Chapter 4
Simple Sorting & Searching Algorithms

Topics

- Searching
  - Linear/Sequential Search
  - Binary Search
- Sorting
  - Bubble Sort
  - Insertion Sort
  - Selection sort

Common Problems

- There are some very common problems that we use computers to solve:
  - **Searching:** Looking for specific data item/record from list of data items or set of records.
  - **Sorting :** reordering a list of items in either increasing or decreasing order
- There are numerous algorithms to perform searches and sorts.
- We will briefly explore a few common ones in this lecture.

Searching

- There exists many searching algorithms you can choose from
- A question you should always ask when selecting a search algorithm is
  - "How fast does the search have to be?"
- **Facts**
  - In general, the faster the algorithm is, the more complex it is.
  - You don't always need to use the fastest algorithm.
  - The list to be searched can either be ordered or unordered list
- Let's explore the following search algorithms, keeping speed in mind.
  - Sequential (linear) search
  - Binary search

Linear/Sequential Search on an Unordered List

- Linear search: a sequential search is made over all items one by one.
  - Meaning, every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Algorithm : Linear Search

- **Linear Search (Array list, Value key)**
  - Step 1: Set i to 0
  - Step 2: if i >= n then go to step 7
  - Step 3: if list[i] = key then go to step 6
  - Step 4: Set i to i + 1
  - Step 5: Go to Step 2
  - Step 6: Print Element key Found at index i and go to step 8
  - Step 7: Print element not found
  - Step 8: Exit

When do we know that there wasn't item in the List that matched the key?

Linear Search (Sequential Search)

- Example Implementation:

```
int linear_search(int list[], int n, int key){
 for (int i=0;i<n; i++){
        if(list[i]==key)
        return i;
 }
 return -1;
}
```

Example with illustration: Linear Search

An array with 10 elements, search for “9”:

56	3	249	518	7	26	94	651	23	9
56	**3**	249	518	7	26	94	651	23	9
56	3	**249**	518	7	26	94	651	23	9
56	3	249	**518**	7	26	94	651	23	9
56	3	249	518	**7**	26	94	651	23	9
56	3	249	518	7	**26**	94	651	23	9
56	3	249	518	7	26	**94**	651	23	9
56	3	249	518	7	26	94	**651**	23	9
56	3	249	518	7	26	94	651	**23**	9
56	3	249	518	7	26	94	651	23	**9**

Time complexity O(n)
--Unsuccessful search --- n times
--Successful search (worst) --- n times
--Successful search (Best) --- 1 time
--Successful search (average) --- n/2 times

Linear Search – Questions ?

- What possible modification we can make if the list is a sorted one so as to make sequential search better?
  – Hint:- when do we know that there wasn't an item in the List that matched the key?

- Assume you have observed that some of the data items are searched more frequently than others in the list. What modification can be made to make this algorithm better ?

- Homework
  – Write an implementation for your answers of Q1 and Q2

Sequential Search of Ordered vs. Unordered List

- If sequential search is used on list of integers say [14,80,39,100,-8], how would the search for 100 on the ordered list compare with the search on the unordered list?
  - Unordered list <14,80,39,100,-8>
    - if 100 was in the list?
      - 4 iterations needed
    - if -50 was not in the list?
      - 5 iterations needed
  - Ordered list <-8,14,39,80,100>
    - if 100 was in the list?
      - 5 iterations
    - if -50 was not in the list?`
      - 1 iteration

Ordered vs. Unordered (con't)

- **Observation**: the search is faster on an ordered list only when the item being searched for is not in the list.

  - Also, keep in mind that the list has to first be placed in order for the ordered search.

- **Conclusion**: the **efficiency** of these algorithms is roughly the same.
  - So, if we need a faster search, on sorted list we need a completely different algorithm.

Binary Search

- Sequential search is not efficient for large lists because, on average, the sequential search searches half the list.

- If we have an ordered list and we know how many things are in the list, we can use a different algorithm named binary-search.

- The **binary search** gets its name because the algorithm continually divides the list into two parts.
  – Uses the divide-and-conquer technique to search the list

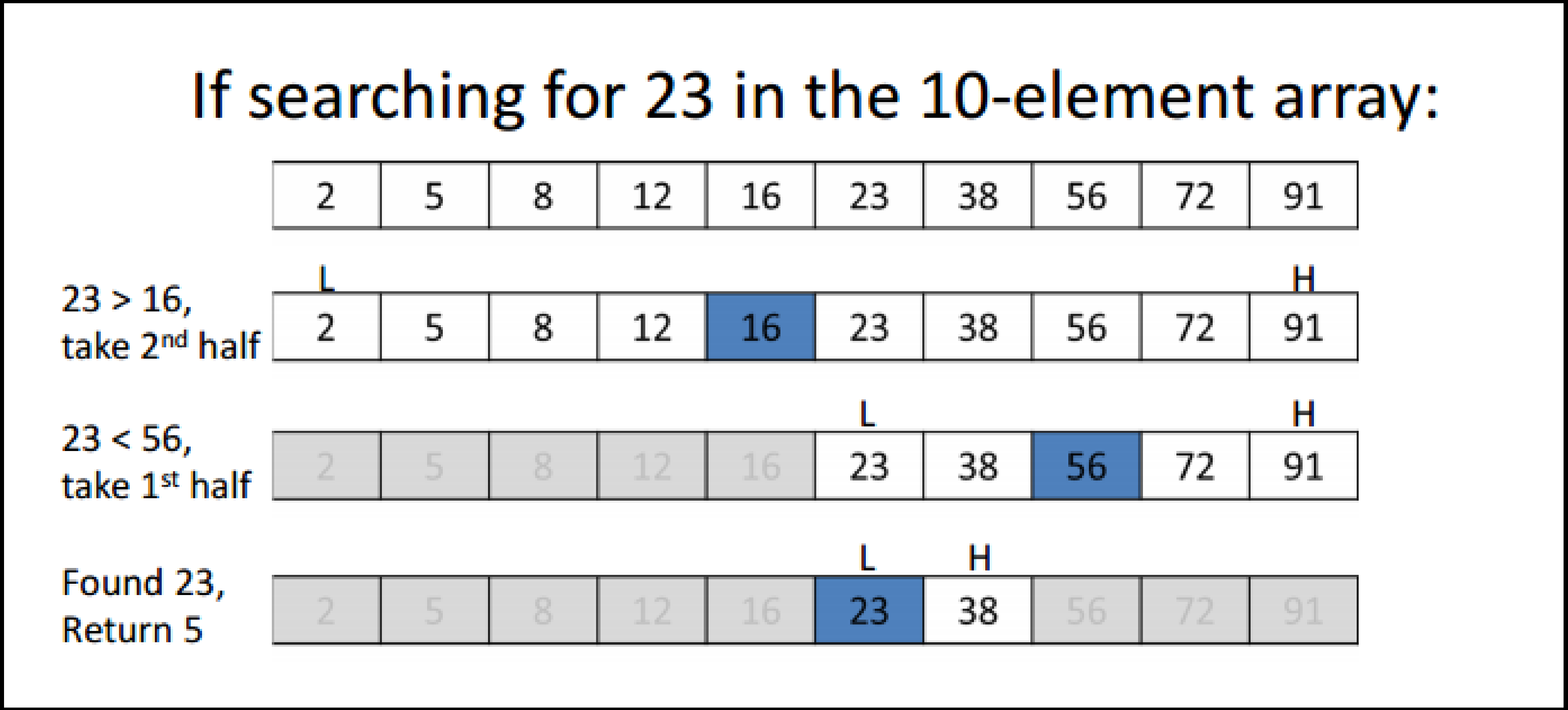
Basic Idea: Binary Search

- Binary search looks for a particular item by comparing **the middle most item** of the collection.
  - If a **match occurs**, then the index of item is returned.
  - If **the item** is **greater than the middle item**, then the item is searched in the sub-array to the **right of the middle item**.
  - Otherwise, the item is searched for in the sub-array to the **left of the middle item.**
  - This process continues on the sub-array as well until the size of the **sub-array reduces to zero**.

Example Implementation

```
int binary_search(int list[],int n, int key)
{       int left=0;  int right=n-1;
        int mid;
        while(left<=right){
                        mid=(left+right)/2;
                if(key==list[mid])
                        return mid;
                else if(key > list[mid])
                        left=mid+1;
                else
                        right=mid-1;
        }
        return -1;
}
```

Example with illustration: Binary Search



How Fast is a Binary Search?

- Worst case: 10 items in the list took 3 tries
- How about the worst case for a list with 32 items ?
  - 1st try - list has 16 items
  - 2nd try - list has 8 items
  - 3rd try - list has 4 items
  - 4th try - list has 2 items
  - 5th try - list has 1 item

How Fast is a Binary Search? (con’t)

List has 250 items

1st try - 125 items

2nd try - 63 items

3rd try - 32 items

4th try - 16 items

5th try - 8 items

6th try - 4 items

7th try - 2 items

8th try - 1 item

List has 512 items

1st try - 256 items

2nd try - 128 items

3rd try - 64 items

4th try - 32 items

5th try - 16 items

6th try - 8 items

7th try - 4 items

8th try - 2 items

9th try - 1 item

Efficiency

- Binary search is one of the fastest Algorithms
- The computational time for this algorithm is proportional to $\log_2 n$
- Log n means the log to the base 2 of some value of n.
  - $8 = 2^3$ log 8 = 3 $16 = 2^4$ log 16 = 4
- Therefore, the time complexity is O(logn)
- How much space?

In-place algorithm

Meaning, binary search requires no additional memory therefore, O(1).

Sorting

- The binary search is a very fast search algorithm.
  - But, the list has to be sorted before we can search it with binary search.
- To be really efficient, we also need a fast sort algorithm.

- There are many known sorting algorithms.

Bubble Sort	Heap Sort
Selection Sort	Merge Sort
Insertion Sort	Quick Sort and others

Internal and external sorting

- Internal sorting:
  - The process of sorting is done in main memory
  - The number of elements in main memory is relatively small (less than millions). The input is fit into main memory
- In this type of sorting the main advantage is memory is directly addressable,
  - which bust-up performance of the sorting process.
- External sorting:
  - Can not be performed in main memory due to their large input size. i.e., the input is much larger to fit into main memory
  - Sorting is done on disk or tape.
  - It is device dependent than internal sorting

Common Sorting Algorithms

- Bubble sort is the slowest, running in **n^2 time**.
- **Quick sort** is the fastest, running in **$n \log n$ time**.
- As with searching, the faster the sorting algorithm, the more complex it tends to be.
- We will examine three sorting algorithms:
  - Bubble sort
  - Insertion sort
  - Selection sort

Bubble Sort

- Bubble sort is a simple algorithm with a memorable name and a simple idea
- **The idea:**
  - Starting at the front, traverse the list, find the largest item, and move (or *bubble*) it to the top
  - **How:**
    - **Compare adjacent elements** and swap the elements if they are not in order.
  - With each subsequent iteration, find the next largest item and *bubble* it up towards the top of the array
- This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n is the number of items.

Obama on bubble sort

When asked the most efficient way to sort a million 32-bit integers, Senator Obama had an answer:

www.youtube.com/watch?v=k4RRi_ntQc8

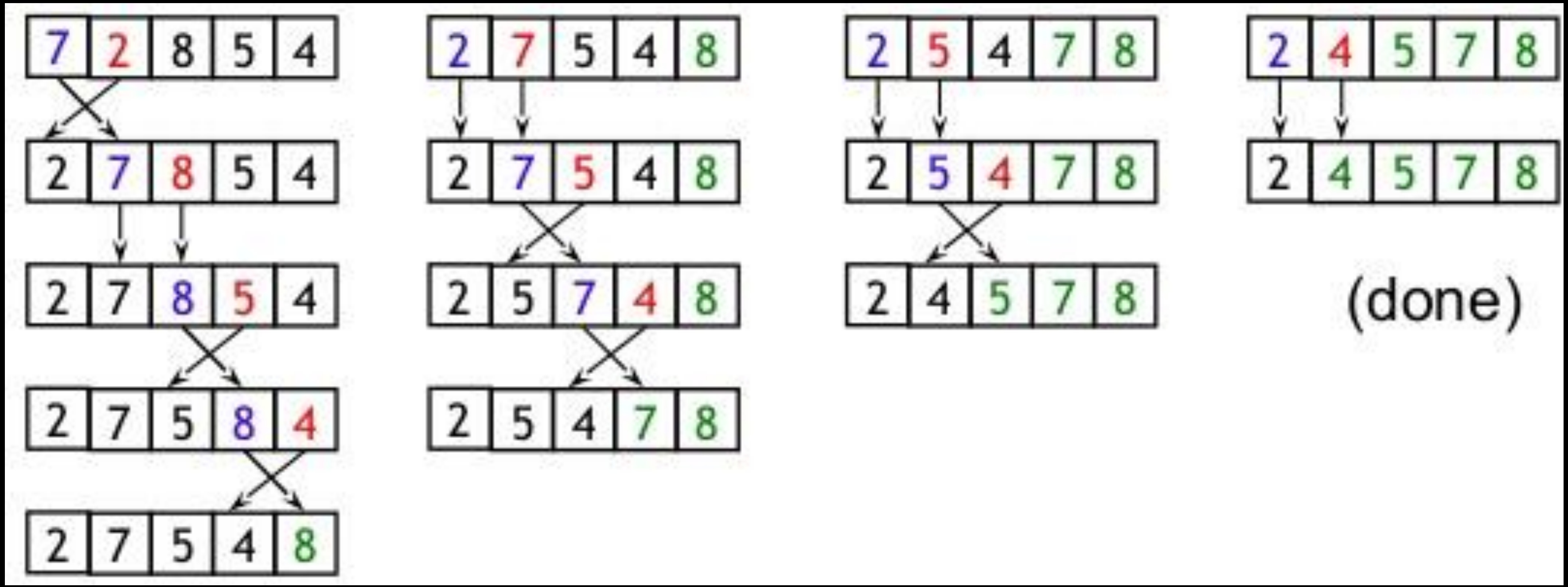
Algorithm: Bubble Sort

- Step 1 – Loop through array from i=0 to n
- Step 2 – Pick two adjacent element
- Step 3 – Compare the elements
- Step 4 – Swap the adjacent elements if they are out of order
- Step 5 – Repeat until list is sorted

Implementation: Bubble Sort

```
void bubbleSort (int a[ ] , int size)
{
    int i, j, temp;
    for ( i = 0; i < size; i++ )/*controls passes through the list */
    {
      for ( j = 0; j < size - 1; j++ )/*performs adjacent comparisons */
      {
            if ( a[ j ] > a[ j+1 ] )/*determines if a swap should occur */
            {
                  temp = a[ j ];         /* swap is performed */
                  a[ j ] = a[ j + 1 ];
                  a[ j+1 ] = temp;
            }}}
}
```

Example with illustration: Bubble Sort



We start with the element in the first location, and move forward:

- if the current and next items are in order, continue with the next item, otherwise swap the two entries
- After one loop, the largest element is in the last location
- Repeat again

Analysis- Bubble Sort

- How many comparisons?

  $(n * n) = O(n^2)$ In all best, average and worst cases

- How many swaps?

  $0 = O(1)$ in best cases where the list is already sorted

  $n^2 = O(n^2)$ in worst case scenario where the list is sorted in reverse order

- How much space?

  In-place algorithm

  Meaning, bubble sort requires no additional memory therefore, $O(1)$.

Insertion Sort

- Consider the following observations:
  - A list with one element is sorted
  - In general, if we have a sorted list of *k* items, we can insert a new item to create a sorted list of size *k* + 1
- Insertion sort works the same way as arranging your hand when playing cards.
  - Out of the pile of unsorted cards that were dealt to you, you pick up a card and place it in your hand in the correct position relative to the cards you're already holding.
- Basic Idea is: Demos\21DemoInsertionSort.mov
  - Find the location for an element and move all others up, and insert the element.

Algorithm : Insertion Sort

- Step 1 – If it is the first element, it is already sorted. return 1;
- Step 2 – Pick next element
- Step 3 – Compare with all elements in the sorted sub-list
- Step 4 – Shift all the elements in the sorted sub-list that is greater than the value to be sorted
- Step 5 – Insert the value
- Step 6 – Repeat until list is sorted

Implementation- Insertion sort

```
void insertion_sort(int list[ ], int n)
{
int temp;
for(int i = 1; i < n; i++){
     temp = list[i];
for(int j = i; j > 0 && temp < list[j - 1]; j--)
{ //work backwards through the array finding where temp should go
          list[j] = list[j - 1];
          list[j - 1] = temp;
   }//end of inner loop
  }//end of outer loop
}//end of insertion_sort
```

Analysis – Insertion sort

- How many comparisons?

  $1 + 2 + 3 + \ldots + (n-1) = O(n^2)$ in worst case scenario where the list is sorted in reverse order

  $1+1+\ldots..1= O(n)$ in the best case

- How many swaps?

  $1 + 2 + 3 + \ldots + (n-1) = O(n^2)$ worst cases

  0 during best case where the list is already sorted

- How much space?

  In-place algorithm

  Meaning, no additional memory is required apart temp. hence it is O(1)

Selection Sort

- **Basic Idea:**
  - Loop through the list from **i = 0** to **n - 1**.
  - Select the smallest element in the array from i to n
  - Swap this value with value at position **i**.
- Demos\21DemoSelectionSort.mov
- This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where n is the number of items.

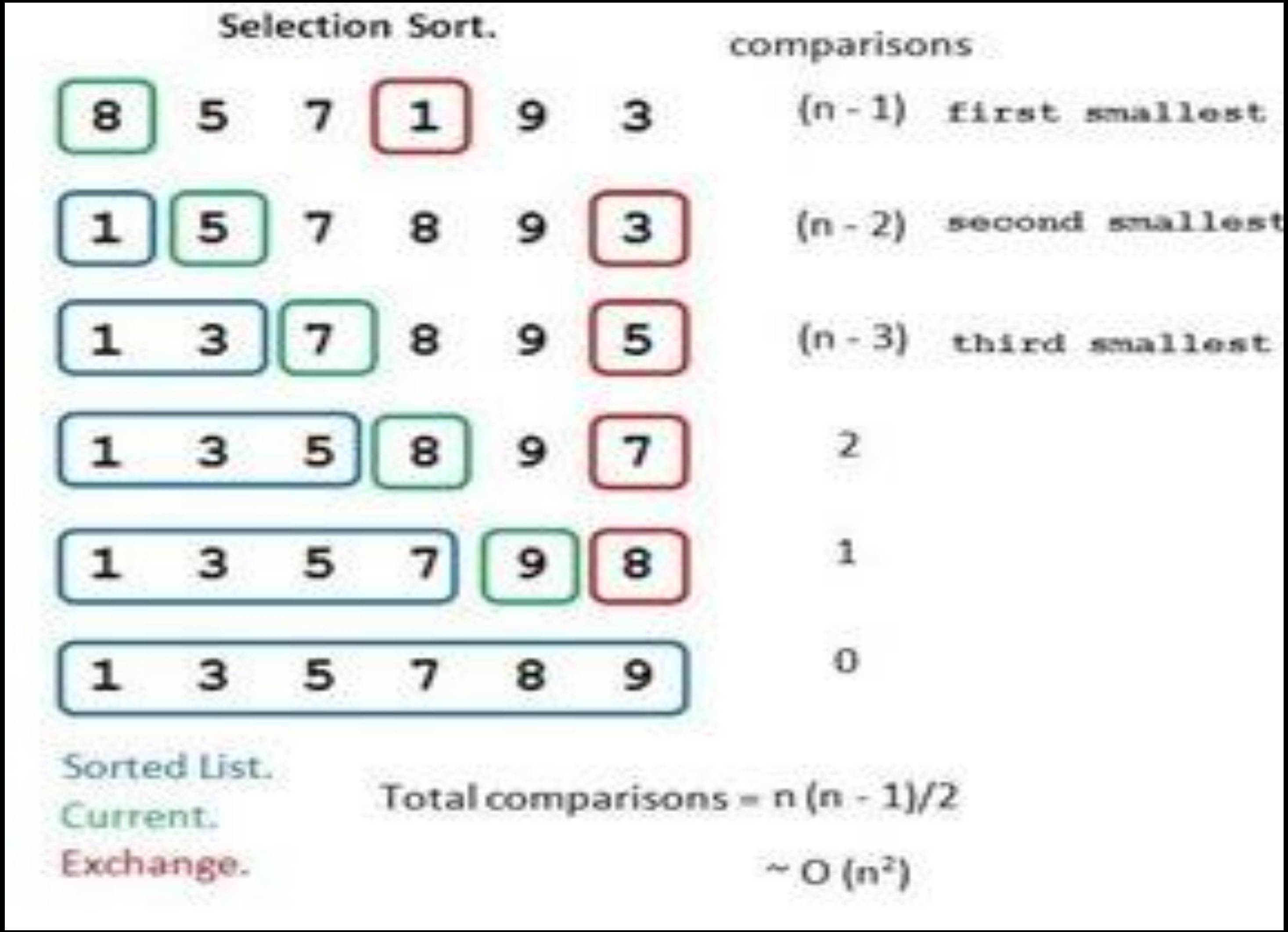
Algorithm: Selection Sort

- Step 1 – Set **SMALLEST** to location **0**
- Step 2 – Search the minimum element in the list
- Step 3 – Swap with value at location **SMALLEST**
- Step 4 – Increment **SMALLEST** to point to next element
- Step 5 – Repeat until list is sorted

Implementation- Selection Sort

```
void selection_sort(int list[], int n)
{
int i, j, smallest;
for(i = 0; i < n; i++){
     smallest = i;
     for(j = i + 1; j < n; j++){
          if(list[j] < list[smallest])
               smallest = j;
      }//end of inner loop
     temp = list[smallest];
     list[smallest] = list[i];
     list[i] = temp;
} //end of outer loop
}//end of selection_sort
```

Example with illustration: Selection Sort



Analysis- Selection Sort

- How many comparisons?

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

- How many swaps?

  $n = O(n)$

- How much space?

  In-place algorithm

  Meaning, Selection Sort requires no additional memory therefore, O(1).

Exercise

- Sort the following data

Element	1	2	3	4	5	6	7	8
Data	27	63	1	72	64	58	14	9

1. Insertion Sort
2. Selection Sort
3. Bubble Sort

Assignment (10%)

Title

1. Shell Sort (Atiklt)
2. Heap Sort(meklit)
3. Quick Sort (Fasil)
4. Merge Sort(Bitanya)
5. Radix Sort(Abel)
6. Counting Sort (Workneh)
7. Cocktail Sort (Saleh)
8. Bucket Sort(Surafel)

Directions

1. What you submit?
   - Algorithm description with sufficient details **(Algorithm + Code)**
     - It has to show what it is? How does it work? Example?
2. When:
   - December 6, 2021
   - Morning: 10:30AM
3. Evaluation:
   - Submission and presentation

Next Time!!

Ch5-List