Chapter Three: Simple Searching and Sorting Algorithms

- Why searching and sorting algorithms in preference from other algorithms?
  - Searching and sorting are the most important tasks performed by computer
  - They consume (take) greater than 25 % of the running time of all computers in the world
    - That is, > 25 % of running time of all computers in the world is spent by searching and sorting data
  - Searching and sorting algorithms play an important role in the field of Theory and Algorithms.
- What is searching?
  - Searching is a process of finding or locating a desired element from the a collection of elements

- Searching a list of values is a common task

- **Examples of searching**: an application program might

  - Retrieve a record in a database matching a certain criteria (e.g. Name = "Scott")
  - Retrieve or Search a student record from a database
  - Retrieve or search a specific document from the Internet
  - Searching a file from the hard disk
  - Retrieve a bank account record, credit record
  - Deleting a record from a database (requires locating the required item from a database)
  - Find the maximum of a set of values
  - Find the minimum of a set of values
  - The median of a set of values
  - ... **Using searching algorithms**
- All these are performed by our computers

- Two types of simple searching algorithms

  - Sequential searching
  - Binary searching

  Sequential Searching
- Is also called linear search, serial search
- Doesn't need sorting of the elements
- Is the most natural way of finding (searching) an element from a collection of elements
- Easy to understand and implement the algorithm

***Sequential Search* of a student database**

	name	student	major	credit hrs.
1	John Smith	58567	physics	36
2	Paula Jones	36794	history	125
3	Led Belly	85674	music	72
	.	.	.	.
	.	.	.	.
	.	.	.	.
n	Chuck Bin	43687	math	89

1. ask user to enter studentNum to search for
2. set i to 1
3. set found to 'no'
4. while i <= n and found = 'no' do
5. if studentNum = $student_i$ then set found to 'yes'
   else increment i by 1
7. if found = 'no' then print "no such student"
   else < student found at array index *i* >

algorithm
to search database
by *student #* :

Algorithm for sequential search

- Loop through the list starting at the first element until the value of target matches one of the list elements.
- If a match is not found, return –1.

i.e : Steps through the list from the beginning one item at a time looking for the desired item

  - The search stops when the item (that is the key) is found or when the search has examined each item without success (that is until end of list is reached)

- The technique is often used because it is easy to write and is applicable to many situations
- Following is the same as the algorithm for sequential search written using the C++ programming language

```
int linearSearch(int list[], int key, int n) {
        int index=0;
        int found=0;
 do{
        if(key==list[index])
        found=1;
        else
        index++;
 }while(found==0&&index<n);

        if(!found)
        index=-1;
        return index;
        }
```

Analysis of Sequential Search

- How do we estimate the complexity of this algorithm?
- Having implemented the sequential search algorithm, how would you measure its efficiency?
- What is the order of sequential searching algorithm?
- A useful metric would be **general** (That is, applicable to any (search) algorithm)
- Since more efficient algorithm take less time to execute, one approach would be to write a program for each of the algorithm, and measure the time each takes to finish
- However, a more efficient metric would be one that allows algorithms to be evaluated before implementing them
- One such metric is the number of main steps the algorithm will require to finish
- That is, as always, we will count the number of operations required by the algorithm rather than measuring actual elapsed time

- Of course, the exact number of steps depends on the input data
- For sequential search algorithm, the number of steps depends on the target is in the list, and if so, where in the list, as well as on the length of the list
- For searching algorithms, the main steps are comparisons of list values with the target value
- Count these to compute the complexity of sequential search for data models representing the best case, worst case and average case

The main task of the searching algorithm is comparison of the key with data elements

Time requirements for sequential search

- *best case* (minimum amount of work):
  studentNum found in $student_1$ one loop iteration
- *worst case* (maximum amount of work):
  studentNum found in $student_n$ *n* loop iterations
- *average case* (expected amount of work):
  studentNum found in $student_{n/2}$ *n*/2 loop iterations

because the amount of work is a constant multiple of n, the time requirement is O(n) in the worst case and the average case

- **Best case (minimum amount of work)**
  - Is defined as the smallest of all the running times on inputs of a particular size
  - As its name suggests, it takes the optimistic view possible
  - We assume the target (the element that we want) is the first element in the list
  - Then there will be only one comparison
  - Thus, For a list (an array) of n elements, the best case-case cpmplexity of sequential search just one comparison (or array access)

That is, $T_{best}(n) = O(1)$

- **Worst case (maximum amount of work):**
  - we consider the hardest input (a target that requires the algorithm to compare the largest number of elements in the list)

For the serial search, the worst case running time occurs either when the target is not in the list or the target is the last element of the list

- For a list of n elements, the worst case running time of sequential search requires **n comparisons**
- **$T_{worst}(n) = O(n)$**
  - Simply the number of comparison in the body is one and this is multiplied by the number of iterations

Time requirements for sequential search

- **O(n) searching is too slow**
- Consider searching AAU's student database using sequential search on a computer capable of 20,000 integer comparisons per second:
- n = 150,000 (students registered during past 10 years)

Average case

150,000 comparisons * 1 seconds/20,000 comparisons = 7.5 seconds

Worst case

150,000 comparisons * 1 second /20,000 comparisons = 7.5 seconds

- Average case (expected amount of work)
  - One way of developing an expression for the average running time of serial search is based on all the targets that are actually in the list
  - Suppose (to be concrete) the list has ten elements
  - So, there are ten possible targets
  - A search for the target that occurs at the first location requires just one comparison (or array access)
  - A search for the target that occurs at the second location requires two comparisons (or array accesses)
  - And so on, through the final target, which requires ten comparisons (or array access) search for
  - In all there are ten possible targets which requires 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 comparisons (or array accesses)
- Average case (expected amount of work)
  - The average of all theses searches is:
  - Generalization
    - The average-case running time of the serial search can be written as an expression with part of the expression being the size of the list (array)
    - Using n as the size of the array, this expression for the average-case running time is:
  - $\mathbf{T_{worst}(n) = O(n)}$

Summary

- The best case running time of sequential search is in O(1)
- Both the worst-case and average case running time for sequential search are in O(n) expression, **but nevertheless, the average case is about half the time of the worst case**

Pros and Cons of Sequential Search Algorithm

- The principle strength of linear search is that it doesn't require that the elements of the list be in any particular order, only that we can step through the list.
- It is easy to implement
- It is easy to analyze
- It is fine to use if you are searching a small list (or array) a few times

However, if a search algorithm will be used over and over, it is worthwhile to find a faster algorithm

- One dramatically faster search algorithm is binary search

- Write a small C++ program that demonstrates sequential search. The program should read a list from the user, prompts for a search target, and then uses sequential search to look for the target in the list. The program should include extra output statements to display steps in the search. The list (which is an array) is partially filled. Since the integers in the list are meant to represent student ID, it is reasonable to assume that they are all positive. Thus, you can use a sentinel entry of – to mark the end of the list

Binary search

- When the values of a list are in sorted order, there are better searches than sequential search for determining whether a particular value is in the list
- For example, when you look up a name on the phone book, you do not start in the beginning and scan through until you find the name
- You use the fact that names are listed in sorted order and use some intelligence to jump quickly to the right page and then start scanning
- Binary search is such an example, a better search than sequential search for determining whether a particular value is in the list
- Works only if the data is sorted
- It uses divide and conquer strategy (approach)
- How it works
    - A flag called **found** is set to false, meaning NOT found.
    - **middle** is calculated

- The value at the **middle index position** in the array is checked to see if it matches the value the user was searching for

- If a match is found, the **found flag** is set to **true** so the loop will fail, preventing unnecessary iterations.
- If the value is not found...
- Since the array is sorted, we can check to see which half of the array the value is in
- If the value is in the lower half, we change **last** so the new array goes **from 0 to middle - 1.**
- This has the effect of discarding the larger half of the array
- We then test to see if the value stored in the (new) **middle** position is the value we were searching for
- If it is not, the process of dividing the array continues until either the value is found or the loop terminates
- A similar procedure develops if the value is in the larger half of the array except the **variable first** is changed to the position just after **middle (middle + 1)** and ends at **last**
- If the value does not exist, the loop will terminate after the array has been subdivided so many times, the **last position** is a **smaller integer** than the first position, which of course is impossible in an array
- **That is, the search ends when the target item is found or the values of first and last cross over, so that last < first, indicating that no list items are left to check**

Psuedocode for Binary Search

1. Locate midpoint of array to search
2. Determine if target is in lower half or upper half of an array.
    i. If in lower half, make this half the array to search
    ii. If in the upper half, make this half the array to search
3. Loop back to step 1 until the size of the array to search is one, and this element does not match, in which case return –1.

Implementation of Binary sereach algorithm in c++

```
int Binary_Search(int list[],int k) {
        int left=0;
        int right=n-1;
        int found=0;
do{
        mid=(left+right)/2;
        if(key==list[mid])
```

```
        found=1;
else
{
if(key<list[mid])
        right=mid-1;
else
        left=mid+1;
}
}while(found==0&&left<right);

if(!found)
  index=-1;
else
   index=mid;
return index;
}
```

- Remark
    - It might appear that this algorithm changes the list because each time through the while loop half of the list is “discarded”, so that at the end the list may contain only one element
    - Two reasons to avoid this
        - The list may be needed again, to search for other targets, for example
        - It creates extra work to delete half of the items in a list, or make a copy of the list
    - Rather than changing the list, the algorithm changes the part of the list to search
    - **Binary search uses the result of each comparison to eliminate half of the list from further searching**
- **Analysis Binary Search**
- What is the complexity (or maximum number of comparisons made) of binary search?
- Is binary search more efficient than sequential search?
- If so, how much more efficient is it?
- To evaluate binary search, count the number of comparisons in the best case and worst case (why?)
- Complexity analysis for the average case is a bit more difficult and hence will be omitted
- Best case
    - Occurs if the middle item happens to be the target
    - Then, only one comparison is needed to find the target
    - Thus, $\mathbf{T_{best}(n) = O(1)}$

        - Remark
            - The best case analysis does not reveal much
- Worst case
    - When does the worst case occur?
    - If the target is not in the list then the processes of dividing the list in half continues until there is only one item left to check
    - Here is the pattern of the number of comparisons after each division, given the simplifying assumptions of an initial list length that is an even power of 2 (1024) and exact division in half on each iteration
    - **One comparison, reduces the number of items left to check by half**
    - For a list of size 1024, there are 10 comparisons to reach a list of size one, given that there is one comparison for each division, and each division splits the list size in half
    - Generalization
        - If n is the size of the list to be searched and C is the number of comparisons to do so in worst case, **C = logn**, base is 2
    - Thus the efficiency of binary search can be expressed as a logarithmic function in which the number of comparisons required to find a target increases logarithmically with the size of the list
    - **$T_{worst}(n) = O(logn)$ (that is, binary search efficiency for the worst case ia a logarithmic function of list size)**

Sequential Vs Binary search

- Here is a table showing how the maximum number of comparisons for sequential and binary searches

Size of List	**Maximum number of comparisons**	
	Sequential search	**Binary search**
100,000	**100,000**	**16**
200,000	**200,000**	**17**

400,000	**400,000**	**18**
800,000	**800,000**	**19**
1,600,000	**1,600,000**	**20**

- **Note**
  - The worst case number of comparisons is just 16 for a list with 100,000 items, versus 100,000 for sequential search
  - Furthermore, if the list were doubled in size to 200,000, the maximum number of comparisons for binary search would only increase by 1 to 17
  - Whereas for sequential search it would double from 100,000 to 200,000
- **In general**
  - **Binary search algorithm is more efficient than sequential searching algorithms because**

logn << n

Sequential search	**Binary search**
Easy to understand and implement	**Difficult to understand**
Doesn't assume sorted data	**Assumes sorted data**
O(n)	**O(logn)**

- One can estimate from the graph which one performs better (Include graph)
- From the graph, you can see that
  - For small number of data sequence, sequential search is more efficient than binary search
  - For sorted and large number of data, binary search is more efficient

Sorting Algorithms

- Another very common operation is *sorting* a list of values or data items
- Sorting is a process of reordering a list of items in either increasing or decreasing order
- Sorting a list places the list in a specified order, such as alphabetical order or rank order (smallest to largest).
- one of most frequently performed tasks
- intensively studied
- many classic algorithms
- there remain unsolved problems
- new algorithms are still being developed
- refinements are very important for special cases
- good for illustrating analysis and complexity issues
- motivate use of file processing (when main memory is too small)
- amenable to parallel implementation
- Example
  - Sorting a list of students in ascending order
  - Sorting file search results by name, date modified, type
- All these are operations performed by our computers

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 in 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.
- Some of the algorithms that are internal are:
  - Bubble Sort
  - Insertion Sort
  - Selection Sort
  - Shell Sort

 Heap Sort

External sorting:

- Cannot 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
- Some of the algorithms that are external are:
  - Simple algorithm- uses merge routine from merge sort
  - Multiway merge
  - Polypbase merge
  - Replacement selection

The reset of this chapter will discus and analyze various internal sorting algorithms.
− There are several algorithms that are easy and are in the order of O (n2).
− There is also algorithm (ShellSort) that is very simple to implement and run in and O(n2) practically efficient.
− There are slightly complicated O(nlogn) sorting algorithms.
− Any general purpose sorting algorithm requires Ω(nlogn) comparisons.

Assumptions:

− Our sorting is comparison based.
− Each algorithm will be passed an array containing N elements.
− N is the number of elements passed to our sorting algorithm
− The operators that will be used are "<",">", and"=="

Simple sorting algorithms

- The following are simple sorting algorithms used to sort small-sized lists.
  - Insertion Sort
  - Selection Sort
  - Bubble Sort

Bubble Sort

Bubble sort is the simplest algorithm to implement and the slowest algorithm on very large inputs.

The basic idea is: Loop through list and compare adjacent pair of elements and when every two elements are out of order with respect to each other interchange them.

Target: Pull a least element to upper position s bubble.
− The process of sequentially traversing through all part of list is known as pass.
− **Implementation:** This algorithm needs two nested loops. The outer loop controls the number of passes through the list and inner loop controls the number of adjacent comparisons.

Example: - suppose the following list of numbers are stored in an array a.

Elements	7	20	15	3	72	13	11	32	9
Index	0	1	2	3	4	5	6	7	8

Index	**Elements**	**Pass1**	**Pass2**	**Pass3**	**Pass4**	**Pass 5**	**Pass 6**	**Pass 7**
7	**7**	**72**	**72**	**72**	**72**	**72**	**72**	**72**
6	**20**	**7**	**20**	**20**	**20**	**20**	**20**	**20**
5	**15**	**20**	**7**	**17**	**17**	**17**	**17**	**17**
4	**3**	**17**	**17**	**7**	**13**	**13**	**13**	**13**
3	**72**	**3**	**13**	**13**	**7**	**11**	**11**	**11**
2	**13**	**13**	**3**	**11**	**11**	**7**	**9**	**9**
1	**11**	**11**	**11**	**3**	**9**	**9**	**7**	**7**
0	**9**	**9**	**9**	**9**	**3**	**3**	**3**	**3**

Implementation:

```
void bubble_sort(list[])
{
 int i,j,temp;
 for(i=0;i<n; i++){
```

```
        for(j=n-1;j>i; j--){
            if(list[j]<list[j-1]){
                temp=list[j];
                list[j]=list[j-1];
                list[j-1]=temp;
                }//swap adjacent elements
            }//end of inner loop
        }//end of outer loop
    }//end of bubble_sort
```

Analysis of Bubble Sort

How many comparisons?

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

How many swaps?

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

Space?

In-place algorithm.

Conclusion: bubble sort is terrible in all cases

- If there is no change in the ith pass it implies that the elements are sorted(sorting is done) earlier. Thus there is no need to continue the remaining pass.
- In exploring this part one can develop a modified bubble sort algorithm that has better performance.

Modified bubble sort:

```
void mBubbleSort(int a[],int n)
  {
         int i=1;
         int swapped =1;

       while(swapped && i< n)
          {
          swapped=0;
          for(j=0;j<i; j++)
              if(a[j]>a[j+1])
                 {
                    swap(a,[j],a[ j+1]);
                    swapped=1;
                  }
                   i++;
              }
}//end of MbubbleSort
```

− **Analysis**: For the modified bubble sort algorithm the big-O is:
Best case: O(n)
Average case: $O(n^2)$

Selection Sort

- The selection sort algorithm is similarly motivated as bubble sort. However in the case of selection sort it attempts to avoid the multitude of interchanges of adjacent entries to which bubble sort is prone.
- *Selection sort* iterates over the list, placing one element in the correct place each time.
- To do this on the i_{th} pass through the array it will determine the position of the smallest/largest entry among the list of elements then this element is swapped.
- This algorithm uses its inner loop to find the largest/smallest entry.
- One version of selection sort finds the smallest element in the list and swaps it with whatever is first, and then repeats with each element of the list.
- Note that this also uses linear search to find the smallest element each time.
- The xth pass of the selection sort will select the xth smallest key in the array, placing that record into position x.

Basic Idea:

- Loop through the array 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.

Implementation of selection sort in C++

```
void selection_sort(int list[])
{
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
```

Index	Elements	Pass 1	Pass 2	Pass 3	Pass 4	Pass 5	Pass 6	Pass 7
7	**7**	**9**	**56**	**56**	**56**	**56**	**56**	**72**
6	**20**	**20**	**20**	**20**	**20**	**24**	**72**	**56**
5	**15**	**15**	**15**	**72**	**72**	**72**	**24**	**24**
4	**19**	**19**	**19**	**19**	**24**	**20**	**20**	**20**
3	**24**	**24**	**24**	**24**	**19**	**19**	**19**	**19**
2	**72**	**72**	**72**	**15**	**15**	**15**	**15**	**15**
1	**56**	**56**	**9**	**9**	**9**	**9**	**9**	**9**
0	**9**	**7**	**7**	**7**	**7**	**7**	**7**	**7**

ANALYSIS of Selection Sort

- Selection sort is a bubble sort (see later), except that rather than repeatedly swapping adjacent elements we remember the position of the element to be selected and do the swap at the end of the inner loop.

How many comparisons?

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

How many swaps?

$n=O(n)$

How much space?

In-place algorithm

Insertion Sort

The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place.

It's the most instinctive type of sorting algorithm. The approach is the same approach that you use for sorting a set of cards in your hand. While playing cards, you pick up a card, start at the beginning of your hand and find the place to insert the new card, insert it and move all the others up one place.

Basic Idea:

Find the location for an element and move all others up, and insert the element.

The process involved in insertion sort is as follows:

1. The left most value can be said to be sorted relative to itself. Thus, we don't need to do anything.
2. Check to see if the second value is smaller than the first one. If it is, swap these two values. The first two values are now relatively sorted.
3. Next, we need to insert the third value in to the relatively sorted portion so that after insertion, the portion will still be relatively sorted.

4. Remove the third value first. Slide the second value to make room for insertion. Insert the value in the appropriate position.
5. Now the first three are relatively sorted.
6. Do the same for the remaining items in the list.

Implementation

```
void insertion_sort(int list[]){

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

How many comparisons?

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

How many swaps?

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

How much space?

In-place algorithm