Guidelines for B.Sc. (H) Computer Science Semester IV (NEP UGCF 2022)

Design and Analysis of Algorithms

S. No.	Topics	Contents	Reference	Hours
1	Unit 1# - Searching, Sorting, Selection	Ch 2 - 2.1, 2.2 Ch 6 - 6.4 Ch 8 - 8.1, 8.2, 8.3 Ch 9 - 9.1	[1]	8
2	Unit 2 - Graphs	Ch 3	[2]	7
3	Unit 3 - Divide and Conquer	Ch 2 - 2.3 Ch 4 - 4.2 Ch 7 - 7.1, 7.2, 7.4 (upto 7.4.1)	[1]	8
4	Unit 4 - Greedy algorithms	Ch 15 - 15.2 Ch 4 - 4.1 (upto page 121, excluding Extensions), 4.5 (only Prim's Algorithm and related theorems/sub-sections to be discussed)	[1] [2]	7
5	Unit 5 - Dynamic Programming	Ch 6 - 6.1, 6.2, 6.4	[2]	7
6	Unit 6 - Hash Functions, Collision resolution schemes	Ch 11 - 11.2 (except Independent Uniform Hashing and Analysis of Hashing with chaining), 11.3 (upto and including multiplication method), 11.4 (except Analysis of open-address hashing)	[1]	8

# - Linear search, binary search, bubble sort and selection sort are to be covered from the appendix attached at the end of the guidelines.

References

1. Cormen, T.H., Leiserson, C.E., Rivest, R. L., Stein C. Introduction to Algorithms, 4th edition, Prentice Hall of India, 2022.
2. Kleinberg, J., Tardos, E. *Algorithm Design,* 1st edition, Pearson, 2013.

Additional References

i. Basse, S., Gelder, A. V., *Computer Algorithms: Introduction to Design and Analysis,* 3rd edition, Pearson, 1999.

Practical List

1) Write a program to sort the elements of an array using Insertion Sort (The program should report the number of comparisons).
2) Write a program to sort the elements of an array using Merge Sort (The program should report the number of comparisons).
3) Write a program to sort the elements of an array using Heap Sort (The program should report the number of comparisons).
4) Write a program to sort the elements of an array using Quick Sort (The program should report the number of comparisons).
5) Write a program to multiply two matrices using the Strassen's algorithm for matrix multiplication.
6) Write a program to sort the elements of an array using Count Sort.
7) Display the data stored in a given graph using the Breadth-First Search algorithm.
8) Display the data stored in a given graph using the Depth-First Search algorithm.
9) Write a program to determine a minimum spanning tree of a graph using the Prim's algorithm.
10) Write a program to solve the 0-1 knapsack problem.

For the algorithms at S. no 1 to 4, test run the algorithm on 100 different input sizes varying from 30 to 1000. For each size find the number of comparisons averaged on 10 different input instances; plot a graph for the average number of comparisons against each input size. Compare it with a graph of n logn.

APPENDIX

MCSC 101: Design and Analysis of Algorithms

Neelima Gupta

ngupta@cs.du.ac.in

December 30, 2020

Pre-requisites

- Familiarity with Basic Data Structures, Algorithms for Problems like Searching, Sorting and Selection, Algorithm Design techniques like Divide and Conquer, Greedy and DP
- Mathematical techniques like Induction and Proof by Contradiction
- We will review some of these quickly
- Those who are not familiar with these will have to work harder
- Some notes are available on my webpage: `http://people.du.ac.in/~ngupta/teaching.html`

Linear Search : Correctness

Let us rewrite the linear search algorithm of chapter 2 as follows:

```
input  : Array: $A[1 \ldots n], Key$
output: index of first occurrence of key if it is found, 0
        otherwise
$i = 1$
while $i \leq n$ do
    if $A[i] = key$ then
        return $i$
    end
    else $i{+}{+}$
end
return 0
```

Steps to Prove

- Loop Invariance
- Prove the Loop Invariance using Mathematical Induction
- Use the Loop Invariance to prove the correctness

Loop Invariance

```
$i = 1$
while $i \leq n$ do
    if $A[i] = key$ then
        return $i$
    end
    else $i{+}{+}$
end
return 0
```

Hypothesis $H(r)$: When the control reaches the "While" statement for the r^{th} time,

1. $i = r$
2. key $\notin A[1 \ldots r-1]$

$\forall\ 1 \leq r \leq n+1$

Proof Of Loop Invariance

Hypothesis $H(r)$: When the control reaches the "While" statement for the r^{th} time,

1. $i = r$
2. key $\notin A[1 \ldots r-1]$

Proof by induction on r.

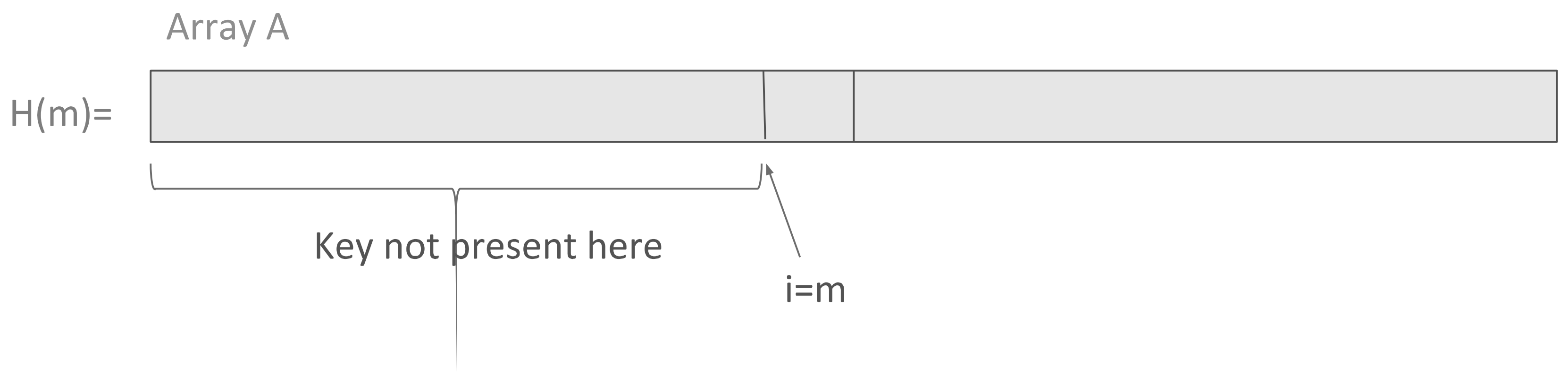
1. Base Case: when $r = 1$, the claim holds vacuously.
2. Induction Hypothesis: $H(m) \Rightarrow H(m+1)$.

Induction Hypothesis

```
$i = 1$
while $i \leq n$ do
    if $A[i] = key$ then
        return $i$
    end
    else $i{+}{+}$
end
return 0
```

Hypothesis $H(r)$: When the control reaches the "While" statement for the r^{th} time,

1. $i = r$
2. key $\notin A[1 \ldots r-1]$

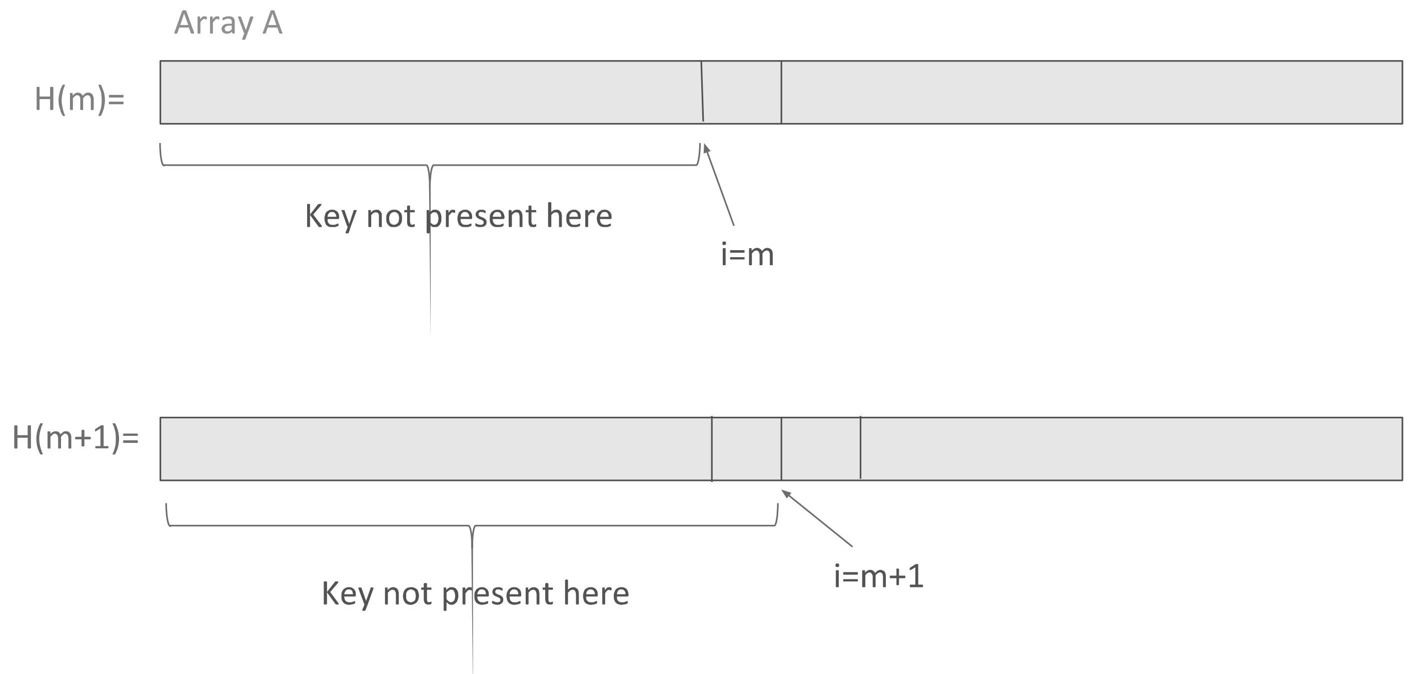


Induction Hypothesis

```
$i = 1$
while $i \leq n$ do
    if $A[i] = key$ then
        return $i$
    end
    else $i$++
end
return 0
```

Hypothesis $H(r)$: When the control reaches the "While" statement for the r^{th} time,

1. $i = r$
2. key $\notin A[1 \dots r-1]$

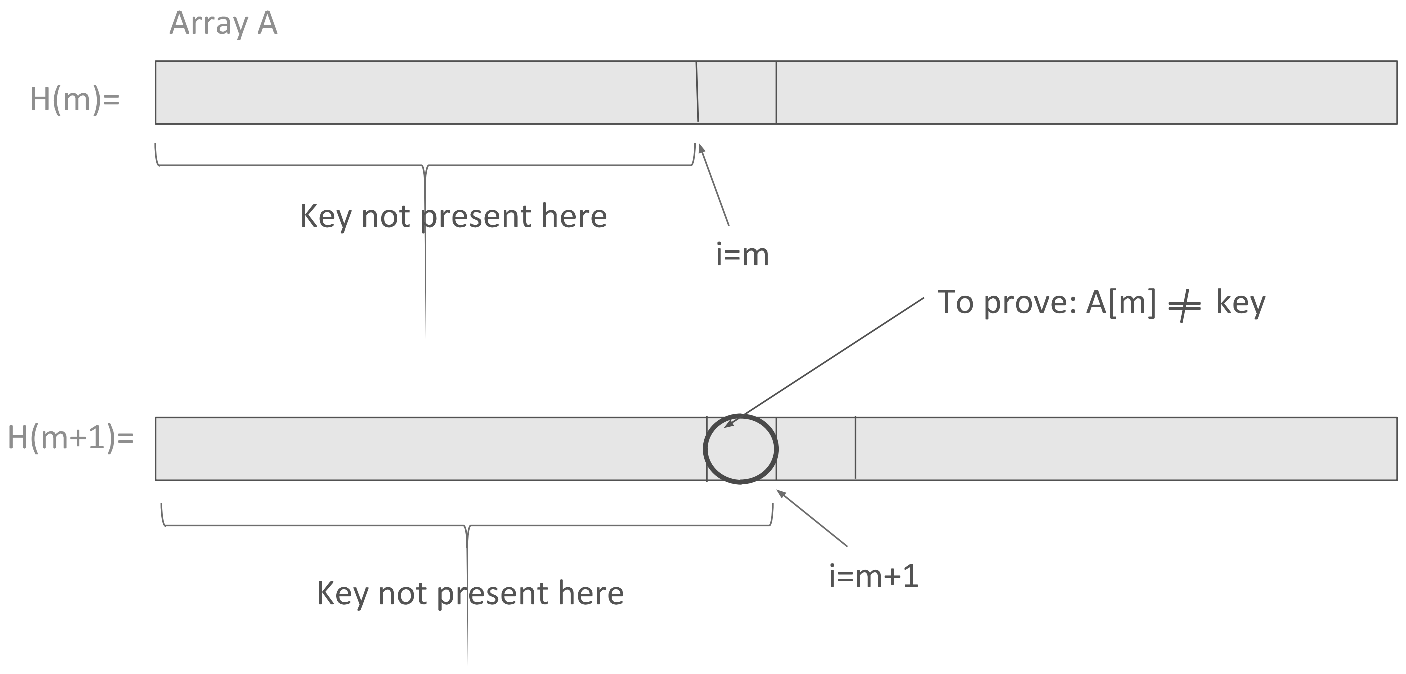


Induction Hypothesis

```
$i = 1$
while $i \leq n$ do
    if $A[i] = key$ then
        return $i$
    end
    else $i$++
end
return 0
```

Hypothesis $H(r)$: When the control reaches the "While" statement for the r^{th} time,

1. $i = r$
2. key $\notin A[1 \dots r-1]$



Proof of Correctness

```
$i = 1$
while $i \leq n$ do
    if $A[i] = key$ then
        return $i$
    end
    else $i{+}{+}$
end
return 0
```

Hypothesis $H(r)$: When the control reaches the "While" statement for the r^{th} time,

1. $i = r$
2. key $\notin A[1 \dots r-1]$

Suppose that the test condition in "While" statement is executed exactly k times. i.e. body of the loop is executed $k-1$ times.

1. Case 1: $k \leq n$. In this case, $i = k$ and $A[1 \dots k-1]$ does not contain the key. Also, the while loop terminated because $A[i](i.eA[k]) = key$. Thus the value returned in statement 4 is the position of the first occurrence of the key in the array.
2. Case 2: $k = n+1$. By loop invariant hypothesis, $i = n+1$ and $A[1 \dots n]$ does not contain the key. Since $i = n+1$, control goes to statement 8 and the algorithm returns 0.

MCAC 301

Design and Analysis of Algorithms

Neelima Gupta

ngupta@cs.du.ac.in

PseudoCode
Eg. Linear Search

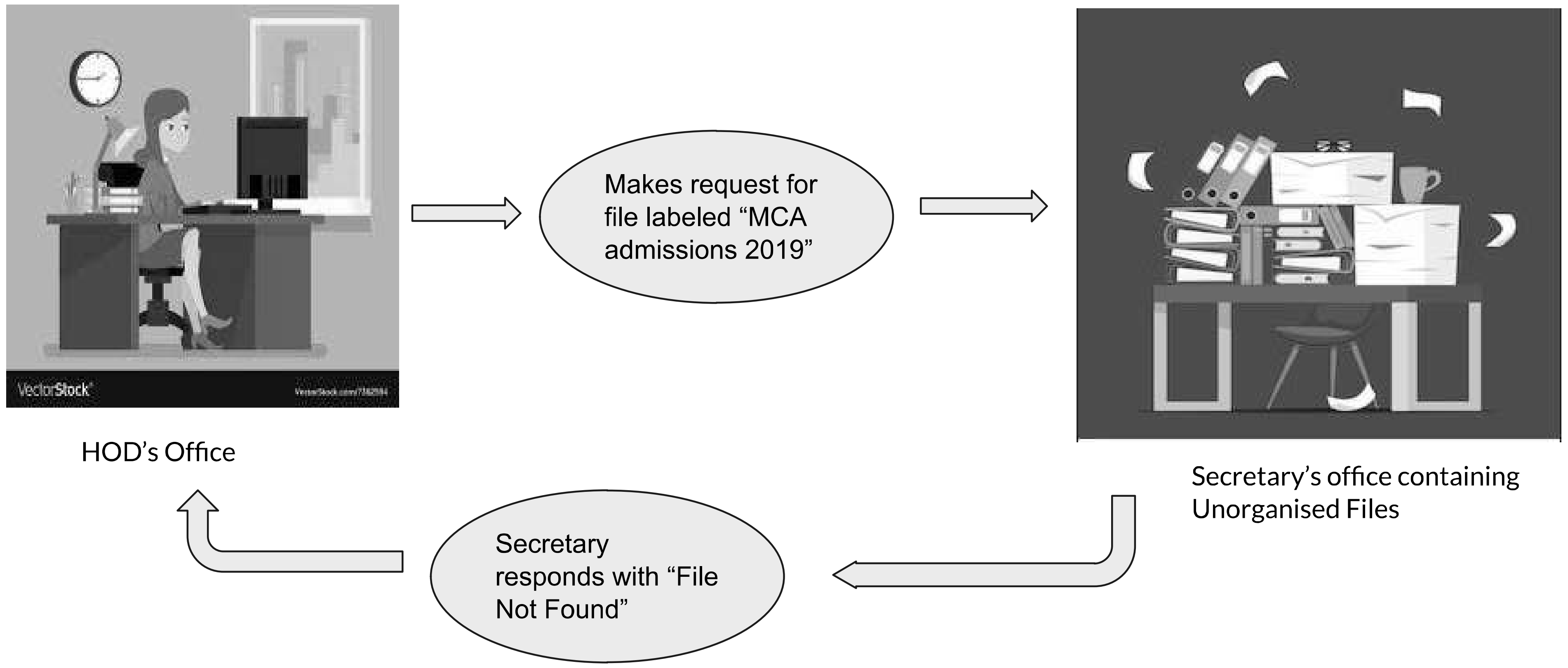
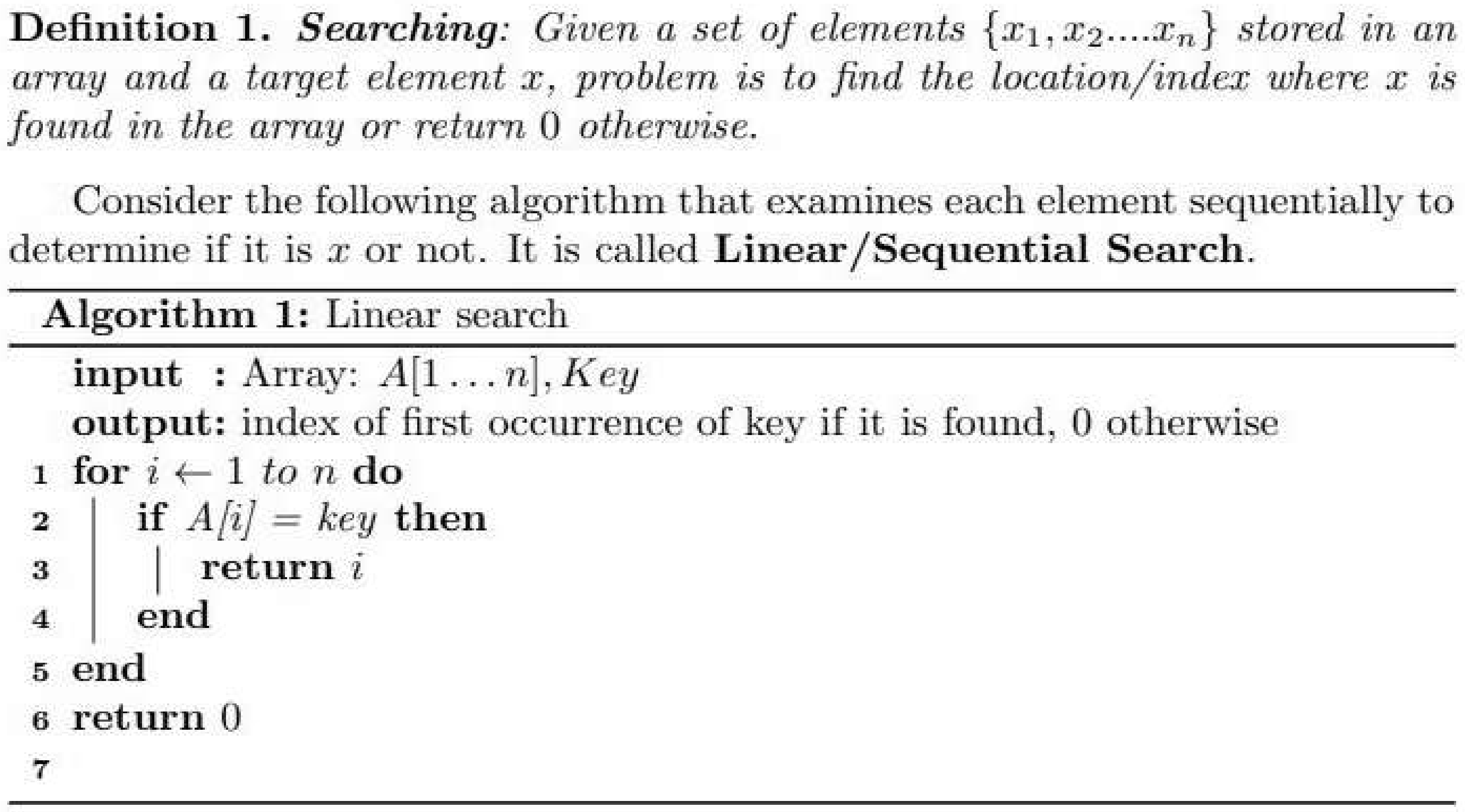

VectorStock
VectorStock.com/7182584
HOD's Office
Makes request for file labeled "MCA admissions 2019"
Secretary's office containing Unorganised Files
Secretary responds with "File Not Found"

Figure A: Secretary at an office responding to the requests of the Head

- If there were n files and the requested file was not found, how many comparisons did the secretary have to perform?

Searching



Definition 1. ***Searching***: *Given a set of elements $\{x_1, x_2....x_n\}$ stored in an array and a target element x, problem is to find the location/index where x is found in the array or return 0 otherwise.*

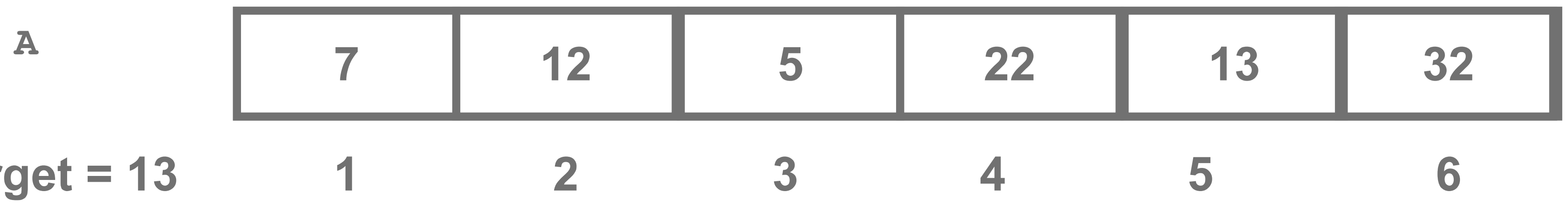
Consider the following algorithm that examines each element sequentially to determine if it is x or not. It is called **Linear/Sequential Search**.

Algorithm 1: Linear search

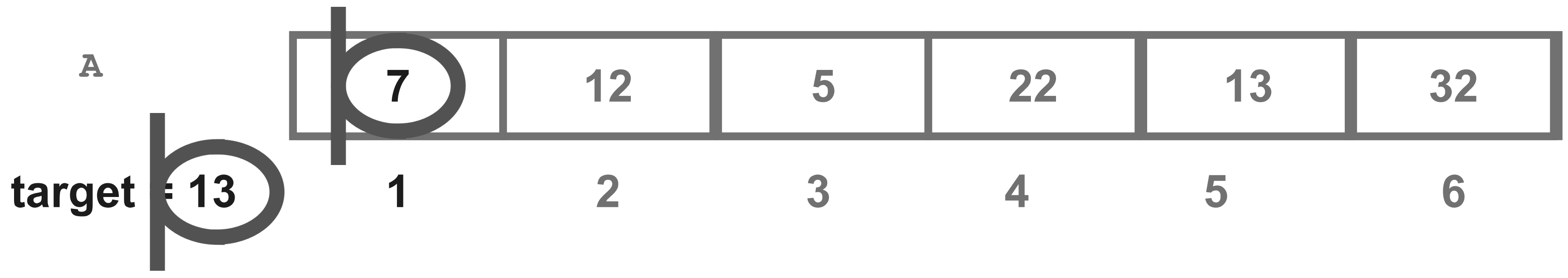
```
input  : Array: $A[1 \dots n], Key$
output: index of first occurrence of key if it is found, 0 otherwise
for $i \leftarrow 1$ to $n$ do
    if $A[i] = key$ then
        return $i$
    end
end
return 0
```

Note: We use indices starting from 1 to n in algorithms (pseudocode) and not 0 to n-1

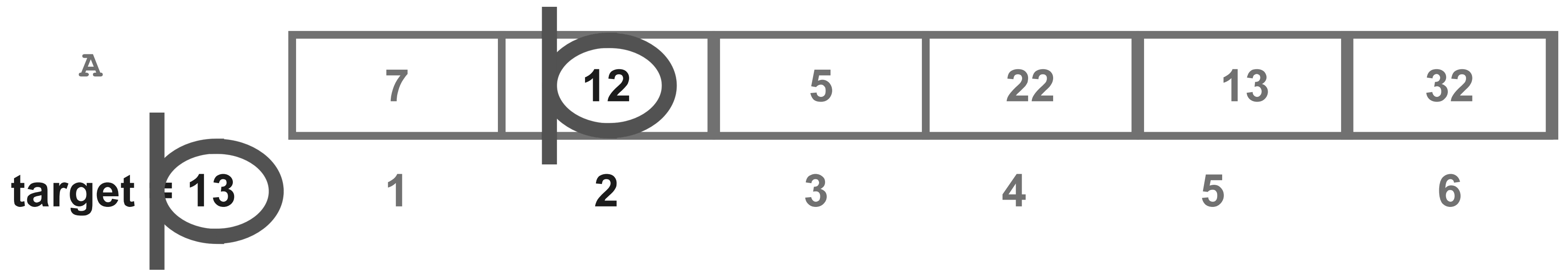
Example of Linear Search



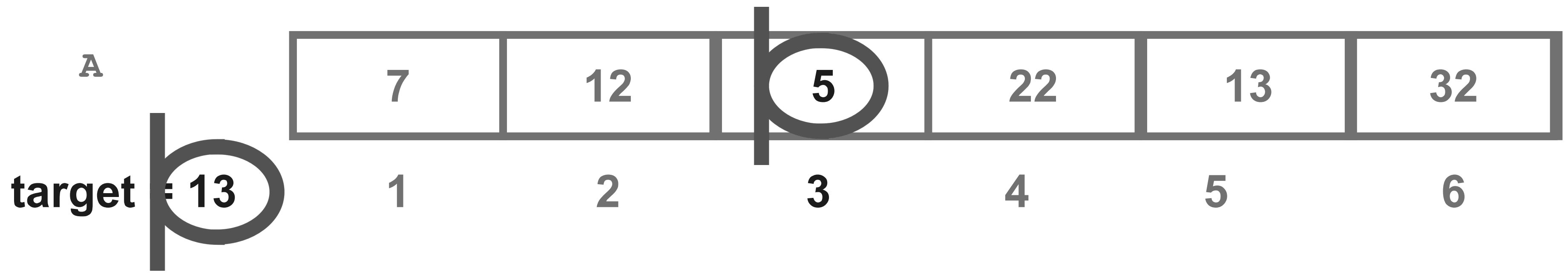
https://www.cc.gatech.edu/home/bleahy/cs1311/cs1311lecture15wdl.



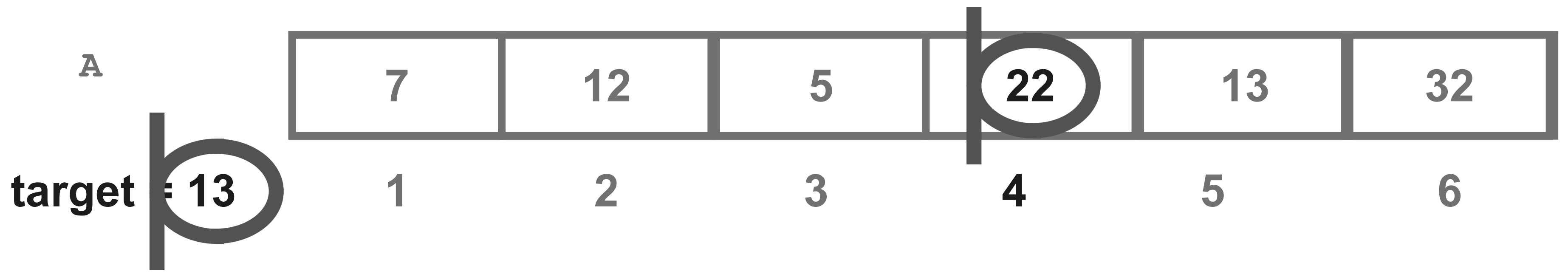
https://www.cc.gatech.edu/home/bleahy/cs1311/cs1311lecture15wdl.



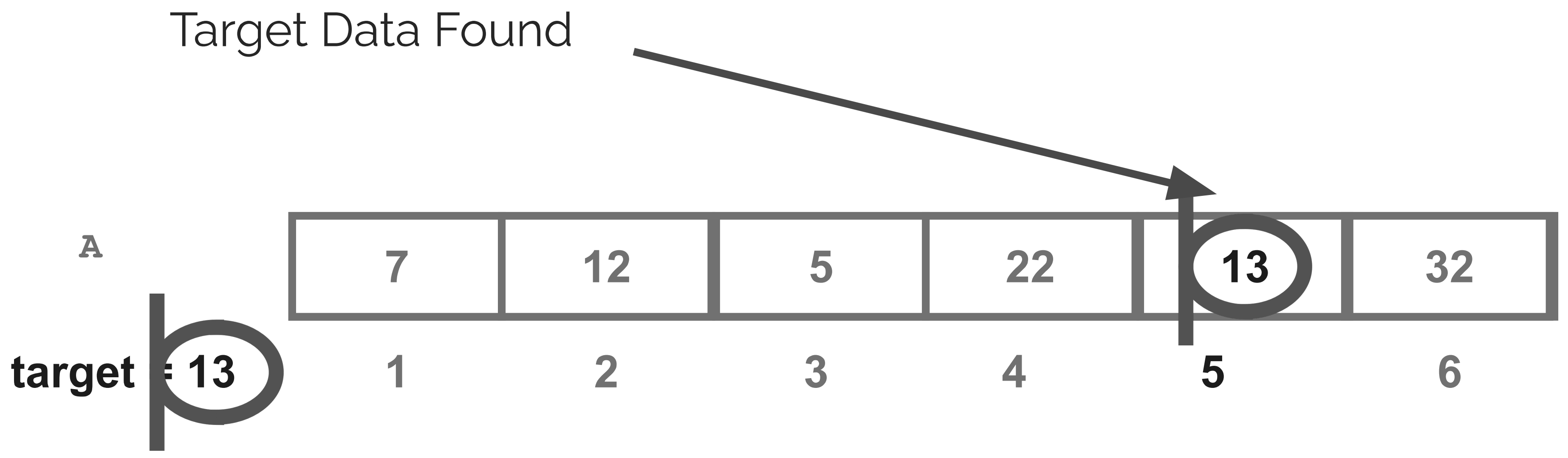
https://www.cc.gatech.edu/home/bleahy/cs1311/cs1311lecture15wdl.



https://www.cc.gatech.edu/home/bleahy/cs1311/cs1311lecture15wdl.

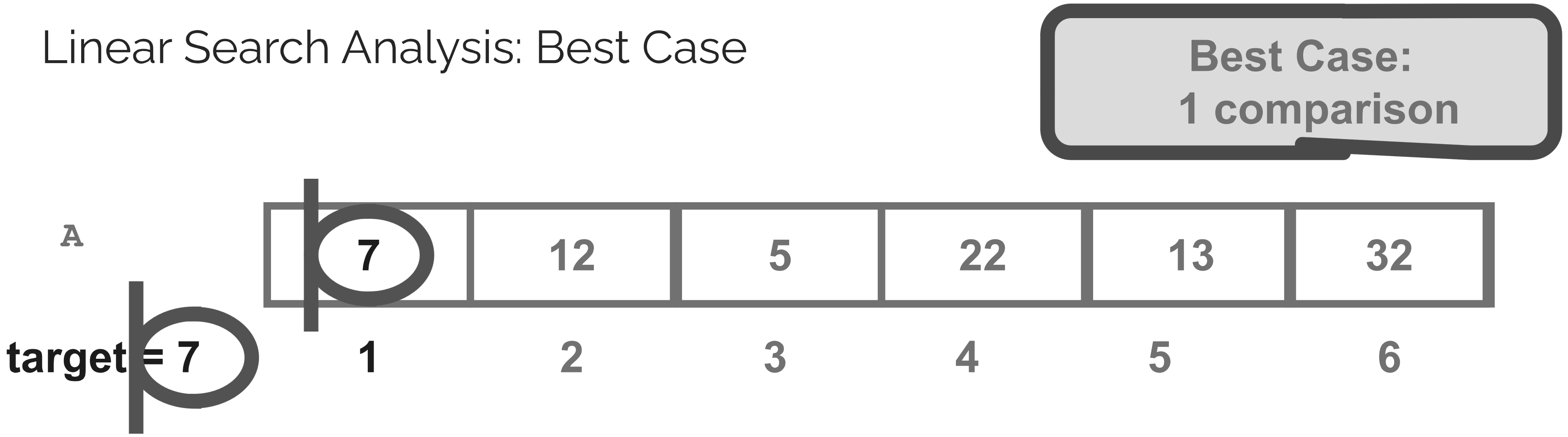


https://www.cc.gatech.edu/home/bleahy/cs1311/cs1311lecture15wdl.



https://www.cc.gatech.edu/home/bleahy/cs1311/cs1311lecture15wdl.

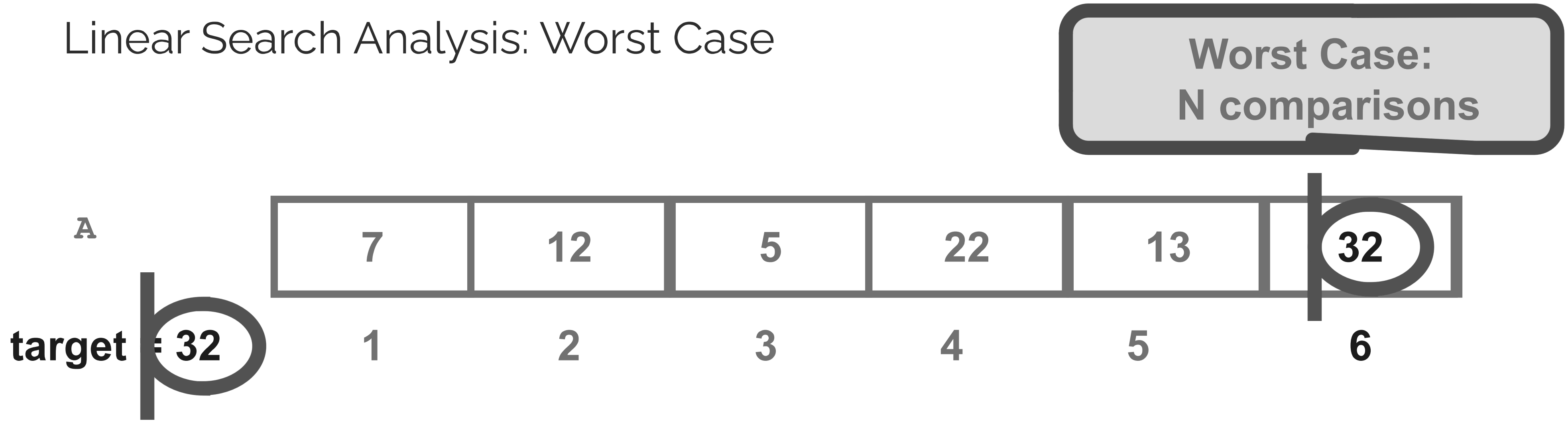
Linear Search Analysis: Best Case



Best Case: **match with the first item**

https://www.cc.gatech.edu/home/bleahy/cs1311/cs1311lecture15wdl.

Linear Search Analysis: Worst Case



Worst Case: match with the last item

https://www.cc.gatech.edu/home/bleahy/cs1311/cs1311lecture15wdl.

Linear Search Analysis: Unsuccessful Search Case

Unsuccessful Search: N comparisons

`A`

	7	12	5	22	13	32
target = 53	1	2	3	4	5	6

Unsuccessful Search: no match

https://www.cc.gatech.edu/home/bleahy/cs1311/cs1311lecture15wdl.

Analysis of Linear Search : Counting the number of primitive steps

Algorithm 1: Linear search

```
input  : Array: $A[1 \dots n], Key$
output: index of first occurrence of key if it is found, 0 otherwise
for $i \leftarrow 1$ to $n$ do          ← T1
    if $A[i] = key$ then                ← T2
        return $i$
    end
end
return 0
```

1. What is the relationship between T1 and T2?
   Ans: T1 = T2 for a successful search, T1 = T2 + 1 = n + 1 for an unsuccessful search
2. In the worst case successful search: T1 is ? T2 is?
   Ans: T1 = T2 =n
3. In the best case successful search: T1 is ? T2 is?
   Ans: T1 = T2 =1
4. In case of an unsuccessful search: T1 is ? T2 is?
   Ans: T1 = T2 + 1 = n + 1

Analysis of Linear Search : Counting the number of primitive steps

Total Number of Primitive Steps: (counting “assignment and comparison” done in statement 1 as one primitive step)

- T1+ T2 + 1(for the “return” statement)

this is =

- n + n + 1 = 2n + 1 in the worst case successful search
- 1 + 1 + 1 = 3 in the best case successful search
- (n + 1) + n + 1 = 2n + 2 in an unsuccessful search

Ignoring the constants

When we compare running times of two algorithms, constants become insignificant for large n.

For example

$\lim_{n\to\infty} \frac{(c1 \log n + c2)}{c3\, n + c4} = 0$, irrespective of the constants c1, c2, c3 and c4.

We will define the notion of asymptotic functions to capture this.

Programming Assignment:

- Implement Linear Search
  - Count the number of key comparisons for various inputs and plot the graph.
  - For every input size n , run it with n+ 1 different keys - n successful plus 1 unsuccessful
  - Compute the minimum, maximum and average of the number of key comparisons for each input size.
  - Plot the graph for each case - best, worst and average number of comparisons.
  - n varies from 10 to 100 in steps of 5

MCAC 301: Design and Analysis of Algorithms

Neelima Gupta

ngupta@cs.du.ac.in

April 27, 2023

Can we search faster?

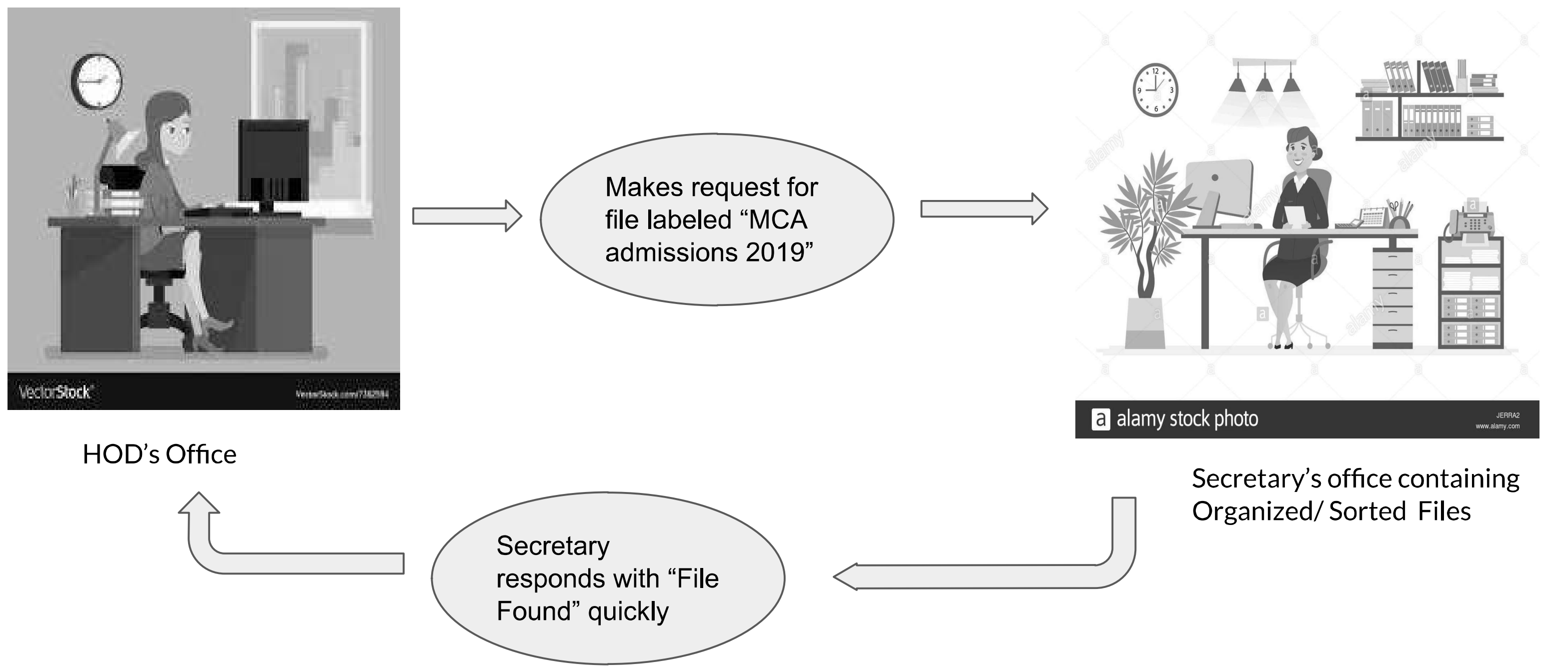

VectorStock
HOD's Office
Makes request for file labeled "MCA admissions 2019"
alamy stock photo
Secretary's office containing Organized/ Sorted Files
Secretary responds with "File Found" quickly

Figure A: Secretary at an office responding to the requests of the Head

1

Searching for a word in a dictionary

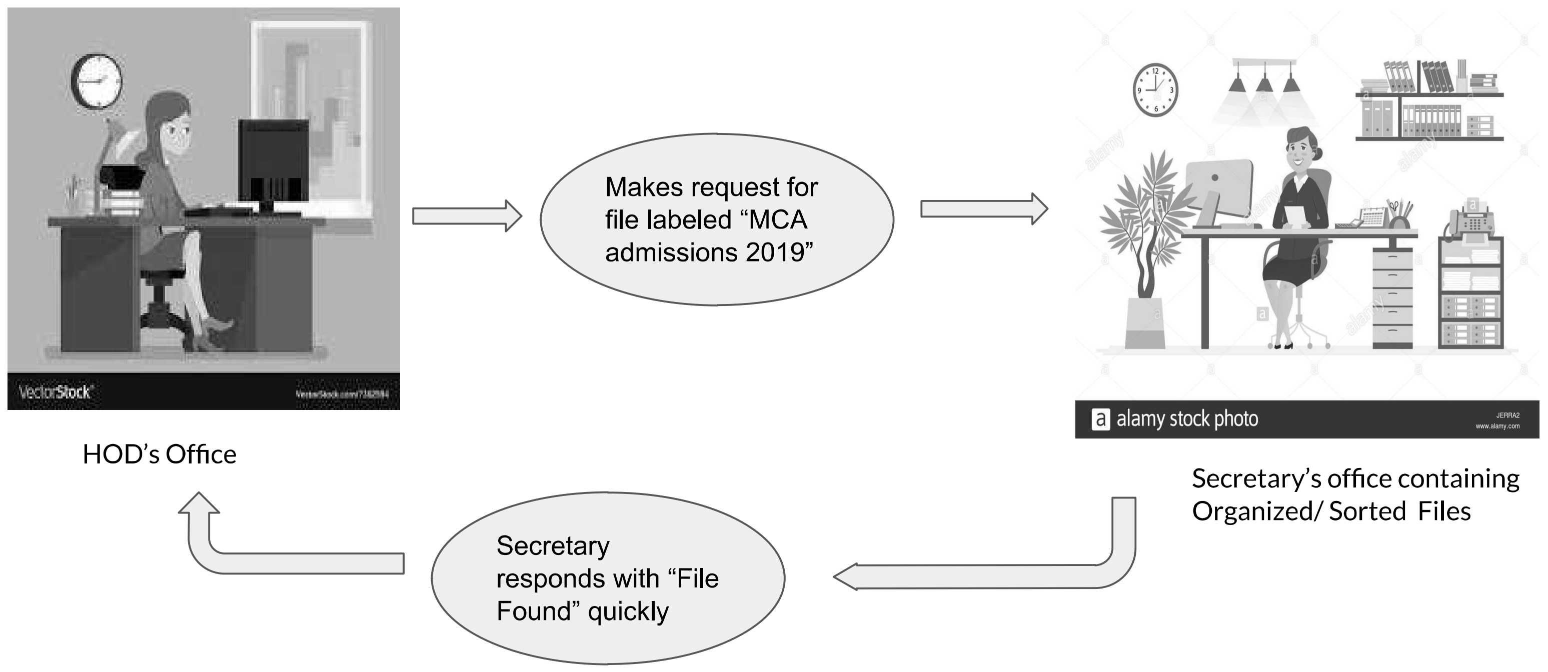
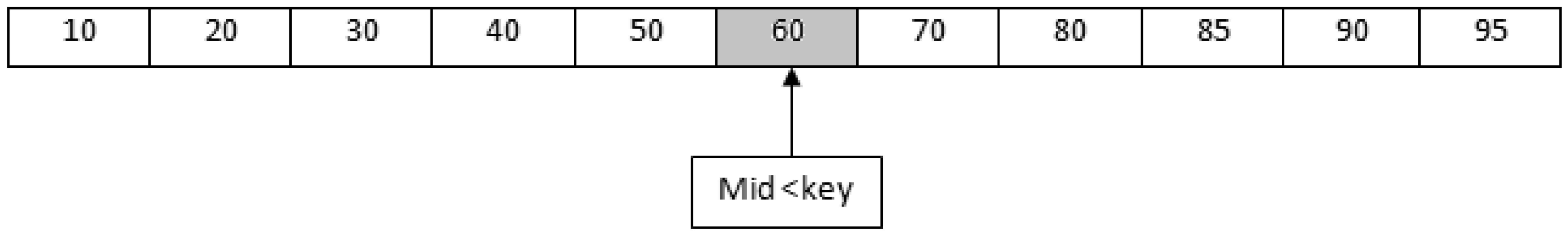

VectorStock
VectorStock.com/7362594
HOD's Office
Makes request for file labeled "MCA admissions 2019"
alamy stock photo
JERRA2
www.alamy.com
Secretary's office containing Organized/ Sorted Files
Secretary responds with "File Found" quickly

Figure A: Secretary at an office responding to the requests of the Head

1

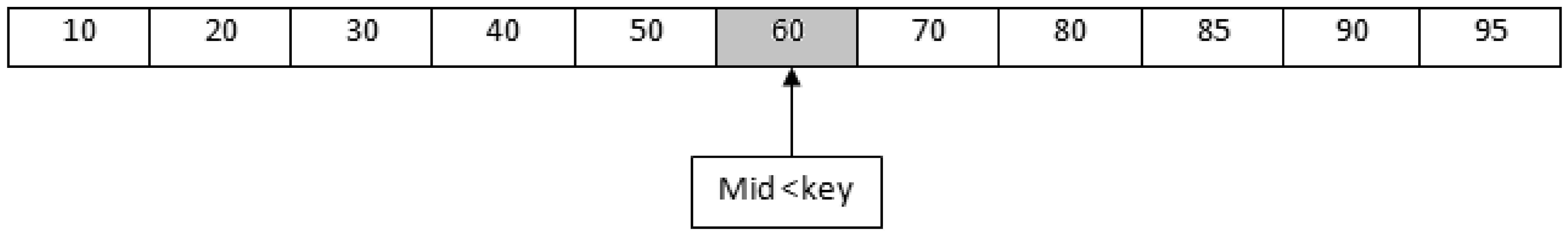
Binary Search

Key: 65



Binary Search

Key: 65



Divide and Conquer Paradigm

1. Divide : Divide the problem into a number of subproblems of smaller Size.
2. Conquer : Solve each sub-problem.
3. Combine : Combine the solution of the subproblems to obtain the solution of the original (bigger) problem.

A subproblem is solved recursively (i.e. by applying the same algorithm on the smaller problem) so long as it's size $>$ some threshold (called the "terminating condition"). Thereafter, it is solved directly.

Binary Search: Split in the middle

```
input  : Array: $A[1 \dots n], Key$
output: Index of key if key found, -1 otherwise
Binary-Search-Recursive(A, first, last, key)
/* "first" and "last" are the first and the last indices of the
  array
if first ≤ last then
    mid = (first + last)/2
    if A[mid]=key then
        return mid
    end
    if A[mid] < key then
        Binary-Search-Recursive(A, mid + 1, last, key)
    end
    else
        Binary-Search-Recursive(A, first, mid -1, key)
    end
end
```

Frame Title

Quick Demo

Time Complexity

Let $W(n)$ be the number of comparisons performed by the binary search algorithm in worst case. Then,
$W(n) = ?$

Time Complexity

Let $W(n)$ be the number of comparisons performed by the binary search algorithm in worst case. Then,
$W(n) =?$
$W(n) = W(n/2)+?$

Time Complexity

Let $W(n)$ be the number of comparisons performed by the binary search algorithm in worst case. Then,
$W(n) =?$
$W(n) = W(n/2)+?$
$W(n) = W(n/2) + 2$

Time Complexity

Let $W(n)$ be the number of comparisons performed by the binary search algorithm in worst case. Then,
$W(n) =?$
$W(n) = W(n/2)+?$
$W(n) = W(n/2) + 2$
$W(n) =?$

Time Complexity

Let $W(n)$ be the number of comparisons performed by the binary search algorithm in worst case. Then,
$W(n) =?$
$W(n) = W(n/2)+?$
$W(n) = W(n/2) + 2$
$W(n) =?$ $W(n) = \log n$
Let $B(n)$ be the number of comparisons performed by the binary search algorithm in best case. Then,

Time Complexity

Let $W(n)$ be the number of comparisons performed by the binary search algorithm in worst case. Then,

$W(n) = ?$

$W(n) = W(n/2) + ?$

$W(n) = W(n/2) + 2$

$W(n) = ?$ $W(n) = \log n$

Let $B(n)$ be the number of comparisons performed by the binary search algorithm in best case. Then,

$B(n) = ?$

Time Complexity

Let $W(n)$ be the number of comparisons performed by the binary search algorithm in worst case. Then,
$W(n) =?$
$W(n) = W(n/2)+?$
$W(n) = W(n/2) + 2$
$W(n) =?$ $W(n) = \log n$
Let $B(n)$ be the number of comparisons performed by the binary search algorithm in best case. Then,
$B(n) =?$
$B(n) = 1$

MCSC 101: Design and Analysis of Algorithms

Neelima Gupta

ngupta@cs.du.ac.in

January 4, 2021

Binary Search

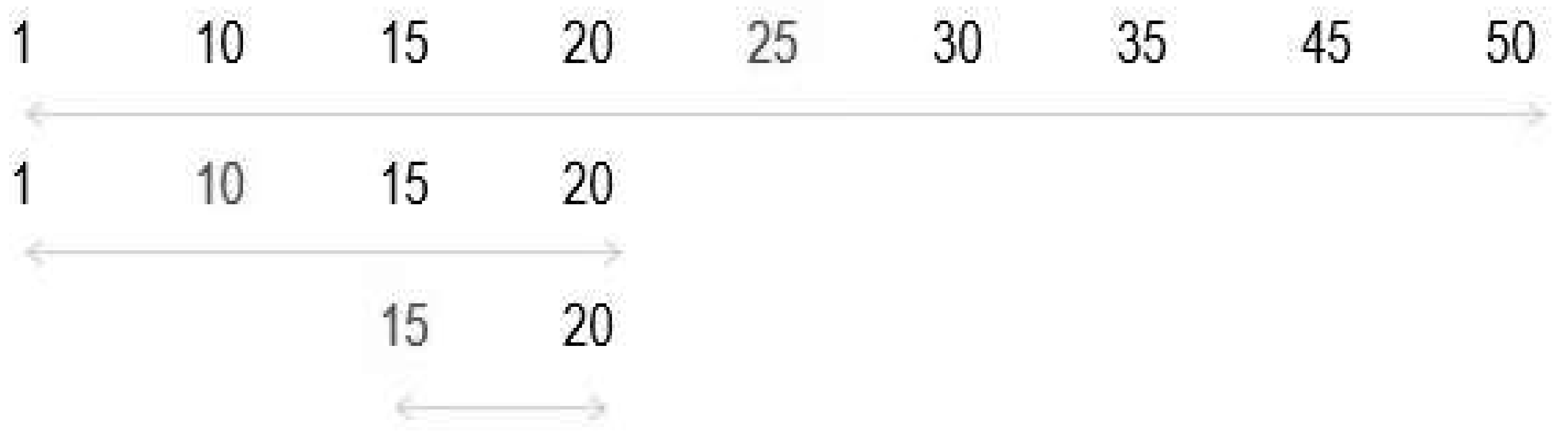
```
Binary-Search($A[]$, $n$, $key$)
left $\leftarrow$ 1, right $\leftarrow$ n
while $left \leq right$ do
    mid $\leftarrow$ (left+right)/2
    if ($A[mid]=key$) then
        return mid
    end
    if ($A[mid] < key$) then
        left $\leftarrow$ mid+1
    end
    else
        right $\leftarrow$ mid-1
    end
end
return 0
```

Defining the I/O of Binary Search

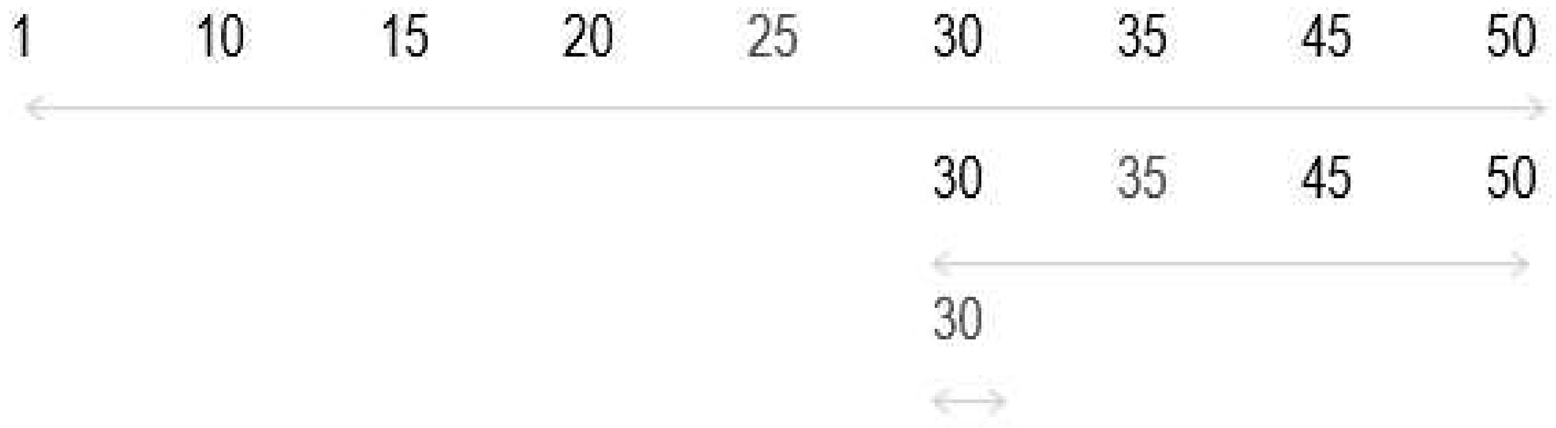
- Input : Given an array L containing n items (n$\geq$ 0) ordered such that A[1] $\leq$ A[2] $\leq$ A[3] $\leq \ldots \leq$ A[n] and given x
- Output : The binary search algorithm terminates with index_mid = an occurrence of x in A, if found and, index_mid= 0 otherwise.

Binary search can only be applied if the array to be searched is already sorted

Search for the no. 15



Search for the no. 30



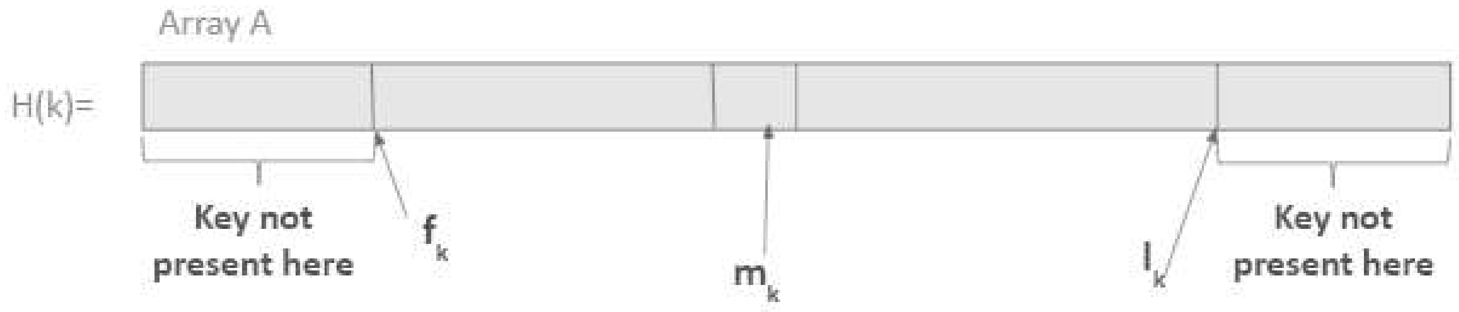
Proof of correctness: The Loop Invariant

Let r be the maximum number of times the loop starting from line 3 runs. For $1 \leq k \leq r$, let f_k and l_k be the values of first and last respectively when the control reaches the while statement for the k^{th} time.

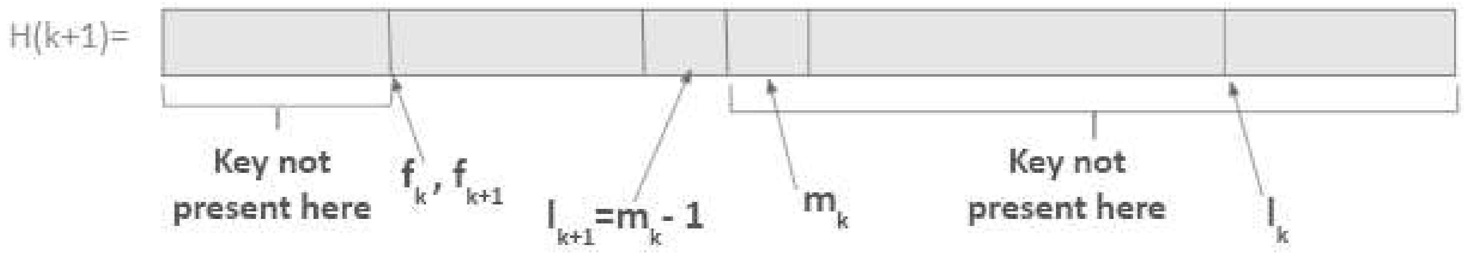
- Loop Invariant: For $1 \leq k \leq r$, $H(k)$: when the control reaches the while statement for the k^{th} time $A[i] \neq x$ for every $i < f_k$ and for every $i > l_k$.
- We will prove the loop invariant by induction.
- Base Case: $k = 1$. $f_k = 1$ and $l_k = n$. Hence the claim is vacuously true.

Inductive Step: Suppose $H(k)$ is true. We will prove that $H(k+1)$ is true. In the k^{th} iteration, we have three possibilities:

- Case 1: $x = A[m_k]$ ....not possible else we wouldn't enter the loop $(k+1)^{th}$ time.

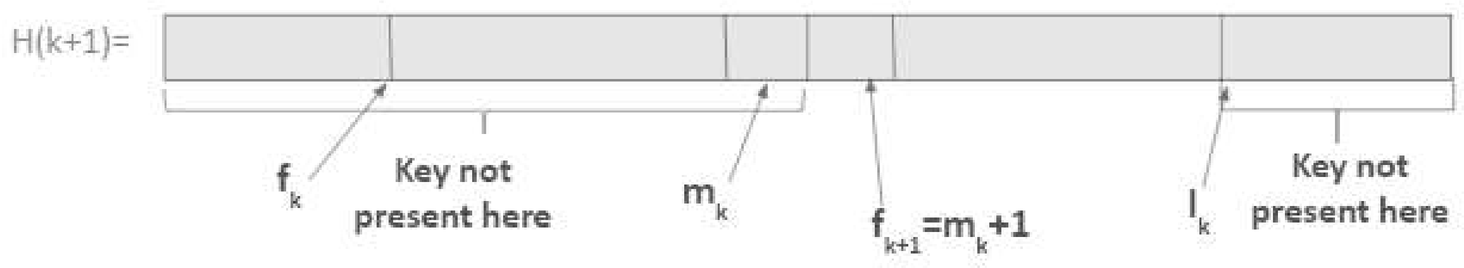


- Case 2. $x < A[m_k]$.



Since the array is sorted, we have
$x < A[m_k] \leq A[i] \; \forall i = m_k + 1 \dots l_k$.
i.e. $x \neq A[i] \; \forall i = m_k \dots l_k$.

- Case 3. $x > A[m_k]$.



Since the array is sorted, we have
$x > A[m_k] \geq A[i]\ \forall i = f_k \ldots m_k - 1$.
i.e. $x \neq A[i]\ \forall i = f_k \ldots m_k$.

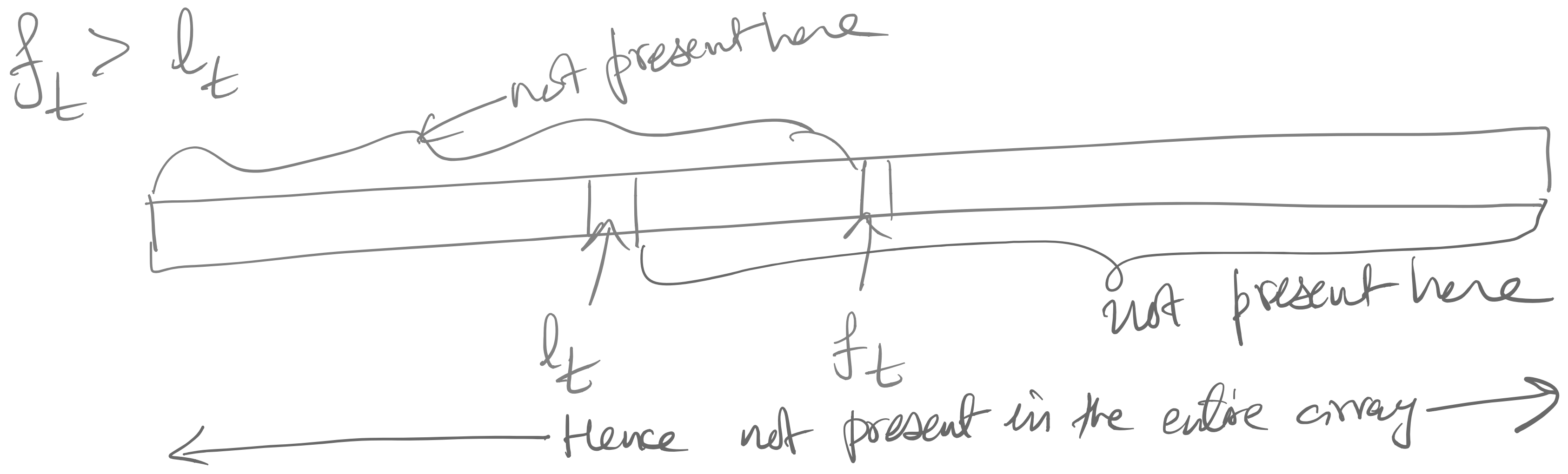
Correctness of the algorithm assuming the loop invariant

Suppose the test condition is executed t times.

- Case 1: $f_t \leq l_t$. We exit from the loop because $A[m_t] = x$. Thus, index_mid is a position of occurrence of x as index_mid $= m_t$
- Case 2: If $f_t > l_t$, we exit the while statement and return 0. We will next argue that the element is not present in the array in this case:
  By loop invariant hypothesis, $A[i] \neq x$ for $i < f_t$ and for $i > l_t$. the claim follows as $f_t > l_t$.

Hence assuming that the statement H(k) is correct the algorithm works correctly.

Correctness of the algorithm assuming the loop invariant



By loop invariant hypothesis, $A[i] \neq x$ for $i < f_t$ and for $i > l_t$. the claim follows as $f_t > l_t$.

Time Complexity

Let $W(n)$ be the number of comparisons performed by the binary search algorithm in worst case. Then,

Time Complexity

Let $W(n)$ be the number of comparisons performed by the binary search algorithm in worst case. Then,
$W(n) = ?$

Time Complexity

Let $W(n)$ be the number of comparisons performed by the binary search algorithm in worst case. Then,
$W(n) =?$
$W(n) = W(n/2)+?$

Time Complexity

Let $W(n)$ be the number of comparisons performed by the binary search algorithm in worst case. Then,
$W(n) =?$
$W(n) = W(n/2)+?$
$W(n) = W(n/2) + 2$

Time Complexity

Let $W(n)$ be the number of comparisons performed by the binary search algorithm in worst case. Then,

$W(n) =?$

$W(n) = W(n/2) + ?$

$W(n) = W(n/2) + 2$

$W(n) =?$

Time Complexity

Let $W(n)$ be the number of comparisons performed by the binary search algorithm in worst case. Then,
$W(n) =?$
$W(n) = W(n/2)+?$
$W(n) = W(n/2) + 2$
$W(n) =?$ $W(n) = \log n$
Let $B(n)$ be the number of comparisons performed by the binary search algorithm in best case. Then,

Time Complexity

Let $W(n)$ be the number of comparisons performed by the binary search algorithm in worst case. Then,
$W(n) =?$
$W(n) = W(n/2)+?$
$W(n) = W(n/2) + 2$
$W(n) =?$ $W(n) = \log n$
Let $B(n)$ be the number of comparisons performed by the binary search algorithm in best case. Then,
$B(n) =?$

Time Complexity

Let $W(n)$ be the number of comparisons performed by the binary search algorithm in worst case. Then,

$W(n) =?$

$W(n) = W(n/2)+?$

$W(n) = W(n/2) + 2$

$W(n) =?$ $W(n) = \log n$

Let $B(n)$ be the number of comparisons performed by the binary search algorithm in best case. Then,

$B(n) =?$

$B(n) = 1$

Sorting

9

The efficiency of data handling can often be substantially increased if the data are sorted according to some criteria of order. For example, it would be practically impossible to find a name in the telephone directory if the names were not alphabetically ordered. The same can be said about dictionaries, book indexes, payrolls, bank accounts, student lists, and other alphabetically organized materials. The convenience of using sorted data is unquestionable and must be addressed in computer science as well. Although a computer can grapple with an unordered telephone book more easily and quickly than a human can, it is extremely inefficient to have the computer process such an unordered data set. It is often necessary to sort data before processing.

The first step is to choose the criteria that will be used to order data. This choice will vary from application to application and must be defined by the user. Very often, the sorting criteria are natural, as in the case of numbers. A set of numbers can be sorted in ascending or descending order. The set of five positive integers (5, 8, 1, 2, 20) can be sorted in ascending order resulting in the set (1, 2, 5, 8, 20) or in descending order resulting in the set (20, 8, 5, 2, 1). Names in the phone book are ordered alphabetically by last name, which is the natural order. For alphabetic and nonalphabetic characters, the American Standard Code for Information Interchange (ASCII) code is commonly used, although other choices such as Extended Binary Coded Decimal Interchange Code (EBCDIC) are possible. Once a criterion is selected, the second step is how to put a set of data in order using that criterion.

The final ordering of data can be obtained in a variety of ways, and only some of them can be considered meaningful and efficient. To decide which method is best, certain criteria of efficiency have to be established and a method for quantitatively comparing different algorithms must be chosen.

To make the comparison machine-independent, certain critical properties of sorting algorithms should be defined when comparing alternative methods. Two such properties are the number of comparisons and the number of data movements. The choice of these two properties should not be surprising. To sort a set of data, the data have to be compared and moved as necessary; the efficiency of these two operations depends on the size of the data set.

Because determining the precise number of comparisons is not always necessary or possible, an approximate value can be computed. For this reason, the number of comparisons and movements is approximated with big-O notation by giving the order of magnitude of these numbers. But the order of magnitude can vary depending on the initial ordering of data. How much time, for example, does the machine spend on data ordering if the data are already ordered? Does it recognize this initial ordering immediately or is it completely unaware of that fact? Hence, the efficiency measure also indicates the "intelligence" of the algorithm. For this reason, the number of comparisons and movements is computed (if possible) for the following three cases: best case (often, data already in order), worst case (it can be data in reverse order), and average case (data in random order). Some sorting methods perform the same operations regardless of the initial ordering of data. It is easy to measure the performance of such algorithms, but the performance itself is usually not very good. Many other methods are more flexible, and their performance measures for all three cases differ.

The number of comparisons and the number of movements do not have to coincide. An algorithm can be very efficient on the former and perform poorly on the latter, or vice versa. Therefore, practical reasons must aid in the choice of which algorithm to use. For example, if only simple keys are compared, such as integers or characters, then the comparisons are relatively fast and inexpensive. If strings or arrays of numbers are compared, then the cost of comparisons goes up substantially, and the weight of the comparison measure becomes more important. If, on the other hand, the data items moved are large, such as structures, then the movement measure may stand out as the determining factor in efficiency considerations. All theoretically established measures have to be used with discretion, and theoretical considerations should be balanced with practical applications. After all, the practical applications serve as a rubber stamp for theory decisions.

Sorting algorithms are of different levels of complexity. A simple method can be only 20 percent less efficient than a more elaborate one. If sorting is used in the program once in a while and only for small sets of data, then using a sophisticated and slightly more efficient algorithm may not be desirable; the same operation can be performed using a simpler method and simpler code. But if thousands of items are to be sorted, then a gain of 20 percent must not be neglected. Simple algorithms often perform better with a small amount of data than their more complex counterparts whose effectiveness may become obvious only when data samples become very large.

9.1 Elementary Sorting Algorithms

9.1.1 Insertion Sort

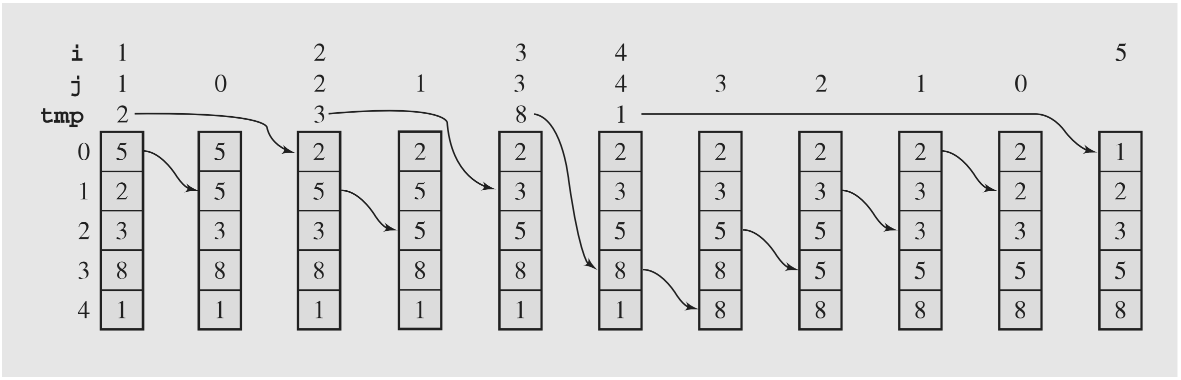
An *insertion sort* starts by considering the two first elements of the array `data`, which are `data[0]` and `data[1]`. If they are out of order, an interchange takes place. Then, the third element, `data[2]`, is considered. If `data[2]` is less than `data[0]` and `data[1]`, these two elements are shifted by one position; `data[0]` is placed at position 1, `data[1]` at position 2, and `data[2]` at position 0. If `data[2]` is less than `data[1]` and not less than `data[0]`, then only `data[1]` is moved to position 2 and its place is taken by `data[2]`. If, finally, `data[2]` is not less than both its predecessors, it stays in its current position. Each element `data[i]` is inserted into its proper location *j* such that $0 \leq j \leq i$, and all elements greater than `data[i]` are moved by one position.

An outline of the insertion sort algorithm is as follows:

```
insertionsort(data[],n)
   for i = 1 to n-1
      move all elements data[j] greater than data[i] by one position;
      place data[i] in its proper position;
```

Note that sorting is restricted only to a fraction of the array in each iteration, and only in the last pass is the whole array considered. Figure 9.1 shows what changes are made to the array [5 2 3 8 1] when `insertionsort()` executes.

FIGURE 9.1 The array [5 2 3 8 1] sorted by insertion sort.



Because an array having only one element is already ordered, the algorithm starts sorting from the second position, position 1. Then for each element `tmp = data[i]`, all elements greater than `tmp` are copied to the next position, and `tmp` is put in its proper place.

An implementation of insertion sort is:

```
template<class T>
void insertionsort(T data[], int n) {
    for (int i = 1,j; i < n; i++) {
        T tmp = data[i];
        for (j = i; j > 0 && tmp < data[j-1]; j--)
            data[j] = data[j-1];
        data[j] = tmp;
    }
}
```

An advantage of using insertion sort is that it sorts the array only when it is really necessary. If the array is already in order, no substantial moves are performed; only the variable `tmp` is initialized, and the value stored in it is moved back to the same position. The algorithm recognizes that part of the array is already sorted and stops execution accordingly. But it recognizes only this, and the fact that elements may already be in their proper positions is overlooked. Therefore, they can be moved from these positions and then later moved back. This happens to numbers 2 and 3 in the example in Figure 9.1. Another disadvantage is that if an item is being inserted, all elements greater than the one being inserted have to be moved. Insertion is not localized and may require moving a significant number of elements. Considering that an element can be moved from its final position only to be placed there again later, the number of redundant moves can slow down execution substantially.

To find the number of movements and comparisons performed by `insertionsort()`, observe first that the outer `for` loop always performs $n - 1$ iterations. However, the number of elements greater than `data[i]` to be moved by one position is not always the same.

The best case is when the data are already in order. Only one comparison is made for each position i, so there are $n - 1$ comparisons, which is $O(n)$, and $2(n - 1)$ moves, all of them redundant.

The worst case is when the data are in reverse order. In this case, for each i, the item `data[i]` is less than every item `data[0]`, . . . , `data[i-1]`, and each of them is moved by one position. For each iteration i of the outer `for` loop, there are i comparisons, and the total number of comparisons for all iterations of this loop is

$$\sum_{i=1}^{n-1} i = 1 + 2 + \cdots + (n - 1) = \frac{n(n - 1)}{2} = O(n^2)$$

The number of times the assignment in the inner `for` loop is executed can be computed using the same formula. The number of times `tmp` is loaded and unloaded in the outer `for` loop is added to that, resulting in the total number of moves:

$$\frac{n(n - 1)}{2} + 2(n - 1) = \frac{n^2 + 3n - 4}{2} = O(n^2)$$

Only extreme cases have been taken into consideration. What happens if the data are in random order? Is the sorting time closer to the time of the best case, $O(n)$, or to the worst case, $O(n^2)$? Or is it somewhere in between? The answer is not immediately evident, and requires certain introductory computations. The outer `for` loop always executes $n - 1$ times, but it is also necessary to determine the number of iterations for the inner loop.

For every iteration i of the outer `for` loop, the number of comparisons depends on how far away the item `data[i]` is from its proper position in the currently sorted subarray `data[0...i-1]`. If it is already in this position, only one test is performed that compares `data[i]` and `data[i-1]`. If it is one position away from its proper place, two comparisons are performed: `data[i]` is compared with `data[i-1]` and then with `data[i-2]`. Generally, if it is j positions away from its proper location, `data[i]` is compared with $j + 1$ other elements. This means that, in iteration i of the outer `for` loop, there are either 1, 2, . . . , or i comparisons.

Under the assumption of equal probability of occupying array cells, the average number of comparisons of `data[i]` with other elements during the iteration *i* of the outer `for` loop can be computed by adding all the possible numbers of times such tests are performed and dividing the sum by the number of such possibilities. The result is

$$\frac{1+2+\cdots+i}{i}=\frac{\frac{1}{2}i(i+1)}{i}=\frac{i+1}{2}$$

To obtain the average number of all comparisons, the computed figure has to be added for all *i*s (for all iterations of the outer `for` loop) from 1 to $n-1$. The result is

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

which is $O(n^2)$ and approximately one-half of the number of comparisons in the worst case.

By similar reasoning, we can establish that, in iteration *i* of the outer `for` loop, `data[i]` can be moved either 0, 1, . . . , or *i* times; that is

$$\frac{0+1+\cdots+i}{i}=\frac{\frac{1}{2}i(i+1)}{i+1}=\frac{i}{2}$$

times plus two unconditional movements (to `tmp` and from `tmp`). Hence, in all the iterations of the outer `for` loop we have, on the average,

$$\sum_{i=1}^{n-1}\left(\frac{i}{2}+2\right)=\frac{1}{2}\sum_{i=1}^{n-1}i+\sum_{i=1}^{n-1}2=\frac{\frac{1}{2}n(n-1)}{2}+2(n-1)=\frac{n^2+7n-8}{4}$$

movements, which is also $O(n^2)$.

This answers the question: is the number of movements and comparisons for a randomly ordered array closer to the best or to the worst case? Unfortunately, it is closer to the latter, which means that, on the average, when the size of an array is doubled, the sorting effort quadruples.

9.1.2 Selection Sort

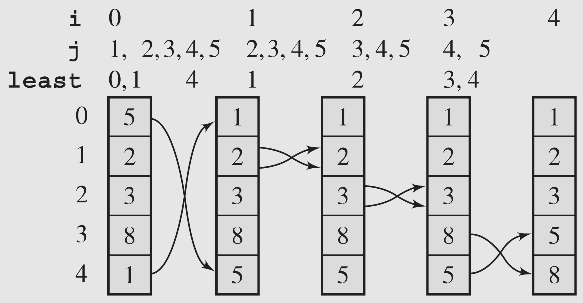
Selection sort is an attempt to localize the exchanges of array elements by finding a misplaced element first and putting it in its final place. The element with the lowest value is selected and exchanged with the element in the first position. Then, the smallest value among the remaining elements `data[1]`, . . . , `data[n-1]` is found and put in the second position. This selection and placement by finding, in each pass `i`, the lowest value among the elements `data[i]`, . . . , `data[n-1]` and swapping it with `data[i]` are continued until all elements are in their proper positions. The following pseudocode reflects the simplicity of the algorithm:

```
selectionsort(data[],n)
    for i = 0 to n-2
        select the smallest element among data[i],...,data[n-1];
        swap it with data[i];
```

It is rather obvious that `n-2` should be the last value for `i`, because if all elements but the last have been already considered and placed in their proper positions, then the *n*th element (occupying position `n-1`) has to be the largest. An example is shown in Figure 9.2. Here is a C++ implementation of selection sort:

```
template<class T>
void selectionsort(T data[], int n) {
    for (int i = 0,j,least; i < n-1; i++) {
        for (j = i+1, least = i; j < n; j++)
            if (data[j] < data[least])
                least = j;
        swap(data[least],data[i]);
    }
}
```

FIGURE 9.2 The array [5 2 3 8 1] sorted by selection sort.



where the function `swap()` exchanges elements `data[least]` and `data[i]` (see the end of Section 1.2). Note that `least` is not the smallest element but its position.

The analysis of the performance of the function `selectionsort()` is simplified by the presence of two `for` loops with lower and upper bounds. The outer loop executes `n` − 1 times, and for each `i` between 0 and `n` − 2, the inner loop iterates `j` = (`n` − 1) − `i` times. Because comparisons of keys are done in the inner loop, there are

$$\sum_{i=0}^{n-2}(n-1-i) = (n-1) + \cdots + 1 = \frac{n(n-1)}{2} = O(n^2)$$

comparisons. This number stays the same for all cases. There can be some savings only in the number of swaps. Note that if the assignment in the `if` statement is executed, only the index `j` is moved, not the item located currently at position `j`. Array elements are swapped unconditionally in the outer loop as many times as this loop executes, which is `n-1`. Thus, in all cases, items are moved the same number of times, 3(`n`−1).

The best thing about this sort is the required number of assignments, which can hardly be beaten by any other algorithm. However, it might seem somewhat unsatisfactory that the total number of exchanges, 3(n−1), is the same for all cases. Obviously, no exchange is needed if an item is in its final position. The algorithm disregards that and swaps such an item with itself, making three redundant moves. The problem can be alleviated by making `swap()` a conditional operation. The condition preceding the `swap()` should indicate that no item less than `data[least]` has been found among elements `data[i+1]`, . . . , `data[n−1]`. The last line of `selectionsort()` might be replaced by the lines:

```
if (data[i] != data[least])
   swap (data[least], data[i]);
```

This increases the number of array element comparisons by n−1, but this increase can be avoided by noting that there is no need to compare items. We proceed as we did in the case of the `if` statement of `selectionsort()` by comparing the indexes and not the items. The last line of `selectionsort()` can be replaced by:

```
if (i != least)
   swap (data[least], data[i]);
```

Is such an improvement worth the price of introducing a new condition in the procedure and adding $n-1$ index comparisons as a consequence? It depends on what types of elements are being sorted. If the elements are numbers or characters, then interposing a new condition to avoid execution of redundant swaps gains little in efficiency. But if the elements in `data` are large compound entities such as arrays or structures, then one swap (which requires three assignments) may take the same amount of time as, say, 100 index comparisons, and using a conditional `swap()` is recommended.

9.1.3 Bubble Sort

A bubble sort can be best understood if the array to be sorted is envisaged as a vertical column whose smallest elements are at the top and whose largest elements are at the bottom. The array is scanned from the bottom up, and two adjacent elements are interchanged if they are found to be out of order with respect to each other. First, items `data[n-1]` and `data[n-2]` are compared and swapped if they are out of order. Next, `data[n-2]` and `data[n-3]` are compared, and their order is changed if necessary, and so on up to `data[1]` and `data[0]`. In this way, the smallest element is bubbled up to the top of the array.

However, this is only the first pass through the array. The array is scanned again comparing consecutive items and interchanging them when needed, but this time, the last comparison is done for `data[2]` and `data[1]` because the smallest element is already in its proper position, namely, position 0. The second pass bubbles the second smallest element of the array up to the second position, position 1. The procedure continues until the last pass when only one comparison, `data[n-1]` with `data[n-2]`, and possibly one interchange are performed.

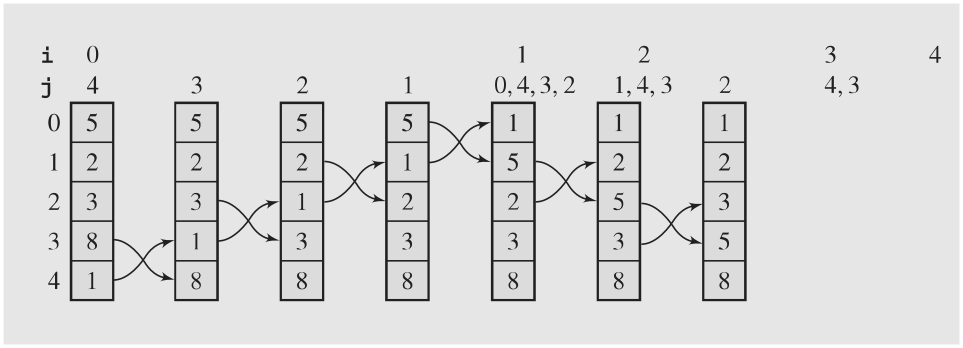
A pseudocode of the algorithm is as follows:

```
bubblesort(data[],n)
   for i = 0 to n−2
       for j = n-1 down to i+1
           swap elements in positions j and j-1 if they are out of order;
```

Figure 9.3 illustrates the changes performed in the integer array [5 2 3 8 1] during the execution of `bubblesort()`. Here is an implementation of bubble sort:

```
template<class T>
void bubblesort(T data[], int n) {
    for (int i = 0; i < n-1; i++)
        for (int j = n-1; j > i; --j)
            if (data[j] < data[j-1])
                swap(data[j],data[j-1]);
}
```

FIGURE 9.3 The array [5 2 3 8 1] sorted by bubble sort.



The number of comparisons is the same in each case (best, average, and worst) and equals the total number of iterations of the inner `for` loop

$$\sum_{i=0}^{n-2}(n-1-i)=\frac{n(n-1)}{2}=O(n^2)$$

comparisons. This formula also computes the number of swaps in the worst case when the array is in reverse order. In this case, $3\,\frac{n(n-1)}{2}$ moves have to be made.

The best case, when all elements are already ordered, requires no swaps. To find the number of moves in the average case, note that if an i-cell array is in random order, then the number of swaps can be any number between zero and $i-1$; that is, there can be either no swap at all (all items are in ascending order), one swap, two swaps, . . . , or $i-1$ swaps. The array processed by the inner `for` loop is `data[i]`, . . . , `data[n-1]`, and the number of swaps in this subarray—if its elements are randomly ordered—is either zero, one, two, . . . , or $n-1-i$. After averaging the sum of all these possible numbers of swaps by the number of these possibilities, the average number of swaps is obtained, which is

$$\frac{0+1+2+\cdots+(n-1-i)}{n-i}=\frac{n-i-1}{2}$$

If all these averages for all the subarrays processed by `bubblesort()` are added (that is, if such figures are summed over all iterations i of the outer `for` loop), the result is

$$\sum_{i=0}^{n-2}\frac{n-i-1}{2} = \frac{1}{2}\sum_{i=0}^{n-2}(n-1) - \frac{1}{2}\sum_{i=0}^{n-2}i$$

$$= \frac{(n-1)^2}{2} - \frac{(n-1)(n-2)}{4} = \frac{n(n-1)}{4}$$

swaps, which is equal to $\frac{3}{4}n(n-1)$ moves.

The main disadvantage of bubble sort is that it painstakingly bubbles items step by step up toward the top of the array. It looks at two adjacent array elements at a time and swaps them if they are not in order. If an element has to be moved from the bottom to the top, it is exchanged with every element in the array. It does not skip them as selection sort did. In addition, the algorithm concentrates only on the item that is being bubbled up. Therefore, all elements that distort the order are moved, even those that are already in their final positions (see numbers 2 and 3 in Figure 9.3, the situation analogous to that in insertion sort).

What is bubble sort's performance in comparison to insertion and selection sort? In the average case, bubble sort makes approximately twice as many comparisons and the same number of moves as insertion sort, as many comparisons as selection sort, and n times more moves than selection sort.

It could be said that insertion sort is twice as fast as bubble sort. In fact it is, but this fact does not immediately follow from the performance estimates. The point is that when determining a formula for the number of comparisons, only comparisons of data items have been included. The actual implementation for each algorithm involves more than just that. In `bubblesort()`, for example, there are two loops, both of which compare indexes: `i` and `n−1` in the first loop, `j` and `i` in the second. All in all, there are $\frac{n(n-1)}{2}$ such comparisons, and this number should not be treated too lightly. It becomes negligible if the data items are large structures. But if `data` consists of integers, then comparing the data takes a similar amount of time as comparing indexes. A more thorough treatment of the problem of efficiency should focus on more than just data comparison and exchange. It should also include the overhead necessary for implementation of the algorithm.

An apparent improvement of bubble sort is obtained by adding a flag to discontinue processing after a pass in which no swap was performed:

```
template<class T>
void bubblesort2(T data[], const int n) {
    bool again = true;
    for (int i = 0; i < n-1 && again; i++)
        for (int j = n-1, again = false; j > i; --j)
            if (data[j] < data[j-1]) {
                swap(data[j],data[j-1]);
                again = true;
            }
}
```

The improvement, however, is insignificant because in the worst case the improved bubble sort behaves just as the original one. The worst case for the number of comparisons is when the largest element is at the very beginning of data before sorting starts because this element can be moved only by one position in each pass. There are $(n - 1)!$ such worst cases in an array in which all elements are different. The cases, when the second largest element is at the beginning or the largest element is in the second position, and there are also $(n - 1)!$ such cases, are just as bad (only one fewer pass would be needed than in the worst case). The cases when the third largest element is in the first position are not far behind, etc. Therefore, very seldom the flag `again` fulfills its duty and very often – because an additional variable has to be maintained by `bubblesort2()` – the improved version is even slower than `bubblesort()`. Therefore, by itself, `bubblesort2()` is not an interesting modification of bubble sort. But comb sort, which builds on `bubblesort2()`, certainly is.