# Teaching Problem-Solving Techniques Utilizing a Running-Example Problem

Mark Fienup
Computer Science Dept.
University of Northern Iowa
Cedar Falls, IA  50614-0507
(fienup@cs.uni.edu)

## Introduction

The problem-solving techniques such as greedy algorithms, divide-and-conquer, dynamic programming, backtracking, and branch-and-bound are important computer science concepts that typically are taught in an Algorithms course.  When each of these problem-solving techniques is introduced to students, illustrative example problems are used to demonstrate the technique and make the abstract problem-solving technique more concrete.  Unfortunately, textbooks [1][2][3] (and teachers) use different example problems when initially introducing each problem-solving technique.  This paper points out the benefits of utilizing the same example problem when introducing each of the problem-solving techniques.  We will call this a "running-example problem."  Additionally, this paper points out specific running-example problems that are useful when introducing all of the problem-solving techniques.

A good running-example problem is the coin-changing problem where the goal is to make change with the fewest number of coins assuming that an unlimited supply of each type of coins is available.  The main criteria for selecting a good running-example problem used to introduce all of the problem-solving technique are that 1)  the problem should be amenable to all problem-solving techniques taught, and 2)  the problem should be easily understood by students, i.e., the problem should be as concrete as possible.  Thus, students are better able to focus on the problem-solving technique and not the details of the problem being solved.

## Running-Example 1:  Coin-changing Problem

In the coin-changing problem, the goal is to make change with the fewest number of coins assuming that an unlimited supply of each type of coin is available.  When asked to solve this problem, students quickly come up with the obvious greedy algorithm of giving back the largest coin first that is less or equal to the change remaining to be returned.  Figure 1 shows the choices this greedy algorithm would making for 41 cents change with coin types of 1, 5, 10, 25, and 50 cents.

Students are surprised to learn that this greedy algorithm does not give the optimal solution for a different set of coin types.  For example, making change for 41 cents with coin types of 1, 5, 10, 12, 25, and 50 cents results in six coins being returns as shown in Figure 2.  Clearly, a better four coin solution exist, i.e., return 25, 10, 5, and 1-cent coins.

```
  41
- 25 <--  Return the largest coin less than or equal to 41 cents, i.e., 25-cent coin
  16
- 10 <--  Return the largest coin less than or equal to 16 cents, i.e., 10-cent coin
   6
 - 5 <--  Return the largest coin less than or equal to 6 cents, i.e., 5-cent coin
   1
 - 1 <--  Return the largest coin less than or equal to 1 cents, i.e., 1-cent coin
   0
```

Figure 1. Greedy algorithm example for 41-cents and coin types of 1, 5, 10, 25, and 50.

To solve the Coin-changing problem optimally, we can employ the problem-solving techniques of divide-and-conquer or dynamic programming. However, in order to use these techniques we must first establish a recursive relationship that allows us to solve the initial problem (e.g., making change for 41 cents) in terms of smaller instances of the same problem. After teaching algorithms for several semester, this is probably the hardest skill for most students to learn when applying any of the problem solving techniques. To help the students see a recursive relationship for making change, I suggest thinking about the choice of the first coin to be given back. If we tried each type of coin, we would be left with a smaller remaining amount of change to return as shown in Figure 3. If we knew the fewest number of coins needed for each of smaller remaining amounts of change, we could decide the best choice for the first coin to give back.

Thus, our recursive relationship FewestCoins for the coin-changing problem would be

$$\text{FewestCoins(change)} = \begin{cases} \underset{\text{coin} \in \text{CoinSet}}{\text{minimum}}(\text{ FewestCoins(change - coin ) }) + 1 & \text{if coin} \notin \text{CoinSet} \\ 1 & \text{if change} \in \text{CoinSet} \end{cases}$$

```
  41
- 25 <--  Return the largest coin less than or equal to 41 cents, i.e., 25-cent coin
  16
- 12 <--  Return the largest coin less than or equal to 16 cents, i.e., 12-cent coin
   4
 - 1 <--  Return the largest coin less than or equal to 4 cents, i.e., 1-cent coin
   3
 - 1 <--  Return the largest coin less than or equal to 3 cents, i.e., 1-cent coin
   2
 - 1 <--  Return the largest coin less than or equal to 2 cents, i.e., 1-cent coin
   1
 - 1 <--  Return the largest coin less than or equal to 1 cents, i.e., 1-cent coin
   0
```

Figure 2. Greedy algorithm example for 41-cents and coin types of 1, 5, 10, 12, 25, and 50.

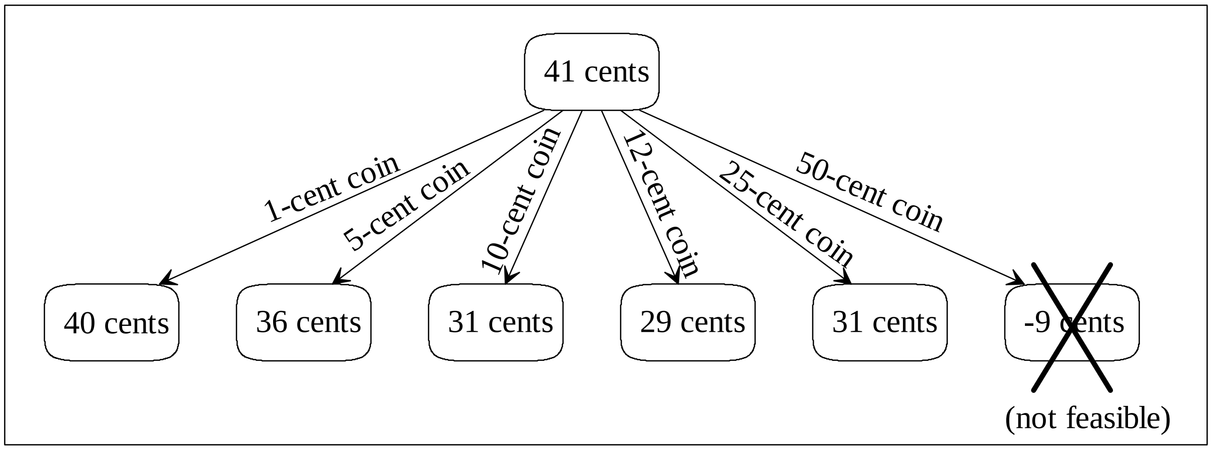


Figure 3. Remaining change after return each possible coin {1, 5, 10, 12, 25, 50} for 41 cents.

While the recursive divide-and-conquer algorithm is relatively straight forward to implement, it performs many redundant calculations which make it impractical. For example, the partial recursion tree for 41 cents with the set of coins {1, 5, 10, 12, 25, 50} leads to the 16-cent subproblem many times as shown in Figure 4. Each time the 16-cent subproblem is encountered it performs the same calculation from scratch.

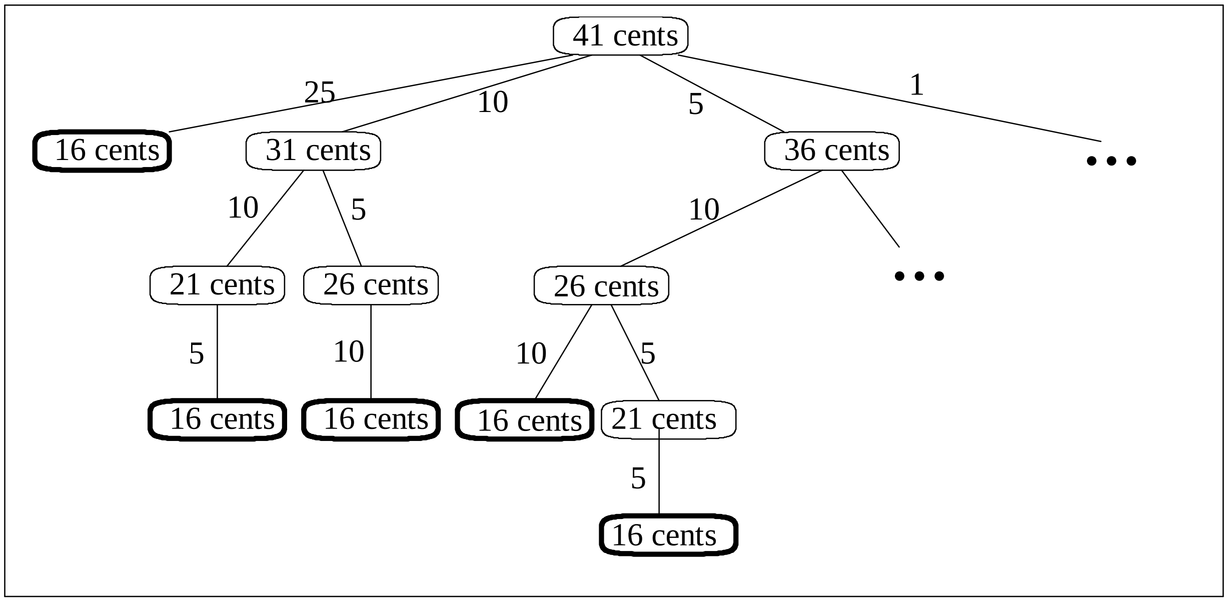
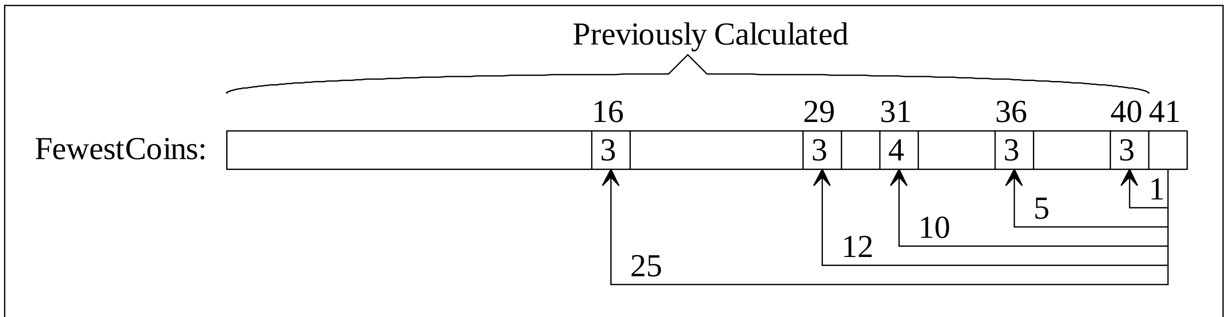


Figure 4. Some of the occurrences of the 16-cent subproblem during the initial 41-cent problem.

The redundant calculations of the divide-and-conquer algorithm makes it an $\theta(n!)$ algorithm, i.e., you don't want to wait for a solution when the change is bigger than about 75 cents. The redundant calculations of divide-and-conquer provides good motivation for the dynamic programming solution where each subproblem is solved exactly once, its answer is stored, and looked up each time it is needed again. The dynamic programming solution of the Coin-change problem fills an array FewestCoins from 0 to the amount of change to be returned. An element of

FewestCoins stores the value of the fewest number of coins necessary for the amount of change corresponding to its index value. The previously defined FewestCoins recursive relationship is used to fill an array element. Since the array is filled from smaller to larger indices, we only need to look up the solution of the smaller subproblems needed. Figure 5 illustrates the subproblem solutions necessary to calculate the fewest number of coins for 41 cents using the set of coin types {1, 5, 10, 12, 25, 50}.



FewestCoins(41) = minimum(FewestCoins(40), FewestCoins(36), FewestCoins(33), FewestCoins(29), FewestCoins(16) ) + 1 = 3 + 1 = 4

Figure 5. Dynamic program calculation for 41 cents using coin types of {1, 5, 10, 12, 25, 50}.

If we record the best first coin to return (found in the "minimum" calculation) for each change amount in an array BestFirstCoin, then we can easily recover the actual coin types to return. Figure 6 shows the BestFirstCoin array for the 41-cent solution with coin types {1, 5, 10, 12, 25, 50}.

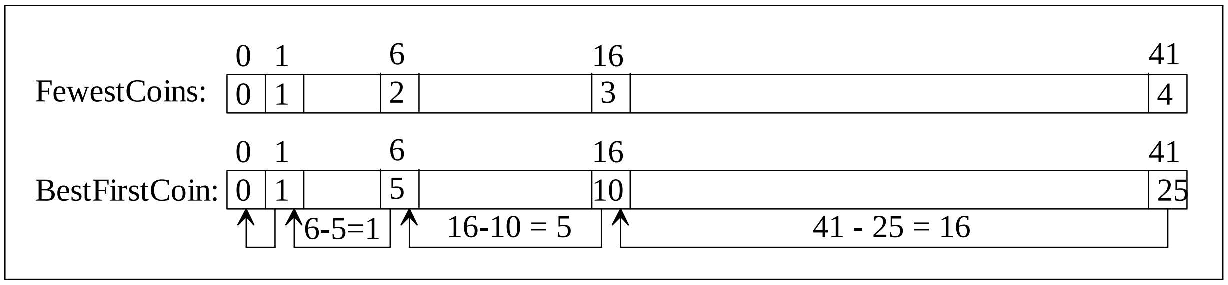


Figure 6. Relevant BestFirstCoin entries for 41 cents using coin types of {1, 5, 10, 12, 25, 50}.

Backtracking is another problem-solving technique for optimization problems that can be demonstrated using the Coin-change problem. The concept of backtracking is probably best explained by using a state-space/search-space tree that shows the recursive calls performed during backtracking. Backtracking performs a depth-first search of the state-space tree. To improve efficiency, we stop following branches of the tree that either cannot lead to a solution or cannot lead to a better solution than you already have found, i.e, you want to "prune" the state-space tree. The criteria for pruning are:

1) giving back a coin that is worth more that the amount of change, or
2) giving back one less coin than the best solution and having more change to return.

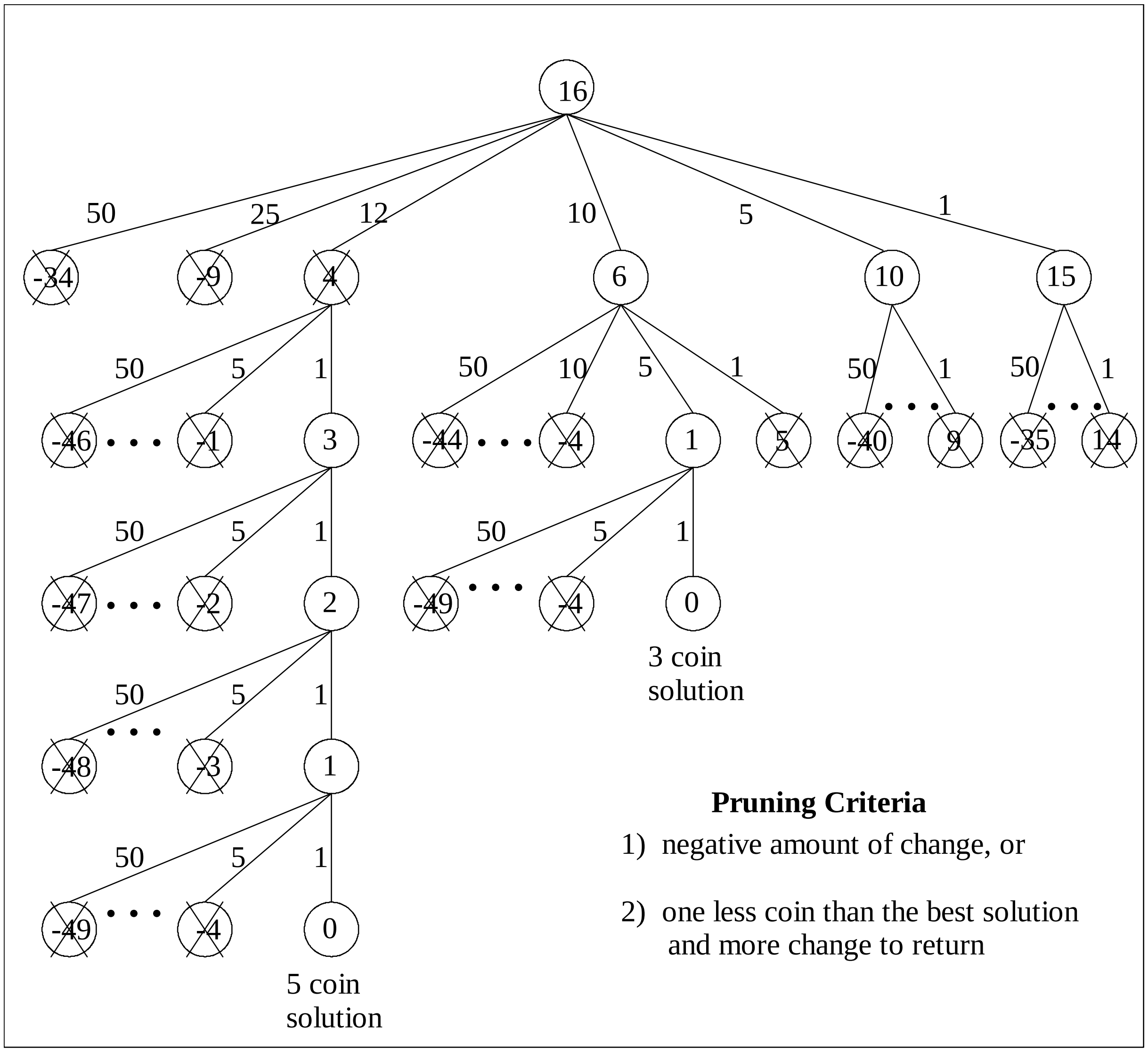


Figure 7. State-space tree for 16-cents change with coin types of {1, 5, 10, 12, 25, 50}.

Figure 7 shows the pruned state-space tree for 16-cents change with coin types of {1, 5, 10, 12, 25, 50}.

The biggest problem with backtracking is that it searches the state-space tree in a depth-first search pattern which might be slow. What we would like to do is march straight down the branch leading to the best solution. Unfortunately, we do not know the best first choice, but we can estimate which of the first choices might lead us to the best solution. Later, if we find out that we made a mistake, then we can backtrack to a node in the tree that looks more promising. By relaxing the order in which we search the state-space tree, we hopefully search less of the tree and reach an answer quicker. If the estimate made when evaluating a nodes potential is a bound on the best solution possible for any node derived from it, then we can use these bounds when pruning the tree. This type of algorithm is know as a breadth-first search with branch-and-bound pruning.

For the coin-change problem, we want the bound calculation to be fast. Typically, the bound calculation is a greedy algorithm. The obvious greedy calculation is the algorithm previously described, i.e., continue giving back the largest possible coin. While this calculation is useful in steering the best-first search, it is not useful as a pruning bound since it does not provide a limit of the best possible solution derivable from that node. As previously shown in figure 2, this greedy algorithm calculates a six-coin solution for 41 cents with coin types {1, 5, 10, 12, 25, 50} when a four coin solution exists. One possible pruning bound would be to calculate for a node

$$\text{bound} = \left( \begin{array}{c} \text{number of coins returned} \\ \text{to reach this node} \end{array} \right) + \left\lceil \frac{\text{change remaining at this node}}{\text{value of largest coin } \leq \text{ change remaining}} \right\rceil,$$

but this bound is not very useful in steering the best-first search since most of the siblings have the same bound. However, by combining the greedy algorithm to steer the best-first search and the bound calculation to prune the state-space tree a reasonable efficient algorithm can be achieved. Figure 8 illustrates the best-first search for the coin-change problem for 41 cents with the set of coins {1, 5, 10, 12, 25, 50}.

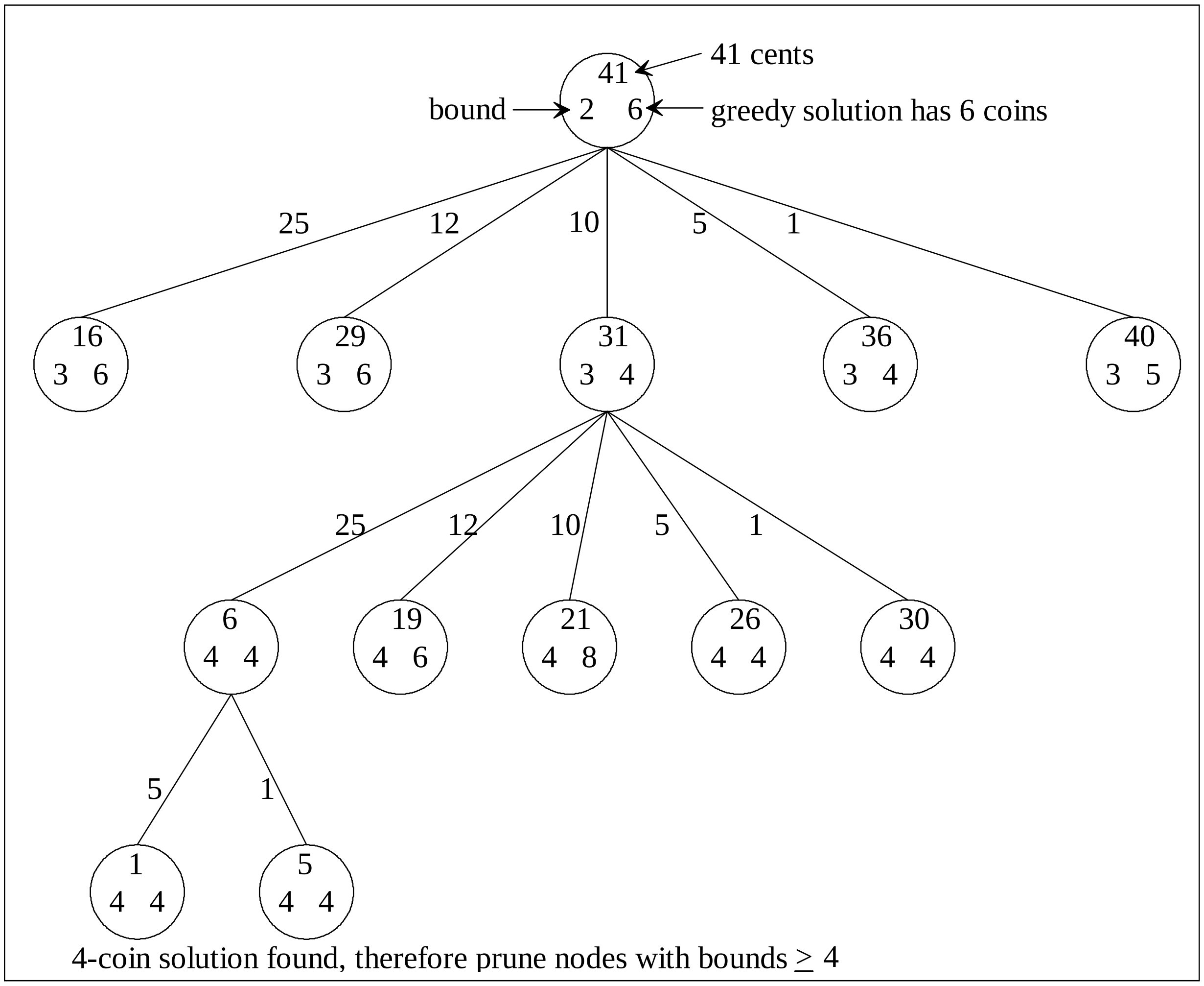


Figure 8. Coin-change problem for 41 cent using Best-First search with bound pruning.

## Selection Criteria and Other Running Examples

The main criteria for selecting a good running-example problem used to introduce all of the problem-solving technique are

1) the problem should be amenable to all problem-solving techniques taught, and
2) the problem should be easily understood by students, i.e., the problem should be as concrete as possible.

Table 1 lists several possible running-example problems with a description of each. All of these satisfy both criteria.

| Problem Name | Description |
|---|---|
| Job-Assignment problem | Given an n x n cost matrix where each row contains the cost of assigning a person to each job. Find the minimum assignment of all n people to the n jobs such that each person is assigned exactly one job and no job has two people assigned to it. |
| Traveling Salesperson Problem (TSP) | Given a weighted, directed graph G = (V, E). Find the minimize cost (tour) simple cycle that visits every vertex in the graph. |
| 0-1 Knapsack problem | Suppose that a thief breaks into a jewelry store will a knapsack with a known weight limit and scale. Find the set of jewelry items that the thief should steal in order to maximum their profit (the thief will use the marked price of each jewelry item) without exceeding the knapsacks weight limit. |

Table 1. Other Possible Running-Example Problems

## Benefits of Using Running-Example Problems

Using the same running-example problem when introducing each problem solving technique has several obvious benefits. Since students are familiar with the running-example problem (after the first problem-solving technique), the students can focus on understanding and learning the new problem-solving technique without being confused by understanding a new problem. Since the same running-example problem is being solved using all of the problem-solving techniques, students are better able to see the similarities and differences of each problem-solving technique. After seeing the problem-solving technique demonstrated with the familiar running-example problem, students are better able to apply the problem-solving technique to additional example problems demonstrating the problem-solving technique being learned.

## References

[1] Brassard, G. and Bratley, P., (1996). Fundamentals of Algorithms, Prentice Hall, Inc.
[2] Cormen, T., Leiserson, C., and Rivest, R., (1989). Introduction to Algorithms, MIT Press.
[3] Neapolitan, R. and Naimipour, K., (1998). Foundations of Algorithms: with C++ Pseudocode, second edition, Jones and Bartlett Publishers, Inc.
[4] Parberry, I., (1995). Problems on Algorithms, Prentice Hall, Inc.