Question 1 – C and pointers

Consider the following program:

```c
int multiply_by_2(int* p) {
    *p = *p * 2;
    return *p;
}

int main() {
    int n = 10, x = 16, y = 32, z = 64;
    int* p1 = &x;
    int* a = malloc(n * sizeof(int));
    int* p2;
    for (int i = 0; i < n; i++) {
        a[i] = i;
    }
    p2 = a + (n / 2);
    *p2 = 0;
    z = multiply_by_2(&x);
    y = multiply_by_2(p1);
}
```

After running this program, what value is represented by each of the following expressions? If the value is a memory address, please write “memory address”. (1 point each)

1. x 64 6. p2 memory address
2. y 64 7. *p2 0
3. z 32 8. a memory address
4. p1 memory address 9. a[0] 0
5. *p1 64 10. a[5] 0

Question 2 – Complexity analysis (10 points)
What is the complexity (expressed in big O notation) of each of the following loop structures? In each case, assume that the function $O(x)$ has complexity $O(x)$, i.e. $O(1)$ has complexity $O(1)$, $O(\log(n))$ has complexity $O(\log n)$, etc. (2.5 points each)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. for (int i = 0; i &lt; n; i++) {</td>
<td>$O(n^2)$</td>
<td></td>
</tr>
<tr>
<td>for (int j = 0; j &lt; n; j++) {</td>
<td></td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>$O(1)$;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$O(1)$;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>$O(n \log n)$</td>
<td></td>
</tr>
<tr>
<td>$O(1)$;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td>(first loop cost dominates the second)</td>
</tr>
<tr>
<td>2. for (int i = 0; i &lt; n; i++) {</td>
<td>$O(n \log n)$</td>
<td></td>
</tr>
<tr>
<td>$O(1)$;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>for (int j = n; j &gt; 0; j /= 2) {</td>
<td></td>
<td>}$O(n \log n)$</td>
</tr>
<tr>
<td>$O(1)$;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. for (int i = 0; i &lt; n; i++) {</td>
<td>$O(n \log n)$</td>
<td></td>
</tr>
<tr>
<td>$O(\log(n))$;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$O(1)$;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td>(first loop cost dominates the second)</td>
</tr>
<tr>
<td>for (int i = 0; i &lt; n; i++) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$O(1)$;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. for (int i = 0; i &lt; n; i++) {</td>
<td>$O(n^3)$</td>
<td></td>
</tr>
<tr>
<td>for (int j = 0; j &lt; i; j++) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$O(n)$;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Question 3 – Estimating wall clock time (15 points)

For each question below, use the provided information to estimate the wall clock running time for the given algorithm on the data set in question. (5 points each)

1. You are benchmarking the linear search operation of an open-source linked list implementation. In your test, you have determined that it takes 10ms on average to search for an item in a collection of 10,000 elements. On the same machine, how long would you expect it to take on average to search for an item in collection
2. You are benchmarking the binary search operation of an open-source ordered array implementation. In your test, you have determined that it takes 5ms on average to search for an item in a collection of 1024 (i.e. $2^{10}$) elements. On the same machine, how long would you expect it to take on average to search for an item in a collection containing 1,073,741,824 (i.e. $2^{30}$) elements?

\[
\frac{\log(n_1)}{\log(n_2)} = \frac{t_1}{t_2} \Rightarrow \frac{\log(2^{10})}{\log(2^{30})} = \frac{5ms}{t_2} \Rightarrow \frac{10}{30} = \frac{5ms}{t_2} \Rightarrow t_2 = 15ms
\]

3. You are benchmarking an open-source quicksort implementation (quicksort is $O(n \log n)$ on average). In your test, you have determined that it takes 16ms on average to sort a collection of 1024 (i.e. $2^{10}$) elements. On the same machine, how long would you expect it to take on average to sort a collection containing 1,048,576 (i.e. $2^{20}$) elements?

\[
\frac{n_1 \log(n_1)}{n_2 \log(n_2)} = \frac{t_1}{t_2} \Rightarrow \frac{2^{10} \log(2^{10})}{2^{20} \log(2^{20})} = \frac{16ms}{t_2} \Rightarrow \frac{10 \cdot 2^{10}}{20 \cdot 2^{20}} = \frac{2^{4} ms}{t_2} \Rightarrow \frac{1}{2^{11}} = \frac{2^{4} ms}{t_2} \Rightarrow t_2 = 2^{15} ms = 32,768ms \approx 33s
\]

Question 4 – Linked list-based queue (30 points)

Assume that you have the basic structure below to represent a single node in a linked list for storing integer values.
struct node {
    int val;
    struct node* next;
};

Provide pseudocode for each of the items below to implement a queue using this node structure. *Importantly, do not forget to indicate where memory should be allocated and freed, and do not forget to indicate when a given pointer should be set to NULL.* Make sure also to check boundary conditions (e.g. list empty) where appropriate.

1. (5 points) Write a new structure to represent your queue.

```c
struct queue {
    struct node* head;
    struct node* tail;
};
```

2. (5 points) Write pseudocode for a function to create a new queue. It should return an instance of the structure you designed above.

```c
struct queue* queue_create() {
    struct queue* q = malloc(sizeof(struct queue));
    q->head = NULL;
    q->tail = NULL;
    return q;
}
```

**Question 4 (continued)**

3. (10 points) Write pseudocode for a function to enqueue a value into a queue represented by the structure you designed above.

```c
void queue_enqueue(int val, struct queue* q) {
    // Create new node with appropriate values.
    struct node* l = malloc(sizeof(struct node));
    l->val = val;
    l->next = NULL;
```
4. (10 points) Write pseudocode for a function to dequeue a value from a queue represented by the structure you designed above. It should return the dequeued value.

```c
int queue_dequeue(struct queue* q) {
    // Only dequeue if queue isn’t empty.
    assert(q->head != NULL);

    // Remember current head and its value.
    struct node* h = q->head;
    int v = h->val;

    // Move the head.
    q->head = h->next;

    // Remove the tail, if it’s the same as the old head.
    if (q->tail == h) {
        q->tail = NULL;
    }

    // Free dequeued node and return its value.
    free(h);
    return v;
}
```

**Question 5 – Dynamic array iterator (20 points)**

Assume you have the basic structure below representing a dynamic array for storing integers.

```c
struct da {
    int size;
    int capacity;
    int* data;
};
```

Provide pseudocode for each of the items below to implement an iterator that moves forward over this dynamic array. **Importantly, do not forget to indicate where memory should be allocated and freed, and do not forget to indicate when a given pointer should be set to NULL.** (5 points each)
1. Write a new structure to represent your iterator.

```c
struct da_iterator {
    int cur;
    struct da* da;
};
```

2. Write a pseudocode for a function to create a new iterator over a given dynamic array. The iterator should start at the beginning of the array. Your function should return an instance of the structure you created above. You may assume here that \texttt{da} is a validly-created dynamic array.

```c
struct da_iterator* da_iterator_create(struct da* da) {
    struct da_iterator* iter = malloc(sizeof(struct da_iterator));
    // Initialize to -1 because we do \texttt{cur++} in next().
    iter->cur = -1;
    iter->da = da;
}
```

**Question 5 (continued)**

3. Write pseudocode for a \texttt{has_next()} function for your iterator. It should return 1 if there is at least one more element in the array to which to iterate and 0 if there are no more elements to which to iterate.

```c
int da_iterator_has_next(struct da_iterator* iter) {
    // There’s another element if the current element is not the last element of the array.
    return iter->cur < iter->da->size - 1;
}
```

4. Write pseudocode for a \texttt{next()} function for your iterator. It should increment the location of the iterator and return the value of the new element the iterator represents. Do appropriate checks to make sure there is in fact a next element.

```c
int da_iterator_next(struct da_iterator* iter) {
    assert(da_iterator_has_next(iter));
iter->cur++;  
return iter->da[iter->cur];