Sessão de Discussão dos Problemas

Pedro Ribeiro

DCC/FCUP

MIUP’2016

MIUP'16
MARATONA
INTER-UNIVERSITÁRIA
DE PROGRAMAÇÃO

Visão Geral do Conjunto de Problemas

#	Problema	Tipo	Dificuldade		Ling.	#Linhas
			Alg	Imp		
A	Greedy Trading	Subsequências, Ordenação	4	5	Java	37
B	The Ant and the Grasshopper	Matemática, Prog. Geométricas	5	2	C	23
C	Election of Representatives	Simulação	3	3	C	34
D	The Modern Feud...	Geometria, Ordenação/ED Espacial	5	4	C	42
E	Administrative Reform	Grafos, Dijkstra	5	5	C++	58
F	Rock-Me-Not	Prog. Dinâmica, Bitmasks	7	6	C	68
G	Eccentrics	Simulação, 3D	4	4	Java	30
H	Problem Setters	Greedy, Swee Linep / ED Intervalos	6	6	C	72
I	Surveillance	Geometria, Visibilidade	8	8	C	167

Esta pequena discussão dos problemas constitui apenas uma visão pessoal minha

A - Greedy Trading

Problema

- **Input:** Sequência de N comandos descrevendo *updates* (adicionar um número à sequência actual) e *queries*
- **Output:** Para cada *query* indicar qual as posições de início e fim da **subsequência de soma máxima** nos números já adicionados, e qual a **mediana** dessa subsequência.

Limites

- $N \leq 1000$

Classificação

- **Categorias:** Subsequência, Ordenação
- **Dificuldade:** Fácil+
- **Autor:** Alexandre Francisco (IST)

A - Greedy Trading

- O problema tem duas partes:
  - (1) Descobrir a sequência de soma máxima até ao momento
  - (2) Descobrir a sua mediana
- Como $N \leq 1000$, precisamos de complexidade mais baixa que $\mathcal{O}(N^3)$
- (1) pode ser feita iterativamente mudando em cada *update*:
  - Com cada *update* apenas muda um número
  - Ter em conta apenas as subequências que terminam nesse novo nº
    - Em $\mathcal{O}(1)$ usando o **algoritmo de Kadane**
    - Em $\mathcal{O}(N)$ fazendo um ciclo desde o novo número
- (2) pode ser feita apenas em cada *query*:
  - Basta ordenar usando por exemplo a biblioteca da vossa linguagem **qsort** (C), **sort** (C++), **Arrays.sort** (Java)
    Todas estas ordenação são linearítmicas $\mathcal{O}(N \log N)$
- O tempo total é dominado pela ordenação (podemos ter tantas *queries* como *updates*): $\mathcal{O}(N^2 \log N)$

B - The Ant and the Grasshopper

Problema

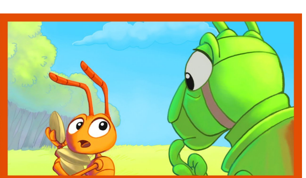
- **Input:** n cenários, cada um indicando um empréstimo com y anos, quantidade de empréstimo B e taxa de juro r.
- **Output:** Para cada cenário, supondo que o montante a pagar anualmente é fixo, o total que a cigarra paga (amortizações + juros).

Limites

- $n \leq 1000$, $y < 100$, $B < 1000$

Classificação

- **Categorias:** Matemática, Progressões Geométricas
- **Dificuldade:** Médio-
- **Autor:** Filipe Araújo (U. Coimbra)



B - The Ant and the Grasshopper

- Podemos visualizar o empréstimo como uma recorrência:
  - B: quantidade emprestada, r: taxa de juro
  - $T(i)$: dívida total no ano i
  - Q_{fixa}: quantia fixa a pagar cada ano

$$T(0) = B$$
$$T(i) = T(i-1) \times (1+r) - Q_{fixa}$$

$$\begin{aligned}
T(3) &= T(2) \times (1+r) - Q_{fixa} = \\
&= [T(1) \times (1+r) - Q_{fixa}] \times (1+r) - Q_{fixa} \\
&= T(1) \times (1+r)^2 - Q_{fixa} \times [(1+R)+1] \\
&= [T(0) \times (1+r) - Q_{fixa}] \times (1+r)^2) - Q_{fixa} \times [(1+R)+1] \\
&= T(0) \times (1+r)^3 - Q_{fixa} \times [(1+r)^2 + (1+r) + 1] \\
&= B \times (1+r)^3 - Q_{fixa} \times [(1+r)^2 + (1+r) + 1] \\
T(y) &= B \times (1+r)^y - Q_{fixa} \times [(1+r)^{y-1} + (1+r)^{y-2} + \ldots + 1]
\end{aligned}$$

B - The Ant and the Grasshopper

Seja y o total de anos. Então

$T(y) = B \times (1+r)^y - Q_{fixa} \times [(1+r)^{y-1} + (1+r)^{y-2} + \ldots + 1]$

Soma de uma progressão geométrica

$$\sum_{k=0}^{n-1}(ac^k) = a(\frac{1-c^n}{1-c})$$

Seja $R = 1 + r$. Sabemos que no final vamos ter de ficar a devolver zero. Então:

$$0 = B \times R^y - Q_{fixa} \times \frac{1-R^y}{1-R}$$
$$Q_{fixa} = \frac{B \times R^y \times (1-R)}{1-R^y}$$

Finalmente, o total a pagar é Q_{fixa} vezes o número de anos y. Então

Total a Pagar $= \frac{y \times B \times R^y \times (1-R)}{1-R^y}$

A resposta pode ser obtida então em $\mathcal{O}(1)$ para cada cenário :)

C - Election of Representatives

Problema

- **Input:** número de lugares S, número de partidos P e o número de votos v_i em cada partido
- **Output:** Aplicar o método de Hondt e determinar a alocaçã de lugares a cada partido

Limites

- $S \leq 250$, $P < 20$

Classificação

- **Categorias:** Simulação, Método de Hondt
- **Dificuldade:** Muito Fácil
- **Autor:** José Saias (U. Évora)

C - Election of Representatives

- O algoritmo para resolver o problema era dado no enunciado
  Basta apenas fazer a simulação pedida!
- Para cada S lugares, descobrir o partido i que maximiza $q_i = \frac{v_i}{s_i+1}$
- Como só temos 250 lugares e 20 partidos podemos até fazer um simples ciclo para descobrir o máximo em cada iteração de um lugar
- Ficamos por isso com uma solução $\mathcal{O}(S \times P)$ muito fácil de implementar

- Se o número de partidos fosse maior poderíamos usar uma estruturas de dados especializada para descobrir o máximo mais rapidamente:
  Exemplo: uma *priority_queue* actualiza e retorna máximo em $\mathcal{O}(n \log n)$

D - The Modern Feud of the Capulets and Montagues

Problema

- **Input:** N pontos de duas "cores"
- **Output:** o par de pontos de cores diferentes mais próximo (distância de manhattan)

Limites

- $N \leq 100\,000$
- Pontos bem "espalhados"

Classificação

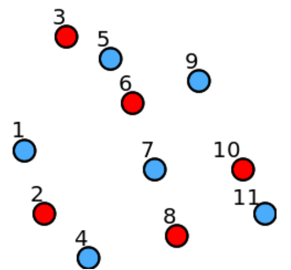
- **Categorias:** Geometria, Ordenação
- **Dificuldade:** Fácil
- **Autor:** Rui Mendes (U. Minho)

D - The Modern Feud of the Capulets and Montagues

- Com 100 000 pontos, um algoritmo quadrático ($\mathcal{O}(N^2)$) que teste todos os pares de pontos possíveis não passa no tempo.
- Saber que os pontos estavam **"spread out evenly over the space"** muda completamente o problema
- Só precisamos de ter bom comportamento num caso com distribuição uniforme dos pontos no plano
- Temos de aproveitar de algum modo a **organização espacial** dos pontos.
- Existiam muitas soluções possíveis:
  - Usar uma estrutura de dados espacial (ex: *quadtree*, *kd-tree*)
  - Pesquisar e cortar quando temos certeza que não melhora solução (*Branch and Bound*)
  - Adaptar algoritmo clássico de par de pontos mais próximo de uma só cor (*Divide and Conquer*)
  - ...

D - The Modern Feud of the Capulets and Montagues

- Uma solução simples passava por ordenar os pontos por um eixo (ex: ordenar pelo x ou pelo y)



- Consideremos que ordenamos pelo eixo dos x
- Comparar cada ponto p_i com os p_j com $j > i$
- Parar quado $p_j.x - p_i.x > menor_distancia$
- Se os pontos estão espalhados esta solução é muito rápida!

- Num caso real ordenar por um eixo de orientação aleatória tornava um worst case muito improvável
- Existe outro tipo de solução (sem D&C) que é sempre $\mathcal{O}(N \log N)$) qualquer que seja a distribuição dos pontos.
  Fica para pensarem em casa :)

E - Administrative Reform

Problema

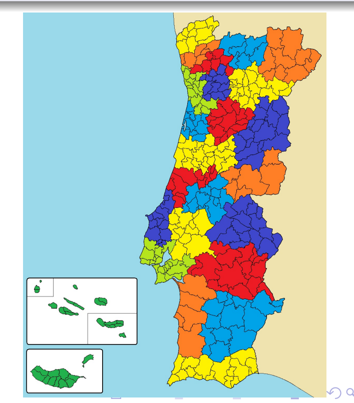
- **Input:** Dado um grafo não dirigido pesado com V nós e E arestas e dois nós c_1 e c_2
- **Output:** contar o número de pontos mais próximos de c_1, mais próximos de c_2 ou à mesma distância de ambos

Limites

- $V \leq 20\,000$
- $E \leq 150\,000$

Classificação

- **Categorias:** Grafos, Caminhos Mínimos, Algoritmo de Dijkstra
- **Dificuldade:** Médio-
- **Autor:** Margarida Mamede (UNL)



E - Administrative Reform

- Para calcular caminhos mínimos em grafos pesados não existe nada mais clássico e ensinado do que o **Algoritmo de Dijkstra**.
- O Dijkstra descobre um caminho mínimo de um nó para todos os outros
- Basta executar duas vezes o Dijkstra, uma a partir de c_1 e outra de c_2 para obter todas as distância necessárias
- Cuidados:
  - $20\,000^2$ não cabe em memória pelo que não podiamos usar matriz de adjacências (usar lista ou outra estrutura $\mathcal{O}(E)$ em espaço)
  - $20\,000^2$ sem matriz é demasiado para ter um Dijkstra quadrático no tempo
  - Uma implementação com $\mathcal{O}(E \log V)$ em termos temporais já passaria. Para isso precisamos de descobrir próximo nó em tempo logarítmico (ex: usar *priority_queue* ou *set*)

F - Rock-Me-Not

Problema

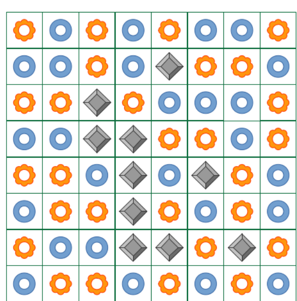
- **Input:** Uma matrix $L \times C$ com "rochas" nalgumas células
- **Output:** O maior número de flores que se pode colocar na matriz sabendo que cada flor precisa de dois aspersores em células adjacentes

Limites

- $L \leq 9$
- $C \leq 9$

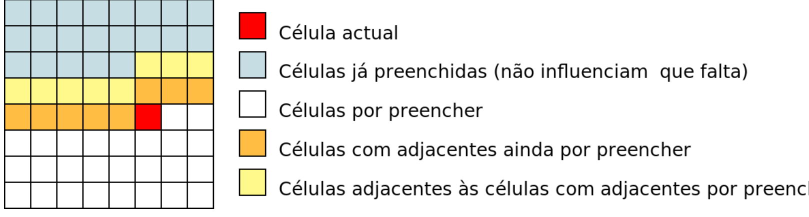
Classificação

- **Categorias:** Programação Dinâmica, Bitmasks
- **Dificuldade:** Médio+
- **Autor:** André Restivo (FEUP)



F - Rock-Me-Not

- Testar todas as configurações possíveis não passa no tempo ($2^{81} > 10^{24}$) (e não chegaria evitar ir por configurações "inválidas")
- Suponha que está a preencher de cima para baixo e da esquerda para a direita. Quando estamos a preencher uma posição, que posições importam para o que falta?



- Apenas as células amarelas e laranjas "influenciam" o que falta preencher.
- Vamos codificar as $2 \times C$ células laranjas e amarelas. Precisamos de um bit para cada (flor ou aspersor) e por isso podemos codificar essas células por um número entre 0 e $2^{2C} - 1$ (uma **bitmask**).

F - Rock-Me-Not

- Seja N um número que descreve o estado das células $2 \times C$ células anteriores à posição actual (x, y).
- O estado (x, y, N) descreve completamente a nossa situação no presente
- Existem múltiplas configurações que vão dar a um mesmo estado (x, y, N) (qualquer configuração nas células cinzentas anteriores)
- Se guardarmos o melhor possível a partir desse estado (x, y, N), não precisamos de recalcular! (**programação dinâmica com memoization**)
- Resolver o problema passa a ser calcular o melhor do estado $(0, 0, 0)$ (qual o melhor a partir da posição $(0, 0)$ quando nas células anteriores não existe nada)
- No máximo existem então $9 \times 9 \times 2^{18}$ estados, que é um número inferior a 10^8 e por isso perfeitamente calculável
- A tabela para guardar o melhor a partir de cada estado só precisa de um byte por cada posição e por isso ocupa cerca de 20MB, cabendo perfeitamente em memória.

G - Eccentrics

Problema

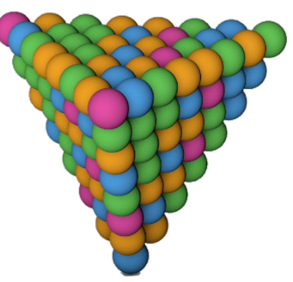
- **Input:** Um tetraedro de N camadas e o peso de cada uma das bolas dentro dele
- **Output:** A ordem em que as bolas caem do tetraedro, supondo que as bolas mais pesadas ocupam as posições deixadas livres nas camadas inferiores.

Limites

- $N \leq 16$

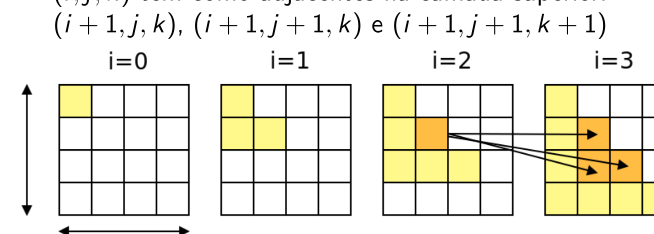
Classificação

- **Categorias:** Simulação, 3D
- **Dificuldade:** Fácil+
- **Autor:** Pedro Guerreiro (U. Algarve)



G - Eccentrics

- O algoritmo para resolver o problema era dado no enunciado. Basta apenas fazer a simulação pedida!
- Repetir N vezes o processo de deixar cair uma bola
- De cada vez começar na camada inferior e ir subindo escolhendo sempre a bola "adjacente" mais pesada da camada superior
- A única dificuldade é como organizar e armazenar o tetraedo.
  - Uma possibilidade é um array tridimensional
  - A posição inferior fica em $(0, 0, 0)$
  - (i, j, k) tem como adjacentes na camada superior: $(i+1, j, k)$, $(i+1, j+1, k)$ e $(i+1, j+1, k+1)$



H - Problem Setters

Problema

- **Input:** Um conjunto de N intervalos $[a_i, b_i]$, cada um com um número de emails k_i associado
- **Output:** O menor número de emails a enviar para que cada intervalo i receba pelo menos k_i emails

Limites

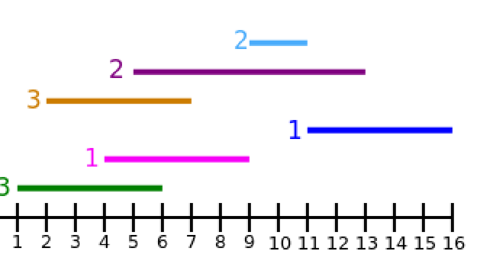
- $N \leq 35\ 000$
- $k_i \leq 5$

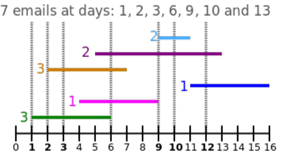
Classificação

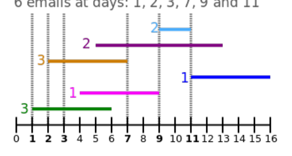
- **Categorias:** Greedy, Ordenação Sweep Line, Set
- **Dificuldade:** Médio+
- **Autor:** Pedro Ribeiro (FCUP)

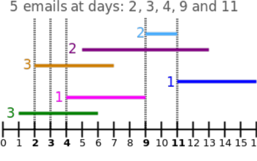
H - Problem Setters

Name	Min Emails	Interval
Steve	(2)	[9, 11]
Edna	(2)	[5, 13]
Mary	(3)	[2, 7]
James	(1)	[11,16]
Barbara	(1)	[4, 9]
David	(3)	[1 ,6]





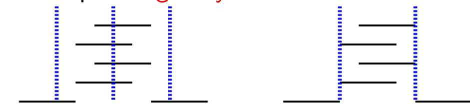




- Testar todas as hipóteses possíveis não é exequível...

H - Problem Setters

- Será que alguma **estratégia greedy** funciona? Qual
- Exemplo de greedy errado: escolher dia com mais pessoas:



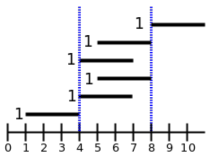
- **Intuição:** quanto mais tarde enviar um email, mais intervalos "apanho"
- Então devemos enviar apenas quando for... inevitável! Se faltam x dias para terminar intervalo i e ainda necessita de x emails, é preciso enviar!

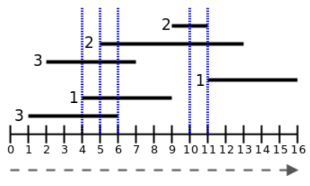
Estratégia greedy correcta (adiar ao máximo envio de emails)

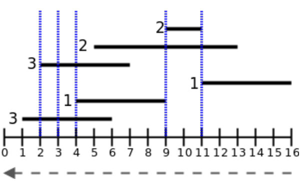
Próximo email a enviar é o mais tarde possível que seja "inevitável"

H - Problem Setters

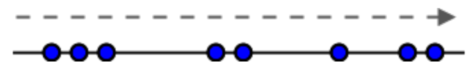
- Vamos ver no exemplo anterior e no exemplo do enunciado







- Como implementar com eficiência sufucente?
- Ter em conta apenas os dias "interessantes" onde podem existir emails são os últimos k_i de cada intervalo i (no total $N \times max(k_i)$ pontos)
- **Ideia:** Ordenar e varrer esses pontos relevantes numa direção (esq $\rightarrow$ dir) **Sweep Line**



H - Problem Setters

- Quando um dia for relevante, actualizar número de emails necessários de cada intervalo "aberto" (iniciado mas não terminado)
- Como saber intervalos que estão "abertos? Podemos acrescentar dois pontos ao nosso sweep: a_i (intervalo i começa) e $b_i + 1$ (intervalo termina)

- Podemos manter intervalos abertos num conjunto e ir adicionando e removendo quando aparecerem os pontos correspondentes
- Quando um ponto relevante estiver a k_i do final intervalo i, enviamos um email e decrementamos o k_i de todos os intervalos abertos
- Ordenação em $\mathcal{O}(NK_i \log NK_i)$; depois fazer sweep a $\mathcal{O}(NK_i)$ pontos.
- Cada intervalo vai ser adicionado uma vez, decrementando k_i, e removido uma vez (podemos usar *sets* para inserir/remover em $\mathcal{O}(\log N)$.
- No final a complexidade global fica $\mathcal{O}(NK_i \log NK_i)$

I - Surveillance

Problema

- **Input:** Um polígono simples com N vértices
- **Output:** O vértice com a maior área de visibilidade

Limites

- $N \leq 50$

Classificação

- **Categorias:** Geometria, Visibilidade, Polígonos, Interseções
- **Dificuldade:** Difícil+
- **Autor:** Fábio Marques (ESTGA)

I - Surveillance

- Baseado na ideia do **polígono de visibilidade**, um problema de **geometria computacional**, para o qual existem vários papers publicados (alguns até com algoritmos... errados)!

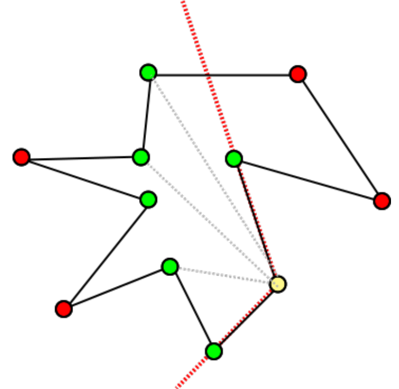
  Joe, Barry, and Richard B. Simpson. *"Corrections to Lee's visibility polygon algorithm."* BIT Numerical Mathematics 27.4 (1987): 458-473.

- Vou descrever uma ideia pensada "como num concurso", sem acesso a trabalho já existente.

- Uma visão geral do algoritmo:
  - Percorrer todos os vértices do polígono
  - Para cada vértice:
    - Descobrir que outros vértices do polígono são visíveis
    - Percorrer a borda do polígono (*ccw*)
    - Ao percorrer adicionar os vértices visíveis ao "polígono de visibilidade"
    - Quando um vértice não é visível atirar um "raio" para descobrir em que pontos as linhas passam a estar visíveis
      (ray tracing)
    - Calcular área do polígono

I - Surveillance

- Descobrir que outros vértices do polígono são visíveis
  - Considerar apenas pontos que estão no "ângulo de visibilidade" (dois vértices adjacentes do ponto)
  - Verificar se nenhuma aresta é "cruzada" ao ir de um vértice ao outro



I - Surveillance

- Percorrer a borda do polígono e construir polígono de visibilidade
  - Vértices visíveis percentem ao polígono de visibilidade
  - Nas zonas com vértices não visíveis usar *ray tracing* para descobrir onde a aresta fica novamente visível
  - Calcular área do polígono ("triangulação" - algoritmo clássico)