“WHY SHOULD I CARE?” MAKING PROGRAMMING ASSIGNMENTS RELEVANT FOR NON-MAJORS

Stephen Davies
University of Mary Washington

ABSTRACT

Introductory courses in computer science are increasingly being offered as electives to non-majors. Many of these students do not find programming examples that involve math, science, or business topics compelling, and would be better reached by applications that deal with concerns relevant to college life. This article presents several examples of programming assignments that have met with approval from non-majors, draws the connection to the college experience, and gives implementation guidelines for instructors.

INTRODUCTION

As technology has permeated our society, introductory programming courses have played to a broader audience. Once a niche field whose barrier to entry could only be crossed by a few devoted specialists, computer science is now accessible to the mainstream student body, at least at the introductory level. These students reflect all walks of life, and all areas of interest. They have the amenities of third-generation languages and tools at their fingertips, and hence the potential to apply computing power successfully. But they need a reason to make the effort. Many of these students are not inclined to math and the sciences, and so the typical introductory examples of converting Celsius to Fahrenheit or computing the first hundred prime numbers are not interesting to them. Indeed, such examples may even be barriers to learning, reinforcing the unfortunate notion that computer science, like math, is a frightening, alien universe full of strange concepts and indecipherable symbols.

Students of all persuasions, and especially those majoring in the humanities, need examples and projects that appeal to their natural interests. Experience has shown that when learners are interested in the topic they are studying, their level of motivation and chances of success rise dramatically.[3] If we can find examples and create assignments that are relevant to the average college student's life, our students are more likely to grasp the features of the problems at hand, and invest the mental effort necessary to solve them.

This article presents a number of sample assignments that have met with student approval simply because of their subject matter. In each case they illustrate the same programming concepts as more traditional examples do, but attempt to appeal to a particular concern or interest of college life. They are applicable to any introductory programming class, even those intended for non-majors. (The examples assume a C++ implementation, since at Mary Washington our 100-level programming course typically contains a small number of future computer science majors who will need C++ expertise for the 200-level course. Nothing depends on C++ in particular, however.)

ALGORITHMS AND HEURISTICS: APPEALING TO THE AESTHETIC

One issue relevant to student life is fashion. The National Retail Federation[1] reports that college students spend nearly $8 billion per year on new clothing and shoes, more than half what they spend on textbooks. And of course acceptance by one's peers – which often involves physical beauty and attractive attire – is of great concern to nearly all teenagers. One way to generate interest in designing algorithms, then, is by tapping into this natural concern with personal appearance.

Studies into the methodologies of fashion designers[9] reveal a multi-pronged process that combines both intuition and rational decision making. Fashion design can be categorized as a "wicked problem" in Rowe's taxonomy of design problems[7], in that the constraints of the problem and solution are interrelated, and furthermore that the problem requirements are so ill-defined that they can be continually reformulated with no definitive stopping point. Such a domain stands in stark contrast to the well-understood, precisely constrained problems usually used to demonstrate algorithm construction. It offers an excellent opportunity, however, to tap into students' natural creativity (in a realm decidedly unconventional for computer science) and to approach the question, "how can one materialize an intuitive thought process tangibly enough to formulate a heuristic that approximates it?"

Consider the highly simplified problem of choosing an appropriate background color for a portrait. Assume for the moment that the subject of the work is female, and that a small subset of her physical characteristics have been captured in a simple C++ struct:

```
struct FemaleSubject {
     enum { BLONDE, BRUNETTE, BLACK, AUBURN } hairColor;
     enum { CRIMSON, FLORAL, PLUM, NAVY, WHITE, BLACK } dressColor;
};
```

A whimsical yet thought-provoking exercise would be to design an algorithm for automatically choosing a background color for a particular subject's portrait. A simple approach might be:

```
typedef enum { LILAC, SLATE, ROSE, AQUA, TAUPE } BACKGROUND_COLOR;
BACKGROUND_COLOR bestBackgroundColor(FemaleSubject subj) {
     switch (subj.dressColor) {
     case NAVY:
          return LILAC;
     case PLUM:
          if (subj.hairColor == BRUNETTE)
               return ROSE;
          else
               return SLATE;
     case FLORAL:
          if (subj.hairColor == BRUNETTE  ||  subj.hairColor == BLACK)
               return LILAC;
          else
               return AQUA;
     default:
          return TAUPE;
     }
}
```

One could imagine such a function being incorporated into the software for an automated photo-taking booth at a party, such that the background color automatically changed in response to the apparel of each user. The algorithm is imperfect and sure to provoke in-class controversy when presented, and this is part of the goal. It demonstrates two intriguing points: first, that computers can be applied not only to scientific problems, but also to aesthetic problems; and second, that computer programs can be used to solve ill-defined problems in an approximate way. There are no right or wrong answers to this "algorithm," and students are encouraged to design their own. The task appeals to the creative side of the math-phobic novice, while disarming possible angst about the presumed coldly logical nature of computers. A follow up assignment would be to create a similar struct for male subjects (perhaps with hair color, suit style, and tie pattern as fields), and then to write a function that took one male and one female as parameters (presumably representing a couple that were to appear in a prom or engagement portrait) and determined the best background color for the pair.

LOOPS: MAKING CREDIT CARD CHOICES COME TO LIFE

Consumer debt is an issue plaguing many Americans, and college students are perhaps the single most impacted demographic group.[6] A recent study[4] found that the average credit card debt in a random sample of college students was over $1500, a considerable sum for an age range with minimal financial resources. Studies also report that applicants are often ignorant about how credit cards are used, and specifically that students who acquire credit cards with no parental involvement are less likely to use them responsibly than those whose parents are engaged in the decision-making process.[5] Such students also frequently request more information about credit cards, suggesting that a programming assignment on this topic would be welcomed.

When one makes a purchase on credit, and repays the sum incrementally over time, one pays more for the item in the end than the original price due to accumulated interest. This is a simple but effective way to think about the implications of using credit: the total amount the customer pays for an individual item is a revised “effective price” somewhat greater than the store's “asking price.” College students, facing certain financial choices for the first time, may not fully understand this process. And even if they grasp the general concept, accurately quantifying the effects of a particular payment strategy is another matter entirely. Considerable light can be shed on these matters by constructing a loop that simulates a credit card's balance over time. This allows students to experiment with different interest rates, item costs, and rates of repayment, and see their long-term financial impact. Such a program further demonstrates the computer program's applicability to practical matters.

The assignment is well within the reach of introductory programming students. First, the program prompts the user for the store price of a purchased item, the annual rate of the credit card used, and a (constant) monthly rate of repayment. The annual interest rate is converted to a monthly rate by solving the equation $\texttt{(1+monthly)}^{12} = \texttt{1+annual}$; this yields a good opportunity for students to create a subroutine, and demonstrates the use of the C++ “pow()” function. A simulation then occurs (using a loop) in which each month the repayment is subtracted and the interest applied. The “effective price” of the item can be found in one of two ways: either by summing up all of the interest applied to the balance each month, and adding it to the original purchase price, or by simply adding up the monthly payments that were made, each of which is a fixed value except the last, which is some lesser amount.[1]

To make the assignment more effective, students should be encouraged to “window shop” for actual purchases they might be interested in, to use the actual interest rate from their credit card or a card they may qualify for, and to specify a realistic monthly payment. One student found that an $850 guitar purchased on his 24%-APR credit card and paid off at $20/month would eventually cost $1,569.57 to finance, whereas if paid at $40/month only $1,057.92 would come out of his pocket. Such hard facts are sobering realizations for students who may not have appreciated the impact of certain financial choices, and, as with all these examples, generate interest in designing and using programs.

STRUCTS: FINDING A DATE (OR MATE)

Finally, one area of intense focus on college campuses is dating. Certainly a good deal of the average student's social life revolves around negotiating romantic relationships, so a programming assignment whose stated aim is to shed light on this process could be appealing indeed. A classic example is the “matchmaker” program, which finds the optimal pairings between members of two

1 The business notion of the “time value of money” (i.e., a dollar paid today is actually more costly than a dollar paid later, due to the effects of inflation) is not taken into account with the basic version of this program, although certainly more advanced and motivated students could incorporate this effect.

groups, based on profiles of characteristics.

Each student begins by designing a brief survey about personal interests, hobbies, or values, and administers it to ten of their male acquaintances, and ten female. The surveys should include perhaps ten questions, and the answers can be multiple-choice, numeric, or scaled. Some examples (from actual student surveys):

If you could be a Star Wars character for a day, which would you choose? *(multiple choice)*
____ Yoda ____ Han Solo ____ Luke Skywalker ___ Darth Vader

How many times a day do you think about food? _______ *(numeric)*

On a scale of 1 to 10, how important is working out to you, where 1 indicates "couch potato" and 10 indicates "very athletic"? _____ *(scaled)*

There are no constraints on topic, and originality is encouraged. The point is to make the students consider the limitations of abstraction: given that a complex individual is going to be represented in a matchmaking application as a small set of discrete values, what should be measured? How can "compatibility between potential dating partners" best be captured, and what questions should be asked to ascertain it?

Answers to the survey questions are included in a "dating profile," excellently illustrating the use of a C++ struct (or class, in an "objects first" curriculum.) The reason why the domain of answers is restricted to the three categories above, of course, is so that two profiles can be easily compared. The next creative endeavor the students undergo, then, is the design of a "compatibilityScore()" function, which takes as inputs two profiles and returns an integer representing the relative likelihood of a successful relationship between them.[2] This, too, is a thought-provoking exercise, and fuels interest in the program: how exactly do we determine whether two people are compatible? Do we weight some questions more heavily than others? Do we consider some differences in answers as nevertheless closer than other differences - "liberal" being considered closer to "moderate" than "liberal" is to "conservative," for example? Do similar answers always mean high compatibility, or should we actually award points for differences in some cases, figuring that "opposites attract?" These are questions that make computing come to life. Students forget that they are writing C++ code, and instead think about a problem that seems worth solving: unraveling the mystery of the sexes.

The program itself is structured in two halves: functions that the students write on their own (the main() and compatibilityScore() functions), and a small number of utility functions written together in class. (See Figure 1.) The assignment seems to work best by combining these two activities. During the (say) three weeks that students are working on the project, a different utility function is written by the instructor during each lecture. The instructor makes clear the purpose of the function, walks through the code, and explains how it will fit into the overall structure of the program. This provides opportunity to introduce a few more advanced programming concepts (e.g., pass-by-reference in the swap() function; indexing one array based on the indices found in another in the printMatchups() and compositeScore() functions) to a captive audience. Students appreciate the necessity of such techniques when they see how they apply to a real-world problem, rather than an artificial example. It also demonstrates the necessity of functions, which many novices seem reluctant to create on their own.

Of the utility functions to be written in class, the algorithm for computing all the permutations is the most difficult. It is perhaps best simply given to the students as a "black box" component for their

2 The integer returned does not need to be chosen from any absolute scale (for instance, "0" meaning "not compatible at all" and "100" meaning "perfectly compatible.") Only relative comparisons are made, so as long as higher return values indicate more compatible couples, the program will work perfectly.

use, though it is not beyond the understanding of more motivated students who wish to study it. Both recursive and iterative solutions exist[8, p.140; 2, p.71]; one way it can be implemented is with a "nextCombination()" function that, each time it is called, modifies an array of integers that represents which men are paired with which women in one possible combination.

The main routine, though somewhat lengthy, is conceptually simple: it prompts the user for the name and the survey answers for each of the ten women, then for each of the ten men. These are stored in two arrays of structs, perhaps called "women" and "men." It instantiates an array of ten integers called "pairings," initialized to { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }, which represents which man is paired with which woman in a particular combination. (This initial value specifies that woman #0 is paired with man #0, woman #1 with man #1, and so on.) Then, the program loops through all possible combinations of pairs between the two groups by calling "nextCombination()" repeatedly: each time this function is called, the contents of the array are shuffled to represent the next combination. compositeScore() can then be called with the two arrays of structs and the "pairings" array; it calls the student's compatibilityScore() function for each of the ten pairs, and returns a grand total score for that combination. A straightforward "find the maximum" algorithm is then used to arrive at the combination with the best overall score (using an arrayCopy() function to copy the "pairings" array to a "bestPairingsSoFar" array each time a new "best" combination is found.)

Functions written by students alone

- main() - Prompts user for survey results, examines all possible combinations of pairs, and prints out the best pair.
- compatibilityScore(Friend man, Friend woman) - Computes a score intending to represent the compatibility between one pair of potential matches.

Utility functions written together during lecture

- compositeScore(Friend men[], Friend women[], int pairings[]) - computes a grand total score for an entire set of ten pairs
- printMatchups(Friend men[], Friend women[], int pairings[]) - prints the names of each pair for a particular combination
- nextCombination(int pairings[]) - modifies an array of indices to hold the next permutation (returns false if no more permutations)
- arrayCopy(int copyTo[], int copyFrom[]) - used by main()
- swap(int &a, int &b) – used by nextCombination()

Figure 1. Functions comprising the entire "Matchmaker" project.

This project offers several compelling features. First, as described above, it appeals naturally to students' curiosity. It also provides good opportunities for creativity, much of which does not involve programming. This is a welcome reprieve for some humanities students who may find math and science subject matter intimidating. The creativity, too, is constrained in a way as to maximize the chances of success. Students are not attempting to design a completely original program on a brand new topic, but rather are given one component of a successfully structured program – the makeup of the "dating profile" structure, and the associated compatibility function – to inject their creativity into. This allows them free rein to decide how people will be represented and compared (which is the most interesting facet of the program) without allowing the scope of the project to drift which would jeopardize the chances of a successful end result.

Finally, this project features one characteristic that many introductory examples do not; namely, its problem cannot be solved *except* by computer. Most programming assignments in entry-level courses involve carrying out computations that the student could actually have performed by hand in a fraction of the time it took to write the program. The entire programming exercise, therefore, is a "suspension of disbelief" in which the student has to imagine the hypothetical scenario of a facing a more involved problem not so easily solved manually. In contrast, this program examines all 10! (or about 3.6 million) combinations of pairings, which, if the compatibility of ten pairs of surveys could be computed by hand in (say) five minutes, would take over 34 years to perform manually. Not only the problem, then, but the program itself have obvious real value, which further strengthens motivation.

An especially valuable "post mortem" exercise is requiring students to iterate on their compatibilityScore() function. After examining the pairings that the program ultimately chose and judging whether they line up with the student's intuition, the student attempts to tweak their compatibility algorithm to get a better outcome. This offers the instructor an opportunity to mention the concept of machine learning, since the students here are essentially training their system by hand on known data, presumably to arrive at a more effective compatibility measure for the general case.

CONCLUSIONS

No formal studies were conducted to try and quantify the appeal of these assignments. Anecdotal evidence merely indicates that they generate considerable interest, both in and out of class, especially compared with more traditional examples used in previous versions of this course (prime numbers, computing tax brackets, etc.) When used, retention rates during the semester seemed admirably high (only two dropouts out of 51 students), as did the overall quality of work. In particular, the number of students "pushing through to completion" was impressive: only six students out of 49, for instance, failed to complete a fully-functional matchmaker program, despite the fact that it was an extraordinarily lengthy and complex project for a 100-level programming course with no prerequisites. This suggests that the interestingness of the problem itself was at least partially what spurred students on past the many obstacles they faced.

Unfortunately, however, no obvious correlation is apparent between the use of these assignments and the number of students continuing on to pursue a computer science major. Eight students from these 51 expressed such interest (and registered for the "for-majors" course at the 200-level), which is similar to the rates we have traditionally seen. If anything, this demonstrates that although "relevant" assignments may make a more appealing programming experience for novices, they do not by themselves dispel the notion that the field as a whole is comprised of less-compelling material.

REFERENCES

[1] BIGRESEARCH, LLC. 2005. *The National Retail Federation 2005 Back-to-College Consumer Intentions and Actions Survey*. Available at http://www.nrf.com.

[2] DIJKSTRA, E.W. 1976. *A Discipline of Programming*, Prentice-Hall.

[3] ELLIOTT, E.S., AND DWECK, C.S. 1988. Goals: An approach to motivation and achievement. *Journal of Personality and Social Psychology 54*, 5-12.

[4] NORVILITIS, J.M., SZABLICKI, P.B., AND WILSON, S.D. 2003. Factors influencing levels of credit-card debt in college students. *Journal of Applied Social Psychology 33 (5)*, pp. 935-947.

[5] PALMER, T.S., PINTO, M.B., AND PARENTE, D.H. 2001. College Students' Credit Card Debt and the Role of Parental Involvement: Implications for Public Policy. *Journal of Public Policy and Marketing 20 (1)* pp.105-113.

[6] ROBERTS, J.A. 1998. Compulsive Buying Among College Students: An Investigation of Its Antedecents, Consequences, and Implications for Public Policy." *Consumer Affairs 32 (2)*, pp.295-319.

[7] ROWE, P.G. 1987. *Design Thinking*. The MIT Press, Cambridge, MA.

[8] SEDGEWICK, R. 1977. Permutation generation methods, *Computing Surveys 9 (2*), pp.137-164.

[9] SINHA, P. 2002. Creativity in Fashion. *Journal of Textile and Apparel, Technology and Management 2,* Issue IV.