Lista de exercícios
Prof. João B. Oliveira

1 Algoritmos gulosos

1. Escreva um algoritmo que resolve o problema de achar o menor número de moedas para dar troco, em um país que tem moedas de 17, 8 e 1 centavos e depois confirme que ele funciona corretamente para todos os centavos entre 1 e 100. "Funcionar" significa dar o troco correto e com o menor número de moedas.

2. Explique por que o algoritmo de ordenação SelectionSort é um algoritmo guloso.

3. Um novo algoritmo para encontrar Árvores de Cobertura Mínima em um grafo G é assim:

   (a) Inicie em algum nodo u de G;

   (b) Siga a aresta de valor mais baixo ligada a u e que leva a um nodo ainda não visitado;

   (c) Repita o passo anterior até visitar todos os nodos de G.

   Agora responda duas perguntas:

   (a) Ele é guloso?

   (b) Ele acha uma Árvore de Cobertura Mínima?

4. Você tem uma série de atividades bem legais que podem ser realizadas no sábado, mas todas tem hora marcada para início e fim. Você gostaria de escolher o máximo de atividades possíveis para fazer, já sabendo que quando iniciar uma delas terá que ir até o final sem mudar para alguma outra. Você pensa em alguns algoritmos para escolher o máximo de atividades:

   (a) Escolher as atividades dando preferência para as mais curtas (por que ocupam o mínimo de tempo);

   (b) Escolher as atividades dando preferência para as que terminam o mais cedo possível;

   Teste seus dois algoritmos com as atividades abaixo:

Atividade	Hora Início	Hora Fim
1	2	4
2	1	4
3	2	7
4	4	8
5	4	9
6	6	8
7	5	10
8	7	9
9	7	10
10	8	11

5. Apresente um algoritmo guloso e de tempo $O(n)$ que opera sobre uma lista de inteiros $[a_1, a_2, a_3, \ldots, a_n]$ e encontra o local onde a soma dos números de um lado da sequencia é igual à soma dos números do outro lado (ou o mais parecida possível). **Seu algoritmo tem que respeitar uma regra extra: cada elemento da lista pode ser acessado somente uma vez.**

6. Uma coloração de um grafo não dirigido é a atribuição de cores aos nodos de forma que nenhum par de nodos vizinhos tenha a mesma cor. No problema de coloração de grafos, desejamos encontrar o número mínimo de cores necessárias para colorir um grafo não dirigido. Proponha um algoritmo guloso para resolver o problema e comprove que ele sempre funciona (se você conseguir provar que sempre funciona pode ganhar um milhão de dólares).

2 Divisão e conquista

1. Implemente pesquisa binária sobre um vetor de inteiros.
2. Teste sua pesquisa binária:
   (a) Crie um vetor de 20 inteiros inicializado com $(0, 1, 2, 3, 4, 5, 6, \ldots, 19)$.
   (b) Para cada um dos 20 elementos, teste sua pesquisa binária para ver se ela encontra o elemento corretamente.
3. Modifique sua pesquisa binária para que ela seja **ternária** e teste outra vez.
4. Implemente Mergesort, dividindo o vetor que deve ser ordenado em duas partes (Mergesort convencional).
5. Modifique seu Mergesort para que ele agora divida o vetor em três partes.
6. Implemente o algoritmo dos camponeses russos para multiplicação de inteiros.
7. Altere o algoritmo dos camponeses russos para que ele não faça mais multiplicação e divisão por 2, pois os camponeses agora querem que a multiplicação e a divisão sejam feitas com 5. Teste seu algoritmo para confirmar que funciona.
8. Escreva um programa que faz uma lista de todos os subconjuntos de elementos de um conjunto S.
9. Dado um array já ordenado de inteiros diferentes $A[0 \ldots n]$, você quer determinar se existe um índice i tal que $A[i] = i$. Forneça um algoritmo de divisão e conquista que resolva o problema em tempo $O(\log n)$.
10. Escreva um programa que recebe um conjunto S e um inteiro k e verifica se algum subconjunto de S tem soma igual a k.
11. Suponha que você esteja escolhendo entre os seguintes três algoritmos:
    (a) Algoritmo A resolve problemas dividindo-os em cinco subproblemas de metade do tamanho, resolvendo recursivamente cada subproblema e, em seguida, combinando as soluções em tempo linear.
    (b) Algoritmo B resolve problemas de tamanho n resolvendo recursivamente dois subproblemas de tamanho $n - 1$ e depois combinando as soluções em tempo constante.
    (c) Algoritmo C resolve problemas de tamanho n dividindo-os em nove subproblemas de tamanho $n/3$, resolvendo recursivamente cada subproblema e, em seguida, combinando as soluções em tempo linear.

    Quais são os tempos de execução de cada um desses algoritmos (em notação big-O) e qual você escolheria?
12. Escreva um programa que escreve todas as permutações de elementos que estão em um conjunto S.

3 Programação dinâmica

1. Implemente um algoritmo para calcular os números de Fibonacci que use memória $O(1)$.
2. Você pode construir uma calçada da sua casa até a casa da vovó usando pedras verdes, azuis e amarelas. As pedras são quadradas, tem 1 metro de largura e 1 metro de comprimento, e a casa da vovó fica a 50 metros de distância. Antes de comprar as pedras na loja de material de construção, você gostaria de responder a estas três perguntas:
   (a) Quantas calçadas com coloridos diferentes são possíveis?

(b) Quantas calçadas com coloridos diferentes são possíveis, se a vovó não quer que duas pedras amarelas fiquem uma ao lado da outra?

(c) ... e se a vovó também não deixa que duas pedras azuis fiquem lado a lado?

Você decide encarar esse problema com uma recursão simples e (se não der certo) pode tentar uma versão com memorização para acelerar as coisas.

3. Você achou esta estranha fórmula em um baú no sótão da vovó:

$$Z_n = \begin{cases} 3, & n = 0 \\ 4, & n = 1 \\ 5, & n = 2 \\ Z_{n-1} - 2Z_{n-2} + 3Z_{n-3}, & n > 2 \end{cases}$$

A partir disso, você decidiu investigar e fez um plano com estas etapas:

(a) Implementar o cálculo de Z_n usando recursão e calcular Z_{40}.

(b) Depois disso você resolve implementar o cálculo sem usar recursão, aproveitando um vetor de valores auxiliares.

(c) E depois você decide se livrar do vetor e usar apenas um punhado de variáveis.

4. O algoritmo abaixo calcula recursivamente alguma coisa interessante:

```
int coisa( int n, int k ) {
  if ( k == n ) return n;
  if ( k == 0 ) return 1;
  return coisa( n-1, k-1 ) - 3 * coisa( n-1, k );
}
```

Você sabe que ele sempre será chamado com $k \leq n$. A partir disso, faça uma versão não-recursiva deste algoritmo usando uma tabela auxiliar.

5. Depois de um tempo você acha uma anotação na última página do caderno de receitas da vovó:

Depois de décadas de tentativas, finalmente consegui descobrir o cálculo correto de Z_n. Eram necessárias ***duas*** *variáveis o tempo todo!! Finalmente posso escrever a fórmula correta!! Ela é*

$$Z_{n,k} = \begin{cases} n, & k = 0 \\ 3, & n = 0 \\ 4, & n = 1 \\ 5, & n = 2 \\ Z_{n-1,k} - 2Z_{n-2,k-1} + 3Z_{n-3,k-2}, & n > 2 \end{cases}$$

Com isso, você resolve refazer suas implementações para completar os cálculos que a vovó sempre desejou fazer.

4 Backtracking

1. Escreva um algoritmo baseado em *backtracking* que recebe um inteiro n e produz uma sequencia contendo todos os inteiros de 1 a n com a condição de que a soma de um inteiro com o seguinte na sequencia seja sempre um quadrado perfeito. Por exemplo, ao receber o número $n = 15$ seu algoritmo poderia produzir

```
8 1 15 10 6 3 13 12 4 5 11 14 2 7 9
```

2. Escreva um algoritmo que seja capaz de resolver o problema das 4 rainhas usando *backtracking*. Altere seu algoritmo para que ele seja capaz de resolver o problema para um número qualquer de rainhas.

3. Adapte o algoritmo produzido para o problema 2 para produzir outro algoritmo que resolva o problema do passeio do cavalo no xadrez. Procure uma descrição deste problema em `http://en.wikipedia.org/wiki/Knight%27s_tour` e tente fazer versões que achem caminhos abertos (mais fácil) e caminhos fechados (mais difícil).

4. Escreva um algoritmo que seja capaz de resolver um Sudoku como o que está abaixo, usando *backtracking*. Preveja a entrada e saída de seu algoritmo.

	2		6		8			
5	8				9	7		
				4				
3	7					5		
6								4
		8					1	3
				2				
		9	8				3	6
			3		6		9	

5. Dado um conjunto de inteiros $S = \{a_1, a_2, \ldots, a_n\}$ escreva um algoritmo baseado em *backtracking* capaz de testar se existe um subconjunto de S que tenha a soma igual a um outro inteiro k.

5 Algoritmos genéticos

1. Você tem vários pontos que pertencem a uma função $f(x)$ desconhecida: (x_1, y_1), (x_2, y_2), ..., (x_n, y_n). Já que $f(x)$ é desconhecida, você resolve imitar a função com um polinômio $p(x) = ax^3 + bx^2 + cx + d$, mas precisa achar os valores de a, b, c e d que fazem $p(x)$ passar perto dos pontos dados e para isso a função

$$d(p()) = \sum_i |p(x_i) - y_i|$$

parece uma escolha bem boa, e ela deve ter o valor mais baixo possível para $p(x)$ passar bem perto dos pontos. Agora cumpra as seguintes etapas:

   (a) Escreva um programa que cria uma população de n polinômios, cada um com seus valores de a, b, c e d;

   (b) Implemente a função $d()$ para medir a qualidade de cada polinômio;

   (c) Implemente uma estratégia de evolução para que sejam criados novos polinômios;

   (d) Faça sua população evoluir para polinômios cada vez melhores para imitar os pontos que foram dados;

(e) Se você não estiver feliz com o resultado que obtiver, mude o polinômio para um polinômio de grau maior ou alguma função mais adequada.