UNIVERSIDADE FEDERAL DE SÃO PAULO
CAMPUS SÃO JOSÉ DOS CAMPOS
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA (DCT)

Exercícios de Projeto e Análise de Algoritmos
Lista 2

UC: **Projeto e Análise de Algoritmos**

Aluno: **Thauany Moedano**

RA: **92486**

Professor: **Dr. Reginaldo Massanobu Kuroshu.**

Entrega: **12/10/2015**

Resumo

Resolução da lista 3 de exercícios da aula de Projeto e Análise de Algoritmos no 2º semestre de 2015.

1. Escreva um algoritmo por backtracking que encontre o número mínimo de moedas para retornar n centavos de troco para qualquer conjunto D de diferentes valores de moedas disponíveis que inclua a moeda de 1 centavo.

O problema consiste em permutar moedas até que se chegue no valor da soma desejada. Os candidatos a permutação sempre são quaisquer valores que não estoure a permutação.

```

is_a_solution(int a[], int k, int n, int soma) {
   if(soma == n) //Quando chegarmos a soma, encontramos uma solucao
      return(1);
   else
      return(0);
}
//No process solution verificamos se a solucao nova eh melhor que a anterior
process_solution(int a[], int k, int n, int best[], int soma) {
   if(a[].size < best[].size) { //A melhor solucao eh aquela que possui
       menos moedas
      int i;
      for(i = 0; i < a.size; i++) {
         best[i] = a[i]; //Armazenamos a solucao encontrada em um vetor best
      }
```

```
      best.size = a.size; //Nao podemos esquecer de atualizar o tamanho do vetor best
  }
  //O vetor de valores eh construido no main, informando quais sao os valores disponiveis para se colocar na permutacao
  construct_candidates(int a[], int k, int n, int c[], int *nCandidates, int soma, int D, int valores[]) {
    int i;

    *nCandidates = 0;

    for(i = 1; i <= n; i++)
      if(valores[i]+soma <= n){ //Esta na permutacao qualquer valor que nao estoure a soma
        c[*nCandidates] = valores[i];
        *nCandidates = *nCandidates + 1;
      }
  }
  //best armazena a melhor solucao encontrada. 'valores' armazena o conjunto de valores disponiveis para colocar
  //No inicio o tamanho de best eh vazio pois nao foi computada nenhuma solucao
  backtrack(int a[], int k, int D, int n, int soma, int best[], int valores[]) {
    int c[D]; //O numero maxmo de candidatos eh defnido por D
    int nCadidates;
    int i;


    if(is_a_solution(a, k, n, soma))
      process_solution(a, k, n, best[]);
    else {
      k+=1;
      construct_candidates(a, k, input, c, &nCandidates,soma, D);

      for(i = 0; i < nCandidates; i++) {
        a[k] = c[i];
        a.size += 1;
        soma += c[i];
        backtrack(a, k, D, n, soma, best, valores);
      }
    }
  }

  Imprime(best); //No final a melhor solucao eh impressa
```

2. Considere o problema das 8 rainhas: em um tabuleiro 8x8 de xadrez é possível encontrar diferentes formas de posicionar 8 rainhas de forma que nenhuma rainha consiga atacar outra rainha em apenas 1 movimento. A rainha é uma peça de xadrez que a cada movimento pode se movimentar múltiplas casas e em diferentes direções: vertical, horizontal e em diagonal. Projete um algoritmo por backtracking que encontre uma soluções para o problema das 8 rainhas de forma eficiente.

A ideia utiliza os princípios do Sudoku. Existe uma struct que é o tabuleiro que representa quais lugares que se pode colocar ou não uma rainha (flags). Quando a flag é zero, significa que aquela é uma posição da qual nenhuma rainha está atacando, logo é uma posição candidata a colocar uma rainha.

```

#define DIMENSION 8
#define NCELL DIMENSION*DIMENSION

typedef struct {
   flag[DIMENSION+1][DIMENSION+1]; //Vetor de flags que verifica quais
       posicoes do tabuleiro nao podem ser preenchidas
} board

void constrcut_candidates(board *board, int c[], int *nCandidates) {
   int i, j;
   *nCandidates = 0;
   for(i = 0; i < DIMENSION; i++) {
      for(j = 0; j < DIMENSION; j++) {
         if(board.flags[x][y] == 0) {
            c[i][j] = 1;
            *nCandidates += 1;
         }
      }
   }
}

void backtrack(int A[], int k) {
   int c[DIMENSION+1][DIMENSION+1];
   int nCandidates;
   int i, j;

   if(is_a_solution) //Se K == DIMENSION
      process_solution;

   else {
      k+=1;
      construct_candidates(board, c, &nCandidates);

         for(i = 0; i < nCandidates; i++) {
            for(j = 0; j < nCandidates; j++) {
               A[i][j] = c[i][j]; //O vetor solucao guarda a posicao que a
                  rainha pode ser colocada
               make_move(A, k); //Marca os flags
```

```
            backtrack(A, k);
            unmake_move(A,k) //Desmarca os flags;
```

3. Implemente um algoritmo por divisão e conquista que encontra a mediana de um vetor de inteiros. Utilize a ideia do algoritmo do Quicksort. Qual a complexidade desta solução no caso médio?

A ideia é utilizar a função de partição para achar a mediana. A partição devolve a posição final do vetor pivô. Quando a posição final do pivô for a metade do vetor, significa que este elemento é a mediana. A funçao mediana retorna o índice do vetor que é a mediana, devendo ser processado de maneira diferente no main de acordo com N (tamanho do vetor).

Se N é par, o main deve imprimir(A[q]+A[q+1])/2. Se N é ímpar, o main deve imprimir A[q].

```

int quickSort_particao(int A[], int p, int r) {
  int x, aux;
  int i, j;
  x = A[r];
  i = p-1;
    for(j = p; j <= r-1; j++)
        if (A[j] <= x) {
            i = i + 1;
            aux = A[i];
            A[i] = A[j];
            A[j] = aux;
        }
  aux = A[i+1];
  A[i+1] = A[r];
  A[r] = aux;
  return (i+1);
}

int mediana(int A[], int p, int r, int n) {
    int q;
    if(p<r) {
        q = quickSort_particao(A, p, r);
        if(q > n/2)
            mediana(A, p, q-1, n);
        else if(q < n/2)
            mediana(A, q+1, r, n);
        else
            return(q);
    }

}
```

Independentemente do caso, quickSort-particao é executada pelo menos uma vez para n elementos (Quando os ranges são zero e n). Mediana faz diversas chamadas recursivas quebrando o problema em dois. Em um caso médio, a mediana chamaria metade dessas recursões, mas isso não excluiria o algoritmo de fazer $O(\log n)$ chamadas recursivas.

Portanto, em um caso médio o algoritmo é de ordem $O(n \log n)$

4. a.) Considere as seguintes funções para max-heap. Argumente sobre a corretude do BUILD-MAX-HEAP() utilizando o seguinte invariante de laço: "No início de cada iteração do laço for, cada nó i+1, i+2.... n é uma raiz de um max-heap". Devemos provar três

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

Figura 1

pontos: **Inicialização, manuntenção e terminação**.

Inicialização: Antes de se inicializar as operações, i recebe $\frac{n}{2}$. Portanto os nós i+1,i+2.... n são todos nós folhas e podem ser considerados raízes de um max-heap.

Manutenção: Durante a execução, Max-Heapfy rearranja o heap considerando que apenas o nó da posição i esteja quebrando as regras do heap. Portanto, o elemento i é trocado com seu filho à esquerda ou a direita, sem alterar as condições iniciais dos nós (i+1....n). Logo, o invariante vale durante a execução.

Terminação: Após a terminação, o nó na posição i é rearranjado de forma que estabeleça as regras de um heap e I é decrementado. Portanto, novamente, os elementos nas posições (i+1....n) atendem as regras de heap e podem ser raízes de um max-heap.

b.) Escreva um algoritmo de complexidade de tempo O(n) que verifica se um vetor A[1...n] é ou não um max-heap. Mostre que sua complexidade de tempo é O(n).

```c
   int verifica(int A[], int n) {
    int i;
    for(i = 0; i < n/2; i++) {
        if(A[i] < A[(2*i)+1] || A[i] < A[(2*i)+2]) //Verifica a condicao de heap
            return(0);
    }
    return(1);

}
```

O laço for roda (n/2)+1 e o if dentro do laço roda (n/2) vezes. Somando as iterações no pior no caso, temos cerca de n operações. Portanto, o algoritmo é de $O(n)$.