Searching and Sorting Algorithms

CS 16: Solving Problems with Computers I
Lecture #11 PRE-RECORDED

Ziad Matni
Dept. of Computer Science, UCSB

Part 1 of 2

Lecture Outline

- Searching Algorithms
  - Sequential Search
  - Binary Search

- Sorting Algorithms
  - Selection Sort
  - Bubble Sort

Note on **getline()** Function

We will use this function often with File I/O operations (next week)

- Strings can contain whitespace characters
- But when using **cin** for user inputs, we can't capture these!!!

- Introducing: `getline()` in `<string>` lib.
- Use: **`getline(cin, string_name);`**
  - Will capture a user input WITH all whitespaces, except newline ('`\n`')
    - The newline is the default *delimiter*
- To use a different *delimiter* (not the default newline), use it like this:

  `getline(cin, string_name, character_delimiter);`

- Examples:

```
string name;
getline(cin, name);
// if user enters:
// Hello there, friend<return>
// then name will have the value
// "Hello there, friend"
```

```
string name;
getline(cin, name, ',');
// if user enters
// Hello there, friend<return>
// then name will have the value
// "Hello there"
```

Searching and Sorting Data

…Very common algorithms in CS that get used in MANY applications…

- There are many algorithms to do this with
  - Usually there's a correlation between complexity and efficiency

Some Examples

Searching

- Sequential Search
- Binary Search
- Jump Search
- Exponential Search

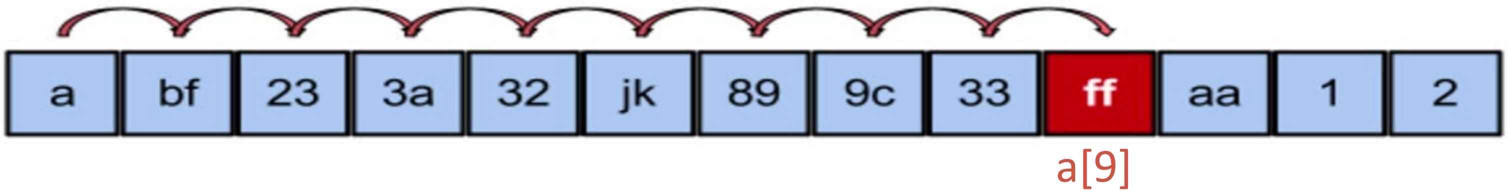
Sorting

- Selection Sort
- Bubble Sort
- Shell Sort
- Merge Sort
- Quick Sort

Sequential Search

aka Linear Search

Task: Search the array for "ff"



Result: in position 9

Sequential Search Algorithm Example

- Given: integer **array** (and it's size), integer **target**
- You want to go thru the array
  - Look for the target BUT Quit the search once you found target

- Sounds like a loop where you look thru the array
  - But it could quit at any time
  - What's better: use **for** or **while** loop??? (and why)

- While loop: keep searching as long as:
  - the target isn't found <u>**AND**</u> you haven't reached the end of the array!
  - If you find it, return the index, otherwise, return **-1** (since that can't be an index!)

```cpp
int SeqSearch
(int arr[], int array_size, int target)
{
    int index(0);
    bool found(false);   // assume not found yet!
    while ((!found) && (index < array_size))
    {
        if (arr[index] == target) {
            found = true;
        } else {
            index++;
        } // end if
    } // end while

    if (found) {
        return index;
    } else {
        return -1;
    }
}
```

See Demo File:
seqSearch.cpp

Simple Sequential Search Function Example

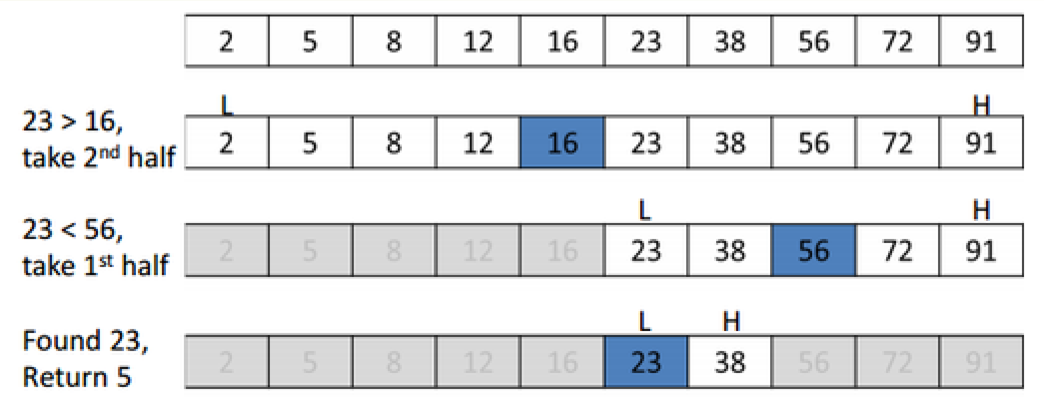
1. *Look for a target value inside of a given array*
2. *If you find it, return its location (i.e. index) in the array*
3. *If you don't find it, return -1*

Pros and Cons of Sequential Search

- PROS:
  - Simple, easy
  - Low complexity
- CONS:
  - Only finds the 1st occurrence of the target
  - Slow
    - Imagine searching a huge array and the target is near the end of it!

Binary Search

- Binary search algorithm assumes the input data is already sorted.
- Example: looking the number "23" in this array:



Binary Search

- Let’s call the array indexes: **0** through **final_index**
- Because the array is sorted, we know:

  a[0] <= a[1] <= a[2] <= … <= a[final_index]
- Like with SeqSearch, if the target is in the list, we want to know *where* it is in the list
- Twist: Since the array is sorted, split it in half!
  - Base your initial search on how target compares to median value

Binary Search: Problem Definition

- Let’s create a function that will have 5 arguments:
  - Integer array a[] and its size
  - Boolean **`found`**
  - Integers **`location`** and **`target`**

- For this example, let’s make the function of type void
  - **`found`** and **`location`** will be passed-by-reference
    - So we get 2 values that caller can use
  - **`target`** only needs to be passed-by-value

Binary Search: Problem Definition

- Pre- and Post-conditions for the function:

```
//precondition:  a[0] through a[final_index] are
//               sorted in increasing order

//postcondition: if target is not in a[0] thru a[final_index]
//            found == false;   otherwise found == true
```

Binary Search: Algorithm Design

Our algorithm:

N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12	N13
first						middle						last

- Start by looking at the item in the <u>middle</u> of the list:
  - If it is the number we are looking for, **we are done!**
  - If it is greater than the number we are looking for, **look in the 1st half of the list**
  - If it is less than the number we are looking for, **look in the 2nd half of the list**

Binary Search: Algorithm Design

1st attempt at the algorithm:

```
found = false;
mid = approx. midpoint between 0 and final_index;

if (target == a[mid])  {      // Hooray! Found it!
   found = true;
   location = mid;
}

else if (target < a[mid])
        search a[0] through a[mid -1]

else if (target > a[mid])
        search a[mid +1] through a[final_index];
```

Has to be repeated, until found or the whole array is searched

Binary Search: Algorithm Design

Next attempt at the algorithm:

```
found = false;
first_index = 0, last_index = array_size - 1;
--------------------------------------------------
mid_index = (first_index + last_index)/2

if (target == a[mid])  {      // Hooray! Found it!
   found = true;
   location = mid;
}

else if (target < a[mid])
        last_index = mid_index - 1;
else if (target > a[mid])
        first_index = mid_index + 1;
```

Has to be repeated, until found or the whole array is searched

Binary Search: A Visualization 1

TARGET

13

10	13	66	87	89	92	93	99	101	111	122	129	145
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
first						mid						last

13 < 93 *found = 0*

Therefore: last = mid – 1 = 5

10	13	66	87	89	92	93	99	101	111	122	129	145
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
first		mid			last							

13 < 66 *found = 0*

Therefore: last = mid – 1 = 1

10	13	66	87	89	92	93	99	101	111	122	129	145
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
mid first	last											

13 > 10 *found = 0*

Therefore: first = mid + 1 = 1

10	13	66	87	89	92	93	99	101	111	122	129	145
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
first	mid last											

13 = 13 *found = 1*

Binary Search: A Visualization 2

TARGET

99

10	13	66	87	89	92	93	99	101	111	122	129	145
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
first						mid						last

99 > 93

found = 0

Therefore: first = mid + 1 = 7

10	13	66	87	89	92	93	99	101	111	122	129	145
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
							first		mid			last

99 < 111

found = 0

Therefore: last = mid – 1 = 8

10	13	66	87	89	92	93	99	101	111	122	129	145
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
							first mid	last				

99 = 99

found = 0

Binary Search: Algorithm Design

- We must ensure that our algorithm eventually ends
  - No infinite loops!

- We can terminate the loop if **target** is found in the array
- BUT what if **target** is not found in the array?
  - If **first** ever becomes larger than (or equal to) **last**, we know that there are no more indices to check and key is not in the array

Final Code

See Demo File:
binSearch.cpp

```
void binSearch(int a[], int size, int target, bool &found, int &loc){
    int first = 0, last = size - 1, mid;
    found = false;

    while((first <= last) && !(found)){
        mid = (first + last)/2;

        if (target == a[mid]){
            found = true;
            loc = mid;
        }
        else if (target < a[mid]){
            last = mid - 1;
        } else {        //i.e. target > a[mid]
            first = mid + 1;
        }
    } // while
}
```

Binary Search Efficiency

- The **binary search** algorithm is *extremely fast* compared to an algorithm that checks each item in order
- The binary search **eliminates about half the elements** between **a[first]** and **a[last]** from consideration at each recursive call
- For an array of **100** items, a simple serial search will average **50** comparisons and may do as many as **100**!
  - N items, N max. comparisons
- For an array of **100** items, the binary search algorithm never compares more than **7** elements to the key!
  - N items, $\log_2 N$ max. comparisons – worst case!
  - As opposed to linear search: N items, N comparisons – worst case!

END OF PART 1

Searching and Sorting Algorithms

CS 16: Solving Problems with Computers I
Lecture #11 PRE-RECORDED

Ziad Matni
Dept. of Computer Science, UCSB

Part 2 of 2

Sorting an Array

- Sorting a list of values is another very common task
  - Create an alphabetical listing
  - Create a list of values in ascending order
  - Create a list of values in descending order

- Many sorting algorithms exist
  - Some are very efficient
  - Some are easier to understand

https://www.toptal.com/developers/sorting-algorithms

Program Example: The **Selection Sort** Algorithm

- When the sort is complete, the elements of the array are ordered in ascending order, such that:

a[0] < a[1] < … < a [number_used -1]

- This leads to an outline of an algorithm:

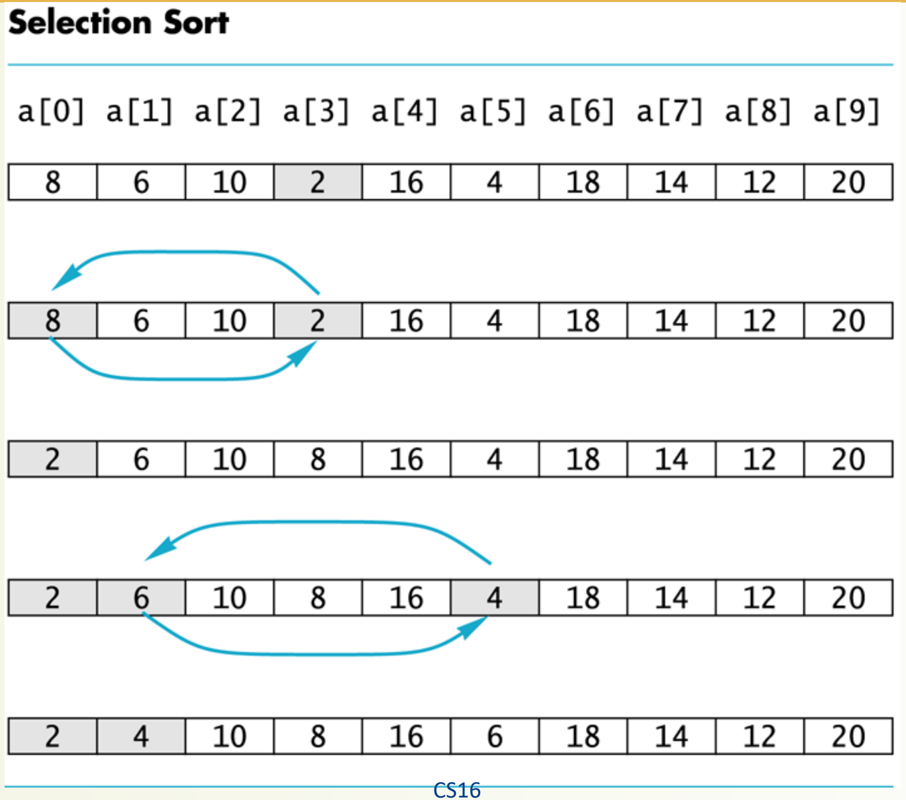
*for (int index = 0; **index < number_used**; index++)*
place the indexth smallest element in a[index]

Program Example: Selection Sort Algorithm Development

(Also, see Display 7.10 in the textbook)

- One array is sufficient to do our sorting
  - i.e. you don't really need 2 arrays
- Search for the *smallest* value in the array
- Place this value in a[0], and place the value that was in a[0] in the location where the smallest was found
  - i.e. swap them
- Starting at a[1], find the smallest remaining value swap it with the value currently in a[1]
- Starting at a[2], continue the process until the array is sorted

Sort from smallest to largest



Final Code

See Demo File: **selectionSort.cpp**

```
void swap_values(int& v1, int& v2){
    int temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}

int index_of_smallest
(int a[], int start_index, int number_used){
    int min = a[start_index];
    int index_of_min = start_index;

    for(int index = start_index + 1; index < number_used; index++){
        if (a[index] < min){
            min = a[index];
            index_of_min = index;
        }
    }
    return index_of_min;
}
```

```
void sort(int a[], int number_used){
    int index_of_next_smallest;

    for(int index = 0; index < number_used - 1; index++){
        index_of_next_smallest = index_of_smallest(a, index, number_used);
        swap_values(a[index], a[index_of_next_smallest]);
    }
}
```

Bubble Sort

- Similar to Selection Sort in terms of efficiency
  - A little easier to understand
- It keeps exchanging (sorting) 2 adjacent elements at a time, if they need to be.
  - When no more exchanges are required, the list/array is sorted

Final Code

See Demo File: bubbleSort.cpp

```
void bubbleSort(int array[], int size)
{
  int temp;
  for (int i = size-1; i >= 0; i--) {
    for (int j = 1; j <= i; j++) {
      if (array[j-1] > array[j]) {
        temp = array[j-1];
        array[j-1] = array[j];
        array[j] = temp;
      } // if
    } // for j
  } // for i
}
```

</LECTURE>