International Journal of Computer Science
Scholarly Peer Reviewed Research Journal - PRESS - OPEN ACCESS
ISSN: 2348-6600

http://www.ijcsjournal.com Volume 5, Issue 1, No 24, 2017 ISSN: 2348-6600
Reference ID: IJCS-272 PAGE NO: 1758-1767

Alagappa University, Karaikudi, India 15th -16th February 2017
IT Skills Show & International Conference on Advancements in Computing Resources (SSICACR-2017)
http://aisdau.in/ssicacr ssicacr2017@gmail.com

Exploring Key Aspects of Computational Thinking

M. Janaki
Assistant Professor, Department of Computer Science,
Dr. Umayal Ramanathan College for Women, Karaikudi.

Abstract

Computational Thinking is a collection of diverse skills to do with problem solving which results from the nature of computation. It involves specific problem solving skills such as the ability to think logically, algorithmically and recursively. Computational thinking is a fundamental skill for everyone, not only for computer scientists. We should add computational thinking to every child's analytical ability. Computational thinking involves solving problems, designing systems, and understanding human behaviour, by drawing on the concepts fundamental to computer science. This paper describes the four key aspects of computational thinking that includes decomposition, pattern recognition, abstraction and algorithms in detail. It talks about why algorithms created through computational thinking need to be evaluated. It also describes a technique called dry run which is used to evaluate solutions before programming. This term has been much discussed amongst educationalists all over the world to grips with a new computing curriculum designed to equip students with such skills, and to reduce the skills gap between education and the workplace.

Index Terms: Problem Solving, Analytical Ability, Abstraction, Decomposition, Pattern Recognition, Algorithms, Dry Run.

1. Introduction

Computational thinking is a basic skill for everyone, not only for computer scientists. I is essential to add computational thinking to every child's analytical ability. Computational thinking involves solving problems, designing systems, and understanding human behaviour, by drawing on the fundamental concepts to computer science. Computational thinking includes a range of mental tools that reflect the breadth of the field of computer science [4]. To solve a problem, two important things are way to find the best solution and to know the complexity of finding solution. Computer science rests on solid theoretical underpinnings to answer such things precisely. We must consider the machine's instruction set, its resource constraints, and its operating environment. Computer science is the study of computation—what is computed and how is computed. Computational thinking thus has the following behaviour: Viewpoint Thinking like a computer scientist means more than being able to program a computer. It requires thinking at multiple levels of abstraction. Computational thinking is a way

International Journal of Computer Science
Scholarly Peer Reviewed Research Journal - PRESS - OPEN ACCESS
ISSN: 2348-6600

http://www.ijcsjournal.com Volume 5, Issue 1, No 24, 2017 ISSN: 2348-6600
Reference ID: IJCS-272 PAGE NO: 1758-1767

Alagappa University, Karaikudi, India 15th -16th February 2017
IT Skills Show & International Conference on Advancements in Computing Resources (SSICACR-2017)
http://aisdau.in/ssicacr ssicacr2017@gmail.com

humans solve problems; it is not trying to get humans to think like computers. Computers are dull and boring; humans are clever and imaginative. We humans make computers exciting. Equipped with computing devices, we use our intelligence to tackle problems we would not dare take on before the age of computing and build systems with functionality limited only by our imaginations; Complements and combines mathematical and engineering thinking. Computational thinking will be a reality when it is so integral to human endeavours, it disappears as an explicit philosophy. Many people equate computer science with computer programming. Some parents see only a narrow range of job opportunities for their children who major in computer science. Many people think the fundamental research in computer science is done and that only the engineering remains. Computational thinking is a grand vision to guide computer science educators, researchers, and practitioners as we act to change society's image of the field.

2. Literature Review

Jeffy Krameris said "I believe that abstraction is a key skill for computing". It is essential during requirements engineering to elicit the critical aspects of the environment and required system while neglecting the unimportant. At design time, we must articulate the software architecture and component functionalities that satisfy functional and non-functional requirements while avoiding unnecessary implementation constraints. Even at the implementation stage we use data abstraction and classes so as to generalize solutions [1].

Alan Bundy described in his paper as Computational thinking is influencing research in nearly all disciplines, both in the sciences and the humanities. Researchers are using computational metaphors to enrich theories as diverse as protoeomics and the mind-body problem. Computing has enabled researchers to ask new kinds of questions and to accept new kinds of answers, for instance, questions that require the processing of huge amounts of data [2].

Vincent Conitzer has summarized in his article as a number of applications where computer scientists have already become involved in the design of markets and other protocols for making decisions based on the preferences of multiple agents. I anticipate that the number and importance of such applications will grow steeply in the years to come. One major reason for this is that computer scientists and economists interested in market design have grown closer together in recent years, and are now seen working together more often (this is necessitated by high value applications such as sponsored search auctions). Computer scientists have caught up on many of the key techniques developed in the microeconomics theory literature. On the other side, economists are becoming increasingly familiar with techniques from modern computer science. This is a very nice example where "computational thinking" is being exported to another discipline (which is certainly not to say that there were no prior instances of economists thinking computationally) [3].

3. Characteristics of Computational Thinking

Computational thinking is thinking in terms of prevention, protection, and recovery from worst-case scenarios through redundancy, damage containment, and error correction. Computational thinking is using heuristic reasoning to find a solution. It is planning, learning, and scheduling in the presence of uncertainty. It has the following characteristics,

- Computational thinking is thinking recursively.
- It is parallel processing.
- It is interpreting code as data and data as code.
- It is type checking as the generalization of dimensional analysis.
- It is recognizing both the cost and power of indirect addressing and procedure call.
- It is separation of concerns.
- It is choosing an appropriate representation for a problem.

Thinking computationally is not programming. It is not even thinking like a computer. Computational thinking enables you to work out exactly what to tell the computer to do. For example, if you want to go on a tour somewhere you have never been before, you would probably plan your route before you step out of your house. You might consider the routes available and which route is 'best' - this might be the route that is the shortest, the quickest. You'd then follow the step-by-step directions to get there [7]. In this case, the planning part is like computational thinking, and following the directions is like programming. Being able to turn a complex problem into one we can easily understand is a skill that is extremely useful. In fact, it's a skill you already have and probably use every day. It might be that you need to decide what to do with your family. If all of you like different things, you would need to decide:

- What you could do?
- Where you could go?
- Who wants to do what?
- What you have previously done that has been a success in the past?
- How much money you have and the cost of any of the options?
- What the weather might be doing?
- How much time you have?

From this information, you and your family could decide more easily where to go and what to do – in order to keep most of your family members happy. You could also use a computer to help you to collect and analyse the data to devise the best solution to the problem, both now and if it arose again in the future, if you wished.

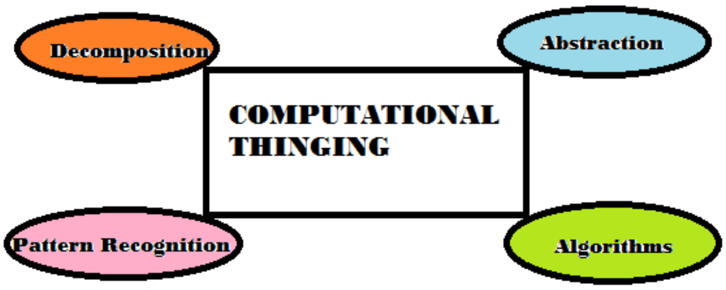
4. Key Techniques of Computational Thinking

Computers can be used to help us solve problems. Before solving a problem, the problem itself and the ways in which it could be solved need to be understood [5]. Computational thinking helps us to do this. Computational thinking allows us to take a complex problem, understand what the problem is and develop possible solutions. We can then present these solutions in a way that a computer, a human, or both, can understand.

There are four key techniques (cornerstones) to computational thinking:

1. **decomposition** - breaking down a complex problem into smaller, more manageable parts
2. **pattern recognition** – looking for similarities within problems
3. **abstraction** – focusing on the important information by ignoring irrelevant detail
4. **algorithms** - developing a step-by-step solution to the problem

Each cornerstone is as important as the others. They are like legs of a chair - if one leg is missing, the chair will probably collapse. Correctly applying all four techniques will help us to find the correct solution for a problem when programming a computer.



A complex problem is one that, at first glance, we don't know how to solve easily. Computational thinking involves following steps to find the best solution.

Step 1: Taking that complex problem and breaking it down into a series of small, more manageable problems (**decomposition**).

Step 2: Each of these smaller problems can then be looked at individually, considering how similar problems have been solved previously (**pattern recognition**).

Step 3: Focusing only on the important details, while ignoring irrelevant information (**abstraction**).

Step 4: Simple steps or rules to solve each of the smaller problems can be designed (**algorithms**).
Step 5: These simple steps or rules are used to **program** a computer to help solve the complex problem in the best way.

I. 4.1 Decomposition

Decomposition is the technique which **involves breaking down a complex problem or system into smaller parts that are more manageable and easier to understand**. The smaller parts can then be examined and solved, or designed individually, as they are simpler to work with [8]. If a problem is not decomposed, it is much harder to solve. Breaking the problem down into smaller parts means that each smaller problem can be examined in more detail. Similarly, trying to understand how a complex system works is easier

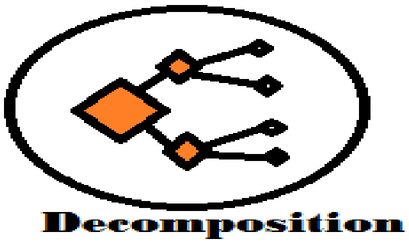


Figure 2: Decomposition

To decompose the problem of how to comb our hair, we would need to consider the following questions,

- Which comb to use?
- How long to comb for?
- How hard to press on our hair?
- What oil to use?

II. 4.2 PATTERN RECOGNITION
III.

When we decompose a complex problem, we often find patterns among the smaller problems we create. The patterns are similarities that some of the problems share [6]. Pattern recognition i**nvolves finding the similarities or patterns among small, decomposed problems that can help us solve more complex problems more efficiently**. Imagine that we want to draw a series of dogs. All

using decomposition. For example, understanding how a bicycle works is more straightforward if the whole cycle is separated into smaller parts and each part is examined to see how it works in more detail.

dogs share common characteristics. Among other things they all have eyes, tail and nails. They also like to eat meat and make barking sounds. As we know that all dogs have eyes, tail and nails, we can make a good attempt at drawing a dog, simply by including these common characteristics. In computational thinking, these characteristics are known as patterns. Once we know how to describe one dog we can describe others, simply by following this pattern. The only things that are different are the specifics:

- one dog may have green eyes, a long tail and 20 nails
- another dog may have yellow eyes, a short tail and 18 nails
- other dog may have brown eyes, a medium tail and 21 nails

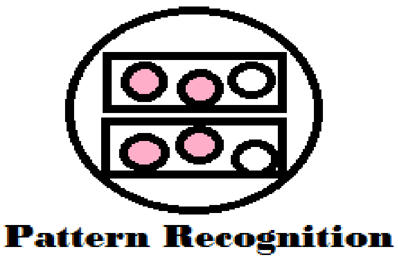


Figure 3: Pattern Recognition

Finding patterns is extremely important. Patterns make our task simpler. Problems are easier to solve when they share patterns, because we can use the same problem-solving solution wherever the pattern exists. The more patterns we can find, the easier and quicker our overall task of problem solving will be. If we want to draw a number of dogs, finding a pattern to describe dogs in general, example: they all have eyes, tail and nails, makes this task quicker and easier. We know that all dogs follow this pattern, so we don't have to stop each time we start to draw a new dog to work this out. From the patterns we know dogs follow, we can quickly draw several dogs. Suppose we hadn't looked for patterns in dogs. Each time we wanted to draw a dog, we would have to stop and work out what a dog looked like. This would slow us down. We could still draw our dogs - and they would look like dogs - but each dog would take far longer to draw. This would be very inefficient, and a poor way to go about solving the dog-drawing task.

4.3 Abstraction

Abstraction allows us to create a general idea of what the problem is and how to solve it. The process instructs us to remove all specific detail that will not help us solve our problem. This helps us form our idea of the problem. This idea is known as a 'model'. If we don't abstract we may end up with the wrong solution to the problem we are trying to solve [9]. With our dog example, if we didn't abstract we might think that all dogs have long tails and 20 nails. Having abstracted, we know that although dogs have tail and nails, not all tails are long and not all have 20 nails. In this case, abstraction has helped us to form a clearer model of a dog.

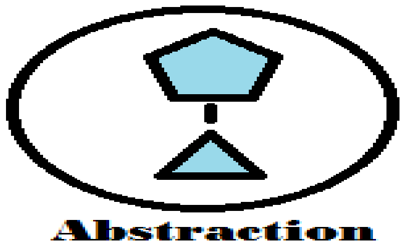


Figure 4: Abstraction

Abstraction is the gathering of the general characteristics we need and the filtering out of the details and characteristics that we do not need. When baking a biscuit, there are some general characteristics between biscuits. For example:

- a cake needs ingredients
- each ingredient needs a specified quantity
- a cake needs timings

When abstracting, we remove specific details and keep the general relevant patterns.

General Patterns	Specific Details
We need to know that a biscuit has ingredients to make	We don't need to know what those ingredients are
We need to know that each ingredient has a specified quantity	We don't need to know what that quantity is
We need to know that each biscuit needs a specified time to bake	We don't need to know how long the time is

Table 1: General Patterns and Specific Details of Cake Baking

I. 4.4 Algorithm

An algorithm is a plan, a set of step-by-step instructions to solve a problem. In an algorithm, each instruction is identified and the order in which they should be carried out is planned [10]. Algorithms are often used as a starting point for creating a computer program, and they are sometimes written as a flowchart or in pseudocode. If we want to tell a computer to do something, we have to write a computer program that will tell the computer, step-by-step, exactly what we want it to do and how we want it to do it. This step-by-step program will need planning, and to do this we use an algorithm. Computers are only as good as the algorithms they are given. If you give a computer a poor algorithm, you will get a poor result – hence the phrase: 'Garbage in, garbage out.' Algorithms are used for many different things including calculations, data processing and automation.

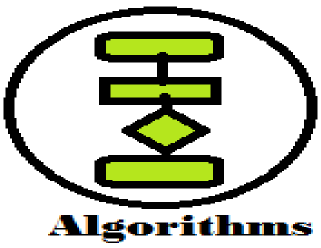


Figure 5: Algorithms

This order can be represented as an algorithm. An algorithm must be clear. It must have a starting point, a finishing point and a set of clear instructions in between.

4.4.1 Representing an algorithm: Pseudocode

There are two main ways that algorithms can be represented – pseudocode and flowcharts. Most programs are developed using programming languages. These languages have specific syntax that must be used so that the program will run properly. Pseudocode is not a programming language, it is a simple way of describing a set of instructions that does not have to use specific syntax. Writing in pseudocode is similar to writing in a programming language. Each step of the algorithm is written on a line of its own in sequence. Usually, instructions are written in uppercase, variables in lowercase and messages in sentence case.

In pseudocode, INPUT asks a question. OUTPUT prints a message on screen. A simple program could be created to ask someone their name and age, and to make a comment based on these. This program represented in pseudocode would look like this:

```
OUTPUT 'What is your name?'
INPUT user inputs their name
OUTPUT 'How old are you?'
INPUT user inputs their age
STORE the user's input in the age variable
IF age >= 18 THEN
    OUTPUT 'You are eligible to vote!'
ELSE
    OUTPUT 'You are not eligible to vote!'
```

4.4.2 Representing an algorithm: Flowcharts

A flowchart is a diagram that represents a set of instructions. Flowcharts normally use standard symbols to represent the different instructions. There are few real rules about the level of detail needed in a flowchart. Sometimes flowcharts are broken down into many steps to provide a lot of detail about exactly what is happening. Sometimes they are simplified so that a number of steps occur in just one step.

A simple program could be created to ask someone their name and age, and to make a comment based on these. This program represented as a flowchart would look like this:

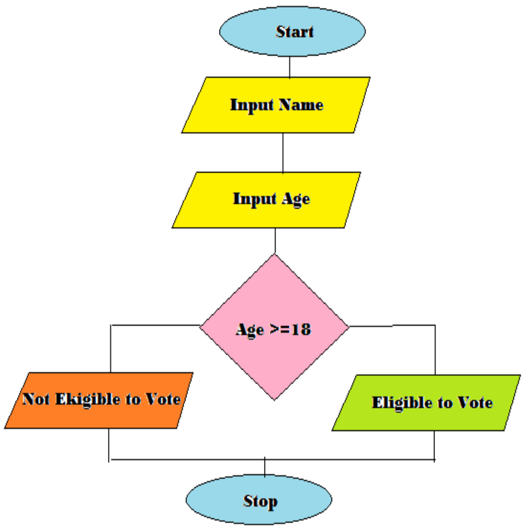


Figure 6: Flow chart for checking eligibility for voting

5. Evaluation of the Solution

Once a solution has been designed using computational thinking, it is important to make sure that the solution is fit for purpose. Evaluation is the process that allows us to make sure our solution does the job it has been designed to do and to think about how it could be improved.

International Journal of Computer Science
Scholarly Peer Reviewed Research Journal - PRESS - OPEN ACCESS
ISSN: 2348-6600

http://www.ijcsjournal.com Volume 5, Issue 1, No 24, 2017 ISSN: 2348-6600
Reference ID: IJCS-272 PAGE NO: 1758-1767

Alagappa University, Karaikudi, India 15th -16th February 2017
IT Skills Show & International Conference on Advancements in Computing Resources (SSICACR-2017)
http://aisdau.in/ssicacr ssicacr2017@gmail.com

Once written, an algorithm should be checked to make sure it:

- is easily understood – is it fully decomposed?
- is complete – does it solve every aspect of the problem? is efficient – does it solve the problem, making best use of the available resources.
- meets any design criteria we have been given

If an algorithm meets these four criteria it is likely to work well. The algorithm can then be programmed. Failure to evaluate can make it difficult to write a program. Evaluation helps to make sure that as few difficulties as possible are faced when programming the solution. We may find that solutions fail because:

- it is **not fully understood** - we may not have properly decomposed the problem
- it is **incomplete** - some parts of the problem may have been left out accidentally
- it is **inefficient** – it may be too complicated or too long
- it **does not meet the original design criteria** – so it is not fit for purpose

There are several ways to evaluate solutions. To be certain that the solution is correct, it is important to ask:

- does the solution make sense?

Do you now fully understand how to solve the problem? If you still don't clearly know how to do something to solve our problem, go back and make sure everything has been properly decomposed. Once you know how to do everything, then our problem is thoroughly decomposed.

- does the solution cover all parts of the problem?

For example, if drawing a dog, does the solution describe everything needed to draw a dog, not just eyes, a tail and nails? If not, go back and keeping adding steps to the solution until it is complete.

- does the solution ask for tasks to be repeated?

If so, is there a way to reduce repetition? Go back and remove unnecessary repetition until the solution is efficient.

5.1 Dry run Technique

One of the best ways to test a solution is to perform what's known as a 'dry run'. With pen and paper, work through the algorithm and trace a path through it. For example, in Algorithms, a simple algorithm was created to ask someone their name and age, and to make a comment based on these. You could try out this algorithm – give it a dry run. Try two ages, 15 and 25. When using age 15, where does the algorithm go? Does it give the right output? If you use age 25, does it take you down a different path? Does it still give the correct output? If the dry run doesn't give the right answer, there is something wrong that needs fixing. Recording the path through the algorithm will help show where the error occurs. Dry runs are also used with completed programs. Programmers use dry runs to help find errors in their program code.

6. Conclusion

Computational thinking is a fundamental skill for everyone, not only for computer scientists. We should add computational thinking to every child's analytical ability. This paper have described the four key aspects of computational thinking that includes decomposition, pattern recognition, abstraction and algorithms in detail along with appropriate examples. It also justified why algorithms created through computational thinking need to be evaluated. It described the dry run technique which is used to evaluate solutions before programming. Computational thinking term has been much discussed amongst educationalists all over the world to grips with a new computing curriculum designed to equip students with such skills, and to reduce the skills gap between education and the workplace. It is essential to keep computational thinking as the technique to be taught in the school level which will act like the brain boosters which improves the analytical ability of the kids.

References

[1] Jeffy Krameris, " Abstraction The Key To Computing?", Communications of the ACM, April 2007/Vol. 50, No. 4.
[2] Alan Bundy, " Computational Thinking Is Pervasive", Journal Of Scientific and Practical Computing, Vol. 1, No. 2 (2007) 67–69.
[3] Vincent Conitzer, "Making Decisions Based on the Preferences of Multiple Agents", communications of the ACM, march 2010, vol. 53, no. 3. doi:10.1145/1666420.1666442.
[4] James J. Lu, George H. L. Fletcher, " Thinking About Computational Thinking", SIGCSE'09, March 3–7, 2009, Chattanooga, Tennessee, USA. Copyright 2009 ACM 978-1-60558-183-5/09/03.
[5] H. Abelson and G. J. Sussman. Structure and Interpretation of Computer Programs, 2nd ed. MIT Press, Cambridge, 1996.
[6] J. L. Bates and R. L. Constable. Proofs as programs. ACM Trans. Program. Lang. Syst. 7(1):113-136, 1985.
[7] L. Carter. Why students with an apparent aptitude for computer science don't choose to major in computer science. SIGCSE 2006, Houston, pp. 27-31.
[8] A. Cohen and B. Haberman. Computer science: a language of technology. SIGCSE inroads 39(4):65-69, 2007.
[9] M. Guzdial. Paving the way for computational thinking. CACM 51(8):25-27, 2008. [8] S. Reges. The mystery of "b := (b = false)." SIGCSE 2008, Portland, pp. 21-25.
[10] J. M. Wing. Computational thinking. CACM 49(3):33-35, 2006.