Aluno(a):____Matrícula:____________

1ª)	1.0	
2ª)	2.0	
3ª)	2.0	
4ª)	2.0	
5ª)	3.0	
	10.0	

a) A prova é individual e sem consulta. Qualquer tentativa de "cola" resultará na anulação da prova do aluno ou de ambos os alunos envolvidos.
b) A interpretação faz parte da questão. Não há perguntas durante a prova. Em caso de dúvida escreva a dúvida e a sua interpretação na resposta.
c) O tempo de prova é 1:45 h.
d) As respostas devem seguir as questões. Caso precise de rascunho use o verso da folha.
e) A prova pode ser feita a lápis.

(1) (1.0 ponto) Os algoritmos abaixo são usados para resolver problemas de tamanho *n*. Determine a complexidade, no pior caso, de cada algoritmo. Explique sua resposta.

a) (0.5 pontos)

```
for ( i=1; i < n; i *= 2 ) {
     for ( j = n; j > 0; j -= 2 ) {
          for ( k = j; k < n; k += 2 ) {
               sum += (-j * k) << i/2;
          }
     }
}
for ( i=1; i < n; i += 1 ) {
     prod *= i;
}
```

b) (0.5 pontos)

```
for ( i=1; i < n; i *= 2 ) {
     for ( j = n; j > 0; j /= 3 ) {
          for ( k = j; k < n; k *= 5 ) {
               sum += (-j * k) << i/2;
          }
     }
}
```

Resposta:

a) *Primeiro ninho de "for":* O laço mais externo é executado $log_2(n)$ vezes, visto que *i* dobra a cada passagem. O laço do meio é executado $n/2$ vezes, já que *j* é decrementado de 2 em 2. O último laço também é executado no máximo $n/2$ vezes, para cada interação do laço do meio. Como os laços estão aninhados, o comando

```
sum += (-j * k) << i/2
```

é executado $log_2(n)*(n/2)^2 = (n^2 * log_2(n))/4$ vezes.

Segundo ninho de "for": O laço é executado *n* vezes, visto que *i* é incrementado de 1 em 1. Logo o comando

```
prod *= i
```

é executado *n* vezes.

Portanto, a complexidade do algoritmo é $O(n^2 \log(n))$.

b) O laço mais externo é executado $log_2(n)$ vezes, visto que *i* dobra a cada passagem. O laço do meio é executado $log_3(n)$ vezes, já que *j* é dividido por 3 a cada passagem. O último laço é executado no máximo $log_5(n)$ vezes, para cada interação do laço do meio. Como os laços estão aninhados, o comando

```
sum += (-j * k) << i/2
```

é executado $log_5(n)*log_3(n)*log_2(n)$ vezes. Logo, a complexidade do algoritmo é $O((\log n)^3)$.

(2) (2.0 pontos) Uma tabela de dispersão de tamanho 11 é implementada com encadeamento interior, utilizando todo o espaço de endereçamento para tratar colições (ou seja sem área de overflow). A função de dispersão é a seguinte:

$$h(x) = (x)\%11$$

a) (1.0 ponto) Desenhe a estrutura de dados após a inserção das chaves (nesta ordem):

7,10,15,14,17,3,21,25

Explique como cada colisão foi tratada.

b) (1.0 ponto) Explique, com base no exemplo anterior, como executar sucessivamente as seguintes operações:
  i) (0.5 pontos) Remoção da chave 3.
  ii) (0.5 pontos) Inserção da chave 4.

Resposta:

a)

x	*h(x)*
7	7
10	10
15	4
14	3
17	6
3	3
21	10
25	3

0	21	*7*	Houve colisão na posição 10; a chave 21 é inserida na primeira posição vazia
1			
2			
3	14	*4*	
4	15	*3*	
5	3	*6*	Houve colisão na posição 3; a chave 3 é inserida na primeira posição vazia
6	17	*5*	
7	7	*1*	
8	25	*8*	Houve colisão na posição 3; a chave 25 é inserida na primeira posição vazia
9			
10	10	*2*	

b) As posições nunca ocupadas são marcadas com BRANCO e as previamente ocupadas por chaves removidas são marcadas com -1.

Remoção de 3: como 3 não está na posição 3, pesquise por 3 (com "loop back" para 0 quando chegar a 10) até achar uma posição que nunca foi ocupada ou até achar 3. Remova 3 então da posição 5, marcando-a com -1.

Inserção de 4: como a posição 4 está ocupada, procure a primeira posição seguinte (com "loop back" para 0 quando chegar a 10) marcada com BRANCO ou com -1; neste caso será a posição 5. Insira 4 nesta posição.

(3) (2.0 pontos) Considere a seguinte sequencia de inteiros: 90, 60, 30, 15, 45.

a) (0.5 ponto) Mostre, passo a passo, como o vetor armazenando um *heap mínimo* é construído pela inserção sucessiva destes 5 elementos, na ordem dada. Comente brevemente cada passo do algoritmo de inserção.

b) (1.0 ponto) Mostre, passo a passo, como o vetor armazenando um *heap mínimo* é construído, para estes mesmos 5 elementos, mas usando o algoritmo eficiente para construção de heaps. Comente brevemente cada passo do algoritmo.

c) (0.5 ponto) Os heaps resultantes do item (a) e do item (b) precisam ser iguais? Explique sua resposta.

Resposta:

a)

0	1	2	3	4	Explicação
90					Acrescente 90 depois do final do heap (que está vazio).
90	60				Acrescente 60 depois do final do heap.
60	90				Compare com o pai ((1-1)/2=0) e troque.
60	90	30			Acrescente 30 depois do final do heap.
30	90	60			Compare com o pai ((2-1)/2=0) e troque.
30	90	60	15		Acrescente 15 depois do final do heap.
30	15	60	90		Compare com o pai ((3-1)/2=1) e troque.
15	30	60	90		Compare com o pai ((1-1)/2=0) e troque.
15	30	60	90	45	Acrescente 45 depois do final do heap.
15	30	60	90	45	Compare com o pai ((4-1)/2=1) e não troque. Pare.

b)

0	1	2	3	4	Explicação (n=5)
		90	60	30	Aloque os n/2+1 primeiros elementos que ocorrem no conjunto
	15	90	60	30	Coloque o próximo elemento, 15, na posição n/2-1
	15	90	60	30	Compare 15 com os seus filhos, nas posições 2i+1 e 2i+2 (com i=1). Não é preciso trocar pois 15<60 e 15<40
45	15	90	60	30	Coloque o próximo elemento, 45, na posição 0
15	45	90	60	30	Compare 45 com os seus filhos, nas posições 2i+1 e 2i+2 (com i=0). É preciso trocar 45 com 15 pois 15<45 e 15<90
15	30	90	60	45	Compare 45 com os seus filhos, nas posições 2i+1 e 2i+2 (com i=1). É preciso trocar 45 com 30 pois 30<45 e 30<60

c) Os heaps não precisam ser iguais. Por definição de min heap, basta que a prioridade do pai seja menor do que a dos filhos, mas a prioridade do filho à esquerda pode ser maior ou menor do que a do filho à direita.

(4) (2.0 pontos) Considere a representação de partições por florestas, com as otimizações da operação de `union` através do balanceamento por altura e da operação de `find` por compressão de caminhos. Caso haja empate na operação de `union`, escolha o menor inteiro para raíz.

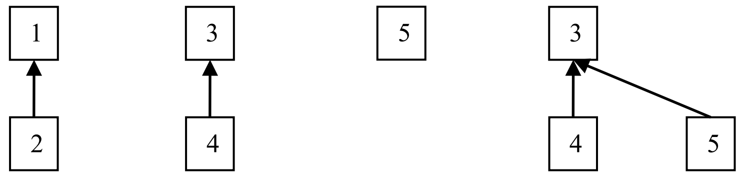
a) (1.0 ponto) Começando com uma partição de {1, 2, 3, 4, 5} em *singletons*, represente a sequência de florestas obtidas sucessivamente pelas operações:

`union(1,2), union(3,4), union(3,5), union(1,3)`

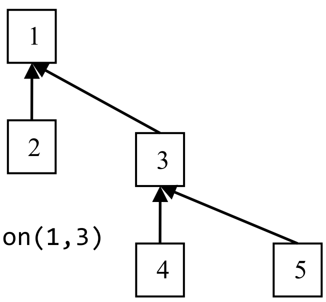
b) (1.0 ponto) Considere a operação `find(4)`. Quantos passos serão executados após realizar `find(4)` por 4 vezes? Explique a sua resposta.

Resposta:

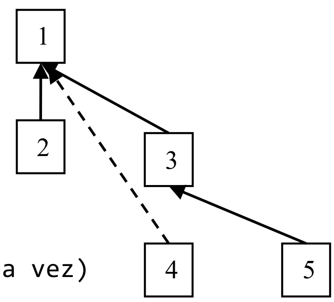
(a)



`union(1,2)` `union(3,4)` `union(3,5)`



`union(1,3)`



`find(4)`
`(primeira vez)`

(b) `find(4)` – 1a vez: 2 passos
`find(4)` – 2a, 3a e 4a vezes: 1 passo
Total de comparações: 5 passos

(5) Considere a implementação de conjuntos como vetores de bits.

a) (1.5 pontos) Implemente em C uma função que computa o complemento de um conjunto. A função recebe como entrada um conjunto *a* e retorna o complemento de *a*. A interface da função é a seguinte:

```
BitVector* bvCompl(BitVector* a);
```

A sua implementação não deverá usar as operações do TAD de conjuntos visto em sala.

b) (1.5 pontos) Implemente em C uma função que computa a cardinalidade de um conjunto. A função recebe como entrada um conjunto *a* e retorna a cardinalidade de *a*. A interface da função é a seguinte:

```
int bvCard(BitVector* a);
```

A sua implementação não deverá usar as operações do TAD de conjuntos visto em sala.

Resposta:

(a)

```
BitVector* bvCompl(BitVector* a)
{
   int i;
   int num   = (a->max-1)/sizeof(int)+1; /* c->vetor tem o tamanho de a->vetor */

   BitVector* t = (BitVector*)malloc(sizeof(BitVector));
   c->max    = a->max;
   c->vector = (int*)malloc(num*sizeof(int));

   for (i=0; i < num; i++)
      c->vector[i] = ~(a->vector[i]);        /* computa o complemento de a     */

   return c;
 }
```

(b)

```
int bvCard(BitVector* a)
{
   int i, j, bitmap, card = 0;
   int size= sizeof(int);
   int mask=1;                              /* máscara com '00...01'            */
   for (i=0; i < a->max; i++) {
      bitmap = a->vector[i];
      for (j=0; j < size; j++) {
            card = card + (bitmap & mask); /* incr. card se o último bit for 1 */
            bitmap = bitmap / 2;           /* remove o último bit              */
      }
   }
   return card;
 }
```