

Recursion (Part 2)



EECS2101 E:
Fundamentals of Data Structures
Summer 2026

CHEN-WEI WANG

Learning Outcomes of this Lecture

This module is designed to help you:

- Learn about the more intermediate ***recursive algorithms***:
 - Binary Search
 - Merge Sort
 - Quick Sort

Recursion: Binary Search (1)

- **Searching Problem**

Given a numerical key k and an array a of n numbers:

Precondition: Input array a **sorted** in a non-descending order
i.e., $a[0] \leq a[1] \leq \dots \leq a[n-1]$

Postcondition: Return whether or not k exists in the input array a .

- **Q.** RT of a search on an **unsorted** array?

A. $O(n)$ (despite being iterative or recursive)

- **A Recursive Solution**

Base Case: Empty array \rightarrow **false**.

Recursive Case: Array of size $\geq 1 \rightarrow$

- Compare the **middle** element of array a against key k .
 - All elements to the left of **middle** are $\leq k$
 - All elements to the right of **middle** are $\geq k$
- If the **middle** element **is** equal to key $k \rightarrow$ **true**
- If the **middle** element **is not** equal to key k :
 - If $k < \text{middle}$, recursively search key k on the left half.
 - If $k > \text{middle}$, recursively search key k on the right half.

Recursion: Binary Search (2)

```
boolean binarySearch(int[] sorted, int key) {
    return binarySearchH(sorted, 0, sorted.length - 1, key);
}
boolean binarySearchH(int[] sorted, int from, int to, int key) {
    if (from > to) { /* base case 1: empty range */
        return false; }
    else if (from == to) { /* base case 2: range of one element */
        return sorted[from] == key; }
    else {
        int middle = (from + to) / 2;
        int middleValue = sorted[middle];
        if (key < middleValue) {
            return binarySearchH(sorted, from, middle - 1, key);
        }
        else if (key > middleValue) {
            return binarySearchH(sorted, middle + 1, to, key);
        }
        else { return true; }
    }
}
```

Running Time: Binary Search (1)

We define $T(n)$ as the *running time function* of a **binary search**, where n is the size of the input array.

$$\begin{cases} T(0) = 1 \\ T(1) = 1 \\ T(n) = T(\frac{n}{2}) + 1 \quad \text{where } n \geq 2 \end{cases}$$

To solve this recurrence relation, we study the pattern of $T(n)$ and observe how it reaches the **base case(s)**.

Running Time: Binary Search (2)

Without loss of generality, assume $n = 2^i$ for some $i \geq 0$.

$$\begin{aligned}
 T(n) &= T\left(\frac{n}{2}\right) + 1 \\
 &= \underbrace{\left(T\left(\frac{n}{4}\right) + 1\right)}_{T\left(\frac{n}{2}\right)} + \underbrace{1}_{1 \text{ time}} \\
 &= \underbrace{\left(\left(T\left(\frac{n}{8}\right) + 1\right) + 1\right)}_{T\left(\frac{n}{4}\right)} + \underbrace{1}_{2 \text{ times}} \\
 &= \dots \\
 &= \left(\left(\left(\underbrace{1}_{T\left(\frac{n}{2^{\log n}}\right) = T(1)} \right) + 1 \right) \dots \right) + 1 \\
 &\qquad\qquad\qquad \underbrace{\hspace{10em}}_{\log n \text{ times}}
 \end{aligned}$$

$\therefore T(n)$ is $O(\log n)$

Recursion: Merge Sort

- **Sorting Problem**

Given a list of n numbers $\langle a_1, a_2, \dots, a_n \rangle$:

Precondition: NONE

Postcondition: A permutation of the input list $\langle a'_1, a'_2, \dots, a'_n \rangle$
sorted in a non-descending order (i.e., $a'_1 \leq a'_2 \leq \dots \leq a'_n$)

- **A Recursive Algorithm**

Base Case 1: Empty list \rightarrow Automatically sorted.

Base Case 2: List of size 1 \rightarrow Automatically sorted.

Recursive Case: List of size $\geq 2 \rightarrow$

1. **Split** the list into two (**unsorted**) halves: **L** and **R**.
2. **Recursively sort** **L** and **R**, resulting in: **sortedL** and **sortedR**.
3. Return the **merge** of **sortedL** and **sortedR**.

Recursion: Merge Sort in Java (1)

```
/* Assumption: L and R are both already sorted. */
private List<Integer> merge(List<Integer> L, List<Integer> R) {
    List<Integer> merge = new ArrayList<>();
    if(L.isEmpty() || R.isEmpty()) { merge.addAll(L); merge.addAll(R); }
    else {
        int i = 0;
        int j = 0;
        while(i < L.size() && j < R.size()) {
            if(L.get(i) <= R.get(j)) { merge.add(L.get(i)); i++; }
            else { merge.add(R.get(j)); j++; }
        }
        /* If i >= L.size(), then this for loop is skipped. */
        for(int k = i; k < L.size(); k++) { merge.add(L.get(k)); }
        /* If j >= R.size(), then this for loop is skipped. */
        for(int k = j; k < R.size(); k++) { merge.add(R.get(k)); }
    }
    return merge;
}
```

RT(merge)?

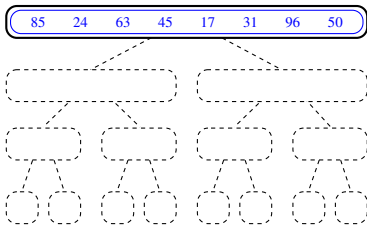
[$O(L.size() + R.size())$]

Recursion: Merge Sort in Java (2)

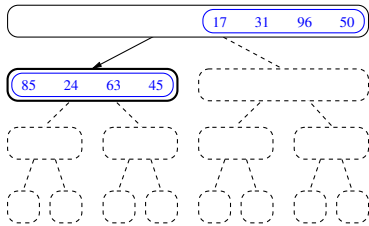
```
public List<Integer> sort(List<Integer> list) {
    List<Integer> sortedList;
    if(list.size() == 0) { sortedList = new ArrayList<>(); }
    else if(list.size() == 1) {
        sortedList = new ArrayList<>();
        sortedList.add(list.get(0));
    }
    else {
        int middle = list.size() / 2;
        List<Integer> left = list.subList(0, middle);
        List<Integer> right = list.subList(middle, list.size());
        List<Integer> sortedLeft = sort(left);
        List<Integer> sortedRight = sort(right);
        sortedList = merge(sortedLeft, sortedRight);
    }
    return sortedList;
}
```

Recursion: Merge Sort Example (1)

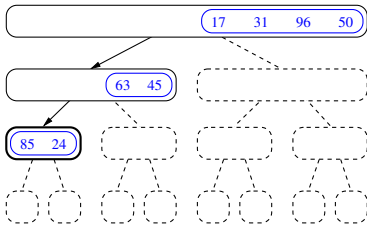
(1) Start with input list of size 8



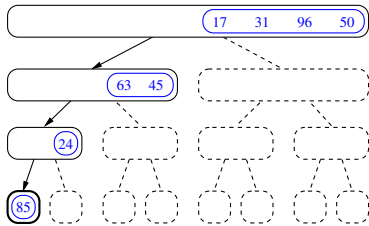
(2) Split and recur on L of size 4



(3) Split and recur on L of size 2

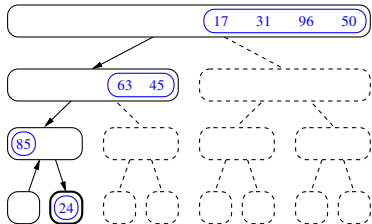


(4) Split and recur on L of size 1, *return*

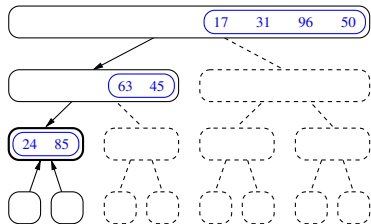


Recursion: Merge Sort Example (2)

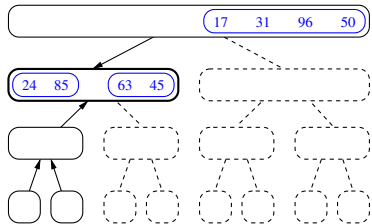
(5) Recur on R of size 1 and *return*



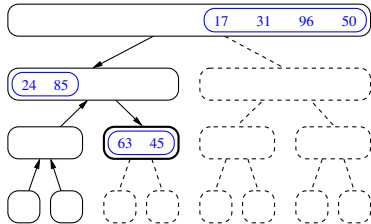
(6) Merge sorted L and R of sizes 1



(7) Return merged list of size 2

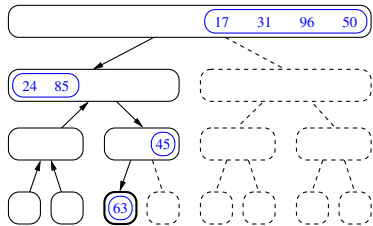


(8) Recur on R of size 2

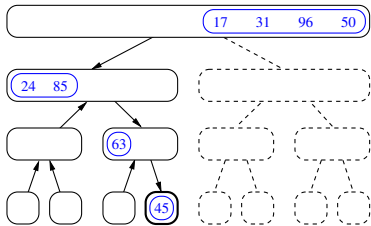


Recursion: Merge Sort Example (3)

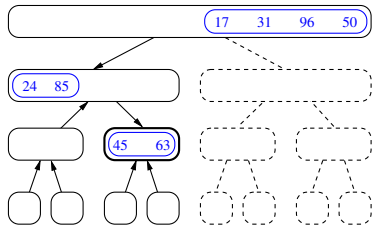
(9) Split and recur on L of size 1, *return*



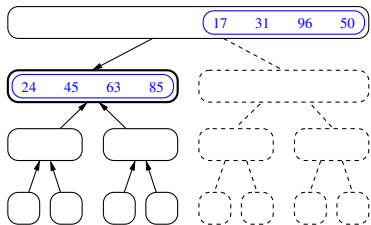
(10) Recur on R of size 1, *return*



(11) Merge sorted L and R of sizes 1, *return*

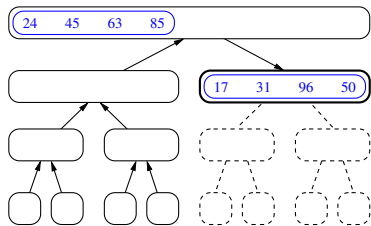


(12) Merge sorted L and R of sizes 2

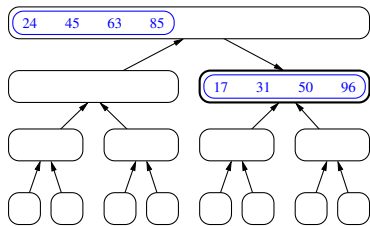


Recursion: Merge Sort Example (4)

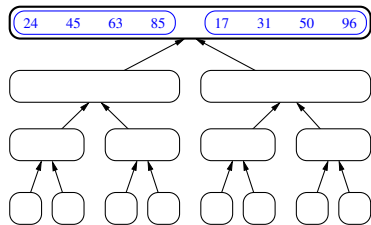
(13) Recur on R of size 4



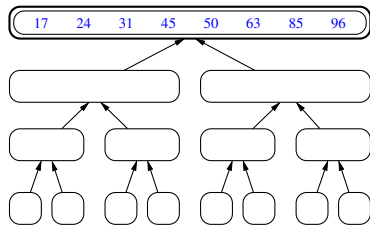
(14) *Return* a sorted list of size 4



(15) Merge sorted L and R of sizes 4



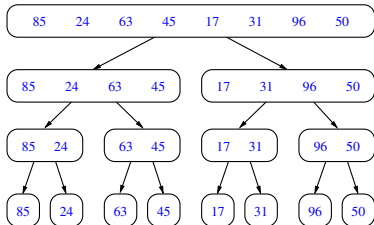
(16) *Return* a sorted list of size 8



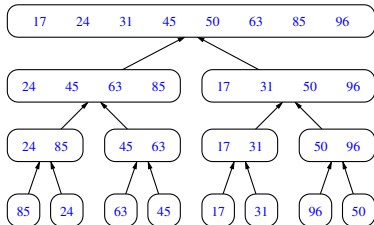
Recursion: Merge Sort Example (5)

Let's visualize the two *critical phases* of **merge sort** :

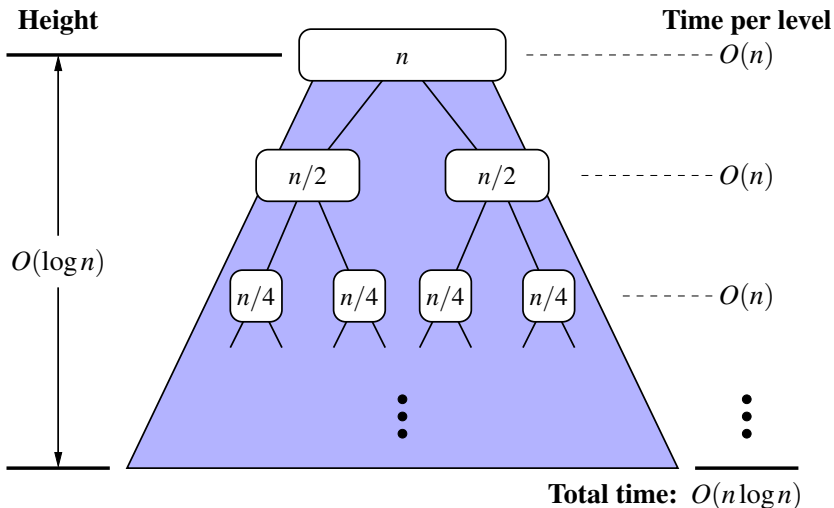
(1) After *Splitting Unsorted* Lists



(2) After *Merging Sorted* Lists



Recursion: Merge Sort Running Time (1)



Recursion: Merge Sort Running Time (2)

- **Base Case 1:** Empty list \rightarrow Automatically sorted. [$O(1)$]
- **Base Case 2:** List of size 1 \rightarrow Automatically sorted. [$O(1)$]
- **Recursive Case:** List of size $\geq 2 \rightarrow$
 1. **Split** the list into two (**unsorted**) halves: **L** and **R**; [$O(1)$]
 2. **Recursively sort** **L** and **R**, resulting in: **sortedL** and **sortedR**
Q. # times to **split** until **L** and **R** have size 0 or 1?
A. [$O(\log n)$]
 3. Return the **merge** of **sortedL** and **sortedR**. [$O(n)$]

Running Time of Merge Sort

$$\begin{aligned} &= (\text{RT each RC}) \times (\# \text{ RCs}) \\ &= (\text{RT merging } \textit{sortedL} \text{ and } \textit{sortedR}) \times (\# \text{ splits until bases}) \\ &= O(n \cdot \log n) \end{aligned}$$

Recursion: Merge Sort Running Time (3)

We define $T(n)$ as the *running time function* of a **merge sort**, where n is the size of the input array.

$$\begin{cases} T(0) = 1 \\ T(1) = 1 \\ T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n \quad \text{where } n \geq 2 \end{cases}$$

To solve this recurrence relation, we study the pattern of $T(n)$ and observe how it reaches the *base case(s)*.

Recursion: Merge Sort Running Time (4)

Without loss of generality, assume $n = 2^i$ for some $i \geq 0$.

$$\begin{aligned}
 T(n) &= 2 \times T\left(\frac{n}{2}\right) + n \\
 &= \underbrace{2 \times \left(2 \times T\left(\frac{n}{4}\right) + \frac{n}{2}\right)}_{2 \text{ terms}} + \underbrace{\frac{n}{2}}_{2 \text{ terms}} + n \\
 &= \underbrace{2 \times \left(2 \times \left(2 \times T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + \frac{n}{2}\right)}_{3 \text{ terms}} + \underbrace{\frac{n}{4}}_{3 \text{ terms}} + \frac{n}{2} + n \\
 &= \dots \\
 &= \underbrace{2 \times \left(2 \times \left(2 \times \dots \times \left(2 \times T\left(\frac{n}{2^{\log n}}\right) + \frac{n}{2^{(\log n)-1}}\right) + \dots + \frac{n}{4} + \frac{n}{2}\right) + n\right)}_{\log n \text{ terms}} \\
 &= \underbrace{2 \cdot \frac{n}{2} + 2^2 \cdot \frac{n}{4} + \dots + 2^{(\log n)-1} \cdot \frac{n}{2^{(\log n)-1}} + \underbrace{n}_{2^{\log n} \cdot \frac{n}{2^{\log n}}}}_{\log n \text{ terms}} \\
 &= \underbrace{n + n + \dots + n + n}_{\log n \text{ terms}}
 \end{aligned}$$

$\therefore T(n)$ is $O(n \cdot \log n)$

Recursion: Quick Sort

- **Sorting Problem**

Given a list of n numbers $\langle a_1, a_2, \dots, a_n \rangle$:

Precondition: NONE

Postcondition: A permutation of the input list $\langle a'_1, a'_2, \dots, a'_n \rangle$
sorted in a non-descending order (i.e., $a'_1 \leq a'_2 \leq \dots \leq a'_n$)

- **A Recursive Algorithm**

Base Case 1: Empty list \rightarrow Automatically sorted.

Base Case 2: List of size 1 \rightarrow Automatically sorted.

Recursive Case: List of size $\geq 2 \rightarrow$

1. Choose a **pivot** element. [ideally the **median**]
2. **Split** the list into two (**unsorted**) halves: **L** and **R**, s.t.:
All elements in **L** are less than or equal to (\leq) the **pivot**.
All elements in **R** are greater than ($>$) the **pivot**.
3. **Recursively sort** **L** and **R**: **sortedL** and **sortedR**;
4. Return the **concatenation** of: **sortedL** + **pivot** + **sortedR**.

Recursion: Quick Sort in Java (1)

```
List<Integer> allLessThanOrEqualTo(int pivotIndex, List<Integer> list)
    List<Integer> sublist = new ArrayList<>();
    int pivotValue = list.get(pivotIndex);
    for(int i = 0; i < list.size(); i++) {
        int v = list.get(i);
        if(i != pivotIndex && v <= pivotValue) { sublist.add(v); }
    }
    return sublist;
}

List<Integer> allLargerThan(int pivotIndex, List<Integer> list) {
    List<Integer> sublist = new ArrayList<>();
    int pivotValue = list.get(pivotIndex);
    for(int i = 0; i < list.size(); i++) {
        int v = list.get(i);
        if(i != pivotIndex && v > pivotValue) { sublist.add(v); }
    }
    return sublist;
}
```

RT(allLessThanOrEqualTo)?

[$O(n)$]

RT(allLargerThan)?

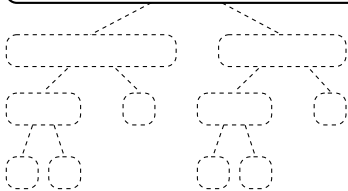
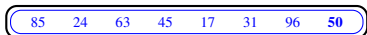
[$O(n)$]

Recursion: Quick Sort in Java (2)

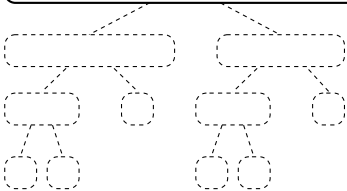
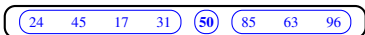
```
public List<Integer> sort(List<Integer> list) {
    List<Integer> sortedList;
    if(list.size() == 0) { sortedList = new ArrayList<>(); }
    else if(list.size() == 1) {
        sortedList = new ArrayList<>(); sortedList.add(list.get(0)); }
    else {
        int pivotIndex = list.size() - 1;
        int pivotValue = list.get(pivotIndex);
        List<Integer> left = allLessThanOrEqualTo(pivotIndex, list);
        List<Integer> right = allLargerThan(pivotIndex, list);
        List<Integer> sortedLeft = sort(left);
        List<Integer> sortedRight = sort(right);
        sortedList = new ArrayList<>();
        sortedList.addAll(sortedLeft);
        sortedList.add(pivotValue);
        sortedList.addAll(sortedRight);
    }
    return sortedList;
}
```

Recursion: Quick Sort Example (1)

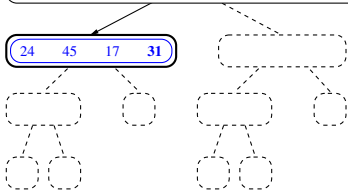
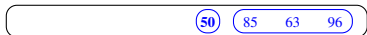
(1) Choose pivot 50 from list of size 8



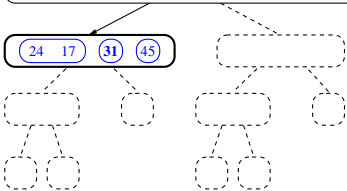
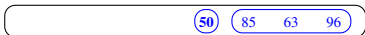
(2) Split w.r.t. the chosen pivot 50



(3) Recur on L of size 4, choose pivot 31

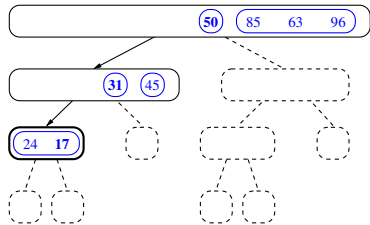


(4) Split w.r.t. the chosen pivot 31

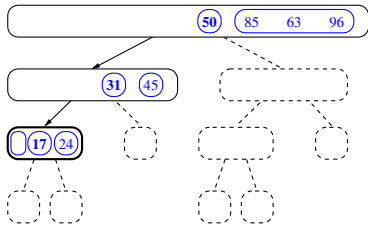


Recursion: Quick Sort Example (2)

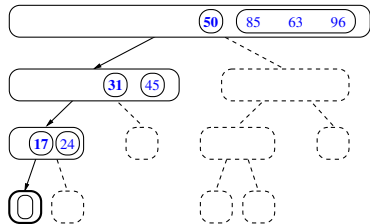
(5) Recur on L of size 2, choose pivot 17



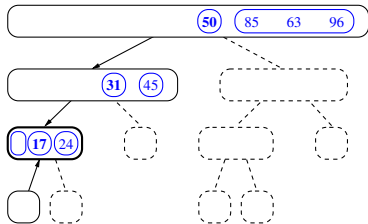
(6) Split w.r.t. the chosen pivot 17



(7) Recur on L of size 0

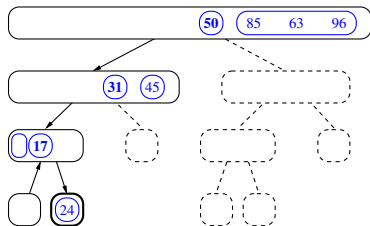


(8) *Return* empty list

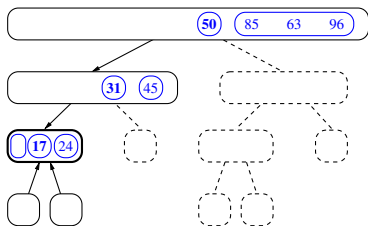


Recursion: Quick Sort Example (3)

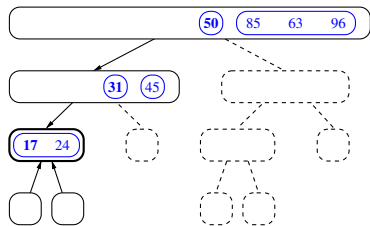
(9) Recur on R of size 1



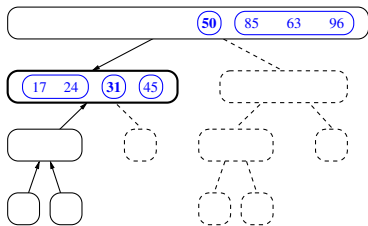
(10) *Return* singleton list <24>



(11) Concatenate {}, <17>, and <24>

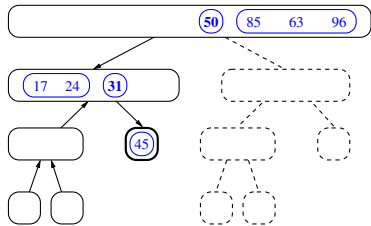


(12) *Return* concatenated list of size 2

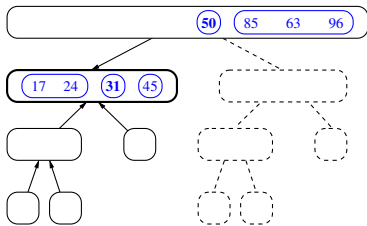


Recursion: Quick Sort Example (4)

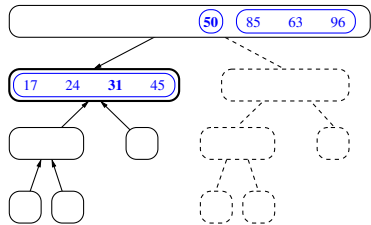
(13) Recur on R of size 1



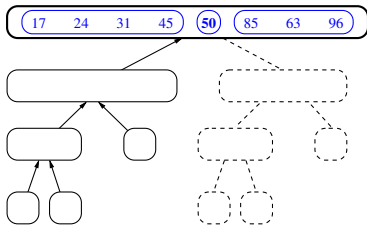
(14) *Return* singleton list <45>



(15) Concatenate <17, 24>, <31>, and <45>

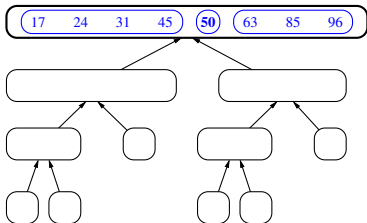


(16) *Return* concatenated list of size 4

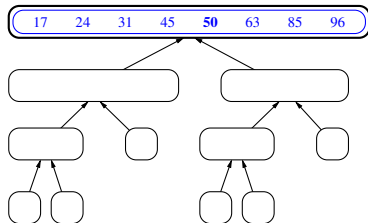


Recursion: Quick Sort Example (5)

(15) Recur on R of size 3



(16) *Return* sorted list of size 3

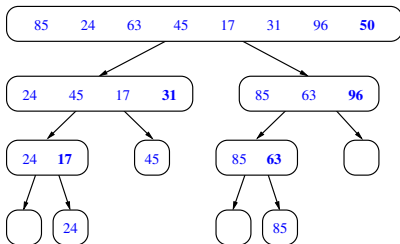


(17) Concatenate $\langle 17, 24, 31, 45 \rangle$, $\langle 50 \rangle$, and $\langle 63, 85, 96 \rangle$, then *return*

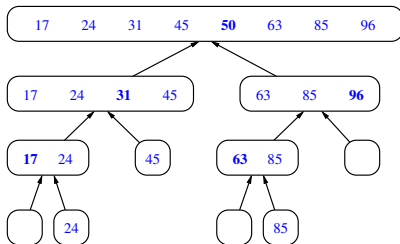
Recursion: Quick Sort Example (6)

Let's visualize the two *critical phases* of **quick sort** :

(1) After *Splitting Unsorted* Lists

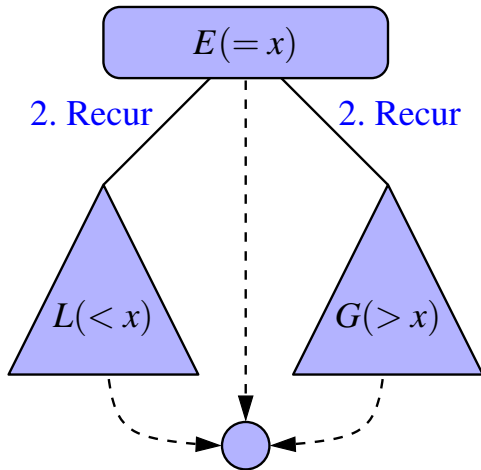


(2) After *Concatenating Sorted* Lists



Recursion: Quick Sort Running Time (1)

1. Split using pivot x



Recursion: Quick Sort Running Time (2)

- **Base Case 1:** Empty list \rightarrow Automatically sorted. [$O(1)$]
- **Base Case 2:** List of size 1 \rightarrow Automatically sorted. [$O(1)$]
- **Recursive Case:** List of size $\geq 2 \rightarrow$
 1. Choose a *pivot* element (e.g., rightmost element) [$O(1)$]
 2. *Split* the list into two (*unsorted*) halves: *L* and *R*, s.t.:
 - All elements in *L* are less than or equal to (\leq) the *pivot*. [$O(n)$]
 - All elements in *R* are greater than ($>$) the *pivot*. [$O(n)$]
 3. **Recursively** *sort* *L* and *R*: *sortedL* and *sortedR*;
 Q. # times to *split* until *L* and *R* have size 0 or 1?
 A. $O(\log n)$ [if pivots chosen are close to **median values**]
 4. Return the *concatenation* of: *sortedL* + *pivot* + *sortedR*. [$O(1)$]

Running Time of Quick Sort

$$\begin{aligned}
 &= (\text{RT each RC}) \times (\# \text{ RCs}) \\
 &= (\text{RT splitting into } L \text{ and } R) \times (\# \text{ splits until bases}) \\
 &= O(n \cdot \log n)
 \end{aligned}$$

Recursion: Quick Sort Running Time (3)

- We define $T(n)$ as the *running time function* of a **quick sort**, where n is the size of the input array.

- Worst Case**

- If the pivot is s.t. the two sub-arrays are “**unbalanced**” in sizes:
e.g., rightmost element in a reverse-sorted array

(“**unbalanced**” splits/partitions: 0 vs. $n - 1$ elements)

$$\begin{cases} T(0) = 1 \\ T(1) = 1 \\ T(n) = T(n-1) + n \quad \text{where } n \geq 2 \end{cases}$$

- As efficient as Selection/Insertion Sorts: $O(n^2)$

[EXERCISE]

- Best Case**

If the pivot is s.t. it is close to the **median** value:

$$\begin{cases} T(0) = 1 \\ T(1) = 1 \\ T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n \quad \text{where } n \geq 2 \end{cases}$$

- As efficient as Merge Sort: $O(n \cdot \log n)$
 - Even with partitions such as $\frac{n}{10}$ vs. $\frac{9 \cdot n}{10}$ elements, RT remains $O(n \cdot \log n)$.

Index (1)

Learning Outcomes of this Lecture

Recursion: Binary Search (1)

Recursion: Binary Search (2)

Running Time: Binary Search (1)

Running Time: Binary Search (2)

Recursion: Merge Sort

Recursion: Merge Sort in Java (1)

Recursion: Merge Sort in Java (2)

Recursion: Merge Sort Example (1)

Recursion: Merge Sort Example (2)

Recursion: Merge Sort Example (3)

Index (2)

Recursion: Merge Sort Example (4)

Recursion: Merge Sort Example (5)

Recursion: Merge Sort Running Time (1)

Recursion: Merge Sort Running Time (2)

Recursion: Merge Sort Running Time (3)

Recursion: Merge Sort Running Time (4)

Recursion: Quick Sort

Recursion: Quick Sort in Java (1)

Recursion: Quick Sort in Java (2)

Recursion: Quick Sort Example (1)

Recursion: Quick Sort Example (2)

Index (3)

Recursion: Quick Sort Example (3)

Recursion: Quick Sort Example (4)

Recursion: Quick Sort Example (5)

Recursion: Quick Sort Example (6)

Recursion: Quick Sort Running Time (1)

Recursion: Quick Sort Running Time (2)

Recursion: Quick Sort Running Time (3)