CS314H Sorting Algorithms

Mi Kyung Han

Please, interrupt and ask questions AT ANY TIME !

Why sorting?

What is the min/max?

- Unsorted array

0	1	2	3	4	5	6	7	8	9
22	18	12	-4	27	36	30	-7	50	25

- Sorted array

0	1	2	3	4	5	6	7	8	9
-7	-4	12	18	22	25	27	30	36	50

Are they identical sets?

- 2 unsorted arrays

0	1	2	3	4	5	6	7	8	9
22	18	12	-4	27	36	30	-7	30	25

0	1	2	3	4	5	6	7	8	9
18	-4	50	27	25	22	30	12	36	-7

How to find the answer? And how long does it take?

Are they identical sets?

- 2 sorted arrays

0	1	2	3	4	5	6	7	8	9
-7	-4	12	18	22	25	27	30	30	36

0	1	2	3	4	5	6	7	8	9
-7	-4	12	18	22	25	27	30	36	50

How to find the answer? How long does it take now?

Sorting reduces the complexity of a problem

Various sorting algorithms

- **bogo sort**: shuffle and pray
- **bubble sort:** swap adjacent pairs that are out of order
- **selection sort:** look for the smallest element, move to front
- **insertion sort**: build an increasingly large sorted front portion
- **merge sort**: recursively divide the array in half and sort it
- **heap sort**: place the values into a sorted tree structure
- **quick sort**: recursively partition array based on a middle value
- **bucket sort**: cluster elements into smaller groups, sort them
- **radix sort**: sort integers by last digit, then 2nd to last, then ...

Why so many of them?

- Some are faster/slower than others
- Some use more/less memory than others
- Some work better with specific kinds of data
- Some can utilize multiple computers/processors, ...

Outline

1. Intro
2. Vocabs: Sorting Stability and Loop Invariants
3. Primer: Where should max go?
4. Selection Sort
5. Bubble Sort
6. Insertion Sort
7. Merge Sort
8. Conclusion

A stable sort maintains the original order of equal items

Original

0	1	2	3	4
100_1	100_2	100_3	90_1	90_2

Stable Sort

0	1	2	3	4
90_1	90_2	100_1	100_2	100_3

Unstable Sort

0	1	2	3	4
90_2	90_1	100_1	100_3	100_2

When can this be useful?

A stable sort can be useful when searching with multiple criteria

Sorted by Name

0	Alice	A
1	Bob	B
2	Carol	A
3	Dave	A
4	Eric	B

Sort by section →

Stable

0	Alice	A
1	Carol	A
2	Dave	A
3	Bob	B
4	Eric	B

Unstable

0	Dave	A
1	Alice	A
2	Carol	A
3	Eric	B
4	Bob	B

How to proof correctness of algorithm?

Loop invariant – a loop property that is true at the start or end of each iteration.

```
int calcSum(int array[], int n){
    int sum = 0;
    for(int i = 0 ; i < n ; i++){
        sum += array[i];
    }
    return sum;
}
```

Can you find the loop invariant here?

Loop invariant – a loop property that is true at the start or end of each iteration.

```
int calcSum(int array[], int n){
    int sum = 0;
    for(int i = 0 ; i < n ; i++){
        sum += array[i];
    }
    return sum;
}
```

After the 1st iteration (iter with index 0), **sum** contains the sum of subarray **A[0:0]**

After the 2nd iteration (iter with index 1), **sum** contains the sum of subarray **A[0:1]**

After the 3rd iteration (iter with index 2), **sum** contain the sum of subarray **A[0:2]**

After kth iteration, sum contains the sum of subarray A[0:k-1]

Proof of correctness

```
int calcSum(int array[], int n){
    int sum = 0;
    for(int i = 0 ; i < n ; i++){
        sum += array[i];
    }
    return sum;
}
```

- Initialization: i = 0. Before the first iteration, the invariant holds true
  - Before the iteration, sum is 0 (contains nothing from this array)
- Maintenance: If the invariant holds before the iteration, it also holds after the iteration
  - After k^{th} iteration, sum contains the sum of array[0:k-1]
- Terminates: Therefore, when the loop terminates XYZ is true
  - When loop terminates, sum contains sum of arry[0:n-1]

Outline

1. Intro
2. Vocabs: Sorting Stability and Loop Invariants
3. Primer: Where should max go?
4. Selection Sort
5. Bubble Sort
6. Insertion Sort
7. Merge Sort
8. Conclusion

Where should max go in a sorted array?

Unsorted array

0	1	2	3	4	5	6	7	8	9
22	12	30	27	18	50	36	-7	-4	25

Our goal

0	1	2	3	4	5	6	7	8	9
..	..	..	..	..	..	..	..	..	50

How to send max to the end of the array?

Unsorted array

0	1	2	3	4	5	6	7	8	9
22	12	30	27	18	50	36	-7	-4	25

Our goal

0	1	2	3	4	5	6	7	8	9
..	..	..	..	..	..	..	..	..	50

- Case 1: Only one swap is allowed
- Case 2: Multiple swaps are are allowed but only among adjacent values
- No other methods of changing values allowed (Use only swapping!)

Case 1: Only one swap is allowed

Unsorted array

0	1	2	3	4	5	6	7	8	9
22	12	30	27	18	50	36	-7	-4	25

- First need to find max by scanning all elements
- Swap the max and the last element

0	1	2	3	4	5	6	7	8	9
..	..	..	..	..	25	..	..	..	50

This is one iteration of **SELECTION** sort

Outline

1. Intro
2. Vocabs: Loop Invariants and Stable Sorting
3. Primer: Where should max go?
4. Selection Sort
5. Bubble Sort
6. Insertion Sort
7. Merge Sort

Selection sort at each iteration selects max from unsorted and swaps with last of unsorted

Initial array

0	1	2	3	4	5	6	7	8	9
22	12	30	27	18	50	36	-7	-4	25

unsorted

After 1st iteration

0	1	2	3	4	5	6	7	8	9
22	12	30	27	18	25	36	-7	-4	50

unsorted sorted

After 2nd iteration

0	1	2	3	4	5	6	7	8	9
22	12	30	27	18	25	-4	-7	36	50

unsorted sorted

After 3rd iteration

0	1	2	3	4	5	6	7	8	9
22	12	-7	27	18	25	-4	30	36	50

unsorted sorted

After 4th iteration

0	1	2	3	4	5	6	7	8	9
22	12	-7	-4	18	25	27	30	36	50

unsorted sorted

...

After 5th iteration

0	1	2	3	4	5	6	7	8	9
22	12	-7	-4	18	25	27	30	36	50

unsorted sorted

After 6th iteration

0	1	2	3	4	5	6	7	8	9
18	12	-7	-4	22	25	27	30	36	50

unsorted sorted

After 7th iteration

0	1	2	3	4	5	6	7	8	9
-4	12	-7	18	22	25	27	30	36	50

unsorted sorted

After 8th iteration

0	1	2	3	4	5	6	7	8	9
-4	-7	12	18	22	25	27	30	36	50

unsorted sorted

After 9th iteration

0	1	2	3	4	5	6	7	8	9
-7	-4	12	18	22	25	27	30	36	50

sorted

Alternatively, we can
select min from unsorted and swap with the first of unsorted

Initial array

0	1	2	3	4	5	6	7	8	9
22	12	30	27	18	50	36	-7	-4	25

unsorted

After 1st iteration

0	1	2	3	4	5	6	7	8	9
-7	12	30	27	18	50	36	22	-4	25

sorted unsorted

After 2nd iteration

0	1	2	3	4	5	6	7	8	9
-7	-4	30	27	18	50	36	22	12	25

sorted unsorted

After 3rd iteration

0	1	2	3	4	5	6	7	8	9
-7	-4	12	27	18	50	36	22	30	25

sorted unsorted

Is selection sort stable (select min)?

min

Select min

0	1	2	3	4	5	6	7	8	9
22	12	30	27	18	-7_1	36	-4	-7_2	25

- Which one should we swap with 22?

0	1	2	3	4	5	6	7	8	9
-7_1	12	30	27	18	-7_2	36	-4	25	22

- Which one should be the min?

When selecting min, scan from the start and do not update min when there is a tie

Is selection sort stable (select max)?

Select max

0	1	2	3	4	5	6	7	8	9
22	12	30	27	18	50_1	36	-7	50_2	25

- Which one should we swap with 25?

0	1	2	3	4	5	6	7	8	9
22	12	30	27	18	50_1	36	-7	25	50_2

- Which one should be the max?

When selecting max, scan from the end and do not update max when there is a tie

What is the loop invariant for selection sort?

What is true after k^{th} iteration (select min)?

After 1^{st} iteration

0	1	2	3	4	5	6	7	8	9
-7	12	30	27	18	50	36	22	-4	25

sorted | unsorted

After 2^{nd} iteration

0	1	2	3	4	5	6	7	8	9
-7	-4	30	27	18	50	36	22	12	25

sorted | unsorted

After 3^{rd} iteration

0	1	2	3	4	5	6	7	8	9
-7	-4	12	27	18	50	36	22	30	25

sorted | unsorted

After 4^{th} iteration

0	1	2	3	4	5	6	7	8	9
-7	-4	12	18	27	50	36	22	30	25

sorted | unsorted

After kth iteration, subarray A[0:k-1] is sorted

All elements in subarray $A[k:n-1] \geq A[k-1]$

O(1) space complexity

Example code for selection sort

n-1

```
void selectionSort(vector<int>& list) {
    for(int i = 0 ; i < list.size() - 1 ; i++) {
        //Before iteration with index i, list[0:i-1] is sorted
        //find min index from unsorted list[i:n-1]
        int min = i;
        for(int j = i + 1 ; j < list.size() ; j++) {
            if(list[min] > list[j]) {
                min = j;
            }
        }
        //send the min value to the start of unsorted list
        swap(list[i], list[min]);
        //After iteration with index i, list[0:i] is sorted
    }
}
```

What is the time complexity? Space complexity?

Space complexity for selection sort is $\Theta(1)$

- One variable for storing min index
- Another temporary variable for swap
- i and j

Time complexity of selection sort

i=0 i=1 i=2

```
void selectionSort(vector<int>& list) {
    for(int i = 0 ; i < list.size() - 1 ; i++) {
        //Before iteration with index i, list[0:i-1] is sorted
        //find min index from unsorted list[i:n-1]
        int min = i;
        for(int j = i + 1 ; j < list.size() ; j++) {
            if(list[min] > list[j]) {
                min = j;
            }
        }
        //send the min value to the start of unsorted list
        swap(list[i], list[min]);
        //After iteration with index i, list[0:i] is sorted
    }
}
```

$\boxed{n-1}$ **total n-1 iterations**

$c_1\{(n-1)+(n-2)+(n-3) + \dots + 1\}$

$\Theta(1) = c_0$

$n-1, n-2, n-3$

$\Theta(n-1-i) = c_1(n-1-i) + c_2$

$\Theta(1) = c_3$

$$(n-1)(c_0 + c_2 + c_3) + c_1 \sum_{k=1}^{n-1} k = (n-1)(c_0 + c_2 + c_3) + \frac{n \times (n-1)}{2} c_1 = \Theta(n^2)$$

Best case vs worst case for selection sort

- Best case?

0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10

- Worst case?

0	1	2	3	4	5	6	7	8	9
10	9	8	7	6	5	4	3	2	1

Best case: Always scan, no swap

Worst case: Always scan, always swap

Both are $\Theta(n^2)$

Outline

1. Intro
2. Vocabs: Loop Invariants and Stable Sorting
3. Primer: Where should max go?
4. Selection Sort
5. Bubble Sort
6. Insertion Sort
7. Merge Sort
8. Conclusion

How to send max to the end of the array?

Unsorted array

0	1	2	3	4	5	6	7	8	9
22	12	30	27	18	50	36	-7	-4	25

Our goal

0	1	2	3	4	5	6	7	8	9
..	..	..	..	..	..	..	..	..	50

- Case 1: Only one swap is allowed
- Case 2: Multiple swaps are are allowed but only among adjacent values
- No shifting in all cases!

Case 2:
Compare two adjacent values and swap if they are “out of order”

0	1	2	3	4	5	6	7	8	9
22	12	30	27	18	50	36	-7	-4	25

0	1	2	3	4	5	6	7	8	9
..	22	30	..	..	..	..	..	..	..

0	1	2	3	4	5	6	7	8	9
..	..	30	27	..	..	..	..	..	..

0	1	2	3	4	5	6	7	8	9
..	..	..	30	18	..	..	..	..	..

0	1	2	3	4	5	6	7	8	9
..	..	..	..	30	50	..	..	..	..

0	1	2	3	4	5	6	7	8	9
..	..	..	..	..	50	36	..	..	..

0	1	2	3	4	5	6	7	8	9
..	..	..	..	..	..	50	-7	..	..

0	1	2	3	4	5	6	7	8	9
..	..	..	..	..	..	..	50	-4	..

0	1	2	3	4	5	6	7	8	9
..	..	..	..	..	..	..	..	50	25

0	1	2	3	4	5	6	7	8	9
..	..	..	..	..	..	..	..	..	50

This is one iteration of BUBBLE sort

In bubble sort, after each iteration max from unsorted “bubbles up” to the top

Initial array

0	1	2	3	4	5	6	7	8	9
22	12	30	27	18	50	36	-7	-4	25

unsorted

After 1st iteration

0	1	2	3	4	5	6	7	8	9
12	22	27	18	30	36	-7	-4	25	50

unsorted sorted

After 2nd iteration

0	1	2	3	4	5	6	7	8	9
12	22	18	27	30	-7	-4	25	36	50

unsorted sorted

After 3rd iteration

0	1	2	3	4	5	6	7	8	9
12	18	22	27	-7	-4	25	30	36	50

unsorted sorted

After 4th iteration

0	1	2	3	4	5	6	7	8	9
12	18	22	-7	-4	25	27	30	36	50

unsorted sorted

After 5th iteration

0	1	2	3	4	5	6	7	8	9
12	18	-7	-4	22	25	27	30	36	50

unsorted sorted

Is bubble sort stable?

0	1	2	3	4	5	6	7	8	9
22_1	22_2	..	..	..	..	..	..	..	..

- Should we swap?

Don’t swap the adjacent values when they tie

What is the loop invariant for bubble sort?

What is true after k^{th} iteration?

After 1st iteration

0	1	2	3	4	5	6	7	8	9
12	22	27	18	30	36	-7	-4	25	50

unsorted sorted

After 2nd iteration

0	1	2	3	4	5	6	7	8	9
12	22	18	27	30	-7	-4	25	36	50

unsorted sorted

After 3rd iteration

0	1	2	3	4	5	6	7	8	9
12	18	22	27	-7	-4	25	30	36	50

unsorted sorted

After 4th iteration

0	1	2	3	4	5	6	7	8	9
12	18	22	-7	-4	25	27	30	36	50

unsorted sorted

After kth iteration, subarray A[n-k:n-1] is sorted

All elements in subarray A[0:n-k-1] is less than or equal to A[n-k]

Example code for bubble sort

```
void bubbleSort(vector<int>& list) {
    for(int i = (int) list.size() - 1 ; i > 0 ; i--) {
        //Before iteration with index i, list[i+1:n-1] is sorted
        //bubble up max of unsorted list[0:i] to list[i]
        for(int j = 0 ; j < i ; j++) {
            if(list[j] > list[j + 1]) {
                swap(list[j], list[j + 1]);
            }
        }
        //After iteration with index i, list[i:n-1] is sorted
    }
}
```

What is the time complexity? Space complexity?

Space complexity for bubble sort is $\Theta(1)$

- One temporary variable for swap
- i and j

Time complexity for bubble sort

```
void bubbleSort(vector<int>& list) {
    for(int i = (int) list.size() - 1 ; i > 0 ; i--) {
        //Before iteration with index i, list[i+1:n-1] is sorted
        //bubble up max of unsorted list[0:i] to list[i]
        for(int j = 0 ; j < i ; j++) {
            if(list[j] > list[j + 1]) {
                swap(list[j], list[j + 1]);
            }
        }
        //After iteration with index i, list[i:n-1] is sorted
    }
}
```

total n-1 iterations

$\Theta(i) = c_0(i) + c_1$

$$(n-1)c_1 + c_0 \sum_{i=1}^{n-1} i = (n-1)c_1 + \frac{n \times (n-1)}{2} c_0 = \Theta(n^2)$$

Best case vs worst case for bubble sort

- Best case?

0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10

- Worst case?

0	1	2	3	4	5	6	7	8	9
10	9	8	7	6	5	4	3	2	1

Both is $\Theta(n^2)$

Is bubble sort faster than selection sort?

- Why or why not?

How about shifting items instead of swapping?

How about shifting items instead of swapping?

It’s called insertion sort

Outline

Case 3: Shifting values are allowed

Initial array

0	1	2	3	4	5	6	7	8	9
22	12	-4	15	18	50	36	-7	30	25

- As we read the first value from the unsorted find out its correct position in the sorted and insert!
- Where should 12 go in the sorted array?

0	1	2	3	4	5	6	7	8	9
12	22	-4	15	18	50	36	-7	30	25

This is one iteration of INSERTION sort

Insertion sort at each iteration
reads the next value from unsorted and inserts it to sorted

Initial array

0	1	2	3	4	5	6	7	8	9
22	12	-4	15	18	50	36	-7	30	25

sorted unsorted

After 1st iteration

0	1	2	3	4	5	6	7	8	9
12	22	-4	15	18	50	36	-7	30	25

sorted unsorted

After 2nd iteration

0	1	2	3	4	5	6	7	8	9
-4	12	22	15	18	50	36	-7	30	25

sorted unsorted

After 3rd iteration

0	1	2	3	4	5	6	7	8	9
-4	12	15	22	18	50	36	-7	30	25

sorted unsorted

After 4th iteration

0	1	2	3	4	5	6	7	8	9
-4	12	15	18	22	50	36	-7	30	25

sorted unsorted

After 5th iteration

0	1	2	3	4	5	6	7	8	9
-4	12	15	18	22	50	36	-7	30	25

sorted unsorted

Is insertion sort stable?

- Where should we insert?

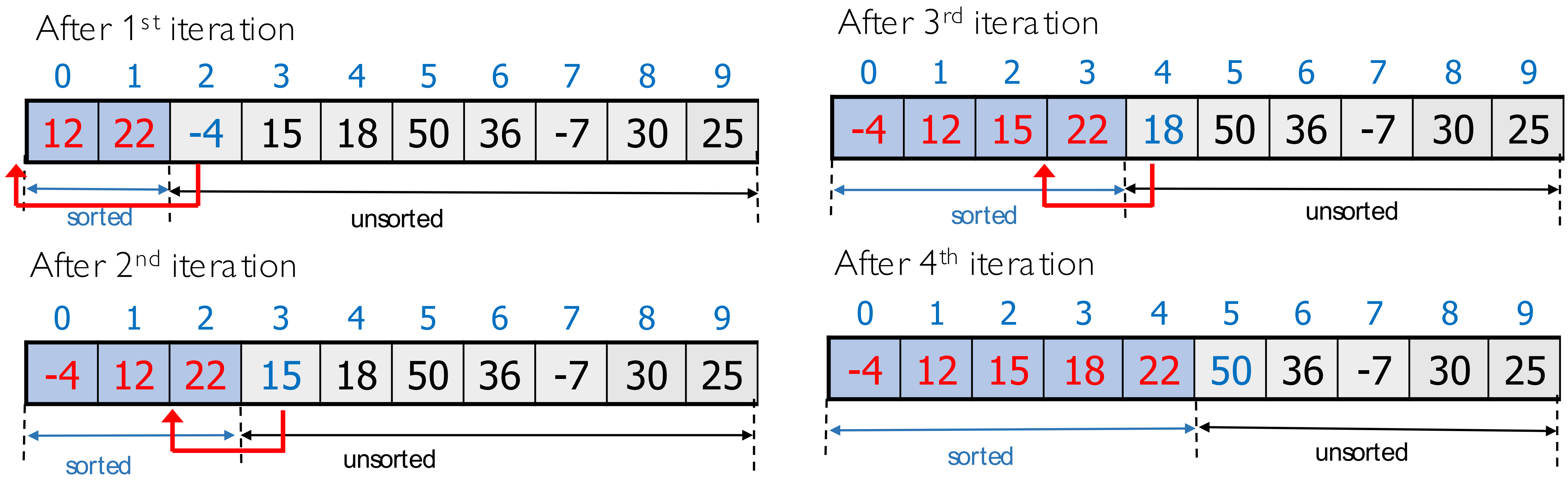
0	1	2	3	4	5	6	7	8	9
25_1	25_2	..	..	..	..	..	..	..	..

0	1	2	3	4	5	6	7	8	9
25_1	25_2	27	28	25_3	..	..	..	..	..

Yes! Insert after the last element with equal value

What is the loop invariant for insertion sort?

What is true after k^{th} iteration?



After kth iteration, subarray A[0:k] is sorted

No claim can be made for elements in subarray A[k+1:n-1]

Example code for insertion sort

```
void insertionSort(vector<int>& list) {
    for(int i = 1 ; i < list.size() ; i++) { //n-1
        //Before iteration with index i, list[0:i-1] is sorted
        //find a hole to place list[i] from sorted list[0:i-1]
        int val = list[i];
        int hole = i;
        while (hole > 0 && val < list[hole - 1]) {//
            list[hole] = list[hole - 1]; //shift right
            hole--;
        }
        list[hole] = val;
        //After iteration with index i, list[0:i] is sorted
    }
}
```

What is the time complexity? Space complexity?

Space complexity for insertion sort is $\Theta(1)$

- One variable for shifting without losing any info
- Variable for hole
- i

Time complexity for insertion sort

```
void insertionSort(vector<int>& list) {
    for(int i = 1 ; i < list.size() ; i++) {
        //Before iteration with index i, list[0:i-1] is sorted
        //find a hole to place list[i] from sorted list[0:i-1]
        int val = list[i];
        int hole = i;
        while (hole > 0 && val < list[hole - 1]) {
            list[hole] = list[hole - 1]; //shift right
            hole--;
        }
        list[hole] = val;
        //After iteration with index i, list[0:i] is sorted
    }
}
```

total n-1 iterations

$\Theta(1) = c_0$

Worst case: $\Theta(i) = c_1 i + c_2$

$\Theta(1) = c_3$

$$(n-1)(c_0+c_2+c_3) + c_1\sum_{i=1}^{n-1} i = (n-1)(c_0+c_2+c_3) + \frac{n\times(n-1)}{2}c_1 = \Theta(n^2)$$

Time complexity for insertion sort

```
void insertionSort(vector<int>& list) {
    for(int i = 1 ; i < list.size() ; i++) {
        //Before iteration with index i, list[0:i-1] is sorted
        //find a hole to place list[i] from sorted list[0:i-1]
        int val = list[i];
        int hole = i;
        while (hole > 0 && val < list[hole - 1]) {
            list[hole] = list[hole - 1]; //shift right
            hole--;
        }
        list[hole] = val;
        //After iteration with index i, list[0:i] is sorted
    }
}
```

total n-1 iterations

$\Theta(1) = c_0$

Best case: $\Theta(1) = c_2$

$\Theta(1) = c_3$

$$(\boldsymbol{n} - \mathbf{1})(\boldsymbol{c_0} + \boldsymbol{c_2} + \boldsymbol{c_3}) = \Theta(\boldsymbol{n})$$

Best case vs worst case for insertion sort

- Best case?

0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10

- Worst case?

0	1	2	3	4	5	6	7	8	9
10	9	8	7	6	5	4	3	2	1

The best case is $\Theta(n)$ and the worst case is $\Theta(n^2)$

Selection vs Bubble vs Insertion sort

- Which one is the fastest?

- The slowest?

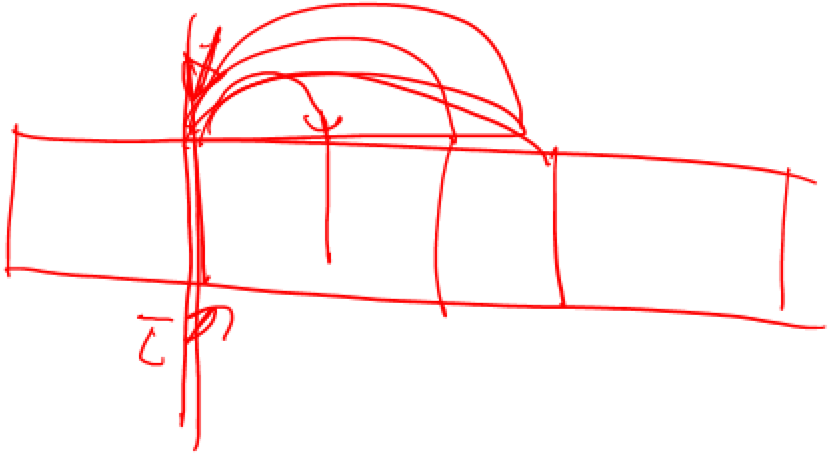
How can we improve insertion sort?

- What could be a better approach to find the hole?

20

0	1	2	3	4	5	6	7	8	9
-4	12	15	18	22	36	50	-7	30	25

sorted | unsorted



  - Hint: searching for something in a sorted array

Binary search!

Worst-case $\Theta(n^2)$ sounds pretty bad.
Can we do better?

Outline

1. Intro
2. Vocabs: Loop Invariants and Stable Sorting
3. Primer: Where should max go?
4. Selection Sort
5. Bubble Sort
6. Insertion Sort
7. Merge Sort
8. Conclusion

Merge sort is a divide-and-conquer algorithm

- Divide the list into two roughly equal halves.
- Sort the left half.
- Sort the right half.
- Merge the two sorted halves into one sorted list.

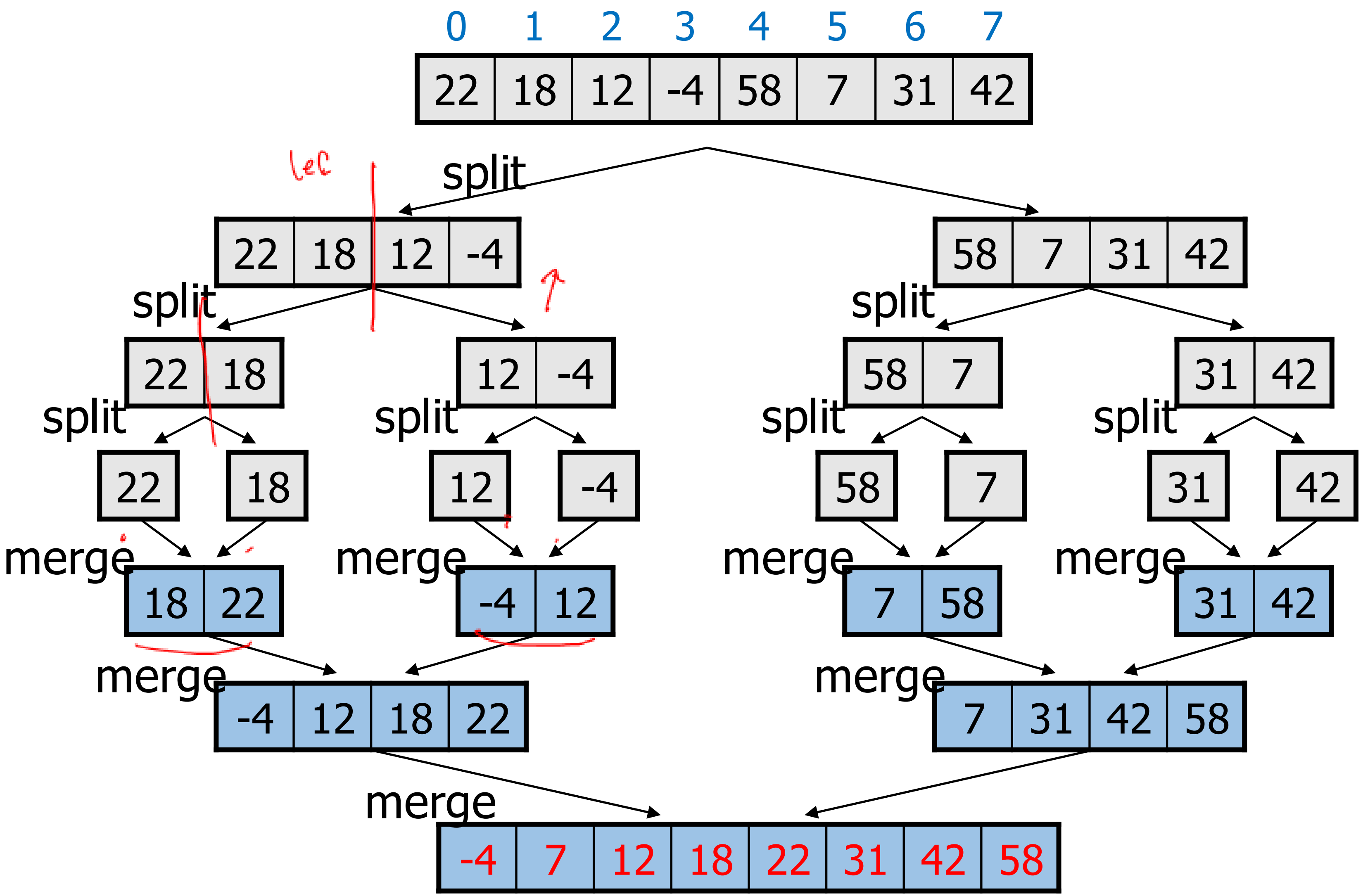
- Divide the list into two roughly equal halves.
- Sort the left half.
- Sort the right half.
- Merge the two sorted halves into one sorted list.

Who is going to sort these halves?

Merge sort is a recursive algorithm

- Divide the list into two roughly equal halves.
- Sort the left half.
- Sort the right half.
- Merge the two sorted halves into one sorted list.

Merge sort example



Let’s implement merge!

Merging two sorted halves into one

Subarrays	Next include	Merged array

0	1	2	3		0	1	2	3
14	32	67	76		23	41	58	85
i1					i2			

Merge halves code: merge2

```
//a version of merge used for mergeSort2
//merges two separate halves (left and right) into list
//pre-condition: 1) left and right is sorted
//               2) size of list is equal to size of left + size of right
void merge2(vector<int>& list, const vector<int> left, const vector<int> right){
    int i1 = 0; //left index
    int i2 = 0; //right index
    for(int i = 0 ; i < list.size(); i++){

        /* Implement me */

    }
}
```

Merge halves code: merge2

```cpp
//a version of merge used for mergeSort2
//merges two separate halves (left and right) into list
//pre-condition: 1) left and right is sorted
//               2) size of list is equal to size of left + size of right
void merge2(vector<int>& list, const vector<int> left, const vector<int> right){
    int i1 = 0; //left index
    int i2 = 0; //right index
    for(int i = 0 ; i < list.size(); i++){
        if(
```

What would be this condition?

```cpp
                                          {
            list[i] = left[i1++]; //take from left
        } else {
            list[i] = right[i2++]; //take from right
        }
    }
}
```

merge2 completed

```
//a version of merge used for mergeSort2
//merges two separate halves (left and right) into list
//pre-condition: 1) left and right is sorted
//               2) size of list is equal to size of left + size of right
void merge2(vector<int>& list, const vector<int> left, const vector<int> right){
    int i1 = 0; //left index
    int i2 = 0; //right index
    for(int i = 0 ; i < list.size(); i++){
        if(i1 < left.size() &&
            (i2 >= right.size() || left[i1] <= right[i2])) {
            list[i] = left[i1++]; //take from left
        } else {
            list[i] = right[i2++]; //take from right
        }
    }
}
```

$\frac{n}{2}$ $\frac{n}{2}$

Time complexity of merge is Θ(n)

mergeSort2 code (version 1)

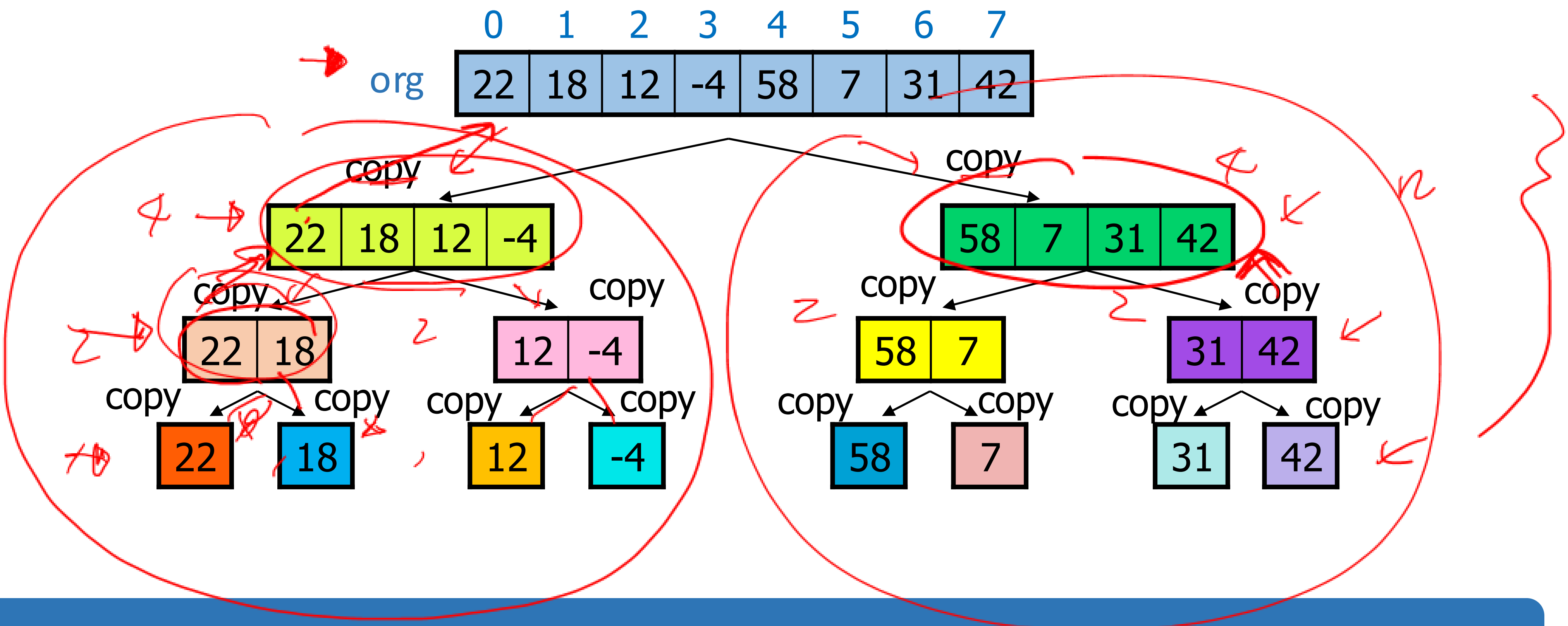
```
//sub-optimal merge sort
void mergeSort2(vector<int>& list){
    if(list.size() <= 1) return; //base case

    // copy list into two halves (left and right)
    vector<int> left(list.size()/2);
    vector<int> right(list.size() - left.size());
    std::copy(list.begin(), list.begin() + left.size() ,left.begin());
    std::copy(list.begin() + left.size(), list.end(), right.begin());

    // sort the two halves
    /* Implement me */


    // merge the sorted havles into list
    /* Implement me */
}
```

mergeSort2 code (version 1) completed

```
//sub-optimal merge sort
void mergeSort2(vector<int>& list){
    if(list.size() <= 1) return; //base case

    // copy list into two halves (left and right)
    vector<int> left(list.size()/2); // n/2 array instantiation
    vector<int> right(list.size() - left.size()); // n/2 array instantiation
    std::copy(list.begin(), list.begin() + left.size() ,left.begin());
    std::copy(list.begin() + left.size(), list.end(), right.begin());

    // sort the two halves
    mergeSort2(left);
    mergeSort2(right);

    // merge the sorted havles into list
    merge2(list, left, right);
```

Does it work? How about space complexity?

Copying array at each step is expensive

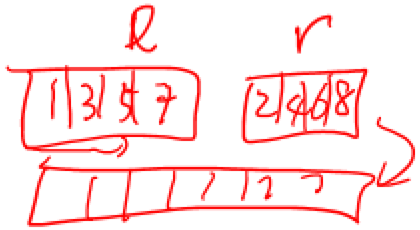


Space complexity is $\Theta(n \log_2 n)$

Additional time needed for copying array - $\Theta(n)$ at each step step

Can we do better?

- For which step we need extra space?
  - Divide the array
  - Sort the left half
  - Sort the right half
  - Merge the sorted left and right arrays into a sorted array



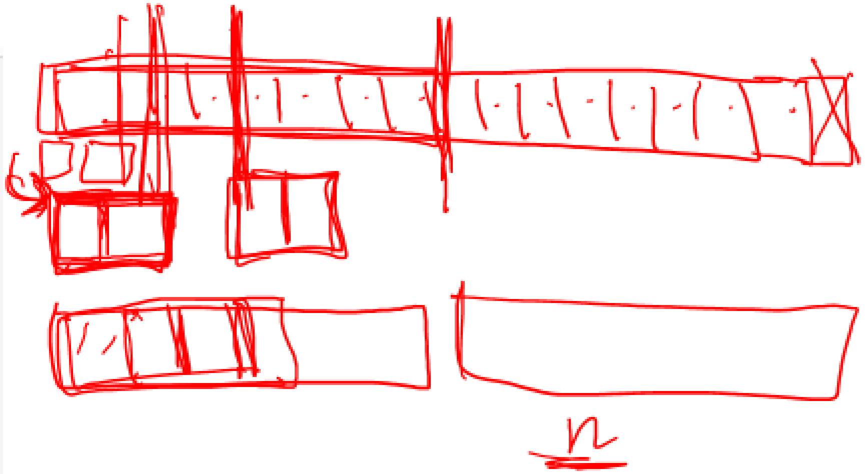
- Where should we put extra array allocation code?

We can move array allocation code just inside merge

$1 \to 2 \to 4 \to 8$

Let's move memory allocation code just inside the merge

```cpp
// Function to merge two sorted subarrays into one sorted array
void merge3(vector<int>& array, int left, int mid, int right) {
    // Calculate the sizes of the two subarrays
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary arrays to hold the elements
    vector<int> L(n1);
    vector<int> R(n2);

    // Copy data to temporary arrays L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = array[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = array[mid + 1 + j];

    // Merge the two temporary arrays back into the original array
    // Code ommited
      ...
    }
}
```




```cpp
// Function to implement merge sort
void mergeSort3(vector<int>& array, int left, int right) {
    if (left >= right) return;
    // Find the middle point
    int mid = left + (right - left) / 2;

    // Recursively sort the two halves
    mergeSort3(array, left, mid);
    mergeSort3(array, mid + 1, right);

    // Merge the sorted halves
    merge3(array, left, mid, right);
}
```

Will this make a difference?

Draw space diagram

We can do better with one auxiliary array

	0	1	2	3	4	5	6	7
org	22	18	12	-4	58	7	31	42

	0	1	2	3	4	5	6	7
aux	22	18	12	-4	58	7	31	42

How?

Better merge sort

```
void mergeSort(vector<int>& list){
    vector<int> copy(list);   // copy of array
    //copy is the source and list is the output
    mergeSort(copy, list, 0, (int) list.size() - 1);
}
```

```
//sort list[start:end] and save to result[start:end]
//start and end is inclusive
void mergeSort(vector<int>& list, vector<int>& result, int start, int end) {
    if (end - start <= 0) return; // base case (size 1 or 0 is already sorted)
    int mid = (start + end) / 2;

    // sort the two halves
     /* Implement me */


    // merge the sorted halves into a sorted whole
     /* Implement me */
}
```

Better merge sort

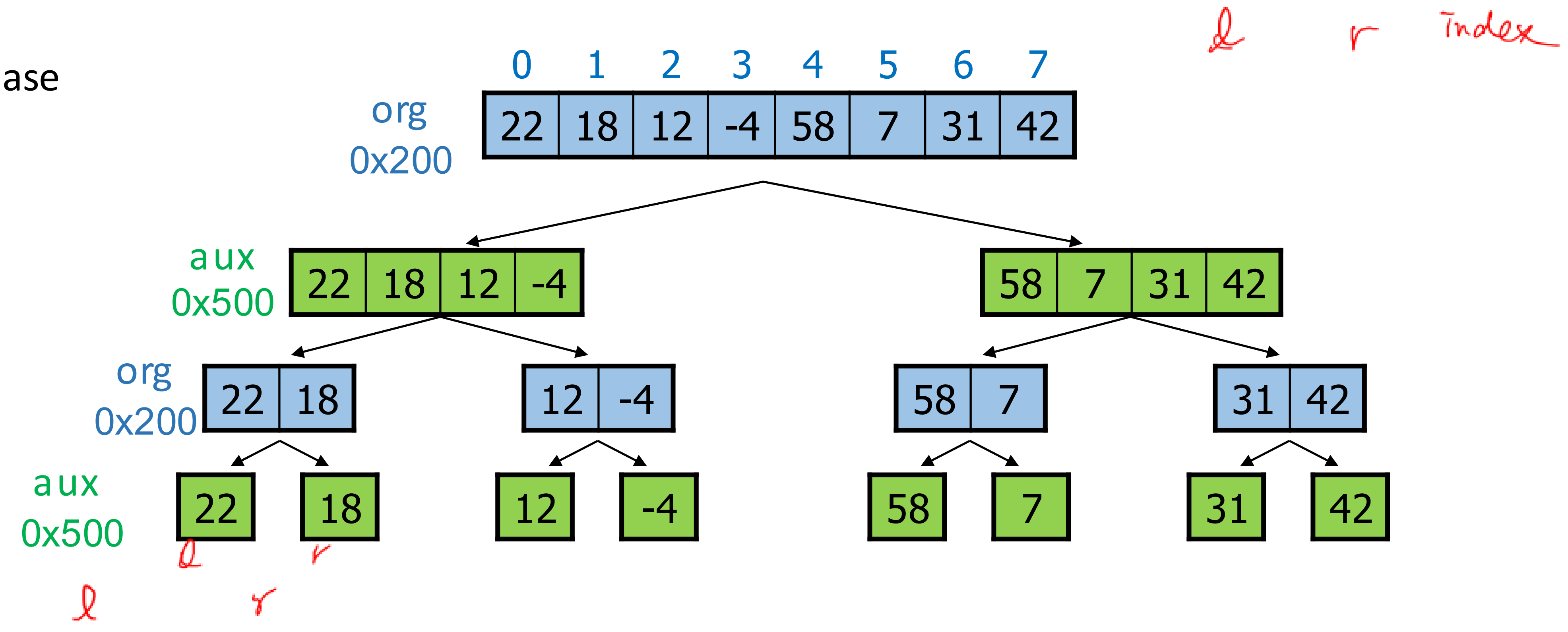
```
void mergeSort(vector<int>& list){
    vector<int> copy(list);  // copy of array
    //copy is the source and list is the output
    mergeSort(copy, list, 0, (int) list.size() - 1);
}
```

```
//sort list[start:end] and save to result[start:end]
//start and end is inclusive
void mergeSort(vector<int>& list, vector<int>& result, int start, int end) {
    if (end - start <= 0) return; // base case (size 1 or 0 is already sorted)
    int mid = (start + end) / 2;

    // sort the two halves
    mergeSort(result, list, start, mid); //sort left and save to list
    mergeSort(result, list, mid + 1, end); //sort right and save to list

    // merge the sorted halves into a sorted whole
    merge(list, result, start, mid, mid + 1, end); //save the merged to result
}
```

Why do we alternate between list and result?

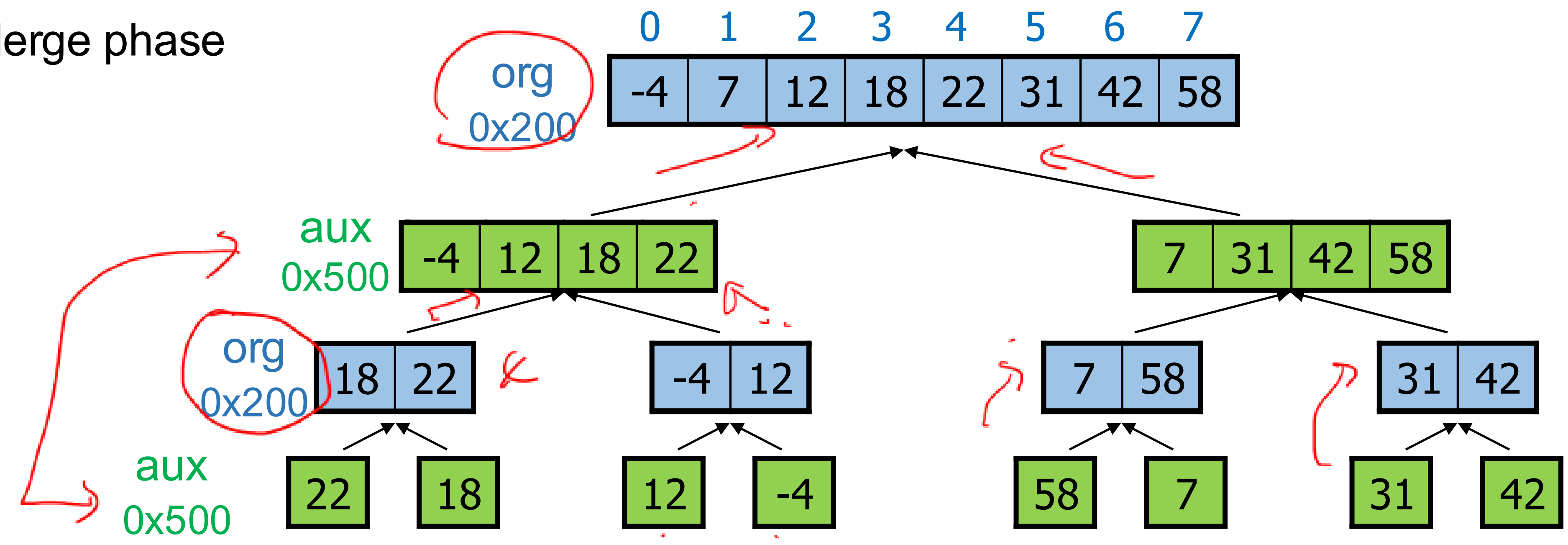
Alternating org and aux achieves space complexity of Θ(n)

Split phase



Alternating org and aux achieves space complexity of Θ(n)

Merge phase



Modified merge - takes two lists of original length

```
// Pre-condition: list[s1:e1] (left) and list[s2:e2] (right) are sorted
// Note: s1, e1, s2, e2 are inclusive and are valid indices
// Merges the left and the right into a sorted result[s1:e2].
void merge(const vector<int>& list, vector<int>& result, int s1, int e1, int s2, int e2) {
    int i1 = s1;   // index of left array
    int i2 = s2;   // index of right array
    for (int i = s1; i <= e2; i++) {
        if (i1 <= e1 &&
            (i2 > e2 || list[i1] <= list[i2] )) {
            result[i] = list[i1];   // take from left
            i1++;
        } else {
            result[i] = list[i2];    // take from right
            i2++;
        }
    }
}
```

Time complexity

$\Theta(n)$

```
void mergeSort(vector<int>& list){
    vector<int> copy(list);  // copy of array
    //copy is the source and list is the output
    mergeSort(copy, list, 0, (int) list.size() - 1);
}
```

$\Theta(1)$

$??$

$\Theta(n)$

```
//sort list[start:end] and save to result[start:end]
//start and end is inclusive
void mergeSort(vector<int>& list, vector<int>& result, int start, int end) {
    if (end - start <= 0) return; // base case (size 1 or 0 is already sorted)
    int mid = (start + end) / 2;

    // sort the two halves
    mergeSort(result, list, start, mid); //sort left and save to list
    mergeSort(result, list, mid + 1, end); //sort right and save to list

    // merge the sorted halves into a sorted whole
    merge(list, result, start, mid, mid + 1, end); //save the merged to result
}
```

Recurrence for merge sort

- Let T(n) be the time complexity of merge sort with n elements

$$T(n) = \begin{cases} \Theta(1) & \text{if } (n = 0) \\ \Theta(1) & \text{if } (n = 1) \\ \Theta(n) + 2\,T(\frac{n}{2}) & \text{otherwise} \end{cases}$$

How to solve this recurrence?

General strategies for solving recurrence

Ex) $T(n) = \begin{cases} 1 & \text{if } (n = 0 \text{ or } 1) \\ 1 + T(n-1) & \text{otherwise} \end{cases}$

$n-1 \quad n-2$

$T(n) = 1 + T(n-1) =$

$= 1 + 1 + T(n-2)$

$= 1 + 1 + 1 + T(n-3) = (n-1) +$

$= \underbrace{\boxed{1 + \cdots + 1}}_{n} + \underbrace{\cancel{T(0)}}_{n-n}$ 0

$O(n)$

- By substitution
  - substitute T(k) with equation containing T(k-1) and simplify
- By telescoping
  - manipulate equation to cancel out!

$$\left.\begin{aligned} \boxed{T(n)} &= 1 + \cancel{T(n-1)} \\ \cancel{T(n-1)} &= 1 + \cancel{T(n-2)} \\ \cancel{T(n-2)} &= 1 + \cancel{T(n-3)} \\ +)\ \cancel{T(2)} &= 1 + T(1) \end{aligned}\right\} n-1$$

$T(n) = (n-1) + 1 = O(n)$

Solve recurrence for merge sort

- Let T(n) be the time complexity of merge sort with n elements

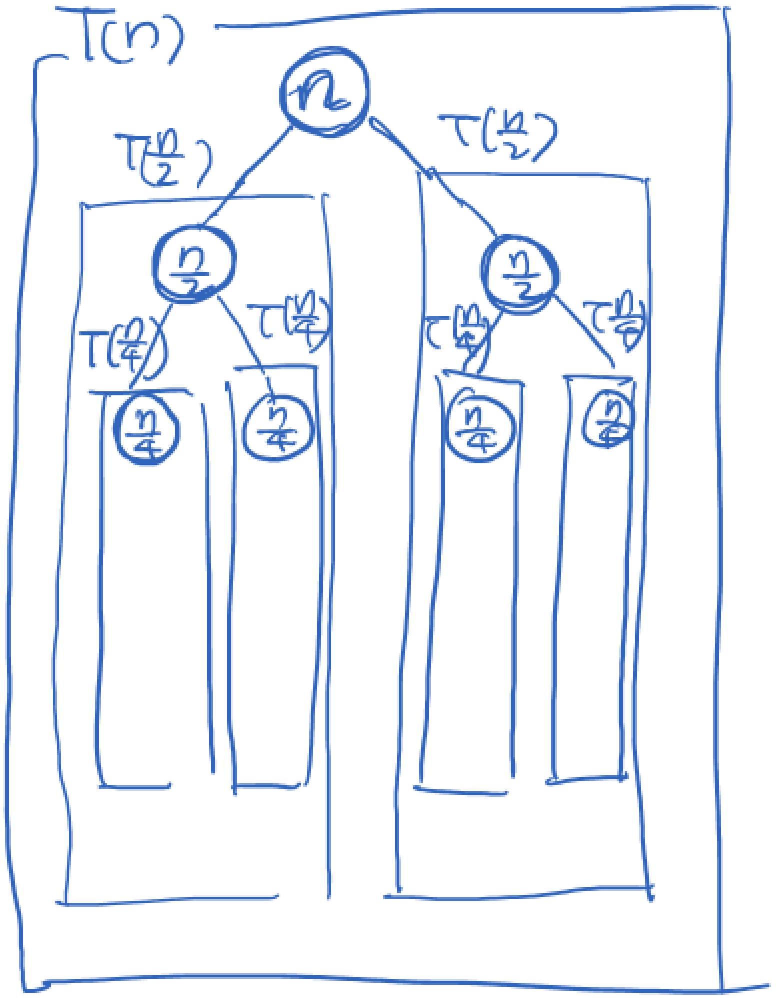
$$T(n) = \begin{cases} \Theta(1) & \text{if } (n = 0) \\ \Theta(1) & \text{if } (n = 1) \\ \Theta(n) + 2\,T(\frac{n}{2}) & \text{otherwise} \end{cases}$$

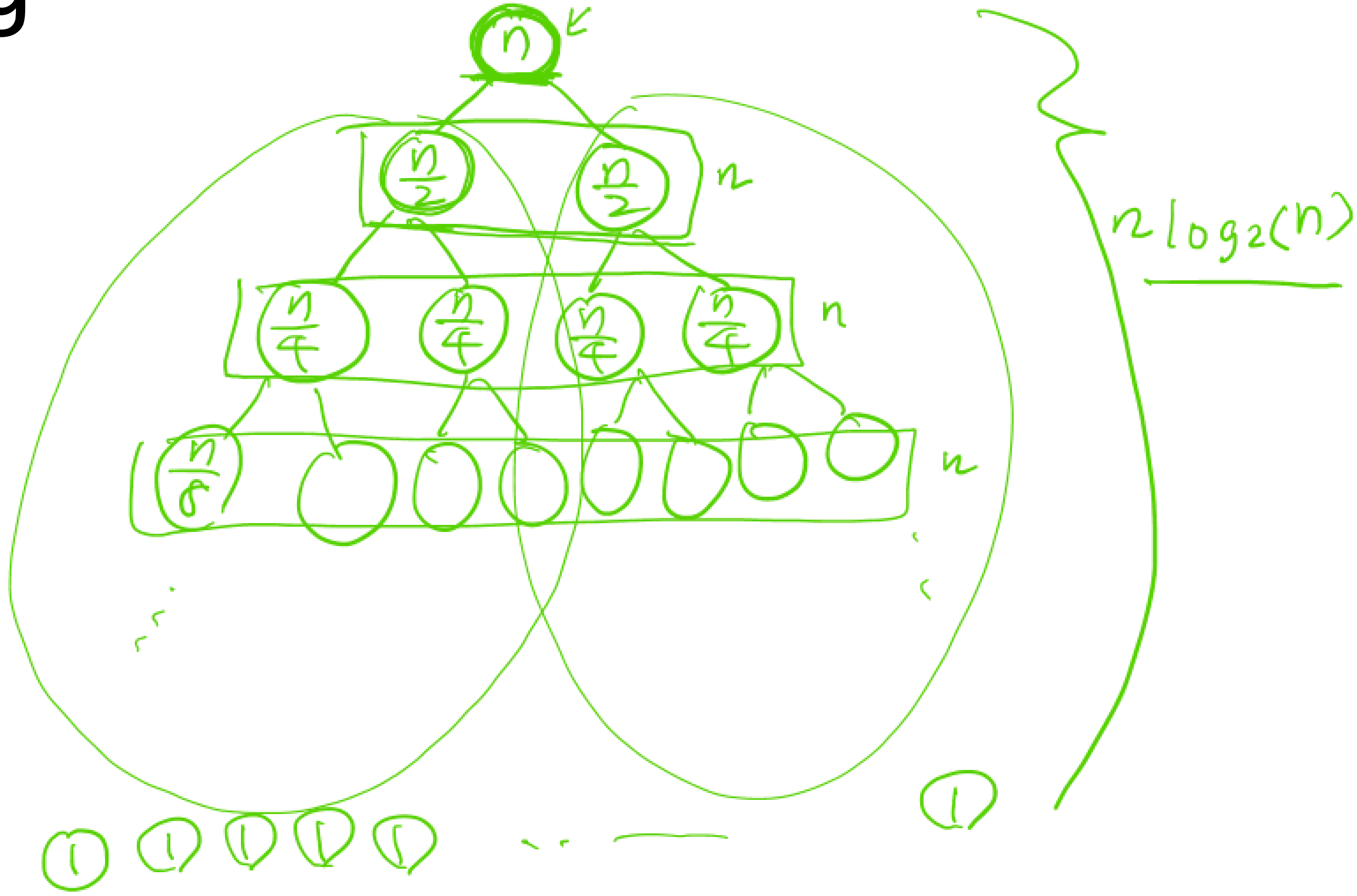
What strategy would you use?

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

art

By ~~t~~elescoping





By ~~art~~ telescoping

$$T(n) = 2T\left(\frac{n}{2}\right) + n \quad \cdot\frac{1}{n}$$

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

$$\frac{T(\frac{n}{2})}{\frac{n}{2}} = \frac{T(\frac{n}{4})}{\frac{n}{4}} + 1$$

$$\frac{T(\frac{n}{4})}{\frac{n}{4}} = \frac{T(\frac{n}{8})}{\frac{n}{8}} + 1$$

$$\vdots$$

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

$\log n$

$+)$

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

$$\frac{T(n)}{n} = \log(n)$$

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

Solving recurrence

- $T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + n$
- Via telescoping
- Via substitution
- Via art(?)

$$T(n) = 2T(\tfrac{n}{2}) + n \quad \times \tfrac{1}{n}$$

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

$$\frac{\cancel{T(\frac{n}{2})}}{\cancel{\frac{n}{2}}} = \frac{\cancel{T(\frac{n}{4})}}{\cancel{n/4}} + 1$$

$$\frac{\cancel{T(\frac{n}{4})}}{\cancel{\frac{n}{4}}} = \frac{\cancel{T(\frac{n}{8})}}{\cancel{n/8}} + 1$$

$\vdots$

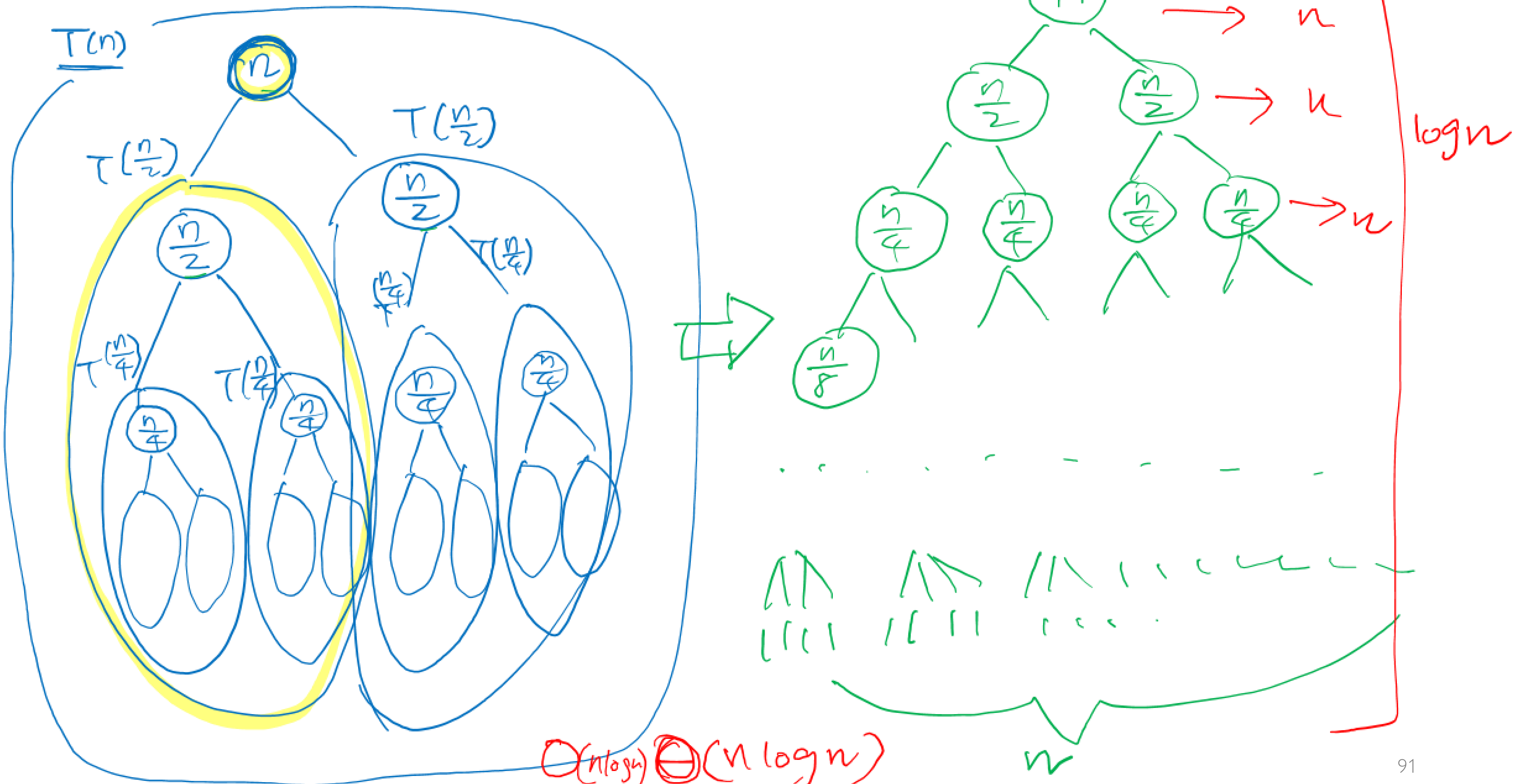
$$\frac{\cancel{T(2)}}{\cancel{2}} = \frac{T(1)}{1} + 1$$

$\log_2 n$

$$\frac{T(n)}{n} = \log n$$

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

$$T(n) = 2T(\tfrac{n}{2}) + n$$



$O(n \log n)$ $\Theta(n \log n)$

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

$T(\frac{n}{2}) = 2T(\frac{n}{4}) + \frac{n}{2}$

$\vdots$

$T(n) = 2\{2T(\frac{n}{4}) + \frac{n}{2}\} + n$

$= \ldots\ldots$

$\vdots$

$T(1)$

$T(n) = T(n-1) + n$

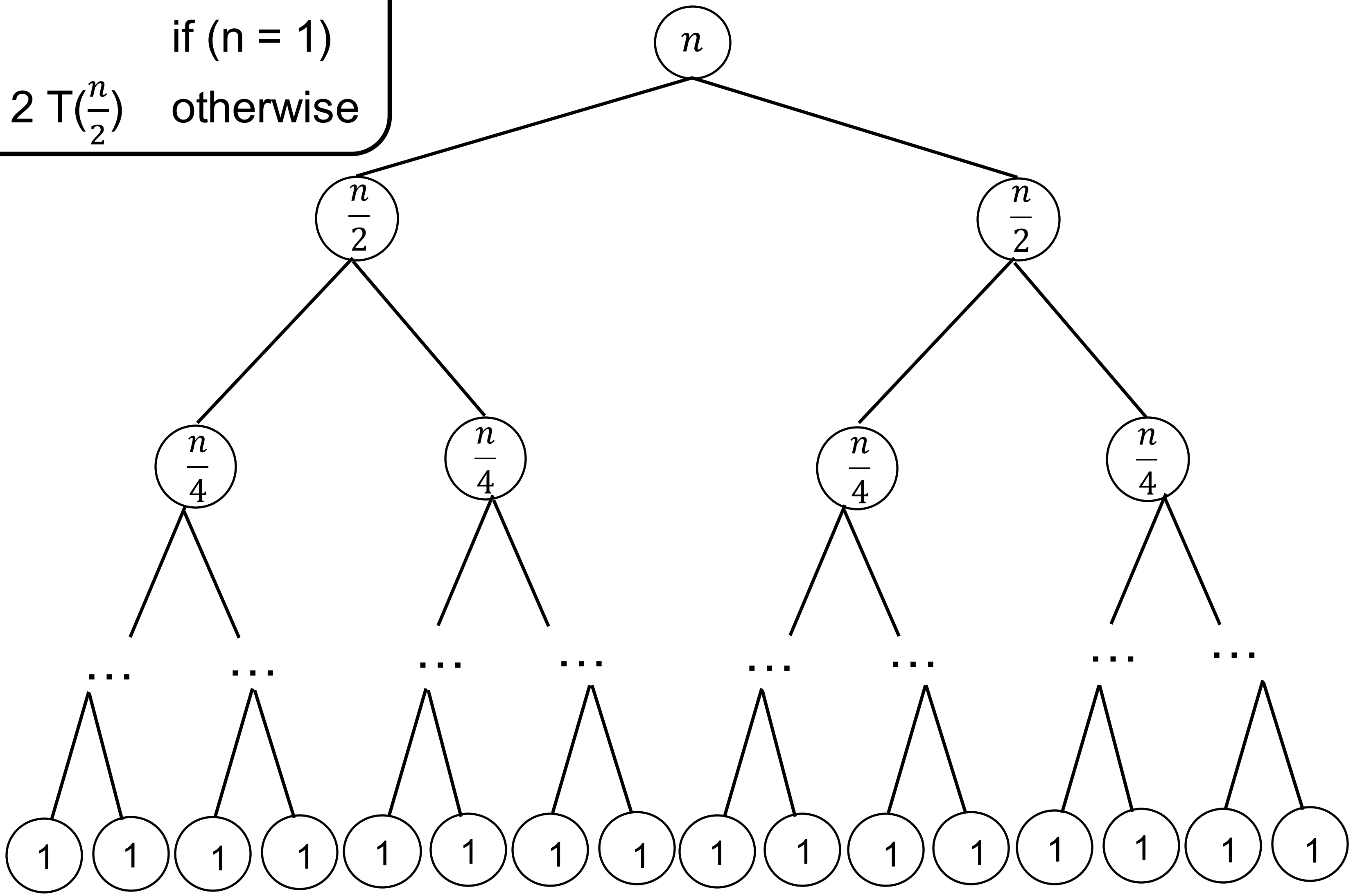
$T(n) = T(n-2) + n-1 + n$

$= T(n-3) + n-2 + n-1 + n$

Substitution can work better linear...

$$T(n) = \begin{cases} \Theta(1) & \text{if } (n = 0) \\ \Theta(1) & \text{if } (n = 1) \\ \Theta(n) + 2\,T(\frac{n}{2}) & \text{otherwise} \end{cases}$$

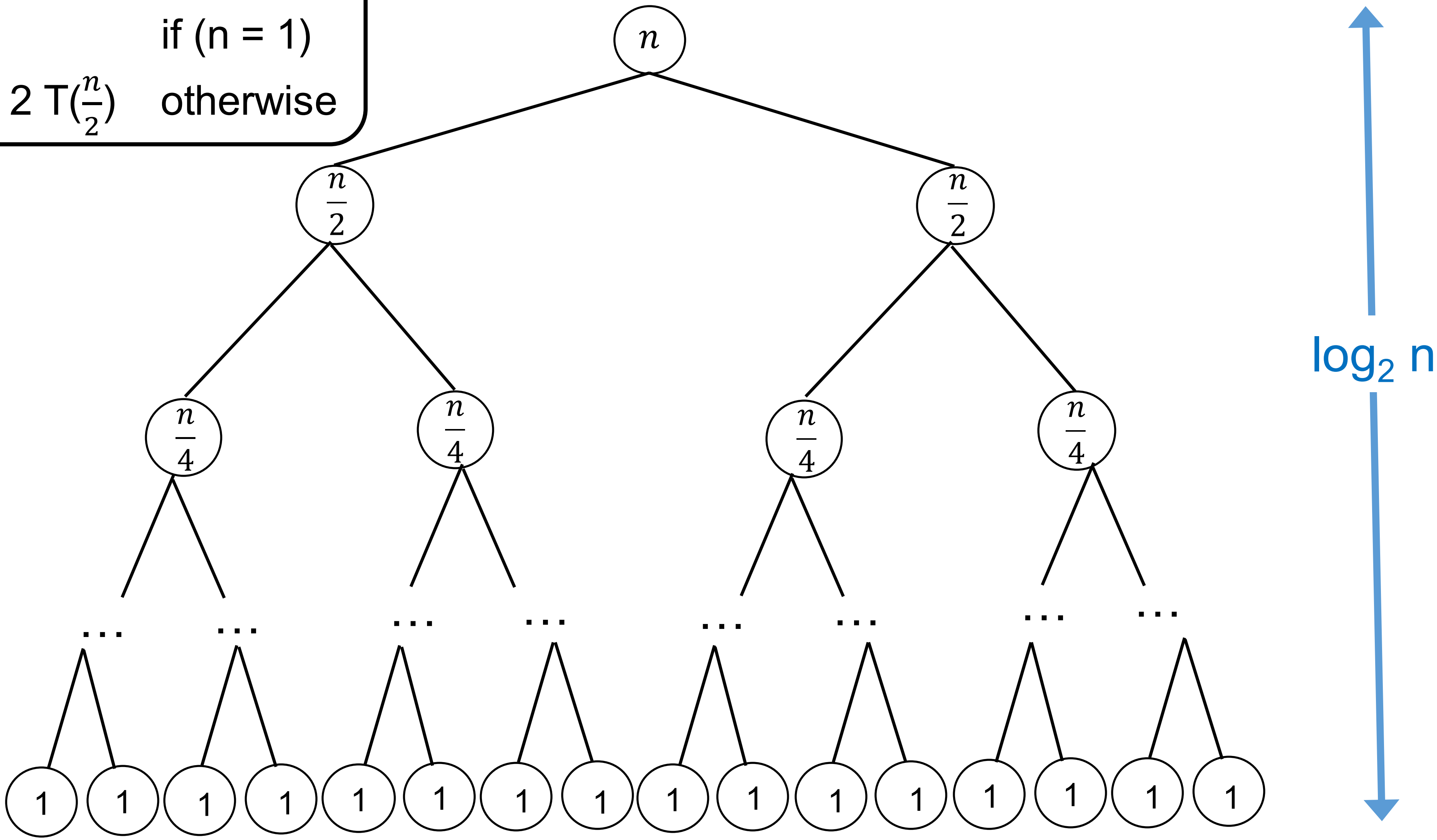
Each node represents Θ



T(n) is the sum of all work done by each node

$$T(n) = \begin{cases} \Theta(1) & \text{if } (n = 0) \\ \Theta(1) & \text{if } (n = 1) \\ \Theta(n) + 2\,T(\frac{n}{2}) & \text{otherwise} \end{cases}$$

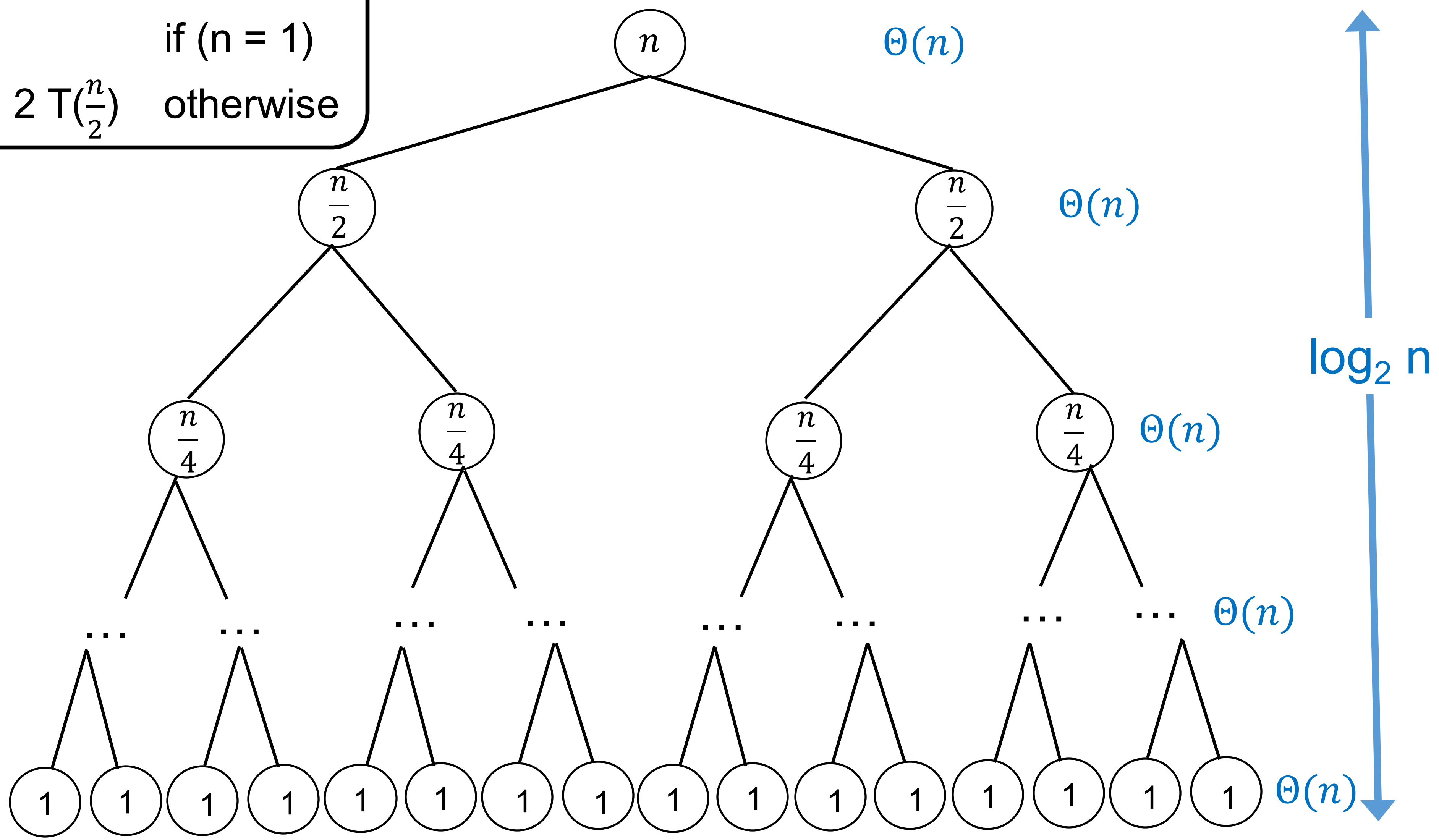
Each node represents Θ



What is the amount of work done at each level?

$$T(n) = \begin{cases} \Theta(1) & \text{if } (n = 0) \\ \Theta(1) & \text{if } (n = 1) \\ \Theta(n) + 2\,T(\frac{n}{2}) & \text{otherwise} \end{cases}$$

Each node represents Θ



Each level does $\Theta(n)$ work. Therefore $T(n) \in \Theta(n \log_2 n)$

Is merge sort stable?

Is merge stable?

- Which 25 to pick to put into result?

left

0	1	2	3	4
20	25_1	..	..	..

right

0	1	2	3	4
25_2	..	..	..	..

result

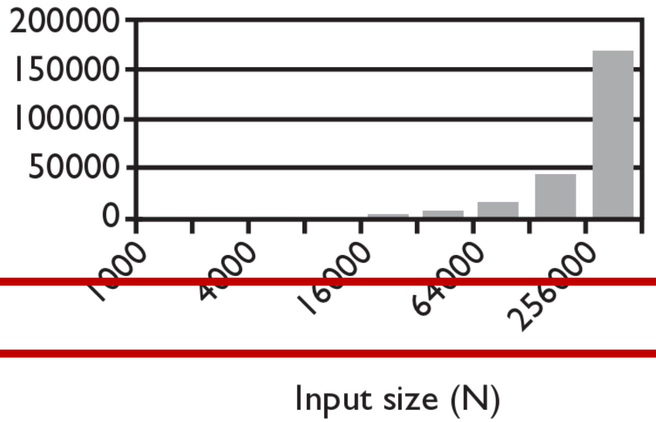
0	1	2	3	4	5	6	7	8	9
20	25_1	25_2	..	..	..	..	..	..	..

Choose from left when there is a tie

Merge sort has significantly lower runtime than selection sort

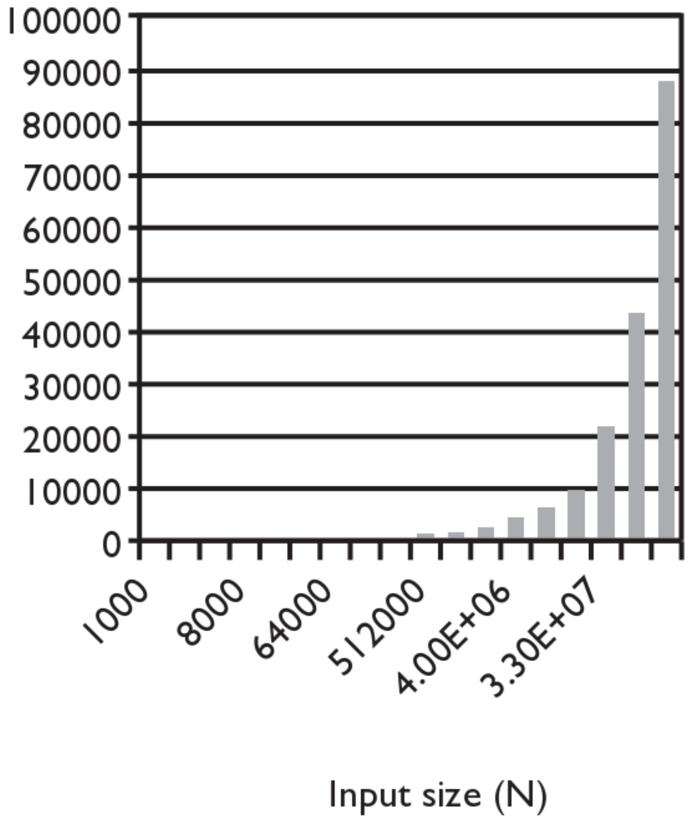
Selection Sort

N	Runtime (ms)
1000	0
2000	16
4000	47
8000	234
16000	657
32000	2562
64000	10265
128000	41141
256000	164985



Merge Sort

N	Runtime (ms)
1000	0
2000	0
4000	0
8000	0
16000	0
32000	15
64000	16
128000	47
256000	125
512000	250
1e6	532
2e6	1078
4e6	2265
8e6	4781
1.6e7	9828
3.3e7	20422
6.5e7	42406
1.3e8	88344



Improving merge sort

- Which data structure instead of array can achieve space complexity of $\Theta(1)$?

- Can we make merge sort even faster?

Improving merge sort

- Merge sort on linked list achieves space complexity of $\Theta(1)$
  - But linked list requires additional pointer per data
- Parallel merge sort!
  - We can parallelize sorting left and right as they are independent
  - We can even parallelize merge! How?

Parallelizing merge!

- A = [1, 2, 3, 4, 9, 13], N = 6
- B = [5, 6, 7, 8, 10, 11], M = 6
- Assume 2 threads: each merge 6 items
  - Thread 1 should merge first smallest 6
  - Thread 2 should merge remaining 6
- How to find smallest k items from two sorted arrays? (k = $\frac{N+M}{2}$)

What makes the smallest k items? (Defining partition)

How to find such partition efficiently? (Determining partition)

Defining partition

- A = [1, 2, 3, 4, 9, 13], N = 6
- B = [5, 6, 7, 8, 10, 11], M = 6
- What are the conditions for i and j?
  -

Why equality?

Determining partition

- A = [1, 2, 3, 4, 9, 13], N = 6
- B = [5, 6, 7, 8, 10, 11], M = 6
- What is better than linear search when things are sorted?

Use binary search!

Outline

1. Intro
2. Vocabs: Loop Invariants and Stable Sorting
3. Primer: Where should max go?
4. Selection Sort
5. Bubble Sort
6. Insertion Sort
7. Merge Sort
8. Quick Sort
9. Conclusion

Quicksort

```
qsort (List S) {
    if (|S| <= 1) return S;                       // Note that S could be empty
    v = element of S;                             // Choose pivot
    // Using set notation for lists
    List L = { x in  S - {v} | x <= v }             // Note x = v goes
    List R = { x in  S - {v} | x >= v }             // in either list
    return (qsort(L) + {v} + qsort (R));          // + is list concatenation
}
```

Why not make it deterministic?

Quicksort

```
qsort (List S) {
    if (|S| <= 1) return S;                   // Note that S could be empty
    v = element of S;                        // Choose pivot
    // Using set notation for lists
    List L = { x in  S - {v} | x <= v }        //consider S = {5,5,5,5,5}
    List R = { x in  S - {v} | x > v }
    return (qsort(L) + {v} + qsort (R));   // + is list concatenation
}
```

Such will result in bad partitioning

Correctness proof of qsort algorithm

- Assume qsort works on lists that are smaller than S
- qsort(L) and qsort(R) works because |L| < n and |R| < n
- qsort(L) is sorted and all elements in L are smaller or equal to v
- qsort(R) is sorted and all elements in R are greater or equal to v
- Thus qsort(L) + {v} + qsort(R) is sorted

Why having |L| < n and |R| < n important?

Is it important how we choose pivot?

- What is a bad pivot?

Bad pivot (partition) analysis via telescoping

- $T(n) = T(n-1) + n$
- Why n?

$$
\begin{aligned}
T(n) &= \cancel{T(n-1)} + n \\
\cancel{T(n-1)} &= \cancel{T(n-2)} + n-1 \\
\cancel{T(n-2)} &= \cancel{T(n-3)} + n-2 \\
&\ \vdots \\
&\quad + 3 \\
&\quad + 2 \\
+)\ \cancel{T(1)} &= \underline{T(0)} + 1 \\
\hline
T(n) &= c + \frac{(n+1)\,n}{2} = O(n^2)
\end{aligned}
$$

$n+1$

$n+1$

$n+1$

This is a worst-case analysis

Good partition analysis

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

$$\times \frac{1}{n}$$

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

$$\frac{T(\frac{n}{2})}{\frac{n}{2}} = \frac{T(\frac{n}{4})}{n/4} + 1$$

$$\frac{T(\frac{n}{4})}{\frac{n}{4}} = \frac{T(\frac{n}{8})}{n/8} + 1$$

$$\vdots$$

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

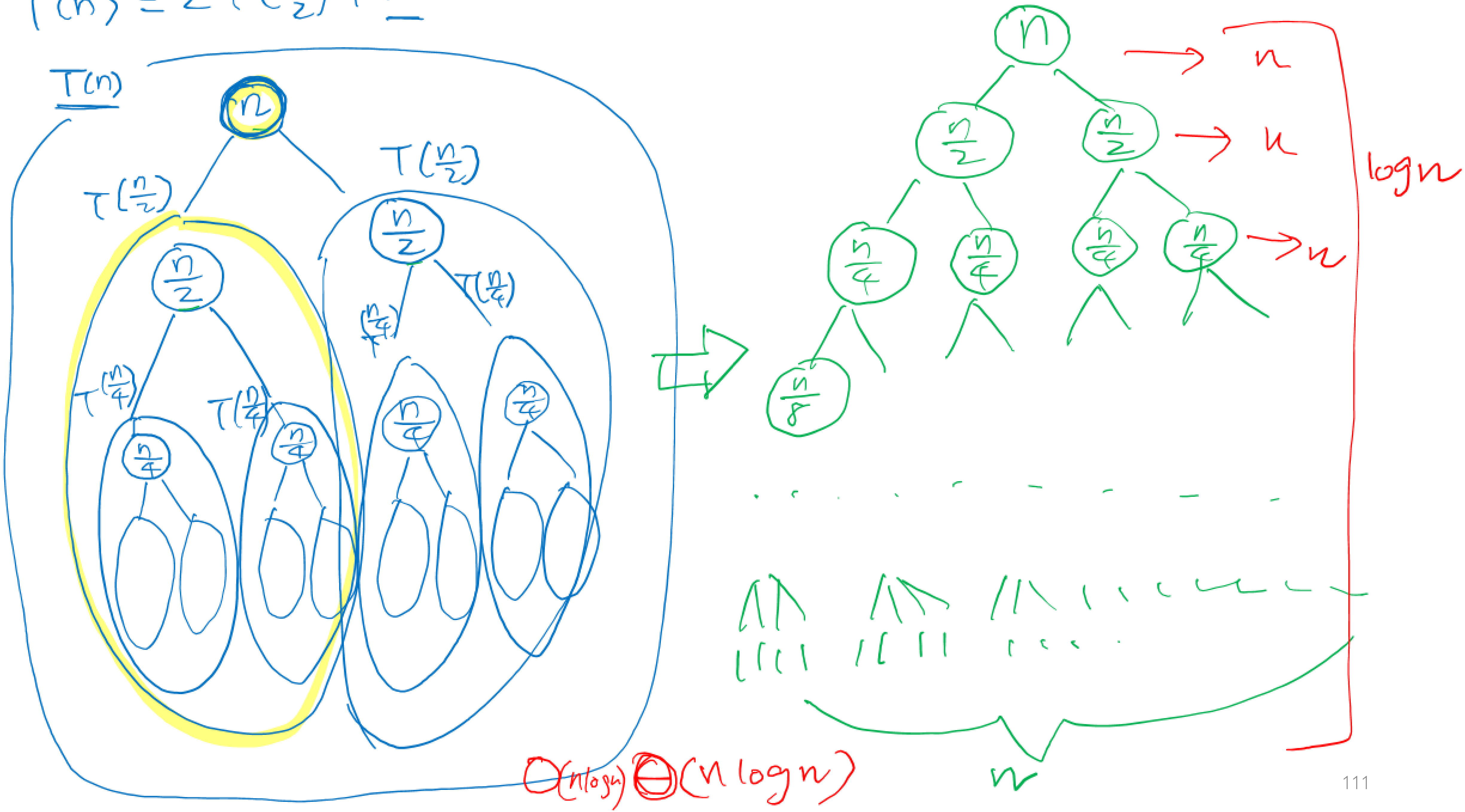
$\log_2 n$

$$\frac{T(n)}{n} = \log n$$

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

- $T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + n$
- Via telescoping
- Via substitution
- Via art(?)

$$T(n) = 2T\left(\frac{n}{2}\right) + \underline{n}$$



$O(n \log n)$ $\Theta(n \log n)$

General results for divide-n-conquer

- If we divide the work two half-size problems + linear additional work
  - What is the linear additional work in quicksort?
- How many times do we need to divide into half?
  - From n to get to 1
  - This is the definition of log n (base 2)
- Thus, we will have O(n log n) algorithm

For the case of quicksort, picking the right pivot is critical

Outline

1. Administrative
2. Recap on Big-O, Big-Omega, Big-Theta
3. Quicksort analysis
4. How to choose pivot?

How to choose pivot?

- Solution 0: Random
- Solution 1: Pick the middle
  - When will this create worst case behavior?
- Solution 2: Pick the real median
  - How can you find median value?
- Solution 3: Pick median of 3 elements
  - Pick a median from leftmost, rightmost, and the middle element
  - Can you come up with a pathological example (worst case)?

Which one is better – random vs median of 3?

Probability of worst case happening

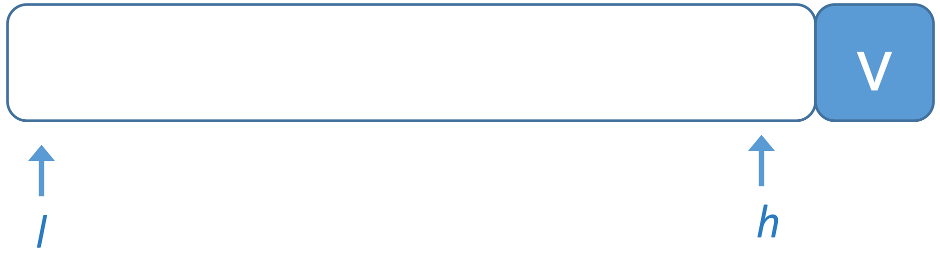
- Random:
  - What is a probability of choosing min or max out of n?
- Median of 3
  - What is the probability of having the two smallest or two largest values in [leftmost, rightmost, middle]?

Outline

1. Administrative
2. Recap on Big-O, Big-Omega, Big-Theta
3. Case study: Quicksort analysis
4. How to choose pivot?
5. Partitioning the list

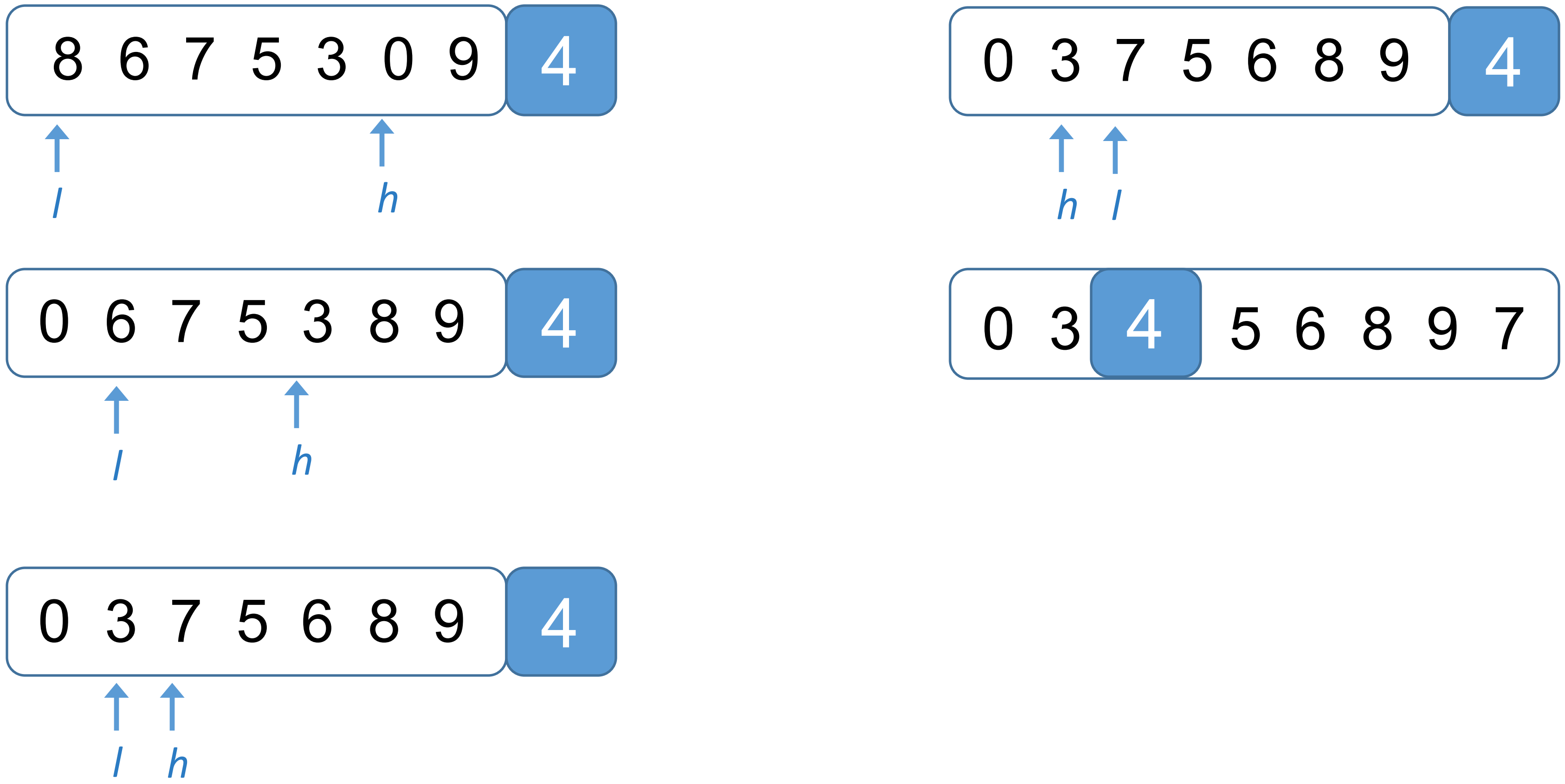
Basic idea

- Put the pivot *v* at the end of the list
- One cursor at the low end *l* and another at the high end *h*.



- Move *l* to the right until it finds L[l] > v
- Move *h* to the left until it finds L[h] < v
- Swap
- Stop when *l* >= *h*
- Restore pivot to *l*

Example



Stop since *h* and crossed over and restore pivot to *l*

What if

5	5	5	5	5	5	5	5

Do we not swap? Or still swap?

Skip to swap:
looking for strictly larger/smaller than pivot

$5_1\ 5_2\ 5_3\ 5_4\ 5_5\ 5_6\ 5_7$ 5_v

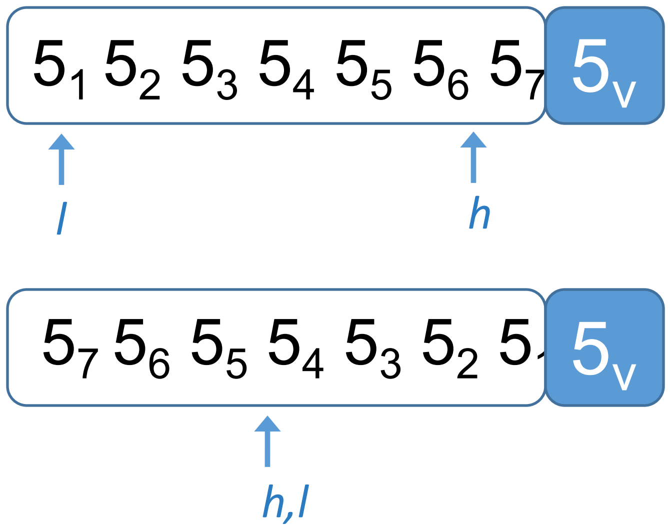
l *h*

$5_1\ 5_2\ 5_3\ 5_4\ 5_5\ 5_6\ 5_7$ 5_v

h *l*

Results in bad partitioning. R is empty.

Yes, to swap!



Results in even partition

In the actual implementation, we can skip the swapping but let h, l progress as shown above

Code

```
l = low;
h = high-1;
while (1)
{
    while (a[l] < pivot)
        l++;
    while (a[h] > pivot)
        h--;
    if (a[l] == a[h])
    {
        l++;
        h--;
        continue;
    }
    if (l>=h)
        break;
    swap(a[l], a[h]);
}
```

Use < not <=

Use > not >=

The analysis has guided the details of the algorithm development

How to make quicksort even quicker?

- Parallelize! How?
- If there are many duplicates…

3 way-partitioning Quicksort

```
qsort (List S) {
    if (|S| <= 1) return S;              // Note that S could be empty
    v = element of S;                    // Choose pivot
    // Using set notation for lists
    List L = { x in S - {v} | x < v }
    List R = { x in S - {v} | x > v }
    return (qsort(L) + E{all elements equal to v} + qsort (R));
}
```

How to implement this algorithm?

We have 3 partitions in this version
So, we should maintain 3 pointers!

Pointers: `lt`, `i`, `gt`

Pointer	Meaning
`lt`	Boundary of `< pivot` region
`i`	Current element being inspected
`gt`	Boundary of `> pivot` region

We have 3 partitions in this version
So, we should maintain 3 pointers!

Pointers: `lt`, `i`, `gt`

Pointer	**Meaning**
`lt`	End of "less than pivot" region (`arr[low..lt-1] < pivot`)
`i`	Current element being inspected (`arr[lt..i-1] == pivot`)
`gt`	Start of "greater than pivot" region (`arr[gt+1..high] > pivot`)

We start at i = lt , we stop at i > gt

Complete the given code in the notes

Outline

1. Intro
2. Vocabs: Loop Invariants and Stable Sorting
3. Primer: Where should max go?
4. Selection Sort
5. Bubble Sort
6. Insertion Sort
7. Merge Sort
8. Conclusion

In conclusion

- Sorting reduces the complexity of problems
- Various sorting algorithms exist each with unique traits

	Time Complexity	Space Complexity	Stable?
Selection	$\Theta(n^2)$	$\Theta(1)$	Y
Bubble	$\Theta(n^2)$	$\Theta(1)$	Y
Insertion	$\Theta(n^2)$	$\Theta(1)$	Y
Merge sort	$\Theta(n \log n)$	$\Theta(n)$	Y
Quick sort	$\Theta(n \log n)$	$\Theta(n)$	Y

- Lecture example code available
  https://github.com/mikiehan/sorting-example.git