HW2.

Q1. a) fork: A fork is a "copy" of a repository

b) clone: brings a copy of the specified repository onto your local machine

c) push: uploads local updates to the remote repository

d) pull: downloads commits from your remote repository

e) remote: remotes indicates remote locations where we can push and pull data from. A remote is an association of a name to a repo URL

f) develop branch: serves as an integration branch for features

g) master branch: default branch to store the official release history.

h) feature branch: created from develop everytime developers start work on a new feature

i) release branch: contains production ready new features and bug fixes that come from stable develop branch.

j) hotfix branch: used to quickly patch production release; fork directly off of master branch.

k) fetch: downloads, but not merge them with your current branch.

l) branch: useful when you want to make individual changes on a collaboration project.

m) cherry-pick: enables arbitrary git commits to be picked by reference and appended to the current working HEAD.

n) revert: reverts a commit that has already been pushed and made public

o) rebase: allows one to move the first commit of a branch to a new starting location.

p) diff: shows changes between commits, commit and working tree, etc.

q) bisect: uses binary search to find the commit that introduced a bug.

Q2. (a): $T(n) = cn + cn/4 + cn/16 + \cdots + O(1) \le cn + cn/2 + cn/4 + cn/8 + \cdots + c$

$= cn(1 + \frac{1}{2} + \frac{1}{4} + \cdots) = 2cn$

$\Rightarrow T(n) = O(2cn) = O(cn)$

Q3. 2^x is a positive increasing function for any x

We have $f(n) = O(g(n)) \Rightarrow \exists\, n_0 > 0,\ c_1 > 0$ s.t. $\forall\, n \geq n_0\ f(n) \leq c_1 \cdot g(n)$

$\Rightarrow \exists\, n_0 > 0,\ c_1 > 0$ s.t. $\forall\, n \geq n_0,\ 2^{f(n)} \leq 2^{c_1 \cdot g(n)}$

But this does not necessarily lead to $2^{f(n)} \leq c_2 \cdot 2^{g(n)}$.

Counterexample: $f(n) = n^2$, $g(n) = n$, obviously $2^{n^2} \neq O(2^n)$

Q4. If the array is unsorted, in the worst case we may have to iterate all elements in the array. So the time complexity is $O(n)$.

Q5. If the array is sorted, we can use binary sort with time complexity $O(\log n)$. or we can use merge sort with time complexity $O(n \log n)$ or use bubble sort time complexity $O(n^2)$

Q6. $\sqrt{\log n}$, $2^{\sqrt{2\log n}}$, $(\sqrt{2})^{\log n}$, $4^{\log n}$, $n!$

$n!$ is definitely the one with fastest growth rate

$a^{\log n}$ grows faster than $a^{\sqrt{\log n}}$, so both $(\sqrt{2})^{\log n}$ and $4^{\log n}$ are faster than $2^{\sqrt{2\log n}}$

Exponential grows faster than linear function, so $2^{\sqrt{2\log n}}$ is faster than $\sqrt{\log n}$

Q7. Using quick sort, average time complexity is $O(N \log 2k)$

In this case is $O(700 \log 400)$

```
func Topk(arr, k):
    // corner case
    if (k <= 0)  return []
    if len(arr) <= k  return arr

    (arr1, arr2) = Partition(arr)
    return Topk(arr1, k).Append(Topk(arr2, k - len(arr1)))
```

```
func Partition(arr):
    arr1 = [] , arr2 = []
    rand = random number choosen from [0, len(arr)]
    swap( arr[1] , arr[rand] )
    p = arr[1]
    for i from 2 to len(arr):
        arr[i] > p ? arr1.Append(arr[i]) : arr2.Append(arr[i])
    len(arr1) < len(arr2) ? arr1.Append(p) : arr2.Append(p)
    return (arr1, arr2)
```

Q8. Suppose the size of dataset is n.

(1) Linear search for unsorted array : time complexity is $O(n)$

(2) Sort then binary search : time complexity is $O(n\log n) + O(\log n)$

When only searching one time : $O(n) < O(n\log n) + O(\log n)$, which means it's better to do linear search ;

When we need to search multiple times : $kO(n) > O(n\log n) + kO(\log n)$, which means it's better to sort the array then do multiple binary search.

Q9. TDD is a technology to ensure other software components can work properly. Before implementing other skills , TDD should be used to generate test case for upcoming situation.

Q10. a) Problem in page 37 asks us to find which two nodes have the shortest distance among n nodes. To find shortest distance, we must know distance between every two nodes and update shortest distance. So we need to go over $\binom{n}{2} = \frac{n(n-1)}{2}$ times $= O(n^2)$

b) Problem in page 38 : There are n sets $S_1, S_2, \cdots S_n$, each of which is a subset of $\{1, 2, \cdots, n\}$, the problem asks us to find whether some pair of these sets has no elements in common.

It takes $\binom{n}{2} = O(n^2)$ to choose two sets from $S_1, S_2, \cdots S_n$, and it takes linear

time for every pair of sets to determine whether p in S_i also belongs to S_j. So the total time is $O(n^3)$

c) Problem in page 39: A set of nodes is independent if no two are joined by an edge. So for a fixed constant k, the problem ask us to judge if a given n-node input graph has an independent set of size k.
It takes $\binom{n}{k} = O(n^k)$ to choose k elements from n elements. Besides, it takes $\binom{k}{2} = \frac{k(k-1)}{2}$ to check if those k nodes has edge with each other. Since k is a constant, the total time is $O(n^k)$.

d) The problem asks us to calculate the subsets of nodes of n-node graph.
$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n} = 2^n$

e) The problem asks us to calculate the time complexity of searching for a largest independent set using naive way.
The time it takes to iterate all subsets of the graph is 2^n, and it takes at most $O(n^2)$ to check all pairs from a n-node set. Therefore, the total time complexity is $O(n^2 \cdot 2^n)$.