2024 Algorithm Mid-Term Reference Answer

1. \(O(n^2)+\Theta(n^{2})=\Theta(n^2)\)

   briefly example: \(2n+3n^2=\Theta(n^2)\)

2. Note that both `BUILD-HEAP` and `HEAPIFY` must be described.

   (a)

```
MAX-HEAPIFY(A, i)
    // Input: A: an array where the left and right children of i root heaps (but i may not), i: an array index
    // Output: A modified so that i roots a heap
    // Running Time: O(log n) where n = heap-size[A] − i
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ≤ heap-size[A] and A[l] > A[i]
 4        largest ← l
 5  else largest ← i
 6  if r ≤ heap-size[A] and A[r] < A[largest]
 7        largest ← r
 8  if largest ≠ i
 9        exchange A[i] and A[largest]
10        MAX-HEAPIFY(A, LARGEST)
```

```
BUILD-MAX-HEAP(A)
    // Input: A: an (unsorted) array
    // Output: A modified to represent a heap.
    // Running Time: O(n) where n = length[A]
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3        MAX-HEAPIFY(A, i)
```

   (b)

Tighter Bound: Each call to MAX-HEAPIFY requires time $O(h)$ where h is the height of node i. Therefore running time is

$$\sum_{h=0}^{\log n} \underbrace{\frac{n}{2^h+1}}_{\text{Number of nodes at height } h} \times \underbrace{O(h)}_{\text{Running time for each node}} = O\left(n\sum_{h=0}^{\log n}\frac{h}{2^h}\right)$$

$$= O\left(n\sum_{h=0}^{\infty}\frac{h}{2^h}\right)$$

$$= O(n) \qquad (1)$$

Note $\sum_{h=0}^{\infty} h/2^h = 2$.

3. Selection Problem:

   (a)

   The complexity of the selection algorithm:

$$T(n) \le \begin{cases} O(1) & \text{if } n < 140\,, \\ T(\lceil n/5\rceil) + T(7n/10+6) + O(n) & \text{if } n \ge 140\,. \end{cases}$$

with substitution method:

$$\begin{aligned} T(n) &\le c\lceil n/5\rceil + c(7n/10+6) + an \\ &\le cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an)\,, \end{aligned}$$

which is at most cn if

$-cn/10 + 7c + an \le 0\,.$

Note that it's possible to prove this using a simplified version:

$T(n) = T(n/5) + T(3n/4) + O(n)$

and derive the upper bound using recursion tree method:

$T(n) \le cn(1 + (19/20) + (19/20)^2 + \ldots + (19/20)^k) \le cn(1/(1 - 19/20)) = O(n)$

(b)

Note that "The lower bound $\Omega(n)$ of selection problem" indicates that the worst case of any algorithm must take at least linear time.

Since even the simplest special case, such as finding the largest (or smallest), requires at least $\Theta(n)$ comparisons, that means no algorithm's cost can be lower than this case, indicating that the lower bound of selection problem is $\Omega(n)$

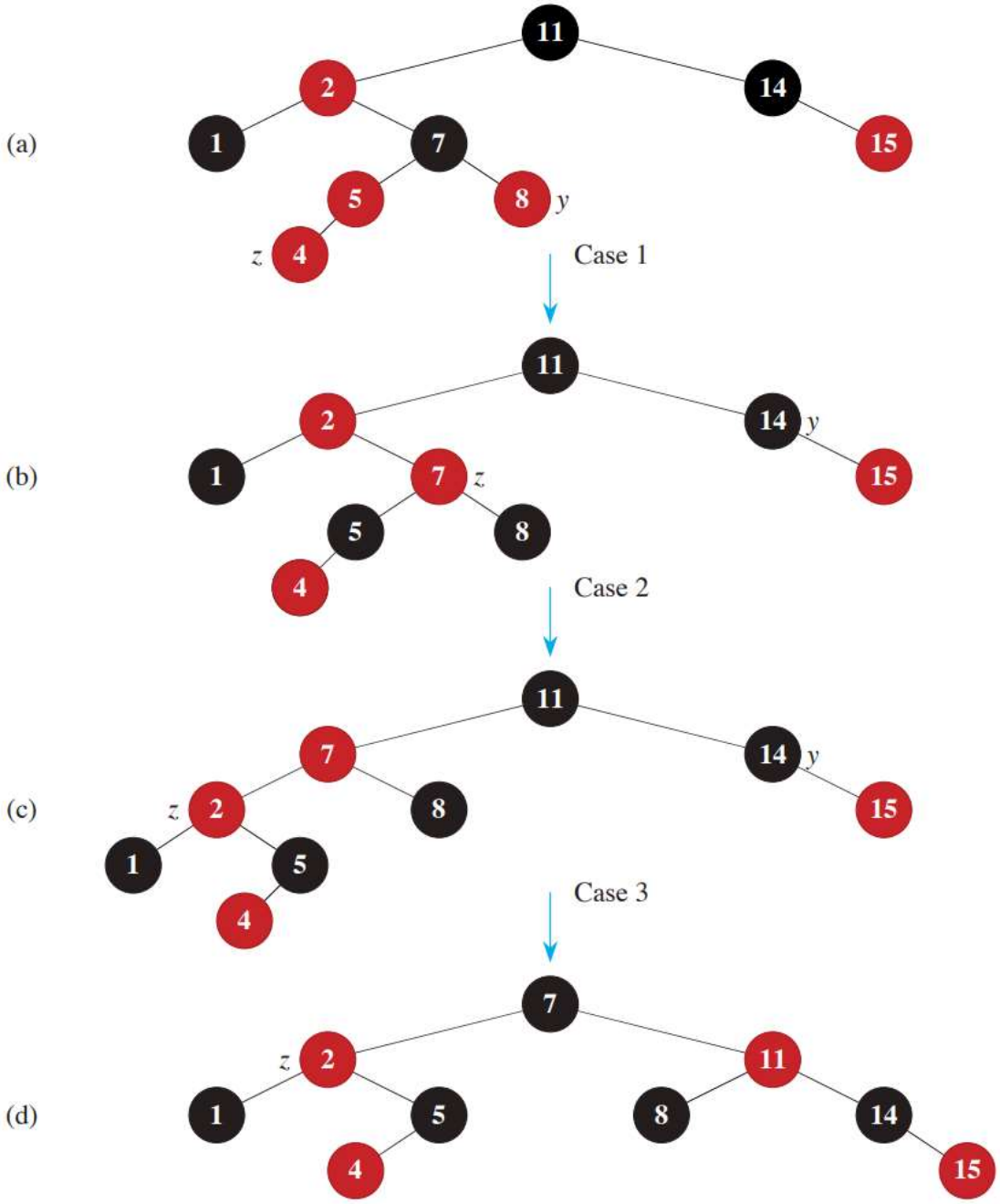
4. (a)

> Consider a red-black tree formed by inserting n nodes with $\mathrm{RB\text{-}INSERT}$. Argue that if $n > 1$, the tree has at least one red node.

- **Case 1:** z and $z.p.p$ are RED, if the loop terminates, then z could not be the root, thus z is RED after the fix up.
- **Case 2:** z and $z.p$ are RED, and after the rotation $z.p$ could not be the root, thus $z.p$ is RED after the fix up.
- **Case 3:** z is RED and z could not be the root, thus z is RED after the fix up.

Therefore, there is always at least one red node.

(b)



5. $r_{10} = 35$
   Psudocode:

```
for i in range(1, n+1):
    r[i] = max(p[i], max([p[k]+r[i-k] for k=range(1, i)]))
return r[n]
```

6. ◦ Correctness:
   - Prove by induction
   - Single-digit integers can obviously be sorted correctly by counting sort.

        - Assume that i-digit integers can be sorted correctly.
        - (i+1)-digit integers will first be sorted by their last i digits.
        - Next, the (i+1)th digit will be sorted. If there are integers with the same (i+1)th digit, their order will remain the same relative to the last i digits, as counting sort is stable.
        - Therefore, (i+1)-digit integers can be correctly sorted. Thus, any d-digit integers can be sorted correctly.
    - Linear time:
        - For each digit, a O(n) counting sort will be performed.
        - There are 10 digits -> $T(n) = 10 \times O(n) = O(n)$ which is linear time.

7. Invariant:

```
- v[1..k] is a heap
- v[1..k] ≤ v[k+1..n]
- v[k+1..n] is in increasing order
```

   Base case:

```
n=0 or n=1: trivial
```

   Induction Step:

```
Assume that heapsort works correctly for lists of length up to k.

1. We build a max-heap from the list of length k+1
2. We extract the maximum element from the heap, e.g., swap v[1] and v[k+1],
   and heapify the remaining elements.
By the inductive hypothesis, heapsort correctly sorts the remaining sublist of
length k.
```

8. Add a value in each node of the BST to record the number of nodes in the left subtree called \(n_left\).

```
FIND-K-LARGEST(root, k):
    if k == root.n_left + 1:
        return root.value;
    if k <= root.n_left:
        return FIND-K-LARGEST(root.left, k);
    return FIND-K-LARGEST(root.right, k - root.n_left - 1);
```

9. Recommended pesudocode:

```
1. Sort the points based on x-coordinate

2. FindCurve(points):

    Base case:
        If there are fewer than 2 points,
        return the points as the curve

    Divide points into two halves

    left_curve = FindCurve(left)
    right_curve = FindCurve(right)

    return MergeCurve(left_curve, right_curve)


3. MergeCurve(left_curve, right_curve):

    Compare the peaks from the two curves, then
    choose the highest point as the new peak.

    Along with the two peaks, collect the increasing part of the left curve
    and any decreasing portions that are higher than the peak of the right curve
    then applying the same principle to the right curve.

    Connect the collected points to form a new curve
```