CMPSC 465: Data Structures and Algorithms

Problem 1

Fibonacci Numbers

Fibonacci numbers

The Fibonacci numbers grow almost as fast as the powers of 2: for example, F_{30} is over a million, and F_{100} is already 21 digits long! In general, $F_n \approx 2^{0.694n}$. No other sequence of numbers has been studied as extensively, or applied to more fields: biology, demography, art, architecture, music, to name just a few. And, together with the powers of 2, it is computer science's favorite sequence.

Source: Algorithms by Dasgupta, Papadimitriou, Vazirani. McGraw-Hill. 2006.

Problem

The Fibonacci numbers $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$ are generated by the following simple rule

$$F_n = \begin{cases} F_{n-1} + F_{n-2}, & n > 1\,, \\ 1, & n = 1\,, \\ 0, & n = 0\,. \end{cases}$$

Given: A positive integer $n \leq 25$.

Return: The value of F_n.

Source: Algorithms by Dasgupta, Papadimitriou, Vazirani. McGraw-Hill. 2006.

Sample Dataset

```
6
```

Sample Output

```
8
```

Discussion

One idea is to slavishly implement the recursive definition of F_n. Here is the pseudocode of the resulting algorithm:

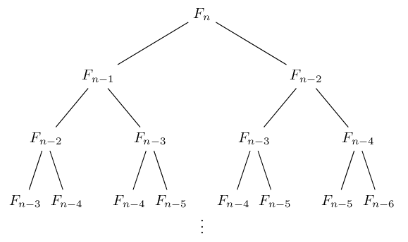


Figure 1. A tree showing the branching, exponential nature of recursive calls in fib1. The bottom row of the tree has calls; once the row grows to depth 20, there are 1,048,576 recursive calls.

```
procedure FIB1($n$)
    if $n = 0$ then
        return 0
    if $n = 1$ then
        return 1
    return FIB1($n - 1$)+FIB1($n - 2$)
```

Let $T(n)$ be the number of steps needed to compute `fib1`(n); what can we say about this function? For starters, if $n \leq 2$, then the procedure halts almost immediately, after just a couple of steps. Therefore, $T(n) \leq 2$ for $n \leq 1$. For larger values of n, there are two recursive calls of `fib1`, taking respective times $T(n-1)$ and $T(n-2)$, plus three computer steps (checks on the value of n and a final addition). Therefore,

$$T(n) = T(n-1) + T(n-2) + 3 \text{ for } n > 1.$$

Compare this to the recurrence relation for F_n: we immediately see that $T(n) \geq F_n$.

This is very bad news: the running time of the algorithm grows as fast as the Fibonacci numbers themselves! $T(n)$ is **exponential** in n, which implies that the algorithm is impractically slow except for very small values of n.

Let's try to understand why `fib1` is so slow. **Figure 1** shows the cascade of recursive invocations triggered by a single call to `fib1`(n). Notice that many computations are repeated! A more sensible scheme would store the intermediate results—the values $F_0, F_1, \ldots, F_{n-1}$—as soon as they become known.

```
procedure FIB2($n$)
    if $n = 0$ then
        return 0
    create an array $f[0 \ldots n]$
    $f[0] \leftarrow 0$, $f[1] \leftarrow 1$
    for $i \leftarrow 1$ to $n$ do
        $f[i] \leftarrow f[i-1] + f[i-2]$
    return $f[n]$
```

As with `fib1`, the correctness of this algorithm is self-evident because it directly uses the definition of F_n. How long does it take? The inner loop consists of a single computer step and is executed $n - 1$ times. Therefore the number of computer steps used by `fib2` is linear in n. From exponential we are down to **polynomial**, a huge breakthrough in running time. It is now perfectly reasonable to compute F_{200} or even $F_{200,000}$.

Source: Algorithms by Dasgupta, Papadimitriou, Vazirani. McGraw-Hill. 2006.

Visualization

To better feel the difference between polynomial and exponential time, try to compute F_{20} by using the following visualization by David Galles: http://www.cs.usfca.edu/~galles/visualization/DPFib.html. Selecting "Fibonacci Recursive" shows the enormous number of recursive calls that the program tries to make in `fib1`. Stop the animation when you get tired of watching it, and instead click "Fibonacci Table" to see just how much quicker using `fib2` is. (Note however that a slightly different definition is used there: $F_0 = F_1 = 1$.)

Problem 2

Binary Search

Binary search is the ultimate **divide-and-conquer** algorithm. To find a key k in a large file containing keys $A[1..n]$ in sorted order, we first compare k with $A[n/2]$, and depending on the result we recurse either on the first half of the file, $A[1..n/2]$, or on the second half, $A[n/2+1..n]$. The recurrence now is $T(n) = T(n/2) + O(1)$. Plugging into the **master theorem** (with $a = 1, b = 2, d = 0$) we get the familiar solution: a running time of just $O(\log n)$.

Source: Algorithms by Dasgupta, Papadimitriou, Vazirani. McGraw-Hill. 2006.

Problem

The problem is to find a given set of keys in a given array.

Given: Two positive integers $n \leq 10^5$ and $m \leq 10^5$, a sorted **array** $A[1..n]$ of integers from -10^5 to 10^5 and a list of m integers $-10^5 \leq k_1, k_2, \ldots, k_m \leq 10^5$.

Return: For each k_i, output an index $1 \leq j \leq n$ s.t. $A[j] = k_i$ or "-1" if there is no such index.

Sample Dataset

```
5
6
10 20 30 40 50
40 10 35 15 40 20
```

Sample Output

```
4 1 -1 -1 4 2
```

Problem 3

Insertion Sort

Computing the number of swaps in insertion sort

Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as "Quick Sort", "Heap Sort", or "Merge Sort". However, insertion sort provides several advantages: simple implementation, efficient for (quite) small data sets, $O(1)$ extra space.

When humans manually sort something (for example, a deck of playing cards), most use a method that is similar to insertion sort.

Source: Wikipedia

Although it is one of the elementary sorting algorithms with $O(n^2)$ worst-case time, insertion sort is the algorithm of choice either when the data is nearly sorted (because it is adaptive) or when the problem size is small (because it has low overhead).

For these reasons, and because it is also stable, insertion sort is often used as the recursive base case (when the problem size is small) for higher overhead **divide-and-conquer** sorting algorithms, such as "Merge Sort" or "Quick Sort".

Visualization by David R. Martin: http://www.sorting-algorithms.com/insertion-sort

Problem

Insertion sort is a simple algorithm with quadratic running time that builds the final sorted array one item at a time.

```
procedure InsertionSort($A[1..n]$)
    for $i \leftarrow 2$ to $n$ do
        $k \leftarrow i$
        while $k > 1$ and $A[k] < A[k-1]$ do
            Swap($A[k-1]$, $A[k]$)
            $k \leftarrow k - 1$
```

Given: A positive integer $n \leq 10^3$ and an array $A[1..n]$ of integers.

Return: The number of swaps performed by insertion sort algorithm on $A[1..n]$.

Sample Dataset

```
6
6 10 4 5 1 2
```

Sample Output

```
12
```

Discussion

For this problem, it is enough to implement the pseudocode above with a quadratic running time and to count the number of swaps performed. Note however that there exists an algorithm counting the number of swaps in $O(n \log n)$.

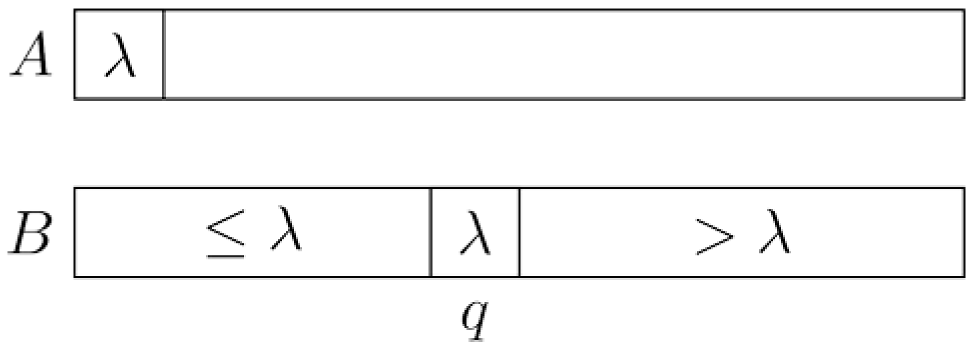
Note also that Insertion Sort is an **in-place** algorithm as it only requires storing a few counters.

Problem 4

2-Way Partition

Problem

A partition procedure is an essential part of the Quick Sort algorithm, the subject of one of the following problems. Its main goal is to put the first element of a given array to its proper place in a sorted array. It can be implemented in linear time, by a single scan of a given array. Moreover, it is not hard to come up with an in-place algorithm.



Given: A positive integer $n \leq 10^5$ and an array $A[1..n]$ of integers from -10^5 to 10^5.

Return: A permuted array $B[1..n]$ such that it is a permutation of A and there is an index $1 \leq q \leq n$ such that $B[i] \leq A[1]$ for all $1 \leq i \leq q-1$, $B[q] = A[1]$, and $B[i] > A[1]$ for all $q+1 \leq i \leq n$.

Sample Dataset

```
9
7 2 5 6 1 3 9 4 8
```

Sample Output

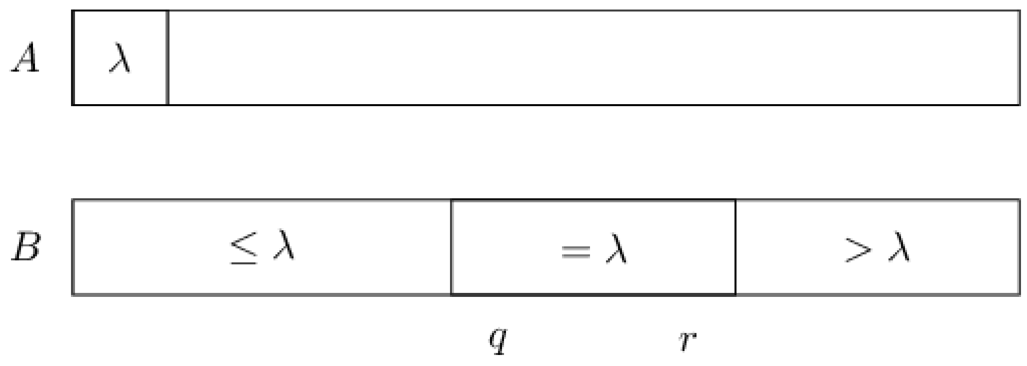
```
5 6 3 4 1 2 7 9 8
```

Problem 5

3-Way Partition

Problem

This problem is very similar to “2-Way Partition”, but now the goal is to partition an input array more carefully.



Given: A positive integer $n \leq 10^5$ and an array $A[1..n]$ of integers from -10^5 to 10^5.

Return: An array $B[1..n]$ such that it is a permutation of A and there are indices $1 \leq q \leq r \leq n$ such that $B[i] < A[1]$ for all $1 \leq i \leq q-1$, $B[i] = A[1]$ for all $q \leq i \leq r$, and $B[i] > A[1]$ for all $r+1 \leq i \leq n$.

Sample Dataset

```
9
4 5 6 4 1 2 5 7 4
```

Sample Output

```
2 1 4 4 4 5 7 6 5
```

Problem 6

Quick Sort

Problem

Comparing the algorithms for sorting and “Median” finding we notice that, beyond the common divide-and-conquer philosophy and structure, they are exact opposites. “Merge Sort” splits the array in two in the most convenient way (first half, second half), without any regard to the magnitudes of the elements in each half; but then it works hard to put the sorted subarrays together. In contrast, the median algorithm is careful about its splitting (smaller numbers first, then the larger ones), but its work ends with the recursive call.

Quick sort is a sorting algorithm that splits the array in exactly the same way as the median algorithm; and once the subarrays are sorted, by two recursive calls, there is nothing more to do. Its worst-case performance is $\Theta(n^2)$, like that of median-finding. But it can be proved that its average case is $O(n \log n)$; furthermore, empirically it outperforms other sorting algorithms. This has made quicksort a favorite in many applications— for instance, it is the basis of the code by which really enormous files are sorted.

Source: Algorithms by Dasgupta, Papadimitriou, Vazirani. McGraw-Hill. 2006.

Given: A positive integer $n \le 10^5$ and an array $A[1..n]$ of integers from -10^5 to 10^5.

Return: A sorted array $A[1..n]$.

Sample Dataset

```
7
5 -2 4 7 8 -10 11
```

Sample Output

```
-10 -2 4 5 7 8 11
```

Visualization

http://www.sorting-algorithms.com/quick-sort-3-way

Running time

To prove an upper bound $O(n \log n)$ on the expected running time of Quick Sort show that is satisfies the recurrence relation

$$T(n) \le O(n) + \frac{1}{n}\sum_{i=1}^{n}(T(i) + T(n-i)).$$

Problem 7

Counting Inversions

Problem

An inversion of an array $A[1..n]$ is a pair of indices (i, j) such that $1 \le i < j \le n$ and $A[i] > A[j]$. The number of inversions shows how far the array is from being sorted: if it is already sorted then there are no inversions while if it sorted in reverse order then the number of inversions is maximal.

Given: A positive integer $n \le 10^5$ and an array $A[1..n]$ of integers from -10^5 to 10^5.

Return: The number of inversions in A.

Sample Dataset

```
5
-6 1 15 8 10
```

Sample Output

```
2
```

Problem 8

Merge Two Sorted Arrays

Problem

The merging procedure is an essential part of "Merge Sort" (which is considered in one of the next problems).

Given: A positive integer $n \leq 10^5$ and a sorted array $A[1..n]$ of integers from -10^5 to 10^5, a positive integer $m \leq 10^5$ and a sorted array $B[1..m]$ of integers from -10^5 to 10^5.

Return: A sorted array $C[1..n+m]$ containing all the elements of A and B.

Sample Dataset

```
4
2 4 10 18
3
-5 11 12
```

Sample Output

```
-5 2 4 10 11 12 18
```

> ### Hint
>
> The very first element of C is either $A[1]$ or $B[1]$, whichever is smaller. The rest of C can then be constructed recursively.
>
> Source: Algorithms by Dasgupta, Papadimitriou, Vazirani. McGraw-Hill. 2006.

Problem 9

Merge Sort

Problem

The problem of **sorting** a list of numbers lends itself immediately to a divide-and-conquer strategy: split the list into two halves, recursively sort each half, and then merge the two sorted sublists (recall the problem “Merge Two Sorted Arrays”).

Source: Algorithms by Dasgupta, Papadimitriou, Vazirani. McGraw-Hill. 2006.

Given: A positive integer $n \leq 10^5$ and an array $A[1..n]$ of integers from -10^5 to 10^5.

Return: A sorted array $A[1..n]$.

Sample Dataset

```
10
20 19 35 -18 17 -20 20 1 4 4
```

Sample Output

```
-20 -18 1 4 4 17 19 20 20 35
```

Visualization

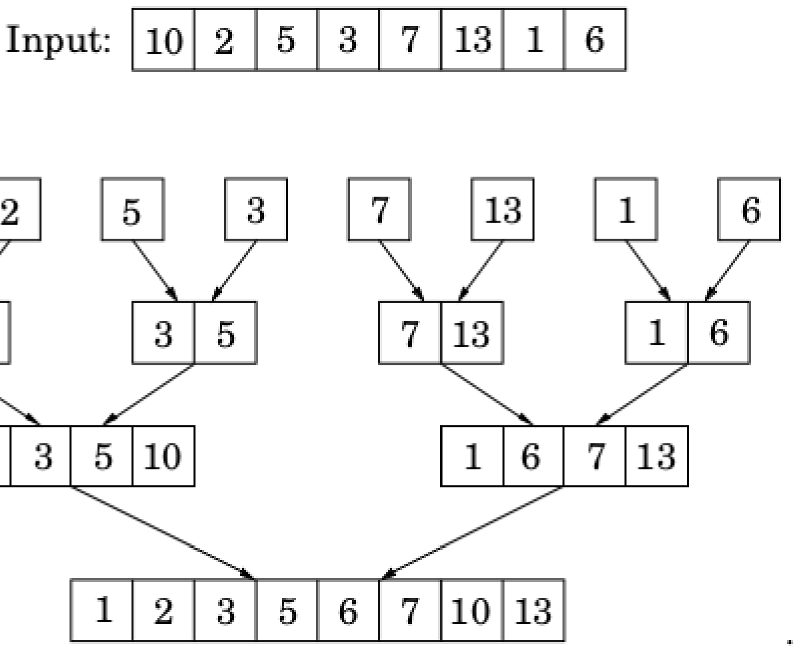
Visualization by David R. Martin: http://www.sorting-algorithms.com/merge-sort

Running time

The running time of Merge Sort satisfies a recurrence relation $T(n) = 2T(n/2) + O(n)$. The master theorem implies that $T(n) = O(n \log n)$.

Non-recursive implementation

Looking back at the Merge Sort algorithm, we see that all the real work is done in merging, which doesn’t start until the recursion gets down to singleton arrays. The singletons are merged in pairs, to yield arrays with two elements. Then pairs of these 2-tuples are merged, producing 4-tuples, and so on. The following figure shows an example.



This viewpoint also suggests how Merge Sort might be made iterative. At any given moment, there is a set of “active” arrays—initially, the singletons—which are merged in pairs to give the next batch of active arrays. These arrays can be organized in a queue, and processed by repeatedly removing two arrays from the front of the queue, merging them, and putting the result at the end of the queue.

Source: Algorithms by Dasgupta, Papadimitriou, Vazirani. McGraw-Hill. 2006.

Problem 10

Building a Heap

Problem

A **binary heap** is a binary tree based data structure that is often used to implement **priority queues**. Binary heaps, in turn, can be easily implemented using an array if the underlying tree is a complete binary tree. The tree nodes have a natural ordering: row by row, starting at the root and moving left to right within each row. If there are n nodes, this ordering specifies their positions $1, 2, \ldots, n$ within the array. Moving up and down the tree is easily simulated on the array, using the fact that node number j has parent $\lceil j/2 \rceil$ and children $2j$ and $2j+1$.

The goal of this problem is to build a heap from the given array. For this, go from the end to the beginning of a given array and let each element "bubble up".

Source: Algorithms by Dasgupta, Papadimitriou, Vazirani. McGraw-Hill. 2006.

Given: A positive integer $n \leq 10^5$ and array $A[1..n]$ of integers from -10^5 to 10^5.

Return: A permuted array A satisfying the binary max heap property: for any $2 \leq i \leq n$, $A[\lfloor i/2 \rfloor] \geq A[i]$.

Sample Dataset

```
5
1 3 5 7 2
```

Sample Output

```
7 5 1 3 2
```

> ### Running time
>
> Since each "bubble up" operation requires only $O(\log n)$ time the running time of this algorithm is $O(n \log n)$. A more careful analysis shows that the running time is in fact just linear.

Problem 11

Heap Sort

Problem

The heap sort algorithm first transforms a given array into a max heap (recall the problem "Building a Heap"). It then repeats the following two simple steps $n - 1$ times:

- Swap the last element of the heap with its root and decrease the size of the heap by 1.
- "Sift-down" the new root element to its proper place.

Given: A positive integer $n \leq 10^5$ and an array $A[1..n]$ of integers from -10^5 to 10^5.

Return: A sorted array A.

Sample Dataset

```
9
2 6 7 1 3 5 4 8 9
```

Sample Output

```
1 2 3 4 5 6 7 8 9
```

> ### Running time

Transforming a given array into a heap is done in linear time, $O(n)$. The next stage requires $n-1$ "sift-down" operations and hence take time $O(n \log n)$. Hence the overall running time of the heap sort algorithm is $O(n \log n)$.

Visualization

To see how the heap sort algorithm performs on various types of arrays use the following visualization by David R. Martin: http://www.sorting-algorithms.com/heap-sort. The visualization by David Galles also shows how the heap sort algorithm performs on a random array: http://www.cs.usfca.edu/~galles/visualization/HeapSort.html.

Problem 12

Degree Array

Why graphs?

A wide range of problems can be expressed with clarity and precision in the concise pictorial language of graphs. For instance, consider the task of coloring a political map. What is the minimum number of colors needed, with the obvious restriction that neighboring countries should have different colors? One of the difficulties in attacking this problem is that the map itself, even a stripped-down version like **Figure 1**, is usually cluttered with irrelevant information: intricate boundaries, border posts where three or more countries meet, open seas, and meandering rivers. Such distractions are absent from the mathematical object of **Figure 2**, a graph with one vertex for each country (1 is Brazil, 11 is Argentina) and edges between neighbors. It contains exactly the information needed for coloring, and nothing more. The precise goal is now to assign a color to each vertex so that no edge has endpoints of the same color.

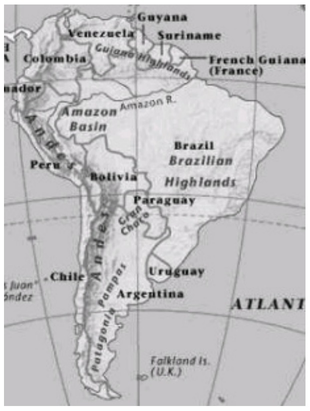


Figure 1

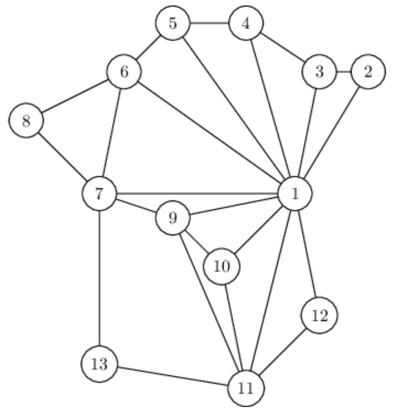


Figure 2

Graph coloring is not the exclusive domain of map designers. Suppose a university needs to schedule examinations for all its classes and wants to use the fewest time slots possible. The only constraint is that two exams cannot be scheduled concurrently if some student will be taking both of them. To express this problem as a graph, use one vertex for each exam and put an edge between two vertices if there is a conflict, that is, if there is somebody taking both endpoint exams. Think of each time slot as having its own color. Then, assigning time slots is exactly the same as coloring this graph!

Formally, a graph is specified by a set of vertices (also called *nodes*) V and by edges E between select pairs of vertices. In the map example, $V = \{1, 2, 3, \ldots, 13\}$ and E includes, among many other edges, $\{1, 2\}$, $\{9, 11\}$, and $\{7, 13\}$. Here an edge between x and y specifically means "x shares a border with y." This is a symmetric relation—it implies also that y shares a border with x—and we denote it using set notation, $e = \{x, y\}$. Such edges are *undirected* and are part of an *undirected* graph.

Sometimes graphs depict relations that do not have this reciprocity, in which case it is necessary to use edges with directions on them. There can be directed edges e from x to y (written $e = (x, y)$), or from y to x (written (y, x)), or both. A particularly enormous example of a directed graph is the graph of all links in the World Wide Web. It has a vertex for each site on the Internet, and a directed edge (u, v) whenever site u has a link to site v: in total, billions of nodes and edges! Understanding even the most basic connectivity properties of the Web is of great economic and social interest. Although the size of this problem is daunting, we will soon see that a lot of valuable information about the structure of a graph can, happily, be determined in just **linear** time.

Source: Algorithms by Dasgupta, Papadimitriou, Vazirani. McGraw-Hill. 2006.

Problem

In an undirected graph, the *degree* $d(u)$ of a vertex u is the number of neighbors u has, or equivalently, the number of edges incident upon it.

Given: A **simple graph** with $n \leq 10^3$ vertices in the **edge list format**.

Return: An array $D[1..n]$ where $D[i]$ is the degree of vertex i.

See **Figure 3** for visual example from the sample dataset.

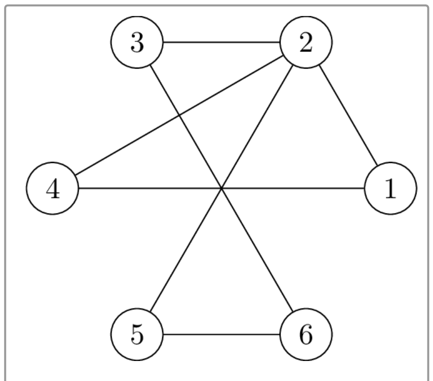


Figure 3. The graph from the dataset

Sample Dataset

```
6 7
1 2
2 3
6 3
5 6
2 5
2 4
4 1
```

Sample Output

```
2 4 2 2 2 2
```

Problem 13

Double-Degree Array

Problem

Source: Algorithms by Dasgupta, Papadimitriou, Vazirani. McGraw-Hill. 2006.

Given: A simple graph with $n \leq 10^3$ vertices in the edge list format.

Return: An array $D[1..n]$ where $D[i]$ is the sum of the degrees of i's neighbors.

See **Figure 1** for visual example from the sample dataset.

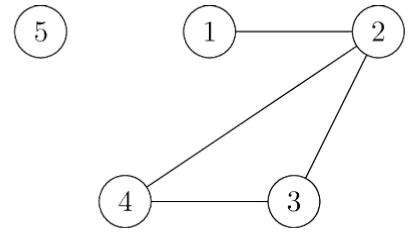


Figure 1. The graph from the dataset

Sample Dataset

```
5 4
1 2
2 3
4 3
2 4
```

Sample Output

```
3 5 5 5 0
```

Adjacency list data structure

An **adjacency list** data structure may come in useful.

Problem 14

Connected Components

Problem

The task is to use **depth-first search** to compute the number of **connected components** in a given undirected graph.

Given: A simple graph with $n \leq 10^3$ vertices in the edge list format.

Return: The number of connected components in the graph.

See **Figure 1** for visual example from the sample dataset.

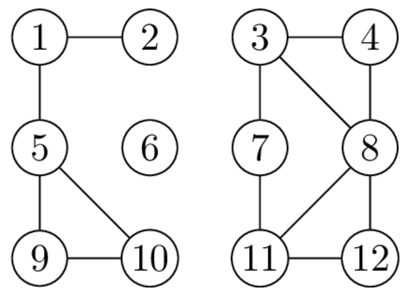


Figure 1. The graph from the dataset

Sample Dataset

```
12 13
1 2
1 5
5 9
```

```
5 10
9 10
3 4
3 7
3 8
4 8
7 11
8 11
11 12
8 12
```

Sample Output

```
3
```

Problem 15

Testing Bipartiteness

Problem

A *bipartite* graph is a graph $G = (V, E)$ whose vertices can be partitioned into two sets ($V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$) 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).

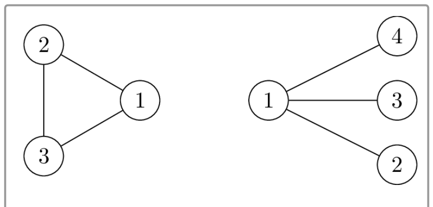


Figure 1. The graphs from the dataset

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. Another one: an undirected graph is bipartite if and only if it contains no cycles of odd length.

Source: Algorithms by Dasgupta, Papadimitriou, Vazirani. McGraw-Hill. 2006.

Given: A positive integer $k \le 20$ and k simple graphs in the edge list format with at most 10^3 vertices each.

Return: For each graph, output "1" if it is bipartite and "-1" otherwise.

See **Figure 1** for visual example from the sample dataset.

Sample Dataset

```
2

3 3
1 2
3 2
3 1
```

```
4 3
1 4
3 1
1 2
```

Sample Output

```
-1 1
```

Problem 16

Testing Acyclicity

Problem

Given: A positive integer $k \leq 20$ and k simple directed graphs in the edge list format with at most 10^3 vertices and $3 \cdot 10^3$ edges each.

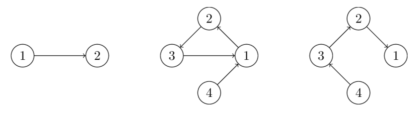


Figure 1. The graphs from the dataset

Return: For each graph, output "1" if the graph is **acyclic** and "-1" otherwise.

See **Figure 1** for visual example from the sample dataset.

Sample Dataset

```
3

2 1
1 2

4 4
4 1
1 2
2 3
3 1

4 3
4 3
3 2
2 1
```

Sample Output

```
1 -1 1
```

Problem 17

Breadth-First Search

Problem

The task is to use **breadth-first search** to compute single-source shortest distances in an unweighted directed graph.

Given: A simple directed graph with $n \le 10^3$ vertices in the edge list format.

Return: An array $D[1..n]$ where $D[i]$ is the length of a shortest path from the vertex 1 to the vertex i ($D[1] = 0$). If i is not reachable from 1 set $D[i]$ to -1.

See **Figure 1** for visual example from the sample dataset.

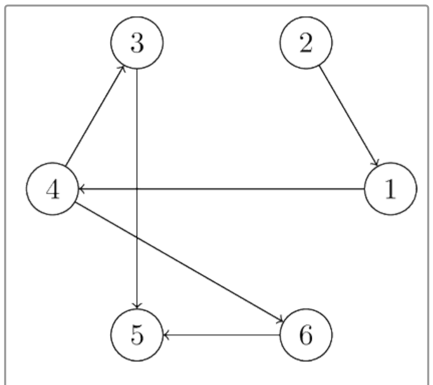


Figure 1. The graph from the dataset

Sample Dataset

```
6 6
4 6
6 5
4 3
3 5
2 1
1 4
```

Sample Output

```
0 -1 2 1 3 2
```

Problem 18

Dijkstra's Algorithm

Problem

The task is to use **Dijkstra's algorithm** to compute single-source shortest distances in a directed graph with positive edge weights.

Given: A simple directed graph with positive edge weights from 1 to 10^3 and $n \leq 10^3$ vertices in the edge list format.

Return: An array $D[1..n]$ where $D[i]$ is the length of a shortest path from the vertex 1 to the vertex i ($D[1] = 0$). If i is not reachable from 1 set $D[i]$ to -1.

See **Figure 1** for visual example from the sample dataset.

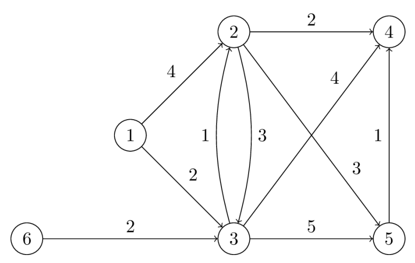


Figure 1. The graph from the dataset

Sample Dataset

```
6 10
3 4 4
1 2 4
1 3 2
2 3 3
6 3 2
3 5 5
5 4 1
3 2 1
2 4 2
2 5 3
```

Sample Output

```
0 3 2 5 6 -1
```

Problem 19

Bellman-Ford Algorithm

Problem

The task is to use **Bellman-Ford algorithm** to compute single-source shortest distances in a directed graph with possibly negative edge weights (but without negative cycles).

Given: A simple directed graph with integer edge weights from -10^3 to 10^3 and $n \leq 10^3$ vertices in the edge list format.

Return: An array $D[1..n]$ where $D[i]$ is the length of a shortest path from the vertex 1 to the vertex i ($D[1] = 0$). If i is not reachable from 1 set $D[i]$ to `x`.

See **Figure 1** for visual example from the sample dataset.

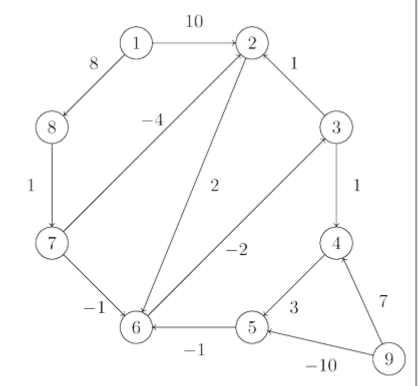


Figure 1. The graph from the dataset

Sample Dataset

```
9 13
1 2 10
3 2 1
3 4 1
4 5 3
5 6 -1
7 6 -1
8 7 1
1 8 8
7 2 -4
2 6 2
6 3 -2
9 5 -10
9 4 7
```

Sample Output

```
0 5 5 6 9 7 9 8 x
```

Problem 20

Negative Weight Cycle

Problem

The task is to use **Bellman-Ford algorithm** to check whether a given graph contains a cycle of negative weight.

Given: A positive integer $k \le 20$ and k simple directed graphs with integer edge weights from -10^3 to 10^3 and $n \le 10^3$ vertices in the edge list format.

Return: For each graph, output "1" if it contains a negative weight cycle and "-1" otherwise.

Sample Dataset

```
2

4 5
```

```
1 4 4
4 2 3
2 3 1
3 1 6
2 1 -7

3 4
1 2 -8
2 3 20
3 1 -1
3 2 -30
```

Sample Output

```
-1 1
```

Problem 21

Median

Problem

The task is to implement a linear time randomized algorithm for the selection problem.

Given: A positive integer $n \leq 10^5$ and an array $A[1..n]$ of integers from -10^5 to 10^5, a positive number $k \leq n$.

Return: The k-th smallest element of A.

Source: Algorithms by Dasgupta, Papadimitriou, Vazirani. McGraw-Hill. 2006.

Sample Dataset

```
11
2 36 5 21 8 13 11 20 5 4 1
8
```

Sample Output

```
13
```

Problem 22

Majority Element

Problem

An array $A[1..n]$ is said to have a *majority element* if more than half of its entries are the same.

Given: A positive integer $k \leq 20$, a positive integer $n \leq 10^4$, and k arrays of size n containing positive integers not exceeding 10^5.

Return: For each array, output an element of this array occurring strictly more than $n/2$ times if such element exists, and "-1" otherwise.

Source: Algorithms by Dasgupta, Papadimitriou, Vazirani. McGraw-Hill. 2006.

Sample Dataset

```
4 8
5 5 5 5 5 5 5 5
8 7 7 7 1 7 3 7
7 1 6 5 10 100 1000 1
5 1 6 7 1 1 10 1
```

Sample Output

```
5 7 -1 -1
```

Discussion

It is not difficult to develop a divide-and-conquer algorithm checking whether a given array of size n contains a majority element in $O(n \log n)$ time. It is interesting to note that there is also a linear time algorithm and it is also based on divide-and-conquer.

Problem 23

2SUM

Problem

Given: A positive integer $k \leq 20$, a positive integer $n \leq 10^4$, and k arrays of size n containing integers from -10^5 to 10^5.

Return: For each array $A[1..n]$, output two different indices $1 \leq p < q \leq n$ such that $A[p] = -A[q]$ if exist, and "-1" otherwise.

Sample Dataset

```
4 5
2 -3 4 10 5
8 2 4 -2 -8
-5 2 3 2 -4
5 4 -5 6 8
```

Sample Output

```
-1
2 4
-1
1 3
```

Problem 24

3SUM

Problem

Given: A positive integer $k \leq 20$, a postive integer $n \leq 10^4$, and k arrays of size n containing integers from -10^5 to 10^5.

Return: For each array $A[1..n]$, output three different indices $1 \leq p < q < r \leq n$ such that $A[p] + A[q] + A[r] = 0$ if exist, and "-1" otherwise.

Sample Dataset

```
4 5
2 -3 4 10 5
8 -6 4 -2 -8
-5 2 3 2 -4
2 4 -5 6 8
```

Sample Output

```
-1
1 2 4
1 2 3
-1
```

Problem 25

Partial Sort

Problem

Given: A positive integer $n \leq 10^5$ and an array $A[1..n]$ of integers from -10^5 to 10^5, a positive integer $k \leq 1000$.

Return: The k smallest elements of a sorted array A.

Sample Dataset

```
10
4 -6 7 8 -9 100 12 13 56 17
3
```

Sample Output

```
-9 -6 4
```

Problem 26

Semi-Connected Graph

Problem

A directed graph is *semi-connected* if for all pairs of vertices i, j there is either a path from i to j or a path from j to i.

Given: A positive integer $k \leq 20$ and k simple directed graphs with at most 10^3 vertices each in the edge list format.

Return: For each graph, output "1" if the graph is semi-connected and "-1" otherwise.

Sample Dataset

```
2

3 2
3 2
2 1

3 2
3 2
1 2
```

Sample Output

```
1 -1
```

Problem 27

Square in a Graph

Problem

Given: A positive integer $k \leq 20$ and k simple undirected graphs with $n \leq 400$ vertices in the edge list format.

Return: For each graph, output "1" if it contains a simple cycle (that is, a cycle which doesn’t intersect itself) of length 4 and "-1" otherwise.

Source: Algorithms by Dasgupta, Papadimitriou, Vazirani. McGraw-Hill. 2006.

Sample Dataset

```
2

4 5
3 4
4 2
3 2
3 1
1 2

4 4
1 2
3 4
```

```
2 4
4 1
```

Sample Output

```
1 -1
```

> ### Hint
>
> The **adjacency matrix** may come in useful for this problem.

Problem 28

Shortest Cycle Through a Given Edge

Problem

Given: A positive integer $k \leq 20$ and k simple directed graphs with positive integer edge weights and at most 10^3 vertices in the edge list format.

Return: For each graph, output the length of a shortest cycle going through the first specified edge if there is a cycle and "-1" otherwise.

Sample Dataset

```
2

4 5
2 4 2
3 2 1
1 4 3
2 1 10
1 3 4

4 5
3 2 1
2 4 2
4 1 3
2 1 10
1 3 4
```

Sample Output

```
-1 10
```

Problem 29

Topological Sorting

Problem

Given: A simple directed acyclic graph with $n \leq 10^3$ vertices in the edge list format.

Return: A **topological sorting** (i.e., a permutation of vertices) of the graph.

Sample Dataset

```
4 5
1 2
3 1
3 2
4 3
4 2
```

Sample Output

```
4 3 1 2
```

> **Visualization**
>
> Visualization by David Galles: http://www.cs.usfca.edu/~galles/visualization/ConnectedComponent.html.

Problem 30

Shortest Paths in DAG

Problem

There are two subclasses of graphs that automatically exclude the possibility of negative cycles: graphs without negative edges, and graphs without cycles. We already know how to efficiently handle the former (see the problem "Negative Weight Cycle"). We will now see how the single-source shortest-path problem can be solved in just linear time on directed acyclic graphs.

As before, we need to perform a sequence of updates (recall Bellman-Ford algorithm) that includes every shortest path as a subsequence. The key source of efficiency is that

In any path of a dag, the vertices appear in increasing linearized order.

Therefore, it is enough to linearize (that is, topologically sort) the dag by depth-first search, and then visit the vertices in sorted order, updating the edges out of each. The algorithm is given below.

Notice that our scheme doesn't require edges to be positive. In particular, we can find *longest paths* in a dag by the same algorithm: just negate all edge lengths.

Source: Algorithms by Dasgupta, Papadimitriou, Vazirani. McGraw-Hill. 2006.

Given: A weighted DAG with integer edge weights from -10^3 to 10^3 and $n \leq 10^5$ vertices in the edge list format.

Return: An array $D[1..n]$ where $D[i]$ is the length of a shortest path from the vertex 1 to the vertex i ($D[1] = 0$). If i is not reachable from 1 set $D[i]$ to `x`.

Sample Dataset

```
5 6
2 3 4
4 3 -2
1 4 1
1 5 -3
2 4 -2
5 4 1
```

Sample Output

```
0 x -4 -2 -3
```

Problem 31

Strongly Connected Components

Problem

Given: A simple directed graph with $n \leq 10^3$ vertices in the edge list format.

Return: The number of **strongly connected components** in the graph.

Sample Dataset

```
6 7
4 1
1 2
2 4
5 6
3 2
5 3
3 5
```

Sample Output

```
3
```

Problem 32

2-Satisfiability

Problem

In the 2SAT problem, you are given a set of *clauses*, where each clause is the disjunction (OR) of two literals (a literal is a Boolean variable or the negation of a Boolean variable). You are looking for a way to assign a value `true` or `false` to each of the variables so that all clauses are satisfied — that is, there is at least one true literal in each clause. For example, here's an instance of 2SAT:

$$(x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee \overline{x}_3) \wedge (x_1 \vee x_2) \wedge (\overline{x}_3 \vee x_4) \wedge (\overline{x}_1 \vee x_4)\ .$$

This instance has a satisfying assignment: set x_1, x_2, x_3, and x_4 to `true`, `false`, `false`, and `true`, respectively.

1. Are there other satisfying truth assignments of this 2SAT formula? If so, find them all.
2. Give an instance of 2SAT with four variables, and with no satisfying assignment.

The purpose of this problem is to lead you to a way of solving 2SAT efficiently by reducing it to the problem of finding the strongly connected components of a directed graph. Given an instance I of 2SAT with n variables and m clauses, construct a directed graph $G_I = (V, E)$ as follows.

- G_I has $2n$ nodes, one for each variable and its negation.
- G_I has $2m$ edges: for each clause $(\alpha \vee \beta)$ of I (where α, β are literals), G_I has an edge from the negation of α to β, and one from the negation of β to α.

Note that the clause $(\alpha \vee \beta)$ is equivalent to either of the implications $\overline{\alpha} \Rightarrow \beta$ or $\overline{\beta} \Rightarrow \alpha$. In this sense, G_I records all implications in I.

3. Carry out this construction for the instance of 2SAT given above, and for the instance you constructed in 2.
4. Show that if G_I has a strongly connected component containing both x and $\overline{x}$ for some variable x, then I has no satisfying assignment.

5. Now show the converse of 4: namely, that if none of G_I's strongly connected components contain both a literal and its negation, then the instance I must be satisfiable. (Hint: Assign values to the variables as follows: repeatedly pick a sink strongly connected component of G_I. Assign value `true` to all literals in the sink, assign `false` to their negations, and delete all of these. Show that this ends up discovering a satisfying assignment.)
6. Conclude that there is a linear-time algorithm for solving 2SAT.

Source: Algorithms by Dasgupta, Papadimitriou, Vazirani. McGraw-Hill. 2006.

Given: A positive integer $k \leq 20$ and k 2SAT formulas represented as follows. The first line gives the number of variables $n \leq 10^3$ and the number of clauses $m \leq 10^4$, each of the following m lines gives a clause of length 2 by specifying two different literals: e.g., a clause $(x_3 \vee \overline{x}_5)$ is given by `3 -5`.

Return: For each formula, output 0 if it cannot be satisfied or 1 followed by a satisfying assignment otherwise.

Sample Dataset

```
2

2 4
1 2
-1 2
1 -2
-1 -2

3 4
1 2
2 3
-1 -2
-2 -3
```

Sample Output

```
0
1 1 -2 3
```

Problem 33

Hamiltonian Path in DAG

Problem

A Hamiltonian path is a path in a graph that visits each vertex exactly once. Checking whether a graph contains a Hamiltonian path is a well-known hard problem. At the same time it is easy to perform such a check if a given graph is a DAG.

Given: A positive integer $k \leq 20$ and k simple directed acyclic graphs in the edge list format with at most 10^3 vertices each.

Return: For each graph, if it contains a Hamiltonian path output "1" followed by a Hamiltonian path (i.e., a list of vertices), otherwise output "-1".

Sample Dataset

```
2

3 3
1 2
2 3
1 3

4 3
4 3
3 2
4 1
```

Sample Output

```
1 1 2 3
-1
```