

10 Algorithms

10.1 Deterministic and Non-deterministic algorithm

Deterministic algorithms can be defined in terms of a state machine: a state describes what a machine is doing at a particular instant in time. State machines pass in a discrete manner from one state to another. Just after we enter the input, the machine is in its initial state or start state. If the machine is deterministic, this means that from this point onwards, its current state determines what its next state will be; its course through the set of states is predetermined. Note that a machine can be deterministic and still never stop or finish, and therefore fail to deliver a result.

A nondeterministic algorithm is one in which for a given input instance each intermediate step has one or more possibilities. This means that there may be more than one path from which the algorithm may arbitrarily choose one. Not all paths terminate successfully to give the desired output. The nondeterministic algorithm works in such a way so as to always choose a path that terminates successfully, thus always giving the correct result.

What makes algorithms non-deterministic?

- If it uses external state other than the input, such as user input, a global variable, a hardware timer value, a random value, or stored disk data.
- If it operates in a way that is timing-sensitive, for example if it has multiple processors writing to the same data at the same time. In this case, the precise order in which each processor writes its data will affect the result.
- If a hardware error causes its state to change in an unexpected way.

Procedures of a Nondeterministic Algorithm:

The nondeterministic algorithm uses three basic procedures as follows:

1. CHOICE(1,n) or CHOICE(S) : This procedure chooses and returns an arbitrary element, in favour of the algorithm, from the closed interval [1,n] or from the set S.
2. SUCCESS : This procedure declares a successful completion of the algorithm.
3. FAILURE : This procedure declares an unsuccessful termination of the algorithm.

10.2 Divide and conquer algorithm

Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related sub problems. These algorithms typically follow a divide-and-conquer approach: they break the problem into several sub problems that are similar to the original problem but smaller in size, solve the sub problems recursively, and then combine these solutions to create a solution to the original problem.

The divide-and-conquer paradigm involves three steps at each level of the recursion:

- Divide the problem into a number of sub problems.
- Conquer the sub problems by solving them recursively. If the sub problem sizes are small enough, however, just solve the sub problems in a straightforward manner.
- Combine the solutions to the sub problems into the solution for the original problem. The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.
- **Divide:** Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.
- **Conquer:** Sort the two subsequences recursively using merge sort.
- **Combine:** Merge the two-sorted subsequences to produce the sorted answer.

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. To perform the merging, we use an auxiliary procedure MERGE(A, p, q, r), where A is an array and p, q, and r are indices numbering elements of the array such that $p \leq q < r$. The procedure assumes that the subarrays $A[p \dots q]$ and $A[q + 1 \dots r]$ are in sorted order. It merges them to form a single sorted subarray that replaces the current subarray $A[p \dots r]$.

10.3 Series and Parallel Algorithm

In computer science, a parallel algorithm or concurrent algorithm, as opposed to a traditional sequential (or serial) algorithm, is an algorithm which can be executed a piece at a time on many different processing devices, and then combined together again at the end to get the correct result. Some algorithms are easy to divide up into pieces in this way. For example, splitting up the job of checking all of the numbers from one to a hundred thousand to see which are primes could be done by assigning a subset of the numbers to each available processor, and then putting the list of positive results back together. Algorithms are also used for things such as rubik's cubing and for hash decryption.

Most of the available algorithms to compute π (π), on the other hand, cannot be easily split up into parallel portions. They require the results from a preceding step to effectively carry on with the next step. Such problems are called inherently serial problems. Some problems are very difficult to parallelize, although they are recursive. One such example is the depth-first search of graphs.

Parallel algorithms are valuable because of substantial improvements in multiprocessing systems and the rise of multi-core processors. In general, it is easier to construct a computer with a single fast processor than one with many slow processors with the same throughput. But processor speed is increased primarily by shrinking the circuitry, and modern processors are pushing physical size and heat limits. These twin barriers have flipped the equation, making multiprocessing practical even for small systems.

The cost or complexity of serial algorithms is estimated in terms of the space (memory) and time (processor cycles) that they take. Parallel algorithms need to optimize one more resource, the communication between different processors. There are two ways parallel processors communicate, shared memory or message passing.

Shared memory processing needs additional locking for the data, imposes the overhead of additional processor and bus cycles, and also serializes some portion of the algorithm.

Message passing processing uses channels and message boxes but this communication adds transfer overhead on the bus, additional memory need for queues and message boxes and latency in the messages.

10.4 Heuristic and Approximate Algorithm

Many important computational problems are difficult to solve optimally. In fact, many of those problems are NP-hard, which means that no polynomial-time algorithm exists that solves the problem optimally unless $P=NP$. A well-known example is the Euclidean traveling salesman problem (Euclidean TSP): given a set of points in the plane, find a shortest tour that visits all the points.

Another famous NP-hard problem is independent set: given a graph $G = (V, E)$, find a maximum size independent set $V' \subseteq V$. (A subset is independent if no two vertices in the subset are connected by an edge.) What can we do when faced with such difficult problems, for which we cannot expect to find polynomial-time algorithms? Unless the input size is really small, an algorithm with exponential running time is not useful. We therefore have to give up on the requirement that we always solve the problem optimally, and settle for a solution close to optimal. Ideally, we would like to have a guarantee on how close to optimal the solution is. For example, we would like to have an algorithm for Euclidean TSP that always produces a tour whose length is at most a factor ρ times the minimum length of a tour, for a (hopefully small) value of ρ . We call an algorithm producing a solution that is guaranteed to be within some factor of the optimum an approximation algorithm. This is in contrast to heuristics, which may produce good solutions but do not come with a guarantee on the quality of their solution.

The objective of a heuristic is to produce quickly enough a solution that is good enough for solving the problem at hand. This solution may not be the best of all the actual solutions to this problem, or it may simply approximate the exact solution. But it is still valuable because finding it does not require a prohibitively long time. Heuristics may produce results by themselves, or they may be used in conjunction with optimization algorithms to improve their efficiency (e.g., they may be used to generate good seed values).

It is difficult to imagine the variety of existing computational tasks and the number of algorithms developed to solve them. Algorithms that either give nearly the right answer or provide a solution not for all instances of the problem are called heuristic algorithms. This group includes a plentiful spectrum of methods based on traditional techniques as well as specific ones.

10.5 Big Oh Notation (Algorithm Analysis)

Big Oh is a characteristic scheme that measures properties of algorithm complexity performance and/or memory requirements. The algorithm complexity can be determined by eliminating constant factors in the analysis of the algorithm. Clearly, the complexity function $f(n)$ of an algorithm increases as 'n' increases. Let us find out the algorithm complexity by analyzing the sequential searching algorithm. In the sequential search algorithm we simply try to match the target value against each value in the memory. This process will continue until we find a match or finish scanning the whole elements in the array. If the array contains 'n' elements, the maximum possible number of comparisons with the target value will be 'n' i.e., the worst case. That is the target value will be found at the nth position of the array. $f(n) = n$. The best case is when the number of steps is less as possible. If the target value is found in a sequential search array of the first position (i.e., we need to compare the target value with only one element from the array) we have found the element by executing only one iteration $f(n) = 1$.

Average case falls between these two extremes (i.e., best and worst). If the target value is found at the $n/2$ position, on an average we need to compare the target value with only half of the elements in the array, so $f(n) = n/2$. The complexity function $f(n)$ of an algorithm increases as 'n' increases. The function $f(n) = O(n)$ can be read as "f of n is big Oh of n" or as "f(n) is of the order of n". The total running time (or time complexity) includes the initializations and several other iterative statements through the loop.

10.6 Growth Rate

We try to estimate the approximate computation time by formulating it in terms of the problem size N. If we consider that the system dependent factor (such as the compiler, language, computer) is constant; not varying with the problem size we can factor it out from the growth rate. The growth rate is the part of the formula that varies with the problem size. We use a notation called O-notation ("growth rate", "big-O"). The most common growth rates in data structure are:

Based on the time complexity representation of the big Oh notation, the algorithm can be categorized as : 1. Constant time $O(1)$

2. Logarithmic time $O(\log n)$
3. Linear time $O(n)$
4. Polynomial time $O(n^c)$ Where $c > 1$
5. Exponential time $O(c^n)$ Where $c > 1$

If we calculate these values we will see that as N grows $\log(N)$ remains quite small and $N \log(N)$ grow fairly large but not as large as N^2 . Eg. Most sorting algorithms have growth rates of $N \log(N)$ or N^2 . Following table shows the growth rates for a given N.

N	Constant	$\log(N)$	$N \log(N)$	N^2
1	1	0	0	1
2	1	1	2	4
4	1	2	8	16
8	1	3	24	64
32	1	5	160	1024
256	1	8	2048	65536
2048	1	11	22528	4194304

10.6.1 Estimating the growth Rate

Algorithms are developed in a structured way; they combine simple statements into complex blocks in four ways:

- Sequence, writing one statement below another
- Decision, if-then or if-then-else
- Loops
- Subprogram call

Let us estimate the big-O of some algorithm structures.

Simple statements: We assume that statement does not contain a function call. It takes a fixed amount to execute. We denote the performance by $O(1)$, if we factor out the constant execution time we are left with 1.

Sequence of simple statements: It takes an amount of execution time equal to the sum of execution times of individual statements. If the performance of individual statements are $O(1)$, so is their sum.

Decision: For estimating the performance, then and else parts of the algorithm are considered independently. The performance estimate of the decision is taken to be the largest of the two individual big Os. For the case structure, we take the largest big O of all the case alternatives.

Simple counting loop: This is the type of loop in which the counter is incremented or decrement each time the loop is executed (for loop). If the loop contains simple statements and the number of times the loop executes is a constant; in other words, independent of the problem size then the performance of the whole loop is $O(1)$. On the other hand if the loop is like Eg: for ($i=0$; $i < N$; $i++$)

The number of trips depends on N ; the input size, so the performance is $O(N)$. Nested loops: The performance depends on the counters at each nested loop. For ex:

```
for (i=0; i < N; i++)
{
    for (j=0; j < N; j++)
    {
        sequence of simple statements
    }
}
```

$$\sum_{i=0}^{N-1} \sum_{j=0}^{N-1} 1 = \sum_{i=0}^{N-1} N = N \sum_{i=0}^{N-1} 1 = N^2$$

The outer loop count is N but the inner loop executes N times for each time. So the body of the inner loop will execute $N*N$ and the entire performance will be $O(N^2)$.

```
for (i=1; i <= N; i++)
{
    for (j=0; j < i; j++)
    {
        sequence of simple statement
    }
}
```

$$\sum_{i=1}^N \sum_{j=0}^{i-1} 1 = \sum_{i=1}^N i = \frac{N(N+1)}{2} = \frac{N^2}{2} + \frac{N}{2}$$

In this case outer count trip is N , but the trip count of the inner loop depends not only N , but the value of the outer loop counter as well. If outer counter is 1, the inner loop has a trip count 1 and so on. If outer counter is N the inner loop trip count is N .

How many times the body will be executed?

$1+2+3+\dots+(N-1)+N = N(N+1)/2 = ((N^2) + N)/2$ Therefore the performance is $O(N^2)$.

Generalization: A structure with k nested counting loops where the counter is just incremented or decrement by one has performance $O(N^k)$ if the trip counts depends on the problem size only.

Note:**✚ Extra:**

Big O notation is used in Computer Science to describe the performance or complexity of an algorithm.

O(1)

O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

```
bool IsFirstElementNull(ICollection<string> elements)
{
    return elements[0] == null;
}
```

O(N)

O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set. The example below also demonstrates how Big O favours the worst-case performance scenario; a matching string could be found during any iteration of the for loop and the function would return early, but Big O notation will always assume the upper limit where the algorithm will perform the maximum number of iterations.

```
bool ContainsValue(ICollection<string> elements, string value)
{
    foreach (var element in elements)
    {
        if (element == value) return true;
    }

    return false;
}
```

O(N²)

O(N²) represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in O(N³), O(N⁴) etc.

```
bool ContainsDuplicates(ICollection<string> elements)
{
    for (var outer = 0; outer < elements.Count; outer++)
    {
        for (var inner = 0; inner < elements.Count; inner++)
        {
            // Don't compare with self
            if (outer == inner) continue;

            if (elements[outer] == elements[inner]) return true;
        }
    }

    return false;
}
```


O(2^N)

O(2^N) denotes an algorithm whose growth doubles with each addition to the input data set. The growth curve of an O(2N) function is exponential - starting off very shallow, then rising meteorically. An example of an O(2N) function is the recursive calculation of Fibonacci numbers:

```
int Fibonacci(int number)
{
    if (number <= 1) return number;

    return Fibonacci(number - 2) + Fibonacci(number - 1);
}
```

Logarithms

Logarithms are slightly trickier to explain so I'll use a common example:

Binary search is a technique used to search sorted data sets. It works by selecting the middle element of the data set, essentially the median, and compares it against a target value. If the values match it will return success. If the target value is higher than the value of the probe element it will take the upper half of the data set and perform the same operation against it. Likewise, if the target value is lower than the value of the probe element it will perform the operation against the lower half. It will continue to halve the data set with each iteration until the value has been found or until it can no longer split the data set.

This type of algorithm is described as O(log N). The iterative halving of data sets described in the binary search example produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase e.g. an input data set containing 10 items takes one second to complete, a data set containing 100 items takes two seconds, and a data set containing 1000 items will take three seconds. Doubling the size of the input data set has little effect on its growth as after a single iteration of the algorithm the data set will be halved and therefore on a par with an input data set half the size. This makes algorithms like binary search extremely efficient when dealing with large data sets.

HEURISTIC AND APPROXIMATE ALGORITHMS

A heuristic is a much broader term than approximation algorithm.

An absolute approximation algorithm as the name suggests has an absolute error with respect to the objective value of an optimal solution. The important other property is that it terminates in polynomial time with respect to the input size.

A heuristic does not need these two properties. A heuristic is a vague term that can capture a "sometimes right, sometimes wrong" approach of approximating a solution (may not even be an algorithm) or an approach of designing algorithms that may not have any properties other than giving you a feasible solution. There are lots of types of heuristic algorithms, even for approximation algorithms.

SERIES AND PARALLEL ALGORITHM

In computer science, a sequential algorithm or serial algorithm is an algorithm that is executed sequentially – once through, from start to finish, without other processing executing

In computer science, a parallel algorithm, as opposed to a traditional serial algorithm, is an algorithm which can be executed a piece at a time on many different processing devices, and then combined together again at the end to get the correct result

The cost or complexity of serial algorithms is estimated in terms of the space (memory) and time (processor cycles) that they take. Parallel algorithms need to optimize one more resource, the communication between different processors. There are two ways parallel processors communicate, shared memory or message passing.

DIVIDE AND CONQUER ALGORITHM

In computer science, divide and conquer (D&C) is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

This divide and conquer technique is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g., quicksort, merge sort)