CS 457, Fall 2016

Drexel University, Department of Computer Science

Lecture 3

What we have learned so far…

- Why study algorithms
- Asymptotic notation
- Algorithm analysis (correctness and running time)
  - E.g., Insertion Sort

Insertion Sort (Correctness)

INSERTION_SORT (*A*)

```
for j =2 to A.length
        key = A[ j ]
        // Insert A[ j ] into the sorted sequence A[1 .. j − 1].
        i = j − 1
        while i > 0 and A[ i ] > key
                A[ i +1 ] = A[ i ]
                i = i − 1
        A[i + 1] = key
```

<u>Loop Invariant:</u>

At the start of each iteration of the **for** loop, the subarray A[1 ,…, j-1] consists of elements originally in A[1 ,…, j-1], but in sorted order

Things to show about invariant:

1. Initialization
2. Maintenance
3. Termination

Asymptotic Notation

- $O(g(n)) = \left\{ \begin{array}{c} f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$
- $\Omega(g(n)) = \left\{ \begin{array}{c} f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \end{array} \right\}$

- $o(g(n)) = \left\{ \begin{array}{c} f(n) : \text{for any constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ s.t.} \\ 0 \le f(n) < cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$
- $\omega(g(n)) = \left\{ \begin{array}{c} f(n) : \text{for any constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ s.t.} \\ 0 \le cg(n) < f(n) \text{ for all } n \ge n_0 \end{array} \right\}$

Asymptotic Notation Properties

- Transitivity:
  - $f(n)=O(g(n))$ and $g(n)=O(h(n))$, then $f(n)=O(h(n))$
  - $f(n)=\Omega(g(n))$ and $g(n)=\Omega(h(n))$, then $f(n)=\Omega(h(n))$
- Reflexivity: $f(n)=\Theta(f(n))$
- Transpose Symmetry: $f(n)=O(g(n))$ if and only if $g(n)=\Omega(f(n))$
- Symmetry: $f(n)=\Theta(g(n))$ if and only if $g(n)=\Theta(f(n))$
- Trichotomy: For any two real numbers a and b:
  - $a > b$, or $a = b$, or $a < b$
  - Does not hold for asymptotic notation!

Today's Lecture

- Divide and Conquer
  - Divide
    - Split the problem into smaller sub-problems of the same structure
  - Conquer
    - If sub-problem size is small enough, solve directly, o/w, solve sub-problems recursively
  - Combine
    - Merge the solutions of sub-problems into a solution of the original problem
- Merge Sort
  - Correctness
  - Running time

(N-1) + fib (N-2)); }

Fibonacci Numbers (recursion)

```
// iterative version
int fib (int N) {
        int k1, k2, k3;
        k1 = k2 = k3 = 1;
        for (int j = 3; j <= N; j++) {
                k3 = k1 + k2;
                k1 = k2;
                k2 = k3;
        }
        return k3;
}

// recursive version
int fib (int N) {
        if ((N == 1) || (N == 2)) return 1;
        else return (fib (N-1) + fib (N-2));
}
```

Merge Sort

Sorting: Given a list $\boldsymbol{A}$ of $\boldsymbol{n}$ integers, create a sorted list of these integers

- Divide
  - *Split the problem into smaller sub-problems of the same structure*
  - Split the list $\boldsymbol{A}$ into two smaller lists of size $\boldsymbol{n_1}$ and $\boldsymbol{n_2}$
- Conquer
  - *If sub-problem size is small enough, solve directly, o/w, solve sub-problems recursively*
  - Sort the two smaller lists recursively using merge sort, unless their size is small
- Combine
  - *Merge the solutions of sub-problems into a solution of the original problem*
  - Merge the two sorted lists into one, and return the result

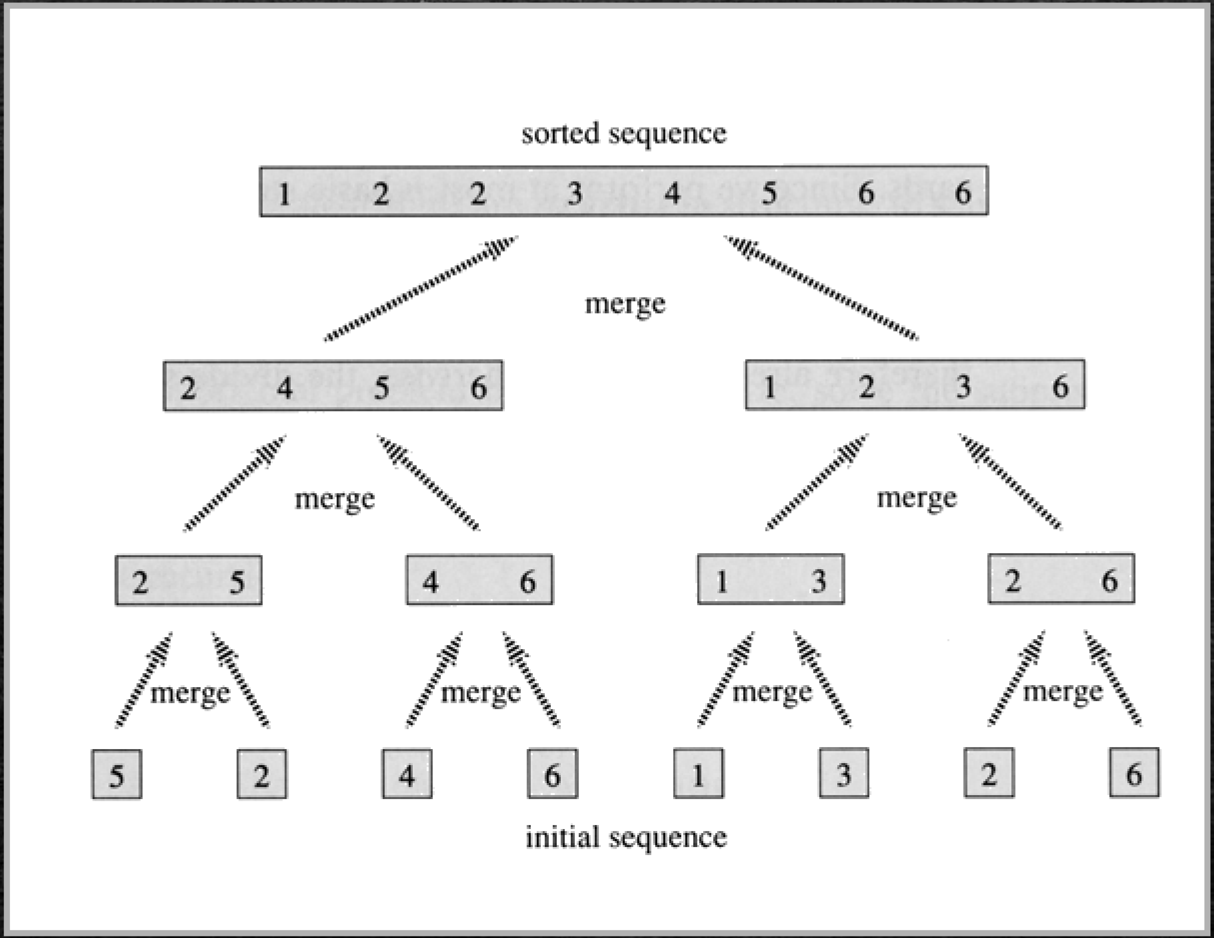
Merge Sort

To sort A[1 .. n], make initial call to MERGE-SORT (A, 1, n).

MERGE-SORT (A, p, r)

1. **if** $p < r$ // Check for base case
2. $q = \lfloor (p + r)/2 \rfloor$ // Divide step
3. MERGE-SORT (A, p, q) // Conquer step.
4. MERGE-SORT (A, $q + 1$, r) // Conquer step.
5. MERGE (A, p, q, r) // Conquer step.

Merge Sort



sorted sequence
1 2 2 3 4 5 6 6
merge
2 4 5 6
1 2 3 6
merge
merge
2 5
4 6
1 3
2 6
merge
merge
merge
merge
5 2 4 6 1 3 2 6
initial sequence

Merging Two Sorted Lists

MERGE (A, p, q, r)

1. $n_1 = q - p + 1$
2. $n_2 = r - q$
3. Create arrays $L[1\,..\,n_1 + 1]$ and $R[1\,..\,n_2 + 1]$
4. **for** $i = 1$ **to** n_1
5. $L[i] = A[p + i - 1]$
6. **for** $j = 1$ **to** n_2
7. $R[j] = A[q + j\,]$
8. $L[n_1 + 1] = \infty$
9. $R[n_2 + 1] = \infty$
10. $i = 1$
11. $j = 1$
12. **for** $k = p$ **to** r
13.   **if** $L[i\,] \leq R[\,j]$
14. $A[k] = L[i]$
15. $i = i + 1$
16.   **else**
17. $A[k] = R[j]$
18. $j = j + 1$

<u>Loop Invariant:</u>

At the start of each iteration of the for loop of lines 12-17, the subarray A[p,.. k-1] contains the k-p smallest elements of $L[1\,..\,n_1+1]$ and $R[1\,..\,n_2+1]$, in sorted order.

Moreover, L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

Things to show about invariant:

1. Initialization
2. Maintenance
3. Termination

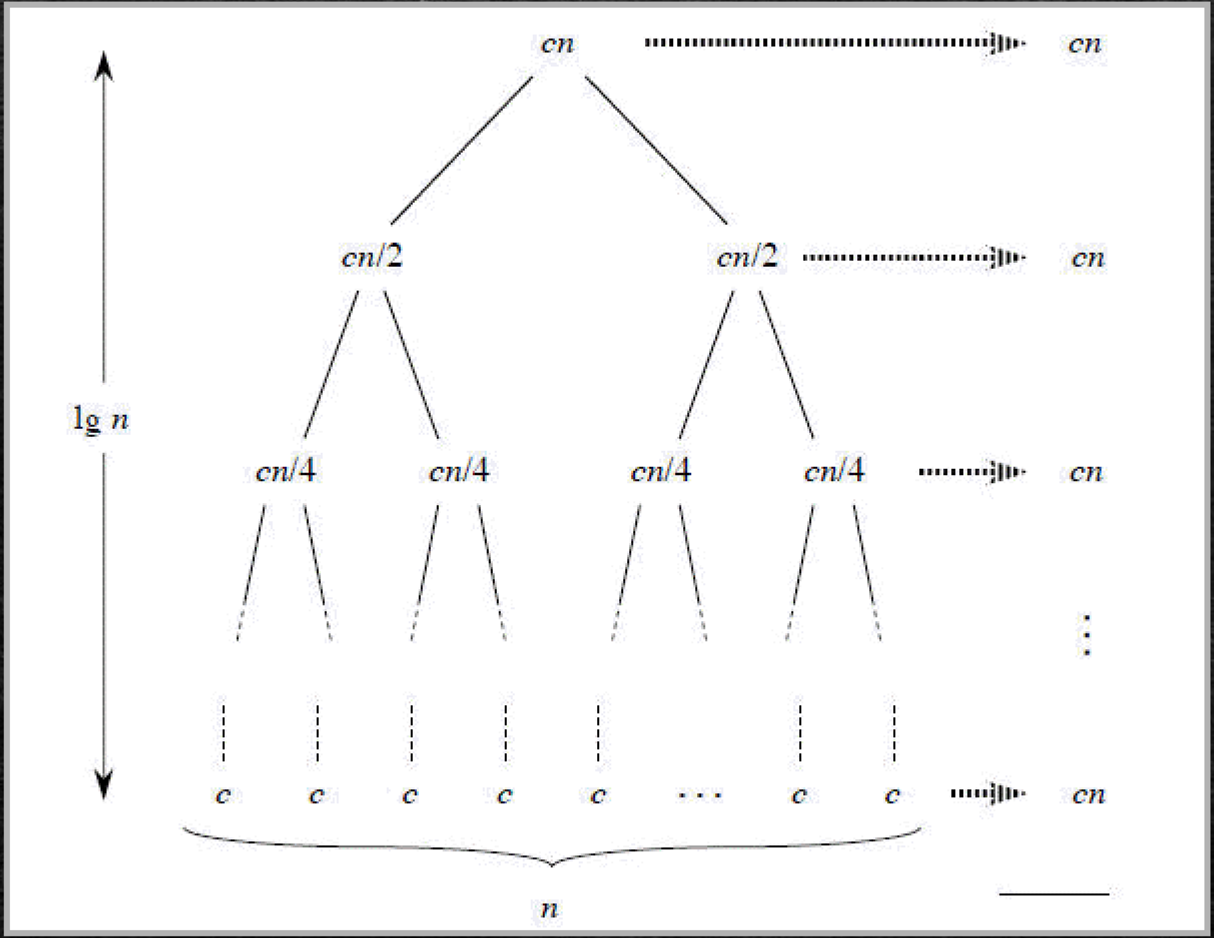
Running Time

- MERGE (A, p, q, r) needs time $\Theta(n_1 + n_2)$

- Recurrence equation for divide and conquer algorithms:
  - $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{otherwise} \end{cases}$

- Recurrence equation for Merge Sort
  - $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{otherwise} \end{cases}$

Recursion Tree



Methods for Solving Recurrences

Three methods:

1. Substitution method
   - Guess a bound and use mathematical induction to prove its correctness
2. Recursion-tree method
   - Covert into a tree and measure cost incurred at the various levels
3. Master method
   - Directly provides bounds for recurrences of the form $T(n) = T\left(\frac{n}{b}\right) + f(n)$