# Homework #01 Solution

**Guidelines:**

- Attempt all questions by yourself before discussing with peers, this is practice to strengthen your concepts.
- For coding questions, try writing clean, readable code on paper.
- Since this homework is ungraded, focus on learning rather than using LLMs to generate codes and get answers.
- If you get stuck, you are encouraged to:
  - Post your doubts on the course Slack channel.
  - Visit the TAs during office hours for guidance.

**Topics:** Experimental Analysis, Big-Oh Analysis, Arrays, Linked Lists, Stacks, Queues and their Applications

## Section 1: Experimental and Asymptotic Analysis

**Q1a. Is Ben's statement correct? Explain why the experimental results might differ from what the Big-O notation suggests for these small inputs.**

Ben's statement is incorrect. The experimental results on small datasets are misleading because Big-O notation describes the *asymptotic* behavior of an algorithm, meaning its performance as the input size (n) grows very large.

For small values of n, the constants and lower-order terms, which Big-O notation ignores, can have a significant impact on the actual runtime. Ben's $O(2^n)$ algorithm might have a smaller constant factor of overhead than Alex's $O(n^2)$ algorithm. For instance, if Alex's runtime is $100n^2$ and Ben's is $5 * 2^n$:

- For n = 4, Alex's runtime would be $100 * 4^2 = 1600$, while Ben's would be $5 * 2^4 = 80$.
  In this scenario, Ben's algorithm is faster for small `n`. However, the exponential growth of $O(2^n)$ will eventually surpass the polynomial growth of $O(n^2)$.

**Q1b. Which algorithm (Alex's or Ben's) would be faster for a large input size, such as n = 500,000? Provide your reasoning based on their time complexities.**

For a large input size like n = 500,000, Alex's $O(n^2)$ algorithm would be faster. The growth rate of an exponential function ($O(2^n)$) is vastly greater than that of a polynomial function ($O(n^2)$). As `n` becomes large, the $2^n$ term will become astronomically larger than $n^2$, making the $O(2^n)$ algorithm computationally infeasible, regardless of the constant factors involved.

**Q2. (a) Express the following functions in terms of Big-O notation (tightest upper bound):**

a. $10n^4 + 50n^2 + 300$ is $\mathbf{O(n^4)}$

b. $n * \log(n) + 3n + 500$ is $\mathbf{O(n \log n)}$

c. $n^2 + 2^n$ is $\mathbf{O(2^n)}$

d. $n^3 + \log(n^4)$ is $\mathbf{O(n^3)}$

e. sqrt(n) + log(n) is **O(sqrt(n))**

**Q2. (b) State if each of the following is True or False.**

a. $100n^2 + 2n + 5 \in O(n^3)$ is **True**

b. $n \log n \in O(n)$ is **False**

c. $5^n \in O(2^n)$ is **False**

d. $n! \in O(n^n)$ is **True**

e. $1000 \in O(1)$ is **True**

**Q3. Consider the following code snippet. Determine its best-case and worst-case time complexity in Big-O notation. Explain your reasoning.**

```
void processData(int arr[], int n, int key) {
 if (arr[0] == key) {
    cout << "Key found at the beginning!" << endl;
    return;
 }

 for (int i = 0; i < n; i++) {
    for (int j = 1; j < n; j = j * 2) {
        cout << "Processing item: " << arr[i] << " and " << j << endl;
    }
 }
}
```

<u>**Best-Case Time Complexity: O(1)**</u>

The best-case scenario occurs if the first element of the array `arr[0]` is equal to `key`. In this situation, the initial `if` statement is true, and the function returns immediately after a single <u>comparison, which is a constant time operation.</u>

**Worst-Case Time Complexity: O(n log n)**

The worst case occurs when `arr[0]` is not equal to `key`, and the nested loops are executed. The outer loop runs `n` times. The inner loop's variable `j` doubles in each iteration (1, 2, 4, 8...), meaning it executes $\log_2(n)$ times. Since these loops are nested, their complexities are multiplied, resulting in a total time complexity of O(n * log n).

**Q4. (a) Determine the worst-case time complexity in Big-O notation for the**

```
void complexFunction(int n) {
 for (int i = 0; i < n; i++) {
     for (int j = 0; j < i; j++) {
         for (int k = 0; k < j; k++) {
             // some O(1) operation
         }
     }
 }
}
```

**Worst-Case Time Complexity: O(n³)**

The function has three nested loops. The outer loop runs approximately `n` times, the middle loop runs up to `n` times, and the inner loop runs up to `n` times, leading to a cubic growth rate.

**Q4. (b) Count the number of primitive operations in the**

```
int countOperations(int n) {
 int operations = 0;
 operations++; // for initialization
 int i = n;
 operations++; // for initialization
 while (i > 1) {
     operations++; // for the while check
     // some O(1) work
     operations++;
     i = i / 2;
     operations++; // for the division/assignment
```

```
 }
 operations++; // for the final while check
 return operations;
}
```

**<u>Number of Primitive Operations:</u>** The number of operations is approximately `3 + 3 * floor(log₂(n))`.

**<u>Worst-Case Time Complexity: O(log n)</u>**

The `while` loop is the dominant part of the function. Since `i` is halved in each iteration, the loop runs a logarithmic number of times with respect to `n`.

## Section 2: Arrays and Linked Lists

**Q5. Which data structure would you choose for an "Undo" feature: a dynamic array or a stack? Justify your choice.**

The best choice is a stack. The "Undo" feature is a classic example of a Last-In, First-Out (LIFO) process. The last action performed is the first one to be undone. A stack is a LIFO data structure by definition. Recording an action: This corresponds to a `push` operation on the stack, which is a highly efficient O(1) operation. Performing an "undo": This corresponds to a `pop` operation, which is also O(1). While a dynamic array could be used, a stack is the most natural and conceptually clean data structure for this task, with guaranteed O(1) time complexity for the required operations.

**Q6. State if each of the following is True or False. If a statement is false, provide a brief justification.**

**a. Accessing the element at index k in a singly linked list is an O(1) operation.**

**False.** You must traverse the list from the beginning to reach the k-th element, which takes O(k) time.

**b. Inserting an element at the beginning of a dynamic array is an O(1) operation on average.**

**False.** This is an O(n) operation because all existing elements must be shifted one position to the right.

**c. In a doubly linked list, deleting a given node (for which you have a direct pointer) is an O(1) operation.**

**True.**

**d. A key advantage of a circular linked list is that it allows traversal from the last node to the first node in O(1) time.**

**True.**

**e. If memory usage is the absolute top priority, a dynamic array is always more memory-efficient than a linked list.**

**False.** A dynamic array can have a lot of unused allocated space (excess capacity), potentially using more memory than a linked list, which only allocates space as needed (plus pointer overhead).

**Q7. Which data structure would be best for a web browser's history: a singly linked list, a doubly linked list, or a dynamic array? Justify your answer.**

The best choice is a doubly linked list.

| Operation | Singly Linked List | Doubly Linked List | Dynamic Array |
|---|---|---|---|
| **Visit new page** | O(n) (or O(1) with tail pointer) | O(1) | O(1) amortized |
| **Go back** | O(n) | O(1) | O(1) |
| **Go forward** | O(1) | O(1) | O(1) |

A doubly linked list is the only structure that provides O(1) time complexity for all three essential operations. A singly linked list fails at the "go back" operation. While a dynamic array seems efficient, a doubly linked list more naturally handles the case where a user goes back and then visits a new page, which invalidates the old "forward" history. This is simpler to implement with pointers than with array index management.

## Section 3: Stacks and Queues

**Q8a. What is the final content of the stack, from top to bottom?**

*Operations: push('A'), push('B'), pop(), push('C'), push('D'), pop(), pop(), push('E')*

**Final content (top to bottom):** E,A

**Q8b. What is the final content of the queue, from front to rear?**

*Initial: 10, 20, 30. Operations: enqueue(40), dequeue(), enqueue(50), enqueue(dequeue())*

**Final content (front to rear):** 30, 40, 50, 20

**Q9a. What would be the final contents of a queue after the following sequence of operations?**

*Operations: enqueue(5), enqueue(10), enqueue(15), dequeue(), enqueue(20), enqueue(dequeue()), dequeue(), enqueue(25)*

**Final content (front to rear):** 20, 10, 25

**Q9b. What would be the final contents of a stack after the following sequence of operations?**

*Operations: push(1), push(2), push(3), pop(), push(pop()), push(4), pop(), push(5)*

**Final content (top to bottom):** 5, 2, 1