Algorithms and Data Structures
Exercises

Antonio Carzaniga
University of Lugano

Edition 1.2
January 2009

1. Answer the following questions on the big-oh notation.

   (a) Explain what $g(n) = O(f(n))$ means. *time: 5'*

   (b) Explain why the statement: "The running time of algorithm A is at least $O(n^2)$" is meaningless. *time: 5'*

   (c) Given two functions $f = \Omega(\log n)$ and $g = O(n)$, consider the following statements. For each statement, write whether it is true or false. For each false statement, write two functions f and g that show a counter-example. *time: 5'*

   - $g(n) = O(f(n))$
   - $f(n) = O(g(n))$
   - $f(n) = \Omega(\log(g(n)))$
   - $f(n) = \Theta(\log(g(n)))$
   - $f(n) + g(n) = \Omega(\log n)$

   (d) For each one of the following statements, write two functions f and g that satisfy the given condition. *time: 5'*

   - $f(n) = O(g^2(n))$
   - $f(n) = \omega(g(n))$
   - $f(n) = \omega(\log(g(n)))$
   - $f(n) = \Omega(f(n)g(n))$
   - $f(n) = \Theta(g(n)) + \Omega(g^2(n)))$

2. Write the pseudo-code of a function called *findLargest* that finds the largest number in an array using a divide-and-conquer strategy. You may use a syntax similar to Java. Also, write the time complexity of your algorithm in terms of big-oh notation. Briefly justify your complexity analysis. *time: 20'*

   ```
   int findLargest(int [] A) {
   ```

3. Illustrate the execution of the *merge-sort* algorithm on the array

   $$A = \langle 3, 13, 89, 34, 21, 44, 99, 56, 9 \rangle$$

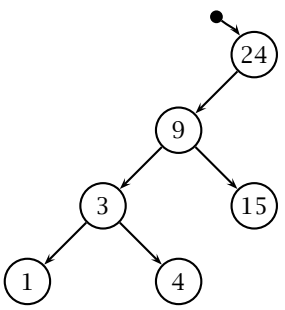
   For each fundamental iteration or recursion of the algorithm, write the content of the array. Assume the algorithm performs an in-place sort. *time: 20'*

4. Consider the array $A = \langle 29, 18, 10, 15, 20, 9, 5, 13, 2, 4, 15 \rangle$.

   (a) Does A satisfy the *max-heap* property? If not, fix it by swapping two elements. *time: 5'*

   (b) Using array A (possibly corrected), illustrate the execution of the *heap-extract-max* algorithm, which extracts the max element and then rearranges the array to satisfy the *max-heap* property. For each iteration or recursion of the algorithm, write the content of the array A. *time: 15'*

5. Consider the following binary search tree (BST).



(a) List all the possible insertion orders (i.e., permutations) of the keys that could have produced this BST. *time: 5'*

(b) Draw the same BST after the insertion of keys: 6, 45, 32, 98, 55, and 69, in this order. *time: 5'*

(c) Draw the BST resulting from the deletion of keys 9 and 45 from the BST resulting from question 5b. *time: 5'*

(d) Write at least three insertion orders (permutations) of the keys remaining in the BST after question 5c that would produce a balanced tree (i.e., a minimum-height tree). *time: 5'*

6. Implement a function that returns the successor of a node in a binary search tree (the BST stores integer keys). A successor of a node n is defined as the smallest key x in the BST such that x is bigger than the value of n, or *null* if that does not exist. You may assume that the BST does not contain duplicate keys. The signature of the function you have to implement and the interface of the *TreeNode* class, which implements the BST, are given below. Note that *getLeft()*, *getRight()*, and *getParent()* return *null* if the node does not have a left, a right child, or is the *root*, respectively. *time: 10'*

```
interface TreeNode {
    int getValue();
    TreeNode getLeft();
    TreeNode getRight();
    TreeNode getParent();
}

/* Returns -1 if no successor exists */
int successor(TreeNode x) {
```

7. Consider a hash table that stores integer keys. The keys are 32-bit unsigned values, and are always a power of 2. Give the minimum table size t and the hash function $h(x)$ that takes a key x and produces a number between 1 and t, such that no collision occurs. *time: 10'*

8. Explain why the time complexity of searching for elements in a hash table, where conflicts are resolved by chaining, decreases as its load factor α decreases. Recall that α is defined as the ratio between the total number of elements stored in the hash table and the number of slots in the table.

9. For each statement below, write whether it is true or false. For each false statement, write a counter-example. *time: 10'*

   - $f(n) = \Theta(n) \wedge g(n) = \Omega(n) \Rightarrow f(n)g(n) = \Omega(n^2)$
   - $f(n) = \Theta(1) \Rightarrow n^{f(n)} = O(n)$
   - $f(n) = \Omega(n) \wedge g(n) = O(n^2) \Rightarrow g(n)/f(n) = O(n)$
   - $f(n) = O(n^2) \wedge g(n) = O(n) \Rightarrow f(g(n)) = O(n^3)$
   - $f(n) = O(\log n) \Rightarrow 2^{f(n)} = O(n)$
   - $f = \Omega(\log n) \Rightarrow 2^{f(n)} = \Omega(n)$

10. Write tight asymptotic bounds for each one of the following definitions of $f(n)$. *time: 10'*

    - $g(n) = \Omega(n) \wedge f(n) = g(n)^2 + n^3 \Rightarrow f(n) =$
    - $g(n) = O(n^2) \wedge f(n) = n \log(g(n)) \Rightarrow f(n) =$
    - $g(n) = \Omega(\sqrt{n}) \wedge f(n) = g(n + 2^{16}) \Rightarrow f(n) =$
    - $g(n) = \Theta(n) \wedge f(n) = 1 + 1/\sqrt{g(n)} \Rightarrow f(n) =$
    - $g(n) = O(n) \wedge f(n) = 1 + 1/\sqrt{g(n)} \Rightarrow f(n) =$
    - $g(n) = O(n) \wedge f(n) = g(g(n)) \Rightarrow f(n) =$

11. Write the ternary-search trie (TST) that represents a dictionary of the strings: “gnu” “emacs” “gpg” “else” “gnome” “go” “eps2eps” “expr” “exec” “google” “elif” “email” “exit” “epstopdf” *time: 10'*

12. Answer the following questions.

    (a) A hash table with chaining is implemented through a table of K slots. What is the expected number of steps for a search operation over a set of $N = K/2$ keys? Briefly justify your answers.

    (b) What are the worst-case, average-case, and best-case complexities of *insertion-sort*, *bubble-sort*, *merge-sort*, and *quicksort*?

    time: 5'

13. Write the pseudo code of the in-place *insertion-sort* algorithm, and illustrate its execution on the array

$$A = \langle 7, 17, 89, 74, 21, 7, 43, 9, 26, 10 \rangle$$

    Do that by writing the content of the array at each main (outer) iteration of the algorithm. *time: 20'*

14. Consider a binary tree containing N integer keys whose values are all less than K, and the following FIND-PRIME algorithm that operates on this tree.

```
FIND-PRIME(T)
1  x ← TREE-MIN(T)
2  while x ≠ NIL
3        do x ← TREE-SUCCESSOR(x)
4           if IS-PRIME(key(x))
5              then return x
6  return x
```

```
IS-PRIME(n)
1  i ← 2
2  while i · i ≤ n
3        do if i divides n
4              then return FALSE
5           i ← i + 1
6  return TRUE
```

    In case you don't remember, these are the relevant binary-tree algorithms

```
TREE-SUCCESSOR(x)
1  if right(x) ≠ NIL
2  return TREE-MINIMUM(right(x))
3  y ← parent(x)
4  while y ≠ NIL ∧ x = right(y)
5        do x ← y
6           y ← parent(y)
7  return y
```

```
TREE-MINIMUM(x)
1  while left(x) ≠ NIL
2        do x ← left(x)
3  return x
```

    Write the time complexity of FIND-PRIME. Justify your answer. *time: 10'*

15. Consider the following *max-heap*

$$H = \langle 37, 12, 30, 10, 3, 9, 20, 3, 7, 1, 1, 7, 5 \rangle$$

    Write the exact output of the following EXTRACT-ALL algorithm run on H

```
EXTRACT-ALL(H)
1  while heap-size(H) > 0
2        do HEAP-EXTRACT-MAX(H)
3           for i ← 1 to heap-size(H)
4                 do output H[i]
5           output "." END-OF-LINE
```

```
HEAP-EXTRACT-MAX(H)
1  if heap-size(H) > 0
2     then k ← H[1]
3          H[1] ← H[heap-size(H)]
4          heap-size(H) ← heap-size(H) − 1
5          MAX-HEAPIFY(H)
6  return k
```

time: 20'

16. Develop an efficient in-place algorithm called PARTITION-EVEN-ODD(A) that partitions an array A in *even* and *odd* numbers. The algorithm must terminate with A containing all its *even* elements preceding all its *odd* elements. For example, for input $A = \langle 7, 17, 74, 21, 7, 9, 26, 10\rangle$, the result might be $A = \langle 74, 10, 26, 17, 7, 21, 9, 7\rangle$. PARTITION-EVEN-ODD must be an *in-place* algorithm, which means that it may use only a constant memory space in addition to A. In practice, this means that you may not use another temporary array.

    (a) Write the pseudo-code for PARTITION-EVEN-ODD. *time: 20'*

    (b) Characterize the complexity of PARTITION-EVEN-ODD. Briefly justify your answer. *time: 10'*

    (c) Formalize the correctness of the partition problem as stated above, and prove that PARTITION-EVEN-ODD is correct using a loop-invariant. *time: 20'*

    (d) If the complexity of your algorithm is not already linear in the size of the array, write a new algorithm PARTITION-EVEN-ODD-OPTIMAL with complexity $O(N)$ (with $N = |A|$). *time: 20'*

17. The binary string below is the title of a song encoded using Huffman codes.

    00110001011111011001110111011000001001110100101 01

    Given the letter frequencies listed in the table below, build the Huffman codes and use them to decode the title. In cases where there are multiple "greedy" choices, the codes are assembled by combining the first letters (or groups of letters) from left to right, in the order given in the table. Also, the codes are assigned by labeling the left and right branches of the prefix/code tree with '0' and '1', respectively.

letter	a	h	v	w	' '	e	t	l	o
frequency	1	1	1	1	2	2	2	3	3

    time: 20'

18. Consider the *text* and *pattern* strings:

    text: `momify my mom please`
    pattern: `mom`

    Use the Boyer-Moore string-matching algorithm to search for the pattern in the text. For each character comparison performed by the algorithm, write the current *shift* and highlight the character position considered in the pattern string. Assume that indexes start from 0. The following table shows the first comparison as an example. Fill the rest of the table. *time: 10'*

n.	*shift*	m	o	m	i	f	y		m	y		m	o	m		p	l	e	a	s	e
1	0	m	o	m																	
2																					
...	...										...										

19. You wish to create a database of stars. For each star, the database will store several megabytes of data. Considering that your database will store billions of stars, choose the data structure that will provide the best performance. With this data structure you should be able to find, insert, and delete stars. Justify your choice. *time: 10'*

20. You are given a set of persons P and their friendship relation R. That is, $(a, b) \in R$ iff a is a friend of b. You must find a way to introduce person x to person y through a chain of friends. Model this problem with a graph and describe a strategy to solve the problem. *time: 10'*

21. Answer the following questions

    (a) Explain what $f(n) = \Omega(g(n))$ means. *time: 5'*

    (b) Explain what kind of problems are in the P complexity class. *time: 5'*

    (c) Explain what kind of problems are in the NP complexity class. *time: 5'*

(d) Explain what it means for problem A to be *polynomially-reducible* to problem B. *time: 5'*

(e) Write *true*, *false*, or *unknown* depending on whether the assertions below are true, false, or we do not know. *time: 5'*

- $P \subseteq NP$
- $NP \subseteq P$
- $n! = O(n^{100})$
- $\sqrt{n} = \Omega(\log n)$
- $3n^2 + \frac{1}{n} + 4 = \Theta(n^2)$

(f) Consider the *exact change problem* characterized as follows. *Input:* a multiset of values $V = \{v_1, v_2, \ldots, v_n\}$ representing coins and bills in a cash register; a value X; *Output:* 1 if there exists a subset of V whose total value is equal to X, or 0 otherwise. Is the exact-change problem in NP? Justify your answer. *time: 5'*

22. A thief robbing a gourmet store finds n pieces of precious cheeses. For each piece i, v_i designates its value and w_i designates its weight. Considering that W is the maximum weight the robber can carry, and considering that the robber may take any fraction of each piece, you must find the quantity of each piece the robber must take to maximize the value of the robbery. *time: 20'*

    (a) Devise an algorithm that solves the problem using a *greedy* or *dynamic programming* strategy.

    (b) Prove the problem exhibits an *optimal substructure*. Moreover, if you used a greedy strategy, show that the *greedy choice property* holds for your algorithm. (*Hint:* the *greedy-choice* property holds iff every greedy choice is contained in an optimal solution; the optimization problem exhibits an *optimal substructure* iff an optimal solution to the problem contains within it optimal solutions to subproblems.)

    (c) Compute the time complexity of your solution.

```
/* Outputs the quantity of each piece taken */
float[] knapSack(int[] v, int[] w, int W) {
```

23. You are in front of a stack of pancakes of different diameter. Unfortunately, you cannot eat them unless they are sorted according to their size, with the biggest one at the bottom. To sort them, you are given a spatula that you can use to split the stack in two parts and then flip the top part of the stack. Write the pseudo-code of a function `sortPancakes` that sorts the stack. The i-th element of array `pancakes` contains the diameter of the i-th pancake, counting from the bottom. The `sortPancakes` algorithm can modify the stack only through the `spatulaFlip` function whose interface is specified below.

    (*Hint:* Notice that you can move a pancake at position x to position y, without modifying the positions of the order of the other pancakes, using a sequence of spatula flips.) *time: 20'*

```
/* Flips over the stack of pancakes from position pos and returns the result */
int[] spatulaFlip(int pos, int[] pancakes);

int[] sortPancakes(int[] pancakes) {
```

24. The following matrix represents a directed graph over vertices $a, b, c, d, e, f, g, h, i, j, k, \ell$. Rows and columns represent the source and destination of edges, respectively.

	a	b	c	d	e	f	g	h	i	j	k	ℓ
a					1	1						
b										1		
c								1			1	
d			1									
e		1								1		
f		1								1		
g			1	1								
h											1	1
i			1				1					
j												
k												1
ℓ												

Sort the vertices in a *reverse topological order* using the *depth-first search* algorithm. (Hint: if you order the vertices from left to right in reverse topological order, then all edges go from right to left.) Justify your answer by showing the relevant data maintained by the depth-first search algorithm, and by explaining how that can be used to produce a reverse topological order. *time: 15'*

25. Answer the following questions on the complexity classes **P** an **NP**. Justify your answers.

    (a) **P** $\subseteq$ **NP**? *time: 5'*

    (b) A problem Q is in **P** and there is a polynomial-time reduction from Q to Q'. What can we say about Q'? Is $Q' \in$ **P**? Is $Q' \in$ **NP**? *time: 5'*

    (c) Let Q be a problem defined as follows. *Input:* a set of numbers $A = \{a_1, a_2, \ldots, a_N\}$ and a number x; *Output:* 1 iff there are two values $a_i, a_k \in A$ such that $a_i + a_k = x$. Is Q in **NP**? Is Q in **P**? *time: 5'*

26. The *subset-sum* problem is defined as follows. *Input:* a set of numbers $A = \{a_1, a_2, \ldots, a_N\}$ and a number x; *Output:* 1 iff there is a subset of numbers in A that add up to x. Formally, $\exists S \in A$ such that $\sum_{y \in S} y = x$. Write a dynamic-programming algorithm to solve the subset-sum problem and informally analyze its complexity. *time: 20'*

27. Explain the idea of *dynamic programming* using the shortest-path problem as an example. (The shortest path problem amounts to finding the shortest path in a given graph $G = (V, E)$ between two given vertices a and b.) *time: 15'*

28. Consider an initially empty B-Tree with minimum degree $t = 3$. Draw the B-Tree after the insertion of the keys $27, 33, 39, 1, 3, 10, 7, 200, 23, 21, 20$, and then after the additional insertion of the keys $15, 18, 19, 13, 34, 200, 100, 50, 51$. *time: 10'*

29. There are three containers whose sizes are 10 pints, 7 pints, and 4 pints, respectively. The 7-pint and 4-pint containers start out full of water, but the 10-pint container is initially empty. Only one type of operation is allowed: pouring the contents of one container into another, stopping only when the source container is empty, or the destination container is full. Is there a sequence of pourings that leaves exactly two pints in either the 7-pint or the 4-pint container?

    (a) Model this as a graph problem: give a precise definition of the graph involved (type of the graph, labels on vertices, meaning of an edge). Provide the set of all reachable vertices, identify the initial vertex and the goal vertices. (Hint: all vertices that satisfy the condition imposed by the problem are reachable, so you don't have to draw a graph.)

(b) State the specific question about this graph that needs to be answered?

(c) What algorithm should be applied to solve the problem? Justify your answer.

time: 15'

30. Write an algorithm called MOVETOROOT(x, k) that, given a binary tree rooted at node x and a key k, moves the node containing k to the root position and returns that node if k is in the tree. If k is not in the tree, the algorithm must return x (the original root) without modifying the tree. Use the typical notation whereby $key(x)$ is the key stored at node x, $left(x)$ and $right(x)$ are the left and right children of x, respectively, and $parent(x)$ is x's parent node. *time: 15'*

31. Given a sequence of numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$, an *increasing subsequence* is a sequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of elements of A such that $1 \leq i_1 < i_2 < \ldots < i_k \leq n$, and such that $a_{i_1} < a_{i_2} < \ldots < a_{i_k}$. You must find the *longest increasing subsequence*. Solve the problem using dynamic programming.

(a) Define the *subproblem structure* and the solution of each subproblem. *time: 5'*

(b) Write an iterative algorithm that solves the problem. Illustrate the execution of the algorithm on the sequence $A = \langle 2, 4, 5, 6, 7, 9 \rangle$. *time: 10'*

(c) Write a recursive algorithm that solves the problem. Draw a tree of recursive calls for the algorithm execution on the sequence $A = \langle 1, 2, 3, 4, 5 \rangle$. *time: 10'*

(d) Compare the time complexities of the iterative and recursive algorithms. *time: 5'*

32. One way to implement a *disjoint-set* data structure is to represent each set by a linked list. The first node in each linked list serves as the representative of its set. Each node contains a key, a pointer to the next node, and a pointer back to the representative node. Each list maintains the pointers *head*, to the representative, and *tail*, to the last node in the list.

(a) Write the pseudo-code and analyze the time complexity for the following operations:

- MAKE-SET(x): creates a new set whose only member is x.
- UNION(x, y): returns the representative of the union of the sets that contain x and y.
- FIND-SET(x): returns a pointer to the representative of the set containing x.

Note that x and y are nodes. *time: 15'*

(b) Illustrate the linked list representation of the following sets:

- $\{c, a, d, b\}$
- $\{e, g, f\}$
- UNION(d, g)

time: 5'

33. Explain what it means for a hash function to be perfect for a given set of keys. Consider the hash function $h(x) = x \mod m$ that maps an integer x to a table entry in $\{0, 1, \ldots m-1\}$. Find an $m \leq 12$ such that h is a perfect hash function on the set of keys $\{0, 6, 9, 12, 22, 31\}$. *time: 10'*

34. Draw the binary search tree obtained when the keys $1, 2, 3, 4, 5, 6, 7$ are inserted in the given order into an initially empty tree. What is the problem of the tree you get? Why is it a problem? How could you modify the insertion algorithm to solve this problem. Justify your answer. *time: 10'*

35. Consider the following array:

$$A = \langle 4, 33, 6, 90, 33, 32, 31, 91, 90, 89, 50, 33 \rangle$$

(a) Is A a *min-heap*? Justify your answer by briefly explaining the *min-heap* property. *time: 10'*

(b) If A is a *min-heap*, then extract the minim value and then rearrange the array with the *min-heapify* procedure. In doing that, show the array at every iteration of *min-heapify*. If A is not a *min-heap*, then rearrange it to satisfy the *min-heap* property. *time: 10'*

36. Write the pseudo-code of the *insertion-sort* algorithm. Illustrate the execution of the algorithm on the array $A = \{3, 13, 89, 34, 21, 44, 99, 56, 9\}$, writing the intermediate values of A at each iteration of the algorithm. *time: 20'*

37. Encode the following sentence with a Huffman code

    Common sense is the collection of prejudices acquired by age eighteen

    Write the complete construction of the code. *time: 20'*

38. Consider the *text* and *query* strings:

    text: `It ain't over till it's over.`
    query: `over`

    Use the Boyer-Moore string-matching algorithm to search for the query in the text. For each character comparison performed by the algorithm, write the current *shift* and highlight the character position considered in the query string. Assume that indexes start from 0. The following table shows the first comparison as an example. Fill the rest of the table. *time: 10'*

n.	*shift*	I	t		a	i	n	'	t		o	v	e	r		t	i	l	l		i	t	'	s		o	v	e	r	.
1	0	o	v	e	r																									
2																														
...	...	...																												

39. Briefly answer the following questions

    (a) What does $f(n) = \Theta(g(n))$ mean? *time: 5'*

    (b) What kind of problems are in the P class? Give an example of a problem in P. *time: 5'*

    (c) What kind of problems are in the NP class? Give an example of a problem in NP. *time: 5'*

    (d) What does it mean for a problem A to be *reducible* to a problem B? *time: 5'*

40. For each of the following assertions, write "true," "false," or "?" depending on whether the assertion is true, false, or it may be either true or false. *time: 10'*

    (a) $P \subseteq NP$

    (b) The *knapsack* problem is in P

    (c) The *minimal spanning tree* problem is in NP

    (d) $n! = O(n^{100})$

    (e) $\sqrt{n} = \Omega(log(n))$

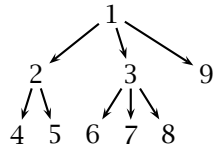
    (f) *insertion-sort* performs like *quicksort* on an almost sorted sequence

41. An application must read a long sequence of numbers given in no particular order, and perform many searches on that sequence. How would you implement that application to minimize the overall time-complexity? Write exactly what algorithms you would use, and in what sequence. In particular, write the high-level structure of a *read* function, to read and store the sequence, and a *find* function too look up a number in the sequence. *time: 10'*

42. Write an algorithm that takes a set of (x, y) coordinates representing points on a plane, and outputs the coordinates of two points with the maximal distance. The signature of the algorithm is MAXIMAL-DISTANCE(X, Y), where X and Y are two arrays of the same length representing the x and y coordinates of each point, respectively. Also, write the asymptotic complexity of MAXIMAL-DISTANCE. Briefly justify your answer. *time: 10'*

43. A *directed tree* is represented as follows: for each vertex v, *first-child*$[v]$ is either the first element in a list of child-vertices, or NIL if v is a leaf. For each vertex v, *next-sibling*$[v]$ is the next element in the list of v's siblings, or NIL if v is the last element in the list. For example, the arrays on the left represent the tree on the right:

v	1	2	3	4	5	6	7	8	9
first-child	2	4	6	NIL	NIL	NIL	NIL	NIL	NIL
next-sibling	NIL	3	9	5	NIL	7	8	NIL	NIL



(a) Write two algorithms, MAX-DEPTH(*root*) and MIN-DEPTH(*root*), that, given a tree, return the maximal and minimal depth of any leaf vertex, respectively. (E.g., the results for the example tree above are 2 and 1, respectively.) *time: 15'*

(b) Write an algorithm DEPTH-FIRST-ORDER(*root*) that, given a tree, prints the vertices in depth-first visitation order, such that a vertices is always preceded by all its children (e.g., the result for the example tree above is 4, 5, 2, 6, 7, 8, 3, 9, 1). *time: 10'*

(c) Write the asymptotic complexities of MAX-DEPTH, MIN-DEPTH, and DEPTH-FIRST-ORDER. Briefly justify your answers. *time: 5'*

44. Write an algorithm called IN-PLACE-SORT(A) that takes an array of numbers, and sorts the array *in-place*. That is, using only a constant amount of extra memory. Also, give an informal analysis of the asymptotic complexity of your algorithm. *time: 10'*

45. Given a sequence $A = \langle a_1, \ldots, a_n \rangle$ of numbers, the *zero-sum-subsequence* problem amounts to deciding whether A contains a subsequence of consecutive elements $a_i, a_{i+1}, \ldots, a_k$, with $1 \leq i \leq k \leq n$, such that $a_i + a_{i+1} + \cdots + a_k = 0$. Model this as a dynamic-programming problem and write a dynamic-programming algorithm ZERO-SUM-SEQUENCE(A) that, given an array A, returns TRUE if A contains a zero-sum subsequence, or FALSE otherwise. Also, give an informal analysis of the complexity of ZERO-SUM-SEQUENCE. *time: 30'*

46. Give an example of a randomized algorithm derived from a deterministic algorithm. Explain why there is an advantage in using the randomized variant. *time: 10'*

47. Implement the TERNARY-TREE-SEARCH(x, k) algorithm that takes the root of a ternary tree and returns the node containing key k. A ternary tree is conceptually identical to a binary tree, except that each node x has two keys, $key_1(x)$ and $key_2(x)$, and three links to child nodes, $left(x)$, $center(x)$, and $right(x)$, such that the left, center, and right subtrees contains keys that are, respectively, less than $key_1(x)$, between $key_1(x)$ and $key_2(x)$, and greater than $key_2(x)$. Assume there are no duplicate keys. Also, assuming the tree is balanced, what is the asymptotic complexity of the algorithm? *time: 10'*

48. Answer the following questions. Briefly justify your answers.

(a) A hash table that uses chaining has M slots and holds N keys. What is the expected complexity of a search operation? *time: 5'*

(b) The asymptotic complexity of algorithm A is $\Omega(N \log N)$, while that of B is $\Theta(N^2)$. Can we compare the two algorithms? If so, which one is asymptotically faster? *time: 5'*

(c) What is the difference between "Las Vegas" and "Monte Carlo" randomized algorithms? *time: 5'*

(d) What is the main difference between the Knuth-Morris-Pratt algorithm and Boyer-Moore string-matching algorithms in terms of complexity? Which one as the best worst-case complexity? *time: 5'*

49. A ternary search trie (TST) is used to implement a dictionary of strings. Write the TST corresponding to the following set of strings: "doc" "fun" "algo" "cat" "dog" "data" "car" "led" "function". Assume the strings are inserted in the given order. Use '#' as the terminator character. *time: 10'*

50. The following declarations define a ternary search trie in C and Java, respectively:

```
struct TST {
   char value;
   struct TST * higher;
   struct TST * lower;
   struct TST * equal;
};
void print(const struct TST * t);
```

```
public class TST {
   byte value;
   TST higher;
   TST lower;
   TST equal;
   void print() {/* ...  */}
};
```

The TST represents a dictionary of byte strings. The `print` method must output all the strings stored in the given TST, in alphabetical order. Assume the terminator value is 0. Write an implementation of the `print` method, either in C or in Java. You may assume that the TST contains strings of up to 100 characters. (Hint: store the output strings in a static array of characters.) *time: 20'*

51. Consider *quick-sort* as an in-place sorting algorithm.

    (a) Write the pseudo-code using only *swap* operations to modify the input array. *time: 10'*

    (b) Apply the algorithm of exercise 51a to the array $A = \langle 8, 2, 12, 17, 4, 8, 7, 1, 12 \rangle$. Write the content of the array after each swap operation. *time: 10'*

52. Consider this *minimal vertex cover* problem: given a graph $G = (V, E)$, find a minimal set of vertices S such that for every edge $(u, v) \in E$, u or v (or both) are in S.

    (a) Model *minimal vertex cover* as a dynamic-programming problem. Write the pseudo-code of a dynamic-programming solution. *time: 15'*

    (b) Do you think that your model of *minimal vertex cover* admits a greedy choice? Try at least one meaningful greedy strategy. Show that it does not work, giving a counter-example graph for which the strategy produces the wrong result. (*Hint:* one meaningful strategy is to choose a maximum-degree vertex first. The degree of a vertex is the number of its incident edges.) *time: 5'*

53. The graph $G = (V, E)$ represents a social network in which each vertex represents a person, and an edge $(u, v) \in E$ represents the fact that u and v know each other. Your problem is to organize the largest party in which nobody knows each other. This is also called the *maximal independent set* problem. Formally, given a graph $G = (V, E)$, find a set of vertices S of maximal size in which no two vertices are adjacent. (I.e., for all $u \in S$ and $v \in S$, $(u, v) \notin E$.)

    (a) Formulate a decision variant of *maximal independent set*. Say whether the problem is in NP, and briefly explain what that means. *time: 10'*

    (b) Write a verification algorithm for the *maximal independent set* problem. This algorithm, called TESTINDEPENDENTSET(G, S), takes a graph G represented through its adjacency matrix, and a set S of vertices, and returns TRUE if S is a valid independent set for G. *time: 10'*

54. A *Hamilton cycle* is a cycle in a graph that touches every vertex exactly once. Formally, in $G = (V, E)$, an ordering of *all* vertices $H = v_1, v_2, \ldots, v_n$ forms a Hamilton cycle if $(v_n, v_1) \in E$, and $(v_i, v_{i+1}) \in E$ for all i between 1 and $n-1$. Deciding whether a given graph is *Hamiltonian* (has a Hamilton cycle) is a well known NP-complete problem.

    (a) Write a verification algorithm for the *Hamiltonian graph* problem. This algorithm, called TESTHAMILTONCYCLE(G, H), takes a graph G represented through adjacency lists, and an array of vertices H, and returns TRUE if H is a valid Hamilton cycle in G. *time: 10'*

    (b) Give the asymptotic complexity of your implementation of TESTHAMILTONCYCLE. *time: 5'*

    (c) Explain what it means for a problem to be NP-complete. *time: 5'*

55. Consider using a b-tree with minimum degree $t = 2$ as an in-memory data structure to implement dynamic sets.

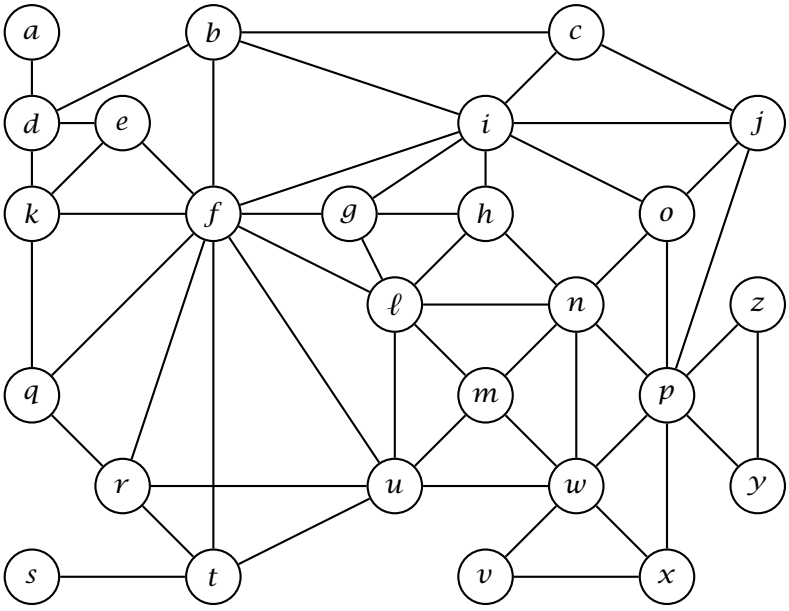
(a) Compare this data structure with a red-black tree. Is this data structure better, worse, or the same as a red-black tree in terms of time complexity? Briefly justify your answer. In particular, characterize the complexity of insertion and search. *time: 10'*

(b) Write an iterative (i.e., non-recursive) *search* algorithm for this degree-2 b-tree. Remember that the data structure is *in-memory*, so there is no need to perform any disk read/write operation. *time: 10'*

(c) Write the data structure after the insertion of keys $10, 3, 8, 21, 15, 4, 6, 19, 28, 31$, in this order, and then after the insertion of keys $25, 33, 7, 1, 23, 35, 24, 11, 2, 5$. *time: 10'*

(d) Write the insertion algorithm for this degree-2 b-tree. (*Hint:* since the minimum degree is fixed at 2, the insertion algorithm may be implemented in a simpler fashion without all the loops of the full b-tree insertion.) *time: 15'*

56. Consider a breadth-first search (BFS) on the following graph, starting from vertex a.



Write the two vectors π (previous) and d (distance), resulting from the BFS algorithm. *time: 10'*