CHAPTER 1

SORTING AND TRIES

1.0 Algorithm Analysis

The time complexity of an algorithm can be thought of as a function of the input size. For example, if we are given an input of size n, the time complexity denotes how long it takes for the algorithm to run. It is either the worst case or the average case of the algorithm.

The worst case analysis is the one specified most commonly. It gives a guarantee of the algorithm's performance. A key factor is the asymptotic behaviour- this indicates what will happen as the input size grows. It is generally expressed using the 'Big Oh' notation. In some cases, space complexity can also be significant.

Big-Oh notation

In 'Big-Oh' notation, we say that $f(n) = O(g(n))$ if f and g grow at approximately the same rate. Here, the domain of f and g is the natural numbers. Formally, we say that $f(n) = O(g(n))$ if there exists a real constant c and an integer constant N such that $|f(n)| \leq |c \cdot g(n)|$ for all $n \geq N$. Usually, we use this notation to better understand the function f; f is typically a complicated function while g is a well-understood function.

When using the 'Big-Oh' notation, we use the tightest g that we can. For example, selection sort has worst-case complexity $O(n^2)$. This also means that selection sort is $O(n^3)$, $O(n^4)$, $O(n^2 \log n)$, etc. but we do not say that any of these are the worst-case complexity. However, we cannot state that selection sort is $O(n \log n)$.

The logarithmic function

We say that $x = \log_a n$ if $n = a^x$. The following are few standard properties of the function:

- $\log_a m \cdot n = \log_a m + \log_a n$;
- $\log_a m/n = \log_a m - \log_a n$;
- $\log_a n^c = c \log_a n$.

The logarithmic function grows very slowly. For example, $\log_2 1000000 \approx 20$. For this reason, $O(n^c \log_2 n)$ is only a little worse than $O(n^c)$.

Using the standard properties of the logarithmic function, we can show that logarithms to different bases related by a constant factor. So, let $x = \log_a n$. In that case, $n = a^x$. Now, if we take $\log_b$ of both sides, we get

$$\log_b n = \log_b a^x = x \log_b a = (\log_a n) \cdot (\log_b a).$$

Since b and a are constants, $\log_b a$ is also a constant. Therefore, $\log_b n$ and $\log_a n$ are related by a constant factor. In terms of 'Big-Oh' notation, this implies that $O(\log_a n) = O(\log_b n)$. For this reason, we specify both of these as just $O(\log n)$.

Polynomial-time algorithms

A polynomial-time algorithm runs in $O(n^c)$. The execution time of algorithms with various complexity functions is shown below:

	1 000	10 000	100 000	1 000 000
n	10^{-6} secs	10^{-5} secs	10^{-4} secs	10^{-3} secs
$n \log_2 n$	10^{-5} secs	1.3×10^{-4} secs	1.6×10^{-3} secs	0.02 secs
n^2	0.001 secs	0.1 secs	10 secs	16 mins
n^3	1 sec	16 mins	11 days	32 years

Table 1.1: The execution time of polynomial-time algorithms with various complexity functions provided that we can complete 10^9 operations per second.

Clearly, the higher the power, the longer it takes for the algorithm to be solved. Nonetheless, the table above denotes the worst-case behaviour- it is possible for the average case behaviour might be much faster.

Exponential-time algorithms

An exponential-time algorithm runs in $O(c^n)$. The execution time of algorithms with various complexity functions is shown below:

	10	20	30	40
2^n	10^{-6} secs	0.001 secs	1.1 secs	18 mins
$n!$	0.004 secs	77 years	$\star$	$\star$

Table 1.2: The execution time of exponential-time algorithms with various complexity functions provided that we can complete 10^9 operations per second.

A star ($\star$) indicates that it takes longer than the age of the Earth for the algorithm to terminate. Looking at the two tables, we can say that polynomial-time algorithms are potentially useful in practice, while exponential-time algorithms are potential useless in practice.

1.1 Abstract Data Types

Stack

A stack is an abstract data type that holds a collection of objects. The objects are accessed using a last-in-first-out (LIFO) policy. The basic operations defined on a stack are:

- `create`, which creates an empty stack;
- `isEmpty`, which checks if the stack is empty;
- `push`, which inserts a new item on the top of the stack; and
- `pop`, which removes the item from the top of the stack.

We can represent a stack as an array. Here, the bottom of the stack is 'anchored' to one end of the array. All the operations are $O(1)$ in that case. We can also represent a stack as a linked list. Again, all the operations are $O(1)$.

Queue

A queue is an abstract data type that too holds a collection of objects. The objects are accessed using a first-in-first-out (FIFO) policy. The basic operations defined on a queue are:

- `create`, which creates an empty queue;
- `isEmpty`, which checks if the queue is empty;
- `insert`, which inserts a new item at the back of the queue; and
- `delete`, which removes the item at the front of the queue.

We can represent a queue as an array. We need to wrap the queue around the array so that all the operations are $O(1)$. We can also use a linked list to represent a queue. Again, all the operations are $O(1)$.

Priority Queue

A priority queue is an abstract data type that is very similar to a queue. Instead of holding the elements in the given order, each element has a priority. When removing an element, we remove the one with the highest priority first. We can achieve this by keeping the queue sorted at all times, for example.

The basic operations defined on a priority queue are the same as a queue: `create`, `isEmpty`, `insert` and `delete`.

If we represent a priority queue as an unordered list, then the operation `insert` is $O(1)$ while the operation `delete` is $O(n)$. On the other hand, if we represent it as an ordered list, then the operation `delete` is $O(1)$ while the operation `insert` is $O(n)$. If we store it as a heap, then both `insert` and `delete` are both $O(\log n)$. In all cases, the other operations `create` and `isEmpty` are $O(1)$.

1.2 Sorting algorithms

We have seen before that naive sorting algorithms such as selection sort, insertion sort and bubble sort are $O(n^2)$ in worst (or average) case. There are also clever sorting algorithms such as heap sort, merge sort and quick sort that are $O(n \log n)$ in worst (or average) case. In practice, the fastest sorting algorithm is quicksort, even though its worse case is $O(n^2)$. This is because, in practice, the worst case is not encountered often.

The best runtime of these algorithms is $O(n \log n)$. In fact, it is not possible for us to use a comparison-based sorting algorithm with runtime better than $O(n \log n)$.

Comparison-based sorting

The algorithms we have above involve comparing elements to establish the order of the sorted list. For this reason, they are called comparison-based sorting algorithms.

An execution of a comparison-based sorting algorithm can be represented as a decision tree. In the decision tree, we have a node that compares two elements- this leads to two branches depending on which of the node is bigger. An node from the tree is shown below.

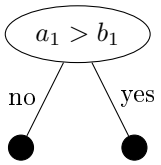

Figure 1.1: A node in a decision tree where we compare the elements a_1 and b_1 and proceed differently depending on which element is bigger.

In the decision tree, every node can be associated with a permutation of the list, each representing a different step in the sorting procedure (including those that do not sort the list). The leaf nodes represent a sorted list.

If we are given an unsorted list of size n, then any of the $n!$ permutations of the list could be the sorted version of the list. Therefore, we need at least $n!$ distinct leaves, each leading to a different permutation of the unsorted list. We could have more leaves, but in that case there will be a permutation associated with more than one leaf.

This implies that the worst-case complexity of a sorting-based algorithm is $O(h)$, where h is the height of the decision tree. An execution is a path from the root to the leaf node. Moreover, we need to perform an operation to go from one node to another, so we need to perform h operations in the worst-case.

If we have a binary tree of height h, the it has at most $2^{h+1} - 1$ leaf nodes. We can prove this by induction:

- If $h = 1$, then we have one node, which is a leaf. In that case, we have $n = 1 \leq 3 = 2^2 - 1 = 2^{h+1} - 1$.
- Now, assume that for some height h, every tree is of size $n_h \leq 2^{h+1} - 1$. Then, consider a tree of height $h + 1$. It has at most $2n_h$ more nodes

compared to a tree of height h. Therefore,

$$n_{h+1} \leq 2 \cdot n_h \leq 2 \cdot (2^{h+1} - 1) = 2^{h+2} - 2 \leq 2^{h+2} - 1.$$

So, by induction, we find that a binary tree of height h has at most $2^{h+1} - 1$ leaf nodes.

A decision tree is a binary tree (since every node has two branches- 'yes' or 'no'). Therefore, we find that

$$n! \leq 2^{h+1} - 1 \leq 2^{h+1}.$$

Taking $\log_2$ of both sides, we find that

$$\begin{aligned} h + 1 &\geq \log_2(n!) \\ &\geq \log_2(n/2)^{n/2} \\ &= (n/2)\log_2(n/2) \\ &= (n/2)\log_2 n - (n/2)\log_2 2 \\ &= (n/2)\log_2 n - n/2. \end{aligned}$$

This implies that h is $O(n \log n)$. Here, we use the fact that $n! \geq \log_2(n/2)^{n/2}$. We can see this as follows:

$$\begin{aligned} n! &= n \cdot (n-1) \cdots (n/2) \cdot (n/2 - 1) \cdots 1 \\ &\geq n \cdot (n-1) \cdots (n/2) \\ &\geq (n/2) \cdot (n/2) \cdots (n/2) \\ &= (n/2)^{n/2}. \end{aligned}$$

Radix Sorting

We have proven that no sorting algorithm based on comparison can be better than $O(n \log n)$ in the worst case. So, if we want to improve on this worst-case bound, we have to create a method based on something other than comparisons.

Radix sorting uses a different approach to achieve $O(n)$ complexity. However, the algorithm has to exploit the structure of the items being sorted- it is less versatile. Moreover, in practice, it is faster than $O(n \log n)$ algorithms only for very large n.

This algorithm breaks down a number into smaller chunks with fewer digits, sorts them in place and continues sorting a more significant chunk. We consider sorting integers, but by transforming them into binary.

So, assume that every integer in the list can be written as a bit-sequence of length m. Moreover, let b be a chosen factor of m. Here, m and b are constants for a particular instance. We can label each digit from 0 to $m - 1$, with bit 0 being the most significant/leftmost bit.

The algorithm iterates m/b times. In each iteration, it places an element in one of the 2^b lists (or buckets), and joins each list to give the list to be used in the next iteration. The buckets correspond to a value between 0 and $2^b - 1$. During the i-th iteration, the bits from position $b \cdot (i-1)$ to $b \cdot i - 1$ are used to find the corresponding bucket.

Since the buckets get concatenated starting from the bucket representing the smallest value and the sorting is in place, we end up with a sorted list after all the loop terminates.

Now, we apply the radix sorting algorithm to sort the list below.

15	43	5	27	60	18	26	2

First, we convert it into binary. The longest bit string is of length 6. So, we have $m = 6$. We choose $b = 2$. The list is shown below as bit strings.

00 11 11	10 10 11	00 01 01	00 10 11	11 11 00	01 00 10	01 10 10	00 00 10

In this case, we have $2^2 = 4$ buckets: 00, 01, 10 and 11. In the first iteration, we add these numbers into the corresponding buckets based on positions $2\cdot(1-1) = 0$ to $2 \cdot 1 - 1 = 1$.

00	60=111100		
01	5=000101		
10	18=010010	26=011010	2=000010
11	15=001111	43=101011	27=001011

We now concatenate the buckets to get the list we will use in the next iteration.

60	5	18	26	2	15	43	27

The binary representation of the list is now the following.

11 11 00	00 01 01	01 00 10	01 10 10	00 00 10	00 11 11	10 10 11	01 10 11

We are now in the second iteration. So, we are considering the positions $2 \cdot (2-1) = 2$ to $2 \cdot 2 - 1 = 3$, as highlighted in blue above. We place them in the four buckets depending on the value within the highlighted position, and that gives us the following list of buckets.

00	18=010010	2=00 00 10	
01	5=000101		
10	26=011010	43=101011	27=011011
11	60=111100	15=001111	

Note that the value 26 comes before 27 since 26 came before 27 in the list at the start of this iteration. This happens because we are sorting in place. Concatenating this, we get the following list.

18	2	5	26	43	27	60	15

The list in binary, with the highlighted positions 4 and 5, is shown below.

01 00 10	00 00 10	00 01 01	01 10 10	10 10 11	01 10 11	11 11 00	00 11 11

Now, we are in the final iteration. In total, there were $m/b = 6/2 = 3$ iterations. We add each element in the list into the bucket.

00	2=000010	5=000101	
01	18=010010	26=011010	27= 01 1011
10	43=101011		
11	60=111100		

We then concatenate the list.

2	5	15	18	26	27	43	60

This gives us the sorted list.

The pseudocode for radix sorting is given below.

```
List<int> radixSort(List<int> list, int m, int b):
    // the number of iterations
    int numIterations = m/b
    // the number of buckets
    int numBuckets = 2**b

    // initialise the buckets
    List<List<int>> buckets = List.generate(i => [])

    for int i in range(1, numIterations+1):
        // clear the buckets
        for List<int> bucket in buckets:
            buckets.clear()

        // add each number to the right bucket
        for int value in list:
            int k = value.toBitString()[b*(i-1):b*i].toInt(2)
            buckets[k].add(value)

        // concatentate the buckets at the end
        list = buckets.reduce(concatenate)
    return list
```

Next, we show that radix sorting is correct. That is, if we have x and y in a list with $x < y$, then we need to show that x precedes y in the final sequence. Suppose that j is the last iteration for which x and y differ (i.e. every bit after j is the same in x and y). Since $x < y$, the relevant bits of x must be smaller than those of y. Therefore, x goes to an 'earlier' bucket that y- after concatenation, x precedes y in the sequence after this iteration. Since any bit after this is the same in both x and y, they will both end up at the same bucket. Since the algorithm is in-place, x will still precede y at the end. Therefore, the final list will be sorted.

Finally, we consider the complexity of radix sort. The number of iterations is m/b and the number of buckets is 2^b. In each iteration, we scan the sequence and add them to one of the buckets- this takes $O(n)$ time. The buckets then get concatenated- this takes $O(2^b)$ time. So, the overall complexity is $O(m/b \cdot (n + 2^b))$. Since m and b are constants, this simplifies to $O(n)$.

When choosing the value b, there is a time-space trade-off. If we choose a large value of b, the multiplicative constant m/b will be small. So, the algorithm will become faster. But, we will need an array of size 2^b- this increases the space requirements exponentially.

1.3 Tries

A trie is used to store items that can be interpreted as a sequence of bits or characters. There is a multi-way branch at each node where each branch has an associated symbol and no two siblings have the same symbol. The branch taken at level i during a search is determined by the i-th element of the value (i-th bit, i-th character, etc.). Tracing a path from the root to a node spells out the value of the item.

A trie can be used to store items with string values, e.g. words in a dictionary. An example is of a trie is given below.

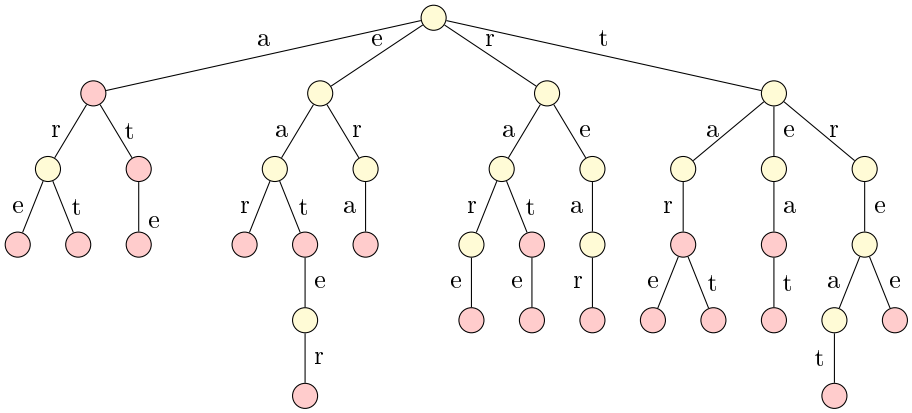

Figure 1.2: A Trie

We follow a path to spell out a word from left to right. A red node represents values that are words, while a yellow node represents an intermediate node. So, `tree` is a word but `trea` is not.

The search algorithm in a trie is given below:

```
bool search(Trie trie, String word):
    Node node = trie.root

    for char letter in word:
        // if the node has the required child,
        if node.hasChild(letter):
            // move to the next character
            node = node.getChild(letter)
        // otherwise, the word isn't present
        else:
            return false

    // for the trie to have a word, it actually needs to be a word
    return child.isWord
```

The insertion algorithm in a trie is given below:

```
void insert(Trie trie, String word):
    Node node = trie.root
    for char letter in word:
        // if the node has the required child,
        if node.hasChild(letter):
            // move to the next character
            node = node.getChild(letter)
```

```
        else:
            // create a node for that character,
            Node child = Node(letter)
            // attach it to the node,
            node.addChild(child)
            // and move to the next character
            node = child

    // the final character represents a word
    node.isWord = true
```

The deletion algorithm in a trie is given below:

```
bool delete(Trie trie, String word):
    Node node = trie.root

    for char letter in word:
        // if the node has the required child,
        if node.hasChild(letter):
            // move to the next character
            node = node.getChild(letter)
        // otherwise, the word isn't present
        else:
            return false

    // we can only delete a word
    if not node.isWord:
        return false

    // the node no longer represents a word
    node.isWord = false

    // while a node can be deleted,
    while node != trie.root and node.hasChild and not node.isWord:
        // delete it
        node.remove()
        node = node.parent

    // the word was present in the trie
    return true
```

Most trie operations are independent of the number of items present. They are essentially linear in the length of the string. There are various possible implementations of a trie, e.g. arrays or linked lists. In a linked list implementation, a node stores the character it represents, along with pointers to the next child of the parent node (called its sibling) and the (first) child of this node. The code can be optimised by sorting the children nodes using the lexicographic order of the character.