**Họ và tên:** Phan Khắc Bão Nghĩa

**Lớp:** CNTT - K61

**MSSV:** 6151071076

**1.2 Fundamentals of Algorithmic Problem Solving**

**Understanding the Problem**

From a practical perspective, the first thing you need to do before designing an algorithm is to fully understand the problem. Read the description of the problem carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think of special cases and ask again if necessary .

If the problem in question is one of them, you can use a known algorithm to solve it. Of course, it helps to understand what an algorithm is like and to know its strengths and weaknesses, especially if you have to choose between available algorithms.

But often you won't find the algorithm available and will have to design your own. Keep in mind that an exact algorithm is not one that works most of the time, but it does work for all legitimate inputs.

Don't skimp on the first step of the algorithmic problem-solving process; otherwise, you run the risk of making unnecessary adjustments.

**Ascertaining the Capabilities of the Computational Device**

Understand the problem

Decide on:

computational means,exact vs. approximate solving,algorithm design technique

Design an algorithm

Prove correctness

Analyzing the algorithm

Code the algorithm

Once you fully understand a problem, you need to determine the capabilities of the computing device the algorithm is for. Most algorithms in use today are still destined to be programmed for a computer that closely resembles the von Neumann machine, a computer architecture drafted by the famous Hungarian-American mathematician John von Neumann. (1903–1957), collaborating with A. Burks and H. Goldstine, in 1946.

The essence of this architecture was captured by the so-called random access machine (RAM). Its central assumption is that instructions are executed one after the other, one operation at a time. Mathematical algorithms designed to execute on such machines are called sequential algorithms.

**Choosing between Exact and Approximate Problem Solving**

The next major decision is to choose between solving the problem correctly or solving it roughly. Why would one choose an approximate algorithm? First, there are important problems that cannot be solved precisely for most of their cases. Second, the algorithms available to exactly solve a problem can be unacceptably slow due to the intrinsic complexity of the problem. Third, an approximation algorithm can be part of a more complex algorithm that solves exactly one problem.

**Algorithm Design Techniques**

What is algorithm design?

Algorithm design (or “strategy” or “model”) is a general approach to solving problems algorithmically that can be applied to many problems from different fields of computing.

Check out this book's table of contents and you'll find that most of the book's chapters are devoted to individual design techniques. They distill a few key ideas that have proven useful in the design of algorithms. Learning these techniques is extremely important for the following reasons.

First, they provide guidelines for designing algorithms for new problems, that is, problems for which no satisfactory algorithm is known. It is not true, of course, that each of these general techniques will necessarily apply to every problem you may encounter. But taken together, they form a powerful collection of tools that you will find quite handy in your studies and work.

Second, algorithms are the foundation of computer science. Every science is concerned with classifying its main object, and computer science is no exception. Algorithmic design techniques make it possible to classify algorithms according to a basic design idea; therefore, they can serve as a natural way to both classify and study algorithms.

**Designing an Algorithm and Data Structures**

Although algorithm design techniques provide a powerful set of general approaches to algorithmic problem solving, designing an algorithm for a particular problem can still be a task. Challenging. With practice, it becomes easier both to choose among general techniques and to apply them, but they are rarely easy.

Of course, one should pay attention to the selection of the appropriate data structure for the operations performed by the algorithm. Years ago, an influential textbook declared the fundamental importance of both algorithms and data structures to computer programming by its very title: Algorithms + Data Structures = Program [Wir76]. In the new world of object-oriented programming, data structures are still extremely important to both algorithm design and analysis.

**Methods of Specifying an Algorithm**

Once you've designed an algorithm, you need to specify it in some way. In Section 1.1, to give you an example, Euclid's algorithm is described verbally (in both free and step-by-step form) and in pseudocode. These are the two most widely used options today for defining algorithms.

Using a natural language has a clear appeal; however, the inherent ambiguity of any natural language makes it surprisingly difficult to describe algorithms succinctly and clearly. However, being able to do this is an important skill that you should strive to develop as you learn algorithms.

Pseudocode is a mixture of natural language and programming language like constructs. Pseudocode is often more precise than natural language, and its use often yields more concise algorithmic descriptions.

In the early days of computing, the dominant means of specifying algorithms was a flowchart, a method of representing an algorithm by a set of connected geometric shapes containing descriptions of steps. of the algorithm. This representation technique has proven to be inconvenient for all algorithms, except very simple ones; nowadays, it can only be found in old algorithm books.

Computer engineering has not yet reached the point where describing algorithms in natural language or pseudocode can be directly fed into electronic computers. Instead, it needs to be converted into a computer program written in a specific computer language. We can think of such a program as another way of specifying an algorithm, although it is better to think of it as an implementation of the algorithm.

**Proving an Algorithm’s Correctness**

Once an algorithm has been specified, you must prove its correctness. That is, you have to prove that the algorithm yields the required result for every legal input in a finite amount of time.

For some algorithms, a proof of correctness is quite easy; For others, it can be quite complicated. A common technique for proving correctness is to use mathematical induction because the iterations of the algorithm provide the natural sequence of steps required for such proofs.

The concept of correctness for approximation algorithms is less straightforward than for exact algorithms. For approximation algorithms, we often want to be able to show that the error generated by the algorithm does not exceed a predefined limit.

**Analyzing an Algorithm**

We often want our algorithms to possess some quality. After correctness, by far the most important thing is efficiency. In fact, there are two types of algorithmic efficiency: time efficiency, which indicates how fast the algorithm runs, and space efficiency, which indicates how much extra memory it uses.

Another desirable property of an algorithm is simplicity. Unlike efficiency, which can be precisely defined and studied with mathematical rigor, simplicity is, like beauty, to a considerable extent in the eye of the beholder. Sometimes simpler algorithms are also more efficient than more complex alternatives. Unfortunately, it is not always the case, in which case it is necessary to make a wise compromise.

However, another desirable property of an algorithm is generality. There are actually two problems here: the generality of the problem the algorithm solves and the set of inputs it accepts. Regarding the first problem, note that it is sometimes easier to design an algorithm for a problem posed in more general terms.

For a set of inputs, your main concern should be to design an algorithm that can handle a natural set of inputs for the current problem.

If you are not satisfied with the efficiency, simplicity, or generality of your algorithm, you must go back to the drawing board and redesign the algorithm. In fact, even if your review is positive, it's still worth looking for other algorithmic solutions. You will do well to keep in mind the following remark by Antoine de Saint-Exupery, French writer, pilot, and aircraft designer: “A designer knows he has reached perfection not when he has reached perfection. There is no longer anything to add, but when there is nothing left there is nothing to take away.”

**Coding an Algorithm**

Most algorithms are ultimately implemented as computer programs. Programming an algorithm presents both risk and opportunity. The danger lies in the possibility of making the conversion from an algorithm to a program incorrect or very inefficient.

As a matter of fact, the validity of programs is still established by testing. Testing computer programs is more of an art than a science, but that doesn't mean there's nothing in it to learn.

Of course, implementing an algorithm correctly is necessary but not enough: you don't want to reduce your algorithm's power by inefficient implementation. Modern compilers provide a certain safety net in this respect, especially when they are used in their code optimization mode. Usually, such improvements can only speed up the program by a constant factor, while a better algorithm can make a difference in running time of orders of magnitude. But once an algorithm is chosen, a 10–50% speedup can be worth the effort.

In fact, this is good news as it makes the end result a lot more interesting. (Yes, I thought about naming this book The Joy of Algorithms.) On the other hand, how does one know when to stop? In the real world, it's often not the project schedule or the boss's impatience that will stop you. And so it should be: perfection is expensive and in fact not always required. Designing an algorithm is an engineering-like activity that requires a compromise between competing goals under the constraints of available resources, with the designer's time being one of the resources.