Algoritmos gulosos

Fabio Lubacheski
fabio.lubacheski@mackenzie.br

Algoritmos gulosos – gananciosos ou greedy

- São aqueles que, a cada iteração:
  - Sempre escolhem a alternativa que parece mais **apetitosa** (**promissora**) naquele instante. O objeto escolhido passa a fazer parte da solução que o algoritmo constrói.
  - **Nunca** reconsideram essa decisão, ou seja, uma escolha que foi feita nunca é revista. As escolhas que faz em cada iteração são definitivas.

- O que é uma alternativa **promissora** (**apetitosa**)?
  - Depende do problema, do que se quer maximizar ou minimizar

    Ex: caminho mais curto ou mais rápido, menor número de jogadas etc

Algoritmos gulosos – será que sempre funciona ?

- Exemplo problema da banda U2:

  A banda U2 vai começar um show daqui a 17 minutos. Os quatros integrantes estão na margem esquerda de um rio e precisam cruzar uma ponte para chegar ao palco na margem direita. É noite. Há somente uma lanterna. Só podem passar uma ou duas pessoas juntas pela ponte, e sempre com a lanterna. A lanterna não pode ser jogada, etc.

  Cada integrante possui um tempo diferente para atravessar a ponte: O Bono leva 1 minuto, o Edge 2 minutos, o Adam 5 minutos e o Larry 10 minutos. Quando os dois atravessam juntos, eles vão pelo tempo do mais lento. Você precisa ajudá-los a atravessar com o menor tempo possível.

Algoritmos gulosos – solução ótima global

- Por fazer a escolha que parece ser a melhor a cada **iteração**, diz-se que a escolha é feita de acordo com um critério guloso – **escolha ótima local**! Ou seja, fazemos uma **escolha ótima local**, na esperança de obter uma **solução ótima global**.
- Dizemos que os algoritmos gulosos são “míopes”: eles tomam decisões com base nas informações disponíveis na iteração corrente, sem olhar as consequências que essas decisões terão no futuro, sem se arrepender ou voltar atrás.
- Por conta disso, os algoritmos gulosos podem não chegar a uma **solução ótima** para uma determinada entrada de um problema.
- Tipicamente algoritmos gulosos são utilizados para resolver **problemas de otimização combinatória** que funcionem através de uma sequência de passos.

Exemplo de um problema que se resolve com algoritmo guloso

- **Troco mínimo:**

  Dados os valores de moedas determinar o **numero mínimo de moedas** para dar um valor de troco.
- **Escolha gulosa:** A cada passo do algoritmo escolher a moeda de maior valor, sem refazer suas decisões.
- **Exemplo:**

  Moedas = {100, 50, 25, 10, 1} valor do troco = 37, qual o número mínimo de moedas para o valor acima ?

Exemplo de um problema que se resolve com algoritmo guloso

- **Troco mínimo:**

  Dados os valores de moedas determinar o **numero mínimo de moedas** para dar um valor de troco.
- **Escolha gulosa:** A cada passo do algoritmo escolher a moeda de maior valor, sem refazer suas decisões.
- **Exemplo:**

  Moedas = {100, 50, 25, 10, 1} valor do troco = 37, qual o número mínimo de moedas para o valor acima ?

  Para o troco = 37 utilizamos o seguinte conjunto mínimo de moedas:

  Solução = {25, 10, 1, 1}

Ideia do algoritmo para cálculo do troco

1) O algoritmo recebe um conjunto `C={100,50,25,10,1}` com os valores das moedas já em ordem decrescente e um determinado **valor do troco** que se deseja calcular. A saída do algoritmo é a sequência **`S`** de moedas usadas para determinar o troco.

2) Enquanto a soma das moedas não atingir o valor passado como parâmetro e ainda tivermos moedas no conjunto `C`, são feitos os dois passos:

   2.1) Escolhe-se a moeda de maior valor em `C`;

   2.2) Se o valor da moeda mais soma de valores for menor ou igual ao que valor do troco, então soma-se o valor da moeda na soma de valores e acrescente a moeda a sequência `S`. Caso contrário a moeda é retirada do conjunto `C`.

Algoritmo troco

```
Algoritmo Troco( C, valor)
Entrada: um conjunto de moedas C contendo os n valores distintos, em ordem decrescente
         de moedas  e um valor de troco.
Saída: S é a solução com as moedas usadas
início
     S ← ∅
     soma ← 0
     enquanto  soma < valor E C ≠ {} faça
        x ← moeda de maior valor em C
        se soma + x  <= valor então
            soma ← soma + x;
            S ← S U {x}
        senão
            C ← C – {x}
        fim-se
     fim-enquanto
     se soma = valor então
        retornar S
     senão
        retornar "Não encontrei solução"
     fim-se
fim
```

Tem algum valor de troco que o algoritmo encontra uma solução que não é ótima ?

Problema de seleção de atividades

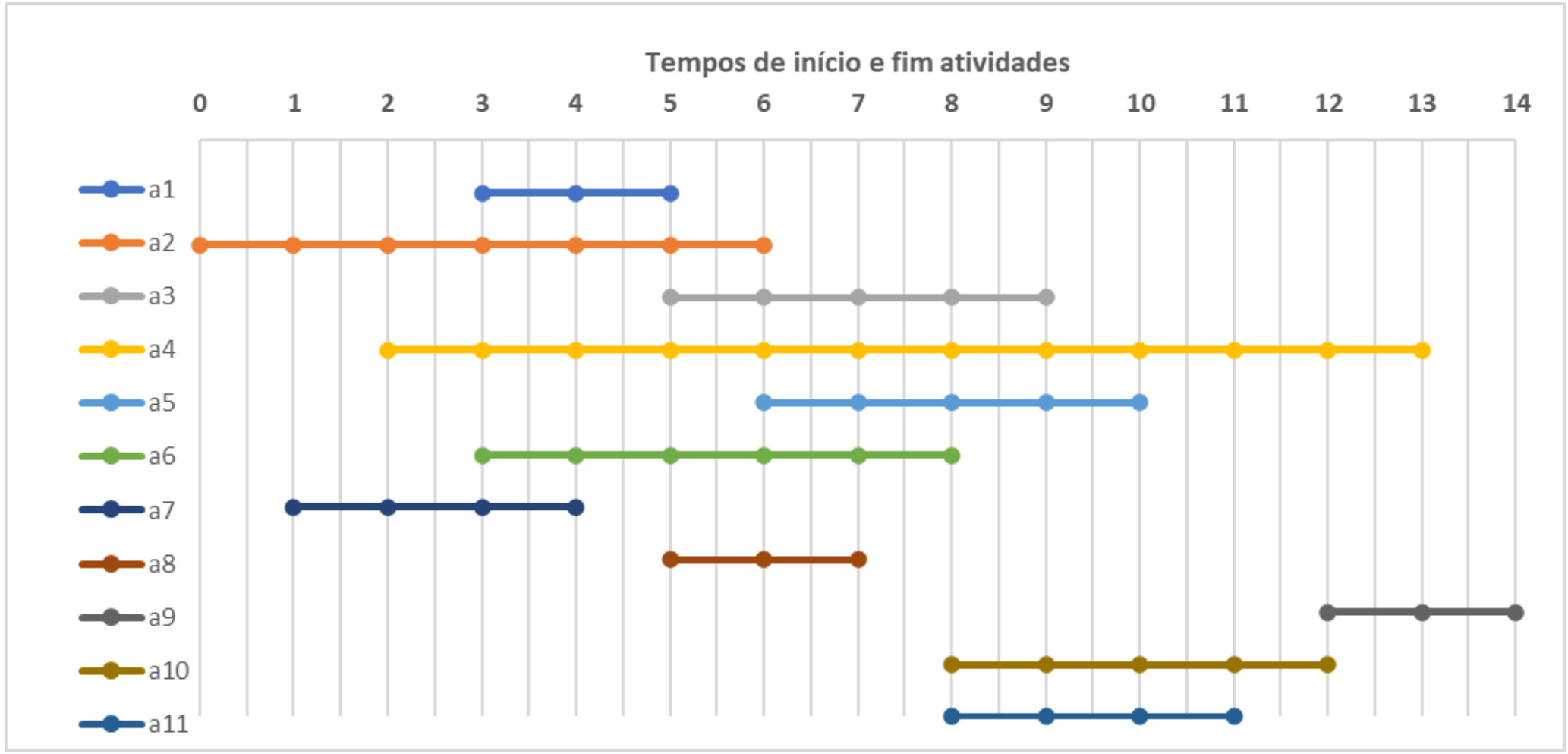
- **Seleção de atividades:**
  - Existem diversas atividades (por exemplo aulas) que querem usar um mesmo recurso (por exemplo uma sala de aulas).
  - Cada atividade tem um horário de início e um horário de fim.
  - Duas atividades não podem usar o mesmo recurso ao mesmo tempo (por exemplo duas aulas não podem ser ministradas na mesma sala ao mesmo tempo).

- Como selecionar um conjunto máximo de atividades (aulas) sem que haja sobreposição de tempo (compatíveis).

Exemplo de atividades e o seus respectivos tempo de início e fim

11 atividades com **15** unidades de tempo

A={(3,5),(0,6),(5,9),(2,13),(6,10),(3,8),(1,4),(5,7),(12,14), (8,12),(8,11)}



Formalizando o problema da Seleção de atividades

Dado uma coleção de atividades, $A=\{a_1, a_2, ..., a_n\}$ para serem executadas onde cada atividade a_i tem tempo de início s_i, e um tempo de término f_i, onde $s_i < f_i$, ou seja,

$a_i = (s_i, f_i)$. O primeiro número do par é o *início* do intervalo e o segundo é o *término*. (As letras s e f lembram *start* e *finish* respectivamente.)

Como determinar um subconjunto de atividades, sem sobreposição de tempo (**compatíveis**), de tamanho máximo de A, ou seja, uma solução ótima ?

Seleção de atividades

- Exemplo de atividades do conjunto:

$$A=\{(3,5),(0,6),(5,9),(2,13),(6,10),(3,8),(1,4),(5,7),(12,14),\ (8,12),(8,11)\}$$

- Dado duas atividades a_i e a_j dizemos que elas são **compatíveis** se:

$$f_i \leq s_j \text{ ou } f_j \leq s_i$$

Se $f_i \leq s_j$ a atividade a_j inicia depois que a_i termina,
exemplo as atividades a_i= (1,4) e a_j=(5,7)

Se $f_j \leq s_i$ a atividade a_i inicia depois que a_j termina,
exemplo as atividades a_i= (12,14) e a_j=(8,11)

Seleção de atividades

- **Ideia do algoritmo**
  - Ordene as atividades por ordem crescente de término
  - A cada iteração do algoritmo, escolha uma atividade compatível que acaba mais cedo, ou seja, a atividade a_j inicia depois que a_i termina ($s_j >= f_i$).

Algoritmo seleção de atividades

```
Algoritmo SelecaoAtividades(A)
  Entrada: Um conjunto de tarefas A={a1,a2,....,an} onde ai é uma tarefa com
           (si,fj) e temos n tarefas no conjunto A.
  Saída: Um subconjunto de atividades S com a solução ótima.
início
    Ordene as atividades em ordem crescente de término
    S ← {a1}
    i ← 1
    para j de 2 até n faça
        se sj >= fi então # a atividade aj inicia depois que ai termina
            S ← S U {aj}
            i ← j
        fim-se
    fim-para
    retorne S
fim.
```

Algoritmo seleção de atividades

- **Complexidade do algoritmo**

  O algoritmo consome $O(n)$ unidades de tempo. Isso não inclui o tempo $O(n \log n)$ necessário para fazer a ordenação prévia das atividades, mas mesmo levando em conta esse tempo adicional.

Código de Huffman

- O **código de *Huffman*** é uma codificação de caracteres que permite compactar arquivos de texto, ou seja, converter um **arquivo de texto** (**entrada**) em um arquivo de bits bem menor (**compactado/saída**).

- A partir da frequência dos caracteres no arquivo texto é obtido um **código binário único** (**código prefixo**), de forma que os caracteres que tenham a maior frequência no arquivo tenham os códigos prefixo de menor comprimento, caracterizando assim uma estratégia gulosa.

- Para gerar o **código prefixo** utilizamos o **Algoritmo de Huffman**, proposto por David Huffman (1952), tem como entrada a **lista de frequência de caracteres** e gera como saída a **Árvore de Huffman,** uma **árvore estritamente binária**, ou seja, cada nó tem sempre dois ou nenhum filho.

Algoritmo de Huffman

```
Algoritmo de Huffman
   Entrada: Uma lista L de caracteres e suas frequências.
   Saida: raiz da árvore de Huffman
Inicio
   n = |L|
   para i = 1 até n-1 faça
      z <- CriaNo()
      z.esq <- ExtraiMinimoLista( L )
      z.dir <- ExtraiMinimoLista( L )
      z.freq <- z.esq.freq + z.dir.freq
      InsereNaLista(L,z)
   fim-para
   retorne ExtraiMinimoLista( L )
Fim
```

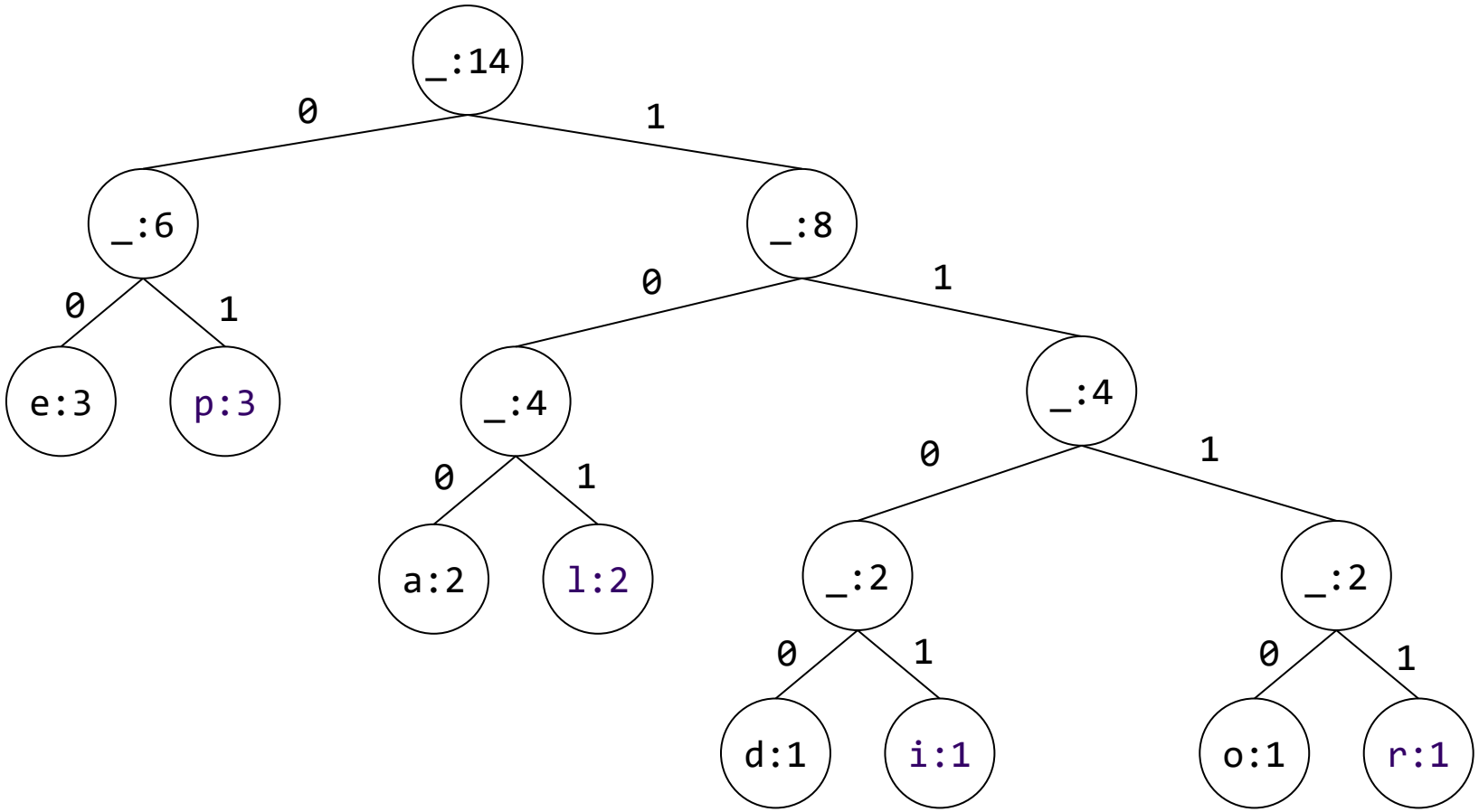
Execução do Algoritmo de Huffman

- Suponha que o arquivo de entrada tenha a sentença `paralelepipedo` para codificação. A sentença tem a seguinte **tabela de frequência de caracteres**:

Caractere	frequência
‘a’	2
‘d’	1
‘e’	3
‘i’	1
‘l’	2
‘o’	1
‘P’	3
‘r’	1

- Com esse conjunto de caracteres e frequências poderemos construir a seguinte **Árvore de Huffman**.

Árvore de *Huffman*



Execução do Algoritmo de Huffmam

- Após construir a **Árvore *Huffman*** associa-se os caracteres do conjunto a seu código binário, isso é feito, realizando um percurso na árvore, para cada um dos caracteres, a partir da raiz da árvore até a folha que o caractere representa.

Caractere	frequência	código prefixo
‘a’	2	100
‘d’	1	1100
‘e’	3	00
‘i’	1	1101
‘l’	2	101
‘o’	1	1110
‘P’	3	01
‘r’	1	1111

Importante: Note que nenhum código binário de um caractere é prefixo de outro código binário de outro caractere, e os caracteres com maiores frequências tem o código prefixo de menor comprimento.

Execução do Algoritmo de Huffam

- Para converter a sentença do arquivo de entrada em uma sequência de `0` e `1` basta substituir os seus caracteres pela código prefixo correspondente, na mesma ordem que aparece na entrada, como mostra a seguir:

  Sentença entrada: `paralelepipedo`

  Sentença codificada: `01100111110010100101000111010100110011110`

Complexidade do Algoritmo de Huffam

- Para facilitar a busca do elemento do caractere de menor frequência podemos trabalhar com uma lista em ordem crescente de frequência dos caracteres, para gerar a lista ordenada teríamos uma complexidade de $O(nlogn)$.

- A cada iteração do algoritmo de *Huffman* os dois menores elementos são obtidos na lista e inserido um novo elemento, a inserção na lista em ordem crescente pode consumir tempo $O(n)$, onde n é o tamanho da lista, e como o algoritmo repete de 1 até $n-1$ vezes e a cada iteração temos operações de inserir na lista podemos dizer que a complexidade total do algoritmo é $O(n^2)$.

- Se a lista fosse implementada como um **Heap** a inserção poderia ser feita em $O(logn)$, com isso o tempo total seria $O(nlogn)$.

Exercícios

1) Leia um valor de ponto flutuante com duas casas decimais. Este valor representa um valor monetário. A seguir, calcule o menor número de notas e moedas possíveis no qual o valor pode ser decomposto. As notas consideradas são de `100, 50, 20, 10, 5, 2`. As moedas possíveis são de `1, 0.50, 0.25, 0.10, 0.05 e 0.01`. O seu programa deve a relação de notas necessárias, por exemplo para o valor `576.73` teríamos:

```
NOTAS:                          MOEDAS:
5 nota(s) de R$ 100.00          1 moeda(s) de R$ 1.00
1 nota(s) de R$ 50.00           1 moeda(s) de R$ 0.50
1 nota(s) de R$ 20.00           0 moeda(s) de R$ 0.25
0 nota(s) de R$ 10.00           2 moeda(s) de R$ 0.10
1 nota(s) de R$  5.00           0 moeda(s) de R$ 0.05
0 nota(s) de R$  2.00           3 moeda(s) de R$ 0.01
```

Exercícios

2) No correio local há somente selos de 3 e de 5 centavos. A taxa mínima para correspondência é de 8 centavos. Faça um programa que determina o menor número de selos de 3 e de 5 centavos que completam o valor de uma taxa dada.

3) Execute o algoritmo *SelecaoAtividades*(A) o conjunto de atividades especificadas pelos pares (s_i,f_i) descritos abaixo, onde s_i é o tempo de início da atividade e f_i o tempo de término da atividade :

   A={ (7,9), (5.6), (4,9), (1,2), (3,7), (6,8), (2,5), (1,3), (1,4)}

4) Considere a estratégia a seguir para o problema de Seleção de Atividade: A cada iteração escolher uma atividade compatível que comece primeiro. Essa estratégia produz uma solução ótima?

5) Implemente versão recursiva do o algoritmo *SelecionaAtividades*(*A*) na linguagem C.

Exercícios

6) Determine os códigos e as árvores de *Huffman* para um texto com os seguintes caracteres e frequências:

   a) $a = 7$, $b = 5$, $c = 10$, $d = 21$, $e = 90$, $f = 11$, $g = 7$ e $h = 2$;

   b) $a = 1$, $b = 1$, $c = 2$, $d = 3$, $e = 5$, $f = 8$, $g = 13$ e $h = 21$;

   c) $a = 45$, $b = 13$, $c = 12$, $d = 16$, $e = 9$, $f = 5$.

7) Descreva a árvore de *Huffman* quando as frequências são os primeiros n números de ***Fibonacci***. Pra começar teste para $n = 10$.

8) Escreva uma função que tendo como entrada uma sequência de caracteres, o seu algoritmo gera uma tabela de frequência de caracteres gastando tempo $O(n)$, onde n é o tamanho da sequência de caracteres.

Exercícios

9) Implemente na linguagem C o algoritmo de *Huffman* apresentando, considere que o algoritmo tem como entrada uma **lista de caracteres** e com suas frequências e como saída a **árvore binária de Huffman**.

10) Escreva uma função que tendo como entrada a raiz da árvore de *Huffman,* apresenta o código prefixo de cada um dos caracteres na árvore.

11) Escreva uma função que tendo como entrada a raiz da árvore de *Huffman* e um código prefixo, o método retorna o caractere correspondente ao código prefixo.

12) Dado uma sequência de códigos prefixos e a raiz de uma árvore de *Huffman*, escreva uma função que “descompacta” a sequência retornando uma sequência com os caracteres “descompactados”.

Exercícios

13) ENADE 2017:

QUESTÃO 22

Um país utiliza moedas de 1, 5, 10, 25 e 50 centavos. Um programador desenvolveu o método a seguir, que implementa a estratégia gulosa para o problema do troco mínimo. Esse método recebe como parâmetro um valor inteiro, em centavos, e retorna um *array* no qual cada posição indica a quantidade de moedas de cada valor.

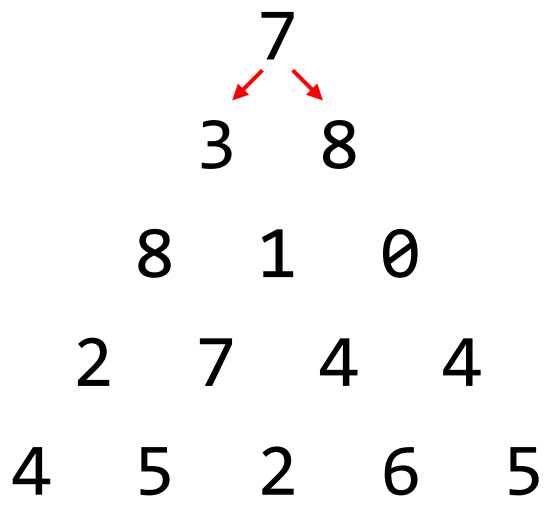
```
public static int[] troco(int valor){
    int[] moedas = new int[5];

    moedas[4] = valor / 50;
    valor = valor % 50;
    moedas[3] = valor / 25;
    valor = valor % 25;
    moedas[2] = valor / 10;
    valor = valor % 10;
    moedas[1] = valor / 5;
    valor = valor % 5;
    moedas[0] = valor;
    return(moedas);
}
```

O algoritmo ao lado sempre encontra a solução global ótima para todas as entradas ?

Desafio

Escreva um algoritmo que calcula o caminho, que começa no topo da pirâmide e acaba na base, com maior soma. Em cada passo podemos ir diagonalmente para baixo e para a esquerda ou para baixo e para a direita.



Restrições: todos os números da pirâmide são inteiros entre `0` e `99` e o número de linhas do triângulo é no máximo `100`.

Fim