Part 1: Big-O Notation

For each of the following code snippets, determine the Big-O time complexity and explain your reasoning:

Code 1

```
public void example1(int[] arr)
{
  for (int i = 0; i < arr.length; i++)
  {
    for (int j = i; j < arr.length; j++)
    {
      System.out.println(arr[i] + arr[j]);
    }
  }
}
```

Solution:

The Big-O time complexity of Code 1 is $O(n^2)$ where n is equal to arr.length. The outer for loop runs from i = 0 to i < arr.length, which means it iterates n times. The inner for loop runs from j = 1 to j < arr.length, which means it runs n times for every iteration of the outer loop. The 1 represents the line of code in the nested for loop that only iterates once. So the total number of iterations is n(n(1)) or n^2.

Code 2

```
public void example2(int[] arr)
{
  for (int i = 0; i < arr.length; i++)
  {
    if (arr[i] % 2 == 0)
    {
      System.out.println(arr[i]);
    }
  }
}
```

Solution:

The Big-O time complexity of Code 2 is $O(n)$ where n is the length of arr. The for loop iterates from 0 to i < arr.length, meaning that it runs n times, where n is arr.length. Inside the loop there is an if statement that operates for the duration of the loop. The print statement in the if statement also iterates for the duration of the loop. Since the loop runs n times, the time complexity is $O(n)$

Code 3

```java
public void example3(int n)
{
  int i = 1;
  while (i < n)
  {
    System.out.println(i);
    i *= 2;
  }
}
```

Solution:

The Big-O time complexity of Code 3 is O(log n). The while loop iterates where i starts at 1 and doubles itself until i is $\geq$ n. Because i doubles every iteration, after x iterations, i = 2^n. Because the loop only runs while i $<$ n, we need to solve $2^x \geq n$ to determine when the loop stops. Taking the log on both sides gives us $x \geq log_2(n)$. This gives us a time complexity of O(log n).

Part 2: Time Complexity

A. Algorithm Analysis:

Consider **arr** is a reference to an unsorted array, analyze the following function:

```java
public boolean search(int[] arr, int target)
{
  for (int i = 0; i < arr.length; i++)
  {
    if (arr[i] == target)
    {
      return true;
    }
  }
  return false;
}
```

What is the time complexity of the best-case and worst-case scenarios for this search?

Solution:

Best-Case: O(1)

If target is the first index then the function returns true immediately.

Worst-Case: O(n) if n is equal to the length of the array

If the target is not present or in the last index, then the loop runs n times.

B. Algorithm Comparison:

Three algorithms solve the same problem:

- Algorithm A: $O(N^2)$
- Algorithm B: $O(n \log n)$
- Algorithm C: $O(n!)$

Which algorithm would you choose and why? Provide a detailed explanation comparing their performance as n grows.

Solution:

I would Choose Algorithm B:$O(n \log n)$. As seen from the table below

	$O(N^2)$	$O(n \log n)$	$O(n!)$
n			