Assessing the Centrality of Motion in Instructional Multimedia :

Algorithm Animation Revisited

A Thesis
Presented to
The Academic Faculty

by

A. Fleming Seay

In Partial Fullfillment
Of the Requirements for the Degree
Masters of Science in Psychology

Georgia Institute of Technology
December 1999

Assessing the Centrality of Motion in Instructional Multimedia :

Algorithm Animation Revisited

Approved:

____

Richard Catrambone

____

Wendy A. Rogers

____

John Stasko

Date Approved__________________

DEDICATION

To my father and grandmother:

“I’ll be missing you.”

And to Mom:

“We’re all we got.”

ACKNOWLEDGEMENTS

The author would like to thank Lorann Miles Birr without whom this would not have been possible. Thank you for believing in me more than I believed in myself.

A special thanks goes to Chris Scheil for his endless efforts in creating the study–aids used in this and previous studies.

Thanks also to Dr. Richard Catrambone for his guidance, advice, and countless re-readings of the document at hand.

TABLE OF CONTENTS

CHAPTER

I.	INTRODUCTION	1
	Foundations in Algorithm Animation Research	2
	Task Analysis	3
	Roots of the Current Work	4
	Diagrams, Animations, and Learning	6
	The Importance of Motion	11
	A Perspective on Transfer	12
II.	OVERVIEW OF PROBLEM DOMAIN	14
	Arrays, Stacks, and Queues	14
III.	OVERVIEW OF EXPERIMENT	19
	Time Measures	21
	Cognitive Ability Measures	22
	The Problem Sets	23
	Expectations	23
	The Three Principle Hypotheses	25
IV.	METHOD	28
	Participants	28
	Apparatus	28
	Design and Procedure	29
	Data Analysis	32
V.	RESULTS	34
VI.	DISCUSSION	39

APPENDIX A – COVER SHEET 44

APPENDIX B – A TASK ANALYSIS OF THE ARRAY DATA STRUCTURE 45

APPENDIX C – A TASK ANALYSIS OF THE STACK DATA TYPE 47

APPENDIX D – A TASK ANALYSIS OF THE QUEUE DATA TYPE 49

APPENDIX E – ISOMORPHIC PROBLEM SET 52

APPENDIX F – NON-ISOMORPHIC PROBLEM SET 56

APPENDIX G – ANSWER KEY 60

APPENDIX H – SCORING RATIONALE 65

APPENDIX I – INTRODUCTORY TEXT 66

APPENDIX J – TASK ANALYSIS BASED TEXT 69

APPENDIX K – CONTROL STACK TEXT 72

REFERENCES 75

LIST OF TABLES

Table		Page
5.1	Means and standard deviations for SAT verbal/quantitative, ACT, and GPA.	34
5.2	Means and standard deviations for near and far transfer performance by group.	35

LIST OF FIGURES

Figure		Page
3.1	3x2, fully crossed design contrasting three levels of study-aid type with two levels of text type.	19
4.1	An example of a three keyframe sequence. The animation also depicted the transitional motion between these static frames.	31
4.2	The problem, and answer for question number 6 of the isomorphic problem set.	32
5.1	The actual trend obtained with respect to hypothesis two.	37
5.2	The significant linear trend obtained with respect to hypothesis three	38

SUMMARY

This study explores the effect of dynamic displays of information on isomorphic and non- isomorphic problem-solving performance in the computer algorithms domain. Specifically, the study assesses the importance of dynamic motion in producing generalization and transfer of problem solving knowledge. Problem solving performance data were analyzed to investigate the differential effects of six instructional conditions on learning performance. These conditions included control text, control text with frames, control text with animation, task analysis text, task analysis text with frames, and task analysis text with animation. Students in the frame conditions viewed a reduced version of the full animation containing keyframes and pseudocode. Near and far transfer performance was measured during a post-test phase containing isomorphic and non-isomorphic problems. The results indicated a significant performance advantage on the isomorphic problems for those participants studying task analysis based materials. In addition, an incremental performance difference based on study aid type was observed on the non-isomorphic problems for participants using the control text. The results recapitulate the importance of formal domain analysis in the creation of instructional materials, and establish the importance of dynamic motion as a contributor to the pedagogical advantage rendered by algorithm animations in certain learning situations.

CHAPTER I

INTRODUCTION

Computer-driven simulations and study-aids continue to be increasingly popular components of instructional curricula. Early experiments in the use of computerized simulations described their ability to enhance comprehension of complex subject matter (Rigney & Lutz, 1976), giving birth to an entire field of study known as Computer Based Training or CBT. In addition to other benefits, proponents of CBT claim that use of such instructional systems can reduce training time by an average of 33% and show an effect size of .30 when compared to more traditional techniques (Stephenson, 1994). In spite of the demonstrable success of the overall field, a sub-field called algorithm animation has lagged behind the rest of the CBT literature in producing significantly positive training outcomes for its users. However, recent work has demonstrated a significant effect of the use of algorithm animations on problem solving performance (Hansen, Schrimpsher, & Narayanan, 1998; Seay & Catrambone, 1998). The current study expands upon and refines the findings of Seay and Catrambone by more deeply investigating the two pedagogical tools addressed in their study: task analysis based text and animated graphical study aids. Both tools seem to have a significant positive impact on the problem solving performance of the students who use them. However, the interaction or additivity of their effects has not been established. By looking more closely at the impact of both of these two tools on problem solving performance, we hope to further solidify and illuminate the theory and practice of algorithm animation.

Foundations in Algorithm Animation Research

A field related to CBT known as software visualization (Price, Baecker, & Small, 1993) is based on the use of images and other graphical representations to help describe the operations of a computer. The subset of software visualization on which this study focuses is known as algorithm animation (Brown, 1988). Algorithm animation is concerned with the use of animations to teach the function and proper implementation of complex computer algorithms to students and programmers. This is done through the use of graphical depictions of the implied structure and motion of the elements of an algorithm as they work through execution. The first work in the field came in the form of a video presented at SIGGRAPH in 1981 entitled Sorting Out Sorting (Brown, 1988). Following from this, a diverse series of implementations of the algorithm animation concept have come into existence. From multimedia presentations to entire animation development environments like John Stasko's Tango, XTango, and Samba, the field of algorithm animation is an expanding one (Byrne, Catrambone, & Stasko, in press).

In spite of its growth, the field has encountered one key problem. It has been difficult to empirically demonstrate any consistent and significant improvement in student performance as a result of exposure to algorithm animations (Byrne et al., in press). In one study, Byrne et al. questioned both the content of their animations and their adopted measure of performance following their inability to demonstrate a significant effect. With regard to animation content, Byrne et al. postulated that animations based on a "careful task analysis" of the algorithm might be more effective

teaching tools than those employed in their study. Byrne et al. also felt that the post-test they used to measure learning performance may have been too simple (in press). They expressed the belief that the tests they had were not able to fully and accurately assess the participants' learning or the effectiveness of the training materials. Stephenson (1994) also supports this notion of the limited descriptive power of brief objective measures of learning in his review of the CBT literature.

Task Analysis

Task analysis is characterized by the detailed examination of a task and the partitioning of said task into its components. For the purposes of the current study the following formalism was used. The components used were definitions, rules, conventions, examples, and exceptions that were designed to clearly state the operators and operands of the problem domain and describe how these items might operate in a given context. Under this formalism a task analysis of opening a door might include definitions of door, hinge, handle or knob, open, closed, and the actions of grasping, turning, pulling, and perhaps pushing. Rules for opening a door might describe the swing of the door on the axis created by its hinges. A convention might be that all doors both open and close, and some can be left open or left closed. Examples would simply describe the operation of various parts of or types of doors. An exception might be the case of revolving doors.

A complete task analysis can be a valuable tool in the construction of instructional/training materials. It allows the designer to break down the material into

content units that can each be addressed explicitly in order to achieve complete coverage of a topic area. For the purposes of this study a task analysis was performed on the array data structure (Appendix B), the stack data type (Appendix C), and the queue data type (Appendix D). Each task analysis was performed by first establishing the definitions and rules required to successfully solve problems in the given domain. Following this, conventions, examples and exceptions were also generated in order to support the definitions and rules and govern their application in specific instances. To test a completed task analysis, pilot participants were asked to step through solving a set of problems using nothing but the information contained in the task analysis itself. The participants were asked to think aloud as they solved, referring to the element in the task analysis that governed each action they performed. This method was used to uncover any holes that might have existed in the analyses since the participant would have to stop solving the problem if a situation arose that was not explicitly addressed in the provided task analysis.

Roots of the Current Work

A study (Seay & Catrambone, 1998) undertaken at the Georgia Institute of Technology sought to extend the research of Byrne et al. (in press). This study explored the effects of algorithm animations on students' ability to learn the Stack and Queue algorithms as measured by performance on isomorphic and non-isomorphic problem sets. Three conditions were compared: 1) a control condition in which students studied a text based on a standard algorithms textbook; 2) a "task analysis" text condition in which

students studied a text based on a careful task analysis of the information needed to solve problems dealing with the Stack algorithm; 3) a "task analysis plus animation" condition in which students studied the task analysis-based text and also viewed a task analysis-based animation of the algorithm. Students in the task analysis conditions (2 and 3 above) outperformed the control condition students on the near transfer, isomorphic problems. These problems dealt specifically with the Stack algorithm. The far transfer, non-isomorphic problems dealt with an analogous, but functionally distinct algorithm called the Queue (a brief review of Stacks and Queues is provided later). Solving the far transfer problems required students to generalize the knowledge gained from the instructional materials, modifying it to "fit" the operation of the new algorithm. Students who viewed the animation outperformed both paper-based groups on the far transfer problems, indicating that the animation had in some way uniquely prepared them to solve the non-isomorphic set.

Seay and Catrambone (1998) thus obtained a rarely demonstrated, significant effect of an animation based study-aid on problem-solving performance. What remains unclear is the aspect or aspects of the animation that fostered the observed performance advantage. Was it the robustness of the graphical representations, the interaction of the study-aid with the task analysis based text, or the motion of the problem elements themselves that yielded the observed performance increase? The present study intended to answer these questions as well as attempt to cast the results in an explanatory framework.

Diagrams, Animations, and Learning

Considerable debate exists surrounding the theoretical basis of graphic instructional materials, spanning the spectrum from simple diagrams to high fidelity immersive simulations. In calling for a more systematic approach to the study of instructional graphics, Scaife and Rogers (1996) point to "a fragmented and poorly understood account of how graphical representations work" and a further paucity of theoretical foundation for this type of research (p. 185).

Reiber and Kini (1991) undertook a review of these theoretical foundations, endorsing what is a widely accepted, yet still controversial, theory regarding the efficacy of instructional graphics: the dual-coding theory. Developed by Paivio (1979, 1986) and supported empirically by a number of studies including Anderson (1978), dual-coding theory holds that the study of pictures and words activate independent visual and verbal codes. Since these separate mechanisms are thought to be additive in their effect, information encoded in both is more likely to be remembered than information encoded in only one. Further, graphics are more likely than text to be encoded both verbally and pictorially. Therefore the probability of information recall from graphics is greater due to the availability of the two representations. Also, such dual-coded knowledge is thought to be more durable since difficulty retrieving one trace in memory does not affect the other, rendering the information still available.

Critics of the dual-coding position point to its lack of specificity with respect to the relationship between and mechanisms of internal and external representations (Scaife & Rogers, 1996). Endorsing what they call "external cognition" Scaife and Rogers point

to three characteristics of instructional graphics they see as central to their efficacy: computational offloading, re-representation, and graphical constraining. Computational offloading and re-representation are simply a restatement of the accepted notion that reduction in cognitive load benefits the problem solving process by freeing cognitive resources and making them available for alternative tasks. In as much as instructional graphics reduce the cognitive load imposed upon an individual operating in a certain problem space, the graphics benefit that individual by facilitating the problem solving or learning task.

With respect to re-representation, Scaife and Rogers (1996) contend that certain external representations are easier to operate upon than others (ie, multiplying roman numerals vs. arabic numerals). Though made explicitly distinct in their framework, this issue still seems to be one of cognitive load reduction, in that external representations not lending themselves to problem solving operations or requiring some form of translation (ie, complex notations, unclear or ambiguous symbols) add cognitive load to the problem solving task. Further, lucid and unambiguous graphical representations that do not add to the cognitive load are more effective in aiding the problem solver or learner.

Finally, Scaife and Rogers (1996) advance the concept of graphical constraining, characterizing it as the way in which :

> …the relations between graphical elements in a graphical representation are able to map onto the relations between features of a problem space in such a way that they restrict (or enforce) the kinds of interpretations that can be made. (p. 198)

This seems like an obfuscated way of making a very useful, if not obvious, point: that clear instructional graphics reduce the opportunity for illusory interpretations of the problem space. Simply, in as much as graphical representations are explicit and deterministic in representing the elements and operations of a problem space, they disambiguate that problem space, making spurious interpretations less likely to occur. Unfortunately, these three factors (computational offloading, re-representation, and graphical constraining) are couched within the overarching framework of "external cognition" and forced into service alongside an opaque notion of the interaction between external and internal representations that is respondent to what the authors call the "resemblance fallacy" (Scaife & Rogers, 1996). As a result, Scaife and Rogers' theoretical stance offers little that is novel, but does speak to some of the central aspects of the utility of instructional graphics.

A more parsimonious, yet perhaps not atheoretical, approach to the role of instructional graphics in learning was adopted for the purposes of the current study. While not endorsing a specific model of cognition or representation, we believe that the efficacy of graphical representations in instructional materials is determined by two key factors: 1) the degree to which cognitive load is reduced, and 2) the degree to which the problem space is disambiguated. As supported by Kalyuga, Chandler, and Sweller (1998), reduction in cognitive load, or the number of task operators and elements that must be held in working memory, allows more efficient and less effortful learning and task completion. Specifically, Kalyuga et al. have shown that reduced cognitive load during the acquisition phase of a learning task improved certain types of subsequent test

performance. Well-constructed instructional graphics that are incorporated into learning materials should, therefore, assist the learner by freeing cognitive resources to be directed at other aspects of the problem space or learning environment. One might then ask, how or through what mechanism are these cognitive resources freed? This brings us to the second point.

Cognitive load can be defined as the amount of mental effort required to complete a problem. Disambiguation of the problem space through clear graphical depiction of its elements and operators reduces cognitive load by reducing the need for interpretation or speculation about any indeterminate aspects. Quite simply, if one is shown in an unambiguous fashion how things look and work then that individual is less likely to need to allocate resources to interpretation. Such effort can be redirected to actually solving the problem or learning the material instead of speculation about its structure.

So what does animation add to the mix? What are the qualities that differentiate it from simple static diagrams? Reiber and Kini (1991) see it this way:

> Animation makes the cognitive task more concrete by providing motion and trajectory attributes directly to the learner, thus reducing the processing demands…and hopefully increasing the potential for successful and accurate encoding… (p. 86)

Even static diagrams with well understood, "classical" motion cues (i.e., arrows or flow symbols) suffer from a certain amount of ambiguity that is overcome by an animation that actually performs the suggested motion. With such diagrams, readers must work to connect and "mentally animate" the elements of the display, providing the

opportunity for illusory or even patently incorrect assumptions (Hegarty, 1992; Reiber & Kini, 1991). In the case of animation, the trajectory and motion information is provided in complete and unambiguous form.

In investigating cases in which animation garners the greatest pedagogical advantage, Hays (1996) asserts that animation is most useful when the instructional domain involves dynamic and/or spatial processes as key elements. The results of Hays' studies lead him to the conclusion that animation is better than text at communicating concepts involving time and motion. The present study was designed to test this conclusion in that the algorithms used here, at their center, involve concepts of time evolving and systematic state changes that involve the motion of their elements.

It should be noted that the animations employed in this study are designed to enhance the procedural problem solving skills of students operating in a specific quantitative domain; computer algorithms. Hansen et al. (1998) indicate that learning to understand and analyze algorithms depends on the student's grasp of the dynamic nature of the domain. What better way to exhibit this than through accurate and clear algorithm animations, animations that can evoke accurate understanding of dynamical factors that might bear generalization to novel problem spaces (Kaiser, Proffitt, Whelan, & Hecht, 1992)?

The Importance of Motion

In order to assess the contribution of dynamic motion to the learning experience, a frame-based study-aid was designed for inclusion in this study. The animations employed in this study were like flip-book drawings, in which the illusion of apparent

motion is induced by viewing consecutive screens in rapid succession. As in most advanced computer animation packages, keyframes are defined that exhibit important finite states of the system being animated. The computer can then automate the job of creating and inserting the interstitial screens that must exist in order to create a smooth transition between the states defined by the keyframes. By themselves, the keyframes statically represent the important finite states through which the algorithm passes as it runs to completion. However, they do not explicitly depict any of the transitional motion that is exhibited by the full animation. For the purposes of the current study, the frame-based study-aid was a set of these keyframes connected by the pseudocode associated with the transitions between them.

For example, for participants in the "Frames" conditions the experiment the beginning state of the algorithm was statically represented. After a short period of time elapsed, a line of pseudocode was displayed above the beginning keyframe. Once the user hit the "forward" key, the beginning keyframe was replaced by another keyframe statically showing the state of the algorithm resulting from the execution of the displayed line of pseudocode. The only difference between the frame study-aid and the animation study-aid was the absence of any depiction of the time evolving, transitional motion that occurs as a result of the execution of the algorithm's various lines of pseudocode.

Users of the frame-based aid were left to infer two things; 1) that movement of the problem elements has occurred, and 2) the way in which this movement might have occurred. Users of the animation-based aid were not required to make such inferences, as the nature of the movement of the problem elements was clearly displayed to them. If

explicit and unambiguous depiction of the movement of the problem elements about the problem space made any contribution to the performance of animation users, then performance differences would be apparent between those participants using the frame-based study-aid and those using the animation study-aid. Such differences could then be attributed to the reduction in ambiguity provided by the clear depiction of the dynamics of the algorithm. If no differences were to arise, then this might indicate that the actual motion is not the central factor contributing to the performance advantage rendered by animations. However, performance differences might also be linked to the relative difficulty of "filling in the gaps" left in the frames based aid when the dynamic motion is removed. If the veridical motion of the problem elements is sufficiently implied by the frame-based aid, then participants would be able to "mentally animate" the elements and override any advantage that the animated aid might render.

A Perspective on Transfer

Paas (1992) refers to a considerable amount of empirical evidence in support of his assertion that traditional instruction techniques are not effective in the formal sciences of physics, mathematics, and computer science. This criticism is based on the deductive nature of traditional instruction. To Paas, the central tenet of traditional instruction is a reliance on practice through conventional problem solving in which students are taught a set of general operators that they must apply to a problem in order to reach some specific goal state. This is, by definition, a deductive process. As a result students often resort to inefficient or inappropriate solution methods such as means-end analysis and hill

climbing (Paas, 1992). With regard specifically to transfer, Paas criticizes traditional instruction for its inability to consistently enable far transfer, that is students are often unable to solve problems that differ, even in appearance, from the specific ones they practiced during training.

Paas (1992) concludes that the study of partially and fully worked problems is an appropriate alternative to conventional instruction. The algorithm animations and static text exemplars employed in the present study constitute worked and partially worked examples. Completion type test problems also include partially worked components. In general, it is hypothesized that students presented with such materials more accurately and definitively understand the problem space, allowing them to alter that problem space in a specific fashion and complete problems requiring far transfer. For the purposes of this study, it was expected that those participants exposed to the animations would outperform participants in the "static" or text only conditions, as the dynamic nature of the animations would provide for the formation of more durable, robust, and unambiguous understanding of the problem space. The frames-based study-aid was included as an ablation of the animation that would allow assessment of the centrality of actual motion of the problem elements while still presenting a "serialized" display of the algorithm as it runs through a number of finite states on the way to completion.

CHAPTER II

OVERVIEW OF PROBLEM DOMAIN

Arrays, Stacks, and Queues

An array is a group of locations used to store values. Arrays are often given names corresponding to the type of information they store. For example, an array used to store the results from a final exam might be called "Scores." Each individual value stored in an array is called an element. The size of an array determines the number of elements that can be stored just as the number of dimples in an egg carton determines how many eggs can be held in it. In general, array size describes the form of an array by specifying the number of rows and columns in the array. For example, an array with two rows and three columns could hold a maximum of six elements. Stack and Queues are both one dimensional arrays. One dimensional arrays have either one row and one or more columns or one column and one or more rows. Because of this one dimensional property, only one number is needed to describe the size of one dimensional arrays.

In a one dimensional array, elements are indexed or addressed using their ordinal (numerically successive) position in the array. By convention, indices are often a series of numbers, starting with "1," that increment from bottom to top or left to right across the array. Elements are entered into the array in the same order in which the indices are incremented. As such, each element value is associated with one and only one index. Individual elements can be referred to using the name of the array and the index at which the element in question is stored.

A pointer is a variable used to track indices of interest in an array. Top, head, and tail (covered later) are examples of pointers. Pointers point to indices, never directly to the element values those indices store. However, through the use of correct notation,

pointers can be used to indicate element values as follows. The name of the pointer is followed by the name of the array in which it is involved; a pointer "top" in array Scores would be written top[Scores]. If top[Scores] points at the index "1" in the array above, it would be written top[Scores] = 1. The value "17" stored in index "1" of array Scores would be indicated by Scores[top[Scores]] = 17 which would be equivalent to writing Scores[1] = 17 since top[Scores] = 1.

A general familiarity with the array allows understanding of a special kind of array called a Stack. Stacks adhere to a number of special rules and constraints to which not all other arrays adhere. Because of this, stacks are very useful for performing some operations for which other types of arrays are less suited. Generally, Stacks are referred to as LIFO or last-in-first-out data structures. That is to say that Stacks are programmed so that the most recent element entered into the Stack (or the "last" one) will be the first to be removed. It might be useful here to think of the analogy of a stack of plates. The plate on the top of the stack (presumably the one put there last) will be the first to be removed. Essentially, this is how Stack data structures operate. Just like a stack of plates, Stacks fill with elements from the bottom up.

A Stack has two attributes: its size and a pointer called "Top." Stack size defines the number of elements a stack can hold just like array size did above. Therefore, an array of size "6" defined as a stack would more appropriately be called a stack of size six. In reality, stack size and array size are equivalent and interchangeable concepts, but since stack size is a more precise designation, one should use it when applicable. Returning to

the stack of plates analogy for moment, a stack of size six would have room enough for storage of a maximum of six plates.

Top is the name given to the pointer that "points" at the index of the element most recently inserted into the stack. As such, Top stores the value of the index, <u>not</u> the value of the element stored at that index. A computationally useful implication of this treatment of Top is the following: if Top = 0 then we know that the Stack is empty. Also, the initial value of Top is always 0 since Stacks always start out empty. In the same way, when Top = Stack Size then the Stack is full since no other available indices would exist to store additional elements. For notational purposes, Top[Ages] could be used to refer to the pointer Top in a Stack named "Ages."

Push is the command used to insert a new element into the Stack. Push is accompanied by what is called an argument or additional information necessary to perform the command. In the case of Push, the argument is always the element that is being inserted into the Stack. Though Push is a single command, a few subcommands run each time a Push is attempted. First, the value of Top is increased by one to point at the next available index in the stack. If there is such an available storage location then the element value is inserted into the Stack at the index Top points to.

A subcommand of the Push command checks for an error condition called Overflow. The Overflow error occurs when one attempts to Push a new element onto an already full Stack. Since the Stack is unable to handle the new element, the "Stack Overflow" error message is returned.

Pop is the command used to remove the top element from the stack. Pop is not accompanied by any argument since the element corresponding to the index stored in Top will always be removed by the Pop command. A Pop command first checks to see if the Stack is empty. If it is not empty, the element stored in the index pointed to by Top is removed from the Stack and then Top is decreased by 1. This reduction in Top is necessary to indicate that the next element stored in the array is now the most recently inserted or top element in the Stack.

A subcommand of the Pop command checks for an error condition called Underflow. The Underflow error occurs when one attempts to Pop (or remove) an element from an already empty stack. Since there are no elements in the Stack, the "Stack Underflow" error message is returned.

A Queue operates in much the same way as a Stack, but with a few notable distinctions. While the Stack operates under a last-in-first-out, or LIFO policy, the Queue operates under a first-in-first-out, or FIFO, policy. To institute this it is necessary that the Queue track two pointers called Head and Tail. Head points to the index of the oldest element in the Queue, while Tail points at the index of the most recently inserted element in the Queue. When a Dequeue command is issued, the element stored in the index pointed to by Head is removed from the Queue. This is analogous to the Pop command in the Stack. When an Enqueue command is issued, the new element is entered into the Queue at the rear and its index is pointed to by the Tail pointer. This is analogous to the Push command in the Stack. Both Overflow and Underflow errors occur in the same way with Queues as they do with Stacks.

As mentioned before, successful problem solving in any domain would necessitate the formation of a robust problem solving schema, or pattern of knowledge describing what is typical in a particular situation (Reisberg, 1997). Though there are many possibilities of what the schema might contain, a schema for solving array based problems might be comprised of a core understanding of the array as a versatile data storage structure that can operate in various ways; a playing field of sorts. The Stack and Queue amount to two distinct ways that an array might operate; two sets of rules or games that can be played out on that playing field.

CHAPTER III

OVERVIEW OF EXPERIMENT

In the present study, a 3 x 2, between subjects design was employed to investigate the effect of the independent variables of study-aid type and text type on the dependent variable of test score. There were three levels of the study-aid type independent variable: (1) animation, a full-motion dynamic display of the algorithm in operation; (2) frames, a static display of keyframes from the animation displaying beginning and end states; and (3) no aid, a control level containing no study-aid. There were two levels of the text type independent variable: (1) task analysis based text, derived from a task analysis of the stack algorithm, and (2) control text, derived from a standard algorithms textbook. The levels of the two independent variables were fully crossed to create six separate conditions; (1) task analysis text + animation (TA+Anim), (2) task analysis text + frames (TA+Frames), (3) task analysis text + no aid (TA-Only), (4) Control text + animation (CO+Aanim), (5) Control text + frames (CO+Frames), and (6) Control text + no aid (CO-Only).

		Study-Aid Type		
Text Type		Animation	Frames	No Aid
	Task Analysis Text	TA+Anim	TA+Frames	TA-Only
	Control Text	CO+Anim	CO+Frames	CO-Only

Figure 1. 3x2, fully crossed design contrasting three levels of study-aid type with two levels of text type.

The task analysis text and control text employed were the same texts as were employed in Seay and Catrambone (1998). Only individuals with extremely limited knowledge of computer science were allowed to participate in the study. Each individual's a priori domain knowledge was assessed through the use of a Likert type instrument (Appendix A). The use of participants familiar with computer science would have confounded results, since these persons would have had an enormous advantage over other participants regardless of condition.

Participants in all conditions read the same introductory text covering an introduction to algorithms, pseudocode, and arrays. Afterward, participants read a brief text on the stack data type the exact form of which was defined by their experiemental condition; participants in the TA conditions read the task analysis based text while participants in the CO conditions read the control text. Following completion of the appropriate text, participants in the TA+Anim, TA+Frames, CO+Anim, and CO+Frames conditions were presented with a computer based study-aid. In the TA+Anim and CO+Anim conditions, this study-aid was a full frame-by-frame, dynamic animation of the stack data type in operation. In the TA+Frames and CO+Frames conditions, various "keyframes" of the full animation were displayed. Specifically, the frames study-aid statically presented a graphical representation of a beginning state of the stack, displayed a line of pseudocode, displayed the resulting state of the stack, then displayed another line of pseudocode, and so on. This differed from the animation shown in the TA+Anim and CO+Anim condition in that no transitional motion between states were displayed.

Participants in the TA-Only and CO-Only conditions read the text appropriate to their respective condition, and were not presented with a study-aid.

After this initial exposure to the study materials, participants in all conditions completed a series of near-transfer problems designed to assess their level of understanding of the Stack algorithm. Upon completion of these problems, the participants read a very brief second text on queues and continued on to the far-transfer test problems on queues.

Time Measures

The amount of time each participant spent with the materials was measured at several points throughout the procedure in order to render fairly fine-grained measures as follows:

Participants in the TA+Anim, TA+Frames, CO+Anim, and CO+Frames conditions were asked to write down the time at which they:

- began reading the introductory text
- finished reading the introductory text
- began reading the stack text
- finished reading the stack text
- began using the study-aid
- finished using the study-aid
- began the near-transfer problems
- finished the near transfer problems
- began the second reading
- finished the second reading
- started the far-transfer problems
- completed the far-transfer problems

Participants in the TA-Only and CA-Only conditions were asked to write down the time at which they:

- began reading the introductory text
- finished reading the introductory text
- began reading the stack text
- finished reading the stack text
- began the near-transfer problems
- finished the near transfer problems
- began the second reading
- finished the second reading
- started the far-transfer problems
- completed the far-transfer problems

A time measurement was also taken at the transition from completion to from-scratch problems for both problem sets. In this way, all pertinent time measures were collected explicitly instead of being estimated from larger, less precise measures. These time measures were collected in order to determine whether performance on the problem sets was merely a function of time spent with the materials.

Cognitive Ability Measures

General cognitive ability was assessed through the use of self report measures of GPA and SAT/ACT scores. Collection of these measures allowed us to test whether cognitive ability was evenly distributed among experimental groups. If cognitive ability was not evenly distributed across the conditions, performance results could be attributed to the relative cognitive ability of individuals in each condition rather than the type of instructional material to which they were exposed. A positive correlation between individual GPA or SAT/ACT scores and performance on both problem sets was expected

since solution of the problems was likely to be influenced by the cognitive ability of the participant.

The Problem Sets

The isomorphic (near transfer) and non-isomorphic (far transfer) problems (see Appendices E and F, respectively) were comprised of two types: completion and "from-scratch." Completion problems required the student to perform some small number of operations on a pre-existing data structure. From-scratch problems required the student to draw the data structure and perform a series of operations on it. The from-scratch problems were more difficult since less information is supplied by the problem itself, thus the participant had to rely more on his or her domain specific knowledge. The far transfer problems were based on a data structure related to the Stack, called the Queue.

Expectations

The authors expected that the three task analysis text conditions (TA+Anim, TA+Frames, and TA-Only) would outperform the control text conditions (CO+Anim, CO+Frames, and CO-Only) on the near-transfer problems. This was based on the fact that the task-analysis based text employed in the TA+Anim, TA+Frames, and TA-Only conditions of this experiment seemed to provide a superior level of information necessary for mastery of the Stack data structure. Seay and Catrambone's (1998) participants in the task analysis text only condition performed just as well as those in the task analysis text + animation condition on the near-transfer problem set. However, both

these conditions significantly outperformed the control text condition on the near-transfer problem set. Results in the current study were expected to mimic those of Seay and Catrambone in this fashion.

With regard to the far-transfer problems, significant differences in performance were expected to arise among the groups due to the differential pedagogical advantage rendered by the use of the study-aids. Seay and Catrambone's (1998) participants in the task analysis text + animation condition significantly outperformed those in the task analysis text only and control text conditions on the far-transfer problem set. It was concluded that this result was due to the more robust and generalizable problem solving schema created as a result of viewing the animation in the animation condition. As the present study was aimed at assessing the importance of the actual motion of the problem elements in the animation, the keyframe based study-aid used in the TA+Frames and CO+Frames conditions were designed to fill the instructional gap between the fully dynamic display of the TA+Anim and CO+Anim conditions and the text only treatments in the TA-Only and CO-Only conditions. As such, it was expected that performance on the far-transfer problems in the TA+Frames condition would be higher than that in the TA-Only condition but lower than that in the TA+Anim condition. In the same way, it was expected that performance on the far-transfer problems in the CO+Frames condition would be higher than that in the CO-Only condition but lower than that in the CO+Anim condition. Again, this expectation was based on the centrality of the role of full and dynamic motion in the extraction of knowledge from algorithm animations and other multi/hypermedia displays.

The Three Principle Hypotheses

The expectations described above were broken down into three predictive hypotheses:

(1) A significant advantage in near-transfer performance for the experimental conditions using the task analysis based text (TA+Anim, TA+Frames, and TA-Only) when compared to the corresponding condition using the control text (CO+Anim, CO+Frames, CO-Only).

(2) A linear trend in the performance data on the far-transfer problems for the conditions using the task analysis based text (TA+Anim, TA+Frames, and TA-Only). Specifically, this linear trend was predicted to go from lowest in the TA-Only condition to highest in the TA+Anim condition with TA+Frames falling somewhere in between. In order to truly support the notion that dynamic motion is a necessary contributor to the performance advantage rendered by the algorithm animations, a significant difference in performance between TA+Anim and TA+Frames would have to be obtained in addition to the linear trend. A linear trend could be obtained in the data without the existence of the described significant difference if both TA+Anim and TA+Frames differed significantly from TA-Only, but not from each other. However, such a scenario would not be considered substantively supportive of hypothesis two.

(3) In the very same way a linear trend was expected to arise in the performance data on the far-transfer problems for the conditions using the control text (CO+Anim, CO+Frames, CO-Only). Specifically, this linear trend was expected to go from lowest in the CO-Only condition to highest in the CO+Anim condition with CO+Frames falling

somewhere in between. In order to truly support the notion that dynamic motion is a necessary contributor to the performance advantage rendered by the algorithm animations, a significant difference in performance between CO+Anim and CO+Frames would have to be obtained in addition to the linear trend. A linear trend could be obtained in the data without the existence of the described significant difference if both CO+Anim and CO+Frames differed significantly from CO-Only, but not from each other. However, such a scenario would not be considered substantively supportive of hypothesis three.

Due to the fully crossed nature of the experimental design employed in this study, the possibility existed for interactive effects to arise between the levels of the independent variables of text type and study aid type. However, no a priori predictions were made in this regard.

The collected measures of cognitive ability were expected to correlate highly with overall performance on both the near and far transfer problems. Measures of time spent with materials were expected to be roughly equivalent across conditions though perhaps slightly higher in the animation (TA+Anim and CO+Anim) and frame (TA+Frames and CO+Frames) conditions due to the fact that there was more material to view.

The argument could be made that any effect for the study-aids may be attributable to the fact that participants in the study-aid conditions (TA+Anim, CO+Anim, TA+Frames, and CO+Frames) simply received more materials than those in the other condition. This argument is not totally relevant to the research question proposed here. This study was proposed to continue the investigation of algorithm animations as a

worthy addition to existing computer science curricula, that is, as a supplement to static representations of algorithms. It was not designed to assess the value of an animation as a substitute for text-based treatments. Methodologically, such a design would require a "study-aid only" condition, one that does not exist here. A "study-aid only" treatment would be unlikely to convey the necessary breadth and depth of information to its participants and therefore, was not included in this study.

CHAPTER IV

METHOD

Participants

506 participants were involved in the current study. These participants were undergraduate non-computer science majors at the Georgia Institute of Technology who had not taken an algorithms class. These individuals were allowed to participate in the study for credit in their psychology course. Before participating in the study, each participant was asked to complete a questionnaire concerning their computer science background to ensure that they had minimal familiarity with algorithms (Appendix A). Individuals with programming experience and/or computer science classwork at the high school or college level were eliminated from the sample. Following this screening process based on their responses to the questionnaire, the sample was trimmed to 300, providing 50 participants per condition.

Apparatus

A PC running the Microsoft Windows95 operating system and Macromedia Director was used to create the study-aids employed in this study. The study-aids were displayed on PCs running the Microsoft Windows95 operating system.

Design and Procedure

Participants were randomly assigned to six separate groups: (1) task analysis text + animation (TA+Anim), (2) task analysis text + frames (TA+Frames), (3) task analysis text + no aid (TA-Only), (4) Control text + animation (CO+Anim), (5) Control text + frames (CO+Frames), and (6) Control text + no aid (CO-Only). Training sessions for all six conditions took place in a large computer lab. Between one and sixteen individuals participated during any one session.

Participants in all six conditions read a brief text containing a general introduction to algorithms followed by a task analysis-based coverage of the Array data structure (see Appendix I), the basis of both the Stack and Queue algorithms. Following this, all participants in the study read a text covering the Stack algorithm. For participants in the task analysis text conditions (TA+Anim, TA+Frames, and TA-Only) this text was derived from a careful task analysis of the Stack algorithm (see Appendix J). For participants in the control text conditions (CO+Anim, CO+Frames, and CO-Only) this text was borrowed from an algorithms textbook (see Appendix K). After reading the appropriate text, participants in the text only conditions (TA-Only and CO-Only) completed the near-transfer problem set (see Appendix E), read a very brief transitional text and then completed the far-transfer problem set (see Appendix F). Participants in the TA-Only and CO-Only conditions used a the digital clock in the bottom corner of their computer's screen to record the times as described above.

Participants in the study-aid conditions (TA+Anim, TA_+Frames, CO+Anim, and CO+Frames) read the same text as those in the corresponding text only condition. In

addition, these participants made use of a computer generated study-aid based on the Stack algorithm after reading the text. Participants in these conditions recorded the time measurement data on their materials, using the system clock on their PC also in the manner described above. Time using the study-aid was also tracked by the study-aid system itself. While working on the problem sets, participants were allowed to refer back to the training materials. For participants in the study-aid conditions, referring back to the study-aid created a new time record so that review time could be measured independently of initial study time.

Both the keyframe display and the animation graphically presented the general properties of a Stack (i.e., size, top, and indexing) (see Figure 2). They also depicted the behavior of the Stack data structure under several Push and Pop conditions, and finally under an Overflow condition. The difference was that the transitional motion occurring between states in the algorithm was not displayed in the keyframe study-aid. Accompanying the images on the screen in both aids was textual representation of the pseudo-code commands associated with the behavior being performed by the graphical elements. The line of pseudo-code corresponding to the current action was displayed during its execution. This was designed to allow the participant to clearly associate the abstraction on the screen with the execution of the corresponding steps in the algorithm. It bears mentioning that the study aids used here were modest relative to other existing treatments of algorithm animation in that the operations performed were predetermined and did not permit user defined parameters.

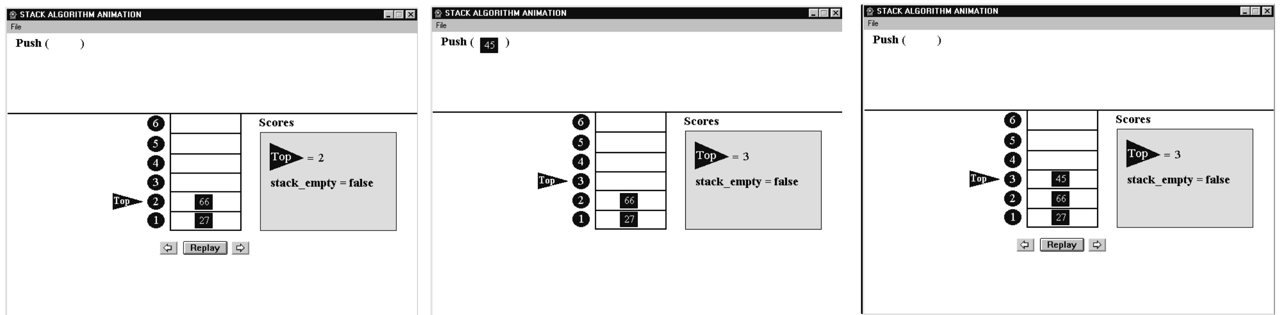


Figure 2. An example of a three keyframe sequence. The animation also depicted the transitional motion between these static frames.

The nine near-transfer problems based on the Stack algorithm consisted of five completion-type problems and four from-scratch problems (see Appendix E). The latter four problems required the participant to create a Stack from scratch and perform a series of operations upon it. The transitional text was a short paragraph at the top of the far-transfer problem set introducing the concept of the Queue data structure and briefly describing its relationship to the Stack (see Appendix F). The nine far-transfer problems consisted of five completion and four from-scratch problems concerning the operation of queues (see Appendix F). Two short-answer problems requiring the participant to apply their knowledge of both the Stack and Queue data structures to a "real world" situation were also included in the far-transfer problem set, bringing the total for that set to eleven.

Problem sets were scored by two raters. Since each problem required several independent responses, a binary scoring approach (Appendix H) to various features of each solution was used to allow partially correct responses to be credited. To illustrate this, the scoring for problem six on the near transfer exercises is presented in Figure 3. Each problem was assigned a point value based on the number of individual responses it

required. For each correct response a "1" was scored, for each incorrect or blank response a "0" was scored.

Figure 3. - The problem, and answer for question number 6 of the isomorphic problem set.

The scoring rationale for problem six was that one point was awarded for correct size, one point was awarded for the proper indices, and one point was awarded for an appropriate set of commands, for a total three available points. As a whole, the near-transfer problem set contained 53 individual responses while the far-transfer set contained 139.

Data Analysis

A multivariate analysis of variance (MANOVA) was conducted to determine if a significant difference in performance on the near-transfer problems existed among the experimental groups (Hypothesis One). Planned comparisons were then conducted in order to determine the existence of significant differences and the pattern of results in the far-transfer problem data (Hypotheses Two and Three). Tukey HSD analyses (pairwise post hoc comparison) were used to determine the nature of any group differences suggested by the performed ANOVAs. An analysis of covariance (ANCOVA) was

applied to the near and far transfer performance data using study time as the covariate. This was done to test whether or not study time accounts for a significant portion of any variance among groups in the performance data for both problem sets. A one way analysis of variance was used to test whether the measures of cognitive ability collected were equivalent across groups.

CHAPTER V

RESULTS

Condition	SAT Verbal		SAT Quantitative		ACT		GPA	
	Mean	*SD*	Mean	*SD*	Mean	*SD*	Mean	*SD*
Control	573.70	65.23	641.09	65.70	29.0	2.8	2.81	.58
TA	574.19	105.45	612.56	112.46	27.0	2.5	2.85	.66
Co+Frame	606.43	81.47	653.57	86.30	28.6	3.5	2.83	.58
Ta+Frame	585.96	71.74	643.83	86.91	27.8	4.2	2.84	.52
Co+Anim	630.48	61.09	638.33	76.12	28.1	3.1	2.97	.53
TA+Anim	606.96	90.08	656.52	58.24	26.3	2.8	2.87	.64

Table 1. Means and Standard deviations for SAT Verbal/Quantitative, ACT, and GPA

An alpha level of .05 was used for all of the following statistical tests. The means and standard deviations for the data collected on cognitive ability are displayed in Table 1. An ANOVA performed on these data indicated that there were no significant differences in cognitive ability among the groups. A significant positive correlation existed between far transfer performance and all collected measures of cognitive ability except ACT score, $r=.212$, $p<.000$; $r=.136$, $p=.027$; $r=.125$, $p=.034$, for SATV, SATQ, and GPA respectively. Only SATQ correlated in a significantly positive manner with near transfer performance, $r=.132$, $p=.032$.

Condition	Near Transfer		Far Transfer	
	Mean	*SD*	Mean	*SD*
Control	39.88	8.40	100.02	35.55
TA	44.59	6.76	112.48	29.32
Co+Frame	44.79	6.58	103.96	29.31
Ta+Frame	47.42	5.99	122.06	23.77
Co+Anim	43.08	8.56	116.80	28.95
TA+Anim	46.73	5.54	118.18	26.64
	(max poss. = 53)		(max poss. = 139)	

Table 2. Means and Standard Deviations for Near and Transfer Performance by Group.

Means and standard deviations for near and far transfer performance by experimental group are displayed in Table 2. A Multivariate ANOVA was performed on these data. A significant main effect for text type, $F(2, 292)=19.56$, $p<.000$, and study aid type, $F(2, 292)=7.60$, $p=.001$ was identified for near transfer performance. With respect to far transfer performance, significant main effects were identified for both text type $F(2, 292)=9.74$, $p=.002$ and study aid type $F(2, 292)=3.65$, $p=.027$. The interaction of text type and study aid type was not statistically significant, $F(2, 292)=.523$, $p=.593$, and $F(2, 292)=2.083$, $p=.13$, for near and far transfer, respectively.

When total training time, that is, time spent reading or time spent viewing the study aide but not solving problems, was entered in to the analysis as a covariate, both study aid type and text type lost their statistically significant main effect on far transfer performance, $F(2,282)=2.148$, $p=.119$, and $F(2,282)=3.13$, $p=.078$, respectively. The main effect of study aid type and text type on near transfer performance remained statistically significant, $F(2,282)=4.21$, $p=.016$, and $F(2,282)=9.930$, $p=.002$, respectively. The interaction of study aid type and text type also remained statistically

insignificant after the introduction of total training time as a covariate, $F(2, 282)=.914$, $p=.402$, and $F(2, 282)=2.29$, $p=.104$, for near and far transfer respectively.

Regarding near transfer performance, Tukey HSD pairwise comparisons performed with respect to experimental group indicated that Control Only was significantly different from TA Only, Control+Frame, TA+Frame, and TA+Anim. Also, TA+Frame was significantly different from Control+Anim. No other significant differences existed among the groups on near transfer performance.

Regarding far transfer performance, Tukey HSD pairwise comparisons performed with respect to experimental group indicated that Control Only was significantly different from TA+Frame, Control+Anim, and TA+Anim. Also, Control+Frame was significantly different from TA+Frame. No other significant differences existed among the groups on far transfer performance.

An independent samples t-test was performed in order to specifically test hypothesis one, which stated that participants reading the task analysis text would outperform those reading the control text on the near transfer problems. Mean near transfer performance for the groups receiving the control text was lower than that for groups receiving the task analysis text, $M = 42.55$ for control text compared to $M = 46.29$ for task analysis text. The t-test showed that the difference was statistically robust, $t(290)= -4.42$, $p<.000$.

Hypothesis two predicted that a significant linear trend would exist in the far transfer performance of the groups receiving the task analysis text. Trend analysis indicated that no significant linear trend in far transfer performance existed for the groups receiving the task analysis text, $\underline{F}(2, 144) = 1.57$, $\underline{p}=.212$; $\underline{t}(90.71)=.991$, $\underline{p}=.325$. In fact, no statistically significant differences at all existed in the performance of these groups on the far transfer problems. Figure 4 visually depicts the lack of a significant linear trend in the far transfer data for readers of the task analysis text.

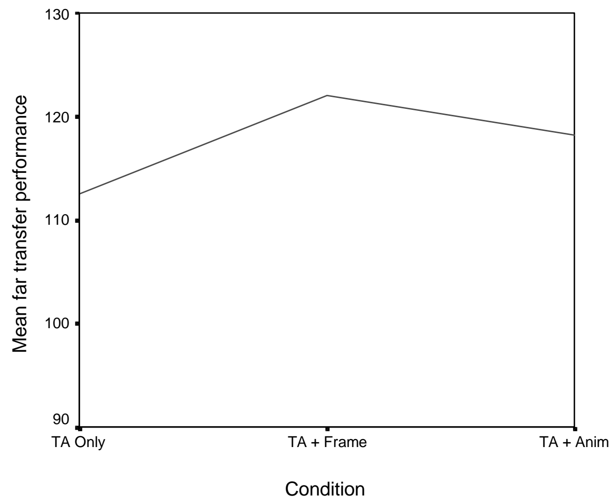
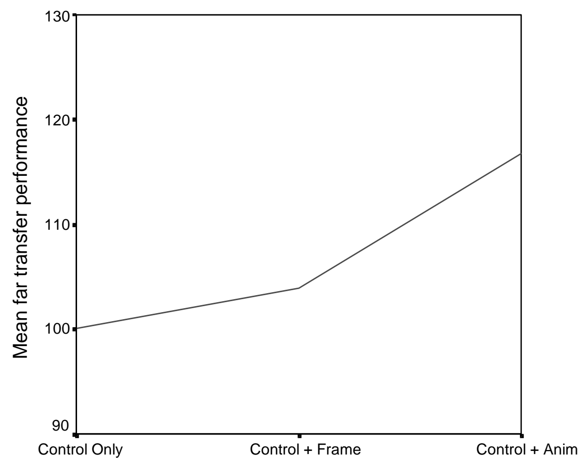


Figure 4. The actual trend obtained with respect to hypothesis two.

Hypothesis three predicted a significant linear trend in the far transfer performance of the groups receiving the control text. Trend analysis showed a significant



linear trend in far transfer performance for these groups, $F(2, 146) = 3.83$, $p=.024$; $t(93.87)=2.58$, $p=.012$. Pairwise comparisons showed that there was no significant difference between Control Only, $M = 100$, $SD=35.5$, and Control+Frame $M=104$, $SD=29.3$ but that one did exist between Control-Only and Control+Anim $M=117$, $SD=28.9$. Figure 5 shows the significant linear trend observed in the far transfer data for readers of the control text.

Figure 5. The significant linear trend obtained with respect to hypothesis three.

CHAPTER VI

DISCUSSION

Hypothesis one predicted that the three conditions using task analysis based text (TA+Anim, TA+Frames, and TA-Only) would outperform the three control text conditions (CO+Anim, CO+Frames, and CO-Only) on the near-transfer problems. Consistent with Seay and Catrambone (1998), this result was obtained in the current study. However, it seems that the 7% performance advantage rendered by use of the task analysis text might not be as substantively as it is statistically significant. By way of comparison, Seay and Catrambone (1998) obtained an almost 20% performance advantage for participants reading the task analysis text compared to those using the control text. So, while this result replicates that of Seay and Catrambone (1998) in letter, it must be suggested that the current study does not fully demonstrate the performance advantage rendered by the use of carefully constructed instructional texts on isomorphic problem performance. Nevertheless, a statistically robust 7% advantage is beneficial and points to the importance of the use of formal analysis of the content domain in the production of instructional materials.

Hypothesis two is clearly not supported by the data. Study aid type did not have a significant linear effect on the far transfer performance of the participants reading the task analysis text. This result refutes the prediction that there would be a graded performance increase on non-isomorphic problems due to the use of the various study-

aides for individuals who were exposed to the task analysis text. In addition, the fact that no significant performance difference existed on the far transfer problems between the TA+Frame and TA+Anim conditions indicates that the type of dynamic motion utilized in the animation rendered no significant performance advantage for these participants.

On the other hand, hypothesis three is supported by the data. Study aid type did have a significant linear effect on the far transfer performance of individuals who read the control text. As predicted this effect went from highest for CO+Anim, to lower for CO+Frame, to lowest for Control only. Also, a statistically significant performance difference was identified between CO+Anim and CO+Frame. This suggests that the dynamic motion of the problem elements displayed in the animation did render a performance advantage over and above that to be gained from the static frames displayed in the CO+Frame condition.

While not fully supporting the centrality of dynamic motion in instructional multimedia, the results of this study do seem to support the parsimonious framework for the role of instructional graphics in learning set forth earlier in this document. To restate the framework briefly, it was suggested that the efficacy of graphical representations in instructional materials is determined by two key factors 1) the degree to which cognitive load is reduced, and 2) the degree to which the problem space is disambiguated. With this in mind, it can be argued that the failure of hypothesis two is based on the fact that the task analysis text used in those three conditions sufficiently disambiguated the problem space such that the study aids, regardless of type, were unable to add any value to the learning process. Further, we see from the success of hypothesis three that in the

more ambiguous problem space left by use of the control text, the study aids operate to render the predicted linear performance advantage. This explanation may suggest an interaction of study aid type and text type, one that we have already shown as statistically insignificant. However, with total study time as a covariate, the interaction does approach statistical significance with respect to far transfer performance, $F(2, 282)=2.285$, $p=.104$.

The task analyses of both the array and Stack were used to inform the design process of both of the study aids. Given that the pedagogical quality rendered by the task analyses is reflected in both study aids, these results are yet another endorsement of the contribution to learning made by the process of explicit and precise specification of the problem domain. Simply, the strength of the task analysis is shown to enhance problem solving performance regardless of its vehicle of delivery. The lack of support for hypothesis two seems to indicate that that strength is finite.

Another reason for the somewhat modest effect of the animation relative to the frame-based study aid may lie in the type of motion depicted in the animation. The animation used in this study depicted what might be called "first order" motion. That is, simple "point-A to point-B" motion. Therefore, the dynamic motion that is displayed in the animation, yet concealed in the frame based aid, is easily inferred allowing users of the frame based aid to accurately mentally animate the scene. Had the motion of the problem elements been more complex, perhaps including an intervening point, say "point-A'," then viewers of the frames based study aid might not have been so successful in inferring the correct pattern of motion and mentally animating the scene in the way

made explicit by the animation. Future work in this area should address the relative effect of dynamic motion in situations where more complex "second" or "third order" motion must be described.

To summarize, the findings of this study replicate those of Seay and Catrambone (1998) regarding the relative effect of task analysis based text on isomorphic problem solving performance, but do so in a less dramatic manner. The combined results of the analyses testing hypotheses two and three suggest that study aids depicting dynamic motion have a greater pedagogical impact upon problem spaces that are left relatively ambiguous by the other supporting instructional materials. That is to say, in situations where reading about a complex problem domain involving time evolving change does not sufficiently disambiguate the problem space for a given learner, a full motion graphical display may serve to substantively enhance the problem solving ability of that learner. Further, under such circumstances, a study aid depicting dynamic motion of the problem elements will have greater pedagogical impact than one that displays the problem elements statically.

So, it seems that under circumstances where reading alone does not sufficiently disambiguate the problem space, dynamic motion proves to be a central aspect of the instructional utility of multimedia study aids. However, in situations where the problem space is already clearly delineated by other instructional materials, the contribution of dynamic motion specifically, and study aids in general, is less crucial.

This study constitutes another step in identifying and codifying the important components of computer generated instructional displays and the circumstances under

which these displays render the greatest pedagogical purchase. We hope that the findings of this study will be valuable in directing future research into the design, creation, and application of computer generated instructional materials, particularly for those illustrating computer algorithms and other complex, rule-governed, and time evolving processes.

APPENDIX A

COVER SHEET

Name: ____________________

Email: ____________________

Have you ever taken a computer science course before? Y N

If so, which course(s)? __________________________

1) I am somewhat familiar with the concepts of computer algorithms, their operation, and use.

Strongly Agree 5 4 3 2 1 Strongly Disagree

2) I have experience programming computers in some language other than HTML (Web page design).

Strongly Agree 5 4 3 2 1 Strongly Disagree

3) I use computers frequently for school and entertainment and am comfortable with their basic operation.

Strongly Agree 5 4 3 2 1 Strongly Disagree

Instructions

In this experiment you will read a series of passages about computer algorithms. Once you have read these passages, you will complete a group of exercises. Following the exercises you will complete a post-test and then be dismissed.

You will see spaces at the bottom of certain pages that look like this:

Write time here_____

When you come to one of these, please right down the time appearing **on the clock at the front of the room**. Do not write in the time at the bottom of a page until you are done reading that page and/or working the exercises on that page entirely.

You are encouraged to work at your own pace, **this is not a race.** How quickly you complete this experiment is not of interest to the experimenter. Rather, it is preferred that you work thoughtfully, trying to understand the concepts so that you can solve the problems successfully. There is no time limit per se, but you will most likely complete this experiment in less than 50 minutes.

If something is not clear or seems confusing, take time to re-read it a few times. If something remains unclear, don't get frustrated, just keep going. The same thing goes for the exercises and post-test; give each problem a solid effort, but if you still don't feel like you know how to do it, proceed to the next one. The experimenter will not answer questions regarding the reading material.

APPENDIX B

A TASK ANALYSIS OF THE ARRAY DATA STRUCTURE

Definitions, Rules, and Implications

Def1 – Array – a group of locations for storing values; often given a name (e.g. ‘b’ or ‘scores’)

Def2 – Element – an individual value stored in an array

Def3 – Size of an array – describes the topological makeup of an array by specifying the number of rows and columns in the array; a 2 by 3, expressed 2x3, might look something like the one below.

A 3x2 array might look like this one.

Rule1 – An array of size ‘r’ by ‘c’ can store ‘r’ times ‘c’ elements.

Examp1 – An array ‘scores’ of size 2x3 can hold six elements.

Def4 – One dimensional array – an array having one row and ‘c’ columns or ‘r’ rows and one column.

Imp1 – A one dimensional array that is one row by 'c' columns can store ‘c’ elements. A one dimensional array that is 'r' rows by one column can store 'r' elements.

Def5 – Index – in a one dimensional array elements are indexed or addressed using their position in the array. By convention, indicess are often a series of numbers starting with '1' that increment from bottom to top or left to right across the array as below. Elements are often entered into an array from bottom to top or left to right as well. As such, each element value is associated with one and only one index.

Examp2 - A one dimensional array ‘Scores’ of size six would have elements Scores[1] through Scores[6] as
below.

Scores

1	2	3	4	5	6
17	12	42	19	28	

Here Scores[3] = 42, because the value stored in Scores[3] is 42.

Def6 - Pointer - a variable used to track indexes of interest in an array. Top, head, and tail are examples of pointers.

Pointers store index values, and never store element values. However, through the use of correct notation,

pointers can be used to indicate element values as follows. The name of the pointer is followed by the name of

the array in which it is involved; a pointer 'top' in array Scores would be written top[Scores].

Examp3 - If top[Scores] is used to point at the index '1' in the array above, then we can write top[Scores] = 1. In order to indicate the value '17' stored in index '1' of array Scores we could write Scores[top[Scores]] = 17 which would be equivalent to writing Scores[1] = 17 since top[Scores] = 1.

APPENDIX C

A TASK ANALYSIS OF THE STACK DATA TYPE

Definitions, Rules, and Implications

Scores

6	
5	28
4	19
3	42
2	12
1	17

Given - Task analysis of array.

Def1 – Stack – a special kind of array with two attributes (size and top) and two operators (push and pop).

Def2 – Stack size – the number of elements a stack can hold; equivalent to array size in that an array of size 'r' defined as a stack can hold 'r' elements.

Def3 – Top – pointer representing the index of the element that was most recently inserted into a stack. The value ‘top’ is the index of the array where the most recently inserted element has been stored. Top is not the value of the stored element itself.

Rule1 –Top is increased or decreased by one every time an element is inserted or removed from the stack.

Examp1 – For the stack ‘Scores’ pictured above, top[Scores]=5 since 28 was the last value inserted into the array. We know that 28 was the last value inserted because arrays are always filled from bottom to top and a stack is a type of array.

Rule2 – When top[Scores]=0 then the array is said to be empty. The initial value of top is always '0' since stacks always start out empty .

Rule3 – When top[Scores]=’r’ the array is said to be full.

Examp2 – For the array ‘Scores’ above, if top[Scores]=6 then the array is said to be full.

Def4 – Push – the command that inserts a value into the stack.

Examp3 – Given the stack above with top[Scores] = 5, Push(32) would first increase the value of top[Scores] by one to top[Scores] = 6. Then the element value '32' would be inserted into the stack at index 6. As a result '32' becomes the most recently inserted element in the stack.

Def5 – Pop – the command for removing the top element from the stack. Pop is not accompanied by any argument since the element value corresponding to the index stored in top will always be removed by the Pop command.

Examp4 - Given the stack above with top[Scores] = 5, Pop would remove the value '28' from the stack since '28' is the value stored in top[Scores] (and, thereby, the most recently inserted element in the stack). After the removal of '28', top[Scores] would decrease by one to top[Scores] = 4 indicating that '19' is now the most recently inserted element in the stack.

Rule4 – If top[Scores]=0 then the Pop command will return an Underflow error since an attempt has been made to remove an element from an empty stack.

Rule5 – If increasing top by one at the beginning of the Push command creates an index value greater than or equal to ‘r’+1 then an Overflow error will be returned since an attempt has been made to push an element onto a full stack.

Pseudocode for the Stack Data Type

```
Define stack Scores

Define variables
        size, type integer
        stack_empty, type boolean
        top[Scores], type integer
        value, type dynamic

Initialize variables
        top[scores]=0
        stack_empty = true

Status check, stack_empty
        If top[scores]=0
                Then stack_empty = true
                Else stack_empty = false

Push(value)
        Top[scores] = top[scores] + 1

        If top[scores] > or = (r+1)
                Then return ‘Error: Stack Overflow’
                Else scores[tops[scores]] = value

Pop
        If stack empty = true
                Then return ‘Error: Stack Underflow’
                Else return scores[top[scores]]

        Top[scores] = top[scores] -1
```

APPENDIX D

A TASK ANALYSIS OF THE QUEUE DATA TYPE (TAIL EMPTY)

Definitions, Rules, and Implications

Scores

1	2	3	4	5	6
17	12	42	19		

Given - Task analysis of array.

Def1 - Queue - a special kind of array with three attributes (size, head, and tail) and two operators (enqueue and dequeue).

Def2 - Queue size - the number of elements a queue can hold; differs from array size in that an array of size 'c' defined as a queue can hold 'c-1' elements.

Def3 - Head - Pointer representing the index of the element that has been in the queue longest. Head is the index where the element that has been in the queue longest is stored. Head is not the value of the stored element itself.

Def4 - Tail - Pointer representing the index where the next element entered into the queue will be stored. The pointer 'tail' is the index where the next element to be entered into the queue will be stored. Tail represents the next available element storage space in the queue.

Rule1 - The value of 'head' changes only when the oldest element is removed from the queue. After the oldest element in the queue is removed, the value of head increases by one, moving right to the next oldest element in the queue. If head[Scores] is at the last index in the queue, it will wrap around to the first index in the array if that index stores an element value.

Examp1 - In the queue above, head[Scores] = 1. If the oldest element is then removed from the queue, the value of head[Scores] will change to head[Scores] = 2.

Rule2 - The index stored in tail[Scores] never holds an element value since tail[Scores] always points to an empty index. When a new element value is inserted into the queue, tail[Scores] is increased by one, moving right to point at the next empty index. If tail[Scores] is at the last index in the queue it will wrap around to the first index in the queue if that index is available (empty).

Examp2 - In the queue above, tail[Scores] = 5. If a new element is then entered into the queue, the value of tail[Scores] will change to tail[Scores] = 6.

Rule3 - When head[Scores]=tail[Scores] then the queue is said to be empty. The initial value of head[Scores] = initial value of tail[Scores] = 1 since queues always start out empty and are filled from left to right.

Rule4 - When head[Scores]-1 = tail[Scores] or head[Scores] = 1 and tail[Scores]=size the queue is said to

be full.

Scores

1	2	3	4	5	6
76	17		45	56	43

Examp3 - For the queue above, if head[Scores] = 4 and tail[Scores] = 3 then the queue would be full.

Scores

1	2	3	4	5	6
98	29	33	51	63	

Examp4 - For the queue above, if head[Scores] = 1 and tail[scores] = 6, then the queue would be full.

Def5 - Enqueue - The command that inserts another element value into the queue; requires an argument.

Scores

1	2	3	4	5	6
78	56	35	45		

Examp4 - In the above queue, if head[Scores] = 1 and tail[Scores] = 5, Enqueue(14) would insert the value '14' into the queue at index 5 and increase tail[Scores] to tail[Scores] = 6. The queue would then be full.

Note1 - Following execution of the enqueue command as outlined in Examp4 the queue would be full since head[Scores] = 1 and tail[Scores] = size.

Def6 - Dequeue - The command that removes the element value that has been in the queue the longest; not accompanied by an argument since the value to be removed is always that stored in Scores[head[Scores]].

Examp5 - In the above queue, if head[Scores] = 1 and tail[Scores] = 5, Dequeue would cause the element value stored in Scores[head[scores]] (in this case 78) to removed from the queue. In addition, head[scores] would be increased to head[Scores] = 2.

Def7 - Queue Underflow - error condition created when a command is issued to remove an element from an empty queue.

Rule5 - A dequeue command issued to an empty queue will cause a queue underflow.

Def8 - Queue Overflow - error condition created when a command is issued to insert an element into a full queue.

Rule6 - An enqueue command issued to a full queue will cause a queue overflow.

Pseudocode for the Queue Data Type

Define queue Scores

```
Define variables
        Size, type = integer
        head[Scores], type = integer
        tail[Scores], type = integer
        value, type dynamic

Initialize variables
        head[Scores] = 1
        tail[Scores] = 1


Define Command - Enqueue(value)
        If ((head[Scores] - 1 = tail[Scores]) OR (head[Scores] = 1 and tail[Scores] = size))
                Return error code "**Queue Overflow**"
        Else
                Scores[tail[Scores]] = value

        If tail[Scores] = size
                tail[Scores] = 1
                tail[Scores ] = tail[Scores ] + 1

Define Command - Dequeue
        If head[Scores] = tail[Scores]
                Return error code "**Queue Underflow**"
        Else
                value = Scores[head[scores]]

        If head[Scores] = size
                head[Scores] = 1
        Else
                head[Scores] = head[Scores] + 1

        Return value
```

Alternate Task Analysis of the Queue Data Type (tail full)

Same as above, but allows a value to be stored in the tail. Only real effect is a change of number of storable elements (Queue Size) and error checking for Queue full.

Alternate Task Analysis of the Queue Data Type (head always one)

Same as above, but head always equals one. This amounts to a line at the bank:

Head guy leaves
the next guy in line moves to the head position
everyone slides up one place closer to the head
new people are appended to the end

Can be used with tail empty or tail full treatments and changes error checking (queue full) and queue size accordingly.

APPENDIX E

ISOMORPHIC PROBLEM SET

(1) Consider the following stack:

Scores

6	
5	28
4	19
3	42
2	12
1	17

A) Show the outcome of each of the following operations on this Stack using the blank Stacks below:

Push(16)
Pop
Pop
Push(48)

6		6		6		6		6	
5	28	5		5		5		5	
4	19	4		4		4		4	
3	42	3		3		3		3	
2	12	2		2		2		2	
1	17	1		1		1		1	

B) What would be the value of Top after this series of operations?

(2) Consider the following stack:

Scores

8	
7	
6	
5	
4	
3	
2	
1	

A) What is the size of this stack?

B) Show the outcome of each of the following operations on this Stack using the blank Stacks below:

Push(42)
Push(23)
Push(39)
Push(97)
Push(40)
Pop
Pop
Push(12)
Push(83)
Push(36)
Push(60)
Pop
Push(12)
Push(17)
Push(43)

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

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

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

8	
7	
6	
5	
4	
3	
2	
1	

(3) Consider the following stack:

Scores

8	
7	
6	
5	10
4	59
3	45
2	22
1	23

A) The last command executed on this stack was Push (10). What is the value of Top?

B) Show the outcome of each of the following operations on this Stack using the blank Stacks below:

Pop
Pop
Pop
Push(14)
Pop
Pop
Pop

8	
7	
6	
5	10
4	59
3	45
2	22
1	23

8	
7	
6	
5	
4	
3	
2	
1	

8	
7	
6	
5	
4	
3	
2	
1	

8	
7	
6	
5	
4	
3	
2	
1	

8	
7	
6	
5	
4	
3	
2	
1	

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

(4) Draw an empty stack of size 4 with indexes.

(5) Draw a stack of size 6 and perform the following operations on it, be sure to show the outcome of each operation, as well as momentary values for top after each operation is completed

Push(12)
Push(51)
Pop
Push(19)
Pop
Pop
Push(62)
Pop
Pop

(6) Draw a stack of size 5 and write out a series of commands that would cause an OVERFLOW error.

(7) Draw a stack of size 6 containing the values 16, 72, 45, and 28 in any order. Then write a series of exactly seven commands that would cause an UNDERFLOW.

APPENDIX F

NON-ISOMORPHIC PROBLEM SET WITH TRANSITIONAL TEXT

The Queue

There is an algorithm similar to the Stack called the Queue. While the Stack is a Last-In-First-Out (LIFO) data structure with one pointer (Top), the Queue is a First-In-First-Out (FIFO) data structure with two pointers (Head and Tail). Recall that in the Stack the index where elements enter and leave the array (top) changes while the other end, lets call it the bottom, stays the same. In a Queue, elements enter at one end (tail) and leave at the other end (head). Drawing on your knowledge of the operation of Stacks, and making these key modifications (FIFO and use of two pointers), answer the following questions. Hint: Overflow errors can only occur when **all** indexes are full.

Non-Isomorphic Problem Set

(1) Consider the following queue with Head = 1 and Tail = 4:

1	2	3	4	5	6	7
23	44	51	36			

A) In the table below, please show the outcome of the following operations. In addition, please indicate the values of Head and Tail after each operation.

	1	2	3	4	5	6	7	Head	Tail
	23	44	51	36				1	4
Enqueue(19)									
Dequeue									
Enqueue(53)									
Dequeue									
Dequeue									

(2) Consider the following Queue:

1	2	3	4	5	6
13	34	97			

A) In total, how many elements can this Queue hold?

B) In the table below, please show the outcome of the following operations. In addition, please indicate the values of Head and Tail after each operation.

	1	2	3	4	5	6	Head	Tail
	13	34	97				1	3
Enqueue(25)								
Dequeue								
Dequeue								
Dequeue								
Enqueue(63)								
Enqueue(52)								
Enqueue(71)								

(2) Consider the following Queue named Ratings with Head = 1 and Tail = 7:

1	2	3	4	5	6	7	8
15	64	72	38	20	59	70	

A) What is the value of Ratings[5]?

B) In the table below, please show the outcome of the following operations. In addition, please indicate the values of Head and Tail after each operation.

	1	2	3	4	5	6	7	8	Head	Tail
	15	64	72	38	20	59	70		1	7
Dequeue										
Dequeue										
Dequeue										
Dequeue										
Dequeue										
Enqueue(19)										
Dequeue										
Dequeue										
Dequeue										
Dequeue										

(4) Consider an empty Queue of size 7. Perform the following operations on it, being sure to show the outcome of each operation as well as momentary values for head and tail.

Enqueue(13)
Enqueue(40)
Enqueue(75)
Enqueue(80)
Dequeue
Enqueue(23)
Enqueue(65)
Enqueue(31)
Dequeue
Enqueue(63)
Enqueue(87)
Dequeue

(5) Consider an empty Queue of size 5. Perform the following operations on it, being sure to show the outcome of each operation as well as momentary values for head and tail.

Enqueue(33)
Enqueue(47)
Enqueue(52)
Dequeue
Dequeue
Enqueue(34)
Dequeue
Dequeue
Dequeue

(6) Consider an empty Queue of size 8. Perform the following operations on it, being sure to show the outcome of each operation as well as momentary values for head and tail.

Enqueue(23)
Enqueue(71)
Enqueue(22)
Enqueue(99)
Dequeue
Enqueue(34)
Enqueue(70)
Enqueue(43)
Enqueue(17)
Enqueue(39)

(7) Consider an empty Queue of size 5. Perform the following operations on it, being sure to show the outcome of each operation as well as momentary values for head and tail.

Enqueue(31)
Enqueue(56)
Enqueue(28)
Enqueue(59)
Enqueue(78)
Dequeue
Enqueue(45)
Enqueue(16)

(8) Of the two algorithms you have dealt with today (Stack and Queue) which would be most appropriate for reading in an entire list of numbers and then outputting them in reverse order? Why is the one you chose most appropriate?

(9) You are designing a waiting list application for the owner of a local restaurant. She is interested in inputting guests' names as they enter the restaurant and then having the name of the guest who has waited longest pop up when a table is available. Since all tables in the restaurant are the same size, party size is not an issue. Which of the two algorithms you have dealt with today (Stack and Queue) is most appropriate for this application? Why is the one you chose most appropriate?

APPENDIX G

ANSWER KEY

Isomorphic Problem Set

(1)

A)

6		6	16	6		6		6	
5	28	5	28	5	28	5		5	48
4	19	4	19	4	19	4	19	4	19
3	42	3	42	3	42	3	42	3	42
2	12	2	12	2	12	2	12	2	12
1	17	1	17	1	17	1	17	1	17

B) 5

(2)

A) 8

B)

8		8		8		8		8	
7		7		7		7		7	
6		6		6		6		6	
5		5		5		5		5	40
4		4		4		4	97	4	97
3		3		3	39	3	39	3	39
2		2	23	2	23	2	23	2	23
1	42	1	42	1	42	1	42	1	42

8		8		8		8		8	
7		7		7		7		7	
6		6		6		6		6	36
5		5		5		5	83	5	83
4	97	4		4	12	4	12	4	12
3	39	3	39	3	39	3	39	3	39
2	23	2	23	2	23	2	23	2	23
1	42	1	42	1	42	1	42	1	42

8		8		8		8	17	8	O

7	60	7		7	12	7	12	7	V
6	36	6	36	6	36	6	36	6	E
5	83	5	83	5	83	5	83	5	R
4	12	4	12	4	12	4	12	4	F
3	39	3	39	3	39	3	39	3	L
2	23	2	23	2	23	2	23	2	O
1	42	1	42	1	42	1	42	1	W

(3)

A) 5

B)

8		8		8		8		8	
7		7		7		7		7	
6		6		6		6		6	
5	10	5		5		5		5	
4	59	4	59	4		4		4	
3	45	3	45	3	45	3		3	14
2	22	2	22	2	22	2	22	2	22
1	23	1	23	1	23	1	23	1	23

8		8		8	
7		7		7	
6		6		6	E
5		5		5	M
4		4		4	P
3		3		3	T
2	22	2		2	Y
1	23	1	23	1	

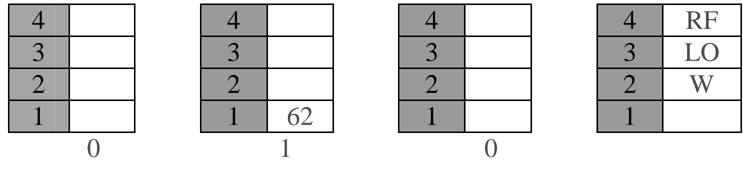
(4)

4	
3	
2	
1	

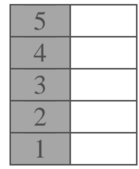
(5)

6		6		6		6		6	
5		5		5		5		5	
4		4		4		4		4	
3		3		3		3		3	
2		2	51	2		2	19	2	
1	12	1	12	1	12	1	12	1	12
	1		2		1		2		1

6		6		6		6	UN
5		5		5		5	DE

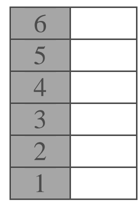


(6)



6 Push commands w/ no Pop commands or
7 Push commands w/ one Pop command or
8 Push commands w/ two Pop commands
etc.

(7)



One Push command and Six Pop commands in any order except :
-Push command cannot be last or second to last command

Non-Isomorphic Problems

(1)

	1	2	3	4	5	6	7	Head	Tail
	23	44	51	36				1	4
Enqueue(19)	23	44	51	36	19			1	5
Dequeue	44	51	36	19				1	4
Enqueue(53)	44	51	36	19	53			1	5
Dequeue	51	36	19	53				1	4
Dequeue	36	19	53					1	3

(2)

A) 6

B)

	1	2	3	4	5	6	Head	Tail
	13	34	97				1	3
Enqueue(25)	13	34	97	25			1	4
Dequeue	34	97	25				1	3
Dequeue	97	25					1	2
Dequeue	25						1	1
Enqueue(63)	25	63					1	2
Enqueue(52)	25	63	52				1	3
Enqueue(71)	25	63	52	71			1	4

(3)

A) 20

B)

	1	2	3	4	5	6	7	8	Head	Tail
	15	64	72	38	20	59	70		1	7
Dequeue	64	72	38	20	59	70			1	6
Dequeue	72	38	20	59	70				1	5
Dequeue	38	20	59	70					1	4
Dequeue	20	59	70						1	3
Dequeue	59	70							1	2
Enqueue(19)	59	70	19						1	3
Dequeue	70	19							1	2
Dequeue	19								1	1
Dequeue										
Dequeue	Underflow									

(4)

	1	2	3	4	5	6	7	Head	Tail
Enqueue(13)	13							1	1
Enqueue(40)	13	40						1	2
Enqueue(75)	13	40	75					1	3
Enqueue(80)	13	40	75	80				1	4
Dequeue	40	75	80					1	3
Enqueue(23)	40	75	80	23				1	4
Enqueue(65)	40	75	80	23	65			1	5
Enqueue(31)	40	75	80	23	65	31		1	6
Dequeue	75	80	23	65	31			1	5
Enqueue(63)	75	80	23	65	31	63		1	6
Enqueue(87)	75	80	23	65	31	63	87	1	7
Dequeue	80	23	65	31	63	87		1	6

(5)

	1	2	3	4	5	Head	Tail
Enqueue(33)	33					1	1
Enqueue(47)	33	47				1	2

Enqueue(52)	33	47	52			1	3
Dequeue	47	52				1	2
Dequeue	52					1	1
Enqueue(34)	52	34				1	2
Dequeue	34					1	1
Dequeue							
Dequeue	Underflow						

(6)

	1	2	3	4	5	6	7	8	Head	Tail
Enqueue(23)	23								1	1
Enqueue(71)	23	71							1	2
Enqueue(22)	23	71	22						1	3
Enqueue(99)	23	71	22	99					1	4
Dequeue	71	22	99						1	3
Enqueue(34)	71	22	99	34					1	4
Enqueue(70)	71	22	99	34	70				1	5
Enqueue(43)	71	22	99	34	70	43			1	6
Enqueue(17)	71	22	99	34	70	43	17		1	7
Enqueue(39)	71	22	99	34	70	43	17	39	1	8

(7)

	1	2	3	4	5	Head	Tail
Enqueue(31)	31					1	1
Enqueue(56)	31	56				1	2
Enqueue(28)	31	56	28			1	3
Enqueue(59)	31	56	28	59		1	4
Enqueue(78)	31	56	28	59	78	1	5
Dequeue	56	28	59	78		1	4
Enqueue(45)	56	28	59	78	45	1	5
Enqueue(16)	Overflow						

(8) Stack because it is a last in first out data structure.

(9) Queue because it is a first in first out data structure.

APPENDIX H

SCORING RATIONALE

1a - 1 awarded for each correct stack, 0 for each incorrect; total - 4
1b - 1 awarded for answer "5", 0 for all other answers; total -1
2a - 1 awarded for answer "8", 0 for all other answers; total - 1
2b - 1 awarded for each correct stack, 0 for each incorrect; total - 15
3a - 1 awarded for answer "5", 0 for all other answers; total - 1
3b - 1 awarded for each correct stack, 0 for each incorrect; total - 7
4 - 1 awarded for correct size of stack, 1 awarded for indices; total - 2
5 - 1 awarded for each correct stack, 1 awarded for each correct to; total - 17
6 - 1 awarded for correct size, 1 awarded for indices, 1 awarded for commands; total - 3
7 - 1 awarded for correct size, 1 awarded for indices, 1 awarded for commands; total - 3

Total for isomorphic set - 54

1 - 1 awarded for each correct queue, 1 awarded for each correct set of head and tail; total - 10
2a - 1 awarded for answer "6", 0 for all other answers; total -1
2b - 1 awarded for each correct queue, 1 awarded for each correct set of head and tail; total - 14
3a - 1 awarded for answer "20", 0 for all other answers; total -1
3b - 1 awarded for each correct queue, 1 awarded for each correct set of head and tail; total - 18
4 - 1 awarded for each correct queue, 1 awarded for each correct set of head and tail; total - 24
5 - 1 awarded for each correct queue, 1 awarded for each correct set of head and tail; total - 16
6 - 1 awarded for each correct queue, 1 awarded for each correct set of head and tail; total - 16
7 - 1 awarded for each correct queue, 1 awarded for each correct set of head and tail; total - 15
8 - five 1's for "stack" and five 1's for proper rationale
9 - five 1's for "queue" and five 1's for proper rationale

Total for non-isomorphic set - 139

APPENDIX I

INTRODUCTORY TEXT

An Introduction to Algorithms

The purpose of this experiment is to introduce you to computer algorithms. As you may know, programmers issue instructions to computers using lists of steps that the computer will use to solve problems and process information. These lists of steps are called **algorithms**. In the following pages we will describe the operation of a particular computer algorithm called the **Stack**. Hopefully, this will illustrate the simplicity and computational elegance of the basic algorithm.

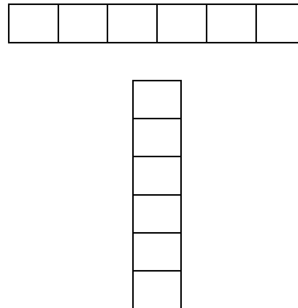
Computer programs and the algorithms they contain are usually written in some form of computer language like BASIC, C++, or Lisp. Interestingly enough, it's not at all necessary to know one of these languages in order to understand algorithms and how they work. All computer instructions, or **code**, can be expressed in an English-like shorthand form called **pseudocode**. Pseudocode is often used to plan and design computer programs before the real "coding" or writing of the actual program takes place. Pseudocode for opening a door might look like this:

```
Grab doorknob
Twist doorknob
Push door open
```

We will use pseudocode here to help explain the operation of the stack algorithm, but before we do that we will need to learn about a data structure called an **Array**.

What is an Array?

An **array** is a group of locations used to store values. Arrays are often given names corresponding to the type of information they store. For example, an array used to store the results from a final exam might be called 'Scores'. Each individual value stored in an array is called an **element**. The size of an array determines the number of elements that can be stored just as the number of dimples in an egg carton determines how many eggs can be held in it. In general, **array size** describes the form of an array by specifying the number of rows and columns in the array. For example, an array with two rows and three columns could hold a maximum of six elements. For our purposes, we will deal only with one dimensional arrays. **One dimensional arrays** have either one row and one or more columns or one column and one or more rows. To illustrate this, below are two examples of one dimensional arrays.



Because of this one dimensional property, we need only one number to describe the size of one dimensional arrays. For example, the one dimensional arrays above are both of size 6. Whether we choose to think of the array as vertical or horizontal is of no real consequence, since the computer does not represent them in such spatial terms.

In a one dimensional array, elements are **indexed** or addressed using their ordinal (numerically successive) position in the array. By convention, indexes are often a series of numbers, starting with '1', that increment from bottom to top or left to right across the array as below. Elements are entered into the array in the same order in which the indexes are incremented. As such, each element value is associated with one and only one index. A one dimensional array named ‘Scores’ of size 6 might look like this.

Scores

1	2	3	4	5	6
17	12	42	19	28	

Note the shaded area under index 6, this indicates an empty index in the array.

Individual elements can be referred to using the name of the array and the index at which the element in question is stored. Here, Scores[3] stores the element value 42. Therefore, Scores[3] = 42. In the same way, Scores[1]=17.

A **pointer** is a variable used to track indexes of interest in an array. Top, head, and tail (covered later) are examples of pointers. Pointers point to indexes, never directly to the element values those indexes store. However, through the use of correct notation, pointers can be used to indicate element values as follows. The name of the pointer is followed by the name of the array in which it is involved; a pointer 'top' in array Scores would be written top[Scores]. If top[Scores] points at the index '1' in the array above, then we would write top[Scores] = 1. In order to indicate the value '17' stored in index '1' of array Scores we would write Scores[top[Scores]] = 17 which would be equivalent to writing Scores[1] = 17 since top[Scores] = 1.

Key Terms

- Array
- Element
- Array Size
- One Dimensional Array
- Index
- Pointer

APPENDIX J

TASK ANALYSIS BASED STACK TEXT

The Stack

Now that we are generally familiar with what an array is, we can talk about a special kind of array called a Stack. Stacks adhere to a number of special rules and constraints that not all other arrays adhere to. As such, Stacks are very useful for performing some operations that other types of arrays are less suited for. Generally, Stacks are referred to as LIFO or last-in-first-out data structures. That is to say that Stacks are programmed so that the most recent element entered into the Stack (or the "last" one) will be the first to be removed. It might be useful here for us to think of the analogy of a stack of plates. The plate on the top of the stack (presumably the one put there last) will be the first to be removed. Essentially, this is how Stack data structures operate. Just like a stack of plates, Stacks fill with elements from the bottom up.

A **Stack** has two attributes; its size and a pointer called "Top." **Stack size** defines the number of elements a stack can hold just like array size did above. Therefore, an array of size 6 defined as a stack would more appropriately be called a stack of size 6. In reality, stack size and array size are equivalent and interchangeable concepts, but since stack size is a more precise designation one should use it when applicable. Returning to our stack of plates analogy for moment, a stack of size six would have room enough for us to store a maximum of six plates.

Top is the name given to the pointer that "points" at the index of the element most recently inserted into the stack. As such, Top stores the value of the index, not the value of the element stored at that index. For example, in the stack below, 34 is the element entered into the Stack last, and therefore, is "on top" of the other value in the Stack. Since it was entered into the Stack at index 2, we would say that Top = 2.

	Index	Value
	5	
	4	
	3	
→	2	34
	1	17

Since Top tracks the index of the element at the top of the Stack in this way, it is increased or decreased by one every time an element is added to or removed from the Stack. If the element value 52 were added to the Stack above, Top would change to Top = 3, just as it would change to Top = 1 if 34 were to be removed. A computationally useful implication of this treatment of Top is the following: if Top = 0 then we know that

the Stack is empty. Also, the initial value of Top is always 0 since Stacks always start out empty. In the same way, when Top =Stack Size then the Stack is full since no other available indexes would exist to store additional elements.

For notational purposes, we would use Top[Ages] to refer to the pointer Top in a Stack named 'Ages'. Further, while Top[Ages] = 2 in the Stack above, Ages[Top[Ages]] = 34. You can see how we substitute the syntax 'Top[Ages]' for the index value '2' in the Ages[<*index*>] statement and are able to get directly at the element value '34'. A bit cumbersome perhaps, but useful as the Stack's operators Push and Pop come into play next.

Push is the command used to insert a new element into the Stack. Push is accompanied by what we call an argument or additional information necessary to perform the command. In the case of Push, the argument is always the element that is being inserted into the Stack. Remember above when we inserted 52 into the Stack? The command that would have allowed us to do so would be written 'Push(52)'. Though Push is a single command, a few subcommands run each time a Push is attempted. First, the value of Top is increased by one to point at the next available index in the stack. If there is such an available storage location then the element value is inserted into the Stack at the index Top points to. We will later talk about what happens if such an index is not available.

Pop is the command used to remove the top element from the stack. Pop is not accompanied by any argument since the element corresponding to the index stored in Top will always be removed by the Pop command. Consider the Stack named "Ratings" of size 5 with Top[Ratings] = 4 below.

	5	
→	4	11
	3	67
	2	89
	1	23

A Pop command issued at this time would first check to see if the Stack is empty. Detecting that it is not empty, the element value '11' is removed from the Stack and then Top[Ratings] is decreased by 1 to Top[Ratings] = 3. This reduction in Top is necessary to indicate that the element stored at Index 3 (which happens to be '67') is now the most recently inserted or top element in the Stack. A subsequent Pop command would remove '67' from the Stack and decrease Top[Ratings] to Top[Ratings] = 2.

Having covered this material, it is now useful to look at the Pseudocode for the Stack data structure so that we might talk about error checking. Below, note that text to the right of '//' represents explanatory comments about the corresponding line of pseudocode.

```
            Name = Scores                                   // defines name of stack
            Size = r                                        // size equals number of rows or 'r'

Define variables
            stack_empty, type boolean                       // sets a true/false flag indicating whether or not Stack is empty
            Top[Scores], type integer                       // indicates that Top will always be an integer value
            element, type integer                           // indicate that the elements will always be integer values

Initialize variables
            Top[scores]=0                                   // indicates that Top[Scores] will start out at 0
            stack_empty = true                              // indicates that true/false will be set to true to start

Begin Loop

Status check, stack_empty                                   // checks whether stack is empty and resets flag accordingly
            If Top[Scores]=0
                        Then stack_empty = true
                        Else stack_empty = false
Push(element)                                               // Specifies code run when Push command is issued
            Top[Scores] = Top[Scores] + 1                   // Top[Scores] is increased by one

            If Top[Scores] > or = (r+1)                     // If  Top[Scores] is greater than or equal to 'r+1',
                        Then return 'Error: Stack Overflow' // then an Overflow error has occurred (Push on a full Stack)
                        Else Scores[Top[Scores]] = element  // If  these conditions do not exist, enter element into Stack at index
                                                            // corresponding to Top[Scores]
Pop                                                         // Specifies code run when Pop command is issued
            If stack_empty = true                           // If stack is empty,
                        Then return 'Error: Stack Underflow' // then an Underflow error has occurred (Pop on an empty Stack)
                        Else return Scores[Top[Scores]]     // If  this condition does not exist, then return the element value
                                                            // stored in the index corresponding to Top[Scores]

            Top[Scores] = Top[Scores] -1                    // Top[Scores] is decreased by one
```

Notice the 'If' statements included in the code sections for both Push and Pop. These 'If' statements check for two specific error conditions that might arise during the repeated performance of Push and Pop operations on a stack. The Push Command checks for an error condition called Overflow. The **Overflow** error occurs when one attempts to Push a new element onto an already full Stack. Since the Stack is unable to handle the new element, the 'Stack Overflow' error message is returned. The Pop command checks for an error condition called Underflow. The **Underflow** error occurs when one attempts to Pop (or remove) an element from an already empty stack. Since there are no elements in the Stack, the 'Stack Underflow' error message is returned.

Key Terms

- Stack
- Stack Size
- Top
- Push
- Pop
- Overflow
- Underflow

APPENDIX K

CONTROL STACK TEXT

What is a Stack?

A **Stack** is a special type of array in which the element removed from the array by the DELETE operation is prespecified. In a stack, the element deleted from the array is the one most recently inserted: the stack implements a *last-in, first-out,* or *LIFO*, policy. In this section we show how to use a simple array to implement a stack.

The INSERT operation on a stack is often called **PUSH**, and the DELETE operation, which does not take an element argument, is often called **POP**. These names are allusions to physical stacks, like stacks of plates. The order in which plates are popped from the stack is the reverse if the order in which they were pushed onto the stack, since only the top plate is accessible.

As shown in Figure1, we can implement a stack of at most *n* elements with an array named Scores. The **size** of a stack, denoted by the numerical value *n*, determines the number of elements that the stack can hold. The stack has an attribute **top**[Scores] that indexes the most recently inserted element. The stack consists of elements Scores[1..top[Scores]], where s[1] is the element at the bottom of the stack and Scores[top[Scores]] is the element on top.

When top[Scores]=0, the stack contains no elements and is empty. The stack can be tested for emptiness by the query operation STACK_EMPTY. When the stack contains *n* elements it is full and can be tested for fullness by the query operation STACK_FULL

FIGURE 1 An array implemented as a Stack and named Scores. Stack elements appear only in the unshaded positions. (a) the Stack named Scores has 4 elements. The top element is 91. (b) Stack Scores after Push(17) and Push(32) have been executed. (c) Stack Scores after the Pop has removed the element 32, which is the one most recently pushed. This makes 17 the top element.

If an empty stack is popped, we say the stack **underflows**, which is an error. If top[Scores] exceeds *n*, the stack **overflows***,* also an error.

The stack operations can each be implemented with a few lines of pseudocode.

```
Check STACK_EMPTY
1   if top[Scores]=0
2       then Stack_Empty =  TRUE
3       else return Stack_Empty = FALSE

Check STACK_FULL
1   if top[Scores] = n
2       then Stack_Full = TRUE
3       else return Stack_Full = FALSE

PUSH (x)
1   if Stack_Full = TRUE
2   then error "overflow"
3   else top[Scores] = top[Scores]+1
4         Scores[top[Scores]] = x

POP
1   if Stack_Empty = TRUE
2      then error "underflow"
3      else return Scores[top[Scores]+1]
4              top[Scores] = top[Scores]-1
```

Key Terms

- Stack
- Push
- Pop
- Size
- Top
- Underflow
- Overflow

REFERENCES

Brown, M.H. (1988) . Perspectives on algorithm animation. Proceedings of the ACM SIGCHI Conference on Human Factors, USA, 88, 33-38.

Baecker, R. (1998). Sorting out sorting: A case study of software visualization for teaching computer science. In J. Stasko, J. Domingue, Brown, M.H., and Price, B.A. (Eds.), Software visualization: Programming as a multimedia experience. Cambridge, MA: MIT Press, 369-381.

Byrne M. D. , Catrambone R. , & Stasko J. T. (in press). Examining the effects of animation and prediction in student learning of computer algorithms. Computers and Education.

Hansen S., Schrimpsher, D., & Narayanan N.H. (1998). From algorithm animations to animation-embedded hypermedia visualizations. Submitted to HyperText '98 Conference.

Hays, T. (1996). Spatial abilities and the effects of computer animation on short term and long term comprehension. Journal of Educational Computing Research, 14(2), 139-155.

Hegarty, M. (1992). Mental animation: Inferring motion from static displays of mechanical systems. Journal of Experimental Psychology: Learning, Memory, and Cognition, 18(5), 1084-1102.

Kaiser, M.K., Proffitt, D.R. Whelan, S.M., & Hecht, H. (1992). Influence of animation on dynamical judgements. Journal of Experimental Psychology: Human Perception and Performance, 18(3), 669-690.

Kalyuga S., Chandler, P., & Sweller, J. (1998). Levels of expertise and instructional design. Human Factors, 40(1), 1-17.

Paas, F. G. W. C. (1992). Training strategies for attaining transfer of problem-solving skill in statistics: A cognitive-load approach. Journal of Educational Psychology, 84(4), 429-434.

Price, B. A. , Baecker, R. M. , & Small, I. S. (1993). A principled taxonomy of software visualization. Journal of Visual Languages and Computing, 4(3), 211-266.

Reiber, L. P. (1991). Animation, incidental learning, and continuing motivation. Journal of Educational Psychology, 83(3), 318-328.

Reiber, L. P. & Kini, A. S. (1991). Theoretical foundations of instructional applications of computer-generated animated visuals. Journal of Computer-Based Instruction, 18(3), 83-88.

Rigney, J. , & Lutz, K. A. (1976) . Effect of graphic analogies of concepts in chemistry on learning and attitude. Journal of Educational Psychology, 68(3), 305-311.

Scaife, M., Rogers, Y. (1996). External cognition: How do graphical representations work? International Journal of Human-Computer Studies, 45, 185-213.

Seay. A.F. & Catrambone. R. (1998). The effect of dynamic displays of information on learning and transfer: An algorithm animation approach. Unpublished Manuscript.

Stasko, J.T. (1990). TANGO: A framework and system for algorithm animation. Computer, 23 (9), 27-39.

Stephenson, S. D. (1994) . The use of small groups in computer-based training: A review of recent literature. Computers in Human Behavior, 10(3), 243-259.