Construção e Análise de Algoritmos

aula 15: Algoritmos gulosos

1 Introdução

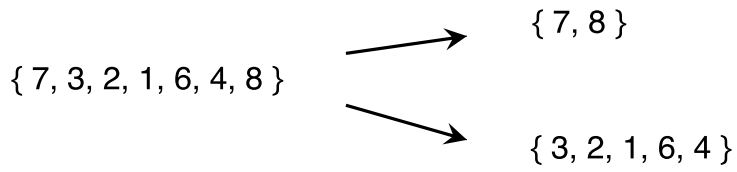
Até o momento, na disciplina, nós construímos algoritmos de dois tipos

- algoritmos que *fazem alguma coisa* — por exemplo, ordenar uma lista, multiplicar inteiros, matrizes ou polinômios, etc.
- algoritmos que *descobrem alguma coisa* — por exemplo, a posição de um certo elemento em uma lista, ou a mediana de um conjunto de números, etc.

A partir de hoje, nós vamos investigar algoritmos de um terceiro tipo:

- algoritmos que *encontram a melhor solução possível* para um problema

Por exemplo, considere o problema de dividir uma lista de números em dois subconjuntos cujas somas sejam as mais próximas possíveis

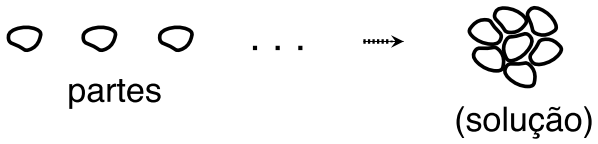


(Você consegue construir um algoritmo (eficiente) que resolve esse problema?)

Problemas desse tipo são conhecidos como *problemas de otimização*.

Algoritmos gulosos

Tipicamente, a solução de um problema de otimização é um agregado de várias partes, e o processo que encontra a solução envolve identificar essas partes uma a uma.



A ideia geral dos algoritmos gulosos consiste em

- a cada passo, *encontrar melhor parte possível* e incorporá-la à solução que está sendo construída

Vejamos um exemplo.

A solução do problema de divisão mencionado acima pode começar colocando o maior elemento da lista, digamos, no primeiro subconjunto (pois, afinal de contas, em algum lugar ele deve ficar).

Esse movimento cria uma diferença entre os dois subconjuntos.

A ideia, a seguir, consiste em tentar reduzir essa diferença ao máximo a cada passo.

Isso corresponde a adotar a seguinte regra:

> - a cada passo, escolher o elemento cujo valor é o mais próximo da diferença entre os dois subconjuntos, e colocá-lo no subconjunto com a menor soma

Por exemplo,

- $\{\, 7,\ 3,\ 2,\ 1,\ 6,\ 4,\ \not{8}\,\} \quad \Rightarrow \quad [\,8\,];\ [\,]$
- $\{\, \not{7},\ 3,\ 2,\ 1,\ 6,\ 4,\ \not{8}\,\} \quad \Rightarrow \quad [\,8\,];\ [\,7\,]$
- $\{\, \not{7},\ 3,\ 2,\ \not{1},\ 6,\ 4,\ \not{8}\,\} \quad \Rightarrow \quad [\,8\,];\ [\,7,\ 1\,]$
- $\{\, \not{7},\ 3,\ \not{2},\ \not{1},\ 6,\ 4,\ \not{8}\,\} \quad \Rightarrow \quad [\,8,\ 2\,];\ [\,7,\ 1\,]$
- $\{\, \not{7},\ \not{3},\ \not{2},\ \not{1},\ 6,\ 4,\ \not{8}\,\} \quad \Rightarrow \quad [\,8,\ 2\,];\ [\,7,\ 1,\ 3\,]$
- $\{\, \not{7},\ \not{3},\ \not{2},\ \not{1},\ 6,\ \not{4},\ \not{8}\,\} \quad \Rightarrow \quad [\,8,\ 2,\ 4\,];\ [\,7,\ 1,\ 3\,]$
- $\{\, \not{7},\ \not{3},\ \not{2},\ \not{1},\ \not{6},\ \not{4},\ \not{8}\,\} \quad \Rightarrow \quad [\,8,\ 2,\ 4\,];\ [\,7,\ 1,\ 3,\ 6\,]$

Mas, infelizmente, essa não é uma solução ótima para o problema.

Quer dizer, o nosso algoritmo não tem a garantia de encontrar a melhor solução possível em todos os casos.

E a razão é a seguinte:

- ele procura manter a diferença entre os dois subconjuntos no menor valor possível *durante* o processo de solução
- mas, o problema requer apenas que a diferença seja a menor possível ao *final* do processo

De fato, essa é a limitação básica de todo algoritmo guloso:

> - *nem sempre a escolha pela opção que parece melhor no momento nos leva ao melhor resultado possível no longo prazo*[*]

Mas o interessante é que às vezes essa ideia funciona!

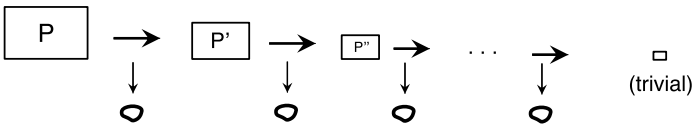
É bom saber disso porque os algoritmos gulosos, em geral, são algoritmos muito simples, e faz sentido testar a estratégia gulosa quando encontramos um problema de otimização.

[*] como muitos precisam aprender de maneira dolorosa ...

Outra razão para estudar os algoritmos gulosos é que eles nos dão mais um exemplo da *técnica de divisão e conquista*.

Tipicamente, a aplicação da regra gulosa reduz o nosso problema de otimização original a um novo problema do mesmo tipo, um pouquinho menor.

E isso significa que a aplicação sucessiva da regra gulosa eventualmente transforma o problema original em um problema trivial



A etapa de conquista consiste apenas em juntar as partes selecionadas pelas escolhas gulosas para formar a solução.

Portanto, como é usual nas aplicações de divisão e conquista, uma vez que você define a regra de escolha gulosa, você já tem essencialmente o seu algoritmo guloso para o problema.

(A única dúvida é saber se ele encontra uma solução ótima ou não ...)

A seguir, nós vamos ver um exemplo onde a estratégia gulosa tem sucesso.

2 O problema da seleção de atividades

> *Todo semestre a mesma situação se repete no departamento de Computação.*
>
> *Os professores ficam bolando nas férias atividades práticas que tornariam as suas disciplinas mais interessantes.*
>
> *Mas, quando começam as aulas, não é possível alocar todas elas no único laboratório do departamento.*
>
> *E daí, começa o debate.*
>
> *Um diz: "Eu deveria ter prioridade, porque a minha disciplina começa bem cedinho, e nessa hora não tem ninguém querendo utilizar o laboratório ainda".*
>
> *Outro diz: "Eu deveria ter prioridade, porque a minha atividade é bem curtinha.*
>
> *Ainda um outro diz: "Eu deveria ter prioridade porque na hora da minha atividade não tem quase ninguém querendo utilizar o laboratório".*
>
> *E por aí vai ...*
>
> *Como é impossível atender a todos, a coordenação tenta fazer as coisas de modo a atender a maior quantidade possível de professores.*
>
> *Mas, como é que se pode fazer isso?*

Pronto, nós já temos aqui o nosso problema de otimização.

A primeira observação é que as sugestões dos professores correspondem a diferentes estratégias.

O primeiro está dizendo

> `R1`: escolha a atividade que começa mais cedo,
> elimine as atividades que conflitam horário com ela,
> e continue dessa maneira

O segundo está dizendo

> `R2`: escolha a atividade mais curta,
> elimine as atividades que conflitam horário com ela,
> e continue dessa maneira

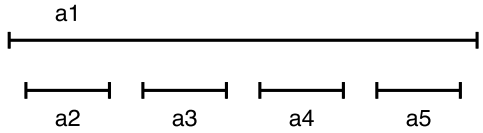
O terceiro está dizendo

> `R3`: escolha a atividade com menos conflitos,
> elimine as atividades que conflitam horário com ela,
> e continue dessa maneira

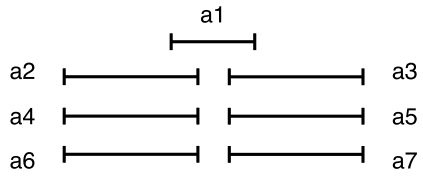
Mas, nenhuma delas tem a garantia de produzir a solução ótima (em todos os casos).

Vejamos.

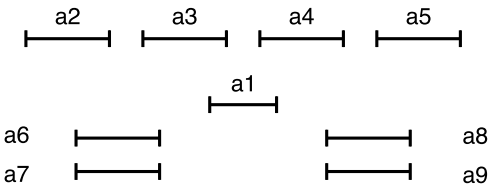
No caso da regra `R1`, é fácil ver que a atividade que começa mais cedo pode ocupar o laboratório o dia inteiro e conflitar com todas as outras



No caso da regra `R2`, também é fácil ver que a atividade mais curta também pode conflitar com todas as outras



Já a regra `R3` parece um pouco mais promissora, mas o exemplo abaixo mostra que ela também não é uma estratégia ótima



Note que a_1 é a atividade com menos conflitos.

Mas, quando nós incluímos a_1 na solução, nós só podemos selecionar mais duas atividades.

Por outro lado, deixando a_1 de fora, nós podemos construir a seguinte solução: $\{a_2, a_3, a_4, a_5\}$.

Mas, se as regras `R1, R2, R3` não funcionam, será que existe alguma estratégia gulosa que pode funcionar?

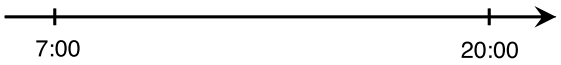
3 Estratégia gulosa ótima

Sim! esse problema possui uma estratégia gulosa ótima.

Mas, para ver como ela funciona, é preciso olhar para o problema de maneira ligeiramente diferente.

Isto é, ao invés de focar nas atividades, nós vamos concentrar a nossa atenção no período total de tempo que nós temos o laboratório à nossa disposição.

Por exemplo,



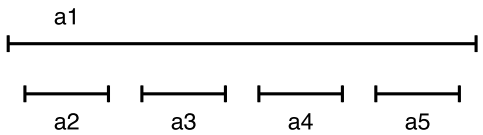
E a ideia, agora, é administrar esse período de maneira gulosa.

Isto é, nós vamos alocar atividades nesse período da esquerda para a direita, tentando manter a maior quantidade possível de tempo livre para outras atividades.

Em outras palavras, nós vamos selecionar, a cada passo, a atividade que termina mais cedo (e eliminar todas as atividades que conflitam com ela).

Veja como a coisa funciona.

No primeiro exemplo,



isso corresponde a escolher a atividade a_2 primeiro.

Com essa escolha, a atividade a_1 é eliminada, e em seguida as atividades a_3, a_4 e a_5 são selecionadas em sequência.

Isso nos dá a solução $\{a_2, a_3, a_4, a_5\}$, que é a solução ótima.

Não é difícil ver que essa estratégia também encontra a solução ótima nos outros dois exemplos acima.

(Verifique isso!)

Mas, será que essa estratégia gulosa encontra a solução ótima em todos os casos?

4 Prova de otimalidade

Essa é a parte que dá mais trabalho na construção de um algoritmo guloso.

Quer dizer, é fácil encontrar estratégias gulosas para um problema de otimização.

Mas, quando encontramos uma que parece funcionar bem, é preciso formular um argumento que comprova que o algoritmo realmente encontra uma solução ótima em todos os casos.

Felizmente, existem algumas estratégias padrão para construir esse argumento.

A mais comum delas consiste em comparar a solução encontrada pelo nosso algoritmo guloso com uma outra solução qualquer.

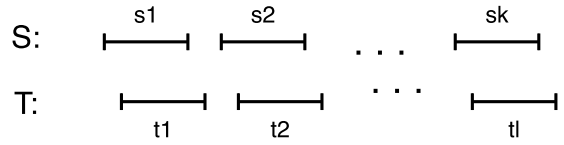
Vejamos.

Denote a solução encontrada pelo algoritmo guloso por S.

E seja T uma outra solução que alguém encontrou (sabe-se lá como ...).

Toda solução para o problema consiste de um subconjunto de atividades que não conflitam entre si.

Logo, as atividades das duas soluções podem ser organizadas na ordem em que elas ocorrem.



E a ideia agora é comparar as duas soluções.

A primeira observação é que s_1 é a atividade que termina mais cedo dentre todas as atividades.

Isso significa que t_1 não pode terminar antes que s_1, o que nos dá duas possibilidades:

E aqui nós podemos ver claramente a ideia da nossa estratégia gulosa em ação: a solução S sempre fica atrás de outras soluções (ou ao menos empatada).

E agora vem a observação chave:

- Toda atividade que conflita com s_1 também conflita com t_1.

Porque?

A resposta é simples: para ter uma atividade que conflita com s_1 mas não com t_1, nós deveríamos ter algo como

Mas isso não pode ser, porque s_1 é a atividade que termina mais cedo de todas.

Essa observação é útil pois ela implica que t_2 não conflita com s_1.

(Porque?)

Isso significa que t_2 é uma das atividades que está à disposição do algoritmo guloso na sua segunda escolha.

E isso, por sua vez, significa que s_2 termina antes, ou no mesmo instante, que t_2

(Você consegue ver isso?)

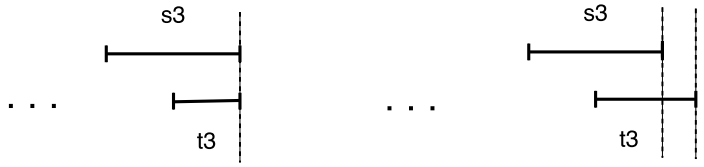
Ou seja, a solução gulosa continua um pouquinho atrás da solução T (ou ao menos empatada).

Nesse ponto, nós podemos repetir o raciocínio acima para observar que

- Toda atividade que conflita com $\{s_1, s_2\}$ também conflita com $\{t_1, t_2\}$.

(Tente fazer isso!)

E isso nos permite avançar mais um passo no argumento e concluir que s_3 termina antes, ou no mesmo instante, que t_3



A observação final é que, prosseguindo dessa maneira, a solução S nunca pode acabar antes da solução T.

Isso é o caso porque todas as atividades que ainda restam para T, em algum momento, também estão disponíveis para S, de modo que o algoritmo guloso sempre pode fazer mais uma escolha.

(Pense um pouco sobre isso.)

A consequência imediata dessa última observação é que o número de atividades em S é maior ou igual ao número de atividades em T.

Esse argumento mostra que não pode existir uma solução para o problema com mais atividades que S.

Logo, S é uma solução ótima! :)

5 Algoritmo e análise de complexidade

Agora que sabemos que a nossa estratégia gulosa é ótima, só nos resta escrever o algoritmo e analisar o seu tempo de execução.

Como todo algoritmo de divisão e conquista, é natural implementá-lo na forma de um procedimento recursivo.

```
   Procedimento  SelAtiv-Gul ( A: coleção com n atividades )
    {
1.      Se ( n = 0 )  Retorna ( vazio )

2.   a-gul  <--  atividade que termina mais cedo em A

3.      A'  <--  Elimina-conflitos ( A, a-gul )

4.       S  <--  SelAtiv-Gul ( A')

5.       S  <--  S + a-gul

6.       Retorna (S)
   }
```

Para analisar a complexidade do algoritmo, é preciso estimar os tempos de execução das linhas `2` e `3` (que não são necessariamente constantes).

E, para fazer essa estimativa, é preciso pensar sobre como essas operações são implementadas.

Uma ideia simples consiste em ordenar a lista de atividades `A` logo no início (i.e., antes da primeira chamada a `SelAtiv-Gul`), de acordo com o tempo de encerramente das atividades — o que leva tempo $\Theta(n \log n)$.

Fazendo isso, a atividade que termina mais cedo é sempre a primeira da lista — e, assim, a linha `2` do algoritmo executa em tempo $O(1)$.

A seguir, é preciso remover da lista todas as atividades que começam antes do término da atividade `a-gul` (pois essas são as atividades que conflitam com ela).

O problema é que essas atividades podem estar espalhadas por toda a lista `A`.

Aqui, é preciso uma pequena esperteza.

A ideia é manter uma outra versão da lista `A`, digamos `A-aux`, ordenada de acordo com o tempo de início das atividades — o que significa mais $\Theta(n \log n)$ no tempo total.

Dessa maneira, as atividades que conflitam com `a-gul` aparecem no início da lista `A-aux`.

Ótimo!

Mas, não basta remover essas atividades da lista `A-aux`, é preciso removê-las da lista `A`.

E isso é feito da seguinte maneira:

- Para cada atividade em `A-aux` que começa antes do termino de `a-gul`
- nós examinamos o seu tempo de término, e fazemos uma busca binária por ela na lista `A`
- uma vez que a atividade é encontrada, ela é marcada como removida

Dessa maneira, cada atividade conflitante é removida na linha `3` em tempo $O(\log n)$.

No final das contas, o procedimento `SelAtiv-Gul` basicamente percorre a lista `A`, selecionando ou removendo atividades, a um custo de $O(\log n)$ por atividade.

Isso significa que o tempo de execução do procedimento é $O(n \log n)$.

Juntando a isso o tempo das ordenações, antes do procedimento começar, nós obtemos

$$\Theta(n \log n) \quad + \quad O(n \log n) \quad = \quad \Theta(n \log n)$$