Elementary Sorting Algorithms

Sorting – Definitions

- Input: n *records*, $R_1 \ldots R_n$, from a *file*.
- Each record R_i has
  - a key K_i
  - possibly other (satellite) information
- The keys must have an ordering relation $\prec$ that satisfies the following properties:
  - *Trichotomy*: For any two keys a and b, exactly one of $a \prec b$, $a = b$, or $a \succ b$ is true.
  - *Transitivity*: For any three keys a, b, and c, if $a \prec b$ and $b \prec c$, then $a \prec c$.

The relation $\preceq$ is a *total ordering* (*linear ordering*) on keys.

Sorting – Definitions

- *Sorting*: determine a permutation $\Pi = (p_1, \ldots, p_n)$ of n records that puts the keys in non-decreasing order $K_{p_1} \leq \ldots \leq K_{p_n}$.
- *Permutation*: a one-to-one function from {1, …, n} onto itself. There are $n!$ distinct permutations of n items.
- *Rank*: Given a collection of n keys, the *rank* of a key is the number of keys that precede it. That is, $rank(K_j) = |\{K_i | K_i < K_j\}|$. If the keys are distinct, then the rank of a key gives its position in the output file.

Sorting Terminology

- *Internal* (the file is stored in main memory and can be randomly accessed) vs. *External* (the file is stored in secondary memory & can be accessed sequentially only)
- *Comparison-based sort*: uses only the relation among keys, not any special property of the representation of the keys themselves.
- *Stable sort*: records with equal keys retain their original relative order; i.e., $i < j \ \& \ Kp_i = Kp_j \Rightarrow p_i < p_j$
- *Array-based* (consecutive keys are stored in consecutive memory locations) *vs. List-based sort* (may be stored in nonconsecutive locations in a linked manner)
- *In-place sort*: needs only a constant amount of extra space in addition to that needed to store keys.

Sorting Categories by operation

- Sorting by Insertion *insertion sort, shell sort*
- Sorting by Exchange *bubble sort, quick sort*
- Sorting by Selection *selection sort, heap sort*
- Sorting by Merging *merge sort*
- Sorting by Distribution *radix sort*

Elementary Sorting Methods

- Easier to understand the basic mechanisms of sorting.
- Good for small files.
- Good for well-structured files that are relatively easy to sort, such as those almost sorted.
- Can be used to improve efficiency of more powerful methods.

Bubble Sort

```
BubbleSort(A)
1   for i = 1 to A.length − 1
2       for j = A.length to i + 1
3           if A[j] < A[j − 1]
4               Swap(A[j], A[j − 1]
```

Analysis of Bubble Sort

- In the first pass, n-1 comparisons will be done, n-2 in 2nd pass, n-3 in 3rd pass and so on.
- So the total number of comparisons will be,
- (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1
- Sum = n(n-1)/2
- i.e O(n^2)
- Hence the time complexity of Bubble Sort is O(n^2).

Analysis of Bubble Sort

- The main advantage of Bubble Sort is the simplicity of the algorithm.
- In place sorting
- Stable sorting
- The space complexity for Bubble Sort is O(1), because only a single additional memory space is required i.e. for temp variable.
- Can be optimized to run in O (n) time in best case (When the array is already sorted) by stopping the algorithm if the inner loop didn’t cause any swap.

Insertion Sort

```
InsertionSort(A, n)
1.  for j = 2 to n do
2.      key ← A[j]
3.      i ← j – 1
4.      while i > 0 and key < A[i]
5.          A[i+1] ← A[i]
6.          i ← i – 1
7.      A[i+1] ← key
```

Analysis of Insertion Sort

- Time Complexity:
- Worst case (Reversely sorted list): $O(n^2)$
- Average case (any list): $O(n^2)$
- Best case (Already sorted list): $O(n)$
- Auxiliary Space (Space complexity): $O(1)$
- In Place sorting
- Stable sorting
- Online algorithm

Analysis of Insertion Sort

- The worst case, $\Theta(n^2)$: if the inputs A contain distinct items in reverse order:
- Maximum possible no of comparisons in each iteration j = j (j-1…0)
- And j = 2, . . . , n (Total n-1)
- Comparisons in total for: 1 + 2 + . . . + n − 1 = (n−1)n/2. i.e. $\Theta(n^2)$

$$C_{wc}(n) \leq \sum_{j = 2 \text{ to } n} (j - 1)$$
$$= \sum_{i = 1 \text{ to } n-1} i$$
$$= n(n-1)/2$$
$$= \Theta(n^2)$$

Average-case Analysis of Insertion sort

- Want to determine the average number of comparisons taken over all possible inputs.
- Determine the average no. of comparisons for a key $A[j]$.
- $A[j]$ can belong to any of the j locations, $1..j$, with equal probability.
- The number of *key* comparisons for $A[j]$ is $j–k+1$, if $A[j]$ belongs to location k, $1 < k \leq j$ and is $j–1$ if it belongs to location 1.

Average no. of comparisons for inserting key $A[j]$ is:

$$\sum_{k=1}^{j-1}\left(\frac{1}{j}k\right)+\frac{1}{j}(j-1)$$

$$=\frac{1}{j}\sum_{k=1}^{j-1}(k)+1-\frac{1}{j}$$

$$=\frac{j-1}{2}+1-\frac{1}{j}=\frac{j+1}{2}-\frac{1}{j}$$

Average-case Analysis of Insertion sort

Summing over the no. of comparisons for all keys,

$$C_{avg}(n) = \sum_{i=2}^{n} \left(\frac{i+1}{2} - \frac{1}{i} \right)$$

$$= \frac{n^2}{4} + \frac{3n}{4} - 1 - \sum_{i=2}^{n} \frac{1}{i}$$

$$= \Theta(n^2) - O(\ln n)$$

$$= \Theta(n^2)$$

Therefore, $T_{avg}(n) = \Theta(n^2)$

Selection Sort

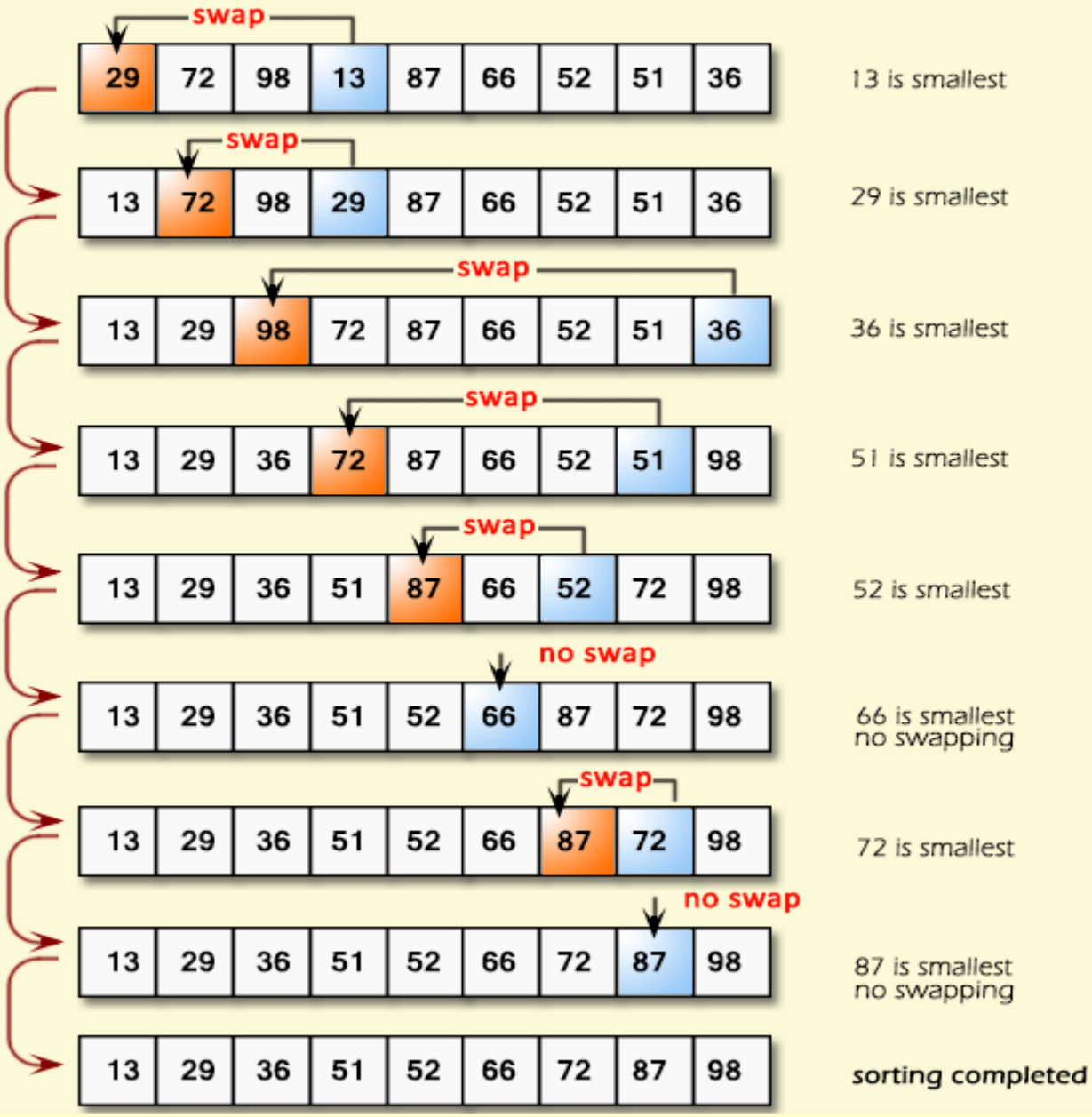
Selection-Sort(*A*, *n*)
input: array *A*[1..*n*] of n unsorted integers
output: same integers in array *A* now in sorted order

```
1 for i = 1 to n-1
2         min = i
3         for j = i+1 to n
4                 if A[j] < A[min]
5                         min = j
6         if min ≠ i then
7             swap A[i] with A[min]
```

Selection Sort

Selection-Sort(*A*, *n*)
input: array *A*[1..*n*] of n
unsorted integers
output: same integers in array *A*
now in sorted order

```
1 for i = 1 to n-1
2       min = i
3       for j = i+1 to n
4           if A[j] < A[min]
5               min = j
6       if min ≠ i
6           swap (A[i], A[min])
```



Correctness of Selection sort

Selection-Sort(*A*, *n*)
input: array *A*[1..*n*] of n unsorted integers
output: same integers in array *A* now in sorted order
1 for *i* = 1 to *n*-1
2 *min* = *i*
3 for *j* = *i*+1 to *n*
4 if *A*[*j*] < *A*[*min*]
5 *min* = *j*
6 if *min* ≠ *i* then
7 swap *A*[*i*] with *A*[*min*]

- **Loop Invariant:** Before the start of each loop (line 3), A[min] is less than or equal to A[i..j-1].

Correctness of Selection sort

Selection-Sort(*A*, *n*)
input: array *A*[1..*n*] of n unsorted integers
output: same integers in array *A* now in sorted order
1 for *i* = 1 to *n*-1
2 *min* = *i*
3 for *j* = *i*+1 to *n*
4 if *A*[*j*] < *A*[*min*]
5 *min* = *j*
6 if *min* ≠ *i* then
7 swap *A*[*i*] with *A*[*min*]

- **Initialization:** Prior to the first iteration of the loop, j=i+1. So the array segment A[i..j-1] is really just spot A[i]. Since line 2 of the code sets min = i, we have that, min indexes the smallest element (the only element) in subarray A[i..j-1] and hence the loop invariant is true.

Correctness of Selection sort

- **Maintenance:** Before pass j, we assume that min indexes the smallest element in the subarray A[i..j-1].
- During iteration j we have two cases: either A[j] < A[min] or A[j] ≥ A[min].
- In the second case, the if statement on line 4 is not true, so nothing is executed. But now min indexes the smallest element of A[i..j].
- In the first case, line 5 switches min to index location j since it is the smallest. If min indexes an element less than or equal to subarray A[i..j-1] and now A[j] < A[min], then it must be the case that A[j] is less than or equal to elements in subarray A[i..j-1]. Line 5 switches min to index this new location and hence after the loop iteration finishes, min indexes the smallest element in subarray A[i..j]

Correctness of Selection sort

Selection-Sort(*A*, *n*)
input: array *A*[1..*n*] of n unsorted integers
output: same integers in array *A* now in sorted order
1 for *i* = 1 to *n*-1
2 *min* = *i*
3 for *j* = *i*+1 to *n*
4 if *A*[*j*] < *A*[*min*]
5 *min* = *j*
6 if *min* ≠ *i* then
7 swap *A*[*i*] with *A*[*min*]

- **Termination:** At termination of the inner loop, min indexes an element less than or equal to all elements in subarray A[*i*..*n*] since *j* = *n*+1 upon termination. This finds the smallest element in this subarray and is useful to us in the outer loop because we can move that next smallest item into the correct location.

Analysis of Selection sort

- To sort n elements, selection sort performs n-1 passes:
- On 1st pass, it performs n-1 comparisons to find index of the smallest element
- On 2nd pass, it performs n-2 comparisons …
- On the (n-1)th pass, it performs 1 comparison
- Adding up the comparisons for each pass, we get:

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

Analysis of Selection sort

- The resulting formula for C(n) is the sum of an arithmetic sequence: $C(n) = 1 + 2 + \ldots + (n - 2) + (n - 1) = \sum_{i=1}^{n-1} i$
- Formula for the sum of this type of arithmetic sequence:

$$\sum_{i=1}^{m} i = \frac{m(m+1)}{2}$$

- Thus, we can simplify our expression for C(n) as follows:

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1} i \\ &= \frac{(n-1)((n-1)+1)}{2} \\ &= \frac{(n-1)n}{2} \\ &= O(n^2) \end{aligned}$$