Class Notes: Algorithms

Phil Mayer

The following document contains my notes from the Algorithms course I took at Fairfield University in the Fall 2015 semester. The course was taught by Phil LaMastra and covered Levitin's *Introduction to The Design and Analysis of Algorithms*, 3rd edition. My notes were originally typed using Apple Pages using embedded formula objects. They have been transcribed into LaTeX. A large language model was used for the transcription, so there may be errors in the formulas or content below.

The most recent version of this document was prepared on June 12, 2026. I plan to continue revising this document for readability. Some sections need to be extended, shortened, or re-formatted.

Contents

1 Getting Started

Recall that an **algorithm** is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time. We can consider algorithms to be procedural solutions to problems. Yet, algorithms themselves are not the answers to particular problems—they are instructions for arriving at an answer.

The first step in working with algorithms is understanding the problem. The second step is to see that the specific inputs specified when an algorithm runs are mere instances of a more general problem. Algorithms should work for all legitimate inputs, rather than specific sets of information. Next, one should determine the method the algorithm uses to solve the problem: **exact algorithms** solve the problem exactly, while **approximation algorithms** solve the problem by estimation.

Once an algorithm has been specified, its **correctness** should be verified. This process determines whether the algorithm yields a correct result for every legitimate input in a finite amount of time. In addition, our algorithms should be as efficient as possible in terms of time and space complexity.

2 Analyzing Iterative Algorithms

When we analyze algorithms, we should examine their **time efficiency** (also called time complexity; how fast the algorithm runs) and **space efficiency** (how much extra memory the algorithm uses). For now, we will limit our discussion to time complexity.

To analyze an algorithm, the first step is to select a parameter to represent the size of a possible input. While this can be obvious at times, the choice influences the rest of the analysis significantly. Next we identify the **basic operation** performed by the algorithm: the action that contributes the most to the total running time. While we can occasionally analyze algorithms by counting the number of times each operation is executed, this is excessive. It is possible that some operations are executed in negligible time, while others dominate the total computation time.

The established framework for analyzing an algorithm's time efficiency is to count the number of times the algorithm's basic operation is executed for inputs of size n. To approximate an algorithm's running time, $T(n)$, we can use the following formula:

$$T(n) \approx c_{op}\, C(n)$$

where c_{op} represents the cost of the basic operation and $C(n)$ represents the number of times the algorithm executes it for an input of size n.

More exact analysis of time complexity can be achieved through the use of worst-case, best-case, and average-case efficiency analysis. The **worst-case** efficiency is the running time given an input (or inputs) for which the algorithm runs the longest among all possible inputs of that size. The **best-case** efficiency of an algorithm is its running time given an input (or inputs) for which the algorithm runs the fastest among all possible inputs of that size.

Neither the worst-case nor best-case efficiencies can typically yield the necessary information to understand the **average-case** efficiency, or the algorithm's efficiency given a "typical" or "random" input. To analyze an algorithm's average-case efficiency, one should consider the probability of success and failure at each iteration of the algorithm.

Overall, analyzing algorithm efficiency concentrates significantly on the order of growth of an algorithm's basic operation count. To compare and rank **orders of growth**, we use three major notations: O (big-oh), Θ (big-theta), and Ω (big-omega). Informally, given two functions $f(n)$ and $g(n)$:

$f(n) \in O(g(n))$ means that $f(n)$ grows no faster than $g(n)$. In other words, this means $f(n) \leq cg(n)\ \forall n \geq n_0$.

$f(n) \in \Theta(g(n))$ means that $f(n)$ grows at roughly the same rate as $g(n)$ for some constants c_1, c_2. In mathematical language, this translates to: $c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0$.

$f(n) \in \Omega(g(n))$ means that $f(n)$ grows at least as fast as $g(n)$, or $f(n) \geq cg(n)\ \forall n \geq n_0$.

Growth classes for two functions $f(n)$ and $g(n)$ can be determined relatively quickly using limits:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & f(n) \in O(g(n)) \\ c & f(n) \in \Theta(g(n)) \\ \text{dne} & f(n) \in \Omega(g(n)) \end{cases}$$

To evaluate some limits, it may be necessary to use **L'Hôpital's rule** or **Stirling's formula**. See Section 17 for these equations.

When analyzing order of growth using limits, it becomes apparent that all basic logarithmic functions belong to the same order of growth class. Differences in base only produce a constant-multiple difference between two logarithmic functions. In contrast, exponential functions of different bases do not belong to the same growth classes. This result can be seen using L'Hôpital's rule: for some exponential functions $f(n)$ with base a and $g(n)$ with base b, $f(n) \notin \Theta(g(n))$ because their rates of growth are significantly different.

3 Analyzing Recursive Algorithms

When analyzing recursive algorithms, nearly all of the same techniques apply. The major difference between the analysis of iterative and recursive algorithms is that counting the number of executions of some basic operation is typically much more difficult for a recursive algorithm. Most of the time, a **recurrence relation** will need to be set up and solved. When setting up recurrences, the first step is to determine how the input is changed after each successive step: in some recurrences, the input size may be reduced by two. In others, the input size might be decreased by one until it reaches a **base-case** value.

Recurrences can be solved by means of substitution, characteristic equation, and the **master method**. Substitution is relatively straightforward, yet requires care because mistakes can be made easily. To solve a recurrence by characteristic equation, set up a homogeneous linear recurrence equation (HLRE) then convert to its characteristic equation:

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0 \quad \longrightarrow \quad a_0 r^k + a_1 r^{k-1} + \cdots + a_k = 0$$

Then find the roots of the equation and solve using the initial conditions:

$$t_n = c_1 r_1^n + c_2 r_2^n + \cdots + c_n r_n^n$$

where each r represents a root of the characteristic equation and each constant c_i is the corresponding constant value for each term. There are several special cases when solving a recurrence by characteristic equation:

- Where roots are repeated, it is said that there are **roots of multiplicity**. For multiplicity m, $t_n = n^{m-1} r^n$.
- For recurrences of the form

  $$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = b^n p(n)$$

  the recurrence has a characteristic equation that incorporates the polynomial function $p(n)$ and its degree, d:

  $$(a_0 r^k + a_1 r^{k-1} + \cdots + a_k)(r - b)^{d+1} = 0$$

Though substitution and the method of characteristic equation are reliable methods of solving recurrences, they can be tedious. Fortunately, for recurrences where $T(n)$ is eventually decreasing and has the form

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^d)$$

and the constants $a \geq 1$, $b \geq 2$, $d \geq 0$, then if $n > 1$:

$$T(n) \in \begin{cases} \Theta(n^d) & a < b^d \\ \Theta(n^d \log_b(n)) & a = b^d \\ \Theta(n^{\log_b(a)}) & a > b^d \end{cases}$$

Finally, when recursive algorithms require specific, discrete input sizes, solve for those specific input sizes, then apply the **smoothness rule** to extend the result to all n.

4 The Quadratic Sorts

Of the sorting algorithms, the simplest are the quadratic-time sorts: selection sort, bubble sort, and insertion sort. For the most part, these algorithms operate in $O(n^2)$ time, making them acceptable but with room for improvement.

4.1 Selection Sort

First, consider **selection sort**:

Algorithm 1 SelectionSort($A[0 \dots n-1]$)

```
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min] then
            min ← j
        end if
    end for
    swap A[i] and A[min]
end for
```

The double-*for* loop structure immediately suggests quadratic efficiency, and further analysis confirms it. Selection sort has quadratic time complexity regardless of the input. Considering the basic operation to be array element comparison:

$$\begin{aligned}
C(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} \big[(n-1) - (i+1) + 1\big] \\
&= \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1 \\
&= n(n-1) - \frac{(n-2)(n-1)}{2} - (n-1) \\
&= n^2 - n - \tfrac{1}{2}(n^2 - 3n + 2) - n + 1 \\
&= \tfrac{1}{2}n^2 - \tfrac{1}{2} \\
&\therefore C(n) \in \Theta(n^2)
\end{aligned}$$

Because the algorithm's execution does not vary with the input, its time complexity is always $O(n^2)$. Interestingly, selection sort's strength is in its number of swaps: in total, the algorithm performs only $n-1$ swaps.

4.2 Bubble Sort

Next, consider **bubble sort**:

Algorithm 2 BubbleSort($A[0 \dots n-1]$)

```
for i ← 0 to n − 2 do
    for j ← 0 to n − 2 − i do
        if A[j + 1] < A[j] then
            swap A[j] and A[j + 1]
        end if
    end for
end for
```

Once again, this algorithm features a double-*for* loop structure. Considering the basic operation to be array key comparisons:

$$\begin{aligned} C(n) &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} \big[(n-2-i) - 0 + 1\big] \\ &= \sum_{i=0}^{n-2} (n-1-i) = (n)(n-1) - (n-1) - \frac{(n-2)(n-1)}{2} \\ &= \frac{(n-1)n}{2} \\ &\therefore C(n) \in \Theta(n^2) \end{aligned}$$

Generally, bubble sort is a mediocre algorithm but represents a brute-force solution that can motivate more efficient solutions.

4.3 Insertion Sort

Finally, **insertion sort** is the last relatively simple sorting algorithm that runs in quadratic time for its worst-case and average-case efficiencies. Interestingly, it is actually one of the best sorting algorithms given a nearly-sorted array.

Algorithm 3 InsertionSort($A[0 \dots n-1]$)

```
for i ← 1 to n − 1 do
    j ← i − 1
    while j ≥ 0 and A[j] > A[i] do
        A[j + 1] ← A[j]
        j ← j − 1
    end while
    A[j + 1] ← A[i]
end for
```

The insertion sort does not use a double-*for* structure, instead using an inner *while* loop to ensure that "insertions" are only made when necessary. Overall, the algorithm works by a decrease-by-one strategy: assuming that the sub-array $A[0 \dots n-2]$ has already been sorted, the algorithm places the element $A[n-1]$ in its appropriate place on the array.

Considering "insertion" to be the basic operation, the worst-case and average-case efficiencies are:

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \qquad \therefore C_{worst}(n) \in \Theta(n^2)$$

$$C_{average}(n) = \sum_{i=1}^{n-1} \left\lfloor \frac{1}{2} \sum_{j=0}^{i} 1 \right\rfloor \approx \frac{n^2}{4} \qquad \therefore C_{average}(n) \in \Theta(n^2)$$

The best-case efficiency, however, is actually linear. In the case that the algorithm has to perform a negligible number of insertions:

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \qquad \therefore C_{best}(n) \in \Theta(n)$$

5 The Closest Pair Problem

In computational geometry, many applications call for finding the pair of two points p_1 and p_2 in a set P of n points such that the Euclidean distance between the two points is the minimum on the set. Recall that the Euclidean distance for two points $p_i(x_i, y_i)$ and $p_j(x_j, y_j)$ is calculated as:

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

The brute-force strategy for finding the minimum distance on the set P is to compute the distance between each pair of distinct points, then find the two points whose distance is the smallest. To prevent calculating the distance between two points twice, we consider only the pairs of points p_i, p_j such that $i < j$. The algorithm is:

Algorithm 4 BFClosestPair(P)

```
d ← ∞
for i ← 1 to n − 1 do
    for j ← i + 1 to n do
        d ← min(d, √((x_i − x_j)^2 + (y_i − y_j)^2))
    end for
end for
```

For this algorithm, we take squaring to be the basic operation. This is because the algorithm can be implemented without actually taking the square root of $(x_i - x_j)^2 + (y_i - y_j)^2$ at all. Square distance comparison between two points gives a sufficient measure of distance between two points (for the sake of comparison), and ultimately determines the closest pair of points equivalently, so the algorithm is still correct. Square roots are also typically relatively expensive to compute, so it is more efficient to avoid them when possible.

Since the number of executions of the basic operation does not change depending on the input:

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 = 2 \sum_{i=1}^{n-1} (n - i) \\ &= 2\left[n(n-1) - \frac{(n-1)n}{2}\right] = n(n-1) \\ &\therefore C(n) \in \Theta(n^2) \end{aligned}$$

6 The Convex Hull Problem

The next problem to review is known as the convex hull problem. The premise is: given a set of points P, determine the subset of points P_c that form a convex polygon that contains all points in P. More rigorously defined, a **convex hull** is the smallest set P_c such that all other points in P are bounded inside the area formed by P_c.

The convex hull problem has a variety of useful applications. Often considered one of the most important computational geometry problems, the convex hull has applications in linear programming. The brute-force algorithm that solves the convex hull problem is:

Algorithm 5 BFConvexHull(P)

```
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        if (x_i, y_i) ≠ (x_j, y_j) then
            draw a line from (x_i, y_i) to (x_j, y_j)
            for k ← 0 to n − 1 do
                if i ≠ k and j ≠ k then
                    check if all other points lie on the same side of line (x_i, y_i)(x_j, y_j)
                    add (x_i, y_i) → (x_j, y_j) to the list of extreme points
                end if
            end for
        end if
    end for
end for
```

To determine whether all other points in the set P lie on the same side of any of the test lines, all points must either be greater than or less than the line's value: $ax + by < c$, or $ax + by > c$ for constants $a = y_2 - y_1$, $b = x_1 - x_2$, and $c = x_1y_2 - x_2y_1$. Considering the basic operation to be this verification:

$$C(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3$$
$$\therefore C(n) \in O(n^3)$$

7 Breadth-First and Depth-First Search

Consider the class of brute-force search techniques called **exhaustive search**. Given a list or array, exhaustive search would determine whether or not some key exists on the list by sequentially accessing each list element until either finding the element or reaching the end of the list. But what if we need to search a more complex structure, such as a graph? In this section, we will examine two such algorithms for finding an element on a graph.

The **depth-first search (DFS)** is considered the "brave" approach: starting at an arbitrary vertex, it marks the current vertex as visited and proceeds to an adjacent unvisited vertex. Depth-first searches can be accompanied by a stack to trace the path taken to find the element (if it exists on the graph), as well as to find dead-end vertices. In addition, the depth-first search can also be used to check for component connectivity and graph acyclicity.

Algorithm 6 DepthFirstSearch($G = \{V, E\}$)

```
mark each vertex in V as unvisited
count ← 0
for each vertex v in V do
    if v is unvisited then
        dfs(v)
    end if
end for
```

Algorithm 7 dfs(v)

```
count ← count + 1
mark v as visited
for each vertex w in V adjacent to v do
    if w is unvisited then
        dfs(w)
    end if
end for
```

The time complexity of the depth-first search is of class $\Theta(|V|^2)$ for implementations using an adjacency matrix, and $\Theta(|V| + |E|)$ for implementations using an adjacency list (given a number of vertices $|V|$ and a number of edges $|E|$).

The second technique for searching for elements on a graph structure is the **breadth-first search (BFS)**. While the DFS can be considered "brave" because it tends to look for the desired element as far as possible from the starting vertex, the BFS is more "cautious" in its approach. The breadth-first search starts at an arbitrary vertex, checks all unvisited adjacent vertices, then all unvisited vertices on the next level. On each iteration, the algorithm identifies all unvisited vertices adjacent to the current vertex, marks them as visited, adds them to a queue, then dequeues the current vertex.

Algorithm 8 BreadthFirstSearch($G = \{V, E\}$)

```
mark each vertex in V as unvisited
count ← 0
for each vertex v in V do
    if v is unvisited then
        bfs(v)
    end if
end for
```

Algorithm 9 bfs(v)

```
count ← count + 1
mark v with count
initialize a queue with v
while queue is not empty do
    for each vertex w in V adjacent to current vertex do
        if w is unvisited then
            count ← count + 1
            mark w with count
            add w to the queue
        end if
    end for
    remove front vertex from the queue
end while
```

The breadth-first search can also check component connectivity and acyclicity, but it has some unique strengths. Due to the FIFO nature of the queue, BFS always reaches each vertex via a shortest path (fewest edges) from the starting vertex. In addition, the algorithm's time complexity is exactly the same as the DFS.

	Depth-first search (DFS)	**Breadth-first search (BFS)**
Data structure	a stack	a queue
Vertex orderings	two orderings	one ordering
Edge types (undirected)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Adjacency matrix	$\Theta(\|V\|^2)$	$\Theta(\|V\|^2)$
Adjacency list	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

8 Topological Sorting

To understand the next problem, we first need to understand the idea of a **directed acyclic graph (DAG)**, a directed graph where no cycles exist. Cycles are considered to be sequences of two or more vertices that start and end at the same vertex, potentially with other distinct vertices between.

The basic idea of the **topological sorting problem** is to find an ordering of vertices on a graph such that for every edge, the vertex where the edge starts is listed before the vertex where the edge ends. It becomes easy to see that the problem has no solution if the graph has a directed cycle.

One algorithm that solves this problem is to simply perform a depth-first search while maintaining a stack. Keeping track of the order in which vertices become dead-ends (in other words, the order in which vertices are popped off the traversal stack). Reversing this order yields a valid topological sort.

A second algorithm uses a decrease-by-one technique: repeatedly identify a vertex on the graph with no incoming edges, then delete it and all edges leading to other vertices from it. The order in which vertices are deleted yields a solution by method of **source removal**.

9 Generating Permutations

When we generate permutations in mathematics, for example when we deal with power sets, we sometimes take our ability to enumerate combinations and permutations for granted. Devising algorithms for generating the different permutations of an object can be difficult. In this section, we will discuss two algorithms that solve this problem.

The first of the two techniques is the **Johnson-Trotter** algorithm. This approach uses the notion of a **mobile** element. An element is said to be mobile if its arrow points toward an adjacent element with a smaller value.

Algorithm 10 JohnsonTrotter(n)

```
initialize the first permutation as 1 2 ··· n with all arrows pointing left
while the last permutation has a mobile element do
    find its largest mobile element k
    swap k with the adjacent element k's arrow points to
    reverse the direction of all elements larger than k
    add the new permutation to the list
end while
```

Johnson-Trotter is one of the most efficient algorithms for generating permutations. Yet, its time complexity at best still belongs to the class $\Theta(n!)$. Since any solution to the problem must output all $n!$ permutations, $\Theta(n!)$ is an unavoidable lower bound. For example, the algorithm applied to input size $n = 3$ produces:

$$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{3} \;\rightarrow\; \overleftarrow{1}\,\overleftarrow{3}\,\overleftarrow{2} \;\rightarrow\; \overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{2} \;\rightarrow\; \overleftarrow{3}\,\overleftarrow{2}\,\overleftarrow{1} \;\rightarrow\; \overleftarrow{2}\,\overleftarrow{3}\,\overleftarrow{1} \;\rightarrow\; \overleftarrow{2}\,\overleftarrow{1}\,\overleftarrow{3}$$

The second technique for generating permutations uses **binary reflected Gray code**. The algorithm was designed by Frank Gray at AT&T Bell Laboratories:

Algorithm 11 BRGC(n)

```
if n = 1 then
    make list L containing bit strings 0 and 1 in this order
else
    generate list L1 of bit strings of size n − 1 by calling BRGC(n − 1)
    copy list L1 to L2 in reversed order
    add 0 in front of each bit string in L1
    add 1 in front of each bit string in L2
    append L2 to L1 to get list L
end if
```

To do a sample run, consider input size $n = 3$. The algorithm would first call BRGC(2) and then BRGC(1). The latter call would return $L = \{0, 1\}$, which would then be named L_1 corresponding to the call BRGC(2). After reversing L_1 to obtain $L_2 = \{1, 0\}$, the appending step would result in $L_1 = \{00, 01\}$ and $L_2 = \{11, 10\}$. Returning the resulting list $L = \{00, 01, 11, 10\}$ to the starting call BRGC(3), L would then be labeled L_1. Once again reversing L_1 to obtain $L_2 = \{10, 11, 01, 00\}$, the

appending step would result in $L_1 = \{000, 001, 011, 010\}$ and $L_2 = \{110, 111, 101, 100\}$. Appending these two lists together, the final resulting list of all permutations would be:

$$L = \{000, 001, 011, 010, 110, 111, 101, 100\}$$

10 Lomuto Partitioning

Another interesting area in which the decrease-and-conquer methodology yields a useful solution is the **selection problem**. The task is to find the k-th smallest element in a list of n numbers. One particular case of this problem is in computing medians: its index value is computed as the ceiling of $n/2$.

One way to solve the problem of computing medians is to simply sort the given list and find the middle element. The time complexity for this approach would be restricted heavily by the complexity of the sorting algorithm however, so this is not an optimal approach. Instead, we can use **partitioning** to rearrange elements in the list based on the value of some **pivot** p. A partitioned array A, for example, will be arranged such that all values $A[i] \leq p$ are to the left of p, followed by p, then all values $A[i] \geq p$ will be to the right of the pivot. The two techniques for performing this task are **Lomuto partitioning** and **Hoare partitioning**. In this section, we will explore the former.

The algorithm generally works by first maintaining three contiguous sub-arrays of the array to be partitioned: a segment with values known to be less than or equal to p, a segment with values known to be greater than or equal to p, and a segment whose values have not yet been compared.

Algorithm 12 LomutoPartition($A[l \dots r]$)

```
p ← A[l]
s ← l
for i ← l + 1 to r do
    if A[i] ≤ p then
        s ← s + 1
        swap A[s] and A[i]
    end if
end for
swap A[l] and A[s]
return s
```

Lomuto partitioning has a time complexity of $\Theta(n)$, considering array element comparison to be the basic operation. Yet, how can we leverage this efficient partitioning algorithm in order to solve the selection problem? In the case that the final pivot position $s = k - 1$, then the value of the pivot is the k-th smallest element on the array. But if $s > k - 1$, then the k-th smallest element must be at index $k - 1$ of the left part of the partitioned array. If $s < k - 1$, then it can be found as the $(k - s)$-th smallest element in the right part. The algorithm that uses this fact is called **quickselect**:

Algorithm 13 QuickSelect($A[l \dots r]$, k)

```
s ← LomutoPartition(A[l ... r])
if s = k − 1 then
    return A[s]
else if s > (l + k − 1) then
    QuickSelect(A[l ... s − 1], k − 1)
else
    QuickSelect(A[(s + 1) ... r], k − 1 − s)
end if
```

Overall quickselect is also a very efficient algorithm, and when coupled with Lomuto partitioning, can be used to solve the selection problem.

11 Variable-Size Algorithms

Along with the selection problem discussed in the previous section, two other algorithms that use the decrease-and-conquer strategy by variable size include **Euclid's algorithm** and another technique for searching.

Euclid's algorithm is a technique used to find the greatest common divisor of two integers m and n. After two iterations of Euclid's algorithm, the problem size decreases by at least two:

Algorithm 14 EuclidGCD(m, n)

```
while n ≠ 0 do
    t ← n
    n ← m mod n
    m ← t
end while
```

Overall, Euclid's algorithm performs very efficiently. When expressed as a recurrence relation, its running time $T(n)$ can be approximated as:

$$T(n) \approx T\left(\frac{n}{\sqrt{2}}\right) + 1$$

By the master method, it can then be shown that the algorithm is of efficiency class $\Theta(\log n)$. Intuitively, this makes sense since the input size is continually reduced by a factor of $\sqrt{2}$ with each iteration.

The other noteworthy algorithm that works by continually cutting the input size by some variable amount is **interpolation search**. Unlike most other search algorithms, the interpolation search takes into account the value of the search key in order to find the optimal element to compare with on the array. The best way to intuitively understand how it works is by thinking about how one might look for a word in a dictionary. Rather than opening to the middle as binary search would, interpolation search estimates where the target is likely to appear — similar to flipping near the front of a dictionary when searching for a word starting with the letter A.

12 Merge Sort

In reviewing the divide-and-conquer class of algorithms, the first important example we will consider is merge sort. This sorting algorithm sorts a given array $A[0 \dots n-1]$ by dividing it into two halves: $A[0 \dots \lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor \dots n-1]$. Each of the two subarrays is then sorted recursively, and the two sorted halves are merged.

Algorithm 15 MergeSort($A[0 \dots n-1]$)

```
if n > 1 then
    copy A[0 ... ⌊n/2⌋ − 1] to B[0 ... ⌊n/2⌋ − 1]
    copy A[⌊n/2⌋ ... n − 1] to C[0 ... ⌈n/2⌉ − 1]
    MergeSort(B[0 ... ⌊n/2⌋ − 1])
    MergeSort(C[0 ... ⌈n/2⌉ − 1])
    Merge(B, C, A)
end if
```

Algorithm 16 Merge($B[0 \dots p-1]$, $C[0 \dots q-1]$, $A[0 \dots p+q-1]$)

```
i ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j] then
        A[k] ← B[i]; i ← i + 1
    else
        A[k] ← C[j]; j ← j + 1
    end if
    k ← k + 1
end while
if i = p then
    copy C[j ... q − 1] to A[k ... p + q − 1]
else
    copy B[i ... p − 1] to A[k ... p + q − 1]
end if
```

Merge sort's efficiency can be understood by solving the recurrence relation $C(n) = 2C(\lfloor n/2 \rfloor) + n - 1$ when $n > 1$ and $C(1) = 0$. By the master method ($a = 2$, $b = 2$, $d = 1$, so $a = b^d$):

$$C(n) \in \Theta(n \log n)$$

13 Quick Sort

Unlike merge sort, which divides its input elements according to their position in the original array, quick sort divides the array elements by their value. Quick sort is made possible by partitioning the array to be sorted into segments about some pivot value.

Consider the array $A[0 \dots n-1]$. If $A[s]$ is chosen as the pivot value for some s, then the partition about $A[s]$ looks like:

$$\underbrace{A[0], \dots, A[s-1]}_{\leq A[s]}, \quad A[s], \quad \underbrace{A[s+1], \dots, A[n-1]}_{\geq A[s]}$$

Once $A[s]$ is in its final position, the left and right subarrays can both be sorted independently by the same method. No merge operation is required; all work is done in the division stage.

Algorithm 17 QuickSort($A[l \dots r]$)

```
if l < r then
    s ← Partition(A[l ... r])
    QuickSort(A[l ... s − 1])
    QuickSort(A[s + 1 ... r])
end if
```

Quick sort can be optimized by using **Hoare partitioning**, a technique that performs more efficiently in many cases:

Algorithm 18 HoarePartition($A[l \dots r]$)

```
p ← A[l]
i ← l;  j ← r + 1
repeat
    repeat
        i ← i + 1
    until A[i] ≥ p
    repeat
        j ← j − 1
    until A[j] ≤ p
    swap A[i] and A[j]
until i ≥ j
swap A[i] and A[j]                          ▷ undo last swap when i ≥ j
swap A[l] and A[j]
return j
```

Quick sort's efficiency is tied closely to the partitioning operation. If the split consistently occurs close to either end of the array, the worst case occurs:

$$C_{worst}(n) = \sum_{i=0}^{n-1}(n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2)$$

In summary: $\Theta(n^2)$ in the worst case, $\Theta(n \log n)$ in the average case, and $\Theta(n \log n)$ in the best case. An efficiency of $\Theta(n)$ can be achieved with a three-way partition and equal keys, but this is a very specific theoretical case.

14 Karatsuba Multiplication

The following technique for multiplying large integers is often called Karatsuba multiplication. Given two n-digit numbers written as $A = a \cdot 10^{n/2} + b$ and $B = c \cdot 10^{n/2} + d$:

$$A \cdot B = ac \cdot 10^n + (ad + bc) \cdot 10^{n/2} + bd$$

The key observation is that the middle term $(ad + bc)$ can be computed with one multiplication rather than two:

$$ad + bc = (a + b)(c + d) - ac - bd$$

This reduces the four multiplications of the grade-school method to three. The corresponding recurrence for the number of single-digit multiplications $M(n)$ is:

$$M(n) = 3M(n/2) + cn$$

By the master method ($a = 3$, $b = 2$, $d = 1$, and $3 > 2^1$):

$$M(n) \in \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$$

This is better than the standard grade-school $\Theta(n^2)$ approach.

15 The Closest-Pair Problem Revisited

To more efficiently solve the closest pair problem (previously $\Theta(n^2)$ by brute force), consider a set of points sorted in nondecreasing order of their x-coordinate (array P) and the same list sorted in nondecreasing order of their y-coordinate (array Q). A $\Theta(n \log n)$ sort such as merge sort can be applied if the arrays are not already sorted.

If $n \leq 3$, solve by brute force. If $n > 3$, divide the points into P_{left} and P_{right} (and corresponding Q_{left}, Q_{right}) by drawing a median vertical line $x = m$. Solve each half recursively to obtain d_{left} and d_{right}, then $d = \min(d_{left}, d_{right})$.

Points in Q within distance d of the median form a strip S. Because at most 8 points can lie within any $d \times 2d$ rectangle, the strip check is $\Theta(n)$.

Algorithm 19 EfficientClosestPair(P, Q)

if $n \leq 3$ **then**
  return minimal distance by brute force
else
  copy first $\lceil n/2 \rceil$ points of P to P_{left}; same points of Q to Q_{left}
  copy remaining $\lfloor n/2 \rfloor$ points of P to P_{right}; same to Q_{right}
  $d_{left} \leftarrow$ EfficientClosestPair(P_{left}, Q_{left})
  $d_{right} \leftarrow$ EfficientClosestPair(P_{right}, Q_{right})
  $d \leftarrow \min(d_{left}, d_{right})$
  copy all points of Q with $|x - m| < d$ into $S[0 \ldots num - 1]$
  $d^2_{min} \leftarrow d^2$
  for $i \leftarrow 0$ to $num - 2$ **do**
    $k \leftarrow i + 1$
    while $k \leq num - 1$ and $(S[k].y - S[i].y)^2 < d^2_{min}$ **do**
      $d^2_{min} \leftarrow \min\big((S[k].x - S[i].x)^2 + (S[k].y - S[i].y)^2,\ d^2_{min}\big)$
      $k \leftarrow k + 1$
    end while
  end for
  return $\sqrt{d^2_{min}}$
end if

The efficiency of this solution is $\Theta(n \log n)$, derived from the recurrence $T(n) = 2T(n/2) + \Theta(n)$ (master method: $a = 2$, $b = 2$, $d = 1$, $a = b^d$). Note that checking minimum distances within the strip ends up being trivial since only six to eight points on average end up inside the area.

16 Horner's Rule and Binary Exponentiation

Horner's Rule is an efficient algorithm for evaluating a polynomial $p(x) = p_n x^n + p_{n-1}x^{n-1} + \cdots + p_0$ at some value of x. It is a good example of a representation-change algorithm that reduces both the number of multiplications and additions $M(n)$. The algorithm works by continuously factoring out each x-term:

$$p(x) = (\cdots((p_n x + p_{n-1})x + p_{n-2})x + \cdots + p_1)x + p_0$$

Algorithm 20 Horner($P[0 \ldots n]$, x)

```
p ← P[n]
for i ← n − 1 down to 0 do
    p ← x · p + P[i]
end for
return p
```

The number of multiplications (equal to the number of additions) is:

$$M(n) = n$$

Interestingly, the coefficient byproducts produced during the evaluation turn out to be the coefficients of the quotient $p(x)/(x - x_0)$.

Now consider an interesting application of Horner's Rule: **binary exponentiation**. The goal is to compute a^n using the binary representation $b(n) = b_I b_{I-1} \cdots b_0$ of n.

Algorithm 21 LeftRightBinaryExp(a, $b(n) = b_I \cdots b_0$)

```
product ← a
for i ← I − 1 down to 0 do
    product ← product × product
    if b_i = 1 then
        product ← product × a
    end if
end for
return product
```

The total number of multiplications satisfies:

$$(b - 1) \leq M(n) \leq 2(b - 1)$$

where $b = \lfloor \log_2 n \rfloor + 1$ is the number of bits in the binary representation of n. Since $b - 1 = \lfloor \log_2 n \rfloor$:

$$M(n) \in \Theta(\log n)$$

17 Reference: Useful Formulas

For analyzing order of growth classes using limits, **L'Hôpital's rule** is particularly useful:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

When analyzing order of growth classes where factorials are involved, **Stirling's formula** can help to evaluate the limit expressions:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$