1. a. Write pseudocode for a divide-and-conquer algorithm for finding the position of the largest element in an array of n numbers.
b. What will be your algorithm's output for arrays with several elements of the largest value?
c. Set up and solve a recurrence relation for the number of key comparisons made by your algorithm.
d. How does this algorithm compare with the brute-force algorithm for this problem?

a)

```
int findMaxIndex(int arr[], int left, int right) {
    if (left == right) {
        return left; // Only one element
    }
    int mid = (left + right) / 2;
    // Recur for left and right halves
    int leftMaxIndex = findMaxIndex(arr, left, mid);
    int rightMaxIndex = findMaxIndex(arr, mid + 1, right);
    // Return the index of the larger element
    return (arr[leftMaxIndex] >= arr[rightMaxIndex]) ? leftMaxIndex : rightMaxIndex;
}

int main() {
    int arr[] = {5, 3, 9, 2, 8, 15, 1, 7};
    int n = sizeof(arr) / sizeof(arr[0]);
    int maxIndex = findMaxIndex(arr, 0, n - 1);
    printf("The largest element is %d at index %d\n", arr[maxIndex], maxIndex);
    return 0;
}
```

b) The algorithm's output will be the index of the Largest element in the Array, if there are multiple index of the largest element, the algorithm returns the left most index of the largest element.

c) Recurrence relation for the number of key comparison is: C(n)=C(n/2)+C(n/2)+1

Solve by backward substitution:

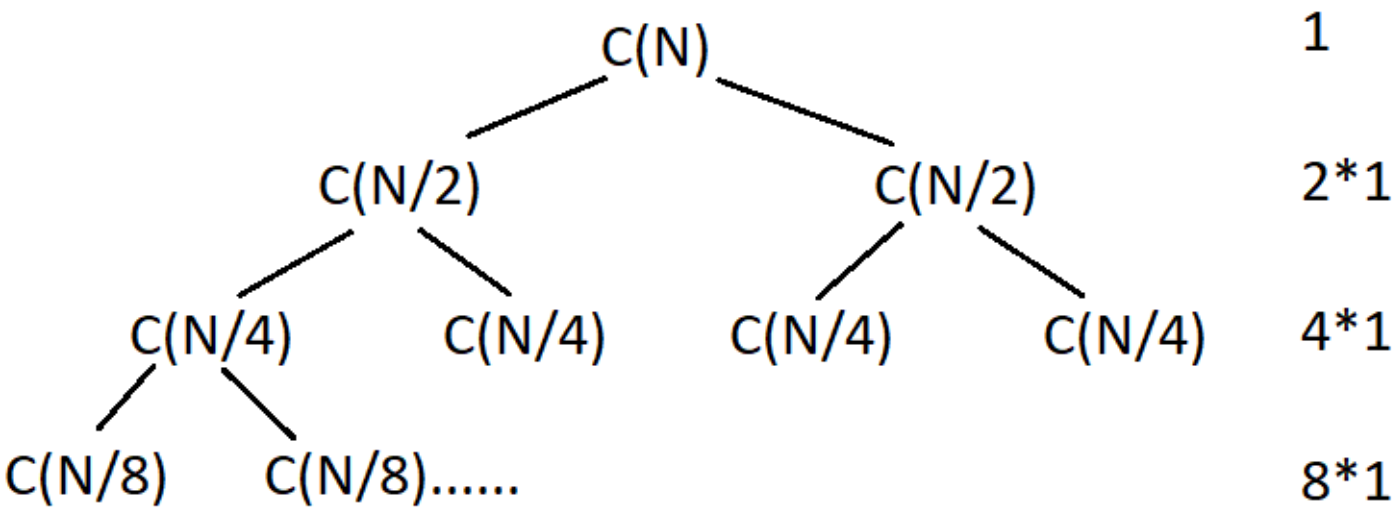
$$
\begin{aligned}
C(n) &= C(n/2) + C(n/2) + 1 \qquad C(1) = 0 \\
&= 2C(n/2) + 1
\end{aligned}
$$

Let $n = 2^k$

$$
\begin{aligned}
C(2^k) &= C\left(\frac{2^k}{2}\right) + C\left(\frac{2^k}{2}\right) + 1 \\
&= C\left(2^{k-1}\right) + C\left(2^{k-1}\right) + \\
&= 2C\left(2^{k-1}\right) + 1 \qquad C\left(2^{k-1}\right) = 2C\left(2^{k-2}\right) + 1 \\
&= 2\left(2C\left(2^{k-2}\right) + 1\right) + 1 \\
&= 2^2 C\left(2^{k-2}\right) + 2 + 1 \qquad C\left(2^{k-2}\right) = 2C\left(2^{k-3}\right) + 1 \\
&= 2^2\left(2C\left(2^{k-3}\right) + 1\right) + 2 + 1 \\
&= 2^3 C\left(2^{k-3}\right) + 2^2 + 2 + 1 \\
&\ldots\ldots \\
&= 2^i C\left(2^{k-i}\right) + 2^{i-1} + 2^{i-2} + \ldots + 1 \\
&\ldots\ldots \\
&= 2^k C\left(2^{k-k}\right) + 2^{k-1} + 2^{k-2} + \ldots + 1 \\
&= 2^k C\left(2^0\right) + 2^{k-1} + 2^{k-2} + \ldots + 1 \\
&= 2^k C(1) + 2^{k-1} + 2^{k-2} + \ldots + 1 \\
&= 2^k(0) + 2^{k-1} + 2^{k-2} + \ldots + 1 \\
&= 2^{k-1} + 2^{k-2} + \ldots + 1 \\
&= 2^k - 1 \\
&= n - 1
\end{aligned}
$$

Therefore, $C(n) = n - 1$

Solve by recurrence tree:



$$\sum_{i=0}^{\log n} 2^i = 2^{\log_2 n+1} - 1$$

d. Brute force and divide and conquer algorithm both take the same number of comparisons, however, there are no overhead of recursion calls in Brute force. Also, divide and conquer needs more space requirement because of the recursions as oppose to brute force that runs linearly.

2 -Find the order of growth for solutions of the following recurrences.

a. $T(n) = 4T(n/2) + n$, $T(1) = 1$

b. $T(n) = 4T(n/2) + n^2$, $T(1) = 1$

c. $T(n) = 4T(n/2) + n^3$, $T(1) = 1$

$$T(n) = aT(n/b) + f(n)$$

If $f(n) \in \Theta(n^d)$ where $d \geq 0$

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

a) A=4 b=2 d=1 a>b^d $n^{\log_2 4} = n^2$

b) A=4 b=2 d=2 a=b^d $n^2 \log n$

c) A=4 b=2 d=3 a<b^d n^3

3- Bir matris üzerindeki gezinen bir robot için her hücre değeri o hücreye basabilmek için gerekli enerji tüketimini ifade etmektedir. Robot yalnızca sağa veya aşağı yöne doğru hareket edebilir. Sol en üst hücreden başlandığında, sağ en alt hücreye gelebilmek için harcanacak minumum enerji miktarını dynamic programming ile hesaplayan C kodunu yazınız.

```c
#include <stdio.h>
#include <limits.h>

#define INF INT_MAX
#define MAX 100


// Recursive function with memoization
int minPath(int i, int j, int E[][MAX], int memo[][MAX]) {
    // Base case: starting cell
    if (i == 0 && j == 0)
        return E[0][0];

    // Out of bounds
    if (i < 0 || j < 0)
        return INF;

    // If already computed, return stored value
    if (memo[i][j] != 0)
        return memo[i][j];

    // Recursive calls: from top and from left
    int from_top = minPath(i - 1, j, E, memo);
    int from_left = minPath(i, j - 1, E, memo);

    // Store and return minimum path cost
    int min_val = (from_top < from_left) ? from_top : from_left;
    memo[i][j] = min_val + E[i][j];
    printf("i:%d j:%d %d\n",i,j,min_val);
    return memo[i][j];
}
```

```
int main() {
    int m = 3, n = 3;
      int i,j;
    // Example grid (change size as needed)
    int E[MAX][MAX] = {
        {1, 3, 1},
        {1, 5, 1},
        {4, 2, 1}
    };

      // Initialize memo table to 0
      int memo[MAX][MAX]={};

    int result = minPath(m - 1, n - 1, E,memo);
    printf("Minimum energy path cost: %d\n", result);

    return 0;
}
```