Lecture#1

Sorting & Searching Arrays

CENG 102- Algorithms and Programming II,

2024-2025, Spring

Contains materials from:
P. Deitel, H. Deitel, ''C How to Program with an Introduction to C++'', 8th edition, Pearson

Lecture 2.1

Sorting Arrays

6.8 Sorting Arrays

- Sorting data (i.e., placing the data into a particular order such as ascending or descending) is one of the most important computing applications.
- In this lecture, we will discuss what is perhaps the simplest known sorting scheme.

Performance Tip 6.4

Often, the simplest algorithms perform poorly. Their virtue is that they're easy to write, test and debug. More complex algorithms are often needed to realize maximum performance.

6.8 Sorting Arrays

- Figure 6.15 sorts the values in the elements of the 10-element array a into **ascending** order.
- The technique we use is called the bubble sort or the sinking sort because the smaller values gradually “bubble” their way upward to the top of the array like air bubbles rising in water, while the larger values sink to the bottom of the array.
- The technique is to make **several passes** through the array.
  - Several passes → n-1, where the length of the array is n.

6.8 Sorting Arrays

- On each pass, successive pairs of elements (element 0 and element 1, then element 1 and element 2, etc.) are compared.
- If a pair is in increasing order (or if the values are identical), we leave the values as they are.
- If a pair is in decreasing order, their values are swapped in the array.

```c
// Fig. 6.15: fig06_15.c
// Sorting an array's values into ascending order.
#include <stdio.h>
#define SIZE 10

// function main begins program execution
int main(void)
{
   // initialize a
   int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};

   puts("Data items in original order");

   // output original array
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%4d", a[i]);
   }

```

Fig. 6.15 | Sorting an array's values into ascending order. (Part 1 of 3.)

```
   // bubble sort
   // loop to control number of passes
   for (unsigned int pass = 1; pass < SIZE; ++pass) {

      // loop to control number of comparisons per pass
      for (size_t i = 0; i < SIZE - 1; ++i) {

         // compare adjacent elements and swap them if first
         // element is greater than second element
         if (a[i] > a[i + 1]) {
            int hold = a[i];
            a[i] = a[i + 1];
            a[i + 1] = hold;
         }
      }
   }

```

Fig. 6.15 | Sorting an array's values into ascending order. (Part 2 of 3.)

```
    puts("\nData items in ascending order");

    // output sorted array
    for (size_t i = 0; i < SIZE; ++i) {
       printf("%4d", a[i]);
    }

    puts("");
 }
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Fig. 6.15 | Sorting an array's values into ascending order. (Part 3 of 3.)

6.8 Sorting Arrays

- First the program compares `a[0]` to `a[1]`, then `a[1]` to `a[2]`, then `a[2]` to `a[3]`, and so on until it completes the pass by comparing `a[8]` to `a[9]`.
- Although there are 10 elements, only nine comparisons are performed.
- Because of the way the successive comparisons are made, a large value may move down the array many positions on a single pass, but a small value may move up only one position.
- On the first pass, the largest value is guaranteed to sink to the bottom element of the array, `a[9]`.

6.8 Sorting Arrays

- On the second pass, the second-largest value is guaranteed to sink to `a[8]`.
- On the ninth pass, the ninth-largest value sinks to `a[1]`.
- This leaves the smallest value in `a[0]`, so only *nine* passes of the array are needed to sort the array, even though there are *ten* elements.
- The sorting is performed by the nested `for` loops.

6.8 Sorting Arrays

- If a swap is necessary, it's performed by the three assignments

    -
      ```
      hold = a[i];
      a[i] = a[i + 1];
      a[i + 1] = hold;
      ```

  where the extra variable `hold` temporarily stores one of the two values being swapped.
- The swap cannot be performed with only the two assignments

    -
      ```
      a[i] = a[i + 1];
      a[i + 1] = a[i];
      ```
- If, for example, `a[i]` is 7 and `a[i + 1]` is 5, after the first assignment both values will be 5 and the value 7 will be lost—hence the need for the extra variable `hold`.

6.8 Sorting Arrays

- The chief virtue of the **bubble sort** is that **it’s easy to program**.
- **However, it runs slowly** because every exchange moves an element only one position closer to its destination.
- This becomes apparent **when sorting large arrays**.
- In the exercises, we’ll develop more efficient versions of the bubble sort.
- Much more efficient sorting algorithms than the bubble sort have been developed.

6.9 Case Study: Computing Mean, Median and Mode Using Arrays

- Computers are commonly used for survey data analysis to compile and analyze the results of surveys and opinion polls.
- Figure 6.16 uses array `response` initialized with 99 responses to a survey.
- Each response is a number from 1 to 9.
- The program computes the mean, median and mode of the 99 values.
- Figure 6.17 contains a sample run of this program.

```c
// Fig. 6.16: fig06_16.c
// Survey data analysis with arrays:
// computing the mean, median and mode of the data.
#include <stdio.h>
#define SIZE 99

// function prototypes
void mean(const unsigned int answer[]);
void median(unsigned int answer[]);
void mode(unsigned int freq[], unsigned const int answer[]) ;
void bubbleSort(int a[]);
void printArray(unsigned const int a[]);

// function main begins program execution
int main(void)
{
   unsigned int frequency[10] = {0}; // initialize array frequency

```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 1 of 8.)

```
    // initialize array response
    unsigned int response[SIZE] =
       {6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
        7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
        6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
        7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
        6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
        7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
        5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
        7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
        7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
        4, 5, 6, 1, 6, 5, 7, 8, 7};

    // process responses
    mean(response);
    median(response);
    mode(frequency, response);
}

```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 2 of 8.)

```c
// calculate average of all response values
void mean(const unsigned int answer[])
{
   printf("%s\n%s\n%s\n", "********", "  Mean", "********");

   unsigned int total = 0; // variable to hold sum of array elements

   // total response values
   for (size_t j = 0; j < SIZE; ++j) {
      total += answer[j];
   }

   printf("The mean is the average value of the data\n"
          "items. The mean is equal to the total of\n"
          "all the data items divided by the number\n"
          "of data items (%u). The mean value for\n"
          "this run is: %u / %u = %.4f\n\n",
          SIZE, total, SIZE, (double) total / SIZE);
}

```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 3 of 8.)

```
// sort array and determine median element's value
void median(unsigned int answer[])
{
   printf("\n%s\n%s\n%s\n%s",
          "********", " Median", "********",
          "The unsorted array of responses is");

   printArray(answer); // output unsorted array

   bubbleSort(answer); // sort array

   printf("%s", "\n\nThe sorted array is");
   printArray(answer); // output sorted array

   // display median element
   printf("\n\nThe median is element %u of\n"
         "the sorted %u element array.\n"
         "For this run the median is %u\n\n",
         SIZE / 2, SIZE, answer[SIZE / 2]);
}

```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 4 of 8.)

```c
// determine most frequent response
void mode(unsigned int freq[], const unsigned int answer[])
{
   printf("\n%s\n%s\n%s\n","********", "  Mode", "********");

   // initialize frequencies to 0
   for (size_t rating = 1; rating <= 9; ++rating) {
      freq[rating] = 0;
   }

   // summarize frequencies
   for (size_t j = 0; j < SIZE; ++j) {
      ++freq[answer[j]];
   }

   // output headers for result columns
   printf("%s%11s%19s\n\n%54s\n%54s\n\n",
          "Response", "Frequency", "Histogram",
          "1    1    2    2", "5    0    5    0    5");

   // output results
   unsigned int largest = 0; // represents largest frequency
   unsigned int modeValue = 0; // represents most frequent response
```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 5 of 8.)

```c

   for (rating = 1; rating <= 9; ++rating) {
      printf("%8u%11u          ", rating, freq[rating]);

      // keep track of mode value and largest frequency value
      if (freq[rating] > largest) {
         largest = freq[rating];
         modeValue = rating;
      }

      // output histogram bar representing frequency value
      for (unsigned int h = 1; h <= freq[rating]; ++h) {
         printf("%s", "*");
      }

      puts(""); // being new line of output
   }

   // display the mode value
   printf("\nThe mode is the most frequent value.\n"
          "For this run the mode is %u which occurred"
          " %u times.\n", modeValue, largest);
}
```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 6 of 8.)

```
// function that sorts an array with bubble sort algorithm
void bubbleSort(unsigned int a[])
{
   // loop to control number of passes
   for (unsigned int pass = 1; pass < SIZE; ++pass) {

      // loop to control number of comparisons per pass
      for (size_t j = 0; j < SIZE - 1; ++j) {

         // swap elements if out of order
         if (a[j] > a[j + 1]) {
            unsigned int hold = a[j];
            a[j] = a[j + 1];
            a[j + 1] = hold;
         }
      }
   }
}

```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 7 of 8.)

```c
// output array contents (20 values per row)
void printArray(const unsigned int a[])
{
   // output array contents
   for (size_t j = 0; j < SIZE; ++j) {

      if (j % 20 == 0) { // begin new line every 20 values
         puts("");
      }

      printf("%2u", a[j]);
   }
}
```

Fig. 6.16 | Survey data analysis with arrays: computing the mean, median and mode of the data. (Part 8 of 8.)

```
********
  Mean
********
The mean is the average value of the data
items. The mean is equal to the total of
all the data items divided by the number
of data items (99). The mean value for
this run is: 681 / 99 = 6.8788
```

Fig. 6.17 | Sample run for the survey data analysis program. (Part 1 of 3.)

```
********
 Median
********
The unsorted array of responses is
 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
 6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9
 6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
 5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
 7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7

The sorted array is
 1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5
 5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8
 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

The median is element 49 of
the sorted 99 element array.
For this run the median is 7
```

Fig. 6.17 | Sample run for the survey data analysis program. (Part 2 of 3.)

```
********
  Mode
********
Response  Frequency          Histogram

                                      1    1    2    2
                                 5    0    5    0    5

       1         1           *
       2         3           ***
       3         4           ****
       4         5           *****
       5         8           ********
       6         9           *********
       7        23           ***********************
       8        27           ***************************
       9        19           *******************

The mode is the most frequent value.
For this run the mode is 8 which occurred 27 times.
```

Fig. 6.17 | Sample run for the survey data analysis program. (Part 3 of 3.)

6.9 Case Study: Computing Mean, Median and Mode Using Arrays

Mean

- The *mean* is the arithmetic average of the 99 values.
- Function `mean` (Fig. 6.16) computes the mean by totaling the 99 elements and dividing the result by 99.

6.9 Case Study: Computing Mean, Median and Mode Using Arrays (Cont.)

Median

- The median is the "*middle* value."
- Function `median` determines the median by calling function `bubbleSort` to sort the array of responses into ascending order, then picking `answer[SIZE / 2]` (the middle element) of the sorted array.
- When the number of elements is even, the median should be calculated as the mean of the two middle elements.
- Function `median` does not currently provide this capability.

6.9 Case Study: Computing Mean, Median and Mode Using Arrays (Cont.)

Mode

- The *mode* is the *value that occurs most frequently* among the 99 responses.
- Function `mode` determines the mode by counting the number of responses of each type, then selecting the value with the greatest count.
- This version of function `mode` does not handle a tie.
- Function `mode` also produces a histogram to aid in determining the mode graphically.

Lecture 2.2

Searching Arrays

6.10 Searching Arrays

- The process of finding a particular element of an array is called searching.
- In this section we discuss two searching techniques—the simple linear search technique and the more efficient (but more complex) binary search technique.

6.10 Searching Arrays (Cont.)

Searching an Array with Linear Search

- The linear search (Fig. 6.18) compares each element of the array with the search key.
- Because the array is not in any particular order, it's just as likely that the value will be found in the first element as in the last.
- On average, therefore, the program will have to compare the search key with *half* the elements of the array.

```c
// Fig. 6.18: fig06_18.c
// Linear search of an array.
#include <stdio.h>
#define SIZE 100

// function prototype
size_t linearSearch(const int array[], int key, size_t size);

// function main begins program execution
int main(void)
{
   int a[SIZE]; // create array a

   // create some data
   for (size_t x = 0; x < SIZE; ++x) {
      a[x] = 2 * x;
   }

   printf("Enter integer search key: ");
   int searchKey; // value to locate in array a
   scanf("%d", &searchKey);

```

Fig. 6.18 | Linear search of an array. (Part 1 of 3.)

```
   // attempt to locate searchKey in array a
   size_t index = linearSearch(a, searchKey, SIZE);

   // display results
   if (index != -1) {
      printf("Found value at index %d\n", index);
   }
   else {
      puts("Value not found");
   }
}

```

Fig. 6.18 | Linear search of an array. (Part 2 of 3.)

```
// compare key to every element of array until the location is found
// or until the end of array is reached; return index of element
// if key is found or -1 if key is not found
size_t linearSearch(const int array[], int key, size_t size)
{
   // loop through array
   for (size_t n = 0; n < size; ++n) {

      if (array[n] == key) {
         return n; // return location of key
      }
   }

   return -1; // key not found
}
```

```
Enter integer search key: 36
Found value at index 18
```

```
Enter integer search key: 37
Value not found
```

Fig. 6.18 | Linear search of an array. (Part 3 of 3.)

6.10 Searching Arrays (Cont.)

Searching an Array with Binary Search

- The **linear searching** method **works well for *small* or *unsorted* arrays**.
- However, for large arrays linear searching is *inefficient*.
- **If the array is sorted**, the high-speed **binary search technique can be used**.
- The binary search algorithm **eliminates** from consideration ***one-half*** of the elements in a sorted array after each comparison.
- The algorithm locates the ***middle*** element of the array and **compares** it to the search **key**.

6.10 Searching Arrays (Cont.)

- If they're equal, the search key is found and the index of that element is returned.
- If they're not equal, the problem is reduced to searching *one-half* of the array.
- If the search key is less than the middle element of the array, the *first half* of the array is searched, otherwise the *second half* of the array is searched.
- If the search key is not found in the specified subarray (piece of the original array), the algorithm is repeated on one-quarter of the original array.

6.10 Searching Arrays (Cont.)

- The search continues until the search key is equal to the middle element of a subarray, or until the subarray consists of one element that is not equal to the search key (i.e., the search key is not found).
- In a worst case-scenario, searching an array of 1023 elements takes *only* 10 comparisons using a binary search.
- Repeatedly dividing 1024 by 2 yields the values 512, 256, 128, 64, 32, 16, 8, 4, 2 and 1.
- The number 1024 (2^{10}) is divided by 2 only 10 times to get the value 1.

6.10 Searching Arrays (Cont.)

- An array of 1048576 (2^{20}) elements takes a maximum of *only* 20 comparisons to find the search key.
- An array of one billion elements takes a maximum of *only* 30 comparisons to find the search key.
- This is a tremendous increase in performance over the linear search that required comparing the search key to an average of half of the array elements.
- For a one-billion-element array, this is a difference between an average of 500 million comparisons and a maximum of 30 comparisons!

6.10 Searching Arrays (Cont.)

- The maximum comparisons for any array can be determined by finding the first power of 2 greater than the number of array elements.

- Figure 6.19 presents the *iterative* version of function `binarySearch`
- The function receives four arguments—an integer array `b` to be searched, an integer `searchKey`, the `low` array index and the `high` array index (these define the portion of the array to be searched).
- If the search key does *not* match the middle element of a subarray, the `low` index or `high` index is modified so that a smaller subarray can be searched.

6.10 Searching Arrays (Cont.)

- If the search key is *less than* the middle element, the `high` index is set to `middle - 1` and the search is continued on the elements from `low` to `middle - 1`.
- If the search key is *greater than* the middle element, the `low` index is set to `middle + 1` and the search is continued on the elements from `middle + 1` to `high`.
- The program uses an array of 15 elements.
- The first power of 2 greater than the number of elements in this array is 16 (2^4), so no more than 4 comparisons are required to find the search key.

6.10 Searching Arrays (Cont.)

- The program uses function `printHeader` to output the array indices and function `printRow` to output each subarray during the binary search process.
- The middle element in each subarray is marked with an asterisk (*) to indicate the element to which the search key is compared.

```c
// Fig. 6.19: fig06_19.c
// Binary search of a sorted array.
#include <stdio.h>
#define SIZE 15

// function prototypes
size_t binarySearch(const int b[], int searchKey, size_t low, size_t high);
void printHeader(void);
void printRow(const int b[], size_t low, size_t mid, size_t high);

// function main begins program execution
int main(void)
{
   int a[SIZE]; // create array a

   // create data
   for (size_t i = 0; i < SIZE; ++i) {
      a[i] = 2 * i;
   }

   printf("%s", "Enter a number between 0 and 28: ");
   int key; // value to locate in array a
   scanf("%d", &key);

```

Fig. 6.19 | Binary search of a sorted array. (Part 1 of 7.)

```
   printHeader();

   // search for key in array a
   size_t result = binarySearch(a, key, 0, SIZE - 1);

   // display results
   if (result != -1) {
      printf("\n%d found at index %d\n", key, result);
   }
   else {
      printf("\n%d not found\n", key);
   }
}

// function to perform binary search of an array
size_t binarySearch(const int b[], int searchKey, size_t low, size_t high)
{
   // loop until low index is greater than high index
   while (low <= high) {

      // determine middle element of subarray being searched
      size_t middle = (low + high) / 2;

```

Fig. 6.19 | Binary search of a sorted array. (Part 2 of 7.)

```
         // display subarray used in this loop iteration
         printRow(b, low, middle, high);

         // if searchKey matched middle element, return middle
         if (searchKey == b[middle]) {
            return middle;
         }

         // if searchKey is less than middle element, set new high
         else if (searchKey < b[middle]) {
            high = middle - 1; // search low end of array
         }  if

         // if searchKey is greater than middle element, set new low
         else {
            low = middle + 1; // search high end of array
         }
      } // end while

      return -1; // searchKey not found
   }

```

Fig. 6.19 | Binary search of a sorted array. (Part 3 of 7.)

```c
// Print a header for the output
void printHeader(void)
{
   puts("\nIndices:");

   // output column head
   for (unsigned int i = 0; i < SIZE; ++i) {
      printf("%3u ", i);
   }

   puts(""); // start new line of output

   // output line of - characters
   for (unsigned int i = 1; i <= 4 * SIZE; ++i) {
      printf("%s", "-");
   }

   puts(""); // start new line of output
}

```

Fig. 6.19 | Binary search of a sorted array. (Part 4 of 7.)

```c
// Print one row of output showing the current
// part of the array being processed.
void printRow(const int b[], size_t low, size_t mid, size_t high)
{
   // loop through entire array
   for (size_t i = 0; i < SIZE; ++i) {

      // display spaces if outside current subarray range
      if (i < low || i > high) {
         printf("%s", "    ");
      }
      else if (i == mid) { // display middle element
         printf("%3d*", b[i]); // mark middle value
      }
      else { // display other elements in subarray
         printf("%3d ", b[i]);
      }
   }

   puts(""); // start new line of output
}
```

Fig. 6.19 | Binary search of a sorted array. (Part 5 of 7.)

```
Enter a number between 0 and 28: 25

Indices:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
                                 16  18  20  22* 24  26  28
                                                 24  26* 28
                                                 24*

25 not found
```

Fig. 6.19 | Binary search of a sorted array. (Part 6 of 7.)

```
Enter a number between 0 and 28: 8

Indices:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12
                  8  10* 12
                  8*

8 found at index 4
```

```
Enter a number between 0 and 28: 6

Indices:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12

6 found at index 3
```

Fig. 6.19 | Binary search of a sorted array. (Part 7 of 7.)