Algorithms and Data Structures II.
Exercises

8 Sorting in Linear Time

8.2-1 Using the lecture as a model, illustrate the operation of COUNTING-SORT on the array
`A = < 31; 21; 30; 22; 01; 23 >`
of base-4 numbers with respect to the least significant digits. Then take the result and sort it with respect to the most significant digits.

8.3-1 Using the lecture as a model, illustrate the operation of RADIX-SORT on the array
`A = < 013; 200; 010; 321; 213; 201 >`
of base-4 numbers.

8.3-2 Which of the following sorting algorithms are stable: insertion sort, merge sort, heap-sort, and quick-sort?

8.3-3 How to modify RADIX-SORT, in order to sort strings?

8.4-1 Using the lecture as a model, illustrate the operation of BUCKET-SORT on the array
`A = < .79;.13;.16;.64;.39;.20;.89;.53;.71;.42 >`.

8.4-2 Explain why the worst-case running time for bucket sort is $\Theta(n^2)$. What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time $O(n \lg n)$?

11 Hash Tables

11.1-1 Suppose that a dynamic set S is represented by a direct-address table T of length m. Describe a procedure that finds the maximum element of S. What is the worst-case performance of your procedure?

11.1-2 A bit vector is simply an array of bits (0s and 1s). A bit vector of length m takes much less space than an array of m pointers. Describe how to use a bit vector to represent a dynamic set of distinct elements with no satellite data. Dictionary operations should run in $O(1)$ time.

11.1-3 Suggest how to implement a direct-address table in which the keys of stored elements do not need to be distinct and the elements can have satellite data. All three dictionary operations (INSERT, DELETE, and SEARCH) should run in $O(1)$ time. (Do not forget that DELETE takes as an argument a pointer to an object to be deleted, not a key.)

11.1-4* We wish to implement a dictionary by using direct addressing on a huge array. At the start, the array entries may contain garbage, and initializing the entire array is impractical because of its size. Describe a scheme for implementing a direct address dictionary on a huge array. Each stored object should use $O(1)$ space; the operations SEARCH, INSERT, and DELETE should take $O(1)$ time each; and initializing the data structure should take $O(1)$ time. (Hint: Use an additional array, treated somewhat like a stack whose size is the number of keys actually stored in the dictionary, to help determine whether a given entry in the huge array is valid or not.)

11.2-2 Demonstrate what happens when we insert the keys 5; 28; 19; 15; 20; 33; 12; 17; 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \bmod 9$.

11.2-3x Given a hash table T with m slots that stores n elements. (The load factor $\alpha = n/m$.) Collisions are resolved by chaining. Write detailed pseudocode for the operations SEARCH, INSERT, and DELETE, if the linked lists referred to by the pointers in the slots are (a) doubly linked, and if (b) they are singly linked. (The lists are acyclic, without sentinels.) Each operation should run in $O(1 + \alpha)$ expected time.

11.2-4* Suggest how to allocate and deallocate storage for elements within the hash table itself by linking all unused slots into a free list. Assume that one slot can store a flag and either one element plus a pointer or two pointers. All dictionary and free-list operations should run in $O(1)$ expected time (except the initialization of the free list, which takes $\Theta(m)$ time, where m is the size of the hash table). Does the free list need to be doubly linked, or does a singly linked free list suffice? (We suppose that the lists of the reserved items are doubly linked, acyclic, without sentinels.)

11.2-5 Suppose that we are storing a set of n keys into a hash table of size m. Show that if the keys are drawn from a universe U with $|U| > nm$, then U has a subset of size n consisting of keys that all hash to the same slot, so that the worst-case searching time for hashing with chaining is $\Theta(n)$.

11.3-1 Suppose we wish to search a linked list of length n, where each element contains a key k along with a hash value $h(k)$. Each key is a long character string. How might we take advantage of the hash values when searching the list for an element with a given key?

11.3-2 Suppose that we hash a string of r characters into m slots by treating it as a radix-128 number and then using the division method. We can easily represent the number m as a 32-bit computer word, but the string of r characters, treated as a radix-128 number, takes many words. How can we apply the division method to compute the hash value of the character string without using more than a constant number of words of storage outside the string itself?

11.3-4 Consider a hash table of size $m = 1000$ and a corresponding hash function $h(k) = \lfloor m(kA \bmod 1) \rfloor$ for $A = (\sqrt{5}-1)/2$. Compute the locations to which the keys 61, 62, 63, 64, and 65 are mapped.

11.4-1

We have a hash table of length $m = 11$. We use open addressing and double hashing with the auxiliary hash functions $h_1(k) = k \bmod m$, and $h_2(k) = 1 + (k \bmod (m-1))$.

For each of the following operations show the probing sequence, and the resulting hash table.

Insert subsequently the keys 10; 22; 31; 4; 15; 28; 17; 88; 59 into the hash table. Then delete key 17. Next search for 18, and 59. Finally insert 18.

Illustrate the above operations with the auxiliary hash function $h'(k) = k \bmod m$ using linear probing, and using quadratic probing with $c1 = 1$ and $c2 = 3$.

11.4-2 Suppose we have the hash table $T[0..m-1]$, the hash function $h(k, i)$ ($i \in 0..m-1$), and we use open addressing. Write pseudo-code for HASH-SEARCH, HASH-DELETE, and HASH-INSERT. Do not forget to handle the special values NIL (for empty slots), and DEL (for deleted slots).

22 Elementary Graph Algorithms

22.1 Representations of graphs

22.1-1 Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?

22.1-2 Give an adjacency-list representation for a complete binary tree on 7 vertexes. Give an equivalent adjacency-matrix representation. Assume that vertexes are numbered from 1 to 7 as in a binary heap.

22.1-3 The transpose of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. Thus, G^T is G with all its edges reversed. Describe efficient algorithms for computing G^T from G, for both the adjacency-list and adjacency-matrix representations of G. Analyze the running times of your algorithms.

Note: A multi-graph is like an undirected graph, but it can have both multiple edges between vertexes and self-loops. In the big-O notation, a set stands for its size, for example: $O(V + E) = O(|V| + |E|)$.

22.1-4 Given an adjacency-list representation of a multi-graph $G = (V, E)$, describe an $O(V+E)$-time algorithm to compute the adjacency-list representation of the "equivalent" undirected graph $G' = (V, E')$, where E' consists of the edges in E with all multiple edges between two vertexes replaced by a single edge and with all self-loops removed.

22.1-5 The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$ such that $(u, v) \in E^2$ if and only G contains a path with at most two edges between u and v. Describe efficient algorithms for computing G^2 from G for both the adjacency-list and adjacency-matrix representations of G. Analyze the running times of your algorithms.

22.1-6* Most graph algorithms that take an adjacency-matrix representation as input require time $\Omega(V^2)$, but there are some exceptions. Show how to determine whether a directed graph G contains a *universal sink* — a vertex with in-degree $|V| - 1$ and out-degree 0 — in time $O(V)$, given an adjacency matrix for G.

22.2 Breadth-first search

Note: Each graph-searching algorithm selects a vertex in an indeterministic way at some points. When you illustrate its run, you should always select the vertex with the smallest index in such cases.

22.2-1 Present breadth-first search on the directed graphs below[1], using the given vertex as source. Using the lecture as a model, illustrate the run of the algorithm. For each vertex, show the discovery/finishing times, the d and π values; and for all the times, show the transformations of the queue. Draw the breadth-first tree represented by the final π values.

22.2-1a Source vertex: 3

$1 \to 2;\ 4.$	$2 \to 5.$	$3 \to 5;\ 6.$
$4 \to 2.$	$5 \to 4.$	$6.$

22.2-1b Source vertex: 5

$1 \to 4.$	$2 \to 1;\ 3;\ 5.$	$3.$
$4 \to 2;\ 5.$	$5 \to 3;\ 4.$	$6 \to 3;\ 5.$

22.2-2 Illustrate the run of the breadth-first search on the undirected graph below[2], using vertex 4 as the source.

$1 - 2;\ 5.$	$2 - 1;\ 6.$	$3 - 4;\ 6;\ 7.$	$4 - 3;\ 7;\ 8.$
$5 - 1.$	$6 - 2;\ 3;\ 7.$	$7 - 3;\ 4;\ 6;\ 8.$	$8 - 4;\ 7.$

22.2-3 Show that using a single bit to store each vertex color suffices by arguing that the BFS procedure would produce the same result provided that the gray and black nodes are not distinguished.

22.2-4 What is the running time of BFS if we represent its input graph by an adjacency matrix and modify the algorithm to handle this form of input?

22.2-5 Argue that in a breadth-first search, the value $u.d$ assigned to a vertex u is independent of the order in which the vertexes appear in each adjacency list. Using the graph in exercise 22.2-2 as an example, show that the breadth-first tree computed by BFS can depend on the ordering within adjacency lists.

[1] $u \to v_1; \dots v_n.$ means that the graph has the directed edges $(u, v_1), \dots (u, v_n)$.
[2] $u - v_1; \dots v_n.$ means that the graph has the undirected edges $(u, v_1), \dots (u, v_n)$.

22.3 Depth-first search

22.3-1 Make a 3-by-3 chart with row and column labels WHITE, GRAY, and BLACK. In each cell (i, j), indicate whether, at any point during a depth-first search of a directed graph, there can be an edge from a vertex of color i to a vertex of color j . For each possible edge, indicate what edge types it can be. Make a second such chart for depth-first search of an undirected graph.

22.3-2 Show how depth-first search works on the graph below. Assume that in the indeterministic cases the DFS procedure considers the vertexes in alphabetical order. Show the discovery and finishing times for each vertex, and show the classification of each edge.

q → s; t; w.	r → u; y.	s → v.	t → x; y.
u → y.	v → w.	w → s.	x → z.
y → q.	z → x.		

22.3-5a Argue that when DFS discovers an edge (u, v), v is
- white, iff (u, v) is a tree edge
- gray, iff (u, v) is a back edge
- black, iff (u, v) is a forward or cross edge
with respect to the depth-first forest computed by the DFS.

22.3-5b How can we distinguish the forward and cross edges with respect to the discovery times of u and v?

22.3-5c How can we classify the edges (as tree, back, forward, and cross edges) with respect to only the discovery and finishing times of the vertexes?

22.3-7 Rewrite the procedure DFS, using a stack to eliminate recursion.

22.3-10 Modify the pseudo-code for depth-first search so that it prints out every edge in the directed graph G, together with its type.

22.3-11 Explain how a vertex u of a directed graph G can end up in a depth-first tree containing only u, even though u has both incoming and outgoing edges in G.

22.3-12 Show that we can use a depth-first search of an undirected graph G to identify the connected components of G, and that the depth-first forest contains as many trees as G has connected components. More precisely, show how to modify depth-first search so that it assigns to each vertex v an integer label $v.cc$ between 1 and k, where k is the number of connected components of G, such that $u.cc = v.cc$ if and only if u and v are in the same connected component.

22.4 Topological sort

22.4-1 Show the ordering of vertexes produced by TOPOLOGICAL-SORT when it is run on the DAG below, under the assumption of Exercise 22.3-2.

q → s; t; w.	r → u; y.	s → v.	t → x; y.
u → y.	v → w.	w.	x → y; z.
y.	z → v.		

22.4-2 Give a linear-time ($O(V+E)$) algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two vertexes s and t, and returns the number of simple paths from s to t in G. For example, the directed acyclic graph below contains exactly four simple paths from vertex p to vertex u: pqu, pqsvu, prsvu, and prtvu. (Your algorithm needs only to count the simple paths, not list them.)

p → q; r.	q → s; u.	r → s; t.	s → v.
t → v.	u.	v → u.	

22.4-3 Give an algorithm that determines whether or not a given undirected graph $G = (V, E)$ contains a cycle. Your algorithm should run in $O(V)$ time, independent of $|E|$.

22.4-5 Another way to perform topological sorting on a directed acyclic graph $G = (V, E)$ is to repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges from the graph. Explain how to implement this idea so that it runs in time $O(V + E)$. What happens to this algorithm if G has cycles?

(Hint: For each $u \in V$: $u.ind :=$ the in-degree of u. $H := \{u \in V : u.ind = 0\}$. In a loop, while H is not empty, remove an element of H, output it, decrease the ind attributes of its adjacent vertexes by one, and put those with zero ind value into H. })

23 Minimum Spanning Trees

23.1 Kruskal's algorithm

23.1.1 Show how the spanning forest of the graph is transformed while processing each edge of the graph in exercise 23.2.1 using Kruskal's algorithm.

23.2 Prim's algorithm

23.2.1 Show the d and π values that result from running Prim's algorithm on the undirected graph below[3], using vertex 2 as source. Using the lecture as a model, illustrate the run of the algorithm. Show the initial d and π values. Then line by line show the vertex selected for expansion, and the d and π values of the vertexes after the expansion[4]. Draw the minimum spanning tree represented by the final π and d values[5].

1 – 2, 2; 5, 1.	2 – 1, 2; 6, 0.
3 – 4, 4; 6, 1; 7, 1.	4 – 3, 4; 7, 3; 8, 2.
5 – 1, 1; 6, 1.	6 – 2, 0; 3, 1; 5, 1; 7, 2.
7 – 3, 1; 4, 3; 6, 2; 8, 1.	8 – 4, 2; 7, 1.

23.2-2a Suppose that we represent the graph $G = (V, E)$ as an adjacency matrix. Give a simple implementation of Prim's algorithm for this case that runs in $O(V^2)$ time.

23.2-2b Suppose that we represent the graph $G = (V, E)$ as an adjacency list. Give a simple implementation of Prim's algorithm for this case that runs in $O(V^2)$ time.

23.2-2c* Suppose that we represent the graph $G = (V, E)$ as an adjacency list. Give a sophisticated implementation of Prim's algorithm for this case that runs in $O((V + E) \lg V)$ time.

Hint: Use a binary minimum heap to represent the priority queue of the vertexes (organized according to the d values of the vertexes) . When we decrease $v.d$ for a vertex v, it must be compared with its parent in the heap

[3] $u - v_1, w_1; \ldots v_n, w_n$. means that the graph has the undirected edges $(u, v_1), \ldots (u, v_n)$ with weights $w_1, \ldots w_n$.

[4] Given an undirected graph $G = (V, E)$, and a subtree $T = (U, A)$, where $U \subset V$, $A \subset U \times U$, and $A \subset E$. Let $v \in V \setminus U$. The attribute $v.d$ is the minimum weight of any edge connecting v to a vertex in the tree; by convention, $v.d = \infty$ if there is no such edge ($v.d$ is also called $v.key$).

[5] In the output of Prim's algorithm $v.d$ is the weight of the edge $(v.\pi, v)$ in the minimum spanning tree, except for the root r of the tree ($r.d = 0$).

(with respect to their d attributes), and they possibly must be swapped, recursively. Therefore, we need an indexing array in order to know the place of each vertex in the heap.

24 Single-Source Shortest Paths

24.1 Queue-based Bellman-Ford algorithm

<http://algs4.cs.princeton.edu/44sp/>
<https://en.wikipedia.org/wiki/Shortest_Path_Faster_Algorithm>

(The **Queue-based Bellman-Ford algorithm** is also known as **Tarjan's breadth-first scanning algorithm**, and **Shortest Path Faster Algorithm (SPFA)**.)

24.1-1 Run the Queue-based Bellman-Ford algorithm on the directed graph below[6], using vertex z as the source.

Using the lecture as a model, illustrate the run of the algorithm.

Show the initial d and π values, and the initial queue. Show the numbering of each pass of the algorithm. During each pass, when expanding (scanning) a vertex makes change, show the **new** d and π values of its successor vertices, and the queue after the expansion. Provided that during the expansion of a vertex more vertices are put into the queue, they are put there in alphabetical order (or in the order of their indexes). Draw the shortest-paths tree represented by the final π and d values.

Now, change the weight of edge (z, x) to 4 and run the algorithm again, using s as the source.

$s \to t, 6; y, 7.$ $\quad t \to x, 5; y, 8; z, -4.$
$x \to t, -2.$ $\quad y \to x, -3; z, 9.$
$z \to s, 2; x, 7.$

24.1.2a Suppose that we represent the graph $G = (V, E)$ as an adjacency matrix. Give a simple implementation of the Queue-based Bellman-Ford algorithm for this case that runs in $O(V^3)$ time.

24.1.2b Suppose that we represent the graph $G = (V, E)$ as an adjacency list. Give a simple implementation of the Queue-based Bellman-Ford algorithm for this case that runs in $O(VE)$ time.

24.2 Single-source shortest paths in directed acyclic graphs

24.2-1 Run DAG-SHORTEST-PATHS on the directed graph below, using vertex r as the source.

$r \to s, 5; t, 3.$ $\quad s \to t, 2; x, 6.$ $\quad t \to x, 7; y, 4; z, 2.$
$x \to x, -1; z, 1.$ $\quad y \to z, -2.$ $\quad z.$

[6] $u \to v_1, w_1; \ldots v_n, w_n.$ means that the graph has the directed edges $(u, v_1), \ldots (u, v_n)$ with weights $w_1, \ldots w_n$.

24.3 Dijkstra's algorithm

24.3-1 Show the d and π values that result from running Dijkstra's algorithm on the directed graph below, using vertex z as source.

Run Dijkstra's algorithm on the directed graph below, first using vertex s as the source and then using vertex z as the source.

Using the lecture as a model, illustrate the run of the algorithm. When the algorithm is indeterministic, prefer the vertex with lower index.

Show the initial d and π values. Then line by line show the vertex selected for expansion, and the new d and π values of the vertexes after the expansion. Draw the shortest-paths tree represented by the final π and d values.

$s \to t, 3; y, 6.$ $\quad t \to x, 8; y, 2.$ $\quad x \to z, 2.$
$y \to t, 1; x, 4; z, 6.$ $\quad z \to s, 3; x, 7.$

24.3-2 Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces incorrect answers.

24.3.3a Suppose that we represent the graph $G = (V, E)$ as an adjacency matrix. Give a simple implementation of Dijkstra's algorithm for this case that runs in $O(V^2)$ time.

24.3.3b Suppose that we represent the graph $G = (V, E)$ as an adjacency list. Give a simple implementation of Dijkstra's algorithm for this case that runs in $O(V^2)$ time.

24.3.3c* Suppose that we represent the graph $G = (V, E)$ as an adjacency list. Give a sophisticated implementation of Dijkstra's algorithm for this case that runs in $O((V + E) \lg V)$ time.

Hint: Use a binary minimum heap to represent the priority queue of the vertexes (organized according to the d values of the vertexes) . When we decrease $v.d$ for a vertex v, it must be compared with its parent in the heap (with respect to their d attributes), and they possibly must be swapped, recursively. Therefore, we need an indexing array in order to know the place of each vertex in the heap.

25 All-Pairs Shortest Paths

25.2 The Floyd-Warshall algorithm

25.2-1 Run the Floyd-Warshall algorithm on the weighted graph below. Show the matrix pairs $(D^{(0)}, \Pi^{(0)}), \ldots, (D^{(4)}, \Pi^{(4)})$.

	1	2	3	4
1	0	5	3	1
2	5	0	1	∞
3	3	1	0	1
4	1	∞	1	0

25.2.2 Run Warshall's transitive-closure algorithm on the unweighted, directed graph below. Show the matrices $T^{(0)}, \ldots, T^{(4)}$.

	1	2	3	4
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	1	0	0	0

32 String Matching

32.1 The naive string-matching algorithm

32.1-1 Show the comparisons the naive string matcher makes for the pattern $P = 0001$ in the text $T = 000010001010001$.

32.1-2 Suppose that all characters in the pattern P are different. Show how to accelerate NAIVE-STRING-MATCHER to run in time $O(n)$ on an n-character text T .

32.2 The Rabin-Karp algorithm

32.2-1 Working modulo $q = 11$, how many spurious (i.e. false) hits does the Rabin-Karp matcher encounter in the text $T = 3141592653589793$ when looking for the pattern $P = 26$?

32.2-2 How would you extend the Rabin-Karp method to the problem of searching a text string for an occurrence of any one of a given set of k patterns? Start by assuming that all k patterns have the same length. Then generalize your solution to allow the patterns to have different lengths.

32.2-3 Show how to extend the Rabin-Karp method to handle the problem of looking for a given $m \times m$ pattern in an $n \times n$ array of characters $(1 \leq m \leq n)$. (The pattern may be shifted vertically and horizontally, but it may not be rotated.)

32.4 The Knuth-Morris-Pratt algorithm

32.4-1 Compute $next[1..19]$ (also called $prefix$ or π function) for the pattern $ababbabbabbababbabb$.

32.4.2a Compute $next[1..4]$ for the pattern $P = 0001$. Show the comparisons the Knuth-Morris-Pratt string matcher makes for this pattern in the text $T = 000010001010001$.

32.4.2b Compute $next[1..5]$ for the pattern $P = abaab$. Show the comparisons the Knuth-Morris-Pratt string matcher makes for this pattern in the text $T = aaababaabaababaab$.

32.4.2c Compute $next[1..6]$ for the pattern $P = aabaab$. Show the comparisons the Knuth-Morris-Pratt string matcher makes for this pattern in the text $T = aaabaabaabaababaab$.

32.4.2d Compute $next[1..6]$ for the pattern $P = babbab$. Show the comparisons the Knuth-Morris-Pratt string matcher makes for this pattern in the text $T = ababbabbababbababbabb$.

32.4.2e Compute $next[1..7]$ for the pattern $P = ABABAKI$.
Show the comparisons the Knuth-Morris-Pratt string matcher makes for this pattern in the text
$T = BABALABABATIBABABAKI$.

32.4-3 Explain how to determine the occurrences of pattern P in the text T by examining $next[1..|PT|]$.

32.4-7 Give a linear-time algorithm to determine whether a text T is a cyclic rotation of another string T'. For example, *arc*, *rca*, and *car* are cyclic rotations of each other.

32.x The Quick Search algorithm

`<http://aszt.inf.elte.hu/~asvanyi/ds/Quick Searching.ppt>`

32.x.1 Compute $shift[0..1]$ for the pattern $P = 000$. Show the comparisons the Quick Search string matcher makes for this pattern in the text $T = 000010001010001$.

32.x.2 Compute $shift['A'..'F']$ for the pattern
$P = ABABACD$. Show the comparisons the Quick Search string matcher makes for this pattern in the text
$T = BABAEABABAFDBABABACD$.

32.x.3 Compute $shift['A','C','G','T']$ for the pattern
$P = GCAGAGAG$. Show the comparisons the Quick Search string matcher makes for this pattern in the text
$T = GCATCGCAGAGAGTATACAGTACG$.

DC Data Compression

DC.1 The Huffman coding

<http://en.wikipedia.org/wiki/Huffman_coding>

DC.1.1 We would like to compress the text

MATEKFELELETEMKETTESLETT

with Huffman coding. Draw the Huffman tree, give the Huffman code of each letter, the Huffman code of the text, and the length of the latest in bits. Show the letters of the original text in the Huffman code of it.

DC.1.2 Solve Exercise DC.1.1 with the following text.

EMESE MAI SMSE NEM NAIV MESE

DC.2 The Lempel–Ziv–Welch (LZW) algorithm

<http://en.wikipedia.org/wiki/Lempel-Ziv-Welch>

DC.2.1 We have compressed a text with the Lempel-Ziv-Welch algorithm. The text contains letters 'A', 'B', and 'C'. Their codes are 1, 2, and 3 in turn. While building the dictionary, for the code of each new word the first unused positive integer was selected. In this way we have received the following code.

1 2 4 3 5 6 9 7 1

Give the original text, and the complete dictionary.

DC.2.2 Solve Exercise DC.2.1 with the following LZW code.

1 2 4 3 5 8 1 10 11 1