University of Science, VNU-HCM
Faculty of Information Technology

Data Structure and Algorithm

Sort Algorithms

Lecturer: Lê Ngọc Thành
Email: lnthanh@fit.hcmus.edu.vn

HCM City

Contents

- What is sorting?
- Why care to sort?
- Sorting application
- Sorting Types
- Implement
- Sorting Algorithms

What is sorting?

- Need to arrange groups of people in ascending order of height, how to do this and what the results will be?

What is sorting?

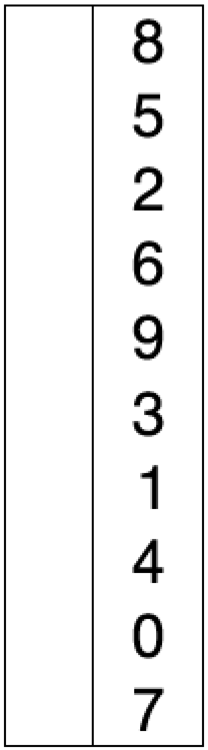
Is it what you think?

How did you do that?

How many steps can you complete?

What is sorting?

- ***Sorting*** is the process of placing the elements of a list in a specified order.



Why care to sort?

- *Because:*
  - Sorting is a fundamental building block in many other algorithms.
  - n the history of development, computers spent more time sorting than doing anything else. According to [Knu73b], a quarter of all mainframe running cycles are used for sorting.
  - Most of the great ideas in designing the algorithm come from sorting like divide-and-conquer, random algorithms ...

Knu73b: D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading MA, 1973.

Why care to sort?

n	$n^2/4$	$n \lg n$
10	*25*	33
100	2,500	*664*
1,000	250,000	*9,965*
10,000	25,000,000	*132,877*
100,000	2,500,000,000	*1,660,960*

Sorting algorithms of different complexity can be performed at very different times.

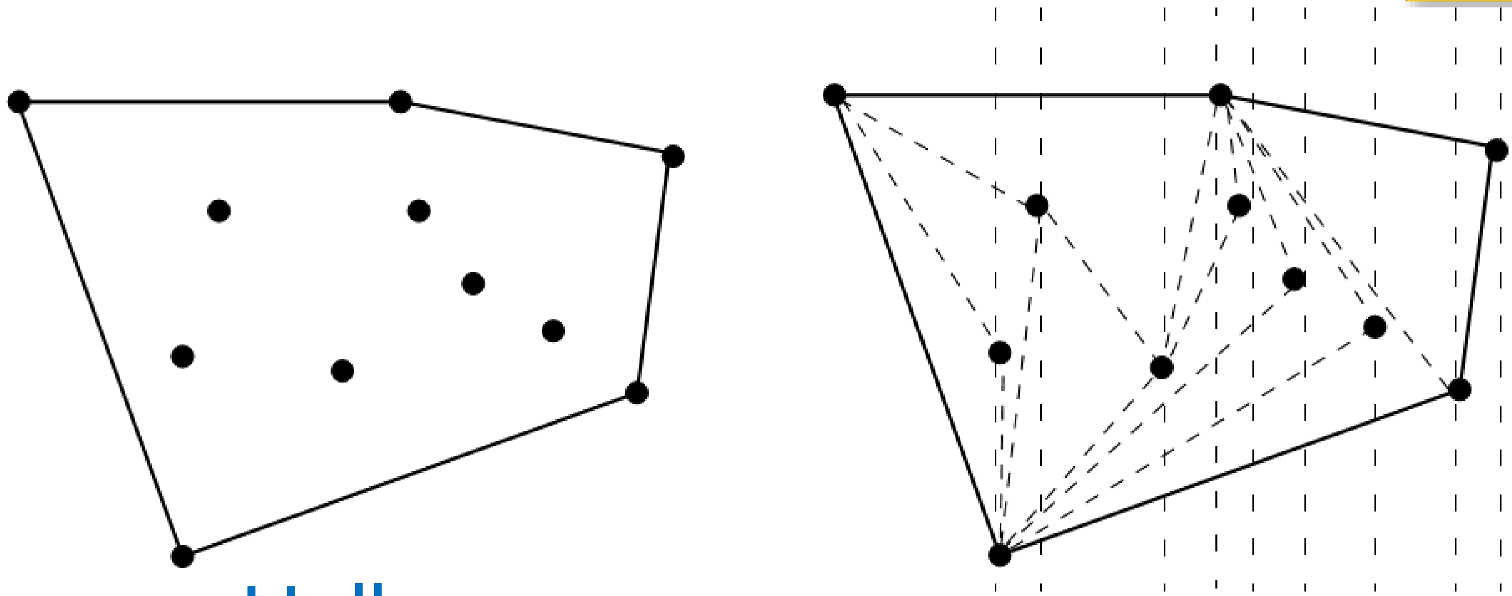
Sort Applications

- Searching:
  - Binary search allows searching for an item in the list with complexity $O(\log n)$ when the array is sorted. Whereas sequential search takes $O(n)$.
- The closest pair:
  - Given n numbers, how can I find a pair of numbers with the smallest difference? When sorted, this closest pair will be adjacent to each other in the list, so when searching sequentially, complexity $O\ (n\ \log n)$ includes sorting.

Sort Applications

- Check duplicate:
  - Need to check whether there are duplicates in the list of n elements? The most effective algorithm is to sort them and traverse them sequentially to check for a zero-distance adjacent pair.
- Element frequency:
  - For n elements, determine the number of occurrences for each element?
- The k^{th} largest element:
  - Find the k^{th} largest element in the array?

Sort Applications



- Convex Hull:
  - Given n points in two-dimensional space, what is the smallest polygon to contain all of them?
  - Arranges the elements in ascending order by the x coordinate, the left most and right element is definitely on the polygon. Subsequent points are considered based on these points.

Sort Applications

- Listing the practical applications you see that use sort?
  – List of classes by Id, or full name
  – Sort the countries by population
  – Sort results in search engines
  – ...

Sorting algorithm structure

- Input:
  - Array A consists of n elements
- Output:
  - A permutation of A such that: $A_0 \le A_1 \le \cdots \le A_{n-1}$ (ascending order)
- Basic operation:
  - Compare
  - Swap (reposition two elements)

Sorting Types

In memory sorting				External sorting
Comparison sorting $\Omega(N \log N)$		Specialized Sorting		
$O(N^2)$	$O(N \log N)$	$O(N)$	# of tape accesses	
• Bubble Sort • Selection Sort • Insertion Sort • Shell Sort	• Merge Sort • Quick Sort • Heap Sort	• Bucket Sort • Radix Sort	• Simple External Merge Sort • Variations	

Implement

Algorithm	Worst-case running time
Insertion sort	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$
Quicksort	$\Theta(n^2)$
Counting sort	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$

Implement

Bubble sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Yes	Exchanging
Cocktail sort	—	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Yes	Exchanging
Comb sort	—	—	$\mathcal{O}(1)$	No	Exchanging
Gnome sort	—	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Yes	Exchanging
Selection sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	No	Selection
Insertion sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Yes	Insertion
Shell sort	—	$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(1)$	No	Insertion
Binary tree sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	Yes	Insertion
Library sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	Yes	Insertion
Merge sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	Yes	Merging
-place merge sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	No	Merging
Heapsort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	No	Selection
Smoothsort	—	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	No	Selection
Quicksort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(\log n)$	No	Partitioning
Introsort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	No	Hybrid
Patience sorting	—	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	No	Insertion & Selection
Strand sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	Yes	Selection

Wikipedia

Other problems

- *Should be sorted increase or decrease?*
- *Sort only the key value or an entire record?*
  - A record: name, address, phone number, …
- *What to do with duplicate values?*
  - Whether it can be viewed as a single key and arranged as usual or grouped together.
- *If the data is not numeric?*
  - String is arranged in alphabet?

Contents

- What is sorting?
- Why care to sort?
- Sorting application
- Sorting Types
- Implement
- **Sorting Algorithms**

Sorting Algorithms

- Selection Sort
- Insertion Sort
- Interchange Sort
- Bubble Sort
- Shaker Sort
- Binary Insertion Sort
- Shell Sort
- Heap Sort
- Quick Sort
- Merge Sort
- Radix Sort

Selection Sort

Idea:

- Finds the element that satisfies the requirements (minimum, maximum ...) from the current position to the end of the array.
- Swap these two elements.

Steps:

- S1: $i = 0$;
- S2: Find the position of the min/max element from i to n-1;
- S3: Swap.
- S4: $i = i + 1$.

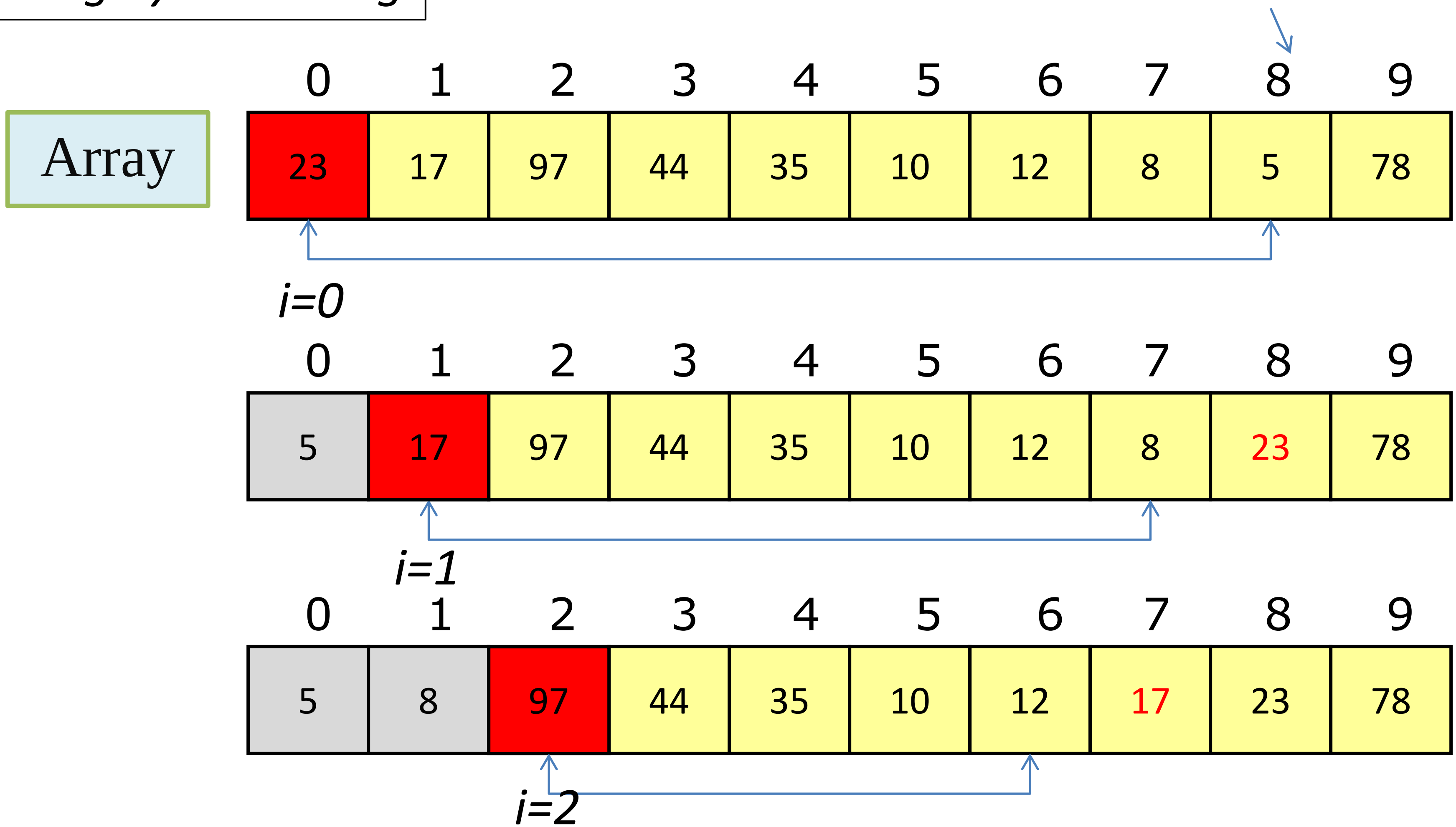
  If $i < n-1$ go to S2.

  Otherwise, end.

Selection Sort

Sorting by ascending

min=8 is position of smallest element



Selection Sort

```
for (int i = 0; i < n -1; i ++){
    int min = i;
    for (int j = i + 1; j < n; j ++)
            if (a[min] > a[j])
                    min  = j;
    swap (a[i],a[min]);
}
```

Comments

- Advantages:
  - Ease of implementation
  - In-place sorting (does not require additional space)
- Disadvantage:
  - High complexity: $O(n^2)$

Sorting Algorithms

- Selection Sort
- Insertion Sort
- Interchange Sort
- Bubble Sort
- Shaker Sort
- Binary Insertion Sort
- Shell Sort
- Heap Sort
- Quick Sort
- Merge Sort
- Radix Sort

Insertion Sort

Idea:

Suppose that $a_0,\ldots, a_i$ has the order, find the position to insert element a_{i+1} into that sequence such that it still has order.

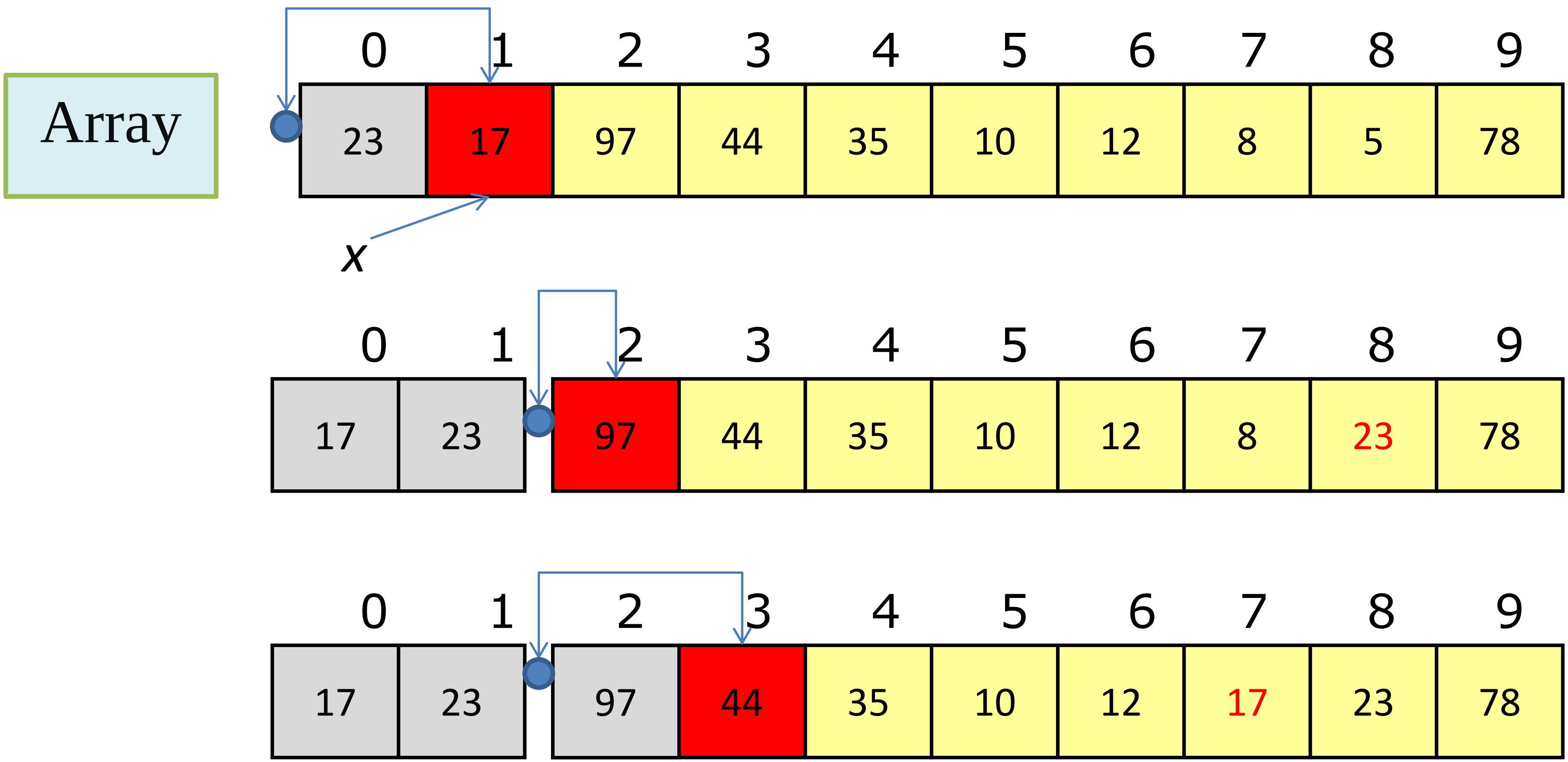
Steps:

- S1: i = 1; (a[0] sorted because there is only 1 element).
- S2: x = a[i];
- S3: Find position pos to insert x into the array from a [0] to a [i-1];
- S4: Move the elements from a [pos] to a [i-1] to the right by 1 position to accommodate the insertion of x in this pos position.
- S5: a[pos] = x;
- S6: i = i + 1 ; If i < n, go to S2.

  Otherwise, go to end.

Insertion Sort

Sorting by ascending



Insertion Sort

Sorting by ascending

```
for (i ←1 to n-1) do
    x ← a[i];
    pos ← i -1;
    while ( pos ≥ 0 && a[pos] > x) do
            a[pos + 1] = a [pos];
            pos ← pos – 1;
    end while
    a[pos + 1] = x;
end for
```

Comments

- Advantages:
  - Ease of implementation
  - In-place sorting (does not require additional space)
  - Real-time sorting, data may be incomplete or coming, but the array is still sortable.
- Disadvantage:
  - High complexity: $O(n^2)$

Sorting Algorithms

- Selection Sort
- Insertion Sort
- Interchange Sort
- Bubble Sort
- Shaker Sort
- Binary Insertion Sort
- Shell Sort
- Heap Sort
- Quick Sort
- Merge Sort
- Radix Sort

Interchange Sort

Idea:

- For each position in the array a, swap with the following elements if in wrong order.

Steps:

- S1: i = 0;
- S2: Go through each element j following i;
- S3: If a[i] and a[j] are in the wrong order, swap them;
- S4: i = i + 1 ;

  If i < n-1, go back to S2.

  Otherwise, go to end.

Interchange Sort

Sorting by ascending

Array

0	1	2	3	4	5	6	7	8	9
23	17	97	44	35	10	12	8	5	78

Wrong order: 23 > 17

0	1	2	3	4	5	6	7	8	9
17	23	97	44	35	10	12	8	5	78

OK

0	1	2	3	4	5	6	7	8	9
17	23	97	44	35	10	12	8	5	78

Wrong order: 17 > 10

Interchange Sort

Sorting by ascending

```
for (i ←0 to n-2) do
    for (j ←i+1 to n-1) do
        if (a[i] > a[j]) then a[i] ↔ a[j]
    end
end
```

Sorting Algorithms

- Selection Sort
- Insertion Sort
- Interchange Sort
- Bubble Sort
- Shaker Sort
- Binary Insertion Sort
- Shell Sort
- Heap Sort
- Quick Sort
- Merge Sort
- Radix Sort

Bubble Sort

Idea: small values "bubble" up to the top of list

- Begin from the end of the array, in turn, swap two adjacent elements if they are in the wrong order.
- The lightest element will float to the top of the array, the next step doesn't take it into account.

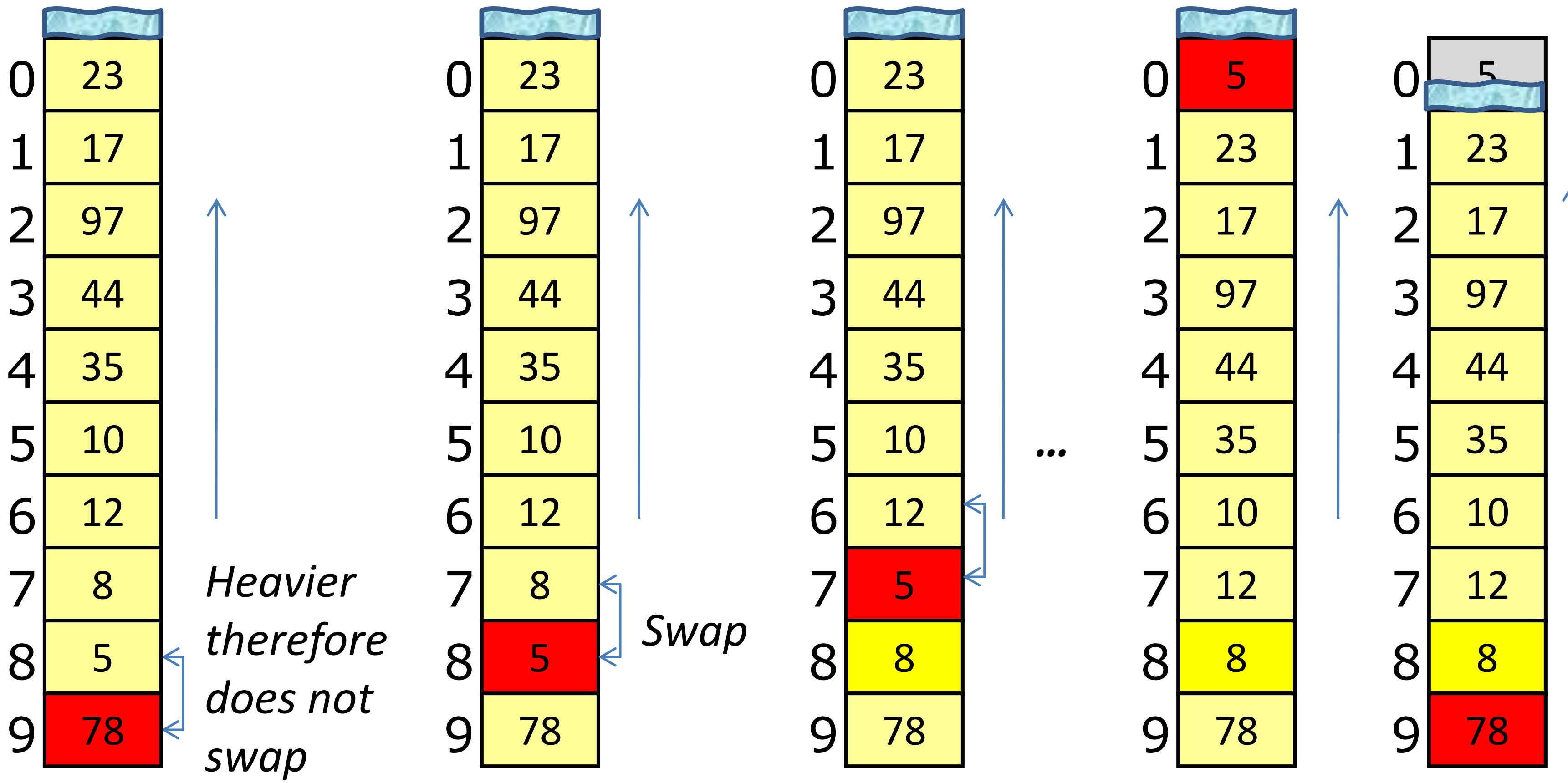
Steps:

- S1: i = 0;   //the floating surface
- S2: j = n-1; //begin from end of the array
- S3: If a[i] and a[j-1] are in the wrong order, swap them; //bubble
- S4: j = j -1; If j > i, go back to S3. // "bubble" up to the top of list
- S5: i = i + 1 ; If i < n-1, go back to S2.

  Otherwise, go to end.

Bubble Sort

Sorting by ascending



Bubble Sort

Sorting by ascending

```
for (i ←0 to n-2) do
    for (j ←n-1 to i+1) do
            if  (a[j-1] > a[j]) then a[j-1] ↔ a[j]
    end
end
```

Comments

- Advantage:
  - Ease of implementation
- Disadvantage:
  - High complexity: $O\ (n$ ^ 2), even in the best case

→ improved algorithm: let the surface drop to position where the last swapping occurs.

Sorting Algorithms

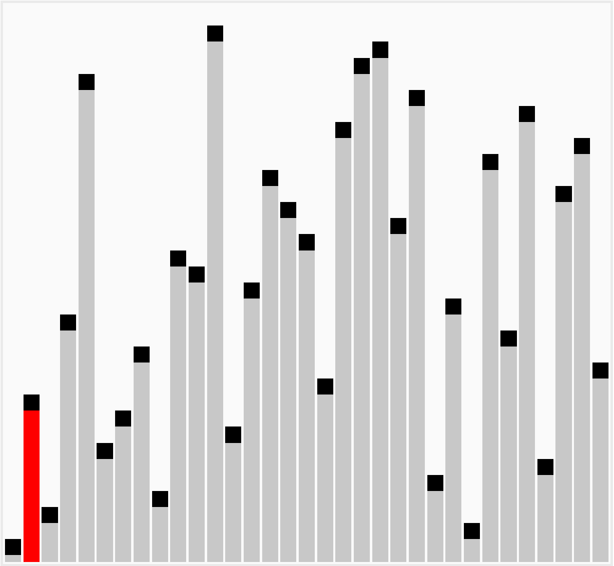
- Selection Sort
- Insertion Sort
- Interchange Sort
- Bubble Sort
- Shaker Sort
- Binary Insertion Sort
- Shell Sort
- Heap Sort
- Quick Sort
- Merge Sort
- Radix Sort

Shaker Sort

Bidirectional Bubble Sort, Cocktail Sort.

Idea: the light elements will float, the heavy ones will sink

- Similar to Bubble sort, but in addition to sinking heavy elements.
- There are also improvements, reducing redundant comparisons by:
  + The floating surface for the next stage will be where the last floating swapping occurred.
  + The sink bottom for the next stage will be where the last sink swapping occurs.



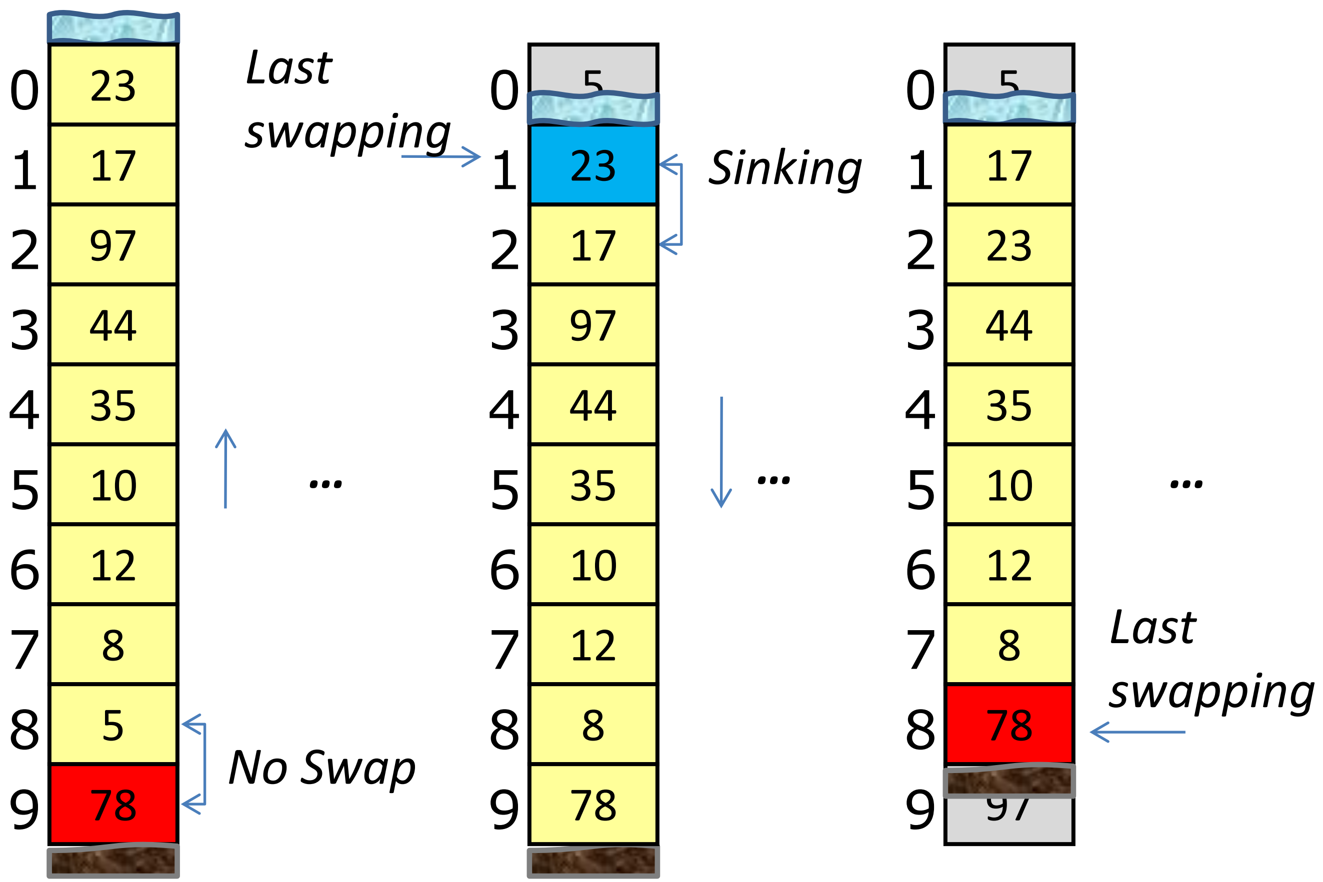
Shaker Sort

Steps:

- S1: surface = 0; //the floating surface
  bottom = n-1; //the sinking bottom
  k = n-1; //saves the location where the last swapping occurred
- S2: j = bottom; //push the light element up from the bottom
  - S2a: If a[j] and a[j-1] are in wrong order, swap them; //floating
    and k = j; // save where the permutation occurs
  - S2b: j = j -1; If j > surface, go to S2. // floating to the surface
- S3: surface = k; // the new surface is the last swapping because the previous sequence is ordered
- S4: j = surface;
  - S4a: If a[j] and a[j+1] are in wrong order, swap them; //sinking
    and k = j;
  - S4b: j = j +1; If j < bottom, go to S4.
- S5: bottom = k;
- S6: If surface < bottom, go to S2. Otherwise, go to end.

Shaker Sort

Sorting by ascending



Shaker Sort

Sorting by ascending

```
surface ←0; bottom ← n-1; k ← n-1;
while (surface < bottom) do
    for (j ← bottom to surface +1) do
            if  (a[j-1] > a[j]) then
                        a[j-1] ↔ a[j];
                        k ← j;
            end if
     surface ← k;
     for (j ← surface to bottom-1) do
            if  (a[j] > a[j+1]) then
                        a[j+1] ↔ a[j];
                        k ← j;
            end if
     bottom ← k;
end while
```

Sorting Algorithms

- Selection Sort
- Insertion Sort
- Interchange Sort
- Bubble Sort
- Shaker Sort
- Binary Insertion Sort
- Shell Sort
- Heap Sort
- Quick Sort
- Merge Sort
- Radix Sort

Binary Insertion Sort

Idea:

- In the insertion sort algorithm, finding the insert position is sequential. For a better way, we use the binary search method to find this position.

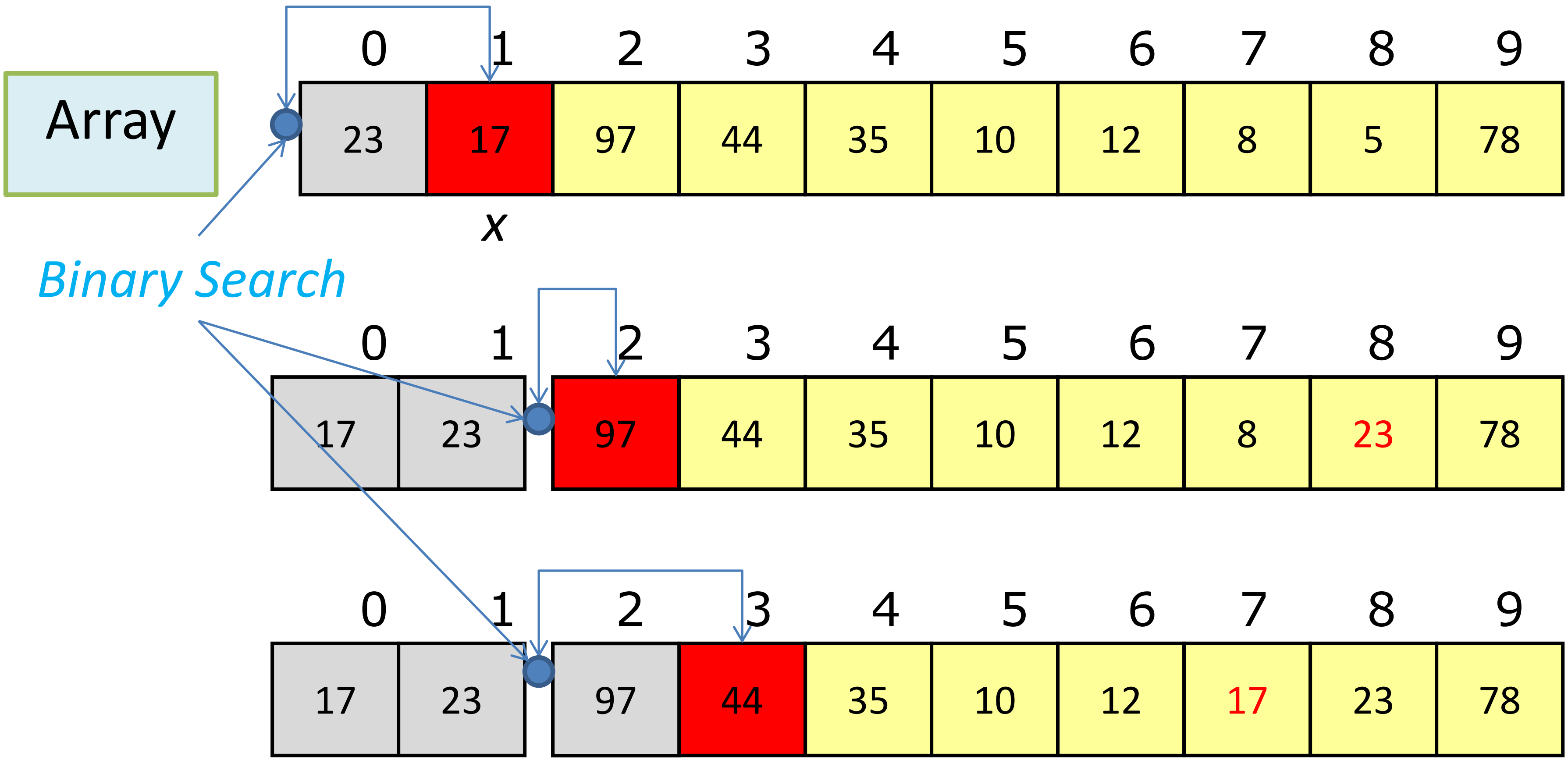
Steps:

- S1: i = 1; (a[0] sorted because there is only 1 element).
- S2: x = a[i];
- S3: Find position pos to insert x into the array from a [0] to a [i-1] *by binary search*
- S4: Move the elements from a[pos] to a[i-1] to the right by 1 position to accommodate the insertion of x in this pos position.
- S5: a[pos] = x;
- S6: i = i + 1 ; If i < n, go to S2.

  Otherwise, go to end.

Binary Insertion Sort

Sorting by ascending



Binary Insertion Sort

Sorting by ascending

```
for (i ←0 to n-1) do
    x ← a[i];
    //find the position to insert x using the binary search method
    pos ← BinarySearch(a,i,x);
    //move the elements backwards to create spaces
    for (j ← i to pos+1) do
            a[j] = a [j-1];
    end for
    a[pos] = x;
end for
```

Binary Insertion Sort

```
function BinarySearch(...,k,x)
    // k is the limit for the search position, x is the value to be compared
    left ← 0; right ← k;
    do
            mid ← (left+right)/2;
            if (x ≤ a[mid]) then right ← mid – 1;
            else      left ← mid + 1;
            end if
    while ( left ≤ right);
    return mid;
end function
```

Sorting Algorithms

- Selection Sort
- Insertion Sort
- Interchange Sort
- Bubble Sort
- Shaker Sort
- Binary Insertion Sort
- Shell Sort
- Heap Sort
- Quick Sort
- Merge Sort
- Radix Sort

Shell Sort

<u>Idea:</u>

- Divide the original array into sub arrays of elements h positions apart.
- Sorts elements in sub arrays.
- Reduce the distance h to form a new sub array. Stop when h = 1.

<u>Steps:</u>

- S1: Initialize shell size $h_0, h_1, \ldots, h_{k-1}$ such that $h_i > h_{i+1}$ and $h_{k-1} = 1$;
- S2: i = 0;
- S3: Divide the original array into sub arrays with h[i] spacing. Sort each sub-array using the insertion sort.
- S4: i = i + 1 ; If i < k, go to S2.

  Otherwise, go to end.

Shell Sort

Sorting by ascending

Shell sizes are 3, 1

<u>*h=3*</u>

0	1	2	3	4	5	6	7	8	9
23	17	97	44	35	10	12	8	5	78

Sort each sub-array

0	1	2	3	4	5	6	7	8	9
12	8	5	23	17	10	44	35	97	78

<u>*h=1*</u>

0	1	2	3	4	5	6	7	8	9
12	8	5	23	17	10	44	35	97	78

…

Shell Sort

Sorting by ascending

```
Choose h[0], h[1], …, h[k-1]=1;
for (step ← 0 to k-1) do
    len = h[step];
    for (i ← len to n-1) do
        x = a[i];
        j = i – len; // j is the preceding position in the same sub-sequence
        //sort the sub-array containing x by insertion sort
        while ( j ≥ 0 && x < a[j]) do
            a[j+len] = a[j];
            j = j – len;
        end while
        a[j+len] = x;
    end for
end for
```

Sorting Algorithms

- Selection Sort
- Insertion Sort
- Interchange Sort
- Bubble Sort
- Shaker Sort
- Binary Insertion Sort
- Shell Sort
- Heap Sort
- Quick Sort
- Merge Sort
- Radix Sort

Heap Sort

Idea:

- When finding the smallest element in step i, the insertion sort does not take advantage of the information obtained by the comparisons in step i-1.

- Use Heap tree to solve the above problem.

Heap Tree:

- Heap is a binary tree: if B is a child of A then key (B) $\leq$ key (A), often referred to as max-heap. The reverse comparison is called the min-heap.
- Consider the case of ascending order and counting from 0 then a_l, $a_{l+1}, \ldots, a_r$ is heap structure if $\forall i \in [l,r]$:

  + $a_i \geq a_{2i+1}$ (left child)

  + $a_i \geq a_{2i+2}$ (right child)

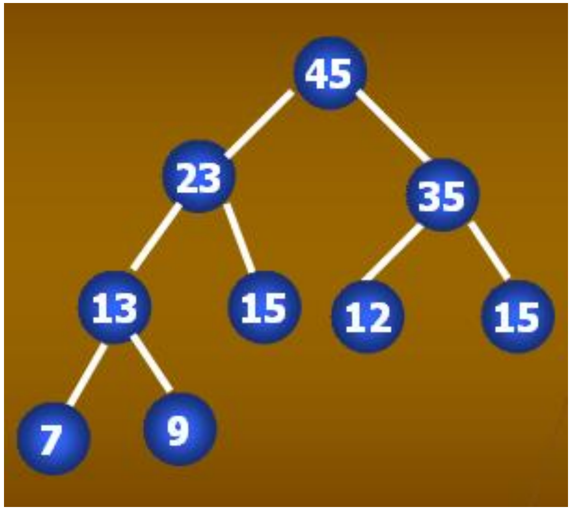
  In this case, (a_i, a_{2i+1}) and (a_i, a_{2i+2}) are sibling.

Heap

- Max-Heap:
  - Each array a can be seen as a binary tree with the root as the beginning of the array a[0].
  - The left child of vertex a[i] is a[2 * i + 1], the right child of vertex a[i] is a[2 * i + 2] if 2 * i + 1 <= n.

  $\rightarrow$ elements have index $i > \left\lfloor \frac{n}{2} \right\rfloor$ will not have children, called leaves

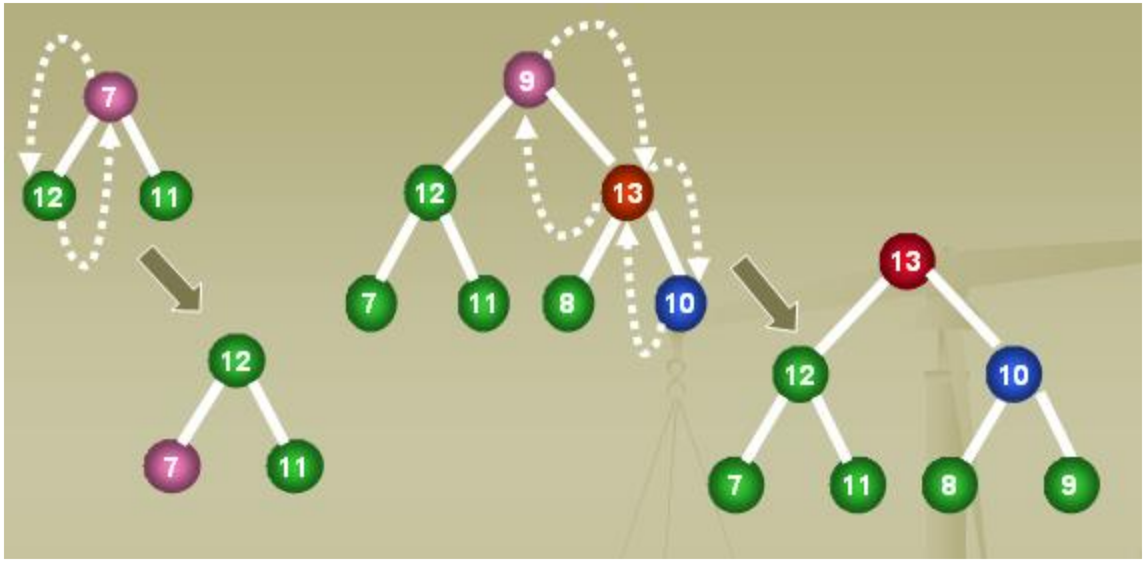
  - Child nodes always have a smaller value than their parent.



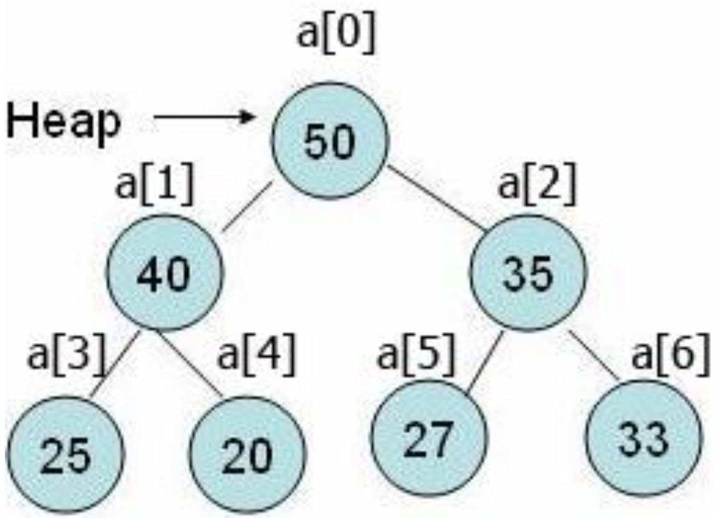
Array (45,23,35,13,15,12,15,7,9) is seen as max-heap tree.

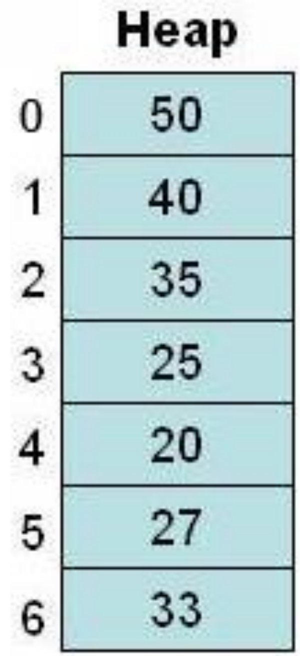
Heapify

- Sorting elements of the original array so that it becomes heap is called heapify.



Heap properties





<u>Heap properties:</u>

[1] If $a_l, a_{l+1}, \ldots, a_r$ is heap, $a_i, a_{i+1}, \ldots, a_j$ $(l \leq i \leq j \leq r)$ is also a heap.

[2] If $a_l, a_{l+1}, \ldots, a_r$ is heap, a_l is always the largest element (max-heap).

[3] All sub-array of $a_l, a_{l+1}, \ldots, a_r$ with $i > \frac{r}{2}$ is always heap.

Heap Sort

Algorithm: consists of 2 phases

- Phase 1: Modify the original array to the heap.
- Phase 2: Sort arrays based on heap.
  - + S1: Swap the largest element and the last element in array;
  - + S2: Removes the last element from the heap, modifying the rest to the heap.
  - + S3: If the heap has an element, then repeat S1.

    Otherwise, go to end.

Note: Based on the third property, we can deduce $a_{(n-1)/2+1}, \ldots, a_{n-1}$ is always a heap, thus adding the elements in turn $a_{(n-1)/2}, \ldots, a_0$ and modifying them to valid heap.

Heap Sort

Sorting by ascending

Phase 1

child

Valid Heap

0	1	2	3	4	5	6	7	8	9
23	17	97	44	35	10	12	8	5	**78**

0	1	2	3	4	5	6	7	8	9
23	17	97	44	78	10	12	**8**	**5**	35

...

0	1	2	3	4	5	6	7	8	9
97	78	23	44	35	10	12	8	5	17

Heap Sort

Sorting by ascending

Phase 2

0	1	2	3	4	5	6	7	8	9
97	78	23	44	35	10	12	8	5	17

Swap

0	1	2	3	4	5	6	7	8	9
17	78	23	44	35	10	12	8	5	97

Heapify

0	1	2	3	4	5	6	7	8	9
78	44	23	17	35	10	12	8	5	97

Heap Sort

Sorting by ascending

```
function HeapSort(…)
    CreateHeap(…);  //phase 1
    //phase 2
    for (i ← n-1 to 1) do
            a[0] ↔ a[i];
            siftdown(a, 0, i-1);        //heapify on the reduced heap
    end for
end function
```

```
function CreateHeap(…)
    //elements from (n-1)/2+1 to end of array satisfy heap
    for (i ← (n-1)/2 to 0) do
            siftdown(a, i, n-1);        //heapify
    end for
end function
```

Heap Sort

Sorting by ascending

```
function siftdown(a, left, right)
    p ← 2*left + 1;                  //position of left child
    if (p > right) then return;
    end if
    if (a[p] < a[p+1]) then          //left child smaller than right child
            p ← p + 1;
    end if
    if (a[left] < a[p]) then
            a[left] ↔ a[p];          //swap
            siftdown(a, p,right); //recursively heapify the affected sub-tree
    end if
end function
```

Comments

- Advantages:
  - Low complexity: $O(nlogn)$
- Disadvantage:
  - Although the complexity is lower than the Quick Sort, the implementation of the Quick Sort is better.

Sorting Algorithms

- Selection Sort
- Insertion Sort
- Interchange Sort
- Bubble Sort
- Shaker Sort
- Binary Insertion Sort
- Shell Sort
- Heap Sort
- Quick Sort
- Merge Sort
- Radix Sort

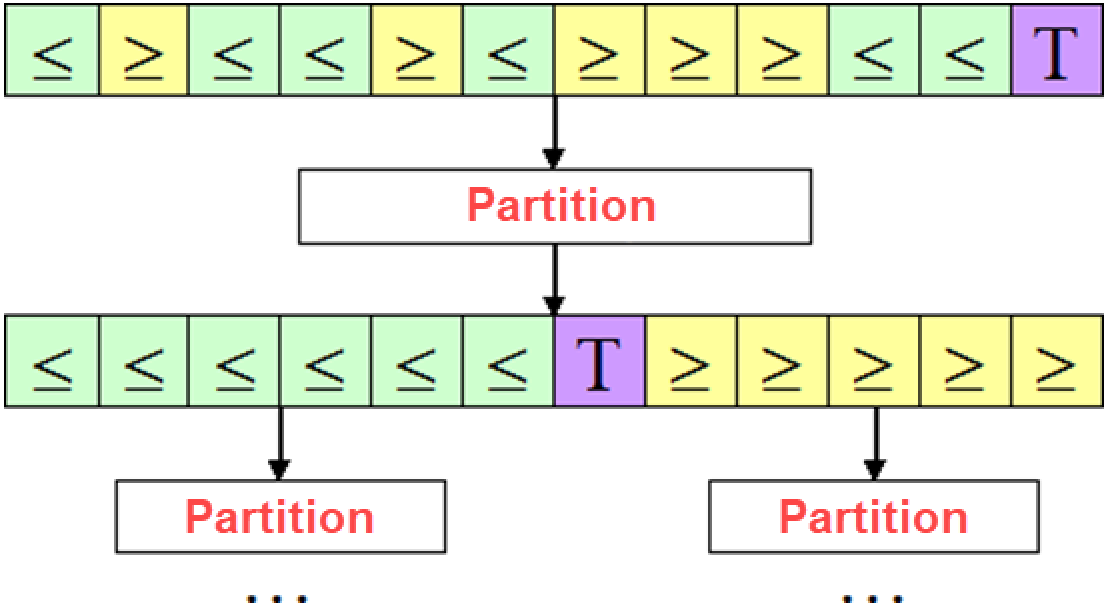
Quick Sort

This is called divide-and-conquer.

<u>Idea:</u>

- Partition the original array into two arrays: the first sub-array consists of elements less than x, and the second sub-array consists of elements greater than x. (x is an optional element in the sequence)

- The partitioning process will be repeated on each sub-array until the sub-array has only 1 element.



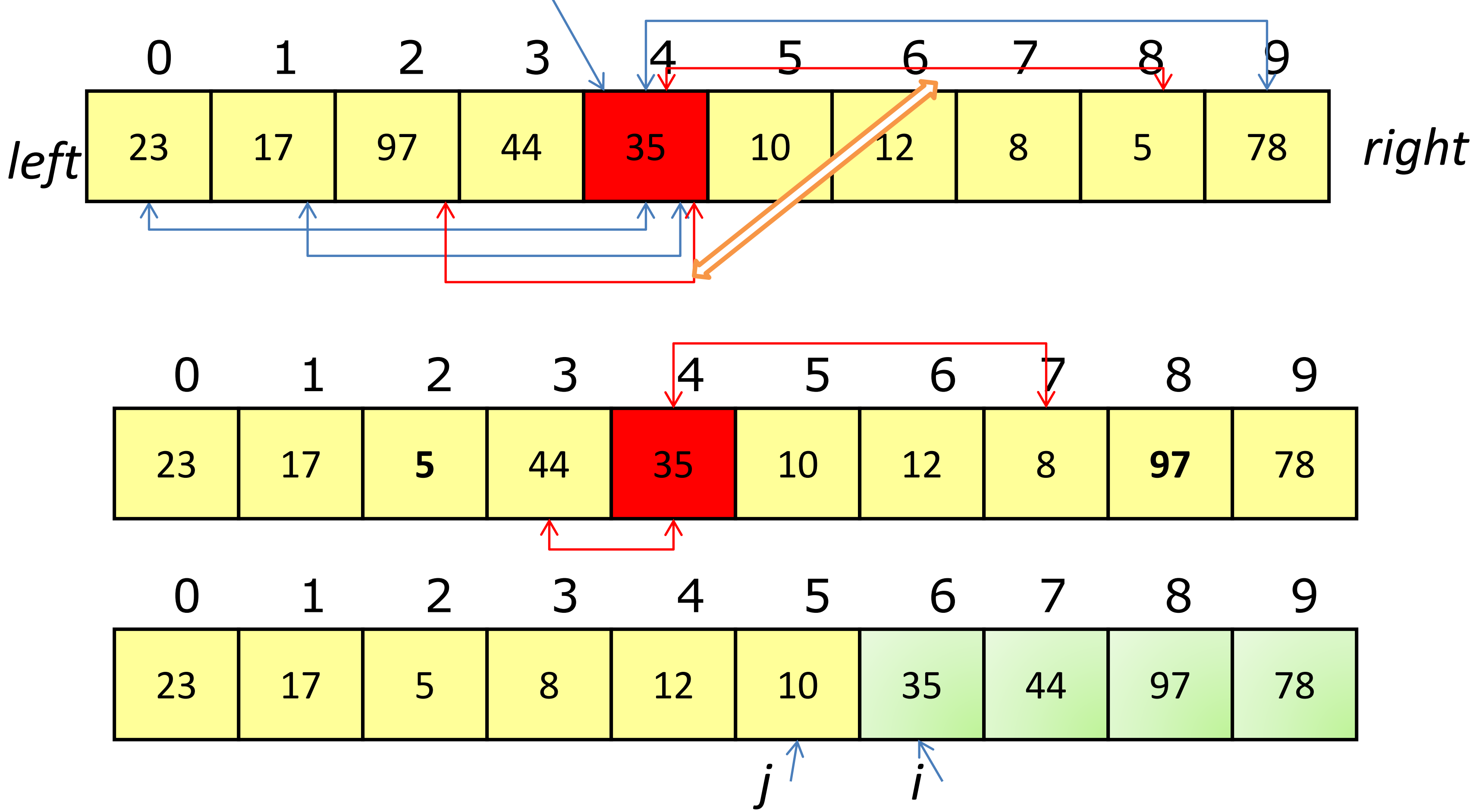
Quick Sort

Steps:

- S1: Selects any element in the array as the partition value. i = left; j = right-1;
- S2: Find and correct pairs of elements a[i], a[j] in the wrong place.
  - S2a: While (a[i] < x) i++;
  - S2b: While (a[j] > x) j--;
  - S2c: If $i \leq j$, swap (a[i], a[j]);
- S3: If $i \leq j$, go back to S2.
- S4: Recursively call to partition the left sub-array (left, j).
- S5: Recursively call to partition the right sub-array (i, right).

Sorting by ascending

Select x = a[4] = 35



Quick Sort

Sorting by ascending

```
function QuickSort(a,left,right)
    i ← left; j ← right;
    while (i ≤ j) do
            while (a[i] < x) do i ← i+1; end while
            while (a[j] > x) do j ← j-1; end while
            if (i ≤ j) then
                    a[i] ↔ a[j];
                    i ← i+1; j ← j-1;
            end if
    end while
    if (left < j) then QuickSort(a,left,j); end if
    if (i < right) then QuickSort(a,i,right); end if
end function
```

Sorting Algorithms

- Selection Sort
- Insertion Sort
- Interchange Sort
- Bubble Sort
- Shaker Sort
- Binary Insertion Sort
- Shell Sort
- Heap Sort
- Quick Sort
- Merge Sort
- Radix Sort

Merge Sort

Idea:

- Partition the original array into two sub arrays. Repeat partitioning on the sub array until the array has 1 element. (top-down)

- Merge each pair of sub arrays into an array in order and repeat for its two parent arrays, until the original array size is reached. (bottom-up)

6 5 3 1 8 7 2 4

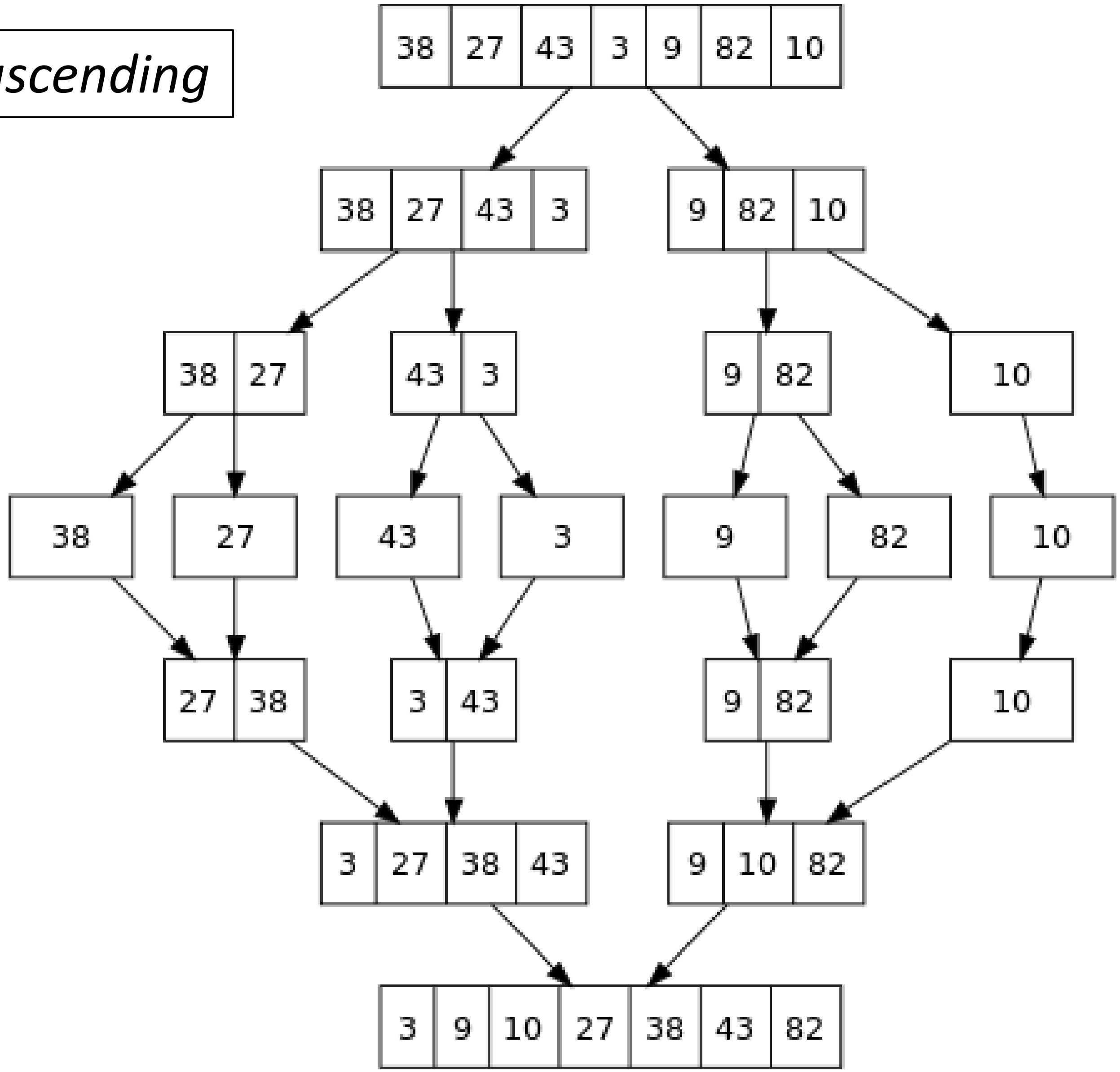
Merge Sort

Steps:

- S1: mid = (l+r)/2;
- S2: Split array a into 2 subarrays b, c
- S3: If the subarray b, c has more than 2 elements, then repeat S2.
- S4: Merge two child arrays to get an ordered parent array.
  Repeat until the array is full of elements.

Merge Sort

Sorting by ascending



Merge Sort

Sorting by ascending

```
function MergeSort(a[],l,r,temp[])
    if (r-l <2) return;
    mid = (l+r)/2;
    MergeSort(a,l,mid,temp);        //divide and merge the left sequence
    MergeSort(a,mid, r, temp);      //divide and merge the right sequence
    Merge(a, l, mid, r, temp);      //merge the split halves
    CopyArray(temp, l,r,a);         //copy back to array a
end function
```

Merge Sort

Sorting by ascending

```
function Merge(a[], l, mid, r, temp)
    iLeft ← l;        iRight ← mid; //The element begins at each child sequence
    for (j ← l; j < r; j++) do
        if ( iLeft < mid && (iRight >= r || a[iLeft] <= a[iRight])) then
            temp[j] ← a[iLeft];
            iLeft++;
        else
            temp[j] ← a[iRight];
            iRight++;
        end if
    end for
end function
```

Sorting Algorithms

- Selection Sort
- Insertion Sort
- Interchange Sort
- Bubble Sort
- Shaker Sort
- Binary Insertion Sort
- Shell Sort
- Heap Sort
- Quick Sort
- Merge Sort
- Radix Sort

Radix Sort

Idea:

- Suppose each element x in the array $a_0,\ldots, a_{n-1}$ is an integer with up to m digits.

- Sort the elements in turn according to its number of units, tens, hundreds ....

Steps:

- S1: k = 0; //sort according to the unit digit
- S2: Initialize 10 empty bins $B_0,\ldots,B_9$ (stack-like).
- S3: Place the elements in array a into bins B_t with t is the number in its k^{th} digit.
- S4: Collect the numbers in bins B in order to construct new array a.
- S5: k = k+1; If k<m, go back to S2;
  Otherwise, go to end.

Radix Sort

The unit digit

0	1	2	3	4	5	6	7	8	9	10	11
70**1**	172**5**	99**9**	917**0**	325**2**	451**8**	700**9**	142**4**	42**8**	123**9**	842**5**	701**3**

Bins B

									099**9**
					172**5**			451**8**	700**9**
917**0**	070**1**	325**2**	701**3**	142**4**	842**5**			042**8**	123**9**
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

0	1	2	3	4	5	6	7	8	9	10	11
9170	0701	3252	7013	1424	8425	1725	0428	4581	1239	7009	0999

Tens digit …

Radix Sort

329		720		720		329
457		355		329		355
657		436		436		436
839	→	457	→	839	→	457
436		657		355		657
720		329		457		720
355		839		657		839

Radix Sort

Sort by ascending

```
Init B[0,…9];
for (t ← 0 to m-1) do
    for (i ← 0 to n-1) do
            Add a[i] to B[Digit(a[i],t)];
    end for
     for (j ← 0 to 9) do
            Retrieve the elements from B [j] into a;
    end for
end for
```

References

[1] http://en.wikipedia.org/wiki/Selection_sort

[2] http://en.wikipedia.org/wiki/Heap_(data_structure)

[3] http://www.tech-faq.com/heaps.shtml

[4] http://www.math.hcmuns.edu.vn/~ptbao/DataStructure/01.pdf

[5] Dương Anh Đức, Trần Hạnh Nhi, 2003, "Cấu Trúc Dữ Liệu và Thuật Toán", trường Đại Học Khoa Học Tự Nhiên Tp.HCM.

The End.