Recursion

EECS2011 N & Z:
Fundamentals of Data Structures
Winter 2022

CHEN-WEI WANG

Background Study: Basic Recursion

- It is assumed that, in EECS2030, you learned about the basics of ***recursion*** in Java:
  - What makes a method recursive?
  - How to trace recursion using a ***call stack***?
  - How to define and use ***recursive helper methods*** on arrays?
- If needed, review the above assumed basics from the relevant parts of EECS2030 (`https://www.eecs.yorku.ca/~jackie/teaching/lectures/index.html#EECS2030_F21`):
  - Parts A – C, Lecture 8, Week 12

  Tips.
  - Skim the ***slides***: watch lecture videos if needing explanations.
  - Recursion lab from EECS2030-F19: **here** [Solution: **here**]
  - Ask questions related to the assumed basics of ***recursion***!
- Assuming that you know the basics of ***recursion*** in Java, we will proceed with more advanced examples.

Extra Challenging Recursion Problems

1. groupSum
   - Problem Specification: ***here*** Solution: ***here***
   - Solution Walkthrough: ***here***
   - Notes: ***here [pp. 7–10]*** & ***here***
2. parenBit
   - Problem Specification: ***here*** Solution: ***here***
   - Solution Walkthrough: ***here***
   - Notes: ***here [pp. 4–5]***
3. climb
   - Problem Specification: ***here*** Solution: ***here***
   - Solution Walkthrough: ***here*** & ***here***
   - Notes: ***here [pp. 7–8]*** & ***here [p. 4]***
4. climbStrategies
   - Problem Specification: ***here*** Solution: ***here***
   - Solution Walkthrough: ***here***
   - Notes: ***here [pp. 5 – 6]***

Learning Outcomes of this Lecture

This module is designed to help you:

- Know about the resources on ***recursion basics***.
- Learn about the more intermediate ***recursive algorithms***:
  - Binary Search
  - Merge Sort
  - Quick Sort
  - Tower of Hanoi
- Explore extra, ***challenging*** recursive problems.

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., $a[0] \leq a[1] \leq \ldots \leq a[n-1]$
  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 $\longrightarrow$ ***false***.
  Recursive Case: Array of size $\geq 1 \longrightarrow$
  - 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 $k \longrightarrow$ ***true***
  - If the ***middle*** element ***is not*** equal to key k:
    - If $k <$ ***middle***, **recursively** *search* key k on the left half.
    - If $k >$ ***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 $i \geq 0$.

$$\begin{aligned} T(n) &= T(\tfrac{n}{2}) + 1 \\ &= \underbrace{(T(\tfrac{n}{4}) + 1)}_{T(\frac{n}{2})} + \underbrace{1}_{1 \text{ time}} \\ &= \underbrace{((T(\tfrac{n}{8}) + 1)}_{T(\frac{n}{4})} + \underbrace{1) + 1}_{2 \text{ times}} \\ &= \ldots \\ &= (((\underbrace{1}_{T(\frac{n}{2^{\log n}}) = T(1)}) + \underbrace{1) \ldots) + 1}_{\log n \text{ times}} \end{aligned}$$

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

Recursion: Merge Sort

- **Sorting Problem**

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

  Precondition: **NONE**

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

- **A Recursive Algorithm**

  Base Case 1: Empty list ⟶ Automatically sorted.

  Base Case 2: List of size 1 ⟶ Automatically sorted.

  Recursive Case: List of size ≥ 2 ⟶

  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)

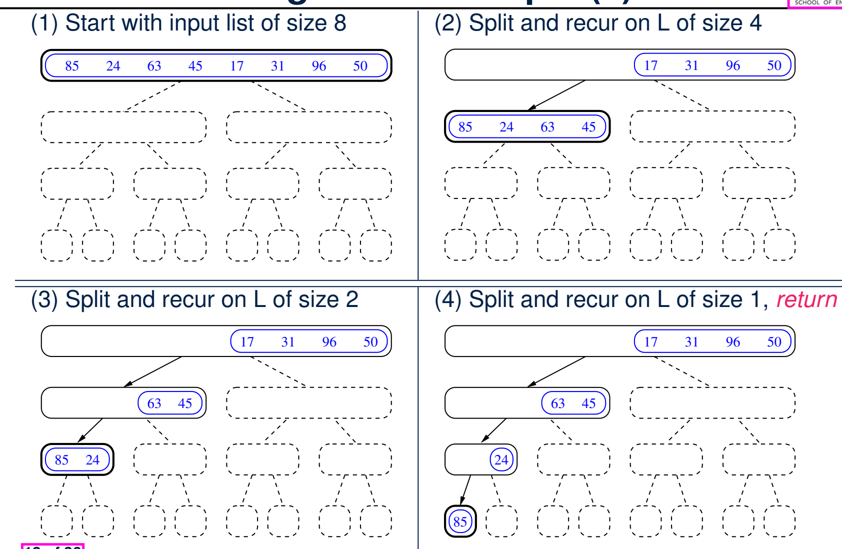
```
/* Assumption:  L and R are both already sorted. */
private List<Integer> merge(List<Integer> L, List<Integer> R) {
  List<Integer> merge = new ArrayList<>();
  if(L.isEmpty()||R.isEmpty()) { merge.addAll(L); merge.addAll(R); }
  else {
    int i = 0;
    int j = 0;
    while(i < L.size() && j < 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)

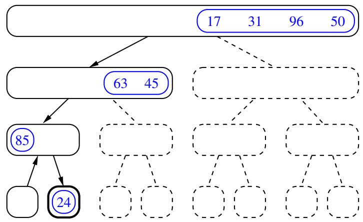
```
public List<Integer> sort(List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>();
    sortedList.add(list.get(0));
  }
  else {
    int middle = list.size() / 2;
    List<Integer> 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)

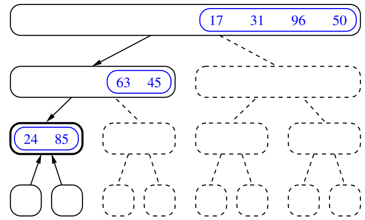


Recursion: Merge Sort Example (2)

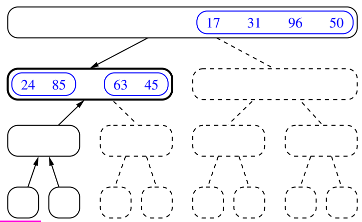
(5) Recur on R of size 1 and *return*



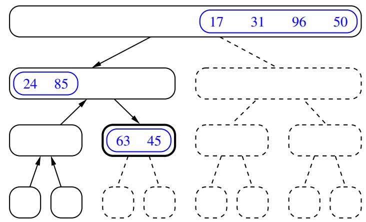
(6) Merge sorted L and R of sizes 1



(7) Return merged list of size 2

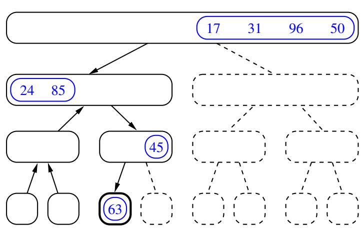


(8) Recur on R of size 2

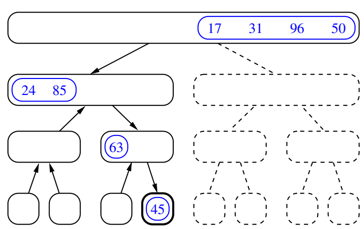


Recursion: Merge Sort Example (3)

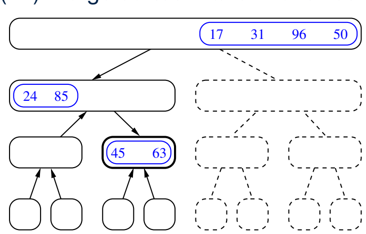
(9) Split and recur on L of size 1, *return*



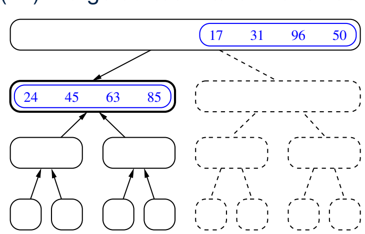
(10) Recur on R of size 1, *return*



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

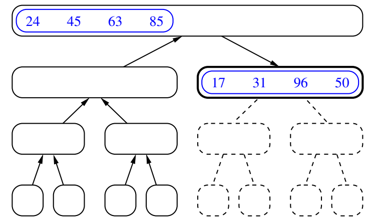


(12) Merge sorted L and R of sizes 2

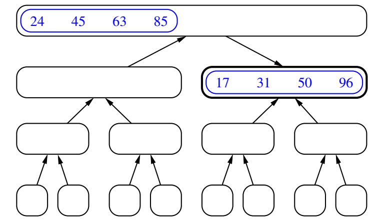


Recursion: Merge Sort Example (4)

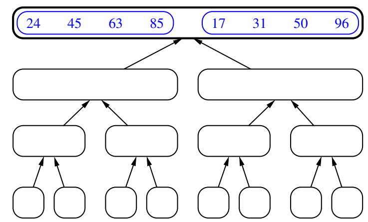
(13) Recur on R of size 4



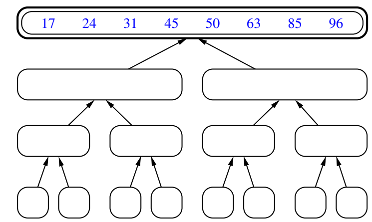
(14) *Return* a sorted list of size 4



(15) Merge sorted L and R of sizes 4



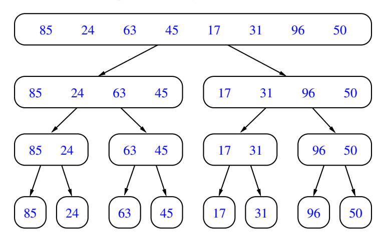
(16) *Return* a sorted list of size 8



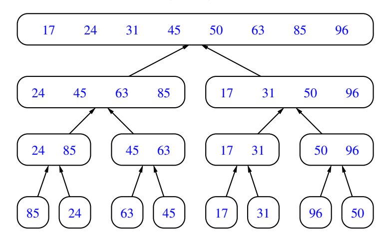
Recursion: Merge Sort Example (5)

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

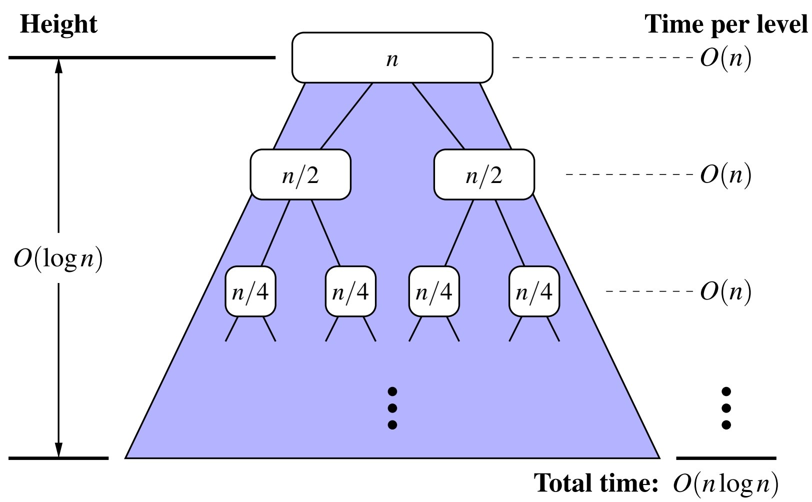
(1) After ***Splitting Unsorted*** Lists



(2) After ***Merging Sorted*** Lists



Recursion: Merge Sort Running Time (1)



Recursion: Merge Sort Running Time (2)

- **Base Case 1**: Empty list ⟶ Automatically sorted. [***O(1)***]
- **Base Case 2**: List of size 1 ⟶ Automatically sorted. [***O(1)***]
- **Recursive Case**: List of size ≥ 2 ⟶
  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
  = (**RT** each RC) × (# **RC**s)
  = (**RT** merging ***sortedL*** and ***sortedR***) × (# splits until bases)
  = $O(n \cdot log\ n)$

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.

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

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 $i \geq 0$.

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

∴ ***T(n)*** is ***O(n · log n)***

Recursion: Quick Sort

- **Sorting Problem**

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

  Precondition: **NONE**

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

- **A Recursive Algorithm**

  Base Case 1: Empty list $\longrightarrow$ Automatically sorted.

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

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

  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)

```java
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;
}
```

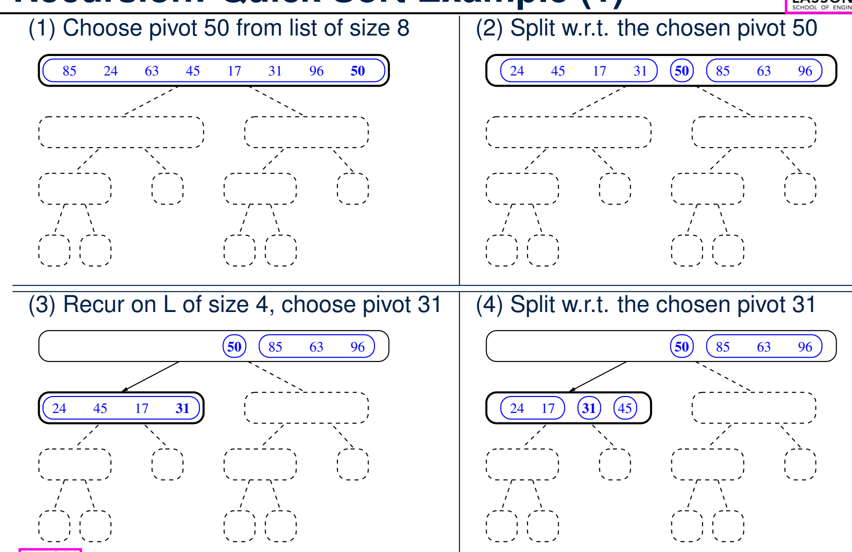
RT(`allLessThanOrEqualTo`)? [***O(n)***]

RT(`allLargerThan`)? [***O(n)***]

Recursion: Quick Sort in Java (2)

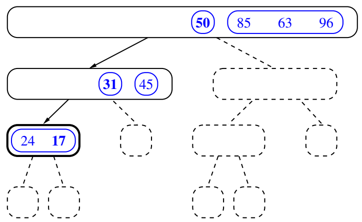
```java
public List<Integer> sort(List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>(); sortedList.add(list.get(0)); }
  else {
    int pivotIndex = list.size() - 1;
    int pivotValue = 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)

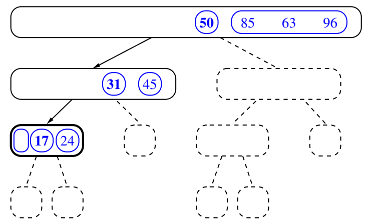


Recursion: Quick Sort Example (2)

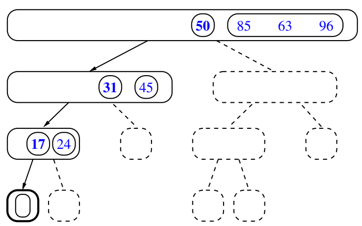
(5) Recur on L of size 2, choose pivot 17



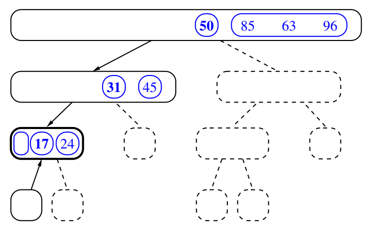
(6) Split w.r.t. the chosen pivot 17



(7) Recur on L of size 0

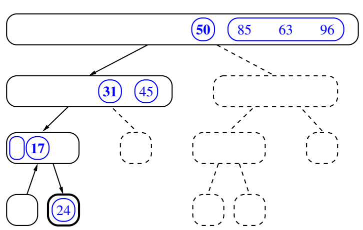


(8) *Return* empty list

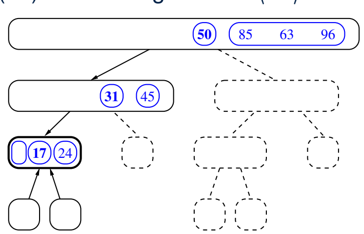


Recursion: Quick Sort Example (3)

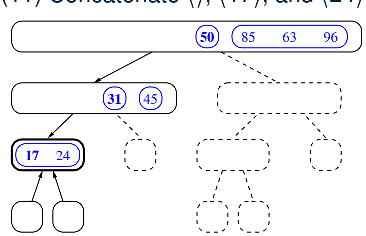
(9) Recur on R of size 1



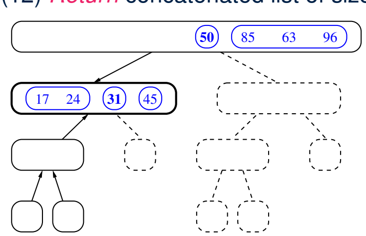
(10) *Return* singleton list ⟨24⟩



(11) Concatenate ⟨⟩, ⟨17⟩, and ⟨24⟩

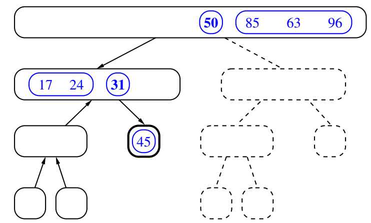


(12) *Return* concatenated list of size 2

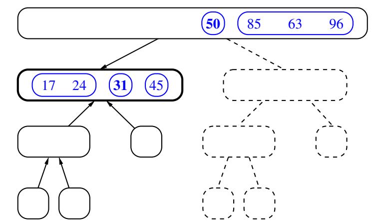


Recursion: Quick Sort Example (4)

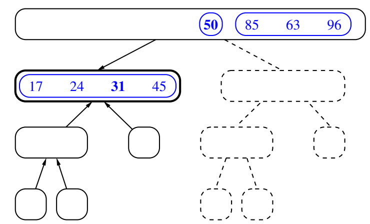
(13) Recur on R of size 1



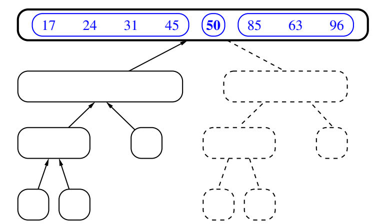
(14) *Return* singleton list ⟨45⟩



(15) Concatenate ⟨17, 24⟩, ⟨31⟩, and ⟨45⟩

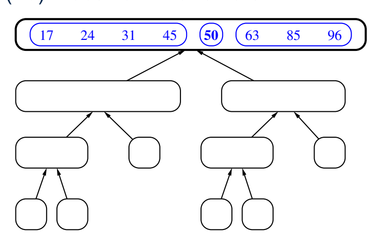


(16) *Return* concatenated list of size 4

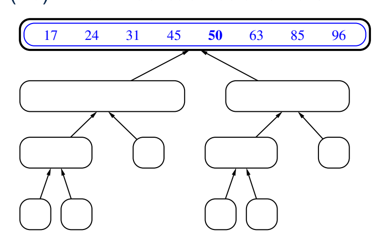


Recursion: Quick Sort Example (5)

(15) Recur on R of size 3



(16) *Return* sorted list of size 3

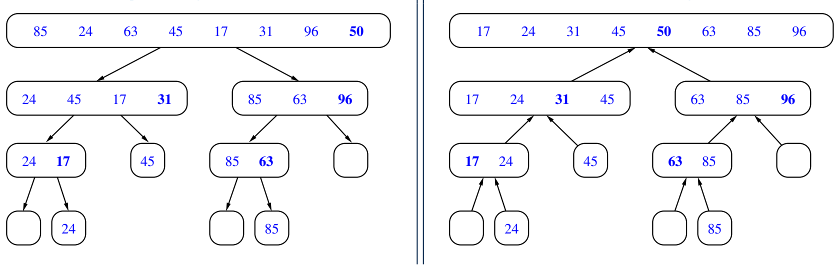


(17) Concatenate ⟨17, 24, 31, 45⟩, ⟨50⟩, and ⟨63, 85, 96⟩, then *return*

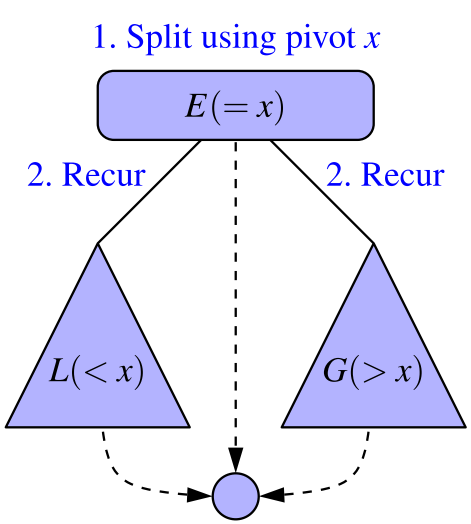
Recursion: Quick Sort Example (6)

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)



Recursion: Quick Sort Running Time (2)

- **Base Case 1**: Empty list ⟶ Automatically sorted. [*O(1)*]
- **Base Case 2**: List of size 1 ⟶ Automatically sorted. [*O(1)*]
- **Recursive Case**: List of size ≥ 2 ⟶
  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 (≤) 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

= (**RT** each RC) × (# **RC**s)
= (**RT** splitting into ***L*** and ***R***) × (# splits until bases)
= $O(n \cdot log\ n)$

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)

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

  - As efficient as Selection/Insertion Sorts: ***O(n²)*** [EXERCISE]
- ***Best Case***
  If the pivot is s.t. it is close to the ***median*** value:

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

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

Beyond this lecture . . .

- Notes on Recursion:
  https://www.eecs.yorku.ca/~jackie/teaching/lectures/2021/F/EECS2030/notes/EECS2030_F21_Notes_Recursion.pdf
- The *best* approach to learning about recursion is via a functional programming language:
  Haskell Tutorial: https://www.haskell.org/tutorial/

Index (1)

Index (2)

Index (3)