ALGORITHMS

Choosing between Exact and Approximate Problem Solving

- One of the most important decision is to choose between solving the problem exactly or solving it approximately.
- In the former case, an algorithm is called an exact algorithm; in the latter case, an algorithm is called an approximation algorithm.
- Why would one opt for an approximation algorithm?
- First, there are important problems that simply cannot be solved exactly for most of their instances; examples include extracting square roots, solving nonlinear equations, and evaluating definite integrals.
- Second, available algorithms for solving a problem exactly can be unacceptably slow because of the problem's intrinsic complexity.
- This happens, in particular, for many problems involving a very large number of choices.

Algorithm Design Techniques

- An algorithm design technique (or “strategy” or “paradigm”) is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

Designing an Algorithm and Data Structures

- While the algorithm design techniques do provide a powerful set of general approaches to algorithmic problem solving, designing an algorithm for a particular problem may still be a challenging task.
- Some design techniques can be simply inapplicable to the problem in question.
- Sometimes, several techniques need to be combined, and there are algorithms that are hard to pinpoint as applications of the known design techniques.
- Even when a particular design technique is applicable, getting an algorithm often requires a nontrivial ingenuity on the part of the algorithm designer.
- With practice, both tasks—choosing among the general techniques and applying them—get easier, but they are rarely easy.

Methods of Specifying an Algorithm

- Once you have designed an algorithm, you need to specify it in some fashion.
- Sometimes an algorithm is described in words (in a free and also a step-by-step form) or in pseudocode.
- Using a natural language has an obvious appeal; however, the inherent ambiguity of any natural language makes a succinct and clear description of algorithms surprisingly difficult.
- Nevertheless, being able to do this is an important skill that you should strive to develop in the process of learning algorithms.

Methods of Specifying an Algorithm

- Pseudocode is a mixture of a natural language and programming language-like constructs.
- Pseudocode is usually more precise than natural language, and its usage often yields more succinct algorithm descriptions.
- Surprisingly, computer scientists have never agreed on a single form of pseudocode, leaving textbook authors with a need to design their own “dialects.”
- Fortunately, these dialects are so close to each other that anyone familiar with a modern programming language should be able to understand them all.

Methods of Specifying an Algorithm

- For the sake of simplicity, someone can omit declarations of variables and use indentation to show the scope of such statements as for, if, and while.

- An arrow “←” can be used for the assignment operation and two slashes “//” for comments.

Methods of Specifying an Algorithm

- In the earlier days of computing, the dominant vehicle for specifying algorithms was a flowchart, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm’s steps.

- This representation technique has proved to be inconvenient for all but very simple algorithms; nowadays, it is not used usually.

Proving an Algorithm’s Correctness

- Once an algorithm has been specified, you have to prove its correctness.
- That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.
- For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex.
- A common technique for proving correctness is to use mathematical induction because an algorithm’s iterations provide a natural sequence of steps needed for such proofs.
- It might be worth mentioning that although tracing the algorithm’s performance for a few specific inputs can be a very worthwhile activity, it cannot prove the algorithm’s correctness conclusively.
- But in order to show that an algorithm is incorrect, you need just one instance of its input for which the algorithm fails.

Proving an Algorithm's Correctness

- The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms.

- For an approximation algorithm, we usually would like to be able to show that the error produced by the algorithm does not exceed a predefined limit.

Analyzing an Algorithm

- We usually want our algorithms to possess several qualities.
- After correctness, by far the most important is efficiency.
- In fact, there are two kinds of algorithm efficiency: time efficiency, indicating how fast the algorithm runs, and space efficiency, indicating how much extra memory it uses.
- Another desirable characteristic of an algorithm is simplicity.
- Unlike efficiency, which can be precisely defined and investigated with mathematical rigor, simplicity, like beauty, is to a considerable degree in the eye of the beholder.

Analyzing an Algorithm

- For example, most people would agree that Euclid’s algorithm is simpler than the middle-school procedure for computing gcd(m, n), but it is not clear whether Euclid’s algorithm is simpler than the consecutive integer checking algorithm.
- Still, simplicity is an important algorithm characteristic to strive for. Why?
- Because simpler algorithms are easier to understand and easier to program; consequently, the resulting programs usually contain fewer bugs.
- There is also the undeniable aesthetic appeal of simplicity.
- Sometimes simpler algorithms are also more efficient than more complicated alternatives.
- Unfortunately, it is not always true, in which case a judicious compromise needs to be made.

Analyzing an Algorithm

- Yet another desirable characteristic of an algorithm is generality.
- There are, in fact, two issues here: generality of the problem the algorithm solves and the set of inputs it accepts.
- On the first issue, note that it is sometimes easier to design an algorithm for a problem posed in more general terms.
- Consider, for example, the problem of determining whether two integers are relatively prime, i.e., whether their only common divisor is equal to 1.
- It is easier to design an algorithm for a more general problem of computing the greatest common divisor of two integers and, to solve the former problem, check whether the gcd is 1 or not.
- There are situations, however, where designing a more general algorithm is unnecessary or difficult or even impossible.
- For example, it is unnecessary to sort a list of n numbers to find its median, which is its th smallest element. $\lceil n/2 \rceil$
- To give another example, the standard formula for roots of a quadratic equation cannot be generalized to handle polynomials of arbitrary degrees.

Analyzing an Algorithm

- As to the set of inputs, your main concern should be designing an algorithm that can handle a set of inputs that is natural for the problem at hand.
- For example, excluding integers equal to 1 as possible inputs for a greatest common divisor algorithm would be quite unnatural.
- On the other hand, although the standard formula for the roots of a quadratic equation holds for complex coefficients, we would normally not implement it on this level of generality unless this capability is explicitly required.

Analyzing an Algorithm

- If you are not satisfied with the algorithm's efficiency, simplicity, or generality, you must return to the drawing board and redesign the algorithm.
- In fact, even if your evaluation is positive, it is still worth searching for other algorithmic solutions.
- Recall the three different algorithms in the previous section for computing the greatest common divisor: generally, you should not expect to get the best algorithm on the first try.
- At the very least, you should try to fine-tune the algorithm you already have.

Coding an Algorithm

- Most algorithms are destined to be ultimately implemented as computer programs.
- Programming an algorithm presents both a peril and an opportunity.
- The peril lies in the possibility of making the transition from an algorithm to a program either incorrectly or very inefficiently.
- Some influential computer scientists strongly believe that unless the correctness of a computer program is proven with full mathematical rigor, the program cannot be considered correct.
- They have developed special techniques for doing such proofs but the power of these techniques of formal verification is limited so far to very small programs.

Coding an Algorithm

- As a practical matter, the validity of programs is still established by testing.
- Testing of computer programs is an art rather than a science, but that does not mean that there is nothing in it to learn.
- Of course, implementing an algorithm correctly is necessary but not sufficient: you would not like to diminish your algorithm's power by an inefficient implementation.
- Modern compilers do provide a certain safety net in this regard, especially when they are used in their code optimization mode.
- Still, you need to be aware of such standard tricks as computing a loop's invariant (an expression that does not change its value) outside the loop, collecting common subexpressions, replacing expensive operations by cheap ones, and so on.

Coding an Algorithm

- In the academic world, the question leads to an interesting but usually difficult investigation of an algorithm’s optimality.
- Actually, this question is not about the efficiency of an algorithm but about the complexity of the problem it solves:
- What is the minimum amount of effort any algorithm will need to exert to solve the problem?
- For some problems, the answer to this question is known.
- For example, any algorithm that sorts an array by comparing values of its elements needs about n log2 n comparisons for some arrays of size n.
- But for many seemingly easy problems such as integer multiplication, computer scientists do not yet have a final answer.

Coding an Algorithm

- Another important issue of algorithmic problem solving is the question of whether or not every problem can be solved by an algorithm.
- We are not talking here about problems that do not have a solution, such as finding real roots of a quadratic equation with a negative discriminant.
- For such cases, an output indicating that the problem does not have a solution is all we can and should expect from an algorithm.
- Nor are we talking about ambiguously stated problems. Even some unambiguous problems that must have a simple yes or no answer are “undecidable,” i.e., unsolvable by any algorithm.
- Fortunately, a vast majority of problems in practical computing can be solved by an algorithm.