ALGORITHM DESIGN AND ANALYSIS

Godfry Justo
African Virtual University

LibreTexts™

Algorithm Design and Analysis

The LibreTexts libraries are Powered by NICE CXone Expert and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

This text was compiled on 06/05/2025

TABLE OF CONTENTS

Foreword

The African Virtual University (AVU) is proud to participate in increasing access to education in African countries through the production of quality learning materials. We are also proud to contribute to global knowledge as our Open Educational Resources are mostly accessed from outside the African continent.

This module was developed as part of a diploma and degree program in Applied Computer Science, in collaboration with 18 African partner institutions from 16 countries. A total of 156 modules were developed or translated to ensure availability in English, French and Portuguese. These modules have also been made available as open education resources (OER) on oer.avu. org.

On behalf of the African Virtual University and our patron, our partner institutions, the African Development Bank, I invite you to use this module in your institution, for your own education, to share it as widely as possible and to participate actively in the AVU communities of practice of your interest. We are committed to be on the frontline of developing and sharing Open Educational Resources.

The African Virtual University (AVU) is a Pan African Intergovernmental Organization established by charter with the mandate of significantly increasing access to quality higher education and training through the innovative use of information communication technologies. A Charter, establishing the AVU as an Intergovernmental Organization, has been signed so far by nineteen (19) African Governments - Kenya, Senegal, Mauritania, Mali, Cote d'Ivoire, Tanzania, Mozambique, Democratic Republic of Congo, Benin, Ghana, Republic of Guinea, Burkina Faso, Niger, South Sudan, Sudan, The Gambia, Guinea-Bissau, Ethiopia and Cape Verde.

The following institutions participated in the Applied Computer Science Program: (1) Université d'Abomey Calavi in Benin; (2) Université de Ougagadougou in Burkina Faso; (3) Université Lumière de Bujumbura in Burundi; (4) Université de Douala in Cameroon; (5) Université de Nouakchott in Mauritania; (6) Université Gaston Berger in Senegal; (7) Université des Sciences, des Techniques et Technologies de Bamako in Mali (8) Ghana Institute of Management and Public Administration; (9) Kwame Nkrumah University of Science and Technology in Ghana; (10) Kenyatta University in Kenya; (11) Egerton University in Kenya; (12) Addis Ababa University in Ethiopia (13) University of Rwanda; (14) University of Dar es Salaam in Tanzania; (15) Universite Abdou Moumouni de Niamey in Niger; (16) Université Cheikh Anta Diop in Senegal; (17) Universidade Pedagógica in Mozambique; and (18) The University of the Gambia in The Gambia.

Bakary Diallo

The Rector

African Virtual University

Licensing

Production Credits

Author

Godfry Justo

Peer Reviewer

Victor Odumuyiwa

AVU - Academic Coordination

Dr. Marilena Cabral

Overall Coordinator Applied Computer Science Program

Prof Tim Mwololo Waema

Module Coordinator

Jules Degila

Instructional Designers

Elizabeth Mbasu

Benta Ochola

Diana Tuel

Media Team

Sidney McGregor

Barry Savala

Edwin Kiprono

Kelvin Muriithi

Victor Oluoch Otieno

Michal Abigael Koyier

Mercy Tabi Ojwang

Josiah Mutsogu

Kefa Murimi

Gerisson Mulongo

Supported By

AVU Multinational Project II funded by the African Development Bank.

SECTION OVERVIEW

Course Overview

CHAPTER OVERVIEW

1: Fundamental Design and Analysis Techniques

Unit Objectives

Upon completion of this unit you should be able to:

- Describe concepts of algorithm design and analysis
- Demonstrate knowledge of algorithm complexity analysis
- Apply iteration and recursion in problem solving
- Apply dynamic programming in problem solving

Unit Introduction

The design and analysis of algorithms has become an indispensable skill in applied computer science as it is the core for developing efficient software applications, especially, due increasing demand for complex software systems to support the contemporary world applications. This unit acts as prelude to the study of advanced data structure and algorithms discussed in this course. It highlights the basic concepts related to this subject and introduces the notions of algorithm analysis and design.

The unit provides a concise description of the key concepts used in algorithm analysis under the time and space complexity dimensions. Furthermore, a review of algorithm design methods is provided including the iteration, recursion and dynamic programming.

1.1: Activity 1 - Overview of Algorithm Design and Analysis

Introduction

There are compelling reasons to study algorithms. To put it simply, computer programs would not exist without algorithms. And with computer applications becoming indispensable in almost all aspects of our professional and personal lives, studying algorithms becomes a necessity for more and more people.

Another reason for studying algorithms is their usefulness in developing analytical skills. After all, algorithms can be seen as special kinds of solutions to problems, not answers but precisely defined procedures for getting answers. Consequently, specific algorithm design techniques can be interpreted as problem solving strategies that can be useful regardless of whether a computer is involved.

Activity Details

The term algorithm can be defined using any of the following statements:

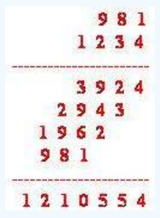
- An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- An algorithm is a finite step-by-step procedure to achieve a required result.
- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a sequence of operations performed on data that have to be organized in data structures.
- An algorithm is an abstraction of a program to be executed on a physical machine (model of Computation).

The most famous algorithm in history dates well before the time of the ancient Greeks: this is the Euclid’s algorithm for calculating the greatest common divisor of two integers.

✓ The Classic Multiplication Algorithm

Multiplication, the American way:

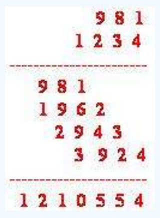
Multiply the multiplicand one after another by each digit of the multiplier taken from right to left.



```
      9 8 1
    1 2 3 4
-----------
    3 9 2 4
  2 9 4 3
1 9 6 2
9 8 1
-----------
1 2 1 0 5 5 4
```

Multiplication, the English way:

Multiply the multiplicand one after another by each digit of the multiplier taken from left to right.



```
        9 8 1
      1 2 3 4
-------------
9 8 1
1 9 6 2
  2 9 4 3
    3 9 2 4
-------------
1 2 1 0 5 5 4
```

Algorithmics is a branch of computer science that consists of designing and analyzing computer algorithms.

The “design” concern to:

- The description of algorithm at an abstract level by means of a pseudo language, and
- Proof of correctness that is, the algorithm solves the given problem in all cases.

The “analysis” deals with performance evaluation (complexity analysis).

We start with defining the model of computation, which is usually the Random Access Machine (RAM) model, but other models of computations can be use such as PRAM. Once the model of computation has been defined, an algorithm can be describe using a simple language (or pseudo language) whose syntax is close to programming language such as C or java.

Algorithm's Performance

Two important ways to characterize the effectiveness of an algorithm are its space complexity and time complexity. Time complexity of an algorithm concerns with determining an expression of the number of steps needed as a function of the problem size. Since the step count measure is somewhat rough, one does not aim at obtaining an exact step count. Instead, one attempts only to get asymptotic bounds on the step count.

Asymptotic analysis makes use of the O (Big Oh) notation. Two other notational constructs used by computer scientists in the analysis of algorithms are Θ (Big Theta) notation and Ω (Big Omega) notation. The performance evaluation of an algorithm is obtained by tallying the number of occurrences of each operation when running the algorithm. The performance of an algorithm is evaluated as a function of the input size n and is to be considered modulo a multiplicative constant.

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm.

Θ-Notation (Same order)

This notation bounds a function to within constant factors. We say f(n) = Θ(g(n)) if there exist positive constants n0, c1 and c2 such that to the right of n0 the value of f(n) always lies between c1 g(n) and c2 g(n) inclusive.

In the set notation, we write as follows:

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c1, c2, \text{ and } n0 \text{ such that } 0 \le c1\, g(n) \le f(n) \le c2\, g(n) \text{ for all } n \ge n0\}$$

We say that g(n) is an asymptotically tight bound for f(n)

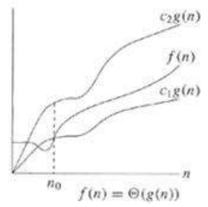


Figure 1.1.1 : Graphs to demonstrate Θ(g(n))

As shown in Figure 1.1.1 , graphically, for all values of n to the right of n0, the value of f(n) lies at or above c1 g(n) and at or below c2 g(n). In other words, for all n ≥ n0, the function f(n) is equal to g(n) to within a constant factor. We say that g(n) is an asymptotically tight bound for f(n).

In the set terminology, f(n) is said to be a member of the set Θ(g(n)) of functions. In other words, because O(g(n)) is a set, we could write

$$f(n) \in \Theta(g(n))$$

to indicate that f(n) is a member of Θ(g(n)). Instead, we write

$$f(n) = \Theta(g(n))$$

to express the same notation.

Historically, this notation is "f(n) = Θ(g(n))" although the idea that f(n) is equal to something called Θ(g(n)) is misleading.

Example: n2/2 − 2n = Θ(n2), with c1 = 1/4, c2 = 1/2, and n0 = 8.

O-Notation (Upper Bound)

This notation gives an upper bound for a function to within a constant factor. We write f(n) = O(g(n)) if there are positive constants n0 and c such that to the right of n0, the value of f(n) always lies on or below c g(n).

In the set notation, we write as follows: For a given function g(n), the set of functions

$$O(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n0 \text{ such that } 0 \le f(n) \le c\, g(n) \text{ for all } n \ge n0\}$$

We say that the function g(n) is an asymptotic upper bound for the function f(n). We use O-notation to give an upper bound on a function, to within a constant factor.

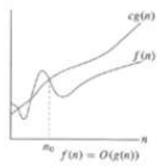


Figure 1.1.2 : Graphs to demonstrate O(g(n))

As shown in Figure 1.1.2 , graphically, for all values of n to the right of n0, the value of the function f(n) is on or below g(n). We write f(n) = O(g(n)) to indicate that a function f(n) is a member of the set O(g(n)) i.e.

$$f(n) \in O(g(n))$$

Note that f(n) = Θ(g(n)) implies f(n) = O(g(n)), since Θ-notation is a stronger notation than O-notation.

Example: 2n2 = O(n3), with c = 1 and n0 = 2.

Equivalently, we may also define f is of order g as follows:

If f(n) and g(n) are functions defined on the positive integers, then f(n) is O(g(n)) if and only if there is a c > 0 and an n0 > 0 such that

$$| f(n) | \leq | g(n) | \text{ for all } n \geq n0$$

Ω-Notation (Lower Bound)

This notation gives a lower bound for a function to within a constant factor. We write f(n) = Ω(g(n)) if there are positive constants n0 and c such that to the right of n0, the value of f(n) always lies on or above c g(n).

In the set notation, we write as follows: For a given function g(n), the set of functions

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants c and n0 such that } 0 \leq c\, g(n) \leq f(n) \text{ for all } n \geq n0\}$$

We say that the function g(n) is an asymptotic lower bound for the function f(n).

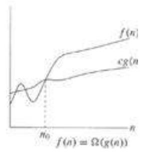


Figure 1.1.3 : Graphs to demonstrate Ω(g(n))

Figure 1.1.3 provides an insight behind Ω-notation. For example, √n = (lg n), with c = 1 and n0 = 16.

Algorithm’s Analysis

The complexity of an algorithm is a function g(n) that gives the upper bound of the number of operation (or running time) performed by an algorithm when the input size is n.

There are two interpretations of upper bound.

- Worst-case Complexity: the running time for any given size input will be lower than the upper bound except possibly for some values of the input where the maximum is reached.
- Average-case Complexity: the running time for any given size input will be the average number of operations over all problem instances for a given size.

Because it is quite difficult to estimate the statistical behaviour of the input, most of the time we content ourselves to a worst case behaviour. Most of the time, the complexity of g(n) is approximated by its family O(f(n)) where f(n) is one of the following functions: n (linear complexity), log n (logarithmic complexity), na where a ≥ 2 (polynomial complexity), an (exponential complexity).

Optimality

Once the complexity of an algorithm has been estimated, the question arises whether this algorithm is optimal. An algorithm for a given problem is optimal if its complexity reaches the lower bound over all the algorithms solving this problem. For example, any algorithm solving “the intersection of n segments” problem will execute at least n2 operations in the worst case even if it does nothing but print the output. This is abbreviated by saying that the problem has Ω(n2) complexity. If one finds an O(n2) algorithm that solves this problem, it will be optimal and of complexity Θ(n2).

Reduction

Another technique for estimating the complexity of a problem is the transformation of problems, also called problem reduction. As an example, suppose we know a lower bound for a problem A, and that we would like to estimate a lower bound for a problem B. If we can transform A into B by a transformation step whose cost is less than that for solving A, then B has the same bound as A.

The Convex hull problem nicely illustrates “reduction” technique. A lower bound of Convex-hull problem is established by reducing the sorting problem (complexity: Θ(n log n)) to the Convex hull problem.

Conclusion

This activity introduced the basic concepts of algorithm design and analysis. The need for analysing the time and space complexity of algorithms was articulated and the notations for describing algorithm complexity were presented.

Assessment

1. Distinguish between space and time complexity.
2. Briefly explain the commonly used notations in performance analysis.

1.2: Activity 2 - Recursion and Recursive Backtracking

Introduction

Recursion and backtracking are important problem solving approaches, which are alternative to iteration. An iterative solution involves loops. Not all recursive solutions are better than iterative solutions, though. Some recursive solutions may be impractical because they are so inefficient. Nevertheless, recursion can provide elegantly simple solutions to problems of great complexity.

Activity Details

When we encounter a problem that requires repetition, we often use iteration, that is, some form of loop structures. Iterative problems include the problem of printing a series of integers from n1 to n2, where n1 <= n2, whose iterative solution is presented below:

```
void printSeries(int n1, int n2) {
for (int i = n1; i < n2; i++) {
    print(i + “, “);}
println(n2);
```

An alternative approach to problems that require repetition is to solve them using recursion. A recursive method is a method that calls itself. Applying this approach to the print-series problem gives the following algorithm:

```
void printSeries(int n1, int n2) {
if (n1 == n2) {
println(n2);}
else {
    print(n1 + “, “);
    printSeries(n1 + 1, n2);}
```

Considering the recursive approach above the following is the trace for the method call

```
printSeries(5, 7):
printSeries(5, 7):
print(5 + “, “);
printSeries(6, 7):
print(6 + “, “);
printSeries(7, 7):
print(7);
return
return
return
```

When we use recursion, we solve a problem by reducing it to a simpler problem of the same kind. We keep doing this until we reach a problem that is simple enough to be solved directly. This simplest problem is known as the base case. The base case stops the recursion, because it doesn’t make another call to the method. In the printSeries function, the base case is when n1 and n2 are equal and the if block executes.

If the base case hasn’t been reached, the recursive case is executed. In the printSeries function, the recursive case is when n1 and n2 are not equal and the else block executes. The recursive case reduces the overall problem to one or more simpler problems of the same kind and makes recursive calls to solve the simpler problems. The general Structure of a Recursive Method is presented below:

```
recursiveMethod(parameters) {
    if (stopping condition) {
// handle the base case
    } else {
// recursive case:
// possibly do something here
    recursiveMethod(modified parameters);
// possibly do something here}
```

Note that there may be multiple base cases and recursive cases. When we make the recursive call, we typically use parameters that bring us closer to a base case.

Consider the problem of Printing a File in Reverse Order. How can we print the lines of a file in reverse order? It's not easy to do this using iteration. Why not? It's easy to do it using recursion! A recursive function for printing lines of a file in reverse order is presented below:

```
void printRecursive(Scanner input) {
    if (!input.hasNextLine()) { // base case
    return;}
    String line = input.nextLine();
    println(line);
    printRecursive(input); // print the rest}
```

When using recursion care must be taken to avoid infinite recursion!! We have to ensure that a recursive method will eventually reach a base case, regardless of the initial input, otherwise, we can get infinite recursion which produces stack overflow, that is, there's no room for more frames on the stack! Below is an example of our sum() method that its test for the base case can potentially cause infinite recursion. What values of n would cause infinite recursion?

```
int sum(int n) {
    if (n == 0) {
    return 0;}
    int total = n + sum(n – 1);
    return total;}
```

The philosophy for defining a recursive solution requires thinking recursively. That is when solving a problem using recursion, ask yourself these questions:

1. How can I break this problem down into one or more smaller subproblems and make recursive method calls to solve the subproblems?
2. What are the base cases? i.e., which subproblems are small enough to solve directly?
3. Do I need to combine the solutions to the subproblems? If so, how should I do so?

Recursive Backtracking

Recursive backtracking perceives that a problem solution space consists of states (nodes) and actions (paths that lead to new states). When in a node, can only see paths to connected nodes, thus if a node only leads to failure go back to its "parent" node and try other alternatives. If these all lead to failure then more backtracking may be necessary. Figure 1.2.1 presents such a tree modeled solution space.

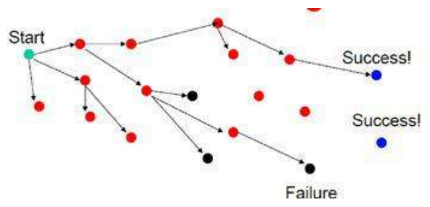


Figure 1.2.1 : Different problem solution paths

For example consider a Sudoku game of 9 by 9 matrixes with some numbers filled in, where all numbers must be between 1 and 9. The goal is that for each row, each column, and each mini matrix must contain the numbers between 1 and 9 once. That is no duplicates in rows, columns, or mini matrices are allowed. Figure 1.2.2 presents the initial game board.

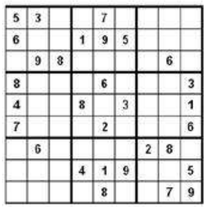


Figure 1.2.2 : Initial Sudoku game board

Solving Sudoku by **Brute Force** - A brute force algorithm is a simple but general approach, and mainly performs the following two steps.

i. Try all combinations until you find one that works.
ii. Then try and improve on the brute force results.

This approach isn't clever, but computers are fast. The Sudoku Brute force Sudoku Solution can proceed by following steps.

i. If not open cells, solved
ii. scan cells from left to right, top to bottom for first open cell
iii. When an open cell is found start cycling through digits 1 to 9.
iv. When a digit is placed check that the set up is legal
v. Now solve the board

Now, after placing a number in a cell is the remaining problem very similar to the original problem? Yes/No? Figure 1.2.3 shows a few such moves

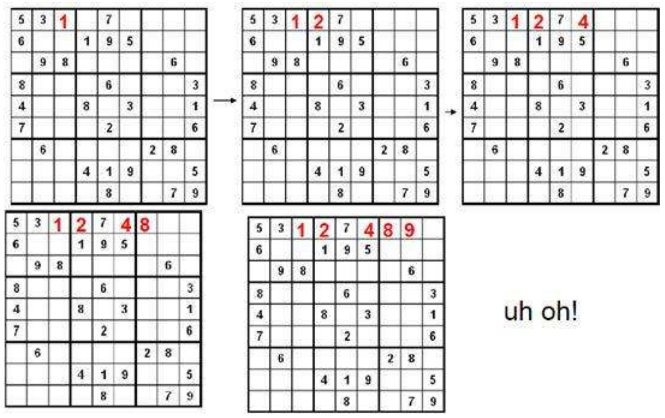
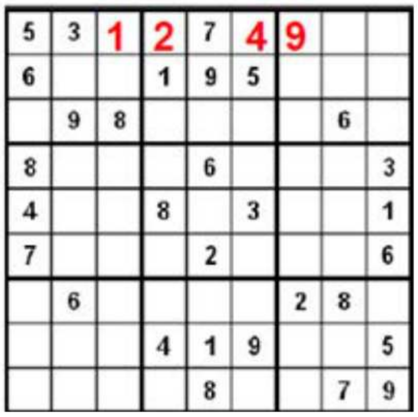


Figure 1.2.3 : Making Brute-force moves

Unfortunately, the performed move reaches a dead end in our search. Note that with the current set up none of the nine digits work in the top right corner. When the search reaches a dead end, we would to back-up to the previous cell the algorithm was trying to fill and go onto to the next digit. In this case, the algorithm back up to the cell with 9 and that turns out to be a dead end as well, so should back up again (therefore, the algorithm needs to remember what digit to try next). Now in the cell with 8, we try 9 and move forward again as shown in Figure 1.2.4 .



5	3	1	2	7	4	9		
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 1.2.4 : A back-up Replaces 8 with 9

The basic characteristics of Brute Force and Backtracking are:

- Brute force algorithms are slow
- They don’t employ a lot of logic - For example we know a 6 can’t go in the last 3 columns of the first row, but the brute force algorithm will plow ahead any way!
- But, brute force algorithms are fairly easy to implement as a first pass solution - many backtracking algorithms are brute force algorithms.

The following are the key insights when making use of the Brute Force and Backtracking:

- After trying placing a digit in a cell we want to solve the new sudoku board. Isn’t that a smaller (or simpler version) of the same problem we started with?
- After placing a number in a cell the we need to remember the next number to try in case things don’t work out.
- We need to know if things worked out (found a solution) or they didn’t, and if they didn’t try the next number
- If we try all numbers and none of them work in our cell we need to report back that things didn’t work

Problems such as Suduko can be solved using recursive backtracking. Recursive because later versions of the problem are just slightly simpler versions of the original. Backtracking because we may have to try different alternatives!

Pseudo code for recursive backtracking algorithms

i. If at a solution, report success
ii. For every possible choice from current state / node
    a. Make that choice and take one step along path
    b. Use recursion to solve the problem for the new node / state
    c. If the recursive call succeeds, report the success to the next lower level, otherwise, Back out of the current choice to restore the state at the beginning of the loop. Report failure

The following are possible goals of backtracking.

- Find a path to success
- Find all paths to success
- Find the best path to success

Figure 1.2.5 show a possible goal search space. Not all problems are exactly alike, and finding one success node may not be the end of the search.

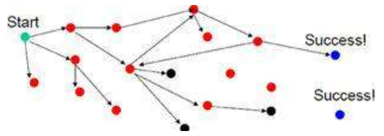


Figure 1.2.5 : Possible goal solution space

The n-Queens Problem

Consider the n-Queens Problem whose goal is to find all possible ways of placing n queens on an n x n chessboard so that no two queens occupy the same row, column, or diagonal. For example, a possible solution for n = 8 is presented in Figure 1.2.6 . This is a classic example of a problem that can be solved using a technique called recursive backtracking.

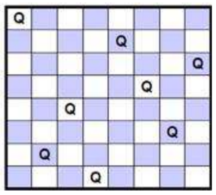


Figure 1.2.6 : Possible solution to the 8-Queens Problem

The recursive strategy for n-Queens can be described as follows: Consider one row at a time, and within the row, consider one column at a time, looking for a “safe” column to place a queen. If we find one, place the queen, and make a recursive call to place a queen on the next row. If we can’t find one, backtrack by returning from the recursive call, and try to find another safe column in the previous row. Figures 1.2.7 -1.2.10 , illustrate this strategy for n = 4.

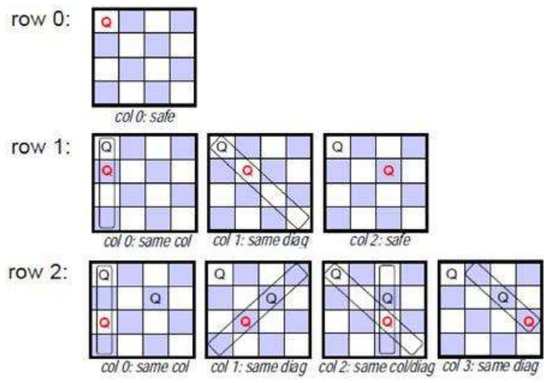


Figure 1.2.7 : A row 2 running out of columns and backtracking to 1

We have run out of columns in row 2! We backtrack to row 1 by returning from the recursive call, and pick up where we left off. We had already tried columns 0-2, so now we try column 3 as shown in Figure 1.2.7 , and continue the recursion as before.

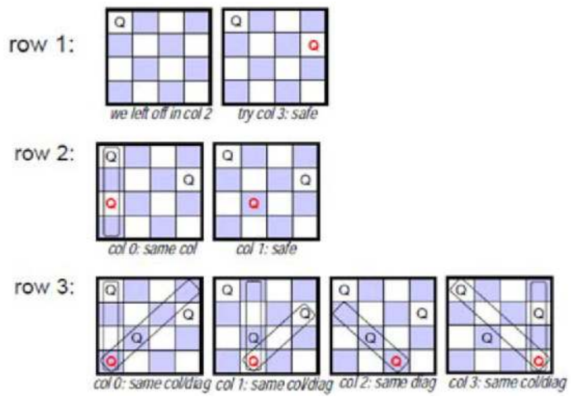


Figure 1.2.8 : A row 3 running out of columns and backtracking to 2

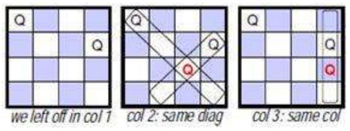


Figure 1.2.9 : A Recursion at row 2

Further, backtrack to row 1, and since no columns are left, next backtrack to row 0, to obtain the final solution for 4-Queens problem as shown in Figure 1.2.10 .

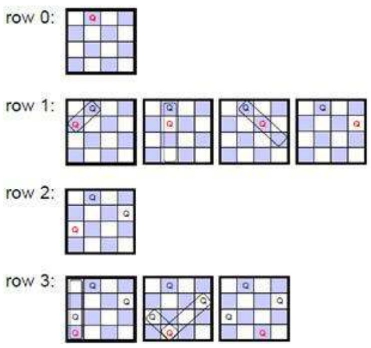


Figure 1.2.10 : A final solution to the 4-Queens Problem

Note that finding a safe column is the core of the n-Queens solution strategy. The algorithm for finding a safe column is presented in the findSafeColumn() function below, and Figure 1.2.11 , presents a snapshot of a trace for its execution.

```
void findSafeColumn(int row) {
if (row == boardSize) { // base case: a solution!
solutionsFound++;
displayBoard();
if (solutionsFound >= solutionTarget)
System.exit(0);
return;}
for (int col = 0; col < boardSize; col++) {
if (isSafe(row, col)) {
placeQueen(row, col);
// Move onto the next row.
//Note applying row++/++row don’t work!!
findSafeColumn(row + 1);
// If we get here, we’ve backtracked.
removeQueen(row, col);}
```

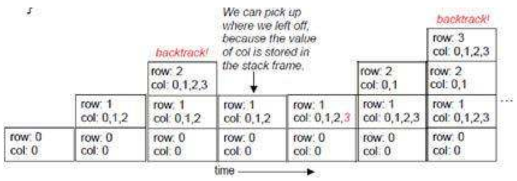


Figure 1.2.11 : Tracing find safe column algorithm

The outline of algorithms for recursive backtracking and for finding a single solution are respectively presented below:

```
//recursive backtracking algorithm
void findSolutions(n, other-params) {
if (found a solution) {
solutionsFound++;
```

```
displaySolution();
if (solutionsFound >= solutionTarget)
System.exit(0);
return;}
for (val = first to last) {
if (isValid(val, n)) {
applyValue(val, n);
findSolutions(n + 1, other params);
removeValue(val, n);}
```

// **finding a single solution**

```
boolean findSolutions(n, other-params) {
if (found a solution) {
displaySolution();
return true;}
for (val = first to last) {
if (isValid(val, n)) {
applyValue(val, n);
if (findSolutions(n + 1, other params))
return true;
removeValue(val, n); }
return false;}
```

The data structures to support the n-Queens solution implementation are now discussed. The solution to the n-Queens problem requires three key operations:

isSafe(row, col) - check to see if a position is safe

placeQueen(row, col)

removeQueen(row, col)

Therefore, a two-dimensional array of booleans would be sufficient, as defined below:

boolean[][] queenOnSquare;

The advantage of the array data structure is the easy to implement the operations place or remove a queen, that is:

```
placeQueen(int row, int col) {
queenOnSquare[row][col] = true;}
removeQueen(int row, int col) {
queenOnSquare[row][col] = false;}
```

But then, the isSafe() operation inherently take a significant number of steps. To work around this problem additional data structures for n-Queens can facilitate improvement for the isSafe() operation. We shall add three arrays of booleans:

boolean[] colEmpty;

boolean[] upDiagEmpty;

boolean[] downDiagEmpty;

An entry in one of these arrays will true if there are no queens in the column or diagonal and false otherwise. Finally, numbering diagonals to get the indices into the arrays is done in two ways as shown in Figure 1.2.12 . In the left board of Figure 1.2.12 “upDiag = row + col” and on the right board on the figure “downDiag = (boardSize – 1) + row – col”.

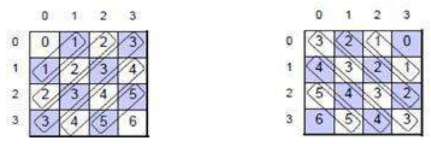


Figure 1.2.12 : Numbering Diagonals

More fine tuning done using the additional arrays lead to improved isSafe()operation performance. Note that the place and remove queen operations now involve updating four arrays instead of just one as shown by the code below:

```
void placeQueen(int row, int col) {
queenOnSquare[row][col] = true;
colEmpty[col] = false;
upDiagEmpty[row + col] = false;
downDiagEmpty[(boardSize - 1) + row - col] = false;}
```

On the other hand, checking if a square is safe become more efficient, that is:

```
boolean isSafe(int row, int col) {
return (colEmpty[col] && upDiagEmpty[row + col] &&
downDiagEmpty[(boardSize - 1) + row - col]);}
```

Conclusion

This activity reviewed the key problem solving methods: iteration, recursion and recursive backtracking. The activity demonstrated use of each method using some concrete examples.

Assessment

i. Distinguish between the following problem solving methods:
   a. Iteration and recursion
   b. Recursion and recursive backtracking
ii. Write both an iterative and recursive solution for the factorial problem.

1.3: Activity 3 - Dynamic Programming

Introduction

Dynamic Programming (DP) is about always remembering answers to the sub-problems you've already solved. Suppose that we have a machine, and to determine its state at time t, we have certain quantities called state variables. There will be certain times when we have to make a decision which affects the state of the system, which may or may not be known to us in advance. These decisions or changes are equivalent to transformations of state variables. The results of the previous decisions help us in choosing the future ones.

What do we conclude from this? We need to break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems. If you are given a problem, which can be broken down into smaller sub-problems, and these smaller sub-problems can still be broken into smaller ones - and if you manage to find out that there are some over-lapping sub-problems, then you've encountered a DP problem.

Famous DP algorithms include:

- Unix diff for comparing two files
- Bellman-Ford for shortest path routing in networks
- TeX the ancestor of LaTeX
- WASP - Winning and Score Predictor

Activity Details

The core idea of Dynamic Programming is to avoid repeated work by remembering partial results and this concept finds application in many real life situations. In programming, DP is a powerful technique that allows one to solve different types of problems in time O(n2) or O(n3) for which a naive approach would take exponential time.

A naive illustration of what dynamic programming really is summed below:

- Writes down "1+1+1+1+1+1+1+1 =" on a sheet of paper.
- "What's that equal to?"
- Counting "Eight!"
- Writes down another "1+" on the left.
- "What about that?"
- "Nine!" "How would you know it was nine so fast?"
- "You just added one more!"
- "So you didn't need to recount because you remembered!

Dynamic Programming and Recursion

Dynamic programming is basically, recursion plus using common sense. What it means is that recursion allows you to express the value of a function in terms of other values of that function. Common sense tells you that if you implement a function in a way that the recursive calls are done in advance and stored for easy access it will make the program faster. This is what we call Memoization - it is memorizing the results of some specific states, which can then be later accessed to solve other sub-problems.

The intuition behind dynamic programming is that we trade space for time, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later. For example, consider the following recurrent definition of Fibonacci numbers:

Fibonacci (n) = 1; if n = 0

Fibonacci (n) = 1; if n = 1

Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)

The first few numbers in this series are: 1, 1, 2, 3, 5, 8, 13, 21,..., and so on! Code for determining the nth number in the series using recursion is as follows:

```
int ibf (int n) {
    if (n < 2)
        return 1;
    return fib(n-1) + fib(n-2);}
```

Using Dynamic Programming approach with memorization we define the following code:

```
void fib () {
    fibresult[0] = 1;
    fibresult[1] = 1;
    for (int i = 2; i<n; i++)
    fibresult[i] = fibresult[i-1] + fibresult[i-2];}
```

Are we using a different recurrence relation in the two codes? No. Are we doing anything different in the two codes? Yes! In the recursive code, a lot of values are being recalculated multiple times. We could do better by calculating each unique quantity only once. Consider the recursive method trace shown in Figure 1.3.1 to understand how certain values were being recalculated in the recursive way:

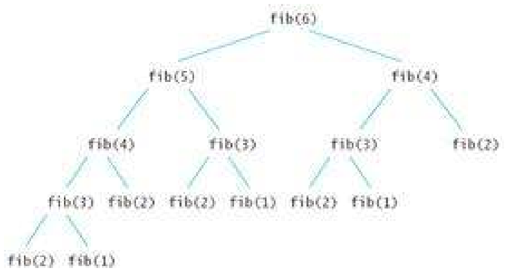


Figure 1.3.1 : Recursive Fibonacci trace for n=6

Most of the DP problems can be categorized into two types:

- Optimization problems
- Combinatorial problems

The optimization problems expect you to select a feasible solution, so that the value of the required function is minimized or maximized. Combinatorial problems expect you to figure out the number of ways to do something, or the probability of some event happening.

Every Dynamic Programming problem has a schema to be followed:

- Show that the problem can be broken down into optimal sub-problems
- Recursively define the value of the solution by expressing it in terms of optimal solutions for smaller sub-problems
- Compute the value of the optimal solution in bottom-up fashion
- Construct an optimal solution from the computed information

One can think of dynamic programming as a table-filling algorithm: you know the calculations you have to do, so you pick the best order to do them in and ignore the ones you don't have to fill in. Consider the following sample problem:

Suppose you are given a number N. Find the number of different ways to write it as the sum of 1, 3 and 4. For example, if N = 5, the answer would be 6:

i.	1 + 1 + 1 + 1 + 1
ii.	1 + 4
iii.	4 + 1
iv.	1 + 1 + 3

v.	1 + 3 + 1
vi.	3 + 1 + 1

Sub-problem: Let DPn be the number of ways to write N as the sum of 1, 3, and 4.

Finding recurrence: Consider one possible solution, n = x1 + x2 + ... xn. If the last number is 1, the sum of the remaining numbers should be n - 1. So, number of sums that end with 1 is equal to DPn-1. Take other cases into account where the last number is 3 and 4. The final recurrence would be:

DPn = DPn-1 + DPn-3 + DPn-4.

Take care of the base cases: DP0 = DP1 = DP2 = 1, and DP3 = 2. The code corresponding implementation is shown below:

```
DP[0] = DP[1] = DP[2] = 1; DP[3] = 2;
for (i = 4; i <= n; i++) {
DP[i] = DP[i-1] + DP[i-3] + DP[i-4]; }
```

The above technique takes a bottom up approach and uses memoization to avoid computing results that have already been computed.

The following narrates a puzzle to aspire the power of dynamic programming solution method:

“Imagine you have a collection of **N** wines placed next to each other on a shelf. For simplicity, let’s number the wines from left to right as they are standing on the shelf with integers from **1 to N**, respectively. Suppose the initial price of the ith wine is pi (prices of wines can be different). Because wines get better every year on year y the price of the ith wine will be y*pi, i.e. y-times the initial current year value.

Suppose we want to sell all the wines but can only sell exactly one wine per year, starting from the current year. More-over, on each year we are only allowed to sell either the leftmost or the rightmost wine on the shelf and not allowed to reorder the wines on the shelf (i.e. they must stay in the same order as they are at the beginning). What is the maximum profit obtained by selling the wines in optimal order?

For example, if the prices of the wines are (in the order as they are placed on the shelf, from left to right): p1=1, p2=4, p3=2, p4=3. The optimal solution would be to sell the wines in the order **p1, p4, p3, p2** for a total profit 1 * 1 + 3 * 2 + 2 * 3 + 4 * 4 = 29.

Wrong solution!

After playing with the problem for a while, you’ll probably get the feeling, that in the optimal solution you want to sell the expensive wines as late as possible. You can probably come up with the following greedy strategy:

Every year, sell the cheaper of the two (leftmost and rightmost) available wines.

Although the strategy doesn’t mention what to do when the two wines cost the same, this strategy feels right. But unfortunately, it isn’t, as the following example demonstrates.

If the prices of the wines are: p1=2, p2=3, p3=5, p4=1, p5=4. The greedy strategy would sell them in the order p1, p2, p5, p4, p3 for a total profit 2 * 1 + 3 * 2 + 4 * 3 + 1 * 4 + 5 * 5 = 49.

But, we can do better if we sell the wines in the order **p1, p5, p4, p2, p3** for a total profit 2 * 1 + 4 * 2 + 1 * 3 + 3 * 4 + 5 * 5 = 50.

This counter-example should convince you, that the problem is not so easy as it can look on a first sight and it can be solved using DP.

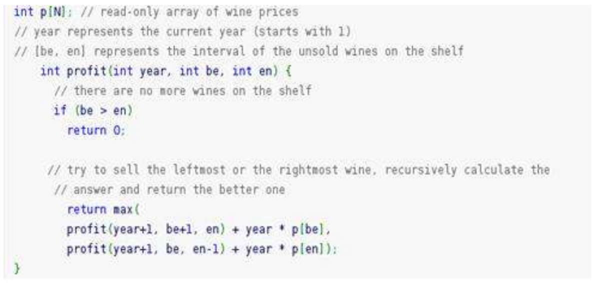
DP Solution using a backtrack

When coming up with the memoization solution for a problem, start with a backtrack solution that finds the correct answer. Backtrack solution enumerates all the valid answers for the problem and chooses the best one. Below are some restrictions on the backtrack solution:

- It should be a function, calculating the answer using recursion
- It should return the answer with return statement, i.e., not store it somewhere

- All the non-local variables that the function uses should be used as read-only, i.e. the function can modify only local variables and its arguments.

The following code uses the backtrack approach:



```
int p[N]; // read-only array of wine prices
// year represents the current year (starts with 1)
// [be, en] represents the interval of the unsold wines on the shelf
    int profit(int year, int be, int en) {
      // there are no more wines on the shelf
      if (be > en)
        return 0;

     // try to sell the leftmost or the rightmost wine, recursively calculate the
      // answer and return the better one
        return max(
        profit(year+1, be+1, en) + year * p[be],
        profit(year+1, be, en-1) + year * p[en]);
}
```

The above solution simply tries all the possible valid orders of selling the wines. If there are **N** wines in the beginning, it will try **2^N** possibilities (each year we have 2 choices). But even though we get the correct answer, the time complexity of the algorithm grows exponentially.

The correctly written backtrack function should always represent an answer to a well-stated question. In our case profit function represents an answer to a question: “What is the best profit we can get from selling the wines with prices stored in the array p, when the current year is year and the interval of unsold wines spans through [be, en], inclusive?”. You should always try to create such a question for your backtrack function to see if you got it right and understand exactly what it does.

We should try to minimize the state space of function arguments. In this step think about, which of the arguments you pass to the function are redundant. Either we can construct them from the other arguments or we don’t need them at all. If there are any such arguments, don’t pass them to the function. Just calculate them inside the function.

In the above function profit, the argument year is redundant. It is equivalent to the number of wines we have already sold plus one, which is equivalent to the total number of wines from the beginning minus the number of wines we have not sold plus one. If we create a read-only **global variable N**, representing the total number of wines in the beginning, we can rewrite our function as follows:

```
int N; // read-only number of wines in the beginning
int p[N]; // read-only array of wine prices
    int profit(int be, int en) {
        if (be > en)
            return 0;
        // (en-be+1) is the number of unsold wines
        int year = N - (en-be+1) + 1; // as in the description above
        return max(
            profit(be+1, en) + year * p[be],
            profit(be, en-1) + year * p[en]); }
```

We are now 99% done. To transform the backtrack function with time complexity O(2N) into the memoization solution with time complexity O(N2), we will use a little trick which doesn’t require almost any thinking. As noted above, there are only O(N2) different arguments our function can be called with. In other words, there are only O(N2) different things we can actually compute.

So where does O(2N) time complexity comes from and what does it compute? The answer is - the exponential time complexity comes from the repeated recursion and because of that, it computes the same values again and again. If you run the above code for an arbitrary array of N=20 wines and calculate how many times the function was called for arguments be=10 and en=10 you will get a number 92378.

That’s a huge waste of time to compute the same answer that many times. What we can do to improve this is to memoize the values once we have computed them and every time the function asks for an already memoized value, we don’t need to run the whole recursion again, as shown in the following code:

int N; // read-only number of wines in the beginning

int p[N]; // read-only array of wine prices

int cache[N][N]; // all values initialized to -1 (or anything you choose)

```
int profit(int be, int en) {
    if (be > en)
        return 0;
    // these two lines save the day
    if (cache[be][en] != -1)
        return cache[be][en];
    int year = N - (en-be+1) + 1;
    // when calculating the new answer, don’t forget to cache it
    return cache[be][en] = max(
        profit(be+1, en) + year * p[be],
        profit(be, en-1) + year * p[en]); }
```

To sum it up, if you identify that a problem can be solved using DP, try to create a backtrack function that calculates the correct answer. Try to avoid the redundant arguments, minimize the range of possible values of function arguments and also try to optimize the time complexity of one function call (remember, you can treat recursive calls as they would run in O(1) time). Finally, you can memoize the values and don’t calculate the same things twice.

Conclusion

This activity presented the dynamic programming problem solving approach. Dynamic programming is basically recursion and use of common sense to store (memoize) some recursive calls results and access the results to speed computation of other sub-problems.

? Assessment

1. Explain the schema to be followed when solving a DP problem.
2. How do the DP with backtrack method differs from recursive backtracking?

1.4: Unit Summary

This unit introduced the basic concepts of algorithm design and analysis and associated techniques. The need for analysing the time and space complexity of algorithms was articulated and the notations for describing algorithm complexity were presented. The unit further discussed the fundamental solution design methods as well as demonstrated each method using concrete examples. The problem solving methods included the iteration, recursion, recursive backtracking and dynamic programming.

? Unit Assessment

Check your understanding!

Comparing performance of DP, Recursion and Iteration methods

Instructions

1. Consider the following iterative solution of the Fibonacci problem:

```
int iterativeFibonacci(int n) {
//iterative solution to the Fibonacci problem
//initialise base cases:
int previous = 1; // initially rabbit(1)
int current = 1; //initially rabbit(2)
int next = 1; // results when n is 1 or 2
//compute next Fibonacci values when n >= 3
    for (int i = 3; i <= n; i++) {
    //current is rabbit(i-1), previous is rabbit(i-2)
    next = current + previous; //rabbit(i)
    previous = current;
    current = next;
}// end for
return next;
} //end iterativeFibbonacci
```

Implement the iterative, recursive, and DP solutions for the Fibonacci problem and compare the time taken by each solution method for n=1000.

Answers

mailto:njulumi@gmail.com

Grading Scheme

As per the offering Institution discretion.

Unit Readings and Other Resources

The readings in this unit are to be found at course level readings and other resources.

CHAPTER OVERVIEW

2: Searching, Binary Search Trees and Heaps

Unit Objectives

Upon completion of this unit you should be able to:

- Demonstrate knowledge of searching algorithms
- Implement a searching algorithm based on efficiency requirements
- Demonstrate knowledge of advanced data structures including binary search trees and binary heap
- Implement a data structure based on efficiency requirements

Unit Introduction

Searching is one of the most fundamental problems in Computer Science. It is the process of finding a particular item in a collection of items. Typically, a search answers whether the item is present in the collection or not. This unit discusses two important search methods: sequential search and binary search. A Sequential (linear) search is the simplest method to solve the searching problem. It finds an item in a collection by looking for it in a collection one at a time from the beginning till the end. Binary Search on the other hand is use a divide and conquer strategy. In divide and conquer, a larger problem is reduced into smaller sub-problems recursively until the problem is small enough that the solution is trivial. A condition for binary search is that the array should be a sorted array.

The unit also discusses two important data structures; a binary search tree (BST) and a heap. A BST is a binary tree based data structure that is viewed to support efficiently the dynamic set operations, including search and insert operations amongst others. A heap on the other hand is a specific tree based data structure in which all the nodes of tree are in a specific order. The maximum number of children of a node in the heap depends on the type of heap. However, a binary heap is the most commonly used heap type, where there are at most 2 children of a node.

2.1: Activity 1 - Searching Techniques

Introduction

Searching is one of the most fundamental problems in Computer Science. It is the process of finding a particular item in a collection of items. Typically, a search answers whether the item is present in the collection or not. For simplification for all our examples, we will be taking an array as the collection. This activity will present two searching methods: the sequential search and the binary search methods.

Activity Details

Sequential Search

Sequential (linear) search is the simplest method to solve the searching problem. It finds an item in a collection by looking one item at a time from the beginning to end of collection if it does not find it. Basically, it checks each item in the sequence until the desired element is found before all the elements of the collection are exhausted.

The code snapshot below presents a linear search method.

```
bool linearSearch (int A[], int length, int item) {
for (int i = 0 ; i < length ; ++i)
if (item == A[i])
return true;      // Item is found in array A
return false;     // Item is not found in array A }
```

Consider an array A = {1, 9, 2, 4, 6, 3, 7, 5, 8, 0} and suppose that we wish to find 3 from the given array. The linear search starts from the beginning and checks if current item matches with the item we are searching. In the provided example the item = 3 and the array length = 10.

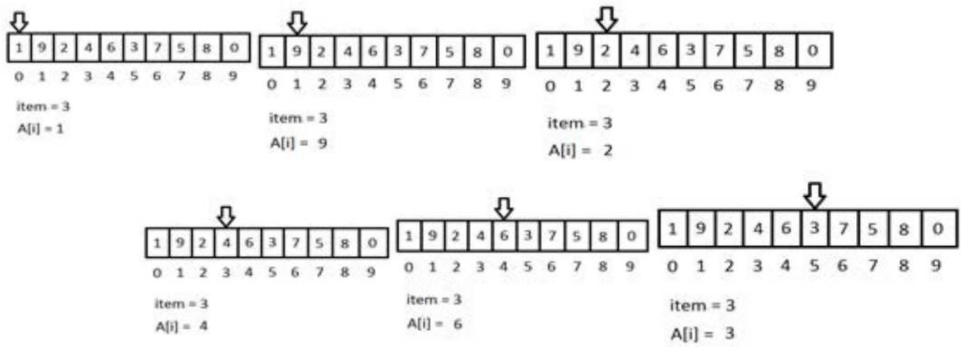


Figure 2.1.1 : Trace of Linear Search for item = 3

Figure 2.1.1 shows the trace of the sequential search algorithm to find item 3. Note that for i = 0, A[i] = 1 and the item being sought is not equal to A[i]. Similarly, for i = 1, A[i] = 9 and item is not equal to A[i], for i = 2, A[i] = 2 and item is not equal to A[i], for i = 3, A[i] = 4 and item is not equal to A[i] and for i = 4, A[i] = 6 and item is not equal to A[i]. But for For i = 5, A[i] = 3 and item is equal to A[i]. Therefore, the linearSearch() function will return true showing that the item is found. Note that, if the item = 10, linearSearch() function will iterate over all the elements in the array A, and return false (showing that the item is not found), because the 'if' condition will never be true.

Now, let us consider the modified search where we need to return the position, where the item is found. To solve this modified search problem, we just need to change two lines in the linearSearch() function as shown below:

```
int linearSearch ( int A[ ], int length, int item) {
    for (int i = 0; i < length ; ++i)
        if (item == A[i])
            return i;     //Returning the index of the element found.
    return -1;            // Item is not found in array A
}
```

In the above codes -1 denotes that the item is not found. Note that if there are duplicate elements, the above code will return the first occurrence of the item. The time complexity of linear search is **O(size of the array).**

Binary Search

Binary Search is a divide and conquer algorithm. In divide and conquer, a larger problem is reduced into smaller sub-problems recursively until the problem is small enough that the solution is trivial. Unlike sequential search, binary search requires that the input array be a sorted array. If the array is sorted in ascending order, binary search compares the item with the middle element of the array. If the item matches, then it returns true. Otherwise, if the item is less than the middle element then it will recursively search on the sub-array to the left of the middle element or if the item is greater it will recursively search on the sub-array to the right of the middle element. Therefore, at each step the size of array will be reduced to half. After repeating this recursion, it will eventually be left with a single element.

The pseudocode for the binary search algorithm is presented below:

```
Algorithm binarySearch(A, left, right, item) {
    if left is less than or equal to right then :
        mid = (left + right) / 2
        if A[mid] is equal to item then return true
        else if item is less than A[mid] then recursively search on the left sub-arra
        else if item is greater than A[mid] recursively on the right sub-array
    else
        return false }
```

Remember that an array should be sorted for the binary search to work!! A recursive implementation of binary search is presented below:

```
//Recursive Binary Search
bool binarySearchRecur(int A[], int left, int right, int item)
{
    if (left <= right)
    {
        int mid = left + (right - left) / 2; // finding middle index
        if (A[mid] == item)
            return true;                     // item found
        else if (item < A[mid])
        {
            // recursively search on the left sub-array
            return binarySearchRecur(A, left, mid-1, item);
        }
        else
        {
            // recursively search on the right sub-array
            return binarySearchRecur(A, mid+1, right, item);
        }
    }
else
    return false; // item not found )
```

Similarly, an iterative implementation of binary search is given below:

```
//Iterative Binary Search
bool binarySearchIter(int A[], int length, int item)
    { int left = 0, right = length - 1, mid;
        while (left <= right)
        { mid = left + (right - left) / 2;   // finding middle index
            if (A[mid] == item)
                return true;                 // item found
            else if (item < A[mid])
                right = mid - 1;             // search on left sub-array
            else
                left = mid + 1;              // search on right sub-array
          }
        return false;                        // item not found }
```

Suppose now we are given array A = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} and we need to find 3 in the array A. Therefore, we have that search item = 3 and array length = 10. Table 2.1.1 traces the execution of the iterative binary search for item =3 on array A. Note that initially left = 0 and right = length – 1 = 10 – 1 = 9. Figure 2.1.2 depicts that the array diminishes during iterations.

Table 2.1.1 : A trace for Iterative Binary Search on array A

Iteration	left	Right	mid	Remarks
1	0	9	mid = left + (right – left) / 2 = 0 + (9 – 0) / 2 = 4	A[mid] = 4 is greater than item. Since the array A is sorted we conclude that the item being found is in the left sub-array
2	0	mid – 1 = 4 –1 = 3	left + (right – left) / 2 = 0 + (3 – 0) / 2 = 1	mid = A[mid] = 1 is smaller than item. Since the array A is sorted the item must be in the right sub-array.
3	mid + 1 = 1 + 1 = 2	right = 3	left + (right – left) / 2 = 2 + (3 – 2) / 2 = 2	mid = A[mid] = 2 is smaller than item. Since the array A is sorted we can say that the item must be in the right sub-array
4	mid + 1 = 2 + 1 = 3	right = 3	mid = left + (right – left) / 2 = 3 + (3 – 3) / 2 = 3	A[mid] = 3 is equal to the item, so the binarySearchIter() function will return true and the item is found

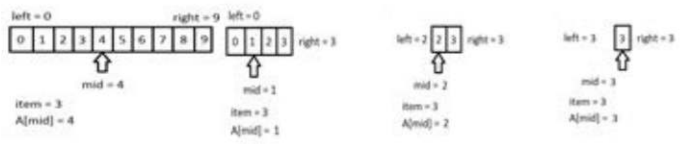


Figure 2.1.2 : Array diminishes during iterations

Time complexity of both binarySearchRecur() and binarySearchIter() is O(logN) where N is the size of the array. This is because after each recursion or iteration the size of problem is reduced into half.

Conclusion

In this activity we presented two search algorithms, sequential search and binary search. The sequential search method is based on exhaustive search strategy while the binary search method is based on divide and conquers search strategy. Binary search also requires that the input array to be sorted.

Assessment

1. Modify the linearSearch() function discussed in this activity such that it will return the last occurrence of the item, if the item is in the array, otherwise, return -1. (Hint: Store the position of the last found occurrence of the item and update it as soon as the item occurs again).
2. Suppose you are given a sorted array with possible duplicate elements within it. Write the code segment that finds the first occurrence of a given input item (assuming the given item exists in the given array).

2.2: Activity 2 - Binary Search Tree

Introduction

A binary tree is made of nodes, where each node contains a left pointer, a right pointer, and a data element also called a key. The root pointer points to the topmost node in the tree. The left and right pointers recursively points to left and right sub-tree respectively. A null pointer represents a binary tree with no elements, that is, an empty tree. Figure 2.2.1 depicts a binary tree.

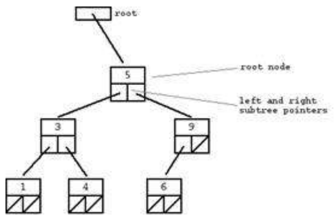


Figure 2.2.1 : A binary tree

A **binary search tree** (BST) also called an ordered binary tree is a type of binary tree where the nodes are arranged in order. That is, for each node, all elements in its left sub-tree are less-or-equal to its element, and all the elements in its right sub-tree are greater than its element. The tree shown in Figure 2.2.1 is a binary search tree since the root node element is a 5, and its left sub-tree elements (i.e., 1, 3, 4) are less than 5, and its right sub-tree elements (i.e., 6, 9) are greater than 5. Recursively, each of the sub-trees must also obey the binary search tree constraint, that is, in the (1, 3, 4) sub-tree, the node with element 3 is the root, and that 1 <= 3 and 4 > 3. In this regard, note that a binary search tree is different from a binary tree. The nodes at the bottom edge of the tree which have empty sub-trees and are called leaf nodes (i.e., 1, 4, 6), while the others are called internal nodes (i.e., 3, 5, 9).

Activity Details

A binary tree is recursively defined as follows:

i. An empty tree is a binary tree
ii. A node with two child sub-trees is a binary tree
iii. Only what you get from (i) by a finite number of applications of (ii) is a binary tree.

Now check whether Figure 2.2.2 is a binary tree.

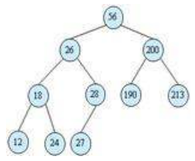


Figure 2.2.2 : A tree

A binary search tree (BST) is a fundamental data structure that can support dynamic set operations including: Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete. The efficiency of its operations (especially, the search and insert operations) make it be suited to use of building dictionaries and priority queues. Its basic operations take time proportional to the height, h, of the tree: i.e., O(h).

Like a binary tree, binary search tree is represented by a linked data structure of nodes. The root (T) points to the root of tree T. Each node contains three fields: key, left (pointer to left child/root of left sub-tree) and right (pointer to right child/root of right sub-tree).

A binary tree is a binary search tree if the stored keys satisfy the following two binary search tree properties:

- ∀ y in left sub-tree of x, then key[y] ≤ key[x].
- ∀ y in right sub-tree of x, then key[y] ≥ key[x].

Do the keys of the binary tree in Figure 2.2.3 satisfy binary search tree properties?

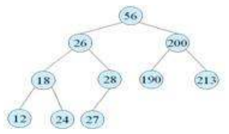
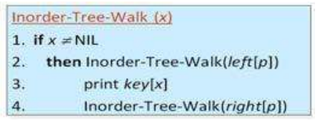


Figure 2.2.3 : A binary tree

A binary search tree property allows its keys to be recursively visited/printed in monotonically increasing order, called an in-order traversal (walk). Figure 2.2.4 presents a pseudo-code for the in-order traversal. The correctness of the in-order walk can be elaborated through induction approach on the size of tree as follows: For an empty tree (i.e., size=0), this is easy. Suppose size >1, then

- Prints left sub-tree in order by induction
- Prints root, which comes after all elements in left sub-tree (still in order)
- Prints right sub-tree in order (all elements come after root, so still in order)

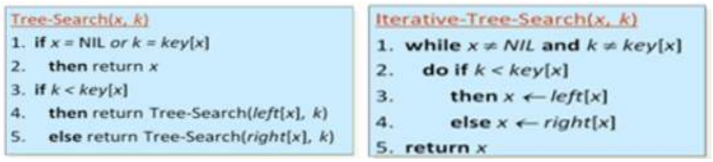


```
Inorder-Tree-Walk (x)
1. if x ≠ NIL
2.   then Inorder-Tree-Walk(left[p])
3.        print key[x]
4.        Inorder-Tree-Walk(right[p])
```

Figure 2.2.4 : In-order traversal

Recall that all dynamic set search operations on a binary search tree can be supported in O(h) time, where h is the tree height. Note that h = Θ(lg n) for a balanced binary tree (and for an average tree built by adding nodes in random order) and that h = Θ(n) for an unbalanced tree that resembles a linear chain of n nodes in the worst case.

In the following we present the algorithms for some of the dynamic set operations. In Figure 2.2.5 and 2.2.6 presents the algorithms for the tree search operation. Figure 2.2.5 presents the algorithm for the search operation using both the recursive and iterative method respectively. The running time for both algorithms is O(h). However, the iterative tree search method is more efficient on most computers, while the recursive tree search is elegantly straightforward.

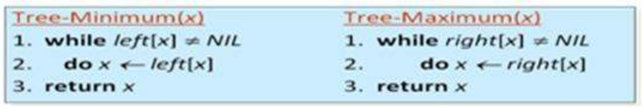


```
Tree-Search(x, k)
1. if x = NIL or k = key[x]
2.   then return x
3. if k < key[x]
4.   then return Tree-Search(left[x], k)
5.   else return Tree-Search(right[x], k)
```

```
Iterative-Tree-Search(x, k)
1. while x ≠ NIL and k ≠ key[x]
2.   do if k < key[x]
3.        then x ← left[x]
4.        else x ← right[x]
5. return x
```

Figure 2.2.5 : A recursive and iterative search algorithm

The other important dynamic set operation is finding Minimum and Maximum key respectively. The binary search tree properties guarantees that the minimum key is located at the left-most node and the maximum key is located at the right-most node. Figure 2.2.6 presents the algorithm for finding minimum and maximum keys.



```
Tree-Minimum(x)
1. while left[x] ≠ NIL
2.   do x ← left[x]
3. return x
```

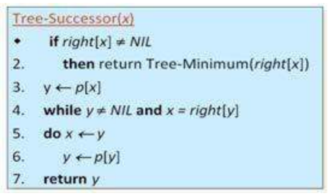
```
Tree-Maximum(x)
1. while right[x] ≠ NIL
2.   do x ← right[x]
3. return x
```

Figure 2.2.6 : Finding Min and Max

The other dynamic set operations on a binary search tree are finding the Predecessor and Successor keys. The successor of node x is the node y such that key[y] is the smallest key greater than key[x]. The successor of the largest key is NIL. The search for successor key consists of two cases:

- If node x has a non-empty right sub-tree, then x’s successor is the minimum in the right sub-tree of x.
- If node x has an empty right sub-tree, then
  - As long as we move to the left up the tree (move up through right children), we are visiting smaller keys.
  - x’s successor y is the node that x is the predecessor of (x is the maximum in y’s left sub-tree).
    That is, x’s successor y, is the lowest ancestor of x whose left child is also an ancestor of x.

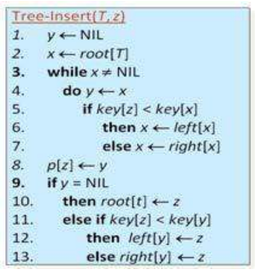
Figure 2.2.7 presents the pseudo-code for the successor operation. The code for predecessor operation is symmetric. The running time is both cases is O(h).



```
Tree-Successor(x)
•   if right[x] ≠ NIL
2.      then return Tree-Minimum(right[x])
3.  y ← p[x]
4.  while y ≠ NIL and x = right[y]
5.  do x ← y
6.      y ← p[y]
7.  return y
```

Figure 2.2.7 : Finding successor

Insertion is also an important dynamic set operation on binary search tree. The insertion operation changes the dynamic set represented by a BST. The operation should ensure the binary search tree properties holds after a change. Generally, insertion is easier than deletion. Figure 2.2.8 presents the pseudo-code for insertion operation.



```
Tree-Insert(T,z)
1.   y ← NIL
2.   x ← root[T]
3.   while x ≠ NIL
4.       do y ← x
5.           if key[z] < key[x]
6.               then x ← left[x]
7.               else x ← right[x]
8.   p[z] ← y
9.   if y = NIL
10.      then root[t] ← z
11.      else if key[z] < key[y]
12.          then  left[y] ← z
13.          else right[y] ← z
```

Figure 2.2.8 : BST Insertion operation

The analysis of insertion operation is as follows:

- Initialization takes O(1)
- The while loop in lines 3-7 searches for place to insert z, maintaining parent y. This takes O(h) time.
- Lines 8-13 insert the value in O(1).

Thus, overall it takes O(h) time to insert a node.

The delete operation in a BST is somehow involving. The BST deletion operation Tree-Delete (T, x) considers three cases:

- case 0 : if x has no children then remove x
- case 1 : if x has one child then make p[x] point to child
- case 2 : if x has two children (sub-trees), then swap x with its successor and perform case 0 or case 1 to delete it.

Overall delete operation takes O(h) time to delete a node. The pseudo-code for deleting a key in a BST is presented below:

```
Tree-Delete(T, z)
/* Determine which node to splice out: either z or z's successor. */
if left[z] = NIL or right[z] = NIL
then y ← z
else y ← Tree-Successor[z]
/* Set x to a non-NIL child of x, or to NIL if y has no children. */
if left[y] ≠ NIL
then x ← left[y]
else x ← right[y]
/* y is removed from the tree by manipulating pointers of p[y] and x */
if x ≠ NIL
then p[x] ← p[y]
if p[y] = NIL
then root[T] ← x
else if y ← left[p[i]]
then left[p[y]] ← x
else right[p[y]] ← x
```

```
/* If z’s successor was spliced out, copy its data into z */
if y ≠ z
then key[z] ← key[y]
copy y’s satellite data into z.
return y
```

How do we know that case 2 should go to case 0 or case 1 instead of back to case 2? Due to the fact that when x has 2 children, its successor is the minimum in its right sub-tree, and that successor has no left child (hence 0 or 1 child). Equivalently, we could swap with predecessor instead of successor. It might be good to alternate to avoid creating unbalanced tree.

Conclusion

In this activity binary search trees were presented. Binary search trees is among advanced data structures of practical interest due to the efficiency of its basic operations including search and insert that make it attractive for implementing dictionaries and priority queues.

? Assessment

1. Explain each of the following terms:
   a. Binary tree
   b. Binary search tree
   c. In-order tree traversal
2. Draw the BST that results when you insert items with keys

   A F R I C A N V I R T U A L U N I V E R S I T Y

   in that order into an initially empty tree.
3. Suppose we have integer values between 1 and 1000 in a BST and search for 363. Which of the following cannot be the sequence of keys examined (Hint: Draw the BST as if you were tracing the keys in the order given).
   a. 2 252 401 398 330 363
   b. 399 387 219 266 382 381 278 363
   c. 3 923 220 911 244 898 258 362 363
   d. 4 924 278 347 621 299 392 358 363
   e. 5 925 202 910 245 363

2.3: Activity 3 - Heaps

Introduction

A heap is a specific tree based data structure in which all the nodes of tree are in a specific order. Suppose that X is a parent node of Y, then the value of X follows some specific order with respect to value of Y and the same order will be followed across the tree. The maximum number of children of a node in the heap depends on the type of heap. However, a binary heap is the more commonly used heap type, in which there are at most 2 children of a node.

Activity Details

In binary heap, if the heap is a complete binary tree with N nodes, then it has smallest possible height which is $\mathbf{\log_2 N}$. For example, from Figure 2.3.1 , observe a particular sequence that each node has greater value than any of its children. Suppose there are N Jobs in a queue to be done, and each job has its own priority. The job with maximum priority will get completed first than others. At each instant we are completing a job with maximum priority and at the same time we are also interested in inserting a new job in the queue with its own priority.

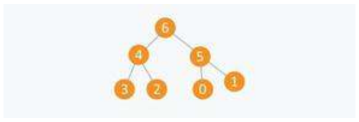


Figure 2.3.1 : A binary heap

So at each instant we have to check for the job with maximum priority to complete it and also insert if there is a new job. This task can be very easily executed using a heap by considering N jobs as N nodes of the tree.

We can use an array to store the nodes of the heap tree. Consider a heap depicted in Figure 2.3.2 and suppose that there are 7 elements with values {6, 4, 5, 3, 2, 0, 1}. We can use an array to simulate a tree in the following way: If we are storing one element at index, i, in array Arr, then its parent will be stored at index i/2 (unless it’s a root, since a root has no parent) and can be access by Arr[i/2], and its left child can be accessed by Arr[2 * i] and its right child can be accessed by Arr[2 * i +1]. The index of root in an array is 1.

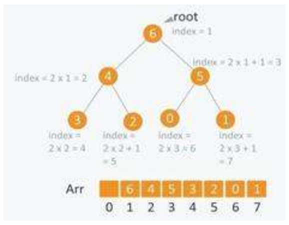


Figure 2.3.2 : Heap and Corresponding Array representation

There are two basic types of heap, max heap and min heap. In a max heap the value of parent node is always greater than or equal to the value of child node across the tree and the node with highest value is the root node of the tree. In a min heap the value of parent node is always less than or equal to the value of child node across the tree. Therefore the node with lowest value is the root node of tree.

Max heap implementation

Suppose that you are given heap elements which are stored in array Arr. We can convert the array Arr into a heap structure as follows: Pick a node in the array, check if the left sub-tree and the right sub-tree are max heaps, among themselves and the node itself is a max heap (i.e., its value should be greater than all the child nodes). To perform this operation we define a function max_heapify that maintain the property of max heap (i.e. each element value should be greater than or equal to any of its child and smaller than or equal to its parent) as presented below:

```
void max_heapify (int Arr[ ], int i, int N)
{   int left  = 2*i             //left child
    int right = 2*i +1          //right child
    if(left<= N and Arr[left] > Arr[i] )
```

```
            largest = left;
        else
            largest = i;
        if(right <= N and Arr[right] > Arr[largest] )
            largest = right;
        if(largest != i )
        { swap (Ar[i] , Arr[largest]);
            max_heapify (Arr, largest,N); }
```

The function max_heapify has time Complexity of O(log N).

Now consider the example given in Figure 2.3.3 depicts the max_heapify tree transformation steps. Initially 1st node (root node) is violating property of max-heap as it has smaller value than its children, so we are performing max_heapify function on this node with value 4. Since 8 is greater than 4, then 8 is swapped with 4 and max_heapify is performed again on 4, but on different position. Now in step 2, 6 is greater than 4, so 4 is swapped with 6 and we obtain a max heap, as now 4 is a leaf node, so further call to max_heapify will not create any effect on the heap. Therefore we can correct a violated max heap property by using **max_heapify** function.

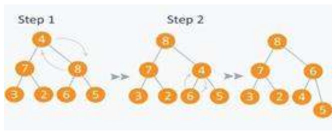


Figure 2.3.3 : Transformation steps to max heap

Now consider the heap property which states:

An N element heap stored in an array has leaves indexed by N/2+1, N/2+2 , N/2+3 …. up to N.

Figure 2.3.4 demonstrates this heap property which is the heap derived from Figure 2.3.3 whose elements have values {8, 7, 6, 3, 2, 4, 5}. Observe that the elements 3, 2, 4, 5 are indexed by N/2 +1 (i.e 4), N/2+2 (i.e 5) and N/2+3 (i.e 6) and N/2+4 (i.e 7) respectively.

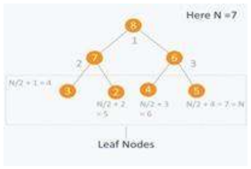


Figure 2.3.4 : Array indices for heap leaf elements

We can build a max heap out of the array elements. That is, suppose that we have N elements stored in the array Arr indexed from 1 to N, and possibly not following the property of a max heap. We use the max-heapify function to make a max heap out of the array. Based on the property stated above, we have that the elements from Arr[N/2+1] to Arr[N] as leaf nodes, and each node is a 1 element heap. The max_heapify function can be used in a bottom up manner on remaining nodes, to cover each node of the heap tree. The code presented below captures these concepts to build up a max heap from a given array:

```
void build_maxheap (int Arr[ ])
{ for(int i = N/2 ; i >= 1 ; i-- )
    { max_heapify (Arr, i) ; }
```

Note that the time complexity for the build_maxheap function is O(N), since max_heapify function has complexity log N and the build_maxheap functions runs for only N/2 times.

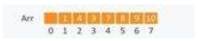


Figure 2.3.5 : An array Arr

Now we demonstrate the max_heap function function using a 7 elements array Arr, shown in Figure 2.3.5 . Here N = 7, so starting from node having index N/2 = 3, (i.e., with value 3 in Figure 2.3.5), call max_heapify from index N/2 to 1. Figure 2.3.6 depicts the build steps in which:

During step 1, in max_heapify(Arr, 3), since 10 is greater than 3, 3 and 10 are swapped and further call to max_heap(Arr, 7) will have no effect as 3 is a leaf node.

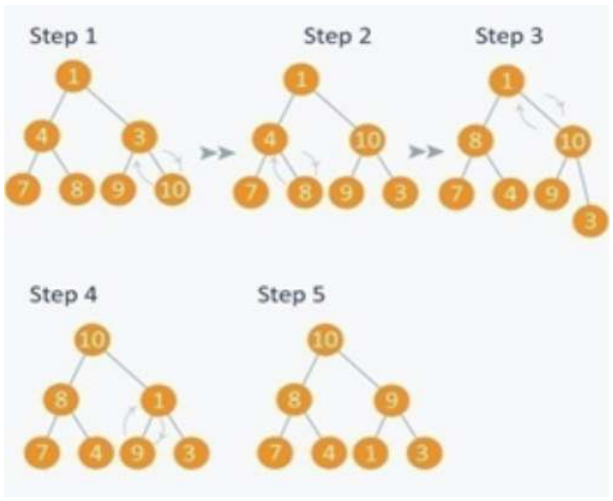


Figure 2.3.6 : Tracing Max heap creation

During step 2, calling max_heapify(Arr, 2), sees node indexed with 2 has value 4, thus 4 is swapped with 8 and further call to max_heap(Arr, 5) will have no effect, since 4 is a leaf node.

During step 3, calling max_heapify(Arr, 1), sees node indexed with 1 has value 1, hence 1 is swapped with 10.

Step 4 is a subpart of step 3, as after swapping 1 with 10, again a recursive call of max_heapify(Arr, 3) will be performed, and 1 will be swapped with 9. Now further call to max_ heapify(Arr, 7) will have no effect, as 1 is a leaf node.

In step 5, we finally get a max- heap and the elements in the array Arr will be as shown in Figure 2.3.7 .

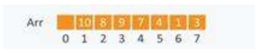


Figure 2.3.7 : Resulting Max-heap array representation

The second heap type, min heap, has the value of parent node always less than or equal to the value of child node across the tree. Therefore the node with lowest value is the root node as depicted in Figure 2.3.8 . This figure also captures the fact that a node has a smaller value than the values of its children.

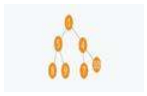


Figure 2.3.8 : A min-heap

In a mini heap we can perform similar operations as in the max heap. Below we present a function min_heapify which maintains the min heap property:

```
void min_heapify (int Arr[ ] , int i, int N)
{ int left = 2*i;
int right = 2*i+1;
int smallest;
if(left <= N and Arr[left] < Arr[ i ] )
    smallest = left;
else
    smallest = i;
if(right <= N and Arr[right] < Arr[smallest] )
    smallest = right;
if(smallest != i)
```

```
{ swap (Arr[ i ], Arr[ smallest ]);
    min_heapify (Arr, smallest,N); }
```

Note that as in the max_heapify function, the time complexity of the min_heapify function is O (log N).

We demonstrate the above function by transforming the array Arr = {4, 5, 1, 6, 7, 3, 2}. Note from Figure 2.3.9 , that the element at index 1 is violating the property of min -heap, so performing min_heapify(Arr, 1) preserves the min-heap property.

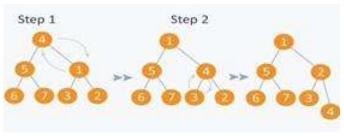


Figure 2.3.9 : Transforming to a min–heap

Apply the min_heapify function on remaining nodes other than leaves (since a leaf node is a 1 element heap), as defined by function build_minheap below to obtain to a min heap.

```
void build_minheap (int Arr[ ])
{   for( int i = N/2 ; i >= 1 ; i--)
    min_heapify (Arr, i);  }
```

Note that as for the build_maxheap function the time complexity for the build_minheap function is O(N).

We demonstrate the build_minheap function by transforming the elements in array {10, 8, 9, 7, 6, 5, 4}. To do so, need to run min_heapify on nodes indexed from N/2 to 1, where the node indexed at N/2 has value 9. The build_minheanp repeats this through a number of steps to obtain a min_heap as depicted in Figure 2.3.10 .

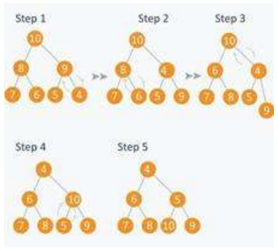


Figure 2.3.10 : Building min-heap from array

Conclusion

In this activity we presented heap data structure. We discussed two types of heap, max heap and min heap, along with the key operation of building a specified heap from an array. Heaps are considered as partially ordered tree, since the nodes of tree do not follow any order with their siblings (nodes on the same level). They are mainly used when setting priority to smallest or the largest node in the tree as we can extract these node very efficiently using heaps.

? Assessment

1. A learner is required to undertake background reading, which is supported by lectures to explain various notions and to show the application of various techniques using examples. The coursework requires the students to solve exercises each week. Feedback for and help with this work is provided in the examples classes.

2.4: Unit Summary

In this unit we presented the searching algorithm and two advanced data structures, the binary search tree and the heap. Two searching algorithms were presented, the sequential search and binary search. The binary search algorithm is generally more efficient than sequential search, although it requires that the input collection be sorted.

The binary search tree operations were presented. The binary search tree data structure is of practical interest due to the efficiency of its basic operations including search and insert that make it attractive for implementing dictionaries and priority queues.

The heap data structure was categorised into two main types, max heap and min heap. For each heap type the operations for building a heap from an array were presented. Heaps are generally useful in priority setting applications through exploiting smallest or largest node in the tree as we can extract these node very efficiently using heaps.

Unit Assessment

Check your understanding!

General Programming Exercises

1. Suppose you are given a sorted array with possible duplicate elements within it. Write a program that finds the last occurrence of a given input item (assuming the given item exists in the given array).
2. Write a program that reads strings (words) as command line arguments and inserts them into a binary search tree and outputs them in in-order fashion. (You should be able to use your program with files by running your program with the standard input redirected from a file)
3. Create a binary heap with a limited heap size. That is, the heap should keep track of a fixed number of most important items (say n). If the heap grows in size to more than n items the least important item should be dropped.

Answers

mailto:njulumi@gmail.com

Grading Scheme

As per the offering Institution grading policy.

Unit Readings and Other Resources

The readings in this unit are to be found at course level readings and other resources.

CHAPTER OVERVIEW

3: Sorting Algorithms

Unit Objectives

Upon completion of this unit you should be able to:

- Demonstrate understanding of various sorting algorithm
- Select best sorting method based on performance requirement
- Apply sorting algorithms in problem solving

Unit Introduction

A sorting algorithm is an algorithm that puts elements of a collection such as array or list in a certain order. The most used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms, such as in search and merge algorithms that require sorted collections to work correctly. It is also often useful for normalization of data and for producing human-readable output. More precisely, the output must satisfy two conditions (i) the output is in non-decreasing order (each element is no smaller than the previous element according to the desired total order) and (ii) the output is a permutation, or reordering, of the input.

The Sorting algorithms used in computer science are often classified by:

1. Computational complexity (worst, average and best case behavior) in terms of the size of the list (n) - For typical sorting algorithms good behavior is O(n log n) and bad behavior is O(n2). Ideal behavior for a sort is O(n). Sort algorithms which only use an abstract key comparison operation often require at least O(n log n) comparisons on average.
2. Memory usage (and use of other computer resources) - In particular, some sorting algorithms are “in place”, such that little memory is needed beyond the items being sorted, while others need to create auxiliary locations for data to be temporarily stored.
3. Stability - Stable sorting algorithms maintain the relative order of records with equal keys (i.e. values). That is, a sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list.
4. Whether or not is a comparison sort - A comparison sort examines the data only by comparing two elements with a comparison operator.
5. General sort method – whether uses insertion, exchange, selection, merging, etc.

Table 3.1 provides a summary of some sorting algorithms and their classification. When equal elements are indistinguishable, such as with integers, stability is not an issue. However, assume that the following pairs of numbers are to be sorted by their first coordinate: (4, 1) (3, 1) (3, 7) (5, 6). In this case, two different results are possible, that is, one which maintains the relative order of records with equal keys, and one which does not as shown below:

(3, 1) (3, 7) (4, 1) (5, 6) (order maintained)

(3, 7) (3, 1) (4, 1) (5, 6) (order changed)

Unstable sorting algorithms may change the relative order of records with equal keys, but stable sorting algorithms never do so. Unstable sorting algorithms can be specially implemented to be stable. One way of doing this is to artificially extend the key comparison, so that comparisons between two objects with otherwise equal keys are decided using the order of the entries in the original data order as a tie-breaker. Remembering this order, however, often involves an additional space cost.

Table 3.1 , presents a classification summary of the sorting algorithms considered in this unit. In the table, n is the number of records to be sorted. The columns “Best”, “Average”, and “Worst” give the time complexity in each case. “Memory” denotes the amount of auxiliary storage needed beyond that used by the collection list itself. “Cmp” indicates whether the sort is a comparison sort.

Table 3.1 : A Sorting Algorithms Classification

Name	Best	Average	Worst	Memory	Stable	Cmp	Method
Selection sort	O(n2)	O(n2)	O(n2)	O(1)	No	Yes	Selection
Insertion sort	O(n)	—	O(n2)	O(1)	Yes	Yes	Insertion
Binary tree sort	O(nlog(n))	—	O(nlog(n))	O(1)	Yes	Yes	Insertion
Merge sort	O(nlog(n))	—	O(nlog(n))	O(n)	Yes	Yes	Merging
Heap sort	O(nlog(n))	—	O(nlog(n))	O(1)	No	Yes	Selection
Quick sort	O(nlog(n))	O(nlog(n))	O(n2)	O(log n)	No	Yes	Partitioning

3.1: Activity 1 - Sorting Algorithms by insertion method

This activity will present two insertion based sorting algorithms, the insertion sort and the binary tree sort algorithms.

Activity Details

Insertion sort Algorithm

The insertion sort is good for sorting moderately sized arrays (up to some tens of elements). For larger arrays, we will see later a much faster algorithm. The basic idea of insertion sort is as follows: Assume that at the start of iteration i of the algorithm, the first i elements of the array are already sorted. The array therefore consists of a left part which is sorted and a right part which is unsorted, as shown in Figure 3.1.1 .

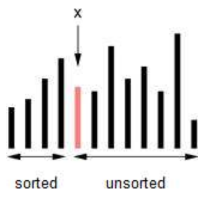


Figure 3.1.1 : Insertion sort progress

Next the algorithm takes the (i+1)-th element x and inserts it into the appropriate position of the sorted part to the left of x by shifting the elements larger than x one position to the right. Thus, we have that the first i+1 element sorted. After n iterations (where n is the length of the array), the whole array is sorted. The insertion sort algorithm is presented below:

```
insertionSort(int[] a)
    {   int n = a.length;
        for (int i = 1; i < n; i++)
        {   // a is sorted from 0 to i-1
            int x = a[i];
            int j = i-1;
            while (j >= 0 && a[j] > x)
            {   a[j+1] = a[j];
                j = j-1; }
            a[j+1] = x; }
```

The algorithm uses the variable x as a temporary buffer to hold the value of a[i] while the larger elements in the sorted part of the array are shifted one position to the right. When all elements have been shifted, this value is written back into the appropriate position of the array.

In the first iteration of the outer loop, the algorithm performs at most one comparison operation, in the second iteration at most two, and so on. The worst case occurs when the input array is sorted in descending order, in which we have that

$$1+2+...+(n-1) = n*(n-1)/2 = n2/2$$

comparisons (the number of element swap operations is approximately the same). Therefore, the time complexity of the algorithm is quadratic in the length n of the array. In average, however, in each iteration of the outer loop only half of the maximum number of comparisons has to be performed until the appropriate insertion point is found. Why? In this case the algorithm performs an average of approximately n2/4 steps.

Sorting an array by this algorithm is therefore much slower than searching for an element in an array (which takes at most n steps, if the array is unsorted, and at most log2 n steps, if the array is sorted). For instance, if the size of the array is 1024, approximately 262000 comparison operations have to be performed to sort the array. If the array has 8192 elements, approximately 16 million comparisons have to be performed.

However, for data that are already sorted or almost sorted, the insertion sort does much better. When data is in order, the inner loop is never executed. Why? In this case, the algorithm needs only n steps. If the data is almost sorted, the inner loop is only very infrequently executed and the algorithm runs in “almost” n steps. It is therefore a simple and efficient way to order a collection that is only slightly out of order.

Binary Tree Sort Algorithm

Given an array of elements to sort, the tree sort algorithms proceeds as follows:

i. Build a binary search tree out of the elements
ii. Traverse the tree in order and copy the elements back into the array.

This algorithm time complexity is as follows:

Worst case - Occurs when the binary search tree is unbalanced, for example when the data is already sorted. Building the binary search tree takes O(n2), and to traverse the binary tree takes O(n) time. The total is O(n2) + O(n) = O(n2).

Best case - Building the binary search tree takes O(n log n) and to traverse the binary tree takes O(n). The total is O(n log n) + O(n) = O(n log n).

Conclusion

This activity presented two sorting algorithms based on the insertion method. Both algorithms' performance were quadratic in the worst case. Such algorithms may be suited to sorting small data collections.

? Assessment

1. Write one to three sentences to answer each of the following questions:
   a. Insertion sort is better for nearly-sorted lists than reverse-ordered lists. Why?
   b. How much extra memory is required for each sort?
   c. If the list is composed of four identical elements, how many elements change position when insertion sort is used? Why?
2. For each of the following lists, draw the result of each insertion of the insertion sorting routine. You do not need to show the result of each comparison, just the final insertion of the element.
   a. [5 | 4 | 1]
   b. [3 | 1 | 3]
   c. [2 | 5 | 4 | 0]
3. Consider the following unsorted list: [6 | 8 | 3 | 5 | 1 | 9 | 2 | 2]. Show the steps used by the tree sort algorithm to obtain a sorted list.

3.2: Activity 2 - Sorting Algorithms by selection method

Introduction

This activity will present two selection based sorting algorithms, the selection sort and the heap sort algorithms.

Activity Details

Selection sort

The selection sort algorithm works by repeatedly exchanging elements: first finds the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted. Below we present the selection sort algorithm:

SELECTION_SORT (A)

```
for i ← 1 to n-1 do
    min j ← i;
    min x ← A[i]
    for j ← i + 1 to n do
        If A[j] < min x then
            min j ← j
            min x ← A[j]
    A[min j] ← A [i]
    A[i] ← min x
```

Selection sort is among the simplest of sorting techniques and it work very well for small files. Furthermore, despite its evident “simplistic approach”, selection sort has a quite important application because each item is actually moved at most once, thus selection sort is a method of choice for sorting files with very large objects (records) and small keys. Below we present the selection sort implementation:

```
void selectionSort(int numbers[], int array_size)
{   int i, j;
    int min, temp;
    for (i = 0; i < array_size-1; i++)
    {   min = i;
        for (j = i+1; j < array_size; j++)
        { if (numbers[j] < numbers[min])
            min = j; }
        temp = numbers[i];
        numbers[i] = numbers[min];
        numbers[min] = temp; }
```

The complexity of the selection sort algorithm is as follows. The worst case occurs if the array is already sorted in descending order. Nonetheless, the time required by selection sort algorithm is not very sensitive to the original order of the array to be sorted: the test “if A[j] < min x” is executed exactly the same number of times in every case. The variation in time is only due to the number of times the “then” part (i.e., min j ← j; min x ← A[j] of this test are executed. Selection sort is quadratic in both the worst and the average case, and requires no extra memory.

Note that for each i from 1 to n - 1, there is one exchange and n - i comparisons, so there is a total of n -1 exchanges and (n -1) + (n -2) + . . . + 2 + 1 = n(n -1)/2 comparisons. These observations hold no matter what the input data is. In the worst case, this could be quadratic, but in the average case, this quantity is O(n log n). It implies that the running time of selection sort does not depend on how the input may be organized.

Heaps Sort

We can use heap for sorting a collection in a specific order efficiently. Suppose we want to sort elements of array Arr in ascending order. We can use max heap to perform this operation through the following steps:

1. Build a max heap of elements in Arr.
2. Now the root element, Arr[1], contains maximum element of Arr. Exchange this element with the last element of Arr and again build a max heap excluding the last element which is already in its correct position. Thus decreases the length of heap by one.
3. Repeat step (ii) until all elements are in their correct position.
4. Get a sorted array.

In the following the implementation of heap sort algorithm is provided (Assuming there are N elements stored in array Arr):

```
void heap_sort(int Ar[ ])
{   int heap_size = N;
    build_maxheap(Arr);
    for(int i = N; i>=2 ; i-- )
    {   swap|(Arr[ 1 ], Arr[ i ]);
        heap_size = heap_size-1;
        max_heapify(Arr, 1, heap_size); }
```

The time complexity of the heap-sort algorithm is as follows: Recall from Unit 2 that max_ heapify has complexity O(logN), build_maxheap has complexity O(N) and the fact that the algorithm calls max_heapify N-1 times, therefore the complexity of heap_sort function is O(N logN).

We now demonstrate the steps of the heap sort algorithm using an example of an unsorted array Arr having 6 elements and corresponding max-heap as in Figure 3.2.1 .

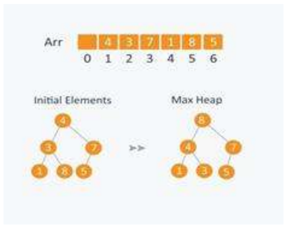


Figure 3.2.1 : An initial array Arr with corresponding max-heap

After building the corresponding max-heap, the elements in the array Arr are now as in Figure 3.2.2 .

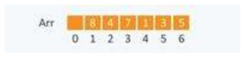


Figure 3.2.2 : An array Arr obtained from max-heap

The subsequent algorithm processing steps are outlined below and further depicted in Figure 3.2.3 .

1. 8 is swapped with 5.
2. 8 is disconnected from heap as 8 is in correct position now and.
3. Max-heap is created and 7 is swapped with 3.
4. 7 is disconnected from heap.
5. Max heap is created and 5 is swapped with 1.
6. 5 is disconnected from heap.
7. Max heap is created and 4 is swapped with 3.
8. 4 is disconnected from heap.
9. Max heap is created and 3 is swapped with 1.

10. 3 is disconnected.

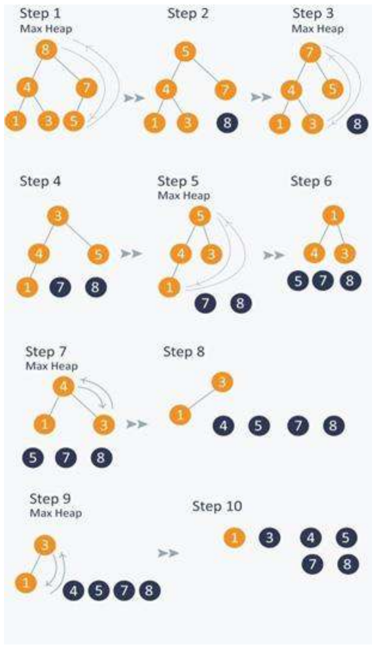


Figure 3.2.3 : Heap-sort algorithm steps

Upon completion of all the steps, we obtain a sorted array shown in Figure 3.2.4 .

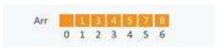


Figure 3.2.4 : A final sorted array

Conclusion

In this activity the selection based sorting algorithms were presented, the selection sort and the heap sort. The complexity analysis for the two algorithms shows that the heap sort is more efficient, since the selection sort has a quadratic worst case performance. The selection sort is suited to sorting small collections, and the order of input does not influence its performance.

? Assessment

1. Show the steps taken when sorting the following list of integers [80, 30, 50, 70, 60, 90, 20, 30, 40] using
   a. Selection sort

3.3: Activity 3 - Sorting Algorithms by Merge and Partition methods

Introduction

This activity will present two sorting algorithms. The first algorithm called merge sort is based on merge method, and the second algorithm called quick sort is based on partition method.

Activity Details

Merge Sort Algorithm

The Merge sort algorithm is based on the divide-and-conquer strategy. Its worst-case running time has a lower order of growth than insertion sort. Since the algorithm deals with sub-problems, we state each sub-problem as sorting a sub-array A[p .. r]. Initially, p = 1 and r = n, but these values change as recursion proceeds through sub-problems.

To sort A[p .. r], the merge sort algorithm proceeds as follows:

1. Divide step - If a given array A has zero or one element, simply return as it is already sorted. Otherwise, split A[p .. r] into two sub-arrays A[p .. q] and A[q + 1 .. r], each containing about half of the elements of A[p .. r].

   That is, q is the halfway point of A[p .. r].

2. Conquer step - Conquer by recursively sorting the two sub-arrays A[p .. q] and A[q + 1 .. r].
3. Combine step - Combine the elements back in A[p .. r] by merging the two sorted sub-arrays A[p .. q] and A[q + 1 .. r] into a sorted sequence. This step will be accomplished by procedure MERGE (A, p, q, r).

Note that the recursion ends when the sub-array has just one element, so that it is trivially sorted (base case). The following are the description of the merge sort algorithm: To sort the entire sequence A[1 .. n], make the initial call to the procedure MERGE-SORT (A, 1, n).

MERGE-SORT (A, p, r)

```
1. IF p < r // Check for base case
2. THEN q = FLOOR[(p + r)/2] // Divide step
3. MERGE (A, p, q) // Conquer step
4. MERGE (A, q + 1, r) // Conquer step
5. MERGE (A, p, q, r) // Conquer step
```

Figure 3.3.1 depicts a bottom-up view of the merge sort procedure for n = 8.

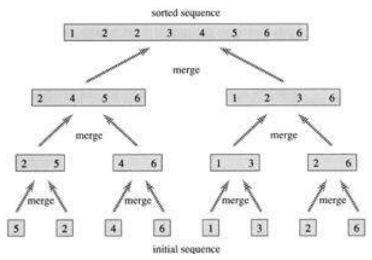


Figure 3.3.1 : merge sort for n=8

What remains is the MERGE procedure. The following is the input and output of the MERGE procedure.

INPUT - Array A and indices p, q, r such that p ≤ q ≤ r and sub-array A[p .. q] is sorted and sub-array A[q + 1 .. r] is sorted. By restrictions on p, q, r, neither sub-array is empty.

OUTPUT- The two sub-arrays are merged into a single sorted sub-array in A[p .. r].

The merge procedure takes Θ(n) time, where n = r − p + 1, which is the number of elements being merged. The idea behind linear time merging can be easily understood by thinking of two piles of cards; each pile is sorted and placed face-up on a table with the smallest cards on top. Merge these into a single sorted pile, face-down on the table, by completing the following basic steps:

1. Choose the smaller of the two top cards.
2. Remove it from its pile, thereby exposing a new top card.

3. Place the chosen card face-down onto the output pile.
4. Repeat steps 1 - 3 until one input pile is empty.
5. Eventually, take the remaining input pile and place it face-down onto the output pile.

Each basic step takes constant time, i.e., O(1), since it involves checking the two top cards. There are at most n basic steps, since each basic step removes one card from the input piles, and we started with n cards in the input piles. Therefore, this procedure should take Θ(n) time.

Now the question is do we actually need to check whether a pile is empty before each basic step?

The answer is no, we do not. Put on the bottom of each input pile a special guard card. It contains a special value that we use to simplify the code. We use ∞, since that's guaranteed to drop to any other value. The only way that ∞ cannot drop is when both piles have ∞ exposed as their top cards. But when that happens, all the non-guard cards have already been placed into the output pile. We know in advance that there are exactly r − p + 1 non-guard cards so stop once we have performed r − p + 1 basic steps. There is no need to check for guards, since they will always drop. Rather than even counting basic steps, just fill up the output array from index p up through and including index r.

The pseudocode of the MERGE procedure is as follows:

MERGE (A, p, q, r)

```
1.  n1 ← q − p + 1
2.  n2 ← r − q
3.  Create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
4.  FOR i ← 1 TO n1
5.  DO L[i] ← A[p + i − 1]
6.  FOR j ← 1 TO n2
7.  DO R[j] ← A[q + j ]
8.  L[n1 + 1] ← ∞
9.  R[n2 + 1] ← ∞
10. i ← 1
11. j ← 1
12. FOR k ← p TO r
13. DO IF L[i ] ≤ R[ j]
14. THEN A[k] ← L[i]
15. i ← i + 1
16. ELSE A[k] ← R[j]
17. j ← j + 1
```

The time complexity of the merge procedure is as follows: The first two for loops (that is, the loop in line 4 and the loop in line 6) take Θ(n1 + n2) = Θ(n) time. The last for loop (that is, the loop in line 12) makes n iterations, each taking constant time, for Θ(n) times. Therefore, the total running time is Θ(n).

The analysis of the merge sort algorithm is as follows: For convenience assume that n is a power of 2 so that each divide step yields two sub-problems, both of size exactly n/2. The base case occurs when n = 1.

When n ≥ 2, time for merge sort steps are:

- Divide - Just compute q as the average of p and r, which takes constant time, i.e., Θ(1).
- Conquer- Recursively solve 2 sub-problems, each of size n/2, which is 2T(n/2).
- Combine- MERGE on an n-element sub-array takes Θ(n) time.

Summed together they give a function that is linear in n, which is Θ(n). Therefore, the recurrence for merge sort running time is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

Solving the merge sort recurrence we can show that T(n) = Θ(n lg n).

The implementation of the merge sort algorithm proceeds as follows:

```
void mergeSort(int numbers[], int temp[], int array_size)
{   m_sort(numbers, temp, 0, array_size - 1);   }

void m_sort(int numbers[], int temp[], int left, int right)
{   int mid;
    if (right > left)
    {   mid = (right + left) / 2;
        m_sort(numbers, temp, left, mid);
    m_sort(numbers, temp, mid+1, right);
    merge(numbers, temp, left, mid+1, right); }

void merge(int numbers[], int temp[], int left, int mid, int right)
{   int i, left_end, num_elements, tmp_pos;
    left_end = mid - 1;
    tmp_pos = left;
    num_elements = right - left + 1;
while ((left <= left_end) && (mid <= right))
{   if (numbers[left] <= numbers[mid])
    {   temp[tmp_pos] = numbers[left];
        tmp_pos = tmp_pos + 1;
        left = left +1; }
    else
    {   temp[tmp_pos] = numbers[mid];
        tmp_pos = tmp_pos + 1;
        mid = mid + 1; }
while (left <= left_end)
    {   temp[tmp_pos] = numbers[left];
        left = left + 1;
        tmp_pos = tmp_pos + 1; }
    while (mid <= right)
    {   temp[tmp_pos] = numbers[mid];
        mid = mid + 1;
        tmp_pos = tmp_pos + 1;}
    for (i = 0; i <= num_elements; i++)
    {   numbers[right] = temp[right];
        right = right - 1;}
```

Quick Sort Algorithm

The quick sort algorithm operates on the principle of divide-and-conquer. Quick sort is an algorithm of choice in many situations because it is not difficult to implement, it is a good "general purpose" sort and it consumes relatively fewer resources during execution. The following are some pros and cons of the quick sort algorithm:

Pros

- well established and uses only a small auxiliary stack

- requires only n log(n) time to sort n items.
- has an extremely short inner loop

Cons

- is recursive. Especially if recursion is not available, the implementation is extremely complicated.
- is fragile i.e., a simple mistake in the implementation can go unnoticed and cause it to perform badly.

The quick sort algorithm works by partitioning a given array A[p . . r] into two non-empty sub array A[p . . q] and A[q+1 . . r] such that every key in A[p . . q] is less than or equal to every key in A[q+1 . . r]. Then the two sub-arrays are sorted by recursive calls to quick sort. The exact position of the partition depends on the given array and index q is computed as a part of the partitioning procedure. In general, the quick sort algorithm performs the following steps:

Quick_Sort

1. If p < r then
2. q Partition (A, p, r)
3. Recursive call to Quick Sort (A, p, q)
4. Recursive call to Quick Sort (A, q + r, r)

Note that to sort the entire array, initially call Quick_Sort(A, 1, length[A]).

As a first step, the quick sort algorithm would choose as a pivot an element from one of the items in the array to be sorted. Subsequently, the array is partitioned on either side of the pivot. Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

The partition procedure is defined below:

PARTITION (A, p, r)

```
x ← A[p]
i ← p-1
j ← r+1
while TRUE do
Repeat j ← j-1
until A[j] ≤ x
Repeat i ← i+1
until A[i] ≥ x
if i < j
then exchange A[i] ← A[j]
else return j
```

The partition procedure selects the first key, A[p] as a pivot key about which the array will be partitioned:

- Keys ≤ A[p] will be moved towards the left
- Keys ≥ A[p] will be moved towards the right

The time complexity of the partition procedure is $\Theta(n)$ where $n = r - p + 1$ which is the number of keys in the array.

The time complexity of quick sort algorithm depends on whether partition is balanced or unbalanced, which in turn depends on which elements of an array to be sorted are used for partitioning. A very good partition splits an array up into two equal sized arrays. A bad partition, on other hand, splits an array up into two arrays of very different sizes. The worst partition puts only one element in one array and all other elements in the other array. If the partitioning is balanced, the quick sort runs asymptotically as fast as merge sort. On the other hand, if partitioning is unbalanced, the quick sort runs asymptotically as slow as insertion sort.

In the best case each partitioning stage divides the array exactly in half. In other words, the best to be a median of the keys in A[p . . r] every time procedure PARTITION is called. The procedure PARTITION always split the array to be sorted into two equal sized arrays. If the procedure PARTITION produces two regions of size n/2, the recurrence relation is then:

$$\begin{aligned} T(n) &= T(n/2) + T(n/2) + \Theta(n) \\ &= 2T(n/2) + \Theta(n) \end{aligned}$$

And

$$T(n) = \Theta(n \lg n).$$

The worst-case occurs if the input array A[1 . . n] is already sorted. The call to PARTITION (A, p, r) always returns p, so successive calls to partition will split arrays of length n, n-1, n-2, . . . , 2. Hence, the running time is proportional to n + (n-1) + (n-2) + . . . + 2 = [(n+2)(n-1)]/2 = Θ(n2). The worst-case also occurs if A[1 . . n] starts out in reverse order.

The implementation of the quick sort algorithm is presented below:

```
void quickSort(int numbers[], int array_size)
{q_sort(numbers, 0, array_size - 1);
}

void q_sort(int numbers[], int left, int right)
{
    int pivot, l_hold, r_hold;
    l_hold = left;
    r_hold = right;
    pivot = numbers[left];
    while (left < right)
    {
        while ((numbers[right] >= pivot) && (left < right))
            right--;
        if (left != right)
        {
            numbers[left] = numbers[right];
            left++;
        }
        while ((numbers[left] <= pivot) && (left < right))
            left++;
        if (left != right)
        {
            numbers[right] = numbers[left];
            right--;
        }
    }
    numbers[left] = pivot;
    pivot = left;
    left = l_hold;
    right = r_hold;
    if (left < pivot)
        q_sort(numbers, left, pivot-1);
    if (right > pivot)
        q_sort(numbers, pivot+1, right);
```

Conclusion

In this activity we presented two sorting algorithms, the merge sort and quick sort algorithm based on merge and partition approach respectively. Both algorithms use the divide-and-conquer strategy and have comparable run time complexity on average. Quick sort is a well established sorting algorithm whose worst-case running time is Θ(n2) and expected running time is Θ(n lg n) where the constants hidden in Θ(n lg n) are small.

? Assessment

1. Why would in practice one use quick sort algorithm than merge sort algorithm even though, merge sort promises a run time complexity of (n lg n) for both worst and average cases?
2. Show the steps taken when sorting the following list of integers [80, 30, 50, 70, 60, 90, 20, 30, 40] using
   a. merge sort
   b. quick sort

3.4: Unit Summary

This unit presented six sorting algorithms based on insertion, selection, merge and partition methods. The insertion sort based algorithms, the insertion sort and binary tree sort have quadratic performance in the worst case, thus may be suited to sorting small data collections. The selection based sorting algorithms, the selection sort and the heap sort, give slight improvement in run time efficiency, though, generally also suffer from the quadratic worst case performance.

The merge sort and quick sort algorithm are based on merge and partition approach respectively, and both algorithms use the divide-and-conquer strategy. They have comparable run time complexity on average of Θ(n lg n). Quick sort is a well established sorting algorithm whose worst-case running time is Θ(n2) and expected running time is Θ(n lg n) where the constants hidden in Θ(n lg n) are small.

? Unit Assessment

Check your understanding!

Programming Exercises

Instructions

1. A merge of two sorted lists, e.g. merge([1;4;9;12], [2;3;4;5;10;13]) leads to a new sorted list [1;2;3;4;4;5;9;10;12;13], made up from the elements of the arguments in the merge function. This operation can be declared so that it has a worst-case running time proportional to the sum of the length of the argument lists. Declare such a function. Test this function so that you are sure that all branches of the declaration work correctly.

Answers

mailto:njulumi@gmail.com

Grading Scheme

As per the offering Institution grading policy.

Unit Readings and Other Resources

The readings in this unit are to be found at course level readings and other resources.

CHAPTER OVERVIEW

4: Hash Tables, Graphs and Graph Algorithms

Unit Objectives

Upon completion of this unit you should be able to:

- Demonstrate understanding of hash tables and graph data structures
- Apply the data structures in problem solving
- Demonstrate understanding of graph algorithms

Unit Introduction

In this unit we will present more advanced data structures, hash tables and graphs and explore some graph algorithms. The hash table data structure uses hash functions to generate indexes where keys are stored or being accessed in an array. They find a wide use including implementation of associative arrays, database indexes, caches and object representation in programming languages.

A graph data structure is represented as a collection of finite sets of vertices or nodes (V) and edges (E). Edges are ordered pairs such as (u, v) where (u, v) indicates that there is an edge from vertex u to vertex v. Edges may contain cost, weight or length. Graphs are widely used in modeling networks including computer networks, social networks or some abstract phenomena of interrelated conceptual objects. Graphs algorithm can be used to solve most problems occurring in situations modeled as networks.

4.1: Activity 1 - Hash Tables

Introduction

To store a key/value pair we can use a simple array like data structure where keys directly can be used as index to store values. But in case when keys are large and can't be directly used as index to store value, we can use a technique of hashing. In hashing, large keys are converted into small ones, by using hash functions and then the values are stored in data structures called hash tables.

The goal of hashing is to distribute the entries (key / value pairs) across an array uniformly. Each element is assigned a key (converted one) and using that key we can access the element in O(1) time. Given a key, the algorithm (hash function) computes an index that suggests where the entry can be found or inserted. Hashing is often implemented in two steps:

- The element is converted into an integer, using a hash function, and can be used as an index to store original element, which falls into the hash table.
- The element is stored in the hash table, where it can be quickly retrieved using a hashed key. That is,

$$\text{hash} = \text{hashfunc(key)}$$

$$\text{index} = \text{hash} \ \% \ \text{array\_size}$$

In this method, the hash is independent of the array size, and it is then reduced to an index (a number between 0 and array_size − 1), using modulo operator (%).

A hash table data structure attracts a wide use including:

- Associative arrays - Hash tables are commonly used to implement many types of in-memory tables. They are used to implement associative arrays (arrays whose indices are arbitrary strings or other complicated objects).
- Database indexing - Hash tables may also be used as disk-based data structures and database indices (such as in dbm).
- Caches - Hash tables can be used to implement caches, auxiliary data tables that are used to speed up the access to data that is primarily stored in slower media.
- Object representation - Several dynamic languages, such as Perl, Python, JavaScript, and Ruby, use hash tables to implement objects.

Hash functions are used in various algorithms to make their computing faster.

Activity Details

A hash function is any function that can be used to map dataset of arbitrary size to dataset of fixed size which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes. In order to achieve a good hashing mechanism, it is essential to have a good hash function. A good hash function has three basic characteristics:

1. Easy to compute – A hash function should be easy to compute. It should not become an algorithm in itself.
2. Uniform distribution – It should provide a uniform distribution across the hash table and should not result in clustering.
3. Less collisions – Collision occurs when pairs of elements are mapped to the same hash value. It should avoid collisions as far as possible.

Note that no matter how good a hash function may be, collisions are bound to occur. So, they need to be managed through various collision resolution techniques in order to maintain high performance of the hash table. To understand the need for good hash functions consider the following example: Suppose it is required to store strings in the hash table using hashing {"abcdef", "bcdefa", "cdefab" , "defabc" }. To compute the index for storing the strings, the hash function that states "the index for a particular string will be equal to the sum of the ascii values of the characters modulo 599" should be used. As 599 is prime number, it will reduce the possibility of same index for different strings (collisions). It is thus recommended to use prime number when using modulo.

As we know ascii value of a = 97, b = 98, c= 99, d = 100, e = 101, f = 102. As all the strings contain same characters with different permutations, so the sum for each will be the same, that is, 599. So the hash function will compute same index for all the strings, and strings will be stored in the hash table in the format given in Figure 4.1.1 . As the index of all the strings is same, so we can create a list on that particular index and can insert all the strings in that list. So to access a particular string, it will take O(n) time where n is the number of strings, which shows that the hash function is not a good hash function.

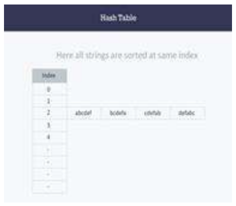


Figure 4.1.1 : A hash table with strings on same index

Table 4.1.1 : String-index mapping for updated hash function

String	Hash function	Index
abcdef	(971 + 982 + 993 + 1004 + 1015 + 1026)%2069	38
Bcdefa	(981 + 992 + 1003 + 1014 + 1025 + 976)%2069	23
Cdefab	(991 + 1002 + 1013 + 1024 + 975 + 986)%2069	14
Defabc	(1001 + 1012 + 1023 + 974 + 985 + 996)%2069	11

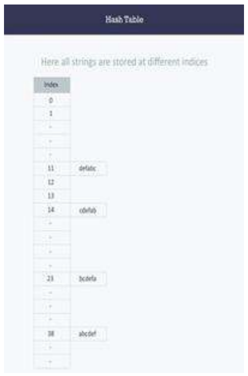


Figure 4.1.2 : A hash table resulting from updated hash function

A hash table is a data structure used to store key / value pairs. Hash tables use a hash function to compute an index into an array where the element will be inserted or searched. By using a good hash function, hashing can perform extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is O(1).

Consider a string S defined below:

string S = “ababcd”

Suppose we need to count the frequency of all the characters in the string S. The simplest thing to do would be to iterate over all the possible characters and count their frequency one by one. The time complexity of this approach is **O(26*N)** where **N** is the size of the string and there are 26 possible characters (in English alphabet). The function defined below performs this task.

```
void countFre(string S)
{ for(char c = ‘a’;c <= ‘z’;++c)
    {   int frequency = 0;
        for(int i = 0;i < S.length();++i)
            if(S[i] == c)
                frequency++;
cout << c << ‘ ‘ << frequency << endl; }
```

The output of the function above for input S is: a(2), b(2), c(1), d(1), e(0), f(0), …, z(0), where the number in brackets denotes the frequency.

To apply hashing to this problem, we define an array ‘Frequency’ of size 26 and hash the 26 characters with indices of the array using the hash function. Subsequently, iterate over the string and for each character, and increase the value in ‘Frequency’ at the corresponding index. The complexity of this approach is **O(N)** where **N** is the size of the string. Below we define a function that uses this method.

```
int Frequency[26];
int hashFunc(char c)
{   return (c - ‘a’);    }
void countFre(string S)
{   for(int i = 0;i < S.length();++i)
    {   int index = hashFunc(S[i]);
        Frequency[index]++;    }
    for(int i = 0;i < 26;++i)
        cout << (char)(i+’a’) << ‘ ‘ << Frequency[i] << endl;}
```

The output of this function for the input S is a(2), b(2), c(1), d(1), e(0), f(0), …, z(0), where the number in brackets denotes frequency.

We define a load factor as the number of entries divided by the size of the hash table, i.e., n / k where n is the number of entries and k is the size of the hash table. If the load factor is kept reasonable, the hash table will perform well, if the hash function used is good. If the load factor grows too large, the hash table will become slow, or it may fail to work (depending on the hash function used). The expected constant time property of a hash table assumes that the load factor is kept below some bound. A low load factor is not especially beneficial, since as the load factor approaches 0, the proportion of unused areas in the hash table increases. This will result in wasted memory.

Collision Resolution Techniques

Separate chaining (Open hashing) - Separate chaining is one of the most common and widely used collision resolution techniques. It is usually implemented using linked lists. In separate chaining, each element of the hash table is a linked list. To store an element in the hash table, simply insert into a particular linked list. If there is any collision, i.e., two different elements have same hash value, then store both elements in the same linked list. Figure 4.1.3 depicts a hash table with separate chaining resolution.

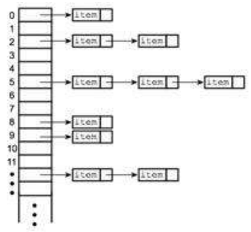


Figure 4.1.3 : Separate chaining

The cost of a lookup is that of scanning the entries of the selected linked list for the desired key. If the distribution of keys is sufficiently uniform, the average cost of a lookup depends only on the average number of keys per linked list, i.e., it is roughly proportional to the load factor. For this reason, chained hash tables remain effective even when the number of table entries n is much higher than the number of slots.

For separate-chaining, the worst-case scenario is when all entries are inserted into the same linked list. The lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number n of entries in the table.

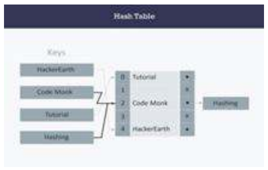


Figure 4.1.4 : Hash table with separate chaining

In Figure 4.1.4 , the keys Code Monk and Hashing both hash to the value 2. But, the linked list at index 2 can hold only one entry, thus the next entry, in this case Hashing is linked (attached) to the entry of Code Monk. The implementation of hash table with separate chaining is presented below with an assumption that the hash function will return an integer from 0 to 19.

```
vector <string> hashTable[20];
int hashTableSize=20;
//insertion
void insert(string s)
{             // Compute the index using Hash Function
    int index = hashFunc(s);
    // Insert the element in the linked list at the particular index
    hashTable[index].push_back(s); }
//searching
void search(string s)
{   // Compute the index using Hash Function
    int index = hashFunc(s);
    // Search the linked list at that particular index
    for(int i = 0;i < hashTable[index].size();i++)
    {    if(hashTable[index][i] == s)
        {    cout << s << " is found!" << endl;
        return;    }
    cout << s << " is not found!" << endl; }
```

Linear probing (Open addressing or Closed hashing) - In open addressing, all entry records are stored in the array itself, instead of linked lists. When a new entry need to be inserted, a hashed index of hash value is computed and the array is examined, starting with the hashed index. If the slot at hashed index is unoccupied, the entry record is inserted in slot at hashed index; otherwise, proceed with a probe sequence, until an unoccupied slot is found.

A probe sequence is the sequence followed while traversing through the entries. There may be different interval between successive entry slots or probes in different probe sequence. When searching for an entry, the array is scanned in the same sequence, until either the target element is found, or an unused slot is found, which indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location ("address") of the item is not determined by its hash value. Linear probing is when the interval between successive probes is fixed (usually to 1).

Suppose the hashed index for a particular entry is index. A probing sequence for linear probing is given by

index = index % hashTableSize

index = (index + 1) % hashTableSize

index = (index + 2) % hashTableSize

index = (index + 3) % hashTableSize

and so on…

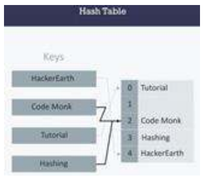


Figure 4.1.5 : Hash table with linear probing

Figure 4.1.5 shows an example of hash collision resolved by open addressing with linear probing.

Note that since the keys Code Monk and Hashing are hashed to the same index, i.e., 2, thus the key Hashing is stored at index 3 as the interval between successive probes is 1. The implementation of hash table with linear probing is presented below with an assumption that there are no more than 20 elements in the data set and the hash function return an integer from 0 to 19, and that data set have unique elements.

```
string hashTable[21];
int hashTableSize = 21;
//insertion
void insert(string s)
{    // Compute the index using the Hash Function
    int index = hashFunc(s);
    // Search for an unused slot and if the index will exceed the hashTableSize
    // we will roll back
    while(hashTable[index] != “”)
        index = (index + 1) % hashTableSize;
    hashTable[index] = s; }
//searching
void search(string s)
{   // Compute the index using the Hash Function
    int index = hashFunc(s);
    // Search for an unused slot and if the index will exceed the hashTableSize
    // we will roll back
    while(hashTable[index] != s and hashTable[index] != “”)
        index = (index + 1) % hashTableSize;
    // Element is present in the Hash Table or not
    if(hashTable[index] == s)
        cout << s << “ is found!” << endl;
    else
        cout << s << “ is not found!” << endl; }
```

Quadratic probing - Quadratic probing is same as linear probing, the only difference being the interval between successive probes or entry slots. When at hashed index for an entry record whose slot is already occupied, start traversing until there is no more unoccupied slot. The interval between slots is computed by adding the successive value of arbitrary polynomial in the original hashed index. Suppose that the hashed index for an entry is index, and the slot at index is occupied, the probe sequence proceeds as follows:

index = index % hashTableSize

index = (index + 12) % hashTableSize

index = (index + 22) % hashTableSize

index = (index + 32) % hashTableSize

and so on…

The implementation of hash table with quadratic probing is given below with assumptions that there are no more than 20 elements in the data set thus hash function will return an integer from 0 to 19, and the data set have unique elements.

```
string hashTable[21];
int hashTableSize = 21;
//insertion
void insert(string s)
{   // Compute the index using the Hash Function
    int index = hashFunc(s);
        // Search for an unused slot and if the index will
        // exceed the hashTableSize we will roll back
    int h = 1;
    while(hashTable[index] != “”)
    {   index = (index + h*h) % hashTableSize;
        h++;    }
    hashTable[index] = s; }
//Searching
void search(string s)
{   // Compute the index using the Hash Function
    int index = hashFunc(s);
    // Search for an unused slot and if the index will exceed the hashTableSize
    // we will roll back
    int h = 1;
    while(hashTable[index] != s and hashTable[index] != “”)
    {   index = (index + h*h) % hashTableSize;
        h++;    }
    // Element is present in the Hash Table or not
    if(hashTable[index] == s)
        cout << s << “ is found!” << endl;
    else
        cout << s << “ is not found!” << endl; }
```

Double hashing - Double hashing is same as linear probing, the only difference being the interval between successive probes. The interval between probes is computed by using two hash functions. That is, suppose that the hashed index for an entry record is index which is computed by one hashing function and at index the slot is already occupied, then we start traversing in a particular probing sequence to look for unoccupied slot as shown below where indexH is the hash value computed by another hash function.

index = (index + 1 * indexH) % hashTableSize;

index = (index + 2 * indexH) % hashTableSize;

and so on….

The implementation of hash table with double hashing is given below with the assumptions that there are no more than 20 elements in the data set thus hash functions will return an integer from 0 to 19, and the data set have unique elements.

```
string hashTable[21];
int hashTableSize = 21;
//insertion
void insert(string s)
{   // Compute the index using the Hash Function1
    int index = hashFunc1(s);
    int indexH = hashFunc2(s);
    // Search for an unused slot and if the index will exceed the hashTableSize
    // we will roll back
    while(hashTable[index] != “”)
        index = (index + indexH) % hashTableSize;
    hashTable[index] = s; }
//searching
void search(string s)
{   // Compute the index using the Hash Function
    int index = hashFunc1(s);
    int indexH = hashFunc2(s);
    // Search for an unused slot and if the index will exceed the hashTableSize
    // we will roll back
    while(hashTable[index] != s and hashTable[index] != “”)
        index = (index + indexH) % hashTableSize;
    // Element is present in the Hash Table or not
    if(hashTable[index] == s)
        cout << s << “ is found!” << endl;
    else
        cout << s << “ is not found!” << endl;}
```

Conclusion

This activity presented the hash table data structure. Hash tables use hash functions to generate indexes for keys to be stored or accessed from array/list. They find a wide use including implementation of associative arrays, database indexes, caches and object representation in programming languages. A good hash function design is instrumental to the hash table performance whose expected access time is O(1). Hash functions suffer from collision problems where two different keys get hashed/mapped to the same index. Collision can be addressed through use of collision resolution techniques, including separate chaining, linear probing, double hashing and quadratic probing.

? Assessment

1. Given a hash table with m=11 entries and the hash function h1 and second hash function h2:

   h1(key) = key mod m

   h2(key) = {key mod (m-1)} + 1

   Insert the keys {22, 1, 13, 11, 24, 33, 18, 42, 31} in the given order, i.e., from left to right, to the hash table using each of the following hash methods:

   a. Separate Chaining
   b. Linear probing
   c. Double hashing

2. Suppose a hash table with capacity m=31 gets to be over ¾ full. We decide to rehash. What is a good size choice for the new table to reduce the load factor below 0.5 and also avoid collisions?

4.2: Activity 2 - Graphs

Introduction

Consider a social network such as facebook which is an enormous network of persons including you, your friends, family, their friends and their friends, etc. This is referred to as a social graph. In this graph, every person is considered as a node of the graph and the edges are the links between two people. In facebook, friendship is a bidirectional relationship. That is if A is B’s friend it implies also that B is A’s friend, thus the associated graph is an undirected graph.

A graph is defined as a collection of finite sets of vertices or nodes (V) and edges (E). Edges are represented as ordered pairs (u, v), where (u, v) indicates that there is an edge from vertex u to vertex v. Edges may contain cost, weight or length. The degree of a vertex is the number of edges that connect to it. A path is a sequence of vertices in which every consecutive vertex pair is connected by an edge and no vertex is repeated in the sequence except possibly the start vertex. There are two types of graphs:

Undirected graph - a graph in which all the edges are bidirectional, essentially the edges don’t point in a specific direction. Figure 4.2.1 depicts an undirected graph on four vertices and four edges.

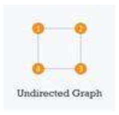


Figure 4.2.1 : An undirected graph

Directed graph (digraph) - a graph in which all the edges are unidirectional. Figure 4.2.2 depicts a digraph on four vertices and five directed edges (arcs).

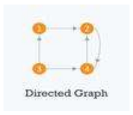


Figure 4.2.2 : A directed graph

A weighted graph has each edge assigned with a weight or cost. Consider a graph of 4 nodes depicted in Figure 4.2.3 . Observe that each edge has a weight/cost assigned to it. Suppose we need to traverse from vertex 1 to vertex 3. There are 3 possible paths: 1 -> 2 -> 3, 1 -> 3 and 1 -> 4 -> 3. The total cost of path 1 -> 2 -> 3 is (1 + 2) = 3 units, the total cost of path 1 -> 3 is 1 units and the total cost of path 1 -> 4 -> 3 is (3 + 2) = 5 units.

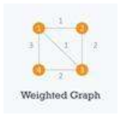


Figure 4.2.3 : A weighted graph

A graph is called cyclic if there is a path in the graph which starts from a vertex and ends at the same vertex, and this path is called a cycle. An acyclic graph is a graph which has no cycle. A tree is an undirected graph in which any two vertices are connected by only one path. A tree is acyclic graph and has N - 1 edges where N is the number of vertices. Figure 4.2.4 shows a tree on 5 nodes.

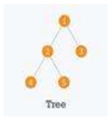
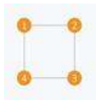


Figure 4.2.4 : A tree

Activity Details

Graph Representation

There is variety of ways to represent a graph, but the two most popular approaches are the adjacency matrix and adjacency list. Adjacency matrix is a V x V binary matrix (i.e., a matrix in which the cells can have only one of two possible values, either a 0 or 1), where an element Ai,j is 1 if there is an edge from vertex i to vertex j else Ai,j is 0. The adjacency matrix can also be modified for the weighted graph in which instead of storing 0 or 1 in Ai,j, store the weight or cost of the edge from vertex i to vertex j. In an undirected graph, if Ai,j = 1 then Aj,i = 1. In a directed graph, if Ai,j = 1 then Aj,i may or may not be 1. Adjacency matrix provide a constant time access, that is, **O(1)** to check whether there is an edge between two nodes. The space complexity of adjacency matrix is **O(V^2)**. Figure 4.2.5 presents a graph and its adjacency matrix.



i/j: **1 2 3 4**

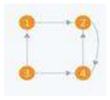
1: 0 1 0 1

2: 1 0 1 0

3: 0 1 0 1

4: 1 0 1 0

Figure 4.2.5 : A four node graph and corresponding adjacency matrix



i/j: **1 2 3 4**

1: 0 1 0 0

2: 0 0 0 1

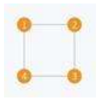
3: 1 0 0 1

4: 0 1 0 0

Figure 4.2.6 : A digraph and corresponding adjacency matrix

Adjacency list represents a graph as an array of separate lists. Each element of the array Ai is a list which contains all the vertices that are adjacent to vertex i. For weighted graph we can store weight or cost of the edge along with the vertex in the list using pairs. In an undirected graph, if vertex j is in list Ai then vertex i will be in list Aj.

The space complexity of adjacency list is **O(V + E)** because in adjacency list we store information for only those edges that exist in the graph. Note that for cases where a matrix is sparse (i.e., most matrix elements entries, by contrast, if most entries are nonzero the matrix is considered dense), using an adjacency matrix might not be very useful, since a lot of space is used, though mostly occupied by 0 entries. In such cases an adjacency list would be a better choice. Figure 4.2.7 presents an undirected graph along with it adjacency list. Similarly, Figure 4.2.8 presents a digraph with its corresponding adjacency list.



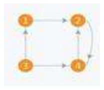
A1→2→4

A2→1→3

A3→2→4

A4→1→3

Figure 4.2.7 : A graph and its adjacency list



A1→2

A2→4

A3→1→4

A4→2

Figure 4.2.8 : A digraph along with its adjacency list

Graph Traversals

Some graph algorithms may require that every vertex of a graph be visited exactly once. The order in which the vertices are visited may be important, and may depend upon a particular algorithm or particular question being solved. During a traversal, we may keep track of which vertices have been visited so far. The most common way is to mark the vertices which have been visited. A graph traversal visits every vertex and every edge exactly once in some well-defined order. There are many approaches to traverse a graph. Two most popular traversal methods are the depth first search (DFS) and breadth first search (BFS).

The DFS traversal is a recursive algorithm that uses the idea of backtracking. Basically, it involves exhaustive searching of all the nodes by going ahead whenever possible, otherwise it backtracks. Backtracking means that when cannot get any further node in the current path, move back to the node from where can find further nodes to traverse. That is, continue visiting nodes as long as unvisited nodes are found along the current path, when current path is completely traversed select a next path.

The recursive nature of DFS can be implemented using stacks. The basic idea is, pick a starting node and push all its adjacent nodes into a stack. Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack. Keep on repeating this process until the stack is empty. DFS do not visit a node more than once, otherwise, might end up in an infinite loop. To avoid an infinite loop, DFS marks the nodes as soon as they are visited. The pseudo-code for DFS is presented below both for an iterative and recursive version. Figure 4.2.9 illustrates the DFS steps on a five node tree.

```
DFS-iterative (G, s):     //where G is graph and s is source vertex.
    let S be stack
    S.push( s )     // inserting s in stack mark s as visited.
    while ( S is not empty):
        // pop a vertex from stack to visit next
        v = S.top( )
        S.pop( )
        //push all the neighbours of v in stack that are not visited
        for all neighbours w of v in Graph G:
            if w is not visited :
                S.push( w )
                mark w as visited
DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:     DFS-recursive(G, w)
```

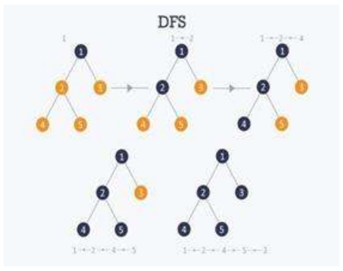


Figure 4.2.9 : A DFS traversal on a tree

A BFS traversal starts from a selected node (source or starting node) and proceeds to traverse the graph layer-wise, that is, explores the neighboring nodes (nodes which are directly connected to source node) and subsequently proceed towards the next level neighbor nodes. Therefore BFS move in breadth of the graph, that is, moves horizontally first and visit all the nodes of the current layer and then to the next layer. Figure 4.2.10 illustrates a BFS traversal.

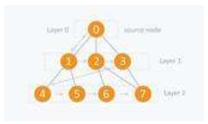


Figure 4.2.10 : A BFS traversal

Note from Figure 4.2.10 that a BFS traversal, visits all nodes on layer 1 before proceeding to nodes of layer 2, since nodes on layer 1 have less distance from source node compared to nodes on layer 2. Since a graph may contain cycles, BFS may possibly visit the same node more than once when traversing the graph. To avoid processing a node more than once, a boolean array may be used to track already processed nodes. While visiting the nodes of current layer of graph, they are stored such that the child nodes are also visited in the same order as the respective parent node.

In Figure 4.2.10 the BFS starts from 0, visit its children 1, 2, and 3 and store them in the order they get visited. Upon visiting all vertices of current layer, BFS proceeds to visit the children of 1 (i.e., 4 and 5), 2 (i.e., 6 and 7) and 3 (i.e., 7) in that order, and so on.

To simply the implementation of the above process, use a queue to store the node and mark it as visited, and should remain in queue until all its neighbors (vertices which are directly connected to it) are marked as visited. Since a queue follows FIFO order (First In First Out), BFS will first visit the neighbors of that node, which were inserted first in the queue. Below a pseudo code for BFS is presented, and Figure 4.2.11 shows a BFS traversal on a six node graph.

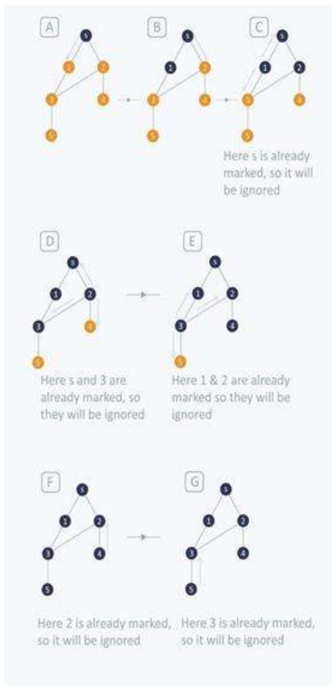
```
//BSF Pseudocode
BFS (G, s)     //where G is graph and s is source node.
    let Q be queue.
    Q.enqueue( s ) // inserting s in queue until all its neighbour vertices are marke
    mark s as visited.
    while ( Q is not empty)
        // removing that vertex from queue, whose neighbour will be visited now.
        v = Q.dequeue( )
        //processing all the neighbours of v
        for all neighbours w of v in Graph G
            if w is not visited
                Q.enqueue( w )     //stores w in Q to further visit its neighbour
                mark w as visited.
```

The BFS traversal on Figure 4.2.11 proceeds as follows: Starting from source node s push in queue and mark it visited. In first iteration, pops s from queue and traverse on neighbors of s, that is 1 and 2. Since 1 and 2 are unvisited, they are pushed in queue and be marked as visited.

In second iteration, pops 1 from queue and traverse its neighbors s and 3. Since s is already marked visited, it is ignored and 3 is pushed in queue and marked as visited.

In third iteration, pops 2 from queue and traverse its neighbors s, 3 and 4. Since 3 and s are already marked visited, they are ignored and 4 is pushed in queue and marked as visited. In fourth iteration, pops 3 from queue and traverse its neighbors 1, 2 and 5. Since 1 and 2 are already marked visited, they are ignored and 5 is pushed in queue and marked as visited.

In fifth iteration, pops 4 from queue and traverse its neighbor, that is, 2. Since 2 is already marked visited it is ignored. In sixth iteration, pops 5 from queue and traverse its neighbors, that is, 3. Since 3 is already marked visited it is ignored. Eventually, the queue is empty so it comes out of the loop. Through this process all the nodes have been traversed by BFS.


A
B
C
Here s is already marked, so it will be ignored
D
E
Here s and 3 are already marked, so they will be ignored
Here 1 & 2 are already marked so they will be ignored
F
G
Here 2 is already marked, so it will be ignored
Here 3 is already marked, so it will be ignored
Figure 4.2.11 : A BFS traversal

The time complexity of BFS is O(V + E), where V is the number of nodes and E is the number of Edges.

Conclusion

This activity presented the graph data structure which is defined by a set of vertices and edges. Graph data structure is very useful in problems which can be modelled as a network. Two graph representation techniques were presented, the adjacency matrix and adjacency list approach. When processing a graph a systematic visitation of its nodes can be facilitated by DFS and BFS traversals.

? Assessment

1. Consider a weighted graph in Figure 4.2.12 , with A being a start vertex.

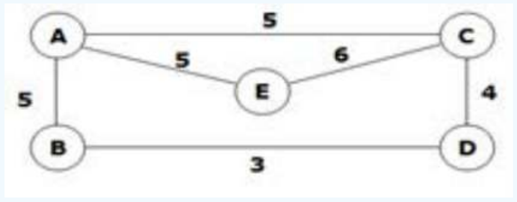


Figure 4.2.12

   a. Define its adjacency matrix
   b. Define its adjacency list
   c. List the vertices using DFS
   d. List the vertices using BFS

4.3: Activity 3 - Graph Algorithms

Introduction

In this activity we will explore two graph algorithms, the topological sort and shortest path algorithms. The topological sort algorithm takes as input a directed acyclic graph and generates a linear sequence of vertices that are totally order by vertex precedence. This algorithm finds wide applications including in projects planning and scheduling, detection of cycles and in creation of software builds with module dependency.

The shortest path problem seeks to find a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized. We consider the Dijkstra algorithm that finds the shortest path for a single source graph with non-negative edge weights, and the Bellman Ford algorithm which works even for graphs with negative edge weights. Furthermore, we consider the Floyd-Warshall algorithm which finds shortest paths in multiple sourced graph setting with arbitrary edge weights. Shortest path algorithms have wide range of applications, both direct and as a sub-problem of other problems. Some of the prominent applications include in vehicle routing and network design and routing.

Activity Details

Topological Sort

A cycle in a digraph G is a set of edges, {(v1, v2), (v2, v3), ..., (vr −1, vr)} where v1 = vr. A digraph is acyclic if it has no cycles. Such a graph is often referred to as a directed acyclic graph, or DAG, for short. DAGs are used in many applications to indicate precedence among events. For example, applications of DAGs include depict/model the following:

- inheritance in object oriented programming classes/interfaces
- prerequisites between courses of a degree program
- scheduling constraints between tasks in a projects

Thus DAG can be fundamental tool for modelling processes and structures that have a partial order: a > b and b > c imply a > c. But may have a and b such that neither a > b or b > a. One can always make a total order (either a > b or b > a for all a ≠ b) from a partial order. In fact, that is the prime goal of a topological sort.

The vertices of a DAG can be ordered in such a way that every edge goes from an earlier vertex to a later vertex. This is called a topological sort or topological ordering. Formally, we say a topological sort of a directed acyclic graph G is an ordering of the vertices of G such that for every edge (vi, vj) of G we have i < j. That is, a topological sort is a linear ordering of all its vertices such that if DAG G contains an edge (vi, vj), then vi appears before vj in the ordering. If DAG is cyclic then no linear ordering is possible. A topological ordering is an ordering such that any directed path in DAG G traverses vertices in increasing order. It is important to note that if the graph is not acyclic, then no linear ordering is possible. That is, circularities should not exist in the directed graph for topological sort to achieve its goal.

Note that a digraph may have several different topological sorts and that topological sort is different from the usual notion of sorting. One way to obtain a topological sort of a DAG is to run a depth-first search (DFS) algorithm on G and order the vertices so that the keys are in descending order. This requires Θ(|V| lg |V|) time if a comparison sort algorithm is used. Thus DFS can be modified for a DAG to produce a topological sort, such that a vertex is visited is pushed on a stack or put on the front of a linked list. In which case, the running time is Θ(|V| + |E|). Consider Figure 4.3.1 . It illustrates an example that arises when a Professor gets dressed in the morning. The DAG of dependencies for putting clothing is given (not shown in the figure). In Figure 4.3.1 (a) the discovery and finishing times from depth-first search are shown next to each vertex. In Figure 4.3.1 (b), the same DAG is shown as topologically sorted.

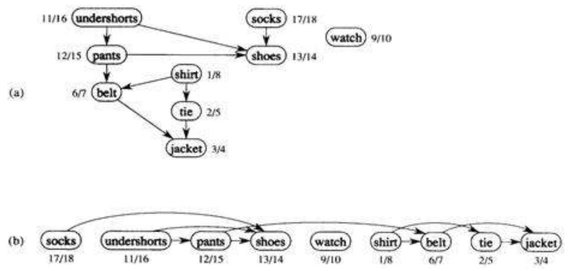


Figure 4.3.1 : Topological sorting example

A topological sort of a DAG is a linear ordering of vertices such that if (u, v) in E, then u appears somewhere before v (not like sorting numbers!). The topological sorting algorithm is described below:

TOPOLOGICAL-SORT(V, E)

Call DFS(V, E) to compute finishing times f[v] for all v in V

Output vertices in order of decreasing finish times

It should be clear from above discussion that we don't need to sort by finish times.

- We can just output vertices as they are finished and understand that we want the reverse of this list.
- Or we can put vertices onto the front of a linked list as they are finished. When done, the list contains vertices in topologically sorted order.

For example, from Figure 4.3.1 the order of vertices:

18 Socks

16 underpants

15 Pants

14 Shoes

10 Watch

8 Shirt

7 Belt

5 Tie

4 jacket

Shortest Paths

A path in a graph is a sequence of vertices in which every two consecutive vertices are connected. If the first vertex of the paths is v and the last is u, we say that the path is a path from v to u. Most of the time, we are interested in paths which do not have repeated vertices. These paths are called simple paths. Since we want to find shortest paths, we define a length of a path as the sum of lengths of its edges.

The shortest path problem is broadly divided into two major versions, the single source shortest path (SSSP) and all pairs shortest paths (APSP). In the SSSP version, given a source vertex s, the goal is to find the shortest paths from s to all other vertices in the graph. In the APSP version, for every pair of vertices (u, v), the goal is to in find the shortest paths between them. Clearly, if we are able to solve SSSP, we are also able to solve APSP by solving SSSP problem for every vertex as the source vertex. While this method is absolutely correct, as we will see, it is not always optimal.

SSSP Problem

Notice that if a graph has uniform length of all edges we already know how to solve the SSSP problem. That is if all edges have the same length, then the shortest path between two vertices is a path which has the fewer number of edges from all such paths and we already know that it can be solved with breadth first search.

The problem becomes harder if lengths are not uniform, in which case the shortest path may have many more edges that any other path. Consider the graph in Figure 4.3.2 where the edge lengths are shown near the edges. Note that the shortest path between vertex 1 and 6 has length 8 and contains 5 edges, while any other path from vertex 1 to 6 with fewer edges is longer. It is easy to see that BFS does not work.

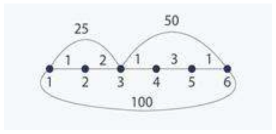


Figure 4.3.2 : A weighted graph

Having highlighted about the nature of the problem we proceed to describe the solution method. The most famous algorithm for solving the SSSP problem is Dijsktra algorithm. It is based on a crucial assumption that all edges have non-negative weights. Its correct logic strongly depends on this assumption that the algorithm does not work if the graph has negative edges.

Let d(s, v) be the length of shortest path from source vertex s to vertex v. The Dijkstra algorithm partition the set V of vertices into two disjoint sets X := {v : such that d(s, v) has its final value already computed} and V \ X. Initially V \ X is empty and initialize d(s, s):= 0 and d(s, v):= infinity for all v != s. Then as long as V \ X in not empty (i.e., there is at least one vertex v for which d(s, v) is not already computed), repeat the following step.

Let v be a vertex in V \ X such that d(s, v) is minimal from all such vertices. Move v to X and for each edge (v, u) such that v is in V \ X, update the current distance to u, i.e., do d(s, u) = min(d(s, u), d(s, v) + w(v, u)), where w(v, u) is the length of the edge from v to u.

Note that for the Dijkstra algorithm the following invariant holds: As long as V \X in not empty, there exists a vertex v in V \ X such that d(s, v) has already assigned its final value. This is true because we are dealing only with non negative edges and this vertex has to have the minimal value of d(s, v) from all vertices in V \ X.

The time complexity of Dijkstra algorithm is as follows. Note that there are n (number of nodes) iterations of the main loop, because during each iteration one vertex is moved to the set X. Also during iteration a vertex v which has the minimum value of d(s, v) is selected from vertices in V \ X, and the distance value for all neighbours of v is updated. It is easy to see that during the execution, this minimum is selected exactly n times and the distance values is updated exactly m (number of edges) times, because we do it once for every edge. Let f(n) be the time complexity of selecting a minimum and let g(n) be the time complexity of the update operation. Therefore, the total time complexity is O(n * f(n) + m * g(n)).

Both of these functions can be implemented in many different ways. The most common implementation is to use a heap based priority queue with decrease-key operation for updating distance values or to use a balanced binary tree. Both these methods give time complexity O(n log n + m log n). There are many cases that we can do better than that, for example, if length are small integers or if we can use more advanced priority queues like Fibonacci heap. But these methods are beyond the scope of this module; nevertheless it is worth knowing that they exist.

Now observe what can happen in case we have negative length edges. Consider the graph in Figure 4.3.3 . The Dijkstra algorithm does not work because it assigns d(s, s):= 0, then d(s, u):= 3 and finally d(s, v):= 1, but actually there exists a path from s to u of length 2.

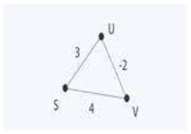


Figure 4.3.3 : A negative weighted graph

In this case a different method called Bellman-Ford (BF) algorithm can be used. The idea behind this algorithm is to have computed shortest paths which uses at most k edges in the kth iteration of the algorithm. After n - 1 iterations, all shortest paths are computed, because a simple path in a graph with n vertices can have at most n - 1 edges. The BF algorithm proceeds as shown below:

```
for v in V:
    if v == s:
        d[v] := 0
    else:
        d[v] := INFINITY
```

```
for i = 1 to n - 1:
    for (u, v) in E:
        d[v] := min(d[v], d[u] + length(u, v))
```

In order to see why the algorithm is correct, notice that if u, ..., w, v is the shortest path from u to v, then it can be constructed from the shortest path from u to w by adding the edge between w and v to it. This is a very important and widely used fact about shortest paths. Bellman-Ford algorithm works for negative edges also. One important assumption is that the input graph cannot contain a negative length cycle since the problem is not well defined in such graphs (you can construct a path of arbitrary small length). In fact, this algorithm can be used to detect if a graph contains a negative length cycle.

The time complexity of Bellman-Ford is O(n * m), because it performs n iterations and in each it examines all edges in the graph. Although this algorithm was invented a long time ago, it is still the fastest strongly polynomial algorithm, i.e., its polynomial complexity depends only on the number of vertices and the number of edges. It is the method commonly used in finding shortest paths in general graphs with arbitrary edges lengths.

APSP Problem

In this version of the problem, we need to find the shortest paths between all pairs of vertices. The most straightforward method to do that is to use any known algorithm for SSSP version of the problem and run it from every vertex as a source. This results in O(n * m log n) time if we use Dijkstra algorithm and O(n2 * m) if we use Bellman-Ford algorithm. If we do not have to deal with negative edges it works well because Dijkstra algorithm is quite fast.

Let us now take a look at Floyd-Warshall (FW) algorithm which runs in O(n3) even if there are negative length edges, which is based dynamic programming. The idea behind the FW algorithm is following: Let d(i, j, k) be the shortest path between vertex i and j which uses only vertices with numbers in {1, 2, ..., k}. Then d(i, j, n) is the desired result which we want to compute for all i and j. Assume that we have computed d(i, j, k) for all i and j, and we want to compute d(i, j, k + 1). Two cases need to be considered, namely, the shortest path between i and j which uses vertices in {1, 2, ..., k + 1} may either use the vertex with number k+1 or not.

If it does not use it, then it is equal to d(i, j, k).

If it uses vertex k+1, it is equal d(i, k + 1, k) + d(k + 1, j, k), since the vertex k+1 can occur only once in such a path.

The Floyd-Warshall algorithm is presented below.

```
for v in V:
    for u in V:
        if v == u:
            d[v][u] := 0
        else:
            d[v][u] := INFINITY
for (u, v) in E:
    d[u][v] := length(u, v)
for k = 1 to n:
    for i = 1 to n:
        for j = 1 to n:
            d[i][j] = min(d[i][j], d[i][k] + d[k][j])
```

This algorithm largely uses dynamic programming as it may have been observed. Its running time is O(n3), because it performs n iterations and in each it examines all pairs of vertices. In fact, it can be implemented so that it uses only O(n2) space and the paths can also be reconstructed very fast.

Conclusion

In this activity we presented the topological sort algorithm and algorithms for the shortest path problem. The topological sort takes an acyclic digraph as input and uses the DFS traversal to enumerate nodes in order precedence. We described the Dijkstra algorithm that finds the shortest path for a single source graph with non-negative edge weights, and the Bellman Ford algorithm which also works for graphs with negative edge weights. For the case of multiple sourced graphs, we showed that the two algorithms can be adapted with some loss of efficiency, though. For this reason, the Floyd-Warshall algorithm which finds shortest paths in multiple sourced graphs setting with arbitrary edge weights was discussed.

Assessment

1. Given a weighted graph in Figure 4.3.4 , find the shortest path from point 1 to all other nodes.

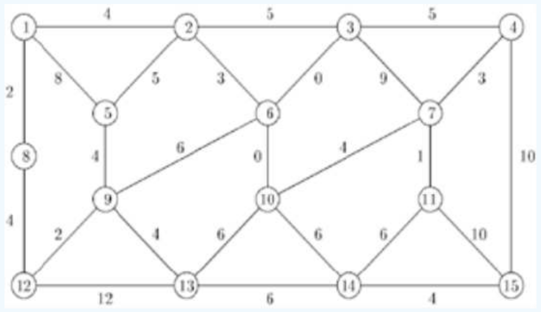


Figure 4.3.4 : A weighted graph

Apr 19, 2021, 5:50 PM

Figure 16

2. Show the steps of Bellman-Ford algorithm to find shortest path from A to E using the graph in Figure 4.3.5 . What is the result? How would the Floyd-Warshall algorithm behave when this graph is used as input?

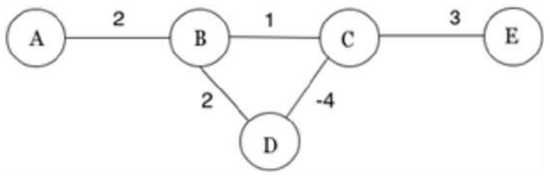


Figure 4.3.5 : A negative weighted graph

4.4: Unit Summary

This unit presented two advanced data structures, hash table and graph, along with some popular graph algorithms. Hash tables use hash functions to generate indexes for keys to be stored or accessed from array/list. A good hash function design is instrumental to the hash table performance expected to have a constant access time. The graph data structure is defined by a set of vertices and edges. It is very useful in problems modeled as networks.

A topological sort runs a depth-first search algorithm on acyclic digraph to order the vertices so that the keys are enumerated in descending order. The shortest path problem seeks to find a shortest path between a pair of vertices. The Dijkstra algorithm finds the shortest path for a single source graph with non-negative edge weights, and the Bellman Ford algorithm is like the Dijkstra algorithm but also works for graphs with negative edge weights. For the case of multiple sourced graphs, the two algorithms can be adapted with some loss of efficiency, though. For this reason, the Floyd-Warshall algorithm is used to finds shortest paths in multiple sourced graphs and arbitrary edge weights.

? Unit Assessment

Check your understanding!

Graphs, DFS and BFS

1. Consider a graph in Figure 4.4.1 .

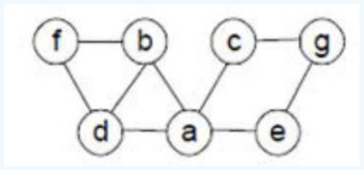


Figure 4.4.1 : A graph

Apr 20, 2021, 11:41 AM

Figure 18

a. Write the adjacency matrix and adjacency list for this graph
b. Starting from vertex a (and resolving ties by the vertex alphabetical order), list the vertices using a DFS traversal.
c. Starting from vertex a (and resolving ties by the vertex alphabetical order), list the vertices using a BFS traversal.

- One can model a maze by having a vertex for a starting point, finishing point, dead ends, and all the points in the maze where more than one path can be taken and then connecting the vertices according to the paths. Consider a maze shown in Figure 4.4.2 .

a. Construct a graph for the maze
b. Which traversal between DFS and BFS would you use if you found yourself in a maze and why?

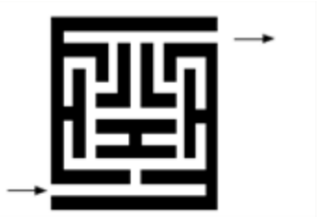


Figure 4.4.2 : A maze

Answers

mailto:njulumi@gmail.com

Grading Scheme

As per the offering Institution grading policy.

Unit Readings and Other Resources

The readings in this unit are to be found at course level readings and other resources.

Index

Glossary

Algorithm | A finite step-by-step procedure to achieve a required result

Algorithmics | A branch of computer science that consists of designing and analyzing computer algorithms

Binary search | A search based on divide and conquer strategy

Binary search tree (BST) | A type of binary tree where the nodes are arranged in order, that is, for each node all keys in its left sub-tree are less-or-equal to its key and all the keys in its right sub-tree are greater than its key

Binary tree | A tree made of nodes, where each node contains a “left” pointer, a “right” pointer, and a data element

Dynamic programming | Solves a problem by always remembering answers to the sub-problems already solved

Graph | A data structure represented as a collection of finite sets of vertices and edges

Hash table | A data structure that stores a key/value pair

Hashing | A process of converting large keys into small ones using a hash function

Heap | A specific tree based data structure in which all the nodes of tree are in a specific order

In-order traversal (Walk) | A BST walk that recursively prints the BST keys in monotonically increasing order

Linear search | A search that sequentially looks for an item in a collection from beginning to the end

Recursion | Solves a problem by reducing it to a simpler problem of the same kind

Red-black trees | A variant of BST that ensure that the tree is balanced, that is, its height is O(lg n), where n is the number of nodes

Searching | A process of finding a particular item in a collection of items

Sorting | A process that puts elements of a collection such as array or list in a certain order

Time complexity | An expression of the number of steps needed as a function of the problem size

Summary

Course Assessment

Miscellaneous Problems

1. Which sorting algorithm is always slow and time complexity O(n2)?
2. Which sorting algorithm is sometimes O(n)?
3. What sorting algorithm runs in O(log n)?
4. What other sorting algorithms have O(n log n) expected time?
5. Which sorting algorithm needs extra memory space?
6. Consider the N-Queens Problem discussed in Unit 1, which was used to demonstrate the recursive backtracking strategy. Assemble the code snapshot provided for N-Queens problem and write a complete program.
7. Suppose you are given a sorted array with possible duplicate elements, write a program to find the number of occurrences of a given input item (assuming the element is present in the array).
8. Consider two sets of integers, S={s1, s2, ..., sm} and T = {t1, t2, ..., tn}, m ≤ n.
   a. Design an algorithm that uses a hash table of size m to test whether S is a subset of T.
   b. What is the average running time of your algorithm?

Answers

mailto:njulumi@gmail.com

Grading Scheme

As per the offering Institution grading policy.

Course References

- Introduction to the Design and Analysis of Algorithms, Anany Levitin, 2nd Edition, Pearson Education, Inc. 2007
- Algorithm Design, John Kleinberge and Eva Tardos, 1st Edition, Perason Education Inc., 2006
- Design and Analysis of Algorithms: Course Notes, Samir Khuller, University of Maryland, 1996
- https://www.hackerearth.com/notes
- http://staff.fh-hagenberg.at/schrein...otes/trending/

Glossary

Sample Word 1 | Sample Definition 1

Detailed Licensing

Overview

Title: Algorithm Design and Analysis (Justo)

Webpages: 45

All licenses found:

- Undeclared: 55.6% (25 pages)
- CC BY-SA 3.0: 44.4% (20 pages)

By Page

- Algorithm Design and Analysis (Justo) - *CC BY-SA 3.0*
  - Front Matter - *CC BY-SA 3.0*
    - TitlePage - *CC BY-SA 3.0*
    - InfoPage - *CC BY-SA 3.0*
    - Table of Contents - *Undeclared*
    - Foreword - *Undeclared*
    - Licensing - *Undeclared*
    - Production Credits - *Undeclared*
    - Copyright Notice - *Undeclared*
    - Course Overview - *CC BY-SA 3.0*
      - 1.1: Welcome to Algorithms Design and Analysis 0 - *Undeclared*
      - 1.2: Prerequisites - *Undeclared*
      - 1.3: Materials - *Undeclared*
      - 1.4: Course Goals - *Undeclared*
      - 1.5: Units - *Undeclared*
      - 1.6: Assessment - *Undeclared*
      - 1.7: Schedule - *Undeclared*
      - 1.8: Readings and Other Resources - *Undeclared*
      - 1.9: Course References - *Undeclared*
  - 1: Fundamental Design and Analysis Techniques - *CC BY-SA 3.0*
    - 1.1: Activity 1 - Overview of Algorithm Design and Analysis - *Undeclared*
    - 1.2: Activity 2 - Recursion and Recursive Backtracking - *Undeclared*
    - 1.3: Activity 3 - Dynamic Programming - *Undeclared*
    - 1.4: Unit Summary - *Undeclared*
  - 2: Searching, Binary Search Trees and Heaps - *CC BY-SA 3.0*
    - 2.1: Activity 1 - Searching Techniques - *CC BY-SA 3.0*
    - 2.2: Activity 2 - Binary Search Tree - *CC BY-SA 3.0*
    - 2.3: Activity 3 - Heaps - *CC BY-SA 3.0*
    - 2.4: Unit Summary - *CC BY-SA 3.0*
  - 3: Sorting Algorithms - *CC BY-SA 3.0*
    - 3.1: Activity 1 - Sorting Algorithms by insertion method - *CC BY-SA 3.0*
    - 3.2: Activity 2 - Sorting Algorithms by selection method - *CC BY-SA 3.0*
    - 3.3: Activity 3 - Sorting Algorithms by Merge and Partition methods - *CC BY-SA 3.0*
    - 3.4: Unit Summary - *CC BY-SA 3.0*
  - 4: Hash Tables, Graphs and Graph Algorithms - *CC BY-SA 3.0*
    - 4.1: Activity 1 - Hash Tables - *Undeclared*
    - 4.2: Activity 2 - Graphs - *Undeclared*
    - 4.3: Activity 3 - Graph Algorithms - *Undeclared*
    - 4.4: Unit Summary - *Undeclared*
  - Back Matter - *CC BY-SA 3.0*
    - Index - *CC BY-SA 3.0*
    - Glossary - *CC BY-SA 3.0*
    - Summary - *Undeclared*
    - Glossary - *Undeclared*
    - Detailed Licensing - *Undeclared*