Problems with linear or sublinear time complexity

1 Single scan with logarithmic memory

Problem 1: Leaders in an array

Write a program to print all the *leaders* in an array. An element is a leader if it is greater than all the elements to its right side. The rightmost element is always a leader. For example in the array 16, 17, 4, 3, 5, 2, the leaders are 17, 5 and 2. What is the fewest number of passes you need to make through the array?

Problem 2: Maximum difference in an array

Given an array A of integers of length n, find the maximum value of $A(i) - A(j)$ over all choices of indexes such that $j > i$.

Problem 3: Maximum contiguous sum

Use the divide-and-conquer approach to write an efficient recursive algorithm that finds the maximum sum in any contiguous sublist of a given list of n real (positive or negative) values. Analyse your algorithm, and show the results in order notation. Can you do better? Obtain a linear-time algorithm.

Problem 4: Maximum sum among nonadjacent subsequences

Find an efficient algorithm for the following problem:

We are given an array of real numbers $V[1..n]$. We wish to find a subset of array positions, $S \subseteq [1...n]$ that maximizes $\sum_{i \in S} V[i]$ subject to no two consecutive array positions being in S. For example, say $V = [10, 14, 12, 6, 13, 4]$, the best solution is to take elements $1, 3, 5$ to get a total of $10 + 12 + 13 = 35$. If instead, we try to take the 14 in position 2, we must exclude the 10 and 12 in positions 1 and 3, leaving us with the second best choice $2, 5$ giving a total of $14 + 13 = 27$.

Problem 5: Maximum contiguous rectangle

Use divide-and-conquer approach to write an efficient recursive algorithm that finds the maximum *area contiguous subsequence* in a given sequence of n nonnegative real values. Analyse your algorithm, and show the results in order notation. Can you do better? Obtain a linear-time algorithm.

The area of a contiguous subsequence is the product of the length of the subsequence and the minimum value in the subsequence.

Problem 6: Computing mode

The mode of a set of numbers is the number that occurs most frequently in the set. The set (4,6,2,4,3,1) has a mode of 4.

1. Give an efficient and correct algorithm to compute the mode of a set of n numbers;
2. Suppose we know that there is an (unknown) element that occurs $\lfloor n/2 \rfloor + 1$ times in the set. Give a worst-case linear-time algorithm to find the mode. For partial credit, your algorithm may run in expected linear time.

Problem 7: Frequent elements

Design an algorithm that, given a list of n elements in an array, finds all the elements that appear more than $n/3$ times in the list. The algorithm should run in linear time. n is a nonnegative integer.

You are expected to use comparisons and achieve linear time. You may not use hashing. Neither can you use excessive space. Linear space is sufficient. Moreover, you are expected to design a deterministic algorithm. You may also not use the standard linear-time deterministic selection algorithm. Your algorithm has to be more efficient (in terms of constant factors) than the standard linear-time deterministic selection algorithm.

Problem 8: Testing chips

Professor Diogenes has supposedly identical VLSI chips that in principle are capable of testing each other. The professor's test jig accommodates two chips at a time. When the jig is loaded, each chip tests the other and reports wheter it is good or bad. A good chip always reports accurately whether the other chip is good or bad, but the answer of a bad chip cannot be trusted. Thus, the four possible outcomes of a test are as follows:

Chip A says	Chip B Says	Conclusion
B is good	A is good	both are good, or both are bad
B is good	A is bad	at least one is bad
B is bad	A is good	at least one is bad
B is bad	A is bad	at least one is bad

1. Show that if more than $n/2$ chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.
2. Consider the problem of finding a single good chip from among n chips, assuming that more than $n/2$ of the chips are good. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.
3. Show that the good chips can be identified with $\Theta(n)$ (proportional to n) pairwise tests, assuming that more than $n/2$ of the chips are good. Give and solve the recurrence that describes the number of tests.

Problem 9: Horner's rule

Design an algorithm for efficiently evaluating the polynomial $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ at point x.

2 Single scan from both ends with logarithmic memory

Problem 10: Pair closest to a number

Given a sorted array and a number x, find a pair in array whose sum is closest to x.

3 Single scan of two or more lists with logarithmic memory

Problem 11: Sorted merge

You are given two sorted arrays, A and B, where A has a large enough buffer at the end of hold B. Develop an algorithm to merge B into A in sorted order.

Problem 12: Common elements in three arrays

Given three arrays sorted in non-decreasing order, print all common elements in these arrays.

4 Single scan with a stack

Problem 13: Balanced Parentheses

You are given a string consisting of right (}) and left ({) brackets. The string may or may not be balanced. You are allowed to make the following alterations to the string: change a left bracket to a right bracket or a right bracket to a left one. Balance the string with minimum number of changes.

Problem 14: Next greater element

Given an array, print the Next Greater Element (NGE) for every element. The Next Greater Element of an item x is the first greater element to its right in the array.

5 Single scan with linear space

Problem 15: Maximum difference in a matrix

Given an $n \times n$ matrix $M[i, j]$ of integers, find the maximum value of $M[c, d] - M[a, b]$ over all choices of indexes such that both $c > a$ and $d > b$.

6 Multiple scans

Problem 16: Finding Minimum and Maximum

Given $n > 1$ items and a two pan balance scale with no weights, determine the lightest and the heaviest items in $\lceil 3n/2 \rceil - 2$ weighings.

Problem 17: Computing mode

The mode of a set of numbers is the number that occurs most frequently in the set. The set (4,6,2,4,3,1) has a mode of 4.

1. Give an efficient and correct algorithm to compute the mode of a set of n numbers;
2. Suppose we know that there is an (unknown) element that occurs $\lfloor n/2 \rfloor + 1$ times in the set. Give a worst-case linear-time algorithm to find the mode. For partial credit, your algorithm may run in expected linear time.

Problem 18: Testing chips

Professor Diogenes has supposedly identical VLSI chips that in principle are capable of testing each other. The professor's test jig accommodates two chips at a time. When the jig is loaded, each chip tests the other and reports wheter it is good or bad. A good chip always reports accurately whether the other chip is good or bad, but the answer of a bad chip cannot be trusted. Thus, the four possible outcomes of a test are as follows:

Chip A says	Chip B Says	Conclusion
B is good	A is good	both are good, or both are bad
B is good	A is bad	at least one is bad
B is bad	A is good	at least one is bad
B is bad	A is bad	at least one is bad

1. Show that if more than $n/2$ chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.

2. Consider the problem of finding a single good chip from among n chips, assuming that more than $n/2$ of the chips are good. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.

3. Show that the good chips can be identified with $\Theta(n)$ (proportional to n) pairwise tests, assuming that more than $n/2$ of the chips are good. Give and solve the recurrence that describes the number of tests.

7 Order

Problem 19: Olay

Professor Olay is consulting for an oil company, which is planning a large pipeline running east to west through an oil field of n wells. From each well, a spur pipeline is to be connected directly to the main pipeline along a shortest path (either north or south). Given x and y coordinates of the wells, how should the professor pick the optimal location of the main pipeline (the one that minimizes the the total length of the spurs?) Show that the optimal location can be determined in linear time.

8 Graph exploration

Problem 20: Pond sizes

You have an integer matrix representing a plot of land, where the value at a location represents the height above sea level. A value of zero indicates water. A pond is a region of water connected vertically, horizontally, or diagonally. The size of a pond is the total number of connected water cells. Write a method to compute the sizes of all ponds in the matrix.

Problem 21: Bad intersections

Consider a city whose streets are defined by an $X \times Y$ grid. We are interested in walking from the upper left-hand corner of the grid to the lower right-hand corner.

Unfortunately, the city has bad neighborhoods, which are defined as intersections we do not want to walk in. We are given an $X \times Y$ matrix BAD, where $BAD[i, j] = "Yes"$ if and only if the intersection between streets i and j is somewhere we want to avoid.

1. Given an example of the contents of BAD such that there is no path across the grid avoiding bad neighborhoods.

2. Give an $O(XY)$ algorithm to find a path that avoids bad neighborhoods.

3. Give an $O(XY)$ algorithm to find the shortest path across the grid that avoids bad neighborhoods. You may assume that blocks are of equal length.

Problem 22: Bipartite graphs (DPV)

A *bipartite graph* is a graph $G = (V, E)$ whose vertices can be partitioned into two sets ($V = V_1 \cup V_2$ and $V_1 \cap V_2 = \varnothing$) such that there are no edges between vertices in the same set (for instance, if $u, v \in V_1$, then there is no edge between u and v).

1. Give a linear-time algorithm to determine whether an undirected graph is bipartite.

2. There are many other ways to formulate this property. For instance, an undirected graph is bipartite if and only if it can be colored with just two colors.

   Prove the following formulation: an undirected graph is bipartite if and only if it contains no cycles of odd length.

3. At most how many colors are needed to color in an undirected graph with exactly *one* odd-length cycle?

Problem 23: Bipartite maximum matching

Consider bipartite graphs of the following special form:

- the vertex set is $\{x_1, ..x_n\}$, $\{y_1, ...y_n\}$.
- if (x_i, y_j) is an edge, there are no edges of the form (x_u, y_v) with both $u > i$ and $v < j$.

A *matching* is a set of edges, no two of which are incident on the same vertex. Design an algorithm that finds a maximum cardinality matching in linear time.

Problem 24: Hamiltonian path (DPV)

Give a linear-time algorithm for the following task.
Input: A directed acyclic graph G
Question: Does G contain a directed path that touches every vertex exactly once?

9 Graphs

Problem 25: Pouring water (DPV)

We have 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. We are allowed one type of operation: pouring the contents of one container into another, stopping only when the source container is empty or the destination container is full. We want to know if there is a sequence of pourings that leaves exactly 2 pints in the 7- or 4-pint container.

1. Model this as a graph problem: give a precise definition of the graph involved and state the specific question about this graph that needs to be answered.
2. What algorithm should be applied to solve the problem?

Problem 26: Coupons

Consider the following "coupon collector" problem. There are different varieties of cereal, and each comes with a single coupon for a discount on another box of cereal, perhaps of another variety. You can use multiple coupons when purchasing a new box, up to getting the new box free, but you never get money back. You want to buy one box of each variety, for as little money as possible. Describe an efficient algorithm, which, given as input, for each variety, its price , the value of the enclosed coupon, and the brand for which the coupon gives a discount, computes an optimal order in which to buy the cereal.

Problem 27: Dijkstra with small weights (CLRS)

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \{0, 1, \ldots, W\}$ for some nonnegative integer W. Modify Dijkstra's algorithm to compute the shortest paths from a given source vertex s in $O(WV + E)$ time.

10 Miscellaneous

Problem 28: Number with odd frequency

You are given an array of positive integers. All numbers occur an even number of times, except one number which occurs an odd number of times. Find the number in $O(n)$ time and constant space.

Problem 29: Maximum product of three numbers

You are given an unsorted array of integers. You need to find the maximum product formed by multiplying three numbers in the array.

Problem 30: Random selection

Given an array A of integers with at least one negative element, return a negative element of A chosen uniformly at random.

Problem 31: Next smallest and largest integer

Given a positive integer x, print the next smallest and the next largest number that have the same number of 1's in their binary representation as x. If such numbers do not exist, indicate accordingly.

11 Problems with sublinear complexity

Problem 32: Searching in a rotated array

Given a sorted array of n elements that has been rotated an unknown number of times, develop an algorithm to find an element in the array. You may assume that the array was sorted originally in increasing order.

Problem 33: Sorted matrix search

Given an $m \times n$ matrix in which each row and column is sorted in ascending order, design an algorithm to find an element.

Problem 34: Count zeros in a matrix

Given an $n \times n$ binary matrix (elements in matrix can be either 1 or 0) where each row and column of the matrix is sorted in ascending order, count the number of zeros present in it. You are expected to solve this problem in time $O(n)$.

Problem 35: Celebrity identification

Given an $n \times n$ adjacency matrix, determine whether there exists an i such that all the entries in the i'th column (except for the i'th entry) are 1, and all the entries in the i'th row (except for the ij-th entry) are 0.

This problem is also referred to as a the celebrity identification problem. Among n persons, a *celebrity* is defined as someone who is known by everyone but does not know anyone.

12 Solved Problems

Problem 36: Computing mode

The mode of a set of numbers is the number that occurs most frequently in the set. The set (4,6,2,4,3,1) has a mode of 4.

1. Give an efficient and correct algorithm to compute the mode of a set of n numbers;
2. Suppose we know that there is an (unknown) element that occurs $\lfloor n/2 \rfloor + 1$ times in the set. Give a worst-case linear-time algorithm to find the mode. For partial credit, your algorithm may run in expected linear time.

Solution: Computing mode

Part 1

High-level description of the algorithm

A simple algorithm for this problem is to sort the n elements, and then traverse the sorted array keeping track of the most frequent element. Such a traversal can accurately keep track of the most frequent element in linear time, since identical elements appear contiguously in a sorted list. The following expands upon this high-level description.

We maintain a counter to compute the length of the current contiguous block of identical elements. We also maintain the most frequent element (together with its frequency) among the elements that occurred prior to the current block. As long as the next element is the same as the current element, we increment the counter. If the next element is distinct from the current element, we compare the counter to the frequency of the most frequent element and update the most frequent element and its frequency accordingly. The counter is then set to one, the current element is set to the next element, and the process continues from the next element onward. Initially, the current element is set to the first element of the sorted array and the traversal starts with the first element of the array. At the end of the traversal, we output the most frequent element.

Pseudocode

Input: an array of numbers S indexed from 1 to n.
Output: a number in the array with the highest frequency.

```
FindMode($S$,$n$)

 1. Mergesort($S$,$n$)
 2. current = $S[1]$
 3. mode = current
 4. count = 1
 5. maxcount = 0
 6. for $i = 2$ to $n$
 7.    if ($S[i]$ = current)
 8.       count = count +1
 9.    else
10.       if (count > maxcount)
11.          mode = current
12.          maxcount = count
13.       endif
14.       count = 1; current = $S[i]$
15.    endif
16. endloop
17. return mode
```

Correctness

The algorithm works because after the set is sorted identical numbers will be adjacent.

Complexity

This algorithm's worst-case running time is $\Theta(n \log n)$ due to mergesort in step 1. Traversing the scanned array (the rest of the steps in the psuedocode) takes linear time.

Part 2

High-level ideas

Let us call the element that occurs at least $\lfloor n/2 \rfloor + 1$ times the majority element. The basic idea is that if an array A containing n elements has a majority element a, and if we remove any two distinct elements from the array, then the array still contains a majority element and it is the same majority element as before. Indeed, if the number of occurrences of a in A is at least $\lfloor n/2 \rfloor + 1$ and if A' is the array obtained by removing any two distinct elements from A, then the number of occurrences of a in A' is at least $\lfloor n/2 \rfloor + 1 - 1 = \lfloor (n-2)/2 \rfloor + 1$.

Our basic operation is to identify and remove a pair of distinct elements. Let us call this operation distinct-pair-removal. We perform a sequence of distinct-pair-removal operations until no such operation can be performed. At this point, either no elements are remaining in the list or all the remaining elements are the same. We call such a list reduced. We represent a reduced list (if it is non-empty) by its unique remaining element and its count. If a reduced list is empty, we set the count to zero. Depending on the sequence of removal operations, we may get different reduced lists. However, when a list has a majority element, the reduced list is non-empty and the remaining element is the the majority element. We use these observations to design two different algorithms to find the mode, given that the mode occurs at least $\lfloor n/2 \rfloor + 1$ times.

High-level description of Algorithm 1

We scan the list from left to right starting with the first element in the list, while performing distinct-pair-removal operations to maintain a reduced list. Initially, the list is empty and the count is zero. We process the next element as follows:

1. If the reduced list is empty, the reduced list is set to next element with count equal to one.
2. If the reduced list is non-empty and if the next element is the same as the remaining element in the reduced list, then we increment the count by one.
3. If the reduced list is non-empty and if the next element is different from the remaining element, we decrement the count by one.

In all cases, we continue the process until we no longer have any elements, at which point we output the remaining element.

As an example, if $A = [1; 1; 2; 3; 1; 2; 1]$, and if we represent the state of the algorithm using the triplet (remaining element in the reduced list; count; index of the next element in the list), then the algorithm would proceed as: $(-; 0; 1); (1; 1; 2); (1; 2; 3); (1; 1; 4); (-; 0; 5); (1; 1; 6); (-; 0; 7); (1; 1; 8)$ and output 1, which is the majority.

Pseudocode

Input: An array of numbers S indexed from 1 to n. Some element appears at least $\lfloor n/2 \rfloor + 1$ times in S.
Output: the number in the array with the highest frequency.

```
FindMode($S$,$n$)

1. counter = 0
2. for $i = 1$ to $n$
3.    if (counter = 0)
```

```
4.        candidate = $S[i]$
5.     endif
6.     if (candidate = $S[i]$)
7.        counter = counter +1
8.     else
9.        counter = counter - 1
10.    endif
11. endloop
12. return candidate
```

Complexity

We spend a constant time for each element as we only compare it to one value and increment or decrement the counter. The run time is therefore $\Theta(n)$.

High-level description of Algorithm 2

Let n be the length of the input list A. We partition A into disjoint pairs of adjacent elements so that each element in the odd position is paired with its adjacent element in the even position. We assume that the first element in the list is in position 1. If both the elements in a pair are different, we discard the pair from the list. Let A' be the resulting list. Let n' be the length of A'. From the previous discussion, it is clear that A' has the same majority element as that of A. If n is even, then n' and A' can be partitioned into pairs of identical elements. We now discard one element from each pair of identical elements to get the list A''. We claim that the list A'' also has a majority element and it is the same as that of A'. Indeed, since the majority element must appear at least $\lfloor n'/2 \rfloor + 1$ times in A', it must appear in at least $\lceil (n'/2+1)/2 \rceil$ pairs of identical elements. However, $\lceil (n'/2+1)/2 \rceil = \lceil n'/4 + 1/2 \rceil = \lfloor n'/4 \rfloor + 1 = \lfloor (n'/2)/2 \rfloor + 1$. Therefore, A'' contains the same majority element so we proceed to recurse on A''.

If n is odd, there will be a left-over element which is not paired up with any other element. In this case, the list A' can be viewed as a collection of pairs of identical elements and a left-over element. If we simply keep one element from each pair of identical elements, we may have not the same majority element in the resulting list. We need a more careful analysis to determine how to deal with the left-over element. Let m be the number of pairs of identical elements in A'. We have $n' = 2m + 1$ and the majority element appears at least $m + 1$ times in A'.

Case 1 $m = 2k + 1$

In this case, the majority elements occurs in at least $k + 1$ pairs of identical elements since the majority element appears at least $m + 1 = 2k + 2$ times in A'. If we select one element from each of the pairs and discard the left-over element, then the resulting list A'' contains $2k + 1$ elements and the majority element appears at least $k + 1$ times in it. Hence, A'' contains a majority element and it is the same as that of A' so we can recurse on A''.

Case 2 $m = 2k$

In this case, we select one element from each pair of identical elements in A' and also keep the left-over element. The resulting list A'' contains $2k + 1$ elements. We will argue that the majority elements of A' remains as the majority element of A''. In A', the majority element appears in at least k pairs of identical elements and at least $m + 1 = 2k + 1$ times. If not, the majority element can only appear at most $2k$ times in A', which is a contradiction. If the majority element appears in exactly k pairs of identical elements, then the left-over element must be equal to the majority. Hence, A'' contains a majority element which is the same as that of A'. We can now recurse on A''. If the majority element appears in more than k pairs of identical elements, then the majority element appears at least $k + 1$ time in A'' so we can recurse on A''. In summary: we can recurse on A'' in all cases and the size of A'' is about half the size of A. It is easy to turn this idea into a linear time algorithm to find the majority.

Pseudocode

Input: An array of numbers S indexed from 1 to n. Some element appears at least $\lfloor n/2 \rfloor + 1$ times in S.
Output: the number in the array with the highest frequency.

FindMode(S, n)

```
1.  if $n = 1$ then return $S[1]$
2.  $S_{new} =$
3.  $n_{new} = 0$
4.  for $i = 1$ to $n$ in steps of 2
5.     if ($S[i] = S[i+1]$) then
6.        $n_{new} = n_{new} + 1$
7.        $S_{new} = S_{new} \bigcup S[i]$
8.  if ($n$ mod 2) = 1 // $n$ is odd
9.     if ($n_{new}$ mod 2) = 0
10.       $S_{new} = S_{new} \bigcup S[n]$
11.       $n_{new} = n_{new} + 1$
12. return FindMode($S_{new}$,$n_{new}$)
```

Complexity

FindMode is recursively called with a smaller problem with size at most $\lceil n/2 \rceil$. At each step in the recursion $\lfloor n/2 \rfloor$ comparisons are performed. Therefore, the total running time of the algorithm is

$$T(n) = T(\lceil n/2 \rceil) + \Theta(n)$$

The solution to this recurrence is $T(n) = \Theta(n)$.

Problem 37: Coupons

Consider the following "coupon collector" problem. There are different varieties of cereal, and each comes with a single coupon for a discount on another box of cereal, perhaps of another variety. You can use multiple coupons when purchasing a new box, up to getting the new box free, but you never get money back. You want to buy one box of each variety, for as little money as possible. Describe an efficient algorithm, which, given as input, for each variety, its price , the value of the enclosed coupon, and the brand for which the coupon gives a discount, computes an optimal order in which to buy the cereal.

Solution: Coupons

Algorithm.

1. Model the problem as a graph where each of the n cereal boxes form the vertices of the graph. A directed edge with weight w is present from node i to node j if cereal box i gives a discount of w for box j. Observe that each node in the graph has outdegree equal to one. Label each node with the price of the cereal box. Let p_v denote the price of the node v and let w_{uv} denote the discount offered by box u towards the purchase of box v.

2. As long as there is a node u with no incoming edges, buy the box represented by the node, delete the unique outgoing edge (u, v), and update the price of v to be $\max(0, p_v - w_{uv})$.

3. At this point since we have no nodes with indegree zero and every node has outdegree 1, the graph must be a union of disjoint cycles.

4. The effective discount of any node u in a cycle is $\min(p_u, w_{vu})$ where (v, u) is the unique incoming edge into u. For each cycle, purchase the nodes in the same order as given by the cycle starting with the node with the smallest effective discount.

Correctness. In the first case, where box X has no coupons for it, let S be any schedule which does not buy box X first. Then let S' be the schedule where we buy X first, and then buy the other boxes in the same order as in S. Since X had no coupon, we pay the same price for X in both orders, namely full price. Now, for all other boxes, the price in S' is at most the price in S, the only difference being the box we got a coupon for by buying X, which may have dropped in price. Therefore, the total cost of S' is at most that of S.

In the second case, where we have a cycle, then the box X we buy is the one with the minimum real discount value. Again let S be any schedule that does not buy X first, and define S' as first buying all the boxes in X's cycle in order starting from X and then buying the rest as in S. Let Y be the first box in X's cycle bought in S. No box outside of X's cycle has changed price. For each box in the cycle except X and Y , we pay the full price - real value of the coupon in S', and at least that in S, since there is only one coupon. The price of X in S' may have increased over that in S by the real value of its coupon, but the price of Y has decreased: i.e in S, we pay full price but in S' we deduct the real value of $Y's$ coupon. Since X has the smallest real coupon value, this means the total price for S' is at most that of S.

Thus in both cases, we see that the greedy strategy does at least as well as the so called optimal strategy if not better and hence this algorithm is correct.

Time complexity. All the precomputation like the number of coupons for each box etc. takes $O(n)$. For all boxes with no coupons we do constant number of operations and number of such boxes is at most n. Therefore this step also takes $O(n)$. Now for the cycles, we randomly start at any point in a cycle and we go through the cycle at most twice. The cycle length can be at most only n. Moreover we do only constant-time operations for each node in the cycle. Therefore time complexity for this part of algorithm is $O(n)$.

Therefore total time complexity for the algorithm is $O(n)$.