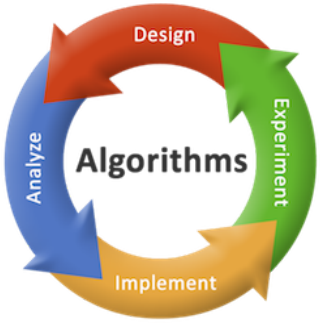
Algorithm Design 1.

Ortiz Vega Angelo
Course Code: CE2103
Name: Algorithms and Data Structures II,
Academic Area of Computer Engineering.
Cartago, Costa Rica.

-Keywords: Algorithm Design, algorithm analysis, asymptotic analysis,

-Content:

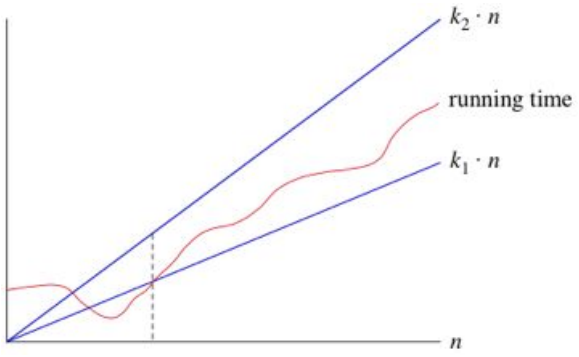
Overview:
An algorithm is a well-defined procedure (sequence of computational steps), it has an input and an output. An algorithm must have a precise and unambiguous specification.
They can be grouped according to purpose: sorting, searching, compression, etc ...
Algorithm Analysis: Allows us to measure efficiency, predict use of resources, Proposed by Alan Turing, Hardware Independent.
The Algorithm Analysis is done through an Asymptotic analysis, this allows to measure the best, average and worst case.



Asymptotic Notation:
The running time of an algorithm depends on how long it takes a computer to run the lines of code of the algorithm—and that depends on the speed of the computer, the programming language, and the compiler that translates the program from the programming language into code that runs directly on the computer, among other factors.
By dropping the less significant terms and the constant coefficients, we can focus on the important part of an algorithm's running time—its rate of growth—without getting mired in details that complicate our understanding. When we drop the constant coefficients and the less significant terms, we use **asymptotic notation**. We'll see three forms of it: big-Theta Θ notation, big-O notation, and big-Omega Ω notation.

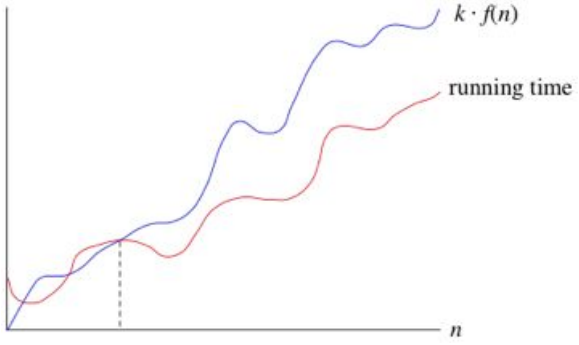
Big-θ (Big-Theta) notation:
When we say that a particular running time is Θ(*n*), we're saying that once *n* gets large enough, the running time is at least k1 * n and a most k2 * n for some

k1 and k2 constants. Here's how to think of Θ(*n*):



Big-O Notation:

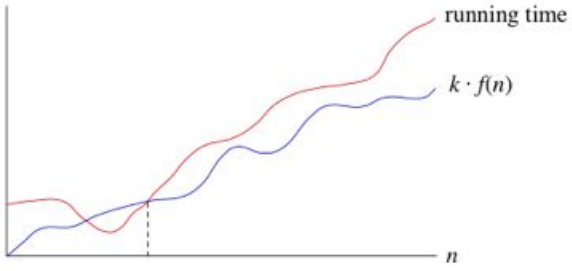
If a running time is O(f(n)), then for a large enough n, the running time is a most k * f(n) for some constant k. Here's how to think of a running time that is O(f(n)):



We use big-O notation for **asymptotic upper bounds**, since it bounds the growth of the running time from above for large enough input sizes.

Big-Ω (Big-Omega) notation:

If a running time is Ω(f(n)), then for a large enough n, the running time is at least k*f(n) for some constant . Here's how to think of a running time that is Ω(f(n))



We use big-Ω notation for **asymptotic lower bounds**, since it bounds the growth of the running time from below for large enough input sizes.

Searching Algorithms:

Hash Tables:

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:

- In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
- In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples the students and books were hashed to a unique number.

Assume that you have an object and you want to assign a key to it to make searching easy. To store the key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store values. However, in cases where the keys are large and cannot be used directly as an index, you should use *hashing*.

In hashing, large keys are converted into small keys by using hash functions. The values are then stored in a data structure called hash table. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key). By using that key you can access the element in O(1) time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Hashing is implemented in two steps:

1. An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.
2. The element is stored in the hash table where it can be quickly retrieved using hashed key.

```
hash = hashfunc(key)
index = hash % array_size
```

In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and array_size − 1) by using the modulo operator (%).

Hash function

A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

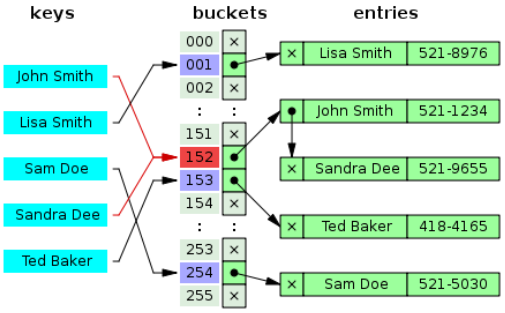
To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:

1. Easy to compute: It should be easy to compute and must not become an algorithm in itself.
2. Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.
3. Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

   Note: Irrespective of how good a hash function is, collisions are bound to occur. Therefore, to maintain the performance of a hash table, it is important to manage collisions through

various collision resolution techniques.

Example of Hash Table:



Searching in Lists:

1. Sequential / Linear Search: Acceptable for small fixes, used for disordered lists, compare key against one element after another. It is easy to understand and program.
2. Binary Search: Similar to look up a word in the dictionary, compare the central element against the key, it has O (log2n).
3. Jump Search: Divide the list into blocks of sqrt (n) elements, in each step compare the border elements. The list must be ordered.
4. Search by interpolation: For ordered lists, it is an improvement over binary search, it changes the way it calculates the central element. Try to estimate where the element is.

Pathfinding:

Pathfinding or pathing is the plotting, by a computer application, of the shortest route between two points. It is a more practical variant on solving mazes. This field of research is based heavily on Dijkstra's algorithm for finding a shortest path on a weighted graph.
https://qiao.github.io/PathFinding.js/visual/

1. **Dijkstra:**

Dijkstra's Algorithm is another algorithm used when trying to solve the problem of finding the shortest path. This algorithm specifically solves the single-source shortest path problem, where we have our start destination, and then can find the shortest path from there to every other node in the graph.

If we are looking for a specific end destination and the path there, we can keep track of parents and once we reach that end destination, backtrack through the end node's parents to reach our beginning position, giving us our path along the way.

Dijkstra's Algorithm Steps

Let's be a even a little more descriptive and lay it out step-by-step.

1. Set all the node's distances to infinity and add them to an unexplored set

2. Set the starting node's distance to 0

3. Repeat the following:

A) Look for the node with the lowest distance, let this be the current node.

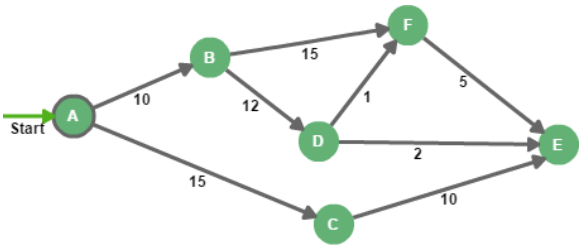
B) Remove it from the unexplored set

C) For each of the nodes adjacent to this node…

- If it is not walkable, ignore it. Otherwise do the following.
- Calculate a potential new distance based on the current node's distance plus the distance to the adjacent node you are at.
- If the potential distance is less than the adjacent node's current distance, then set the adjacent node's distance to the potential new distance and set the adjacent node's parent to the current node

D) Stop when you:

- Remove the end node from the unexplored set, in which case the path has been found, or
- Fail to find the end node, and the unexplored set is empty. In this case, there is no path.



A* Algorithm:

A* is the most popular choice for pathfinding, because it's fairly flexible and can be used in a wide range of contexts.

A* is like Dijkstra's Algorithm in that it can be used to find a shortest path. A* is like Greedy Best-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast as Greedy Best-First-Search.

The secret to its success is that it combines the pieces of information that Dijkstra's Algorithm uses (favoring vertices that are close to the starting point) *and* information that Greedy Best-First-Search uses (favoring vertices that are close to the goal). In the standard terminology used when talking about A*, g(n)represents the *exact cost* of the path from the starting point to any vertex n, and h(n)

represents the heuristic *estimated cost* from vertex n to the goal. In the above diagrams, the yellow (h) represents vertices far from the goal and teal (g) represents vertices far from the starting point. A* balances the two as it moves from the starting point to the goal. Each time through the main loop, it examines the vertex n that has the lowest f(n) = g(n) + h(n).