EE360C: Algorithms

Divide and Conquer

Summer 2019

Department of Electrical and Computer Engineering
University of Texas at Austin

Designing Algorithms

Approaches to Algorithm Design

- **incremental design** – given a sorted subarray, insertion sort inserts a single element into its proper place, generating a longer sorted subarray
- **greedy choice** – frame the problem as a choice; make the *greedy* (obvious, naïve) choice
- **divide and conquer design** – break the problem into several subproblems that are similar to but smaller than the original
  - solve the subproblems recursively
  - combine solutions to subproblems to get a solution to the original problem

Divide and Conquer

The Divide and Conquer Approach

Divide, Conquer, Combime

Divide: the problem into a number of subproblems

Conquer: the subproblems by solving them recursively. If the subproblems are small enough, just solve the subproblems directly.

Combine: the solutions to the subproblems into the solution for the original problem.

Have you used a divide and conquer algorithm before? 

Exponentiation

Given a number a and a positive integer n, consider the problem of calculating the expression a^n. How would you write a program?

SLOWPOWER(a, n)

```
1  x ← a
2  for i ← 2 to n
3        do x ← x × a
4  return x
```

What is the running time of this algorithm?
Assuming multiplication is $O(1)$, it's $\Theta(n)$.
We can do this for anything that can be multiplied, but it would have a different running time if multiplication was more expensive.

Can we do better?

Divide and Conquer Exponentiation

Consider the following equality: $a^n = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}$. This is the constant time addition of solutions to two smaller subproblems. The smaller subproblems are really similar. How would you write this program?

```
FastPower(a, n)
1  if n = 1
2     then return a
3     else
4          x ← FastPower(a, ⌊n/2⌋)
5          if n is even
6            then return x × x
7            else
8                 return x × x × a
```

What's the running time of this algorithm? How many problem instances does it make if it divides the problem in half every recursive call? What is the running time of each of those instances?

lg n instances, each takes $O(1)$ time; overall running time is $\Theta(\lg n)$.

Divide and Conquer Sorting

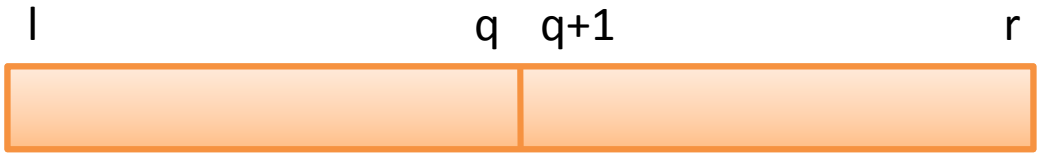
MERGE-SORT

Divide: divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.

Conquer: sort the two subsequences recursively using merge-sort.

Combine: Merge the two sorted subsequences to produce the sorted answer.

The recursion ends when the sequence to be sorted has length 1 and is therefore already sorted.



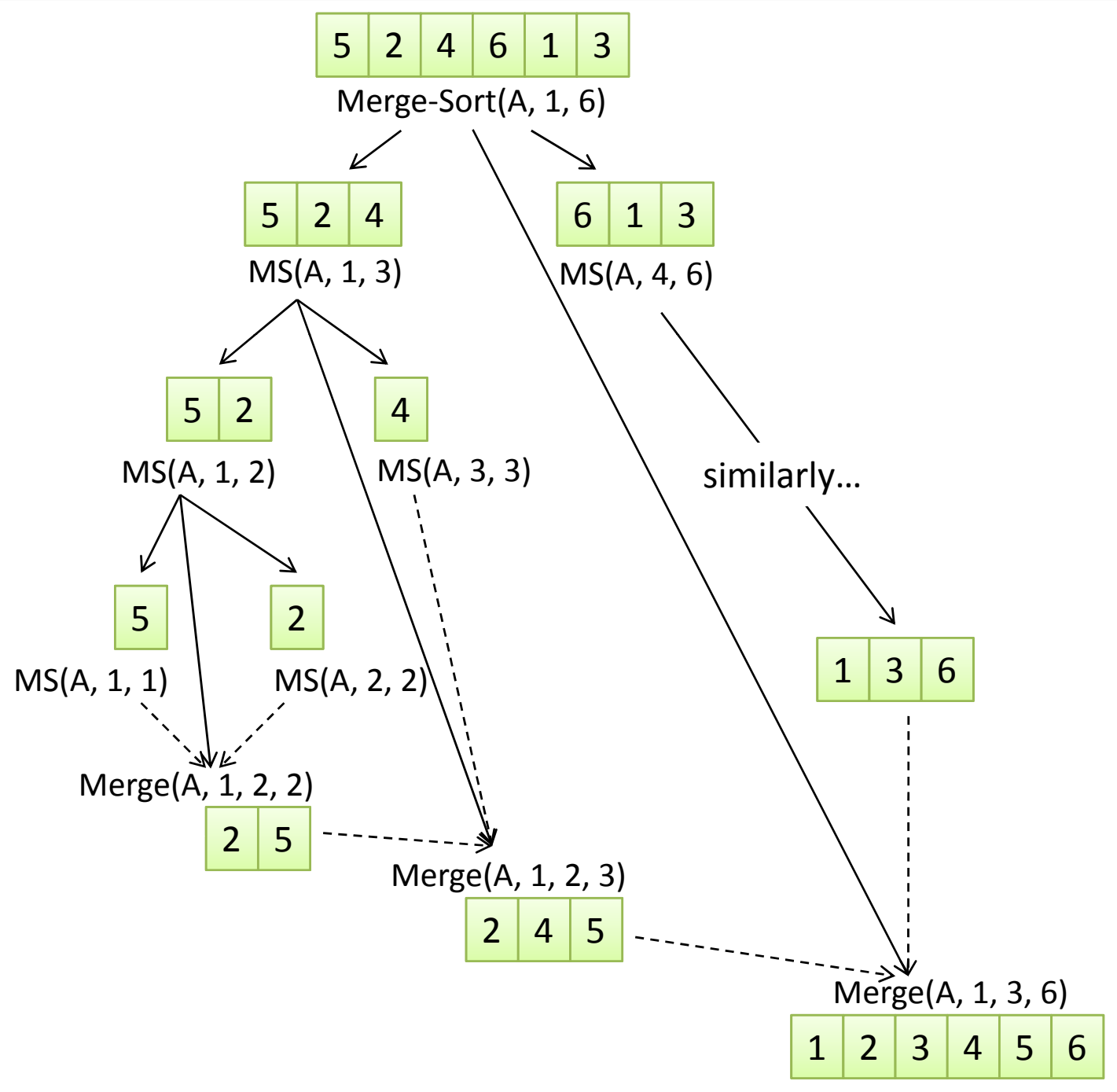
$$q = \left\lfloor \frac{(l+r)}{2} \right\rfloor$$

Merge Sort

MERGE-SORT

- Assume the availability of a subroutine MERGE(A, p, q, r), where A is an array, and p, q, and r are indices numbering elements of the array such that $p \leq q \leq r$.
- MERGE assumes that the subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ are in sorted order.
- It merges them to form a single sorted subarray that replaces the current subarray $A[p\,..\,r]$.

```
MERGE-SORT(A, p, r)
1  if p < r
2     then q ← ⌊(p + r)/2⌋
3          MERGESORT(A, p, q)
4          MERGESORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```


5 2 4 6 1 3
Merge-Sort(A, 1, 6)
5 2 4
MS(A, 1, 3)
6 1 3
MS(A, 4, 6)
5 2
MS(A, 1, 2)
4
MS(A, 3, 3)
similarly…
5
MS(A, 1, 1)
2
MS(A, 2, 2)
1 3 6
Merge(A, 1, 2, 2)
2 5
Merge(A, 1, 2, 3)
2 4 5
Merge(A, 1, 3, 6)
1 2 3 4 5 6

MERGE-SORT

MERGE(A, p, q, r)

```
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
 4  for i ← 1 to n1
 5       do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7       do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13       do if L[i] ≤ R[j]
14             then A[k] ← L[i]
15                  i ← i + 1
16             else A[k] ← R[j]
17                  j ← j + 1
```

Line 1 computes the length of the subarray $A[p\,..\,q]$ Line 2 computes the length of the subarray $A[q+1\,..\,r]$ Line 3 creates the arrays L and R Lines 4–5 copy the subarray $A[p\,..\,q]$ into L Lines 6–7 copy the subarray $A[q+1\,..\,r]$ into R Lines 8 and 9 put sentinels at the ends of L and R Lines 10–17 are the meat of the merging

Merge Details

```
1  for k ← p to r
2        do if L[i] ≤ R[j]
3              then A[k] ← L[i]
4                   i ← i + 1
5              else A[k] ← R[j]
6                   j ← j + 1
```

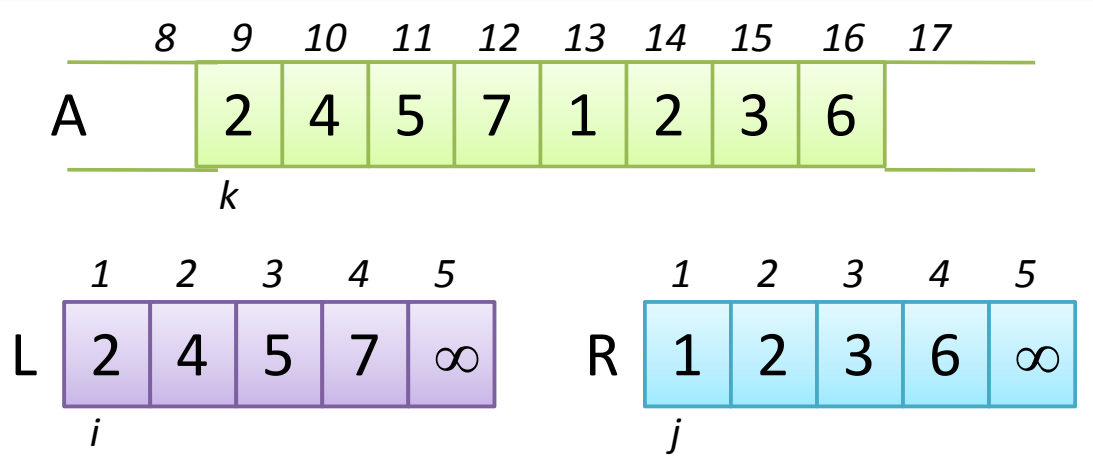
What is the loop invariant?

- At the start of each iteration of the **for** loop, the subarray $A[p \,..\, k-1]$ contains the $k-p$ smallest elements, in sorted order
- $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

The Loop Invariant Intuition

What is the loop invariant?

- At the start of each iteration of the **for** loop, the subarray $A[p\,..\,k-1]$ contains the $k-p$ smallest elements, in sorted order
- $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.



MERGE Loop Invariant Correctness

What is the loop invariant?

- At the start of each iteration of the **for** loop, the subarray $A[p\,..\,k-1]$ contains the $k-p$ smallest elements, in sorted order
- $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Initialization: *the loop invariant is true initially*. Prior to the first iteration of the loop, $k = p$, so the subarray $A[p\,..\,k-1]$ is empty.

Maintenance: *the loop invariant remains true after each iteration of the loop*. Suppose $L[i] \leq R[j]$. Because $A[p\,..\,k-1]$ already contains the $k-p$ smallest elements, after $L[i]$ is copied into $A[k]$, $A[p\,..\,k]$ will

Analyzing MERGE

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        do L[i] ← A[p + i − 1]
 6  for j ← 1 to n_2
 7        do R[j] ← A[q + j]
 8  L[n_1 + 1] ← ∞
 9  R[n_2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13        do if L[i] ≤ R[j]
14              then A[k] ← L[i]
15                   i ← i + 1
16              else A[k] ← R[j]
17                   j ← j + 1
```

Analyzing MERGE

What is the *running time* of MERGE?

- The assignments (Lines 1–3 and 8–11) all take constant time.
- Copying into L and R from A (Lines 4–7) takes $n_1 + n_2 = n$ where n is the number of elements in A.
- The **for** loop (Lines 12–17) takes constant time and is executed n times (once for every element).

$\therefore$ the running time of MERGE is $\Theta(n)$.

What is the running time of MERGE-SORT?

Analyzing Divide and Conquer Algorithms

Analyzing Divide and Conquer Algorithms

- We often express the running time of a recursive algorithm using a **recurrence equation** or just **recurrence**.
- A recurrence for a divide and conquer algorithm is based on the three steps:
  - divide
  - conquer
  - combine

Generic Divide and Conquer Recurrence

- Let $T(n)$ be the running time of a problem of size n.
- If the problem size is small enough (i.e., $n \leq c$ for some constant c), then the straightforward solution takes constant time, or $\Theta(1)$.
- Suppose the **divide** step generates a subproblems, each of which are a fraction $1/b$ of the original problem size.
- Assume that the divide step takes $D(n)$ time and the combine step takes $C(n)$ time.

General Form

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Analyzing Merge Sort

Merge-Sort Analysis

Divide: this step simply computes the middle of the subarray: $\Theta(1)$

Conquer: we define 2 subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.

Combine: the MERGE procedure takes $\Theta(n)$ times

Therefore, the *worst case running time* of MERGE-SORT is:

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

- This is a final running time of $\Theta(n \lg n)$.
- This is intuitive in this case. Why?
- We'll solve these generically using the *master method*.

The Master Method

The Master Method

Consider:

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

where $a \geq 1$ and $b > 1$ and $f(n)$ is an asymptotically positive function.

These often result from divide and conquer approaches.

The Master Method

The **master method** is a cookbook method for solving these common recurrences

The Master Method (cont.)

The Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence:

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

where we interpret n/b to be either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for constant $\epsilon > 0$, and $f(n/b) \leq cf(n)$ for constant $c < 1$, and n sufficiently large, then $T(n) = \Theta(f(n))$.

The ϵ factor is really n^{ϵ}; this is required because the function $f(n)$ must be polynomially different from $n^{\log_b a}$.

A Quick Master Method Example

- What is the running time of:

$$T(n) = 2T(n/2) + \Theta(n)$$

More Master Method Examples

- What is the running time of:

$$T(n) = 9T(n/3) + n$$

More Master Method Examples

- What is the running time of:

$$T(n) = T(2n/3) + 1$$

More Master Method Examples

- What is the running time of:

$$T(n) = 3T(n/4) + n \lg n$$

More Master Method Examples

- What is the running time of:

$$T(n) = 2T(n/2) + n \lg n$$

The Master Method Revisited

The master method can also be written as:

New Master Method

Solve any recurrence of the form:

$$\begin{aligned} T(n) &= aT(n/b) + \Theta(n^l (\lg n)^k) \\ T(c) &= \Theta(1) \text{ for some constant } c \end{aligned}$$

where $a \geq 1$, $b > 1$, $l \geq 0$, and $k \geq 0$.

The goal is to compare l and $\log_b a$. The intuition is that $n^{\log_b a}$ is the number of times the termination condition ($T(c)$) is reached.

The Master Method Revisited (cont.)

The New Master Method Cases

Then we can restate the three cases as:

Case 1: $l < \log_b a$. Then $T(n) = \Theta(n^{\log_b a})$.

Case 2: $l = \log_b a$. Then $T(n) = \Theta(f(n) \lg n)$. Which is equivalent to $T(n) = \Theta(n^l (\lg n)^{k+1})$. Which is ultimately equivalent to (a more generic statement of) what we had before: $T(n) = \Theta(n^{\log_b a} (\lg n)^{k+1})$.

Case 3: $l > \log_b a$. Then $T(n) = \Theta(f(n)) = \Theta(n^l (\lg n)^k)$.

Questions?