Algorithms and Data Structures
Exercises

Antonio Carzaniga
University of Lugano

Edition 1.10
February 2014

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

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

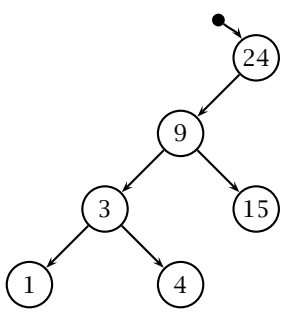
   For each fundamental iteration or recursion of the algorithm, write the content of the array. Assume the algorithm performs an in-place sort. *time: 20'*

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        x = TREE-SUCCESSOR(x)
4        if IS-PRIME(x.key)
5              return x
6  return x
```

```
IS-PRIME(n)
1  i = 2
2  while i · i ≤ n
3        if i divides n
4              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 x.right ≠ NIL
2  return TREE-MINIMUM(x.right)
3  y = x.parent
4  while y ≠ NIL and x == y.right
5        x = y
6        y = y.parent
7  return y
```

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2        x = x.left
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 H.heap-size > 0
2        HEAP-EXTRACT-MAX(H)
3        for i = 1 to H.heap-size
4              output H[i]
5        output "." END-OF-LINE
```

```
HEAP-EXTRACT-MAX(H)
1  if H.heap-size > 0
2        k = H[1]
3        H[1] = H[H.heap-size]
4        H.heap-size = H.heap-size − 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, $(a, b) \in R$ if and only if 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'*

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

(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 if and only if every greedy choice is contained in an optimal solution; the optimization problem exhibits an *optimal substructure* if and only if 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 if and only if 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 if and only if there is a subset of numbers in A that add up to x. Formally, $\exists S \subseteq A$ 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 $x.key$ is the key stored at node x, $x.left$ and $x.right$ are the left and right children of x, respectively, and $x.parent$ 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 minimum 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 $A = \langle 3, 13, 89, 34, 21, 44, 99, 56, 9 \rangle$, 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) $\mathbf{P} \subseteq \mathbf{NP}$

    (b) The *knapsack* problem is in **P**

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

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

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

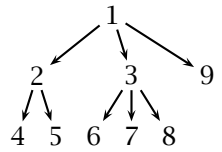
    (f) *insertion-sort* performs like *quicksort* on an almost sorted sequence

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, $v.$*first-child* is either the first element in a list of child-vertices, or NIL if v is a leaf. For each vertex v, $v.$*next-sibling* 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, $x.key_1$ and $x.key_2$, and three links to child nodes, $x.left$, $x.center$, and $x.right$, such that the left, center, and right subtrees contains keys that are, respectively, less than $x.key_1$, between $x.key_1$ and $x.key_2$, and greater than $x.key_2$. 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.

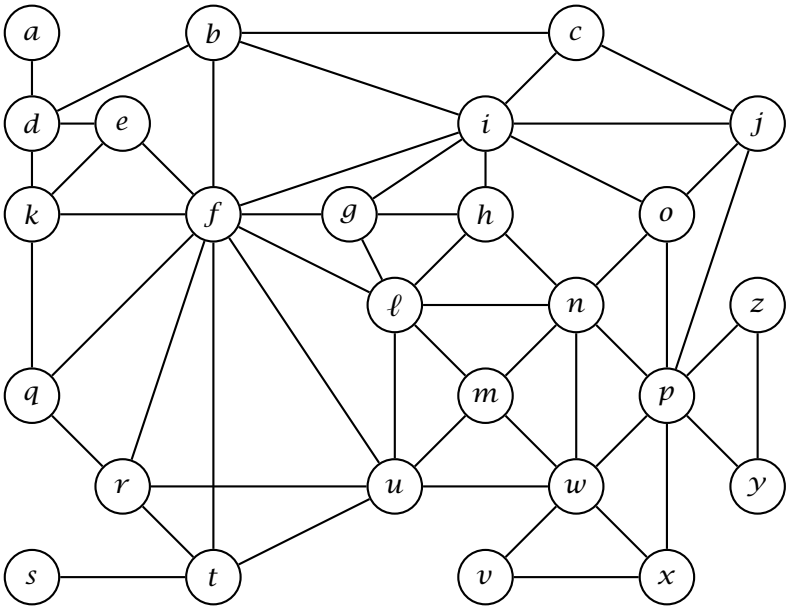
(a) Compare this data structure with a red-black tree. Is this data structure better, worse, or the same as a red-black tree in terms of time complexity? Briefly justify your answer. In particular, characterize the complexity of insertion and search. *time: 10'*

(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'*

57. Write a sorting algorithm MySort(A) that, in the average case, runs in time $O(n \log n)$ on arrays of size $n = |A|$. Also, characterize the best- and worst-case complexity of your solution. *time: 20'*

58. The following algorithms take an array A of integers. For each algorithm, write the asymptotic, best- and worst-case complexities as functions of the size of the input $n = |A|$. Your characterizations should be as tight as possible. Justify your answers by writing a short explanation of what each algorithm does. *time: 20'*

(a) ALGORITHM-I(A)

```
1  for i = 1 to |A| − 1
2      s = TRUE
3      for j = i + 1 to |A|
4          if A[i] > A[j]
5              swap A[i] ↔ A[j]
6              s = FALSE
7      if s == TRUE
8          return
```

(b) ALGORITHM-II(A)

```
1  i = 1
2  j = |A|
3  while i < j
4       if A[i] > A[j]
5            swap A[i] ↔ A[i + 1]
6            swap A[i] ↔ A[j]
7            i = i + 1
8       else j = j − 1
```

59. The following algorithms take a binary tree T containing n keys. For each algorithm, write the asymptotic, best- and worst-case complexities as functions of n. Your characterizations should be as tight as possible. Justify your answers by writing a short explanation of what each algorithm does. *time: 20'*

(a) ALGORITHM-III(T, k)

```
1  if T == NIL
2       return FALSE
3  if T.key == k
4       return TRUE
5  if ALGORITHM-III(T.left)
6       return TRUE
7  else return ALGORITHM-III(T.right)
```

(b) ALGORITHM-IV(T, k_1, k_2)

```
1  if T == NIL
2       return 0
3  if k1 > k2
4       swap k1 ↔ k2
5  r = 0
6  if T.key < k2
7       r = r + ALGORITHM-IV(T.right, k1, k2)
8  if T.key > k1
9       r = r + ALGORITHM-IV(T.left, k1, k2)
10 if T.key < k2 and T.key > k1
11      r = r + 1
12 return r
```

60. Answer the following questions on complexity theory. Justify your answers. All problems are decision problems. (*Hint:* answers are not limited to "yes" or "no.") *time: 20'*

(a) An algorithm A solves a problem P of size n in time $O(n^3)$. Is P in **NP**?

(b) An algorithm A solves a problem P of size n in time $\Omega(n \log n)$. Is P in **P**? Is it in **NP**?

(c) A problem P in **NP** can be polynomially reduced into a problem Q. Is Q in **P**? Is Q in **NP**?

(d) A problem P can be polynomially reduced into a problem Q in **NP**. Is P in **P**? Is P **NP-hard**?

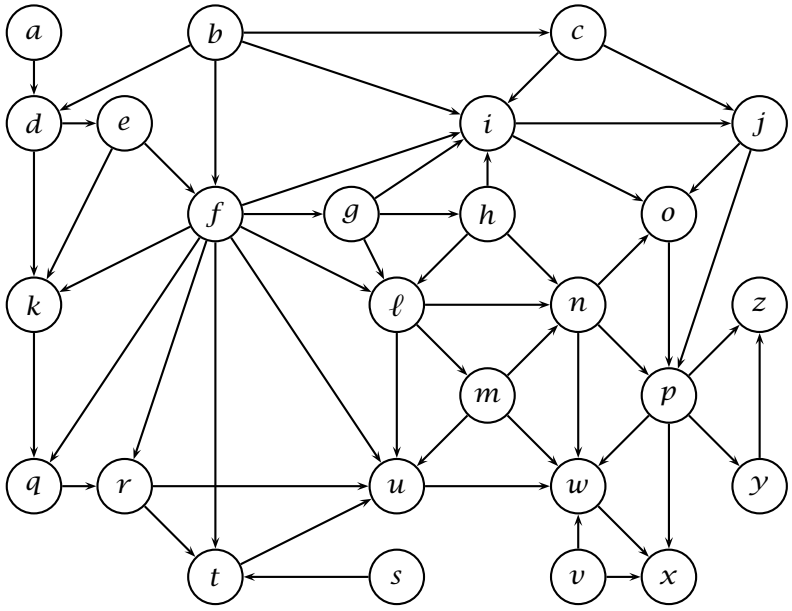
(e) A problem P of size n does not admit to any algorithmic solution with complexity $O(2^n)$. Is P in **P**? Is P in **NP**?

(f) An algorithm A takes an instance of a problem P of size n and a "certificate" of size $O(n^c)$, for some constant c, and *verifies* in time $O(n^2)$ that the solution to given problem is affirmative. Is P in **P**? Is P in **NP**? Is P **NP-complete**?

61. Write an algorithm TSTCOUNTGREATER(T, s) that takes the root T of a ternary-search trie (TST) and a string s, and returns the number of strings stored in the trie that are lexicographically greater than s. Given a node T, $T.left$, $T.middle$, and $T.right$ are the left, middle, and

right subtrees, respectively; $T.value$ is the value stored in T. The TST uses the special character '#' as the string terminator. Given two characters a and b, the relation $a < b$ defines the lexicographical order, and the terminator character is *less than* every other character. (**Hint:** first write an algorithm that, given a tree (node) counts *all* the strings stored in that tree.) *time: 20'*

62. Consider a depth-first search (DFS) on the following graph.



Write the three vectors π, d, and f that, for each vertex represent the *previous* vertex in the depth-first forest, the *discovery* time, and the *finish* time, respectively. Whenever necessary, iterate through vertexes in alphabetic order. *time: 20'*

63. Write an implementation of a *radix tree* in Java. The tree must store 32-bit integer keys. Each node in the tree must contain at most 256 links or keys, so each node would cover at most 8 bits of the key. You must implement a class called `RadixTree` with two methods, `void insert(int k)` and `boolean find(int k)`. You must also specify every other class you might use. For example, you would probably want to define a class `RadixTreeNode` to represent nodes in the tree. *time: 20'*

64. Answer the following questions about *red-black trees*.

    (a) Describe the structure and the properties of a red-black tree. *time: 5'*

    (b) Write an algorithm RB-TREE-SEARCH(T, k) that, given a red-black tree T and a key k, returns TRUE if T contains key k, or FALSE otherwise. *time: 5'*

    (c) Let h be the height of a red-black tree containing n keys, prove that

$$h \leq 2 \log (n + 1)$$

    Hint: first give an outline of the proof. Even if you can not give a complete proof, try to explain informally how the red-black tree property limits the height of a red-black tree. *time: 10'*

65. Sort the following functions in ascending order of asymptotic growth rate:

$$f_1(n) = 3^n \qquad f_2(n) = n^{1/3} \qquad f_3(n) = \log^2 n$$
$$f_4(n) = n^{\log n} \qquad f_5(n) = n^3 \qquad f_6(n) = 4^{\log n}$$
$$f_7(n) = n^2\sqrt{n} \qquad f_8(n) = 2^{2n} \qquad f_9(n) = \sqrt{\log n}$$

That is, write the sequence of sorted indexes $a_1, a_2, \ldots, a_9$ such that for all indexes a_i, a_j with $i < j$, $f_{a_i}(n) = O(f_{a_j}(n))$. (Notice that $\log n$ means $\log_2 n$.) *time: 10'*

66. Consider the following algorithm:

```
ALGO-A(X)
1  d = ∞
2  for i = 1 to X.length − 1
3       for j = i + 1 to X.length
4            if |X[i] − X[j]| < d
5                 d = |X[i] − X[j]|
6  return d
```

(a) Interpreting X as an array of coordinates of points on the x-axis, explain concisely what algorithm ALGO-A does, and give a tight asymptotic bound for the complexity of ALGO-A. *time: 5'*

(b) Write an algorithm BETTER-A(X) that is functionally equivalent to ALGO-A(X), but with a better asymptotic complexity. *time: 15'*

67. The following defines a *ternary search trie* (TST) for character strings, in Java and in pseudo-code notation:

`char c;`	$x.c$	character at node x
`boolean have_key;`	$x.have\text{-}key$	TRUE if node x represents a key
`TSTNode left;`	$x.left$	left child of node x
`TSTNode middle;`	$x.middle$	middle child of node x
`TSTNode right;`	$x.right$	right child of node x

```
class TSTNode {
    char c;
    boolean have_key;
    TSTNode left;
    TSTNode middle;
    TSTNode right;
}
```

Write an algorithm, `void TSTPrint(TSTNode t)` in Java or TST-PRINT(x) in pseudo-code that, given the root of a TST, prints all its keys in alphabetical order. *time: 20'*

68. A set of keys is stored in a *max-heap* H and in a *binary search tree* T. Which data structure offers the most efficient algorithm to output all the keys in descending order? Or are the two equivalent? Write both algorithms. Your algorithms may change the data structures. *time: 20'*

69. Answer the following questions. Briefly justify your answers. *time: 10'*

(a) Let A be an array of numbers sorted in descending order. Does A represent a max-heap (with $A.heap\text{-}size = A.length$)?

(b) A hash table has T slots and uses chaining to resolve collisions. What are the worst-case and average-case complexities of a search operation when the hash table contains N keys?

(c) A hash table with 9 slots, uses chaining to resolve collision, and uses the hash function $h(k) = k \mod 9$ (slots are numbered $0, \ldots, 8$). Draw the hash table after the insertion of keys 5, 28, 19, 15, 20, 33, 12, 17, and 10.

(d) Is the operation of deletion in a binary search tree *commutative* in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why it is or give a counter-example.

70. Draw a binary search tree containing keys $8, 27, 13, 15, 32, 20, 12, 50, 29, 11$, inserted in this order. Then, add keys $14, 18, 30, 31$, in this order, and again draw the tree. Then delete keys 29 and 27, in this order, and again draw the tree. *time: 10'*

71. Consider a *max-heap* containing keys $8, 27, 13, 15, 32, 20, 12, 50, 29, 11$, inserted in this order in an initially empty heap. Write the content of the array that stores the heap. Then, insert keys 43 and 51, and again write the content of the array. Then, extract the maximum value three times, and again write the content of the array. In all three cases, write the heap as an array. *time: 10'*

72. Consider a *min-heap* H and the following algorithm.

```
BST-FROM-MIN-HEAP(H)
1  T = NEW-EMPTY-TREE()
2  for i = 1 to H.heap-length
3       TREE-INSERT(T, H[i])    // binary-search-tree insertion
4  return T
```

Prove that BST-FROM-MIN-HEAP does not always produce minimum-height binary trees. *time: 10'*

73. Consider an array A containing n numbers and satisfying the *min-heap* property. Write an algorithm MIN-HEAP-FAST-SEARCH(A, k) that finds k in A with a time complexity that is better than linear in n whenever at most $\sqrt{n}$ of the values in A are less than k. *time: 20'*

74. Write an algorithm B-TREE-TOP-K(R, k) that, given the root R of a b-tree of minimum degree t, and an integer k, outputs the largest k keys in the b-tree. You may assume that the entire b-tree resides in main memory, so no disk access is required. (*Reminder:* a node x in a b-tree has the following properties: $x.n$ is the number of keys, $X.key[1] \leq x.key[2] \leq \ldots x.key[x.n]$ are the keys, $x.leaf$ tells whether x is a leaf, and $x.c[1], x.c[2], \ldots, x.c[x.n+1]$ are the pointers to x's children.) *time: 30'*

75. Your computer has a special machine instruction called SORT-FIVE(A, i) that, given an array A and a position i, sorts in-place and in a single step the elements $A[i \ldots i+5]$ (or $A[i \ldots |A|]$ if $|A| < i+5$). Write an in-place sorting algorithm called SORT-WITH-SORT-FIVE that uses only SORT-FIVE to modify the array A. Also, analyze the complexity of SORT-WITH-SORT-FIVE. *time: 20'*

76. For each of the following statements, briefly argue why they are true, or show a counterexample. *time: 10'*

    (a) $f(n) = O(n!) \Longrightarrow \log(f(n)) = O(n \log n)$

    (b) $f(n) = \Theta(f(n/2))$

    (c) $f(n) + g(n) = \Theta(\min(f(n), g(n)))$

    (d) $f(n)g(n) = O(\max(f(n), g(n)))$

    (e) $f(g(n)) = \Omega(\min(f(n), g(n)))$

77. Characterize the complexity of the following algorithm. Briefly justify your answer. *time: 10'*

```
SHUFFLE-A-BIT(A)
 1  i = 1
 2  j = A.length
 3  if j > i
 4       while j > i
 5            p = CHOOSE-UNIFORMLY({0, 1})
 6            if p == 1
 7                 swap A[i] ↔ A[j]
 8            j = j − 1
 9            i = i + 1
10       SHUFFLE-A-BIT(A[1 ... j])
11       SHUFFLE-A-BIT(A[i ... A.length])
```

78. Answer the following questions. For each question, write "yes" when the answer is always true, "no" when it is always false, "undefined" when it can be true or false. *time: 10'*

(a) Algorithm A solves decision problem X in time $O(n \log n)$.

i. Is X in **NP**?

ii. Is X in **P**?

(b) Decision problem X in **P** can be polynomially reduced to problem Y.

i. Is there a polynomial-time algorithm to solve Y?

(c) Decision problem X can be polynomially reduced to a problem Y for which there is a polynomial-time verification algorithm.

i. Is X in **NP**?

ii. Is X in **P**?

(d) An **NP-hard** decision problem X can be polynomially reduced to problem Y.

i. Is Y in **NP**?

ii. Is Y **NP-hard**?

(e) Algorithm A solves decision problem X in time $\Theta(2^n)$.

i. Is X in **NP**?

ii. Is X in **P**?

79. Write a minimal character-based binary code for the following sentence:

in theory, there is no difference between theory and practice; in practice, there is.

The code must map each character, including spaces and punctuation marks, to a binary string so that the total length of the encoded sentence is minimal. Use a Huffman code and show the derivation of the code. *time: 20'*

80. The following matrix represents a directed graph over 12 vertices labeled $a, b, c, \ldots, \ell$. Rows and columns represent the source and destination of edges, respectively. For example, the value 1 in row a and column f indicates an edge from a to f.

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

Run a *breadth-first search* on the graph starting from vertex a. Using the table below, write the two vectors π (previous) and d (distance) at each main iteration of the BFS algorithm. Write the pair π, d in each cell; for each iteration, write only the values that change. Also, write the complete BFS tree after the termination of the algorithm. *time: 20'*

a	b	c	d	e	f	g	h	i	j	k	ℓ
$a, 0$	$-, \infty$	$-, \infty$	$-, \infty$	$-, \infty$	$-, \infty$	$-, \infty$	$-, \infty$	$-, \infty$	$-, \infty$	$-, \infty$	$-, \infty$

81. A *graph coloring* associates a color with each vertex of a graph so that adjacent vertices have different colors. Write a greedy algorithm that tries to color a given graph with the least number of colors. This is a well known and difficult problem for which, most likely, there is no perfect greedy strategy. So, you should use a *reasonable* strategy, and it is okay if your algorithm does not return the absolute best coloring. The result must be a *color* array, where $v.color$ is a number representing the color of vertex v. Write the algorithm, analyze its complexity, and also show an example in which the algorithm does not achieve the best possible result. *time: 20'*

82. Given an array A and a positive integer k, the *selection* problem amounts to finding the largest element $x \in A$ such that at most k elements of A are less than or equal to x, or NIL if no such element exists. A simple way to implement it is as follows:

```
SIMPLESELECTION(A, k)
1  if k > A.length
2       return NIL
3  else sort A in ascending order
4       return A[k]
```

Write another algorithm that solves the selection problem without first sorting A. (**Hint:** use a divide-and-conquer strategy that "divides" A using one of its elements.) Also, illustrate the execution of the algorithm on the following input by writing its state at each main iteration or recursion.

$$A = \langle 29, 28, 35, 20, 9, 33, 8, 9, 11, 6, 21, 28, 18, 36, 1 \rangle \qquad k = 6$$

time: 20'

83. Consider the following *maximum-value contiguous subsequence* problem: given a sequence of numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$, find two positions i and j, with $1 \le i \le j \le n$, such that the sum $a_i + a_{i+1} + \cdots + a_j$ is maximal.

    (a) Write an algorithm to solve the problem and analyze its complexity. *time: 10'*

    (b) If you have not already done so for exercise 83a, write an algorithm that solves the maximum-value contiguous subsequence problem in time $O(n)$. (**Hint:** one such algorithm uses dynamic-programming.) *time: 20'*

84. Consider the following intuitive definition of the *size* of a binary search (sub)tree t: $size(t) = 0$ if t is NIL, or $size(t) = 1 + size(t.left) + size(t.right)$ otherwise. For each node t in a tree, let attribute $t.size$ denote the size of the subtree rooted at t.

    (a) Prove that, if for each node t in a tree T, $\max\{size(t.left), size(t.right)\} \le \frac{2}{3} size(t)$, then the height of T is $O(\log n)$, where $n = size(T)$. *time: 10'*

    (b) Write the rotation procedures ROTATE-LEFT(t) and ROTATE-RIGHT(t) that return the left- and right rotation of tree t maintaining the correct *size* attributes. *time: 10'*

    (c) Write an algorithm called SELECTION(T, i) that, given a tree T where each node t carries its size in $t.size$, returns the i-th key in T. *time: 10'*

    (d) A tree T is *perfectly balanced* when $\max\{size(t.left), size(t.right)\} = \lfloor size(t)/2 \rfloor$ for all nodes $t \in T$. Write an algorithm called BALANCE(T) that, using the rotation procedures defined in exercise 84b, balances T perfectly. (**Hint:** the essential operation is to move the median value of a subtree to the root of that subtree.) *time: 30'*

85. Write the *heap-sort* algorithm and illustrate its execution on the following sequence.

$$A = \langle 1, 1, 24, 8, 3, 36, 34, 23, 4, 30 \rangle$$

Assuming the sequence A is stored in an array passed to the algorithm, for each main iteration (or recursion) of the algorithm, write the content of the array. *time: 10'*

86. A radix tree is used to represent a dictionary of words defined over the alphabet of the 26 letters of the English language. Assume that letters from A to Z are represented as numbers from 1 to 26. For each node x of the tree, $x.links$ is the array of links to other nodes, and $x.value$ is a Boolean value that is true when x represents a word in the dictionary. Write an algorithm PRINT-RADIX-TREE(T) that outputs all the words in the dictionary rooted at T. *time: 10'*

87. Consider the following algorithm that takes an array A of length $A.length$:

```
ALGO-X(A)
1  for i = 3 to A.length
2      for j = 2 to i − 1
3          for k = 1 to j − 1
4              if |A[i] − A[j]| == |A[j] − A[k]| or |A[i] − A[k]| == |A[j] − A[k]|
5                  return TRUE
6  return FALSE
```

Write an algorithm BETTER-ALGO-X(A) equivalent to ALGO-X(A) (for all A) but with a strictly better asymptotic complexity than ALGO-X(A). *time: 20'*

88. For each of the following statements, write whether it is correct or not. Justify your answer by briefly arguing why it is correct, or otherwise by giving a counter example. *time: 10'*

    (a) If $f(n) = O(g^2(n))$ then $f(n) = \Omega(g(n))$.

    (b) If $f(n) = \Theta(2^n)$ then $f(n) = \Theta(3^n)$.

    (c) If $f(n) = O(n^3)$ then $\log(f(n)) = O(\log n)$.

    (d) $f(n) = \Theta(f(2n))$

    (e) $f(2n) = \Omega(f(n))$

89. Write an algorithm PARTITION(A, k) that, given an array A of numbers and a value k, changes A in-place by only swapping two of its elements at a time so that all elements that are less then or equal to k precede all other elements. *time: 10'*

90. Consider an initially empty B-Tree with minimum degree $t = 2$.

    (a) Draw the tree after the insertion of keys 81, 56, 16, 31, 50, 71, 58, 83, 0, and 60 in this order. *time: 10'*

    (b) Can a different insertion order produce a different tree? If so, write the same set of keys in a different order and the corresponding B-Tree. If not, explain why. *time: 10'*

91. Consider the following decision problem. Given a set of integers A, output 1 if some of the numbers in A add up to a multiple of 10, or 0 otherwise.

    (a) Is this problem in NP? If it is, then write a corresponding verification algorithm. If not, explain why not. *time: 5'*

    (b) Is this problem in P? If it is, then write a polynomial-time solution algorithm. Otherwise, argue why not. (*Hint:* consider the input values modulo 10. That is, for each input value, consider the remainder of its division by 10.) *time: 15'*

92. The following greedy algorithm is intended to find the shortest path between vertices u and v in a graph $G = (V, E, w)$, where $w(x, y)$ is the length of edge $(x, y) \in E$.

GREEDY-SHORTEST-PATH($G = (V, E, w), u, v$)

```
1  Visited = {u}                    // this is a set
2  path = ⟨u⟩                       // this is a sequence
3  while path not empty
4      x = last vertex in path
5      if x == v
6          return path
7      y = vertex y ∈ Adj[x] such that y ∉ Visited and w(x, y) is minimal
                                    // y is x's closest neighbor not already visited
8      if y == UNDEFINED            // all neighbors of x have already been visited
9          path = path − ⟨x⟩        // removes the last element y from path
10     else Visited = Visited ∪ {y}
11         path = path + ⟨y⟩        // append y to path
12 return UNDEFINED                 // there is no path between u and v
```

Does this algorithm find the shortest path always, sometimes, or never? If it always works, then explain its correctness by defining a suitable invariant for the main loop, or explain why the greedy choice is correct. If it works sometimes (but not always) show a positive example and a negative example, and briefly explain why the greedy choice does not work. If it is never correct, show an example and briefly explain why the greedy choice does not work. *time: 20'*

93. Write the quick-sort algorithm as a deterministic in-place algorithm, and then apply it to the array

$$\langle 50, 47, 92, 78, 76, 7, 60, 36, 59, 30, 50, 43 \rangle$$

Show the application of the algorithm by writing the content of the array after each main iteration or recursion. *time: 20'*

94. Consider an undirected graph G of n vertices represented by its adjacency matrix A. Write an algorithm called IS-CYCLIC(A) that, given the adjacency matrix A, returns TRUE if G contains a cycle, or FALSE if G is acyclic. Also, give a precise analysis of the complexity of your algorithm. *time: 20'*

95. A palindrome is a sequence of characters that is identical when read left-to-right and right-to-left. For example, the word "racecar" is a palindrome, as is the phrase "rats live on no evil star." Write an algorithm called LONGEST-PALINDROME(T) that, given an array of characters T, prints the longest palindrome in T, or any one of them if there are more than one. For example, if T is the text "radar radiations" then your algorithm should output "dar rad". Also, give a precise analysis of the complexity of your algorithm. *time: 20'*

96. Write an algorithm called OCCURRENCES that, given an array of numbers A, prints all the distinct values in A each followed by its number of occurrences. For example, if $A = \langle 28, 1, 0, 1, 0, 3, 4, 0, 0, 3 \rangle$, the algorithm should output the following five lines (here separated by a semicolon) "28 1; 1 2; 0 4; 3 2; 4 1". The algorithm may modify the content of A, but may not use any other memory. Each distinct value must be printed exactly once. Values may be printed in any order. The complexity of the algorithm must be $o(n^2)$, that is, strictly lower than $O(n^2)$. *time: 20'*

97. The following algorithm takes an array of line segments. Each line segment s is defined by its two end-points $s.a$ and $s.b$, each defined by their Cartesian coordinates $(s.a.x, s.a.y)$ and $(s.b.x, s.b.y)$, respectively, and ordered such that either $s.a.x < s.b.x$ or $s.a.x = s.b.x$ and $s.a.y < s.b.y$. That is, $s.b$ is never to the left of $s.a$, and if $s.a$ and $s.b$ have the same x coordinates, then $s.a$ is below $s.b$.

EQUALS(p, q)

```
   // tests whether p and q are the same point
1  if p.x == q.x and p.y == q.y
2      return TRUE
3  else return FALSE
```

```
ALGO-X(A)
1  for i = 1 to A.length
2      for j = 1 to A.length
3          if EQUALS(A[i].b, A[j].a)
4              for k = 1 to A.length
5                  if EQUALS(A[j].b, A[k].b) and EQUALS(A[i].a, A[k].a)
6                      return TRUE
7  return FALSE
```

(a) Analyze the asymptotic complexity of ALGO-X *time: 10'*

(b) Write an algorithm ALGO-Y that does exactly what ALGO-X does but with a better asymptotic complexity. Also, write the asymptotic complexity of ALGO-Y. *time: 20'*

98. Write an algorithm called TREE-TO-VINE that, given a binary search tree T, returns the same tree changed into a *vine*, that is, a tree containing exactly the same nodes but restructured so that no node has a left child (i.e., the returned tree looks like a linked list). The algorithm must not destroy or create nodes or use any additional memory other than what is already in the tree, and therefore must operate through a sequence of *rotations*. Write explicitly all the rotation algorithms used in TREE-TO-VINE. Also, analyze the complexity of TREE-TO-VINE. *time: 15'*

99. We say that a binary tree T is *perfectly balanced* if, for each node n in T, the number of keys in the left and right subtrees of n differ at most by 1. Write an algorithm called IS-PERFECTLY-BALANCED that, given a binary tree T returns TRUE if T is perfectly balanced, and FALSE otherwise. Also, analyze the complexity of IS-PERFECTLY-BALANCED. *time: 15'*

100. Two graphs G and H are *isomorphic* if there exists a *bijection* $f : V(G) \to V(H)$ between the vertexes of G and H (i.e., a one-to-one correspondence) such that any two vertices u and v in G are adjacent (in G) if and only if $f(u)$ and $f(v)$ are adjacent in H. The *graph-isomorphism* problem is the problem of deciding whether two given graphs are isomorphic.

(a) Is graph isomorphism in NP? If so, explain why and write a verification procedure. If not, argue why not. *time: 10'*

(b) Consider the following algorithm to solve the graph-isomorphism problem:

```
ISOMORPHIC(G, H)
1  if |V(G)| ≠ |V(H)|
2      return FALSE
3  A = V(G) sorted by degree    // A is a sequence of the vertices of G
4  B = V(H) sorted by degree    // B is a sequence of the vertices of H
5  for i = 1 to |V(G)|
6      if degree(A[i]) ≠ degree(B[i])
7          return FALSE
8  return TRUE
```

Is ISOMORPHIC correct? If so, explain at a high level what the algorithm does and informally but precisely why it works. If not, show a counter-example. *time: 10'*

101. Write an algorithm HEAP-PRINT-IN-ORDER(H) that takes a min heap H containing unique elements (no element appears twice in H) and prints the elements of H in increasing order. The algorithm must not modify H and may not use any additional memory. Also, analyze the complexity of HEAP-PRINT-IN-ORDER. *time: 20'*

102. Write an algorithm called BST-RANGE-WEIGHT(T, a, b) that takes a well balanced binary search tree T (or more specifically the root T of such a tree) and two keys a and b, with $a \leq b$, and returns the number of keys in T that are between a and b. Assuming there are $o(n)$ such keys, then the algorithm should have a complexity of $o(n)$, that is, strictly better than linear in the size of the tree. Analyze the complexity of BST-RANGE-WEIGHT. *time: 10'*

103. Let (a, b) represent an interval (or range) of values x such that $a \leq x \leq b$. Consider an array $X = \langle a_1, b_1, a_2, b_2, \ldots, a_n, b_n \rangle$ of $2n$ numbers representing n intervals (a_i, b_i), where $a_i = X[2i-1]$ and $b_i = X[2i]$ and $a_i \leq b_i$. Write an algorithm called SIMPLIFY-INTERVALS(X) that takes an array X representing n intervals, and simplifies X in-place. The "simplification" of a set of intervals X is a minimal set of intervals representing the *union* of all the intervals in X. Notice that the union of two disjoint intervals can not be simplified, but the union of two partially overlapping intervals can be simplified into a single interval. For example, a correct solution for the simplification of $X = \langle 3, 7, 1, 5, 10, 12, 6, 8 \rangle$ is $X = \langle 10, 12, 1, 8 \rangle$. An array X can be shrunk by setting its length (effectively removing elements at the end of the array). In this example, $X.length$ should be 4 after the execution of the simplification algorithm. Analyze the complexity of SIMPLIFY-INTERVALS. *time: 30'*

104. Write an algorithm SIMPLIFY-INTERVALS-FAST(X) that solves problem 103 with a complexity of $O(n \log n)$. If your solution for problem 103 already has an $O(n \log n)$ complexity, then simply say so. *time: 20'*

105. Consider the following algorithm:

```
ALGO-X(A, k)                    ALGO-Y(A, i)
1  i = 1                        1  while i < A.length
2  while i ≤ A.length           2        A[i] = A[i + 1]
3        if A[i] == k           3        i = i + 1
4              ALGO-Y(A, i)     4  A[i] = NULL
5        else i = i + 1
```

Analyze the complexity of ALGO-X and write an algorithm called BETTER-ALGO-X that does exactly the same thing, but with a strictly better asymptotic complexity. Analyze the complexity of BETTER-ALGO-X. *time: 20'*

106. Write an in-place partition algorithm called MODULO-PARTITION(A) that takes an array A of n numbers and changes A in such a way that (1) the final content of A is a permutation of the initial content of A, and (2) all the values that are equivalent to $0 \mod 10$ precede all the values equivalent to $1 \mod 10$, which precede all the values equivalent to $2 \mod 10$, etc. Being an in-place algorithm, MODULO-PARTITION must not allocate more than a constant amount of memory. For example, for an input array $A = \langle 7, 62, 5, 57, 12, 39, 5, 8, 16, 48 \rangle$, a correct result would be $A = \langle 12, 62, 5, 5, 16, 57, 7, 8, 48, 39 \rangle$. Analyze the complexity of MODULO-PARTITION. *time: 30'*

107. Write the *merge sort* algorithm and analyze its complexity. *time: 10'*

108. Write an algorithm called LONGEST-REPEATED-SUBSTRING(T) that takes a string T representing some text, and finds the longest string that occurs at least twice in T. The algorithm returns three numbers $begin_1, end_1$, and $begin_2$, where $begin_1 \leq end_1$ represent the first and last position of the *longest* substring of T that also occurs starting at another position $begin_2 \neq begin_1$ in T. If no such substring exist, then the algorithm returns "None." Analyze the time and space complexity of your algorithm. *time: 20'*

109. Answer the following questions on complexity theory. Reminder: SAT is the Boolean satisfiability problem, which is a well-known NP-complete problem.

    (a) A decision problem Q is polynomially-reducible to SAT. Can we say for sure that Q is NP-complete? *time: 2'*

    (b) SAT is polynomially-reducible to a decision problem Q. Can we say for sure that Q is NP-complete? *time: 2'*

    (c) A decision problem Q is polynomially reducible to a problem Q' and Q' is polynomially reducible to SAT. Can we say for sure that Q is in NP? *time: 2'*

(d) An algorithm A solves every instance of a decision problem Q of size n in $O(n^3)$ time. Also, Q is polynomially reducible to another problem Q'. Can we say for sure that Q' is in NP? *time: 2'*

(e) A decision problem Q is polynomially reducible to another decision problem Q', and an algorithm A solves Q' with complexity $O(n \log n)$. Can we say for sure that Q is in NP? *time: 2'*

(f) Consider the following decision problem Q: given a graph G, output 1 if G is connected (i.e., there exists a path between each pair of vertices) or 0 otherwise. Is Q in P? If so, outline an algorithm that proves it, if not argue why not. *time: 10'*

(g) Consider the following decision problem Q: given a graph G and an integer k, output 1 if G contains a cycle of size k. Is Q in NP? If so, outline an algorithm that proves it, if not argue why not. *time: 10'*

110. Consider an initially empty B-tree with minimum degree $t = 3$. Draw the B-tree after the insertion of the keys 84, 13, 36, 91, 98, 14, 81, 95, 12, 63, 31, and then after the additional insertion of the keys 65, 62, 187, 188, 57, 127, 6, 195, 25. *time: 10'*

111. Write an algorithm B-TREE-RANGE(T, k_1, k_2) that takes a B-tree T and two keys $k_1 \leq k_2$, and prints all the keys in T between k_1 and k_2 (inclusive). *time: 20'*

112. Write an algorithm called FIND-TRIANGLE(G) that takes a graph represented by its *adjacency list* G and returns true if G contains a triangle. A triangle in a graph G is a triple of vertices u, v, w such that all three edges (u, v), (v, w), and (u, w) are in G. Analyze the complexity of FIND-TRIANGLE. *time: 15'*

113. Write an algorithm MIN-HEAP-INSERT(H, k) that inserts a key k in a min-heap H. Also, illustrate the algorithm by writing the content of the array H after the insertion of keys 84, 13, 36, 91, 98, 14, 81, 95, 12, 63, 31, and then after the additional insertion of the key 15. *time: 15'*

114. Implement a priority queue by writing two algorithms:

- ENQUEUE(Q, x, p) enqueues an object x with priority p, and
- DEQUEUE(Q) extracts and returns an object from the queue.

The behavior of ENQUEUE and DEQUEUE is such that, if a call ENQUEUE(Q, x_1, p_1) is followed (not necessarily immediately) by another call ENQUEUE(Q, x_2, p_2), then x_1 is dequeued before x_2 unless $p_2 > p_1$. Implement ENQUEUE and DEQUEUE such that their complexity is $o(n)$ for a queue of n elements (i.e., strictly less than linear). *time: 20'*

115. Write an algorithm called MAX-HEAP-MERGE-NEW(H_1, H_2) that takes two max-heaps H_1 and H_2, and returns a new max-heap that contains all the elements of H_1 and H_2. MAX-HEAP-MERGE-NEW must create a *new* max heap, therefore it must allocate a new heap H and somehow copy all the elements from H_1 and H_2 into H without modifying H_1 and H_2. Also, analyze the complexity of MAX-HEAP-MERGE-NEW. *time: 20'*

116. Write an algorithm called BST-MERGE-INPLACE(T_1, T_2) that takes two binary-search trees T_1 and T_2, and returns a new binary-search tree by merging all the elements of T_1 and T_2. BST-MERGE-INPLACE is *in-place* in the sense that it must rearrange the nodes of T_1 and T_2 in a single binary-search tree without creating any new node. Also, analyze the complexity of BST-MERGE-INPLACE. *time: 20'*

117. Let A be an array of points in the 2D Euclidean space, each with its Cartesian coordinates $A[i].x$ and $A[i].y$. Write an algorithm MINIMUM-BOUNDING-RECTANGLE(A) that, given an array A of n points, in $O(n)$ time returns the smallest axis-aligned rectangle that contains all the points in A. MINIMUM-BOUNDING-RECTANGLE must return a pair of points corresponding to the bottom-left and top-right corners of the rectangle, respectively. *time: 10'*

118. Let A be an array of points in the 2D Euclidean space, each with its Cartesian coordinates $A[i].x$ and $A[i].y$. Write an algorithm LARGEST-CLUSTER(A, ℓ) that, given an array A of points and a length ℓ, returns the maximum number of points in A that are contained in a square of size ℓ. Also, analyze the complexity of LARGEST-CLUSTER. *time: 30'*

119. Consider the following algorithm that takes an array of numbers:

```
ALGO-X(A)
 1  i = 1
 2  j = 1
 3  m = 0
 4  c = 0
 5  while i ≤ |A|
 6        if A[i] == A[j]
 7              c = c + 1
 8        j = j + 1
 9        if j > |A|
10              if c > m
11                    m = c
12              c = 0
13              i = i + 1
14              j = i
15  return m
```

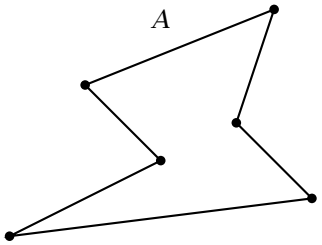
(a) Analyze the complexity of ALGO-X. *time: 5'*

(b) Write an algorithm that does exactly the same thing as ALGO-X but with a strictly better asymptotic time complexity. *time: 15'*

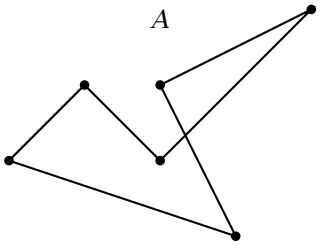
120. Write a THREE-WAY-MERGE(A, B, C) algorithm that merges three sorted sequences into a single sorted sequence, and use it to implement a THREE-WAY-MERGE-SORT(L) algorithm. Also, analyze the complexity of THREE-WAY-MERGE-SORT. *time: 20'*

121. Write an algorithm IS-SIMPLE-POLYGON(A) that takes a sequence A of 2D points, where each point $A[i]$ is defined by its Cartesian coordinates $A[i].x$ and $A[i].y$, and returns TRUE if A defines a simple polygon, or FALSE otherwise. Also, analyze the complexity of IS-SIMPLE-POLYGON. A polygon is *simple* if its line segments do not intersect.

Example:



IS-SIMPLE-POLYGON(A) = TRUE

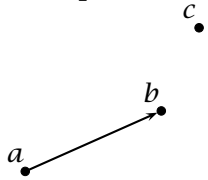


IS-SIMPLE-POLYGON(A) = FALSE

Hint: Use the following DIRECTION-ABC algorithm to determine whether a point c is *on the left side*, *collinear*, or *on the right side* of a segment ab:

```
DIRECTION-ABC(a, b, c)
1  d = (b.x − a.x)(c.y − a.y) − (b.y − a.y)(c.x − a.x)
2  if d > 0
3        return LEFT
4  elseif d == 0
5        return CO-LINEAR
6  else return RIGHT
```

Example:



DIRECTION-ABC(a, b, c) = LEFT

time: 20'

122. Implement a dictionary data structure that supports *longest prefix matching*. Specifically, write the following two algorithms:

- BUILD-DICTIONARY(W) takes a list W of n strings and builds the dictionary.
- LONGEST-PREFIX(k) takes a string k and returns the longest prefix of k found in the dictionary, or NULL if none exists. The time complexity of LONGEST-PREFIX(k) must be $o(n)$, that is, sublinear in the size n of the dictionary.

For example, assuming the dictionary was built with strings, "luna", "lunatic", "a", "al", "algo", "an", "anto", then if k is "algorithms", then LONGEST-PREFIX(k) should return "algo", or if k is "anarchy" then LONGEST-PREFIX(k) should return "an", or if k is "lugano" then LONGEST-PREFIX(k) should return NULL. *time: 20'*

123. Consider the following decision problem: given a set S of character strings, with characters of a fixed alphabet (e.g., the Roman alphabet), and given an integer k, return TRUE if there are at least k strings in S that have a common substring.

    (a) Is the problem in NP? Write an algorithm that proves it is, or argue the opposite. *time: 5'*

    (b) Is the problem in P? Write an algorithm that proves it is, or argue the opposite. *time: 15'*

124. Draw a red-black tree containing the following set of keys, clearly indicating the color of each node.

$$\{8, 7, 7, 35, 23, 35, 13, 7, 23, 18, 3, 19, 22\}$$

time: 10'

125. Consider the following algorithm ALGO-X that takes an array A of n numbers:

```
ALGO-X(A)
1  return ALGO-XR(A, 0, 1, 2)
```

```
ALGO-XR(A, t, i, r)
1  while i ≤ A.length
2      if r == 0
3          if A[i] == t
4              return TRUE
5      else if ALGO-XR(A, t − A[i], i + 1, r − 1)
6              return TRUE
7      i = i + 1
8  return FALSE
```

Analyze the complexity of ALGO-X and then write an algorithm BETTER-ALGO-X that does exactly the same thing but with a strictly better time complexity. *time: 30'*

126. A Eulerian cycle in a graph is a cycle that goes through each edge exactly once. As it turns out, a graph contains a Eulerian cycle if (1) it is connected, and (2) all its vertexes have even degree. Write an algorithm EULERIAN(G) that takes a graph G represented as an adjacency matrix, and returns TRUE when G contains a Eulerian cycle. *time: 10'*

127. Consider a social network system that, for each user u, stores u's friends in a list *friends*(u). Implement an algorithm TOP-THREE-FRIENDS-OF-FRIENDS(u) that, given a user u, recommends the three other users that are not already among u's friends but are among the friends of most of u's friends. Also, analyze the complexity of the TOP-THREE-FRIENDS-OF-FRIENDS algorithm. *time: 20'*

128. Consider the following algorithm:

```
ALGO-X(A)
 1  for i = 3 to A.length
 2      for j = 2 to i − 1
 3          for k = 1 to j − 1
 4              x = A[i]
 5              y = A[j]
 6              z = A[k]
 7              if x > y
 8                  swap x ↔ y
 9              if y > z
10                  swap y ↔ z
11                  if x > y
12                      swap x ↔ y
13              if y − x == z − y
14                  return TRUE
15  return FALSE
```

Analyze the complexity of ALGO-X and write an algorithm called BETTER-ALGO-X(A) that does the same as ALGO-X(A) but with a strictly better asymptotic time complexity and with the same space complexity. *time: 20'*

129. The weather service stores the daily temperature measurements for each city as vectors of real numbers.

(a) Write an algorithm called HOT-DAYS(A, t) that takes an array A of daily temperature measurements for a city and a temperature t, and returns the maximum number of consecutive days with a recorded temperature above t. Also, analyze the complexity of HOT-DAYS(A, t). *time: 5'*

(b) Now imagine that a particular analysis would call the HOT-DAYS algorithm several times with the same series A of temperature measurements (but with different temperature values) and therefore it would be more efficient to somehow index or precompute the results. To do that, write the following two algorithms:

- A preprocessing algorithm called HOT-DAYS-INIT(A) that takes the series of temperature measurements A and creates an auxiliary data structure X (an index of some sort).
- An algorithm called HOT-DAYS-FAST(X, t) that takes the index X and a temperature t and returns the maximum number of consecutive days with a temperature above t. HOT-DAYS-FAST must run in *sub-linear time* in the size of A.

Also, analyze the complexity of HOT-DAYS-INIT and HOT-DAYS-FAST. *time: 25'*

130. Consider the following decision problem: given a sequence A of numbers and given an integer k, return TRUE if A contains either an increasing or a decreasing subsequence of length k. The elements of the subsequence must maintain their order in A but do not have to be contiguous.

(a) Is the problem in NP? Write an algorithm that proves it is, or argue the opposite. *time: 10'*

(b) Is the problem in P? Write an algorithm that proves it is, or argue the opposite. *time: 20'*

131. Write an algorithm HEAP-DELETE(H, i) that, given a max-heap H, deletes the element at position i from H. *time: 10'*

132. Write an algorithm MAX-CLUSTER(A, d) that takes an array A of numbers (not necessarily integers) and a number d, and prints a maximal set of numbers in A that differ by at most d. The output can be given in any order. Your algorithm must have a complexity that is strictly better than $O(n^2)$. For example, with

$$A = \langle 7, 15, 16, 3, 10, 43, 8, 1, 29, 13, 4.5, 28 \rangle \quad d = 5$$

MAX-CLUSTER(A, d) would output $7, 3, 4.5, 8$ (or the same numbers in any other order) since those numbers differ by at most 5 and there is no larger set of numbers in A that differ by at most 5. Also, analyze the complexity of MAX-CLUSTER. *time: 20'*

133. Consider the following algorithm that takes a non-empty array of numbers

```
ALGO-X(A)
 1  B = make a copy of A
 2  i = 1
 3  while i ≤ B.length
 4        j = i + 1
 5        while j ≤ B.length
 6              if B[j] == B[i]
 7                    i = i + 1
 8                    swap B[i] ↔ B[j]
 9              j = j + 1
10        i = i + 1
11  q = B[1]
12  n = 1
13  m = 1
14  for i = 2 to B.length
15        if B[i] == q
16              n = n + 1
17              if n > m
18                    m = m + 1
19        else q = B[i]
20              n = 1
21  return m
```

(a) Briefly explain what ALGO-X does, and analyze the complexity of ALGO-X. *time: 10'*

(b) Write an algorithm called BETTER-ALGO-X that is functionally identical to ALGO-X but with a strictly better complexity. Analyze the complexity of BETTER-ALGO-X. *time: 10'*

134. Write the *heap-sort* algorithm and then illustrate how *heap-sort* processes the following array in-place:

$$A = \langle 33, 28, 23, 48, 32, 46, 40, 12, 21, 41, 14, 37, 38, 0, 25 \rangle$$

In particular, show the content of the array at each main iteration of the algorithm. *time: 20'*

135. Write an algorithm BST-PRINT-LONGEST-PATH(T) that, given a binary search tree T, outputs the sequence of nodes (values) of the path from the root to any node of maximal depth. Also, analyze the complexity of BST-PRINT-LONGEST-PATH. *time: 30'*

136. Consider insertion in a binary search tree.

(a) Write a valid insertion algorithm BST-INSERT. *time: 10'*

(b) Illustrate how BST-INSERT works by drawing the binary search tree resulting from the insertion of the following keys in this order:

$$33, 28, 23, 48, 32, 46, 40, 12, 21, 41, 14, 37, 38, 0, 25$$

Also, if the resulting tree is not already of minimal depth, write an alternative insertion order that would result in a tree of minimal depth. *time: 10'*

(c) Write an algorithm BEST-BST-INSERT-ORDER(A) that takes an array of numbers A and outputs the elements of A in an order that, if used with BST-INSERT would lead to a binary search tree of minimal depth. *time: 10'*

137. Write an algorithm called FIND-NEGATIVE-CYCLE that, given a weighted directed graph $G = (V, E)$, with weight function $w : E \to \mathbb{R}$, finds and outputs a negative-weight cycle in G if one such cycle exists. Also, analyze the complexity of FIND-NEGATIVE-CYCLE. *time: 20'*

138. Consider a text composed of n lines of up to 80 characters each. The text is stored in an array T where each line $T[i]$ is an array of characters containing words separated by a single space.

    (a) Write an algorithm SORT-LINES-BY-WORD-COUNT(T) that, with a worst-case complexity of $O(n)$, sorts the lines in T in non-decreasing order of the number of words in the line. (**Hint:** lines have at most 80 characters, so the number of words in a line is also limited.) *time: 20'*

    (b) If you did not already do that for exercise 138a, write an *in-place* variant of the SORT-LINES-BY-WORD-COUNT algorithm. This algorithm, called SORT-LINES-BY-WORD-COUNT-IN-PLACE, must also have a $O(n)$ complexity to sort the set of lines, and may use only a constant amount of extra space to do that. *time: 20'*

139. Consider a weighted undirected graph $G = (V, E)$ representing a group of programmers and their affinity for team work, such that the weight $w(e)$ of an edge $e = (u, v)$ is a number representing the ability of programmers u and v to work together on the same project. Write an algorithm BEST-TEAM-OF-THREE that outputs the best team of three programmers. The value of a team is considered to be the lowest affinity level between any two members of the team. So, the best team is the group of programmers for which the lowest affinity level between members of the group is maximal. *time: 20'*

140. Write an algorithm MAX-NON-ADJACENT-SEQ-WEIGHT(A) that, given a sequence of numbers $A = \langle a_1, a_2, ..., a_n \rangle$, computes, with worst-case complexity $O(n)$, the maximal weight of any sub-sequence of non-adjacent elements in A. A sub-sequence of non-adjacent elements may include a_i or a_{i+1} but not both, for all i. For example, with $A = \langle 2, 9, 6, 2, 6, 8, 5 \rangle$, MAX-NON-ADJACENT-SEQ-WEIGHT(A) should return 20. (**Hint:** use a dynamic programming algorithm that scans the input sequence once.) *time: 20'*

141. Consider a trie rooted at node T that represents a set of character strings. For simplicity, assume that characters are from the Roman alphabet and that the letters of the alphabet are encoded with numeric values between 1 and 26. Write an algorithm PRINT-TRIE(T) that prints all the strings stored in the trie. *time: 20'*

142. Write an algorithm called PRINT-IN-THREE-COLUMNS(A) that takes an array of words A and prints all the words in A, in the given order left-to-right and top-to-bottom, such that the words are left-aligned in three columns. Words must be separated by at least one space horizontally, but in order to align words, the algorithm might have to print more spaces between words. For example, if A contains the words *exam, algorithms, asymptotic, complexity, graph, greedy, lugano, np, quicksort, retake, september,* then the output should be

```
exam       algorithms asymptotic
complexity graph      greedy
lugano     np         quicksort
retake     september
```

    time: 20'

143. Consider a binary search tree.

    (a) Write an algorithm BST-MEDIAN(T) that takes the root T of a binary search tree and returns the median element contained in the tree. Also analyze the complexity of BST-MEDIAN(T). Can you do better? *time: 10'*

    (b) Assume now that the tree is balanced and also that each node t has an attribute $t.weight$ corresponding to the total number of nodes in the subtree rooted at t (including t itself). Write an algorithm BETTER-BST-MEDIAN(T) that improves on the complexity of BST-MEDIAN. Analyze the complexity of BETTER-BST-MEDIAN. *time: 10'*

144. Consider the following decision problem. Given a set of strings S, a number w, and a number k, output *YES* when there are at least k strings in S that share a common sub-string of length w, or *NO* otherwise. For example, if S contains the strings *exam, algorithms, asymptotic, complexity, graph, greedy, lugano, np, quicksort, retake, september, theory, practice, programming, math, art, truth, justice,* with $w = 2$ and $k = 3$ the output should be *YES*, because the 3 strings *graph, greedy,* and *programming* share a common substring "gr" of length 2. The output should also be *YES* for $w = 3$ and $k = 3$ and for $w = 2$ and $k = 4$, but it should be *NO* for $w = 3$ and $k = 4$.

(a) Is this problem in NP? Write an algorithm that proves it is, or argue that it is not. *time: 10'*

(b) Is this problem in P? Write an algorithm that proves it is, or argue that it is not. (**Hint:** a string of length ℓ has $O(\ell^2)$ sub-strings of any length.) *time: 20'*

145. Consider the following sorting problem: you must reorder the elements of an array of numbers in-place so that odd numbers are in odd positions while even numbers are in even positions. If there are more even elements than odd ones in A (or vice-versa) then those additional elements will be grouped at the end of the array. For example, with an initial sequence

$$A = \langle 50, 47, 92, 78, 76, 7, 60, 36, 59, 30, 50, 43 \rangle$$

the result could be this:

$$A = \langle 47, 50, 7, 78, 59, 76, 43, 92, 36, 60, 30, 50 \rangle$$

(a) Write an algorithm called ALTERNATE-EVEN-ODD(A) that sorts A in place as explained above. Also, analyze the complexity of ALTERNATE-EVEN-ODD. (You might want to consider problem 145b before you start solving this problem.) *time: 20'*

(b) If you have not done so already, write a variant of ALTERNATE-EVEN-ODD that runs in $O(n)$ steps for an array A of n elements. *time: 10'*

146. Write an algorithm called FOUR-CYCLE(G) that takes a directed graph represented with its adjacency matrix G, and that returns *true* if and only if G contains a 4-cycle. A 4-cycle is a sequence of four distinct vertexes a, b, c, d such that there is an arc from a to b, from b to c, from c to d, and from d to a. Also, analyze the complexity of FOUR-CYCLE(G). *time: 20'*