Aula 4 · Provando Soluções e Algoritmos Gananciosos

Desafios de Programação

Fernando Kiotheka

UFPR

27/03/2024

Princípios básicos de provas

Invariantes e monovariantes

- Talvez os conceitos mais importantes de todos!
- Uma invariante é uma quantidade que **não muda** no tempo.
- Exemplo: a soma de todas as probabilidades é 1.
- Uma monovariante varia *monotonicamente* no tempo.
- Uma quantidade é monotonicamente não-decrescente ou fracamente crescente se na função do tempo:

$$s \leq t \implies f(s) \leq f(t)$$

- Com $<$ temos uma quantidade monotonicamente crescente ou estritamente crescente.
- Com $\geq$ e $>$, temos o inverso: não-crescente ou fracamente decrescente e decrescente ou estritamente decrescente.

Problema clássico: Load Balancing[3] (B)

- Problema "idêntico": The Trip[1], A Viagem[2].
- Processamento de tarefas em n servidores.
- Cada servidor i tem m_i tarefas agendadas.
- Queremos minimizar $m_a - m_b$ onde a é o servidor mais carregado e b o menos carregado.
- Você pode transferir uma tarefa entre dois servidores.
- Qual o mínimo de transferências necessárias?

[1] https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=16&page=show_problem&problem=1078

[2] https://www.beecrowd.com.br/judge/pt/problems/view/1220

[3] https://codeforces.com/contest/609/problem/C

Identificar a invariante

- O total de tarefas agendadas **nunca muda**.
- Caso simples: o total de tarefas t é múltiplo de n.
  - Nesse caso $m_a - m_b$ pode ser 0 com todos os servidores com exatamente a mesma quantidade de tarefas: t/n.
- O restante: o total de tarefas t não é múltiplo de n.
  - Sabemos que $m_a - m_b$ pode ser no mínimo 1 e há duas quantidades de tarefas diferentes distribuídas entre os servidores: m_a e m_b, qual o valor de m_b?
  - Podemos resolver analiticamente: sabemos que o total de tarefas t deve ser igual a soma de p servidores com $m_a = m_b + 1$ tarefas e $n - p$ servidores com m_b tarefas:

$$\begin{aligned} t &= p(m_b + 1) + (n - p)m_b \\ &= pm_b + p + nm_b - pm_b \\ &= nm_b + p \\ nm_b &= t - p \\ m_b &= (t - p)/n \end{aligned}$$

Uma definição de módulo

Os inteiros quociente q e resto r de a dividido por n satisfazem:

$$\begin{aligned} a &= nq + r \\ nq &= a - r \\ n &= (a - r)/q \\ q &= \lfloor a/n \rfloor \\ r &= a \pmod{q} \end{aligned}$$

Onde $0 \leq r < n$, se n e a são positivos.

$\rightarrow$ Caso n ou a sejam negativos, use outra definição.

Resolvendo o problema

No nosso caso,

$$m_b = (t - p)/n$$

Como m_b é inteiro e $0 \leq p < n$, então $p = (t \bmod n)$.

- Então no caso onde o total de tarefas t não é múltiplo de n:
  - Temos $p = (t \bmod n)$ servidores com $m_a = m_b + 1$ tarefas sendo $m_b = \lfloor t/n \rfloor$.
- Agora sabemos como os servidores devem ficar!
- Como fazer as transferências da forma mais eficiente possível?
  - Essa pergunta se resume a uma pergunta recorrente:
  - Quantas operações de incremento ou decremento para transformar um vetor em outro, desprezando ordem?
  - Intuição: ordenar origem e destino.
  - A resposta seria a soma das diferenças de cada elemento.
  - No caso das transferências, a resposta seria divisível por dois, e a quantidade de transferências seria o quociente dessa divisão.
  - Isso funciona?

Provando que ordenar funciona

- Temos que provar que se A e B são ordenados de forma não-decrescente, $\sum_i |a_i - b_i|$ é minímo, isto é, não existe outra ordenação que obtenha um valor menor.
- O valor absoluto é muito chato de lidar, geralmente você precisa considerar caso a caso[4], ou empregar uns truques que veremos mais pra frente.
- Vamos provar algo mais fácil com as diferenças ao quadrado[5], isto é, $\sum_i |a_i - b_i|^2 = \sum_i (a_i - b_i)^2$ tem que ser mínimo.
- É válido porque $|x|$ e x^2 são funções monotônicas.

[4] `https://stackoverflow.com/a/65920964`
[5] `https://stackoverflow.com/a/65920010`

Provando que ordenar funciona (continuado)

$$\sum_i (a_i - b_i)^2 = \sum_i a_i^2 + b_i^2 - 2a_i b_i$$
$$= \sum_i a_i^2 + \sum_i b_i^2 - 2\sum_i a_i b_i$$

Como em qualquer ordenação o valor de $\sum_i a_i^2$ e $\sum_i b_i^2$ não se altera, para minimizar o custo total, precisamos maximizar $\sum_i a_i b_i$. Para maximizar $\sum_i a_i b_i$, A e B devem estar ordenados conforme a **desigualdade do rearranjo**.

Desigualdade do rearranjo

A desigualdade do rearranjo[6] afirma que

$$\begin{aligned}
&\sum_{i=1}^{n} x_{n-i+1} y_i = x_n y_1 + \ldots + x_1 y_n \\
&\leq x_{\pi_1} y_1 + \ldots + x_{\pi_n} y_n \\
&\leq \sum_{i=1}^{n} x_i y_i = x_1 y_1 + \ldots + x_n y_n
\end{aligned}$$

Para qualquer escolha de números reais $x_1 \leq \ldots \leq x_n$ e $y_1 \leq \ldots \leq y_n$ e toda permutação π. Informalmente:

- Soma máxima com os maiores pareados com os maiores.
- Soma mínima com os menores pareados com os maiores.

[6] `https://en.wikipedia.org/wiki/Rearrangement_inequality`

As transferências sempre existem?

- Sabendo que a soma dos vetores é a mesma, isso é suficiente pra dizer que as transferências existem?
- Em outras palavras: Isso é suficiente pra dizer que a soma das diferenças positivas é igual a soma das diferenças negativas?
- Sim. Vamos provar por contradição:

Assuma sem perda de generalidade (troque A por B se necessário) que a soma das diferenças positivas d_+ é maior que a soma das diferenças negativas d_- entre os vetores A e B $(a_i - b_i)$.
O vetor A então tem soma $d_+ - d_- > 0$ maior do que o vetor B.
Porém, a soma de A é igual a soma de B, contradição $\lightning$.

- Como a soma das diferenças é a soma de d_+ e d_- que são iguais, outro resultado é que a soma das diferenças é par.

Provas não-construtivas

- A prova anterior especificamente é *irritante*.
- Por quê? Porque é uma prova não-construtiva.
- Provas não-construtivas não te dão algoritmos.
- Provas não-construtivas muitas vezes não ajudam em nada.
- Provas não-construtivas não são satisfatórias.
- Nesse caso, não sabemos como realizar as transferências.
- Para o nosso problema, isso é suficiente: queremos apenas o número de operações, e não como elas devem ser realizadas.
- Provas que nos dão algoritmos são as provas construtivas.
- Provas construtivas são desejáveis, mas muitas vezes é impossível provar que a resposta sempre existe.
- Então provas não-construtivas são um mal necessário.

Algoritmos gananciosos

Algoritmos gananciosas/gulosas

Um algoritmo ganancioso/guloso constrói a solução de um problema escolhendo sempre o caminho que pareça ser o melhor, isto é, através de ótimos locais.

- Geralmente rápidos, geralmente $\mathcal{O}(n \lg n)$ ou $\mathcal{O}(n)$.
- Pode ser difícil determinar a estratégia gananciosa.
- Os ótimos locais realmente levam ao ótimo global?

Aplicações

- Os algoritmos "rápidos" que conhecemos geralmente são algoritmos gananciosos: árvore mínima geradora é um exemplo.
- Na maratona, geralmente os problemas possuem propriedades que permitem a aplicação de uma abordagem gananciosa. No mundo real isso também não é raro.
- E quando temos problemas difíceis, algoritmos com boas heurísticas podem ser usados para aproximar soluções.
- Podemos criar uma solução lenta, de força bruta e comparar com a nossa solução gananciosa pra ver se ela está correta.

Problema do Troco (conjunto fixo)

Qual o menor número de moedas do conjunto S necessárias para obter o valor V?

$$S = \{1, 2, 5, 10\}$$

Para $V = 25$, por exemplo, podemos usar 3 moedas: $\{10, 10, 5\}$.

Solução gananciosa do Problema do Troco (conjunto fixo)

Acontece que existe uma solução gananciosa para este problema, e é de escolher sempre a maior moeda que não passa do valor desejado (e decrementá-lo com o valor da moeda).
Intuição: estamos tirando o máximo possível do valor atual.

Solução gananciosa do Problema do Troco (conjunto fixo)

Isso não funciona sempre. Tomemos por exemplo $S = \{1, 3, 4\}$ e $V = 6$:

1. Para $V = 6$, escolhemos a moeda 4 (`ans++`)
2. Para $V = 2$, escolhemos a moeda 1 (`ans++`)
3. Para $V = 1$, escolhemos a moeda 1 (`ans++`)

Poderíamos, no entanto, escolher a moeda 3 duas vezes!

Solução gananciosa do Problema do Troco (conjunto fixo)

Isso não funciona nem se a soma dos valores menores que um determinado valor for menor. Tomemos por exemplo $S = \{1, 7, 10\}$ e $V = 14$:

1. Para $V = 14$, escolhemos a moeda 10 (`ans++`)
2. Para $V = 4$, escolhemos a moeda 1 (`ans++`)
3. Para $V = 3$, escolhemos a moeda 1 (`ans++`)
4. Para $V = 2$, escolhemos a moeda 1 (`ans++`)
5. Para $V = 1$, escolhemos a moeda 1 (`ans++`)

Poderíamos, no entanto, escolher a moeda 7 duas vezes!

Assim, solução gananciosa nem sempre garantida

- Algumas instâncias funcionam: problema do troco funciona para potências de dois.
- Verificar se um algoritmo ganancioso do problema do troco funciona para um determinado conjunto S: "A polynomial-time algorithm for the change-making problem" (`https://www.sciencedirect.com/science/article/abs/pii/S0167637704000823`)

Problema de Agendamento de Tarefas

- Conjunto de tarefas da forma (s_i, f_i).
- Tarefas são compatíveis se $f_i \leq s_j$ ou $f_j \leq s_i$.

- Isto é, enquanto se faz uma tarefa não se pode fazer outra.
- Queremos maximizar a quantidade de tarefas realizadas.

Em matematiquês, as tarefas são tratadas como *intervalos abertos*. O objetivo é criar um conjunto de intervalos de tamanho máximo de forma que todos os intervalos não tem intersecção par a par.

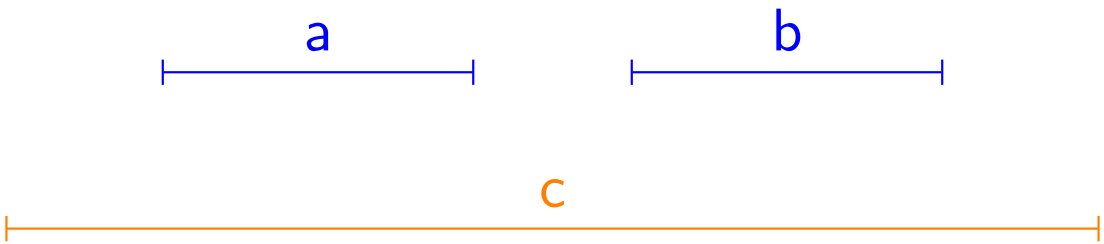
Qual critério utilizar?

Podemos pensar em alguns critérios para selecionar nossos intervalos:

- Escolher o próximo que começa primeiro
- Escolher o próximo que termina primeiro
- Escolher o próximo menor intervalo
- Escolher o próximo intervalo com menos intersecções

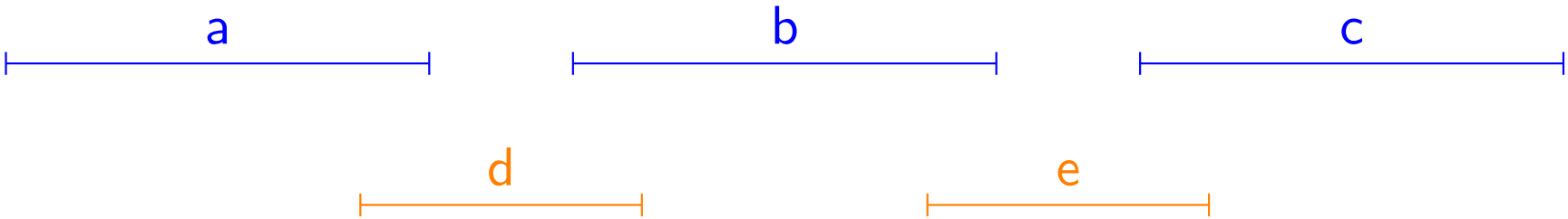
É útil pensar em contra-exemplos para cada um desses casos.

Contra-exemplo para intervalos que começam primeiro



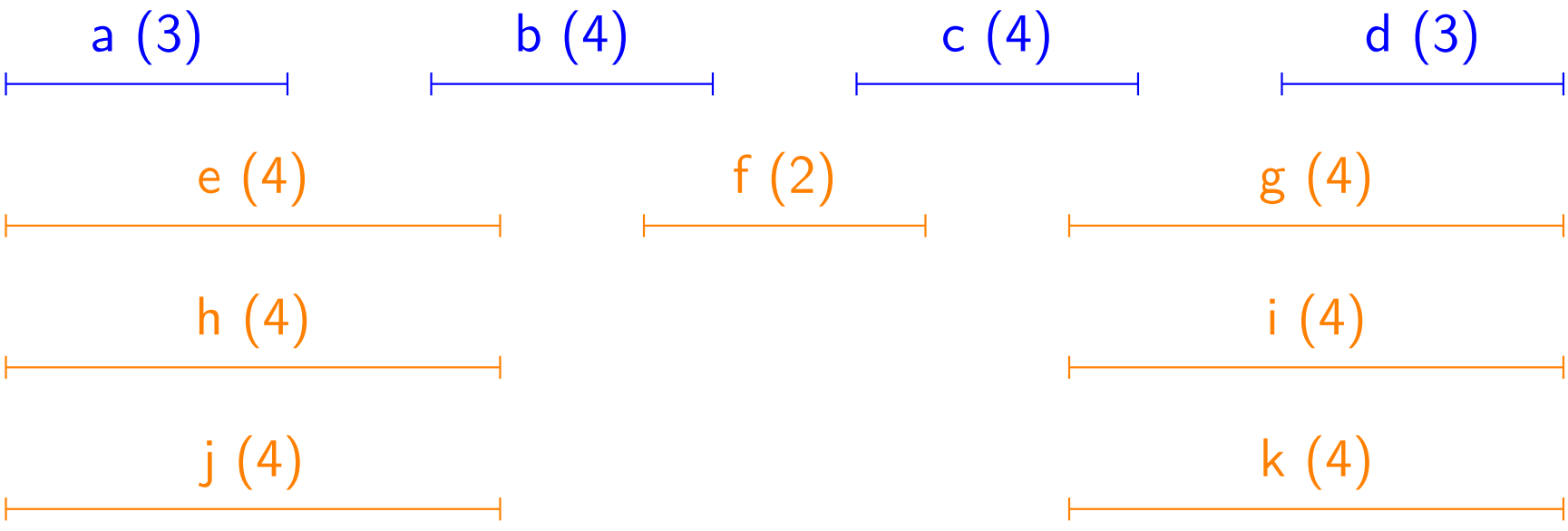
- Solução ótima: $\{a, b\}$
- Solução obtida: $\{c\}$

Contra-exemplo para menores intervalos



- Solução ótima: $\{a, b, c\}$
- Solução obtida: $\{d, e\}$

Contra-exemplo para intervalos com menos intersecções



- Solução ótima: $\{a, b, c, d\}$
- Solução obtida: $\{f, a, d\}$

Como provar a otimalidade de um algoritmo ganancioso?

- Temos que provar que o algoritmo não faz uma escolha ruim.
- A melhor forma é prova por contradição.
- Comparamos a solução do nosso algoritmo com uma suposta solução ótima diferente.
- Prova "sempre na frente": Se conseguirmos modificar a suposta solução ótima e conseguir uma solução melhor ainda, chegamos em uma contradição.
- Prova "não fica pra trás": Podemos mostrar que a solução do algoritmo é igual ou melhor que a solução ótima.
- Podemos pensar nas soluções como uma série de passos e olhamos para o primeiro passo onde as soluções diferem.

Provando que intervalos que terminam primeiro funciona

A solução do nosso algoritmo é S e de uma solução ótima é O. No primeiro passo onde a solução ótima O difere de S (O escolheu um intervalo diferente do que termina primeiro):

- Se o intervalo escolhido por O não tem intersecção com o intervalo de S, o intervalo de S ainda pode ser escolhido. Neste caso, criamos uma solução ótima $O^\star$:
  - Se a solução ótima escolhe o intervalo que termina primeiro em algum momento, apenas movemos a escolha para agora.
  - Se a solução ótima não escolhe o intervalo que termina primeiro em algum momento, adicionamos essa escolha, afinal ela ainda é possível.
- A solução $O^\star$ é maior (uma contradição) ou igual a O.
- Se o intervalo escolhido por O tem intersecção com o de S, podemos trocar os intervalos; como o intervalo de S termina mais cedo, é tão bom quanto o algoritmo ótimo.

Falta formalidade, mas nessa prova temos um argumento indutivo que altera O de forma que o nosso algoritmo "nunca fica pra trás".

Argumento da troca

A chave para achar um algoritmo ganancioso geralmente é a função de comparação.

- Você deve colocar dois elementos lado a lado e pensar quando que vale a pena trocar a ordem.
- Em alguns casos, pode ser necessário pegar três ou quatro elementos.
- As vezes você vai obter comparadores inválidos que com algum trabalho podem ser arrumados.
- Você pode combinar duas regras diferentes somando inequações.

O que comparadores customizados devem respeitar

A sua função de comparação de implementar algo parecido com o operador $<$ e respeitar a ordenação fraca estrita[7]. Ao desrespeitá-las, o comportamento é arbitrário e pode dar erro na execução (Runtime Error).

- $a \not< a$ (irreflexibilidade).
- Se $x < y$, então $y \not< x$ (antissimetria).
- Se $x < y$ e $y < z$, então $x < z$ (transitividade).
- Se $x \not< y$ e $y \not< x$ ($x = y$) e $y \not< z$ e $z \not< y$ ($y = z$), então $x \not< z$ e $z \not< x$ ($x = z$). (transitividade de equivalência).

Antissimetria pode ser deduzido da irreflexibilidade e transitividade.

[7] `https://codeforces.com/blog/entry/72525`

Comparadores que quebram as regras

- O comparador $x \leq y$ quebra a irreflexibilidade ($x = y$).
- O comparador $((x - y) \bmod 3) = 1$ quebra a transitividade ($f(1,2) = 0$ e $f(2,3) = 0$ mas $f(1,3) = 1$).
- O comparador $x + 1 < y$ quebra a transitividade de equivalência ($f(0,1) = 0$ e $f(1,0) = 0$ e também $f(1,2) = 0$ e $f(2,1) = 0$, mas $f(0,2) = 1$).

Recomendação de leitura

O Livro Olympiad Combinatorics de Paranav A. Sriram[8].

[8] https://archive.org/details/olympiad-combinatorics