Learning Guide Unit 5

Site: University of the People
Course: CS 3304-01 Analysis of Algorithms - AY2024-T5
Book: Learning Guide Unit 5

Printed by: Mejbaul Mubin
Date: Saturday, 29 June 2024, 6:44 AM

Description

Learning Guide Unit 5

Table of contents

Overview

Unit 5: Dynamic Programming

Topics:

- Dynamic Programming
- The Knapsack Problem
- Randomized Algorithms

Learning Objectives:

By the end of this Unit, students will be able to:

1. Apply dynamic programming as an algorithm design strategy.
2. Articulate how dynamic programming differs from a divide and conquer approach.
3. Implement a dynamic programming algorithm to solve the knapsack problem.
4. Describe the features and functioning of randomized algorithms.

Tasks:

- Peer assess Unit 4 Programming Assignment
- Read the Learning Guide and Reading Assignments
- Participate in the Discussion Assignment (post, comment, and rate in the Discussion Forum)
- Complete and submit the Programming Assignment
- Make entries to the Learning Journal
- Take the Self-Quiz

Introduction

In this unit we will be covering one new pattern, Dynamic programming. Throughout this course we have been learning about the characteristics of different patterns of algorithms. In unit one, we reviewed some of the material that was covered in CS1303 Data Structures and Algorithms and discussed the patterns that were represented by some of those algorithms. The patterns that we discussed included:

Brute Force which is simple to implement but not but not very efficient when the input size gets large. We also discussed variations of Brute force that make it a bit more efficient including the Backtracking and Branch and Bound patterns.

In unit two, we introduced the 'Divide and Conquer' approach, represented by algorithms such as the merge sort that gain efficiency by breaking a large problem into subsequently smaller problems until those small problems and be easily solved and then combined together to get a solution for the larger problem.

In units 3 and 4 we introduced a new pattern known as 'Greedy algorithms' and looked at how 'greedy' algorithms can solve minimum spanning tree problems using algorithms such as Prim's and Kruskal's algorithms.

In this unit we introduce dynamic programming which is a pattern that is an extension to the divide and conquer pattern. In divide and conquer we take a large problem, break it down into many smaller sub-problems and then solve those sub-problems. These results are then combined for a solution to the problem.

There are many cases where these sub-problems are repetitive. Imagine if you would a large problem that is broken down into hundreds of these sub-problems. Let us further assume that the sub problem is to multiply two numbers and that the same multiplication is repeated many times. This would be inefficient because we are doing the same work repeatedly. If we could eliminate all of the redundancy and only complete the unique multiplications, then we would have an algorithm that is much more efficient.

This principle of eliminating any redundant or repeated operations is the basis of dynamic programming.

The word dynamic is used because it refers to the fact that the algorithm can determine which work is redundant and then make a decision not to repeat it. This same concept of conditional evaluation and computation appears elsewhere in computer science. In functional programming we see a similar concept referred to as Lazy evaluation.

Another facet of dynamic programming is the idea that problems are divided into smaller sub-problems but those sub-problems are ordered from the easiest to solve to the hardest. In applications such as finding a shortest path, if the path that needs to be evaluated requires only the first few nodes then much of the processing of the remaining nodes can be eliminated making the algorithm very efficient.

The Schaffer text will use the classic 'Knapsack' problem to illustrate the benefits of dynamic programming. It would be recommended to do some research on the knapsack problem using the internet

Randomized Algorithms
A randomized algorithm is an algorithm which employs a degree of randomness as part of its logic. The algorithm typically uses a randomly generated number as an input to guide its behavior, in the hope of achieving good performance in the "average case". Of course this means that it is possible that an instance of running the algorithm could have poor running time, however, the goal of a randomized algorithm is to have improved performance in the average case.

They way that randomized algorithms can improve performance is by randomizing decision making. One way of thinking about this is by considering the field of 'decision theory'. In decision theory we learn that there are two basic kinds of decision making processes. The first is called 'rational decision making'. A Rational decision is one where all of the potential alternatives and options have been identified, evaluated, and assessed when every possible option has been examined and the advantages and disadvantages of each tabulated and sorted, the option that has the greatest advantage and the least disadvantage is selected.

Sounds good … but in reality it is simply not practical or possible in most situations. In most cases there are a myriad of alternatives all with different advantages, disadvantages, costs, and impacts. As human beings we simply lack the processing capability to evaluate and assess all of the potential alternatives.

Consider the process of purchasing a new automobile. There are dozens of manufacturers, hundreds of styles, thousands of options ranging from wheels, engine, tires, interior, color, warrantee and an entire host of other factors including cost, financing, serviceability, safety ratings and on and on. Buying a car today is a long ways away from the days of Henry Ford who used to asset that you could

purchase the Model-T in “any color you want … so long as its black!”

As human beings if we were rational …. We would never make any decisions. The very best decision is simply impractical so we have learned to make “Good enough” decisions that perhaps evaluate to most important criteria. In decision theory we refer to this as being “boundedly rational”. This term simply means that we make decisions that are as rational as possible, but there are options, factors, or considerations that we leave up to chance.

Randomized algorithms are very much like this. We can eliminate some of the complexity of an algorithm thus reducing its running time by “randomizing” some decisions. Like the individual, the algorithm simply cannot take the time to evaluate everything so we use randomized input in some decisions to speed the process up.

Consider the brute force approach to the knapsack problem. The brute force approach is equivalent to the ‘rational’ approach to decision making because it evaluates EVERY possible combination of items to find a solution. As we will see in this units development project in the knapsack problem if there were only 16 items in the store then a ‘rational’ or brute force approach to solving the problem would need to examine over 65,000 different combinations. At 32 items in the store the number of combinations to be evaluated is over 4 billion and of course when we get up to 64 items in the store there are ….. well to borrow a phrase from Carl Sagan “billions and billions” of combinations.

The use of randomized input and chance as part of this process can achieve a “good enough” answer and do so with a much more efficient algorithm. You may be wondering under what situations a “good enough” outcome is sufficient in computer science. The reality is that there are a large number of problems that randomized algorithms are suited to and new problems that are emerging (such as Big Data) for which such efficient algorithms may be the only solutions.

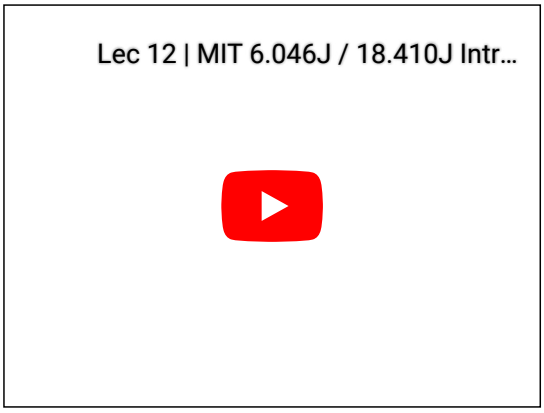
Reading Assignment

Topic 1: Dynamic Programming
Chapter 16: Patterns of Algorithms, Sections 16.1 – 16.2, in A Practical Introduction to Data Structures and Algorithm Analysis by Clifford A. Shaffer.
Chapter 6 Dynamic Programming in Algorithms by S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani available at http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf

Topic 2: Revisiting the Knapsack Problem with Dynamic Programming
Chapter 16: Patterns of Algorithms, Sections 16.1 – 16.2, in A Practical Introduction to Data Structures and Algorithm Analysis by Clifford A. Shaffer.
Chapter 6 Dynamic Programming in Algorithms by S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani available at http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf

Topic 3: Randomized Algorithms and Skip Lists
Chapter 16: Patterns of Algorithms, Sections 16.1 – 16.2, in A Practical Introduction to Data Structures and Algorithm Analysis by Clifford A. Shaffer.

Supplemental Materials
Dynamic Programming, By Charles E. Leiserson – MIT http://videolectures.net/mit6046jf05_leiserson_lec15/
Skip Lists, By Erik Demaine – MIT



Unit 5 Optional Video Lectures
The following video lectures are optional resources that have been made available to students who can take advantage of them. These lectures are strictly optional resources. All of the information in these lectures is available in other learning resources within the course. These lectures are provided for those students who have sufficient network bandwidth and technology capabilities to take advantage of video content. These lectures cannot be used instead of the required assigned resources and there is no information that is not contained in the assigned resources. These lectures simply present some of the information in a different format.

- Unit 5 Lecture 1: Dynamic Programming
- Unit 5 Lecture 2: Randomized Algorithms

Discussion Assignment

In your own words describe the ‘knapsack problem’. Further, compare and contrast the use of a brute force and a dynamic programming algorithm to solve this problem in terms of the advantage and disadvantages of each. An analysis of the asymptotic complexity of each is required as part of this assignment.

Include one or two examples to explain your thought process to show what is occurring and how the methodology works. Use APA citations and references for any sources used.

Programming Assignment

A thief robbing a store can carry a maximum weight of W in their knapsack. There are n items and ith item weighs wi and is worth vi dollars. What items should the thief take to maximize the value of what is stolen?

The thief must adhere to the 0-1 binary rule which states that only whole items can be taken. The thief is not allowed to take a fraction of an item (such as ½ of a necklace or ¼ of a diamond ring). The thief must decide to either take or leave each item.

Develop an algorithm using Java and developed in the Cloud9 environment (or your own Java IDE) environment to solve the knapsack problem.

Your algorithms should use the following data as input.

Maximum weight (W) that can be carried by the thief is 20 pounds

There are 16 items in the store that the thief can take (n = 16). Their values and corresponding weights are defined by the following two lists.

Item Values: 10, 5, 30, 8, 12, 30, 50, 10, 2, 10, 40, 80, 100, 25, 10, 5

Item Weights: 1, 4, 6, 2, 5, 10, 8, 3, 9, 1, 4, 2, 5, 8, 9, 1

Your solution should be based upon dynamic programming principles as opposed to brute force.

The brute force approach would be to look at every possible combination of items that is less than or equal to 20 pounds. We know that the brute force approach will need to consider every possible combination of items which is 2n items or 65536.

The optimal solution is one that is less than or equal to 20 pounds of weight and one that has the highest value. The following algorithm is a 'brute force' solution to the knapsack problem. This approach would certainly work but would potentially be very expensive in terms of processing time because it requires 2n (65536) iterations

The following is a brute force algorithm for solving this problem. It is based upon the idea that if you view the 16 items as digits in a binary number that can either be 1 (selected) or 0 (not selected) than there are 65,536 possible combinations. The algorithm will count from 0 to 65,535, convert this number into a binary representation and every digit that has a 1 will be an item selected for the knapsack. Keep in mind that not ALL combinations will be valid because only those that meet the other rule of a maximum weight of 20 pounds can be considered. The algorithm will then look at each valid knapsack and select the one with the greatest value.

```
import java.lang.*;
import java.io.*;

public class Main {

   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
      int a, i, k, n, b, Capacity, tempWeight, tempValue, bestValue, bestWeight;
      int remainder, nDigits;
      int Weights[] = {1, 4, 6, 2, 5, 10, 8, 3, 9, 1, 4, 2, 5, 8, 9, 1};
      int Values[] = { 10, 5, 30, 8, 12, 30, 50, 10, 2, 10, 40, 80, 100, 25, 10, 5 };
      int A[];

      A = new int[16];

      Capacity = 20; // Max pounds that can be carried
      n = 16;  // number of items in the store
      b=0;

      tempWeight = 0;
```

```
        tempValue = 0;
        bestWeight = 0;
        bestValue = 0;

    for ( i=0; i<65536; i++) {
            remainder = i;

            // Initialize array to all 0's
            for ( a=0; a<16; a++) {
               A[a] = 0;
            }

            // Populate binary representation of counter i
            //nDigits = Math.ceil(Math.log(i+0.0));
            nDigits = 16;

            for ( a=0; a<nDigits; a++ ) {
               A[a] = remainder % 2;
               remainder = remainder / 2;
            }

            // fill knapsack based upon binary representation
            for (k = 0; k < n; k++) {

               if ( A[k] == 1) {
                  if (tempWeight + Weights[k] <= Capacity) {
                     tempWeight = tempWeight + Weights[k];
                     tempValue = tempValue + Values[k];
                  }
               }
            }

            // if this knapsack is better than the last one, save it
            if (tempValue > bestValue) {
               bestValue = tempValue;
               bestWeight = tempWeight;
               b++;
            }
            tempWeight = 0;
            tempValue = 0;
      }
      System.out.printf("Weight: %d Value %d\n", bestWeight, bestValue);
      System.out.printf("Number of valid knapsack's: %d\n", b);
   }
}
```

The brute force algorithm requires 65,536 iterations (216) to run and returns the output defined below. The objective of this assignment will be to develop a java algorithm designed with dynamic programming principles that reduces the number of iterations. The brute force algorithm requires an algorithm with exponential 2n complexity where O(2n). You must create a dynamic programming algorithm using java to solve the knapsack problem. You must run your algorithm using Java and post the results. You results must indicate the Weight of the knapsack, the value of the contents, and the number of iterations just as illustrated in the brute force output below. You must also include a description of the Big O complexity of your algorithm.

Output from the Brute Force Algorithm.

Weight: 20
Value: 280
Number of valid knapsack's: 45

For a hint on the dynamic programming approach see the following:

The basic idea behind the dynamic programming approach: Compute the solutions to the sub-problems once and store the solutions in a table, so that they can be reused (repeatedly) later.
http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf

Some of these algorithms may take a long time to execute. If you have access to a java compiler on your local computer or the Virtual Computing Lab, you may want to test your code by running it and executing it with java directly as it can speed up the process of getting to a result. You should still execute your code within Java to get an understanding of how it executes. (To compile with java use the javac command. To run a compiled class file, use the java command)

Assessment
You will have ONE WEEK to complete this assignment. It will be due the end of this unit. Your assignment will be assessed (graded) by your peers. You should post this assignment, the results, and other requirements such as the asymptotic analysis in one of the following formats:

- Directly cut-and-pasted into the text box for the posting.
- As a document in either RTF or Word 97/2003 format.

Learning Journal

The Learning Journal is a tool for self-reflection on the learning process. In addition to completing directed tasks, you should use the Learning Journal to document your activities, record problems you may have encountered and to draft answers for Discussion Forums and Assignments. The Learning Journal should be updated regularly (on a weekly basis), as the learning journals will be assessed by your instructor as part of your Final Grade.

Your learning journal entry must be a reflective statement that considers the following questions:

- Describe what you did. This does not mean that you copy and paste from what you have posted or the assignments you have prepared. You need to describe what you did and how you did it.
- Describe your reactions to what you did
- Describe any feedback you received or any specific interactions you had. Discuss how they were helpful
- Describe your feelings and attitudes
- Describe what you learned

Another set of questions to consider in your learning journal statement include:

- What surprised me or caused me to wonder?
- What happened that felt particularly challenging? Why was it challenging to me?
- What skills and knowledge do I recognize that I am gaining?
- What am I realizing about myself as a learner?
- In what ways am I able to apply the ideas and concepts gained to my own experience?

Your Learning Journal should be a minimum of 500 words

Self-Quiz

The Self-Quiz gives you an opportunity to self-assess your knowledge of what you have learned so far.

The results of the Self-Quiz do not count towards your final grade, but the quiz is an important part of the University's learning process and it is expected that you will take it to ensure understanding of the materials presented. Reviewing and analyzing your results will help you perform better on future Graded Quizzes and the Final Exam.

Please access the Self-Quiz on the main course homepage; it will be listed inside the Unit.

Checklist

- [ ] Peer assess Unit 4 Programming Assignment
- [ ] Read the Learning Guide and Reading Assignments
- [ ] Participate in the Discussion Assignment (post, comment, and rate in the Discussion Forum)
- [ ] Complete and submit the Programming Assignment
- [ ] Make entries to the Learning Journal
- [ ] Take the Self-Quiz