AST20105 Data Structures & Algorithms

CHAPTER 9 – SORTING I

Instructed by Garret Lai

Before Start

- Sorting is one of the most important operations performed by computers.

- The efficiency of data handling can often be substantially increased if the data are sorted according to some criteria of order.

Before Start

- For example:
  - It would be practically impossible to find a name in the telephone directory if the names were not alphabetically ordered.
  - The same can be said about dictionaries, book indexes, payrolls, bank accounts, student lists, and other alphabetically organized materials.

Sorting

Sorting

- Sorting algorithms refer to algorithms that arrange elements of a list in order
- Categories:
  - Comparison-based sorting
    - Selection sort
    - Insertion sort
    - Bubble sort
    - Merge sort
    - Quick sort
  - Non comparison-based sorting
    - Counting sort
    - Bucket sort
    - Radix sort

Sorting

- The first step is to choose the criteria that will be used to order data.
  - Very often, the sorting criteria are natural, as in the case of numbers.
    - A set of numbers can be sorted in ascending or descending order.
      - Ascending: (1, 2, 5, 8, 20)
      - Descending: (20, 8, 5, 2, 1)

Sorting

- Names in the phone book are ordered alphabetically by last name, which is the natural order.

  - For alphabetic and non-alphabetic characters, the American Standard Code for Information Interchange (ASCII) code is commonly used.

Common Terminology for Sorting

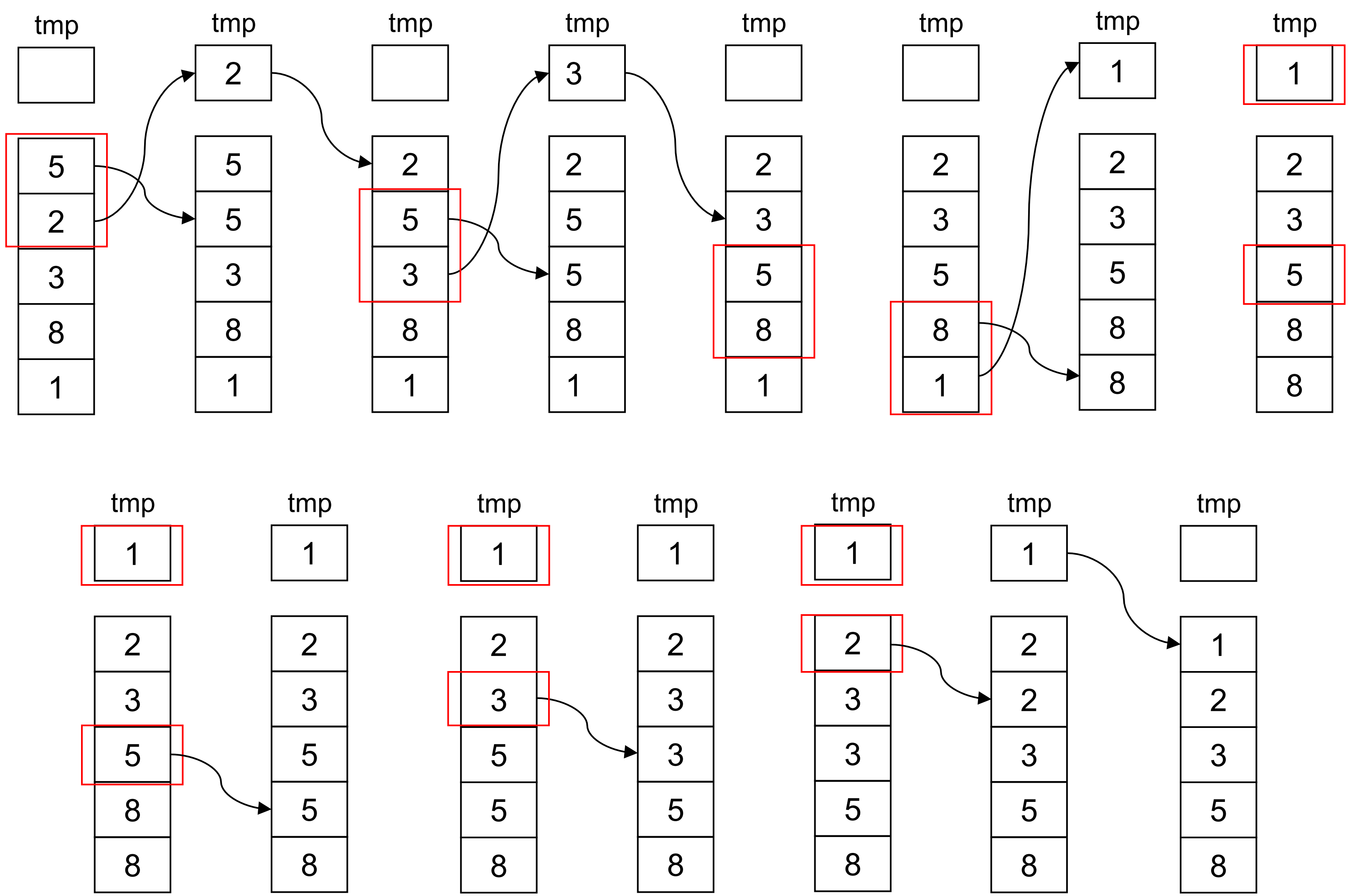
- In-place sorting
  - The amount of extra space required to sort the data is constant with the input size
- Stable sorting
  - A stable sorting preserves relative order of equal values

Insertion Sort

Insertion Sort

- It's the same strategy that you use for sorting your bridge hand.
  - You pick up a card,
  - Start at the beginning of your hand and find the place to insert the new card,
  - Insert it and move all the others up one place.

Insertion Sort



Insertion Sort

```
template<class T>
void insertionsort(T data[], int n) {
    for (int i = 1, j; i < n; ++i) {
        T tmp = data[i];
        for (j = i; j > 0 && tmp < data[j-1]; --j)
            data[j] = data[j-1];
        data[j] = tmp;
    }
}
```

Worst Case Analysis of Insertion Sort

- Occurs if the array is sorted in reverse order
  - Inserting the n^{th} element, we need at most n-1 comparisons and n-1 element movement
    Inserting the $n\text{-}1^{th}$ element, we need at most n-2 comparisons and n-2 element movement
    …
    Inserting the 2^{nd} element, we need 1 comparison and one element movement
  - Total number of operations
    $= 2 * (1 + 2 + 3 + \ldots + n\text{-}1)$
    $= 2 * ((1 + n - 1)(n\text{-}1) / 2)$
    $= n(n\text{-}1)$
    $= O(n^2)$

Best Case Analysis of Insertion Sort

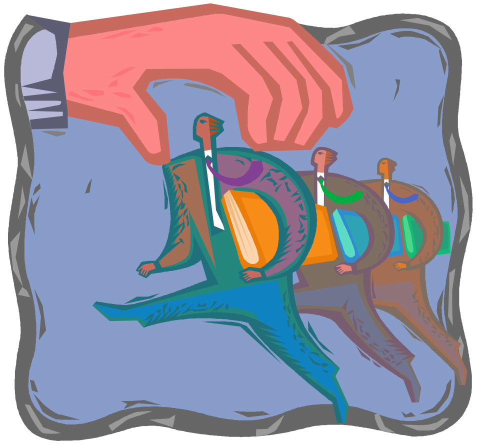
- Occurs if the array is already sorted
  - Inserting the n^{th} element, we need 1 comparison and 0 element movement
    Inserting the $n-1^{th}$ element, we need 1 comparison and 0 element movement
    …
    Inserting the 2^{nd} element, we need 1 comparison and 0 element movement
  - Total number of operations
    $= n - 1$
    $= O(n)$

Insertion Sort - Pros and Cons

- Pros
  - Efficient for sorting of small data
  - Efficient for data that are almost sorted
  - In-place sorting as only constant amount of additional memory space is required
  - Stable sorting algorithm, since it does not change the relative order of elements with equal keys

- Cons
  - Less efficient for sorting of large data

Selection Sort

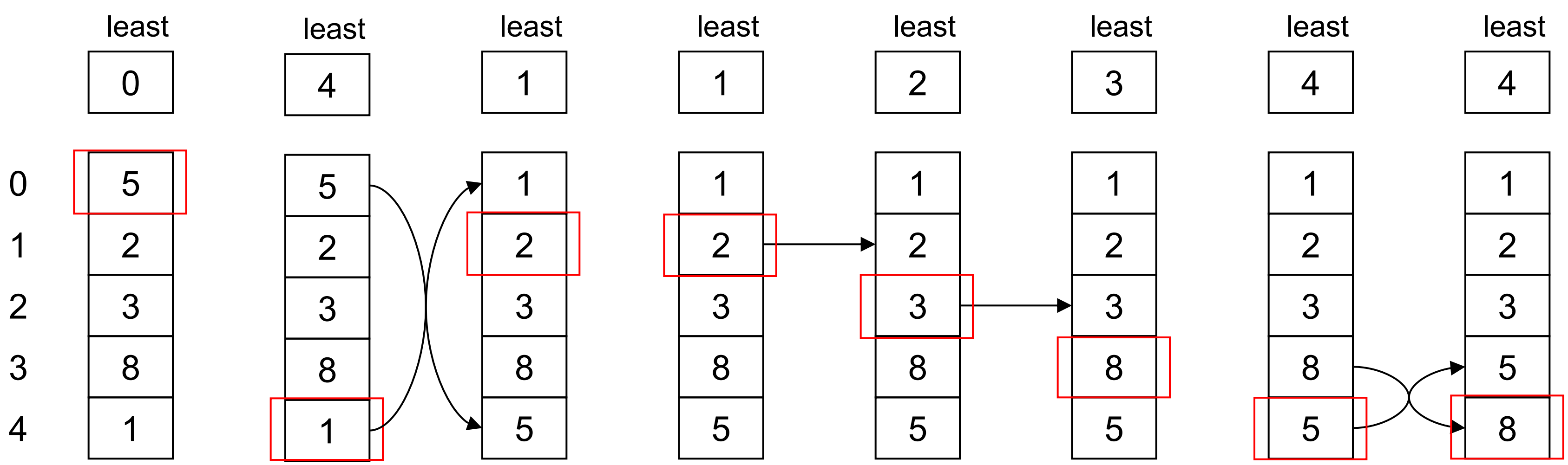


Selection Sort

- Selection sort is an attempt to localize the exchanges of array elements by
  finding a misplaced element first and
  putting it in its final place.

  - The element with the lowest value is selected and exchanged with the element in the first position.

Selection Sort



Selection Sort

```
template<class T>
void selectionsort(T data[], int n) {
    for (int i = 0, j, least; i < n-1; ++i) {
        for (j = i+1, least = i; j < n; ++j)
            if (data[j] < data[least])
                least = j;
        swap(data, least, i);
    }
}
```

Worst Case Analysis of Selection Sort

- Finding the largest element needs n-1 comparisons and 1 element swap
  Finding the second largest element needs n-2 comparisons and 1 element swap
  …
  Finding the n-1th largest element needs 1 comparison and 1 element swap
- Total number of operations
  $= (n-1) + (1 + 2 + 3 + \ldots + n-1)$
  $= (n-1) + (1+n-1)(n-1)/2$
  $= (n-1) + n(n-1)/2$
  $= O(n^2)$

Selection Sort - Pros and Cons

- Pros
  - Easy to implement
- Cons
  - Is no faster on a partially sorted array

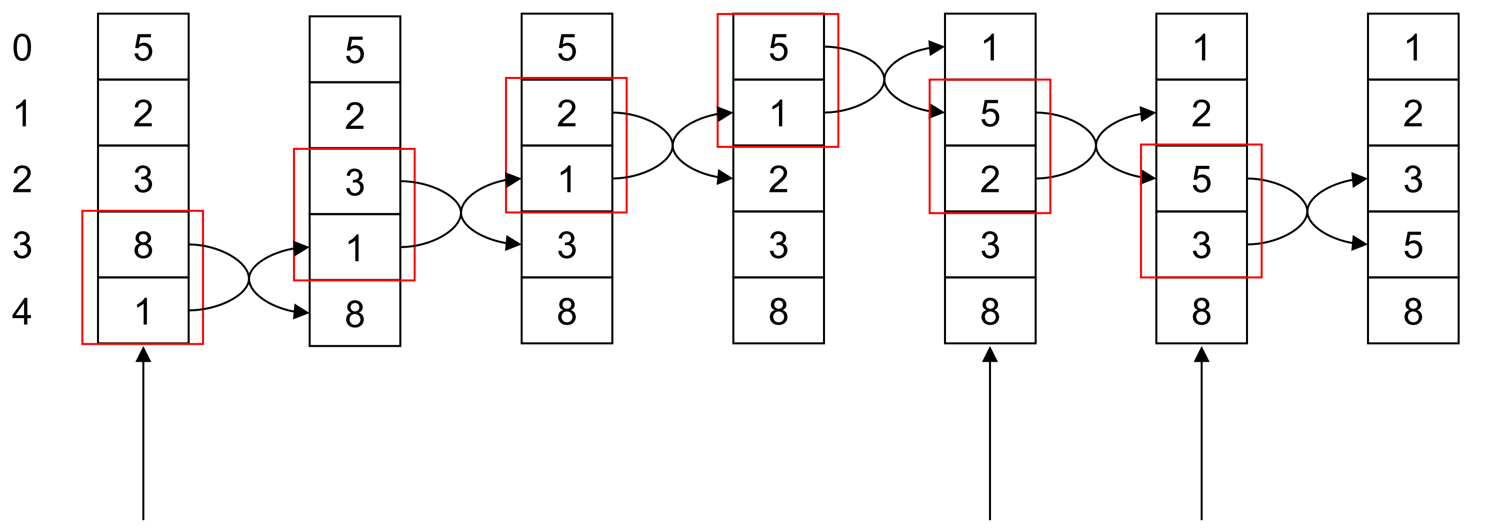
Bubble Sort

Bubble Sort

- A bubble sort can be best understood if the array to be sorted is envisaged as a vertical column whose smallest elements are at the top and whose largest elements are at the bottom.

  - The array is scanned from the bottom up, and two adjacent elements are interchanged if they are found to be out of order with respect to each other.

Bubble Sort



Bubble Sort

```
template<class T>
void bubblesort(T data[], int n) {
    for (int i = 0; i < n-1; ++i)
        for (int j = n-1; j > i; --j)
            if (data[j] < data[j - 1])
                swap(data, j, j-1);
}
```

Quicksort

Quicksort

- The divide-and-conquer strategy is used in quicksort. Below the recursion step is described:
  - **Choose a pivot value**
    - We take the value of the middle element as pivot value,
    - but it can be any value, which is in range of sorted values, even if it doesn't present in the array.

Quicksort

- The divide-and-conquer strategy is used in quicksort. Below the recursion step is described:
  - **Partition**
    - Rearrange elements in such a way,
      - that all elements which are lesser than the pivot go to the left part of the array and
      - all elements greater than the pivot, go to the right part of the array.
      - Values equal to the pivot can stay in any part of the array. Notice, that array may be divided in non-equal parts.

Quicksort

- The divide-and-conquer strategy is used in quicksort. Below the recursion step is described:
  - **Sort both parts**
    - Apply quicksort algorithm recursively to the left and the right parts.

Quicksort

- **Partition algorithm in detail**
  - There are two indices **i** and **j**.
  - At the very beginning of the partition algorithm
    - **i** points to the first element in the array and
    - **j** points to the last one.
  - Then algorithm moves **i** forward, until an element with value greater or equal to the pivot is found.
  - Index **j** is moved backward, until an element with value lesser or equal to the pivot is found.

Quicksort

- **Partition algorithm in detail**
  - If **i** ≤ **j** then they are swapped and i steps to the next position **(i + 1)**, j steps to the previous one **(j - 1)**.
  - Algorithm stops, when **i** becomes greater than **j**.
  - After partition,
    - all values before **i-th** element are less or equal than the pivot

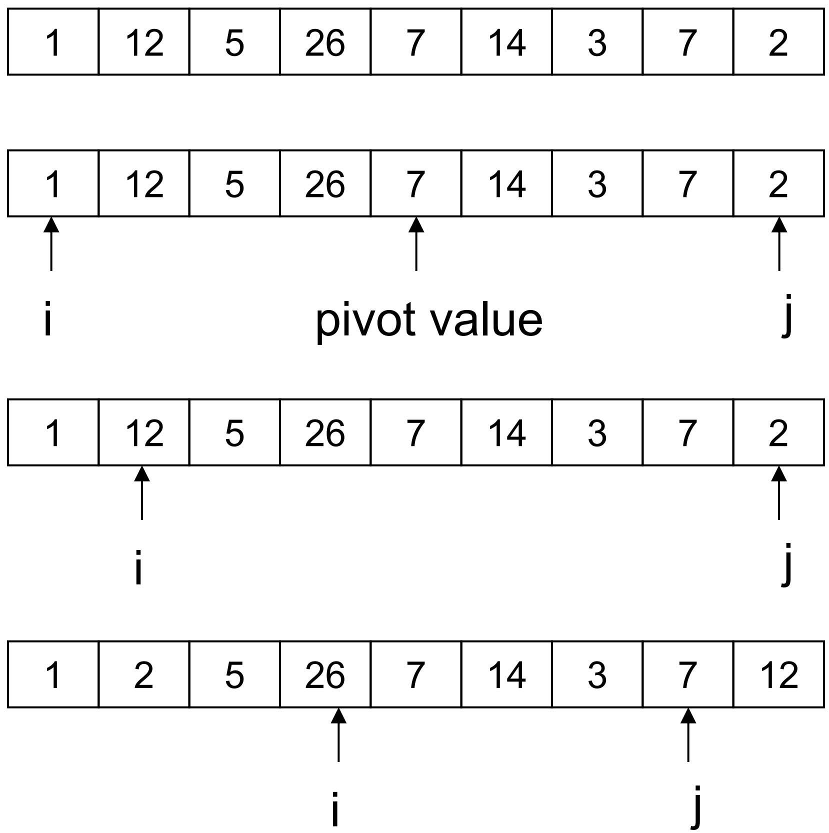
      and
    - all values after **j-th** element are greater or equal to the pivot.

Quicksort

- *Example:*
  - Sort {1, 12, 5, 26, 7, 14, 3, 7, 2} using quicksort.

1	12	5	26	7	14	3	7	2

Quicksort



- Unsorted
- Pivot value = 7
- 12 >= 7 >= 2. swap
- 26 >= 7 >= 7, swap

Quicksort

1	2	5	7	7	14	3	26	12

1	2	5	7	7	14	3	26	12
				i		j		

- 7 >= 7 >= 3, swap

1	2	5	7	3	14	7	26	12
				j	i			

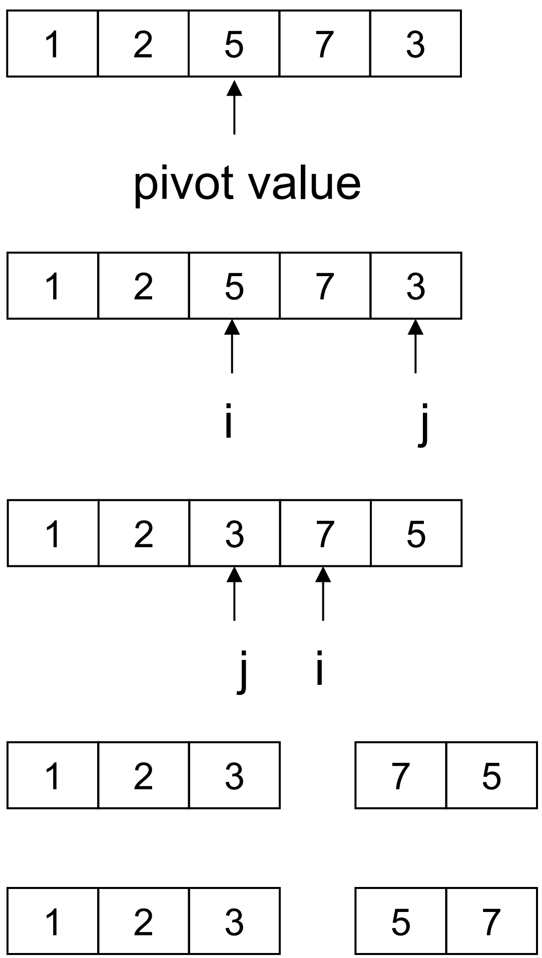
- i > j, stop partition

1	2	5	7	3

14	7	26	12

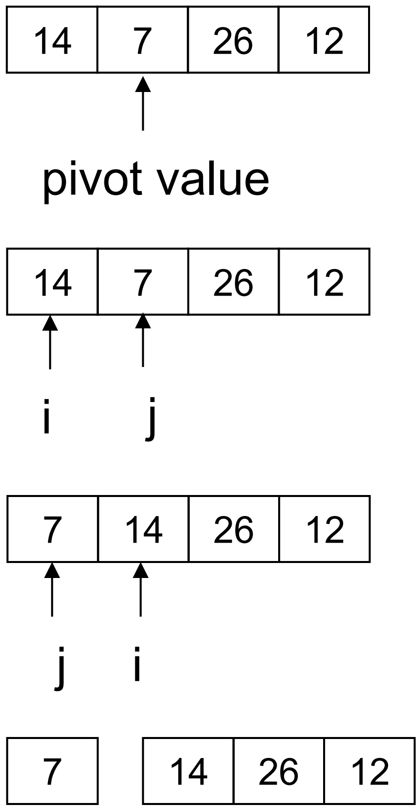
- run quicksort recursively

Quicksort



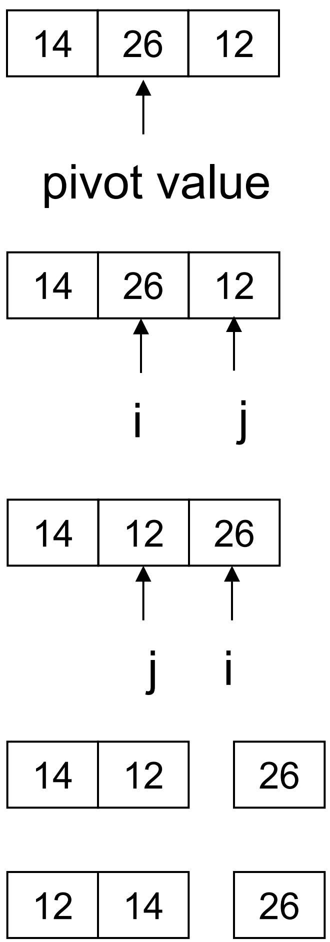
- Pivot value = 5
- 5 >= 5 >= 3. swap
- i > j, stop partition
- run quicksort recursively

Quicksort



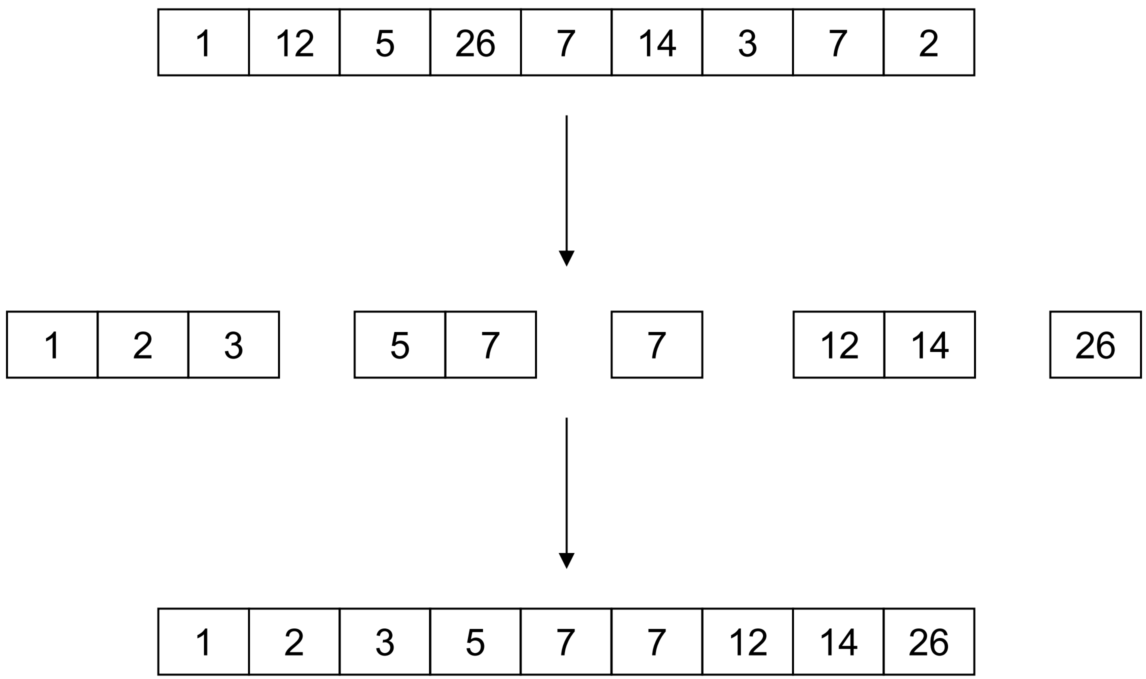
- Pivot value = 7
- 14 >= 7 >= 7. swap
- i > j, stop partition
- run quicksort recursively

Quicksort



- Pivot value = 26
- 26 >= 26 >= 12. swap
- i > j, stop partition
- run quicksort recursively

Quicksort



Quicksort

```
void quickSort(int arr[], int left, int right) {
    int i = left, j = right;
    int tmp;
    int pivot = arr[(left + right) / 2];

    /* partition */
    while (i <= j) {
        while (arr[i] < pivot)
            i++;
        while (arr[j] > pivot)
            j--;

        if (i <= j) {
            tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
            i++;
            j--;
        }
    };

    /* recursion */
    if (left < j)
        quickSort(arr, left, j);
    if (i < right)
        quickSort(arr, i, right);
}
```

Quicksort - How to Pick Pivot?

- Use the last element as pivot
  - Fine if the input is random
  - If the input is already sorted in non-decreasing order or the reverse
    - All elements would be in one sub-array and the other is empty
    - This happens for every recursively calls
    - Resulted in a very bad running time

- Randomly chosen pivot
  - Generally is a good option, but random number generation can be expensive

Quicksort - How to Pick Pivot?

- Use the median as the pivot
  - Partitioning always partitions the input array into two halves of the same size
  - However, it is difficult to find median
  - Solution: Use median of three

- Median of three
  - Compare three elements, the leftmost, rightmost and the center one

Analysis of Quicksort

- Assumptions:
  - A random pivot
  - No use of insertion sort for small array

- Let $T(n)$ be the running time of quick sort to sort n numbers
- Assume n is a power of 2

- Analysis:
  - Pivot selection: $O(1)$ time
  - Partitioning: $O(n)$ time
  - Running time of two recursive calls
    $T(1) = 1$
    $T(n) = T(i) + T(n-i-1) + n$

Worst Case Analysis of Quick Sort

- Worst case when the chosen pivot is the smallest element, all the time
- Partition is always unbalanced
- $T(1) = 1$
  $T(n) = T(n-1) + n$

$T(n)=T(n-1)+n$
$T(n)=T(n-2)+(n-1)+n$
$T(n)=T(n-3)+(n-2)+(n-1)+n$
…
$T(n)=T(n-k)+(n-(k-1)) + \ldots + n$

$n-k = 1$
$k = n - 1$

$T(n) = T(1) + (n-(n-1-1)) + \ldots + n$
$T(n) = 1 + 2 + \ldots + n$
$T(n) = (1+n)(n) / 2$
$T(n) = O(n^2)$

Best Case Analysis of Quick Sort

- Best case when the chosen pivot is always the median of the array, all the time
- Partition is always balanced
- $T(1) = 1$
  $T(n) = 2T(n/2) + n$

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2(2T(n/2^2) + n/2) + n \\
&= 2^2T(n/2^2) + 2n \\
&= 2^2(2T(n/2^3) + n/2^2) + 2n \\
&= 2^3T(n/2^3) + 3n \\
&\;\;\dots \\
&= 2^kT(n/2^k) + kn
\end{aligned}
$$

$$
\begin{aligned}
n &= 2^k \\
\log_2 n &= \log_2 2^k \\
k &= \log_2 n
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= nT(1) + n\log_2 n \\
&= n(1) + n\log_2 n \\
&= n\log_2 n + n \\
&= O(n\log n)
\end{aligned}
$$

Quick Sort - Pros and Cons

- On average, the running time is O(nlogn)

- Pros
  - Extremely fast on average

- Cons
  - Fairly tricky to implement
  - Very slow in the worst case (but not likely to occur)

Merge Sort

Merge Sort

- The problem with quicksort:
  - Its complexity in the worst case is $O(n^2)$
    - Because it is difficult to control the partitioning process
    - There is no guarantee that partitioning results in arrays of approximately the same size.

Merge Sort

- The problem with quicksort:
  - To overcome this problem, another strategy is to make partitioning as simple as possible and concentrate on merging the two sorted array.
  - This strategy is characteristic of mergesort.

Merge Sort

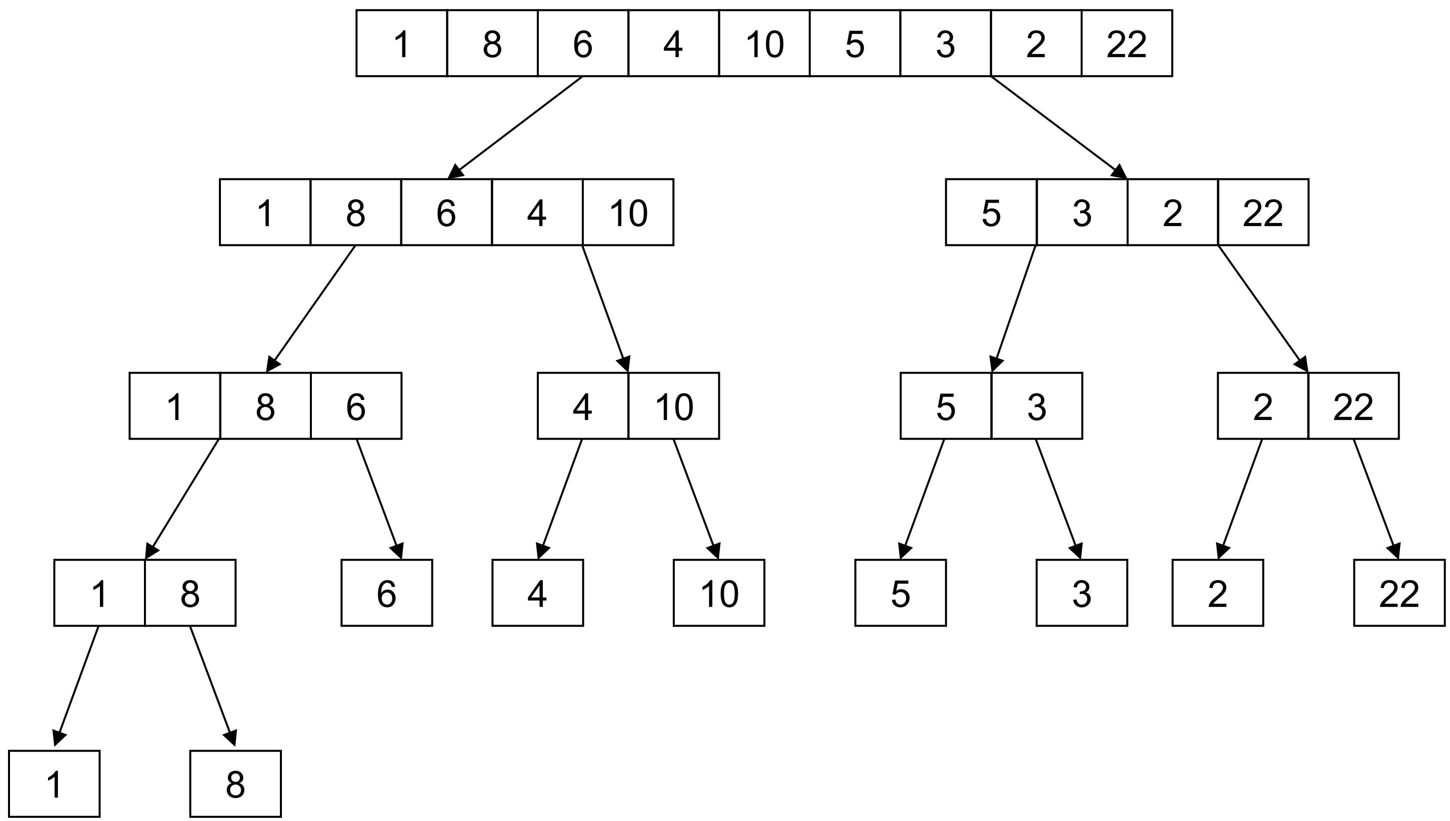
- It was one of the first sorting algorithms used on a computer.

- Developed by John von Neumann.

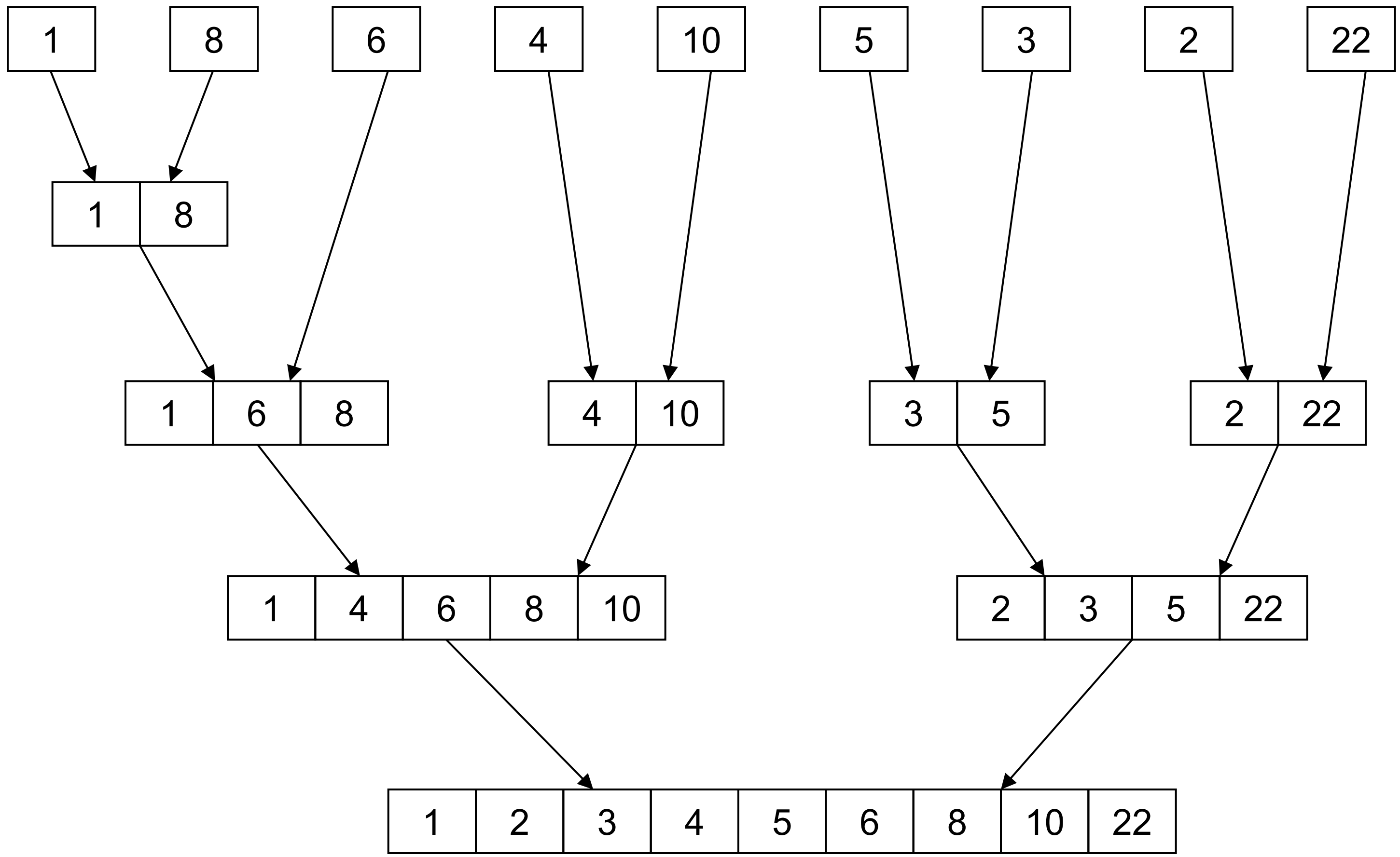
Merge Sort

- The key process in merge sort is
  - Merging sorted halves of an array into one sorted array.
  - However, these halves have to be sorted first, which is accomplished by merging the already sorted halves of these halves.
  - This process of dividing arrays into two halves stops when the array has fewer than two elements.
  - The algorithm is recursive in nature.

Merge Sort

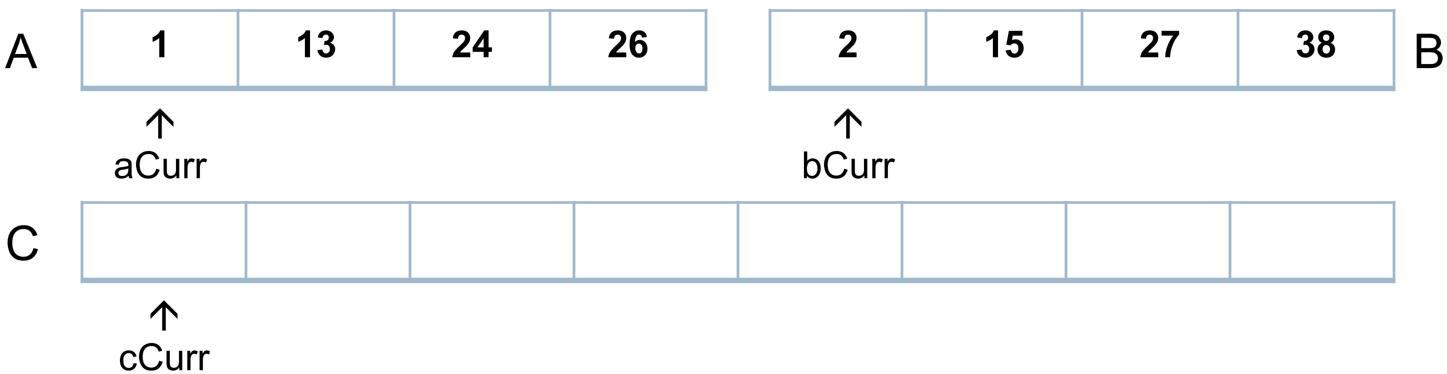


Merge Sort



Merge Sort - How to Merge?

- Input: Two sorted array A and B
- Output: An sorted array C
- Three counters: aCurr, bCurr, cCurr
  - Initially set them to the beginning of their respective arrays



  - The smallest of A[aCurr] and B[aCurr] is copied to the next entry in C, and the counters are increased by 1
  - When either count reached the end, the remaining elements in the other list is copied to C

Example - Merge

Original

A	1	13	24	26	2	15	27	38	B
	↑aCurr				↑bCurr				
C									
	↑cCurr								

Step 1

A	1	13	24	26	2	15	27	38	B
		↑aCurr			↑bCurr				
C	1								
		↑cCurr							

Step 2

A	1	13	24	26	2	15	27	38	B
		↑aCurr				↑bCurr			
C	1	2							
			↑cCurr						

Example - Merge

Step 3

A

1	13	24	26

↑aCurr

B

2	15	27	38

↑bCurr

C

1	2	13					

↑cCurr

Step 4

A

1	13	24	26

↑aCurr

B

2	15	27	38

↑bCurr

C

1	2	13	15				

↑cCurr

Step 5

A

1	13	24	26

↑aCurr

B

2	15	27	38

↑bCurr

C

1	2	13	15	24			

↑cCurr

Example - Merge

Step 6

A

1	13	24	26	2	15	27	38

B

↑bCurr

C

1	2	13	15	24	26		

↑cCurr

Last step

A

1	13	24	26	2	15	27	38

B

Copy all the remaining elements over to C

C

1	2	13	15	24	26	27	28

Analysis of Merge Operation

- The running of merge takes O(n1 + n2) where n1 and n2 are the sizes of the two sub-arrays, which is O(n)

- Space requirements of merge operation:
  - Merging two sorted lists requires O(n) extra memory
  - Additional work to copy the temporary array back to the original array

Merge Sort - C++ Code

```
void mergeSort(int arr[], int left, int right, int size)
{
    if(left < right)
    {
        int center = (left + right)/2;
        mergeSort(arr, left, center, size);
        mergeSort(arr, center + 1, right, size);
        merge(arr, left, center, right, size);
    }
}
```

```
void merge(int arr[], int low, int mid,
                          int high, int size)
{
    int* c = new int[size];
    int l = low;
    int i = low;
    int j = mid+1;

    while((l<=mid) && (j<=high)) {
        if(arr[l] <= arr[j]) { c[i] = arr[l]; l++; }
        else { c[i] = arr[j]; j++; }
        i++;
    }
    if(l > mid) {
        for(int k=j; k<=high; k++) {
            c[i] = arr[k]; i++;
        }
    }
    else {
        for(int k=l; k<=mid; k++)
        {
            c[i] = arr[k]; i++;
        }
    }

    for(int k=low; k<=high; k++)
          arr[k] = c[k];
    delete [] c;
}
```

Analysis of Merge Sort

- Let $T(n)$ be the worst-case running time of merge sort to sort n numbers
- Assume n is a power of 2
- Analysis:
  - Divide: $O(1)$ time
  - Conquer: $2T(n/2)$ time
  - Combine step: $O(n)$ time
  - Recurrence equation:
    $T(1) = 1$
    $T(n) = 2T(n/2) + n$

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2(2T(n/2^2) + n/2) + n \\
&= 2^2T(n/2^2) + 2n \\
&= 2^2(2T(n/2^3) + n/2^2) + 2n \\
&= 2^3T(n/2^3) + 3n \\
&\quad \ldots \\
&= 2^kT(n/2^k) + kn
\end{aligned}
$$

$$
\begin{aligned}
n &= 2^k \\
\log_2 n &= \log_2 2^k \\
k &= \log_2 n
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= nT(1) + n\log_2 n \\
&= n(1) + n\log_2 n \\
&= n\log_2 n + n \\
&= O(n\log n)
\end{aligned}
$$

Merge Sort - Pros and Cons

- Pros
  - It is a stable sort, i.e. it preserves relative order of equal values

- Cons
  - Requires additional storage proportional to the size of the input array for merge operations

CHAPTER 9 END