Lecture 19 -
Makeup for WrittenTest2

(≈ 90 minutes)

Lecture

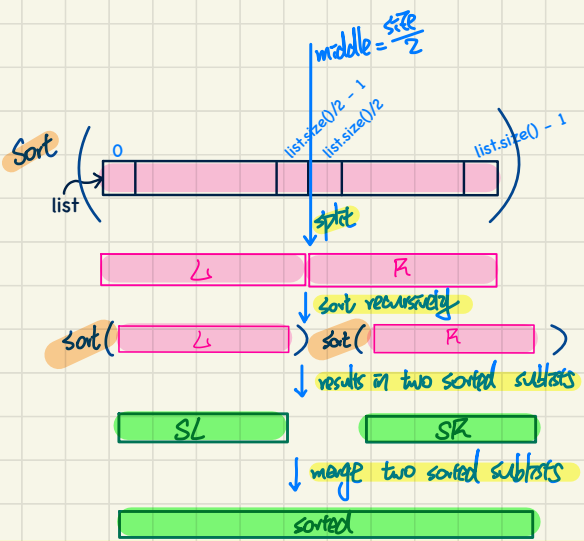
Recursion: Part II (continued)

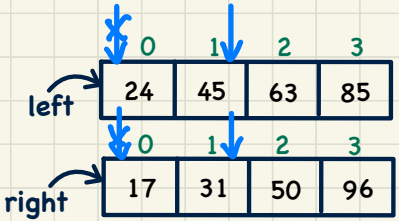
Merge Sort

Merge Sort in Java

```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 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;
}
```

base cases





Precondition

L and R sorted

```java
/* 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;
}
```

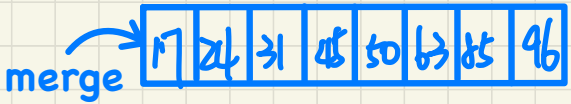
(a) # iterations: min(L.size(), R.size())

O(1)

O(1)

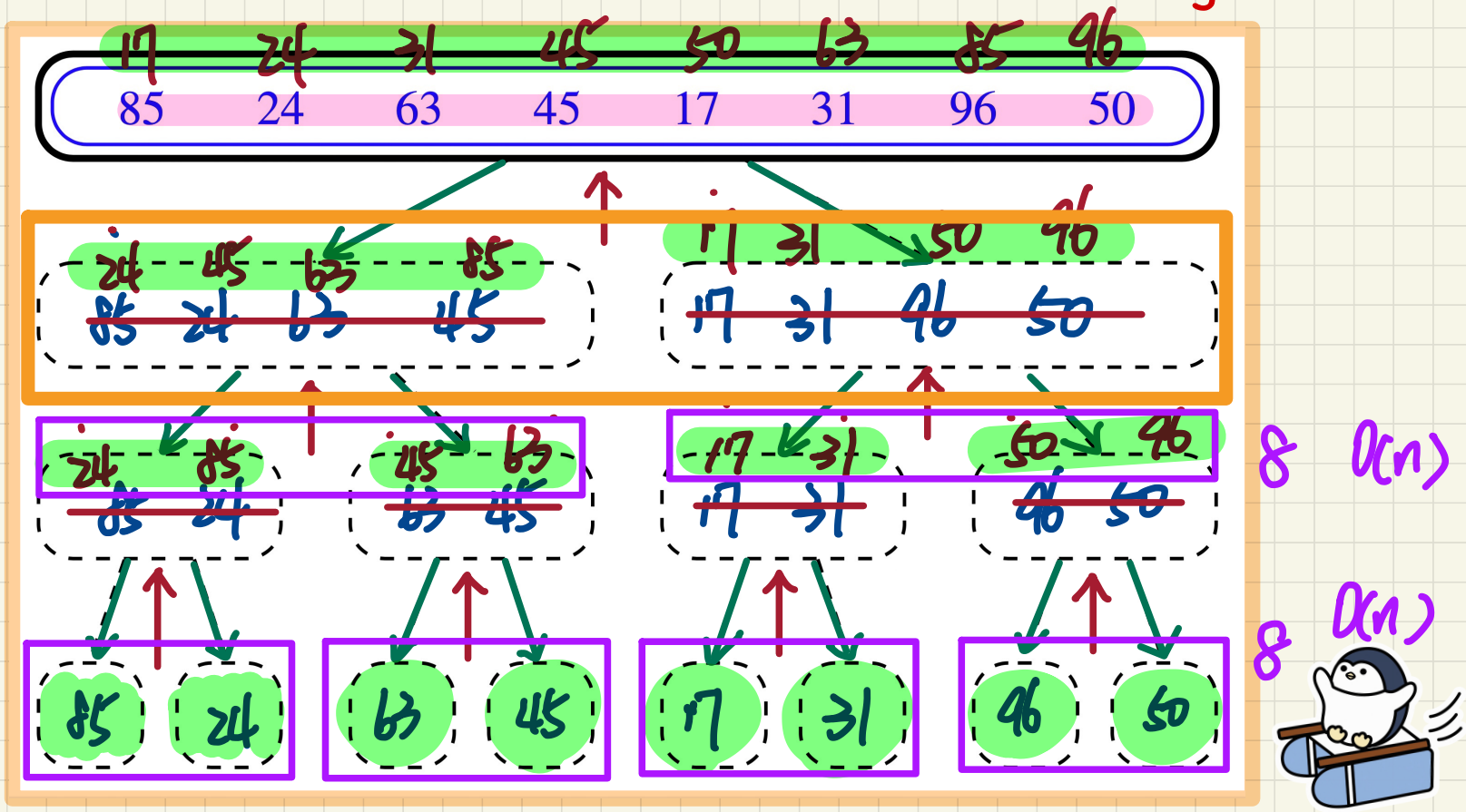
(b) # iterations: remaining # of items to loop over in the longer list

(a) + (b) = L.size() + R.size()



Merge Sort: Tracing

split
merge



Merge Sort: Running Time

```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 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;
}
```

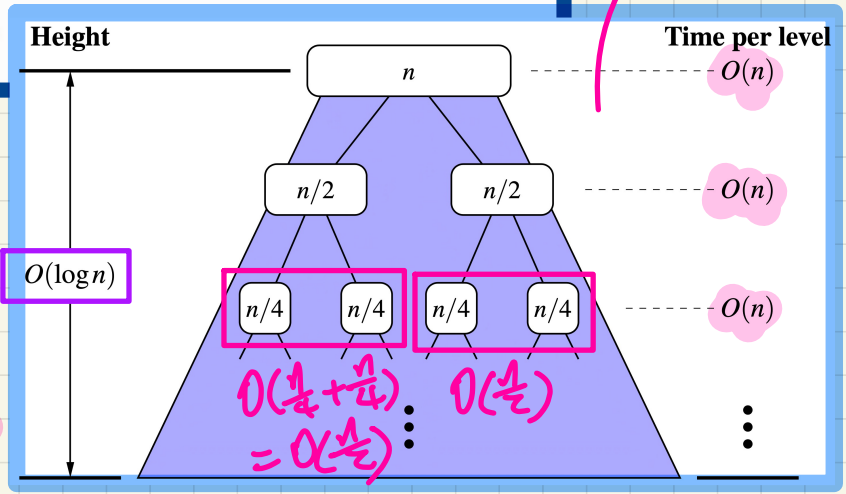
size = $\frac{n}{2}$

sort(left)
sort(right)

O(1)

O(n)

recursion tree is a full BT



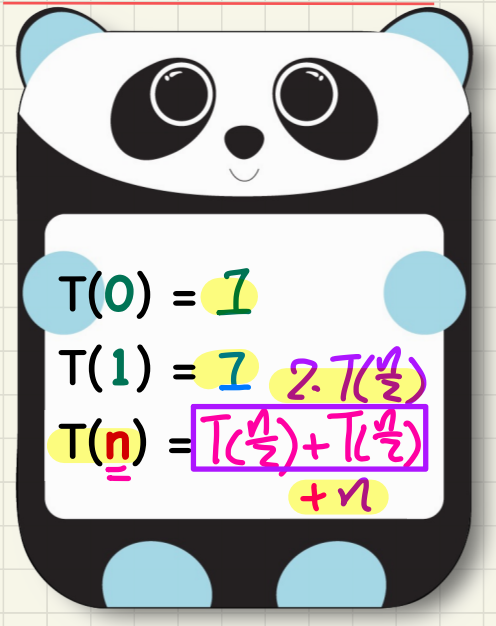
Total RT:

$O(\log n \times n)$

$= O(n \cdot \log n)$

height of balanced BST

Running Time as a Recurrence Relation



Running Time: Unfolding Recurrence Relation

$T(0) = 1$

$T(1) = 1$

$T(n) = 2 \cdot T(n/2) + n$

$T(n) = 2 \cdot T(\frac{n}{2}) + n$

$= 2 \cdot (2 \cdot T(\frac{n}{4}) + \frac{n}{2}) + n$ $\quad [4 \cdot T(\frac{n}{2^2}) + 2n]$

$= 2 \cdot (2 \cdot (2 \cdot T(\frac{n}{8}) + \frac{n}{4}) + \frac{n}{2}) + n$ $\quad [8 \cdot T(\frac{n}{2^3}) + 3n]$

$\vdots$

$= \underbrace{2^{\log n}}_{n} \cdot \underbrace{T(1)}_{\frac{n}{2^{\log n}} \,=\, 1} + \log n \cdot n = n + n \cdot \log n$

$= O(n \cdot \log n)$

$I = \frac{n}{n} = \frac{n}{2^{\log n}}$

$n = 8$

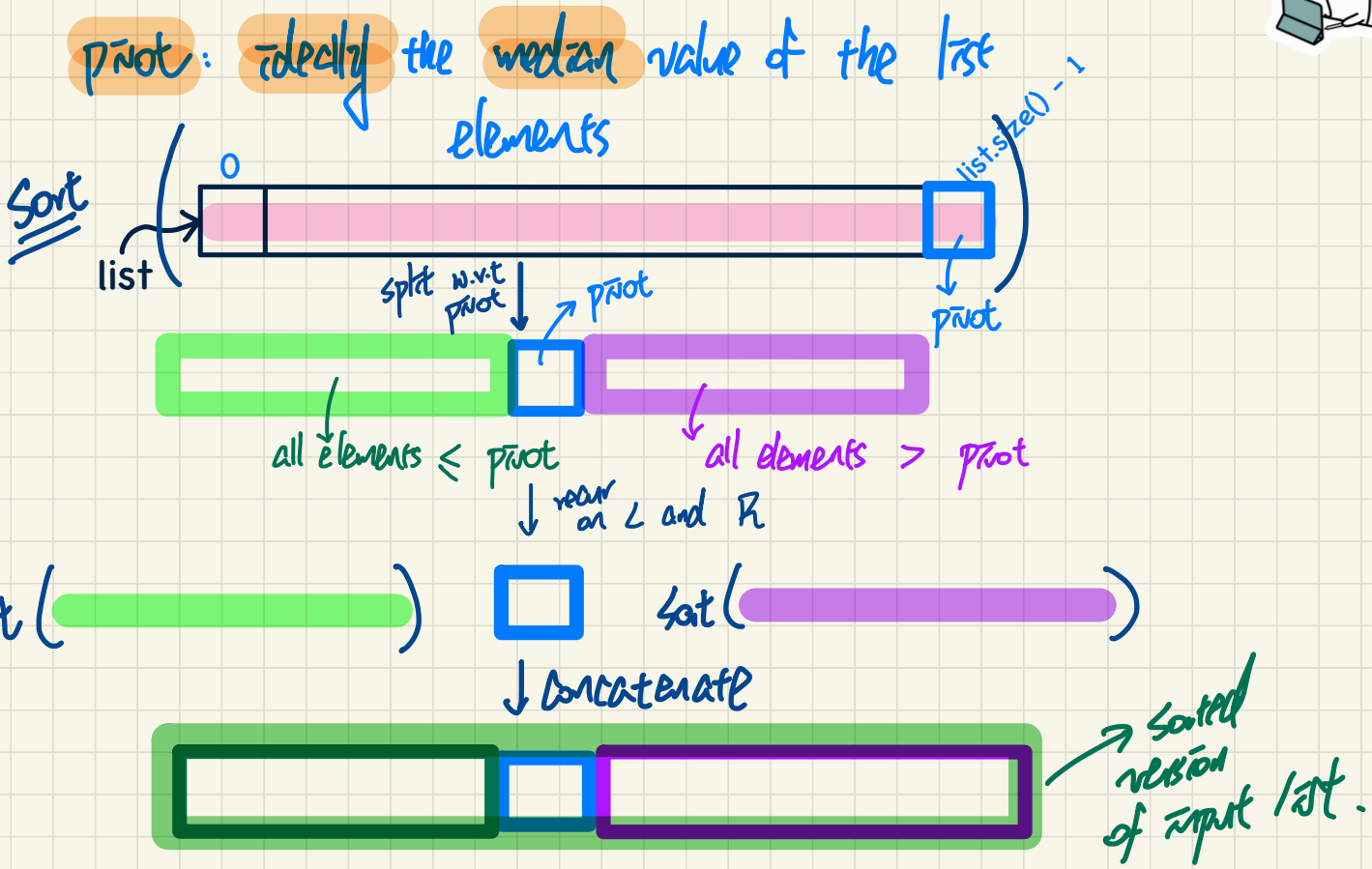
$2^{\log 8} = 8$

Lecture

Recursion: Part II (continued)

Quick Sort

Quick Sort: Ideas



Quick Sort in Java

```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;
}
```

base cases

O(1)

O(1)

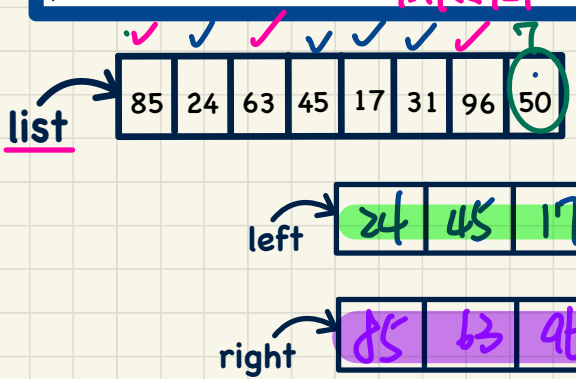
O(n)

O(n)

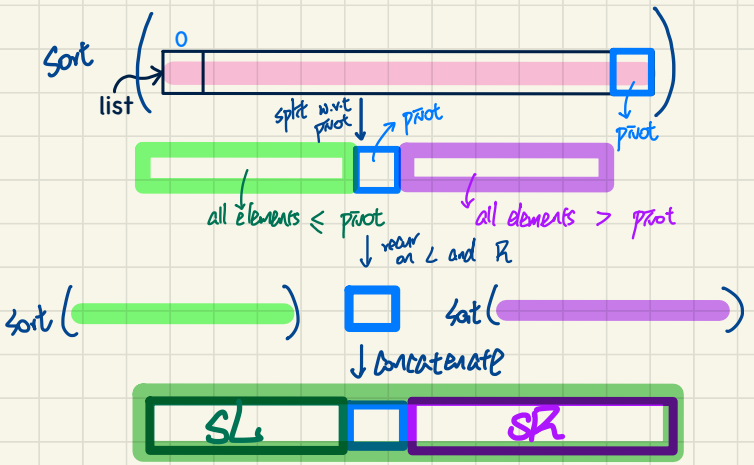
1. Best case: pivot is s.t. $|L| \approx |R|$

2. Worst case: e.g. $|L| = n-1$, $|R| = 0$, or $|R| \ll |L|$

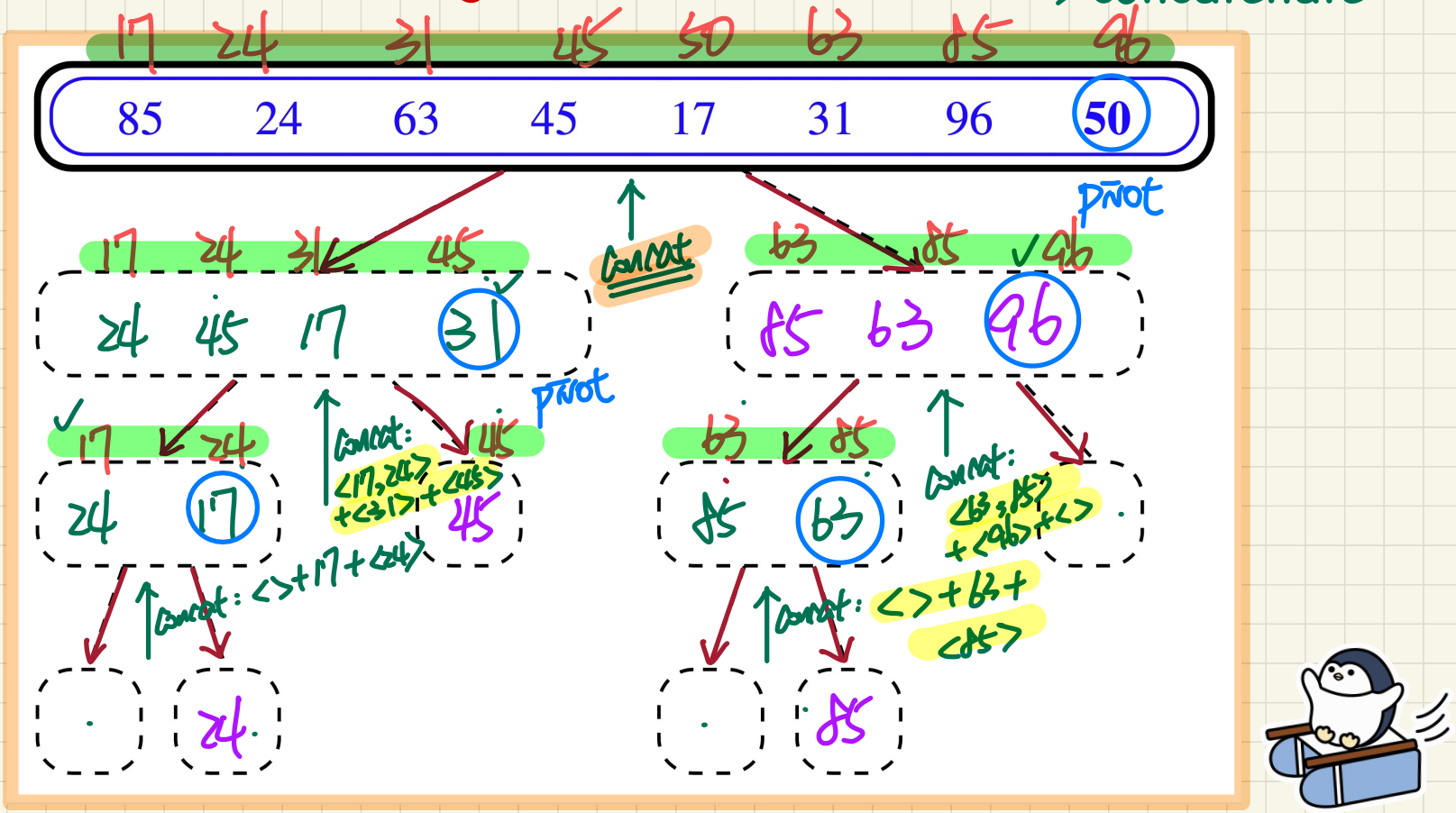
O(n)




```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;
}
```



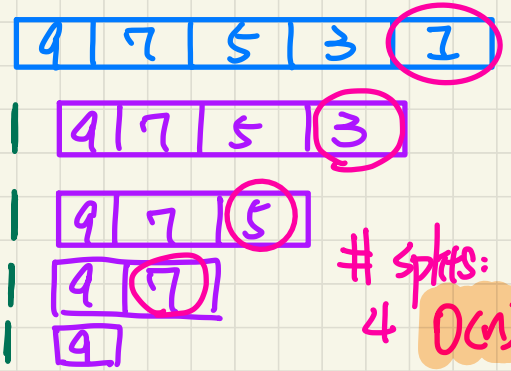
Quick Sort: Tracing

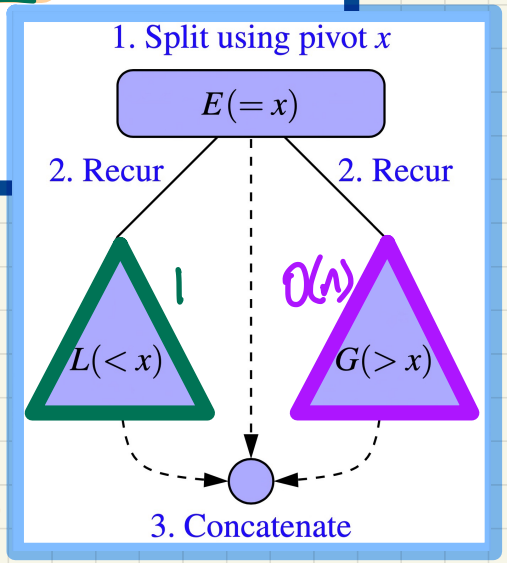


Quick Sort: Worst-Case Running Time

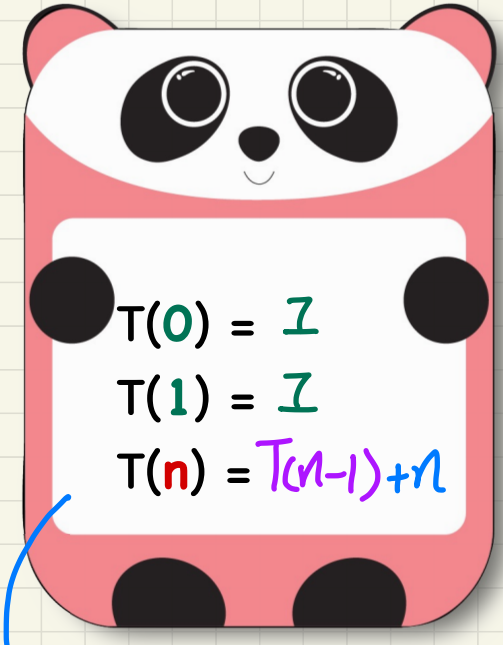
```
public List<Integer> sort(List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>(); sortedList.add(list.get(0)); }
  else {
    int pivotIndex = list.size() - 1;
    int pivotValue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

O(n)





Running Time as a Recurrence Relation



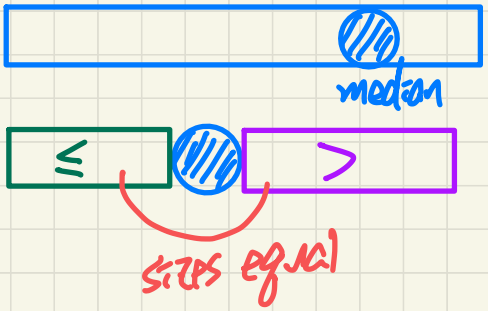
Exercise: Solve by unfolding

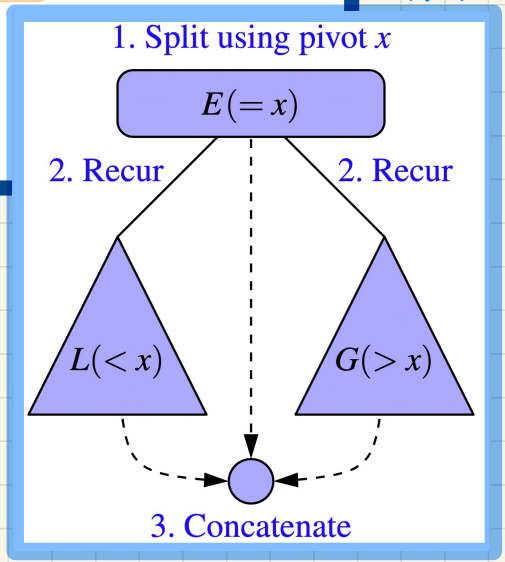
Quick Sort: Best-Case Running Time

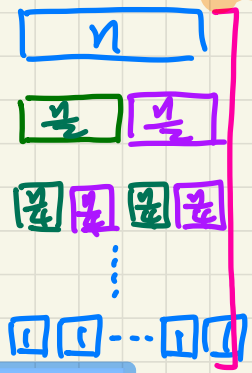
```
public List<Integer> sort(List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>(); sortedList.add(list.get(0)); }
  else {
    int pivotIndex = list.size() - 1;
    int pivotValue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

O(1)

O(n)

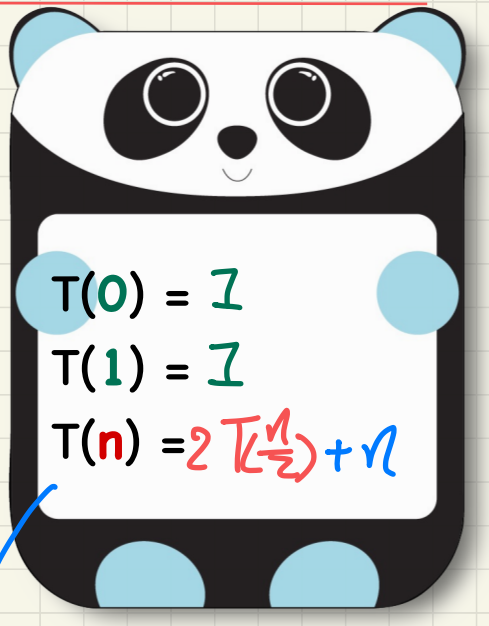






log₂n

Running Time as a Recurrence Relation



Ex.2

Exercise1: Solve by Unfolding.

$T(0) = 1$

$T(1) = 1$

$T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + n$