San Jose State University
SJSU ScholarWorks

Master's Projects | Master's Theses and Graduate Research

10-1-2011

# JDPET: Java Dynamic Programming Educational Tool

Aaron Lemoine

Follow this and additional works at: http://scholarworks.sjsu.edu/etd_projects

Recommended Citation
Lemoine, Aaron, "JDPET: Java Dynamic Programming Educational Tool" (2011). *Master's Projects.* Paper 202.

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

# JDPET:
# Java Dynamic Programming Educational Tool

A Report
Presented to
The Faculty of the Department of Computer Science
San José State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Computer Science

by
Aaron S. Lemoine
November 2011

© 2011
Aaron S. Lemoine
ALL RIGHT RESERVED

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

---

Dr. David Taylor Department of Computer Science

---

Dr. Chris Pollett Department of Computer Science

---

Dr. Teng Moh Department of Computer Science

APPROVED FOR THE UNIVERSITY

---

# Abstract

# JDPET: Java Dynamic Programming Educational Tool

There exist many different algorithm types for solving problems, one of which is dynamic programming. To assist students to learn about dynamic programming algorithms, JDPET was developed. JDPET is an interactive, visual, problem solving tool that allows students to solve several different problems and learn how dynamic programming can be applied to solve these problems. JDPET also provides students with detailed feedback on the problems they attempt to solve.

# Table of Contents


**1. Introduction** ..... 1
**2. Dynamic Programming** ..... 2
  **2.1 Four Steps of Dynamic Programming Algorithm Creation** ..... 2
  **2.2 Solvable Problems with Dynamic Programming** ..... 2
  **2.3 Others Work With Dynamic Programming** ..... 3
**3. JDPET** ..... 4
  **3.1 What Students Must Know About JDPET** ..... 4
  **3.2 What Teachers Must Know About JDPET** ..... 4
**4. How to Use JDPET** ..... 6
  **4.1 Basics of JDPET** ..... 6
    **4.2.2 Knapsack NS Algorithm** ..... 9
    **4.2.3 JDPET and Knapsack NS** ..... 12
    **4.2.4 Knapsack S Algorithm** ..... 15
    **4.2.5 JDPET and Knapsack S** ..... 16
  **4.3 Fractional Knapsack Problem** ..... 18
    **4.3.1 Creating the Algorithm** ..... 18
    **4.3.2 Fractional Knapsack Algorithm** ..... 18
    **4.3.3 JDPET and Fractional Knapsack** ..... 19
  **4.4 Longest Common Subsequence Problem** ..... 21
    **4.4.1 Creating the Algorithm** ..... 21
    **4.4.2 Longest Common Subsequence Algorithm** ..... 22
    **4.4.3 JDPET and Longest Common Subsequence** ..... 24
  **4.5 Longest Increasing Subsequence Problem** ..... 27
    **4.5.1 Creating the Algorithm** ..... 27
    **4.5.2 Longest Increasing Subsequence Algorithm** ..... 27
    **4.5.3 JDPET and Longest Increasing Subsequence** ..... 29
  **4.6 Shuffle Problem** ..... 32
    **4.6.1 Creating the Algorithm** ..... 32
    **4.6.2 Shuffle Algorithm** ..... 33
    **4.6.3 JDPET and Shuffle** ..... 35
**5. Student Reviews** ..... 39
**6. Future Work** ..... 40
**7. Conclusion** ..... 41
**8. References** ..... 43

# 1. Introduction

Teaching students can be a difficult task because students learn in different ways. Some students may prefer in class lectures while others may prefer to read books. To learn how to solve problems, some students may learn by watching an example while others may learn by applying themselves to problems and getting feedback. With learning by problems, teachers' time is limited so they may not be able to show students enough examples or give enough feedback on a students' work. With the time limitation of teachers, automated educational tools can assist students with a wide variety of problems and subjects. One such subject is teaching students about algorithms, more specifically, in this case, dynamic programming algorithms. To help students learn about dynamic programming algorithms, I have created Java Dynamic Programming Educational Tool (JDPET).

Other educational tools have been made to educate students about a wide variety of algorithms, some of which are dynamic programming algorithms. The other educational tools typically educate students with worked examples. Although those are fine for some students, other students may learn more by solving the problems for themselves. Tools such as AlViE provide visual worked examples that are a good starting point to learn, but when students do not solve the problems, what they learn is limited[9]. JDPET goes one step further than tools such as AlViE and allows for both worked examples and for students to solve problems with feedback. Another tool called TRAKLA also allows for both worked examples and student problem solving, however the feedback on this tool is limited[11]. JDPET provides better feedback than TRAKLA by providing information on the student's answers such as what the correct answer is based on a student's previous mistakes, if any are made, and what the correct worked example answer is.

JDPET was developed with a Windows 7 64 bit machine using Java SE development kit 6 update 23. As JDPET does not use any packages outside of the basic Java install environment, the same version of the java development kit or 6.0.290 will be capable of running JDPET. Newer versions should also be able to run JDPET.

JDPET helps students learn about dynamic programming by solvable problems and work examples. It will generate problem instances for the fixed set of problem types. Students are presented six different algorithms, each able to randomly generate problem instances. JDPET will allow students to solve problems, obtain hints, and be provided feedback on the work they have done. Though JDPET alone cannot teach students about dynamic programming, such as how create dynamic programming algorithms, it can help fortify their understanding of the algorithms presented in JDPET and potentially assist them to understand the nature of dynamic programming.

# 2. Dynamic Programming

Dynamic programming is a programming style for solving difficult and complex problems. Similarly to the divide and conquer programming style, dynamic programming breaks a problem into subproblems and breaks those subproblems into smaller and smaller subproblems until no more subproblems are possible[2]. Dynamic programming does differ from divide and conquer by solving a redundant calculation flaw that might be present in some divide and conquer algorithms. Divide and conquer method may recursively solve and resolve the same subproblems and therefore can be wasteful. Dynamic programming remedies this problem by saving subproblem calculations. Although dynamic programming saves calculation time, it does so at the expense of storage space.

Dynamic programming is commonly used for optimization based problems[2]. Although the given problem may have multiple correct answers, dynamic programming will typically solve and present one optimized answer.

### 2.1 Four Steps of Dynamic Programming Algorithm Creation

In Thomas Cormen's book, Introduction to Algorithms, he describes the development of dynamic programming algorithms in four steps:

1) Characterize the structure of an optimal solution.
2) Recursively define the value of an optimal solution.
3) Compute the value of an optimal solution in a bottom-up fashion.
4) Construct an optimal solution from computed information.

The first three steps are used to solve the optimal solution to a problem, where as the fourth step is used to find additional information about the problem, such as what values were used in the optimized solution[2]. The goal of the first step is to describe the nature of the optimal solution, the second step the builds a recursive solution based on the description, and the third step builds an algorithm based of the recursive solution. The final step is an optional step to gain additional computation information and can be done by using the data constructed in third step.

### 2.2 Solvable Problems with Dynamic Programming

The problem types solved by dynamic programming algorithms included in JDPET are the knapsack, longest common subsequence, longest increasing subsequence, and a card shuffling problem. JDPET also solves one knapsack problem by a greedy algorithm to show a little more variety with this problem.

### 2.3 Others Work With Dynamic Programming

Other educational tools for solving problems with dynamic programming algorithms mostly cover algorithms and problem types that differ from JDPET such as Fibonacci or a variation on the knapsack problem[6][9]. AlViE does cover the longest common subsequence problem which JDPET also covers, but AlViE only has a workable example.

# 3. JDPET

JDPET is a Java application that attempts to give students an opportunity to test and improve their knowledge about how a few dynamic programming algorithms work through interactive examples and visualization. JDPET presents six different algorithms to the students which include three knapsack algorithms, a longest increasing subsequence algorithm, a longest common subsequence algorithm, and a shuffle algorithm. Running through the problems and algorithms, students should visually see how dynamic programming breaks down and stores a problem into many subproblems and how an answer can be found by the stored subproblems. By interacting with the algorithms and using other teaching supplements, students should gain a greater understanding about dynamic programming and potentially they will gain enough of an understanding to assist them in creating new dynamic programming based algorithms.

### 3.1 What Students Must Know About JDPET

JDPET provides several interactive algorithm problems which students can use to help them gain a better understanding of dynamic programming. The students will be able to solve a problem based on how the algorithm solves the problem, be provided hints and feedback upon request, and receive grading when finished with the algorithm. Grading does take account of a student's mistakes so if a student makes a mistake in how the algorithm solves the problem, JDPET will attempt to grade the problem as if the incorrectly answered subproblems were correct for other subproblems of the algorithm. Students will be encouraged to attempt the problems based on what lesson or reading material they have been provided before hand, but ultimately must know that these problems follow very specific algorithms which will be provided to them in JDPET and the instructions. Students will need to read the instructions on JDPET to use it. Instructions will describe the problems, provide information on creating an algorithm, the algorithm created and used for JDPET, how to interact with JDPET, and information on the feedback.

### 3.2 What Teachers Must Know About JDPET

JDPET cannot be used as a stand alone educational tool, students must have some additional lesson supplements on dynamic programming, such as the four steps of creating a dynamic programming algorithm. JDPET will only provide the students with an interactive tool based on a specific algorithm with each problem. Instructions will describe the problems, provide information on creating an algorithm, the algorithm created and used for JDPET, how to interact with JDPET, and information on feedback. Teachers are encouraged to go into more detail about the given problems and how to construct a dynamic programming algorithm based on a problem. After students are provided some material on the problems, it is then the best time to introduce them to JDPET. Teachers should encourage the students to attempt the problems before reading

the specific algorithm to see if they can construct their own algorithms and how similar it may be to the JDPET algorithm. Ultimately they must know that these problems follow very specific algorithms which will be provided to them in JDPET and the instructions.

# 4. How to Use JDPET

First the basic instructions for using JDPET are presented. For each problem type, this section will briefly explains each problem type implemented in JDPET and how a dynamic programming algorithm may be constructed based on four steps for creating dynamic programming algorithms from Cormen's book[2][5][7]. After the algorithm is constructed and explained, the final part explains the specific instructions for using the algorithm with JDPET.

### 4.1 Basics of JDPET

The user will be provided with the JDPET.jar file and he must have his system set up to run jar files. Upon running the JDPET.jar file, the user will be presented a menu that has six buttons corresponding to each of the six algorithms JDPET covers. Selecting a button will lead the user to the indicated interactive algorithm which will always start with the same preset problem values, giving the user a familiar problem instance with which to practice.

Within each algorithm there are several buttons at the top of the screen: menu, new, solve, hint, and instructions. Selecting the menu button will bring the user back to the starting menu to select a different algorithm. Selecting the new button will clear any and all work done so far on the problem and generate a new random problem. Selecting the solve button will solve the current subproblem the user is on, provided the subproblem has an answer. Since subproblems may not have valid answers due to mistakes on previous subproblems, JDPET will not provide an answer for subproblems that cannot be answered. Selecting the hint button will provide one or two features depending on the algorithm. The first hint feature will highlight values as the color green to give the user some hints on the current subproblem answer. The first hint feature is not present in every algorithm, more details will be given in each algorithm's section. The second hint feature enables feedback on each step. With the feedback the user may attempt to solve the step again or simply continue and try to answer the remaining subproblems as correctly as possible. The instructions button creates a popup window which gives some general instructions for how to interact with JDPET.

Below the top buttons is where the problems are solved. This screen can be divided into four sections that each holds different information for the problem: top left, top right, bottom left, and bottom right. Each section holds similar functionality between the different algorithms and any differences will be specified in their particular section. The upper left section holds a table or array used for solving the problem and holds answers for phase 1. This section also holds several buttons, a text field, and additional information useful to solving the current subproblem. The upper right section holds information for answers selected in phase 2. This section will also hold feedback about each subproblem if the hint is on. The lower left section holds information about the

problem being solved. The lower right section is the code and report section. During phase 1 and 2, partial pseudo code will be given for what the student is currently trying to solve. After grading, this section gives additional details about answers selected when the hint is on and will also give information on the user's results.

During phase 1, the focus is to fill in the table or array in the upper left section while phase 2 focus is to fill in the upper right array using information from phase 1. The user will see a table cell or array cell and some header values colored cyan, this is the current subproblem the user is solving. In phase 1, the user will need to fill in all the table cells before moving on to the next phase, where as arrays differ in each algorithm. In phase 2 the user will navigate through the upper left table or array until an end location, then the problem is finished.

To enter values in a table, there are different ways to do so. The first method is entering the values in the text box then pressing the enter key or pressing the next button. Some problems are missing the next button which is replaced by different buttons. With those problems clicking on the buttons will provide a answer independent of the text field. The final method is clicking on one of the table or array cells. This will fill in the answer. Specifics of what to answer for are provided in each section. Clicking the undo button will bring the user back a single step in the algorithm as long as the problem has not been graded.

After finishing phase 2, a grade button will appear. Selecting the grade button will end the problem and grade it for the user and display the user's answer above the report section. The upper left section will have the graded answers for phase 1, while the upper right section will have the graded answers for phase 2. Each table or array cell value will be colored differently to indicate the results, a color code can be seen in Table 1. Clicking on the table or array cells will give a general message as well as more details about the problem in the report section, the detailed report can be seen in Table 2. Since general messages differ enough, a separate table is within each algorithm's section. At the bottom of the report section are colored numbers, these correspond to the number of colored graded items according to those colors. An exception to this is black which indicates the number of subproblems the user answered, while blue indicates total number of different subproblems solved for the user at one point. For example if the user has JDPET solve the problem, undo, and solve it themselves then JDPET will still add this to the blue counter, instead of a different colored counter so long as the subproblem is still answered by the user. If the hint button is enabled, then the two correct colored numbers will have two numbers instead of one. The first number will indicate the correct answer while the second will indicate the number of correct answers with hint enabled.

**Table 1. Grading Colors**

| Color | Description |
|---|---|
| Green | Correct answer for subproblem entered. |
| Red | Incorrect answer for subproblem entered. |

| | |
|---|---|
| Blue | JDPET solved the subproblem. |
| Orange | Correct answer based on mistakes of previous subproblems. Answer does not match up with the true correct answer. |
| Purple | Subproblem answers provided has caused the algorithm to miss a step or end earlier than it should have. This shows the true answers to a subproblem that have not been answered |
| Black | Data setup set by JDPET. |

**Table 2. Grading Details**

| Details | Description |
|---|---|
| Correct Value | The correct answer based on previously entered subproblems. May be missing if no correct answer exists. |
| Correct i/j, Correct i, Correct j, Correct Direction | The location of the correct subproblem used to solve the current subproblem. May be missing if no correct answer exists. |
| Selected Value | Present when the user's answer or subproblem selected is different than the correct answer or subproblem. Displays the user's selected answer. |
| Selected i/j, Selected i, Selected j, Selected Direction | Present when the user's answer or subproblem selected is different than the correct answer or subproblem. Also may not be displayed if the answer entered by keyboard and cannot be traced to any subproblem. Displays the user's selected subproblem. |
| True Correct Value | Present when the correct answer differs than what the real answer should be at the subproblem, caused by mistakes in previous subproblems. Displays the real correct answer at the current subproblem if no previous mistakes were made. If algorithm runs longer than it should, "No true answer" will be displayed. |
| True Correct i/j, True Correct i, True Correct j, True Correct Direction | Present when the correct answer differs than what the real answer should be at the subproblem, caused by mistakes in previous subproblems. Displays the real location of the correct subproblem used to solve the current subproblem if no previous mistakes were made. |
| Other Considered i/j, Other Considered j | Present when neither of the two previous subproblems considered by the algorithm are selected. Displays the subproblem location that was not selected for the correct answer. |

### 4.2. 0-1 Knapsack

In the 0-1 knapsack problem, there is a knapsack that has a size limit and number of items with different values and size. In this problem, maximization of the total value of items placed in the knapsack within the limited size is the goal. The items cannot be broken into smaller pieces.

### 4.2.1 Creating the Algorithm

Following the four steps of dynamic programming algorithm creation, the first step is to characterize the structure of the optimal solution. For a knapsack K of size s and with items 1 to n to select from and placed in the knapsack, K(n, s), the goal is to determine the optimal solution. Let V[i] and S[i] be the values and size of the ith item, respectively. Let j be some knapsack size less than or equal to s. One potential solution to this problem is finding all combinations of the 1 to n items to be placed in a knapsack and the greatest value from the combination of items that fit in the knapsack of size s will be the solution, however this solution is wasteful with calculations. The answer produced by this method will show which items were used in the knapsack and which were not, so another approach is to ask for each ith item, will it fit in a knapsack of size j? For each ith item it can be checked with K(i, j) and since all ith items are being checked, it can assume K(i-1, j) has already been optimally solved. Since K(i-1, j) is optimally solved for size j it is assumed the knapsack is full and therefore the ith value cannot added, however the ith item can be added to K(i-1, j-S[i]) which will give a knapsack of size j. Assuming K(i-1, j-S[i]) is also optimal, the two values K(i-1, j) and K(i-1, j-S[i])+V[i] are both candidates for the optimal value at K(i, j). The greater of these two candidate values will be the optimal value.

From characterizing how to get the optimal solution, it should be easy to see the recursive algorithm. To find the optimal solution the two other knapsacks need to be considered, which are used in the recursive solution. Recursive solution is K(n, s) = max(K(n-1,s), K(n-1, s-S[n])+V[n]). From this the pseudo code is built.

### 4.2.2 Knapsack NS Algorithm

N = The number of knapsack items to be added.
S = The max size of a knapsack being solving for.
Item = Object array of size n. Holds the value and size of each knapsack item.
K = A knapsack table of size n and s. Holds optimal knapsack subproblems being calculated through out the algorithm.
answer = Final optimal value calculated. Answer to knapsack NS problem.

Pseudo code for phase 1

```
1.  for i = 0 to N
2.      K[i, 0] = 0
```

```
3.  for j = 1 to S
4.    K[0, j] = 0
5.  for i = 1 to N
6.    for j = 1 to S
7.      if Item[i].size() <= j
8.        if K[i-1, j-Item[i].size()] + Item[i].value() > K[i-1, j]
9.          K[i, j] = K[i-1, j-Item[i].size()] + Item[i].value()
10.       else
11.         K[i, j] = K[i-1, j]
12.     else
13.       K[i, j] = K[i-1, j]
14. answer = K[N, S]
```

This algorithm will determine which of K[i-1, s-Item[i].size()] + Item[i].value() and K[i-1, s] is greater for each knapsack subproblem. First the algorithm needs to set valueless knapsacks which are done in lines 1 through 4. Lines 5 and 6 run through each of the knapsack subproblems leading to the final K[N, S] problem. Next for each K[i, j] knapsack the algorithm will check if the K[i-1, j- Item[i].size()] exists by line 7. If it does exist, next the step will check which of K[i-1, j- Item[i].size()] + Item[i].value() and K[i-1, j] is greater in line 8 and set the optimal value in line 9 or 11. If the K[i-1, j-Item[i].size()] value doesn't exist, then K[i-1, j] is set as optimal in line 13. At the end, line 14 records the K[N, S] as the optimal solution.

With the knapsack table filled up, this can be used to find additional information about which items were used in the optimal knapsack. Starting with the K[N, S] knapsack the K[N-1, S] knapsack can be examined. If both knapsacks have the same value then the nth item wasn't needed and the K[N-1, S] knapsack is examined next. If they do not match, then K[N-1, S- Item[i].size()] was used with the value of the n item, so the n item is in the knapsack. It then move to K[N-1, S- Item[i].size()] and examine if its n-1 value is in the knapsack.

Pseudo Code for Phase 2

```
15. i = N
16. j = S
17. while K[i, j] != 0
18.   if K[i, j] = K[i-1, j]
19.     i = i - 1
20.   else
21.     mark ith item as in the knapsack
22.     j = j – Item[i].size()
23.     i = i – 1
```

To find all the values used in the knapsack, first set the Line 15 and 16 set i and j to the values n and s values. Line 17 continues the algorithm so long as the current

knapsack table has a value in it and therefore added a item at some point. Line 18 checks if the K[i, j] = K[i-1, j] and move to the K[i-1, j] value if the do match. Otherwise the algorithm goes to line 20, adds the ith item as in the knapsack and moves to the K[i-1, j-Item[i].size()] value.

With the algorithms explained, next is to examine JDPET and how to use it for this algorithm.

### 4.2.3 JDPET and Knapsack NS

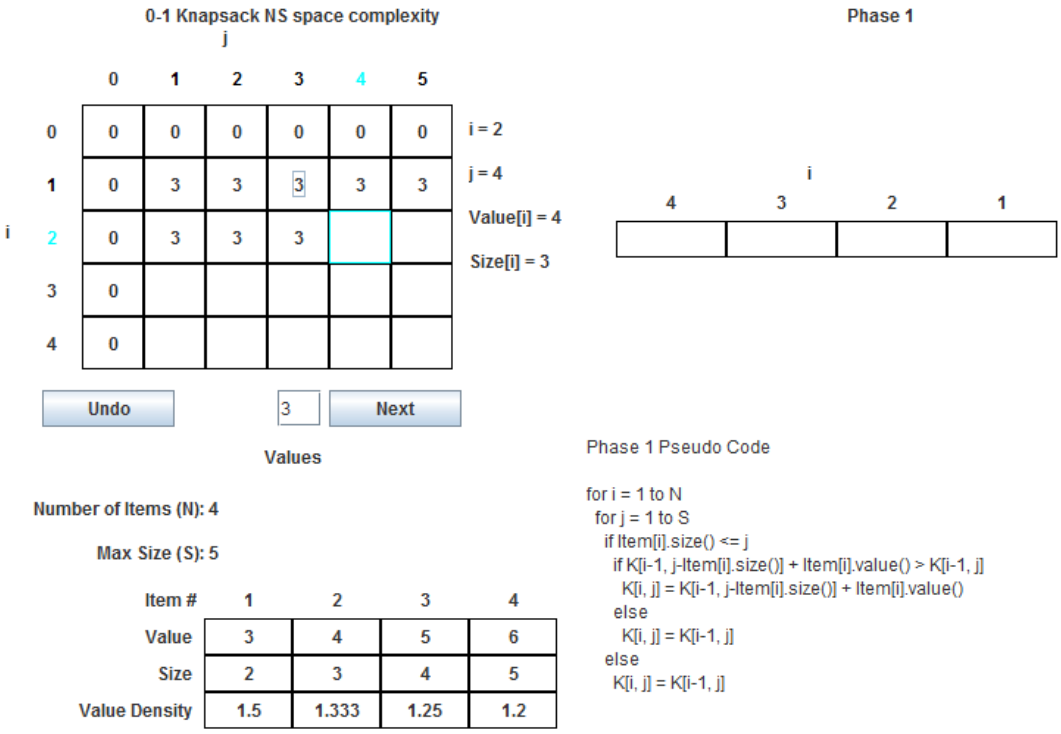


**Figure 1. Knapsack NS Algorithm in Phase 1**

Figure 1 shows the first phase of the knapsack NS problem. The j value indicates the user is considering filling a knapsack of size j while the i value indicates he is considering using items 1 to i to fill the knapsack. For entering values, the value entered in this text field is expected to be the value that will be placed in the current table cell. For selecting a table cell, the value that will be placed in the current cell will be the selected cell's value plus the current item's value which is seen to the right of the table (similar to line 9), though there is an exception to this in which if the table cell above the current is selected then the selected cell's value will be carried over (line 11). Value entered in the final subproblem holds the optimal solution.

0-1 Knapsack NS space complexity

j

| i \ j | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 3 | 3 | 3 | 3 | 3 |
| 2 | 0 | 3 | 3 | 3 | 7 | 7 |
| 3 | 0 | 3 | 3 | 3 | 7 | 7 |
| 4 | 0 | 3 | 3 | 3 | 7 | 7 |

i = 2

j = 5

Size[i] = 3

Undo

Next

Values

Number of Items (N): 4

Max Size (S): 5

| Item # | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Value | 3 | 4 | 5 | 6 |
| Size | 2 | 3 | 4 | 5 |
| Value Density | 1.5 | 1.333 | 1.25 | 1.2 |

Phase 2

i

| 4 | 3 | 2 | 1 |
|---|---|---|---|
| 5 | 5 | | |

Phase 2 Pseudo Code

```
while K[i, j] != 0
 if K[i, j] = K[i-1, j]
  i = i - 1
 else
  mark ith item as in the knapsack
  j = j - Item[i].size()
  i = i - 1
```

**Figure 2. Knapsack NS Algorithm in Phase 2**

Figure 2 shows the second phase of the knapsack NS problem. The goal now is to find the items that have been used in the optimal knapsack answer. Each step the user decrements by one i value and moves to the selected j value. In each step the user selects by text box the j value he wants to move to or press the table cell he wants to move to and the right array will get filled with the selected j value. Changing j values indicates the item was used in the knapsack so the i value will become green (line 21, 22, and 23). This phase will end after landing on a table cell with a zero value.

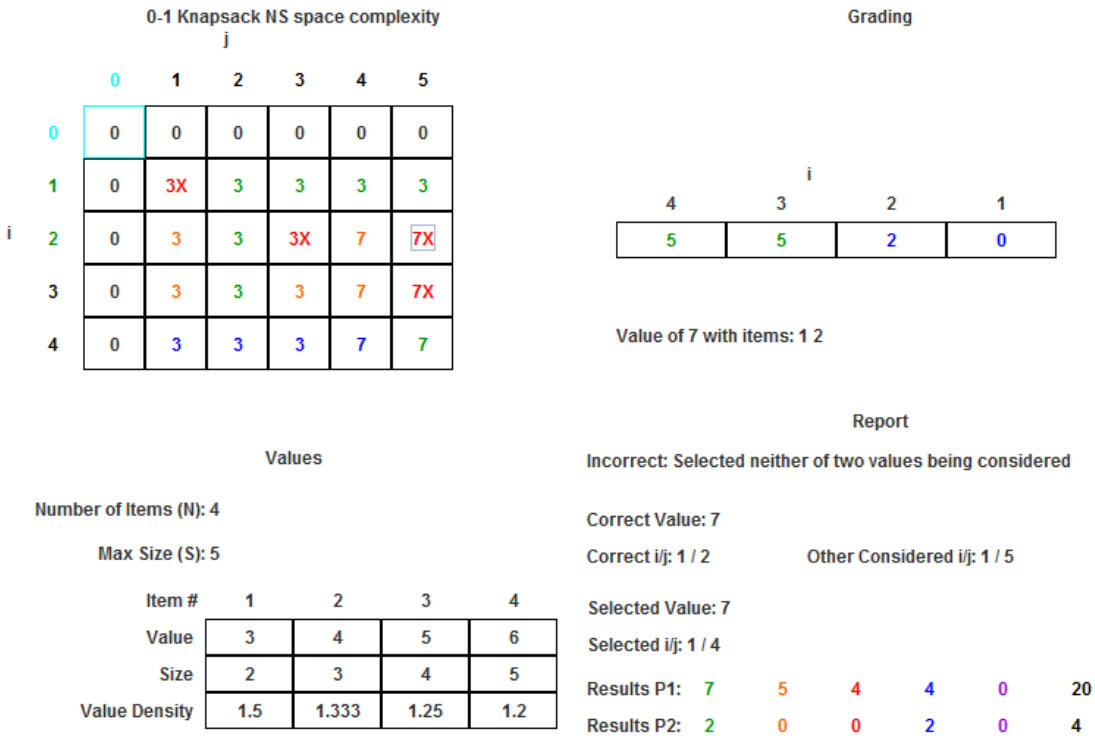


**Figure 3. Knapsack NS Algorithm After Grading**

Figure 3 shows the results for phase 1 and 2 and the user's answer. Answer displayed will show the optimal solution and items placed in the knapsack. Color code, details, and general messages can be seen in Tables 1, 2, and 3.

**Table 3. Knapsack NS General Message**

| General Message | Description |
|---|---|
| Correct | The user has the correct answer for the subproblem step. |
| Correct: Based on previous mistakes | The user has the correct answer for the subproblem based on previous subproblems that are incorrect. True subproblem answer does not match up with the user's answer. |
| Correct: Solved for user | JDPET solved the subproblem step. |
| Correct: Solved for user based on previous mistakes | JDPET solved the subproblem using previous subproblems that are incorrect. True subproblem answer does not match with the solved answer. |
| Incorrect: Selected wrong of two values being considered | The algorithm considers two previous subproblems and uses one to determine the answer to the current subproblem. The user chose the wrong one of these two previous subproblems to solve the current subproblem. |

| | |
|---|---|
| Incorrect: Selected neither of two values being considered | The algorithm considers two previous subproblems and uses one to determine the answer to the current subproblem. The user chose neither of these two previous subproblems to solve the current subproblem. |
| Incorrect: Selected wrong of values with one being considered | Due to item and knapsack size limits, algorithm only has one previous subproblem to look at to determine the current subproblem. The user did not select the subproblem it would have used. |
| Algorithm ended early | Algorithm has ended earlier than it should have with the input problem data, this is due to mistakes made. The true answer is still displayed for this step. |

### 4.2.4 Knapsack S Algorithm

Another algorithm that can be used to solve the 0-1 knapsack problem is the knapsack S algorithm. The knapsack S algorithm saves space as opposed to the knapsack NS algorithm.

N = The number of knapsack items to be added.
S = The max size of a knapsack being solving for.
Item = Object array of size n. Holds the value and size of each knapsack item.
K = A array of size s. Holds optimized subproblems being calculated through out the algorithm.
answer = Final optimal value calculated.

Pseudo Code for Phase 1

```
1. for j = 0 to S
2.    K[0] = 0
3. for i = 1 to N
4.    for j = S to Item[i].size()
5.          if K[j- Item[i].size()] + Item[i].value() > L[j]
6.             K[j] = Item[i].value() + K[j- Item[i].size()]
7. answer = K[s]
```

This algorithm will determine which of K[j- Item[i].size()] + Item[i].value() and K[j] is greater for each knapsack subproblem set iteration. First the algorithm needs to set valueless knapsacks which are done in lines 1 and 2. Lines 3 and 4 run through each of the knapsack sub problems leading to the final K[s] problem. Next for each K[j] of item iteration i, algorithm will check which of K[j- Item[i].size()] + Item[i].value() and K[j] is greater in line 6. If K[j- Item[i].size()] + Item[i].value() is greater it will be updated in line 6, otherwise no change is made. At the end, line 7 records the K[S] as the optimal solution.

Since this algorithm uses only a single array rather than a table, information to find the items used in the knapsack is not present. Therefore this algorithm does not have a second phase. Though it can greatly reduce the storage space needed to calculate the problem, it does so at the sacrifice of finding additional information. With the algorithms explained, next is to examine JDPET and how to use it for this algorithm.

### 4.2.5 JDPET and Knapsack S

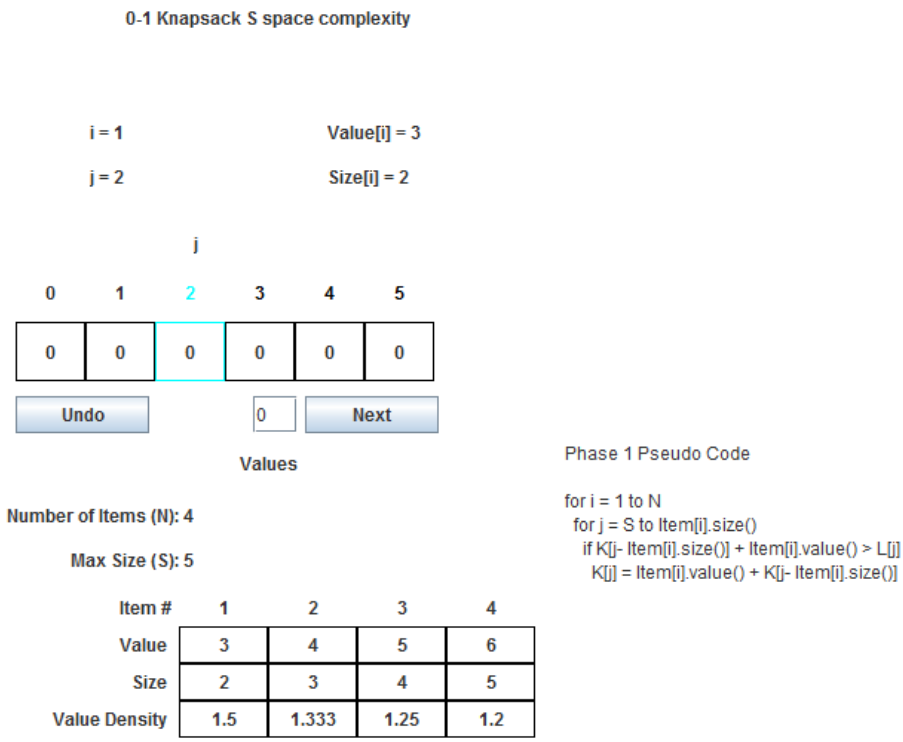


**Figure 4. Knapsack S Algorithm in Phase 1**

Figure 4 shows the first and only phase of the knapsack S problem. The j value indicates the user is considering filling a knapsack of size j while the i value indicates he is trying to fill a knapsack with items 1 to i. For entering values, the value entered in this text field is expected to be the value that will be placed in the current array cell. For selecting a table cell values, the value that will be placed in the current cell will be the selected cell's value plus the current item's value which is seen to the above the array (similar to line 6), though the exception to this in which if the current array cell is selected then the same value will be carried over. The algorithm will loop through the array for each item. After all loops are complete, the algorithm will end and the optimal solution will be in the highest j indexed array cell.

0-1 Knapsack S space complexity

| i \ j | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 3 | 0X | 0X | 0X |
| 2 | 0 | 0 | 3 | 4 | 4 | 7 |
| 3 | 0 | 0 | 3 | 4 | 5 | 7 |
| 4 | 0 | 0 | 3 | 4 | 5 | 7 |

Value of 7

Values

Number of Items (N): 4

Max Size (S): 5

| Item # | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Value | 3 | 4 | 5 | 6 |
| Size | 2 | 3 | 4 | 5 |
| Value Density | 1.5 | 1.333 | 1.25 | 1.2 |

Report

Correct

Correct Value: 7

Correct j: 2

Results: 7 0 3 0 10

**Figure 5. Knapsack S Algorithm After Grading**

In Figure 5 the first thing the user should notice is the knapsack array disappeared and has been replaced by a table. The table shows all answers entered during ith item loop. The user's answer is displayed will show the optimal solution. Color code, details, and general messages can be seen in Tables 1, 2, and 4.

**Table 4. Knapsack S General Message**

| General Message | Description |
|---|---|
| Correct | The user has the correct answer for the subproblem step. |
| Correct: Based on previous mistakes | The user has the correct answer for the subproblem based on previous subproblems that are incorrect. True subproblem answer does not match up with the user's answer. |
| Correct: Solved for user | JDPET solved the subproblem step. |
| Correct: Solved for user based on previous mistakes | JDPET solved the subproblem using previous subproblems that are incorrect. True subproblem answer does not match with the solved answer. |
| Incorrect: Selected wrong of two values being considered | The algorithm considers two previous subproblems and uses one to determine the answer to the current subproblem. The user chose the wrong one of these two previous subproblems to solve the current subproblem. |

| | |
|---|---|
| Incorrect: Selected neither of two values being considered | The algorithm considers two previous subproblems and uses one to determine the answer to the current subproblem. The user chose neither of these two previous subproblems to solve the current subproblem. |

### 4.3 Fractional Knapsack Problem

In the fractional knapsack problem, there is a knapsack that has a size limit and number of items with different values and size. In this problem, maximization of the total value of items placed in the knapsack within the limited size is the goal. The items can be broken into smaller pieces.

#### 4.3.1 Creating the Algorithm

The fractional knapsack problem is the only problem not solved by dynamic programming, it uses a greedy algorithm. Since items can be broken up into any size for this problem, simply grabbing the items with the greatest value/size density would produce the optimal solution to the problem. Since this is not a dynamic programming algorithm, there are no phases.

#### 4.3.2 Fractional Knapsack Algorithm

N = The number of knapsack items to be added.
S = The max size of a knapsack being solving for.
Item = Object array of size n. Holds the value, size, and density of each knapsack item.
Knap = Knapsack being filled. Contains value and size.
answer = Contains the answer to the optimal knapsack.

Pseudo Code for Problem

```
1.  Knap.setValue(0)
2.  Knap.setSizeFilled(0)
3.  while Knap.sizeFilled < S and all items aren't marked as in the knapsack
4.     i = max_Density_Unmarked(Item)
5.     if Item[i].size() + Knap.sizeFilled() < S
6.        mark ith item as in knapsack
7.        Knap.add(Item[i])
8.     else
9.        mark ith item as partially in knapsack
10.       Knap.partialAdd(Item[i])
11. answer = Knap.value();
```

Since the goal is to fill the knapsack with items with the greatest density, that is exactly what this algorithm does. First this algorithm will set a empty knapsack with line 1 and 2. The algorithm then continues to add items in the knapsack until the knapsack is

full or all items are in the knapsack. Line 4 will find the item with the greatest density that is not already in the knapsack. If the whole item fits, it will be marked and added by lines 6 and 7, else if it only partially fits, it will be marked and added partially by lines 9 and 10. After the loop is done, the final answer gets set at line 11.

As with all problems, there are multiple algorithms to complete them. This particular algorithm is less than efficient as there are more efficient greedy algorithms. With the algorithms explained, next step is to examine JDPET and how to use it for this algorithm.

### 4.3.3 JDPET and Fractional Knapsack

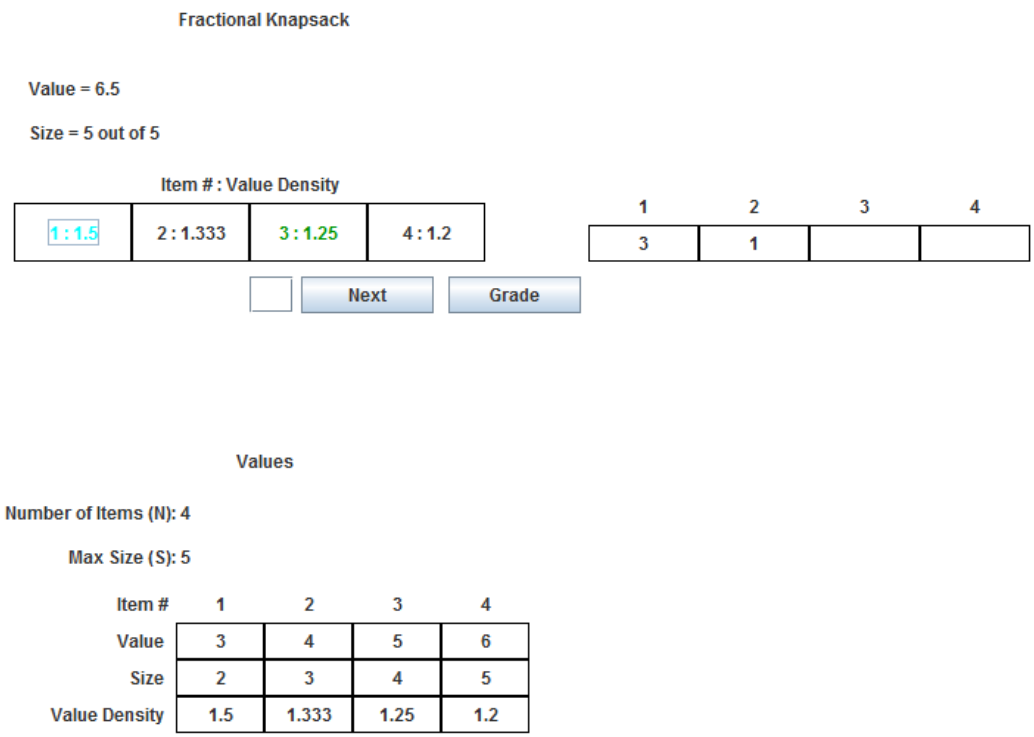


**Figure 6. Fractional Knapsack Algorithm**

Figure 6 shows the problem part of the fractional knapsack problem. The task of this problem is to select all of the items that would be selected for the knapsack and select the items in the proper order. To select an item the user will need to click on the array cell or select the cell index by the text field button. After adding a item to a knapsack, the item in the knapsack table will be colored green or cyan. Green coloring indicates the entire item has been added into the knapsack while cyan indicates it has partially been added into the knapsack. The array to the right will also add the new item to the list of items in the knapsack. Above the knapsack array the size and value information will

change to reflect how much of the knapsack size is filled and how much value is currently placed in the knapsack.

To remove an item from the knapsack, entering the item number in the textbox again or clicking on the array item again will remove the item. The array to the right will update and slide over the order of values entered if needed and the information above the knapsack array will also be updated.

Due to the simplicity of the problem, the hint button will not provide any assistance, instead it will only provide feedback as knapsack items are selected.

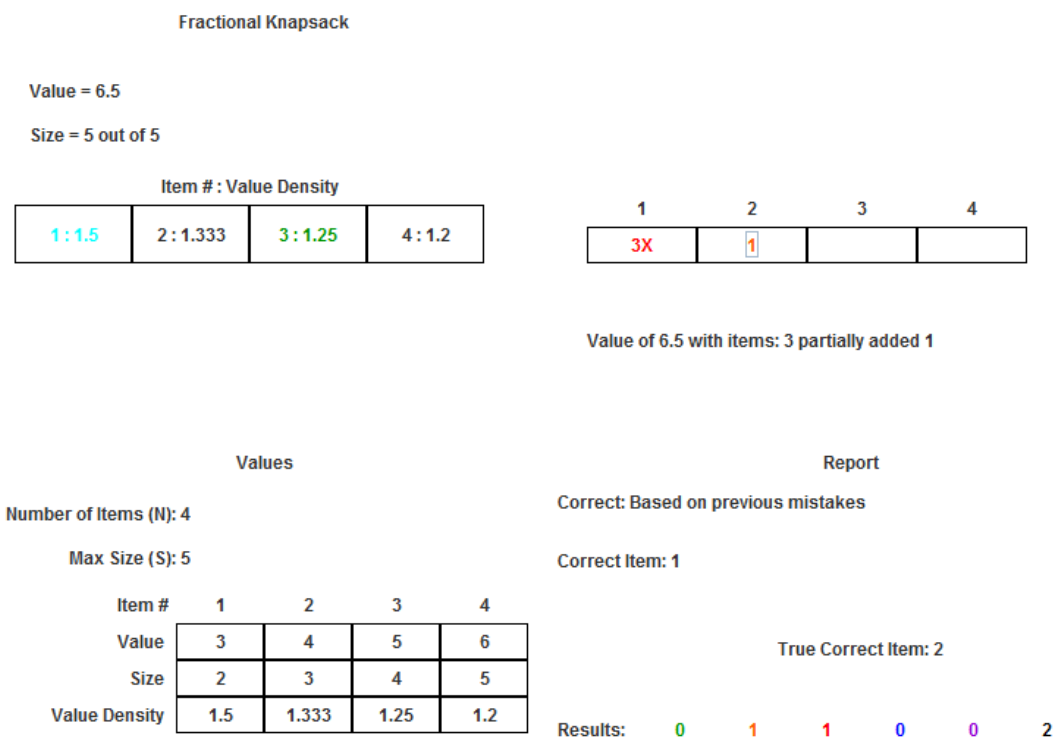


**Figure 7. Fractional Knapsack After Grading**

In Figure 7 shows a number of different colored array cell values in the top right and the user's answer. Answer displayed will show the optimal solution and items placed in the knapsack. Color code, details, and general messages can be seen in Tables 1, 2, and 5.

**Table 5. Fractional Knapsack General Message**

| General Message | Description |
|---|---|
| Correct | The user has the correct answer for the subproblem step. |

| | |
|---|---|
| Correct: Based on previous mistakes | The user has the correct answer for the subproblem based on previous subproblems that are incorrect. True subproblem answer does not match up with the user's answer. |
| Correct: Solved for user | JDPET solved the subproblem step. |
| Correct: Solved for user based on previous mistakes | JDPET solved the subproblem using previous subproblems that are incorrect. True subproblem answer does not match with the solved answer. |
| Incorrect: Selected wrong item | The user selected the wrong item to be placed in the knapsack based on what he has previously selected. |
| Algorithm ended early | Algorithm has ended earlier than it should have with the input problem data, this is due to mistakes made. The true answer is still displayed for this step. |

### 4.4 Longest Common Subsequence Problem

In the longest common subsequence problem there exist two strings. The goal of this problem is to find the greatest common character value length which occurs between the two strings. The subsequence will be in order matching the two other string's characters, but not necessarily matching the characters sequentially.

#### 4.4.1 Creating the Algorithm

Following the four steps, the first step is to characterize the structure of the optimal solution. For a longest common subsequence LCS the goal is to compare string X which has length x and string Y which has length y, LCS(x, y).

Next if there already exist some other optimal LCS using string X and string Y with smaller substrings, this information can use to help find the LCS(x, y). Considering the final characters in string X and Y if they match then they are added to the LCS. Since the final character in X and Y are added, the LCS(x, y) would be increment and carry over the value from a LCS which is missing only the final characters in string X and Y, LCS(x-1, y-1). If the two characters do not match, then a carry over LCS which is not incremented value is needed. Just because the final X and Y character do not match with each other does not indicate that the final X character cannot match with a previous Y character or the final Y character cannot match with a previous X character, so the LCS(x, y) cannot just carry the value over from LCS(x-1, y-1). Since the X and Y character might match with previous characters of other string, it can be assumed that all but the final X or Y character has been used in the solution, LCS(x-1, y) or LCS(x, y-1). At this point the missing final character can simply added in and carry over the optimal value, which would be the greater of LCS(x-1, y) and LCS(x, y-1).

With the problem characterized, next is to determine a recursive algorithm.
if X.char(x) = Y.char(y)

```
    LCS(x, y) = LCS(x-1, y-1) + 1
  else
    LCS(x, y) = max(LCS(x-1, y) , LCS(x, y-1)).
```

Now the dynamic programming pseudo code can be built based of the recursive algorithm.

### 4.4.2 Longest Common Subsequence Algorithm

X = String X
Y = String Y
LCS = Table of X.size() and Y.size(). Holds subproblems being calculated through out the algorithm. Each table cell contains the a numeric value and directional value
answer = Optimal longest common subsequence value.

Pseudo Code for Phase 1

```
1.  for x = 0 to X.size()
2.    LCS[x, 0].setValue(0)
3.  for y = 1 to Y.size()
4.    LCS[0, y].setValue(0)
5.  for x = 1 to X.size()
6.    for y = 1 to Y.size()
7.      if X.char(x) = Y.char(y)
8.        LCS[x, y].setValue(LCS[x-1, y-1].value() + 1)
9.        LCS[x, y].setDirection('↖')
10.     else  if  LCS[x-1, y].value() >= LCS[x, y-1].value()
11.       LCS[x, y].setValue(LCS[x-1, y].value())
12.       LCS[x, y].setDirection('↑')
13.     else
14.       LCS[x, y].setValue(LCS[x, y-1].value())
15.       LCS[x, y].setDirection('←')
16. answer = LCS[X.size(), Y.size()].value();
```

This algorithm is used to determine the optimal solution for each LCS[x, y] subproblem by recursively using LCS with one less string X and/or string Y character. First the algorithm needs to set valueless LCS which are done in lines 1 through 4. Lines 5 and 6 run through each of the LCS subproblems leading to the final LCS[x, y] problem. Line 7 will check if the character in string X and Y match and set the value LCS[x-1, y-1] + 1 if there is a match. If the two characters do not match, then line 10 will find the greater value among LCS[x-1, y] and LCS[x, y-1] set that as the optimal value. If there is a tie with the greater value, then the upper value is selected. Additionally, with each different LCS item used, a direction will be set which is used in the second phase. Finally optimal value is LCS[X.size(), Y.size()] which gets set as the answer in line 16.

With the LCS table filled, this can be used to find additional information about which characters were used in the optimal solution. Starting with the LCS[X.size(), Y.size()] the directions set in phase 1 can be used to find how to get to the answer. Key thing to note is since a characters were only used in LCS when they matched which produces a '↖' direction. Therefore when navigating through the table, every time '↖' is seen the characters are known to be in the optimal solution.

Pseudo Code for Phase 2

```
17. x = X.size()
18. y = Y.size()
19. while x != 0 and y != 0
20.   if LCS[x, y].direction() = '↖'
21.     mark X.char(x) as in LCS
22.     mark Y.char(y) as in LCS
23.     x = x – 1
24.     y = y – 1
25.   else if LCS[x, y].direction() = '↑'
26.     x = x -1
27.   else
28.     y = y – 1
```

Using the directions set in phase 1, phase 2 can quickly run through and mark the characters that are in the optimal solution. Lines 17 and 18 set the values to start as LCS[X.size(), Y.size()]. Line 19 will continue so long as both of the two strings still have characters to consider: if either one runs out of characters then no more matches will exist. Line 20 finds a '↖' direction, marks both characters of string X and Y as in the LCS and moves to the proper direction. Line 25 checks for the '↑' direction and moves in the proper direction at line 26, else move in the other direction at line 28.

### 4.4.3 JDPET and Longest Common Subsequence

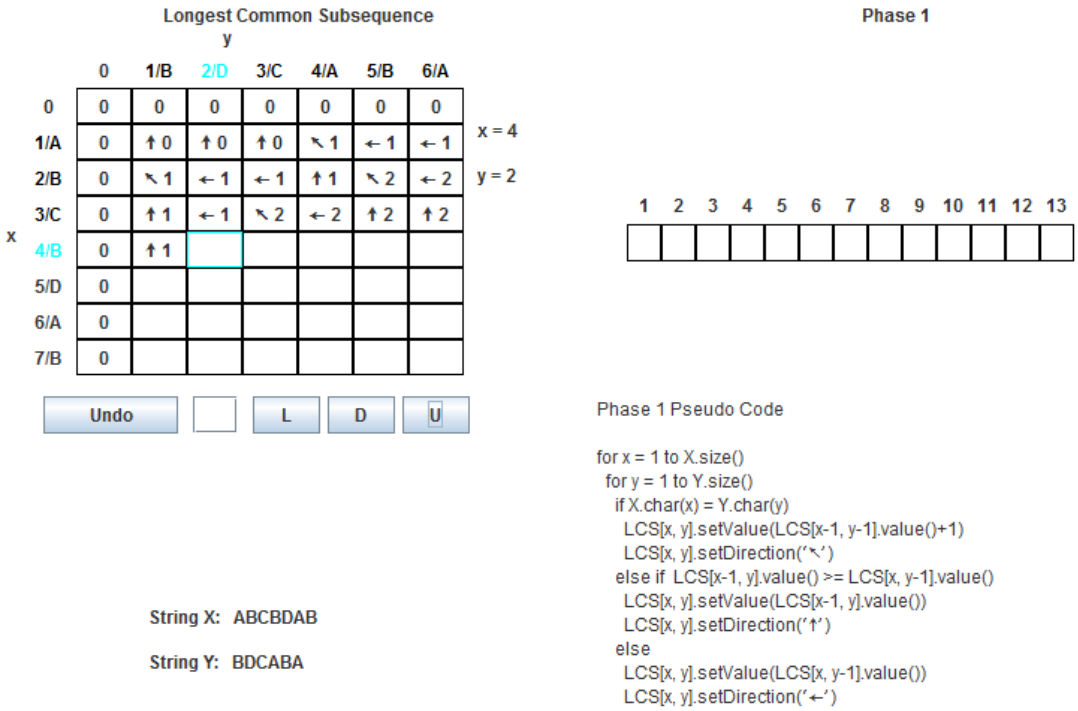


**Figure 8. LCS Algorithm in Phase 1**

Figure 8 shows the first phase of the longest common subsequence problem. The user should notice two values in the table headers, the first number indicates the index of the array while the second number indicates the character value of the corresponding string. Values the user can enter for each subproblem are be L, U, and D. The user can type in any of these letters and hit enter, click the buttons with these letters, or select the left, upper, or upper left table cell. Selecting L indicates the user is carrying over a value left of the current table cell and will also place a '←' character in the current table cell, U indicates the user is carrying over a value from the above cell and will also place a '↑' character in the current table cell, while D indicates a character match and carries the value from the diagonal upper left table cell, increments the value by 1, and will also place a '↖' character in the current table cell. In addition to a arrow, a value will also be placed in each table cell. The optimal solution will be the value in the final table cell answered.

Longest Common Subsequence

y

| x | 0 | 1/B | 2/D | 3/C | 4/A | 5/B | 6/A |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1/A | 0 | ↑ 0 | ↑ 0 | ↑ 0 | ↖ 1 | ← 1 | ← 1 |
| 2/B | 0 | ↖ 1 | ← 1 | ← 1 | ↑ 1 | ↖ 2 | ← 2 |
| 3/C | 0 | ↑ 1 | ← 1 | ↖ 2 | ← 2 | ↑ 2 | ↑ 2 |
| 4/B | 0 | ↑ 1 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ← 3 |
| 5/D | 0 | ↑ 1 | ↖ 2 | ← 2 | ↑ 2 | ↑ 3 | ↑ 3 |
| 6/A | 0 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ← 3 | ↖ 4 |
| 7/B | 0 | ↖ 1 | ↑ 2 | ↑ 2 | ↑ 3 | ↖ 4 | ↑ 4 |

x = 3

y = 3

Undo | | L | D | U

Phase 2

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| U | D | U | D | L | | | | | | | | |

Phase 2 Pseudo Code

```
while x != 0 and y != 0
 if LCS[x, y].direction() = '↖'
  mark X.char(x) as in LCS
  mark Y.char(y) as in LCS
  x = x - 1
  y = y - 1
 else if LCS[x, y].direction() = '↑'
  x = x - 1
 else
  y = y - 1
```

String X: ABCBDAB

String Y: BDCABA

**Figure 9. LCS Algorithm in Phase 2**

The goal in phase 2 is to find the matching string characters that are used in the longest common subsequence. During each step the user selects the direction to move, this should be corresponding to the arrows. Selecting the D or upper left direction will mark both the current x and y characters as matching and part of the longest common subsequence. This phase will end after arriving at a directionless table cell, when x = 0 and/or y = 0.

Longest Common Subsequence

y

| x | 0 | 1/B | 2/D | 3/C | 4/A | 5/B | 6/A |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1/A | 0 | ↑ 0 | ↑ 0 | ↑ 0 | ↖ 1 | ← 1 | ← 1X |
| 2/B | 0 | ↖ 1 | ← 1 | ← 1 | ↑ 1 | ↖ 2 | ← 2 |
| 3/C | 0 | ↑ 1 | ← 1X | ↖ 2 | ← 2 | ↑ 2 | ↑ 2 |
| 4/B | 0 | ↑ 1X | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ← 3 |
| 5/D | 0 | ↑ 1 | ↖ 2 | ← 2X | ↑ 2 | ↑ 3 | ↑ 3 |
| 6/A | 0 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ← 3X | ↖ 4 |
| 7/B | 0 | ↖ 1 | ↑ 2 | ↑ 2 | ↑ 3 | ↖ 4 | ↑ 4 |

Grading

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| U | D | U | D | L | D | L | D | | | | | |

Length of 4 with characters BCBA

Report

Incorrect: Wrong direction selected

Correct Value: 1

Correct Direction: D

Selected Value: 1

Selected Direction: L

Results P1: 32 0 5 5 0 42

Results P2: 5 0 0 3 0 8

String X: ABCBDAB

String Y: BDCABA

**Figure 10. LCS Algorithm After Grading**

Figure 10 shows the results for phase 1 and 2 and the user's answer. Answer displayed will show the optimal solution and characters in the LCS. Color code, details, and general messages can be seen in Tables 1, 2, and 6.

**Table 6. LCS General Message**

| General Message | Description |
|---|---|
| Correct | The user has the correct answer for the subproblem step. |
| Correct: Based on previous mistakes | The user has the correct answer for the subproblem based on previous subproblems that are incorrect. True subproblem answer does not match up with the user's answer. |
| Correct: Solved for user | JDPET solved the subproblem step. |
| Correct: Solved for user based on previous mistakes | JDPET solved the subproblem using previous subproblems that are incorrect. True subproblem answer does not match with the solved answer. |
| Incorrect: Wrong direction selected | In phase 1, the user selected the wrong direction to carry previous subproblem data from to answer the current subproblem. In phase 2, the user selected the wrong direction to move. |

| | |
|---|---|
| Algorithm ended early | Algorithm has ended earlier than it should have with the input problem data, this is due to mistakes made. The true answer is still displayed for this step. |

### 4.5 Longest Increasing Subsequence Problem

In the longest increasing subsequence problem there exists an array or list of integers. The integers can clearly be quantified and compared with each other to determine which of the two values is greater. In this problem the goal is to determine the greatest length of increasing subsequence values in the array.

#### 4.5.1 Creating the Algorithm

For creating the dynamic programming algorithm, the first step is to characterize the structure of the optimal solution. For a longest increasing subsequence LIS the goal is to compare all of the characters in input integer array A which has a size of s, LIS(A). To find the optimal value, the value that would be selected with LIS(A) would be the greatest subsequence value ending with the character A[i], LIS(i). For LIS(i), if the ith integer is greater than any of the previous integer values in the array, then the ith integer's longest increasing subsequence must be at least one greater than the previous integer's longest increasing subsequence. Since trying to find the optimal LIS(i) value, it will be compared for all previous integer 1 to i-1 to find the greatest longest increasing subsequence value. Because LIS(A) needs to find the max longest increasing subsequence value of all the integer, then LIS(i) needs to run from 1 to s.

The next step is to develop a recursive algorithm based on the characterization of the optimal solution. Since for each integer, a comparison is made to the LIS of all previous integers that are smaller, this formula is obtained.

For each $1 \leq i < s$ and $A[s] > A[i]$
$LIS(s) = \max(A[i]+1)$

This will produce the LIS for each character value. The optimal solution for the problem is

For each $1 \leq i \leq s$
$LIS(A) = \max(LIS(i))$

#### 4.5.2 Longest Increasing Subsequence Algorithm

A = Input integer array
LIS = One dimensional array of size s. Holds subproblems being calculated through out the algorithm.
answer = Answer that holds the optimal solution for LIS(A).

Pseudo Code for Phase 1

1. for i = 1 to A.size()

```
2.    LIS[i] = 1
3.    for j = i-1 to 1
4.      if A[i] > A[j]
5.        if LIS[i] <= LIS[j]
6.          LIS[i] = LIS[j] + 1
```

This algorithm is produced by the recursive solution. The algorithm begins by creating a loop to find the LIS for each substring of A up to character i with line 1. Lines 3 through 6 will compare the LIS[i] with all previous LIS and update the solution any time character i is greater and the solution is greater, therefore finding the max optimal solution. By default the solution is set to 1 at line 2, in the event that character i is not greater than any other previous character. This will produce the optimal solution for each character.

Note that the next step for finding the optimal solution for the problem is also the first step for phase 2, so that has been placed in phase 2, though it would be equally valid in phase 1. In the next phase the goal is to find all of the character values used to create the LIS. Starting with the greatest LIS character value, the same order is followed in which its optimal value was constructed to find all of the character values used.

Pseudo Code for Phase 2

```
7.  LISIndex = 1
8.  for i = 2 to A.size()
9.    if LIS[i] > LIS[LISIndex]
10.     LISIndex = i
11. i = LISIndex
12. j = i-1
13. answer = LIS[i]
14. mark A[i] as part of LIS
15. while LIS[i]  > 1
16.   if LIS[i] = LIS[j] – 1
17.     i = j
18.     mark A[i] as part of LIS
19.   j--
```

In phase 2, the algorithm will determine which character values were used in the longest increasing subsequence. The first step is shared with phase 1, which is determining the greatest LIS value and which character holds this value. Lines 7 through 10 will check through the LIS optimal values at each character and will mark the greatest value index in variable LISIndex. After, lines 11 through 14 will set up for the remainder of phase 2, save the optimal LIS answer, and mark the first character value that is in the LIS.

Next Lines 15 through 19 will find and mark the remaining values. Based on the way the LIS array was constructed, finding the first LIS[j] value that is 1 less than the LIS[i] value where A[i] > A[j] is the character which was used in the optimal solution, so this value will be marked and checked against next. The algorithm will continue so long as the value isn't 1 as the first character in the sequence must be 1.

This particular longest increasing subsequence algorithm is not the most efficient longest increasing subsequence dynamic programming algorithm. It does still represent a possibility of an algorithm that can be created by dynamic programming.

### 4.5.3 JDPET and Longest Increasing Subsequence

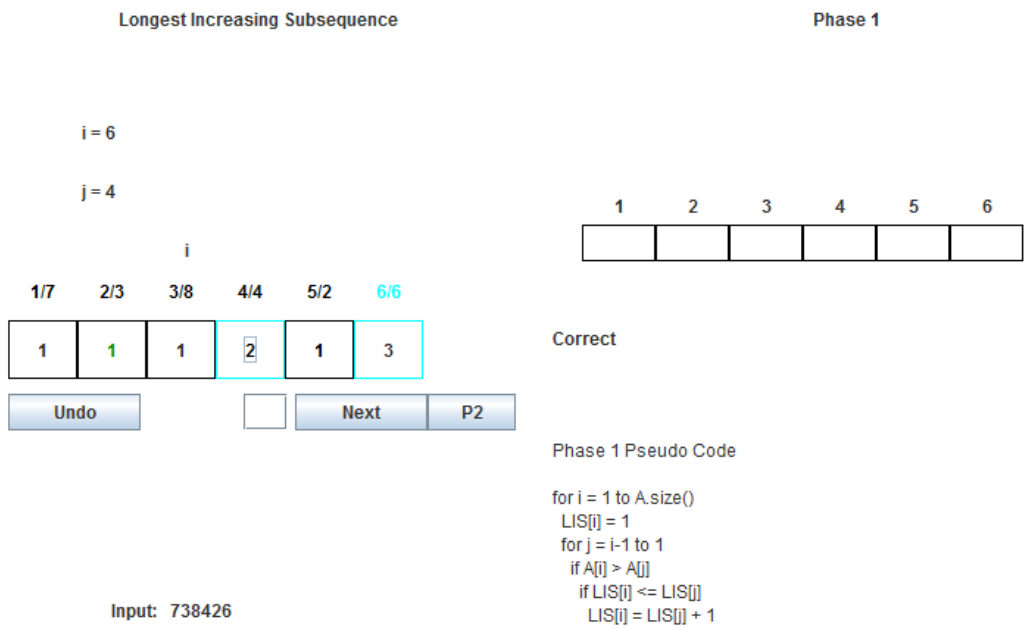


**Figure 11. LIS Algorithm in Phase 1**

Figure 11 shows the first phase of the longest increasing subsequence problem. The i value indicates the index of the numeric character and subproblem the user is solving for while the j value indicates another subproblem which is currently used to solve the i subproblem. The user should also notice the array header for the LIS array contains two numbers. The first number indicates the index of the array while the second number indicates the character value of the input string.

The user should notice one or two cyan colored empty boxes, as well as an i value colored cyan, this indicates the current subproblem being solved as well as what other

subproblem has been used so far in the calculation. The value expected to be entered by the text box or selected by the table array is the subproblem which is expected to solve the current subproblem. The user indicates that a subproblem is complete by selecting the next subproblem to be solved. This will continue until all of the array cells have been filled, and then the user will have the option to move onto phase 2.

Upon starting the algorithm the user should notice nothing is selected as flow of control is different than other algorithms in JDPET. Since the current subproblem checks itself against all previous subproblems it would be more engaging for the user to select when and if the current subproblem is updated or not. To get started the user must select the first array cell to indicate he is starting the first subproblem. In this algorithm, JDPET requires the user to indicate when a subproblem is finished and when he is starting a new subproblem. To indicate when the user is starting a new subproblem he must select the next i value or subproblem and at the very last subproblem he must select the P2 button to indicate the start of phase 2.

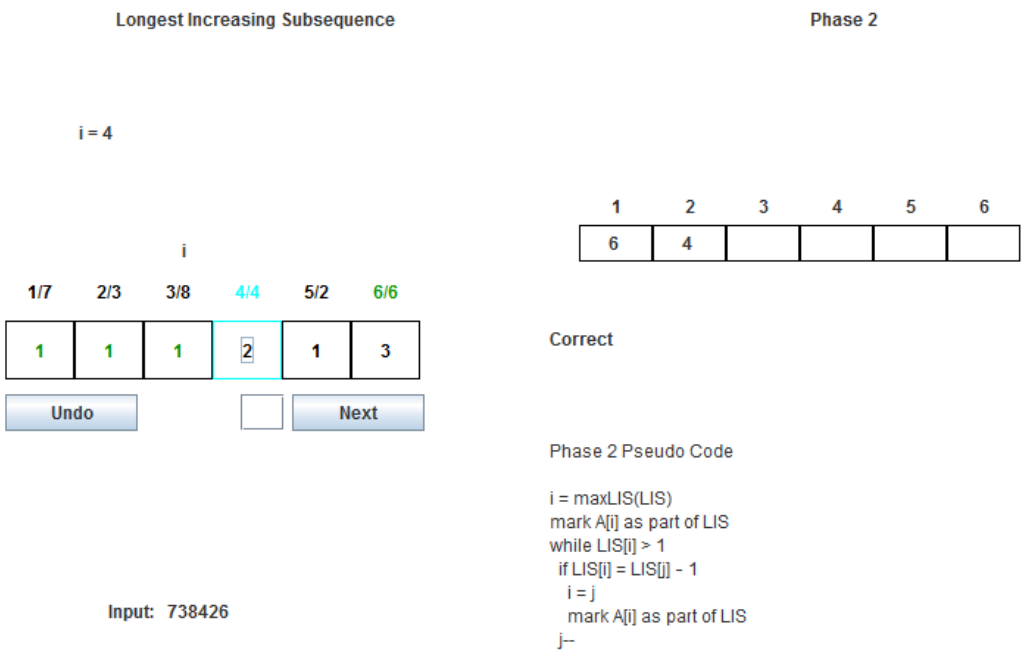


**Figure 12. LIS Algorithm in Phase 2**

The goal in phase 2 is to find the character values that are used in the longest increasing subsequence. In the first step there are no cyan colored items as the user needs to select which value he starts at. In each following step the user can select any array cells left of the current i array cell. Selecting an array cell indicates that the character value at that index is part of the longest increasing subsequence, when moving away from

this array cell, its index will remain green colored. The algorithm will end after selecting a value of 1.

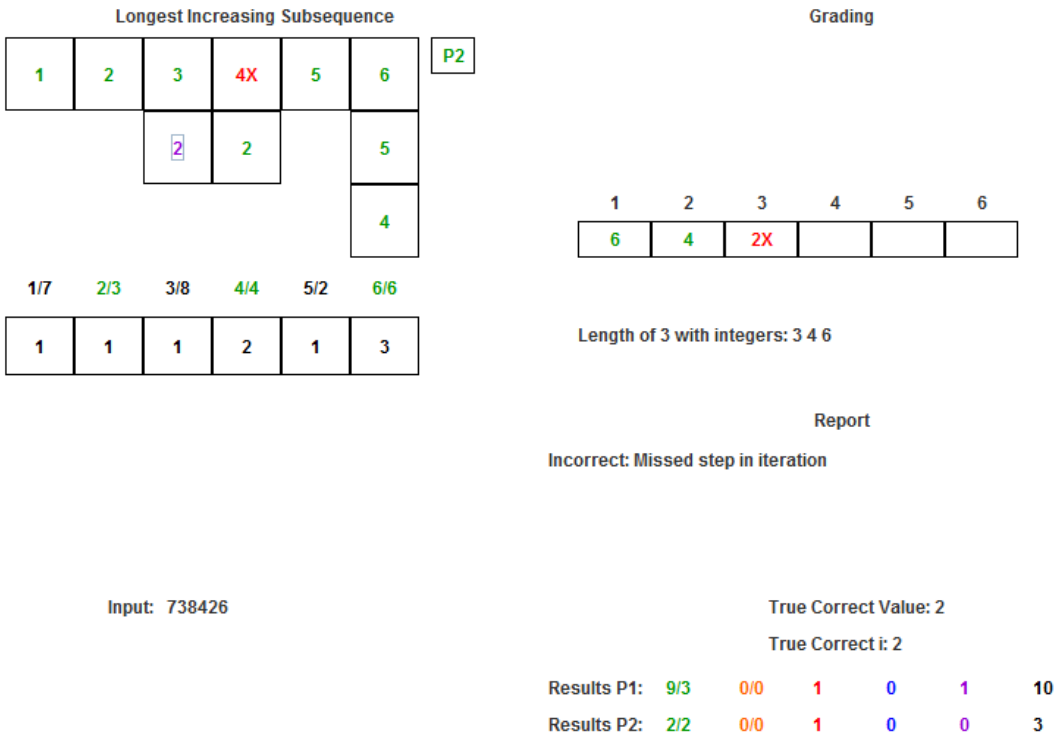


**Figure 13. LIS After Grading**

Figure 13 shows the results for phase 1 and 2 and the user's answer. Answer displayed will show the optimal solution and integers in the LIS. Color code, details, and general messages can be seen in Tables 1, 2, and 7.

**Table 7. LIS General Message**

| General Message | Description |
|---|---|
| Correct | The user has the correct answer for the subproblem step. |
| Correct: Based on previous mistakes | The user has the correct answer for the subproblem based on previous subproblems that are incorrect. True subproblem answer does not match up with the user's answer. |
| Correct: Solved for user | JDPET solved the subproblem step. |
| Correct: Solved for user based on previous mistakes | JDPET solved the subproblem using previous subproblems that are incorrect. True subproblem answer does not match with the solved answer. |
| Incorrect: Wrong value selected | The user selected the wrong previous subproblem to fill the current subproblem. |

| | |
|---|---|
| Incorrect: Should move on to next value | The current subproblem should be finished, but the user continued to update its value. |
| Incorrect: Should have remained at previous value | The previous subproblem the user was working on was not complete, more updates were needed. |
| Incorrect: Selected wrong max value | The user selected a subproblem with the correct max value, but he did not select the correct subproblem. |
| Incorrect: Did not select max value | The user selected a subproblem that did not contain the max value. |
| Incorrect: Selected wrong of one value less | The user selected a subproblem which value is one less than the current subproblem's value, but he did not select the correct subproblem. |
| Incorrect: Did not select one value less | The user did not select a subproblem value which is one less than the current subproblem's value. |
| Incorrect: Missed step in iteration | Current ith subproblem is missing some steps in the ith iteration that are present in the true answer. True answer is displayed. |
| Algorithm ended early | Algorithm has ended earlier than it should have with the input problem data, this is due to mistakes made. The true answer is still displayed for this step. |

### 4.6 Shuffle Problem

The shuffle problem contains three strings and the goal of the problem is to determine if the first two string characters can be shuffled together to become the third string. When shuffled together the two strings will maintain their ordering of character values when combined into the third string. A example of this problem is if there are two hands of cards which are shuffled together once, is there any way possible it can equal a third hand of cards?

#### 4.6.1 Creating the Algorithm

Following the four steps, the first step is to characterize the structure of the optimal solution. For the shuffle problem contains string A of length a, string B of length b, and the shuffle string C of length c and the goal to find out if string A and B can be shuffled into string C, S(a, b, c). S(a, b, c) will simply produce a true or false answer if a shuffle exists. The first thing to note that before continuing with the shuffle is to make sure the length a and b match the length c, otherwise there would be too many or too few characters to shuffle into string C which would not produce a shuffle. Next, notice that if the last character of C doesn't match the last character of either A or B, then C is not a shuffle of A or B. If it matches only one, for example A, and if the shuffle S(a-1, b, c-1) exists, then the last character can be added from A into a shuffle and produce a new shuffle, therefore the S(a, b, c) exists.

With the characterized solution, next step is producing the recursive algorithm. Based on the characterized solution,

```
if (a+b) = c
    if A[a] = C[c] and S(a-1, b, c-1) is a shuffle
      S(a, b, c) = S(a-1, b, c-1)
    if B[b] = C[c] and S(a, b-1, c-1) is a shuffle
      S(a, b, c) = S(a, b-1, c-1)
```

Next is to construct the algorithm based on the recursive solution.

### 4.6.2 Shuffle Algorithm

A = String A
B = String B
C = String C, shuffled string which string A and B are being checked against
S = Table of size a and b. Holds subproblems being calculated through out the algorithm.
answer = Solution to shuffle problem

Pseudo Code for Phase 1

```
1.  if A.size()+B.size() != C.size()
2.    answer = 'N'
3.  else
4.    for a = 0 to A.size()
5.      for b = 0 to B.size()
6.        if a = 0 and b = 0
7.          S[a, b] = 'E'
8.        else
9.          if a > 0 and A.char(a) = C.char(a+b) and S[a-1, b] != 'N' or null
10.           S[a, b] = 'A'
11.         else if b > 0 and B.char(a) = C.char(a+b) and S(a, b-1) != 'N' or null
12.           S[a, b] = 'B'
13.         else
14.           S[a, b] = 'N'
15.   answer = S[A.size(), b.size()]
```

This algorithm covers the shuffle problem. To start, the algorithm checks if a and b match c to continue with line 1, otherwise it ends indicating there is no shuffle. Lines 4 and 5 loop through each of the subproblems to find if there is a shuffle. Line 6 and 7 are the first steps which set a special marker that allows the algorithm to continue for future steps. After the first step, lines 9 through 14 will check the next shuffle character of C matches the current A or B character and it the previous shuffle is true, if so it will it will mark the shuffle as true with characters 'A' or 'B' otherwise if false with character 'N'. In the event that both A and B match the shuffle character C, then A will be chosen as the shuffle character. Finally line 15 will set the answer for the shuffle problem.

Two things to note, first it was mentioned before that the size of string A and B must match string C's size for the shuffle to be true, so once this is confirmed the remainder of the algorithm uses a+b to cover the c value to reduce extra variables. The other thing to mention is that this algorithm uses 'A' and 'B' for true values instead of just a 'T' value. The reason for this is to help with the additional information in phase 2. With the table built, the 'A' and 'B' values can use to find which characters with string A and B match with which characters in string C.

Pseudo Code for Phase 2

```
16. a = A.size()
17. b = B.size()
18. if answer != 'N' // Note ignored be JDPET
19.   while a+b > 0
20.     if S[a, b] = 'A'
21.       C[a+b] marked as A
22.       a = a - 1
23.     else   // Note JDPET check if second result matches B
24.       C[a+b] marked as B
25.       b = b - 1
```

In this phase the goal is to match the characters from string A and B to C. To begin with it is verify that there exists a shuffle answer with line 18 before continuing. Line 19 will continue the algorithm so long as A and B have not been matched with all of the shuffle characters. The remaining lines will check the table characters and depending on if the value 'A' or 'B' is found, it will mark that string to match the current C string character then continue on to the next character in string A and C or B and C.

Two things to mention, since JDPET always produces a shuffle, step 18 is not checked. Also to give better feedback, line 23 does check if S[a, b] = 'B' rather than just assuming. In both cases, this would indicate that the user has arrived at a table cell S[a, b] that has the value 'N.' Based on the way the algorithm is constructed and that JDPET always produces a answer, the user should not arrive at a 'N' value unless he made a mistake and therefore it will be indicated as a mistake with no correct answer.

### 4.6.3 JDPET and Shuffle

Shuffle

| a \ b | 0 | 1/S | 2/D | 3/C | 4/H | 5/S | 6/H |
|---|---|---|---|---|---|---|---|
| 0 | E | N | N | N | N | N | N |
| 1/H | A | B | N | N | N | N | N |
| 2/S | A | N | N | N | N | N | N |
| 3/C | A | B | N | N | N | N | N |
| 4/S | A | A | B | N | N | N | N |
| 5/D | N | A | A | B | B | B | B |
| 6/H | N | N | N | A | N | A | N |
| 7/S | N | N | N | A | | | |

a = 7

b = 4

c = 11

Undo | | N | A | B

Phase 1

| a+b | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | H | S | C | S | S | D | D | C | H | S | H | S | H |
| | | | | | | | | | | | | | |

String A: HSCSDHS

String B: SDCHSH

String C: HSCSSDDCHSHSH

Phase 1 Pseudo Code

```
for a = 0 to A.size()
 for b = 0 to B.size()
  if a = 0 and b = 0
   S[a, b] = 'E'
  else
   if a > 0 and A.char(a) = C.char(a+b) and S[a-1, b] != 'N' or null
    S[a, b] = 'A'
   else if b > 0 and B.char(a) = C.char(a+b) and S(a, b-1) != 'N' or null
    S[a, b] = 'B'
   else
    S[a, b] = 'N'
```

**Figure 14. Shuffle Algorithm in Phase 1**

Figure 14 shows the first phase of the shuffle problem. The focus of this phase is to find if a shuffle exist. If the last table cell is A or B then this indicates there is a solution to the problem. JDPET will always create a problem that has a solution, this is to ensure a second phase is produced otherwise it will not occur if a shuffle does not exist. For each subproblem the user is trying to determine if a shuffle exists at the current subproblem and if string A or B's character at index a or b will be used in the shuffle. The user should notice that the table headers for a and b contain both a number and a letter. The number indicates the index while the letter indicated the character at that index.

Values selected for each subproblem are N, A, and B. N indicates that no shuffle exists while A and B indicates their character was used at the subproblem. The user can select these values by the text field, button, or table. Selecting the current table cell will produce N, selecting the above table cell produces A, while selecting the left table cell produces B.

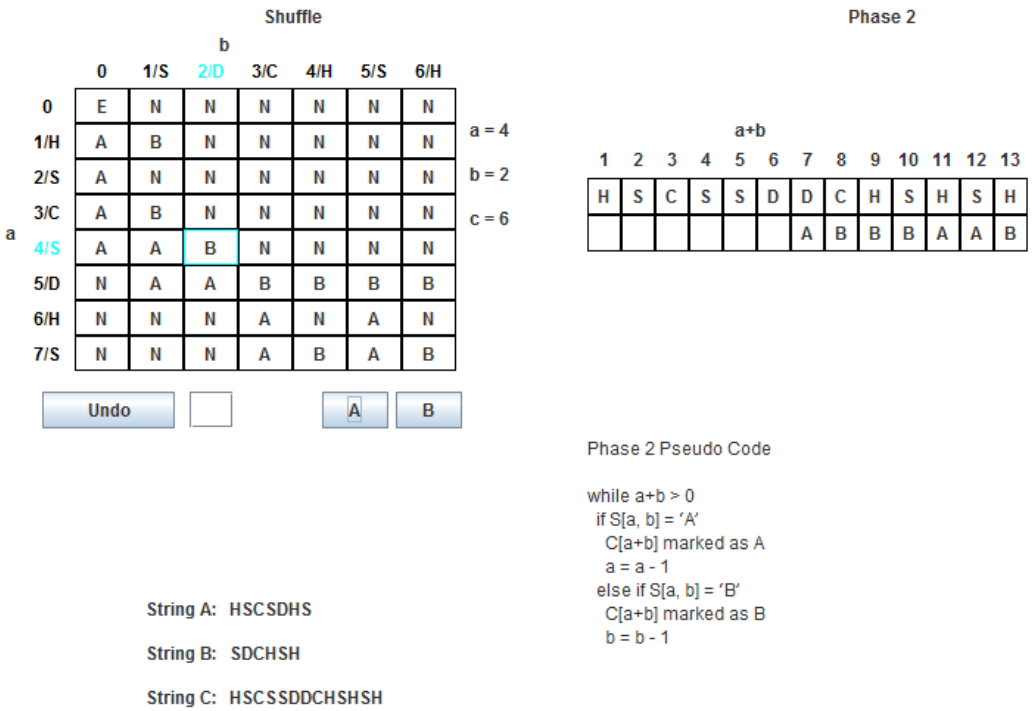


**Figure 15 Shuffle Algorithm in Phase 2**

The goal in phase 2 is to find which string A and string B characters are used in the shuffle with the corresponding shuffle string characters. If there is no solution, then this phase is skipped, however to always have this phase occur, JDPET generates problems that always contain a solution. Even if the final table cell is N, phase 2 will occur because there should be a solution that leads into phase 2.

In each step the user can move in the direction of up or left. Selecting A will move one table cell up while B moves to the left. Additionally, the user can press the left or upper table cell of the current table cell to move in that direction, which will mark A for up or B for left. After coming to E value in the table, the algorithm will end.

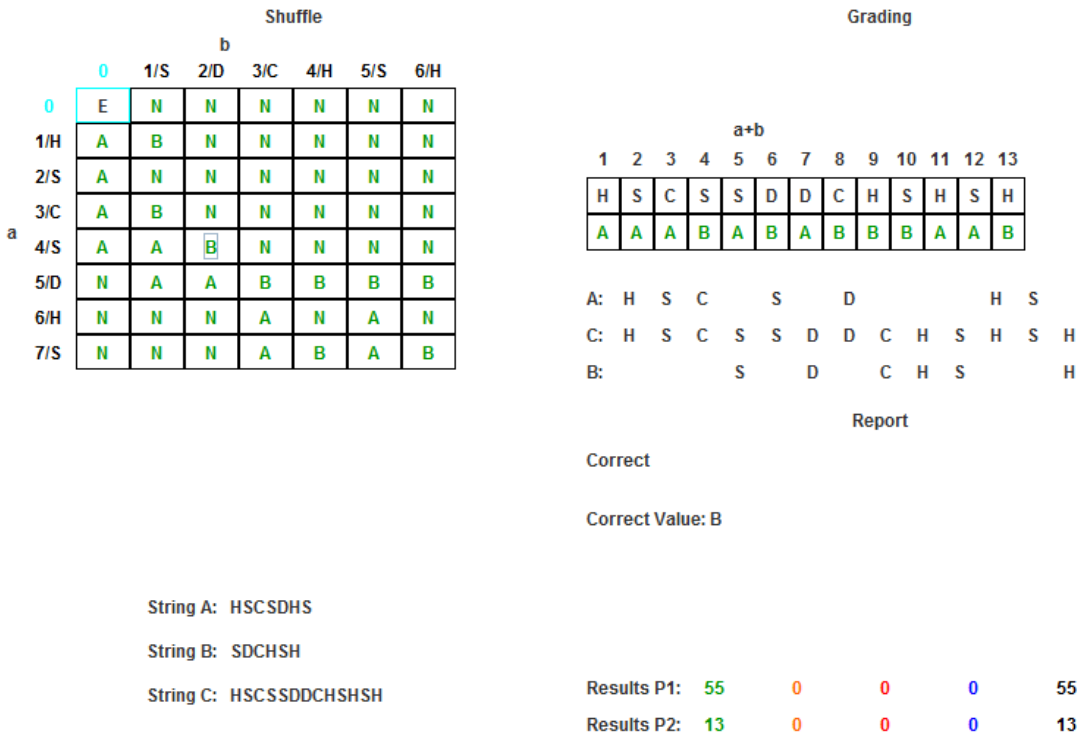


**Figure 16. Shuffle Algorithm After Grading**

Figure 16 shows the results for phase 1 and 2 and the user's answer. Answer displayed will show the matching characters between string A, B, and C. Color code, details, and general messages can be seen in Tables 1, 2, and 8.

**Table 8. Shuffle General Message**

| General Message | Description |
|---|---|
| Correct | The user has the correct answer for the subproblem step. |
| Correct: Based on previous mistakes | The user has the correct answer for the subproblem based on previous subproblems that are incorrect. True subproblem answer does not match up with the user's answer. |
| Correct: Solved for user | JDPET solved the subproblem step. |
| Correct: Solved for user based on previous mistakes | JDPET solved the subproblem using previous subproblems that are incorrect. True subproblem answer does not match with the solved answer. |
| Incorrect: Wrong of two values | A or B is the correct answer, but the user selected the wrong one. |
| Incorrect: Selected neither value when one was selected. | The user selected N when the correct answer was A or B. |

| | |
|---|---|
| Incorrect: Selected wrong value when neither are selected. | The user selected A or B when the correct answer was N. |

# 5. Student Reviews

To get some feedback on JDPET, it was tested with a couple of students. The students had some difficulty using JDPET. Problems included they were unfamiliar with the problems, they were uncertain about what they were solving for, and they were uncertain with some of the interaction with JDPET. JDPET wasn't designed to introduce students to the problems so it is understandable that students are unfamiliar and uncertain for what they are solving for as they had not had the proper teaching supplements before being introduced into JDPET. For the interactions, it has been learned JDPET is not the most intuitive in this aspect, though part of this could also relate to students being unfamiliar with the problems. To help with the interactions the users will be provided with an instruction guide and JDPET will now include a pop up instruction window. These problems are not exclusive to JDPET, other tools such as TRAKLA can also be confusing for users who are unfamiliar with the problem or interaction. Though JDPET was difficult for the testers to use, with proper introduction to the problems and instructions, the students should be able to use JDPET.

# 6. Future Work

Though JDPET helps students learn by means of actively imitating the algorithm or viewing worked examples, other features could be added into JDPET in the future. JDPET could allow for student generated problems within reason. As JDPET makes use of touch buttons, JDPET could be adapted for mobile device use. Though JDPET has only a few dynamic programming algorithms, it can be adapted to include more dynamic programming algorithms, and non-dynamic programming algorithms based on similar problems to show the differences. Before adding new algorithms however, the JDPET should have the code refactored to build a better framework. Building a better framework will make it much easier to add new dynamic programming algorithms.

JDPET could also use more user testing and refinement. JDPET could be tested against students in a continual study to improve its interaction and ease of learning how to use the product. Additionally through these tests, the JDPET instruction manual can be refined and improved. Furthermore, JDPET could be tested on various platform and work environments to better improve its portability.

# 7. Conclusion

JDPET, by means of visualization and interactive examples, assists students to learn about specific dynamic programming algorithms which illustrate the nature of dynamic programming. In conjunction to other teaching supplements, JDPET should give students a strong foundation on dynamic programming. This foundation Could potentially lead students to create their own dynamic programming algorithms.

Although there are other educational tools that educate users about specific algorithms or specific dynamic programming algorithms, none does so like JDPET. JDPET takes the approach of student solvable problems while other tools merely show visual step by step examples. The visual step by step process is quite common as there are a number of websites and programs that take this approach[4][9][11].

One notable educational tool that educates differently is AlViE[9]. AlViE is software that educates students about many algorithms, more than JDPET has to offer, but JDPET does present mostly different algorithms than AlViE. AlViE does have a few dynamic programming algorithms, with only the longest common subsequence in common with JDPET. Though both programs have the same algorithm, they present it by different means. AlViE presents the algorithm and a visual representation of the table filling and does so by showing students a step by step process of how the table gets filled as well as where the code is during each step. JDPET on the other hand allows the users to fill the table themselves, although they still have the option of seeing a worked example by having JDPET solve the problem at each step. Although AlViE shows students good examples of how the algorithm runs, it does not give the students opportunity to test their understanding of the algorithms like JDPET does.

Tools such as TRAKLA also allow students to solve problems much like JDPET but have one flaw that JDPET fixes, feedback[11]. TRAKLA's feedback isn't very detailed, it simply states how many steps the user got correct. JDPET on the other hand, for each subproblem, states the correct answer based on other entered subproblems, users' answers, and worked example correct answer. This gives JDPET much more details on the feedback.

Another notable feature about other programs is the ability to use user generated problems. Some of the programs, including JFLAP allow a users to create their own problems to visually run[4][8]. Although it would be a nice feature to have, JDPET lacks this and makes up by having a preset problem for each algorithm and the capability to randomly generated new problems.

The main focus of JDPET generally differs from other educational tools, but JDPET does share one useful trait with other educational tools. Like many other educational tools, JDPET uses visualization to help students learn. It is a common trait

used by many other tools that cannot only be used to learn about algorithms, but can also be used to learn about programming paradigm styles such as object oriented programming or even subjects like state machines[1][3][8][10].

JDPET helps students learn by solving problems for themselves. JDPET provides feedback for randomly generated instances of a few specific dynamic programming algorithms. It allows students to test their knowledge of those algorithms and provides detailed feedback. It's grading and custom feedback are a step above in other systems for dynamic programming algorithms.

# 8. References


[1] S. M. Cisar, et al., "Software visualization: The educational tool to enhance student learning," *MIPRO, 2010 Proceedings of the 33rd International Convention*, pp. 990-994, 2010.

[2] T. H. Cormen, et al., *Introduction to Algorithms*, 2nd ed. Cambridge, MA: MIT Press, 2001.

[3] M. Esteves and A. J. Mendes, "A simulation tool to help learning of object oriented programming basics," *in Education, 2004. FIE 2004. 34th Annual*, vol. 2, 2004.

[4] D. Galles, "Data Structure Visualization," Internet: http://www.cs.usfca.edu/~galles/visualization/ [Oct, 2011].

[5] S. Goddard, "Dynamic Programming," Internet: http://www.cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf [Oct, 2011].

[6] L. Holder, "The Unbounded Knapsack Problem," Internet: http://www.eecs.wsu.edu/~holder/courses/cse2320/lectures/applets/knapsack/knapsack.html [Oct, 2011].

[7] A. Kolokolova, "Dynamic Programming Algorithms," 2003, Internet: http://www.cs.mun.ca/~kol/courses/2711-w08/dynprog-2711.pdf [Oct, 2011].

[8] S. H. Rodgers, *JFLAP 7.0,* Software: http://www.cs.duke.edu/csed/jflap/.

[9] *AlViE* 3.0, Software: http://alvie.algoritmica.org/.

[10] *Jeliot 3.7.2*, Software: http://cs.joensuu.fi/jeliot/index.php.

[11] *TRAKLA2*, Software: https://trakla.cs.hut.fi/app.