Solver: Liu Mingyu

Whenever you are in doubt, check with course notes. It is very important to adopt the convention from tutorial problem set in order to excel in exams.

1) a) i) 14. This is easily computed by going through the algorithm with the given input.

ii) Best case happens when there's no 'B' in the array, or 'B' is the last element of the array. In either case, the algorithm will scan the array once, which yields a runtime of *O(n)*.
Worst case happens when all elements (with possible exception of the last element) of the array are 'B'. In this case, runtime will be

$$\sum_{i=1}^{n-1}\sum_{j=i}^{n} 1 = O(n^2)$$

iii) From given information, we can see that $P(B) = \frac{1}{3}, P(E) = \frac{1}{3}, P(X) = \frac{1}{3}$ where $P(\delta)$ denotes the probability of the appearance of δ.
Thus, with any $0 \leq i \leq n-1$, the number of comparisons is

$$P(B) \cdot \sum_{j=i}^{n} 1 + \big(1 - P(B)\big) \cdot 0 = \frac{1}{3}\,(n - i + 1)$$

The total runtime is

$$\sum_{i=1}^{n-1} \frac{1}{3}(n - i + 1) = \frac{(n+2)(n-1)}{6}$$

b) i)

```
int ternarySearch(int E[], int first, int last, int k) {
      if (last < first) return -1;
      else {
            int firstThird = (2*first + last) / 3;
            int secondThird = (first + 2*last) / 3;

            if (k == E[firstThird]) return firstThird;
            else if (k == E[secondThird]) return
            secondFirst;
            else {
                  if (k < E[firstThird]) {
                        return tenarySearch(E, first,
                        firstThird - 1, k);
                  }
                  else if (k < E[secondThird]) {
                        return ternarySearch(E,
                        firstThird, secondThird -1, k);
                  }
                  else {
                        return ternarySearch(E,
                        secondThird, last, k);
                  }
```

```
                }
            }
        }
```

ii) In the worst case, item k is greater than all elements in array E.

$$T(i) = T\left(\frac{i}{3}\right) + O(1)$$

iii) $T(i) = T\left(\frac{i}{3}\right) + O(1)$

$= T\left(\frac{i}{3^2}\right) + 2O(i)$

$= T\left(\frac{i}{3^d}\right) + dO(1)$ for some integer d.

Choose $d = \log_3 n$,

$T(n) = T(1) + O(\log_3 n) = O(\log_3 n)$

iv) Yes.

$$\lim_{n\to\infty} \frac{\log_3 n}{\log_2 n} = \frac{\frac{\log_2 n}{\log_2 3}}{\log_2 n} = \log_3 2 \approx 0.631$$

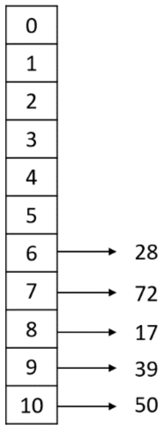
2) a) i) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log_2 n}{\log_4 n^4 + n} \approx \lim_{n\to\infty} \frac{\log_2 n}{\log_4 n^4} = 1$

Therefore, $f = \theta(g)$

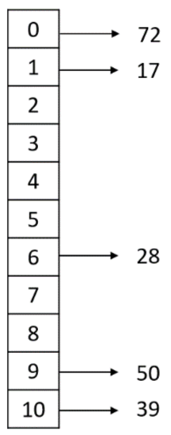
ii) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2^n}{\sqrt{3}^n} = \lim_{n\to\infty} \left(\frac{2}{\sqrt{3}}\right)^n = \infty$

Therefore, $g = O(f)$

b) i)



by linear probing

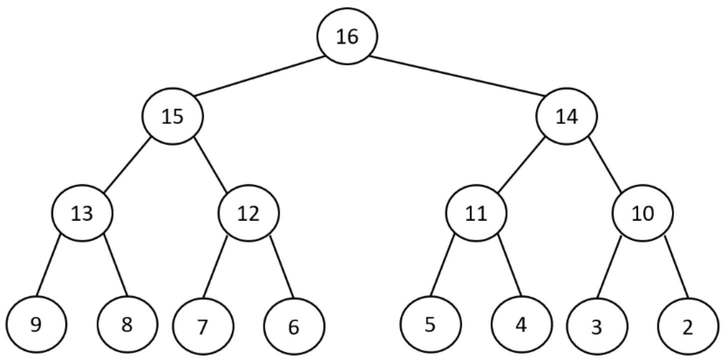
ii) 

by double hashing

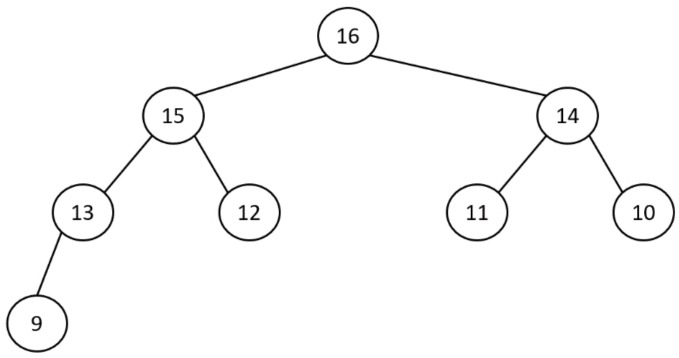
c) **To find 39:**
Linear probing = 4
Double hashing = 2
To find 61:
Linear probing: 5 (assuming checking null does not count as one comparison)
Double hashing: 1

d) Because double hashing is more likely to avoid clashes with the help of the second hash function. This advantage over linear probing is more obvious when multiple keys are hashed to the same key.

3) a) From the question, we may perceive that the initial array is somehow almost sorted already. Hence, the most suitable algorithm to fix the remaining errors is Insertion Sort. This is because for every correctly-ordered key value, we are only required to do 1 key comparison, that is with the previous key. Furthermore, every inversion can be solved in a constant time as we only need to compare the two values and swap. This algorithm on the given array will bring down the time complexity to Θ(n).

b) 

and

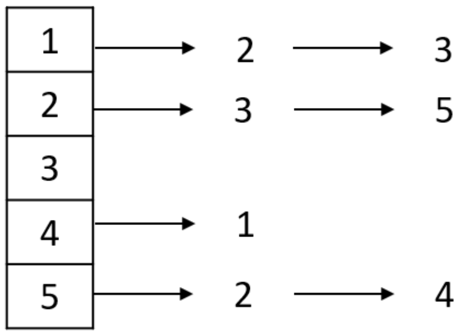


c) One way I can think of doing this is to merge A and B into C, then remove duplicates from C. In the worst case (e.g. A[1, 3, 5], B[2, 4, 6]:
   Merging takes n-1 time.
   Removing duplicates takes n-1 time.
   In total $n - 1 + n - 1 \in O(n)$

d) Insertion sort and merge sort are stable. Quicksort is not stable, because partition is not stable. For example:
   Run partition on 3 in **| 3 | 5 | 1_a | 1_b |**
   The result will be **| 1_b | 1_a | 3 | 5 |**
   It is clear that the order of two 1s are not preserved.

e)
```
Step 1: construct a max heap        // $O(n)$
Step 2: for i = 1 to k - 1          // $O(k-1)$
    deleteMin()                     // $O(1)$
    fixHeap()                       // $O(\log(n))$
Step 3: return deleteMin()          // $O(1)$
```
The time complexity is $O(n) + (k-1)O(\log(n)) + O(1) = O(n + klog(n))$

4) a) i)

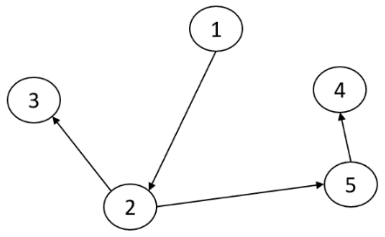


ii) *I am very uncertain of this question. I cannot remember how I handled this question during exam, so I regret that I might not be able to provide sufficient help. However there should be some similar exercise in tutorial problem set.*

Vertex label: 1 word

List reference: 1 word
Edge weight: 1 word
I would go for $12 \times 1 + 7 \times 1 + 7 \times 1 = 26$ words.

b)



Visiting order: 1 → 2 → 3 → 5 → 4

c) This statement is **NOT** true.
Consider a directed graph with 4 vertices, with $l(s,a) = 1, l(a,b) = 1, l(b,t) = 1, l(s,t) = 4$ where $l(u,v)$ denote the length from vertex u to vertex v.
We can easily tell that the statement is false.

d) Just run Dijkstra's algorithm on every vertex. Terminate the program when a path reaches t. You may write more details in your pseudocode.

e) The statement is true. We prove by <u>induction</u>.
For graph $G = (V, E)$ where all edge weights are different:

<u>Base case</u>:
$|V| = 2$. Two vertices and the edge with lowest weight forms the MST.

<u>Inductive step</u>:
Suppose when $|V| = k (k \geq 2)$, the statement holds.
Then, when $|V| = k + 1$:

$$New\ MST = Old\ MST \cup \{New\ vertex\} \cup \{Edge\ contects\ new\ vertex\ to\ old\ MST\ with\ lowest\ weight\}$$

Since all edge lengths are different, there is only one such new MST possible.

Hence, the statement is true.

--End of Answers--