Lecture 17 - March 18

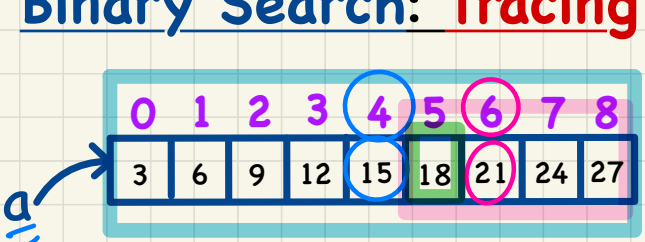
Binary Search, Merge Sort

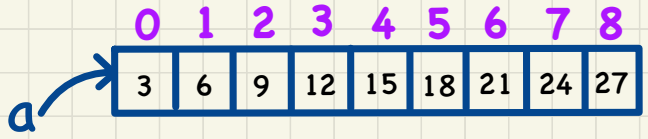
Binary Search: Tracing, Running Time
MergeSort: Ideas, Java, Tracing

Announcements/Reminders

- Assignment 3 (on linked Trees) solution released
- WrittenTest and ProgTest1 results & feedback released
- ProgTest2 guide & example questions to be released
- Makeup Lecture (on Queues) posted
- Lecture notes template, Office Hours, TA Contact

Binary Search: Tracing





Exercise

search(a,7)

binarySearchH(a,0,8,7)

binarySearchH(a,0,3,7)

binarySearchH(a,2,3,7)

binarySearchH(a,2,1,7)

search(a,18) key

binarySearchH(a,0,8,18)

$m = \frac{0+8}{2} = 4$

m.v. a[4] == 15

18 > 15

↳ go R

(4+1, 8)

binarySearchH(a,5,8,18)

$m = \frac{5+8}{2} = 6$

m.v. a[6] == 21

18 < 21

↳ go L.

(5, 6-1)

binarySearchH(a,5,5,18)

a[5] == 18

temp ↳ found !!

Binary Search: Running Time

```
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; }
  }
}
```

n T(n) = ?

T(0) = 1

T(1) = 1

O(1)

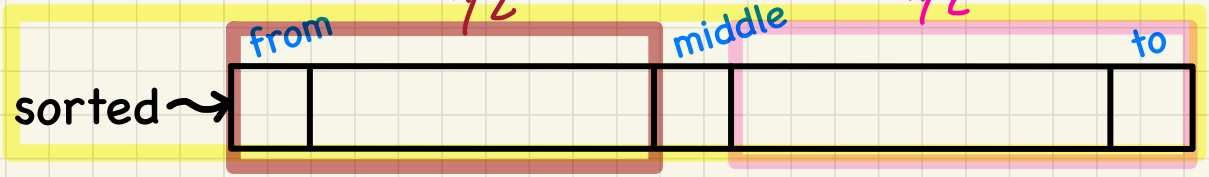
n/2

assume: $T(\frac{n}{2})$

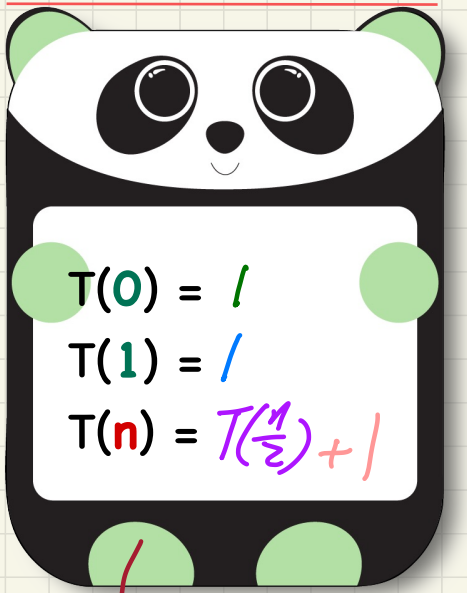
n/2

assume: $T(\frac{n}{2})$

T(n)



Running Time as a Recurrence Relation



wrong: (∵ if-else if)

$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + 1$

Running Time: Unfolding Recurrence Relation

$T(0) = 1$

$T(1) = 1$

✓ $T(n) = T(n/2) + 1$

Assume without loss of generality: $n = 2^x$ for $x \geq 0$

Warm-up: $T(\frac{n}{2}) = T(\frac{n/2}{2}) + 1 = T(\frac{n}{4}) + 1$

$T(n) = T(\frac{n}{2}) + 1$ (2¹)

$= (T(\frac{n}{4}) + 1) + 1$ (2²)

$= ((T(\frac{n}{8}) + 1) + 1) + 1$ (2³)

$\vdots$

$= T(1) + 1 + 1 + \cdots + 1$ (how many? $\log n$.)

$= 1 + \log n \cdot 1 = \log n$

$1 = \frac{n}{n} = \frac{n}{2^{\log n}}$

$2^{\log n} = n$

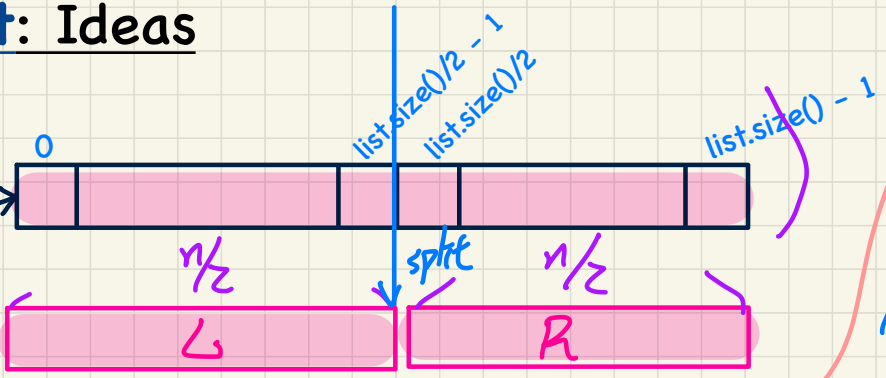
$n = 8$

$\frac{8}{2^{\log 8}} = \frac{n}{n} = 1$

Merge Sort: Ideas

- split
- merge

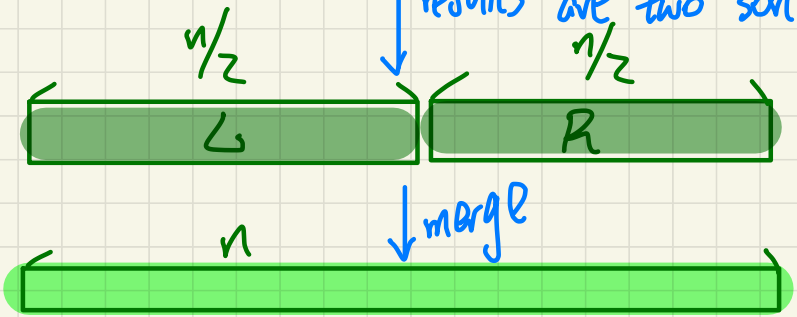
sort (list)



sort recursively

sort (L) sort (R)

results are two sorted lists



Each recursive call:

1. split ($n \to \frac{n}{2}, \frac{n}{2}$) O(1)

$\frac{n}{2} \to \frac{n}{4}, \frac{n}{4}$

$2 \to 1, 1$ (sorted)

2. merge O(n)

$1, 1 \to 2$,

$2, 2 \to 4$,

$\frac{n}{4}, \frac{n}{4} \to \frac{n}{2}$

$\frac{n}{2}, \frac{n}{2} \to n$

Merge Sort in Java

Total # iterations for merge: (L.size + 0) + (R.size − 0)

RT for merge: O(1 · (L.size + R.size))

* Exhaust either L or R = L.size + R.size

↓ O(1)

e.g. L is exhausted

≈ i + j iterations so far. (i = L.size)

** Append remaining elements from R

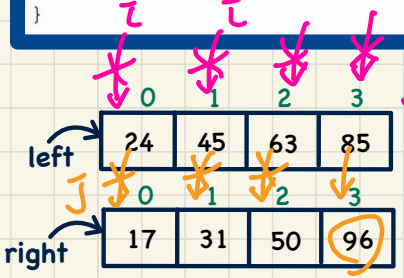
≈ R.size − j iterations.

```
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

O(1)

O(L.size + R.size) L is exhausted



Precondition: L and R are sorted

```
/* 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;
}
```

exit: i >= L.size || j >= R.size

O(1)

O(1)

append the remaining elements from the unexhausted list.

Merge Sort: Tracing

split

merge