DESIGN AND ANALYSIS OF ALGORITHMS (DAA)

- QUICK SORT
- MATRIX MULTIPLICATION
- CLOSEST PAIR OF POINTS

Course Instructor: Dr. Shreya Ghosh

COMPARISON SORTING

- Assumptions
  1. No knowledge of the keys or numbers we are sorting on.
  2. Each key supports a comparison interface or operator.
  3. Sorting entire records, as opposed to numbers, is an implementation detail.
  4. Each key is unique (just for convenience).

COMPARISON SORTING

- Given a set of *n* values, there can be *n!* permutations of these values.

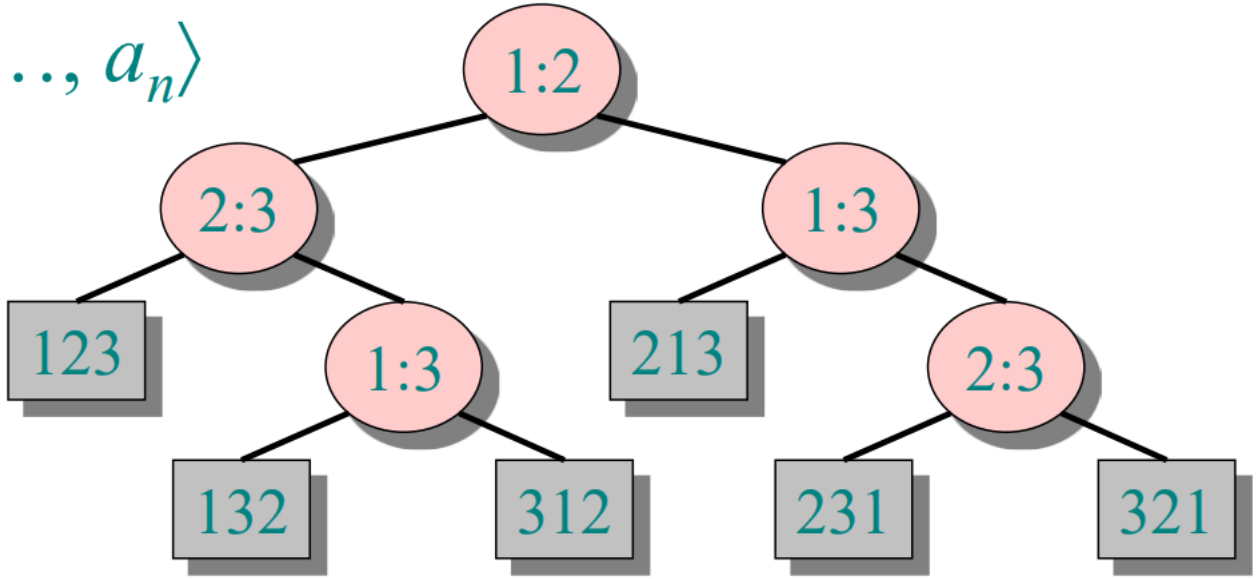
- So if we look at the behavior of the sorting algorithm over all possible *n!* inputs we can determine the worst-case complexity of the algorithm.

DECISION TREE MODEL

- Decision tree model
  - Full binary tree
    - A **proper binary tree** (or **2-tree**) is a tree in which every node other than the leaves has two children
  - Internal node represents a comparison.
    - Ignore control, movement, and all other operations, just see comparison
  - Each leaf represents one possible result (a permutation of the elements in sorted order).
  - The height of the tree (i.e., longest path) is the lower bound.

DECISION-TREE EXAMPLE

Sort $\langle a_1, a_2, \ldots, a_n \rangle$

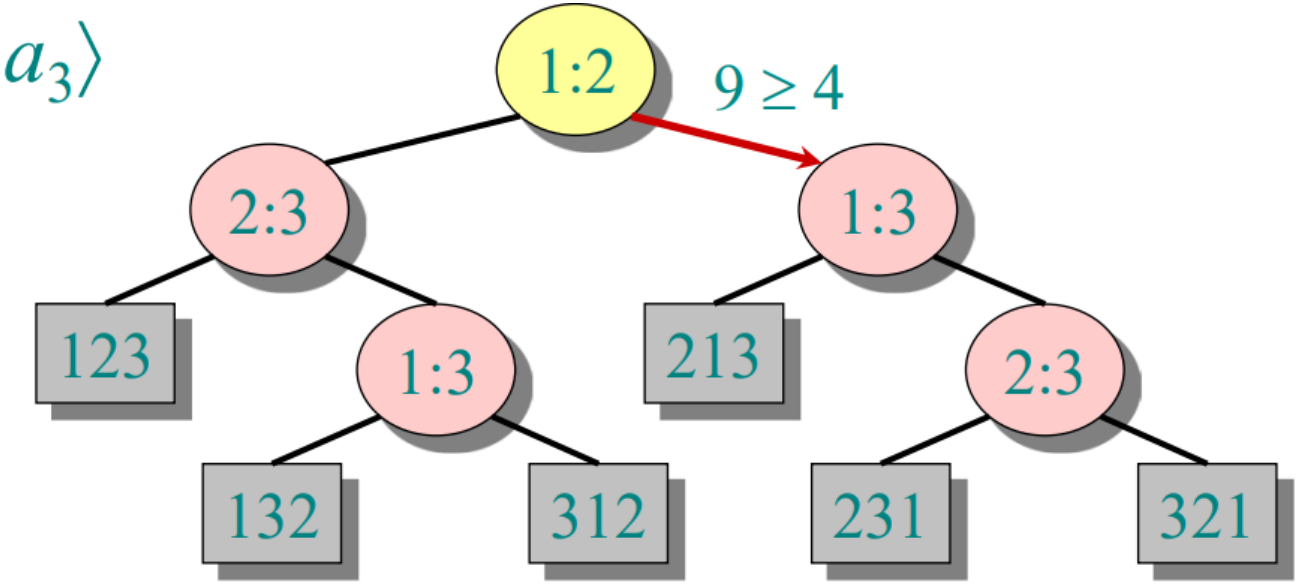


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.

DECISION-TREE EXAMPLE

Sort $\langle a_1, a_2, a_3 \rangle$ $= \langle 9, 4, 6 \rangle$:

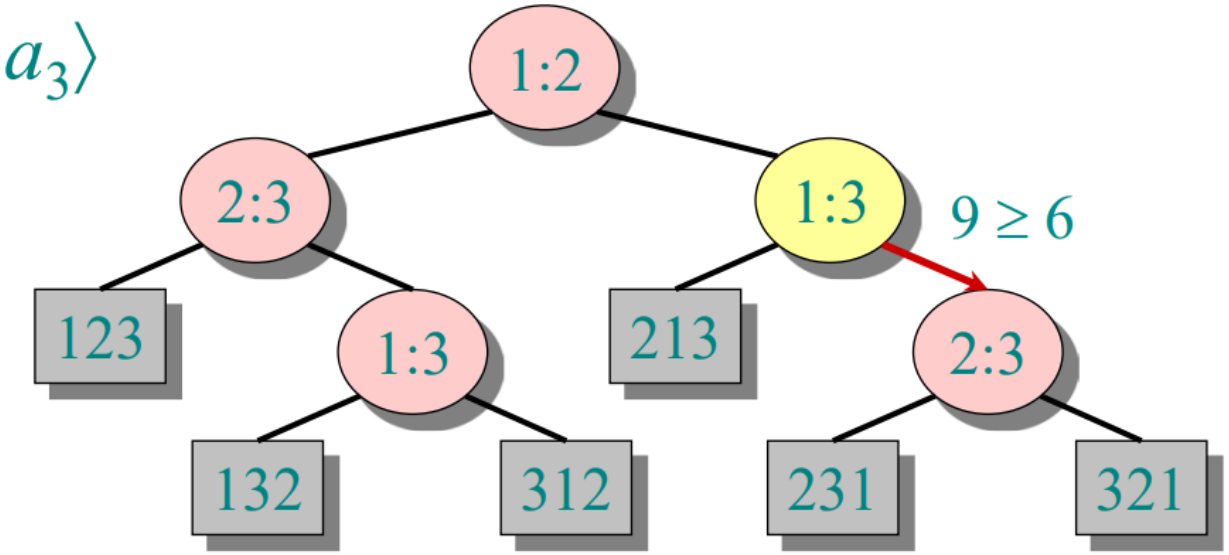


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.

DECISION-TREE EXAMPLE

Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$:

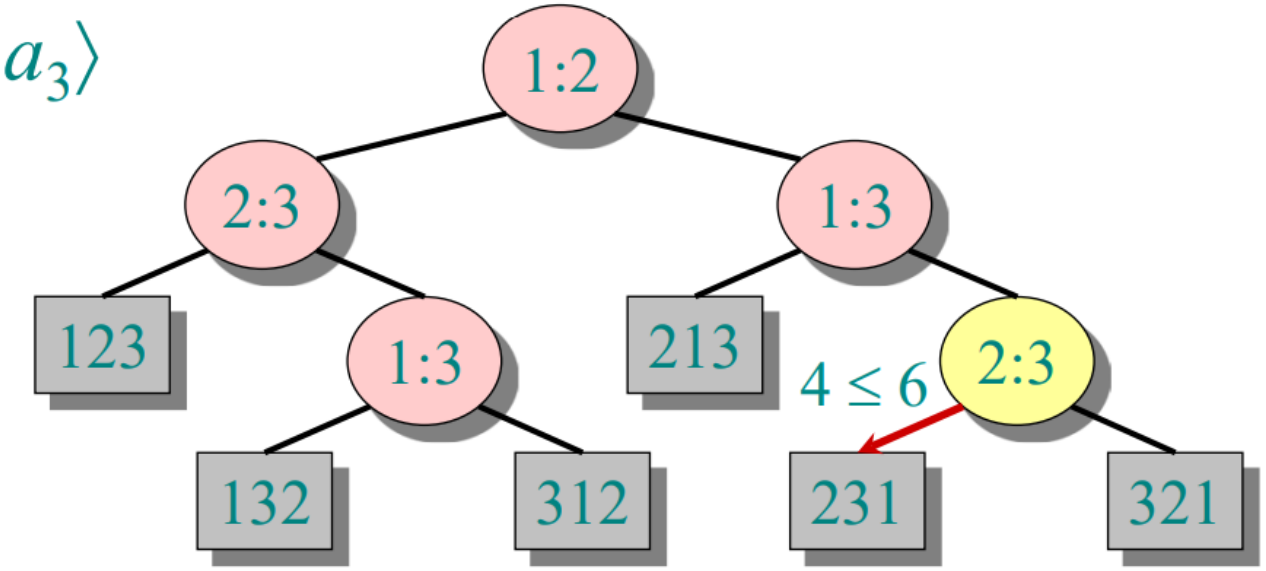


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.

DECISION-TREE EXAMPLE

Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$:

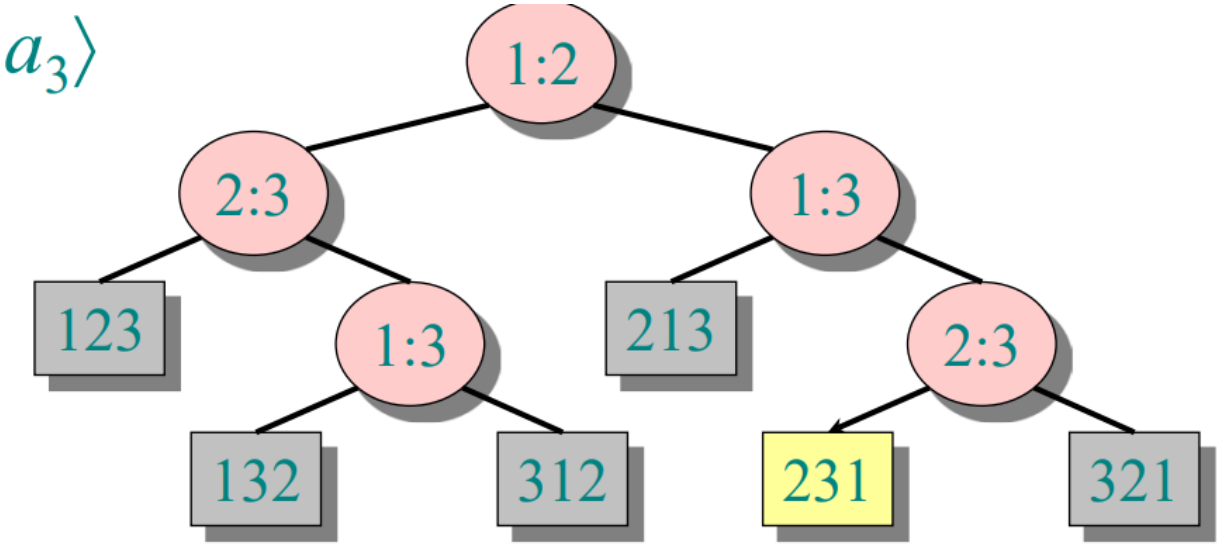


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i \geq a_j$.

DECISION-TREE EXAMPLE

Sort $\langle a_1, a_2, a_3 \rangle = \langle 9, 4, 6 \rangle$:



Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \Lambda \leq a_{\pi(n)}$ has been established.

LOWER BOUND FOR DECISION TREE SORTING

Theorem. Any decision tree that can sort n elements must have height $\Omega(n \lg n)$.

Proof. The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations. A height-h binary tree has $\leq 2^h$ leaves. Thus, $n! \leq 2^h$.

$$
\begin{aligned}
\therefore\ h &\geq \lg(n!) && (\lg \text{ is mono. increasing}) \\
&\geq \lg\left((n/e)^n\right) && (\text{Stirling's formula}) \\
&= n \lg n - n \lg e \\
&= \Omega(n \lg n).
\end{aligned}
$$

□

QUICK SORT

- **Fastest** known sorting algorithm in practice
- Average case: O(N log N)
- Worst case: $O(N^2)$
  - But the worst case can be made exponentially unlikely.
- Another divide-and-conquer recursive algorithm, like merge sort.

QUICK SORT: MAIN IDEA

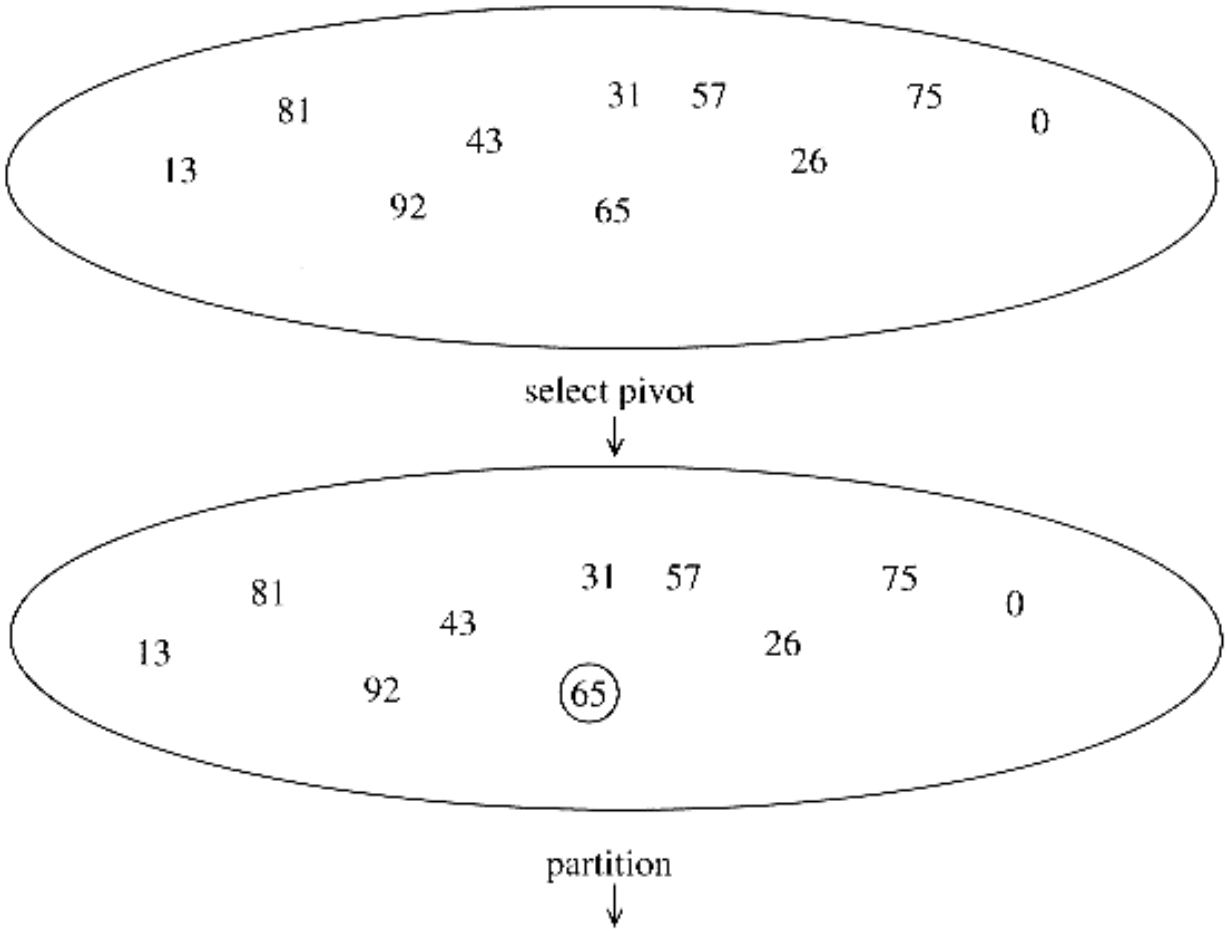
1. If the number of elements in S is 0 or 1, then return (base case).
2. Pick any element v in S (called the pivot).
3. Partition the elements in S except v into two disjoint groups:
    1. $S_1 = \{x \in S - \{v\} \mid x \leq v\}$
    2. $S_2 = \{x \in S - \{v\} \mid x \geq v\}$
4. Return $\{\text{QuickSort}(S_1) + v + \text{QuickSort}(S_2)\}$

- Follows the **divide-and-conquer** paradigm.

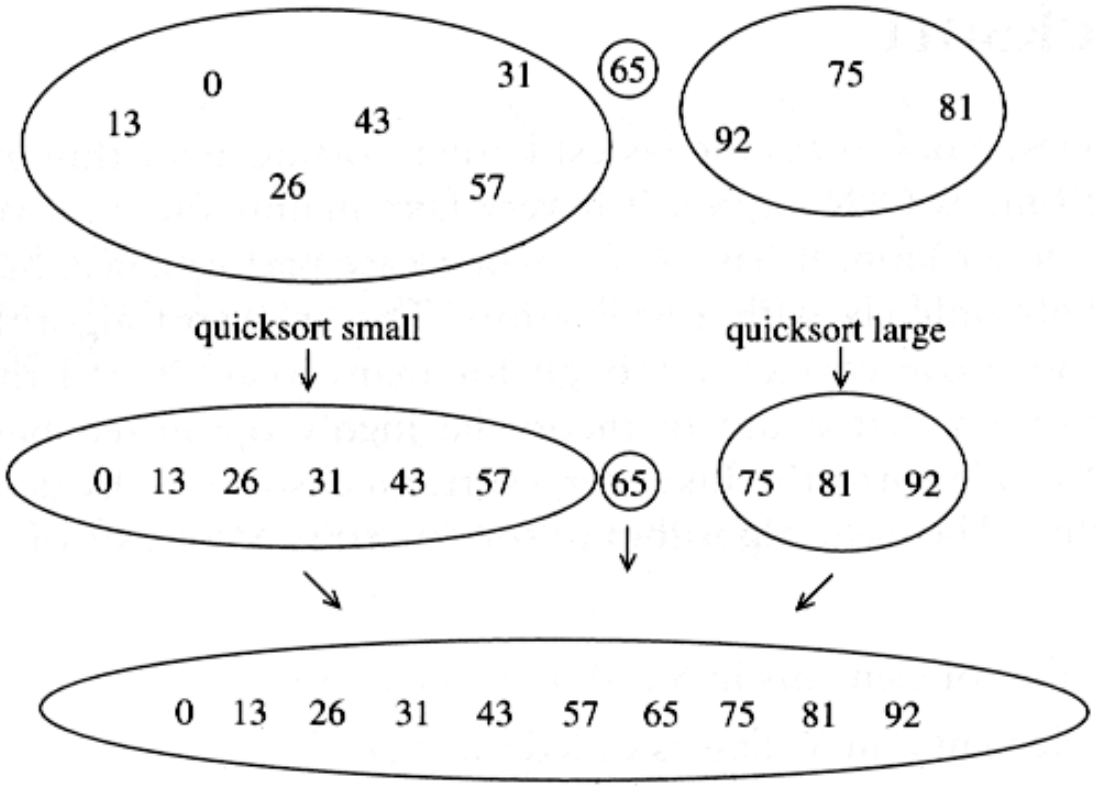
- ***Divide*:** Partition (separate) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$.
  - Each element in $A[p..q-1] < A[q]$.
  - $A[q] <$ each element in $A[q+1..r]$.
  - Index q is computed as part of the partitioning procedure.
- ***Conquer*:** Sort the two subarrays by recursive calls to quicksort.
- ***Combine*:** The subarrays are sorted in place – no work is needed to combine them.

- How do the divide and combine steps of quicksort compare with those of merge sort?

QUICK SORT: EXAMPLE



EXAMPLE OF 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)
```

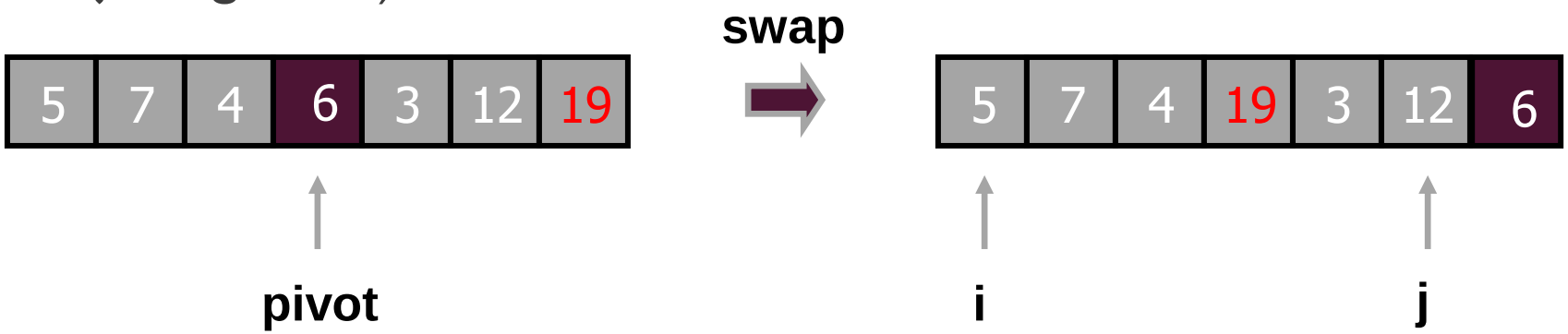
```
Partition(A, p, r)
    x, i  := A[r], p – 1;
    for j := p to r – 1 do
        if A[j]  ≤  x then
            i := i + 1;
            A[i] ↔ A[j]
    A[i + 1] ↔ A[r];
    return i + 1
```

ISSUES TO CONSIDER

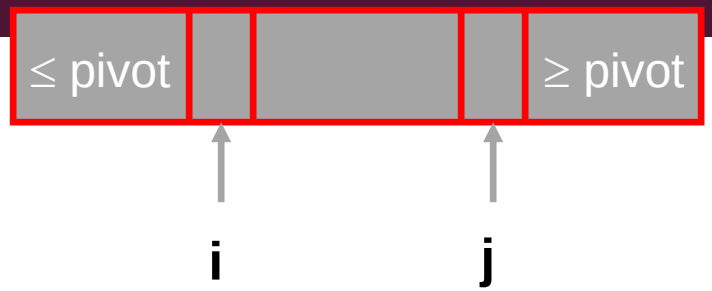
- How to pick the pivot?
  - Many methods (discussed later)
- How to partition?
  - Several methods exist.
  - The one we consider is known to give good results and to be easy and efficient.
  - We discuss the partition strategy first.

PARTITIONING STRATEGY

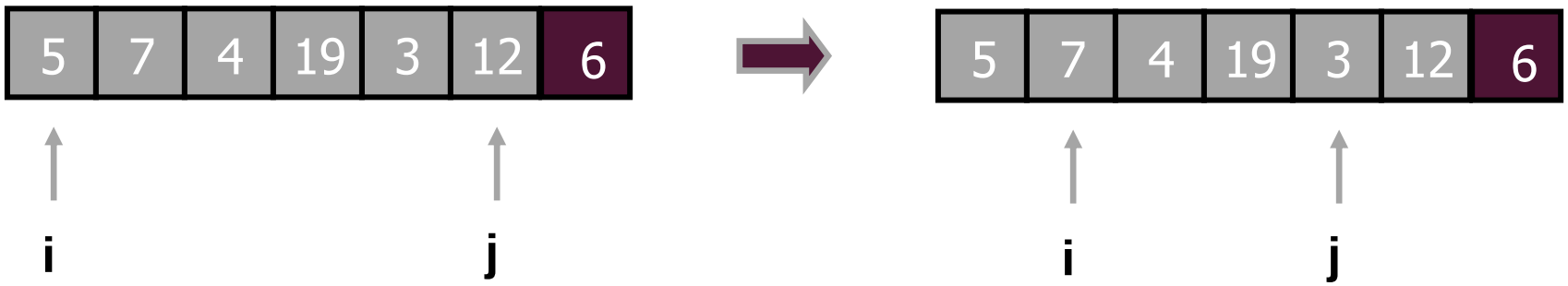
- For now, assume that pivot = A[(left+right)/2].
- We want to partition array A[left .. right].
- First, get the pivot element out of the way by swapping it with the last element (swap pivot and A[right]).
- Let i start at the first element and j start at the next-to-last element (i = left, j = right – 1)



PARTITIONING STRATEGY

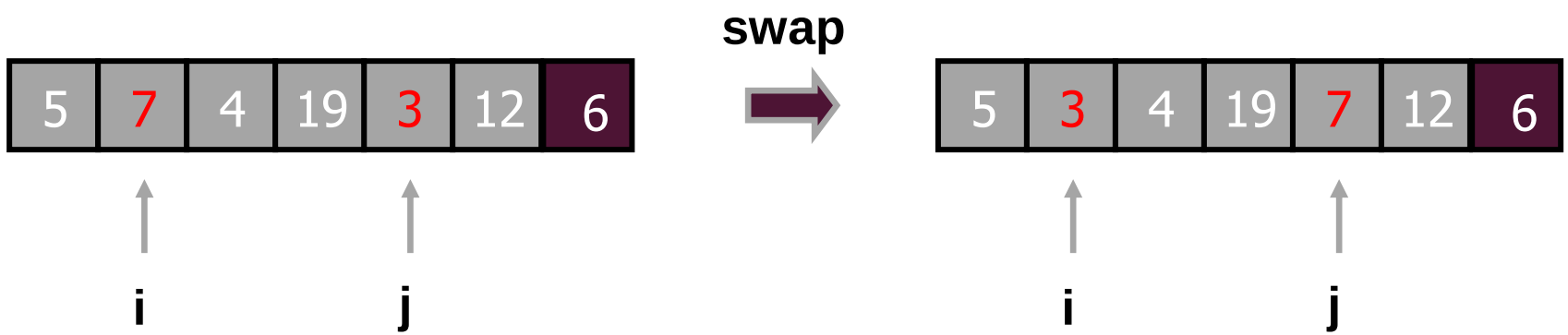


- Want to have
  - A[k] ≤ pivot, for k < i
  - A[k] ≥ pivot, for k > j
- When i < j
  - Move i right, skipping over elements smaller than the pivot
  - Move j left, skipping over elements greater than the pivot
  - When both i and j have stopped
    - A[i] ≥ pivot
    - A[j] ≤ pivot ⇒ A[i] and A[j] should now be swapped



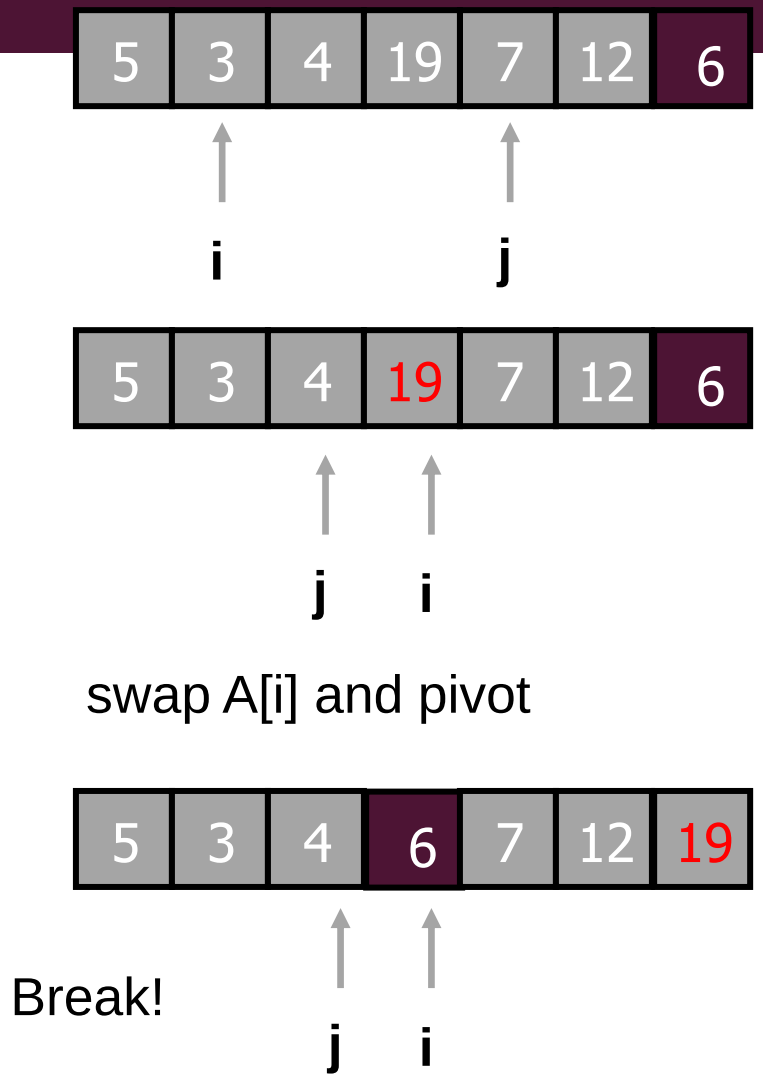
PARTITIONING STRATEGY (2)

- When i and j have stopped and i is to the left of j (thus legal)
  - Swap A[i] and A[j]
    - The large element is pushed to the right and the small element is pushed to the left
  - After swapping
    - A[i] ≤ pivot
    - A[j] ≥ pivot
  - Repeat the process until i and j cross



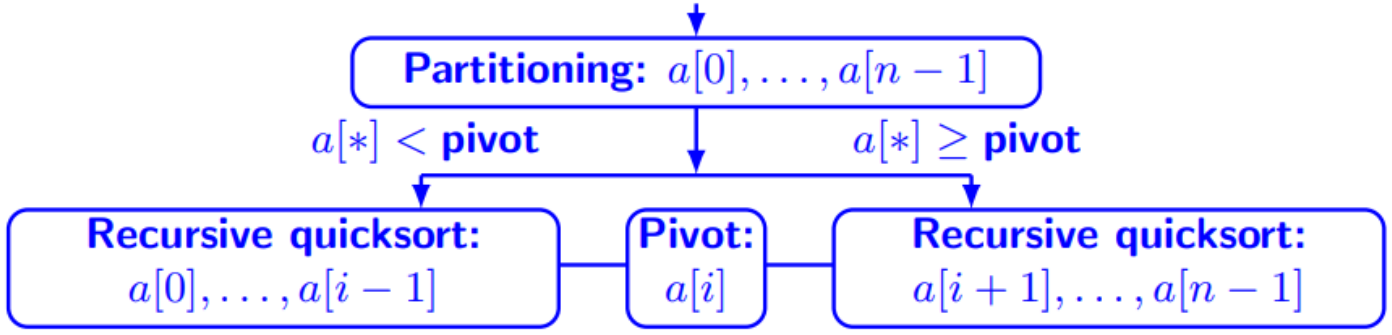
PARTITIONING STRATEGY (3)

- When i and j have crossed
  - swap A[i] and pivot
- Result:
  - A[k] ≤ pivot, for k < i
  - A[k] ≥ pivot, for k > i



If the size, n, of the list, is 0 or 1, return the list. Otherwise:

1. Choose one of the items in the list as a **pivot**.
2. Next, **partition** the remaining items into two disjoint sublists, such that all items greater than the pivot follow it, and all elements less than the pivot precede it.
3. Finally, return the result of quicksort of the "head" sublist, followed by the pivot, followed by the result of quicksort of the "tail" sublist.



BASIC RECURSIVE QUICKSORT

QUICKSORT IS CORRECT

Proof: by math induction on the size n of the list.

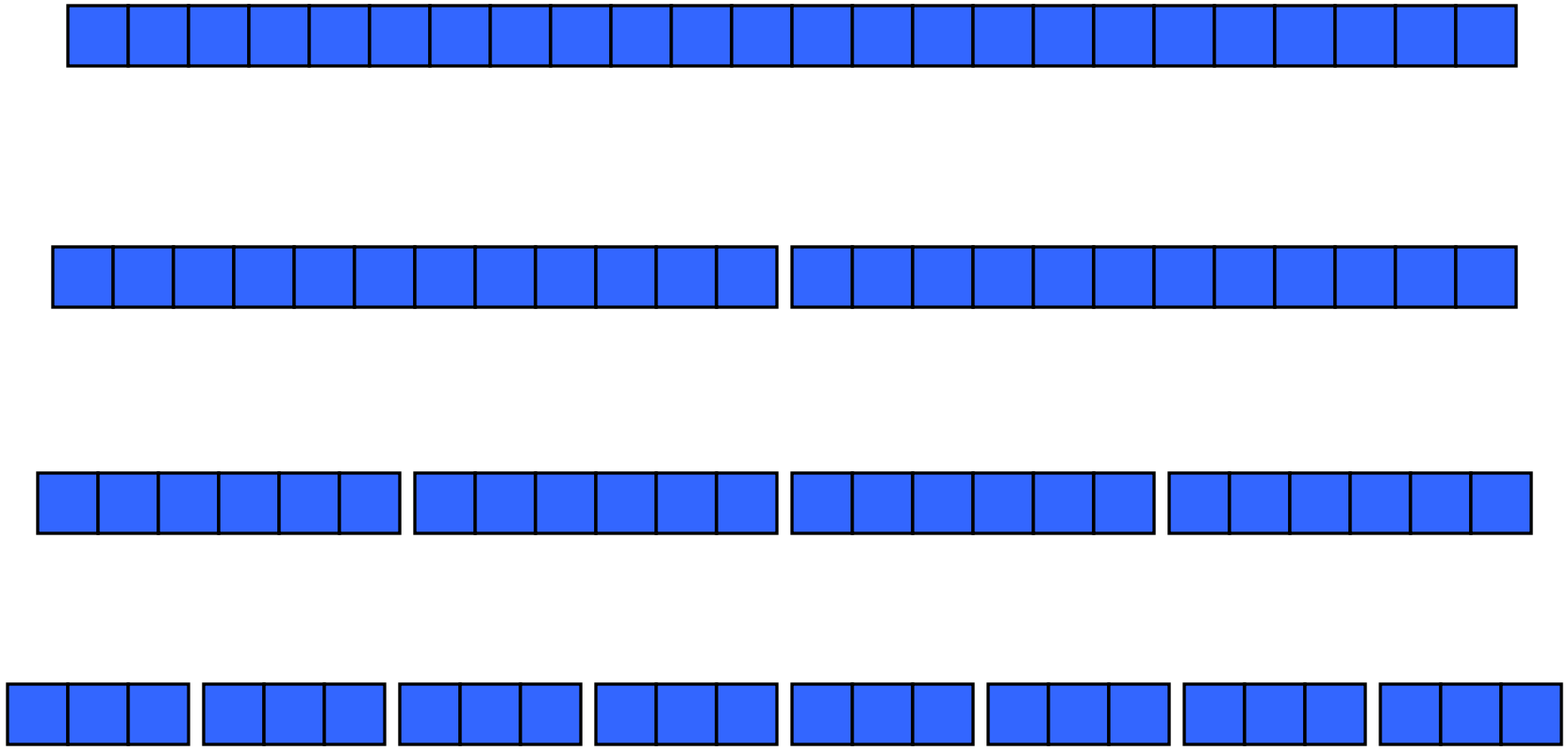
- **Basis.** If $n = 1$, the algorithm is correct.
- **Hypothesis.** It is correct on lists of size smaller than n.
- **Inductive step.** After positioning, the pivot p at position i; $i = 1, \ldots, n-1$, splits a list of size n into the head sublist of size i and the tail sublist of size $n - 1 - i$.
  - Elements of the head sublist are not greater than p.
  - Elements of the tail sublist are not smaller than p.
  - By the induction hypothesis, both the head and tail sublists are sorted correctly.
  - Therefore, the whole list of size n is sorted correctly.

Any implementation specifies what to do with items equal to the pivot.

ANALYSIS OF QUICKSORT—BEST CASE

- Suppose each partition operation divides the array almost exactly in half
- Then the depth of the recursion in $\log_2 n$
  - Because that's how many times we can halve n
- We note that
  - Each partition is linear over its subarray
  - All the partitions at one level cover the array

PARTITIONING AT VARIOUS LEVELS



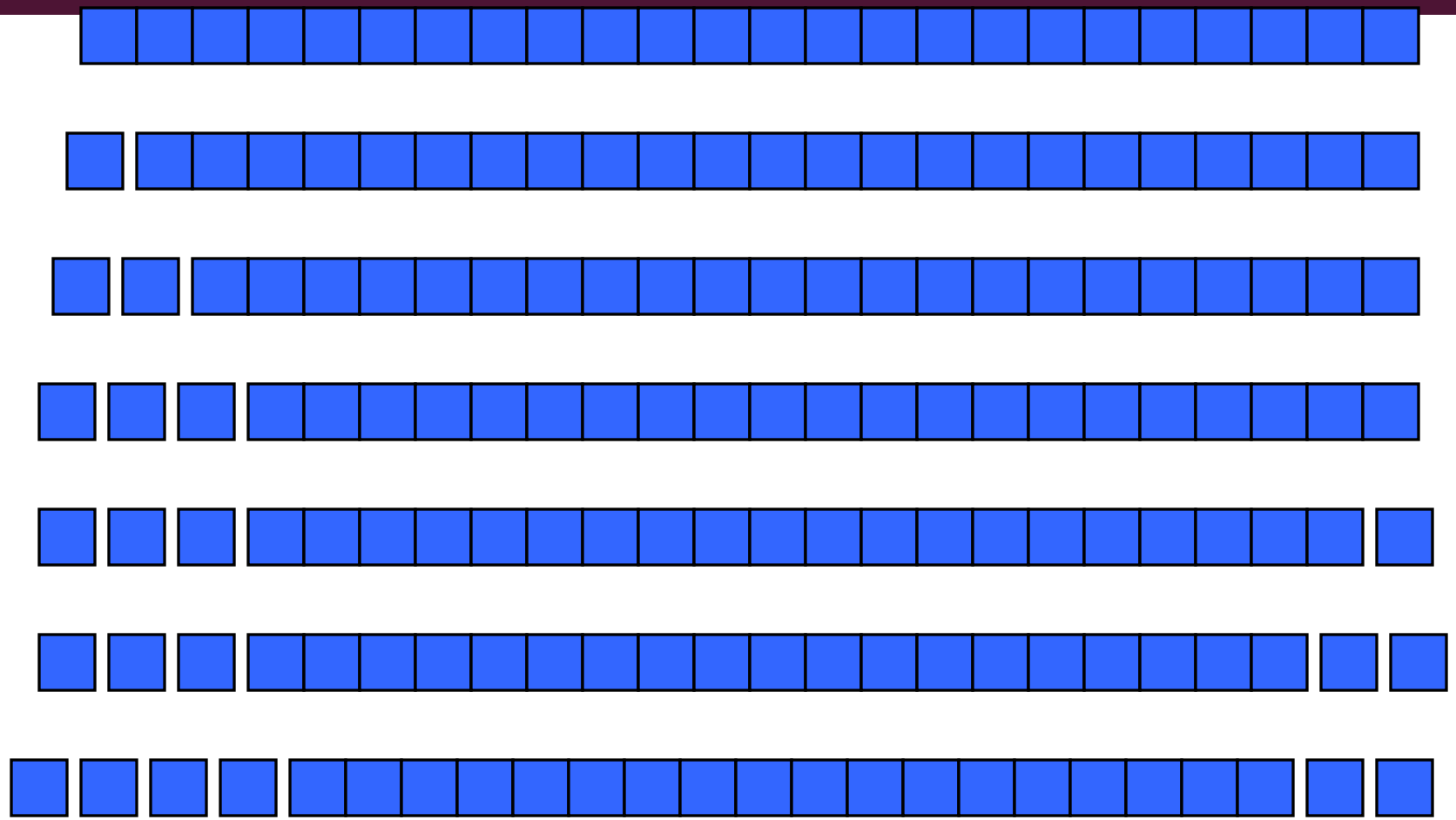
BEST CASE ANALYSIS

- We cut the array size in half each time
- So the depth of the recursion in $\log_2 n$
- At each level of the recursion, all the partitions at that level do work that is linear in n
- $O(\log_2 n) * O(n) = O(n \log_2 n)$
- Hence in the best case, quicksort has time complexity $O(n \log_2 n)$
- What about the worst case?

WORST CASE

- In the worst case, partitioning always divides the size `n` array into these three parts:
  - A length one part, containing the pivot itself
  - A length zero part, and
  - A length `n-1` part, containing everything else
- We don't recur on the zero-length part
- Recurring on the length `n-1` part requires (in the worst case) recurring to depth `n-1`

WORST CASE PARTITIONING



ANALYSIS (2)

- Running time
  - pivot selection: constant time, i.e. O(1)
  - partitioning: linear time, i.e. O(N)
  - running time of the two recursive calls

- $T(N) = T(i) + T(N - i - 1) + cN$
  - i: number of elements in S1
  - c is a constant

WORST-CASE SCENARIO

- What will be the worst case?
  - The pivot is the smallest element, all the time
  - Partition is always unbalanced

$$
\begin{aligned}
T(N) &= T(N-1) + cN \\
T(N-1) &= T(N-2) + c(N-1) \\
T(N-2) &= T(N-3) + c(N-2) \\
&\vdots \\
T(2) &= T(1) + c(2) \\
T(N) &= T(1) + c\sum_{i=2}^{N} i = O(N^2)
\end{aligned}
$$

BEST-CASE SCENARIO

- What will be the best case?
  - Partition is perfectly balanced.
  - Pivot is always in the middle (median of the array).

- $T(N) = T(N/2) + T(N/2) + cN = 2T(N/2) + cN$

- This recurrence is similar to the merge sort recurrence.
- The result is O(NlogN).

WORST CASE FOR QUICKSORT

- In the worst case, recursion may be n levels deep (for an array of size n)
- But the partitioning work done at each level is still n
- $O(n) * O(n) = O(n^2)$
- So worst case for Quicksort is $O(n^2)$
- When does this happen?
  - There are many arrangements that *could* make this happen
  - Here are two common cases:
    - When the array is already sorted
    - When the array is *inversely* sorted (sorted in the opposite order)

TYPICAL CASE FOR QUICKSORT

- If the array is sorted to begin with, Quicksort is terrible: $O(n^2)$
- It is possible to construct other bad cases
- However, Quicksort is *usually* $O(n \log_2 n)$
- The constants are so good that Quicksort is generally the faster algorithm.
- Most real-world sorting is done by Quicksort

PICKING A BETTER PIVOT

- Before, we picked the *first* element of the subarray to use as a pivot
  - If the array is already sorted, this results in $O(n^2)$ behavior
  - It's no better if we pick the *last* element
- We could do an *optimal* quicksort (guaranteed $O(n \log n)$) if we always picked a pivot value that exactly cuts the array in half
  - Such a value is called a **median**: half of the values in the array are larger, half are smaller
  - The easiest way to find the median is to *sort* the array and pick the value in the middle (!)

MEDIAN OF THREE

- Obviously, it doesn't make sense to sort the array in order to find the median to use as a pivot.
- Instead, compare just *three* elements of our (sub)array—the first, the last, and the middle
  - Take the *median* (middle value) of these three as the pivot
  - It's possible (but not easy) to construct cases which will make this technique $O(n^2)$

PICKING THE PIVOT (2)

- Use the first element as pivot
  - if the input is random, ok.
  - if the input is presorted (or in reverse order)
    - all the elements go into S_2 (or S_1).
    - this happens consistently throughout the recursive calls.
    - results in $O(N^2)$ behavior (we analyze this case later).
- Choose the pivot randomly
  - generally safe,
  - but random number generation can be expensive and does not reduce the running time of the algorithm.

PICKING THE PIVOT (3)

- Use the median of the array (ideal pivot)
  - The $\lceil N/2 \rceil$ *th* largest element
  - Partitioning always cuts the array into roughly half
  - An optimal quick sort (O(N log N))
  - However, hard to find the exact median

- Median-of-three partitioning
  - eliminates the bad case for sorted input.
  - reduces the number of comparisons by 14%.

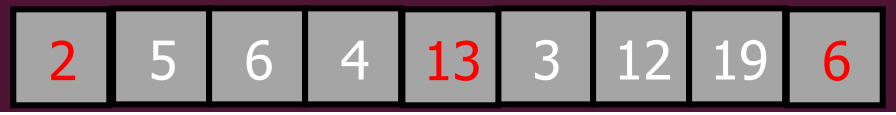
MEDIAN OF THREE METHOD

- Compare just three elements: the leftmost, rightmost and center
  - Swap these elements if necessary so that
    - A[left] = Smallest
    - A[right] = Largest
    - A[center] = Median of three
  - Pick A[center] as the pivot.
  - Swap A[center] and A[right – 1] so that the pivot is at the second last position

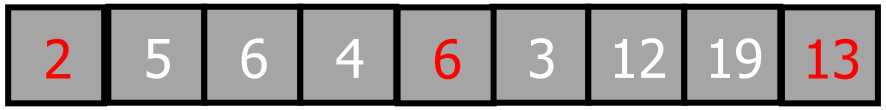
```
int center = ( left + right ) / 2;
if( a[ center ] < a[ left ] )
    swap( a[ left ], a[ center ] );
if( a[ right ] < a[ left ] )
    swap( a[ left ], a[ right ] );
if( a[ right ] < a[ center ] )
    swap( a[ center ], a[ right ] );

    // Place pivot at position right - 1
swap( a[ center ], a[ right - 1 ] );
```

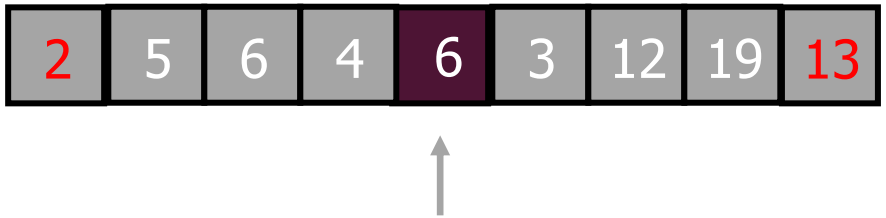
MEDIAN OF THREE: EXAMPLE



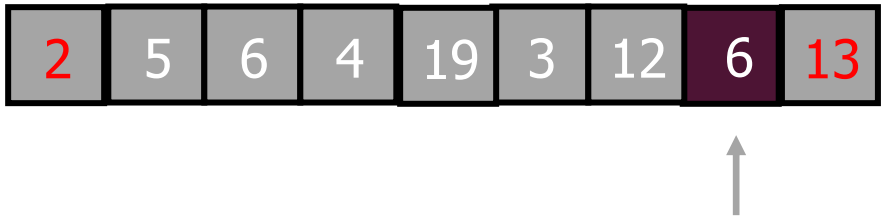
A[left] = 2, A[center] = 13,
A[right] = 6



Swap A[center] and A[right]



Choose A[center] as pivot



Swap pivot and A[right – 1]

We only need to partition A[left + 1, …, right – 2].

SMALL ARRAYS

- For very small arrays, quick sort does not perform as well as insertion sort
- Do not use quick sort recursively for small arrays
  - Use a sorting algorithm that is efficient for small arrays, such as insertion sort.
- When using quick sort recursively, switch to insertion sort when the sub-arrays have between 5 to 20 elements (10 is usually good).
  - saves about 15% in the running time.
  - avoids taking the median of three when the sub-array has only 1 or 2 elements.

QUICK SORT FASTER THAN MERGE SORT

- Both quick sort and merge sort take O(N log N) in the average case.
- But quick sort is faster in the average case:
  - The inner loop consists of an increment/decrement (by 1, which is fast), a test and a jump.
  - There is no extra juggling as in merge sort.

```
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```

inner loop

MULTIPLICATION OF TWO N BIT NUMBERS

Integer Addition

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

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

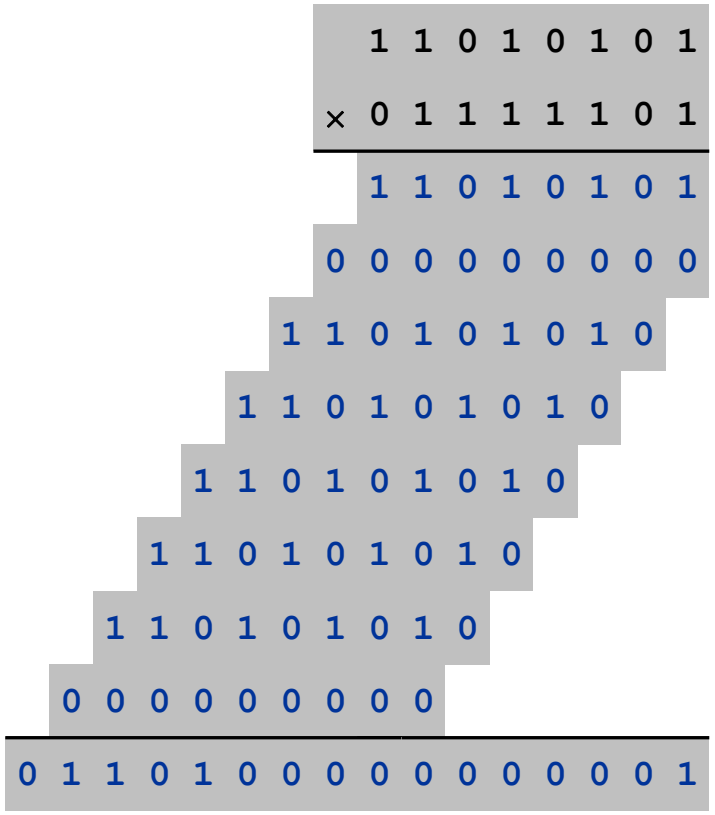
1	1	1	1	1	1	0	1	
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
1	0	1	0	1	0	0	1	0

Remark. Grade-school 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.



Q. Is grade-school 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.

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

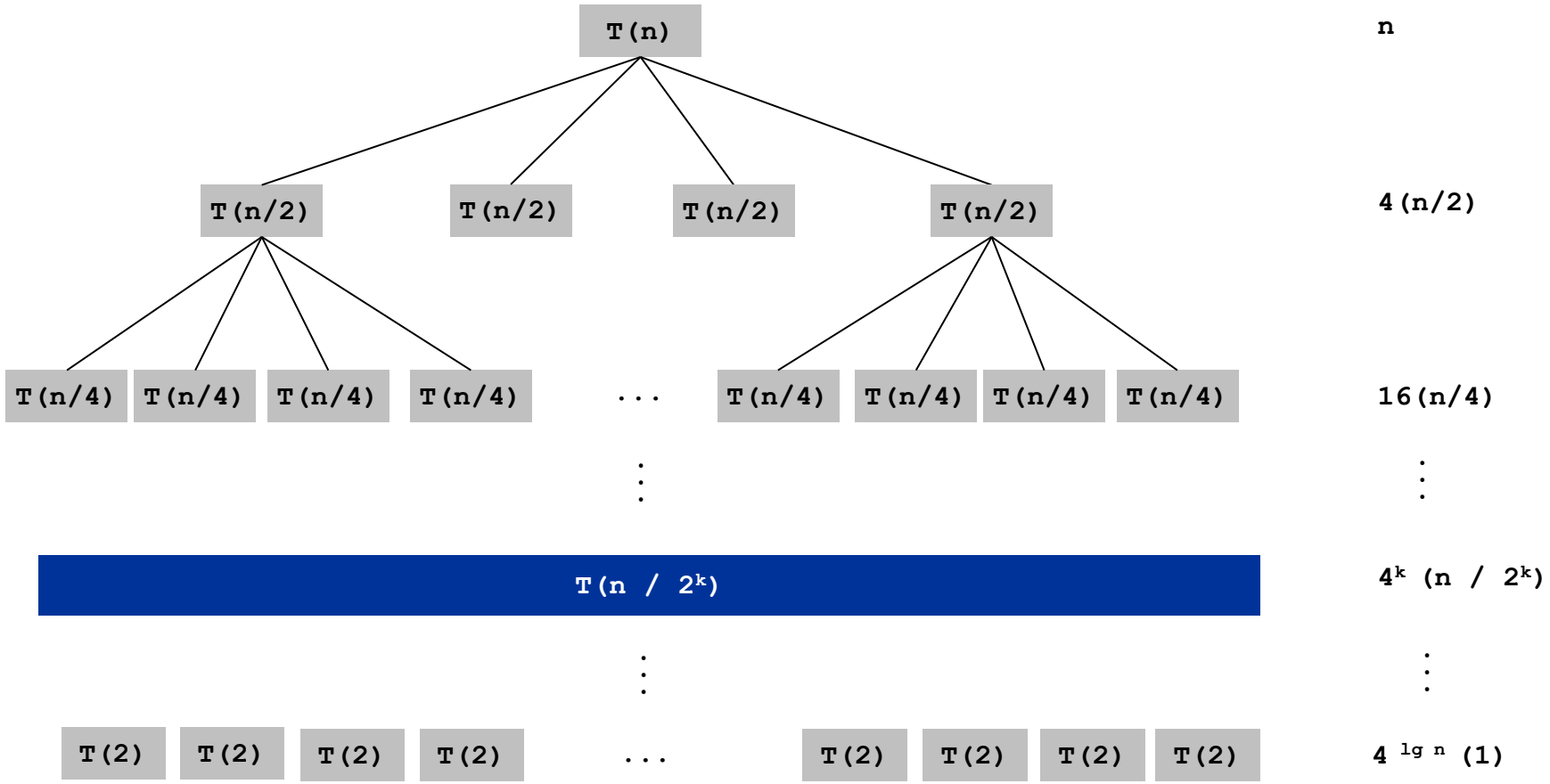
Ex. $\quad a = \underbrace{1000}_{a_1}\underbrace{1101}_{a_0} \qquad b = \underbrace{1110}_{b_1}\underbrace{0001}_{b_0}$

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

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.

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

1 2 1 3 3

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.

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

1 2 1 3 3

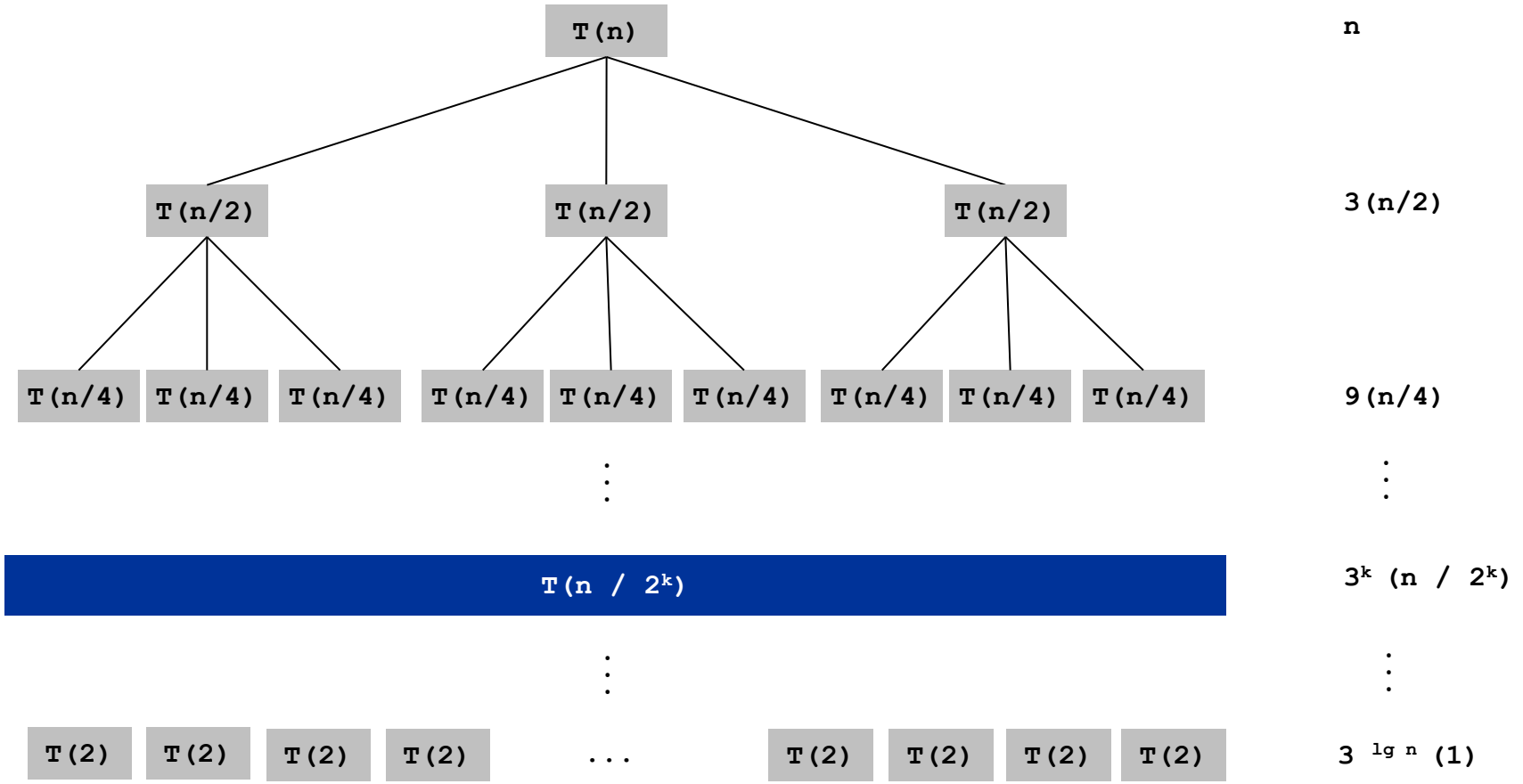
Theorem. [Karatsuba-Ofman 1962] Can multiply two n-bit integers in $O(n^{1.585})$ bit operations.

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

Karatsuba: Recursion Tree

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

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



Matrix Multiplication

Matrix Multiplication

Matrix multiplication. Given two n-by-n matrices A and B, compute $C = AB$.

Grade-school. $\Theta(n^3)$ arithmetic operations.

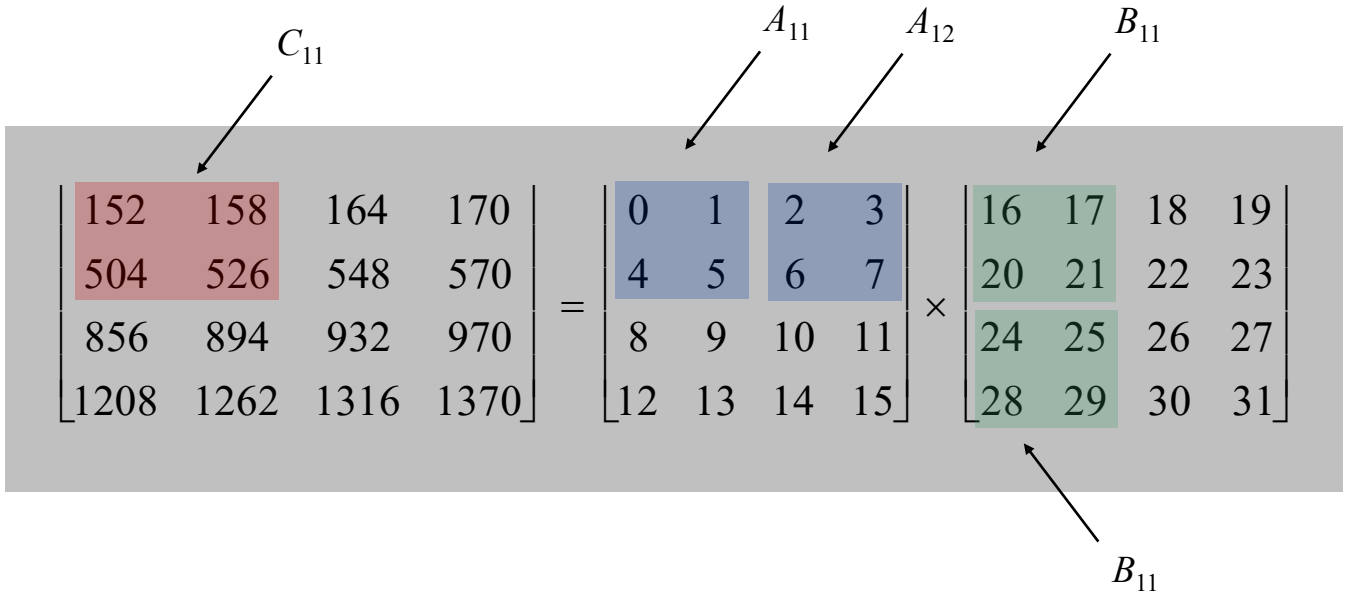
$$c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj}$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$\begin{bmatrix} .59 & .32 & .41 \\ .31 & .36 & .25 \\ .45 & .31 & .42 \end{bmatrix} = \begin{bmatrix} .70 & .20 & .10 \\ .30 & .60 & .10 \\ .50 & .10 & .40 \end{bmatrix} \times \begin{bmatrix} .80 & .30 & .50 \\ .10 & .40 & .10 \\ .10 & .30 & .40 \end{bmatrix}$$

Q. Is grade-school matrix multiplication algorithm optimal?

Block Matrix Multiplication



$$\begin{bmatrix} 152 & 158 & 164 & 170 \\ 504 & 526 & 548 & 570 \\ 856 & 894 & 932 & 970 \\ 1208 & 1262 & 1316 & 1370 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \times \begin{bmatrix} 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 \end{bmatrix}$$

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} = \begin{bmatrix} 0 & 1 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 16 & 17 \\ 20 & 21 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \times \begin{bmatrix} 24 & 25 \\ 28 & 29 \end{bmatrix} = \begin{bmatrix} 152 & 158 \\ 504 & 526 \end{bmatrix}$$

Strassen's Algorithm for Matrix Multiplication

Let $A = n \times n$ matrix, $B : n \times n$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$= \begin{bmatrix} ae \pm bg & af \pm bh \\ ce \pm dg & cf \pm dh \end{bmatrix}$$

$$T(n) = 8\,T\left(\frac{n}{2}\right) + O(n^2)$$

$$= O(n^3)$$

$$P_1 = a(f-h), \quad P_2 = (a+b)h$$

$$P_3 = (c+d)e, \quad P_4 = d(g-e)$$

$$P_5 = (a+d)(e+h) \quad P_6 = (b-d)(g+h)$$

$$P_7 = (a-c)(e+f)$$

$$\begin{bmatrix} P_5+P_4-P_2+P_6 & P_1+P_2 \\ P_3+P_4 & P_1+P_5-P_3-P_7 \end{bmatrix}$$

$$T(n) = \boxed{7}\,T\left(\frac{n}{\boxed{2}}\right) + O(n^2)$$

$$O(n^{\log_2 7}) = O(n^{2.81})$$

Matrix Multiplication: Warmup

To multiply two n-by-n matrices A and B:

- Divide: partition A and B into $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.
- Conquer: multiply 8 pairs of $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices, recursively.
- Combine: add appropriate products using 4 matrix additions.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

$$T(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}} \Rightarrow T(n) = \Theta(n^3)$$

Fast Matrix Multiplication

Key idea. multiply 2-by-2 blocks with only 7 multiplications.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

$$\begin{aligned} P_1 &= A_{11} \times (B_{12} - B_{22}) \\ P_2 &= (A_{11} + A_{12}) \times B_{22} \\ P_3 &= (A_{21} + A_{22}) \times B_{11} \\ P_4 &= A_{22} \times (B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_6 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ P_7 &= (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{aligned}$$

- 7 multiplications.
- $18 = 8 + 10$ additions and subtractions.

Fast Matrix Multiplication

To multiply two n-by-n matrices A and B: [Strassen 1969]

- Divide: partition A and B into $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.
- Compute: 14 $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices via 10 matrix additions.
- Conquer: multiply 7 pairs of $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices, recursively.
- Combine: 7 products into 4 terms using 8 matrix additions.

Analysis.

- Assume n is a power of 2.
- $T(n)$ = # arithmetic operations.

$$T(n) = \underbrace{7T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, subtract}} \Rightarrow T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

CLOSEST PAIR OF POINTS

CLOSEST POINTS

- ***A famous algorithmic problem...***

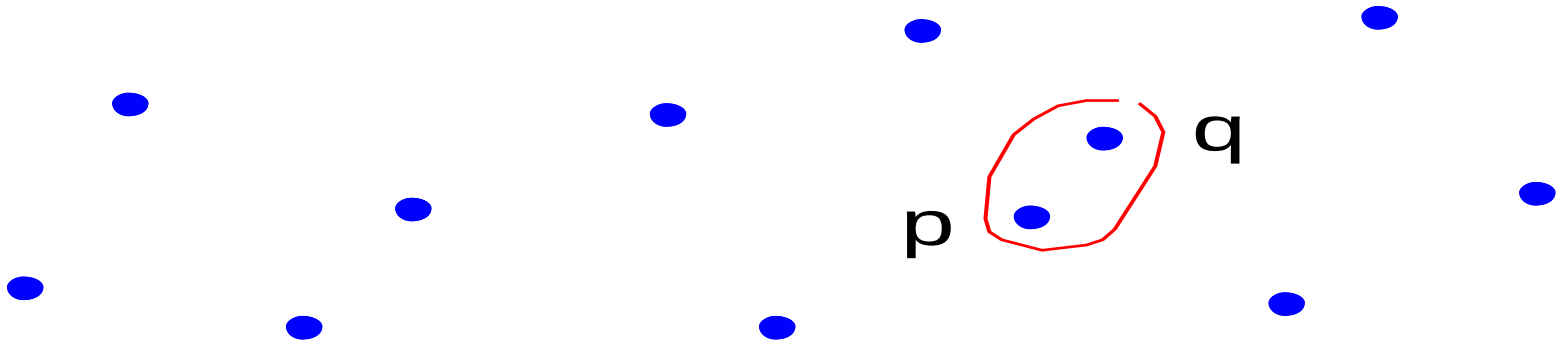
  Given a set of points in the plane (cities in India., transistors on a circuit board, computers on a network, etc.) find which pair of points is the closest together.

- Today we will study algorithms and data structures for solving this problem.

CLOSEST PAIR

FORMAL DEFINITION

Given a set P of N points, find $p,q \in P$ such that the distance d(p,q) is minimum.



- Algorithms for determining the closest pair:

  -brute force $O(N^2)$

  -divide-and-conquer O(N log N)

  -plane-sweep O(N log N)

Compute all the distances d(p,q) and select the minimum distance.

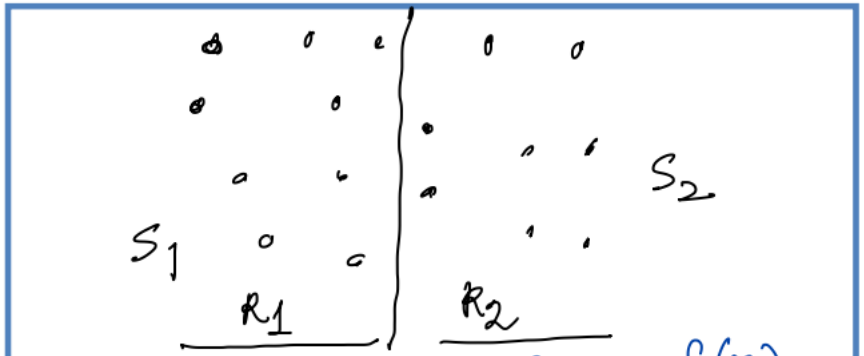
$$d(p1, p2) = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

Time Complexity: O(N^2)

DIVIDE AND CONQUER

```
closestpair (S)
{ Let S = {⟨x1, y1⟩, ⟨x2, y2⟩, ..., ⟨xn, yn⟩}
  split S into 2 disjoint non-empty subsets S1 & S2        O(1)
  R1 = closestpair (S1) }                                  O(n)
  R2 = closestpair (S2) }
  Let R = min (R1, R2)
  R3 = combine (S1, S2, R)                                 f(n)
  return (R3)                                              O(n)
}
```

equal



$T(n) = T(n_1) + T(n_2) + f(n)$

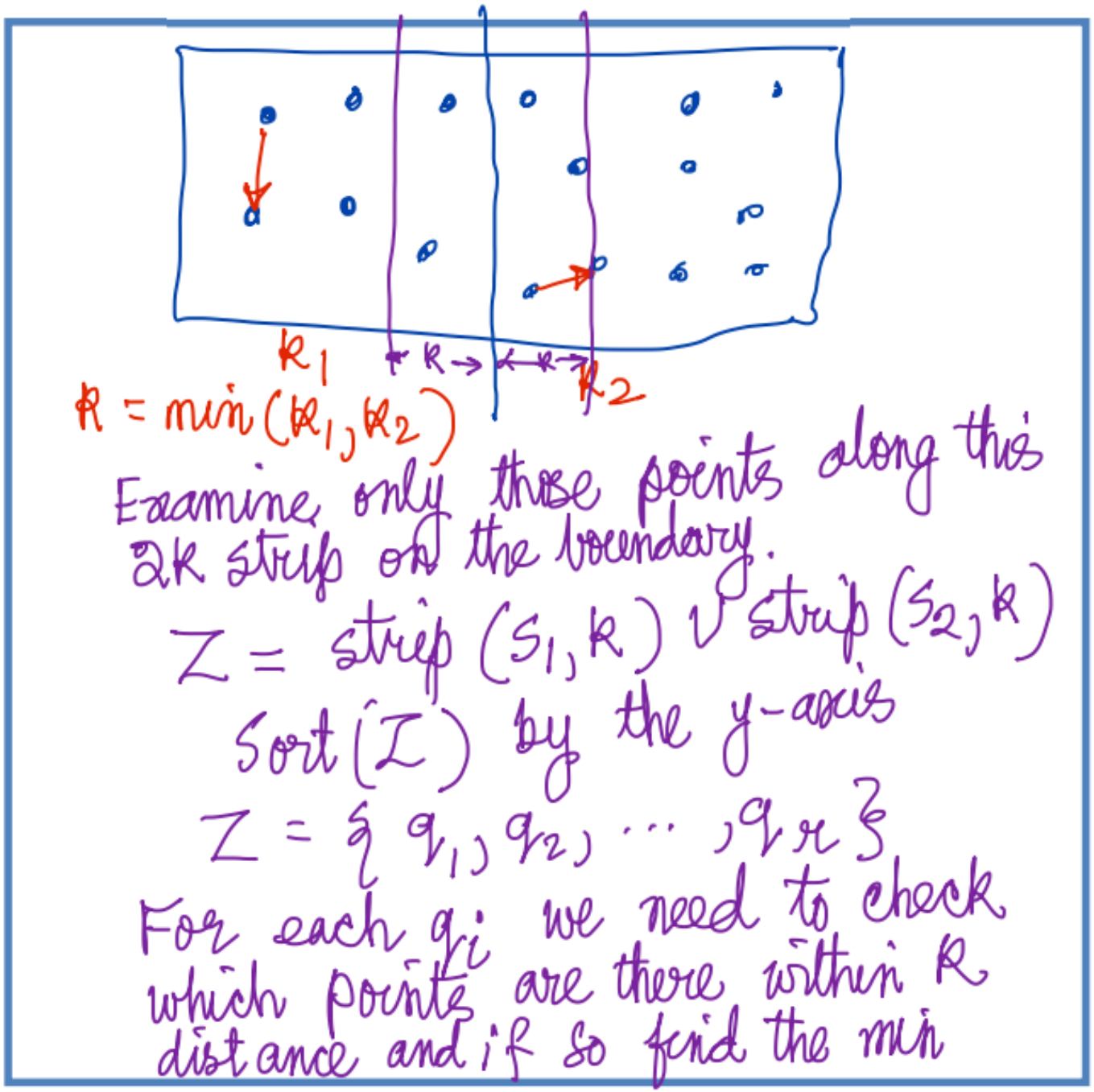
$= 2T(n/2) + f(n)$ ←

1. Divide into 2 equal parts
2. Divide randomly [check for av. case]

→ Median along x-axis

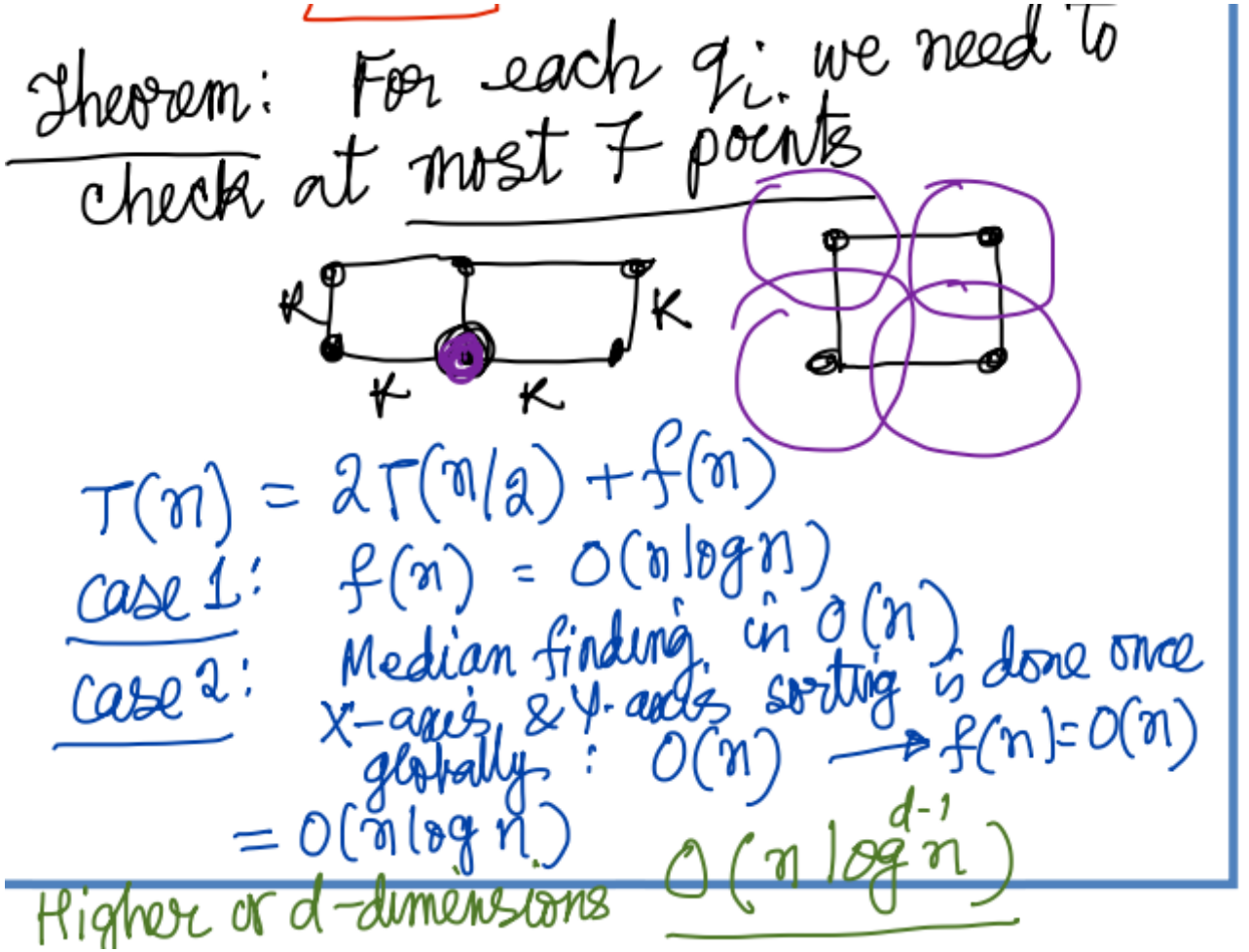
→ sorting $O(n \log n)$

→ Median Finding $O(n)$



CLOSEST PAIR OF POINTS: STRIP COMBINE

DOES THE COMPARISON REQUIRE EVERY POINTS IN Z TO BE COMPARED WITH MANY OR O(N) OTHER POINTS IN Z? **NO**



δ_L: minimum distance of P_L
δ_R: minimum distance of P_R
δ=min{δ_L, δ_R}.

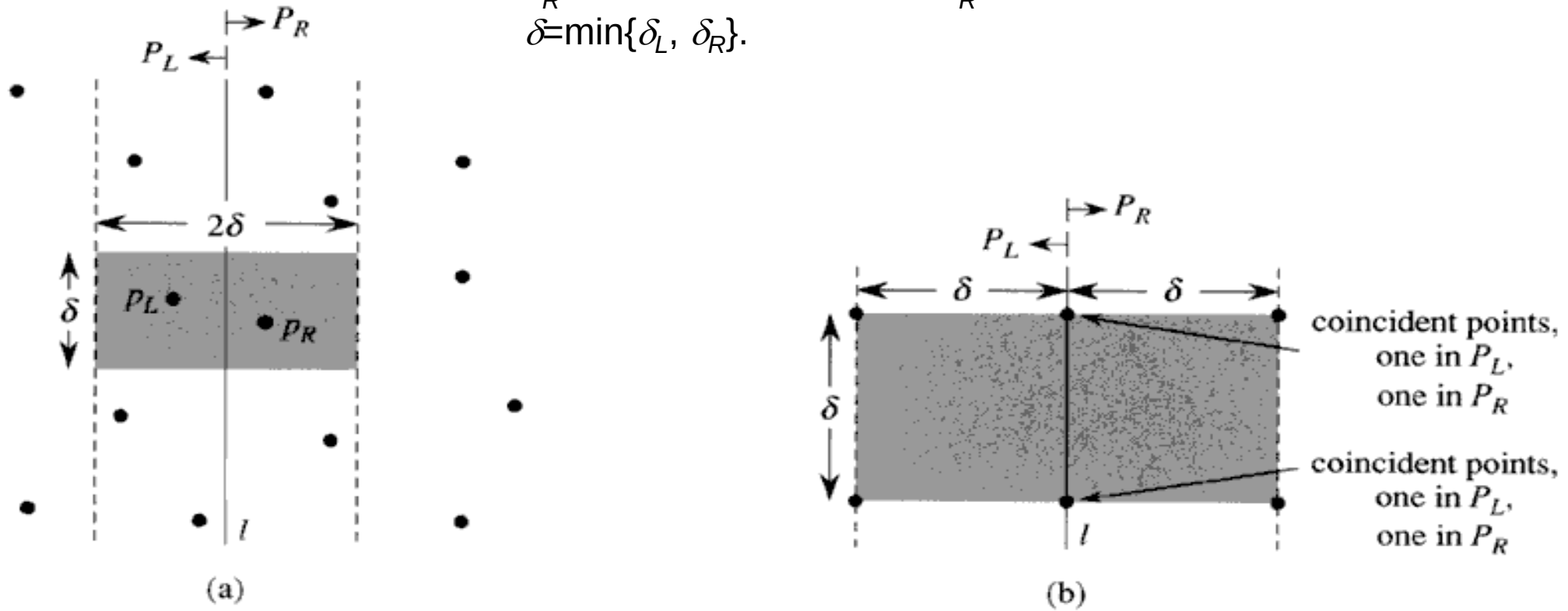


Figure 33.11 Key concepts in the proof that the closest-pair algorithm needs to check only 7 points following each point in the array Y'. **(a)** If $p_L \in P_L$ and $p_R \in P_R$ are less than δ units apart, they must reside within a $\delta \times 2\delta$ rectangle centered at line l. **(b)** How 4 points that are pairwise at least δ units apart can all reside within a $\delta \times \delta$ square. On the left are 4 points in P_L, and on the right are 4 points in P_R. There can be 8 points in the $\delta \times 2\delta$ rectangle if the points shown on line l are actually pairs of coincident points with one point in P_L and one in P_R.

- Divide : into two subsets (according to x-coordinate) : $P_L <= l <= P_R$ (O(n))
- Conquer: recursively on each half.
  - Get δ_L, δ_R
  - 2T(n/2).
- Combine:
  - select closer pair of the above. $\delta=\min\{\delta_L, \delta_R\}$, O(1)
  - Find the smaller pairs, one$\in P_L$ and the other$\in P_R$
    - Creat an array Y' of points within 2δ vertical strip, sorted by y-coor. O($n$lg$n$) or O(n).
    - for each point in Y', compare it with its following 7 points. O(7n).
      - *If a new smaller δ'* is found, then it is new δ and the new pair is found.

- P: set of points, X: sorted by x-coordinate, Y: sorted by y-coordinate
- Divide P into P_L and P_R, X into X_L and X_R, Y into Y_L and Y_R,
  - δ_1=CLOSET-PAIR(P_L,X_L,Y_L) and //T(n/2)
  - δ_2=CLOSET-PAIR(P_R,X_R,Y_R) //T(n/2)
- Combine:
  - δ=min(δ_1, δ_2)
  - compute the minimum distance between the points $p_l \in P_L$ and $p_r \in P_R$. // $O(n)$.
    - Form Y′, which is the points of Y within 2δ-wide vertical strip.
    - For each point p in Y′, 7 following points for comparison.