**자료구조론 CC343\_2207**

**Reading assignment 2**

**경기대학교 컴퓨터공학부**

**201511837 이상민**

**Review Questions**

1. Explain the features of a good program

좋은 프로그램의 특징을 설명합니다.

* **Portability**: Portability refers to the ability of an application to run on different platforms (operating systems) with or without minimal changes. Due to rapid development in the hardware and the software, nowadays platform change is a common phenomenon. Hence, if a program is developed for a particular platform, then the life span of the program is severely affected.
* **Readability:** The program should be written in such a way that it makes other programmers or users to follow the logic of the program without much effort. If a program is written structurally, it helps the programmers to understand their own program in a better way. Even if some computational efficiency needs to be sacrificed for better readability, it is advisable to use a more user-friendly approach, unless the processing of an application is of utmost importance.
* **Efficiency:** Every program requires certain processing time and memory to process the instructions and data. As the processing power and memory are the most precious resources of a computer, a program should be laid out in such a manner that it utilizes the least amount of memory and processing time.
* **Structural:** To develop a program, the task must be broken down into a number of subtasks. These subtasks are developed independently, and each subtask is able to perform the assigned job without the help of any other subtask. If a program is developed structurally, it becomes more readable, and the testing and documentation process also gets easier.
* **Flexibility:** A program should be flexible enough to handle most of the changes without having to rewrite the entire program. Most of the programs are developed for a certain period and they require modifications from time to time. For example, in case of payroll management, as the time progresses, some employees may leave the company while some others may join. Hence, the payroll application should be flexible enough to incorporate all the changes without having to reconstruct the entire application.
* **Generality:** Apart from flexibility, the program should also be general. Generality means that if a program is developed for a particular task, then it should also be used for all similar tasks of the same domain. For example, if a program is developed for a particular organization, then it should suit all the other similar organizations.
* **Documentation:** Documentation is one of the most important components of an application development. Even if a program is developed following the best programming practices, it will be rendered useless if the end user is not able to fully utilize the functionality of the application. A well-documented application is also useful for other programmers because even in the absence of the author, they can understand it.

1. Define the terms: data, file, record, and primary key.

데이터, 파일, 레코드 및 기본 키라는 용어를 정의합니다.

**Data :** Data is set of variables which are used for describing specific topic. It can be meaningless sometimes, which can be translated to efficient manner. Data can be numbers, symbols, figures, characters and many others.

**File :** File is a collection of folders that contain different types of data. It can be any format like document or excel or power point. It has information in different formats like table, text, pictures and many such.

**Record :** Record is a set of fields which has information of at least one variable or item. It describes clearly about the item using fields. For example, a File record has fields which describe its terms like file name, file size and many such.

Primary key : A primary key is a special relational database table column (or combination of columns) designated to uniquely identify all table records.

1. Define data structures. Give some examples.

데이터 구조를 정의합니다. 몇 가지 예를 들어보세요.

**Characteristics of data structures**

Data structures are often classified by their characteristics. Possible characteristics are:

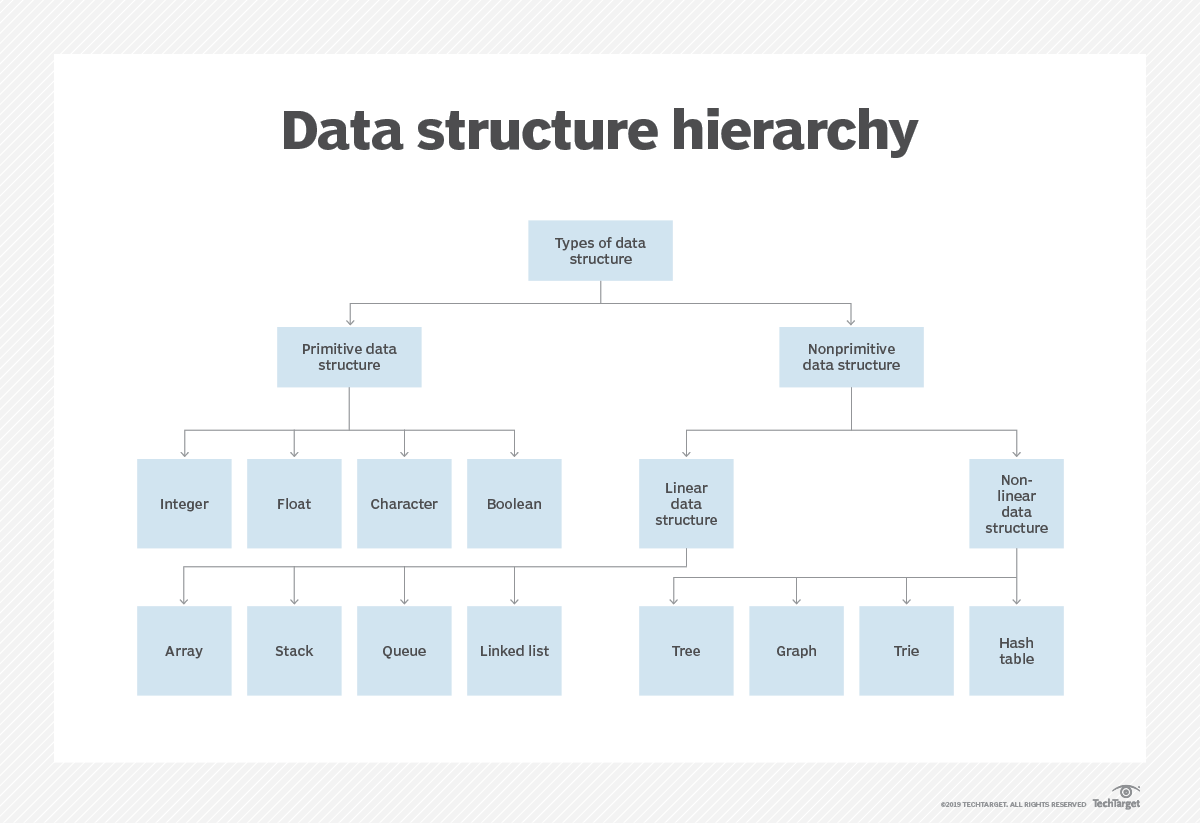
* Linear or non-linear: This characteristic describes whether the data items are arranged in chronological sequence, such as with an array, or in an unordered sequence, such as with a graph.
* Homogeneous or non-homogeneous: This characteristic describes whether all data items in a given repository are of the same type or of various types.
* Static or dynamic: This characteristic describes how the data structures are compiled. Static data structures have fixed sizes, structures and memory locations at compile time. Dynamic data structures have sizes, structures and memory locations that can shrink or expand depending on the use.

**Types of data structures**

Data structure types are determined by what types of operations are required or what kinds of algorithms are going to be applied. These types include:

* Arrays- An array stores a collection of items at adjoining memory locations. Items that are the same type get stored together so that the position of each element can be calculated or retrieved easily. Arrays can be fixed or flexible in length.
  + A stack stores a collection of items in the linear order that operations are applied. This order could be last in first out (LIFO) or first in first out ().
  + A queue stores a collection of items similar to a stack; however, the operation order can only be first in first out.
* Linked lists- A linked list stores a collection of items in a linear order. Each element, or node, in a linked list contains a data item as well as a reference, or link, to the next item in the list.
* Trees- A tree stores a collection of items in an abstract, hierarchical way. Each node is linked to other nodes and can have multiple sub-values, also known as children.
* Graphs- A graph stores a collection of items in a non-linear fashion. Graphs are made up of a finite set of nodes, also known as vertices, and lines that connect them, also known as edges. These are useful for representing real-life systems such as computer networks.
* Tries- A trie, or keyword tree, is a data structure that stores strings as data items that can be organized in a visual graph.
* Hash tables- A hash table, or a hash map, stores a collection of items in an associative array that plots keys to values. A hash table uses a hash function to convert an index into an array of buckets that contain the desired data item.

These are considered complex data structures as they can store large amounts of interconnected data. Examples of primitive, or basic, data structures are , floats, and characters.



1. In how many ways can you categorize data structures? Explain each of them.

데이터 구조를 몇 가지 방법으로 분류할 수 있습니까? 하나하나 설명해 주세요.

* **Primitive and Non Primitive Data Structure:** The data structure that are atomic (indivisible) are called primitive. Examples are integer, real and characters. The Data structures that are not atomic are called non-primitive or composite. Examples are records, array and string.
* **Linear and Non-Linear Data Structures:** In a linear data structure, the data items are arranged in a linear sequence. For Example: array. In a non-linear data structure, the data items that are not in sequence. For Example: trees and graphs.
* **Homogeneous and Non-Homogeneous Data Structures:** In homogeneous data structure, all the elements are of same type. For Example: arrays. In non-homogeneous data structure, the elements may or may not be of the same type. For Example: Records.
* **Static and Dynamic Data Structures:** Static data structures are those whose size and structures, associated location is fixed at compile time.Dynamic structures are ones whose ones which expand or shrink as required during the program execution and there associate memory location change.

1. Discuss the applications of data structures.

데이터 구조의 애플리케이션에 대해 논의합니다.

**Applications of Data Structure**

**Arrays:** An array stores a collection of items at adjoining memory locations. Items that are the same type get stored together so that the position of each element can be calculated or retrieved easily. Arrays can be fixed or flexible in length.

**Stacks:** A stack stores a collection of items in the linear order that operations are applied. This order could be last in first out (LIFO) or first in first out (FIFO).

**Queues:** A queue stores a collection of items similar to a stack; however, the operation order can only be first in first out.

**Linked lists:** A linked list stores a collection of items in a linear order. Each element, or node, in a linked list, contains a data item as well as a reference, or link, to the next item in the list.

**Trees:** A tree stores a collection of items in an abstract, hierarchical way. Each node is linked to other nodes and can have multiple sub-values, also known as children.

**Graphs:** A graph stores a collection of items in a non-linear fashion. Graphs are made up of a finite set of nodes, also known as vertices, and lines that connect them, also known as edges. These are useful for representing real-life systems such as computer networks.

**Tries:** A trie, or keyword tree, is a data structure that stores strings as data items that can be organized in a visual graph.

**Hash tables:** A hash table, or a hash map, stores a collection of items in an associative array that plots keys to values. A hash table uses a hash function to convert an index into an array of buckets that contain the desired data item.

1. Write a short note on different operations that can be performed on data structures.

데이터 구조에서 수행할 수 있는 다양한 작업에 대해 짧은 메모를 작성합니다.

1. **Create :** The create operation results in reserving memory for program elements. This can be done by declaration statement. Creation of data structure may take place either during compile-time or run-time. malloc() function of C language is used for creation.
2. **Destroy :** Destroy operation destroys memory space allocated for specified data structure. free() function of C language is used to destroy data structure.
3. **Selection** : Selection operation deals with accessing a particular data within a data structure.
4. **Updation :** It updates or modifies the data in the data structure.
5. **Searching :** It finds the presence of desired data item in the list of data items, it may also find the locations of all elements that satisfy certain conditions.
6. **Sorting :** Sorting is a process of arranging all data items in a data structure in a particular order, say for example, either in ascending order or in descending order.
7. **Merging :** Merging is a process of combining the data items of two different sorted list into a single sorted list.
8. **Splitting :** Splitting is a process of partitioning single list to multiple list.
9. **Traversal :** Traversal is a process of visiting each and every node of a list in systematic manner.
10. Compare a linked list with an array.

링크된 목록을 어레이와 비교합니다.

|  |  |
| --- | --- |
| **ARRAY** | **LINKED LIST** |
| Array is a collection of elements of similar data type. | Linked List is an ordered collection of elements of same type, which are connected to each other using pointers. |
| Array supports **Random Access**, which means elements can be accessed directly using their index, like arr[0] for 1st element, arr[6] for 7th element etc.  Hence, accessing elements in an array is **fast** with a constant time complexity of O(1). | Linked List supports **Sequential Access**, which means to access any element/node in a linked list, we have to sequentially traverse the complete linked list, upto that element.  To access **nth** element of a linked list, time complexity is O(n). |
| In an array, elements are stored in **contiguous memory location** or consecutive manner in the memory. | In a linked list, new elements can be stored anywhere in the memory.  Address of the memory location allocated to the new element is stored in the previous node of linked list, hence formaing a link between the two nodes/elements. |
| In array, **Insertion and Deletion** operation takes more time, as the memory locations are consecutive and fixed. | In case of linked list, a new element is stored at the first free and available memory location, with only a single overhead step of storing the address of memory location in the previous node of linked list.  Insertion and Deletion operations are **fast** in linked list. |
| Memory is allocated as soon as the array is declared, at **compile time**. It's also known as **Static Memory Allocation**. | Memory is allocated at **runtime**, as and when a new node is added. It's also known as **Dynamic Memory Allocation**. |
| In array, each element is independent and can be accessed using it's index value. | In case of a linked list, each node/element points to the next, previous, or maybe both nodes. |
| Array can **single dimensional**, **two dimensional** or **multidimensional** | Linked list can be **Linear(Singly)**, **Doubly** or **Circular** linked list. |
| Size of the array must be specified at time of array decalaration. | Size of a Linked list is variable. It grows at runtime, as more nodes are added to it. |
| Array gets memory allocated in the **Stack** section. | Whereas, linked list gets memory allocated in **Heap** section. |

1. Write a short note on abstract data type.

추상 데이터 유형에 대해 짧은 메모를 작성합니다.

Abstract Data Type(ADT) is a data type, where only behavior is defined but not implementation.

Opposite of ADT is Concrete Data Type (CDT), where it contains an implementation of ADT.

**Examples:**

Array, List, Map, Queue, Set, Stack, Table, Tree, and Vector are ADTs. Each of these ADTs has many implementations i.e. CDT. The container is a high-level ADT of above all ADTs.

1. Explain the different types of data structures. Also discuss their merits and demerits.

다양한 유형의 데이터 구조를 설명합니다. 또한 그들의 장단점에 대해 토론하세요.

**Data structure** : It is arrangement of various types of data in computer memory, in such a way that it is stored efficiently.

There are various types of data structure available like arrays,Linked List,Trees,graphs,stacks, queues and hash table.

**Advantages :**

1) Allows easier processing of data.

2) It allows information stored on disk very efficiently.

3) These are necessary for designing an efficient algorithm.

4) It provides management of databases like indexing with the help of hash tables and arrays.

5) We can access data anytime and anywhere.

6) It is secure way of storage of data.

7) Graphs models real life problems

8) It allows processing of data on software system

**Disadvantages :**

1) It is applicable only for advanced users.

2) If any issue occurs it can be solved only by experts.

3) Slow access in case of some data types

1. Define an algorithm. Explain its features with the help of suitable examples.

알고리즘을 정의합니다. 적절한 예를 통해 해당 기능을 설명합니다.

**Algorithm:**

Technically, the set of instruction or description in a particular notation of the process is termed as algorithm. An algorithm is a finite step-by-step well defined instructions of the sequence of the activities that constitute a process of getting the desired outputs from the given inputs. The raw material needed at the time of beginning is referred to as input and the rusting entity is referred as output.

**Characteristics of An Algorithm:**

There are five important characteristics of an algorithm that should be considered while designing any algorithm for any problem.

1. **Fineness**: An algorithm should terminate infinite number of steps and each step must finish in finite amount of time.
2. **Definiteness (No Ambiguity):** Each step of algorithm should be clearly and precisely define and there should not be any ambiguity. Example: A program fragment is given below:

x ← 1.

toss a coin, if the result is head then x← 3 else x ← 4.

In the above program, all the steps would be carried out effectively but there is no definiteness since there are two possible values of x i.e., 1 and 3/4

1. **Inputs:** An algorithm must have zero or more but must be finite number of inputs. Example of zero input algorithm. Print the ASCII code of each of the letter in the alphabet of the computer system.
2. **Output:** An algorithm must have at-least one desirable outcome, i.e., output.
3. **Effectiveness:** An algorithm should be effective. Effective means that each step should be referred as principle and should be executing in finite time. Example of Not Effectiveness: Find exact value of e using the following formula:
4. Explain and compare the approaches for designing an algorithm.

알고리즘 설계 방법을 설명하고 비교합니다.

The following is a list of several popular design approaches:

**1. Divide and Conquer Approach:** It is a top-down approach. The algorithms which follow the divide & conquer techniques involve three steps:

Divide the original problem into a set of subproblems.

Solve every subproblem individually, recursively.

Combine the solution of the subproblems (top level) into a solution of the whole original problem.

**2. Greedy Technique:** Greedy method is used to solve the optimization problem. An optimization problem is one in which we are given a set of input values, which are required either to be maximized or minimized (known as objective), i.e. some constraints or conditions.

* Greedy Algorithm always makes the choice (greedy criteria) looks best at the moment, to optimize a given objective.
* The greedy algorithm doesn't always guarantee the optimal solution however it generally produces a solution that is very close in value to the optimal.

**3. Dynamic Programming:** Dynamic Programming is a bottom-up approach we solve all possible small problems and then combine them to obtain solutions for bigger problems.

This is particularly helpful when the number of copying subproblems is exponentially large. Dynamic Programming is frequently related to Optimization Problems.

**4. Branch and Bound:** In Branch & Bound algorithm a given subproblem, which cannot be bounded, has to be divided into at least two new restricted subproblems. Branch and Bound algorithm are methods for global optimization in non-convex problems. Branch and Bound algorithms can be slow, however in the worst case they require effort that grows exponentially with problem size, but in some cases we are lucky, and the method coverage with much less effort.

**5. Randomized Algorithms:** A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation.

**6. Backtracking Algorithm:** Backtracking Algorithm tries each possibility until they find the right one. It is a depth-first search of the set of possible solution. During the search, if an alternative doesn't work, then backtrack to the choice point, the place which presented different alternatives, and tries the next alternative.

**7. Randomized Algorithm:** A randomized algorithm uses a random number at least once during the computation make a decision.

1. What is modularization? Give its advantages.

모듈화란 무엇입니까? 이점을 제공합니다.

One of the most prominent problems in software engineering has been how to program large and complex pieces of software. Often, large projects involve hundreds of programmers working on millions of lines of code. In this kind of environment, it is easy to lose track of what particular code does, or to produce code that must be rewritten elsewhere. To avoid such poor-planning scenarios, computer scientists began to organize around the concept of "modularization," which means to break up and organize code based on the task it executes. In this way, code becomes reusable and easier to debug and manage.

**Easier to Debug :**

When debugging large programs, how and when any bugs occur can become a mystery. This can take much of a programmer valuable time as he searches through lines and lines of code to find out where an error occurred, and problems it causes later in the program. If a program is designed with modularity in mind, however, then each discrete task has its own discrete section of code. So, if there is a problem in a particular function, the programmer knows where to look and can manage a smaller portion of code.

**Reusable Code :**

Modular code allows programmers to easily reuse code. If particular tasks are sectioned off to certain functions or classes, this means that the programmer can reuse that particular code whenever she needs to perform that task again. If code is not organized into discrete parts, then it is harder (or impossible) to reference, separate or implement that code in other programming contexts.

**Readability :**

Modular code is code that is highly organized. To organize code based on task means that the programmer can organize each piece of code based on what it does. Then, she can easily find or reference that code based on her organization scheme. Furthermore, other programmers working on the code can follow her organization scheme to read the code as well. This optimizes code for use among multiple developers with less trouble.

**Reliability :**

All these advantages add up to one big advantage: reliability. Code that is easier to read, easier to debug, easier to maintain and easier to share will always run smoother with less errors. This becomes necessary when working on extremely large projects, with hundreds of developers, all of which have to either share code or work on code that will have to interface with other developers' code in the future. Modularization of code is necessary to create complex software reliably.

1. Write a brief note on trees as a data structure.

데이터에 대한 구조로서 나무에 대한 간단한 메모를 작성합니다.

In , a tree is a widely used (ADT) that simulates a hierarchical , with a root value and subtrees of children with a , represented as a set of linked .

A tree data structure can be defined as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.

Alternatively, a tree can be defined abstractly as a whole (globally) as an , with a value assigned to each node. Both these perspectives are useful: while a tree can be analyzed mathematically as a whole, when actually represented as a data structure it is usually represented and worked with separately by node (rather than as a set of nodes and an of edges between nodes, as one may represent a , for instance). For example, looking at a tree as a whole, one can talk about "the parent node" of a given node, but in general as a data structure a given node only contains the list of its children, but does not contain a reference to its parent (if any).

1. What do you understand by a graph?

그래프로 무엇을 이해합니까?

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,

[A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes.]



In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

1. Explain the criteria that you will keep in mind while choosing an appropriate algorithm to solve a particular problem.

특정 문제를 해결하기 위한 적절한 알고리즘을 선택하는 동안 염두에 둘 기준을 설명합니다.

The first thing to do is look at the fundamental requirements - what are the inputs, what are the outputs, what are the operations in between?

Then, the performance and scale requirements - how much data are we talking about? How frequently will it be accessed?

Then, ponder how those requirements align with the simple classic data structures that we all know and love - lists, arrays, hash sets, dictionaries, queues, trees, heaps, and so on. Usually there's a good match; if there is only a near-match, you can usually wrap one of those simple structures to give it the API that you really need (like turning a list into a stack or queue).

Sometimes, there's no good fit with the usual structures. Then you have to consider stuff that isn't used often, but that works wonderfully for some special cases. Bloom filters are a great example. Or stuff that's "bigger" than the basic structures, like relational databases or the various NoSQL databases.

The last two steps depend on having a mental catalog of structures to consider. That's something that comes from experience. The more code you write, the more familiar you will be with various data structures, algorithms, patterns, and so on. Reading books also helps, as does reading code, getting feedback on your code, reading other peoples' feedback on other peoples' code, and so on. It all takes time. But you can pick up a lot of that stuff without even trying - just by writing software on your own, or better yet as part of a team.

If there's still no good fit that comes to mind, talk to people. Ask classmates or co-workers, find online forums, and so on. Maybe you can find someone who knows of a good fit, or maybe you can find someone who can help you picture your problem from a different perspective, and maybe that can help you think of a good way to meet your requirements.

1. What do you understand by time–space trade-off?

시간적 공간적 절충을 통해 무엇을 이해합니까?

In , a space-time or time-memory tradeoff is a way of solving a problem or calculation in less time by using more storage space (or memory), or by solving a problem in very little space by spending a long time. Most computers have a large amount of space, but not infinite space. Also, most people are willing to wait a little while for a big calculation, but not forever. So if your problem is taking a long time but not much memory, a space-time tradeoff would let you use more memory and solve the problem more quickly. Or, if it could be solved very quickly but requires more memory than you have, you can try to spend more time solving the problem in the limited memory.

The most common condition is an using a . This means that the answers for some question for every possible value can be written down. One way of solving this problem is to write down the entire lookup table, which will let you find answers very quickly, but will use a lot of space. Another way is to calculate the answers without writing down anything, which uses very little space, but might take a long time.

A space-time tradeoff can be used with the problem of . If data is stored uncompressed, it takes more space but less time than if the data were stored compressed (since compressing the data decreases the amount of space it takes, but it takes time to run the ).

Larger code size can be used to increase program speed when using . This makes the longer for each iteration of a loop, but saves the computation time needed for jumping back to the beginning of the loop at the end of each iteration.

In the field of , using space-time tradeoff, the attacker is decreasing the time required for a . use partially precomputed values in the hash space of a to in minutes instead of weeks. Decreasing the size of the rainbow table increases the time required to over the hash space. The meet-in-the-middle attack attack uses a space-time tradeoff to find the cryptographic key in only encryptions (and O() space) compared to the expected encryptions (but only O(1) space) of the normal attack.

is another example where the time of solving problems can be decreased by using more memory.

1. What do you understand by the efficiency of an algorithm?

알고리즘의 효율성으로 무엇을 이해합니까?

algorithm efficiency A measure of the average execution time necessary for an algorithm to complete work on a set of data. Algorithm efficiency is characterized by its order. Typically a bubble sort algorithm will have efficiency in sorting N items proportional to and of the order of , usually written O(). This is because an average of N/2 comparisons are required N/2 times, giving /4 total comparisons, hence of the order of . In contrast, quicksort has an efficiency O(N log2N).

If two algorithms for the same problem are of the same order then they are approximately as efficient in terms of computation. Algorithm efficiency is useful for quantifying the implementation difficulties of certain problems.

1. How will you express the time complexity of a given algorithm?

주어진 알고리즘의 시간 복잡성을 어떻게 표현합니까?

컴퓨터과학에서 알고리즘의 시간복잡도는 입력을 나타내는 문자열 길이의 함수로서 작동하는 알고리즘을 취해 시간을 정량화하는 것이다. 알고리즘의 시간복잡도는 주로 빅-오 표기법을 사용하여 나타내며, 이 빅-오 표기법은 계수와 낮은 차수의 항을 제외시키는 방법이다. 이런 방식으로 표현할 때, (예를 들면, 입력 크기를 무한대로 입력하여) 시간복잡도를 점근적으로 묘사한다고 말한다.

예시로서, 만약 크기 n의 모든 입력에 대한 알고리즘에 필요한 시간이 최대 (어떤 n0보다 크지 않은 모든 n에 대하여) 5n3 + 3n의 식을 가진다면, 이 알고리즘의 점근적 시간 복잡도는 O(n3)이라고 할 수 있다.

시간 복잡도는 기본적인 연산을 수행하는데에 어떤 고정된 시간이 걸릴 때, 알고리즘에 의해서 수행되는 기본 연산의 개수를 세어 예측할 수 있다. 그러므로 걸리는 시간의 총량과 알고리즘에 의해 수행되는 기본적인 연산의 개수는 최대 상수 인자만큼 다르다.

알고리즘의 수행 시간은 동일 크기의 다양한 입력에 의해 달라질 수 있기 때문에, 가장 많이 쓰이는 최악의 시간 복잡도의 알고리즘 시간을 T(n)이라고 했을 때, 이것은 크기 n의 모든 입력에 대해 걸리는 최대의 시간으로 정의할 수 있다.

그 다음으로 덜 흔하게 쓰이면서, 보통 명확하게 서술되는 측정방법은 평균 시간 복잡도이다.

시간 복잡도는 함수 T(n)의 특성에 의해 분류할 수 있다. 예를 들면, T(n)=O(n)인 알고리즘은 선형 시간 알고리즘이라고 부르며, 몇몇 M ≥ n >1에 대해 T(n)=O(Mn)이고 Mn=O(T(n)) 인 알고리즘은 지수 시간 알고리즘이라고 한다.

1. Discuss the significance and limitations of the Big O notation.

Big O 표기법의 중요성과 한계를 논의합니다.

Big O notation is a mathematical notation that describes the of a when the tends towards a particular value or infinity. It is a member of a family of notations invented by , , and others, collectively called Bachmann–Landau notation or asymptotic notation.

In , big O notation is used to according to how their running time or space requirements grow as the input size grows. In , big O notation is often used to express a bound on the difference between an and a better understood approximation; a famous example of such a difference is the remainder term in the .

There are certain limitations with the Big O notation of expressing the complexity of algorithms. These limitations are as follows:

* 1. Many algorithms are simply too hard to analyse mathematically.
  2. There may not be sufficient information to calculate the behaviour of the algorithm in the average case.
  3. Big O analysis only tells us how the algorithm grows with the size of the problem, not how efficient it is, as it does not consider the programming effort.
  4. It ignores important constants. For example, if one algorithm takes O(n2 ) time to execute and the other takes O(100000n2 ) time to execute, then as per Big O, both algorithm have equal time complexity. In real-time systems, this may be a serious consideration.

1. Discuss the best case, worst case, average case, and amortized time complexity of an algorithm.

알고리즘의 최적 사례, 최악의 사례, 평균 사례 및 상각 시간 복잡성에 대해 토론합니다.

최선의 경우 성능(best-case performance)은 전산학에서 최적조건 하에 알고리즘의 동작을 묘사하는데 사용된다. 예를 들어, 리스트(list)의 간단한 선형 탐색 문제에서 최선의 경우는 원하는 원소가 리스트의 처음에 위치하는 경우이다.

알고리즘 선택 및 개발에 있어 최선의 경우 성능은 거의 활용되지 않으며, 대부분 평균 경우 복잡도(average-case complexity)와 최악의 경우 성능(worst-case performance)의 향상에 관심이 있다. 유한 입력 집합의 하드-코딩(hard-coding) 해결법으로 좋은 최선의 경우 실행 시간을 가지도록 알고리즘을 수정한다.

최악의 경우(worst-case) 성능 분석과 평균 경우 성능 분석은 서로 유사성을 가지지만, 두 가지 분석을 위해서 실제로 다른 도구와 접근법이 요구된다.

평균 입력이 무엇인지 정의하는 것은 어렵고, 흔히 평균 입력은 수학적으로 특징 짓기 어려운 특성을 가진다. 예를 들어 문자열 연산을 위한 알고리즘 역시 평균 입력을 정의하기 어렵다. 심지어 특정 “평균 경우(average case)”의 합리적인 기술이 가능하더라도, 이를 분석하기 위하여 매우 어려운 수식들을 도출하는 경향이 있다.

Worst-case 분석은 유사한 문제를 가진다: 일반적으로 정확한 worst-case 시나리오를 결정하는 게 불가능하다. 대신에, 최소 worst case 보다 나쁜 시나리오를 고려한다. 예를 들어, 알고리즘 분석시, 심지어 경로를 생성하는 정확한 입력을 결정하는 것이 불가능하더라도 (실재로 그러한 입력이 존재하지 않더라도), 가장 긴 가능한 경로를 찾는 것이 가능하다. (예를 들어 최대 루프 숫자를 고려하기) 이는 안전한 분석을 도출하나(worst case는 결코 과소평가 될 수 없다), 이 경로를 요구하는 입력이 없을 수 있기 때문에, 이는 비관적이다.

최악의 경우 분석 역시도 비슷한 문제점을 가진다. 일반적으로 최악의 경우 시나리오를 정밀하게 정의하는 것은 불가능하다. 대신에, 최악의 경우 못지않게 안 좋은 시나리오를 고려한다. 예를 들어 경로 생성 알고리즘 분석시, 경로를 생성하기 위한 정확한 입력을 결정하는 것이 불가능하더라도 (실제로 그러한 입력이 존재하지 않더라도) 가장 긴 가능한 경로를 찾는 것은 가능하다(예를 들어 최대 루프 횟수). 최악의 경우 분석은 과소추정(underestimate) 하지 않기 때문에 안정한 분석을 도출한다. 그러나 해당 경로를 요구하는 입력이 없을 수 있기 때문에 매우 비관적인 분석이 될 수 있다.

그 대신에 실제 최악의 경우와 매우 가깝다고 추정되는 시나리오를 고려 할 수 있다(그러나 더 나쁜 경우는 아니다). 이는 낙관적인 결과를 도출 할 수 있으며, 이 분석은 실제 최악의 경우를 과소추정 할 수 있다는 의미를 지닌다.

어떠한 상황에서는 안정성 보장을 위하여 비관적 분석이 필요할 수 있다. 그러나 비관적 분석은 자주 매우 비관적으로 변할 수 있으며, 낙관적으로 실제 값과 근접하게 분석하는 것이 훨씬 더 실질적인 접근법이 될 수 있다.

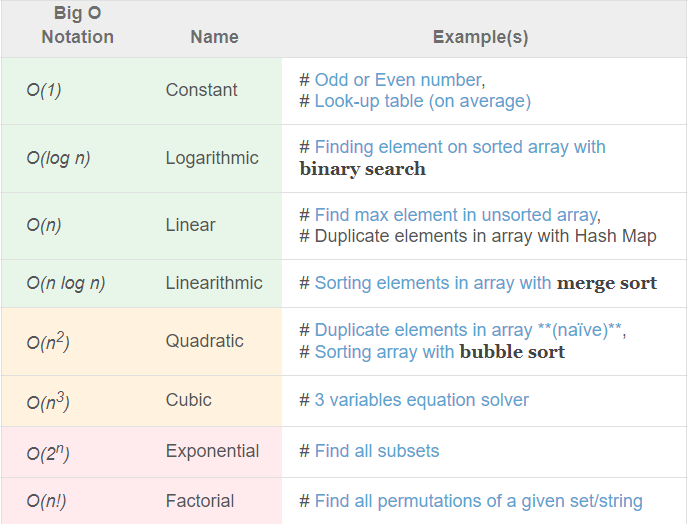
자주 짧은 수행시간이 걸리나 주기적으로 훨씬 긴 시간을 요구하는 알고리즘들을 분석 할 때, 분할상환분석(amortized analysis)은 연속적인 연산들의 최악의 경우 실행 시간을 결정하는데 활용 될 수 있다.

이 분할상환 최악의 경우(amortized worst-case) 비용은 매우 평균 경우 비용(average case cost)와 근접함과 동시에 실행 시간의 상향치(upper limit)를 보장한다.

이 최악의 경우 분석은 최악의 경우 복잡도(worst-case complexity)와 연관된다.

1. Categorize algorithms based on their running time complexity.

실행 시간 복잡도를 기준으로 알고리즘을 분류합니다



1. Give examples of functions that are in Big O notation as well as functions that are not in Big O notation.

Big O 표기법에 없는 기능 뿐 만 아니라 Big O 표기법에도 있는 기능의 예를 제시합니다.

22번 문제의 답변과 일치합니다.

1. Explain the little o notation.

작은 O 표기법을 설명해 주세요.

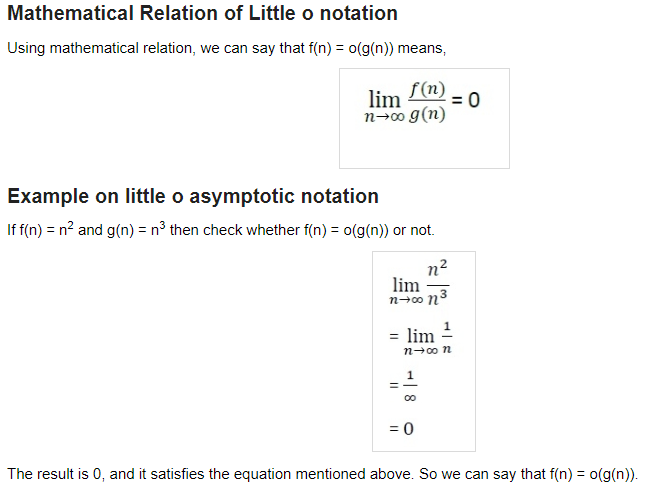
There are some other notations present except the Big-Oh, Big-Omega and Big-Theta notations. The little o notation is one of them.

Little o notation is used to describe an upper bound that cannot be tight. In other words, loose upper bound of f(n).

Let f(n) and g(n) are the functions that map positive real numbers. We can say that the function f(n) is o(g(n)) if for any real positive constant c, there exists an integer constant n0 ≤ 1 such that f(n) > 0.

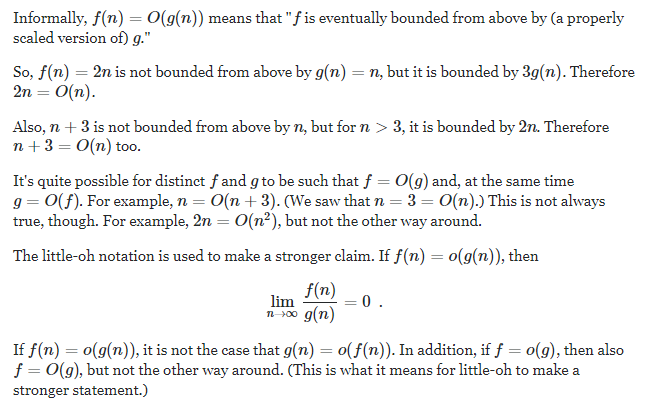
1. Give examples of functions that are in little o notation as well as functions that are not in little o notation.

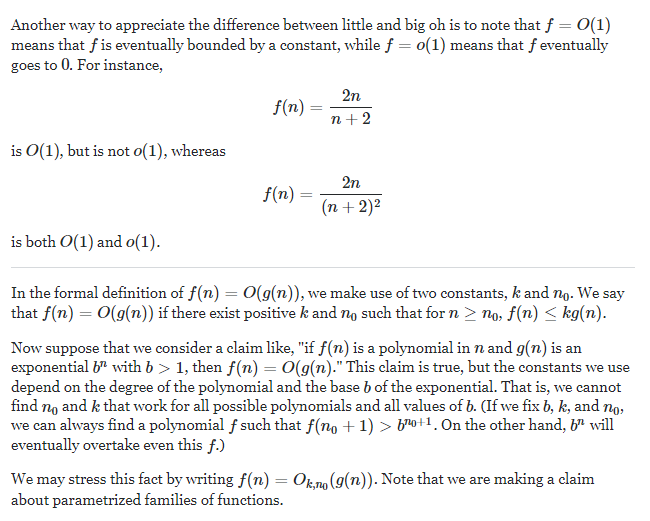
o표기법에 없는 기능 뿐 만 아니라 o표기법에도 없는 기능의 예를 제시합니다.



1. Differentiate between Big O and little o notations.

Big O와 little o 표기법을 구분합니다.





1. Explain the Ω notation.

Ω 표기법을 설명합니다

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.



For example, for a function f(n)

Ω(f(n)) ≥ { g(n) : there exists c > 0 and n0 such that g(n) ≤ c.f(n) for all n > n0. }

1. Give examples of functions that are in Ω notation as well as functions that are not in Ω notation.

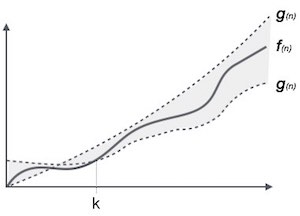
Ω 표기법에 없는 함수와 Ω 표기법에 없는 함수의 예를 제시합니다.

We can also make correct, but imprecise, statements using big-Ω notation. For example, if you really do have a million dollars in your pocket, you can truthfully say "I have an amount of money in my pocket, and it's at least 10 dollars." That is correct, but certainly not very precise. Similarly, we can correctly but imprecisely say that the worst-case running time of binary search is \Omega(1)Ω(1)\Omega, left parenthesis, 1, right parenthesis, because we know that it takes at least constant time.

1. Explain the Θ notation.

Θ 표기법을 설명합니다.

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows –



θ(f(n)) = { g(n) if and only if g(n) = Ο(f(n)) and g(n) = Ω(f(n)) for all n > n0. }

1. Give examples of functions that are in Θ notation as well as functions that are not in Θ notation.

Θ 표기법에 없는 함수와 as Θ 표기법에 없는 함수의 예를 제시합니다.

보통 상수 인자와 낮은 차원의 항목은 생략하고 사용합니다. big-Θ 표기법을 사용하는 또다른 이점은 시간 단위를 고려할 필요가 없다는 것입니다. 예를 들어 실행시간이 6n^2 + 100n + 300us라고 가정해 봅시다. 아니면 ms일 수도 있을겁니다. big-Θ 표기법에서는 이를 언급하지 않습니다. 또한 계수인 6과 저차원 항목인 100n + 300100n+300100, n, plus, 300을 생략하고 그냥 실행 시간이 Θ(n^2) 라고 할 수 있습니다.

big-Θ표기법을 사용하는 것은 실행 시간에 대해 점근적으로 근접한 한계값이 있다고 표현하는 것입니다. "점근적으로"라는 말을 쓰는 이유는 큰 값의 n에 대해서만 적용되기 때문입니다. "근접한 한계값"이라는 말은 위, 아래로 상수값 내에서 실행 시간을 좁힐 수 있다는 뜻입니다.

1. Explain the ω notation.

ω 표기법을 설명합니다.

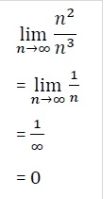
Another asymptotic notation is little omega notation. it is denoted by (ω).

1. Give examples of functions that are in ω notation as well as functions that are in ω notation

ω 표기법에 있는 함수와 ω 표기법에 있는 함수의 예를 제시합니다.

Little omega (ω) notation is used to describe a loose lower bound of f(n).

If f(n) = n2 and g(n) = n3 then check whether f(n) = o(g(n)) or not.



The result is 0, and it satisfies the equation mentioned above. So we can say that f(n) = o(g(n)).

1. Differentiate between Big omega and little omega notations.

큰 오메가 표기법과 작은 오메가 표기법을 구분합니다.

f(n) has a higher growth rate than g(n) so main difference between Big Omega (Ω) and little omega (ω) lies in their definitions.In the case of Big Omega f(n)=Ω(g(n)) and the bound is 0<=cg(n)<=f(n), but in case of little omega, it is true for 0<=c\*g(n)<f(n).

1. Show that  + 50n = O(  ).

In your case, the terms are n2, 2n, and 3, and the dominant term is n2.

So, you want to find a, b, and c such that n2≤an2, 2n≤n2, and 3≤cn3 for all n>n0 for some n0.

Obviously a=1 works for all n.

If 2n≤bn2, then 2≤bn, and this is true for b=1 for n≥2.

Note that big-oh (and little-oh) notation means the bound holds for all large enough n, so you do not need to worry about some initial values for which the bound is false.

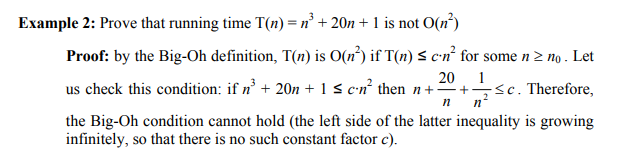
Finally, 3≤cn2 is true for c=3 for all n.

Therefore, for n≥2, n2+2n+2≤n2+1n2+3n2=5n2, so n2+2n+2=O(n2).

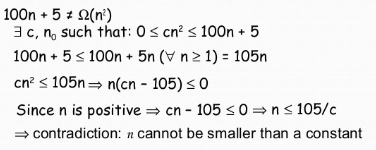
1. Show that  +  +  = 3  = O(  ).



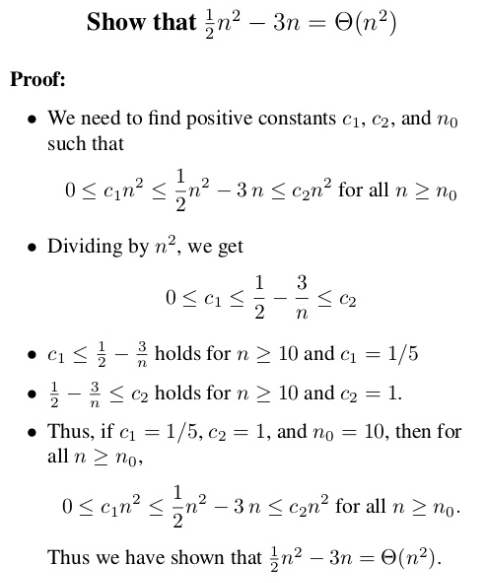
1. Prove that   ≠ O( ).



1. Show that  = Ω(lg n ).
2. Prove that 3n + 5 ≠ Ω(  ).



1. Show that ½ – 3 n ∈ Θ( ).



**Multiple-choice** **Questions**

**1.** Which data structure is defined as a collection of similar data elements?

**(a) Arrays** (b) Linked lists (c) Trees (d) Graphs

**2.** The data structure used in hierarchical data model is

(a) Array (b) Linked list **(c) Tree** (d) Graph

**3.** In a stack, insertion is done at

**(a) Top** (b) Front (c) Rear (d) Mid

**4.** The position in a queue from which an element is deleted is called as

(a) Top **(b) Front** (c) Rear (d) Mid

**5.** Which data structure has fixed size?

**(a) Arrays** (b) Linked lists (c) Trees (d) Graphs

**6.** If TOP = MAX–1 , then that the stack is

(a) Empty **(b) Full** (c) Contains some data (d) None of these

**7.** Which among the following is a LIFO data structure?

**(a) Stacks** (b) Linked lists (c) Queues (d) Graphs

**8.** Which data structure is used to represent complex relationships between the nodes?

(a) Arrays (b) Linked lists (c) Trees **(d) Graphs**

**9.** Examples of linear data structures include

(a) Arrays (b) Stacks (c) Queues **(d) All of these**

**10.** The running time complexity of a linear time algorithm is given as

(a) O(1) **(b) O(n)** (c) O(n log n) (d) O()

**11.** Which notation provides a strict upper bound for f(n)?

(a) Omega notation **(b) Big** O notation (c) Small o notation (d) Theta Notation

**12.** Which notation comprises a set of all functions h(n) that are greater than or equal to cg(n) for all values of n ≥  ?

**(a) Omega notation** (b) Big O notation (c) Small o notation (d) Theta Notation

**13.** Function in o () notation is

(a) 10 **(b)**  (c) /100 (d)

**True** **or** **False**

**1.** Trees and graphs are the examples of linear data structures. : False

**2.** Queue is a FIFO data structure. : True

**3.** Trees can represent any kind of complex relationship between the nodes. : False

**4.** The average-case running time of an algorithm is an upper bound on the running time for any input. : False

**5.** Array is an abstract data type. : False

**6.** Array elements are stored in continuous memory locations. : True

**7.** The pop operation adds an element to the top of a stack. : False

**8.** Graphs have a purely parent-to-child relationship between their nodes. : False

**9.** The worst-case running time of an algorithm is a lower bound on the running time for any input. : False

**10.** In top-down approach, we start with designing the most basic or concrete modules and then proceed towards designing higher-level modules. : False

**11.** o(g(n)) comprises a set of all functions h(n) that are less than or equal to cg(n) for all values of n ≥ . : False

**12.** Simply Ω means same as best case Ω. : True

**13.** Small omega notation provides an asymptotically tight bound for f(n). : False

**14.** Theta notation provides a non-asymptotically tight lower bound for f(n). : False

**15.**   ≠ ω(  ). : False

**Fill** **in** **the** **Blanks**

**1.** \_\_\_\_\_\_ is an arrangement of data either in the computer ’s memory or on the disk storage.

**Answer :** Data structures

**2.** \_\_\_\_\_\_ are used to manipulate the data contained in various data structures.

**Answer :** Functions

**3.** In \_\_\_\_\_\_, the elements of a data structure are stored sequentially.

**Answer :** Arrays

**4.** \_\_\_\_\_\_ of a variable specifies the set of values that the variable can take.

**Answer :** Data type

**5.** A tree is empty if \_\_\_\_\_\_.

**Answer :** Root = NULL

**6.** Abstract means \_\_\_\_\_\_.

**Answer :** Considered apart from the detailed specifications or implementation

**7.** The time complexity of an algorithm is the running time given as a function of \_\_\_\_\_\_.

**Answer :** Input size

**8.** \_\_\_\_\_\_ analysis guarantees the average perfor-mance of each operation in the worst case.

**Answer :** Amortized case

**9.** The elements of an array are referenced by an \_\_\_\_\_\_.

**Answer :** Index or subscript

**10.** \_\_\_\_\_\_ is used to store the address of the topmost element of a stack.

**Answer :** Top

**11.** The \_\_\_\_\_\_ operation returns the value of the topmost element of a stack.

**Answer : Peep or peek**

**12.** An overflow occurs when \_\_\_\_\_\_.

**Answer : An attempt is made to insert an element in an array, stack or queue that is already full.**

**13.** \_\_\_\_\_\_ is a FIFO data structure.

**Answer : Queue**

**14.** The elements in a queue are added at \_\_\_\_\_\_ and removed from \_\_\_\_\_\_.

**Answer : Rear, front**

**15.** If the elements of a data structure are stored sequentially, then it is a \_\_\_\_\_\_.

**Answer : Linear data structure**

**16.** \_\_\_\_\_\_ is basically a set of instructions that solve a problem.

**Answer : Program**

**17.** The number of machine instructions that a pro-gram executes during its execution is called its \_\_\_\_\_\_.

**Answer : Time complexity**

**18.** \_\_\_\_\_\_ specifies the expected behaviour of an algorithm when an input is randomly drawn from a given distribution.

**Answer : Average case running time**

**19.** The running time complexity of a constant time algorithm is given as \_\_\_\_\_\_.

**Answer : O(1)**

**20.** A complex algorithm is often divided into smaller units called \_\_\_\_\_\_.

**Answer : Modules**

**21.** \_\_\_\_\_ design approach starts by dividing the complex algorithm into one or more modules.

**Answer : Top down**

**22.** \_\_\_\_\_\_\_ case is when the array is sorted in reverse order.

**Answer : Worst**

**23.** \_\_\_\_\_\_\_\_ notation provides a tight lower bound for f(n).

**Answer : Omega**

**24.** The small o notation provides a \_\_\_\_\_\_\_\_\_ tight upper bound for f(n).

**Answer : Non-asymptotically**

**25.** 540  + 10 \_\_\_\_ Ω (  )

**Answer : Is in**