Divide and Conquer

Divide-and-Conquer

- Divide-and-conquer.
  - Break up problem into several parts.
  - Solve each part recursively.
  - Combine solutions to sub-problems into overall solution.

- Most common usage.
  - Break up problem of size n into two equal parts of size $\frac{1}{2}$n.
  - Solve two parts recursively.
  - Combine two solutions into overall solution in linear time.

- Consequence.
  - Brute force: n^2.
  - Divide-and-conquer: n log n.

- Analysis.
  - involves solving a recurrence relation that bounds the running time recursively in terms of the running time on smaller instances.

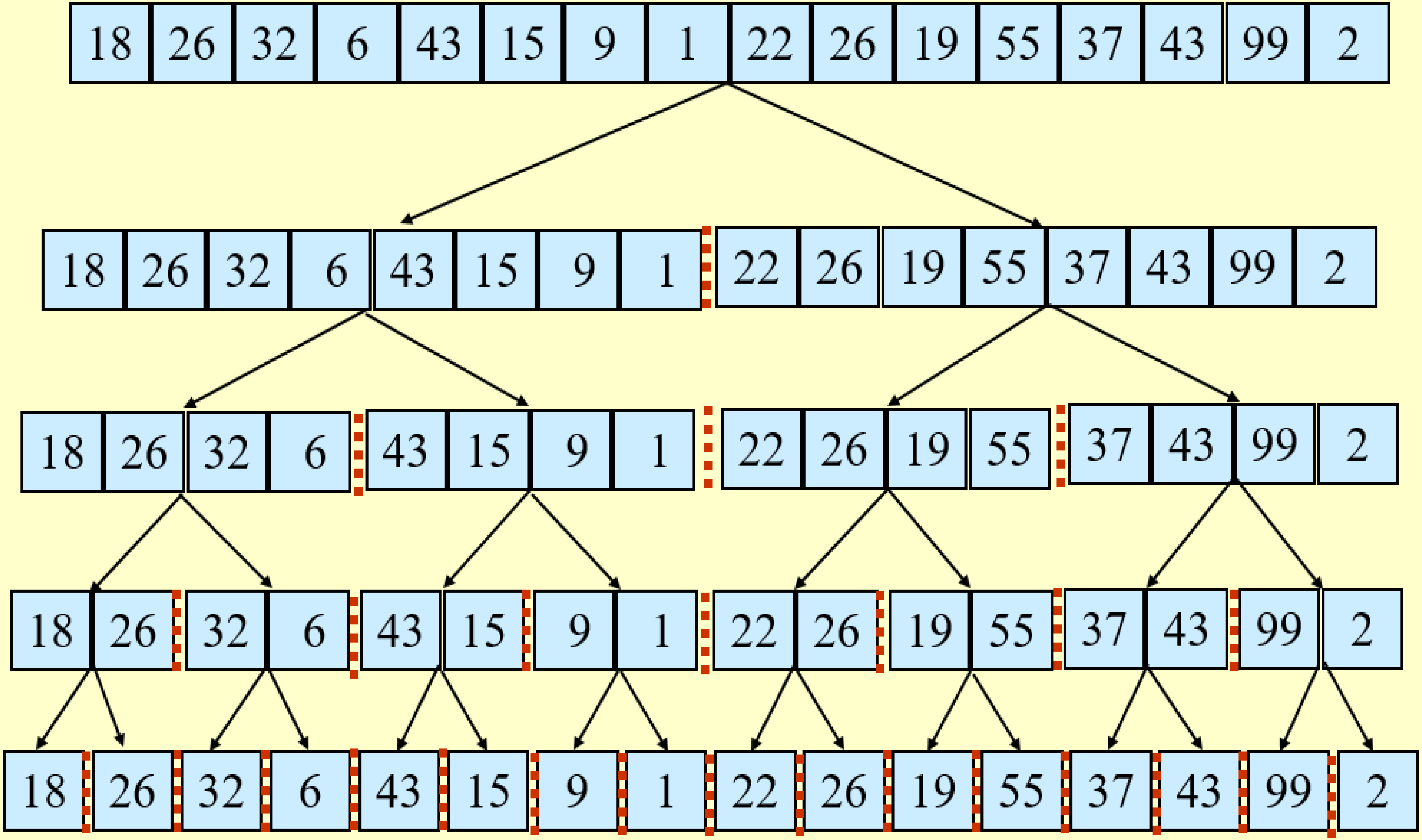
Merge Sort

Merge Sort

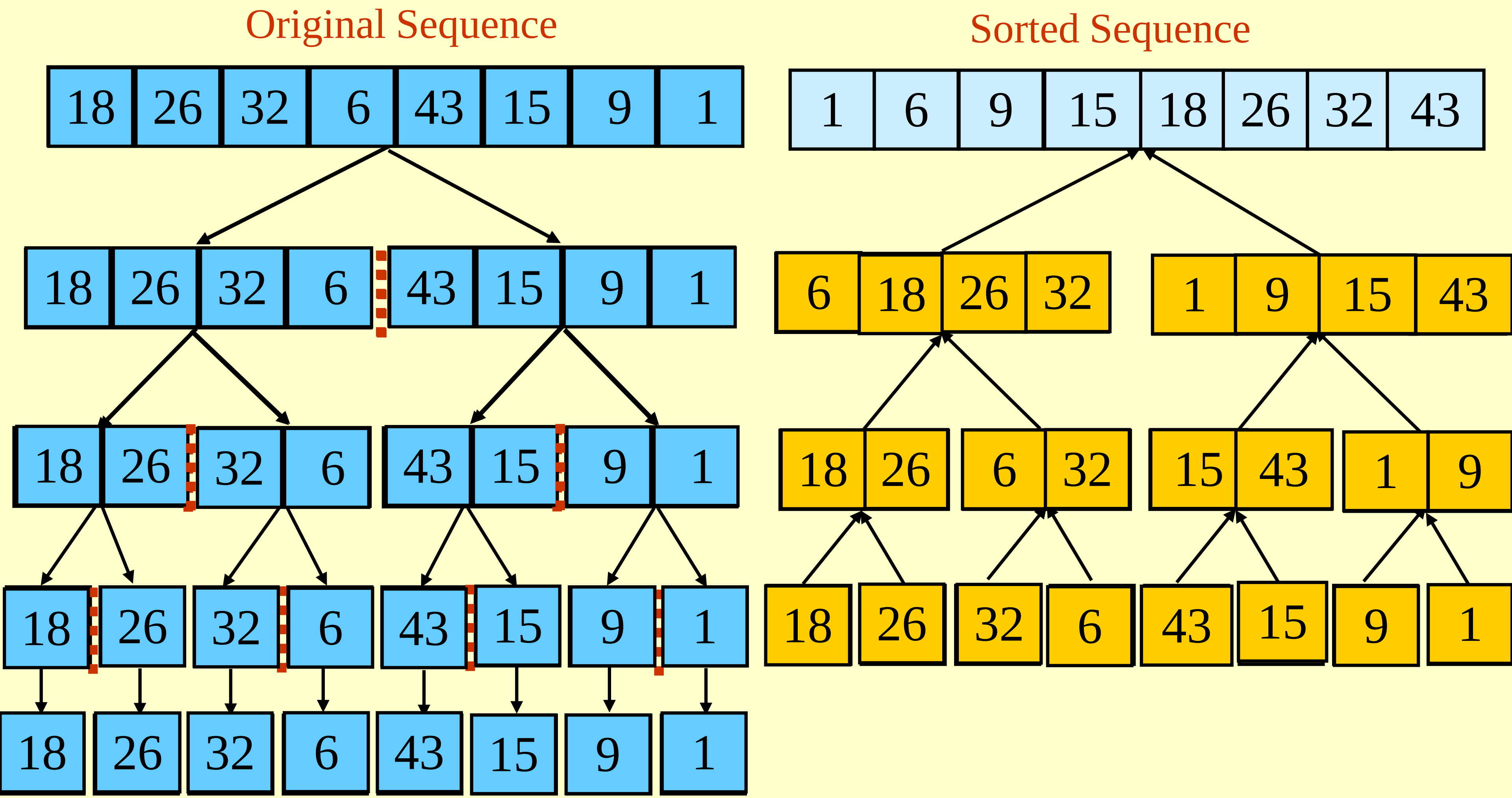
Sorting Problem: Sort a sequence of n elements into non-decreasing order.

- ***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 – 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)
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
```

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

A

…	1	6	8	9	26	32	42	43	…

k

L

6	8	26	32	∞

i

R

1	9	42	43	∞

j

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 for the *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 $n_1 \leftarrow q - p + 1$
2 $n_2 \leftarrow r - q$
3 **for** $i \leftarrow 1$ **to** n_1
4   **do** $L[i] \leftarrow A[p + i - 1]$
5 **for** $j \leftarrow 1$ **to** n_2
6   **do** $R[j] \leftarrow A[q + j]$
7 $L[n_1+1] \leftarrow \infty$
8 $R[n_2+1] \leftarrow \infty$
9 $i \leftarrow 1$
10 $j \leftarrow 1$
11 **for** $k \leftarrow p$ **to** r
12   **do if** $L[i] \leq R[j]$
13   **then** $A[k] \leftarrow L[i]$
14             $i \leftarrow i + 1$
15   **else** $A[k] \leftarrow R[j]$
16             $j \leftarrow j + 1$

Maintenance:

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

- 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.

Similarly for $L[i] > R[j]$.

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. All but the two sentinels have been copied back into A.

Analysis of Merge Sort

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

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

$\Rightarrow T(n) = \Theta(n \lg n)$

Counting Inversions

Counting Inversions

- Music site tries to match your song preferences with others.
- You rank n songs.
- Music site consults database to find people with similar tastes.

- Similarity metric: number of inversions between two rankings.
- My rank: 1, 2, …, n.
- Your rank: $a_1, a_2, \ldots, a_n$.
- Songs i and j inverted if $i < j$, but $a_i > a_j$.

Songs

Inversions 3-2, 4-2

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

Brute force: check all $\Theta(n^2)$ pairs i and j.

Applications

- Applications.
  - Voting theory.
  - Collaborative filtering.
  - Measuring the "sortedness" of an array.
  - Sensitivity analysis of Google's ranking function.
  - Rank aggregation for meta-searching on the Web.
  - Nonparametric statistics (e.g., Kendall's Tau distance).

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

1	5	4	8	10	2	6	9	12	11	3	7

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- Divide: separate list into two pieces.

1	5	4	8	10	2	6	9	12	11	3	7

Divide: $O(1)$.

1	5	4	8	10	2

6	9	12	11	3	7

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- Divide: separate list into two pieces.
- Conquer: recursively count inversions in each half.

1	5	4	8	10	2	6	9	12	11	3	7

Divide: O(1).

1	5	4	8	10	2

6	9	12	11	3	7

Conquer: 2T(n / 2)

5 blue-blue inversions

5-4, 5-2, 4-2, 8-2, 10-2

8 green-green inversions

6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- Divide: separate list into two pieces.
- Conquer: recursively count inversions in each half.
- Combine: count inversions where a_i and a_j are in different halves, and return sum of three quantities.

1	5	4	8	10	2	6	9	12	11	3	7

Divide: O(1).

1	5	4	8	10	2

5 blue-blue inversions

6	9	12	11	3	7

8 green-green inversions

Conquer: 2T(n / 2)

9 blue-green inversions
5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Combine: ???

Total = 5 + 8 + 9 = 22.

Counting Inversions: Combine

Combine: count blue-green inversions

- Assume each half is sorted.
- Count inversions where a_i and a_j are in different halves.
- Merge two sorted halves into sorted whole.

to maintain sorted invariant

3	7	10	14	18	19

2	11	16	17	23	25
6	3	2	2	0	0

13 blue-green inversions: 6 + 3 + 2 + 2 + 0 + 0

Count: $O(n)$

2	3	7	10	11	14	16	17	18	19	23	25

Merge: $O(n)$

$$T(n) \le T\left(\lfloor n/2 \rfloor\right) + T\left(\lceil n/2 \rceil\right) + O(n) \Rightarrow \mathrm{T}(n) = O(n \log n)$$

Counting Inversions: Implementation

Pre-condition. [Merge-and-Count] A and B are sorted.
Post-condition. [Sort-and-Count] L is sorted.

```
Sort-and-Count(L) {
if list L has one element  return 0 and the
list L

Divide the list into two halves A and B
     (rA, A) ← Sort-and-Count(A)
     (rB, B) ← Sort-and-Count(B)
     (r , L) ← Merge-and-Count(A, B)

   return r = rA + rB + r and the sorted list L
}
```

Quick Sort

Quick Sort

```
Quicksort(A, p, r)
if p < r
    then q ← Partition(A, p, r)
         Quicksort(A, p, q − 1)
         Quicksort(A, q + 1, r)
```

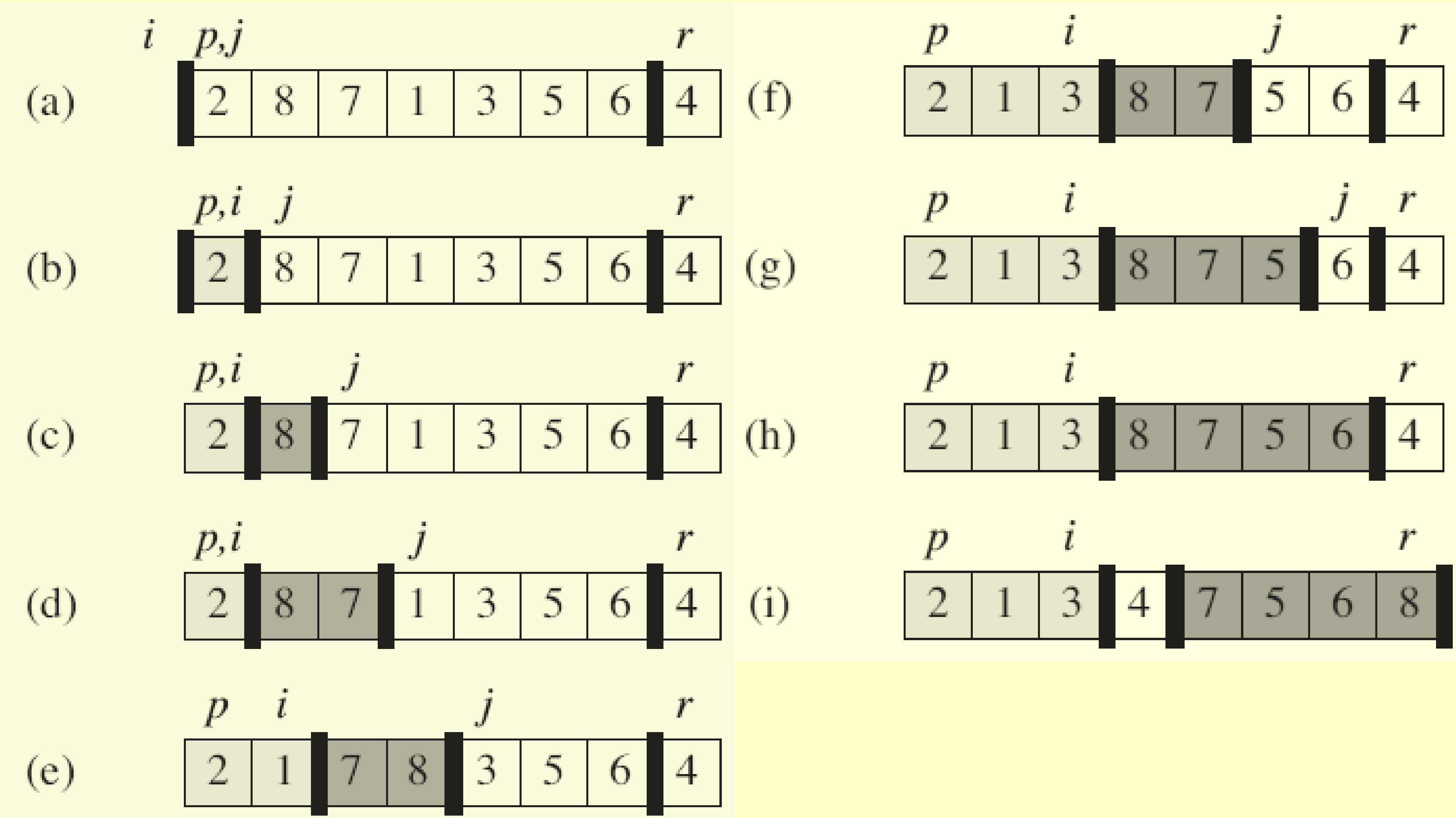
Initial call is Quicksort(A, 1, n)

Quick Sort

Partition(A, p, r)

```
x ← A[r]
i ← p − 1
for j ← p to r − 1
        do if A[j] ≤ x
                then i ← i + 1
                        exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

Quick Sort: Example



Partitioning

- Select the last element A[r] in the subarray $A[p..r]$ as the *pivot* – the element around which to partition.
- As the procedure executes, the array is partitioned into four (possibly empty) regions.
  1. $A[p..i]$ — All entries in this region are $\leq$ ***pivot***.
  2. $A[i+1..j-1]$ — All entries in this region are $>$ ***pivot***.
  3. $A[r] = pivot$.
  4. $A[j..r-1]$ — Not known how they compare to *pivot*.
- The above hold before each iteration of the *for* loop, and constitute a *loop invariant*. (4 is not part of the LI.)

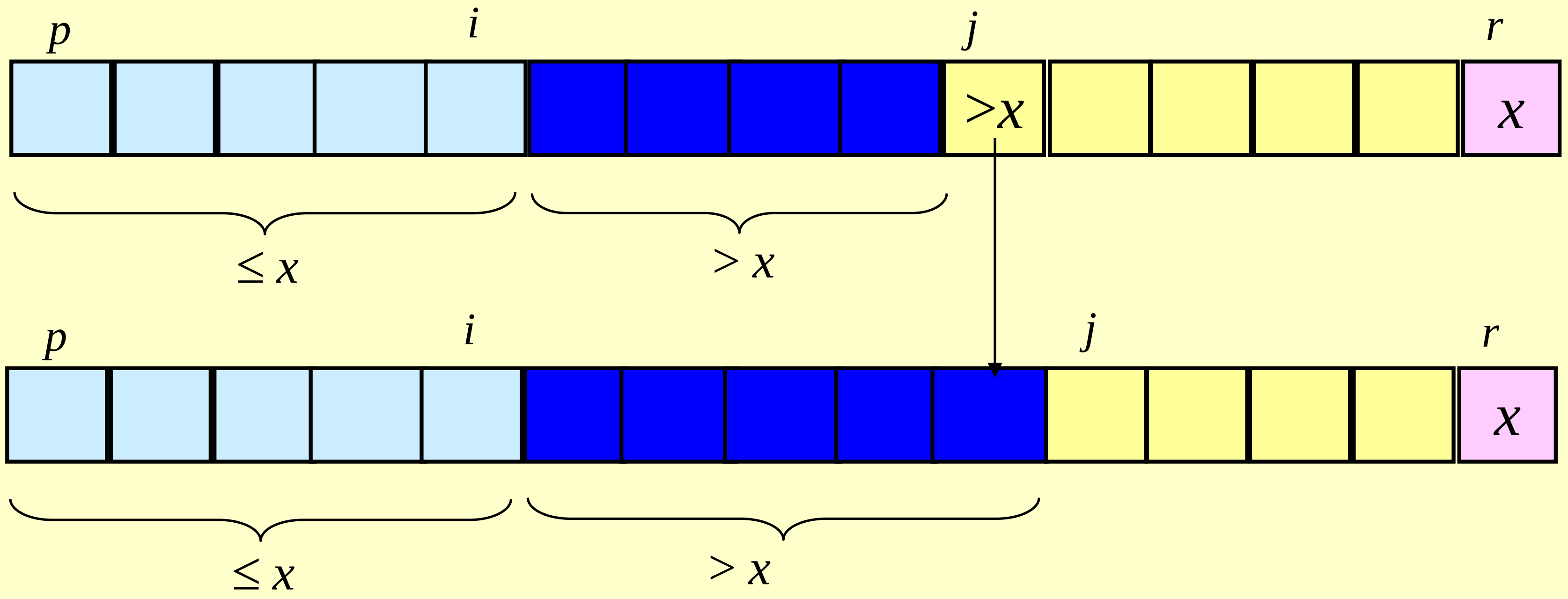
Correctness of Partition

- Use loop invariant.
- **Initialization:**
  - » Before first iteration
    - $A[p..i]$ and $A[i+1..j-1]$ are empty – Conds. 1 and 2 are satisfied (trivially).
    - r is the index of the *pivot* – Cond. 3 is satisfied.
- **Maintenance:**
  - » **Case 1:** $A[j] > x$
    - Increment j only.
    - LI is maintained.

```
Partition(A, p, r)
    x, i  := A[r], p – 1;
    for j := p to r – 1 do
        if A[j]  ≤  x then
            i := i + 1;
            A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```

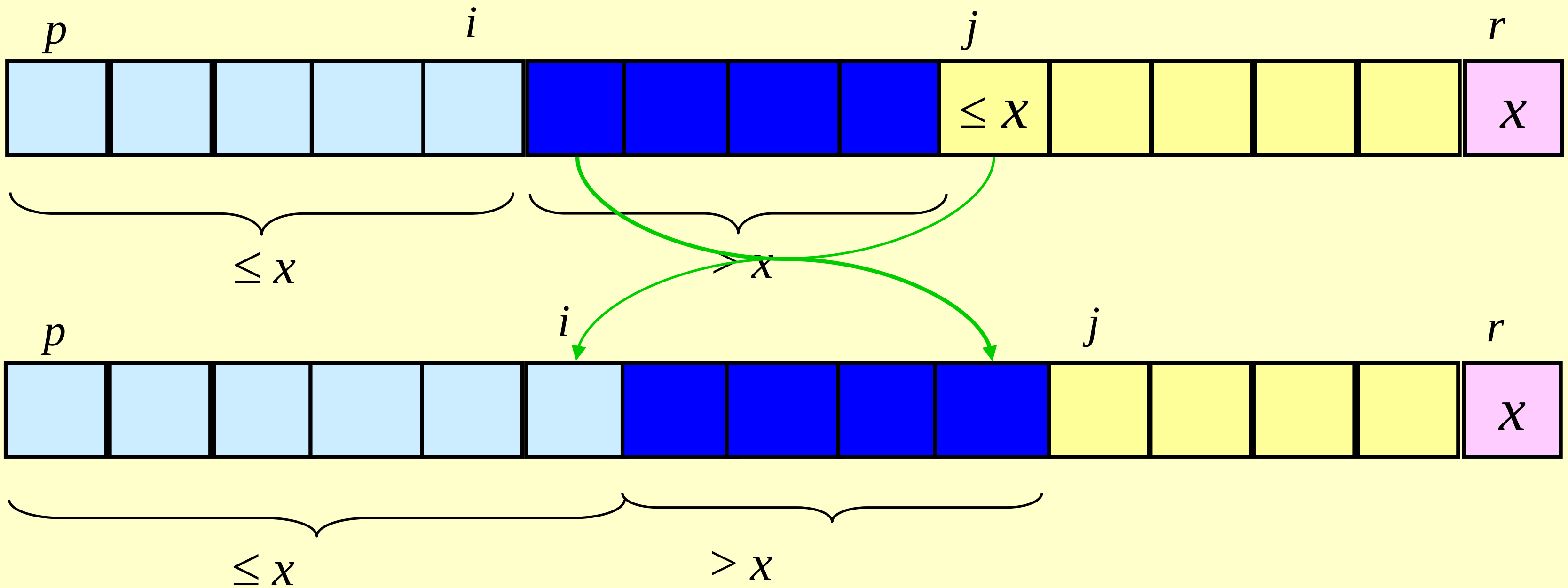
Correctness of Partition

Case 1:



Correctness of Partition

- **Case 2:** $A[j] \leq x$
  - » Increment i
  - » Swap $A[i]$ and $A[j]$
    - Condition 1 is maintained.
  - » Increment j
    - Condition 2 is maintained.
  - » $A[r]$ is unaltered.
    - Condition 3 is maintained.



Correctness of Partition

- **Termination:**
  - When the loop terminates, $j = r$, so all elements in A are partitioned into one of the three cases:
    - $A[p..i] \le$ ***pivot***
    - $A[i+1..j-1] >$ ***pivot***
    - $A[r] =$ ***pivot***
- The last two lines swap $A[i+1]$ and $A[r]$.
  - *Pivot* moves from the end of the array to between the two subarrays.
  - Thus, procedure *partition* correctly performs the divide step.

Time Complexity of Partition

- PartitionTime(n) is given by the number of iterations in the *for* loop.
- Time complexity of Partition algorithm is $\Theta(n)$.
- $\Theta(n)$: $n = r - p + 1$.

```
Partition(A, p, r)
    x, i  := A[r], p – 1;
    for j := p to r – 1 do
        if A[j]  ≤  x then
            i := i + 1;
            A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```

Algorithm Performance

Running time of quicksort depends on whether the partitioning is balanced or not.

- **Worst-Case Partitioning (Unbalanced Partitions):**
  - » Occurs when every call to partition results in the most unbalanced partition.
  - » Partition is most unbalanced when
    - Subproblem 1 is of size $n - 1$, and subproblem 2 is of size 0 or vice versa.
    - $pivot \geq$ every element in $A[p..r - 1]$ or $pivot <$ every element in $A[p..r - 1]$.
  - » Every call to partition is most unbalanced when
    - Array $A[1..n]$ is sorted or reverse sorted!

Worst-case Partition Analysis

Recursion tree for worst-case partition

$$
n \left\{
\begin{array}{c}
n \\
\downarrow \\
n-1 \\
\downarrow \\
n-2 \\
\downarrow \\
n-3 \\
\vdots \\
2 \\
\downarrow \\
1
\end{array}
\right.
$$

Running time for worst-case partitions at each recursive level:

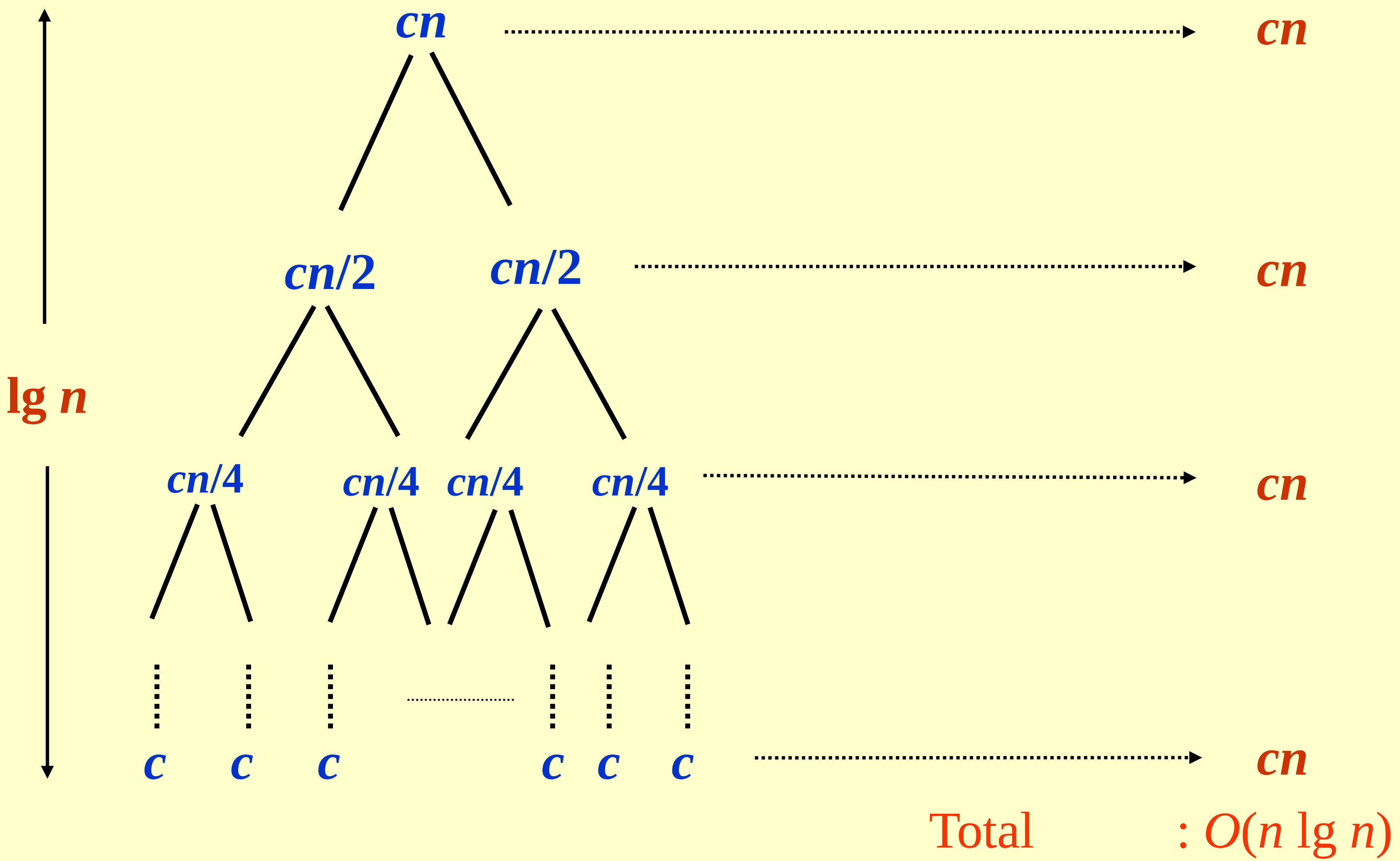
$$
\begin{aligned}
T(n) &= T(n-1) + T(0) + \text{PartitionTime}(n) \\
&= T(n-1) + \Theta(n) \\
&= \textstyle\sum_{k=1 \text{ to } n} \Theta(k) \\
&= \Theta(\textstyle\sum_{k=1 \text{ to } n} k\ ) \\
&= \Theta(n^2)
\end{aligned}
$$

Best-case Partitioning

- Size of each subproblem $\leq n/2$.
  - » One of the subproblems is of size $\lfloor n/2 \rfloor$
  - » The other is of size $\lceil n/2 \rceil - 1$.
- Recurrence for running time
  - » $T(n) \leq 2T(n/2) + \text{PartitionTime}(n)$

    $= 2T(n/2) + \Theta(n)$
- By using the case 2 of Master's method, the solution to the above recurrence is $\boldsymbol{T(n) = \Theta(n \lg n)}$

Recursion Tree for Best-case Partition



Fast Integer Multiplication

Integer Addition

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

Grade-school. $\Theta(n)$ 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. Traditional addition algorithm is optimal.

Integer Multiplication

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

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

Q. Is traditional multiplication algorithm optimal?

Divide-and-Conquer Multiplication: Warmup

To multiply two n-bit integers a and b:

- Multiply four $\frac{1}{2}n$-bit integers, recursively.
- Add and shift to obtain result.

$$
\begin{aligned}
a &= 2^{n/2} \cdot a_1 + a_0 \\
b &= 2^{n/2} \cdot b_1 + b_0 \\
ab &= \left(2^{n/2} \cdot a_1 + a_0\right)\left(2^{n/2} \cdot b_1 + b_0\right) = 2^n \cdot a_1 b_1 + 2^{n/2} \cdot \left(a_1 b_0 + a_0 b_1\right) + a_0 b_0
\end{aligned}
$$

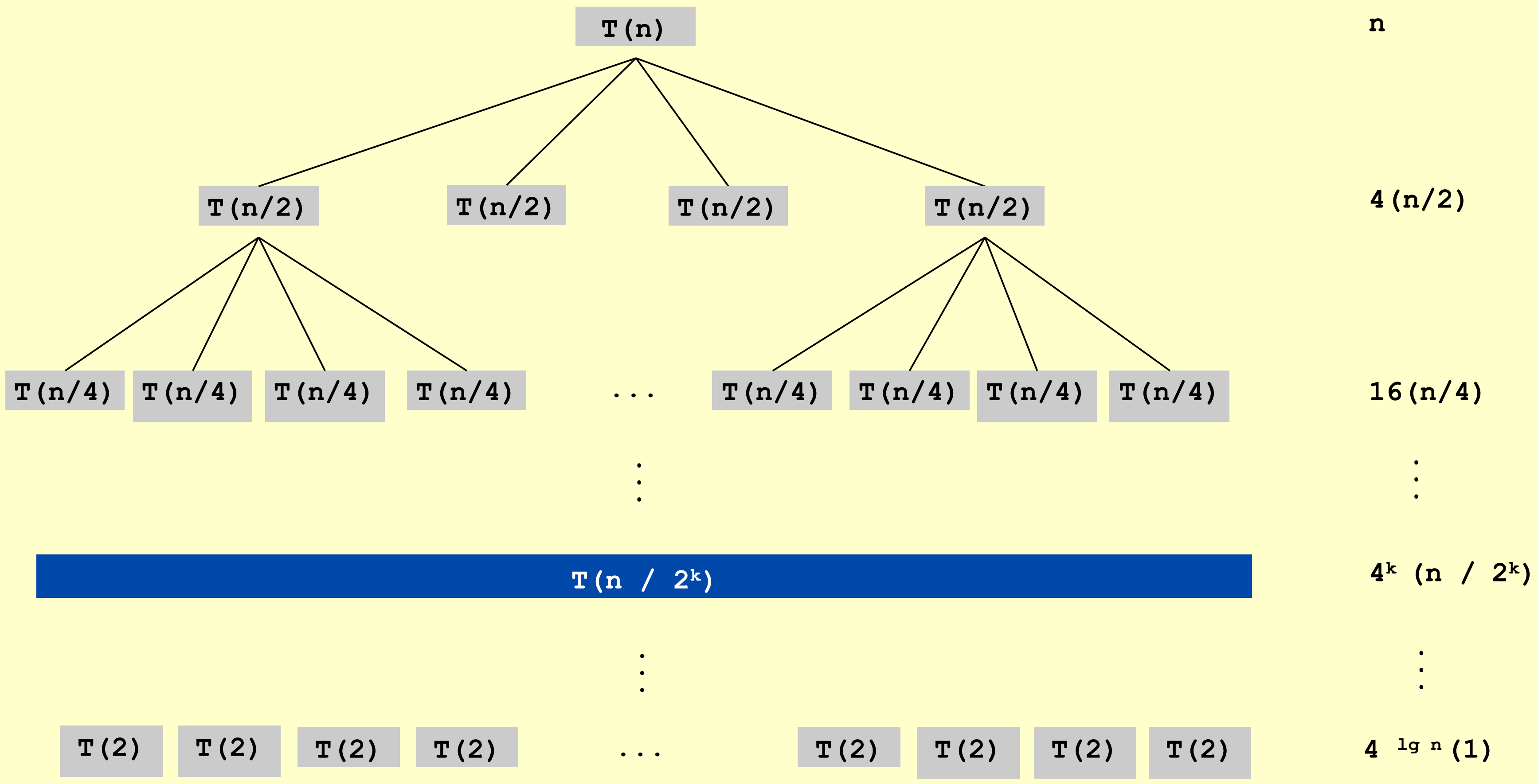
Ex. $a = \underbrace{1000}_{a_1}\underbrace{1101}_{a_0}$ $\quad b = \underbrace{1110}_{b_1}\underbrace{0001}_{b_0}$

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

Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 4T(n/2) + n & \text{otherwise} \end{cases}$$

$$T(n) = \sum_{k=0}^{\lg n} n\,2^k = n\left(\frac{2^{1+\lg n} - 1}{2 - 1}\right) = 2n^2 - n$$



Karatsuba Multiplication

To multiply two n-bit integers a and b:

- Add two $\frac{1}{2}n$ bit integers.
- Multiply three $\frac{1}{2}n$-bit integers, recursively.
- Add, subtract, and shift to obtain result.

$$
\begin{aligned}
a &= 2^{n/2} \cdot a_1 + a_0 \\
b &= 2^{n/2} \cdot b_1 + b_0 \\
ab &= 2^{n} \cdot a_1 b_1 + 2^{n/2} \cdot \left(a_1 b_0 + a_0 b_1\right) + a_0 b_0 \\
&= 2^{n} \cdot \underset{1}{a_1 b_1} + 2^{n/2} \cdot \left(\underset{2}{(a_1 + a_0)(b_1 + b_0)} - \underset{1}{a_1 b_1} - \underset{3}{a_0 b_0} \right) + \underset{3}{a_0 b_0}
\end{aligned}
$$

$$
T(n) \le \underbrace{T\left(\lfloor n/2 \rfloor\right) + T\left(\lceil n/2 \rceil\right) + T\left(1 + \lceil n/2 \rceil\right)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, subtract, shift}} \Rightarrow T(n) = O(n^{\lg 3}) = O(n^{1.585})
$$

Karatsuba Multiplication

```
Recursive-Multiply(x,y):
```

Write $x = x_1 \cdot 2^{n/2} + x_0$

$y = y_1 \cdot 2^{n/2} + y_0$

Compute $x_1 + x_0$ and $y_1 + y_0$

p = Recursive-Multiply($x_1 + x_0$, $y_1 + y_0$)

$x_1 y_1$ = Recursive-Multiply(x_1, y_1)

$x_0 y_0$ = Recursive-Multiply(x_0, y_0)

Return $x_1 y_1 \cdot 2^n + (p - x_1 y_1 - x_0 y_0) \cdot 2^{n/2} + x_0 y_0$

$$T(n) \le \underbrace{T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + T(1 + \lceil n/2 \rceil)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, subtract, shift}} \Rightarrow T(n) = O(n^{\lg 3}) = O(n^{1.585})$$

Karatsuba: Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 3T(n/2) + n & \text{otherwise} \end{cases}$$

$$T(n) = \sum_{k=0}^{\lg n} n \left(\tfrac{3}{2}\right)^k = n\left(\frac{\left(\frac{3}{2}\right)^{1+\lg n} - 1}{\frac{3}{2} - 1}\right) = 3n^{\lg 3} - 2n$$