

CS240 Homework #1

Zhiqiang Xie 77892769

Problem 1

I represent the ascending order in Asymptotic notations:

$$2^{\sqrt{\log n}} = o(n(\log n)^3) = o(n^{4/3}) = o(n^{\log n}) = o(2^n) = o(2^{n^2}) = o(2^{2^n})$$

Problem 2

1. $f(n) = p + q * n$, where p, q are constants. Therefore, $f(n) = O(n)$
2. Treat n as a binary number, so that we only need to calculate the 1th, 2th, 4th... power of x . I'll illustrate a faster algorithm in Python code:

Python

```
1 def fastPower(x, n):
2     """
3     :type x: int != 0
4     :type n: int >= 0
5     :rtype: int
6     """
7     res = 1
8     while n:
9         if n&1:
10             res *= x
11         n >> 1
12         x *= x
13     return res
```

Thus, $g(n) = p_1 + q_1 * \lceil \log_2 n \rceil$, where p_1, q_1 are constants.

Therefore, $g(n) = O(\log n)$, $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$

Problem 3

I summarize and abstract the information about this problem below:

- n scenic spots and m roads mean a graph contains n vertices and m edges.
- Roads in one direction and no cycle mean the graph is a directed acyclic graph (DAG).
- Our task is to determine whether there exists a path which can traverse all vertices (since it's a DAG, no path could visit the same city twice or more)

We need two lemmas to start our procedures:

- Here are always one or more vertices with in-degree zero.
 - At least one city in our map cannot be reached from any other cities.

- Any sub-graph of a DAG is a DAG
 - Every time we have visited a city, the subsequent task is just the same.

We need to prepare an in-degree table first:

- To do this, we'll iterate all vertices once. Now we know the vertices with 0 in-degree.
- This preparation cost $\Theta(n)$ time and $\Theta(n)$ space.

Now we start our algorithm at the cities with 0 in-degree:

- Once we have more than one 0 in-degree city, we know it's impossible to find such a path to visit all cities. The program terminates.
- We remove the selected city from our map and deduct the in-degree of the cities being directed by one.
- Then we choose the cities with 0 in-degree from them. Based on the lemma 2, here must be at least one city to be chosen.
- Repeat the procedures until we traverse all cities or terminated in procedure 1.
- These procedures cost $O(m + n)$ time.

Finally, the time complexity is $O(m + n) + \Theta(n) \in O(m + n)$

Problem 4

I summarize and abstract the information about this problem below:

- n scenic spots and m roads mean a graph contains n vertices and m edges.
- The graph is undirected and connected.
- Our task is to remove a vertex from the graph and make it remain connected.

Now we start our algorithm at a random city:

- Set the vertex to be part of the spanning tree of the graph.
- Do BFS or DFS from the vertex we selected.
- Every time we find a vertex that is not in the spanning tree, add to the spanning tree both the new node and the edge.

Since the graph is connected, we'll get a spanning tree after the procedure above.

- The procedure costs $O(m + n)$ time.
- Then we choose a leaf from the spanning tree and the vertex is our answer.
- Choose a leaf cost $O(n)$ time (worst case, $O(\log n)$ is more general)

Finally, the time complexity is $O(m + n) + O(n) \in O(m + n)$

Problem 5

Proof part:

- Firstly, we can assume that no such node v exists.
- In this case, there must be two totally different paths from node s to node t with the length strictly greater than $n/2$.
- However, it requires the number of nodes $n > (n/2 + 1) * 2 - 2 = n$, which is impossible.
- Therefore, there must exist some node v , not equal to either s or t , such that deleting v from G destroys all $s - t$ paths.

Algorithm part (some details in implementation are omitted):

- Start two BFS from s and t respectively.
- When the two BFS meet at just one node, we get the node v .
- The procedure costs $O(m + n)$ time.

Problem 6

Greedy Algorithm:

- Sort the children by their expected sizes and the cakes by the size of piece in ascending order.
- We scan the list of children from the one with the lowest expected size and the list of pieces of cake from the one with the smallest size. Once a piece of cake satisfies a child, we move the child to another list called S and remove all pieces of cake being scanned.
- Repeat the above procedure, finally output the length of S while the list of children or cake become empty.

Proof:

- Suppose that there exists an optimal solution which satisfies a child with higher expected size than an unsatisfied child.
- If we take the cake for the child with large expected size to the unsatisfied child, the number of satisfied children doesn't change. Thus, we get another optimal solution.
- Repeat the process and we'll find our greedy method in selecting children is optimal.
- Suppose that there exists an optimal solution which contains inversions (larger piece of cake is used to satisfy a child with less expected size, and smaller piece of cake is used to satisfy a child with higher expected size)
- If we invert the two pieces of cake, we'll see both are still satisfied. Thus, we get another optimal solution.
- Repeat the process and we'll find our greedy method in picking cake is optimal.

Problem 7

Greedy Algorithm:

- Sort the set of n points $\{x_1, x_2, \dots, x_n\}$ to get a new set $Y = \{y_1, y_2, \dots, y_n\}$ such that $y_1 \leq y_2 \leq \dots \leq y_n$.
- Then we scan the set started from y_1 . Everytime while encountering y_i ($i \in \{1, \dots, n\}$), we put the closed interval $[y_i, y_i + 1]$ in our optimal solution set S , and remove all the points in Y covered by $[y_i, y_i + 1]$.
- Repeat the above procedure, finally output S while Y becomes empty.

Proof:

- Suppose that there exists an optimal solution S' , such that y_1 is covered by $[x', x' + 1] \in S'$, where $x' < y_1$. Since y_1 is the leftmost element of the sorted set, there is no other point lying in $[x', y_1)$.
- If we replace $[x', x' + 1]$ in S , by $[y_1, y_1 + 1]$, we will get another optimal solution.
- Repeating solving the remaining subproblem (repeating comparing following interval), we'll find our greedy strategy lead to optimal.

Problem 8

Since now we have an MST T of the graph G , we'll create a cycle in T when we directly add the edge (v, w) to it. From the procedures of Kruskal's algorithm, we know:

- If the newly added edge has the maximum cost among the edges of the cycle, then the MST remains as it is.
- If some other edge of the cycle has the maximum cost, then that's the edge to be removed to get a tree with lower cost.

Therefore, our task is to determine the circle and the edge with the maximum cost. Besides, we know that the circle contains $u, v, (u, v)$.

Therefore, we can simply find a path from u to v in T and record the cost of each edge of it.

- Since T is a tree, here's only one path from u to v and we can apply BFS or DFS with backtracking.
- As for a tree, $O(E) = O(V)$, this method can test the MST in $O(V)$ time without any more assumptions.

However, we can take use of the features of a tree to make it better:

- Assumptions:
 - Every node in T has a parent-child relation with the nodes they connected.

We can also implement this job in $O(V)$ time.

- Traverse u, v to the root node, and join the two paths to get what we want.
- This procedure can be terminated if one of them is the ancestor of another. Generally, it costs $O(H)$ time where the H is the height of T , which is often far less than V . In the worst case, it costs $O(V)$ time.

Furthermore, it's possible to accelerate the procedure to find the least common ancestor(LCA) with some more preprocessing:

- Assumptions:
 - Every node in T has a parent-child relation with the nodes they connected.
 - Precompute the 2^j th ancestor for all valid j for each node in T .
 - Put the result above into a sparse 2D table P such that $P[i][j]$ stores the 2^j th ancestor for node i .

This preprocess costs $O(V \log V)$ time and space.

- Every number can be expressed as a *sum* of distinct *powers* of 2 .
- Hence, we may use our table P to keep raising our nodes by distinct powers of 2 to find their LCA.
- Some details about the procedures above is omitted.
- In this way, we can find the LCA in $O(\log V)$ time.

Once we find the LCA of u, v , the rest time cost will depend on the circumference of the cycle. In the worst case, it costs $O(V)$ still.