COMP251: Divide-and-Conquer (1)

Jérôme Waldispühl & Guilia Alberini
School of Computer Science
McGill University

Based on (Kleinberg & Tardos, 2005) and slides by K. Wayne & Snoeyink

Algorithm design techniques

	Greedy choice	Optimal Substructure	Recursion	Examples
Greedy	✓	✓	✓	Kruskall,Huffman, Dijkstra…
Dynamic Programming		✓	✓	Weighted interval scheduling, Bellman-Ford…
Divide-and-conquer			✓	MergeSort, Karatsuba…

Outline

- **MergeSort**
  - Definition
  - Correctness
  - Complexity analysis
- **Integer multiplication**
  - “Naïve” recursive algorithm
  - Karatsuba

Objective: Designing a divide-and-conquer algorithm and characterizing its running time.

Divide and Conquer

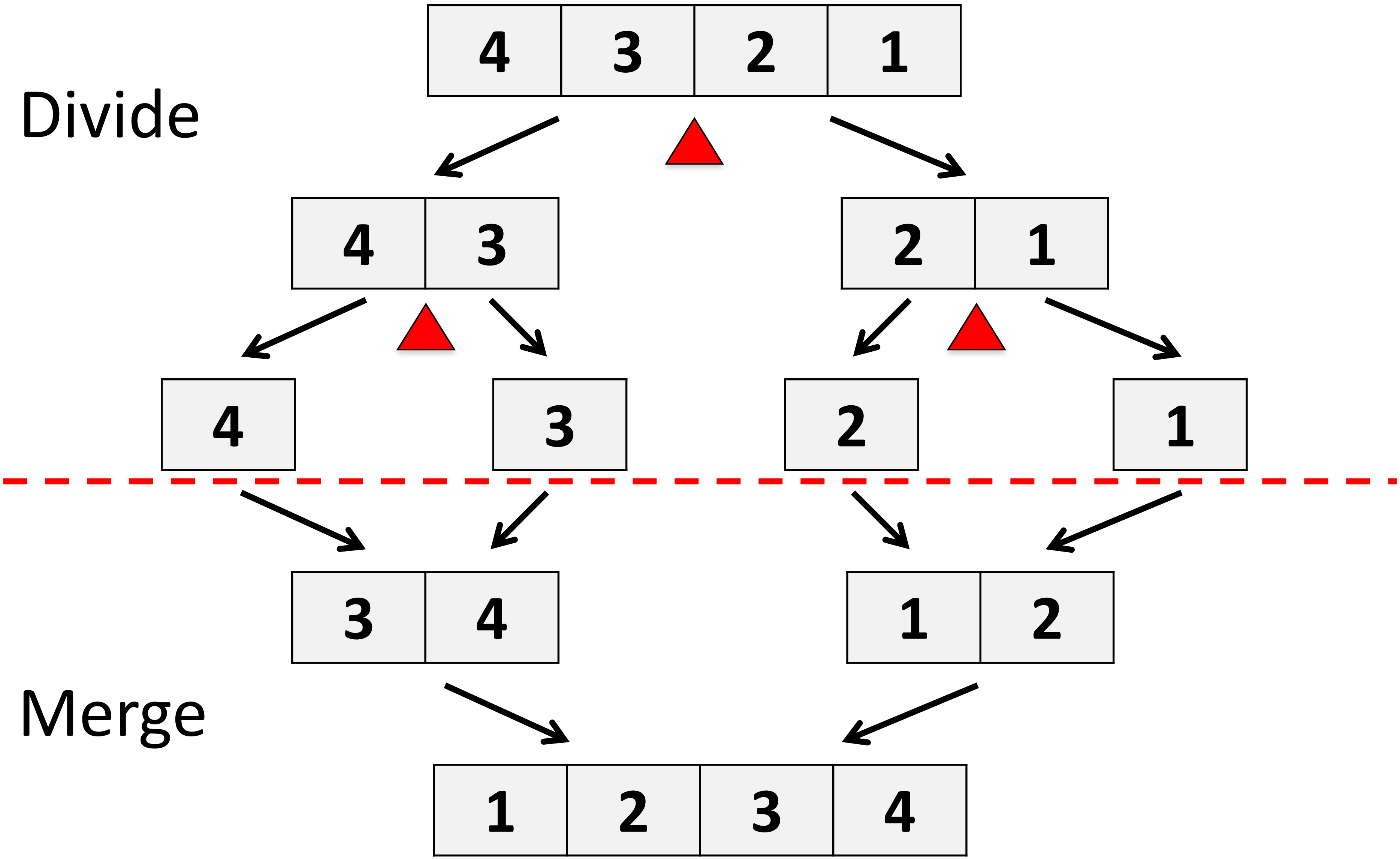
- Recursive in structure
  - ***Divide*** the problem into sub-problems that are similar to the original but smaller in size
  - ***Conquer*** the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  - ***Combine*** the solutions to create a solution to the original problem

An Example: Merge Sort

***Sorting Problem*:** Sort a sequence of *n* elements into non-decreasing order.

- ***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.

Merge Sort - Example



Merge-Sort (A, p, r)

INPUT: a sequence of *n* numbers stored in array A

OUTPUT: an ordered sequence of *n* numbers

```
MergeSort (A, p, r)   // sort A[p..r] by divide & conquer
if p < r
    then q ← ⌊(p+r)/2⌋
        MergeSort (A, p, q)
        MergeSort (A, q+1, r)
        Merge (A, p, q, r) // merges A[p..q] with A[q+1..r]
```

Initial Call: MergeSort(*A*, 1, *n*)

Procedure Merge

Merge(*A, p, q, r*)

$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
for $i \leftarrow 1$ **to** n_1
    do $L[i] \leftarrow A[p + i - 1]$
for $j \leftarrow 1$ **to** n_2
    do $R[j] \leftarrow A[q + j]$
$L[n_1+1] \leftarrow \infty$
$R[n_2+1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow p$ **to** r
    do if $L[i] \leq R[j]$
        then $A[k] \leftarrow L[i]$
            $i \leftarrow i + 1$
        else $A[k] \leftarrow R[j]$
            $j \leftarrow j + 1$

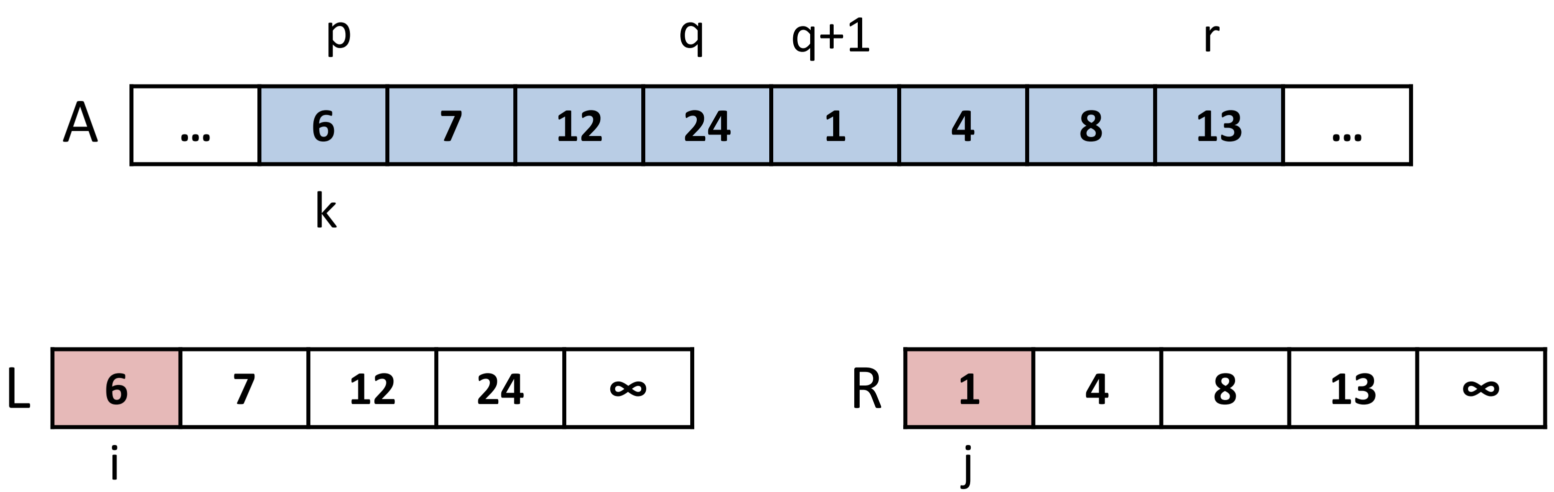
Input: Array containing sorted subarrays $A[p..q]$ and $A[q+1..r]$.

Output: Merged sorted subarray in $A[p..r]$.

Sentinels, to avoid having to check if either subarray is fully copied at each step.

Merge (example)

Execution of a call to Merge(A,p,q,r):

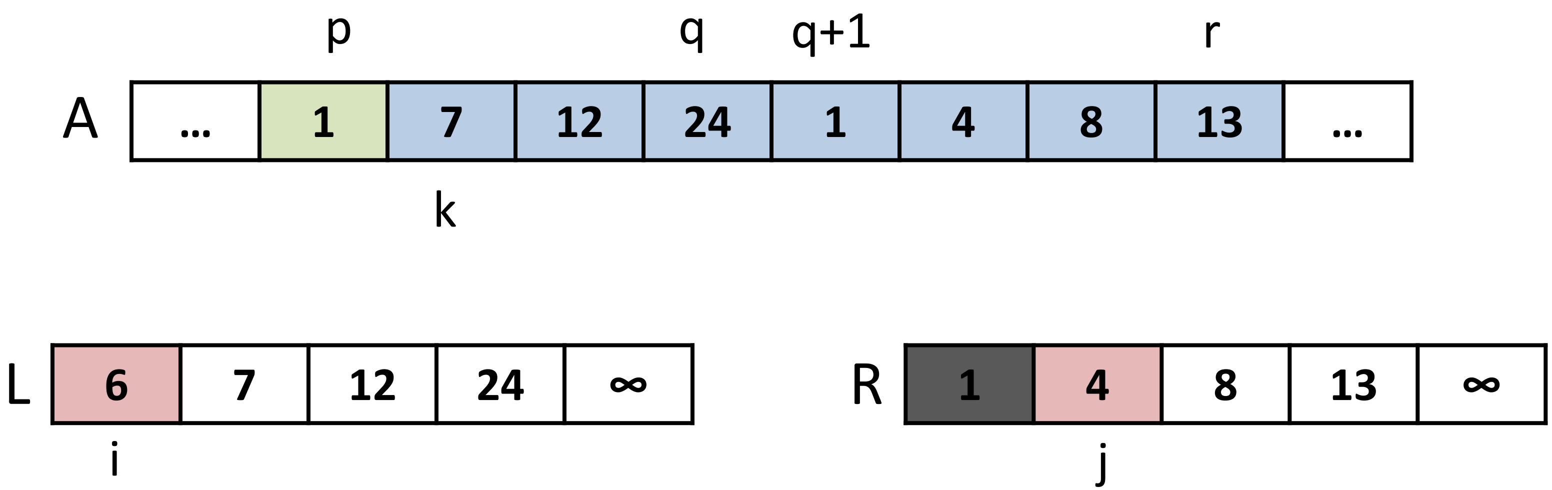


The first and second half of A[p,r] are already sorted.

Merge creates an array for each first and second halves.

Merge (example)

Execution of a call to Merge(A,p,q,r):

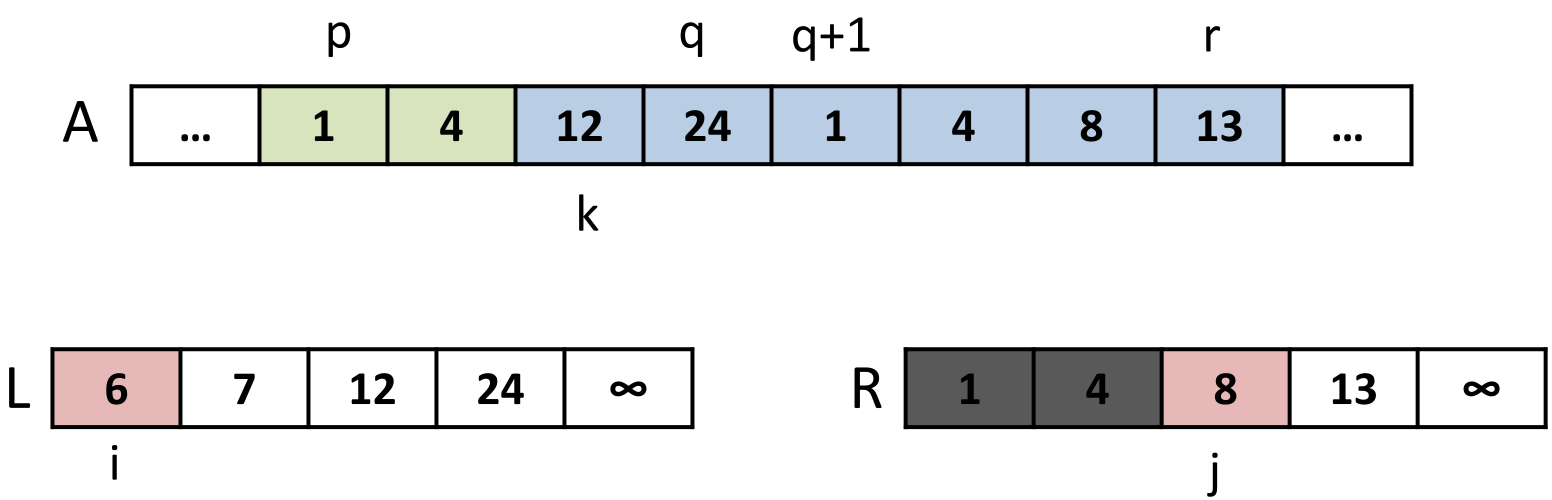


Pick the lowest value in L and R and place it at the lowest index not used yet (i.e., index k)

Note: We sort in-place

Merge (example)

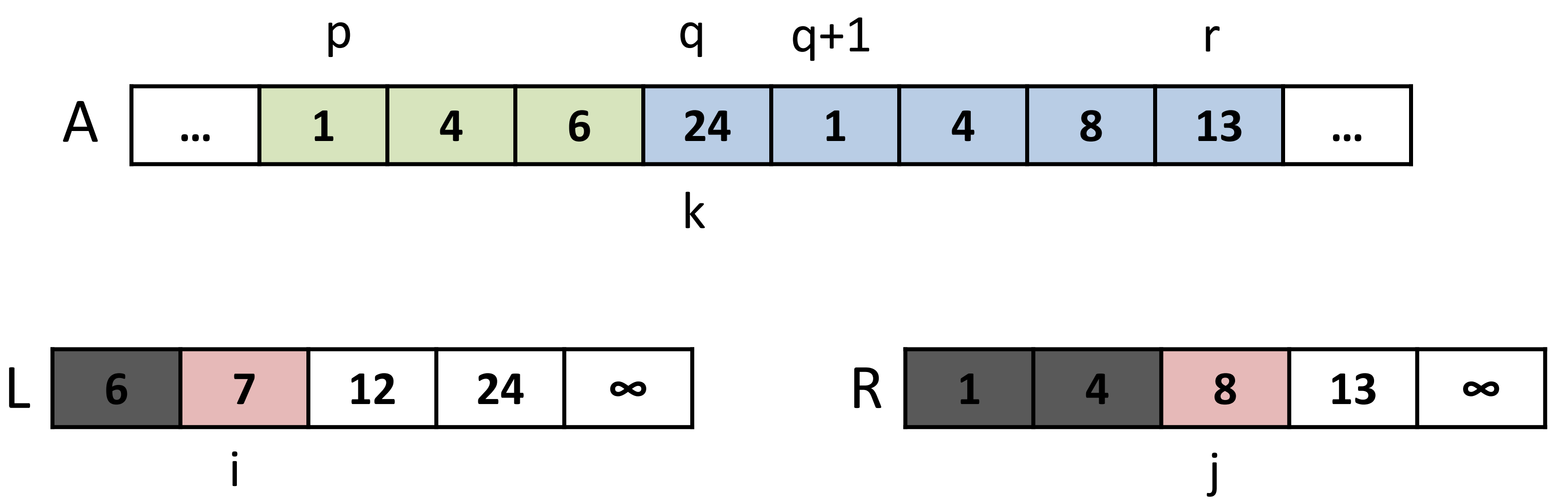
Execution of a call to Merge(A,p,q,r):



Iterate the same process

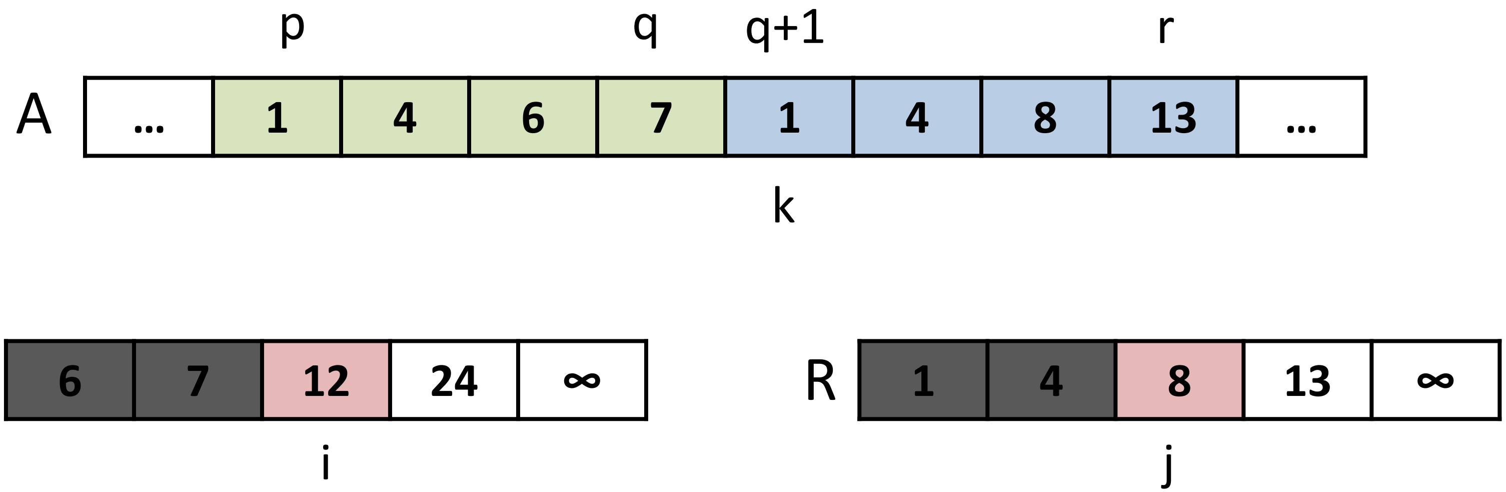
Merge (example)

Execution of a call to Merge(A,p,q,r):



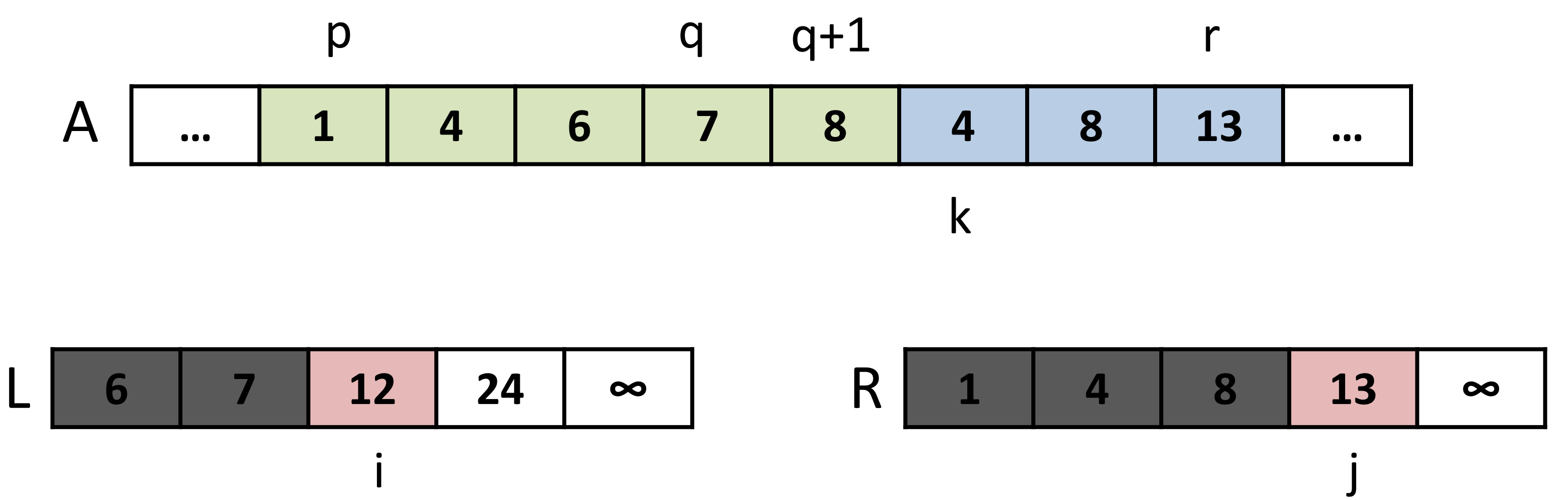
Merge (example)

Execution of a call to Merge(A,p,q,r):



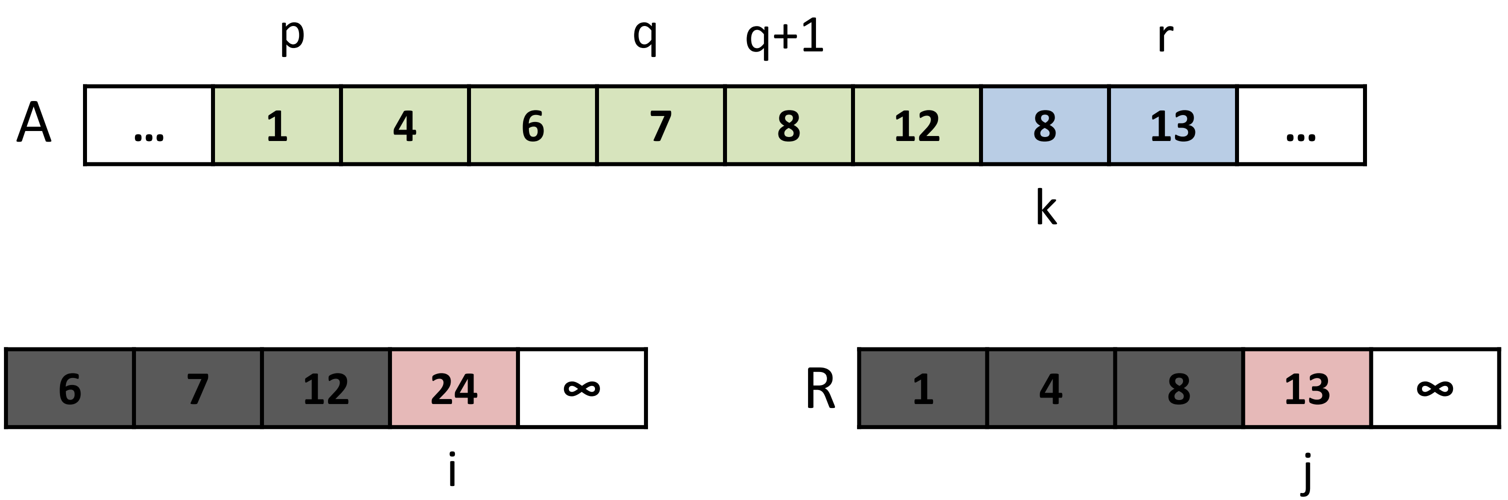
Merge (example)

Execution of a call to Merge(A,p,q,r):



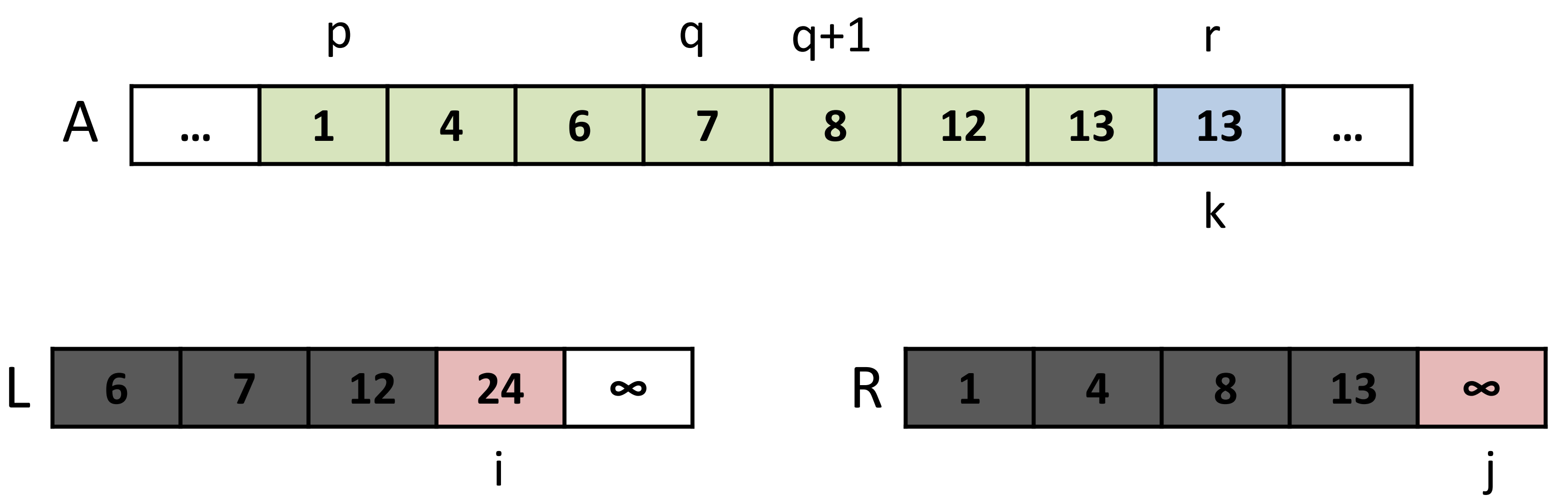
Merge (example)

Execution of a call to Merge(A,p,q,r):



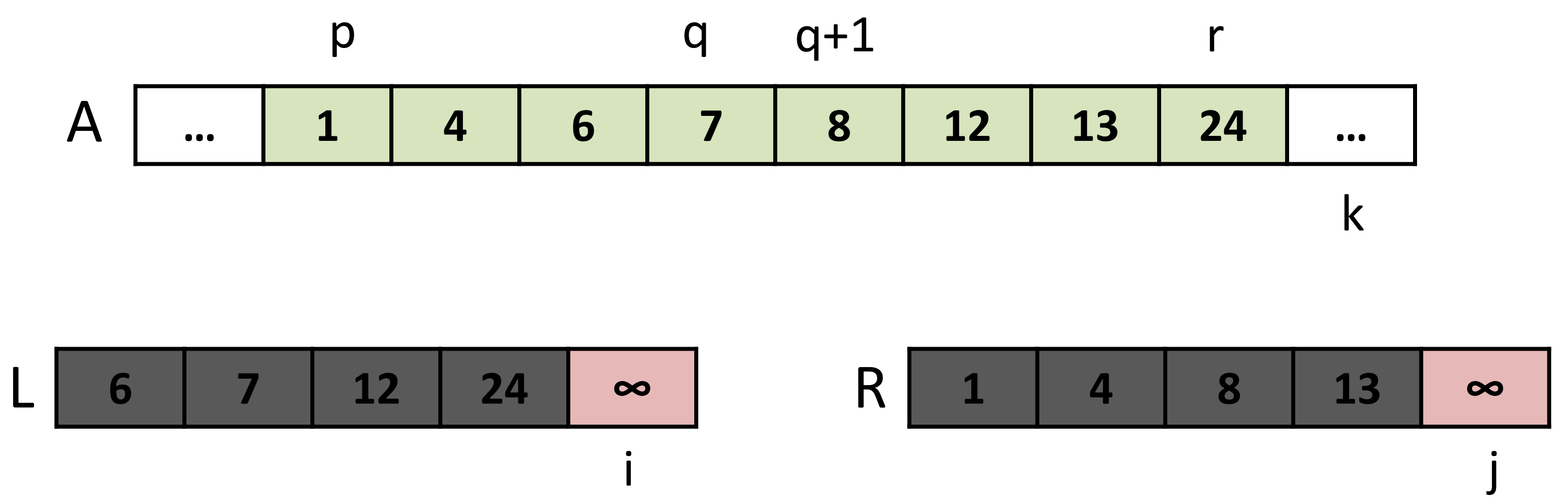
Merge (example)

Execution of a call to Merge(A,p,q,r):



Merge (example)

Execution of a call to Merge(A,p,q,r):



We stop when A[p,r] has been filled

Correctness of Merge

Merge(*A*, *p*, *q*, *r*)

```
n1 ← q – p + 1
n2 ← r – q
for i ← 1 to n1
    do L[i] ← A[p + i – 1]
for j ← 1 to n2
    do R[j] ← A[q + j]
L[n1+1] ← ∞
R[n2+1] ← ∞
i ← 1
j ← 1
for k ←p to r
    do if L[i] ≤ R[j]
        then A[k] ← L[i]
             i ← i + 1
        else A[k] ← R[j]
             j ← j + 1
```

Loop Invariant property (main *for* loop)

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

Initialization:

Before the first iteration:

- $A[p..k-1]$ is empty.
- $i = j = 1$.
- $L[1]$ and $R[1]$ are the smallest elements of L and R not copied to A.

Correctness of Merge

Merge(*A*, *p*, *q*, *r*)

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

Maintenance:

Case 1: $L[i] \le R[j]$

- By LI, A contains $p - k$ smallest elements of L and R in sorted order.
- By LI, $L[i]$ and $R[j]$ are the smallest elements of L and R not yet copied into A.
- Line 13 results in A containing $p - k + 1$ smallest elements (again in sorted order). Incrementing i and k reestablishes the LI for the next iteration.

Case 2: Similar arguments with $L[i] > R[j]$

Termination:

- On termination, $k = r + 1$.
- By LI, A contains $r - p + 1$ smallest elements of L and R in sorted order.
- L and R together contain $r - p + 3$ elements including the two sentinels. So, all elements are sorted!

Analysis of Merge Sort

- Running time ***T*(*n*)** of Merge Sort:
- Divide: computing the middle takes $\Theta(1)$
- Conquer: solving 2 sub-problems takes $2T(n/2)$
- Combine: merging n elements takes $\Theta(n)$
- Total:

$$T(n) = \Theta(1) \qquad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \Theta(n) \quad \text{if } n > 1$$

Solution: $T(n) = \Theta(n \lg n)$

We will describe two ways to prove it. Though, to make our task easier, we will assume that n is a power of 2.

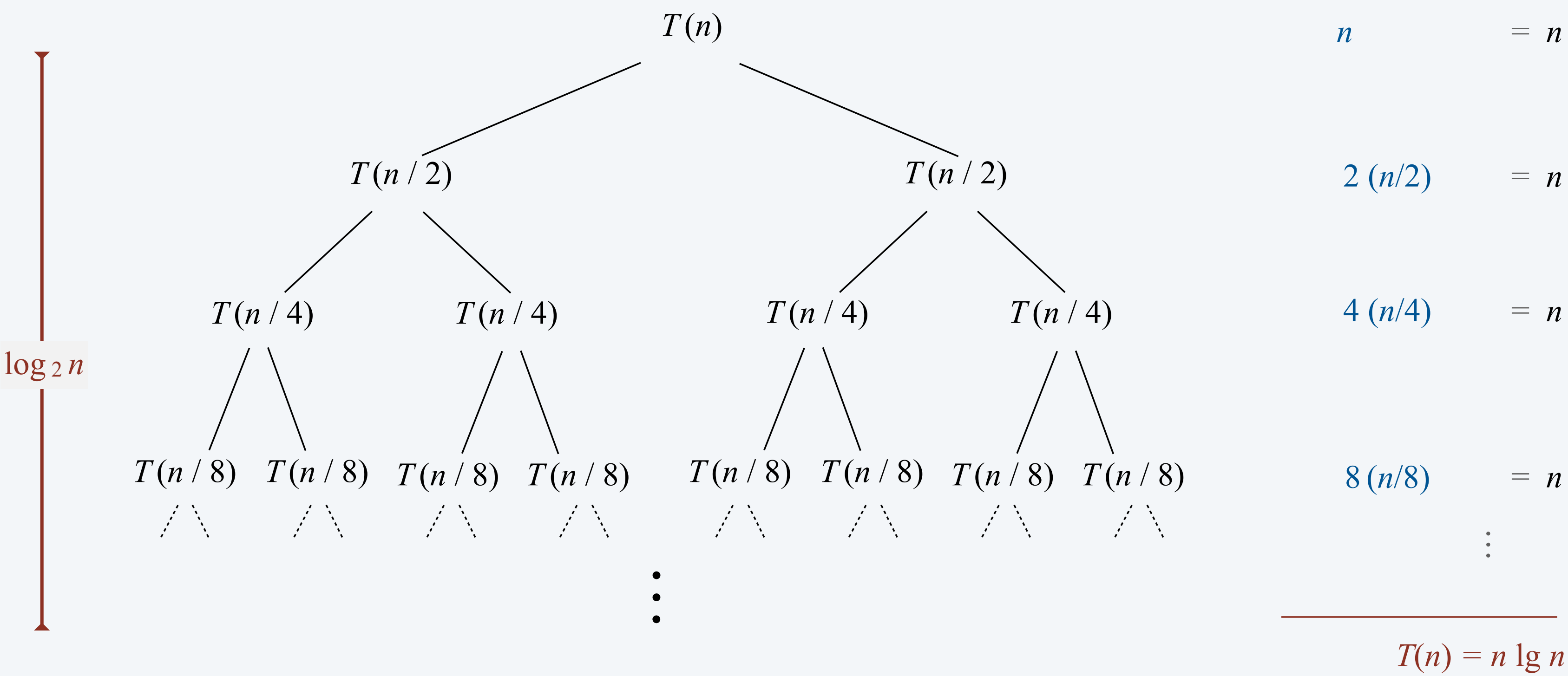
Divide-and-conquer recurrence: proof by recursion tree

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

assuming n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2\,T(n/2) + n & \text{otherwise} \end{cases}$$

Pf 1.



Proof by induction

Proposition. If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.

assuming n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2\,T(n/2) + n & \text{otherwise} \end{cases}$$

Pf 2. [by induction on n]

- Base case: when $n = 1$, $T(1) = 0$.
- Inductive hypothesis: assume $T(n) = n \log_2 n$.
- Goal: show that $T(2n) = 2n \log_2 (2n)$.

This is an example where the inductive hypothesis is used to prove T(2n) instead of T(n+1).

$$\begin{aligned} T(2n) &= 2\,T(n) + 2n \\ &= 2\,n \log_2 n + 2n \\ &= 2\,n\,(\log_2 (2n) - 1) + 2n \\ &= 2\,n \log_2 (2n). \quad \blacksquare \end{aligned}$$

$\log(a \cdot b) = \log(a) + \log(b)$

Arithmetic operations

Given 2 (binary) numbers, we want efficient algorithms to:

- Add 2 numbers
- **Multiply 2 numbers** (using divide-and-conquer!)

Integer addition

Addition. Given two n-bit integers a and b, compute $a + b$.

Subtraction. Given two n-bit integers a and b, compute $a - b$.

Grade-school algorithm. $\Theta(n)$ bit operations.

1	1	1	1	1	1	0	1	
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
1	0	1	0	1	0	0	1	0

Remark. Grade-school addition and subtraction algorithms are asymptotically optimal.

Integer multiplication

Multiplication. Given two n-bit integers a and b, compute $a \times b$.

Grade-school algorithm. $\Theta(n^2)$ bit operations.

								1	1	0	1	0	1	0	1
							×	0	1	1	1	1	1	0	1
								1	1	0	1	0	1	0	1
							0	0	0	0	0	0	0	0	
						1	1	0	1	0	1	0	1		
					1	1	0	1	0	1	0	1			
				1	1	0	1	0	1	0	1				
			1	1	0	1	0	1	0	1					
		1	1	0	1	0	1	0	1						
	0	0	0	0	0	0	0	0							
0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	1

Conjecture. [Kolmogorov 1952] Grade-school algorithm is optimal.

Theorem. [Karatsuba 1960] Conjecture is wrong.

Divide-and-conquer multiplication

To multiply two n-bit integers x and y:

- Divide x and y into low- and high-order bits.
- Multiply four ½n-bit integers, recursively.
- Add and shift to obtain result.

$$m = \lceil n / 2 \rceil$$

$$a = \lfloor x / 2^m \rfloor \quad b = x \bmod 2^m$$

$$c = \lfloor y / 2^m \rfloor \quad d = y \bmod 2^m$$

use bit shifting to compute 4 terms

$$\underbrace{(2^m a + b)}_{x} \underbrace{(2^m c + d)}_{y} = 2^{2m} \underset{①}{ac} + 2^m (\underset{②}{bc} + \underset{③}{ad}) + \underset{④}{bd}$$

Ex. $x = \underbrace{1\,0\,0\,0}_{a}\underbrace{1\,1\,0\,1}_{b} \qquad y = \underbrace{1\,1\,1\,0}_{c}\underbrace{0\,0\,0\,1}_{d}$

Divide-and-conquer multiplication

$\text{MULTIPLY}(x, y, n)$

IF $(n = 1)$

  RETURN $x \times y$.

ELSE

  $m \leftarrow \lceil n / 2 \rceil$.

  $a \leftarrow \lfloor x / 2^m \rfloor$; $b \leftarrow x \bmod 2^m$.

  $c \leftarrow \lfloor y / 2^m \rfloor$; $d \leftarrow y \bmod 2^m$.

  $e \leftarrow \text{MULTIPLY}(a, c, m)$.

  $f \leftarrow \text{MULTIPLY}(b, d, m)$.

  $g \leftarrow \text{MULTIPLY}(b, c, m)$.

  $h \leftarrow \text{MULTIPLY}(a, d, m)$.

  RETURN $2^{2m}\, e + 2^m\, (g + h) + f$.

Divide-and-conquer multiplication analysis

Proposition. The divide-and-conquer multiplication algorithm requires $\Theta(n^2)$ bit operations to multiply two n-bit integers.

Pf. Apply case 1 of the master theorem to the recurrence:

$$T(n) = \underbrace{4T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, shift}} \Rightarrow T(n) = \Theta(n^2)$$

Next class!

Karatsuba trick

To compute middle term $bc + ad$, use identity:

$$bc + ad = ac + bd - (a - b)(c - d)$$

$$m = \lceil n / 2 \rceil$$

$$a = \lfloor x / 2^m \rfloor \quad b = x \bmod 2^m$$

$$c = \lfloor y / 2^m \rfloor \quad d = y \bmod 2^m$$

middle term

$$(2^m a + b)(2^m c + d) = 2^{2m} ac + 2^m \underbrace{(bc + ad)}_{\text{middle term}} + bd$$

$$= 2^{2m} ac + 2^m (ac + bd - (a - b)(c - d)) + bd$$

1 1 3 2 3

Bottom line. Only three multiplication of $n/2$-bit integers.

Karatsuba multiplication

KARATSUBA-MULTIPLY(x, y, n)

IF $(n = 1)$

RETURN $x \times y$.

ELSE

$m \leftarrow \lceil n / 2 \rceil$.

$a \leftarrow \lfloor x / 2^m \rfloor$; $b \leftarrow x \bmod 2^m$.

$c \leftarrow \lfloor y / 2^m \rfloor$; $d \leftarrow y \bmod 2^m$.

$e \leftarrow$ KARATSUBA-MULTIPLY(a, c, m).

$f \leftarrow$ KARATSUBA-MULTIPLY(b, d, m).

$g \leftarrow$ KARATSUBA-MULTIPLY$(a - b, c - d, m)$.

RETURN $2^{2m} e + 2^m (e + f - g) + f$.

Karatsuba analysis

Proposition. Karatsuba's algorithm requires $O(n^{1.585})$ bit operations to multiply two n-bit integers.

Pf. Apply case 1 of the master theorem to the recurrence:

$$T(n) = 3\,T(n/2) + \Theta(n) \quad \Rightarrow \quad T(n) = \Theta(n^{\lg 3}) = O(n^{1.585}).$$

Next class!

Practice. Faster than grade-school algorithm for about 320-640 bits.

Integer arithmetic reductions

Integer multiplication. Given two n-bit integers, compute their product.

problem	arithmetic	running time
integer multiplication	$a \times b$	$\Theta(M(n))$
integer division	a / b, $a \bmod b$	$\Theta(M(n))$
integer square	a^2	$\Theta(M(n))$
integer square root	$\lfloor \sqrt{a} \rfloor$	$\Theta(M(n))$

integer arithmetic problems with the same complexity as integer multiplication

History of asymptotic complexity of integer multiplication

year	algorithm	order of growth
?	brute force	$\Theta(n^2)$
1962	Karatsuba-Ofman	$\Theta(n^{1.585})$
1963	Toom-3, Toom-4	$\Theta(n^{1.465})$, $\Theta(n^{1.404})$
1966	Toom-Cook	$\Theta(n^{1+\varepsilon})$
1971	Schönhage–Strassen	$\Theta(n \log n \log \log n)$
2007	Fürer	$n \log n \; 2^{O(\log^* n)}$
?	?	$\Theta(n)$

These results have even been improved since 2007...

number of bit operations to multiply two n-bit integers

used in Maple, Mathematica, gcc, cryptography, ...

Remark. GNU Multiple Precision Library uses one of five different algorithm depending on size of operands.

GMP «Arithmetic without limitations»