13

String Matching Algorithms

There are many popular string matching algorithms. String matching algorithms have very important applications, such as searching for an element in a text document, plagiarism detection, text editing programs, and so on. In this chapter, we will study the pattern matching algorithms that find the locations of a given pattern or substring in any given text. We will discuss the **brute force algorithm**, along with the **Rabin-Karp**, **Knuth-Morris-Pratt (KMP)**, and **Boyer-Moore pattern matching algorithms**. This chapter aims to discuss algorithms that are related to strings. The following topics will be covered in this chapter:

- Learning pattern matching algorithms and their implementation
- Understanding and implementing the **Rabin-Karp pattern matching algorithm**
- Understanding and implementing the **Knuth-Morris-Pratt (KMP) algorithm**
- Understanding and implementing the **Boyer-Moore pattern matching algorithm**

Technical requirements

All of the programs based on the concepts and algorithms discussed in this chapter are provided in the book as well as in the GitHub repository at the following link: `https://github.com/PacktPublishing/Hands-On-Data-Structures-and-Algorithms-with-Python-Third-Edition/tree/main/Chapter13`.

String notations and concepts

Strings are sequences of characters. Python provides a rich set of operations and functions that can be applied to the string data type. Strings are textual data and are handled very efficiently in Python. The following is an example of a string `(s)`—`"packt publishing"`.

A substring is a sequence of characters that's part of the given string, i.e., specified indices in the string in a continuous order. For example, "`packt`" is a substring of the string "`packt publishing`". On the other hand, a subsequence is also a sequence of characters that can be obtained from the given string by removing some of the characters from the string by keeping the order of occurrence of the characters. For example, "`pct pblishing`" is a valid subsequence for the string "`packt publishing`" that is obtained by removing the characters `a`, `k`, and `u`. However, this is not a substring since "`pct pblishing`" is not a continuous sequence of characters. Hence, a subsequence is different from a substring, and it can be considered a generalization of substrings.

The prefix (`p`) is a substring of the string (`s`) in that it is present at the start of the string. There is also another string (`u`) that exists in the string (`s`) after the prefix. For example, the substring "`pack`" is a prefix for the string `(s) = "packt publishing"` as it is the starting substring and there is another substring `u = "publishing"` after it. Thus, the prefix plus string (`u`) makes "`packt publishing`", which is the whole string.

The suffix (`d`) is a substring that is present at the end of the string (`s`). For example, the substring "`shing`" is one of the many possible suffixes for the string "`packt publishing`". Python has built-in functions to check whether a string starts or ends with a specific string, as shown in the following code snippet:

```
string = "this is data structures book by packt publisher"
suffix = "publisher"
prefix = "this"
print(string.endswith(suffix))  #Check if string contains given suffix.
print(string.startswith(prefix)) #Check if string starts with given
prefix.
```

The output of the above code is as follows:

```
True
True
```

In the above example of the given string, we can see that the given text string ends with another substring "`publisher`", which is a valid suffix, and that also has another substring "`this`", which is a substring of the string start and is also a valid prefix.

Note that the pattern matching algorithms discussed here are not to be confused with the matching statements of Python 3.10.

Pattern matching algorithms are the most important string processing algorithms and we will discuss them in the subsequent sections, starting with pattern matching algorithms.

Pattern matching algorithms

A pattern matching algorithm is used to determine the index positions where a given pattern string (`P`) is matched in a text string (`T`). Thus, the pattern matching algorithm finds and returns the index where a given string pattern appears in a text string. It returns `"pattern not found"` if the pattern does not have a match in the text string.

For example, for the given text string `(s) = "packt publisher"` and the pattern string `(p) = "publisher"`, the pattern-matching algorithm returns the index position where the pattern string is matched in the text string. An example of a string matching problem is shown in *Figure 13.1*:

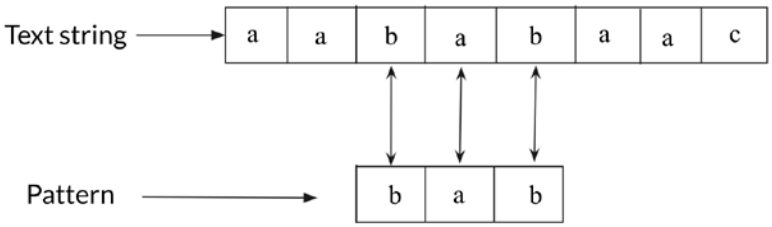


Figure 13.1: An example of a string matching problem

We will discuss four pattern matching algorithms, that is, the brute force method, Rabin-Karp algorithm, and the **Knuth-Morris-Pratt** (**KMP**) and Boyer-Moore pattern-matching algorithms. We start with the brute force pattern matching algorithm.

The brute force algorithm

The brute force algorithm is also called the naive approach to pattern matching algorithms. Naive approach means that it is a very basic and simple algorithm. In this approach, we match all the possible combinations of the input pattern in the given text string to find the position of the occurrence of the pattern. This algorithm is very naive and is not suitable if the text is very long.

In this algorithm, we start by comparing the characters of the pattern string and the text string one by one, and if all the characters of the pattern are matched with the text, we return the index position of the text where the first character of the pattern is located. If any character of the pattern is mismatched with the text string, we shift the pattern by one position to check if the pattern appears at the next index position. We continue comparing the pattern and text string by shifting the pattern by one index position.

To better understand how the brute force algorithm works, let's look at an example. Suppose we have a text string (`T`) = "**acbcabccababcaacbcac**", and the pattern string (`P`) is "**acbcac**". Now, the objective of the pattern matching algorithm is to determine the index position of the pattern string in the given text, `T`, as shown in *Figure 13.2*:

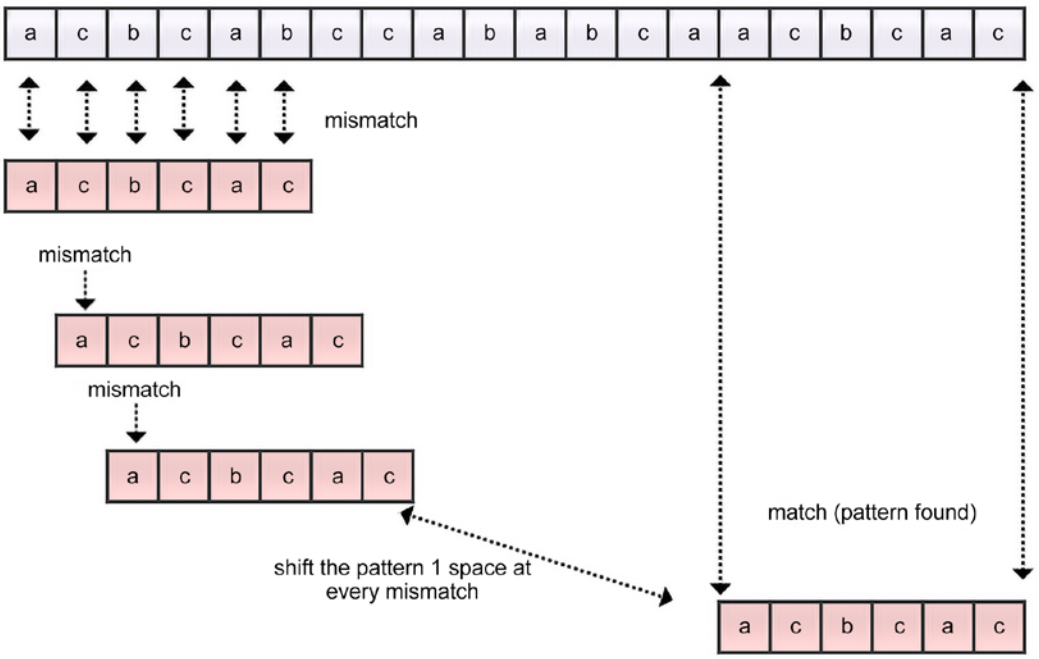


Figure 13.2: An example of the brute force algorithm for string matching

We start by comparing the first character of the text, that is, **a**, and the first character of the pattern. Here, the initial five characters of the pattern are matched, and then there is a mismatch in the last character of the pattern. This is a mismatch, so we shift the pattern by one place. We again start comparing the first character of the pattern and the second character of the text string one by one. Here, character **c** of the text string does not match with the character **a** of the pattern. So, this is also a mismatch, and we shift the pattern by one space, as shown in *Figure 13.2*. We continue comparing the characters of the pattern and the text string until we traverse the whole text string. In this example, we find a match at index position 14, which is shown in *Figure 13.2*.

Let's consider the Python implementation of the brute force algorithm for pattern matching:

```
def brute_force(text, pattern):
    l1 = len(text)      # The length of the text string
```

```
    l2 = len(pattern)   # The length of the pattern
    i = 0
    j = 0               # looping variables are set to 0
    flag = False        # If the pattern doesn't appear at all, then set
this to false and execute the last if statement
    while i < l1:         # iterating from the 0th index of text
        j = 0
        count = 0
        # Count stores the length upto which the pattern and the text have
matched
        while j < l2:
            if i+j < l1 and text[i+j] == pattern[j]:
        # statement to check if a match has occurred or not
                count += 1     # Count is incremented if a character is
matched
            j += 1
        if count == l2:   # it shows a matching of pattern in the text
                print("\nPattern occurs at index", i)
                # print the starting index of the successful match
                flag = True
                # flag is True as we wish to continue looking for more
matching of pattern in the text.
        i += 1
    if not flag:
        # If the pattern doesn't occur at all, means no match of pattern
in the text string
        print('\nPattern is not at all present in the array')
```

The following code snippet can be used to call the function to search the pattern `'acbcac'` in the given string:

```
brute_force('acbcabccababcaacbcac','acbcac')         # function call
```

The output of the above function call is as follows:

```
Pattern occurs at index 14
```

In the preceding code for the brute force approach, we start by computing the length of the given text strings and pattern. We also initialize the looping variables with `0` and set the flag to `False`.

This variable is used to continue searching for a match of the pattern in the string. If the `flag` variable is `False` by the end of the text string, it means that there is no match for the pattern at all in the text string.

Next, we start the searching loop from the 0^{th} index to the end of the text string. In this loop, we have a `count` variable that is used to keep track of the length up to which the pattern and the text have been matched. Next, we have another nested loop that runs from the 0^{th} index to the length of the pattern. Here, the variable `i` keeps track of the index position in the text string and the variable `j` keeps track of the characters in the pattern. Next, we compare the characters of the patterns and the text string using the following code fragment:

```
if i+j<l1 and text[i+j] == pattern[j]:
```

Furthermore, we increment the `count` variable after every match of the character of the pattern in the text string. Then, we continue matching the characters of the pattern and text string. If the length of the pattern becomes equal to the `count` variable, it means there is a match.

We print the index position of the text string if there is a match for the pattern string in the text string and keep the `flag` variable as to `True` as we wish to continue searching for more matches of the patterns in the text string. Finally, if the value of the variable `flag` is `False`, it means that there was not a match for the pattern in the text string at all.

The best-case and worst-case time complexities for the naive string matching algorithms are `O(n)` and `O(m*(n-m+1))`, respectively. The best-case scenario occurs when the pattern is not found in the text and the first character of the pattern is not present in the text at all, for example, if the text string is `ABAACEBCCDAAEE`, and the pattern is `FAA`. Here, as the first character of the pattern will not find a match anywhere in the text, it will have comparisons equal to the length of the text (`n`).

The worst-case scenario occurs when all characters of the text string and the pattern are the same and we want to find out all the occurrences of the given pattern string in the text string, for example, if the text string is `AAAAAAAAAAAAAAA`, and the pattern string is `AAAA`. Another worst-case scenario occurs when only the last character is different, for example, if the text string is `AAAAAAAAAAAAAAF` and the pattern is `AAAAF`. Thus, the total number of comparisons will be `m*(n-m+1)` and the worst-case time complexity will be `O(m*(n-m+1))`.

Next, we discuss the Rabin-Karp pattern matching algorithm.

The Rabin-Karp algorithm

The Rabin-Karp pattern matching algorithm is an improved version of the brute force approach to find the location of the given pattern in the text string. The performance of the Rabin-Karp algorithm is improved by reducing the number of comparisons with the help of hashing. We discussed the concept of hashing in *Chapter 8*, *Hash Tables*. The hashing function returns a unique numeric value for a given string.

This algorithm is faster than the brute force approach as it avoids unnecessary comparisons. In this algorithm, we compare the hash value of the pattern with the hash value of the substring of the text string. If the hash values are not matched, the pattern is shifted forward one position. This is a better algorithm as compared to the brute-force algorithm since there is no need to compare all the characters of the pattern one by one.

This algorithm is based on the concept that if the hash values of the two strings are equal, then it is assumed that both the strings are also equal. However, it is also possible that there can be two different strings whose hash values are equal. In that case, the algorithm will not work; this situation is known as a spurious hit and happens due to a collision in hashing. To avoid this with the Rabin-Karp algorithm, after matching the hash values of the pattern and the substring, we ensure that the pattern is actually matched in the string by comparing the pattern and the substring character by character.

The Rabin-Karp pattern matching algorithm works as follows:

1. First, we preprocess the pattern before starting the search, that is, we compute the hash value of the pattern of length `m` and the hash values of all the possible substrings of the text of length `m`. The total number of possible substrings would be (`n-m+1`). Here, `n` is the length of the text.
2. We compare the hash value of the pattern with the hash value of the substrings of the text one by one.
3. If the hash values are not matched, then we shift the pattern by one position.
4. If the hash value of the pattern and the hash value of the substring of the text match, then we compare the pattern and substring character by character to ensure that the pattern is actually matched in the text.
5. We continue the process of *steps 2-5* until we reach the end of the given text string.

In this algorithm, we compute the numerical hash values using Horner's rule (any other hashing function can also be used) that returns a unique value for the given string. We also compute the hash value using the sum of the ordinal values of all the characters of the string.

Let's consider an example to understand the **Rabin-Karp algorithm**. Let's say we have a text string `(T) = "publisher paakt packt"`, and the pattern `(P) = "packt"`. First, we compute the hash values of the pattern (length `m`) and all the substrings (of length `m`) of the text string. The functionality of the **Rabin-Karp algorithm** is shown in *Figure 13.3*:

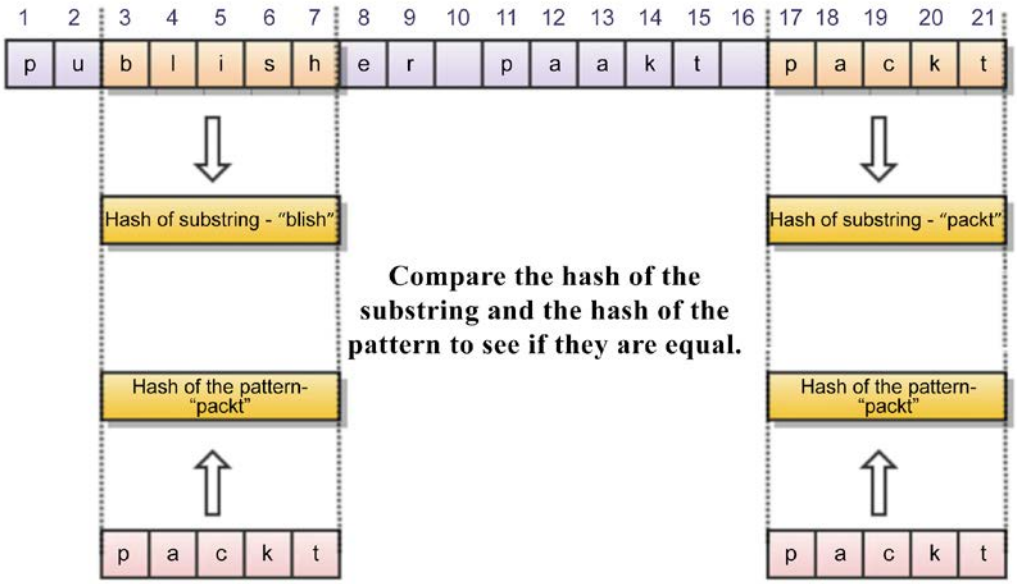


Figure 13.3: An example of the Rabin-Karp algorithm for string matching

We start comparing the hash value of the pattern `"packt"` with the first substring `"publi"`. Since the hash values do not match, we shift the pattern by one position, and then we compare the hash value of pattern with the hash value of the next substring of the text, i.e. `"ublis"`. As these hash values also do not match, we again shift the pattern by one position. We shift the pattern by one position at a time if the hash values do not match. And, if the hash value of the pattern and the hash value of the substring match, we compare the pattern and substring character by character and we return the location of the text string if they match.

In the example shown in *Figure 13.3*, hash values of the pattern and the substring of the text are matched at location 17.

It is important to note that there can be a different string whose hash value can match with the hash of the pattern, i.e. a spurious hit.

Next, let us discuss the implementation of the **Rabin-Karp pattern matching algorithm**.

Implementing the Rabin-Karp algorithm

The implementation of the **Rabin-Karp algorithm** is done in two steps:

1. We implement the `generate_hash()` method, which is used to compute the hash value of the pattern and all the possible combinations of the substrings of length equal to the length of the pattern.
2. We implement the **Rabin-Karp algorithm**, which uses the `generate_hash()` method to identify the substring whose hash value matches the hash value of the pattern. Finally, we match them character by character to ensure we have correctly found the pattern.

Let us first discuss the implementation of generating hash values for the patterns and substrings of the text. For this, we need to first decide on the hash function. Here, we use the sum of all the ordinal values of all the characters of the string as the hashing function.

The complete Python implementation to compute the hashing values is given below:

```
def generate_hash(text, pattern):
    ord_text = [ord(i) for i in text]       # stores unicode value of each
character in text
    ord_pattern = [ord(j) for j in pattern] # stores unicode value of each
character in pattern
    len_text = len(text)                    # stores length of the text
    len_pattern = len(pattern)              # stores length of the pattern
    len_hash_array = len_text - len_pattern + 1 # stores the length of new
array that will contain the hash values of text
    hash_text = [0]*(len_hash_array)        # Initialize all the values in
the array to 0.
    hash_pattern = sum(ord_pattern)
    for i in range(0,len_hash_array):       # step size of the loop will
be the size of the pattern
        if i == 0:                          # Base condition
            hash_text[i] = sum(ord_text[:len_pattern])   # initial value
of hash function
        else:
            hash_text[i] = ((hash_text[i-1] - ord_text[i-1]) + ord
```

```
[i+len_pattern-1])   # calculating next hash value using previous value
    return [hash_text, hash_pattern]                # return the hash
values
```

In the above code, we start by storing the ordinal values of all the characters of the text and the pattern in the `ord_text` and `ord_pattern` variables. Next, we store the length of the text and the pattern in the `len_text` and `len_pattern` variables.

Next, we create a variable called `len_hash_array` that stores the number of all the possible substrings of length (equal to the length of the pattern) using `len_text - len_pattern + 1`, and we create an array called `hash_text` that stores the hash value for all the possible substrings. This is shown in the following code snippet:

```
    len_hash_array = len_text - len_pattern + 1
    hash_text = [0]*(len_hash_array)
```

Next, we compute the hash value for the pattern by summing up the ordinal values of all the characters in the pattern using the following code snippet:

```
    hash_pattern = sum(ord_pattern)
```

Next, we start a loop that executes for all the possible substrings of the text. For this, initially, we compute the hash value for the first substring by summing the ordinal values of all of its characters using `sum(ord_text[:len_pattern])`. Further, the hash values for all of the substrings are computed using the hash value of the previous substrings as shown in the following code snippet:

```
    hash_text[i] = ((hash_text[i-1] - ord_text[i-1]) + ord_text[i+len_
pattern-1])
```

So, we have precomputed the hash values for the pattern and all the substrings of the text that we will use for comparing the pattern and the text in the implementation of the **Rabin-Karp algorithm**. The **Rabin-Karp algorithm** works as follows. Firstly, we compare the hash values of the pattern and substrings of the text. Next, we take the substring for which the hash matches with the hash of the pattern and compare them both character by character.

The complete Python implementation of the **Rabin-Karp algorithm** is as follows:

```
def Rabin_Karp_Matcher(text, pattern):
    text = str(text)                            # convert text into string
format
    pattern = str(pattern)                  # convert pattern into string
format
```

```
    hash_text, hash_pattern = generate_hash(text, pattern) # generate hash
values using generate_hash function
    len_text = len(text)          # length of text
    len_pattern = len(pattern)    # length of pattern
    flag = False                  # checks if pattern is present atleast
once or not at all
    for i in range(len(hash_text)):
        if hash_text[i] == hash_pattern:   # if the hash value matches
            count = 0                      # count the total characters
upto which both are similar
            for j in range(len_pattern):
                if pattern[j] == text[i+j]: # checking equality for each
character
                    count += 1              # if value is equal, then
update the count value
                else:
                    break
            if count == len_pattern:        # if count is equal to length
of pattern, it means there is a match
                    flag = True             # update flag accordingly
                    print('Pattern occurs at index',i)
    if not flag:                            # if pattern doesn't match
even once, then this if statement is executed
        print('Pattern is not at all present in the text')
```

In the above code, firstly, we convert the given text and pattern into string format as the ordinal values can only be computed for strings. Next, we use the `generate_hash` function to compute the hash values of patterns and texts. We store the length of the text and patterns in the `len_text` and `len_pattern` variables. We also initialize the `flag` variable to `False` so that it keeps track of whether the pattern is present in the text at least once.

Next, we start a loop that implements the main concept of the algorithm. This loop executes for the length of `hash_text`, which is the total number of possible substrings. Initially, we compare the hash value of the first substring with the hash of the pattern by using `if hash_text[i] == hash_pattern`. If they do not match; we move one index position and look for another substring. We iteratively move further until we get a match.

If we find a match, we compare the substring and the pattern character by character through a loop by using `if pattern[j] == text[i+j]`.

We then create a `count` variable to keep track of how many characters match in the pattern and the substring. If the length of the count and the length of the pattern are equal, this means that all of the characters match, and the index location where the pattern was found is returned. Finally, if the `flag` variable remains `False`, this means that the pattern does not match at all with the text. The following code snippets can be used to execute the **Rabin-Karp matching algorithm**:

```
Rabin_Karp_Matcher("10111000001110100101011101","1011")
Rabin_Karp_Matcher("ABBACCADABBACCEDF","ACCE")
```

The output of the above code is as follows:

```
Pattern occurs at index 0
Pattern occurs at index 18
Pattern occurs at index 11
```

In the above code, we first check whether the pattern "`1011`" appears in the given text string "`10111000001110100101011101`". The output shows that the given pattern occurs at index position `0` and `18`. Next, the pattern "`ACCE`" occurs at index position `11` in the text string "`ABBACCADABBACCEDF`".

The Rabin-Karp pattern matching algorithm preprocesses the pattern before the searching; that is, it computes the hash value for the pattern that has the complexity of `O(m)`. Also, the worst-case running time complexity of the Rabin-Karp algorithm is `O(m *(n-m+1))`. The worst-case scenario is when the pattern does not occur in the text at all. The average-case scenario is when the pattern occurs at least once.

Next, we will discuss the KMP string matching algorithm.

The Knuth-Morris-Pratt algorithm

The KMP algorithm is a pattern matching algorithm based on the idea that the overlapping text in the pattern itself can be used to immediately know at the time of any mismatch how much the pattern should be shifted to skip unnecessary comparisons. In this algorithm, we will precompute the `prefix` function that indicates the required number of shifts of the pattern whenever we get a mismatch. The KMP algorithm preprocesses the pattern to avoid unnecessary comparisons using the `prefix` function. So, the algorithm utilizes the `prefix` function to estimate how much the pattern should be shifted to search the pattern in the text string whenever we get a mismatch. The **KMP algorithm** is efficient as it minimizes the number of comparisons of the given patterns with respect to the text string.

The motivation behind the **KMP algorithm** can be observed in *Figure 13.4*. In this example, it can be seen that the mismatch occurred at the 6th position with the last character "d" after matching the initial 5 characters. It is also known from the `prefix` function that the character "d" did not appear before in the pattern, and utilizing this information, the pattern can be shifted by six places:

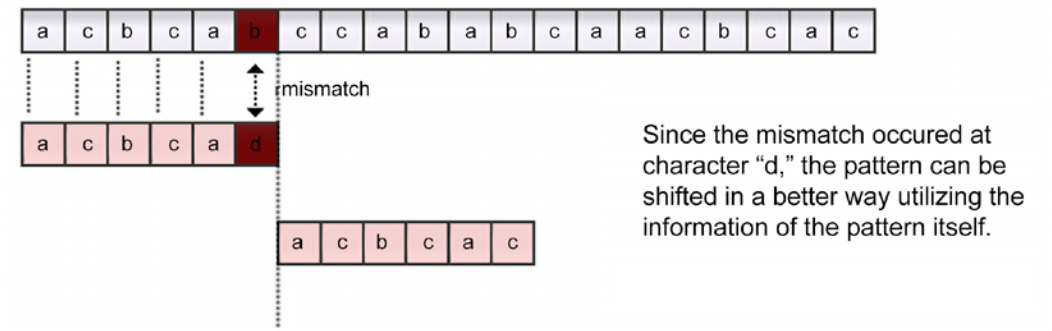


Figure 13.4: Example of the KMP algorithm

So, in this example, the pattern has shifted six positions instead of one. Let us discuss another example to understand the concept of the **KMP algorithm**, as shown in *Figure 13.5*:

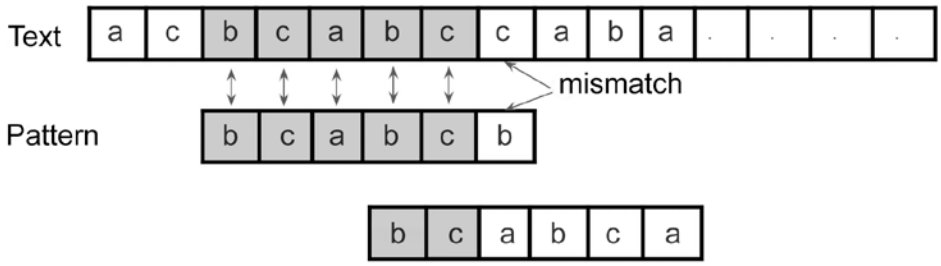


Figure 13.5: Second example of the KMP algorithm

In the above example, the mismatch occurs at the last character of the pattern. Since the pattern at the location of the mismatch has a partial match of the prefix **bc**, this information is given by the `prefix` function. Here, the pattern can be shifted to align with the other occurrence of the matched prefix **bc** in the pattern.

We will look into the `prefix` function next for a better understanding of how we use it to know by how much we should shift the pattern.

The prefix function

The `prefix` function (also known as the failure function) finds the pattern within the pattern. It finds out how much the previous comparisons can be reused due to repetition in the pattern itself when there is a mismatch. The `prefix` function returns a value for each position wherever we get a mismatch, which tells us by how much the pattern should be shifted.

Let us understand how we use the `prefix` function to find the required shift amount with the following examples. Consider the first example: if we had a `prefix` function for a pattern where all of the characters are different, the `prefix` function would have a value of `0`. This means that if we find any mismatch, the pattern will be shifted by the number of characters compared up to that position in the pattern.

Consider an example with the pattern **abcde**, which contains all different characters. We start comparing the first character of the pattern with the first character of the text string, as shown in *Figure 13.6*. As shown in the figure, the mismatch occurs at the 4th character in the pattern. Since the prefix function has the value 0, it means that there is no overlap in the pattern and no previous comparisons would be reused, so the pattern will be shifted to the number of characters compared up until that point:

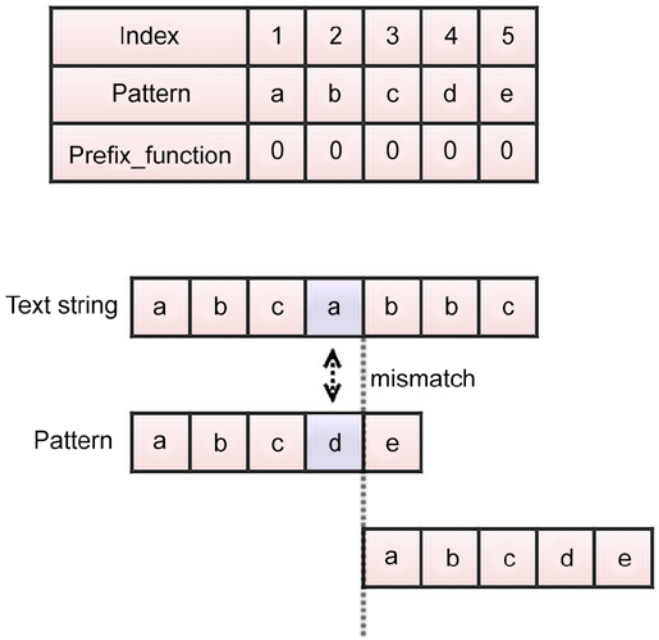


Figure 13.6: Prefix function in the KMP algorithm

Let's consider another example to better understand how the `prefix` function works for the pattern (`P`) **abcabbcab** as shown in *Figure 13.7*:

Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0						

Figure 13.7: Example of the prefix function in the KMP algorithm

In *Figure 13.7*, we start calculating the values of the `prefix` function starting from index 1. We assign the value **0** if there is no repetition of the characters in the pattern. So, in this example, we assign **0** to the `prefix` function for index positions 1 to 3. Next, at index position 4, we can see that there is a character, **a**, which is a repetition of the first character of the pattern itself, so we assign the value **1** here, as shown in *Figure 13.8*:

Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0	1					

Figure 13.8: Value of the prefix function at index 4 in the KMP algorithm

Next, we look at the next character at position 5. It has the longest suffix pattern, **ab**, and so it would have a value of **2**, as shown in *Figure 13.9*:

Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0	1	2				

Figure 13.9: Value of the prefix function at index 5 in the KMP algorithm

Similarly, we look at the next index position of 6. Here, the character is **b**. This character does not have the longest suffix in the pattern, so it has the value **0**. Next, we assign value **0** at index position 7. Then, we look at the index position 8, and we assign the value **1** as it has the longest suffix of length **1**.

Finally, at the index position of 9, we have the longest suffix of **2**. This is shown in *Figure 13.10*:

Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0	1	2	0	0	1	2

Figure 13.10: Value of the prefix function at index 6 to 9 in the KMP algorithm

The value of the `prefix` function shows how much of the start of the string can be reused if there is a mismatch. For example, if the comparison fails at index position `5`, the `prefix` function value is **2**, which means that the two starting characters don't need to be compared, and the pattern can be shifted accordingly.

Next, we discuss the details of the **KMP algorithm**.

Understanding the KMP algorithm

The **KMP pattern matching algorithm** detects overlaps in the pattern itself so that it avoids unnecessary comparisons. The main idea behind the **KMP algorithm** is to detect how much the pattern should be shifted, based on the overlaps in the patterns. The algorithm works as follows:

1. First, we precompute the `prefix` function for the given pattern and initialize a counter **q** that represents the number of characters that matched.
2. We start by comparing the first character of the pattern with the first character of the text string, and if this matches, then we increment the counter **q** for the pattern and the counter for the text string, and we compare the next character.
3. If there is a mismatch, then we assign the value of the precomputed `prefix` function for `q` to the index value of `q`.
4. We continue searching the pattern in the text string until we reach the end of the text, that is, if we do not find any matches. If all of the characters in the pattern are matched in the text string, we return the position where the pattern is matched in the text and continue to search for another match.

Let's consider the following example to understand the working of the **KMP algorithm**. We have a pattern `acacac` along with index positions from `1` to `6` (just for simplicity, we have index positions starting from 1 instead of 0), shown in *Figure 13.11*. The `prefix` function for the given pattern is constructed as shown in *Figure 13.11*:

Index	1	2	3	4	5	6
Pattern	a	c	a	c	a	c
Prefix_function	0	0	1	2	1	2

Figure 13.11: The prefix function for pattern "acacac"

Let us take an example to understand how we use the `prefix` function to shift the pattern according to the **KMP algorithm** for the text string and pattern given in *Figure 13.12*. We start comparing the pattern and the text character by character. When we mismatch at index position 6, we see the prefix value for this position is **2**. Then we shift the pattern according to the return value of the `prefix` function. Next, we start comparing the pattern and text string from the index position of 2 on the pattern (character **c**), and the character **b** of the text string. Since this is a mismatch, the pattern will be shifted according to the value of the `prefix` function at this position. This description is depicted in *Figure 13.12*:

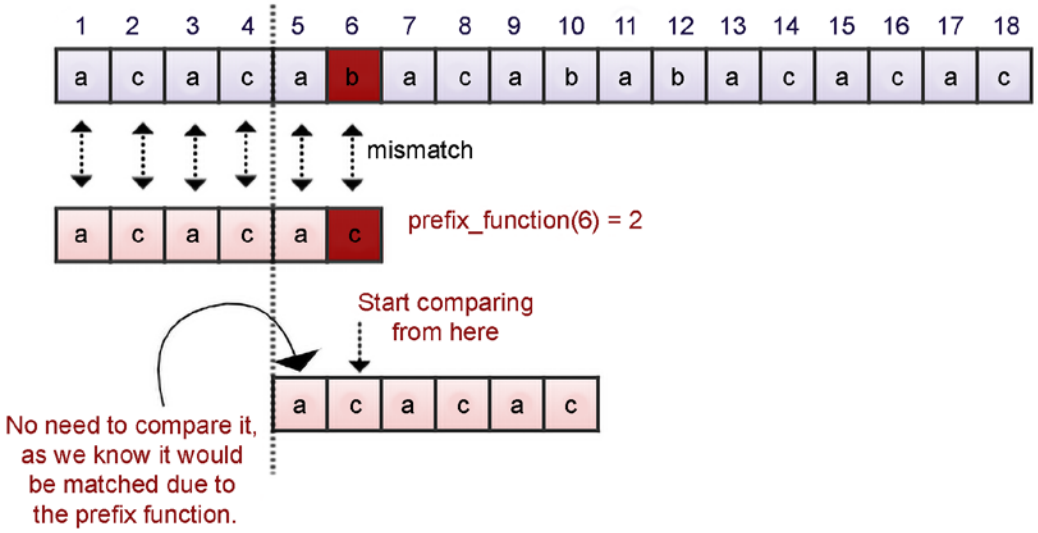


Figure 13.12: The pattern is shifted according to the return value of the prefix function

Now let's take another example shown in *Figure 13.13* where the position of the pattern over the text is shown. When we start comparing the characters **b** and **a**, these do not match, and we see the `prefix` function for index position `1` shows a value of **0**, meaning no overlapping of text in the pattern has occurred. Therefore, we shift the pattern by **1** place as shown in *Figure 13.12*. Next, we compare the pattern and text string character by character, and we find a mismatch at index

position 10 in the text between characters **b** and **c**.

Here, we use the precomputed `prefix` function to shift the pattern – as the `prefix_function(4)` is **2**, we shift the pattern to align over the text at index position 2 of the pattern. After that, we compare characters **b** and **c** at index position `10`, and since they do not match, we shift the pattern by one place. This process is shown in *Figure 13.13*:

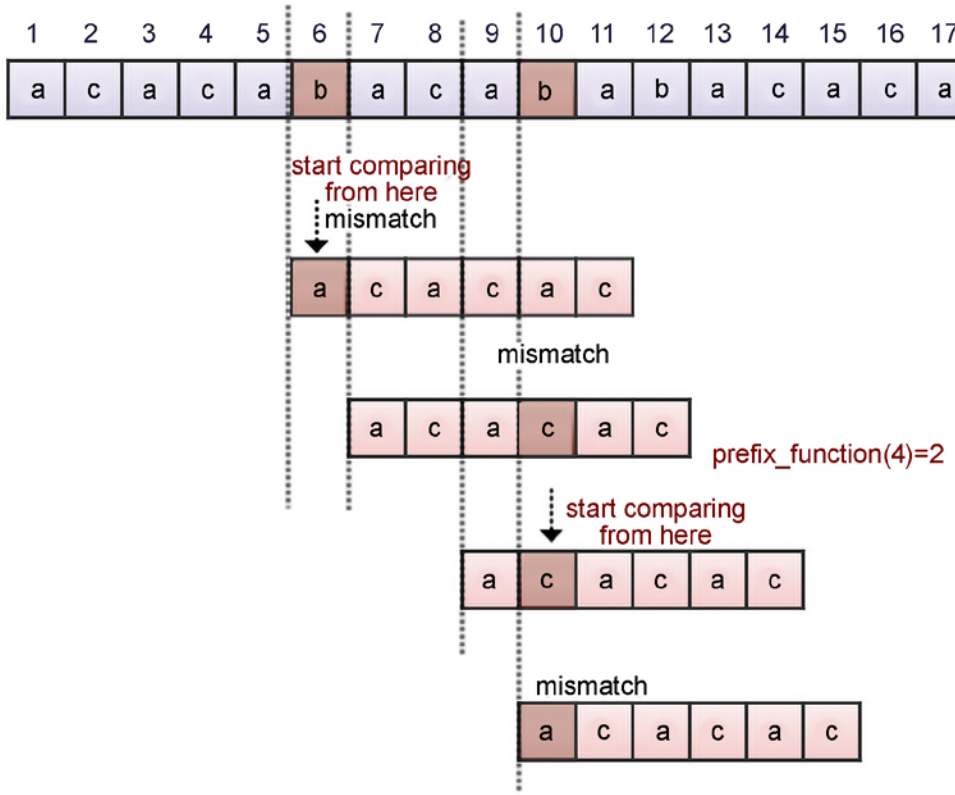


Figure 13.13: Shifting of the pattern according to the return value of the prefix function

Let us continue our searching from index position `11`, as shown in *Figure 13.14*. Next, we compare the characters at index `11` in the text and continue until a mismatch is found. We find a mismatch between characters **b** and **c** at index position `12`, as shown in *Figure 13.14*. We shift the pattern and move it next to the mismatched character since the `prefix_function(2)` is `0`. We repeat the same process until we reach the end of the string. We find a match of the pattern in the text string at index location **13** in the text string, as in *Figure 13.14*:

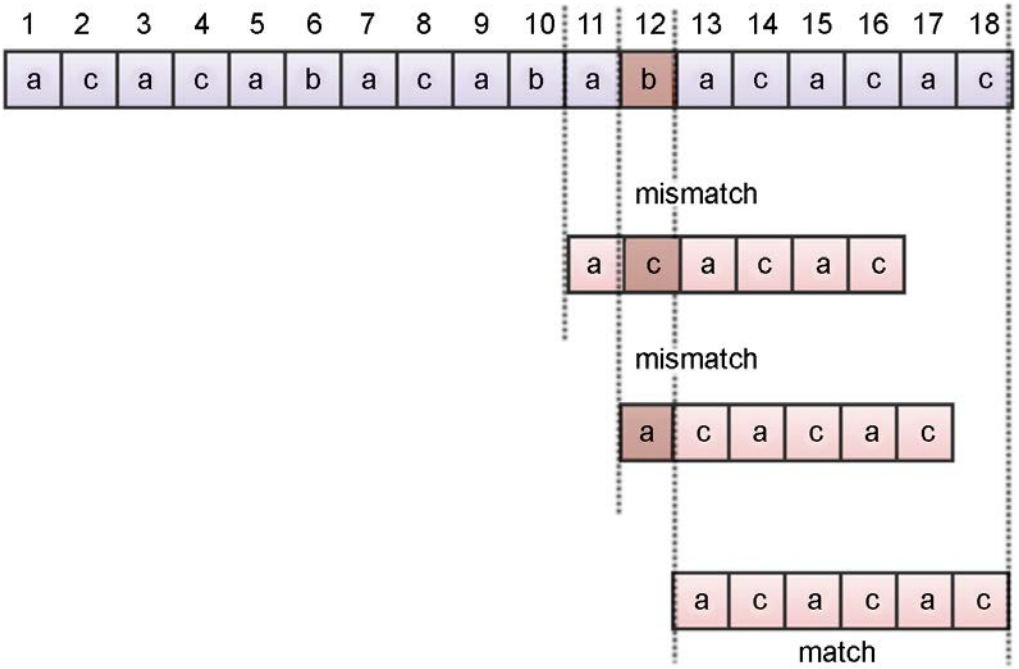


Figure 13.14: Shifting of the pattern for index positions of 11 to 18

The **KMP algorithm** has two phases: first, the preprocessing phase, which is where we compute the `prefix` function, which has the space and time complexity of `O(m)`. Further, the second phase involves searching, for which the **KMP algorithm** has a time complexity of `O(n)`. So, the worst-case time complexity of the **KMP algorithm** is `O(m +n)`.

Now, we will discuss the implementation of the **KMP algorithm** using Python.

Implementing the KMP algorithm

The Python implementation of the KMP algorithm is explained here. We start by implementing the `prefix` function for the given pattern. The code for the `prefix` function is as follows:

```
def pfun(pattern):                  # function to generate prefix function for
the given pattern,
    n = len(pattern)                # length of the pattern
    prefix_fun = [0]*(n)            # initialize all elements of the list to 0
    k = 0
```

```
    for q in range(2,n):
        while k>0 and pattern[k+1] != pattern[q]:
            k = prefix_fun[k]
        if pattern[k+1] == pattern[q]:     # If the kth element of the
pattern is equal to the qth element
            k += 1                         # update k accordingly
        prefix_fun[q] = k
    return prefix_fun           # return the prefix function
```

In the above code, we first compute the length of the pattern using the `len()` function, and then we initialize a list to store the values computed by the `prefix` function.

Next, we start the loop that executes from 2 to the length of the pattern. Then, we have a nested loop that is executed until we have processed the whole pattern. The variable `k` is initialized to `0`, which is the `prefix` function for the first element of the pattern. If the k^{th} element of the pattern is equal to the q^{th} element, then we increment the value of `k` by `1`. The value of `k` is the value computed by the `prefix` function, and so we assign it at the index position of `q` in the pattern. Finally, we return the list of the `prefix` function that has the computed value for each character of the pattern.

Once we have created the `prefix` function, we implement the main **KMP matching algorithm**. The following code shows this in detail:

```
def KMP_Matcher(text,pattern):   # KMP matcher function
    m = len(text)
    n = len(pattern)
    flag = False
    text = '-' + text   # append dummy character to make it 1-based
indexing
    pattern = '-' + pattern # append dummy character to the pattern also
    prefix_fun = pfun(pattern) # generate prefix function for the pattern
    q = 0
    for i in range(1,m+1):
        while q>0 and pattern[q+1] != text[i]: # while pattern and text
are not equal, decrement the value of q if it is > 0
            q = prefix_fun[q]
        if pattern[q+1] == text[i]:                # if pattern and text
are equal, update value of q
            q += 1
        if q == n:                                      # if q is equal to
the length of the pattern, it means that the pattern has been found.
```

```
            print("Pattern occurs at positions ",i-n)     # print the
index, where first match occurs.
            flag = True
            q = prefix_fun[q]
    if not flag:
        print('\nNo match found')
```

In the above code, we start by computing the length of the text string and the pattern, which are stored in the variables `m` and `n`, respectively. Next, we define a variable `flag` to indicate whether the pattern has found a match or not. Further, we add a dummy character - in the text and pattern to make the indexing start from index `1` instead of index `0`. Next, we call the `pfun()` method to construct the array containing the prefix values for all the positions of the pattern using `prefix_fun = pfun(pattern)`. Next, we execute a loop starting from `1` to `m+1`, where `m` is the length of the pattern. Further, for each iteration of the `for` loop, we compare the pattern and text in a `while` loop until we finish searching the pattern.

If we get a mismatch, we use the value of the `prefix` function at index `q` (here, `q` is the index where the mismatch occurs) to find out by how much we have to shift the pattern. If the pattern and text are equal, then the value of `1` and `n` will be equal, and we can return the index where the pattern was matched in the text. Further, we update the `flag` variable to `True` when the pattern is found in the text. If we finished searching the whole text string and still the variable `flag` was `False`, it would mean the pattern was not present in the given text.

The following code snippet can be used to execute the KMP algorithm for string matching:

```
    KMP_Matcher('aabaacaadaabaaba','aabaa')   # function call, with two
parameters, text and pattern
```

The output of the above code is as follows:

```
    Pattern occurs at positions 0
    Pattern occurs at positions 9
```

In the above output, we see that the pattern is present at index positions `0` and `9` in the given text string.

Next, we will discuss another pattern matching algorithm, the Boyer-Moore algorithm.

The Boyer-Moore algorithm

As we have already discussed, the main objective of the string pattern matching algorithm is to find ways of skipping comparisons as much as possible by avoiding unnecessary comparisons.

The Boyer-Moore pattern matching algorithm is another such algorithm (along with the KMP algorithm) that further improves the performance of pattern matching by skipping comparisons using different methods. We have to understand the following concepts in order to understand the Boyer-Moore algorithm:

1. In this algorithm, we shift the pattern in the direction from left to right, similar to the KMP algorithm.
2. We compare the characters of the pattern and the text string from right to left, which is the opposite of what we do in the case of the KMP algorithm.
3. The algorithm skips the unnecessary comparisons by using the good suffix and bad character shift heuristics. These heuristics themselves find the possible number of comparisons that can be skipped. We slide the pattern over the given text with the greatest offsets suggested by both of these heuristics.

Let us understand all about these heuristics and the details of how the Boyer-Moore pattern matching algorithm works.

Understanding the Boyer-Moore algorithm

The Boyer-Moore algorithm compares the pattern with the text from right to left, meaning that in this algorithm if the end of the pattern does not match with the text, the pattern can be shifted rather than checking every character of the text. The key idea is that the pattern is aligned with the text and the last character of the pattern is compared with the text, and if they do not match, then it is not required to continue comparing each character and we can rather shift the pattern.

Here, how much we shift the pattern depends upon the mismatched character. If the mismatched character of the text does not appear in the pattern, it means we can shift the pattern by the whole length of the pattern, whereas if the mismatched character appears in the pattern somewhere, then we partially shift the pattern in such a way that the mismatched character is aligned with the other occurrence of that character in the pattern.

In addition, in this algorithm, we can also see what portion of the pattern has matched (with the matched suffix), so we utilize this information and align the text and pattern by skipping any unnecessary comparisons. Making the pattern jump along the text to reduce the number of comparisons rather than checking every character of the pattern with the text is the main idea of an efficient string matching algorithm.

The concept behind the Boyer-Moore algorithm is demonstrated in *Figure 13.15*:

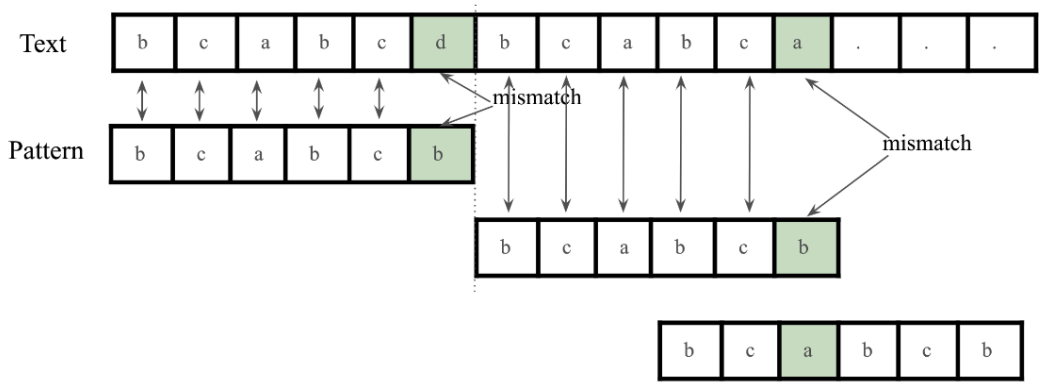


Figure 13.15: A example to demonstrate the concept of the Boyer-Moore algorithm

In the example shown in *Figure 13.15*, where character **b** of the pattern mismatches with character **d** of the text, we can shift the entire pattern since the mismatched character **d** is not present in the pattern anywhere. In the second mismatch, we can see that the mismatched character **a** in the text is present in the pattern, so we shift the pattern to align with that character. This example shows how we can skip unnecessary comparisons. Next, we will discuss further the details of the algorithm.

The Boyer-Moore algorithm has two heuristics to determine the maximum shift possible for the pattern when we find a mismatch:

- Bad character heuristic
- Good suffix heuristic

At the time of a mismatch, each of these heuristics suggests possible shifts, and the Boyer-Moore algorithm shifts the pattern over the text string by a longer distance considering the maximum shift given by bad character and good suffix heuristics. The details of the bad character and good suffix heuristics are explained in detail with examples in the following subsections.

Bad character heuristic

The Boyer-Moore algorithm compares the pattern and the text string in the direction of right to left. It uses the bad character heuristic to shift the pattern, where we start comparing character by character from the end of the pattern, and if they match then we compare the second to-last character, and if that also matches, then the process is repeated until the entire pattern is matched or we get a mismatch.

The mismatched character of the text is also known as a bad character. If we get any mismatch in this process, we shift the pattern according to one of the following conditions:

1. If the mismatched character of the text does not occur in the pattern, then we shift the pattern next to the mismatched character.
2. If the mismatched character has one occurrence in the pattern, then we shift the pattern in such a way that we align with the mismatched character.
3. If the mismatched character has more than one occurrence in the pattern, then we make the most minimal shift possible to align the pattern with that character.

Let us understand these three cases with examples. Consider a text string (T) and the pattern = {**acacac**}. We start by comparing the characters from right to left, that is, character **c** of the pattern and character **b** of the text string. Since they do not match, we look for the mismatched character of the text string (that is **b**) in the pattern. Since the bad character **b** does not appear in the pattern, we shift the pattern next to the mismatched character, as shown in *Figure 13.16*:

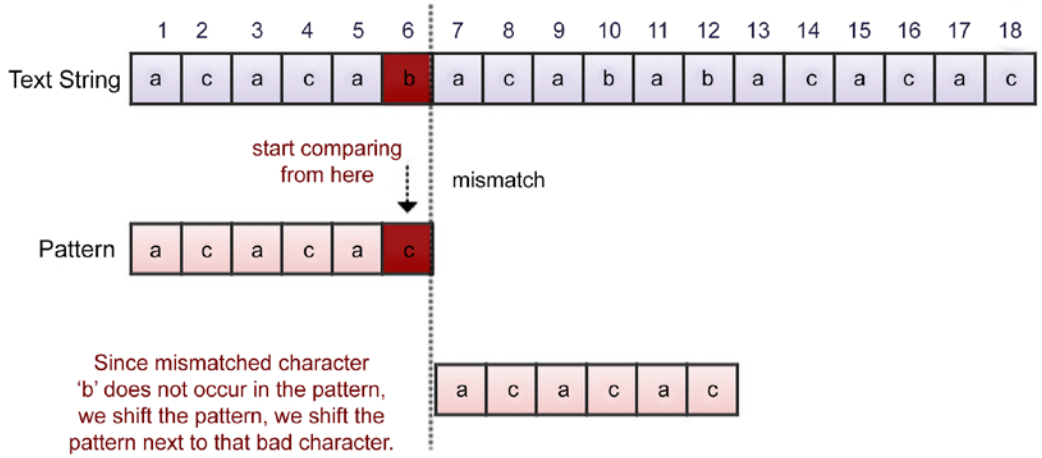


Figure 13.16: Example of the bad character heuristic in the Boyer-Moore algorithm

Let's take another example with a given text string and the pattern = {**acacac**} as shown in *Figure 13.17*. For the given example, we compare the characters of the text string and the pattern from right to left, and we get a mismatch for the character **d** of the text. Here, the suffix **ac** is matched, but the characters **d** and **c** do not match, and the mismatched character **d** does not appear in the pattern. Therefore, we shift the pattern next to the mismatched character, as shown in *Figure 13.17*:

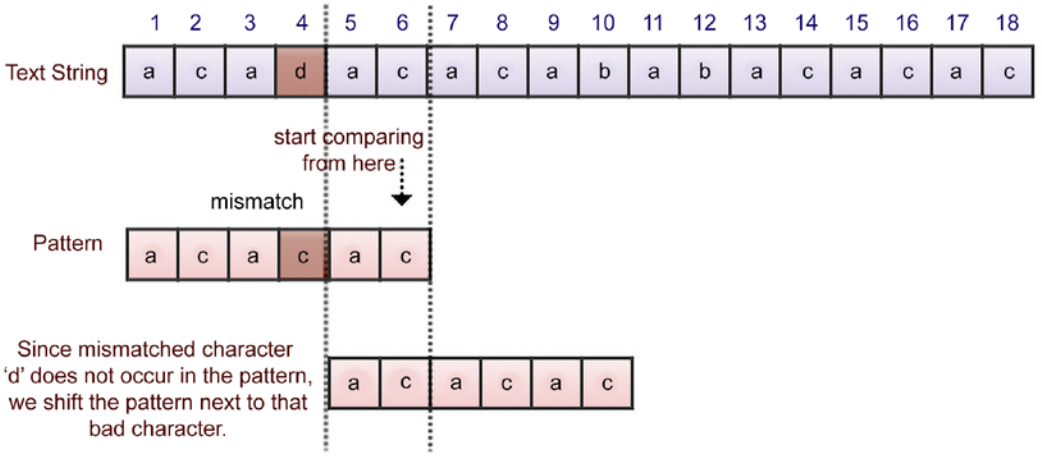


Figure 13.17: Second example of the bad character heuristic in the Boyer-Moore algorithm

Let's consider an example to understand the second and third cases of the bad character heuristic for the given text string and the pattern as shown in *Figure 13.18*. Here, the suffix **ac** is matched, but the next characters, **a** and **c**, do not match, so we search for the occurrences of the mismatched character **a** in the pattern. Since it has two occurrences in the pattern, we have two options for shifting the pattern to align it with the mismatched character. Both of these options are shown in *Figure 13.18*:

In such situations where we have more than one option to shift the pattern, we apply the least possible number of shifts to prevent missing any possible match. If on the other hand we have only one occurrence of the mismatched character in the pattern, we can easily shift the pattern in such a way that the mismatched character is aligned. So, in this example, we would prefer option 1 to shift the pattern as shown in *Figure 13.18*:

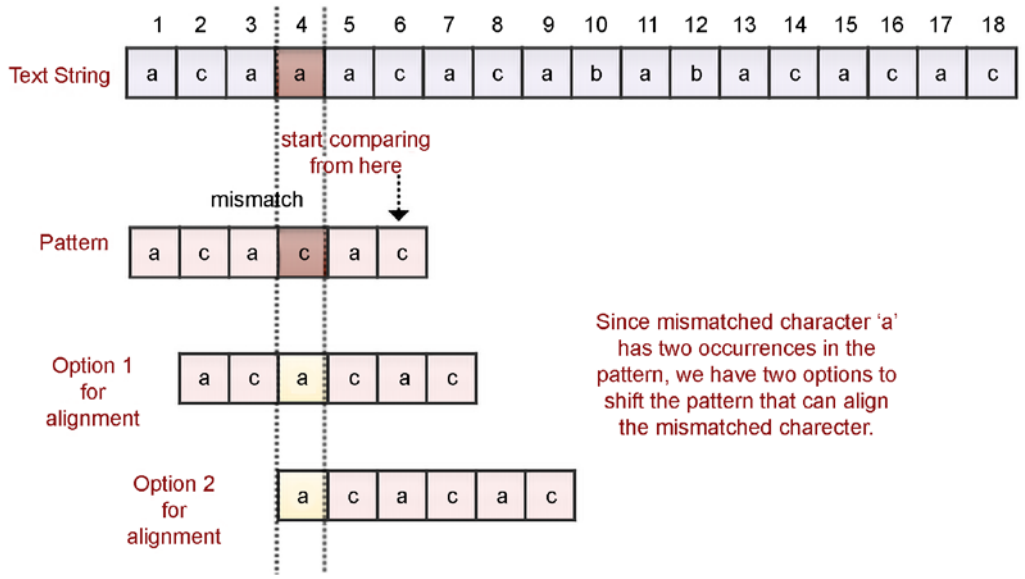


Figure 13.18: Third example of the bad character heuristic in the Boyer-Moore algorithm

We have discussed the bad character heuristic so far, and we consider the good suffix heuristic in the next section.

Good suffix heuristic

The bad character heuristic does not always provide good suggestions for shifting the pattern. The Boyer-Moore algorithm also uses the good suffix heuristic to shift the pattern over the text string, which is based on the matched suffix. In this method, we shift the pattern to the right in such a way that the matched suffix of the pattern is aligned with another occurrence of the same suffix in the pattern.

It works like this: we start by comparing the pattern and the text string from right to left, and if we find any mismatch, then we check the occurrence of the suffix in the pattern that has been matched so far, which is known as a good suffix.

In such situations, we shift the pattern in such a way that we align another occurrence of the good suffix to the text. The good suffix heuristic has two main cases:

1. The matching suffix has one or more occurrences in the pattern
2. Some part of the matching suffix is present at the start of the pattern (this means that the suffix of the matched suffix exists as the prefix of the pattern)

Let's understand these cases with the following examples. Suppose we have a given text string and the pattern **acabac** as shown in *Figure 13.19*. We start comparing the characters from right to left, and we get a mismatch with the character **a** of the text string and **b** of the pattern. By the point of this mismatch, we have already matched the suffix **ac**, which is called the "good suffix." Now, we search for another occurrence of the good suffix **ac** in the pattern (which is present at the starting position of the pattern in this example) and we shift the pattern to align it with that suffix, as shown in *Figure 13.19*:

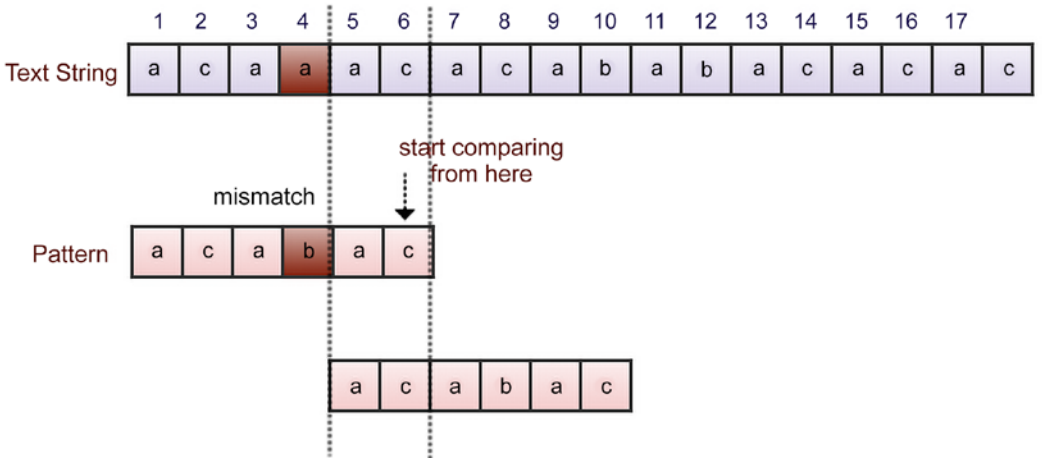


Figure 13.19: Example of the good suffix heuristic in the Boyer-Moore algorithm

Let's take another example to understand the good suffix heuristic. Consider the text string and pattern given in *Figure 13.18*. Here, we get a mismatch between characters **a** and **c**, and we get a good suffix **ac**. Here, we have two options for shifting the pattern to align it with the good suffix string.

In a situation where we have more than one option to shift the pattern, we take the option with the lower number of shifts. For this reason, we take option 1 in this example, as shown in *Figure 13.20*:

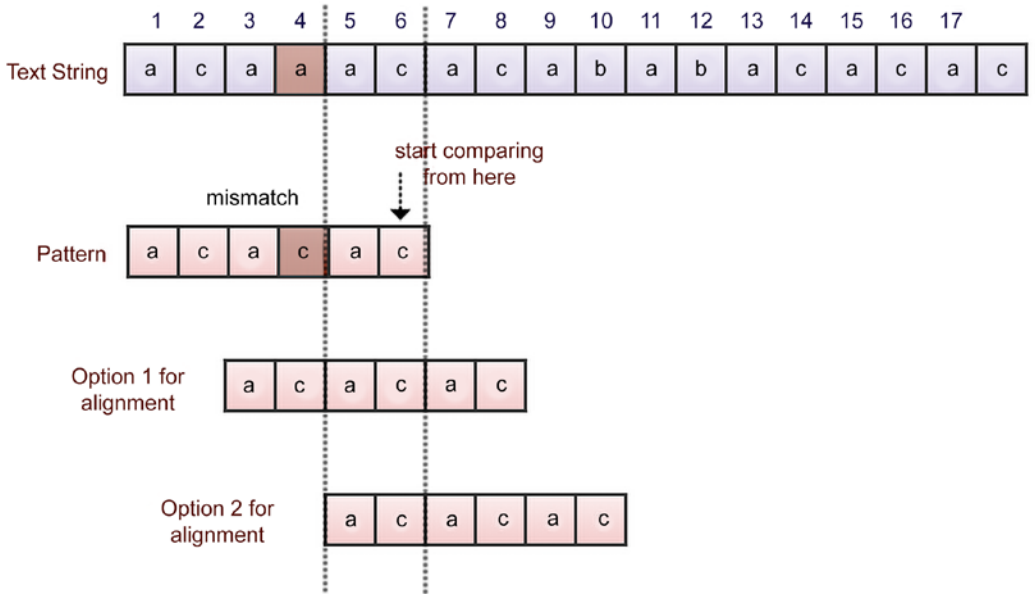


Figure 13.20: Second example of the good suffix heuristic in the Boyer-Moore algorithm

Let's take a look at another example of the text string and pattern shown in *Figure 13.19*. In this example, we get a good suffix string **aac**, and we get a mismatch for the characters **b** of the text string and **a** of the pattern. Now, we search for the good suffix **aac** in the pattern, but we do not find another occurrence of it. When this happens, we check whether the prefix of the pattern matches the suffix of the good suffix, and if so, we shift the pattern to align with it.

For this example, we find that the prefix **ac** at the start of the pattern does not match with the full good suffix, but does match the suffix **ac** of the good suffix **aac**. In such a situation, we shift the pattern by aligning with the suffix of **aac** that is also a prefix of the pattern as shown in *Figure 13.21*:

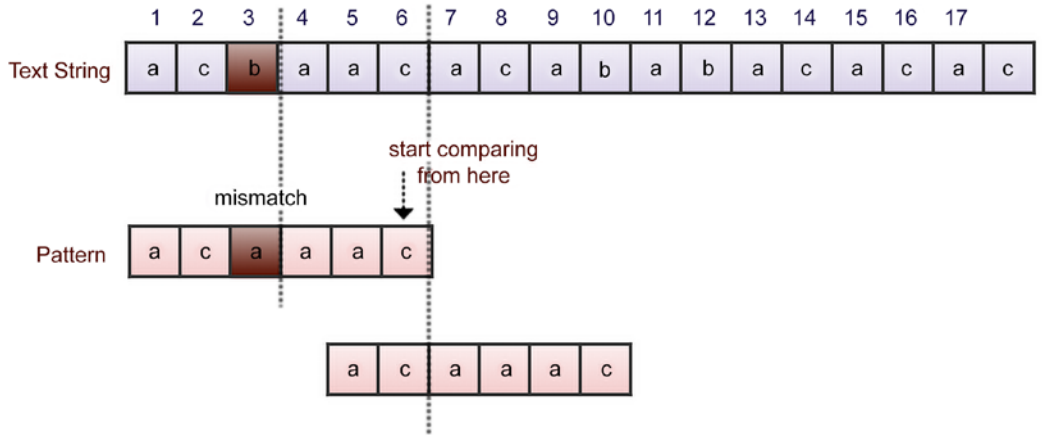


Figure 13.21: Third example of the good suffix heuristic in the Boyer-Moore algorithm

Another case for the good suffix heuristic for the given text string and pattern is shown in *Figure 13.22*. In this example, we compare the text and pattern and find the good suffix **aac**, and we get a mismatch with character **b** of the text and **a** of the pattern.

Next, we search for the matched good suffix in the pattern, but there is no occurrence of the suffix in the pattern, nor does any prefix of the pattern match the suffix of the good suffix. So, in this kind of situation, we shift the pattern after the matched good suffix as shown in *Figure 13.22*:

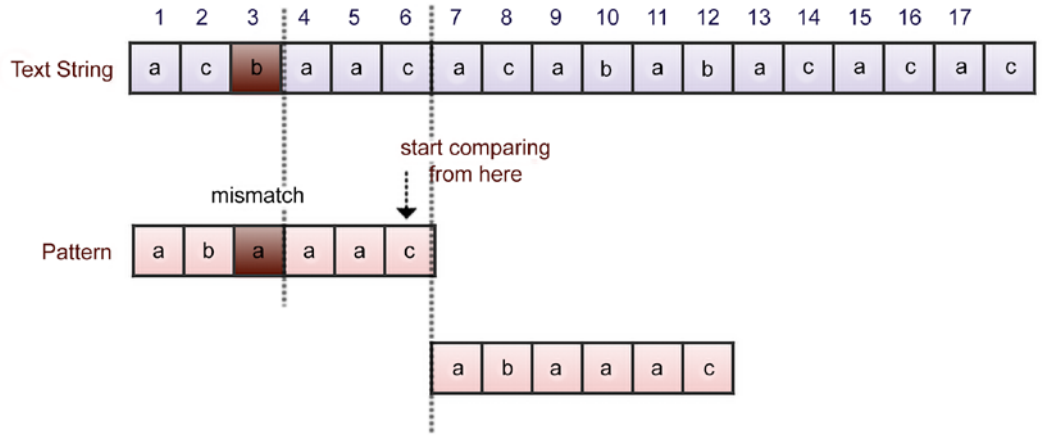


Figure 13.22: Fourth example of the good suffix heuristic in the Boyer-Moore algorithm

In the Boyer-Moore algorithm, we compute the shifts given by the bad character and good suffix heuristics. Further, we shift the pattern by the longer of the distances given by the bad character and good suffix heuristics.

The Boyer-Moore algorithm has a time complexity of `O(m)` for the preprocessing of the pattern, and the searching has a time complexity of `O(mn)`, where `m` is the length of the pattern and `n` is the length of the text.

Next, let us discuss the implementation of the Boyer-Moore algorithm.

Implementing the Boyer-Moore algorithm

Let's understand the implementation of the Boyer-Moore algorithm. The complete implementation of the Boyer-Moore algorithm is as follows:

```
text = "acbaacacababacacac"
pattern = "acacac"


matched_indexes = []

i=0
flag = True
while i<=len(text)-len(pattern):
    for j in range(len(pattern)-1, -1, -1):     #reverse searching
        if pattern[j] != text[i+j]:
            flag = False    #indicates there is a mismatch
            if j == len(pattern)-1:     #if good-suffix is not present, we
test bad character
                if text[i+j] in pattern[0:j]:
                    i=i+j-pattern[0:j].rfind(text[i+j])
                    #i+j is index of bad character, this line is used for
jumping pattern to match bad character of text with same character in
pattern
                else:
                    i=i+j+1     #if bad character is not present, jump
pattern next to it
            else:
```

```
                k=1
                while text[i+j+k:i+len(pattern)] not in
pattern[0:len(pattern)-1]:
                    #used for finding sub part of a good-suffix
                    k=k+1
                if len(text[i+j+k:i+len(pattern)]) != 1:    #good-suffix
should not be of one character
                    gsshift=i+j+k-pattern[0:len(pattern)-1].
rfind(text[i+j+k:i+len(pattern)])
                    #jumps pattern to a position where good-suffix of
pattern matches with good-suffix of text
                else:
                    #gsshift=i+len(pattern)
                    gsshift=0   #when good-suffix heuristic is not
applicable,
                                #we prefer bad character heuristic
                if text[i+j] in pattern[0:j]:
                    bcshift=i+j-pattern[0:j].rfind(text[i+j])
                    #i+j is index of bad character, this line is used for
jumping pattern to match bad character of text with same character in
pattern
                else:
                    bcshift=i+j+1
                i=max((bcshift, gsshift))
            break
    if flag:    #if pattern is found then normal iteration
        matched_indexes.append(i)
        i = i+1
    else:   #again set flag to True so new string in text can be examined
        flag = True

print ("Pattern found at", matched_indexes)
```

An explanation of each of the statements of the preceding code is presented here. Initially, we have the text string and the pattern. After initializing the variables, we start with a `while` loop that starts by comparing the last character of the pattern with the corresponding character of the text.

Then, the characters are compared from right to left by the use of the nested loop from the last index of the pattern to the first character of the pattern. This uses `range(len(pattern)-1, -1, -1)`.

The outer `while` loop keeps track of the index in the text string while the inner `for` loop keeps track of the index position in the pattern.

Next, we start comparing the characters by using `pattern[j] != text[i+j]`. If they are mismatched, we make the `flag` variable `False`, denoting that there is a mismatch.

Now, we check whether the good suffix is present using the condition `j == len(pattern)-1`. If this condition is true, it means that there is no good suffix possible, so we check for the bad character heuristics, that is, whether a mismatched character is present in the pattern using the condition `text[i+j] in pattern[0:j]`, and if the condition is true, then it means that the bad character is present in the pattern. In this case, we move the pattern to align this bad character to the other occurrence of this character in the pattern by using `i=i+j-pattern[0:j].rfind(text[i+j])`. Here, `(i+j)` is the index of the bad character.

If the bad character is not present in the pattern (it isn't in the `else` part of it), we move the whole pattern next to the mismatched character by using the index `i=i+j+1`.

Next, we go into the `else` part of the condition to check the good suffix. When we find the mismatch, we further test to see whether we have any subpart of a good suffix present in the prefix of the pattern. We do this using the following condition:

```
text[i+j+k:i+len(pattern)] not in pattern[0:len(pattern)-1]
```

Furthermore, we check whether the length of the good suffix is `1` or not. If the length of the good suffix is `1`, we do not consider this shift. If the good suffix is more than `1`, we find out the number of shifts by using the good suffix heuristics and store this in the `gsshift` variable. This is the pattern, which leads to a position where the good suffix of the pattern matches the good suffix in the text using the instruction `gsshift=i+j+k-pattern[0:len(pattern)-1].rfind(text[i+j+k:i+len(pattern)])`. Furthermore, we computed the number of shifts possible due to the bad character heuristic and stored this in the `bcshift` variable. The number of shifts possible is `i+j-pattern[0:j].rfind(text[i+j])` when the bad character is present in the pattern, and the number of shifts possible would be `i+j+1` in the case of the bad character not being present in the pattern.

Next, we shift the pattern on the text string by the maximum number of moves given by the bad character and good suffix heuristics by using the instruction `i=max((bcshift, gsshift))`. Finally, we check whether the `flag` variable is `True` or not. If it is `True`, this means that the pattern has been found and that the matched index has been stored in the `matched_indexes` variable.

We have discussed the concept of the Boyer-Moore pattern matching algorithm, which is an efficient algorithm that skips unnecessary comparisons using the bad character and good suffix heuristics.

Summary

In this chapter, we have discussed the most popular and important string matching algorithms that have a wide range of applications in real-time scenarios. We discussed the brute force, Rabin-Karp, KMP, and Boyer-Moore pattern matching algorithms. In string matching algorithms, we try to uncover ways to skip unnecessary comparisons and move the pattern over the text as fast as possible. The **KMP algorithm** detects unnecessary comparisons by looking at the overlapping substrings in the pattern itself to avoid redundant comparisons. Furthermore, we discussed the **Boyer-Moore algorithm**, which is very efficient when the text and pattern are long. It is the most popular algorithm used for string matching in practice.

Exercise

1. Show the KMP `prefix` function for the pattern `"aabaabcab"`.
2. If the expected number of valid shifts is small and the modulus is larger than the length of the pattern, then what is the matching time of the Rabin-Karp algorithm?
    a. Theta (m)
    b. Big O (n+m)
    c. Theta (n-m)
    d. Big O (n)
3. How many spurious hits does the Rabin-Karp string matching algorithm encounter in the text `T = "3141512653849792"` when looking for all occurrences of the pattern `P = "26"`, working modulo `q = 11`, and over the alphabet set `Σ = {0, 1, 2,..., 9}`?
4. What is the basic formula applied in the Rabin-Karp algorithm to get the computation time as Theta (m)?
    a. Halving rule
    b. Horner's rule
    c. Summation lemma
    d. Cancellation lemma

5. The Rabin-Karp algorithm can be used for discovering plagiarism in text documents.

    a. True

    b. False

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://packt.link/MEvK4`

Appendix

Answers to the Questions

Chapter 2: Introduction to Algorithm Design

Question 1

Find the time complexity of the following Python snippets:

a.

```
i=1
while(i<n):
    i*=2
    print("data")
```

b.

```
i =n
while(i>0):
    print("complexity")
    i/ = 2
```

c.

```
for i in range(1,n):
    j = i
    while(j<n):
        j*=2
```

d.

```
i=1
while(i<n):
    print("python")
    i = i**2
```

Solution

a. The complexity will be `O(log(n))`.

   As we are multiplying the integer `i` by `2` in each step there will be exactly `log(n)` steps. (`1`, `2`, `4`, ... till `n`).

b. The complexity will be `O(log(n))`.

   As we are dividing the integer `i` by `2` in each step there will be exactly `log(n)` steps. (`n`, `n/2`, `n/4`, ... till `1`).

c. The outer loop will run `n` times for each `i` in the outer loop, while the inner `while` loop will run `log(i)` times because we are multiplying each of the `j` values by `2` until it is less than `n`. Hence, there will be a maximum of `log(n)` steps in the inner loop. Therefore, the overall complexity will be `O(nlog(n))`.

   In this code snippet, the `while` loop will execute based on the value of `i` until the condition becomes `false`. The value of `i` is incrementing in the following series:

   ```
   2, 4, 16, 256, ... n
   ```

   We can see that the number of times the loop is executing is $\log_2(\log_2(n))$ for a given value of `n`. So, for this series there will be exactly $\log_2(\log_2(n))$ executions of the loop. Hence the time complexity will be $O(\log_2(\log_2(n))$.

Chapter 3: Algorithm Design Techniques and Strategies

Question 1

Which of the following options will be correct when a top-down approach of dynamic programming is applied to solve a given problem related to the space and time complexity?

a. It will increase both time and space complexity
b. It will increase the time complexity, and decrease the space complexity
c. It will increase the space complexity, and decrease the time complexity
d. It will decrease both time and space complexities

Solution

Option c is correct.

Since the top-down approach of dynamic programming uses the memoization technique, which stores the pre-calculated solution of a subproblem. It avoids recalculation of the same subproblem that decreases the time complexity, but at the same time, the space complexity will increase because of storing the extra solutions of the subproblems.

Question 2

What will be the sequence of nodes in the following edge-weighted directed graph using the greedy approach (assume node **A** as the source)?

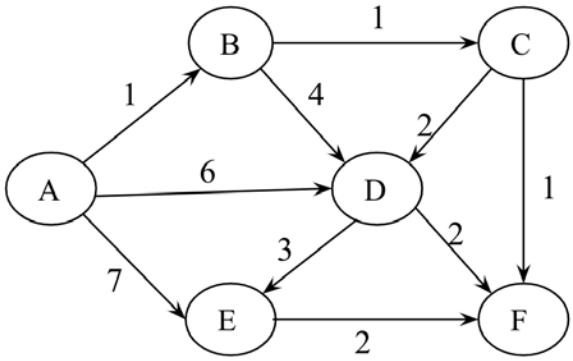


Figure A.1: A weighted directed graph

Solution

A, B, C, F, E, D

In Dijkstra's algorithm, at each, point we choose the smallest weight edge, which starts from any one of the vertices in the shortest path found so far, and add it to the shortest path.

Question 3

Consider the weights and values of the items in *Table 3.8*. Note that there is only one unit of each item.

Item	Weight	Value
A	2	10
B	10	8
C	4	5
D	7	6

Table A.1: The weights and values of different items

We need to maximize the value; the maximum weight should be 11 kg. No item may be split. Establish the values of the items using a greedy approach.

Solution

Firstly, we picked item A (weight 2 kg) as the value is the maximum (10). The second highest value is for item B, but as the total weight becomes 12 kg, this violates the given condition, so we cannot pick it. The next highest value is item D, and now the total weight becomes 2+7 = 9 kg (item A + item D). The next remaining item, C, cannot be picked because after adding it, the total weight condition will be violated.

So, the total value of the items picked up using the greedy approach = 10 + 6 = 16

Chapter 4: Linked Lists

Question 1

What will be the time complexity when inserting a data element after an element that is being pointed to by a pointer in a linked list?

Solution

It will be `O(1)`, since there is no need to traverse the list to reach the desired location where a new element is to be added. A pointer is pointing to the current location, and a new element can be directly added by linking it.

Question 2

What will be the time complexity when ascertaining the length of the given linked list?

Solution

`O(n)`.

In order to find out the length, each node of the list has to be traversed, which will take `O(n)`.

Question 3

What will be the worst-case time complexity for searching a given element in a singly linked list of length n?

Solution

`O(n)`.

In the worst case, the data element to be searched will be at the end of the list, or will not be present in the list. In that case, there will be a total `n` number of comparisons, thus making the worst-case time complexity `O(n)`.

Question 4

For a given linked list, assuming it has only one `head` pointer that points to the starting point of the list, what will be the time complexity for the following operations?

a. Insertion at the front of the linked list
b. Insertion at the end of the linked list
c. Deletion of the front node of the linked list
d. Deletion of the last node of the linked list

Solution

a. `O(1)`. This operation can be performed directly through the head node.
b. `O(n)`. It will require traversing the list to reach the end of the list.
c. `O(1)`. This operation can be performed directly through the head node.
d. `O(n)`. It will require traversing the list to reach the end of the list.

Question 5

Find the n^{th} node from the end of a linked list.

Solution

In order to find out the n^{th} node from the end of the linked list, we can use two pointers – `first` and `second`. Firstly, move the second pointer to `n` nodes from the starting point. Then, move both the pointers one step at a time until the second pointer reaches the end of the list. At that time, the first pointer will point to the n^{th} node from the end of the list.

Question 6

How can you establish whether there is a loop (or circle) in a given linked list?

Solution

To find out the loop in a linked list, it is most efficient to use **Floyd's cycle-finding algorithm**. In this approach, two pointers are used to detect the loop – let's say the first and second pointers. We start moving both the pointers from the starting point of the list.

We move the first and second pointers by one and two nodes at a time. If these two pointers meet at the same node, that indicates that there is a loop, otherwise, there is no loop in the given linked list.

The process is shown in the below figure with an example:

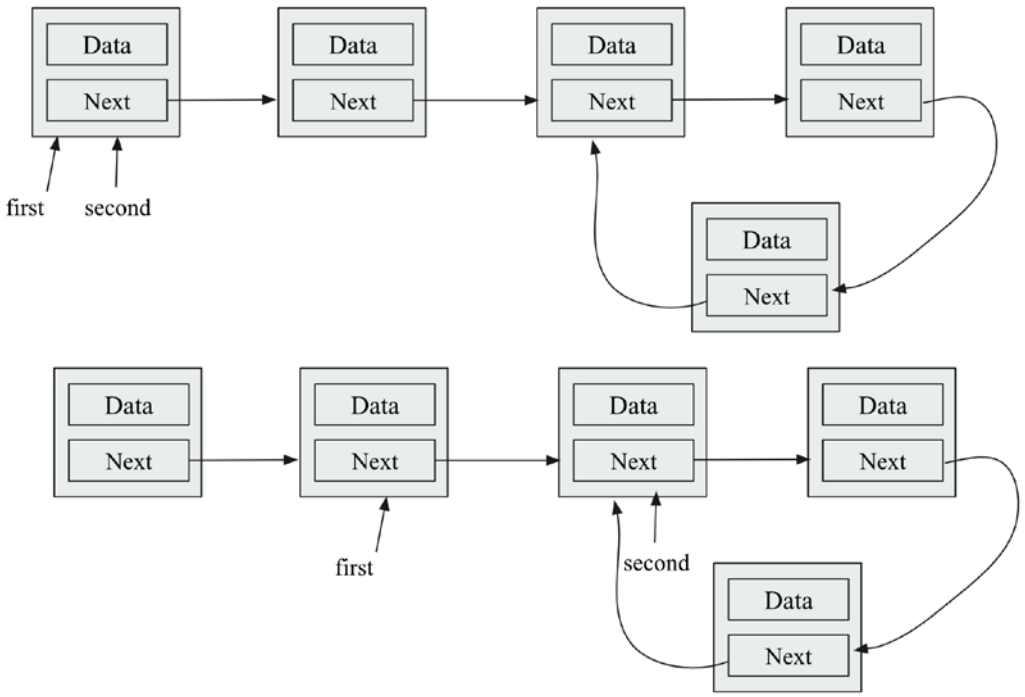


Figure A.2: Loop in a singly linked list

Question 7

How can you ascertain the middle element of the linked list?

Solution

It can be done with two pointers, say, the first and second pointers. Start moving these two pointers from the starting node. The first and second pointers should move one and two nodes at a time, respectively. When the second node reaches the end of the list, the first node will point to the middle element of the singly linked list.

Chapter 5: Stacks and Queues

Question 1

Which of the following options is a true queue implementation using linked lists?

a. If, in the enqueue operation, new data elements are added at the start of the list, then the dequeue operation must be performed from the end.

b. If, in the enqueue operation, new data elements are added to the end of the list, then the enqueue operation must be performed from the start of the list.

c. Both of the above.

d. None of the above.

Solution

B is correct. The queue data structure follows a FIFO order, hence data elements must be added to the end of the list, and then removed from the front.

Question 2

Assume a queue is implemented using a singly linked list that has `head` and `tail` pointers. The enqueue operation is implemented at `head`, and the dequeue operation is implemented at the `tail` of the queue. What will be the time complexity of the enqueue and dequeue operations?

Solution

The time complexity of the enqueue operation will be `O(1)` and `O(n)` for the dequeue operation. As for the enqueue operation, we only need to delete the `head` node, which can be achieved in `O(1)` for a singly linked list. For the dequeue operation, to delete the `tail`, we need to traverse the whole list first to the `tail`, and then we can delete it. For this we need linear, `O(n)`, time.

Question 3

What is the minimum number of stacks required to implement a queue?

Solution

Two stacks.

Using two stacks and the enqueue operation, the new element is entered at the top of `stack1`. In the dequeue process, if `stack2` is empty, all the elements are moved to `stack2`, and finally, the top of `stack2` is returned.

Question 4

The enqueue and dequeue operations in a queue are implemented efficiently using an array. What will be the time complexity for both of these operations?

Solution

`O(1)` for both operations.

If we use a circular array for the implementation of a queue, then we do not need to shift the elements, just the pointers, so we can implement both the enqueue and dequeue operations in `O(1)` time.

Question 5

How can we print the data elements of a queue data structure in reverse order?

Solution

Make an empty stack, then enqueue each of the elements from the queue and push them into the stack. After the queue is empty, start popping out the elements from the stack and then printing them one by one.

Chapter 6: Trees

Question 1

Which of the following is true about binary trees:

a. Every binary tree is either complete or full
b. Every complete binary tree is also a full binary tree
c. Every full binary tree is also a complete binary tree
d. No binary tree is both complete and full
e. None of the above

Solution

Option A is incorrect since it is not compulsory that a binary tree should be complete or full.

Option B is incorrect since a complete binary tree can have some nodes that are not filled in the last level, so a complete binary tree will not always be a full binary tree.

Option C is incorrect, as it is not always true, the following figure is a full binary tree, but not a complete binary tree:

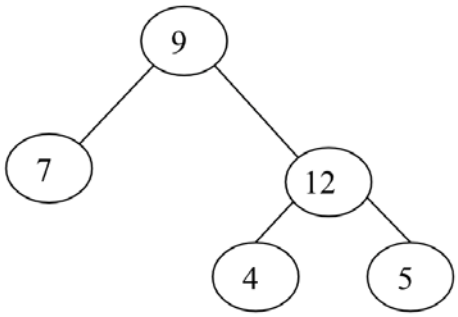


Figure A.3: A binary tree that is full, but not complete

Option D is incorrect, as it is not always true. The following tree is both a complete and full binary tree:

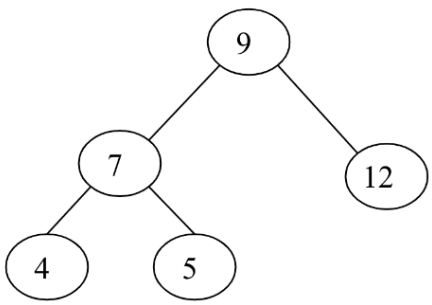


Figure A.4: A binary tree, that is full and complete

Question 2

Which of the tree traversal algorithms visit the root node last?

Solution

`postorder` traversal.

Using `postorder` traversal, we first visit the left subtree, then the right subtree, and finally we visit the `root` node.

Question 3

Consider this binary search tree:

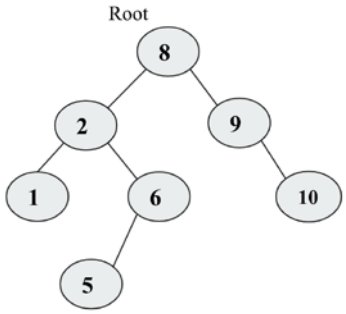


Figure A.5: Sample binary search tree

Suppose we remove the root node 8, and we wish to replace it with any node from the left subtree then what will be the new root?

Solution

The new node will be node 6. To maintain the properties of the binary search tree, the maximum value from the left subtree should be the new root.

Question 4

What will be the `inorder`, `postorder`, and `preorder` traversal of the following tree?

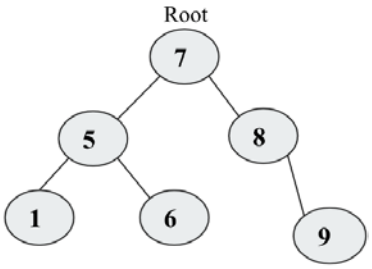


Figure A.6: Example tree

Solution

The `preorder` traversal will be `7-5-1-6-8-9`.

The `inorder` traversal will be `1-5-6-7-8-9`.

The `postorder` traversal will be `1-6-5-9-8-7`.

Question 5

How do you find out if two trees are identical?

Solution

In order to find out if two binary trees are identical or not, both of the trees should have exactly the same data and element arrangement. This can be done by traversing both of the trees with any of the traversal algorithms (it should be the same for both trees) and matching them element by element. If all the elements are the same in traversing both of the trees, then the trees are identical.

Question 6

How many leaves are there in the tree mentioned in *question 4*?

Solution

Three, nodes `1`, `6`, and `9`.

Question 7

What is the relation between a perfect binary tree's height and the number of nodes in that tree?

Solution

$\log_2 (n+1) = h$.

The number of nodes in each level:

Level `0`: $2^0 = 1$ nodes

Level `1`: $2^1 = 2$ nodes

Level `2`: $2^2 = 4$ nodes

Level `3`: $2^3 = 8$ nodes

The total nodes at level `h` can be computed by adding all nodes in each level:

$n = 2^0 + 2^1 + 2^2 + 2^3 + \ldots\ldots 2^{h-1} = 2^h - 1$

So, the relationship between `n` and `h` is: $n = 2^h - 1$

$= \log (n+1) = \log 2^h$

$= \log_2 (n+1) = h$

Chapter 7: Heaps and Priority Queues

Question 1

What will be the time complexity for deleting an arbitrary element from the `min-heap`?

Solution

To delete any element from the `heap`, we first have to search the element that is to be deleted, and then we delete the element.

Total time complexity = Time for searching the element + Deleting the element

= `O(n)` + `O(log n)`

= `O(n)`

Question 2

What will be the time complexity for finding the k^{th} smallest element from the `min-heap`?

Solution

The k^{th} element can be found out from the `min-heap` by performing `delete` operations k times. For each `delete` operation, the time complexity is `O(logn)`. So, the total time complexity for finding out the k^{th} smallest element will be `O(klogn)`.

Question 3

What will be the time complexity to make a `max-heap` that combines two `max-heap` each of size `n`?

Solution

`O(n)`.

Since the time complexity of creating a `heap` from `n` elements is `O(n)`, creating a `heap` of `2n` elements will also be `O(n)`.

Question 4

What will be the worst-case time complexity for ascertaining the smallest element from a binary max-heap and binary min-heap?

Solution

In a max-heap, the smallest element will always be present at a leaf node. So, in order to find out the smallest element, we have to search all the leaf nodes. So, the worst-case complexity will be `O(n)`.

The worst-case time complexity to find out the smallest element in the min-heap will be `O(1)` since it will always be present at the root node.

Question 5

The level order traversal of `max-heap` is `12`, `9`, `7`, `4`, `2`. After inserting new elements `1` and `8`, what will be the final `max-heap` and level order traversal of the final `max-heap`?

Solution

The `max-heap` after the insertion of element `1` is shown in the below figure:

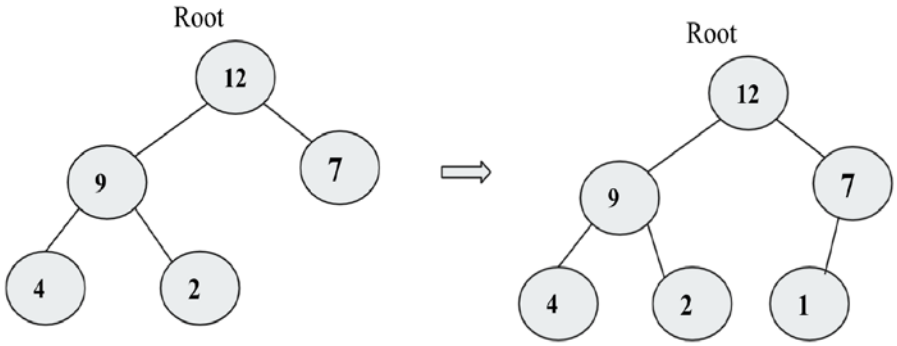


Figure A.7: The max-heap before insertion of element 8

The final `max-heap` after the insertion of element `8` is shown in the below figure:

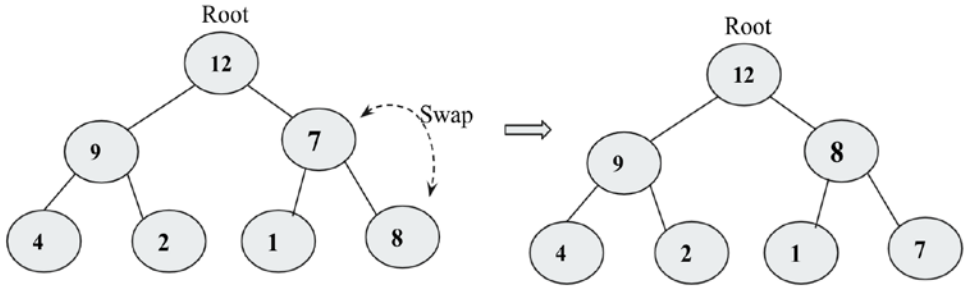


Figure A.8: The max-heap after the insertion of elements 1 and 8

The level order traversal of the final `max-heap` will be `12, 9, 8, 4, 2, 1, 7`.

Question 6

Which of the following is a binary `max-heap`?

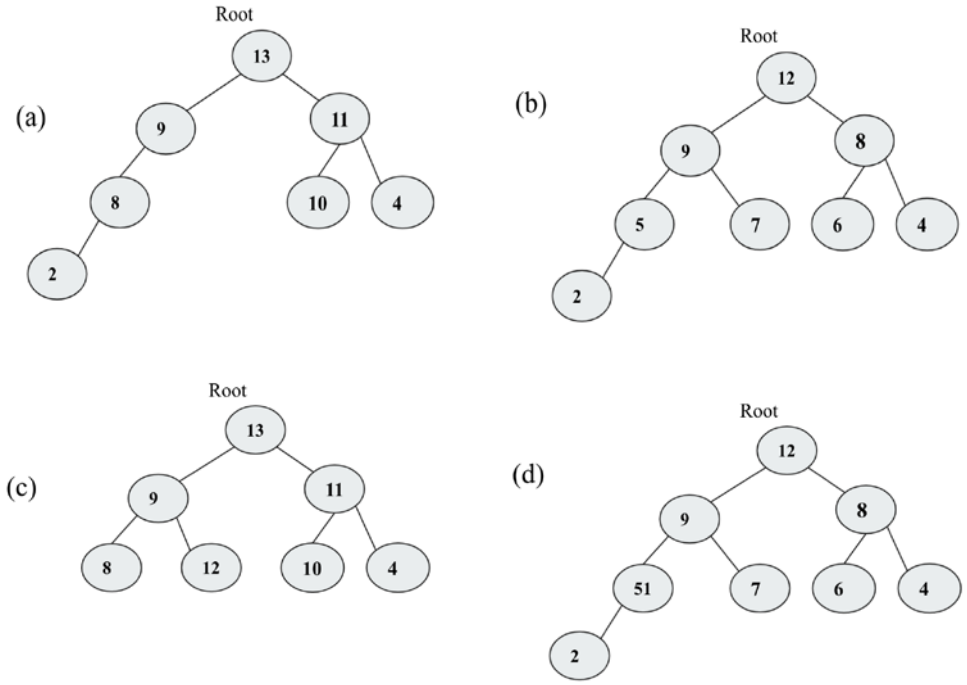


Figure A.9: Example trees

Solution

B.

A binary `max-heap` should be a complete binary tree and all the levels should be filled, except the last level. The value of the parent should be greater or equal to the values of its children.

Option A is not correct because it is not a complete binary tree. Options C and D are not correct because they are not fulfilling the `heap` property. Option B is correct because it is complete and fulfills the `heap` property.

Chapter 8: Hash Tables

Question 1

There is a hash table with 40 slots and there are 200 elements stored in the table. What will be the load factor of the hash table?

Solution

The load factor of the hash table = (no. of elements) / (no. of table slots) = 200/40 = 5.

Question 2

What is the worst-case search time of hashing using a separate chaining algorithm?

Solution

The worst-case time complexity for searching in a separate chaining algorithm using linked lists is `O(n)`, because in the worst case, all the items will be added to `index 1` in a linked list, searching an item will work similarly to a linked list.

Question 3

Assume a uniform distribution of keys in the hash table. What will be the time complexities for the `search`/`insert`/`delete` operations?

Solution

The index of the hash table is computed from the key in `O(1)` time when the keys are uniformly distributed in the hash table. The creation of the table will take `O(n)` time, and other operations such as `search`, `insert`, and `delete` operations will take `O(1)` time because all the elements are uniformly distributed, hence, we directly get the required element.

Question 4

What will be the worst-case complexity for removing the duplicate characters from an array of characters?

Solution

The brute force algorithm starts with the first character and searches linearly with all the characters of the array. If a duplicate character is found, then that character should be swapped with the last character and then the length of the string should be decremented by one. The same process is repeated until all characters are processed. The time complexity of this process is $O(n^2)$.

It can be implemented more efficiently using a hash table in `O(n)` time.

Using this method, we start with the first character of the array and store it in the hash table according to the hash value. We do it for all the characters. If there is any collision, then that character can be ignored, otherwise, the character is stored in the hash table.

Chapter 9: Graphs and Algorithms

Question 1

What is the maximum number of edges (without self-loops) possible in an undirected simple graph with five nodes?

Solution

Each node can be connected to every other node in the graph. So, the first node can be connected to `n-1` nodes, the second node can be connected to `n-2` nodes, the third node can be connected to `n-3` nodes, and so on. The total number of nodes will be:

[(n-1)+(n-2)+ ... +3+2+1] = n(n-1)/2.

Question 2

What do we call a graph in which all the nodes have an equal degree?

Solution

A complete graph.

Question 3

Explain what cut vertices are and identify the cut vertices in the given graph?

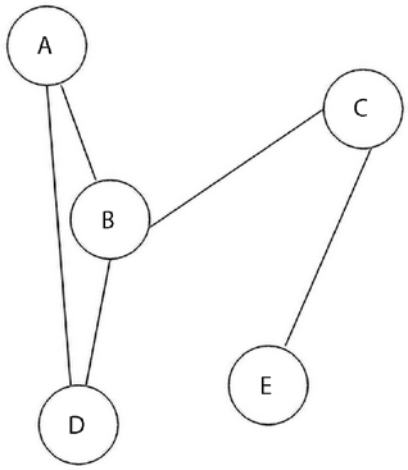


Figure A.10: Sample graph

Solution

Cut vertices also known as articulation points. These are the vertices in the graph, after removal of which, the graph splits in two disconnected components. In the given graph, the vertices `B`, and `C` are cut vertices since after removal of node `B`, the graph will split into `{A, D}`, `{C,E}` vertices. And, after removal of node `C`, the graph will split into `{A,B, D}`, `{E}` vertices.

Question 4

Assuming a graph `G` of order `n`, what will be the maximum number of cut vertices possible in graph `G`?

Solution

It will be `n-2`, since the first and last vertices will not be cut vertices, except those two nodes, all nodes can split the graph into two disconnected graphs. See the below graph:

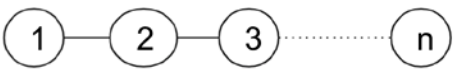


Figure A.11: A graph G

Chapter 10: Searching

Question 1

On average, how many comparisons are required in a linear search of n elements?

Solution

The average number of comparisons in linear search will be as follows. When a search element is found at the 1st position, 2nd position, 3rd position, and similarly at the nth position, correspondingly, it will require 1, 2, 3... n number of comparisons.

Total average number of comparisons

$$= \frac{(1 + 2 + 3 + \cdots n)}{n}$$

$$= \frac{[\frac{n(n+1)}{2}]}{n}$$

$$= \frac{(n+1)}{2}$$

Question 2

Assume there are eight elements in a sorted array. What is the average number of comparisons that will be required if all the searches are successful and if the binary search algorithm is used?

Solution

Average number of comparisons = (1+2+2+3+3+3+3+4)/8

= 21/8

= 2.625

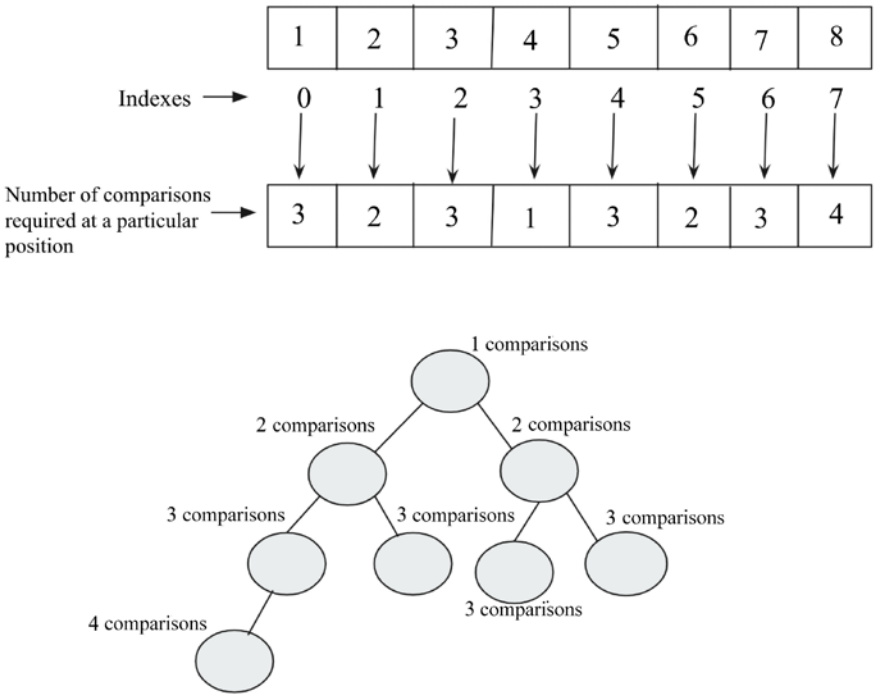


Figure A.12: Demonstration of number of the comparisons in the given array

Question 3

What is the worst-case time complexity of the binary search algorithm?

Solution

`O(logn)`.

The worst-case scenario of the binary search algorithm will occur when the desired element is present in the first position or at the last position. In that case, `log(n)` comparisons will be required. Hence the worst-case complexity will be `O(logn)`.

Question 4

When should the interpolation search algorithm perform better than the binary search algorithm?

Solution

The interpolation search algorithm performs better than the binary search algorithm when the data items in the array are uniformly distributed.

Chapter 11: Sorting

Question 1

If an array `arr = {55, 42, 4, 31}` is given and bubble sort is used to sort the array elements, then how many passes will be required to sort the array?

a. 3

b. 2

c. 1

d. 0

Solution

The answer is a. To sort `n` elements, the bubble sort algorithm requires `(n-1)` iterations (passes), where `n` is the number of elements in the given array. Here in the question, the value of `n = 4`, so `4-1 = 3` iterations will be required to sort the given array.

Question 2

What is the worst-case complexity of bubble sort?

a. O(nlogn)

b. O(logn)

c. O(n)

d. $O(n^2)$

Solution

The answer is d. The worst case appears when the given array is in reverse order. In that case, the time complexity of bubble sort would be $O(n^2)$.

Question 3

Apply quicksort to the sequence (56, 89, 23, 99, 45, 12, 66, 78, 34). What is the sequence after the first phase, and what pivot is the first element?

a. 45, 23, 12, 34, 56, 99, 66, 78, 89

b. 34, 12, 23, 45, 56, 99, 66, 78, 89

c. 12, 45, 23, 34, 56, 89, 78, 66, 99

d. 34, 12, 23, 45, 99, 66, 89, 78, 56

Solution

b.

After the first phase, 56 would be in the right position so that all the elements smaller than 56 will be on the left side of it, and elements bigger than 56 will be on the right side of it. Further, quicksort is applied recursively to the left subarray and right subarray. The process of the quicksort for the given sequence, as shown in the below figure.

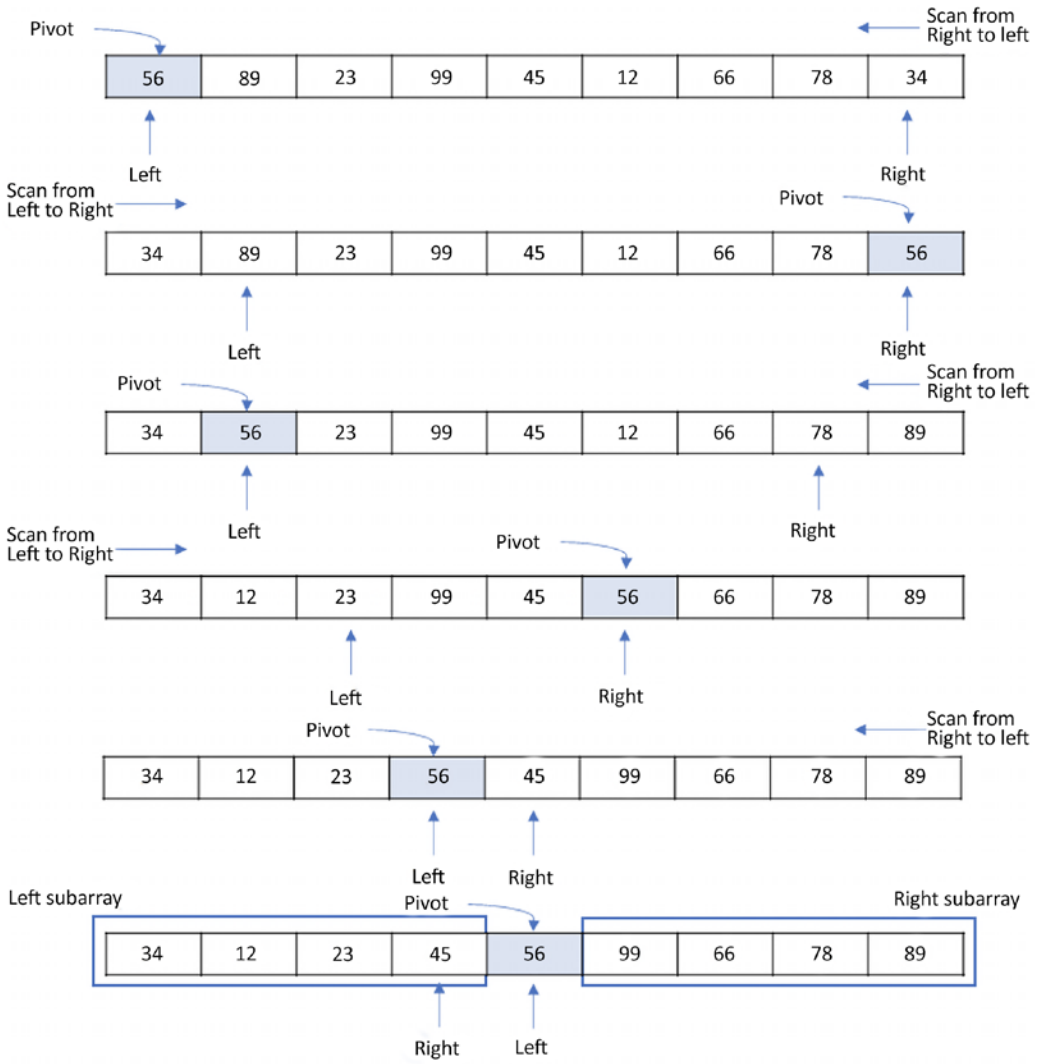


Figure A.13: Demonstration of the quicksort algorithm

Question 4

Quicksort is a _____

a. Greedy algorithm
b. Divide-and-conquer algorithm
c. Dynamic programming algorithm
d. Backtracking algorithm

Solution

The answer is b. Quicksort is a divide-and-conquer algorithm. Quick sort first partitions a large array into two smaller sub arrays and then recursively sorts the sub-arrays. Here, we find the pivot element such that all elements to the left side of the pivot element would be smaller than the pivot element and create the first subarray. The elements to the right side of the pivot element are greater than the pivot element and create the second subarray. Thus, the given problem is reduced into two smaller sets. Now, sort these two subarrays again, finding the pivot element in each subarray, i.e. apply quicksort on each subarray.

Question 5

Consider a situation where a `swap` operation is very costly. Which of the following sorting algorithms should be used so that the number of `swap` operations is minimized?

a. Heap sort
b. Selection sort
c. Insertion sort
d. Merge sort

Solution

b. In the selection sort algorithm, in general, we identify the largest element, and then swap it with the last element so that in each iteration, only one `swap` is required. For `n` elements, the total (`n-1`) swaps will be required, which is the lowest in comparison to all other algorithms.

Question 6

If the input array `A = {15, 9, 33, 35, 100, 95, 13, 11, 2, 13}` is given, using selection sort, what would be the order of the array after the fifth `swap`? (Note: it counts regardless of whether they exchange or remain in the same position.)

a. 2, 9, 11, 13, 13, 95, 35, 33, 15, 100

b. 2, 9, 11, 13, 13, 15, 35, 33, 95, 100

c. 35, 100, 95, 2, 9, 11, 13, 33, 15, 13

d. 11, 13, 9, 2, 100, 95, 35, 33, 13, 13

Solution

The answer is a. In selection sort, select the smallest element. Start the comparison from the beginning of the array and swap the smallest element with the first greatest element. Now, exclude the previous element that was chosen as the smallest element, as it has been put in the right place.

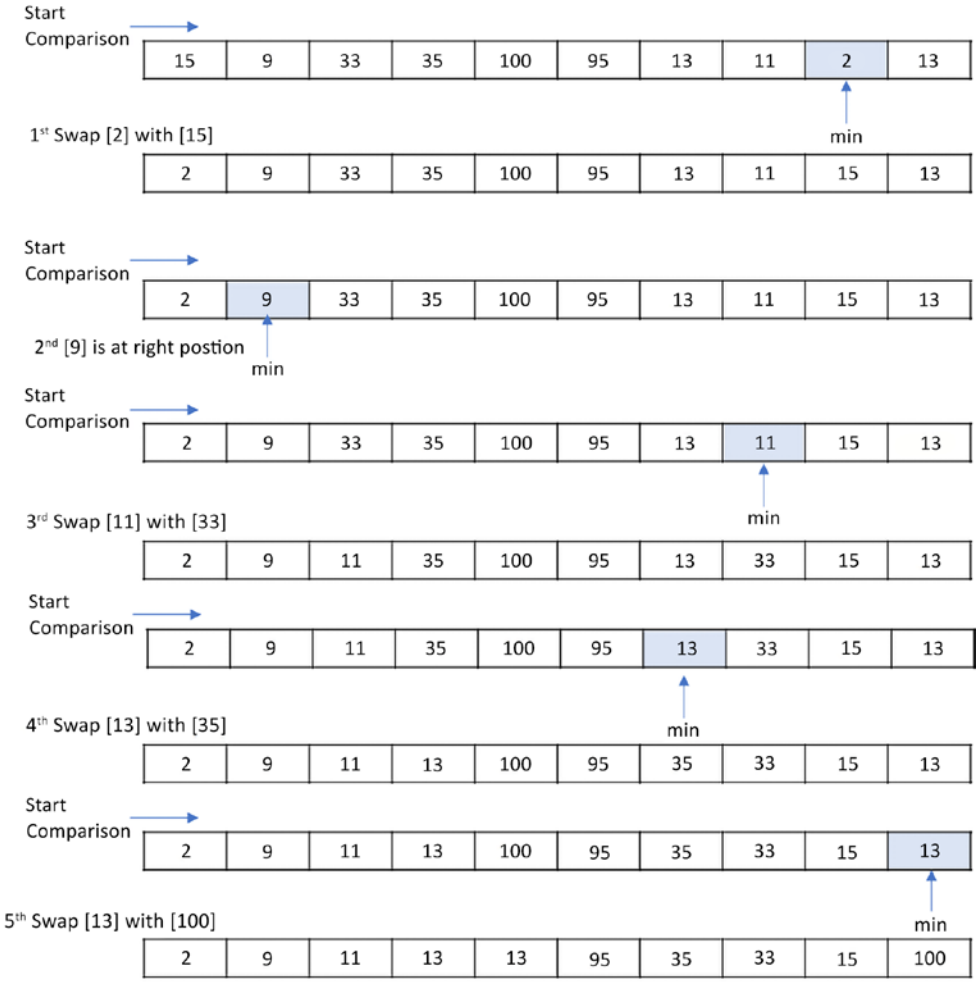


Figure A.14: Demonstration of insertion sort on the given sequence

Question 7

What will be the number of iterations to sort the elements `{44, 21, 61, 6, 13, 1}` using insertion sort?

a. 6

b. 5

c. 7

d. 1

Solution

The answer is a. Suppose there are `N` keys in an input list, then it requires `N` iterations to sort the entire list using insertion sort.

Question 8

How will the array elements `A= [35, 7, 64, 52, 32, 22]` look after the second iteration, if the elements are sorted using insertion sort?

a. 7, 22, 32, 35, 52, 64

b. 7, 32, 35, 52, 64, 22

c. 7, 35, 52, 64, 32, 22

d. 7, 35, 64, 52, 32, 22

Solutions

d. Here `N = 6`. In the first iteration, the first element, that is, `A[1] = 35`, is inserted into array `B`, which is initially empty. In the second iteration, `A[2] = 7` is compared with the elements in `B` starting from the rightmost element of `B` to find its place. So, after the second iteration, the input array would be `A = [7, 35, 64, 52, 32, 22]`.

Chapter 12: Selection Algorithm

Question 1

What will be the output if the quickselect algorithm is applied to the given array `arr=[3, 1, 10, 4, 6, 5]` with `k` given as 2?

Solution

1. Given the initial array: `[3, 1, 10, 4, 6, 5]`, we can find the median of medians: `4` (index = 3).
2. We swap the pivot element with the first element: `[4, 1, 3, 10, 6, 5]`.
3. We will move the pivot element to its correct position: `[1, 3, 4, 10, 6, 5]`.
4. Now we get a split index equal to `2` but the value of `k` is also equal to `2`, hence the value at index `2` will be our output. Hence the output will be `4`.

Question 2

Can quickselect find the smallest element in an array with duplicate values?

Solution

Yes, it works. By the end of every iteration, we have all elements less than the current pivot stored to the left of the pivot. Let's consider when all elements are the same. In this case, every iteration ends up putting a pivot element to the left of the array. And the next iteration will continue with one element shorter in the array.

Question 3

What is the difference between the quicksort algorithm and the quickselect algorithm?

Solution

In quickselect, we do not sort the array, and it is specifically for finding the k^{th} smallest element in the array. The algorithm repeatedly divides the array into two sections based on the value of the pivot element. As we know, the pivot element will be placed such that all the elements to its left are smaller than the pivot element, and all the elements to the right are larger than the pivot element. Thus, we can select any one of the segments of the array based on the target value. This way, the size of the operable range of our array keeps on reducing. This reduces the complexity from $O(n\log_2(n))$ to $O(n)$.

Question 4

What is the main difference between the deterministic selection algorithm and the quickselect algorithm?

Solution

In the `quickselect` algorithm, we find the k^{th} smallest element in an unordered list based on picking up the pivot element randomly. Whereas, in the deterministic selection algorithm, which is also used for finding the k^{th} smallest element from an unordered list, but in this algorithm, we choose a pivot element by using median of medians, instead of taking any random pivot element.

Question 5

What triggers the worst-case behavior of the selection algorithm?

Solution

Continuously picking the largest or smallest element on each iteration triggers the worst-case behavior of the selection algorithm.

Chapter 13: String Matching Algorithms

Question 1

Show the KMP `prefix` function for the pattern `"aabaabcab"`.

Solution

The `prefix` function values are given below:

pattern	a	a	b	a	a	b	c	a	b
`prefix_function` π	0	1	0	1	2	3	0	1	0

Table A.2: Prefix function for the given patten

Question 2

If the expected number of valid shifts is small and the modulus is larger than the length of the pattern, then what is the matching time of the Rabin-Karp algorithm?

a. Theta (`m`)
b. Big O (`n+m`)
c. Theta (`n-m`)
d. Big O (`n`)

Solution

Big O (`n+m`)

Question 3

How many spurious hits does the Rabin-Karp string matching algorithm encounter in the text `T = "3141512653849792"` when looking for all occurrences of the pattern `P = "26"`, working modulo `q = 11` and over the alphabet set `Σ = {0, 1, 2,..., 9}`?

Solution

2.

Question 4

What is the basic formula applied in the Rabin-Karp algorithm to get the computation time as Theta (m)?

a. Halving rule
b. Horner's rule
c. Summation lemma
d. Cancellation lemma

Solution

Horner's rule.

Question 5

The Rabin-Karp algorithm can be used for discovering plagiarism in text documents.

a. True
b. False

Solution

True, the Rabin-Karp algorithm is a string matching algorithm, and it can be used for detecting plagiarism in text documents.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

`https://packt.link/MEvK4`

packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

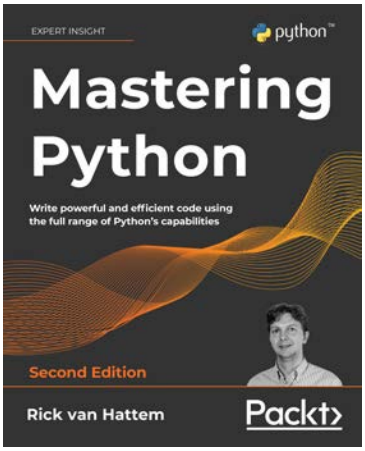
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



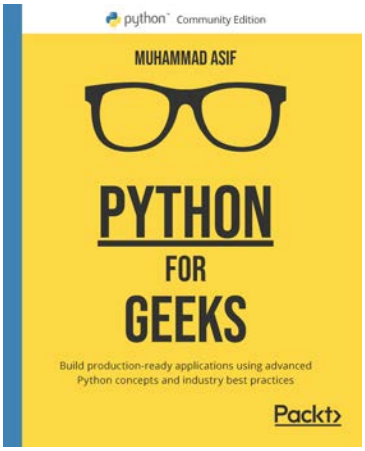
Mastering Python 2E

Rick van Hattem

ISBN: 978-1-80020-772-1

- Write beautiful Pythonic code and avoid common Python coding mistakes
- Apply the power of decorators, generators, coroutines, and metaclasses
- Use different testing systems like pytest, unittest, and doctest
- Track and optimize application performance for both memory and CPU usage
- Debug your applications with PDB, Werkzeug, and faulthandler
- Improve your performance through asyncio, multiprocessing, and distributed computing

- Explore popular libraries like Dask, NumPy, SciPy, pandas, TensorFlow, and scikit-learn
- Extend Python's capabilities with C/C++ libraries and system calls



Python for Geeks 2E

Muhammad Asif

ISBN: 978-1-80107-011-9

- Understand how to design and manage complex Python projects
- Strategize test-driven development (TDD) in Python
- Explore multithreading and multiprogramming in Python
- Use Python for data processing with Apache Spark and Google Cloud Platform (GCP)
- Deploy serverless programs on public clouds such as GCP
- Use Python to build web applications and application programming interfaces
- Apply Python for network automation and serverless functions
- Get to grips with Python for data analysis and machine learning

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Hands-On Data Structures and Algorithms with Python - Third Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z