a) Give at least three reasons as to why someone would use merge sort over quicksort ?

a. Mergesort has Θ(NlogN) worst case runtime versus quicksort's Θ(N2).
b. Mergesort is stable, whereas quicksort typically isn't.
c. Mergesort can be highly parallelized because as we saw in the first problem the left and right sides don't interact until the end. Mergesort is also preferred for sorting a linked list.

b) When will the Quicksort perform poorly (worst case) ?

Quicksort has a worst case runtime of Θ(N2), if the array is partitioned very unevenly at each iteration.

c) We have a system running insertion sort and we find that it's completing faster than expected. What could we conclude about the input to the sorting algorithm?\

Answer: Input already sorted.

d) Explain which sorting algorithm you would use to sort the input array the fastest and why you chose this sorting algorithm. An array of n Comparable objects that is sorted except for k randomly located elements that are out of place (that is, the list without these k elements would be completely sorted)

Sort: Insertion sort

- Runtime: O(nk)
- Explanation: For the n − k sorted elements, insertion sort only needs 1 comparison to check that it is in the correct location (larger than the last element in the sorted section). The re-maining k out-of-place elements could be located anywhere in the sorted section. In the worst case, they would be inserted at the beginning of the sorted section, which means there are O(n) comparisons in the worst-case for these k elements. This leads to an overall runtime of O(nk + n), which simplifies to O(nk).

a) Does this array represent a Max-Heap with the root at index 1? (Ignore the data at index 0.)

index	0	1	2	3	4	5	6	7	8	9	10	11	12
Data	12	78	50	30	8	65	22	28	4	6	12	2	13

Answer: Not a heap.

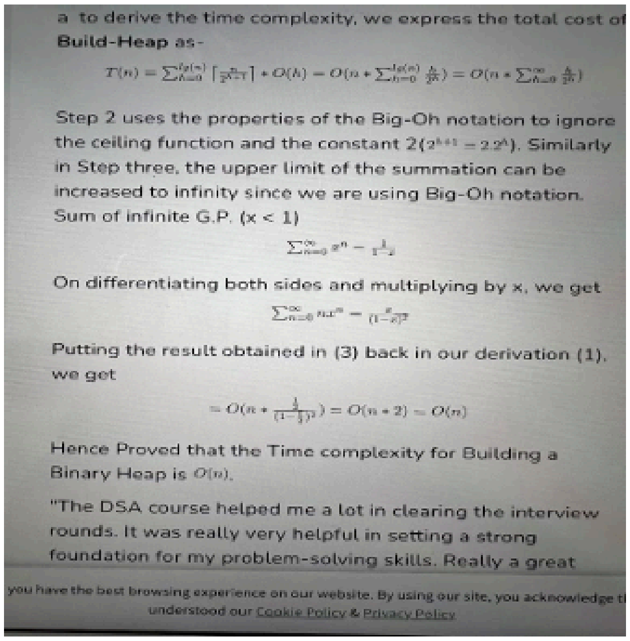
b) Consider 21 20_a 20_b 12 11 8 7 elements , show that whether the Heapsort is stable or not ?

Note: A sorting algorithm is stable if identical elements remain in their original order after sorting.

Answer: Stability for sorting algorithms mean that if two elements in the list are defined to be equal, then they will retain their relative ordering after the sort is complete. Heap operations may mess up the relative ordering of equal items and thus is not stable.

c) Consider the following algorithm for building a Heap of an input array A. Solve in step by step the time complexity of building a heap function below:

```
BUILD-HEAP(A)

   heapsize := size(A);

   for i := floor(heapsize/2) downto 1

      do HEAPIFY(A, i);

   end for

END
```



a to derive the time complexity, we express the total cost of

Build-Heap as-

$$T(n) = \sum_{h=0}^{lg(n)} \lceil \frac{n}{2^{h+1}} \rceil * O(h) = O(n * \sum_{h=0}^{lg(n)} \frac{h}{2^h}) = O(n * \sum_{h=0}^{\infty} \frac{h}{2^h})$$

Step 2 uses the properties of the Big-Oh notation to ignore the ceiling function and the constant 2($2^{h+1} = 2.2^h$). Similarly in Step three, the upper limit of the summation can be increased to infinity since we are using Big-Oh notation. Sum of infinite G.P. (x < 1)

$$\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$$

On differentiating both sides and multiplying by x, we get

$$\sum_{n=0}^{\infty} nx^n = \frac{x}{(1-x)^2}$$

Putting the result obtained in (3) back in our derivation (1), we get

$$= O(n * \frac{\frac{1}{2}}{(1-\frac{1}{2})^2}) = O(n * 2) = O(n)$$

Hence Proved that the Time complexity for Building a Binary Heap is $O(n)$.

"The DSA course helped me a lot in clearing the interview rounds. It was really very helpful in setting a strong foundation for my problem-solving skills. Really a great

Problem 1-2. Given a data structure `D` supporting the four first/last sequence operations: `D.insert_first(x)`, `D.delete_first()`, `D.insert_last(x)`, `D.delete_last()`, each in $O(1)$ time, describe algorithms to implement the following higher-level operations in terms of the lower-level operations. Recall that `delete` operations return the deleted item.

(a) `swap_ends(D)`: Swap the first and last items in the sequence in `D` in $O(1)$ time.

Solution: Swapping the first and last items in the list can be performed by simply deleting both ends in $O(1)$ time, and then inserting them back in the opposite order, also in $O(1)$ time. This algorithm is correct by the definitions of these operations.

```python
def swap_ends(D):
    x_first = D.delete_first()
    x_last  = D.delete_last()
    D.insert_first(x_last)
    D.insert_last(x_first)
```

(b) `shift_left(D, k)`: Move the first k items in order to the end of the sequence n `D` in $O(k)$ time. (After, the k^{th} item should be last and the $(k+1)^{\text{st}}$ item should be first.)

Solution: To implement `shift_left(D, 1)`, delete the first item and insert it into the last position in $O(1)$ time. The list maintains the relative ordering of all items in the sequence, except has moved the first item behind all the others, so `shift_left(D, 1)` is correct. Then to implement `shift_left(D, k)`, move the first item to the last position as above, and then recursively call `shift_left(D, k - 1)` until reaching base case `shift_left(D, 1)`. By induction, if `shift_left(D, k - 1)` is correct, moving the first item to the last position restores correctness. `shift_left(D, k)` runs in $O(k)$ time because it makes $O(k)$ recursive calls until reaching the base case, doing constant work per call.

```python
def shift_left(D, k):
    if (k < 1) or (k > len(D) - 1):
        return
    x = D.delete_first()
    D.insert_last(x)
    shift_left(D, k - 1)
```

Gardening company Wonder-Grow sponsors a nation-wide gardening contest each year where they rate gardens around the country with a positive integer[2] ***score***. A garden is designated by a ***garden pair*** (s_i, r_i), where s_i is the garden's assigned score and r_i is the garden's unique positive integer ***registration number***.

(a) [5 points] To support inclusion and reduce competition, Wonder-Grow wants to award identical trophies to the top k gardens. Given an unsorted array A of garden pairs and a positive integer $k \leq |A|$, describe an $O(|A| + k \log |A|)$-time algorithm to return the registration numbers of k gardens in A with highest scores, breaking ties arbitrarily.

Solution: Build a max-heap from array A keyed on the garden scores s_i, which can be done in $O(|A|)$ time. Then repeatedly remove the maximum pair k times using `delete_max()`, and return the registration numbers of the pairs extracted (say in an array of size k). Because a max-heap correctly removes **some** maximum from the heap with each deletion in $O(\log |A|)$ time, this algorithm is correct and runs in $O(|A| + k \log |A|)$ time.

(b) [5 points] Wonder-Grow decides to be more objective and award a trophy to every garden receiving a score strictly greater than a reference score x. Given a max-heap A of garden pairs, describe an $O(n_x)$-time algorithm to return the registration numbers of all gardens with score larger than x, where n_x is the number of gardens returned.

Solution: For this problem, we cannot afford $O(n_x \log |A|)$ time to repeatedly delete the maximum from A until the deleted pair has score less than or equal to x. However, we can exploit the max-heap property to traverse only an $O(n_x)$ subset of the max-heap containing the largest n_x pairs. First observe that the max-heap property implies all n_x items having key larger than x form a connected subtree of the heap containing the root (assuming any stored score is greater than x). So recursively search node v of the heap starting at the root. There are two cases, either:

- the score at v is $\leq x$, so return an empty set since, by the max-heap property, no pair in v's subtree should be returned; or

- the score at v is $> x$, so recursively search the children of v (if they exist) and return the score at v together with the scores returned by the recursive calls (which is correct by induction).

This procedure visits at most $3n_x$ nodes (the nodes containing the n_x items reported and possibly each such node's two children), so this procedure runs in $O(n_x)$ time.

Question 5

Explain with examples the best and worst case scenario of the Naive String matching algorithm. Marks will only be given for properly writing each step with notations.

Solution

Explanation

The Naïve String-matching algorithm tests all the possible place of a pattern (P) relative to a text (T).
We let the pattern length be **m** and the text length be **n**.
We try shift (s = 0, 1, 2, … n - m) and for each shift we compare T [s+1, s+2, … s + m] to P [1, 2, 3, … m].

Algorithm

```
n= T.length
m=P.length
for s=0 to n-m:
    for i=1 to m:
        if P[i]==T[s+i]:
        then print "Pattern occurs at shift", s
```

Example

Best Case:
In best case, the first element of the pattern does not match with the text in that shift so we do not check further (i.e. the inner loop never runs).

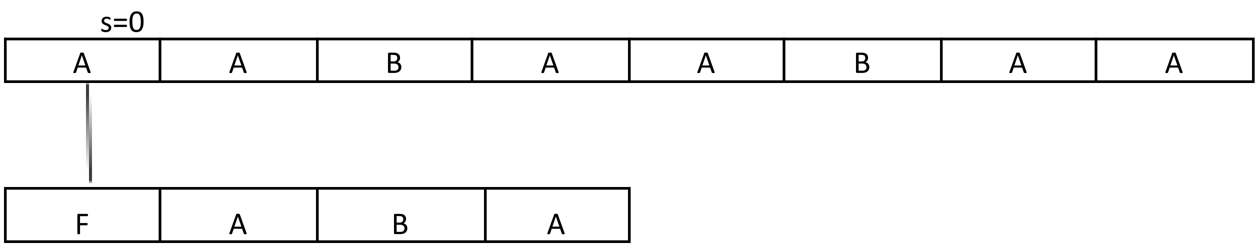
Let

T = A A B A A B A A   n = 8

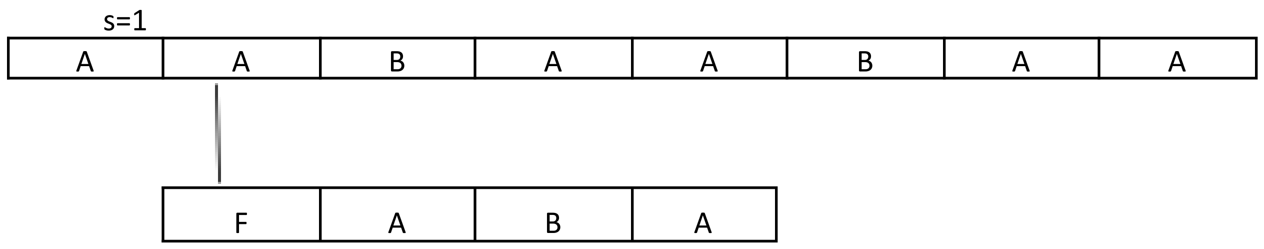
P = F A B A   m = 4

Now,

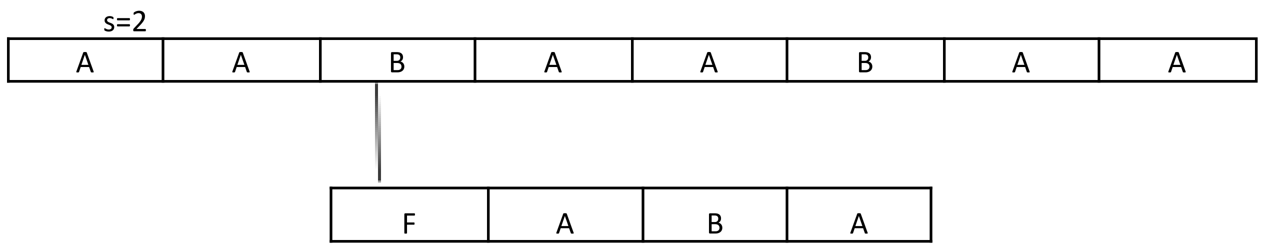
In the below notation, the black lines show when the loop for checking breaks and the shift is changed without any further checking.



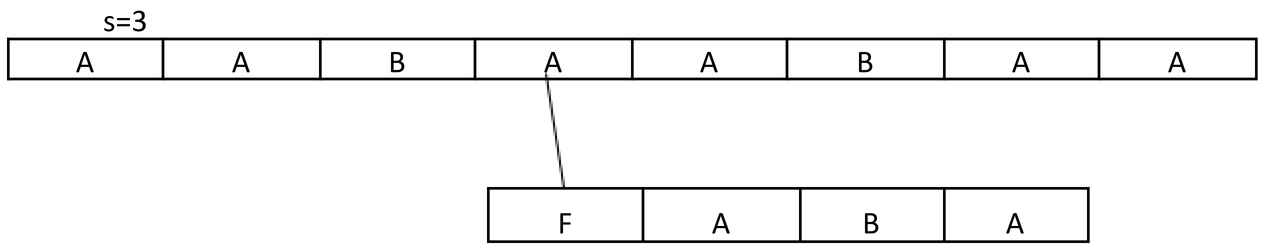
So, s=0 is not a valid shift.



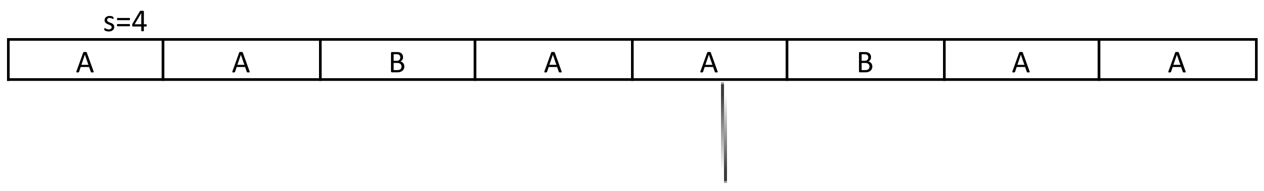
So, s = 1 is not a valid shift.

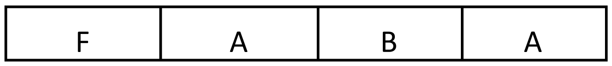


So, s = 2 is not a valid shift.



So, s = 3 is a valid shift.





So, s = 4 is not a valid shift.

As we can see, in each shift after the first mismatch the loop moves on to the next shift instead of checking for the rest of the elements of the pattern i.e. instead of running the inner loop.

The complexity of this best-case scenario is s = n – m +1 as s runs for 0, 1, 2, 3, 4 and inner loop runs 1 time each.

Best case: O(n – m + 1)

Worst Case:

The worst case of Naive Pattern Searching occurs in following scenarios.

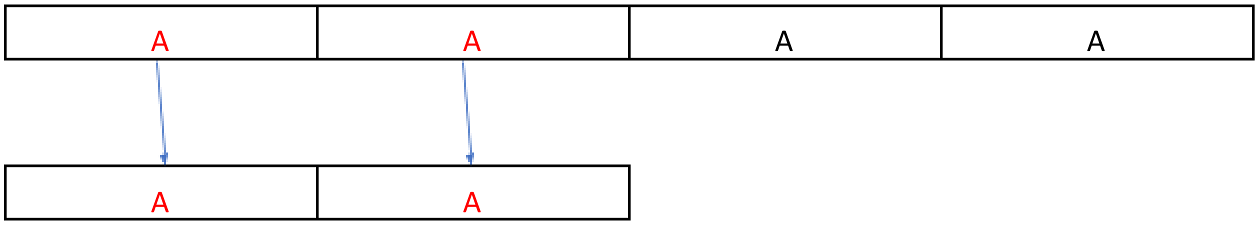
1. When all characters of the text and pattern are same.
2. When only the last character is different.

Let's consider the first case

1) T = AAAA n=4

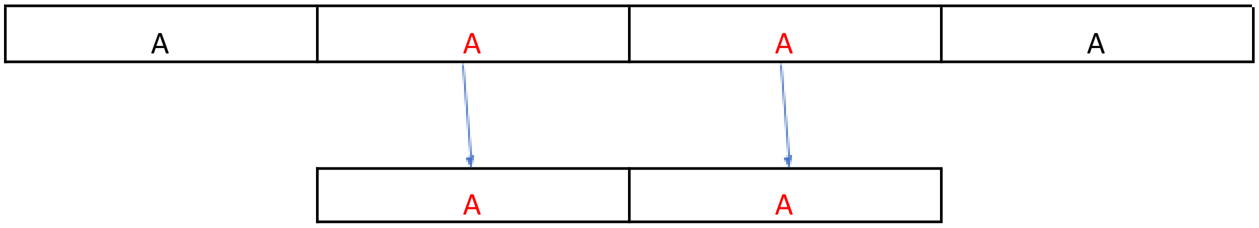
   P = AA m=2

s=0



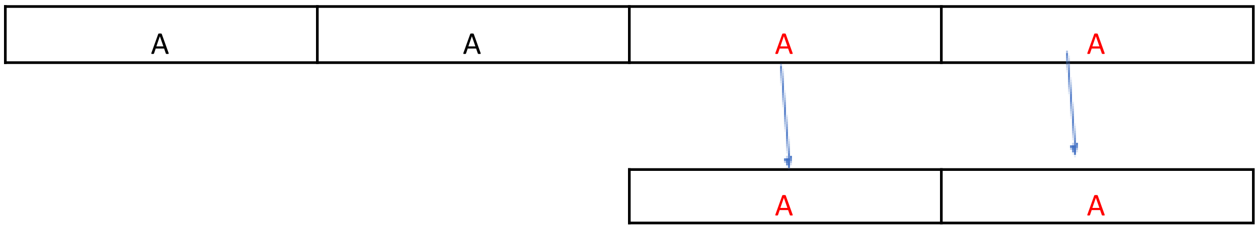
So, s=0 is a valid shift.

s=1



So, s=1 is a valid shift.

s=2



So, s=2 is a valid shift.

As we can see, each the time the shift changes the inner loop also runs the maximum amount of times i.e. m. So, the worst-case complexity is

Worst case: O((n – m + 1)*m)

Question 2:

We have studied multiple string matching algorithms in class and have found naive to be an inefficient algorithm.

- Can you suggest a solution to increase the efficiency of the naive algorithm?

  Naïve string-matching algorithm is inefficient because it does not store the information it has gained in one shift to use for another shift. It rechecks the whole pattern even if the text in the next shift is the same.

  We can increase the efficiency by applying a method in which information gained in the shift changes is stored and used effectively and rechecking of similar patterns is decreased.

  One method to do this is to use the KMP (Knuth Morris Pratt) String matching algorithm. This algorithm's basic idea is that whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match.

  The method to do this is by making a pie-table or a longest-proper-prefix (lps) which is also a suffix. This table is used to skip characters while matching. We search for lps in sub-patterns i.e. we focus on sub-strings that are either prefix or suffix.

- Will it be feasible for all the cases?

  The KMP string matching algorithm is feasible for patterns having the same sub-patterns appearing more than once in the pattern. For example, the pattern ABABC has a repeating sub-pattern AB.

- Explain the working of your suggested solution by writing code.

```
def LPSGeneration():
j=0
        lps[0]=0
        i=1
        while i < len(patterm :
                if lps[j] = = lps[i]:
                        lps[i] = j + 1
                        i=i+1
```

```
9.                  j=j+1
10.             else:
11.                 if  j != 0:
12.                     j = lps[j-1]
13.                 else:
14.                     lps[i]=0
15.                     i=i+1
```

Consider an example pattern to explain LPS Generation

Pattern: ABCDABD   length = 7

Intial:

Index	0	1	2	3	4	5	6
Position of i & j	j	i					
Pattern	A	B	C	D	A	B	D
LPS Array	0						

Loop run 1:

Index	0	1	2	3	4	5	6
Position of i & j	j		i				
Pattern	A	B	C	D	A	B	D
LPS Array	0	0					

Line 13 **else** statement ran.

Loop run 2:

Index	0	1	2	3	4	5	6
Position of i & j	j			i			
Pattern	A	B	C	D	A	B	D
LPS Array	0	0	0				

Line 13 **else** statement ran.

Loop run 3:

Index	0	1	2	3	4	5	6
Position of i & j	j				i		
Pattern	A	B	C	D	A	B	D
LPS Array	0	0	0	0			

Line 13 **else** statement ran.

Loop run 4:

Index	0	1	2	3	4	5	6
Position of i & j		j				i	
Pattern	A	B	C	D	A	B	D
LPS Array	0	0	0	0	1		

Line 6 **if** statement ran.

Loop run 5:

Index	0	1	2	3	4	5	6
Position of i & j			j				i
Pattern	A	B	C	D	A	B	D
LPS Array	0	0	0	0	1	2	

Line 6 **if** statement ran.

Loop run 6:

Index	0	1	2	3	4	5	6
Position of i & j	j						i
Pattern	A	B	C	D	A	B	D
LPS Array	0	0	0	0	1	2	

Line 11 **if** statement ran. (j = p [j – 1] = p [2 – 1] = p [1] = 0)

Index	0	1	2	3	4	5	6

Position of i & j	j						i
Pattern	A	B	C	D	A	B	D
LPS Array	0	0	0	0	1	2	0

Line 13 **else** statement ran.

The upper part was the pre-processing of KMP algorithm. Moving on to the main algorithm,

```
def KMPSearch():
    n=len(text)
    m=len(pattern)
    int LPS[m]
    LPSGeneration()
    while i < n:
        if pattern [j] == text[i]:
            i=i+1
        j=j+1
        if j == n:
            print ("pattern found")
        else:
            if  j != 0:
                j = LPS[ j - 1]
            else:
            i=i+1
```

- We start comparison of pattern[j] with j = 0 with characters of current window of text.
- We keep matching characters txt[i] and pattern[j] and keep incrementing i and j while pattern[j] and text[i] keep **matching**.
- When we see a **mismatch**
    - We know that characters pattern[0..j-1] match with text[i-j…i-1] (Note that j starts with 0 and increment it only when there is a match).
    - We also know that lps[j-1] is count of characters of pattern[0…j-1] that are both proper prefix and suffix.
    - From above two points, we can conclude that we do not need to match these lps[j-1] characters with text[i-j…i-1] because we know that these characters will anyway match.

Q6) **(KMP)**

Part a:

Given below is an arbitrary pattern:

"aabaabcab"

You have to show the KMP prefix function for the above pattern.

Solution

Index	0	1	2	3	4	5	6	7	8
Pattern	**a**	**a**	**b**	**a**	**a**	**b**	**c**	**a**	**b**
LPS Array	**0**	**1**	**0**	**1**	**2**	**3**	**0**	**1**	**0**

Part b:

You are required to write a program in C++ which will show how you achieved the KMP prefix function for part A.

Solution

```
def LPSGeneration():
j=0
        lps[0]=0
        i=1
        while i < len(patterm :
                if lps[j] = = lps[i]:
                        lps[i] = j + 1
                        i=i+1
                        j=j+1
                else:
                        if  j != 0:
                                j = lps[j-1]
                        else:
                                lps[i]=0
                                i=i+1
```

Q7) **(Rabin-Karp)**

Part a:

For this question, you have to mention the number of spurious hits encountered if we run the Rabin-Karp algorithm on the text given below.

Text: 3141592653589793

Assume you are looking for the pattern:

P = 26

working modulo: q=11

Solution

n = length of text = 16
m= length of pattern = 2

s is shift till n – m (14 shifts)

HashP = 26 mod 11 = 4

HashT
s=0 31 mod 11 = 9
s=1 14 mod 11 = 3
s=2 41 mod 11 = 8
s=3 **15 mod 11 = 4**
since, pattern ! = text: spurious hit

s=4 **59 mod 11 = 4**
since, pattern ! = text: spurious hit

s=5 **92 mod 11 = 4**
since, pattern ! = text: spurious hit

s=6 **26 mod 11 = 4**
since, pattern == text: successful hit

s=7 65 mod 11 = 10
s=8 53 mod 11 = 9
s=9 35 mod 11 = 2
s=10 58 mod 11 = 3
s=11 89 mod 11 = 1
s=12 97 mod 11 = 9
s=13 79 mod 11 = 2
s=14 93 mod 11 = 5

While looking for pattern P = 26 total 3 spurious hits will occur.

Part b:

You are required to write a program in C++ to code your solution of part A.

Solution

```
def RabinKarp():
n = len(text)
m = len(pattern)
hashP = Hash (pattern[])
hashT = Hash (text[])
for  s in range(n-m):
    found=true
    if hashP == hashT:
        for i in range(m):
            if pattern[i] != text[s+m-1]:
                found=false
        else:
            found=false
        if found ==true:
            print (“Pattern found”)
        if  s < n - m:
            hashT = Hash(T [s+1…s+m])
```

Q8)

Discuss how to extend the Rabin-Karp method to handle the problem of looking for a given m x m pattern in an n x n array of characters (The pattern may be shifted vertically and horizontally, but it may not be rotated.)

Solution

One way to do this can be to calculate the hash of each of the rows; this would return us m hashes for the pattern matrix.
In the n x n text matrix, do the same for the first m x m submatrix.
You have a submatrix hit if all m-row hashes are the same.
Otherwise move the pattern matrix to the right one column and compute the new hash for the m x m submatrix in text matrix.
Don't throw away these intermediate hashes since when you move down one row, you will have to recompute some of them again unless you save them.
Horizontal: Just as in the 1D-array case for RABIN-KARP, the new hash computations just need the the most significant character removed and the least one added into the hash.
Vertical: Compute a new set of hashes in the text matrix for the new rows and use the old saved hashes as intermediates for repeating rows.

Q9

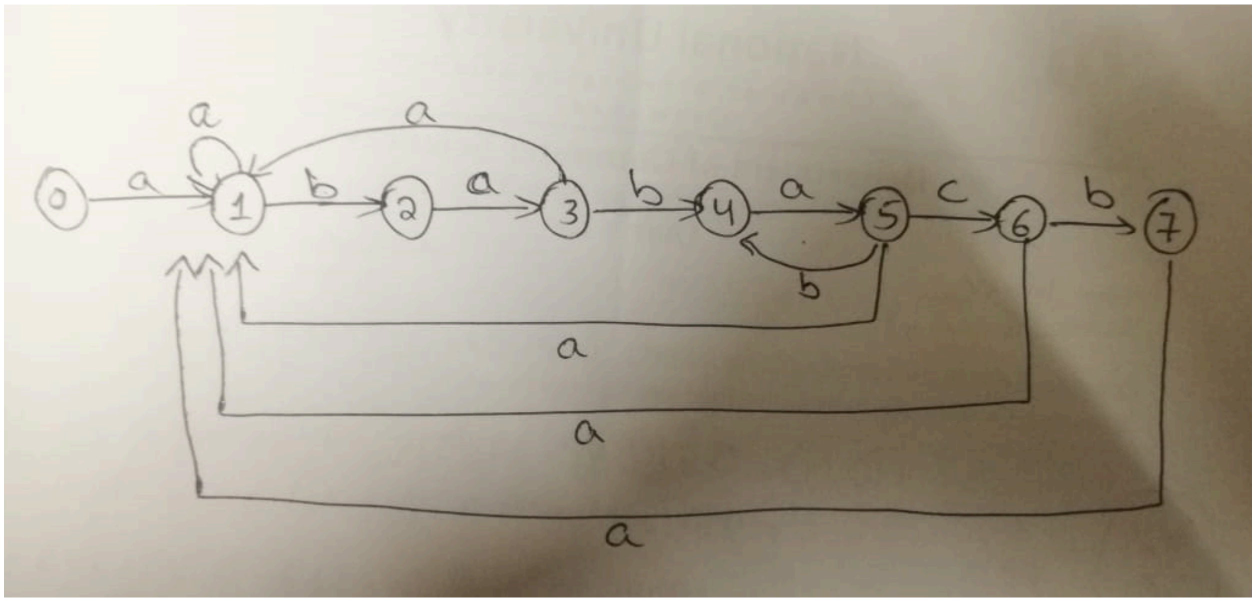
Part C

Question 1:

Design a finite automata machine along with state table for string matching that accepts all strings ending the form

ababacb

Solution



State Table

	a	b	c
0	1	-	-
1	1	2	-
2	3	-	-
3	1	4	-
4	5	-	-
5	1	4	6
6	1	7	-

7	1	-	-

Question 2:

Prove that string S is a substring of string A by drawing a state diagram o.f S, and using Finite Automata.

S: a b a b a d c a

Solution

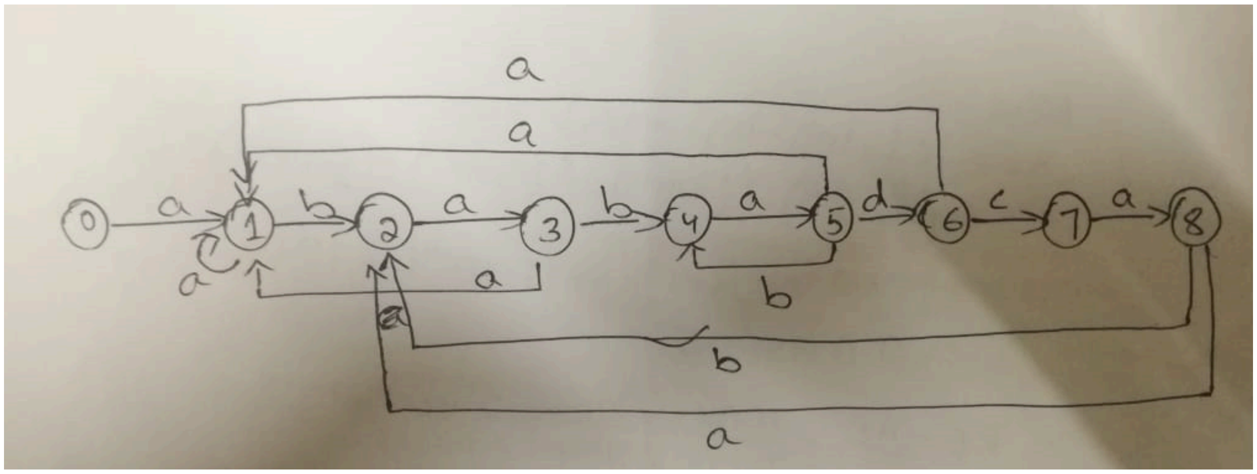


Table of State Diagram

	a	b	c	d
0	1	-	-	-
1	1	2	-	-
2	3	-	-	-
3	1	4	-	-
4	5	-	-	-
5	1	4	-	6
6	1	-	7	-
7	8	-	-	-
8	2	2	-	-

(i)

A: a b a b a b d c a a b a b a d c a

Table of Solution Set

A	**State**
a	1
b	2
a	3
b	4
a	5
b	4
d	4
c	4
a	5
a	**1**
b	**2**
a	**3**
b	**4**
a	**5**
d	**6**
c	**7**
a	**8 (String matched)**

(ii)

A: a b a b a d c a a b a b a d c a d

Table of Solution Set

A	**State**
a	1
b	2
a	3
b	4
a	5
d	6
c	7
a	8 (String matched)
a	2
b	2
a	3
b	4
a	5
d	6
c	7
a	8 (String matched)
d	8

(iii)

A: a b a b a b d c b a b a b a d c b

Table of Solution Set

A	**State**
a	1
b	2
a	3
b	4
a	5
b	4
d	4
c	4
b	5
a	1
b	2
a	3
b	4
a	5
d	6
c	7
b	7