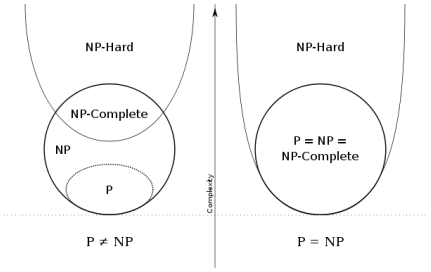
Solver: Zhou Jingyuan

Email Address: zhou0235@e.ntu.edu.sg

1. (a)
   (i) True. $f(n) \in \Omega(g(n))$ implies that f(n) will always be less than g(n) if n is large enough
   (ii) False. Counter example: $f(n) = n, g(n) = n^n, f(n) \notin \Theta(g(n))$
   (iii) True.
   (iv) False. NP is Non-*Deterministic* Polynomially Bounded Problems
   (v) True.



   (b)
   (i) Best case: 1 comparison, where the first pair of the number is not in order
   Worst case: n-1 comparison, where the array is indeed sorted
   (ii) The average number of comparison is give by

$$(n-1)*(1-p)^{n-1} + p*\sum_{i=1}^{n-1} i*(1-p)^{i-1}$$

   Where first term represents the average number of comparison needed if the result is true.
   The second term represents the average number of comparison if the result is false.
   Further simplification of the above expression is possible, but not required.
   (c)

```
bool RecSorted (int array[], int start, int end){
        if (start+1=end) return true;
        int mid = (start+end)/2;
        if (array[mid-1]>array[mid]) return false;
        if (!RecSorted(array,start,mid)) return false;
        if (!RecSorted(array,mid,end)) return false;
        return true;
    }
```

   (d)

$$f(n) = 1 + 2*f\left(\frac{n}{2}\right)$$

   solve the equation, we can easily get $f(n) = n = 2^k$

2. (a)
   (i) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\frac{\log_e(n)}{\log_e(10)}}{\log_e(n)^e} = \lim_{n\to\infty} \frac{1}{\log(10)*\log(n)^{e-1}} = 0$

$$\therefore f(n) \notin \Omega(g(n))$$
$$f(n) \in O(g(n))$$
$$f(n) \notin \Theta(g(n))$$

   (ii) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3^n}{e^n} = \lim_{n\to\infty} (\frac{3}{e})^n = \infty$ Note: $e \approx 2.7$

$$\therefore f(n) \in \Omega(g(n))$$
$$f(n) \notin O(g(n))$$

$$f(n) \notin \Theta(g(n))$$

(b)True. Given

$$f(n) \leq c_1 * h(n) \ \ for\ all\ n \geq n_1$$

$$g(n) \leq c_2 * h(n) \ \ for\ all\ n \geq n_2$$

We could deduce that

$$f(n) + g(n) \leq (c_2 + c_1) * h(n) \ \ for\ all\ n \geq \max(n_1, n_2)$$

Thus

$$f(n) + g(n) \in O(h(n))$$

(c)

A hash function MUST return a value within the hash table range.

It should achieve an even distribution of the keys that actually occur across the range of indices

It should be easy and quick to compute.

The function should not be used as a hash function as the range of the function falls in 0..10 which requires table of size 11 yet the table is only of size 10.

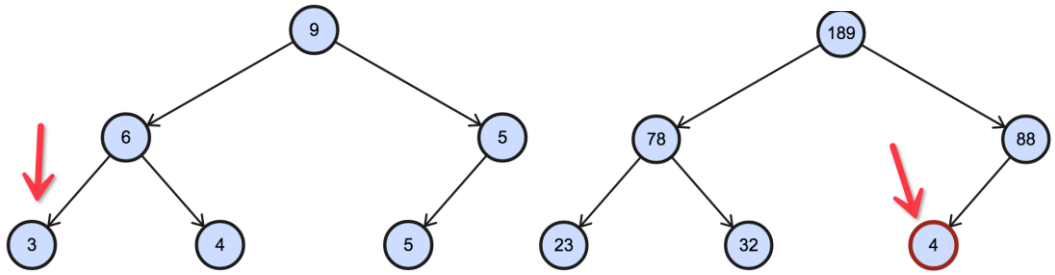
(d)

Index	Value
0	
1	
2	
3	
4	4 ~~17~~
5	
6	17
7	33 ~~20~~
8	20
9	
10	
11	
12	38

(e)

Value	Searched indices	Search times
38	12	1
20	~~7~~ 8	2
10	10	1

3. (a) The smallest element could be located on the leaf nodes.



(b)

(i) A strict descending array, where each element is strictly less than the next element. e.g. 6 5 4 3 2 1. For such array, the number of inversion : $f(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$

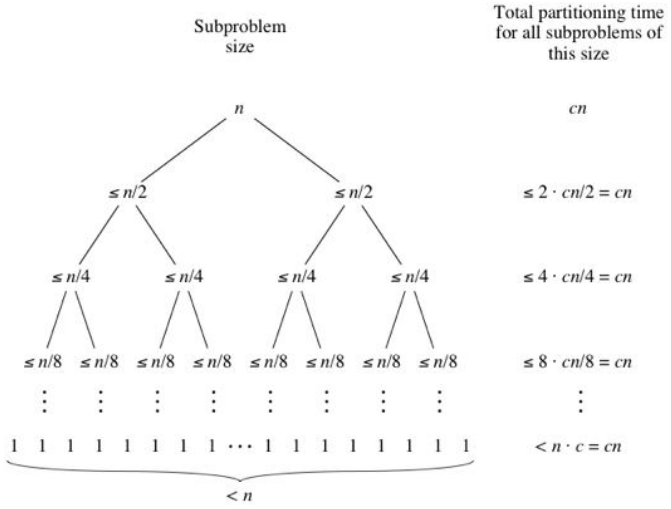
(ii) Consider the i[th] element in an insertion sort, to move the element into its correct position, each element (from 1 to i) that is greater than the i[th] element will have to move backwards by one position. Thus each inversion will correspond to one element movement.

(c) There are two methods available depending on the memory limitation.

Method 1	Detail	Single step Complexity	Repetition	Total Complexity
Step 1	for each element in S, compute $S_i' = x - S_i$	O(1)	n	O(n)
Step 2	Sort array T using quick sort	O(nlgn)	1	O(nlgn)
Step 3	For each S_i', use binary search to find whether the element exist in T	O(lgn)	n	O(nlgn)
Total	No addition memory requirement	-	-	O(nlgn)

Method 2	Detail	Single step Complexity	Repetition	Total Complexity
Step 1	for each element in S, compute $S_i' = x - S_i$	O(1)	n	O(n)
Step 2	Put S' into a hash table	O(1)	n	O(n)
Step 3	For each S_i', use hash table to search whether the element exist in T	O(1)	n	O(n)
Total	Require additional memory for hash table	-	-	O(n)

(d) Quicksort's best case occurs when the partitions are as evenly balanced as possible: their sizes either are equal or are within 1 of each other. The former case occurs if the subarray has an odd number of elements and the pivot is right in the middle after partitioning, and each partition has (n-1)/2 elements. The latter case occurs if the subarray has an even number n of elements and one partition has n/2 elements with the other having n/2-1. In either of these cases, each partition has at most n/2 elements, and the tree of subproblem sizes looks a lot like the tree of subproblem sizes for merge sort, with the partitioning times looking like the merging times:



Using big-Θ notation, we get the same result as for merge sort: $\Theta(nlgn)$

4. (a)
   (i) Assume 450k edges are NOT bi-directional. The full graph will have 1000*999 = 999,000 edges. The utilization rate of an adjacency matric is around 45%(450/999). Even if using adjacency list will incur some overhead for pointing to the next edge (say 50%), the pros still outdone the cons. in this case adjacency list is preferred.
   Assume 450k edges are bi-directional. Utilization rate = $\frac{450{,}000}{\frac{1000*999}{2}} \approx 90\%$ In this case adjacency matric is preferred.
   (ii) Regardless of the property of the edges. The graph is a spares graph (utilization rate = 2,000/ (500*999) <1%), adjacency list is preferred.
   (iii) For adjacency list, in the worst case we need to traverse n-1 edges to determine the adjacency of a given pair of vertices. Time complexity is of O(n)
   Whereas adjacency matric, the time complexity is of O(1)
   Thus, for the time complexity and the space is not a concern, adjacency matric is preferred.

(b) Since the spanning tree represents the search tree for breadth-first search. By the property of BFS, it gives the shortest (in terms of edges) page from s to every other vertex, in this case, L edges. Thus, all other any other path in from s to u contains at least L edges.

(c) use $\log(w_i)$ instead of w_i when computing minimum spanning tree using prim. Note: w_i : the weight of edge i.

For the final result, output the same tree and $e^{\text{final cost}}$ as the cost.

$$ab = e^{\log a + \log b}$$

(d) Pre-process:

1. drop all edges with success rate of 1
2. construct a new graph, where the probability of an edge with success rate of p_i becomes an edge with weight

$$\log 1 - p_i$$

Then run Dijkstra's algorithm on the new graph, the shortest path is the most reliable path.

Key considerations:

i. Use logarithm to convert the multiplication into addition

ii. Dijkstra can only solves for the minimal, and the edge have to be non-negative, have to convert the success rate into failing rate

For reporting of errors and errata, please visit pypdiscuss.appspot.com
Thank you and all the best for your exams! ☺