Answer

3-1:
Answer: (c) 480

No. of ways to arrange ABCDEF = 6!

No. of ways to arrange (AB) CDEF = 5!
No. of ways to arrange AB = 2!
→ no. of ways to arrange such that AB are always next to each other = 5! x 2!

Therefore, using complementation, total no. of ways = 6! - (5! X 2!) = 480

3-2:
Answer: (a) 72

No. of ways for 6 ppl to sit around a table = 6!/6 = 120
No. of ways A & B sit together = 5!/5 x 2 = 48
No. of ways A & B don't sit together = 120 - 48 = 72

3-3:
Answer: (b) 233

Divisible by 3: 500/3 = 166
Divisible by 5: 500/5 = 100
Divisible by 3 & 5 = 500/15 = 33
Divisible by 3 or 5 = 166 + 100 - 33 = 233

3-4:
Answer: (c) 461

At least 1 man = (All possible ways to choose 5 people - ways to choose no man)
All possible ways = 11C5 = 462
Ways to choose no man = 6C0 * 5C5 = 1
Therefore, 462 - 1 = 461

Alternatively,
Case 1: only 1 man chosen = 6C1 x 5C4 = 30
Case 2: only 2 men chosen = 6C2 x 5C3 = 150
Case 3: only 3 men chosen = 6C3 x 5C2 = 200
Case 4: only 4 men chosen = 6C4 x 5C1 = 75
Case 5: 5 men chosen = 6C5 x 5C0 = 6
Answer = case 1 + case 2 + case 3 +case 4 + case 5 = 461

Note: do not do: 6C1 x 10C4 because of a lot of repetitions.

3-5:
Answer: (c) 19

Working:
1st pos: 0 0 _ _ _ = 2^3
2nd pos: 1 0 0 _ _ = 2^2
3rd pos: _ 1 0 0 _ = 2^2
4th pos: _ _ 1 0 0 = 2^2 - **1** | **repeated 0-0-1-0-0**
Answer = 2^3 + 2^2 + 2^2 + 2^2 - 1 = 19

3-6:
Answer: (b) 9000

Case 1: 1 digit (cannot be 0) hence there are only 9 options

Case 2: 2 digits (cannot be 0 at start or end) hence there is only 9 options for digit 1 and 9 options for digit 2

Case 3: 3 digits (cannot be 0 at start or end) hence there is only 9 options for digit 1, 10 options for digit 2 and 9 options for digit 3

Case 4: 4 digits (cannot be 0 at start or end) hence there is only 9 options for digit 1, 10 options for digit 2, 10 options for digit 3 and 9 options for digit 4

Ans: (9) + (9 x 9) + (9 x 10 x 9) + (9 x 10 x 10 x 9) = 9000

3-7:
Answer: (b) 48

Amanda sits at the end.
2C1 to choose a girl to sit beside her.
The remaining 4 people, 4!
Ans: 2C1 x 4! = 48

3-8:
Answer: (a) 2

Since they are in groups of 3, we can treat the groups as an entity and since it is a circular table, the answer would be (3-1)! or 3!/3.

3-9:
Answer: (b) 60

No. of ways we can arrange the 5 letters = 5! = 120.
Within each arrangement, the two Es order does not matter.
No. of different arrangement = 5! / 2! = 60.

3-10:
Answer: (c) 13 * 4C2

There are 13 ways to select the rank, then we multiply by 4C2 i.e. the number of ways to choose any 2 suits from a total of 4. Hence, 13 * 4C2.

4-1:
Answer: (c) O(n^2log(n))

i goes from 1, 2, 2^2, …, 2^k (=n) → O(log(n))
j goes from 2, 3, …, 21 → 20 times
k goes from n, n-2, n-4, …, 1 (assuming n odd) → n/2 times
g(n) → O(n)
∴ log(n) x 20 x n/2 x n = O($n^2 log(n)$)

4-2:
Answer: ~~(b)~~ (c) O(nlog(n))

$[\log(n^{12}) - \log(n^2)] = \log(n^{12}/n^2) = \log(n^{10}) = 10\log(n)$
$10\log(n) \times (3/2)n = (30/2)n\log(n)$ → O(nlog(n))
$(\log 100)^2$ is constant
$\log(n^{20}) = 20\log(n)$
Overall dominant term is nlog(n) → O(nlog(n))

4-3:

Answer: (b) O(n^3)

i goes from 0 .. n-1
f(n) is called (n/2) times → $n^2/2$
g(n) is called (n/2) times → $n^3/2$
Big O of function = $O(n^3/2 + n^2/2) = O(n^3)$

4-4:

Answer: (d) no. of multiples of a + 1

Function f_multiple has parameter(list_m, a) and returns the sum of elements in list_m which are multiples of a.

First assignment operation: sum = 0
Other assignment operations: no. of multiples of a because the assignment statement 'sum += i' is repeated when the if condition i % a == 0 is met.

4-5:

Answer: (b) f2>f1>f3

Function	no. of steps	Big O	Order
f1(n)	$3n * 2n^3 + 7n$	$O(n^4)$	2
f2(n)	$(3n)! + 6n^2 + n(\log_2 n)$	O((3n)!)	1 (worst)
f3(n)	$15 \log(n^2) + 20^3 + 7(\log n)^2$	$O((\log n)^2)$	3 (best)

4-6:

Answer: (a) is true

(a) True: Worse-case scenarios are a realistic and good indicator of "difficulty" to run an algorithm as compared to best-case (a "special situation") or average scenarios (difficult to determine without context)
(b) False: Counting the number of primitive operations (ie comparison, assignment, subtractions etc) performed by the algorithm on a given input size should be the way to determine the Big O. We do not base a measure of efficiency of an algorithm off the size of data input.
(c) False: if n and m are independent inputs to the algo, both terms should be preserved.
(d) False: $O(2^n)$. When n increases by 1, number of operations doubles. Therefore, it runs in exponential time and has the highest complexity out of the other 3 Big O notations.

4-7:

Answer: (a)

1st loop: the print statement executes n times
Therefore, the runtime for the first for-loop is **n**.

2nd nested loop: the print statement executes n^2 times.
Therefore, runtime for this nested loop is n^2.
Total runtime is $n + n^2$
Therefore, Big O complexity = $O(n + n^2) = O(n^2)$

Not $n \times n^2$ because the first part and second part are independent events.

4-8:
Answer: (d) $O(n^2)$

i goes from 0, 2, 4, …, n-1 (assuming n odd) i.e. n/2 times
For each i, j goes from 0 … n-1, i.e. n times
Counting the ==, there are n/2 x n ==
Counting the statement in the loop, it executes when j = 0, 2, …, n-1 (i.e. n/2 times), therefore n/2 x n/2

Overall, $O(n^2)$

4-9:
Answer: (b) iii, ii, iv, v, i

a. $\log (100n)^3 \rightarrow O(\log n)$
b. $100n^2 \rightarrow O(n^2)$
c. $2^{2n+1} + 100 \rightarrow O(2^{2n})$
d. $n\log (n) \rightarrow O(n \log n)$
e. $100n + \log (5n)^2 \rightarrow = O(n)$

Decreasing order = Most complex to least complex,
c) $2^{2n+1} + 100 \rightarrow$ **b)** $100n^2 \rightarrow$ **d)** $n\log (n) \rightarrow$ **e)** $100n + \log (5n)^2 \rightarrow$ **a)** $\log (100n)^3$

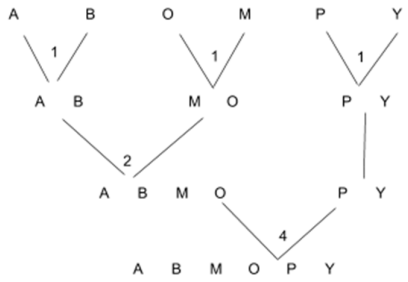
4-10:
Answer: (a) O(n)
Since the input size is n (the image array is considered the input in this case), and every pixel was only iterated through once, the answer is just O(n).

5-1:
Answer: (a)
1st iteration: lower = -1, upper = 9, mid = 4
2nd iteration: lower = -1, upper = 4, mid = 1
3rd iteration: lower = -1, upper = 1, mid = 0

5-2:
Answer: (b)
41 will be compared with 95, 56, 43 and 30. Since it is smaller than 95, 56 and 43 but larger than 30, it will be inserted between 30 and 43. Hence, there will be 4 comparisons but 3 swaps.

5-3:
Answer: (a)
i is 2 so compare num_array[2] and num_array[1]. Since 80 < 90, 80 will swap with 90. Then compare num_array[1] and num_array[0]. Since 80 < 70 is False, loop is exited.

5-4:
Answer: (c) 9
1 + 1 + 1 + 2 + 4 = 9



5-5:

Answer: (c)

As 67 is at the end of the string, a linear search would run 8 times.

Binary search:
lower = -1, upper = 8, mid = (-1+8)//2 = 3, discard left half
lower = 3, upper = 8, mid = (3+8)//2 = 5, discard left half
lower = 5, upper = 8, mid = (5+8)//2 = 6, discard left half
lower = 6, upper = 8, mid = (6+8)//2 = 7 (match!!)
Total: 4 comparisons

5-6:

Answer: (b) 1 + 2 + 3 + 1 + 2 = 9

1st step : Process 56, comparing it with 23 (Stay at original position) – 1 comparison

2nd step: Process 38 , comparing it with 56 (move towards left side) and comparing it with 23 (stay at the original position) – 2 comparisons.

3rd step: Process 10, comparing it with 56 (move towards left side), comparing it with 38 (move towards left side) and comparing it with 23 (move towards left side) – 3 comparisons

4th step: Process 72 and comparing it with 56 (stay at the original position) – 1 comparison

5th step: Process 65, comparing it with 72 (move towards left side), comparing it with 56 (stay at the original position) – 2 comparisons

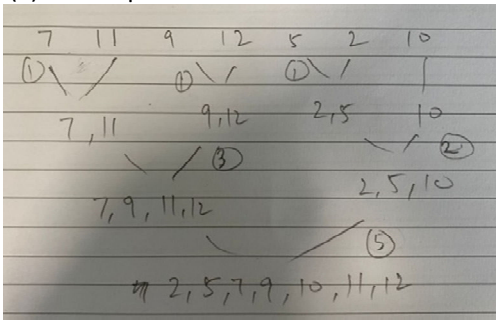
5-7:

Answer: (a) 7

$\log_2 78 = 6.28$ (round up to 7)

5-8:

Answer: (a) 13 comparisons



5-9:

Answer: (a) 90 and 99

First iteration:

- Lower = -1
- Upper = 7
- Mid = 3
- Mid value = 90

Second iteration:

Since 90 < 99

- Lower = 3
- Upper = 7
- Mid = 5
- Mid value = 99

5-10:

Answer: (c) 6

The linear search algorithm given in the slides will return the first position of any found key. Python will start from position 0 and iterate until it finds the key. There are two "42"s in the list, but the search terminates when the first instance of the key is found at position 6.

6-1:

Answer: (c) B13

convert(2835,16) → convert(177,16) + s[3] → convert(11,16) + s[1]
convert(11,16) returns s[11] i.e. 'B'
convert(177, 16) returns 'B1'
conert(2835, 16) returns 'B13'

6-2:

Answer: (d) 99

practice(arr, 4) → practice(arr, 3) → practice(arr, 2) → practice(arr, 1)
practice(arr, 1) returns 2 (i.e. x = 2). Since 2 > 28 is False, practice(arr,2) returns 28
Back in practice(arr, 3), x = 28. Since 28 > 55 is False, practice(arr, 3) returns 55.
Back in practice(arr, 4), x = 55. Since 55 > 99 is False, practice(arr, 4)returns 99.

6-3:

Answer: (c) n + sum_series(n-2)

Since the given sum of the positive integers starts with n followed by n - 2 and n - 4, the reduction step in line 5 will have to first return n, and invoke the sum_series function with n - 2.

6-4:

Answer: (a) O(n)

Assume $n = 2^m$.
$f(k,n) \rightarrow f(k, \frac{n}{2}) \times 2 \rightarrow f(k, n/2^2) \times 4 \rightarrow f(k, n/2^3) \times 8 \rightarrow \ldots \rightarrow f(k, n/2^m) \times 2^m$
Summing up the * at each level of recursion, $1 + 2 + 4 + \ldots + 2^m = 2^{m+1} - 1 = 2n - 1 = O(n)$

6-5:
Answer: (c) 1 2 1 3 1 2 1

tryme(3) → tryme(2) → tryme(1) → tryme(0)
Back to tryme(1), print(1)
Back to tryme(2), print(2), followed by tryme(1), which print(1)
Back to tryme(3), print(3), followed by tryme(2), which print(1), print(2), print(1)

In all, 1 2 1 3 1 2 1

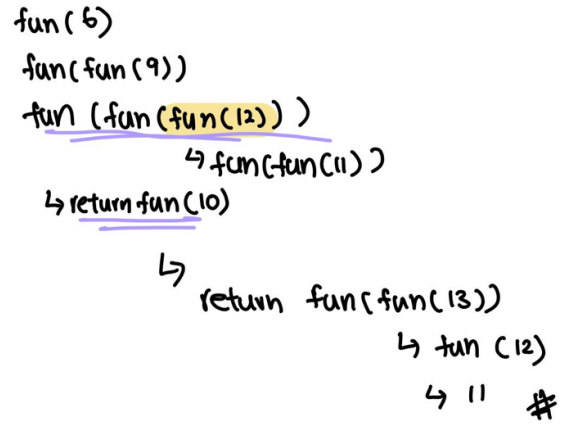
6-6:
Answer: (a) 23

The code sums up all values of x in range(1, x+1) + y.
(6,2) → (5, 8) → (4, 13) → (3, 17) → (2, 20) → (1, 22) → (0, 23) → 23

6-7:
Answer: (a) 120

The code calculates the factorial of N using recursion

6-8:
Answer: (b) 11



6-9:
Answer: (c) 2

The code checks if n is divisible by every integer from 2 to n-1.

6-10:
Answer: (b) 110

7-1:
Answer: (b) Head: 2, Tail: 2

assumption: both head (H) & tail (T) will point to position 0 at the start (when queue is empty). This question is implementation-specific.

assumptions:
- enqueue: insert element at T. T = (T + 1) % capacity
- dequeue: remove element at H. H = (H + 1) % capacity.

- capacity = 4, as stated in question.

line 1: enqueue("a"): H: 0 (unchanged). T: 1
line 2: enqueue("b"): H: 0 (unchanged). T: 2
line 3: enqueue("c"): H: 0 (unchanged). T: 3
line 4: enqueue("d"): H: 0 (unchanged). T: 0
line 5: dequeue(): H: 1. T: 0 (unchanged)
line 6: dequeue(): H: 2. T: 0 (unchanged)
line 7: enqueue("e"): H: 2 (unchanged). T: 1
line 8: enqueue("f"): H: 2 (unchanged). T: 2

7-2:
Answer: (c) s: top ['c', 'a'] , q: head ['f', 'e']

After pushing “a” and “b” into the stack, “b” is at the top. At line 5, it removes the top element in the stack which is “b” and enqueues it into the queue.
At line 6, “c” is pushed into the stack which will place it at the top.(top [‘c’, ‘a’])
At line 7, “f” enqueues into the queue which will place it to the tail.(head [‘b’, ‘f’])
After pushing “d” and “e” into the stacks → top [‘e’, ‘d’, ‘c’, ‘a’].
At line 10, it removes the top element in the stack which is “e” and enqueues it into the queue. (s: top [‘d’, ‘c’, ‘a’] , q: head [‘b’, ‘f’, ‘e’])
Then at line 11 and 12, it will pop “d” and dequeue “b” from stack and queue respectively. (s: top [‘c’, ‘a’] , q: head [‘f’, ‘e’]).

7-3:
Answer: (c) s: top ['4', '3', '2', '1'] , q: head ['a', 'b', 'c', 'd']

The function iterates through every character of the string. If the character is numeric, it is pushed to Stack s. Otherwise, the character is enqueued into Queue q.

Steps:
1234 pushed into Stack s → s: top [‘4’,’3’,’2’,’1’]
abcd enqueued into Queue q → q: head[‘a’,’b’,’c’,’d’]

7-4:
Answer: (a)

To perform dequeue operation you need to pop each element from the (1) stack and push it into the (2) stack. In this case you need to pop ‘m’ times and need to perform push operations also ‘m’ times. Then you pop the first element from this second stack (constant time) and pass all the elements to the (1) stack. Therefore the time complexity is O(m).

In enqueue operation, the new element is entered at the top of stack1. In dequeue operation, if stack (2) is empty then all the elements are moved to stack (2) and finally top of stack (2) is returned.

```
enQueue(q,  x)
  1) Push x to stack1 (assuming size of stacks is unlimited).
Here time complexity will be O(1)

deQueue(q)
  1) If both stacks are empty then error.
  2) If stack2 is empty
       While stack1 is not empty, push everything from stack1 to stack2.
  3) Pop the element from stack2 and return it.
Here time complexity will be O(n)
```

7-5:

Answer: (a) mystery ("a*&b*c&")

The function is used to check if for every '*', there is a closing '&'. The function first creates a stack. It then checks all the characters in the text inputted. If there is a "*", it adds it into the stack. If there is a "&", it pops the "*" from the stack. If there isn't any "*" in the stack, it returns False. If the count of the stack is 0, the function returns True. The function checks the order of the string as well. If the string has a "&" before a "*", the function returns false.

7-6:

Answer: (d) [top]69, 33

Stack starts pushing to the bottom then end at the top

7-7:

Answer: (c)

Just be careful: the push() implementation is atypical. It takes in a list and pushes the items in the list into the stack, starting from position 0.

push(s, a2) → s=[mango, watermelon] (first element is top)
pop(s) mango comes out → s = [watermelon]
push(s,a2) → s =[mango, watermelon, watermelon]
push(s,a1) → s=[orange, pear, apple, mango, watermelon, watermelon]
pop(s) orange comes out → s= [pear, apple, mango, watermelon, watermelon]

7-8:

Answer: ~~(a) 12~~ None

Temporary variable = tmp. These are the 12 steps:

1. tmp = s1.pop() # element 1
2. ~~s2.push(tmp)~~ q1.enqueue(tmp)
3. tmp = s1.pop() # element 2
4. q1.enqueue(tmp)
5. tmp = s1.pop() # element 3
6. q1.enqueue(tmp)
7. tmp = q1.dequeue() # element ~~2~~ 1
8. ~~s1.push(tmp)~~ q1.enqueue(tmp)
9. tmp = q1.dequeue() # element ~~3~~ 2
10. s1.push(tmp)
11. ~~tmp = s2.pop() # element 1~~ tmp = q1.dequeue() # element 3
12. s1.push(tmp)
13. tmp = q1.dequeue() # element 1
14. s1.push(tmp)

7-9:
Answer: (c) C B S R U T Q P
Similar to Q1 of your Week 6 tutorial.

7-10:
Answer: (c) O(n)
In the worst case, you might have to traverse the entire list.
See slides about sorted list vs unsorted list implementation of a queue in your Week 6 slides.