrando todos os casos, é formado o Algoritmo 2 para a função f. O caso da potência de 2 - 1 está tratado na linha 10 do algoritmo. Nas linhas 11 até 15, é calculado o b, do Algoritmo 1. Enquanto na linha 16 é implementada a Equação 2.4. [2]

Aplicando o exemplo (14) ao Algoritmo 2, primeiramente será chamada a função $f(14)$,

[2] Na linguagem C, as funções *deslocaBitsAEsquerda* e *deslocaBitsADireita*, utilizadas no Algoritmo 2, podem ser substituídas pelos operadores << e >> respectivamente.

onde as condições das linhas 9 e 10 retornará falso, e o algoritmo calculará a variável log como 3, e inicializando b como 1. O *loop* da linha 13 será executado 2 vezes com a variável log indo de 3 até 1, deslocando b dois bits a esquerda, preenchendo com uns à direita, gerando o número 7 (111 em representação binária). O cálculo da linha 16 será resolvido para $14 - 7 + f(7) + f(14 mod (7 + 1))$, ou $7 + f(7) + f(6)$. Chamando a função $f(7)$, ela retornará na linha 10, pois a condição $(n \wedge (n + 1)) = 0$ é verdadeira para 7 ($111 \wedge 1000 = 0$), efetuando o cálculo $log2(n+1) \times ((n+1)/2)$, que resultará em 3×4, retornando 12 para a chamada inicial da função f. Em seguida, será chamada a função $f(6)$, onde as condições das linhas 9 e 10 retornarão falso, o algoritmo calculará a variável log como 2, b será calculado como 3 (11 em representação binária) e o cálculo da linha 16 será resolvido para $6-3+f(3)+f(6 mod (3+1))$, ou $3 + f(3) + f(2)$, que resultará em $3 + 4 + 2$, retornando 9 para a chamada inicial da função. Assim a chamada $f(14)$, resultará em $7 + 12 + 9$, ou 28.

Para calcular complexidade da função f, serão analisadas as 3 partes últimas partes do algoritmo. A parte 1 ($f(b)$) é constante, pois b é um número no padrão potência de 2 - 1, e a Equação 2.2 será usada. É trivial que a parte 2 ($n - b$) também é constante. A parte 3 elimina 1 dígito binário a cada chamada recursiva, pois usa o resto da divisão com b (como no exemplo, 14 (1110), chama a função para 6 (110), que chama para 3 (11), que é uma potência de 2 - 1), com isso haverá, no máximo, $log_2(N)$ chamadas recursivas a função f. Como a complexidade de tempo é calculada em função do tamanho da entrada (T), que neste caso é $log_2(N)$, sabe-se que o algoritmo é linear, pois executará aproximadamente T chamadas recursivas da função f.

Este problema também está disponível para resolução no URI Online Judge (https://www.urionlinejudge.com.br/judge/en/problems/view/1492).

2.2 Problema 2: 2014 - G - Galaxy Collision

Neste problema é descrita a colisão das galáxias Via Láctea e Andrômeda, e dadas as posições das N estrelas resultantes. O objetivo é calcular o número mínimo de estrelas que pode pertencer a uma galáxia, sendo que a distância entre as estrelas de uma mesma galáxia deve ser maior que 5 anos luz. O problema também especifica que sempre existe uma separação possível entre as duas galáxias, atendendo a restrição dos 5 anos luz. A entrada é composta pelo número N ($1 \leq N \leq 5 * 10^4$) de estrelas, e as coordenadas X e Y ($1 \leq X, Y \leq 5 * 10^5$) de cada uma.

Algoritmo 2: Implementação da função f.

```
1  Function log2(n: integer) is
2      log ← 0;
3      while (n ← deslocaBitsADireita(n, 1)) ≠ 0 do
4          log ← log + 1
5      end
6      return log;
7  end
8  Function f(n: integer) is
9      if n = 0 then return 0;
10     if (n ∧ (n + 1)) = 0 then return log2(n + 1) ∗ ((n + 1)/2);
11     log ← log2(n);
12     b ← 1;
13     while (log ← log − 1) ≠ 0 do
14         b ← deslocaBitsAEsquerda(b, 1) ∨ 1;
15     end
16     return n − b + f(b) + f(n mod (b + 1))
17 end
```

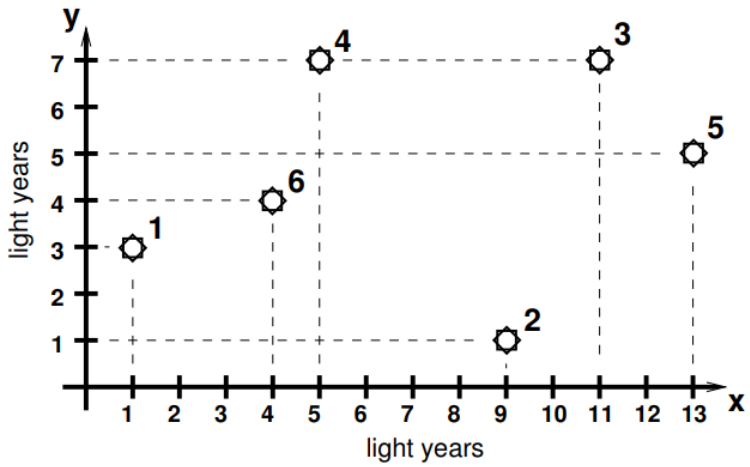


Figura 2.3: Exemplo de entrada do problema Galaxy Collision.

Um exemplo de entrada pode ser visualizado na Figura 2.3, retirada do problema. Neste exemplo, existem 4 formas de separar as estrelas em duas galáxias, mantendo a restrição da distância entre as estrelas ser maior que 5 anos luz: {{1, 2, 4, 5}, {3, 6}}; {{1, 2, 3, 4}, {5, 6}}; {{1, 4, 5}, {2, 3, 6}}; {{1, 3, 4}, {2, 5, 6}}. É possível verificar que, neste caso, o menor número de estrelas que uma galáxia pode ter são 2 estrelas ({3, 6} ou {5, 6}).

Para resolver este problema, será criado um grafo entre as estrelas que tem distância de até 5 anos luz, para depois contar quantas delas podem pertencer a uma das galáxias. Será utilizado o exemplo original do problema, mostrado na Figura 2.3. Para esta entrada, é gerado

o grafo da Figura 2.4. Nele, observa-se que a estrela 1, tem menos de 5 anos luz de distância para a 6, logo, elas terão uma aresta no grafo. A estrela 2 não está próxima a nenhuma outra estrela, logo, ela não terá nenhuma aresta.

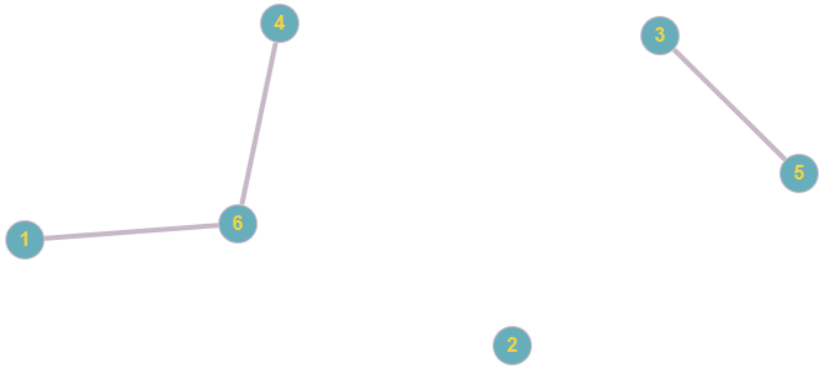


Figura 2.4: Grafo gerado para o problema Galaxy Collision.

Algoritmo 3: Criação trivial do grafo

```
1 for i ← 1 to N do
2 |   for j ← 1 to N do
3 |   |   if i = j then continue ;
4 |   |   if distance(points[i], points[j]) < 5 then
5 |   |   |   graph[i] ← graph[i] ∪ j;
6 |   |   end
7 |   end
8 end
```

Primeiramente, é preciso tratar a criação da estrutura do grafo. O algoritmo trivial para criar este grafo percorre todos as N estrelas para cada estrela, verificando a distância entre elas, como demonstrado no Algoritmo 3. Certamente este código excederá o tempo limite, pois a complexidade é de N^2, sendo $N \leq 5 * 10^4$, dessa forma irá executar até $2,5 * 10^9$ iterações.

Uma solução pra resolver este problema é criar uma matriz para salvar os pontos em suas respectivas coordenadas (Exemplo: uma estrela na posição (2, 2) seria gravada na posição (2, 2) da matriz). Com a matriz criada, para cada ponto, só é preciso percorrer os pontos adjacentes na matriz para encontrar as arestas do grafo. A criação deste mapa está descrita no Algoritmo 4, sendo utilizado no Algoritmo 5 para criação do grafo.

O fato do problema garantir que sempre existe uma solução válida e que não há sobre-posição de estrelas facilita bastante a criação do grafo, pois garante que não há mais que duas

estrelas em uma área de 5 anos luz, isso torna o grafo bastante esparso. Caso contrário, a solução voltaria a ser quadrática, pois poderia acontecer de todos os pontos serem muito próximos, e o algoritmo voltaria a percorrer todas as estrelas para cada estrela. A complexidade da criação do grafo é $O(N \times V(N))$, sendo $V(N)$ a função vizinhança de N. Com o algoritmo trivial, $V(N) = N - 1$, pois todas as outras estrelas serão testadas, tornando o algoritmo quadrático. Com o algoritmo otimizado, $V(N) \leq 100$, porque cada estrela verifica 5 posições para cada lado na matriz, percorrendo um quadrado de 10×10, sendo uma solução linear.

Pelo uso de memória, seria inviável criar uma matriz de (5×10^5) por (5×10^5), então divide-se estes valores por 50, gravando vários pontos em uma mesma posição. Este valor será chamado de MAP_SCALE nos algoritmos seguintes. Este procedimento é demonstrado nas linhas 2 e 3, do Algoritmo 4, e nas linhas 3 e 4 do Algoritmo 5. Nas linhas 6 a 14 do Algoritmo 5, percorre-se os pontos adjacentes da matriz, verificando a distância, e criando a aresta no grafo.

Algoritmo 4: Criação do mapa de posições

```
1  foreach p in points do
2  |  xMap ← p.x/MAP_SCALE;
3  |  yMap ← p.y/MAP_SCALE;
4  |  map[xMap][yMap] ← map[xMap][yMap] ∪ p;
5  end
```

Algoritmo 5: Criação otimizada do grafo

```
1  for i ← 1 to N do
2  |  p ← points[i];
3  |  xMap ← p.x/MAP_SCALE;
4  |  yMap ← p.y/MAP_SCALE;
5  |  for x ← xMap − 1 to xMap + 1 do
6  |  |  for y ← yMap − 1 to yMap + 1 do
7  |  |  |  foreach other in map[x][y] do
8  |  |  |  |  if p = other then continue ;
9  |  |  |  |  if distance(p, other) < 5 then
10 |  |  |  |  |  graph[i] ← graph[i] ∪ other;
11 |  |  |  |  end
12 |  |  |  end
13 |  |  end
14 |  end
15 end
```

A alternativa usada frequentemente em maratonas de programação, quando o problema

envolve comparação de distâncias, é que não é preciso calcular a distância em si, apenas o quadrado dela. No exemplo, ao invés de verificar se $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} <= 5$, verifica-se $(x_1 - x_2)^2 + (y_1 - y_2)^2 <= 25$, que é mais simples, e não envolve cálculos com ponto flutuante.

Com o grafo criado, será usada uma DFS (Depth First Search) para verificar como separar cada componente conexa do grafo em duas galáxias. O algoritmo, apresentado no Algoritmo 6, vai percorrer cada estrela (linha 20), atribuindo uma galáxia a ela (linhas 7 e 10), e marcando a mesma como visitada (linha 3). Após isso, ela percorre as estrelas vizinhas no grafo, executando a DFS recursivamente somando ao resultado (linha 14). Ao final, a DFS vai retornar o número de estrelas das duas galáxias da componente conexa, então é somada a menor galáxia de cada componente conexa para ter o resultado final (linha 22). Como a DFS percorre todas as estrelas apenas uma vez, este algoritmo tem complexidade linear.

Algoritmo 6: Solução do problema G

```
1  Function dfs(i: int, galaxy: int) : {int, int} is
2      if visited[i] then return {0, 0};
3      visited[i] ← true;
4      count ← {0, 0};
5      otherGalaxy ← {0, 0};
6      if galaxy = VIA_LACTEA then
7          count[0] ← 1;
8          otherGalaxy ← ANDROMEDA;
9      else
10         count[1] ← 1;
11         otherGalaxy ← VIA_LACTEA;
12     end
13     foreach j in graph[i] do
14         count = count + dfs(j, otherGalaxy)
15     end
16     return count;
17 end
18 Function f(a: int) : int is
19     minCount ← 0;
20     for i ← 0 to N do
21         count = dfs(i, VIA_LACTEA);
22         minCount = min(count[0], count[1]);
23     end
24     return minCount
25 end
```

No exemplo da Figura 2.4, a componente conexa composta pelas estrelas 1, 6 e 4 vai

retornar 2, 1 na DFS, a das estrelas 3 e 5 retornará 1, 1 e a 2 1, 0. Como o problema pede o número mínimo de estrelas em uma das galáxias, é usado o menor dois dois números retornados pela DFS, e somando os resultados temos $1 + 1 + 0$, totalizando 2 estrelas na menor galáxia.

Este problema também está disponível para resolução no URI Online Judge (https://www.urionlinejudge.com.br/judge/en/problems/view/1749).

3 CONCLUSÃO

Os problemas referenciados neste trabalho ajudam a demonstrar a efetividade do estudo complementar que é feito para as maratonas de programação, aprofundando o conhecimento dos estudantes em grafos, números binários, algoritmos e análise de complexidade de tempo. A preparação para maratona de programação também leva o estudante a praticar mais as habilidades de programação, o que acaba facilita o desenvolvimento das outras atividades acadêmicas.

Também podemos estimar impactos na vida profissional do estudante, pois as grandes empresas frequentemente demandam conhecimentos em otimização de código nas entrevistas de emprego. Algumas até utilizam problemas de maratona como critério de seleção, ou utilizam os eventos da maratona para identificar bons profissionais.

Com estas considerações, podemos concluir que a participação em maratonas de programação tem uma contribuição significativa para a vida acadêmica e profissional dos estudantes.

REFERÊNCIAS

[1] Maratona de Programação IME. https://maratona.ime.usp.br/sobre19.html. Acessado em: 2021-11-17.

[2] Maratona de Programação SBC - 2013. http://maratona.sbc.org.br/hist/2013/index.html. Acessado em: 2021-11-01.

[3] Maratona de Programação SBC - 2014. http://maratona.sbc.org.br/hist/2014/index.html. Acessado em: 2021-11-01.

[4] B. Bloomfield, Aaron; Sotomayor. A programming contest strategy guide. *SIGCSE '16: Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. https://people.cs.uchicago.edu/~borja/pubs/sigcse2016-programming-contests.pdf.

ANEXOS

ANEXO A – 2013 - Problema C - Counting Ones

Problem C

Counting ones

Carl is right now the happiest child in the world: he has just learned this morning what the binary system is. He learned, for instance, that the binary representation of a positive integer k is a string $a_n a_{n-1} \cdots a_1 a_0$ where each a_i is a binary digit 0 or 1, starting with $a_n = 1$, and such that $k = \sum_{i=0}^{n} a_i \times 2^i$. It is really nice to see him turning decimal numbers into binary numbers, and then adding and even multiplying them.

Caesar is Carl's older brother, and he just can't stand to see his little brother so happy. So he has prepared a challenge: "Look Carl, I have an easy question for you: I will give you two integers A and B, and you have to tell me how many 1's there are in the binary representation of all the integers from A to B, inclusive. Get ready". Carl agreed to the challenge. After a few minutes, he came back with a list of the binary representation of all the integers from 1 to 100. "Caesar, I'm ready". Caesar smiled and said: "Well, let me see, I choose $A = 10^{15}$ and $B = 10^{16}$. Your list will not be useful".

Carl hates loosing to his brother so he needs a better solution fast. Can you help him?

Input

A single line that contains two integers A and B ($1 \leq A \leq B \leq 10^{16}$).

Output

Output a line with an integer representing the total number of digits 1 in the binary representation of all the integers from A to B, inclusive.

Sample input 1	**Sample output 1**
`1000000000000000 10000000000000000`	`239502115812196372`

Sample input 2	**Sample output 2**
`2 12`	`21`

Sample input 3	**Sample output 3**
`9007199254740992 9007199254740992`	`1`

ANEXO B – 2013 - Problema C - Counting Ones - Solução em C++

```
1  #define LL unsigned long long
2
3  int log2(LL a) {
4      int log = 0;
5      while(a>>=1) {
6          log++;
7      }
8      return log;
9  }
10
11 LL f(LL a) {
12     if (!a) return 0;
13     if (!(a & (a+1ull))) {
14         return log2(a+1) * ((a+1)/2);
15     }
16     int l = log2(a);
17     LL b = 1;
18     while (--l) {
19         b = (b<<1) | 1;
20     }
21
22     return a-b +f(b) + f(a%(b+1));
23 }
24
25 int main() {
26     LL a, b;
27     while (scanf("%llu %llu", &a, &b) != EOF) {
28         printf("%llu\n", f(b)-f(a-1));
29     }
30
31     return 0;
32 }
```

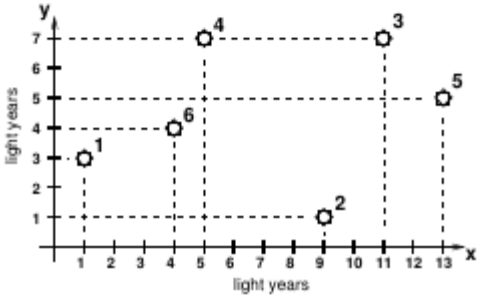
ANEXO C – 2014 - Problema G - Galaxy Collision

Problem G – Galaxy collision

Author: Guilherme Albuquerque Pinto, Universidade Federal de Juiz de Fora

The Andromeda galaxy is expected to collide with our Milky Way in about 3.8 billion years. The collision will probably be a merging of the two galaxies, with no two stars actually colliding. That is because the distance between stars in both galaxies is so huge. Professor Andrew is building a computational model to predict the possible outcomes of the collision and needs your help! A set of points in the two dimensional plane is given, representing stars in a certain region of the already merged galaxies. He does not know which stars came originally from which galaxy; but he knows that, for this region, if two stars came from the same galaxy, then the distance between them is greater than 5 light years. Since every star in this region comes either from Andromeda or from the Milky Way, the professor also knows that the given set of points can be separated into two disjoint subsets, one comprising stars from Andromeda and the other one stars from the Milky Way, both subsets with the property that the minimum distance between two points in the subset is greater than 5 light years. He calls this a *good* separation, but the bad news is that there may be many different good separations. However, among all possible good separations there is a minimum number of stars a subset must contain, and this is the number your program has to compute.

For example, the picture illustrates a given set of six points. Professor Andrew cannot tell which stars came from Andromeda, but note that there are four possible good separations: $\{\{1,2,4,5\},\{3,6\}\}$; $\{\{1,2,3,4\},\{5,6\}\}$; $\{\{1,4,5\},\{2,3,6\}\}$; $\{\{1,3,4\},\{2,5,6\}\}$. Therefore, at least two stars must have come from Andromeda, since this is the minimum number of points a subset may have in a good separation.



Input

The first line contains an integer N ($1 \leq N \leq 5 \times 10^4$) representing the number of points in the set. Each of the next N lines describes a different point with two integers X and Y ($1 \leq X, Y \leq 5 \times 10^5$), indicating its coordinates, in light years. There are no coincident points, and the set admits at least one good separation.

Output

Output a line with an integer representing the minimum number of points a subset may have in a good separation.

Sample input 1	**Sample output 1**
6 1 3 9 1 11 7 5 7 13 5 4 4	2

Sample input 2	**Sample output 2**
2 10 10 50 30	0

ANEXO D – 2014 - Problema G - Galaxy Collision - Solução em C++

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <math.h>
4  #include <vector>
5  using namespace std;
6
7  #define MAX_DISTANCE_SQUARED 25
8  #define MAX_POS 500000
9  #define MAP_SCALE 50
10 #define MAX_MAP_POS MAX_POS/MAP_SCALE
11 #define MAX 50000
12 #define VIA_LACTEA 1
13 #define ANDROMEDA 2
14
15 typedef struct point {
16     int x;
17     int y;
18 } Point;
19
20 Point points[MAX];
21 vector<int>* map[MAX_MAP_POS][MAX_MAP_POS];
22 vector<int> graph[MAX];
23 bool visited[MAX];
24 int N;
25
26
27 long long distanceSquared(Point a, Point b) {
28     long long x = b.x - a.x;
29     long long y = b.y - a.y;
30     return x*x + y*y;
31 }
32
33 pair<int, int> addPair(pair<int, int> a, pair<int, int> b) {
34     return make_pair(a.first + b.first, a.second + b.second);
35 }
36
37 pair<int, int> dfs(int i, int galaxy) {
38     if (visited[i]) {
39         return {0, 0};
40     }
41     visited[i] = true;
42     pair<int, int> count = {0, 0};
43
44     switch (galaxy) {
45         case VIA_LACTEA: count.first++; break;
46         case ANDROMEDA: count.second++; break;
47     }
48
49     for (int j = graph[i].size(); j--; ) {
50         auto other = dfs(
51             graph[i][j],
52             galaxy == VIA_LACTEA ? ANDROMEDA : VIA_LACTEA);
53         count = addPair(count, other);
```

```
54      }
55
56      return count;
57  }
58
59
60  int main() {
61      scanf("%d", &N);
62
63      // ler entrada
64      for (int i = 0; i != N; i++) {
65          int x, y;
66          scanf("%d %d", &x, &y);
67          points[i].x = x;
68          points[i].y = y;
69      }
70
71      // preenche mapa
72      for (int i = 0; i != N; i++) {
73          Point p = points[i];
74          int xMap = p.x / MAP_SCALE;
75          int yMap = p.y / MAP_SCALE;
76          vector<int>* mapxy = map[xMap][yMap];
77          if (!mapxy) {
78              mapxy = map[xMap][yMap] = new vector<int>();
79          }
80          mapxy->push_back(i);
81      }
82
83      // cria grafo
84      for (int i = 0; i != N; i++) {
85          Point p = points[i];
86          int xMap = p.x / MAP_SCALE;
87          int yMap = p.y / MAP_SCALE;
88          for (int x = xMap-1; x <= xMap+1; x++) {
89              if (x < 0 || x > MAX_MAP_POS) continue;
90              for (int y = yMap-1; y <= yMap+1; y++) {
91                  if (y < 0 || y > MAX_MAP_POS) continue;
92                  if (!(map[x][y])) continue;
93
94                  vector<int> mapxy = *map[x][y];
95                  for (int xy = mapxy.size(); xy--; ) {
96                      int otherIdx = mapxy[xy];
97                      if (otherIdx == i) continue;
98                      Point other = points[otherIdx];
99                      if (distanceSquared(p, other)
100                             <= MAX_DISTANCE_SQUARED) {
101                         graph[i].push_back(otherIdx);
102                     }
103                 }
104             }
105         }
106     }
107
108     // busca menor valor
109     int minCount = 0;
110     for (int i = N; i --; ) {
111         auto count = dfs(i, VIA_LACTEA);
```

```
112             minCount += min(count.first, count.second);
113         }
114
115         printf("%d\n", minCount);
116
117         return 0;
118     }
```

ANEXO E – 2013 - Certificado de participação

Certificate of Achievement

South America Brazil Region - Brazilian Finals
Uberlândia, November 10th 2013

Honorable Mention

acm International Collegiate Programming Contest

IBM event sponsor

Universidade Federal da Fronteira Sul

Iago Ulian Berndt
Emerson Dallagnol
Geomar André Schreiner
Leandro Zatesko, Coach
Mateus Trebien

William B. Poucher, Ph. D.
ICPC Executive Director

ANEXO F – 2014 - Certificado de participação

Certificate of Achievement

The 2014 ACM-ICPC South America/Brazil Finals
07 - 08 November 2014

Universidade Federal da Fronteira Sul

Iago Ulian Berndt
Matheus DallRosa
Emerson Dallagnol
Leandro Zatesko, Coach

William B. Poucher, Ph. D.
ICPC Executive Director