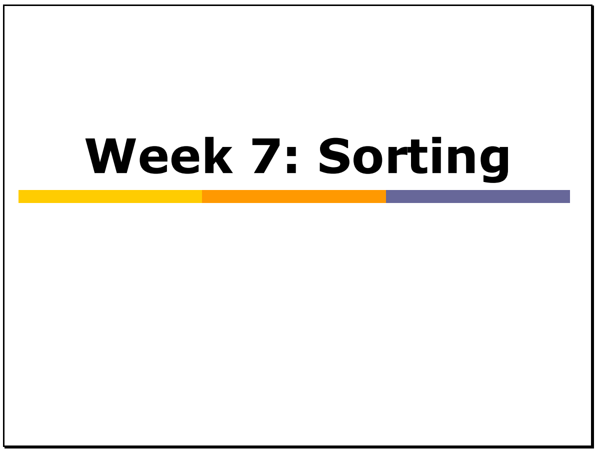
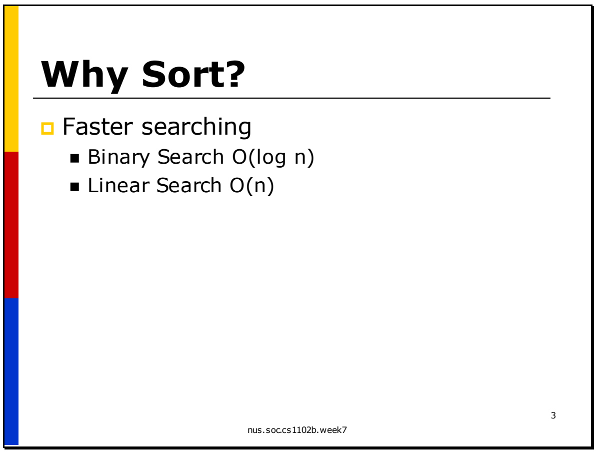


Week 7: Sorting

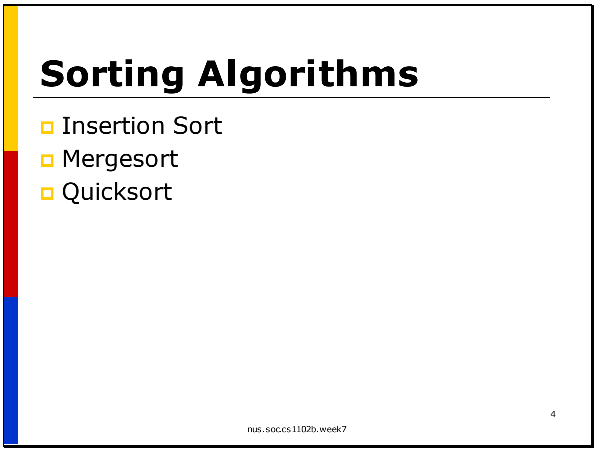


Why Sort?

- Faster searching
  - Binary Search O(log n)
  - Linear Search O(n)

nus.soc.cs1102b.week7 3

- Recall that searching in a sorted array using binary search is much faster than searching in an unsorted array.
- There are many examples in our daily lives, where things are sorted according to some order.
  - Apartments/Room numbers
  - Dictionary
  - Phonebook
  - Books in the library
  - Calendar

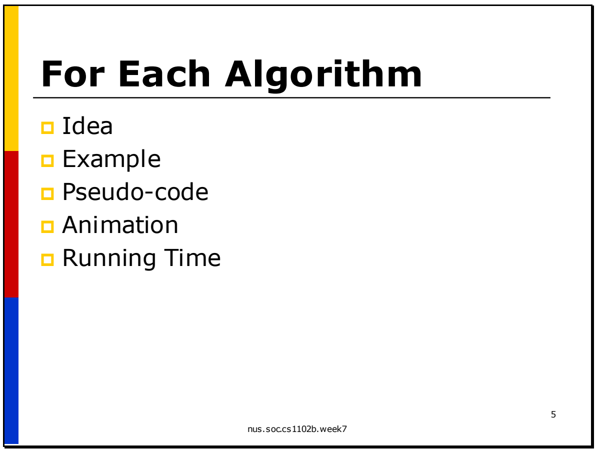


Sorting Algorithms

- Insertion Sort
- Mergesort
- Quicksort

nus.soc.cs1102b.week7 4

- We will look at three different sorting algorithms: insertion sort, Mergesort and Quicksort.

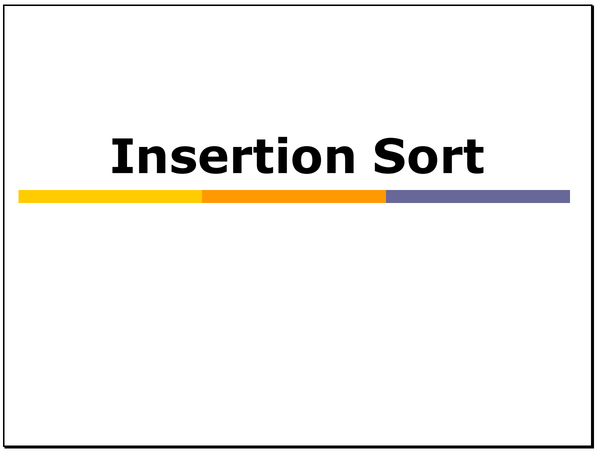


For Each Algorithm

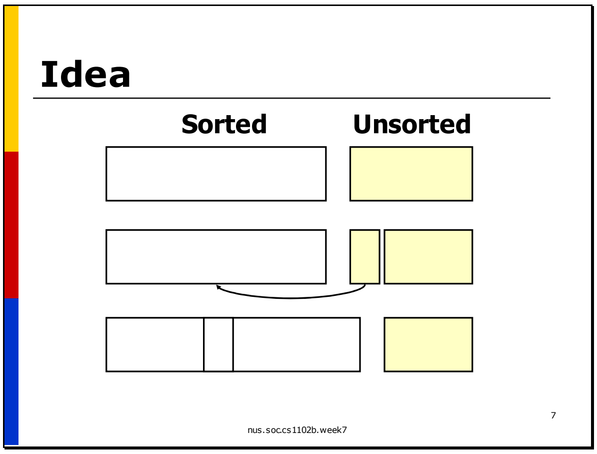
- Idea
- Example
- Pseudo-code
- Animation
- Running Time

nus.soc.cs1102b.week7 5

- This lecture will be organized as follows. For each of the sorting algorithm, I will
  - Give you an idea about how the algorithm works
  - Give you an example
  - Show you some pseudo-code (part English, part Java)
  - Show you an animation of the sorting algorithm
  - Analyze the running time



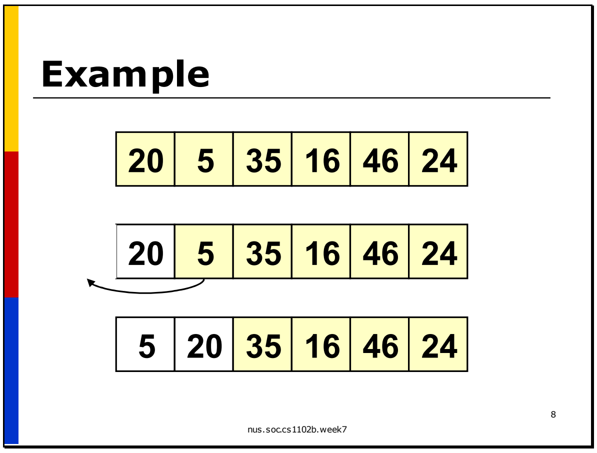
Insertion Sort



Idea

nus.soc.cs1102b.week7 7

- The inputs are partitioned into two parts, one sorted, and one unsorted.
- We take the first element from the unsorted partition, and insert it into the sorted partition (maintaining the sorted order).
- Repeat until there are no more elements in the unsorted portion.



Example

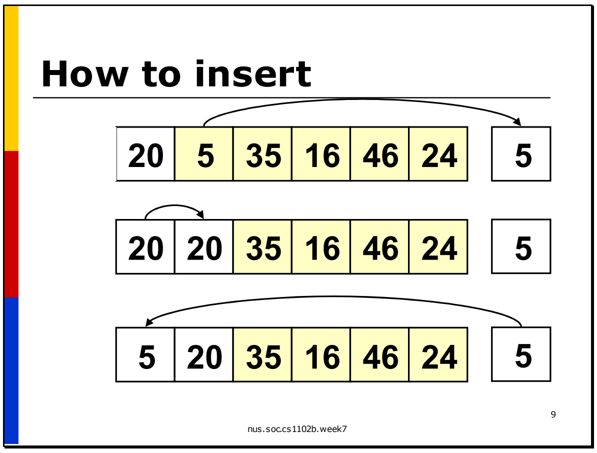
20 5 35 16 46 24

20 5 35 16 46 24

5 20 35 16 46 24

nus.soc.cs1102b.week7 8

- Consider this input. First, we partition it to two parts of size 1 and size N-1 (where N is the size of the input).
- A partition of size 1 is already sorted!
- Now pick the first element (5) from the unsorted portion. Move it into the sorted part (maintaining the sorted order).



How to insert

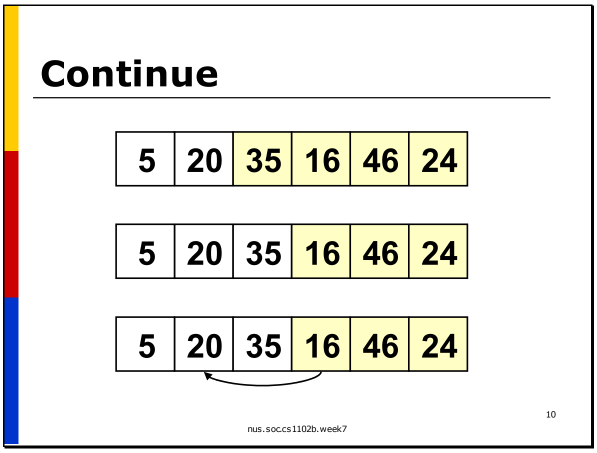
20 5 35 16 46 24 5

20 20 35 16 46 24 5

5 20 35 16 46 24 5

nus.soc.cs1102b.week7 9

- Inserting into sorted linked list is easy. If the input is an array, which is the case here, it is more tedious. We will have to shift the elements to make way for the inserted element.
- We first store 5 in a variable
- Start from the end of the sorted partition, compare 5 with each element.
- If 5 is smaller than the current element, we shift the element one slot to the right.
- If 5 is larger than the current element, we insert 5 to the right of the current element. Done.
- When we reach the beginning of the sorted portion. We insert 5 at the first slot. Done.

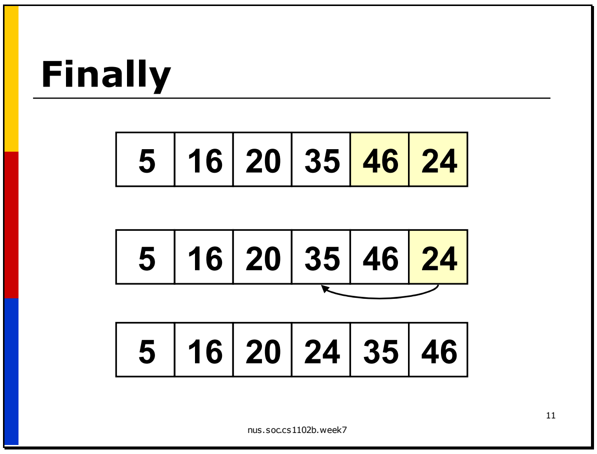


Continue

5	20	35	16	46	24
5	20	35	16	46	24
5	20	35	16	46	24

nus.soc.cs1102b.week7 10

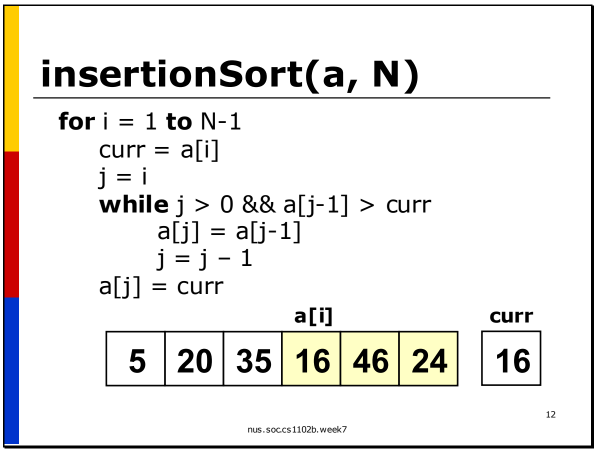
- Now the sorted partition grown to two elements.
- The next element to consider is 35. Since 35 is larger than 20, it is already in its sorted position. The sorted partition grown to 5, 20 and 35.
- The algorithm continues, until all elements are sorted.



Finally

5	16	20	35	46	24
5	16	20	35	46	24
5	16	20	24	35	46

nus.soc.cs1102b.week7 11



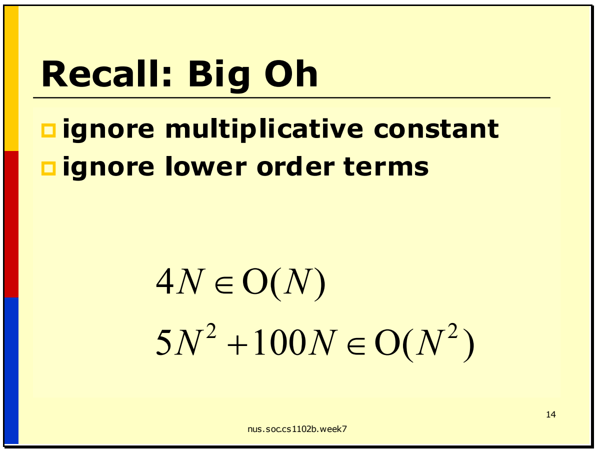
insertionSort(a, N)

```
for i = 1 to N-1
    curr = a[i]
    j = i
    while j > 0 && a[j-1] > curr
        a[j] = a[j-1]
        j = j – 1
    a[j] = curr
```

			a[i]			curr
5	20	35	16	46	24	16

nus.soc.cs1102b.week7 12

- Note that we start from the 2nd element (a[1], not a[0]).
- Pseudo-code is not Java, but part Java, part English.
- During exam, if you are asked to implement or write Java code, you cannot write pseudo-code. But if you are asked to describe an algorithm, you may use pseudo-code.



Recall: Big Oh

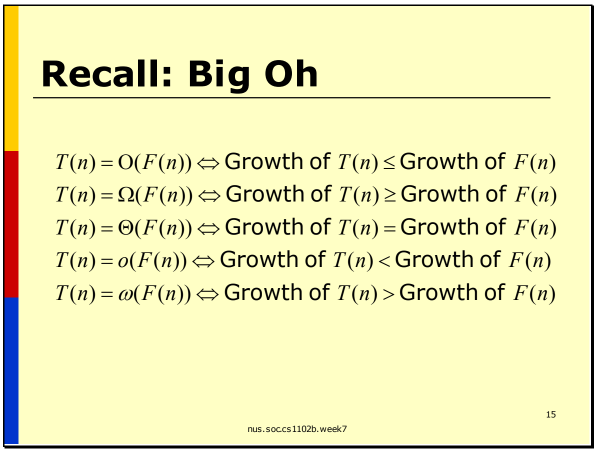
- ignore multiplicative constant
- ignore lower order terms

$$4N \in O(N)$$

$$5N^2 + 100N \in O(N^2)$$

nus.soc.cs1102b.week7 14

- Now we will analyze the running time of insertion sort using Big O notation. Recall that in Big O notation, we are more interested in the "order of magnitude", so we can ignore multiplicative constant and lower order terms.



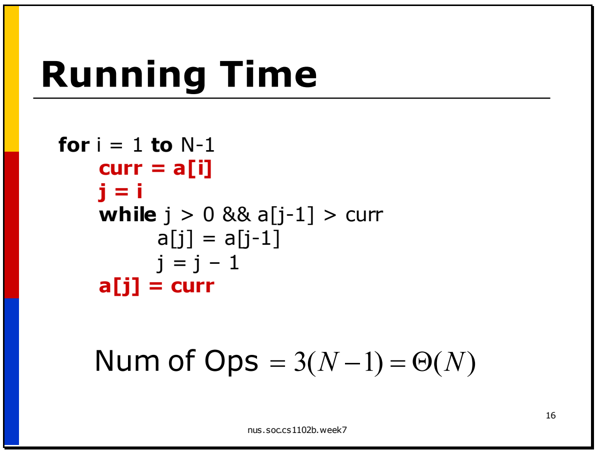
Recall: Big Oh

$$T(n) = \mathrm{O}(F(n)) \Leftrightarrow \text{Growth of } T(n) \le \text{Growth of } F(n)$$
$$T(n) = \Omega(F(n)) \Leftrightarrow \text{Growth of } T(n) \ge \text{Growth of } F(n)$$
$$T(n) = \Theta(F(n)) \Leftrightarrow \text{Growth of } T(n) = \text{Growth of } F(n)$$
$$T(n) = o(F(n)) \Leftrightarrow \text{Growth of } T(n) < \text{Growth of } F(n)$$
$$T(n) = \omega(F(n)) \Leftrightarrow \text{Growth of } T(n) > \text{Growth of } F(n)$$

15

nus.soc.cs1102b.week7

- Recall the different notation for algorithm analysis. We are more interested in Big Oh, which is an upper bound of growth rate, and Big Omega, which is the lower bound of growth rate, and Big Theta, which indicates the same growth rate.



Running Time

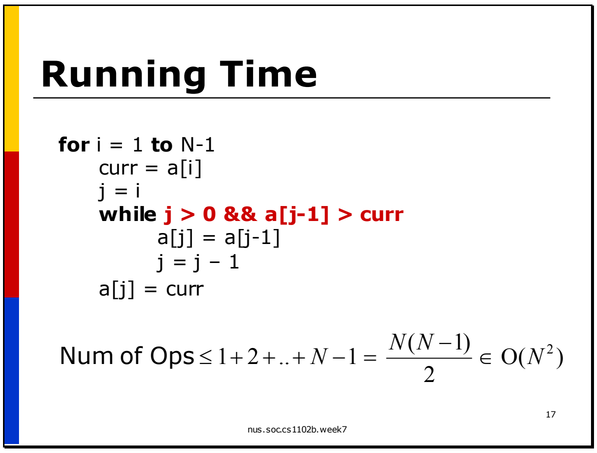
```
for i = 1 to N-1
    curr = a[i]
    j = i
    while j > 0 && a[j-1] > curr
        a[j] = a[j-1]
        j = j - 1
    a[j] = curr
```

Num of Ops $= 3(N-1) = \Theta(N)$

16

nus.soc.cs1102b.week7

- Let’s look at the outer loop first, the outer loop has three operations, and are executed N-1 times. So the total running time is Θ(N).



Running Time

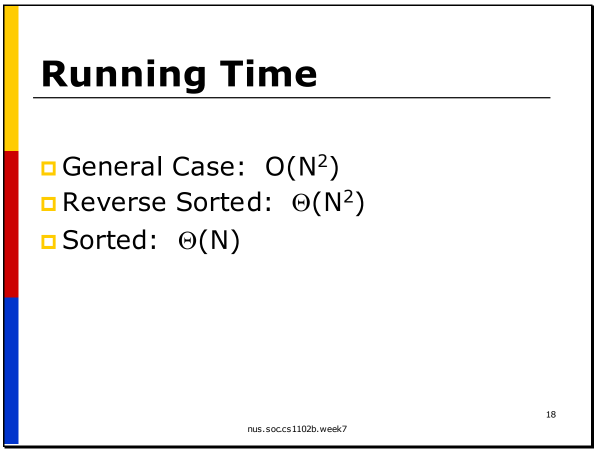
```
for i = 1 to N-1
    curr = a[i]
    j = i
    while j > 0 && a[j-1] > curr
        a[j] = a[j-1]
        j = j - 1
    a[j] = curr
```

Num of Ops $\le 1+2+..+N-1 = \frac{N(N-1)}{2} \in \mathrm{O}(N^2)$

17

nus.soc.cs1102b.week7

- The inner loop is executed at most i times, where i is the count of the outer loop. Hence the number of times it is executed is the summation of i, for i = 1 to N-1. This is an arithmetic series, and the value is equal to N(N-1)/2. By ignoring lower terms and multiplicative constant, the running time of the inner loop is O(N^2).
- Question: Why do we use Big-O instead of Big-Θ?



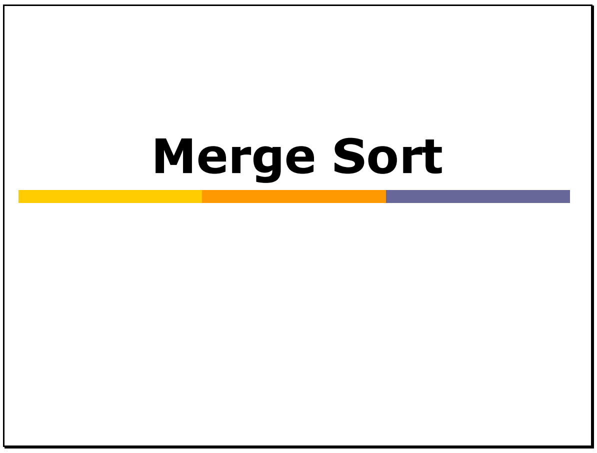
Running Time

- □ General Case: O(N^2)
- □ Reverse Sorted: Θ(N^2)
- □ Sorted: Θ(N)

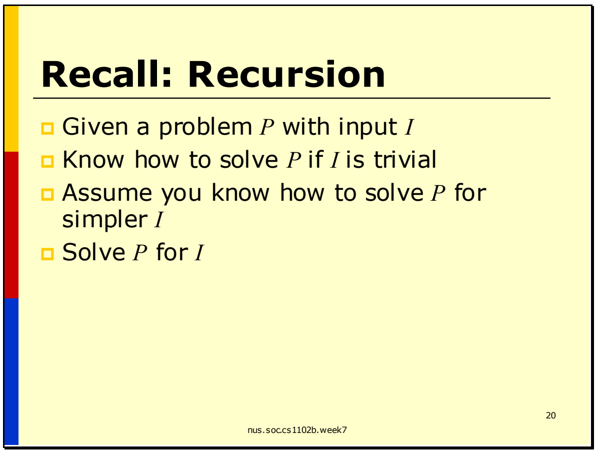
18

nus.soc.cs1102b.week7

- The previous slides show the analysis for general inputs.
- Let’s consider the special cases, when the input is reversely sorted, and when the input is already sorted. The running time for these two cases are Θ(N^2) and Θ(N) respectively. As an exercise, look at the algorithm in the previous slide, and figure out why this is the case.

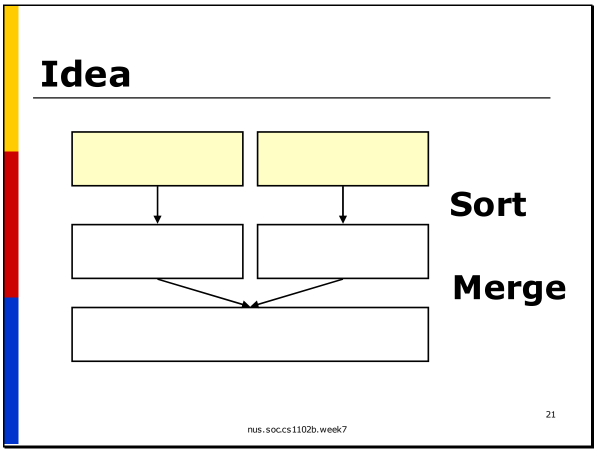


Merge Sort



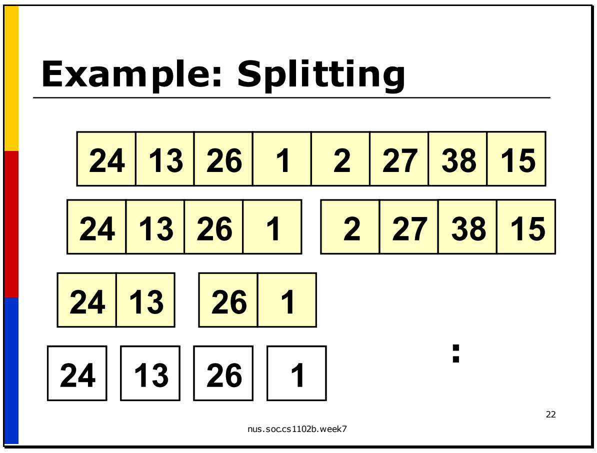
Recall: Recursion

- Given a problem P with input I
- Know how to solve P if I is trivial
- Assume you know how to solve P for simpler I
- Solve P for I



Idea

- The idea of Mergesort is to split the input array into half, recursively sort each half. Then you take the two sorted halves, and merge them together into a sorted array.
- Note: You should know how to merge two sorted list/array by now. This part of the algorithm will be skipped!



Example: Splitting

24 13 26 1 2 27 38 15

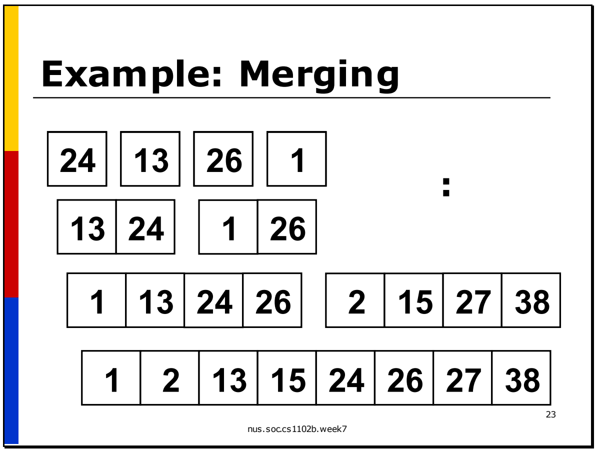
24 13 26 1   2 27 38 15

24 13   26 1

24   13   26   1

:

- We first partition the original array into halves and recursively sort the two halves. This means we take the first half, and partition it into two halves again. Repeat until we have a partition of one element, which is trivially sorted.



Example: Merging

24 | 13 | 26 | 1 :

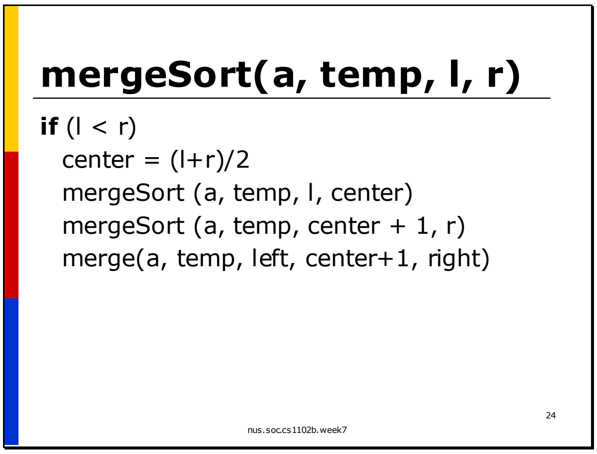
13 24 | 1 26

1 13 24 26 | 2 15 27 38

1 2 13 15 24 26 27 38

nus.soc.cs1102b.week7 23

- We merge the sorted partitions, when we step out of the recursive calls. Note that to merge two arrays, we need a temporary array.

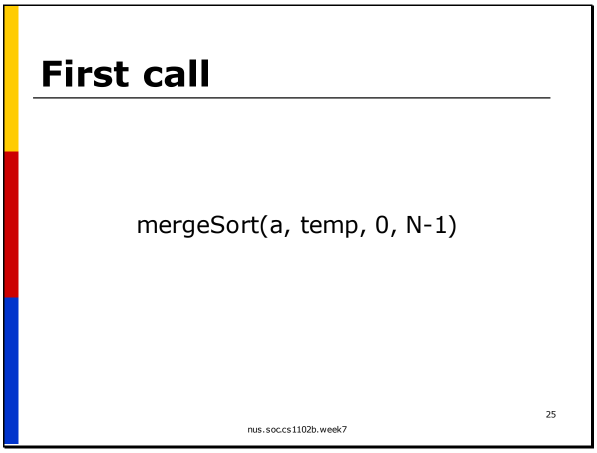


mergeSort(a, temp, l, r)

```
if (l < r)
  center = (l+r)/2
  mergeSort (a, temp, l, center)
  mergeSort (a, temp, center + 1, r)
  merge(a, temp, left, center+1, right)
```

nus.soc.cs1102b.week7 24

- Here is the pseudo-code. *temp* is the temporary array.
- Question: Does this algorithm follow the first three rules of recursion? Why?

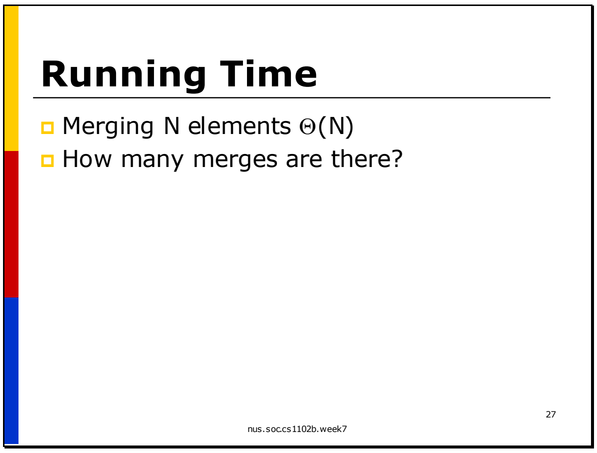


First call

mergeSort(a, temp, 0, N-1)

nus.soc.cs1102b.week7 25

- The first time you call mergeSort, pass in 0 and N-1 as the value of *l* and *r*.

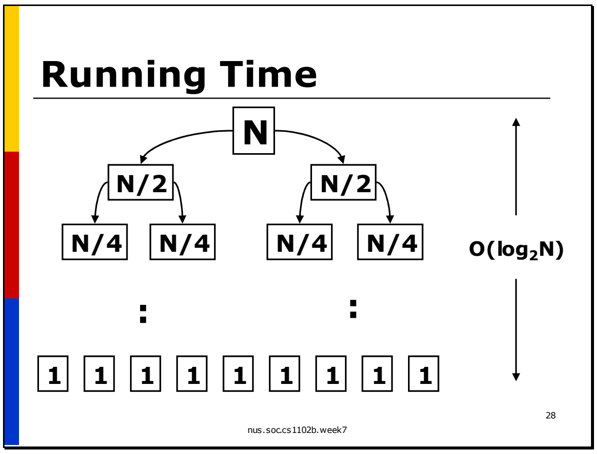


Running Time

- Merging N elements Θ(N)
- How many merges are there?

nus.soc.cs1102b.week7 27

- Merging two lists with a total of N elements takes Θ(N) times.
- Question: Why? Verify this!



Running Time

N

N/2 N/2

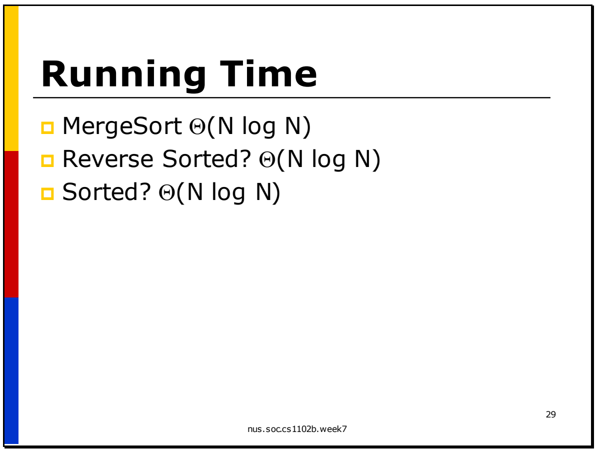
N/4 N/4 N/4 N/4

: :

1 1 1 1 1 1 1 1 1

$O(\log_2 N)$

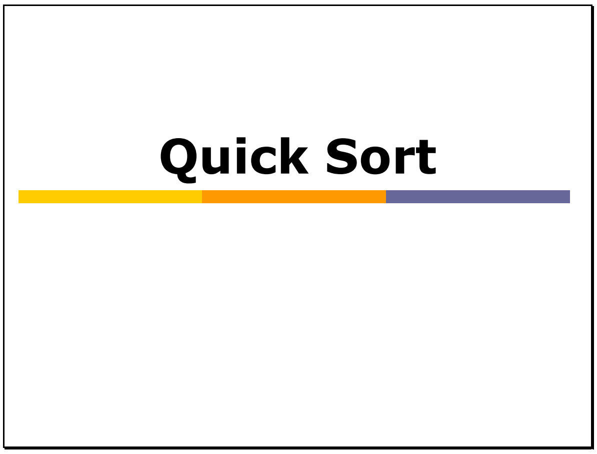
- If we visualize the algorithm like this, we can see that there are actually Θ(log N) levels, since at each level, we half the array.
- Question: Look back at the algorithm analysis of binary search. Do you see a similarity here?
- Question: Is the height of the tree still Θ(log N) if N is not a power of two? Why?
- Merging each level takes Θ(N) times because at level i, we are merging 2^i arrays, each with $N/2^i$ elements. So we always merge a total of N elements, even though they are not being merged into the same array.
- Question: Verify this by calculating the number of merged that occurs at the lowest level, the first level and the second level.



Running Time

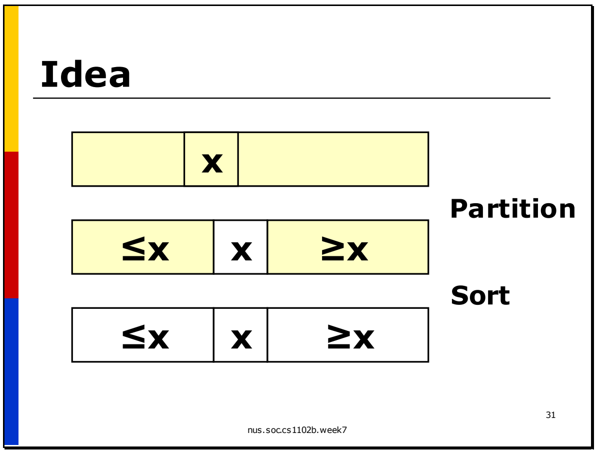
- MergeSort Θ(N log N)
- Reverse Sorted? Θ(N log N)
- Sorted? Θ(N log N)

- The running time of mergesort is hence Θ(N) x Θ(log N) = Θ(N log N)
- The running time of mergesort is the same even if the array is already sorted, or is reverse sorted.
- Question: Verify this! Why?



Quick Sort

- The next sorting algorithm is Quicksort.



Idea

	x	

Partition

≤x	x	≥x

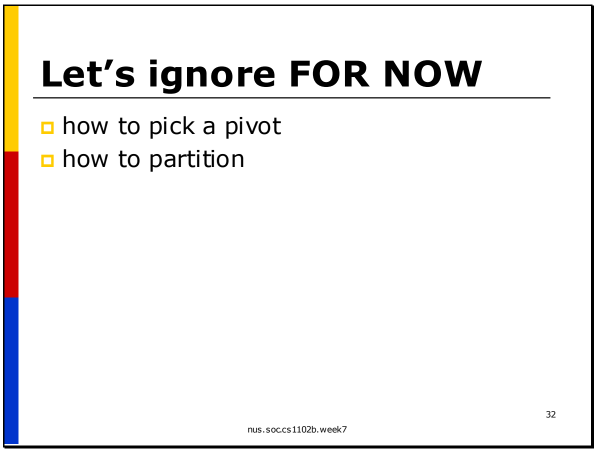
Sort

≤x	x	≥x

nus.soc.cs1102b.week7

31

- The idea is to pick an element from the input, called *pivot*, and use it to partition the array. Let's say we pick x. We partition the array, such that those that are less than or equal to x goes to the left hand side of x, and those that are greater or equal to x goes to the right hand side. After partition, we can be sure that x is its rightful position in a sorted array. (i.e. the position of x, if the array is sorted will be the same as its position now.)
- Question: Why is x in its rightful position?
- We now recursively sort the left partition of the array, and sort the right partition of the array. After sorting the left and right partition, the whole array is sorted.
- Important:
  - The pivot may move from its initial position after partition!
  - The partition may not be of equal length!
  - A partition can be empty! (e.g., if x happen to be the maximum element)

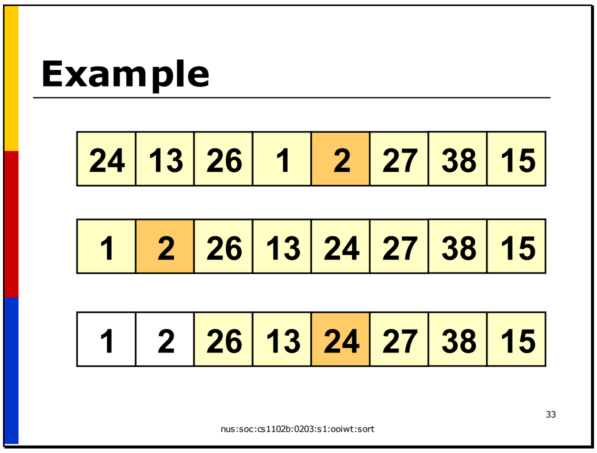


Let's ignore FOR NOW

- how to pick a pivot
- how to partition

nus.soc.cs1102b.week7

32



Example

24	13	26	1	2	27	38	15

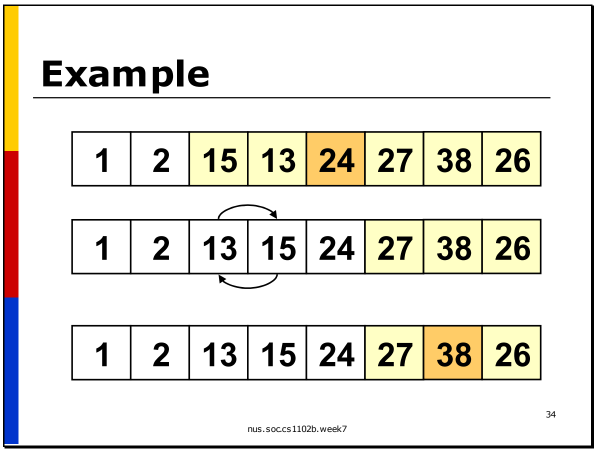
1	2	26	13	24	27	38	15

1	2	26	13	24	27	38	15

nus:soc:cs1102b:0203:s1:ooiwt:sort

33

- We pick 2 as a pivot.
- After partitioning, all elements less than 2 (there are only one in this case, element 1) go to the left of 2, and all elements greater than 2 go to the right of 2.
- Recursively sort the left partition. Nothing to do here, since it is already sorted.
- Recursively sort the right partition. Now, pick 24 as pivot.



Example

1	2	15	13	24	27	38	26

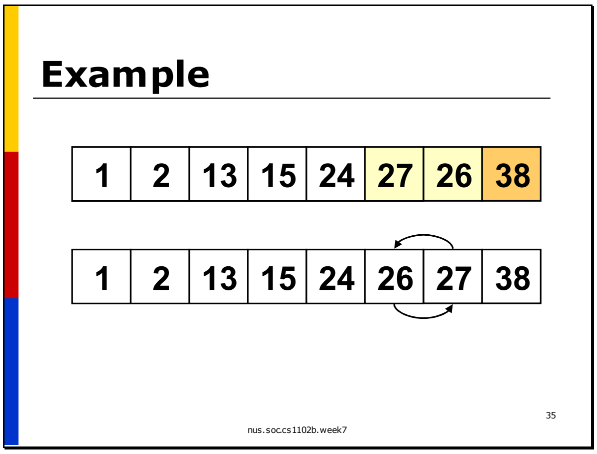
1	2	13	15	24	27	38	26

1	2	13	15	24	27	38	26

nus.soc.cs1102b.week7

34

- Elements less than 24 (13, 15) goes to the left. The rest of elements go to the right.
- Recursively sort the left partition (13,15)
- Recursively sort the right partition (27, 38, 26).
- Pick 38 as pivot.



Example

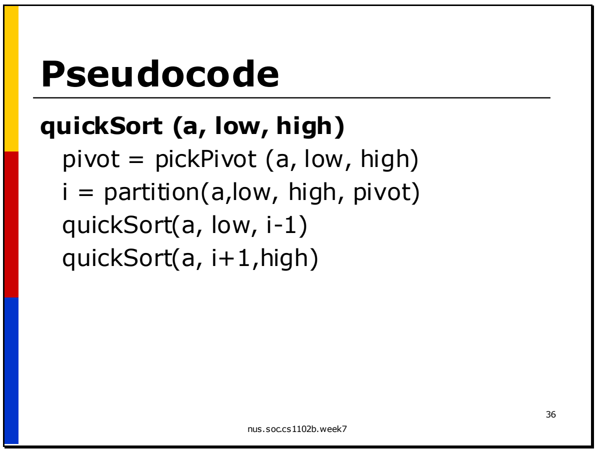
1	2	13	15	24	27	26	38

1	2	13	15	24	26	27	38

nus.soc.cs1102b.week7

35

- Left partition has (27,26), right partition is empty.
- Recursively sort (26,27)
- DONE!



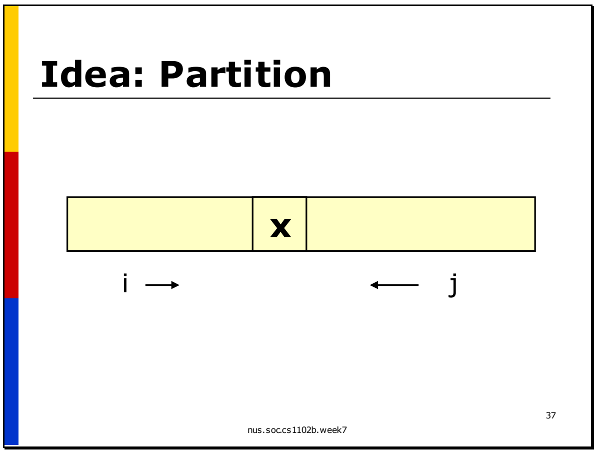
Pseudocode

```
quickSort (a, low, high)
  pivot = pickPivot (a, low, high)
  i = partition(a,low, high, pivot)
  quickSort(a, low, i-1)
  quickSort(a, i+1,high)
```

nus.soc.cs1102b.week7

36

- Here is the pseudo code, again we assume that there exists a method called pickPivot and a method called partition.
- pickPivot takes in low and high, which are variables that tells us the portion of the array we are sorting. It returns the value of the pivot.
- partition takes in the array, the low and high index, and the index of the pivot. It partitions the array and returns the new index of the pivot.
- We then recursively sort the left and right partition.

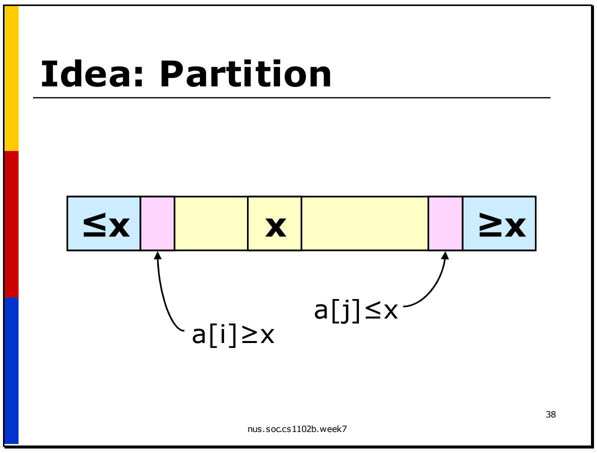


Idea: Partition

nus.soc.cs1102b.week7

37

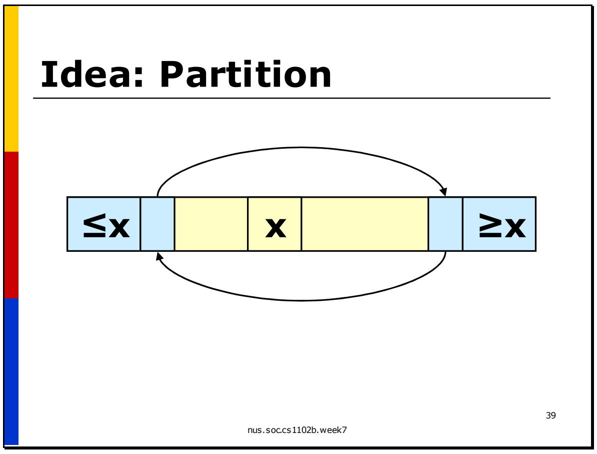
- The idea of partition() is to scan from both end of the array. Maintain two cursors i and j. Move the cursors towards each other. i is used to search for elements that are larger or equal to x, and j is used to search for elements that are smaller or equal to x.



Idea: Partition

nus.soc.cs1102b.week7 38

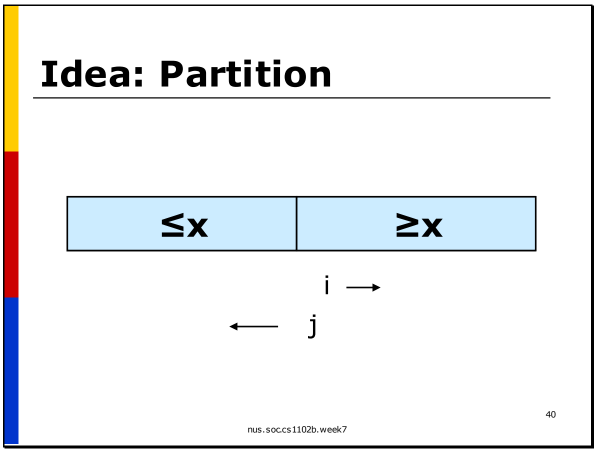
- Stop when we encounter an item that is “out of place” (red items)



Idea: Partition

nus.soc.cs1102b.week7 39

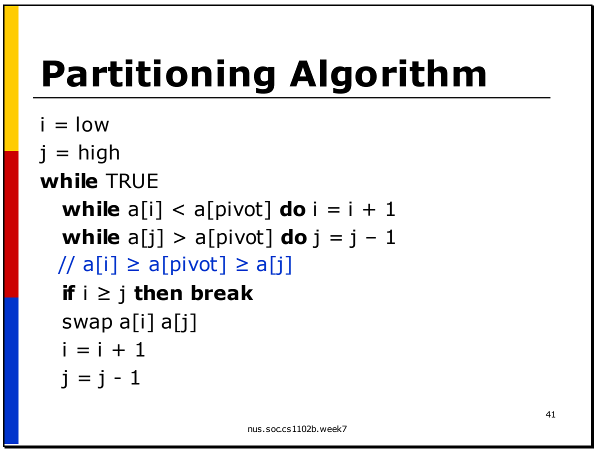
- Swap them and continue.



Idea: Partition

nus.soc.cs1102b.week7 40

- Until i crosses with j

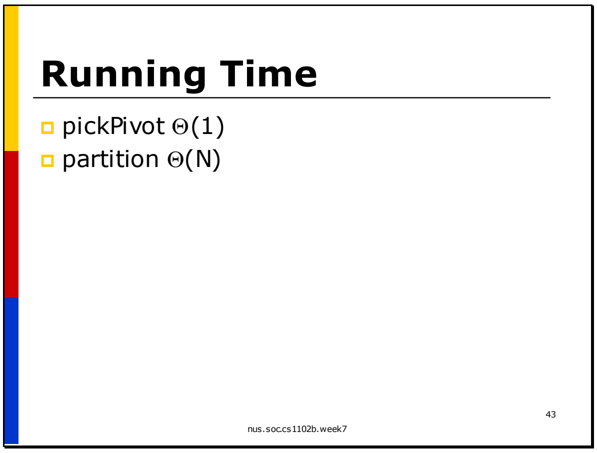


Partitioning Algorithm

```
i = low
j = high
while TRUE
   while a[i] < a[pivot] do i = i + 1
   while a[j] > a[pivot] do j = j − 1
   // a[i] ≥ a[pivot] ≥ a[j]
   if i ≥ j then break
   swap a[i] a[j]
   i = i + 1
   j = j - 1
```

nus.soc.cs1102b.week7 41

- i and j are “cursors” into the array.
- scan i from left, stop when we encounter an out-of-place element (an element that belongs to the right side)
- scan j from right, stop when we encounter an out-of-place element (an element that belongs to the left side)
- swap those elements if i hasn’t cross j.
- Question: Trace through the algorithm using a simple array.

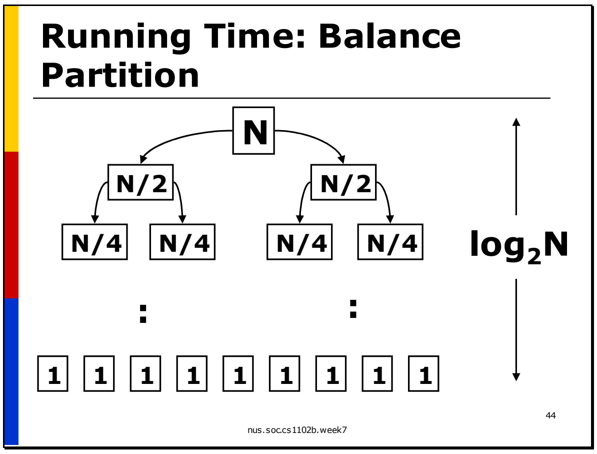


Running Time

- pickPivot Θ(1)
- partition Θ(N)

nus.soc.cs1102b.week7 43

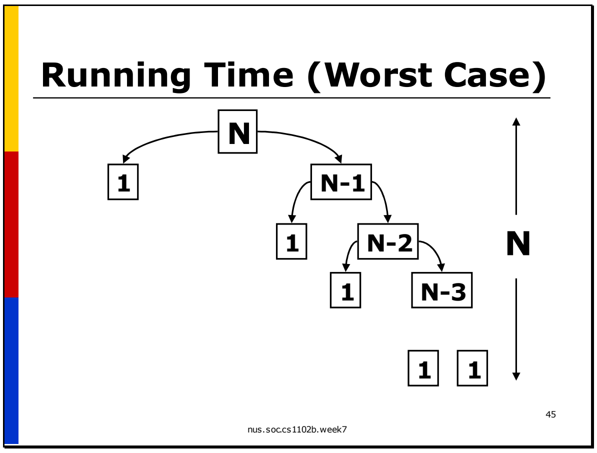
- Let’s assume for now we use a very simple method to pick a pivot, say, we pick the first element. This can be done in Θ(1) time.
- partition requires scanning the whole array. After we examine an element, we know which partition this element belongs to. So we only need to scan the whole array once. Hence, running time for partition is Θ(N).



Running Time: Balance Partition

nus.soc.cs1102b.week7 44

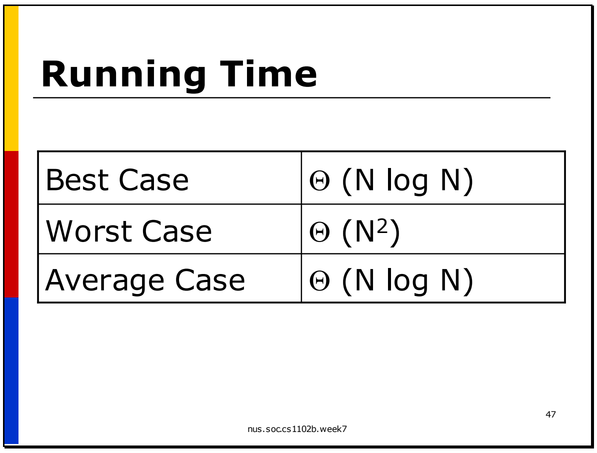
- Recall that partition does not guarantee that we always divide the array into halves. So we have to analyze different cases. Suppose partition always split the array evenly, then we get this picture, similar to merge sort. There will be Θ(log N) levels of recursion. Each level takes Θ(N). So the running time is Θ(Nlog N).



Running Time (Worst Case)

nus.soc.cs1102b.week7 45

- Suppose we pick our pivot badly, and every time we partition the array, one of the partition is always empty. We end up with this case. Note that now we have N levels.
- The running time is thus O(N^2).

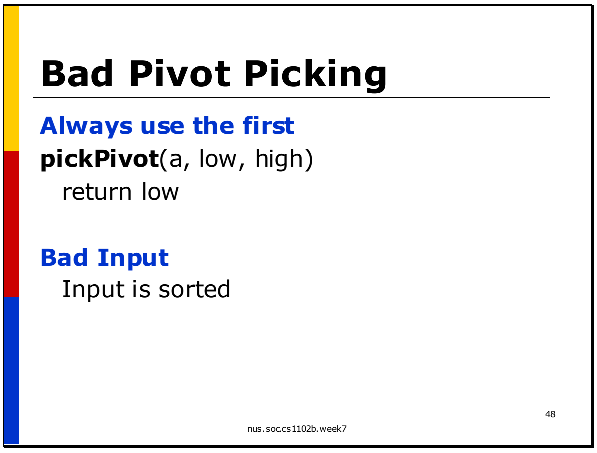


Running Time

Best Case	Θ (N log N)
Worst Case	Θ (N^2)
Average Case	Θ (N log N)

nus.soc.cs1102b.week7 47

- It turn out that balanced partition is the best we can get. Quicksort, therefore, has the best running time of Θ(NlogN) and worst case running time of Θ(N^2).
- Luckily, the average running time of Quicksort is Θ(N log N). The analysis of this is in the book, but is not covered in this course.



Bad Pivot Picking

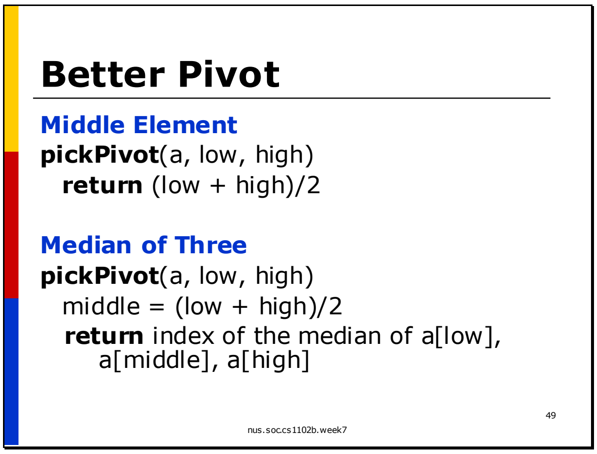
Always use the first

```
pickPivot(a, low, high)
  return low
```

Bad Input

Input is sorted

- We have seen that picking a good pivot is important. An example of bad pivot picking is to always pick the first one. This behaves badly if the input is sorted.
- Question: Why? Verify by a simple example.



Better Pivot

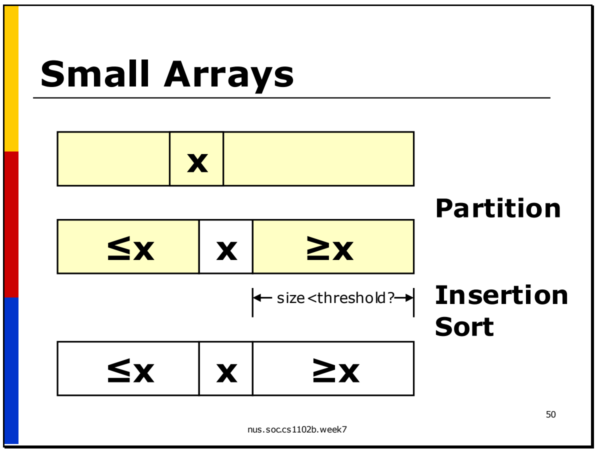
Middle Element

```
pickPivot(a, low, high)
  return (low + high)/2
```

Median of Three

```
pickPivot(a, low, high)
  middle = (low + high)/2
  return index of the median of a[low],
     a[middle], a[high]
```

- A better way to pick a pivot is to pick the middle element in the array. Another way, used in the book, is to pick the median of the first/last and middle element as the pivot.

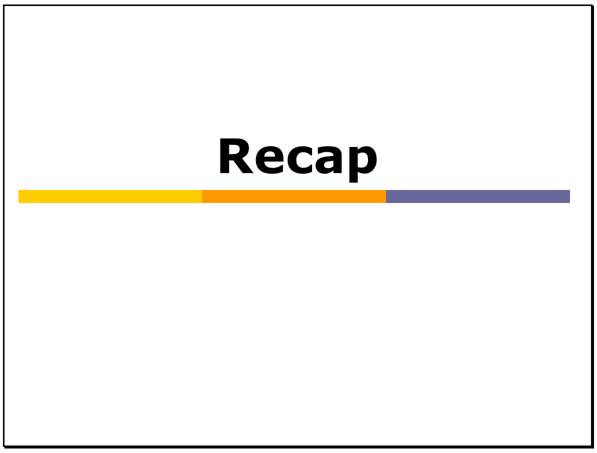


Small Arrays

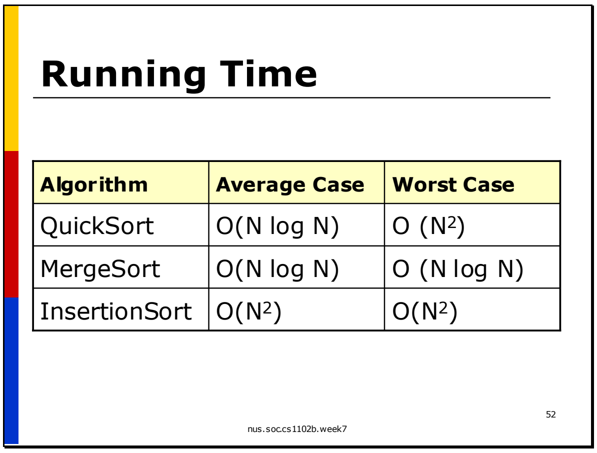
Partition

Insertion Sort

- Since Quicksort has an overhead in picking pivot and partitioning the array, it is not the best algorithm for sorting small arrays, simple (but asymptotically slower) algorithm may be faster in this case.
- An enhancement to Quicksort is to check the size of the partition: if it is smaller than a threshold or cutoff value, we use insertion sort to sort the partition and do not recur further.

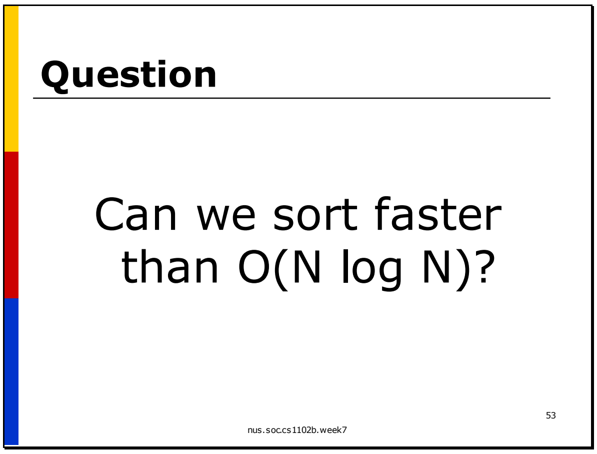


Recap



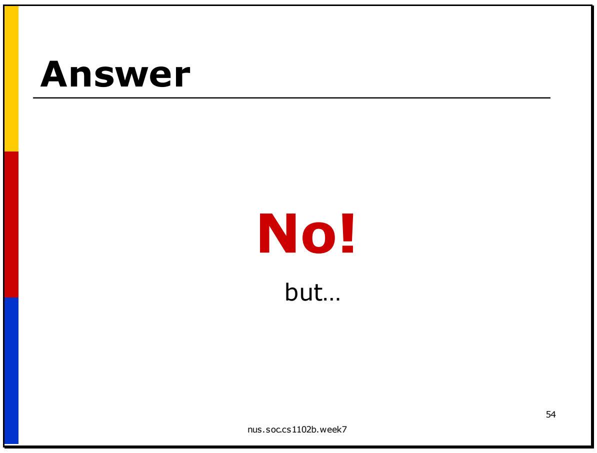
Running Time

Algorithm	Average Case	Worst Case
QuickSort	O(N log N)	O (N^2)
MergeSort	O(N log N)	O (N log N)
InsertionSort	O(N^2)	O(N^2)



Question

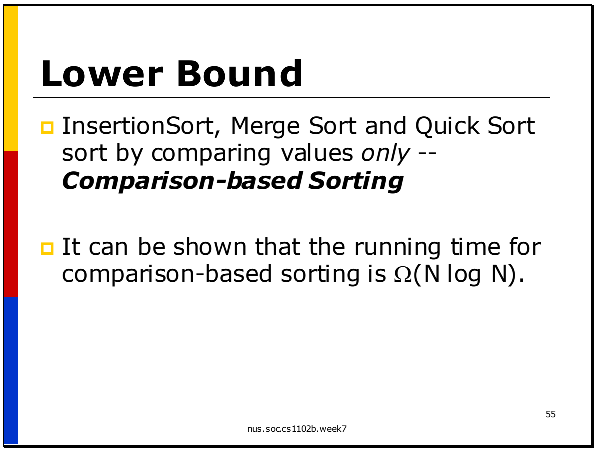
Can we sort faster than O(N log N)?



Answer

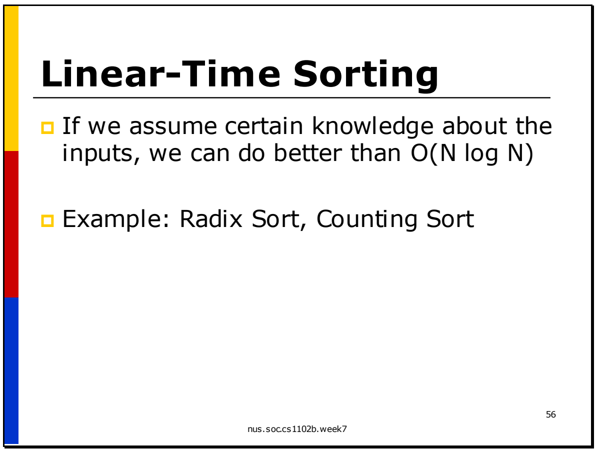
No!

but…



Lower Bound

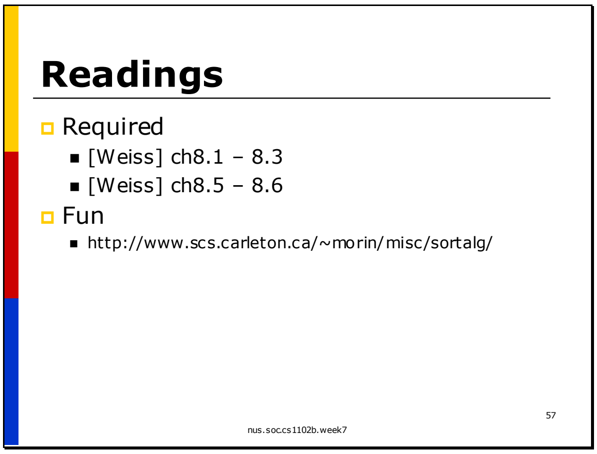
- InsertionSort, Merge Sort and Quick Sort sort by comparing values *only* -- ***Comparison-based Sorting***
- It can be shown that the running time for comparison-based sorting is Ω(N log N).



Linear-Time Sorting

- If we assume certain knowledge about the inputs, we can do better than O(N log N)
- Example: Radix Sort, Counting Sort

- Radix Sort and Counting Sort are two linear time sorting algorithms. They can achieve linear time because they assume some properties in the input. For example, Counting Sort can be use to sort integers within a given range.



Readings

- Required
  - [Weiss] ch8.1 – 8.3
  - [Weiss] ch8.5 – 8.6
- Fun
  - http://www.scs.carleton.ca/~morin/misc/sortalg/