Plan for 241 day 4: RUNNING TIME OF RECURSIVE ALGORITHMS.

RECURSIVE ALGORITHMS:

Process for determining running time of iterative algorithms:

1. Decide on a parameter indicating input size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or somehow ascertain the the order of growth of its execution.

A recursive algorithm can often be described by a recurrence equation that describes the overall runtime on a problem of size n in terms of the runtime on smaller inputs.

```
IterativeBinarySearch(A, T):
  Input:  A is an array of numbers sorted in increasing order
  T is the element to find
  Output: The index of T or -1 if T not found
    n = A.length
    L = 1
    R = n
    while L <= R
        m = floor((L + R) / 2)
        if T > A[m]:
            L := m + 1
        else if T < A[m]:
            R := m - 1
        else:
            return m
    return -1
```

Ask: What is basic operation? Are there b-c and w-c running times? If so, give instances of each. Express running time in asymptotic notation.

```
RecursiveBinarySearch(A, left, right, x):
  {
        // Base condition 1 (search space is exhausted)
        if (left > right) {
            return -1;
        }

        // we find the mid value in the search space and
        // compare it with key value

        int mid = (left + right) / 2;

        // Base condition 2 (key value is found)
        if (x == A[mid]) {
            return mid;
        }

        // Recursive condition 1
        // discard all elements in the right search space including the mid element
        else if (x < A[mid]) {
            return binarySearch(A, left, mid - 1, x);
        }

        // Recursive condition 2
        // discard all elements in the left search space
        // including the mid element
        else {
            return binarySearch(A, mid + 1, right, x);
        }
  }
```

For divide and conquer algorithms, we get recurrences like:

$$
\begin{aligned}
T(n) &= \Theta(1) \text{ if } n < c \text{ or} \\
&= aT(\tfrac{n}{b}) + D(n) + C(n) \text{ otherwise}
\end{aligned}
$$

where

a = number of subproblems we divide the problem into

$\frac{n}{b}$ = size of subproblems (in terms of n)

$D(n)$ = time to divide the size n problems into subproblems

$C(n)$ = time to combine the subproblem solutions to get answer for size n

What is $T(n)$ for BinarySearch?

Go to slides for MergeSort.

NEXT SUBJECT: Heap-Sort (Chapter 6)