References for Today’s Lecture

- Required reading
  - CLRS Chapter 2

- References
Recursion

• A recursive function is one that calls itself.

• There are two basic types of recursive functions.
  – A *linear recursion* calls itself once.
  – A *branching recursion* calls itself two or more times.

• Examples of linear recursion
Divide and Conquer

- Many common problems can be solved using a *divide-and-conquer* approach.
- This means breaking a larger problem into pieces that can be solved independently.
- The solutions to the various pieces may then have to recombined in some way.
- Divide-and-conquer algorithms have natural implementations using branching recursions.
- **Example:** Merge sort
Implementing Merge Sort

• Here is the subroutine for implementing a basic merge sort.

• To sort an entire array the call would be `MergeSort(array, 0, length - 1)`.

```c
MergeSort(int* array, int beg, int end){
    if (beg <= end) { // Otherwise, there is nothing to do
        int mid = (beg + end)/2;
        MergeSort(array, beg, mid);
        MergeSort(array, mid + 1, end);
        Merge(array, beg, mid, end);
    }
}
```
Implementing Merge

- There are many ways to implement the merge, but here is one simple one.

- Note that this involves copying over the elements of the array.

```cpp
Merge(int* array, int beg, int mid, int end){
    int *temp1 = new int[mid - beg + 1];
    int *temp2 = new int[end - mid];
    copy(array + beg, array + mid, temp1);
    copy(array + mid + 1, array + end, temp2);
    for (int i = 0, j = 0, k = 0; k <= end - beg; k++){
        if (i == mid - beg) {array[k] = temp1[i++]; continue;}
        if (j == end - mid) {array[k] = temp2[j++]; continue;}
        array[k] = (temp2[i] < temp2[j]) ?
            temp2[i++] : temp2[j++];
    }
}
```
Correctness of Merge Sort

- Assuming the merge is done correctly, correctness of the main subroutine is straightforward.

- To show the merge works correctly, we can use a loop invariant.

- What is the loop invariant in the merge subroutine?
Some Simple Optimization

- Handling small arrays
- Eliminating copying (reduce memory requirements)
- Using sentinels

```c
Merge(int* array, int beg, int end, int mid){
    int *temp1 = new int[mid - beg + 2];
    int *temp2 = new int[end - mid + 1];
    copy(array + beg, array + mid, temp1);
    copy(array + mid + 1, array + end, temp2);
    temp1[mid - beg + 1] = MAXINT; // the sentinel
    temp2[end - mid] = MAXINT; // the sentinel
    for (int i = 0, j = 0, k = 0; k < end - beg; k++){
        array[k] = (temp2[i] < temp2[j]) ?
            temp2[i++] : temp2[j++];
    }
}
```
Analyzing Merge Sort

- Suppose the running time of merge sort is $T$.
- We analyze each piece of the algorithm separately.
  - **Divide**: This operation involves finding the midpoint of the array, which is in $\Theta(1)$.
  - **Conquer**: We recursively solve two subproblems, each of size $n/2$, which is $2T(n/2)$.
  - **Combine**: The running time of the merge subroutine is in $\Theta(n)$.
- So $T$ satisfies the following recurrence.

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
2T(n/2) + \Theta(n) & \text{if } n > 1 
\end{cases}
\]

- How do we figure out what $T$ is?