Searching and Sorting

LEUNG MAN HO (LMH)

Table Of Content – Searching

Table Of Content – Sorting

- Comparison Sort
  - Bubble Sort
  - Insertion Sort
  - Selection Sort
  - Shell Sort
  - Merge Sort
  - Quick Sort
- Counting Sort
- Radix Sort

Searching – Introduction

- Usage
  - Locating an object in an array
  - Finding an optimal number for a problem

- Often require preprocessing

Linear Search – Introduction

- aka Sequential Search
- Check until a match is found

Situation	Time Complexity
Best	$O(1)$
Worst	$O(N)$
Average	$O(N)$

Binary Search

- Preprocessing: Sorted
- Eliminate impossible region
  - $A_0, A_1, \ldots, A_{n-1}$
  - If key < A_k, then key < A_i for i >= k
  - If key > A_k, then key > A_i for i <= k

Binary Search – Algorithm

- A[low] to A[high]
- Repeat the following process until the key is found
  - Compare key with A[mid]
  - If key < A[midpoint], then A[mid] to A[high] does not contain the key
  - If key > A[midpoint], then A[low] to A[mid] does not contain the key

Binary Search – Buggy Code

```
int search( int key, int *a ) {
  int hi, mid, lo;
  lo=0; hi=n-1;
  while (hi-lo >=0) {
        mid = (hi+lo) / 2;
        if ( key <= a[mid] )  hi = mid;
        else lo  = mid;
     }
     if ( key==a[hi] )  return( hi );
     else   return( -1 );
}
```

Binary Search – Correct Implementation

```
int search( int key, int *a ) {
  int hi, mid, lo;
  lo=-1; hi=n;
  while (hi-lo >1) {
          mid = (hi+lo) / 2;
          if ( key <= a[mid] )  hi = mid;
          else lo  = mid;
   }
   if ( key==a[hi] )  return( hi );
   else   return( -1 );
}
```

Binary Search

- Complexity: $O(\log N)$
- Why `mid = (hi + lo) / 2`
  - The expected area eliminated is largest

Ternary search – Introduction

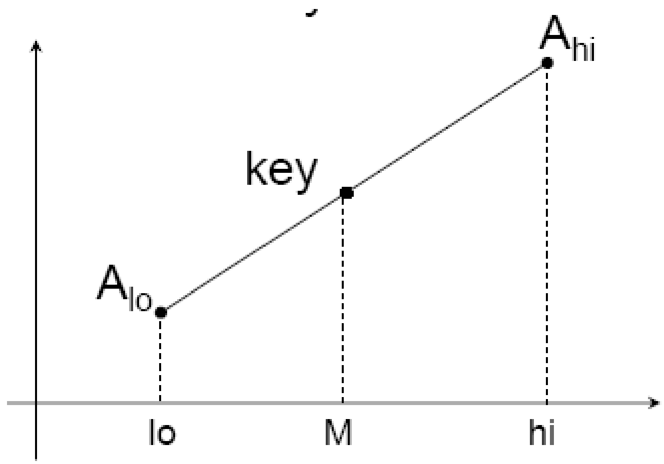
- Finding the max/min of a single-peak function
- Function Requirement
  - there must be some value x such that
  - for all a, b with $A \le a < b \le x$, we have $f(a) < f(b)$, and
  - for all a, b with $x \le a < b \le B$, we have $f(a) > f(b)$
  - Or other way round
- Useful in many optimizing problem

Ternary search – Algorithm

```
// Find maximum of unimodal function f() within [left, right]
// To find the minimum, revert the comparison.
while True do
  // left and right are the current bounds; the maximum is between them
  if abs(right - left) < absolutePrecision
    return (left + right) / 2
  leftThird = left + (right - left) / 3
  rightThird = right - (right - left) / 3
  if f(leftThird) < f(rightThird)  left = leftThird
  else  right = rightThird
```

Interpolation Search – Introduction

- Requirement: Sorted



$$M \approx lo + \frac{hi - lo}{A_{hi} - A_{lo}} \times (key - A_{lo})$$

- Good if the elements in the list are *uniformly distributed*

More on Searching

- Given a key and a list, find *i* such that (A_i >= key) and (A_i – key) is minimized
- Linear Search
  - search all elements
- Binary Search
  - If the key exists, easy
  - If the key does not exists, make use of hi

More on Searching

- Which searching algorithm is the best?
- Depends on situation
  - Size of list
  - Ordering of elements
  - Frequency of search
  - Frequency of list modification
  - Distribution of elements

Sorting – Introduction

- Consistent data ordering

- Algorithm preprocessing
  - Searching
  - Greedy
  - DP

- OI commonly used sorting
  - Counting Sort
  - Bubble Sort
  - Merge Sort

Bubble Sort – Introduction

- Compare two **neighboring** keys, swap if in the wrong order
- In k-th iteration
  - The k-largest key will be ***bubbled*** to its end position
  - Other keys may still be out of order
- N – 1 iterations are needed

Bubble Sort – Example

```
Original       34 8 64 51 32 21   # of swaps
----------------------------------------
After p = 1, 8 34 51 32 21 64    4
After p = 2, 8 34 32 21 51 64    2
After p = 3, 8 32 21 34 51 64    2
After p = 4, 8 21 32 34 51 64    1
After p = 5, 8 21 32 34 51 64    0
```

Bubble Sort – Algorithm

```
for i = 1 to N-1 do
  for j = 1 to N-i do
    if A[j].key > A[j+1].key then
       swap(A[j],A[j+1])
```

Bubble Sort - Improved version

```
for i = 1 to N-1 do
  for j = 1 to N-i do
    if A[j].key > A[j+1].key then
      swap(A[j],A[j+1])
  break if no swapping was done
```

Bubble Sort

- Complexity - $O(n^2)$
- Worst Case: Reverse order, the total number of comparisons
  - $(n-1) + (n-2) + \ldots + 2 + 1 = \frac{(n-1) \times n}{2}$

Bubble Sort – Advantage

- Easy to code, understand and memorize
- Requires little additional space
- $O(n)$ when file is almost completely sorted

Insertion Sort – Introduction

- Simple method
- Commonly used in playing cards
  - Receiving a new card
  - Insert to right position

Insertion Sort – Example

```
Original      34 8 64 51 32 21 # of Moves
------------------------------------------
After p = 1, 8 34 64 51 32 21   1
After p = 2, 8 34 64 51 32 21   0
After p = 3, 8 34 51 64 32 21   1
After p = 4, 8 32 34 51 64 21   3
After p = 5, 8 21 32 34 51 64   4
```

Insertion Sort – Algorithm

```
for p = 2 to N
  for j = p downto 2
    if A[j - 1] > A[j]
      swap(A[j - 1], A[j])
    else break
```

Insertion Sort

- Complexity - $O(n^2)$

Insertion Sort – Advantages

- Similar to those mentioned in Bubble Sort

Selection Sort – Introduction

- aka Straight Selection, Push-Down Sort
- Successive elements are selected in order and placed into their proper sorted positions

Selection Sort – Example

```
Original       34 8 64 51 32 21 # of swaps
----------------------------------------
After p = 1, 8 34 64 51 32 21    1
After p = 2, 8 21 64 51 32 34    1
After p = 3, 8 21 32 51 64 34    1
After p = 4, 8 21 32 34 64 51    1
After p = 5, 8 21 32 34 51 64    1
```

Selection Sort – Algorithm

```
for i = 1 to N - 1
  lowindex = i
  lowkey = A[i]
  for j = i + 1 to N
    if A[j] < lowkey
      lowindex = j
      lowkey = A[j]
  swap(A[i], A[lowindex])
```

Selection Sort

- Complexity - $O(n^2)$

Shell Sort – Introduction

- aka ***Diminishing Increment Sort***
- Consists of phases
- phase k has a number called increment h_k
- The increment h_1 for phase 1 must be 1

For example:

Phase	Increment
4	8
3	4
2	2
1	1

Shell Sort – Overview

- Start from the **highest** phase first
- Proceed to the lowest phase (phase 1)
- For each phase k, we sort the numbers such that
  - For all i, A[i] <= A[i + h_k]
- In fact, we perform an insertion sort on h_k independent sub-arrays
- After each phase, the numbers are h_k-sorted

Shell Sort

77 93 60 53 14 81 38 24 27 92 39 29 88

Phase	Increment
3	5
2	3
1	1

Shell Sort

*Start **phase 3**, as the increment for phase 3 is **5**, therefore we divide the numbers into 5 independent subarrays*

77	93	60	53	14	81	38	24	27	92	39	29	88

Phase	Increment
3	5
2	3
1	1

Shell Sort

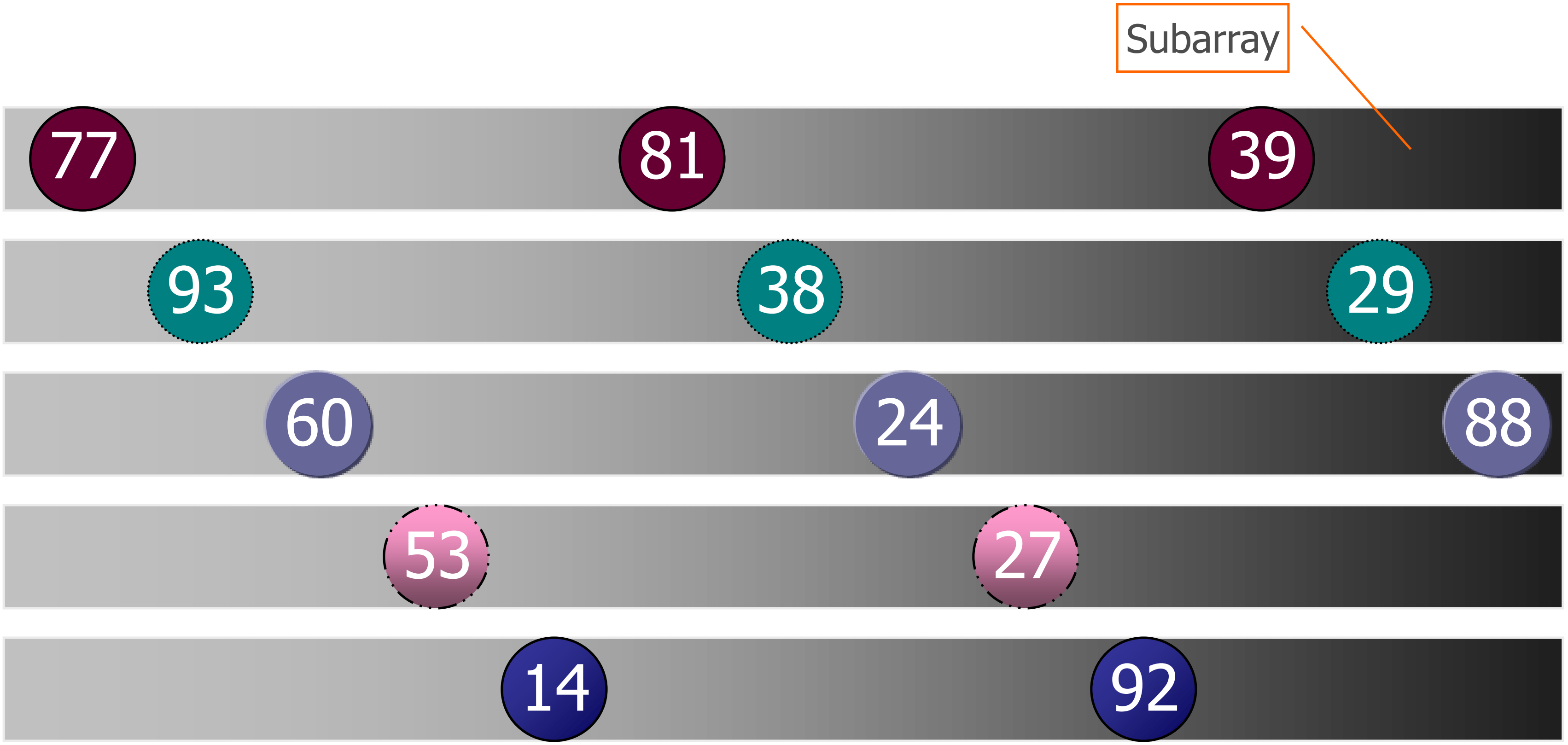
77 93 60 53 14 81 38 24 27 92 39 29 88

*The five subarrays are colored differently. Notice that in each sub-array, the distance between two consecutive elements is **5** (the increment h_3 for **phase 3**)*

77 93 60 53 14 81 38 24 27 92 39 29 88

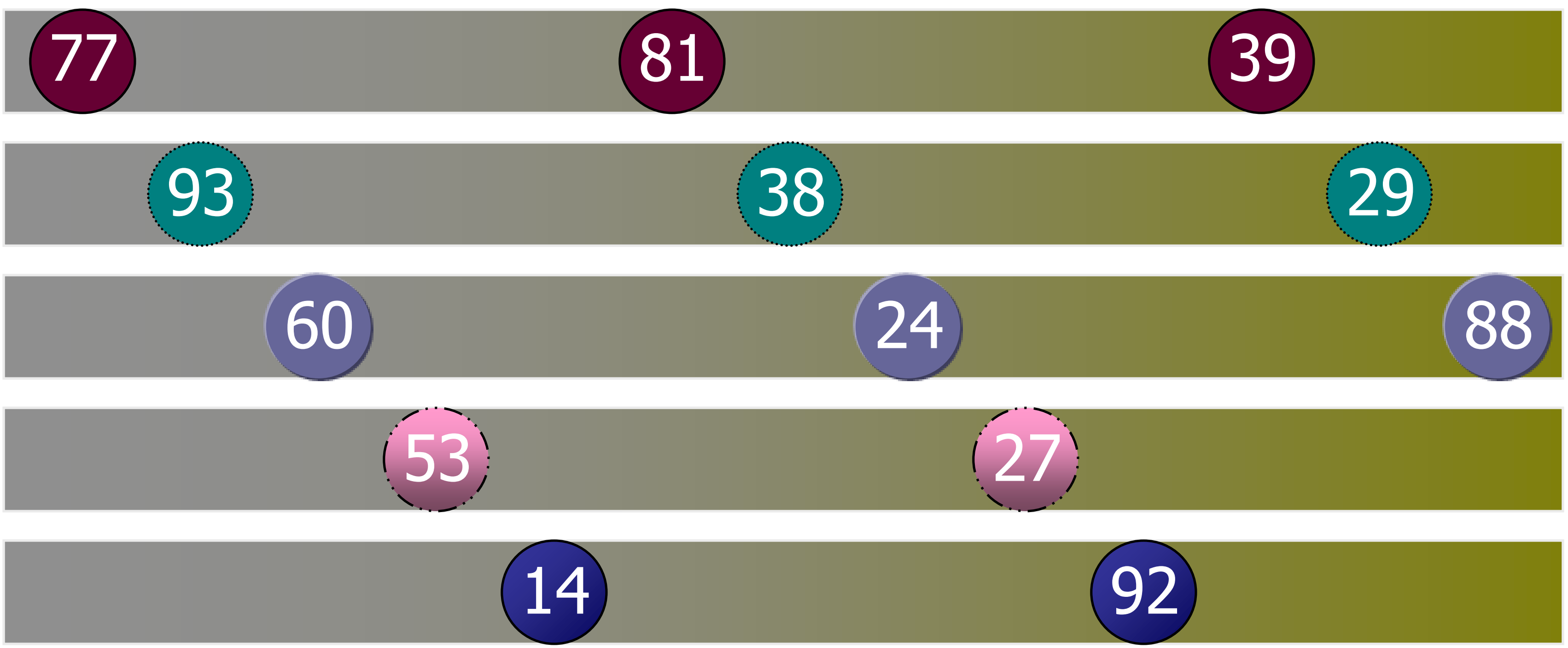
Phase	Increment
3	5
2	3
1	1

Shell Sort



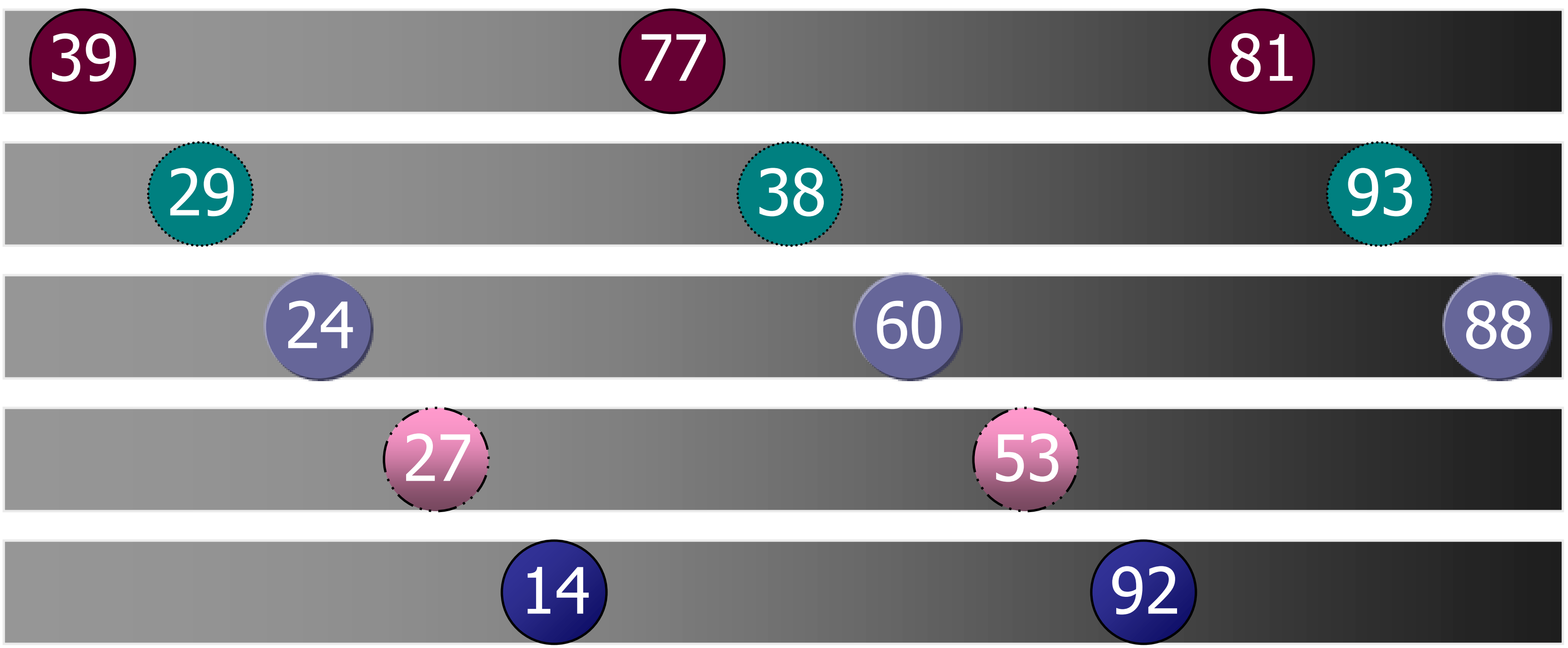
Shell Sort

Now sort each subarray using insertion sort.



Shell Sort

After each sub-array is sorted.



Shell Sort

39	29	24	27	14	77	38	60	53	92	81	93	88

Shell Sort

Phase 3 finished.

39	29	24	27	14	77	38	60	53	92	81	93	88

Shell Sort

*Start **phase 2**, as the increment for phase 3 is **3**, therefore we divide the numbers into 3 independent subarrays*

39 29 24 27 14 77 38 60 53 92 81 93 88

Phase	Increment
3	5
2	3
1	1

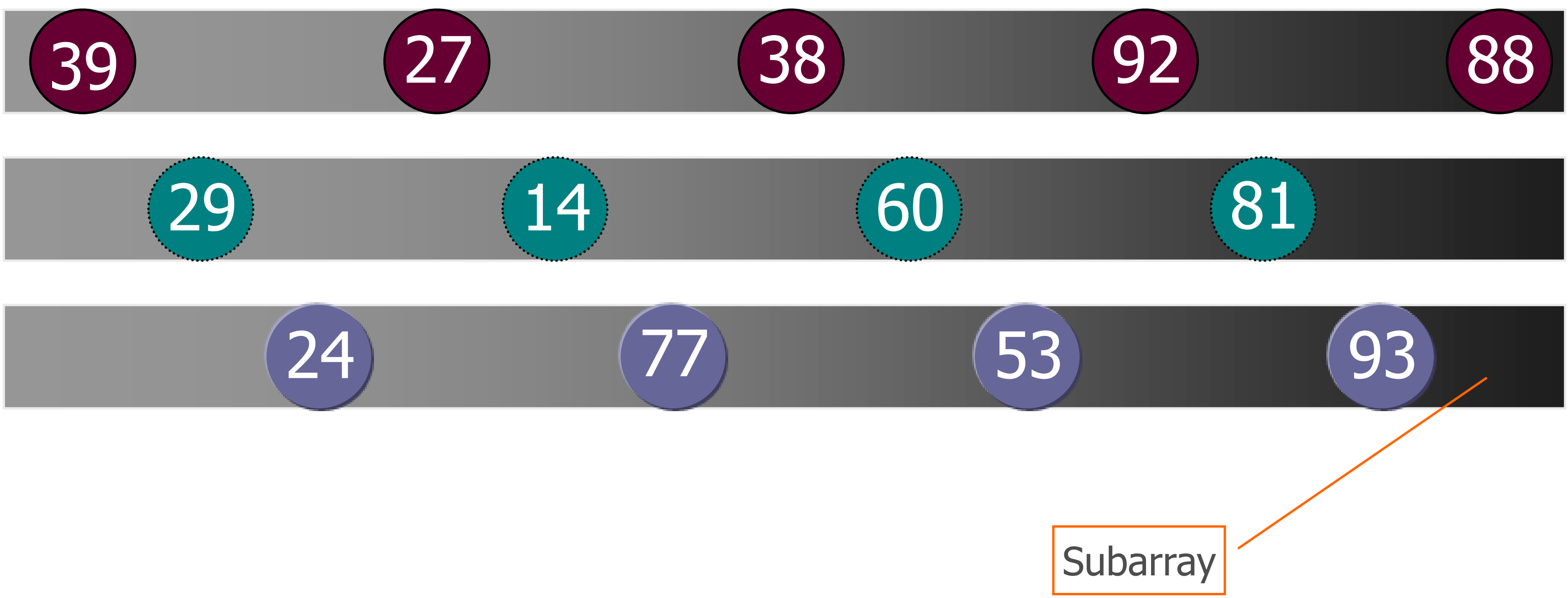
Shell Sort

*The three subarrays are colored differently. Notice that in each subarray, the distance between two consecutive elements is **3** (the increment h_2 for **phase 2**)*

39 29 24 27 14 77 38 60 53 92 81 93 88

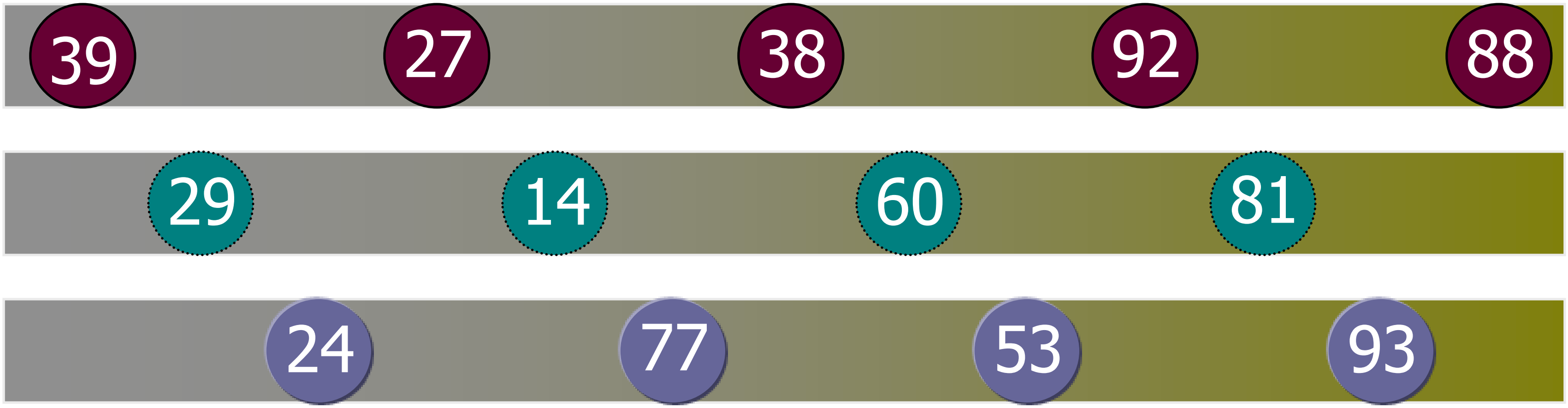
Phase	Increment
3	5
2	3
1	1

Shell Sort



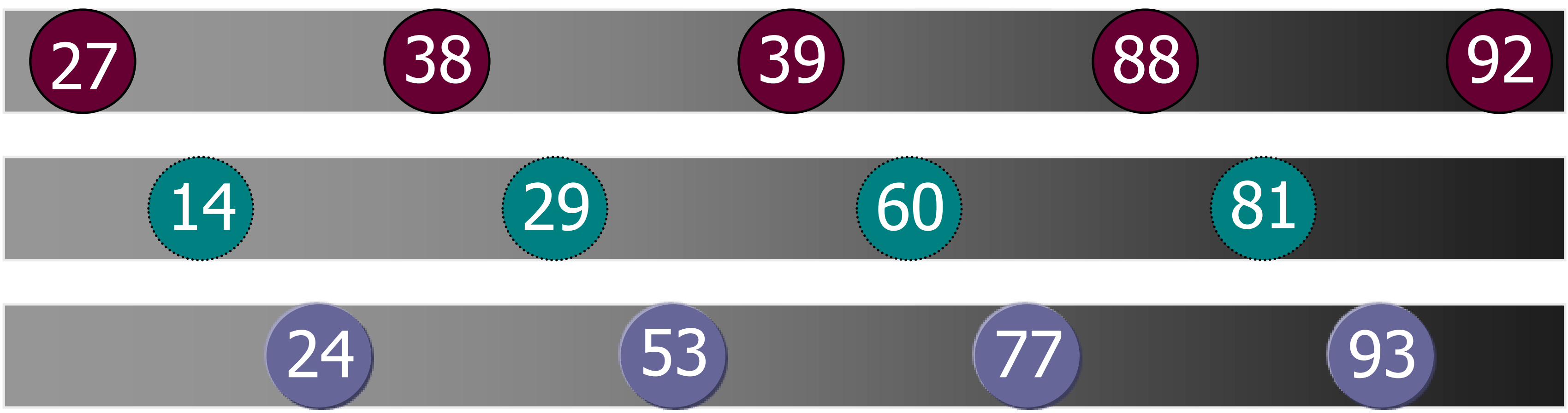
Shell Sort

Now sort each subarray using insertion sort.



Shell Sort

After each subarray is sorted.



Shell Sort

27 14 24 38 29 53 39 60 77 88 81 93 92

Shell Sort

Phase 2 finished.

27	14	24	38	29	53	39	60	77	88	81	93	92

Shell Sort

*Start **phase 1**, as the increment for phase 3 is **1**, therefore the only one subarray is the whole list itself*

27	14	24	38	29	53	39	60	77	88	81	93	92

Phase	Increment
3	5
2	3
1	1

Shell Sort

Now sort using insertion sort.

27 14 24 38 29 53 39 60 77 88 81 93 92

Shell Sort

After the list is sorted. Phase 1 finished.
Shell sort finished !

14 24 27 29 38 39 53 60 77 81 88 92 93

Shell Sort – Code

```
shellsort(int x[], int n){
   int incr, i, j, temp;
   for (incr=n/2;incr>0;incr/=2)
    for (i=incr;i<n;i++){
        temp=x[i];
        for (j=i;j>=incr;j-=incr)
            if (x[j-incr]>temp)
                x[j]=x[j-incr];
            else break;
        x[j]=temp;
    }
   }
}
```

Shell Sort

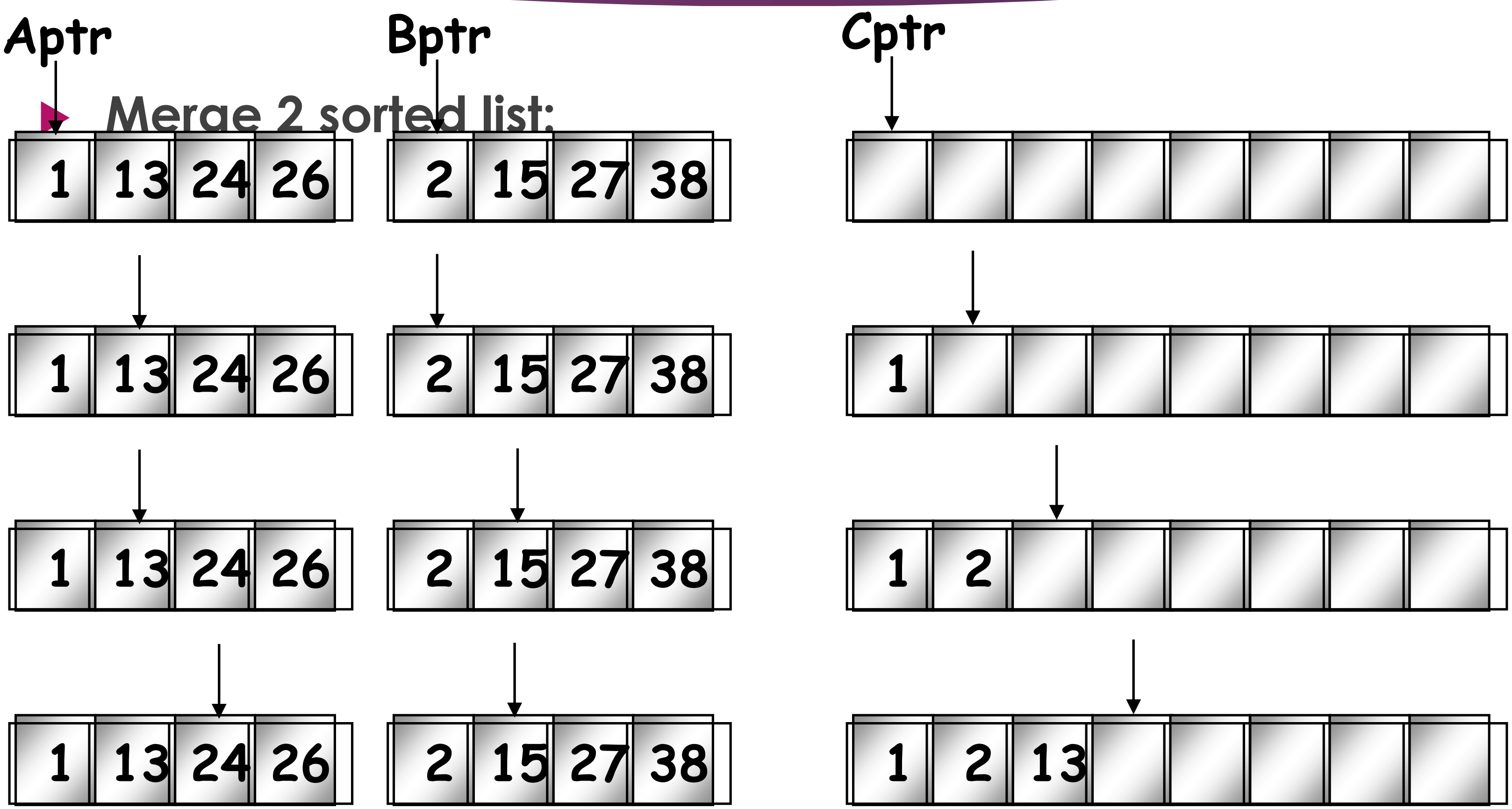
- Average complexity - $O(n^{1.5})$
- Worst case complexity - $O(n^2)$

Merge Sort – Introduction

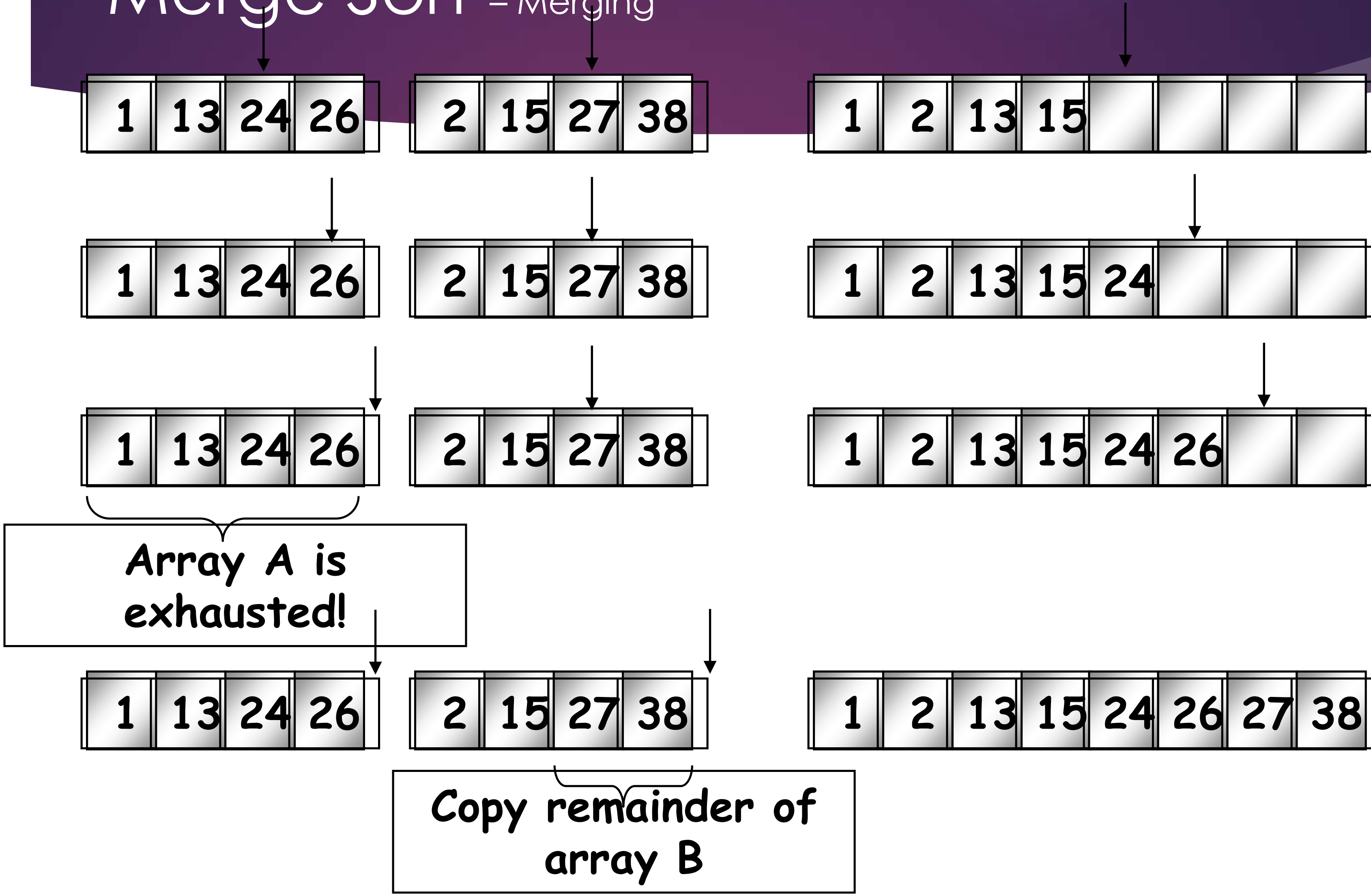
- The basic operation of merge sort is to merge 2 sorted lists
- Divide-and-conquer

Merge Sort – Merging

- Merge 2 sorted list:



Merge Sort – Merging



Merge Sort

- Merge sort follows the divide-and-conquer approach
- **Divide**: Divide the n-element sequence into two (n/2)-element subsequences
- **Conquer**: Sort the two subsequences recursively
- **Combine**: Merge the two sorted subsequence to produce the answer

Merge Sort

- Complexity - $O(n \log n)$

Quick Sort

- Divide-and-conquer process for sorting an array A[p..r]
- Divide: A[p..r] is partitioned into two nonempty subarrays A[p..q] and A[q+1..r] such that each element of A[p..q] is less than each element of A[q+1..r]
- Conquer: The two subarrays A[p..q] and A[q+1..r] are sorted by recursive calls to quicksort.
- Combine: Since the subarrays are sorted in place, no work is needed to combine them

Quick Sort – Algorithm

- Quicksort(S)

  {

  if size of S is 0 or 1 then return;

  pick a pivot v

  divide S - {v} into S_1 and S_2 such that

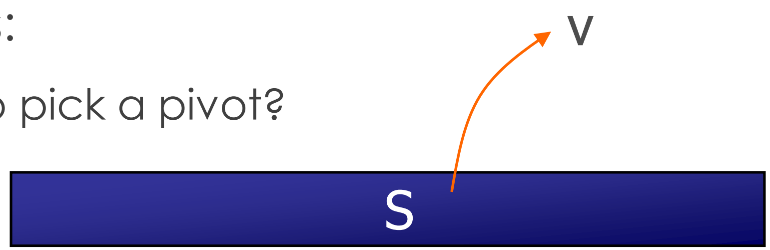
  - **every element in S_1 is <= v**
  - **every element in S_2 is >= v**

  return { Quicksort(S_1) v Quicksort(S_2) }

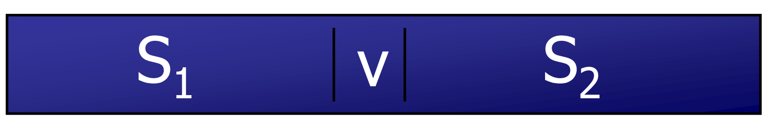
  }

Quick Sort

- Questions:
  - How to pick a pivot?



  - How to divide S into S_1 and S_2 ?



Quick Sort

- How to pick a pivot?
  - First element in S

    Pivot = 42

    For example: 42 34 8 2 6 21 5 32 1

  - Median-of-Three
    - median of first, center and last element

    Pivot = median of 42, 6 and 1 = 6

    For example: 42 34 8 2 6 21 5 32 1

Quick Sort

- How to divide S into S_1 and S_2 ?
  - Based on the value of pivot, we put
    - elements that are <= pivot to the left of pivot
    - elements that are > pivot to the right of pivot
    - For example,

Pivot

10	11	9	15	25	31	43	62	81

    - On the left of pivot: 10, 11, 9, 15 are <= 25
    - On the right of pivot: 31, 43, 62, 81 are > 25

Quick Sort

One of the methods

- Steps:

(1) Swap the first, center and last element in S such that
first element <= center element <= last element

(2) The pivot v is the center element, swap it with the element before the last element

(3) Make two pointers i, j to point to first element and the element before the last elements respectively

(4) Move i pointer in right direction to point to a number > pivot v

(5) Move j pointer in left direction to point to a number <= pivot v

(6) If i pointer is right of j pointer, goto step (7), else swap the elements pointed by i and j and goto step (4)

(7) Swap the element pointed to i with the pivot v (the element before the last element in S)

Quick Sort

Step 1: Swap the first, center and last element in S such that

first element <= center element <= last element

10	11	9	15	5	1	3	2	8

Quick Sort

*Step 1: After swapping. Therefore the **pivot is 8***

5	11	9	15	8	1	3	2	10

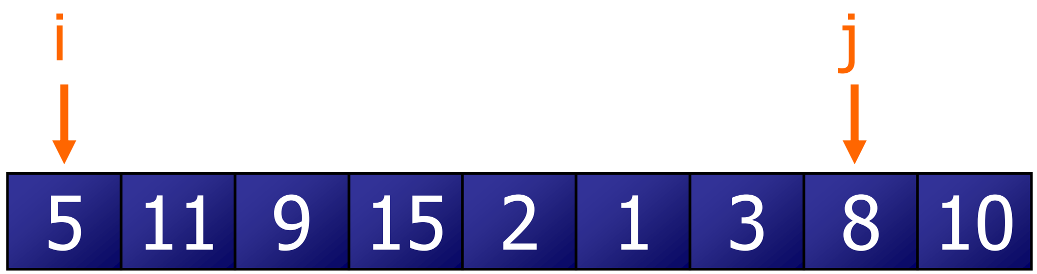
Quick Sort

Step 2: Swap the pivot with the element before the last element.

5	11	9	15	2	1	3	8	10

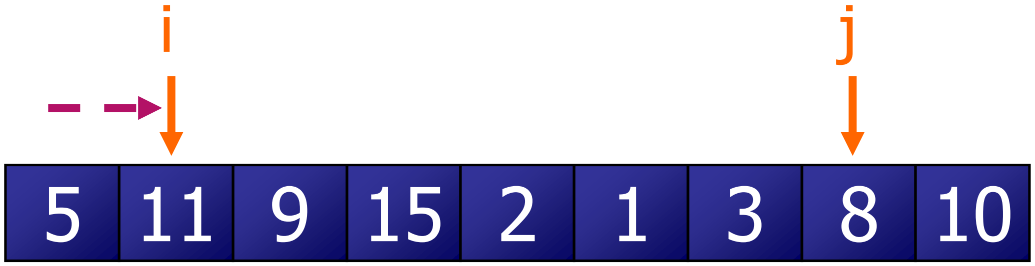
Quick Sort

Step 3: Make two pointers i, j to point to first element and the element just before the last elements respectively



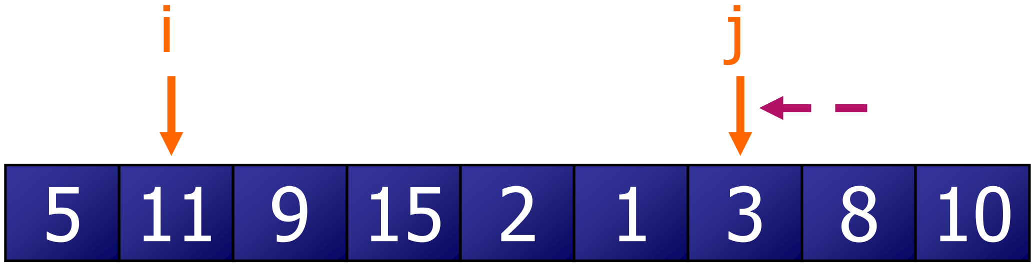
Quick Sort

Step 4: Move i pointer in right direction to point to a number > pivot v



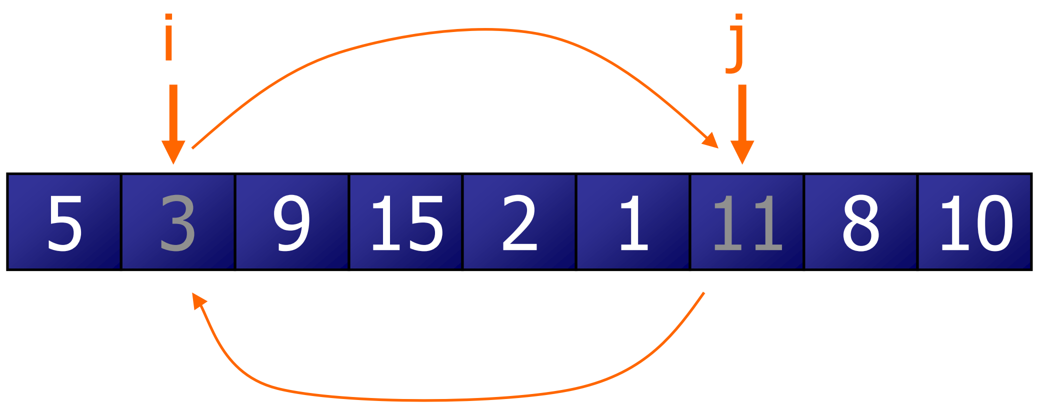
Quick Sort

Step 5: Move j pointer in left direction to point to a number <= pivot v



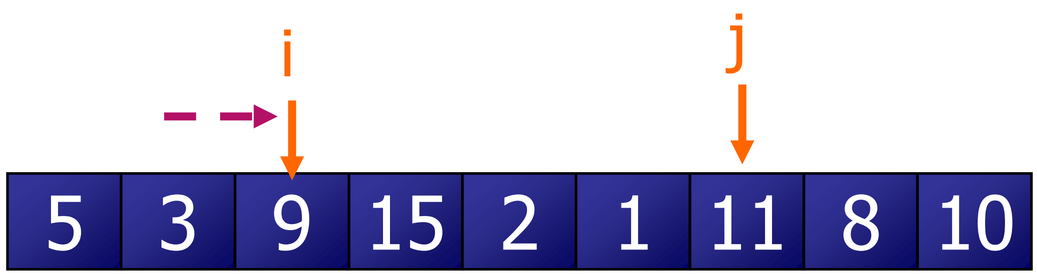
Quick Sort

Step 6: As i pointer is NOT right of j pointer, swap the elements pointed by i and j and goto step (4)



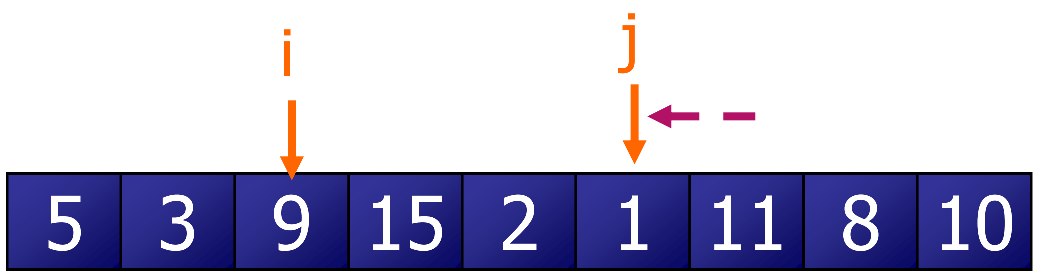
Quick Sort

Step 4: Move i pointer in right direction to point to a number > pivot v



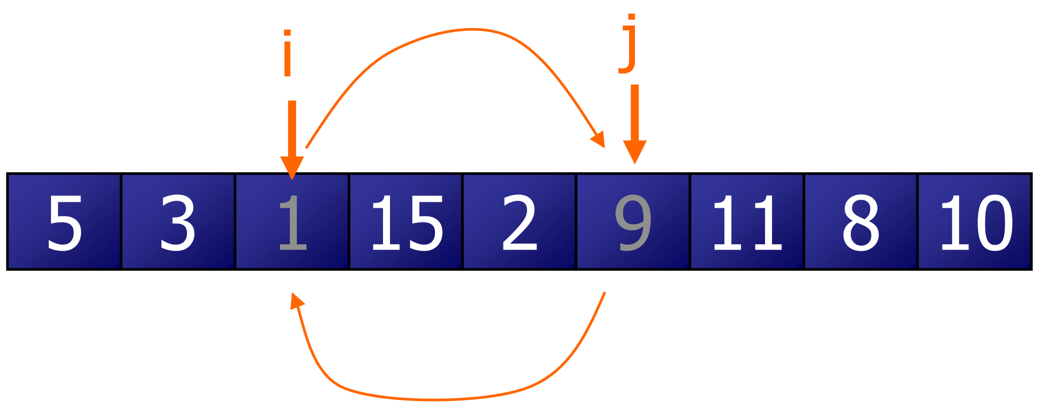
Quick Sort

Step 5: Move j pointer in left direction to point to a number <= pivot v



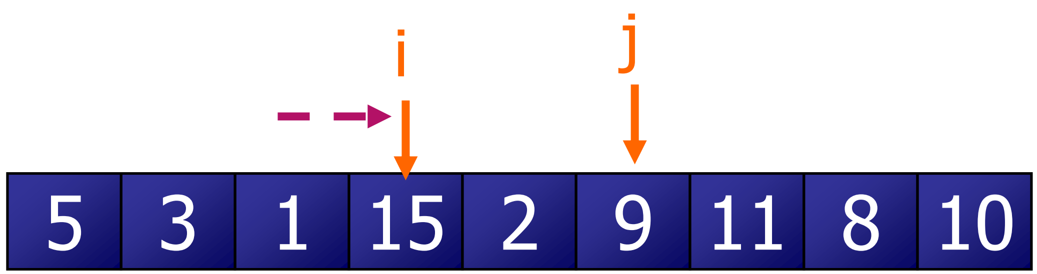
Quick Sort

Step 6: As i pointer is NOT right of j pointer, swap the elements pointed by i and j and goto step (4)



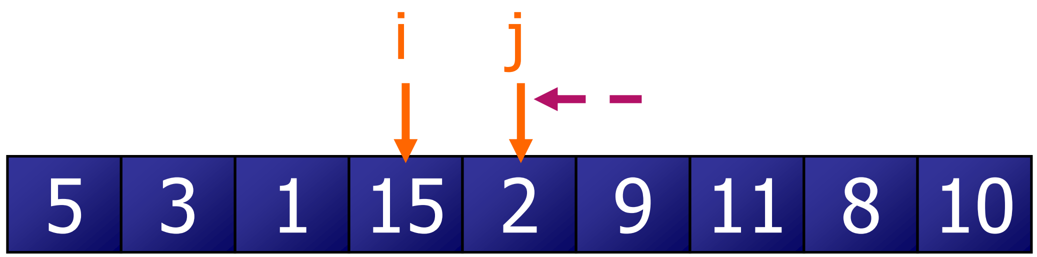
Quick Sort

Step 4: Move i pointer in right direction to point to a number > pivot v



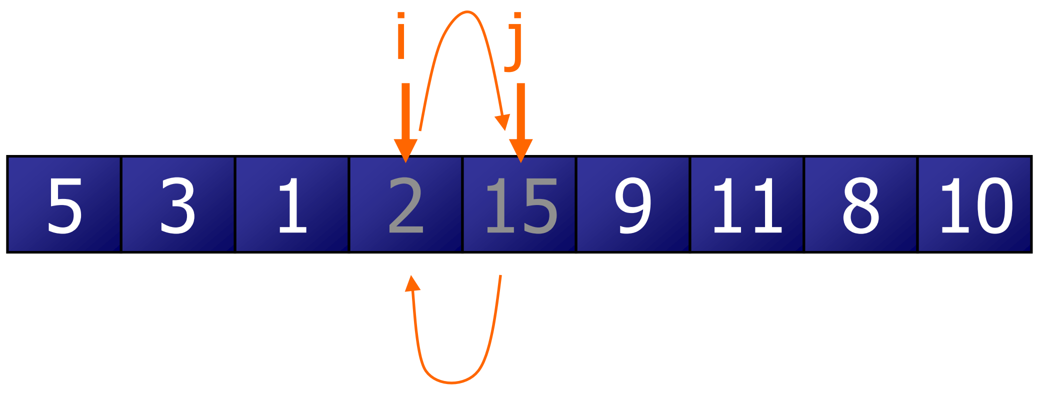
Quick Sort

Step 5: Move j pointer in left direction to point to a number <= pivot v



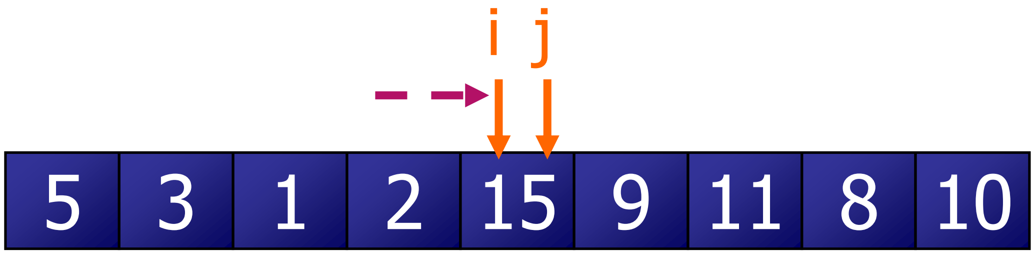
Quick Sort

Step 6: As i pointer is NOT right of j pointer, swap the elements pointed by i and j and goto step (4)



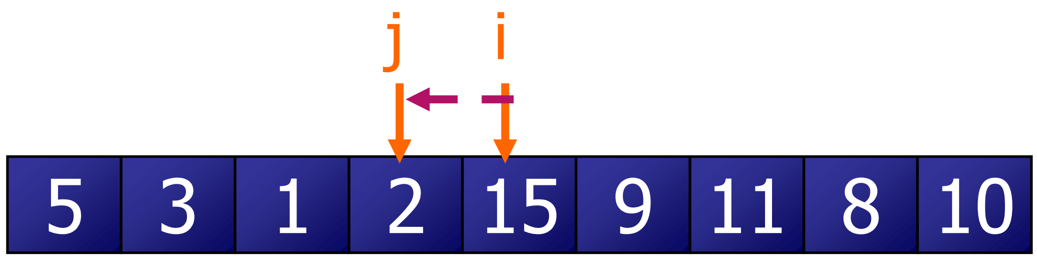
Quick Sort

Step 4: Move i pointer in right direction to point to a number > pivot v



Quick Sort

Step 5: Move j pointer in left direction to point to a number <= pivot v



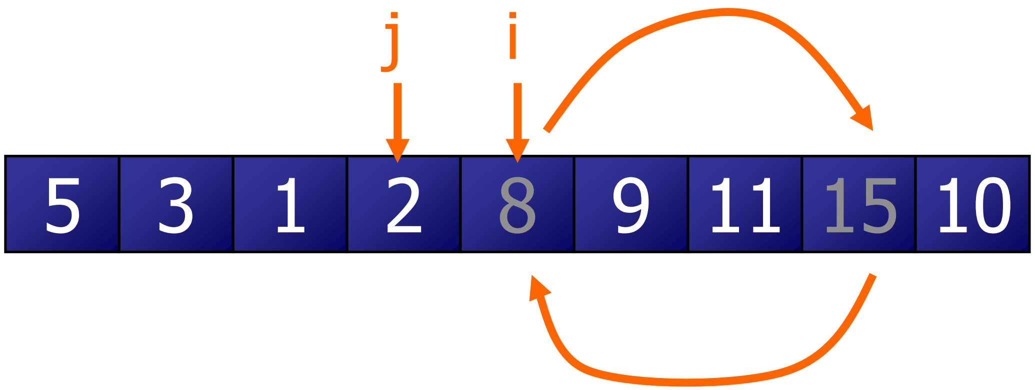
Quick Sort

Step 6: As i pointer is right of j pointer, goto step (7)

			j	i				
5	3	1	2	15	9	11	8	10

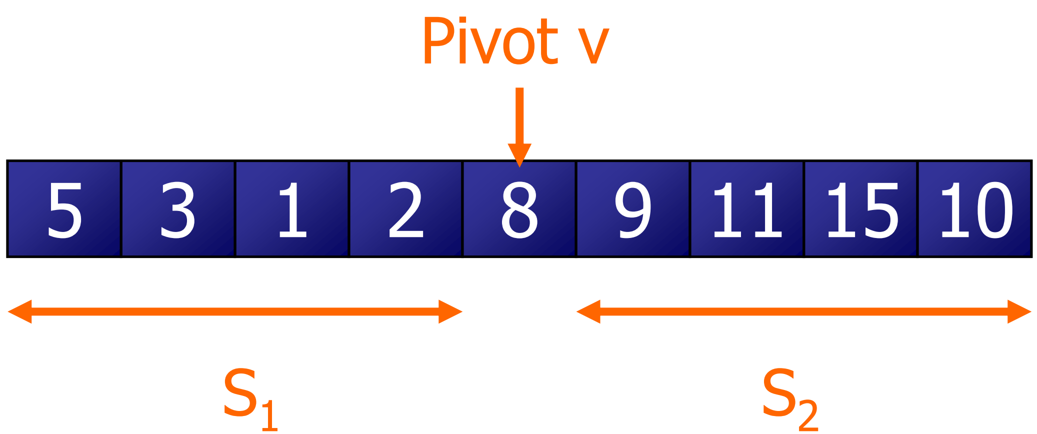
Quick Sort

Step 7: Swap the element pointed to i with the pivot (the element before the last element in S)



Quick Sort

Done!!! You can see that
all elements in S_1 is <= pivot v
and
all elements in S_2 is >= pivot v



Quick Sort

- Complexity?
- Best: $O(n \log n)$
- Worst: $O(n^2)$
- Average: $O(n \log n)$

Comparison Sorting – Summary

- $O(n^2)$
- $O(n^{1.5})$
- $O(n \log n)$
- Can it be faster?
  - Nope

Counting Sort – Introduction

- Assume 0<A[i]<=M
- Algorithm:

  initialize an array Count of size M to all 0's

  for i=1 to N

  inc(Count[A[i]])

  Scan the Count array, and printout the sorted list

Counting Sort

- Complexity - $O(M + N)$
- It can be used for small integers and limited by M

Radix Sort – Introduction

- aka Card Sort.
- Algorithm:

  initialize an array of 10 buckets to empty

  for i=1 to N

  read A[i] and place it into the bucket the last digit

  Use the same process to sort the second last digit

  repeat until the first digit

Radix Sort

- Sort into buckets on the least significant digit first, then proceed to the most significant digit

Initial: 64, 8, 216, 512, 27, 729, 0, 1, 343, 125

By the least significant digit:

0	1	512	343	64	125	216	27	8	729
0	1	2	3	4	5	6	7	8	9

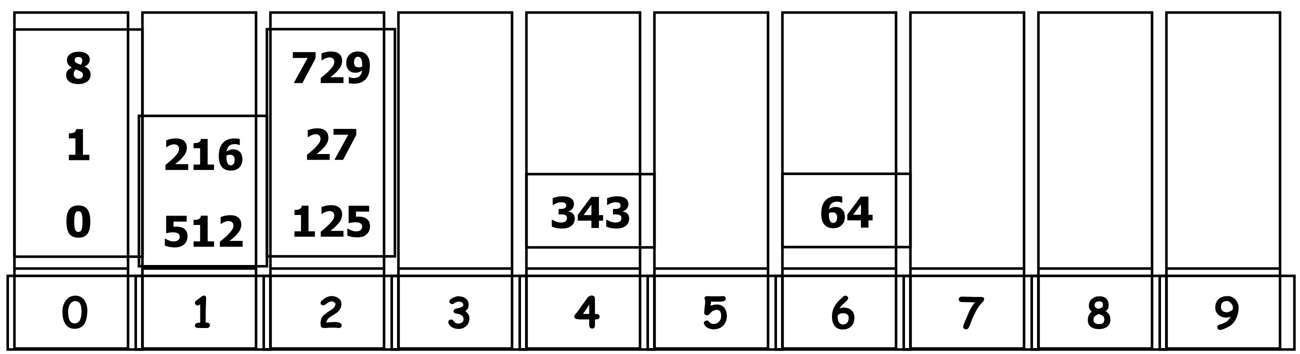
After First Pass: 0, 1, 512, 343, 64, 125, 216, 27, 8, 729

Radix Sort

- Second Pass:

1st Pass Result: 0, 1, 512, 343, 64, 125, 216, 27, 8, 729

By the tens digit:



After Second Pass: 0, 1, 8, 512, 216, 125, 27, 729, 343, 64

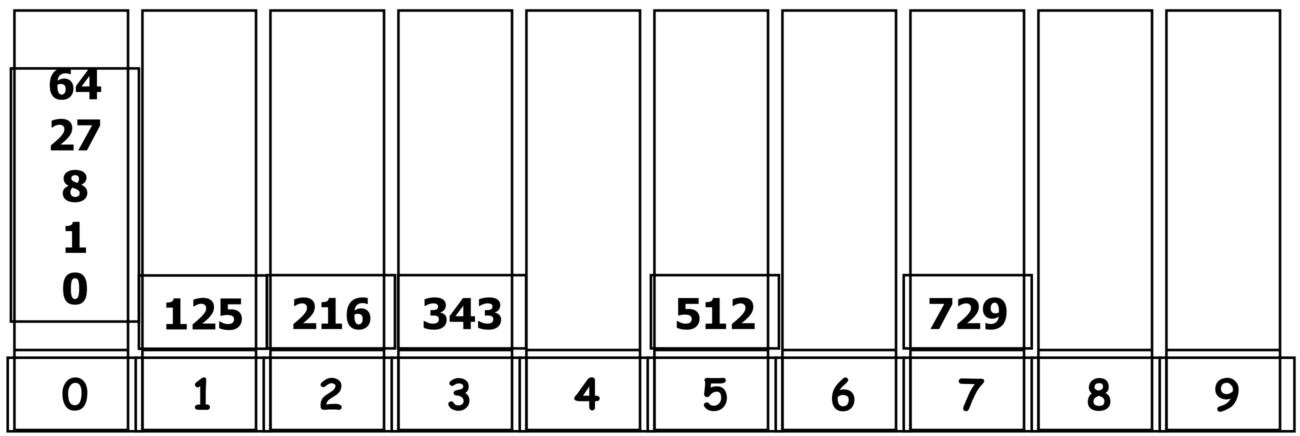
Radix Sort

- Third Pass:

2nd Pass Result:

0, 1, 8, 512, 216, 125, 27, 729, 343, 64

By the most significant digit:



After Last Pass:

0, 1, 8, 27, 64, 125, 216, 343, 512, 729

Radix Sort

- Complexity: $O(d \times n)$, d is the number of digits

STL Support

- Searching
  - lower_bound()
  - upper_bound()
- Sorting
  - sort()