SSC0503 - Introdução à Ciência de Computação II
Respostas da 4ª Lista
Professor: Claudio Fabiano Motta Toledo (`claudio@icmc.usp.br`)
Estagiário PAE: Jesimar da Silva Arantes (`jesimar.arantes@usp.br`)

Resposta 1: Desenvolva manualmente o algoritmo insertion sort sobre o arranjo $A = [13, 19, 9, 5, 12, 8, 7, 4]$.
Passo 1: $A =$ [\| 13 19 9 5 12 8 7 4]
Passo 2: $A =$ [13 \| 19 9 5 12 8 7 4]
Passo 3: $A =$ [13 19 \| 9 5 12 8 7 4]
Passo 4: $A =$ [9 13 19 \| 5 12 8 7 4]
Passo 5: $A =$ [5 9 13 19 \| 12 8 7 4]
Passo 6: $A =$ [5 9 12 13 19 \| 8 7 4]
Passo 7: $A =$ [5 8 9 12 13 19 \| 7 4]
Passo 8: $A =$ [5 7 8 9 12 13 19 \| 4]
Passo 9: $A =$ [4 5 7 8 9 12 13 19 \|]

Resposta 3: Desenvolva manualmente o algoritmo selection sort sobre o arranjo $A = [13, 19, 9, 5, 12, 8, 7, 4]$.
Passo 1: $A =$ [13 19 9 5 12 8 7 4 \|]
Passo 2: $A =$ [13 4 9 5 12 8 7 \| 19]
Passo 3: $A =$ [7 4 9 5 12 8 \| 13 19]
Passo 4: $A =$ [7 4 9 5 8 \| 12 13 19]
Passo 5: $A =$ [7 4 8 5 \| 9 12 13 19]
Passo 6: $A =$ [7 4 5 \| 8 9 12 13 19]
Passo 7: $A =$ [5 4 \| 7 8 9 12 13 19]
Passo 8: $A =$ [4 \| 5 7 8 9 12 13 19]
Passo 9: $A =$ [\| 4 5 7 8 9 12 13 19]

Resposta 6: Use o método de substituição para provar que a recorrência $T(n) = T(n-1) + \Theta(n)$ tem a solução $T(n) = \Theta(n^2)$.
$T(n) = T(n-1) + \Theta(n)$
$T(n) = \Theta(n^2)$
$T(n) \leq cn^2$
$T(n) \leq c(n-1)^2 + \Theta(n)$
Fazendo $\Theta(n) = c_1 n$
$T(n) \leq c(n^2 - 2n + 1) + c_1 n$
$T(n) \leq cn^2 - 2cn + c + c_1 n$
$T(n) \leq cn^2 - (2cn - c - c_1 n)$
Fazendo $2cn - c - c_1 n \geq 0$ (resíduo)
$c \geq \frac{c_1 n}{2n-1}$
Fazendo n tender a infinito temos:
$c \geq \frac{c_1 1}{2}$
Assumindo que c_1 assume o valor $c_1 = 1$ temos

Dessa forma qualquer $c \geq \frac{1}{2}$ resolve o problema.

Resposta 7: Use o método mestre para provar que a recorrência $T(n) = 2T(n/2) + \Theta(n)$ tem a solução $T(n) = \Theta(n \cdot lg\ n)$.
$T(n) = 2T(n/2) + \Theta(n)$
$T(n) = 2T(n/2) + cn$
Resolvendo pelo método mestre temos:
$a = 2$
$b = 2$
$f(n) = cn$
Caso 1: $f(n) = O(n^{log_b a - \varepsilon})$
$cn = O(n^{log_2 2 - \varepsilon})$
$cn = O(n^{1-\varepsilon})$
Falso
Caso 2: $f(n) = \Theta(n^{log_b a})$
$cn = \Theta(n^{log_2 2})$
$cn = \Theta(n^1)$
Verdade, então:
$T(n) = \Theta(n^{log_b a} \cdot lg\ n)$
$T(n) = \Theta(n \cdot lg\ n)$

Resposta 9: Desenvolva um programa em C que faça a ordenação através do método quicksort sobre um vetor de tamanho N. Em seguida, diga qual a análise de complexidade no melhor caso, pior caso e caso médio.

```c
void quicksort(int a[], int p, int r){
 if (p < r){
  int v = (rand()%(r-p)) + p;
  int pivo = a[v];
  a[v] = a[r];
  a[r] = pivo;
  int i = p - 1;
  int j = r;
  do{
   do{
    i++;
   }while(a[i] < pivo);
   do{
    j--;
   }while((a[j] > pivo) && (j > p));
   if (i < j){
    int t = a[i];
    a[i] = a[j];
    a[j] = t;
   }
  }while(i < j);
  a[r] = a[i];
  a[i] = pivo;
  quicksort(a, p, i-1);
  quicksort(a, i+1, r);
```

```
  }
}
```

Listing 1: Resposta do exercício 9 codificado na linguagem C

Análise de Pior Caso: $O(n^2)$
Análise de Melhor Caso: $O(n\ lg\ n)$
Análise de Caso Médio: $O(n\ lg\ n)$

Resposta 10: Desenvolva um programa em C que faça a ordenação (em ordem decrescente) através do método quicksort sobre um vetor de tamanho N. Em seguida, diga qual a análise de complexidade no melhor caso, pior caso e caso médio.

```
void quicksort(int a[], int p, int r){
 if (p < r){
  int v = (rand()%(r-p)) + p;
  int pivo = a[v];
  a[v] = a[r];
  a[r] = pivo;
  int i = p - 1;
  int j = r;
  do{
   do{
    i++;
   }while(a[i] > pivo);
   do{
    j--;
   }while((a[j] < pivo) && (j > p));
   if (i < j){
    int t = a[i];
    a[i] = a[j];
    a[j] = t;
   }
  }while(i < j);
  a[r] = a[i];
  a[i] = pivo;
  quicksort(a, p, i-1);
  quicksort(a, i+1, r);
 }
}
```

Listing 2: Resposta do exercício 10 codificado na linguagem C

Análise de Pior Caso: $O(n^2)$
Análise de Melhor Caso: $O(n\ lg\ n)$
Análise de Caso Médio: $O(n\ lg\ n)$

Resposta 13: Mostre que o quicksort no melhor caso executa em $\Omega(n \cdot lg\ n)$.
O algoritmo quicksort é um algoritmo de divisão e conquista.
Ele subdivide o problema em subproblemas cada vez menores e os resolve.
No cenário de melhor caso o quicksort sempre consegue quebrar o problema pela metade.
Dessa forma, o quicksort gasta $\Theta(n)$ operações para subdividir o problema de tamanho n

em dois problemas de tamanho $n/2$.
Logo, a sua definição de complexidade de tempo é da forma: $T(n) = 2T(n/2) + \Theta(n)$
Foi visto no exercício 7 que tal recursão tem a solução $T(n) = \Theta(n \cdot lg\ n)$.
Lembrando da propriedade que para duas funções quaisquer $f(n)$ e $g(n)$, temos $f(n) = \Theta(g(n))$ se e somente se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.
Portanto, se $T(n) = \Theta(n \cdot lg\ n)$ então também é da ordem $T(n) = \Omega(n \cdot lg\ n)$.

Resposta 14: No algoritmo Randomized-Quicksort, quantas chamadas são feitas ao gerador de números aleatórios Random no pior caso? e no melhor caso?. Dê sua resposta em termos da notação Θ.

Na análise de pior caso a complexidade do quicksort é de $T(n) = \Theta(n^2)$.
O elemento Random é usado na separação das listas L1 e L2 que possui custo computacional $T(n) = O(n)$. Para ser mais exato necessitamos passar um vez pelos n dados para gerar a Lista L1 e passar uma vez pelos n dados para gerar a lista L2.

O fator restante de complexidade n do quicksort é dado pela altura da árvore.

Na análise de melhor caso a complexidade do quicksort é de $T(n) = \Theta(n\ lg\ n)$.
O elemento Random é usado na separação das listas L1 e L2 que possui custo computacional $T(n) = O(n)$, da mesma forma que no pior caso.

O fator restante de complexidade $lg\ n$ do quicksort é dado pela altura da árvore. Repare que nesse caso apesar da mesma quantidade de chamadas a função Random a árvore foi subdividida.

Resposta 15: Desenvolva um programa em C que faça uma implementação combinada do quicksort com o insertion sort. O quicksort será executado até que o partição fique menor que um determinado valor k (por exemplo, $k = 10$, $k = 20$) então faça uma chamada para o método insertion sort. Avalie e compare o desempenho isolado do quicksort e insertion sort.

```
1  void quicksortIS(int a[], int p, int r){
   if (p < r){
3   if (r - p < 20){
     insertionsort(a, p, r);
5    return;
    }
7   int v = (r - p)/2 + p;
    int pivo = a[v];
9   a[v] = a[r];
    a[r] = pivo;
11  int i = p - 1;
    int j = r;
13  do{
     do{
15    i++;
     }while(a[i] < pivo);
17   do{
      j--;
19   }while((a[j] > pivo) && (j > p));
     if (i < j){
21    int t = a[i];
```

```
    a[i] = a[j];
    a[j] = t;
   }
  }while(i < j);
  a[r] = a[i];
  a[i] = pivo;
  quicksortIS(a, p, i-1);
  quicksortIS(a, i+1, r);
 }
}

void insertionsort(int a[], int p, int r){
 for (int j = p + 1; j <= r; j++){
  int chave = a[j];
  int i = j - 1;
  while (i >= 0 && a[i] > chave){
   a[i+1] = a[i];
   i = i-1;
  }
  a[i+1] = chave;
 }
}
```

Listing 3: Resposta do exercício 15 codificado na linguagem C

Resposta 17: Quais são os números mínimo e máximo de elementos numa heap de altura h?

A expressão 2^h revela o número mínimo de elementos em um heap de altura h. Dessa forma, seja $h = 4$ então, número mínimo é igual a $2^4 = 16$.

A expressão $2^{h+1} - 1$ revela o número máximo de elementos em um heap de altura h. Dessa forma, seja $h = 4$ então, número mínimo é igual a $2^{4+1} - 1 = 32 - 1 = 31$.

Resposta 18: Mostre que uma heap com n elementos tem altura $\lfloor lg\ n \rfloor$.

Como uma heap trata-se de uma árvore binária completa ou quase-completa. E em uma árvore binária a quantidade de filhos de um nível para o seguinte dobra (exceto para o último nível). Dessa forma, a quantidade de filhos crescem em uma progressão exponencial de base 2 com a quantidade de nível (veja exercício anterior). Dessa forma, a altura de uma árvore é dada pela operação inversa da exponencial, ou seja, o logaritmo (de base 2). O operador chão $\lfloor \rfloor$ se justifica uma vez que a altura da árvore não pode ser real, mas sim um número inteiro.

Resposta 20: Escreva um algoritmo para a rotina Min-Heapify(A,i). Compare o tempo de execução da Min-Heapify com Max-Heapify.

```
void min_heapify(int A[], int i, int tamHeapA){
 int l = left(i);
 int r = right(i);
 int menor = i;
 if (l < tamHeapA && A[l] < A[menor]){
  menor = l;
```

```
  }
  if (r < tamHeapA && A[r] < A[menor]){
   menor = r;
  }
  if (menor != i){
   swap(A, i, menor);
   min_heapify(A, menor, tamHeapA);
  }
}
```

Listing 4: Resposta do exercício 20 codificado na linguagem C

Resposta 21: Qual o efeito de chamar Max-Heapify(A,i), quando o elemento A[i] é maior que seus filhos? e quando i>A.heap-size/2?

Quando o elemento A[i] for maior que seus filhos o procedimento Max-Heapify executará em $O(1)$. Isto ocorre porque quando A[i] é maior que os seus filhos a heap já corresponde a uma heap máxima. O mesmo ocorrerá quando i>A.heap-size/2, ou seja, Max-Heapify executará em $O(1)$. Isso ocorre porque o elemento i não terá filhos e dessa forma já corresponde também a uma heap máxima.

Resposta 26: Escreva algoritmos para Heap-Minimum, Heap-Extract-Min, Heap-Decrease-Key e Min-Heap-Insert que implemente uma min-priority queue com uma min-heap.

```
int heap_minimum(int A[]){
 return A[0];
}

int heap_extract_min(int A[], int tamHeapA){
 if (tamHeapA < 0){
  printf("heap underflow\n");
  return -1;
 }
 int min = A[0];
 A[0] = A[tamHeapA-1];
 tamHeapA = tamHeapA - 1;
 min_heapify(A, 0, tamHeapA);
 return min;
}

void heap_decrease_key(int A[], int i, int chave){
 if (chave > A[i]){
  printf("nova chave maior que chave atual\n");
  return;
 }
 A[i] = chave;
 while (i > 0 && A[parent(i)] > A[i]){
  swap(A, i, parent(i));
  i = parent(i);
 }
}

void min_heap_insert(int A[], int chave, int tamHeapA){
 tamHeapA = tamHeapA + 1;
```

```
31  A[tamHeapA] = 1000000;
    heap_decrease_key(A, tamHeapA, chave);
33  }
```

Listing 5: Resposta do exercício 26 codificado na linguagem C