Note: So far we have seen selection sort, bubble sort, and insertion sort. All three of these take $O\left(n^2\right)$ time. But there are $O(n \lg n)$ time sorting algorithm, e.g., mergesort, which we shall learn later in the course. In these home work problems, if you need sorting, you may assume that you have an $O(n \lg n)$ sorting algorithm at hand.

1. (10 points) A firm wants to determine the highest floor of its n-story headquarters from which a gadget can fall without breaking. The firm has two identical gadgets to experiment with. If one of them gets broken, it cannot be repaired, and the experiment will have to be completed with the remaining gadget. Design as efficient (as few droppings as you need) an algorithm as you can to solve this problem.

   [Solution]

   We could start at the first floor and check every floor following it upwards. Then we do not need two gadgets. The best-case is when gadget breaks at first floor. The worst-case is when the gadget first breaks at the n-th floor. No other method's best-case is better than this one. But we are usually more concerned about the worst-case (average-case is more important but it is hard to analyze). So, we shall try to improve the worst-case as we have more than one gadget.

   In order to minimize the number of required drops, we should make big jumps. But when the first gadget breaks, we fall back to linear search for the "jump" number of floors in the worst-case. So, we should hit an optimal trade-off between number of jumps and jump-length. Say, jump length is j. Then in the worst-case we need to make $\frac{n}{j}$ jumps and following that we have to perform linear search of length j. The optimal point is when $\frac{n}{j} = j$ or $j = \sqrt{n}$. So, we drop the first gadget from the $\lfloor\sqrt{n}\rfloor$-th floor and if it doesn't break, move to the $2 \cdot \lfloor\sqrt{n}\rfloor$-th floor, and so on. When the first gadget breaks, we have to do $\lfloor\sqrt{n}\rfloor$ drops with the second gadget in the worst-case. So, the algorithm is in $O(\sqrt{n})$.

2. (10 points) Apply topological sorting using Depth First Search on the following directed graph. Show

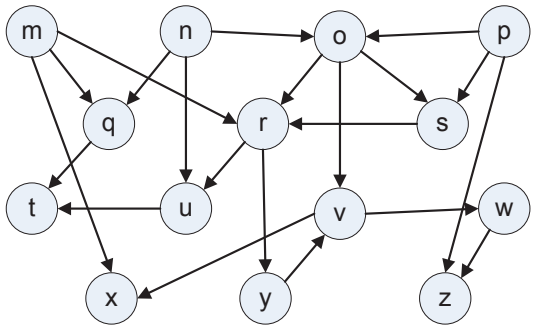

Figure 1: Digraph for Topological Sorting

   the full stack with push and pop indices for each element, DFS forest (with various types of edges), and the final topological order of the vertices.

   [Solution]

   Figure 2 shows the DFS forest and the full stack. The topological order is the reverse of the pop-order: $\langle p, n, o, s, m, r, y, v, x, w, z, u, q, t\rangle$

3. (10 points) There are two types of professional wrestlers: "babyfaces" ("good guys") and "heels" ("bad guys"). Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n professional wrestlers and we have a list of r pairs of wrestlers for which there are rivalries. Give an $O(n + r)$-time algorithm that determines whether it is possible to designate some of the wrestlers as babyfaces and the remainder as heels such that each rivalry is between a babyface and a heel. If it is possible to perform such a designation, your algorithm should produce it. Explain why do you think your algorithm's time efficiency is $O(n + r)$? [**Hint:** Think in terms of graphs.]

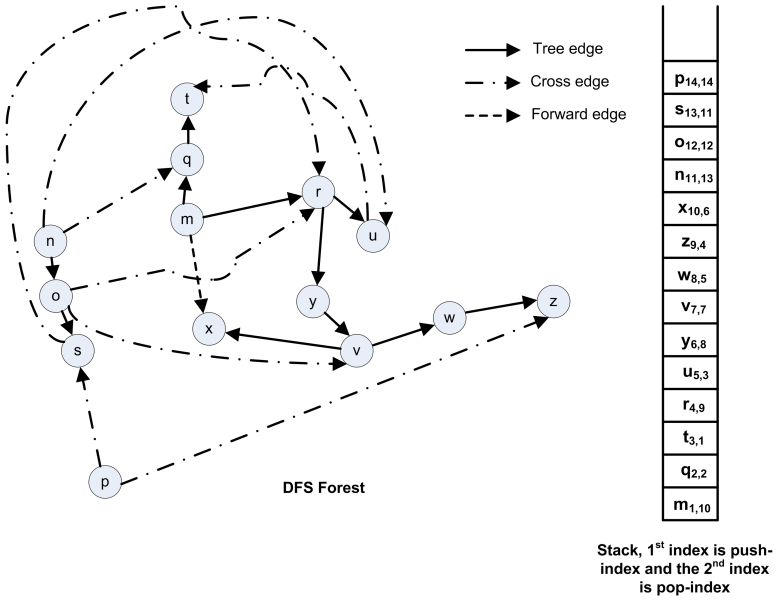

Figure 2: DFS forest and Stack

[Solution] Represent the problem as a graph, where each vertex represents a wrestler and each edge represents a rivalry. The graph will have n vertices and r edges.

Perform as many BFS's as are needed to visit all n vertices. Assign the first wrestler to be a good guy and then assign all its immediate neighbors to be bad guys, and so on. If a wrestler is assigned to be a good guy (or bad guy), but one of its neighbors has already been assigned to be a good guy (or bad guy), report that a desired designation is not possible. The solution has BFS's complexity, which is $O(n+r)$.

4. (10 points) One can model a maze by having a vertex for a starting point, a finishing point, dead ends, and all the points in the maze where more than one path can be taken, and then connecting the vertices according to the paths in the maze.

   (a) Construct such a graph for the maze in Figure 2.

Figure 3: Maze

[Solution] Figure 4 shows the numbering and the graph for the maze.

   (b) Which traversal–DFS or BFS–would you use if you found yourself in a maze and why?

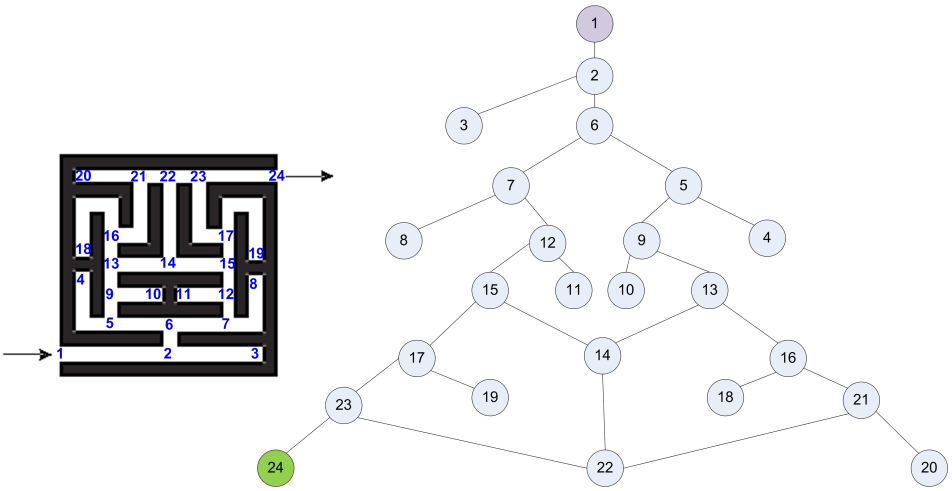

Figure 4: Maze

[Solution]
DFS or BFS, either one is efficient.

5. (10 points) Generate all permutations of $\{3, 5, 6, 9\}$ by

   (a) the bottom-up minimal-change algorithm.

   (b) the Johnson-Trotter algorithm (with arrows and the greatest mobile element marked).

   (c) the lexicographic-order algorithm.

   Solution

   (a) $3, 35, 53, 356, 365, 635, 653, 563, 536, 3569, 3596, 3956, 9356, 9365, 3965, 3695, 3659, 6359, 6395, 6935, 9635, 9653, 6953, 6593, 6539, 5639, 5693, 5963, 9563, 9536, 5936, 5396, 5369$

   (b) $\overleftarrow{3}\,\overleftarrow{5}\,\overleftarrow{6}\,\overleftarrow{9}$, $\overleftarrow{3}\,\overleftarrow{5}\,\overleftarrow{9}\,\overleftarrow{6}$, $\overleftarrow{3}\,\overleftarrow{9}\,\overleftarrow{5}\,\overleftarrow{6}$, $\overleftarrow{9}\,\overleftarrow{3}\,\overleftarrow{5}\,\overleftarrow{6}$, $\overrightarrow{9}\,\overleftarrow{3}\,\overleftarrow{6}\,\overleftarrow{5}$, $\overleftarrow{3}\,\overrightarrow{9}\,\overleftarrow{6}\,\overleftarrow{5}$, $\overleftarrow{3}\,\overleftarrow{6}\,\overrightarrow{9}\,\overleftarrow{5}$, $\overleftarrow{3}\,\overleftarrow{6}\,\overleftarrow{5}\,\overrightarrow{9}$, $\overleftarrow{6}\,\overleftarrow{3}\,\overleftarrow{5}\,\overleftarrow{9}$, $\overleftarrow{6}\,\overleftarrow{3}\,\overleftarrow{9}\,\overleftarrow{5}$, $\overleftarrow{6}\,\overleftarrow{9}\,\overleftarrow{3}\,\overleftarrow{5}$, $\overleftarrow{9}\,\overleftarrow{6}\,\overleftarrow{3}\,\overleftarrow{5}$, $\overrightarrow{9}\,\overrightarrow{6}\,\overleftarrow{5}\,\overleftarrow{3}$, $\overrightarrow{6}\,\overrightarrow{9}\,\overleftarrow{5}\,\overleftarrow{3}$, $\overrightarrow{6}\,\overleftarrow{5}\,\overrightarrow{9}\,\overleftarrow{3}$, $\overrightarrow{6}\,\overleftarrow{5}\,\overleftarrow{3}\,\overrightarrow{9}$, $\overleftarrow{5}\,\overrightarrow{6}\,\overleftarrow{3}\,\overleftarrow{9}$, $\overleftarrow{5}\,\overrightarrow{6}\,\overrightarrow{9}\,\overleftarrow{3}$, $\overleftarrow{5}\,\overleftarrow{9}\,\overrightarrow{6}\,\overleftarrow{3}$, $\overleftarrow{9}\,\overleftarrow{5}\,\overrightarrow{6}\,\overleftarrow{3}$, $\overrightarrow{9}\,\overleftarrow{5}\,\overleftarrow{3}\,\overrightarrow{6}$, $\overleftarrow{5}\,\overrightarrow{9}\,\overleftarrow{3}\,\overrightarrow{6}$, $\overleftarrow{5}\,\overleftarrow{3}\,\overrightarrow{9}\,\overrightarrow{6}$, $\overleftarrow{5}\,\overleftarrow{3}\,\overrightarrow{6}\,\overrightarrow{9}$

   (c) $3\,5\,6\underset{i}{}\underset{i+1=j}{9}$, $3\,5\underset{i}{}\underset{i+1}{9}\,6\underset{j}{}$, $3\,6\,5\underset{i}{}\underset{i+1=j}{9}$, $3\,6\underset{i}{}\underset{i+1=j}{9}\,5$, $3\,9\,5\underset{i}{}\underset{i+1=j}{6}$, $3\underset{i}{}\underset{i+1}{9}\,6\,5\underset{j}{}$, $5\,3\,6\underset{i}{}\underset{i+1=j}{9}$, $5\,3\underset{i}{}\underset{i+1}{9}\,6\underset{j}{}$, $5\,6\,3\underset{i}{}\underset{i+1=j}{9}$, $5\,6\underset{i}{}\underset{i+1=j}{9}\,3$, $5\,9\,3\underset{i}{}\underset{i+1=j}{6}$, $5\underset{i}{}\underset{i+1}{9}\,6\underset{j}{}\,3$, $6\,3\underset{i}{}\underset{i+1}{9}\,5\underset{j}{}$, $6\,3\,5\underset{i}{}\underset{i+1=j}{9}$, $6\,5\underset{i}{}\underset{i+1=j}{9}\,3$, $6\,9\,3\underset{i}{}\underset{i+1=j}{5}$, $6\underset{i}{}\underset{i+1=j}{9}\,5\,3$, $9\,3\,5\underset{i}{}\underset{i+1=j}{6}$, $9\,3\underset{i}{}\underset{i+1}{6}\,5\underset{j}{}$, $9\,5\,3\underset{i}{}\underset{i+1=j}{6}$, $9\,5\underset{i}{}\underset{i+1=j}{6}\,3$, $9\,6\,3\underset{i}{}\underset{i+1=j}{5}$, $9\,6\,5\,3$

6. (10 points) Consider the pseudocode of the algorithm for generating permutations in the listing **Algorithm 1**:

   (a) Trace the algorithm by hand for $n = 2, 3$, and 4. [**Hint:** Try to reuse the work you did for $n = 2$ while you are working for $n = 3 \ldots$]

   [Solution]

   You will find that when n is odd, after $Permute(n)$ finishes, the array remains the same as it was at the beginning of the call $Permute(n)$. And when n is even, say $Permute(n)$ started with an array $\langle 1, 2, 3, \ldots, n-1, n\rangle$, then at the end the array becomes $\langle n-1, 1, 2, \ldots, n-2, n\rangle$, it actually cyclically right shifts the first $n-1$ positions.

   (b) Prove the correctness of $Permute(n)$. [**Hint:** Prove that if Permute$(n-1)$ works correctly, Permute(n) must also work correctly.]

   [Solution]

Algorithm 1 Permute(n)

```
1: {Input: A positive integer n and a global array A[1..n]}
2: {Output: All permutations of elements of A}
3: if n ← 1 then
4:    print A
5: else
6:    for i ← 1 to n do
7:       Permute(n − 1)
8:       if n is odd then
9:          swap A[1] and A[n]
10:      else
11:         swap A[i] and A[n]
12:      end if
13:   end for
14: end if
```

If $Permute(n-1)$ works properly, i.e., if it prints all $(n-1)!$ permutations correctly, then $Permute(n)$ in its **for** loop will make $n!$ prints. If we can prove that all these prints are distinct, then $Permute(n)$ also must work correctly.

When n is even, it makes n calls to $Permute(n-1)$ in its **for** loop. As n is even, $n-1$ must be odd. So after each call to $Permute(n-1)$ the array is left as it was at the beginning of that call and then occurs the "swap A[i] **and** A[n]". If we consider the n-th element of the array to be sitting idle during the first call to $Permute(n-1)$, then this swap following the end of $Permute(n-1)$ now picks the first element which remains idle during this second call to $Permute(n-1)$, and then the second element is chosen to remain idle during the third $Permute(n-1)$ call, and so on. Thus during each of this n calls to $Permute(n-1)$ inside the **for** loop of $Permute(n)$ a distinct element remains idle. So, all permutations must be distinct.

Now, if n is odd, $n-1$ is even, thus the observation in part(a) asserts that an array $\langle 1, 2, 3, \ldots, n-1, n\rangle$ turns into $\langle n-1, 1, 2, \ldots, n-2, n\rangle$ after the end of the first call to $Permute(n-1)$ and then "swap A[1] **and** A[n]" picks a different element as the idle one for each of the remaining iterations. Thus again, all the permutations must be distinct.

(c) What is the time efficiency of $Permute(n)$? [**Hint:** The pseudocode gives you the recurrence relation (number of swaps), solve it. You may need the formula $e \approx \sum_{i=0}^{n} \frac{1}{i!}$ for large n.]

[Solution]

The recurrence for the number of swaps is as follows,

$$T(n) = \begin{cases} n\,[T(n-1)+1] & \text{if } n > 1 \\ 0 & \text{if } n = 1 \end{cases}$$

Let us solve the recurrence using back-substitution:

$$\begin{aligned}
T(n) &= nT(n-1)+n \\
&= n[(n-1)T(n-2)+(n-1)]+n \\
&= n(n-1)T(n-2)+n(n-1)+n \\
&= n(n-1)(n-2)\ldots[n-(n-1)]T([n-(n-1)])+n(n-1)(n-2)\ldots[n-(n-1)]+ \\
&\quad n(n-1)(n-2)\ldots[n-(n-2)]+\ldots+n(n-1)+n \\
&= n!+n!+\frac{n!}{2!}+\frac{n!}{3!}+\ldots+\frac{n!}{(n-1)!} \\
&= n!\left(1+1+\frac{1}{2!}+\frac{1}{3!}+\ldots+\frac{1}{(n-1)!}\right) \\
&= n!\left(\sum_{i=0}^{i=n}\frac{1}{i!}-\frac{1}{n!}\right) \\
&\approx n!(e-0) \quad \left[\text{When } n \text{ is big } \frac{1}{n!} \text{ is small}\right]
\end{aligned}$$

Thus $T(n) \in \Theta(n!)$.

7. (10 points) Consider the following algorithm for searching in a sorted array $A[0..n-1]$. If $n = 1$, simply compare the search key K with the single element of the array; otherwise, search recursively by comparing K with $A\left[\lfloor\frac{n}{3}\rfloor\right]$, and if K is larger, compare it with $A\left[\lfloor\frac{2n}{3}\rfloor\right]$ to determine in which third of the array to continue search.

   (a) Set up a recurrence for the number of key comparisons in the worst-case. You may assume that $n = 3^k$.

   [Solution]

   The recurrence relation is,

$$T(n) = \begin{cases} T\left(\frac{n}{3}\right)+2 & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

   (b) Solve the recurrence for $n = 3^k$.

   [Solution]

$$T(n) = T\left(\frac{n}{3^k}\right)+\underbrace{2+2+\ldots+2}_{k\ 2's}$$

   As $k = \log_3 n$, $T(n) = 2\log_3 n$.

   (c) Compare the algorithm's efficiency with that of binary search.

   [Solution]

   Binary search has a worst-case complexity of $O(\lg n)$. As $\lg n$ and $\log_3 n$ are related by a constant factor, asymptotically binary search and the above algorithm have equivalent speed.

8. (10 points) You need to search for a given number in an $n \times n$ matrix in which every row and every column is sorted in increasing order. Can you design a $O(n)$ algorithm for this problem? Express the algorithm in pseudocode.

   [Solution]

   Algorithm 2 gives the pseudocode. In the worst-case, when the search path has a staircase pattern, the number of comparisons is $2(2n-1)$.

Algorithm 2 $SearchSortedMatrix(A[0..n-1, 0..n-1], K)$

```
1: {Input: An n × n matrix A[0..n − 1, 0..n − 1] whose every row and every column is sorted in increasing order and an integer K}
2: {Output: The row and column indices of the cell that contains a value equal to K, if no such cell exists, −1}
3: r ← 0
4: c ← n − 1
5: while r < n and c ≥ 0 do
6:    if A[r, c] = K then
7:       return ⟨r, c⟩
8:    else if A[r, c] < K then
9:       r ← r + 1
10:   else
11:      c ← c − 1
12:   end if
13: end while
14: return −1
```

9. (10 points) Suppose you are given an array A of n sorted numbers that has been *circularly shifted* k positions to the right. For example, $\langle 35, 42, 5, 15, 27, 29\rangle$ is a sorted array that has been circularly shifted $k = 2$ positions, while $\langle 27, 29, 35, 42, 5, 15\rangle$ has been shifted $k = 4$ positions.

   (a) Suppose you know what k is. Give an as-efficient-as-you-can (AEAYC) algorithm to find the largest number in A. Express the algorithm in pseudocode.

   [Solution] **Algorithm 3** gives the pseudocode.

Algorithm 3 $FindMax(A[0..n-1], k)$

```
1: if k = 0 then
2:    return A[n − 1]
3: else
4:    return A[(k − 1) mod n]
5: end if
```

   (b) Suppose you *do not* know what k is. Give an AEAYC algorithm to find the largest number in A. Express the algorithm in pseudocode.

   [Solution] **Algorithm 4** gives the pseudocode.

Algorithm 4 $FindMax(A[0..n-1])$

```
1: for i ← 0 to n − 2 do
2:    if A[i] ≥ A[i + 1] then
3:       return A[i]
4:    end if
5: end for
6: return A[n − 1]
```

10. (10 points) Given a set S of n integers and an integer t, give an $O\left(n^{k-1} \lg n\right)$ algorithm to test whether k of the integers in S add up to t. Express the algorithm in pseudocode.

    [Solution]

    Let us see what we can do when $k = 3$. We need a $O\left(n^2 \lg n\right)$ algorithm. Sort the array first with a $O(n \lg n)$ sorting algorithm. The n^2 inside the $O()$ indicates a nested loop and the $\lg n$ indicates one

binary search for each of these n^2 iterations. Binary search needs a key, K. Inside the second loop, we could subtract the sum of two numbers from t, to obtain K. We shall generalize this idea for any $k > 1$.

Algorithm 5 $TestSum(A[0..n-1], k, t)$

```
1: {Input: An array A[0..n-1], an integer k << n, and an integer t}
2: {Output: true if k elements of A[0..n-1] add up to t, false otherwise}
3: Sort(A[0..n-1])
4: for i ← 0 to k-2 do
5:    I[i] ← i
6: end for
7: while I[0] ≤ n-k do
8:    s ← 0
9:    for i ← 0 to k-2 do
10:      s ← s + A[I[i]]
11:   end for
12:   d ← t - s
13:   l ← I[k-2] + 1
14:   found ← BinarySeach(A[l..n-1], d)
15:   if found then
16:      return true
17:   else
18:      I[k-2] ← I[k-2] + 1
19:      if I[k-2] > n-k+2 then
20:         carry ← 1
21:         i ← k-3
22:         while i ≥ 0 and I[i] + carry > n-k-i do
23:            i ← i-1
24:         end while
25:         if i < 0 then
26:            return false
27:         else
28:            I[i] ← I[i] + 1
29:            for j ← i+1 to k-2 do
30:               I[j] ← I[j-1] + 1
31:            end for
32:         end if
33:      end if
34:   end if
35: end while
36: return false
```

Lines 17–32 maintain the indexing of the implicit $k-1$ nested loops (for the n^{k-1} part of the complexity).