1. Given an undirected graph G and a particular edge e in it, devise a lineartime algorithm O(m+n) that determines whether G has a cycle containing e.

Assume edge e is (u, v).

We remove the edge from the graph G, and use DFS search to see if v can visited from u. If we can reach v from u, then add edge (u, v) back, it certaintly form a cycle. This algorithm take O(m+n) times.

```
Explore(G, u, target):
  if u = target
    hasCycle <- true
  visited(u) <- 1
  for each edge (u, v) □ E
    if visited(u) = 0
      explore(G, u)

DFSSearch(G, u, target):
  for u □ G
    visited(u) <- 0
  explore (G, u, target)

DFSSearch(G', u, v) where (u, v) is e, and G' is G removing edge e.
```

2. Consider the following statement: if (u,v) is an edge in an undirected graph, and during the depth-first search post(u) < post(v), then v must be an ancestor of u in the DFS tree. Is this statement true or false? If it is true, prove it; otherwise, give a counterexample.

Yes.

We can categories the possibilities to four situations.

1. Visting u first and then v through edge(u, v)
2. Visiting u first and other nodes, then go u without traversing edge(u, v)
3. Visiting v first and then u through edge(u, v)
4. Visiting v first and other nodes, then go u without traversing edge(u, v)

For first case, if we visit v after u through edge (u, v), then post(u) > post(v) because we will back from u after finishing exploreing u. We don't need to handle this case because post(u) > post(v).

For second case, if we visit v after u without traversing edge(u, v). Instead, we visit other node x1, x2, ... xn, then to v. By induction on first case, post(u) > post(x1) > post(x2) .... > post(v). We can get post(u) > post(v). We don't need to handle this case either.

For third case, if we visit u after v immediately through edge (u, v), then post(v) > post(u) and clearly u is the ancestor of v.

For fourth case, if we visit u after v without traversing edge(u, v). Instead, Instead, we visit other node x1, x2, ... xn, then to u. By induction on third case, we can get post(v) > post(x1) > post(x2) ...> post(xn) > post(u), and v is ancestor of x1, x1 is ancestor of x2 ... and xn is ancestor of u. Therefore, we can get conclusion that v is ancestor of u.

3. The police department in the city of Axeville has made all streets oneway. The mayor contends that there is still a way to drive legally from any intersection in the city to any other intersection, but the opposition is not convinced. (1) Formulate a graph algorithmic problem to check whether this statement is true or not, and devise a linear time algorithm to solve it; (2) Suppose it turns out the mayor’s claim is false. She next claims something weaker: if you start driving from the town hall, then no matter where you reach, there is always a way to drive legally back to the town hall. Formulate this weaker statement, and devise a linear time algorithm to check if it is true or not.

The first problem is essential the problem to prove the whole city is a single strongly connected component.

We define a intersection is a vertex v and if there is a way to go from one intersection u to another intersection v then we have a edge (u, v). The we can model the whole city as a graph graph G(V, E).

To proof the whole city is a single SCC, we can runn DFS algorithm twice. First DFS find the sink of the graph. The second DFS start from sink in reversed graph. If second time we still can reach all the vertex in graph, then we can proof it is a single SCC.

In the second problem, we still use the same graph model as the first problem. This time, we firstly use DFS start from the town hall, and remove all intersections that is not reachable from this search. Then we check if such graph is a single SCC by reversed graph and running DFS again.

Both problem can be solved by two DFS, and DFS is a linear time algorithm. Therefore, both of them is also O(n)

4. Devise a linear O(m+n) time to solve the following problem: given a directed acyclic graph G (with n vertices and m edges), check if G has a directed path that visits every vertex once and only once.

This is a Hamiltonian path problem.

Firstly, Using DFS to sort all vertices in topological order, which takes time O(m+n).

Then we check if all successive vertices in the sort are connected. That is, we have edges (1, 2), (2, 3) (3, 4) ... (n-1, n), where the number is vertices sorted by topological order. If so, then we get a Hamiltonian path, which starting from the source, traversing all vertices in topological order and ending at the sink. This operation take O(n) times.

The running time is O(m+n) + O(n)

5. A feedback edge set of an undirected graph G=(V, E) is a subset of edges E’E that intersects every cycle of the graph. Thus, removing the edges in E’ will render the graph acyclic. Give an efficient algorithm for the following problem: Input: Undirected graph G=(V, E) with positive edge weights We Output: A feedback edge set E’E of minimum total weight Σ eE’ We

We know that if we remove all edges in feedback edge set, than the graph will become the spanning tree. If we want to get feedback edge set with min total weight, then it is to say we want to find maximum spanning tree.

We know how to find minimun spanning tree by using greeding algorithm like following code, because spanning tree form a graph matroid. (We proof it on the class, so we skip here)

```
A <- empty
sort e □ E in increasing order of w for each e □ E in the order
  if A U {e} □ l
    A <- A U {e}
```

Actually, we can use such algorithm to find maximum spanning tree too. We just need to sort edge by decreasing instead of increasing order of w, and run the above algorithm again. Now we start to add the originally largest weight edge into A step by step. Therefore, we can get maximum spanning tree. Then we can diff E with those tree edges and get minimum feedback edge set.

6. Given two arrays of n integers, the all-pair-sum S is defined as the sum of every pair of elements: S = Σi,jaibj, where ai and bj are the integers in the two respective arrays. Given an array of n integers A, we want to find an array of integers B, in which each element bj □ {1, -1}, such that the all-pair-sum between A and B is maximized. How to find array B?

Let the f(j) be the max sum of array A and the array B with first j element.

We know there is only two elements { 1 , -1 } in array B. Let ai is the element in array A, SP be the sum of `ai * 1`, and SN be the sum of `ai * -1`.

Then we can get the following relation

```
f(j+1) = f(j) + max { SP , SN }
```

However, max { SP , SN } is always the same, so we only compute SP, SN once, and based on the result to return array of all elements are 1 or array of all elements are -1.

7. Given two sorted arrays, each containing n integers, A[1..n] and B[1..n], and an integer N, we want to find two numbers a and b in each of these input arrays, respectively, a□A and b□B, such that |a-b|=N; if there are no such two numbers, a message "not found" should be output. Here, we want to avoid any additional storage with the size O(n). Devise an algorithm to find the two numbers in O(n) time under this constraint. (Note that you can still use additional constant memory as temporary storage.)

We need to extra memory for index i and j to record current position in array A and B, and we check the difference of B[i] - A[j].

Because both A and B is sorted, we can gradually scan all possibility by increase index i & j based on the difference of B[i] - A[j]. If difference < N, then we increase i; if difference > N, then we increase j.

```
i <- 1
j <- 1

f(A, B):
  while (diff > 0 && i < n && j < n)
    if ( B[i] - A[j] == N )
      return [B[i], A[j]]
    else if ( B[i] - A[j] < N )
      j++
    else
      i++

f(A, B)
// we switch A B and run it again
f(B, A)
```

Eg:

```
A: [2, 3, 4, 5]
B: [2, 3, 4, 9, 10]
N = 3


i=1, j=1, B[i] - A[j] = 0 < 3
i=1, j=2, B[i] - A[j] = 1 < 3
i=1, j=3, B[i] - A[j] = 2 < 3
i=1, j=4, B[i] - A[j] = 7 > 3
i=2, j=4, B[i] - A[j] = 6 > 3
i=3, j=4, B[i] - A[j] = 5 > 3
i=4, j=4, B[i] - A[j] = 4 > 3

// switch A B
i=1 j=1, A[i] - B[j] = 0 < 3
i=1 j=2, A[i] - B[j] = 1 < 3
i=1 j=3, A[i] - B[j] = 2 < 3
i=1 j=4, A[i] - B[j] = 3 == 3 return [5, 2]
```

8. Given an array of n positive integers and an integer N, we want to find if it has a consecutive subarray (i.e., a subarray with consecutive elements between a two positions) with the sum of N. Devise an O(n) algorithm to solve this problem.

Let f(i, j) be the sum from i element to j element in the array A.

We start from f(1, 1) = A[1].

if f(i, j) < N, we go f(i, j+1). if f(i, j) > N, we go f(i+1, j).

Eg:

```
A = [1, 2, 3, 4, 1, 2, 3]
N = 8
```

```
f[1, 1] = 1 < 8  =>
f[1, 2] = 3 < 8  =>
f[1, 3] = 6 < 8  =>
f[1, 4] = 10 > 8 =>
f[2, 4] = 9 > 8  =>
f[3, 4] = 7 < 8  =>
f[3, 5] = 8 == 8
```

The time complexity is 2n, because i and j both from 1 to n, and every step we increase i or j by 1.

9. Given k arrays each with n integers, devise a divide-and-conquer algorithm in O(nklog(nk)) time to find all integers that each appears at least once in each input array. You should use only comparison of the integers, but not use advanced data structures such as hash tables or counting arrays.

We can choose a pivot element "p" from the first array, and split all input array to 3 sub arrays. the {x < p} , {x = p}, {x > p}. This steps cost `n` times `k` operations. Then we check if all arrays have at least one elements in the { x = p } sub array. If so, we put p into final result.

And we resursively on subarrays for {x < p} and {x > p}, until one of array in the arrays set is empty.

The expected hight is log(n), just like the parition in quicksort.

I say the total cost can be nklog(n).

Eg:

```
A = [3, 2, 6, 1]
B = [1, 3, 4, 2]
C = [1, 1, 3, 1]

choose 3 as pivot element
split A = [ [2, 1], [3], [6] ]
split B = [ [1, 2], [3], [4] ]
split C = [ [1, 1, 1] [3] [] ]

Then we know 3 appear in all input arrays.

Because one array of the set of {x > 3} arrays is empty, we dont need to handle {x
> 3} case anymore.

On {x < 3} recursion,

A' = [2, 1]
B' = [1, 2]
C' = [1, 1, 1]

choose 2 as pivot element

split A' = [ [1], [2], []]
split B' = [ [1], [2], []]
split C' = [ [1, 1, 1] [], []]
```

```
Then we know 2 don't appear in all input arrays, so we discard it.

{x > 2} arrays has at least one empty array, we don't need to handle that.

On {x < 2} recursion

A'' = [1]
B'' = [1]
C'' = [1, 1, 1]

choose 1 as pivot element,

split A'' = [ [], [1], []]
split B'' = [ [], [1], []]
split C'' = [ [], [1, 1, 1] []]

we know 1 appear in all elements.

so the final result is [1, 3]
```

10. In an array of n integers A[1..n], the increasing subsequence is a subsequence of k consecutive elements in the array, A[i], A[i+1], ..., A[i+k], such that A[i] ≤ A[i+1] ≤ ...≤ A[i+k]. Devise a linear time O(n) algorithm to find the longest increasing subsequence of a given array of n integers.

let f(j) be the currently longest sequence that ends at j in array A. let E be the end index of the longest sequence, and L be the length of the longest sequence.

```
f(1) = 1
E = 1
L = 1

for(i = 2 ; i < length(A); i++){
  if A[i] >= A[i-1] {
    f(i)++;
    if(f(i) >= L){
      L = f(i);
      E = i;
    }
  }
  else  {
    f(i) = 1;
  }
}
```

11. The overlap between two intervals i and i' is defined as: if i □ i' ≠ ø, that is, if low[i] ≤ low[i'], overlap = max(0, high[i] - low[i’]); if low[i’] ≤ low[i], overlap = max(0, high[i’] - low[i]). Given a set of intervals S and a query interval q, we want to find the interval in S with the greatest overlap with q. Devise a data structure based on binary search tree (BST) to maintain the interval set S (so that they can be inserte/delected, etc), and a search algorithm using the data structure to solve the above problem in O(log n), where n = |S|. You may modify the BST data structure to incorporate additional auxiliary information.

We sort intervals by low of interval, and we need to store the max of all children to find the overlapy. In addition, we also need to store the min of all children to find maximum overlap.

There are four cases for overlapping.

Ths first case overlap, the query node is on the right side of current searching node. We need to check if max value is from left branch or right branch. If max from left branch, then we go left. Otherwise, we compare the high[x] - low[i] and high[i] - min[right[x]]. If high[x] - low[i] < high[i] - min[right[x]], then we go right, otherwise we return current node.

```
query:          |--------|
current node |-----|
```

Ths second case overlap, the query node is on the left side of current searching node. We need to check if max value is from left branch or right branch. If max from left branch, then we go left. Otherwise, we reutrn current node.

```
query:        |--------|
current node       |-----|
```

Ths third case overlap, the query node overshadow the current node. We need to check if max value is from left branch or right branch. If max from left branch, then we go left. Otherwsier, we need to compre high[x] - low[x] and high[i] - min[right[x]]. If high[x] - low[x] < high[i] - min[right[x]], then we go right, otherwise we return current node.

```
query:        |----------|
current node     |-----|
```

Ths fourth case overlap, the query node is overshadowed by current node. We return current node directly.

```
query:         |--------|
current node  |------------|
```

12. Given n integers, and a query integer k, we want to computer how many integers among the n integers are greater than k. Assuming we maintain a binary search tree (BST) of the n integers (so that they can be inserted/deleted, etc), we want to solve the above problem in O(h) time, where h is the height of the BST, and for a balanced BST, h=O(log n). Devise an algorithm in O(h) based on the BST of the n integers. You may modify the BST data structure to incorporate additional auxiliary information.

We need to maintain extra info: the number of children in each node.

The insert and delete operations are still the same as BST. The only difference is that, we need to update the children count upward to the root.

For example, the BST with children count is like this.

```
                          (20, 7)
        (14, 2)                              (28, 3)
(11, 0)         (16, 0)      (24, 0)                (31, 1)
                                                (29, 0)
```

For root, (20, 7) means the integer is 20 and this node has 7 children.

If we insert 32 into tree, then we need to increase the children count of node 31, 28, 20 by 1.

```
                          (20, 8)
        (14, 2)                              (28, 4)
(11, 0)         (16, 0)      (24, 0)                (31, 2)
                                                (29, 0)     (32, 0)
```

If we delete 29 from the node, we also need to decrease the children count of node 31 28 20 by 1.

```
                          (20, 6)
        (14, 2)                              (28, 2)
(11, 0)         (16, 0)      (24, 0)                (31, 0)
```

With such data structure, we now define the function f to calculate the number bigger than k at the root of tree T

```
f(T, k)
  if root[T] > k
    if right[T] != Nil
      f(left[T], k) + childrenCount[right[T]] + 2
    else
      f(left[T], k) + 1
  else
    if right[T] != Nil
      f(right[T], k)
    else
      0
```

Eg: We want to find the answer on above tree with k = 13.

At root f(20, 13), 13 < 20, so we go left. However, we need to add the children count of the right branch.

```
f(14, 13) + 3 + 2
```

Now 13 < 14, we go left again, and add right branch node count.

```
f(11, 13) + 0 + 2 + (3 + 2)
```

Now 13 > 11 and right branch is NIL, so we stop here and return 0.

```
0 + (0 + 2) + (3 + 2)
```

The answer is 7.

Eg2: We want to find the answer on above tree with k = 29.

At root f(20, 29), 29 > 20, so we go right.

```
f(28, 29)
```

Now 29 > 28, we go right again.

```
f(31, 29)
```

Now 29 < 31, so we go left. Besides, right branch is empty so we only need to add 1

```
f(29, 29) + 1
```

now 29 <= 29, so we need to go right, but it is Nil. Therefore we stop here.

```
0 + 1
```

The answer is 1.

Every step we go 1 layer deep, so the time complexity is O(h).

13. Given two set of elements, A with m elements and B with n elements (n ≥ m), devise an algorithm to check if A is a subset of B. Note that you can only compare the elements to tell if they are the same or not. What is the run time of your algorithm in big-O notation?

```
isSubset(A, B)
  for element a in A
    if not existedInSet(a, B)
      return false
  return true
```

```
existedInSet(a , B)
  for element b in B
    if a == b
    return true
  return false
```

isSubset run m times, and existedInSet is O(n) The time complexity is O(nm).

14. You want to schedule a subset of n given jobs on a resource. Each job i needs to run on the resource for ti hours, and has a benefit of bi. You cannot schedule more than one job on the resource at a time. Devise an algorithm to schedule the subset of the jobs on the resource for a total of N hours (N is given in addition to ti and bi).

Originally, I think we can sort jobs by bi/ti, and using greedy algorithm to solve it.

```
  f(N, jobs) :
    for ji in jobs which ti < N
      choose highest bi/ti  ji □ f(N-ti, jobs / ji)
```

However, this is wrong because we have time limit.

For example:

```
Job 1: t1: 8, b1: 16  b1/t1 = 2
Job 2: t2: 6, b2: 11  b2/t2 = 1.83
Job 3: t3: 4, b3: 7   b3/t3 = 1.75
N = 10
```

Choosing job 2 + job 3 is better than just allocate job 1.

So I think we need to solve this problem by dynamic programming.

```
f(N, jobs):
  let ji in jobs which ti < N
  max { f(N-ti, jobs / ji) , f(N, jobs / ji) }
```

15. Alice wants to organize a party and is deciding whom to call. She has n people to choose from, and she has made up a list of which pairs of these people know each other. She wants to pick as many people as possible, subject to two constraints: at the party, each person should have at least five other people whom they know and five people whom they don't know. Given as input the list of n people and the list of pairs who know each other, devise an algorithm to output the best choice of party invitees.

We define S be the set of people {p1, p2, p3, ... pi} that we want to invite.

Function K(S, pi) is the number of people the people pi in S knows. Function N(S, pi) is the number of people the people pi in S doesn't know.

We start from inviting all the people, that is S = {p1, p2, ... pn}

```
f(S) :
  if existed pi in S st ( K(S, pi) < 5 || N(S, pi) < 5 )
    return max { for pi which ( K(S, pi) < 5 || N(S, pi) < 5 ) f(S/pi) }
  else
    return S
```

The algorithm is quite straightforward, we invite all people at first. If someone don't fit the constraints, then we kick him/her off from invitation list. If there are several people don't qualify those two constraints, than we recursively check the subproblem that, which person we remove can get the biggest invitation list.

16. Consider the following variation on the Scheduling Problem. You have a processor that can operate 24 hours a day, every day. People submit requests to run daily jobs on the processor. Each such job comes with a start time and an end time; if the job is accepted to run on the processor, it must run continuously, every day, for the period between its start and end times. Given a list of n such jobs, your goal is to accept as many jobs as possible (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in n. You may assume for simplicity that no two jobs have the same start or end times.

Example. Consider the following four jobs, specified by (start time, end-time) pairs. (6 P.M., 6 A.M.), (9 P.M., 4 A.M.), (3 A.M., 2 P.M.), (1 P.M., 7 P.M.). The optimal solution would be to pick the two jobs (9 P.M., 4 A.M.) and (1P.M., 7 P.M.), which can be scheduled without overlapping

Firstly, we change the time format to 24 hours foramt. If a jobs cross 12 AM, such as (9 P.M., 4 A.M.) we take them as(21, 28), and (21, 28) is still overlap with (0, 4).

Moreover, we need to define relative starting time. For example, we can say the schedule start from 6 (6 A.M.). When we say this job is end early than another job, we will have a relative starting time in mind. For example, 9 P.M. is earier than 4 A.M. when relative starting time is 8 P.M..

Then we sort all jobs by finish time and using greedy algorithm to solve it.

```
max {
  for rst(relative ) from 0 to 23
    sj <- sort jobs by finish time related to rst
    A <- empty schedule
    for j in sj
      if j is not conficit with jobs in A
        A <- A U j
    return A
}
```

Now we prove it correctness.

We get the solutions by greedying algorithm is g1, g2, ... gk.

If our solutions is not the optimal, and suppose we have a set of jobs j1, j2, ... jm is a optimal solution with g1 = j1, g2 =j2, ... gi = ji for the largest possible value of i.

However, job gi+1 finish before ji+1 because our algo choose ths job finish first, then we can replace job ji+1 with gi+1, and solution is still feasible and optimal, but contradict "the largest possible value of i".