Revision :

key indexed counting :

0 - 255

or let us consider

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

Lets take an array like ①

① d 3
f 5
a 0
a 0
c 2
e 4
e 4
e 4
f 5
d 3
d 3
b 1
c 2
b 1
a 0

Now create an array that represents the alphabets as its index and add up the count in each index.

We will limit to a, b, c, d, e, f

Let the 0th index be unpopulated

0	
d	
f	

0	
a	3
b	2
c	2
d	2
e	3
f	2

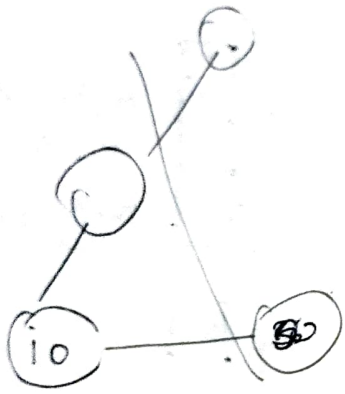
After adding the counts now cumulate it

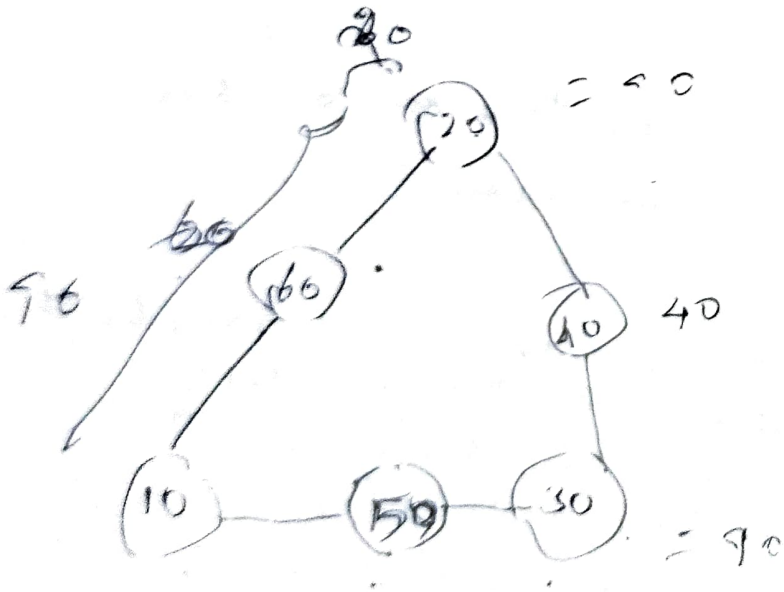
0	
a	3
b	5
c	7
d	9
e	12
f	14

Now there are 3 chars lesser than b

5 < c 2 > a

```
int[] a = new int [array.length]
int[] count = new int [R+1]
for (int i=0; i<a.length; a++)
    count[a[i]+1]++
for (int j=0; j<count.length; j++)
    count[j+1] = count[j]+count[j-1]

for (int k=0; k<aux.length; k++)
    aux[count[a[k]]++] = a[k]
aux[7] =
```





20, 40, 50, 60

0	1	2	3	4	5	6	7	8	9

~~9~~	~~1~~	~~9~~	~~3~~	~~4~~	~~5~~	~~9~~	~~7~~	~~8~~	~~9~~

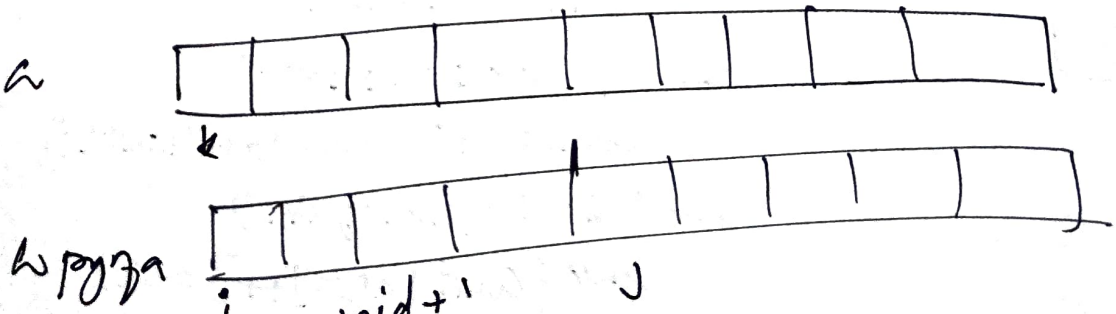
4	1	2	3	6	5	9	~~8~~	~~8~~	9

find root(0)

```
{ id[0] == 0
    return 0
else
    0's root is 4
    4's root is 6
    6's root is 9
    9's root is 9
while (i != id[i]) {
    i = id[i]
}
return i
```

Linked list

Merge Sort




```
k=0 i=0 j= mid+1
{ i<j
    move copya[i] -> a[k]
    i++;
    k++
else
    move copya[j] - a[k]
    j++
    k++
```

Key indexed counting demo:

Goal: Sort an array a[] of N integers between 0 and R-1

- Count frequencies of each letter using key as index
- Compute frequency cumulates which specify distribution
- Access cumulates using key as index to move items.
- Copy back into original array

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

use a for 0
b for 1
c for 2
d for 3
e for 4
f for 5

```
int N= a.length;
int[] Count = new Count [R+1];
for(int i=0; i<N; i++)
    Count[a[i]+1]++;
for(int r=0 ; r<R; r++)
    Count[r+1]= Count[r+1]+Count[r]
for (int i=0; i<N; i++)
    aux[Count [a[i]]++] =a[i]
for(int i=0; i<N; i++)
    a[i]=aux[i]
```

LSD Radix sort.

least-significant-digit-first string sort.

Consider characters from right to left

stably sort using d^{th} character as the key (using key-indexed counting)

```
Public class LSD {

    public static void sort(String[] a, int w)
    {
        int R = 256
        ~~int[] count~~
        int N = a.length;
        String aux[] = new String[N];
        for (int d = w; d >= 0; d--)
        {
            int[] count = new int[R+1];
            for(int i=0; i<N; i++) {
                ~~count[a[i]+1]++~~
                ~~count[a[i].charAt(d)]++~~
                count[a[i].charAt(d)+1]++;
            }
            for (int r=0; r<R; r++) {
                count[r+1] = count[r+1] + count[r]
            }
            for (int i=0; i<N; i++) {
                ~~aux[count[a[i]]++] = a[i]~~
                aux[count[a[i].charAt(d)]++] = a[i]
            for (int i=0; i<N; i++) {
                a[i] = aux[i];
            }
        }
```

Most Significant digit first Sort

MSD String Sort

Study again

```
public static void sort(String[] a)
{
    aux = new String[a.length]
    Sort = (a, aux, 0, a.lenght-1, 0);
}
private static void sort(String[] a, String[] aux,
                         int lo, int hi, int d)
{
    if (hi<=lo) return;
    int[] count = new int[R+2];
    for(int i=lo; i<=hi; i++) {
        count(a[i].charAt(d))++
```

~~R-Way Tries~~

Ternary search tries:

store characters and values in nodes (not keys)

Each node has 3 children

Smaller (left), equal (middle), larger (right)

A TST node is 5 fields

A value

A character c

A reference to a left TST

A reference to a right TST

A reference to a middle TST

```
Private class Node {
  private Value val;
  Private Char c;
  private Node left, mid, right;
}
```

```
public class TST<value>
{
  private Node root;

  private class Node { }

  Public void put (String key, Value val) {
      root= put (root, key, val, 0);
  }

  private Node put (Node x, String key, Value val, int d )
  {

      char c = key . charAt(d);
      if (x == null) { x = new Node(); x.c = c; }
      if (c < x.c) x.left = put (x.left, key, val, d);
      if (c > x.c) x.right = put (x.right, key, val, d);
      else if (d < key.length()) x.mid = put (x.mid, key, val, d+1)
      else x.val = val

      return x;
  }
```

```
public boolean contains (String key)
{
    return get(key) != null;
}

public Value get (String key)
{
    Node x = get (root, key, 0);
    if (x == null) return null;
    return x.val;
}

private Node get (Node x, String key, int d) {
    if (x == null) return null;
    char c = key.charAt (d);
    if (c < x.c)  return get (x.left, key, d);
    if (c > x.c)  return get (x.right, key, d);
    if (d < keylength()-1) return get (x.mid, key, d+1);
    else return x;
}
```

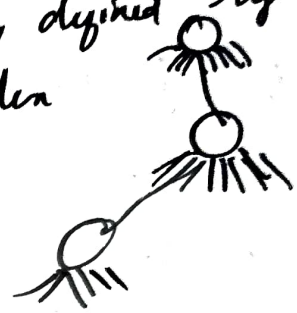
<u>Ordered Iteration</u> :

```
public Iterable<String> keys() {
    Queue<String> queue = new Queue<String>();
    Collect (root, "", queue);
    return queue; }
private void collect (Node x, String prefix, Queue<String> q) {
    if (x == null) return;
    if (x.val != null) q.enqueue (prefix);
    for (char c=0; c< R; c++)
        collect (x.next[c], prefix+c, q)
}
```

R. Way Tries:

```
private static class Node {
    private Object value;
    private Node[] next = new Node[R];
}
```

Characters are implicitly defined by link index



```
public class TrieST<value>
{
    private static final int R = 256;
    private Node root = new Node();

    private static class Node { }

    public void put (String key, Value val) {
        root = put(root, key, val, 0); }

    private Node put (Node x, String key, Value val, int d) {
        if (x == null) x = new Node();
        if (d == key.length) { x.val = val; return x; }
        char c = key.charAt(d);
        x.next[c] = put(x.next[c], key, val, d+1)
        return x;
    }
    public boolean contains (String key) { return get(key) != null; }
    public Value get (String key) {
        Node x = get(root, key, 0);
        if (x == null) return null;
        return (Value) x.val; }
    private Node get (Node x, String key, int d) {
        if (x == null) return null;
        if (d == key.length()) return x;
        char c = key.charAt(d);
        return get(x.next[c], key, d+1)
```

Substring Search:

Goal: find pattern of lenght M in a text of length N
typically N>>M

Pattern → NEEDLE

text → IN A HAYSTACK $\underbrace{\text{NEEDLE}}_{\text{Match}}$ IN A

Brute force:

i	j	i+j	0	1	2	3	4	5	6	7	8	9	10
	text		A	B	A	C	A	D	A	B	R	A	C
0	2	2	A	B	R	A	← Pattern						
1	0	1		A	B	R	A						
2	1	3			A	B	R	A					
3	0	3				A	B	R	A				

```
public static int search (string text, string pat) {
    int M= pat.length();
    int N= text.length();
    for(int i=0; i< N-M; i++) {
        int j;
        for(j=0; j< M; j++) {
            if (text.charAt(i+j) != pat.charAt(j))
                break;
        if(j==M) return i;
    }
    return N
}
```

Boyer-Moore Substring Search - mismatched character Heuristic

Intuition:

- Scan characters in pattern from right to left
- Can skip as many as M text chars when finding not one in the pattern

i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	5																								

. . . T L E . . .
N E E D L E

How much to skip:

N E E D L E

Case 1: Mismatch character not in pattern

increment i one character beyond 'T'

Case 2a: Mismatch character in pattern.

before text . . . . . N L E . . . . .
pat N E E D L E

after text . . . . . N L E . . . . .
N E E D L E

mismatch character 'N' in pattern; align text 'N' with rightmost pattern 'N'.

Boyer Moore Skip table computation:

c		N 0	E 1	E 2	D 3	L 4	E 5	right[c]
A	-1							-1
B	-1							-1
C	-1							-1
D	-1	-1	-1	-1	3	3	3	3
E	-1	-1	1	2	2	2	5	5
...								
L	-1							
M	-1							
N	-1	0	0	0	0	0	0	
...								

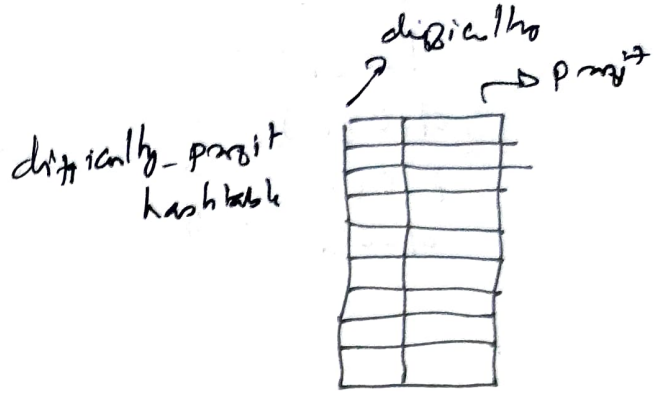
```
right = new int[R];
for (int c=0; c<R; c++)
    right[c] = -1;
for (int j=0; j<M; j++)
    right[pat.charAt(j)] = j;

public int search (String txt)
{   int N = txt.lenght();
    int M = pat.length();
    int skip;
    for (int i=0; i<N-M; i+=skip){
        skip=0;
        for (int j=M-1; j>=0; j--){
            if (pat.charAt(j) != txt.charAt(i+j)){
                skip= Math.max(1, j-right[txt.charAt(i+j)]);
                break;
            }
        }
        if (skip==0) return i;
    }
    return N;
}
```

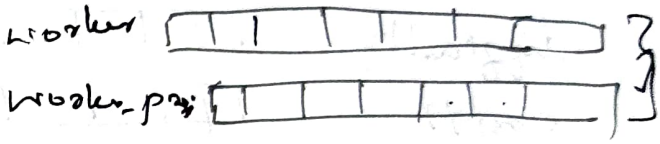
```
private static void merge (Comparable[] a,
                           Comparable[] aux,
                           int lo, int mid, int hi)
{
  for (int k = lo; k <= hi; k++) {
    aux[k] = a[k]; }

  int i = lo, j = mid+1;
  for (int k = lo; k <= hi; k++) {

    if (i > mid) a[k] = aux[j++]
    else if (j > hi) a[k] = aux[i++]
    else if (aux[j] < aux[i]) a[k] = aux[j++]
    else a[k] = aux[i++];
}

Public class Merge {
  private static void merge(...) { }
  private static void sort(Comparable[] a,
                           Comparable[] aux,
                           int lo, int hi)
      if (hi <= lo) return;
      int mid = lo + (hi - lo)/2;
      sort(a, aux, lo, mid);
      sort(a, aux, mid, hi);
      merge(a, aux, lo, mid, hi);
  }
  public static void sort(Comparable[] a) {
      aux = new Comparable[a.length]
      sort(a, aux, 0, a.length-1)
```

Jan-6-2020

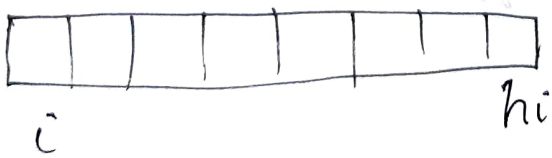
Algorithms :

We have jobs difficulty[i] is the difficulty of the ith job and profit[i] is the profit of ith job.






```
int sum=0;
for (int i=0; i<worker.length; i++) {
    int profit=0
    for (int j=0; j<difficulty.length; j++) {
        if (worker[i] == difficulty[j]) {
            worker_profit[i] = profit[j]
            profit = profit[j]
            sum += profit[j];
            continue; }
        ~~if (difficulty[j] > worker[i]~~
        if (difficulty[j] > worker[i]) {
            worker_profit[i] = 0
            profit = 0;
            sum += 0;
        } else {
            profit = max(profit, difficulty[j]
        }
    worker_profit[i] = profit
    sum = profit
}
```




```
int sum=0
for (int i=0; i<worker.length; i++) {
    int profit = -∞;
    for (int j=0; j<; j++) {
        if (difficulty_pro.contains(worker[i]) {
            worker.profit[i] =
                difficult_pro.get(worker[i]);
            continue;
        profit = max(profit, ~~worker[i]~~);
        if (difficulty[j] > worker[j])
            profit = 0
            continue;
        profit = max(profit, difficulty[j]);
    }
    worker profit[i] = profit
    ~~sum = profit~~
}
```

difficulty = [2, 4, 6, 8, 10]

for i = 0 to difficulty.length - 1

difficulty_indices[difficulty[i]] = i

→ i=0

d_i[2] = 0

d_i[4] = 1

d_i[6] = 2

d_i[8] = 3

d_i[10] = 4

Worst approach

~~while (true) {~~

~~int k=0 int z=0~~

~~while (copy_difficulty[z++] <= worker[k]~~

int k=0;

int z=0;

while (true) {

~~while (copy_difficulty[z++]~~

{ (copy_difficulty[z++] ≤ worker[k]) {

min = Math.min(min, copy_difficulty[z] - wk[k])

1) Make a copy of the difficulty matrix

2) Sort difficulty matrix

3) Binary search Check if worker ability in sorted difficulty matrix if not decrement ability and replace the original worker ability to the nearest one in sorted difficulty array which is lesser in value

4) If there are worker abilities that are less than in the difficulty array, make them zeros

5) Create a worker_profit array and populate the corresponding profit

i=0 j=5

2 4 6 8 10 12

↑ 5

0 i++ j--

↑ ↑4

i++ ↑ ↑3 j--

1 i++ j--

2

Given an array of n positive integers and a positive integer s, find the minimal length of a contiguous sub array of which the sum ≥ s. If there isn't one return 0

s = 7

nums :- 2 3 1 2 4 3

Done

2. Add two numbers

You are given two non-empty linked lists representing two nonnegative integers. The digits are stored in a reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

You may assume there are no leading zeros

$2 \to 4 \to 3 \quad + 5 \to 6 \to 4$

$\Rightarrow$ 465
+ 342
807

output $7 \to 0 \to 8$

1) ~~l1.val~~ $l_1 + l_2$!= null

$l_1.val + l_2.val < 9 \Rightarrow$ Add to the result linked list

$l_1.val + l_2.val > 9 \Rightarrow$ carry = 1

Result = Sum − 10

ii) If l_1 == null && l_2 != null

while(l_2.next != null)

Send l_2 l_2.next

iii) If l_2 == null && l_1 != null

Send l_1 l_1.next

4) If l_1 & l_2 == null & Carry != 0

insert the carry into the listNode

Add two numbers :

Input $(7 \to 2 \to 4 \to 3) + (5 \to 6 \to 4)$

output $7 \to 8 \to 0 \to 7$

```
while (l1.next != null
       || l2.next != null)
{
```

Aim:

To align the two numbers

Take two numbers

iterate / recurse over the two numbers until both the numbers have next number

If one of the numbers ends sooner we will have to roll back and offset the smaller number by 1 pos

How to identify the smaller number

l_1, l_2

offset

~~while (l1.next && l2.next)~~

```
while (true) {
    l1 = l1.next;
    l2 = l2.next;
    if (l1 == null || l2 == null)
        break
}
```

l_1 l_2

$l_1.next$ $l_2.next$

```
if(l1.next == null || l2.next == null) {

}

if (l1.next == null || l2.next != null)
```

```
addTwonumbers(listNode l1, listNode l2) {
    if (l1.next == null || l2.next == null) return true;
    ~~if(l1.next != null) l1=l1.next~~;

while (true) {

        addTwonumbers (l1, l2.next);
        if(l1.next != null) l1 = l1.next
}
```

7 → 2 → 4 → 3 5 → 6 → 4

1 4 5 6 7 5 5 13 13 20 22

4 5 13 16 20 22

Median

5 4 7 5 10 12 13

Mean

1 3 7 9 11 13 15 19 22

1 3 7 9 11 13 15 19 22

17/2 = 8.5

1 3 7 5 11

1 3 4 6 8 13

1 3 4 6 7 8 9 11 13

Leet Code

Algorithms

1. Two Sum ✓ 14 — 15/9/20
2. Add two numbers ✓ 15 — 15/9/20
3. Longest substring without repeating characters 15 — 15/9/20
4. Median of two sorted arrays 19 24/9/20
5. Longest Palindromic Substring 24/09
6. Zig Zag Conversion 3-Jan
7. Regular Expression Matching 19-Feb
Dynamic Programming

Median of two sorted arrays

0 1 2 3 4

1 3 5 | 7 9

Median = 5

0 1 2 3 4 5 6

23 25 27 | 29 31 33

Median = 28

23 25 27 29 31 33

1 3 5 7 9 23

Median of 1 5

Median of 2 $\frac{27+29}{2} = \frac{56}{2} = 28$

Partition x

1 3 5 7 9 (23) 25 27 29 31 33

~~If left~~

If $L_x \le R_y$ && $L_y \le R_x$

If $L_x > R_y$

~~1 3 5 7 9 23 25 27 29~~

18/

~~1 3 5 7 9~~

1 5 7 11 14 16 = $18/2 = 9$

2 4 8 12 18 20 = $20/2 = 10$

$\frac{13}{2}$

Median of two sorted Arrays

1 2 4 5 7 8 11 12 14 16 18 20

9.5

$\frac{6}{2}$ 9 9

is the median

$\frac{5}{2} = 3$

5 Nos

0 1 2

1 5 7 11 14 16

2 4 8 12 18 20

5 Nos

1 5 9 11 14 16

2 4 8 12 18 20

5

$\frac{5}{2} = 2.5 = 2$

1 2 4 5 8 9 11 ~~12 14 16~~

1 5 7 11 14 16

2 4 8 12 18 20

$\left(\frac{x+y+1}{2}\right) = 6$

Partition of both Arrays combined

$6 - 2 = 4$

1 3 5 7 9 13 15 17 20 22 25

↳ 10/2=5

4 6 8 10 12

1/2

=5

15 - 5 = 11

1 3 4 5 6 7 8 9 10 12 13 15 17 20 22 25

7.2=5

2

2

7.5=2

0 4 1 6 2 8 3 10 4 12 5 6 7 8 9 10 11

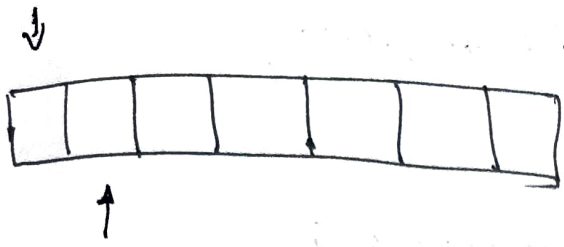
1 3 5 7 9 13 15 17 20 22 25

Split this into two groups so that each group has the same number of elements

$LX \leq RY$ && $LY \leq RX$

Return

$LX > RY$



BB

ABBA

RAMAR

DAMAD

MADAM

CIVIC

ROTOR

↑

```
int lo = 0
int hi = S.length-1
ispal  int fwdPtr =  lo + hi/2
while (true) {
  if (S.length % 2 != 0){
      int fwdPtr = 0+hi
      /2
int lo=0;
int hi = S.length() - 1;
if (S.length() % 2 == 0) {
    int backwardPtr = Math.floor(S.length/2);
    int fwdPtr = backwardPtr+1;
} else {
    int backwardPtr = S.length/2;
    int fwdPtr = backwardPtr;
}
while (true) {
    backwardptr --;
    forwardPtr ++;
if (S.charAt(backwardPtr) == S.charAt(forwardPtr)) {
        backwardPtr --;
        forwardPtr ++;
} else {
        break;
};
```

int mid = 0

length == 2

leng > 2

ispalindrome (String)

length of the string

low = 0

high = length of the string -1

partition $\frac{0-4}{2} = 2$

fwdPointer

bwd Pointer

RAMMAR length = 6

low = 0

high = 5

int lo = 0; int bptr, fptr = 0;

int hi = S.length() - 1;

if (S.length() % 2 == 0) {

bptr = Math.floor(S.length/2);

fptr = bptr + 1;

} else {

bptr = S.length()/2;

fptr = bptr;

}

isPalindrom(S, bptr, fptr);

~~public boolean isPalindrom (String S) {~~

public boolean isPalindrom (String S, int bptr, int fptr)

{ (bptr < 0 && fptr == S.length) return true;

~~if (S.charAt(bptr) == S.charAt(fptr)) {~~

boolean isPali = true;

if (S.charAt(bptr) == S.charAt(fptr)) {

return isPalindrom(S, bptr--, fptr++);

} else {

isPalindrom = false;

return false;

}

return isPalindrom;

R A R A M A R A D

string maker

— start = 0

for int i = 1 i < S.length i++

= S.substring (0, i)

1) String feeder :-

Feed strings of varying length to the palindrome finder

2) Palindrome finder :-

Figure out even/odd length of the string

a) Odd :- Check if

Median fwdPtr++ equals

Median bwdPtr-- equals

Eg (median == fwdPtr++ || median = bwdPtr--

|| fwdPtr++ != bwdPtr--)

else if (fwdPtr == bwdPtr

Eg (fwdPtr == fwdPtr++ || bwdPtr == bwdPtr--

|| fwdPtr++ != bwdPtr--)

b) Even

```
public String getPalindrom (String s, int fwd Ptr,
                            int bwd Ptr)
{ if (s.length() == 2 && s.charAt(0) == s.charAt(1))
                    return s;

  if (s.length() % 2 == 0) {
      if (s.charAt(fwd Ptr) == s.charAt(bwd Ptr)) {
```

0 1 2 3 4   0 1 2 3 4 5

1 1 1 1 1   1 1 1 1 1 1

1 2 3 4 5   1 2 5 9 5 6

```
for (int i=0; i<s.length; i++) {

    int median = i+1;
    String candidateString = ~~s.substring(~~
    ~~String candidateString~~

    if (median >= s.length()/2) {
        candidateString = s.substring(
                                  median - (s.length() - median))
                                  median + (s.length() - median))
    else
        candidateStr = s.sub(median - (median - 1), median + median - 1)
    }
```

s.substring(0, 1)

i = (median - 1)

i = 3

i = 2

1 2 3 4 5 6 7 8 9 10

1 1 1 1 1 1 1 1 1 1

0 1 2 3 4 5 6 7 8 9

1 1 1 1

10/2 = 5

11/2 = 5

1 2 3 4

s.length = median

10 - 6 = 4

```
public String getPalindrome (String s, int bwdPtr, int fwPtr)
String PalindromeString = "";
if (s.length() == 1) return "";

if (s.length() == 2 && s.charAt(0) == s.charAt(1)) return s;
while (true) {
    if (s.length % 2 == 0) {
        if (s.charAt(bwdPtr) == s.charAt(fwdPtr)) {
            PalindromeStr = s.substring(bwdPtr, fwPtr);
            ~~PalindromeString~~ =
            bwdPtr = bwdPtr - 1;
            fwPtr = fwdPtr + 1;
        else
            break;
    else
        if (s.charAt(bwdPtr) == s.charAt(fwPtr)) {
            palindrStr = s.substr(fwPtr, fwdPtr)
        } else {
            break;
    }
return PalindrStr;
```

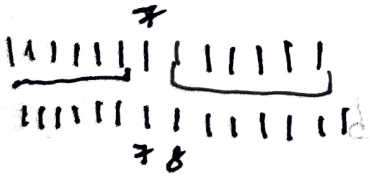
if the media is

2

21

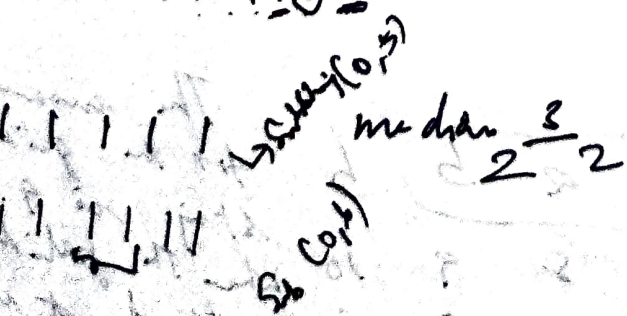
3 length = 5

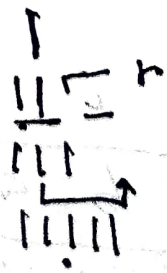
4

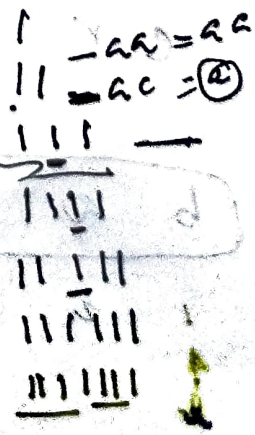


= leng = 20 $21/2 = 10$

= leng = 21 $22/2 = 11$







0 1 2 3 4 5 → Indices

| a | b | a | a | b | a |

0 1 2 3 4 5 6 7 8 9 10 11 12 → Positions

LPS length value can be interpreted in two ways, one in terms of index and second in terms of positions.

LPS value d at position i L(i) = d

Substring from position i-d to i+d is a palindrome of length d (in terms of ~~index~~ position)

Substring from index $i-d/2$ to $i+d/2-1$ is a palindrome of length d (in terms of index)

Lets focus on odd length palindromes for simplicity. We go about the string from left to right

Notations:- let 'c' be the center of the longest path length palindrome. Let 'l' and 'r' be the left and right boundaries of this palindrome i.e the left most character index and the right most character index respectively. Now lets take an example to understand c, l and r.

Eg: "abacabacabb"

When going from left to right, when i is at index 1, the longest palindrome substring is "aba" length = 3

i

a b a c a b a c a b b

0 1 2 3 4 5 6 7 8 9 10

l c r

i

a b a c a b a c a b b

0 1 2 3 4 5 6 7 8 9 10

l c r

The answer for the given string is 9 when the palindrome is centered at index 5

c, l and r are

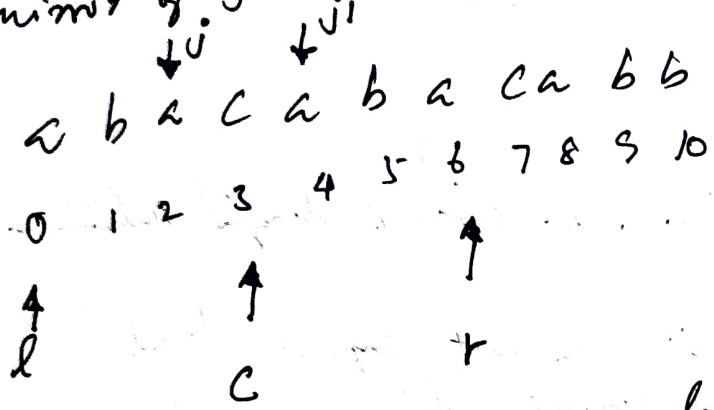
Palindrome properties:

For any palindrome centered at a center 'c', the mirror of index j is j' such that

$$c - j = j' - c$$

So mirror index of j : $j' = (2 \times c) - j$

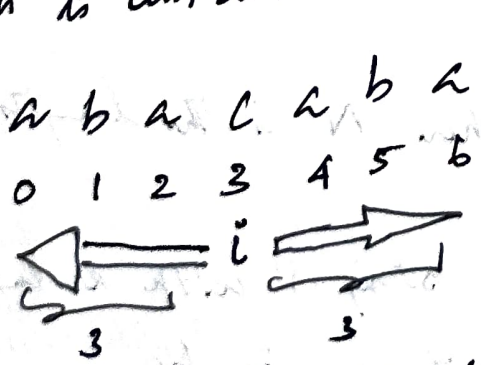
For palindrome "abacaba", the mirror of j is j' and the mirror of j' is j



When we move c from left to right, we try to "expand" the palindrome at each 'i'. When I say the root expand, it means that I'll check whether there exists a palindrome centered at i and if there exists one, I'll store the "expansion length" to the left or to the right in a new array called P[] array or some proper LPS[].

If the palindrome at i expands beyond the current right boundary r then c is updated to i and new l, r are found and updated.

Lets take previously discussed problem palindrome "abacaba" which is centered at i=3.



Expansion length = 3

Hence P[i] = 3

Therefore the P[] array shows the expansion length of the palindrome centered at each index. But we want need to manually go to each index and expand

Important recap: c indicates the center of the current largest odd length palindrome. And l, r denote the palindromes

Let's see the P[] array for the string "abacaba".

	l		i'		i		r				
	a	b	a	c	a	b	a	c	a	b	b
i	0	1	2	3	4	5	6	7	8	9	10
P[]	0	1	0	3	?						

When i = 4 the index is inside the current largest palindrome ie i < r.

As long as the palindrome at i' does not expand beyond the left boundary l of the current largest palindrome, minimum certainty possible expansion length at i is P[i']

	l		i'		i		r		
	c	a	c	a	c	a	c	b	c
	0	1	2	3	4	5	6	7	8
P	0	1	2	3	3	1	0		

If palindrome at i' exceeds l the minimum possible expansion at i is $\underline{r - i}$

So P[5] = 6 − 5 = 1

Why can't the palindrome centered at i expand after r in this case. If it did, then it would have already if the palindrome at i were to expand beyond r, then the character at index 7 should have been a but that would cause the current palindrome

Manacher's Algorithm: O(N) where N is the length of the input string

Let us assume we are given a string S with a length N. The objective of this algorithm is to a table P[] such that knowing the value of P[i] allows us to find the length of the longest

palindromic substring with center at position i for all $1 \le i \le N$

Suppose S = "abba" and consider 1-based indexing of the string. Then the palindromic substring "bb" may have an ambiguity in terms of its center as index 2 and index 3 are both its centers. To avoid this ambiguity, we insert an arbitrary character (that doesn't appear in the string), say '#', between every two characters of S and in the beginning and end of S.

So the string now transforms to T: "a#b#b#a". Notice that every palindromic substring in the original string, regardless of length, corresponds to a palindromic substring in the transformed string of odd length, so we can now define $P[i]$ to mark the radius of the largest odd-length palindromic substring centered at index i.

Suppose R is the rightmost boundary of the longest palindrome centered at i. Then $P[i] = R - i$ $(1 \le i \le 2N-1)$

Thus then, the length of the longest palindromic substring for every index in the original string can be easily recovered.

Let us see how the P values will be filled for the given string S.

i :	1	2	3	4	5	6	7	8	9
T :	#	a	#	b	#	b	#	a	#
P =	0	1	0	1	4	1	0	1	0

From the $P[]$ table we can see that the length of the longest palindrome substring in S is 4.

Now the hardest and the most essential part of the algorithm is the calculation of $P[]$ table.

Let T be the string formed after inserting characters "#" between the characters of string S.

Let C be the center of the palindrome currently known to include the boundary closest to the right end of our string T.

Let R be the rightmost boundary of the palindrome centered at C.

Let i be the position of an element in T whose palindromic span is being determined with i always to the right of C.

i' be the mirrored position of i with respect to c. It can be seen that $i' = c - (i - c) = 2c - i$.

We will iterate i from 1 to length of T and determine $P[i]$ assuming that $P[i']$ for all $i' < i$ has already been calculated.

Consider we are given a string S: "babcbabcbcecba"

Thus we have T: #b#a#b#c#b#a#b#c#b#a#c#c#b#c#a"

after inserting "#" between every two characters.

Consider the following three cases:

Case 1: The length of the longest palindrome centered at i' is such that the left boundary of the palindrome does not extend beyond or until the boundary of the longest palindrome centered at c

Let the left boundary of palindrome at c be L. We know that $T[c+k] = T[c-k]$ for all $k < R - c$. Since the substring of T centered at c and radius $R-c$ is palindrome.

Notice that $P[x]$ gives the number of odd length palindromes centered at x and hence $2P[x]-1$ is the length of the longest palindrome centered at x.

So consider $T[i'-k]$ and $T[i+k]$ for all $k \le P[i']$

$k \le P[i']$ implies $k < i' - L$ as the subpalindrome centered at i'

i' $\quad$ i

As i' is a mirror of i about c

5 4 3 2 1 0 1 2 3 4 5

3

2

$k < R - i$

& $P[i'] = 3$

$k = 2$

$P[i']$ $\quad$ $L = 5$

4 $\quad$ $P[i']$

R A R A R A M A R A R A R

$P[i']$ $\quad$ $P[i]$

$= 2$

$c - i' = i - c$

$-i' = i - c$

$i' = 2c - i$

k $\quad$ i' $\quad$ i

M A R A M A R A M

1 2 3 4 5 6 7 8 9

c

As i' is a mirror of i about c

$k < R - i$

$i' = 2c - i$

$T[i'-k] = T[2c - i - k] = T[c - (i+k-c)]$

$i + k - c < i + R - i - c$ $\quad T[i+k] = T[c + (i+k-c)]$

$i + k - c < R - c$

Hence $T[i'-k] = T[c-k']$

$T[i+k] = T[c+k']$

where $k' < R - c$

Hence $T[i+k] = T[i'-k]$

A R A M A R A

k c k

$T[c+k] = T[c-k]$

R A R A M A R A R

L C R

P[] 0 1 1 0 4 0 1 1 0

$T[C-k] = T[C+k]$ for all $k < R-C$ since the substring of T centered at C and radius $R-C$ is a palindrome.

R A R A M A R A R

P[] 0 1 1 0 4 0 1 1 0

C

k

so consider $T[i'-k]$ and $T[i+k]$ for all $k \le P[i']$

$k \le P[i']$. $k \le P[i]$ implies $k < i' - L$

R A M A R A M A R A M A R

P[] 0 0 0 0 4 0 0 0 4 0 0 0

C i k R

L k i'

$k \le P[i']$ implies $k < i' - C$ as the subpalindrome centered at i' has a left boundary to the right of the left boundary of the palindrome centered at C

As i' is a mirror of i about C

$$k < R - i$$

6 5 4 3 2 1 0 1 2 3 4 5 6

i' i R

$P[i'] = 2$

$k < R - i$

When $k < 4$ the $k \le P[i]$ 4

5

P[]	0	0	2	0	4	0	6	0	4	0	2	0	0	$k < R - C$
k	1	2	3	4	5	6	7	8	9	10	11	12	13	
	R	A	M	A	R	A	M	A	R	A	M	A	R	

i' C i

L k R

$i' = 5 \Rightarrow C - (i - C) = 2C - i$

$\Rightarrow 7 - (9 - 7)$ | $2 \times 7 - 9$

$7 - (2) = 5$ | $14 - 9$

5

We will iterate i from 1 to length of T and determine $P[i]$ assuming that $P[i']$ for all $i' < i$ has already been calculated.

Case 1: The length of the longest Palindrome centered at i' such that the left boundary of the palindrome does not extend beyond or until the left boundary of the longest palindrome centered at C.

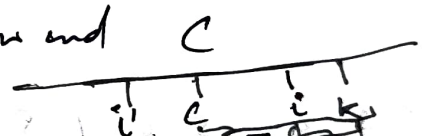
Let the left boundary of palindrome at C be L. We know that $T[C+k] = T[C-k]$ for all $k < R-C$.

Notice that $P[x]$ gives the number of odd length palindromes centered at X and hence $2P[x]-1$ is the length of the largest palindrome centered at X.

So consider $T[i'-k]$ and $T[i+k]$ for all $k < P[i]$

$k \leq P[i]$ implies $k < i' - L$ as the subpalindrome centered at i' such the L is to the ~~left~~ right of the boundary of the bigger palindrome centered at C.

As i' is a mirror of i around C



$k < R - i$

~~$i' = 2$~~ $C - i' = C + i$

$C - i' = i - C$

$-i' = i - 2C$

$i' = 2C - i$

$T[i'-k] = T[2C - i - k]$

$= T[C + C - i - k] = T[C + (C - i - k)]$

$\Rightarrow T[C - (i + k - C)]$

$T[i+k] = T[C + \underbrace{(i+k-C)}_{k'}]$

$T[i+k] = T[C+k'] = T[C-k']$

$T[i'-k] = T[C-k']$

$T[i+k] = T[i'-k]$

Where $k' < R - C$

for all $k < i' - L$

Hence $P[i] = P[i']$

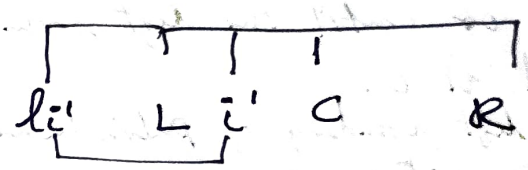
Also as the right boundary centered at i does not exceed R, we should not update R and C

Case 2:

Consider the palindrome centered at i' extends beyond the left boundary of the palindrome centered at C.

Consider $T[L-1]$

mirror $(L-1)$ wrt i'

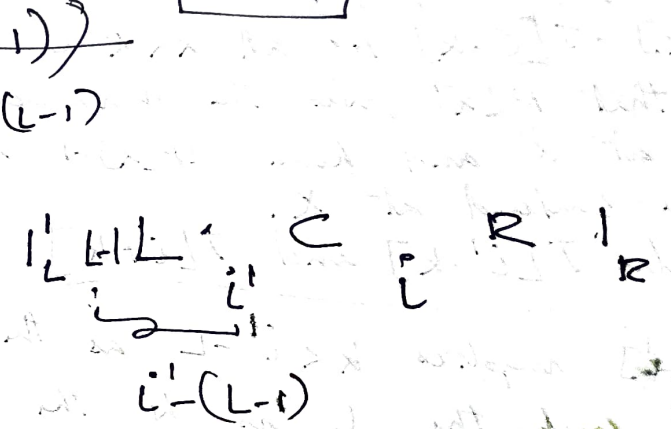


$\cancel{= M_1 = 2(i' - (L-1))}$

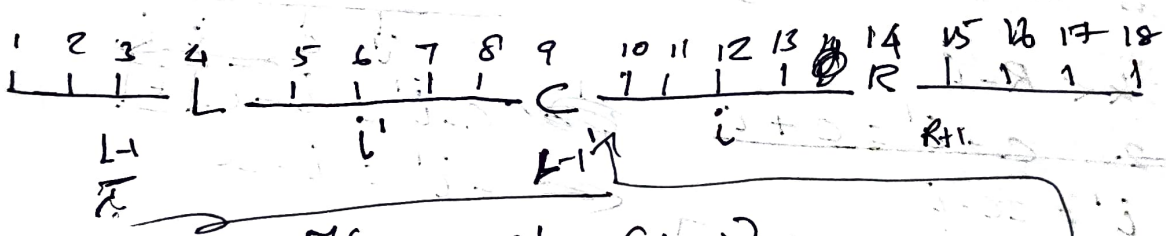
$M_1 = 2i' - (L-1)$

$\cancel{L-1 = 2i}$

$T(L-1) = T(M1)$



Mirroring makes it twice the value $2 \times i' - (L-1)$



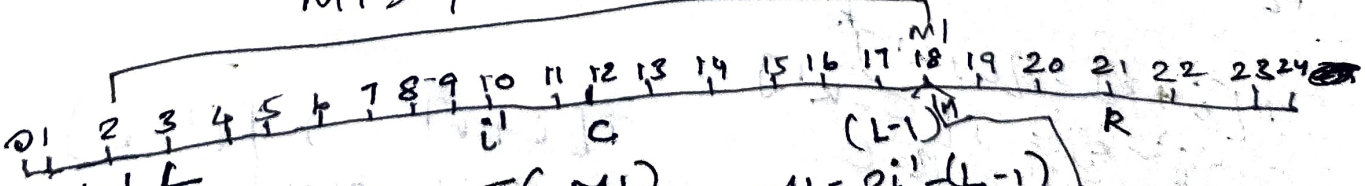
$M1 = 2i' - (L-1)$

$\cancel{2 \times 6 - 3 = 12 - 3 = 9}$

$\cancel{2 \times 6 - (3-1) = 12 - 2 = 10}$

$2 \times 6 - 3 = 9$

$M1 = 9$



$T[L-1] = T(M1)$

$M1 = 2i' - (L-1)$

$= (2 \times 10) - 2 = 18$

Mirror M1 with respect to C

$M2 = 2C - M1$

$= 2\times12 - 18$

$= 24 - 18 = 6$

$2C - i = i'$

why $2i' - (L-1) = L-1$

why $i' = 2C - i \Rightarrow T[i'] = T[2C-i]$

$M1 = 2i' - (L-1)$ $\quad C - i' = i - C$

$T[M1] = T(L-1)$ $\quad -i' = i - C - C$

$-i' = i - 2C$

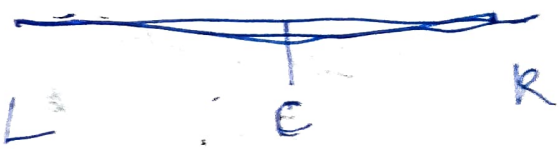
Mirror M1 wrt C $\quad i' = 2C - i$

therefore $\quad M1 = 2i' - (L-1)$

M2 should be equal to

$M2 = 2C - M1$

$T[M1] = T[M2]$



mirror M2 wrt i

$R = 2C - L$

$M3 = 2i - M2$

$= 2i - (2C - M1)$

$= 2i - 2C + M1$

$= 2i - 2C + 2i' - L + 1$

$= 2(i - C + i') - L + 1$

$= 2(2C - i - C + i') - L + 1$

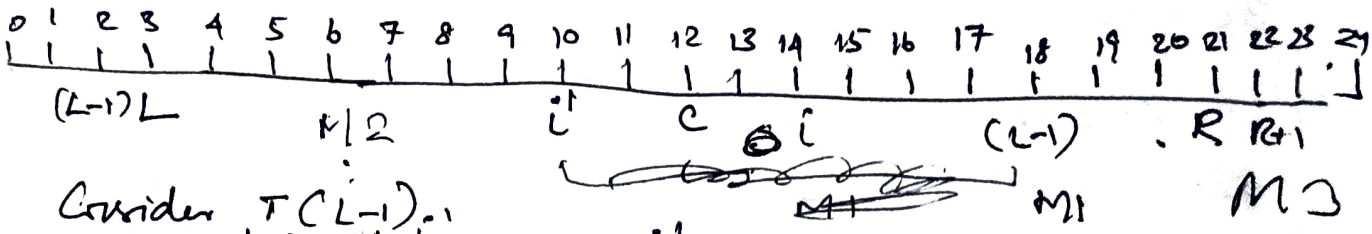
$(2C - L + 1 = R + 1$

Now suppose $T(M3) = T(M2)$ Hence

$T(M3) = T(M1) = T(L-1)$

$T(M3) = T[R+1]$

Case 2 :- Consider that the palindrome centered at 'i' extends beyond the left boundary of the palindrome centered at C.



Consider $T(L-1)$:-
mirror L-1 wrt i
$M1 = 2i - (L-1)$

$\Rightarrow 20 - 2 = 18$

$i' = 2C - i$
$= (2 \times 12) - 14 = 10$

$i = 2C - i'$
$= 2 \times 12 - 10 = 14$

mirror M1 wrt. C

$M2 = 2C - M1$
$= 24 - 18 = 6$

$T[M1] = T[M2]$

Mirror M2 wrt i

$M3 = 2i - M2 \Rightarrow$
$28 - 6 = 22$

$M3 = i$

$2i - 2C + M1$
$2i - 2C + 2i - L + 1$
$2(i - C + i) - L + 1$
$2(2C - i - C + 1) - L + 1$
$(2C - L) + 1 = R + 1$

$T[M3]$ is $T[M2]$ centered around i
$T[M2]$ is $T[M1]$ centered around C
$T[M1]$ is $T[L-1]$ cent. on i'.

Eg $T[M3] = T[M2]$

$T[M2] = T[M1]$
$\Rightarrow T[M3] = T[M1]$
$\Rightarrow T[M3] = T[L-1]$
which is true

Then $T[R+1] = T[L-1]$ which is not possible since it would mean that it is possible to extend the palindrome centered at C

Therefore $T[M3] \neq T[M2]$

So $T[M3] \neq T[M2] \Rightarrow T[R+1] \neq T[M2]$

which means it is not possible to extend the palindrome centered at i

Case 2:

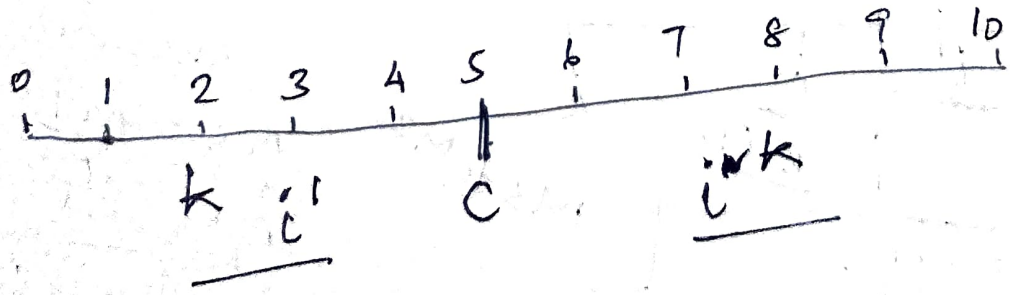
The length of the longest palindrome centered at i' is such that the left boundary of the palindrome does not extend beyond or until the ~~longest boundary~~ left boundary of the longest palindrome centered at C.

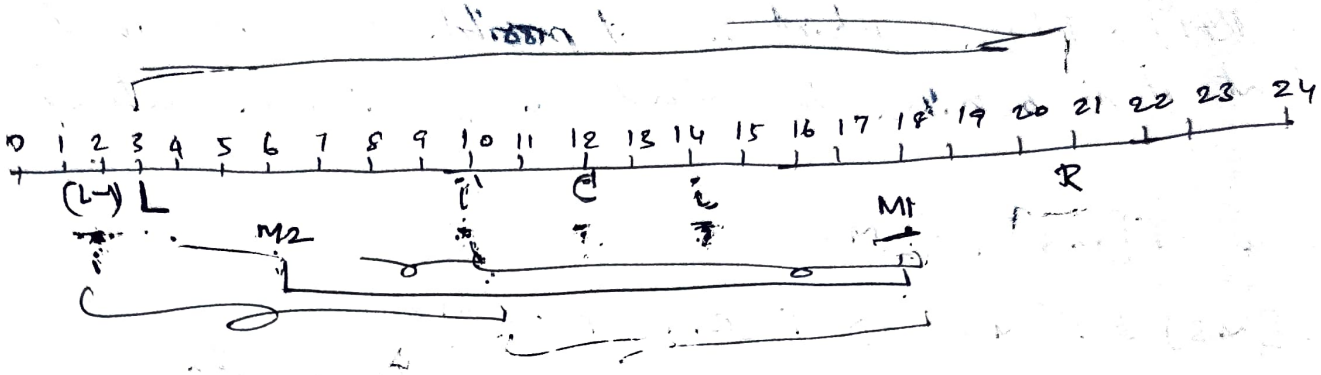
Let the left boundary of palindr centered at C be L.

We know that $T[C+k] = T[C-k]$ for all $k < R-C$ since the substring of T centered at C and radius $R-C$ is a palindrome.

Notice that $P[x]$ gives the number of odd length palindrome centered at x and hence $2P[x]-1$ is the length of the longest palindrome centered at x.

So consider $T[i'-k]$ and $T[i+k]$ for all $k \leq P[i']$





$R = 2C - L$

$= 24 - 3 = 21$

$i' = 2C - i$

~~$i' = 2C - i = 24 - 14 = 10$~~

OR

R A M A R

0 i

1) Mirror i' $\quad i = 2C - i' = 24 - 10 = 14$

2) Mirror $L-1$ wrt i'

$M1 = 2i' - (L-1) = 20 - 2 = 18$

$T[M1] = T[L-1]$

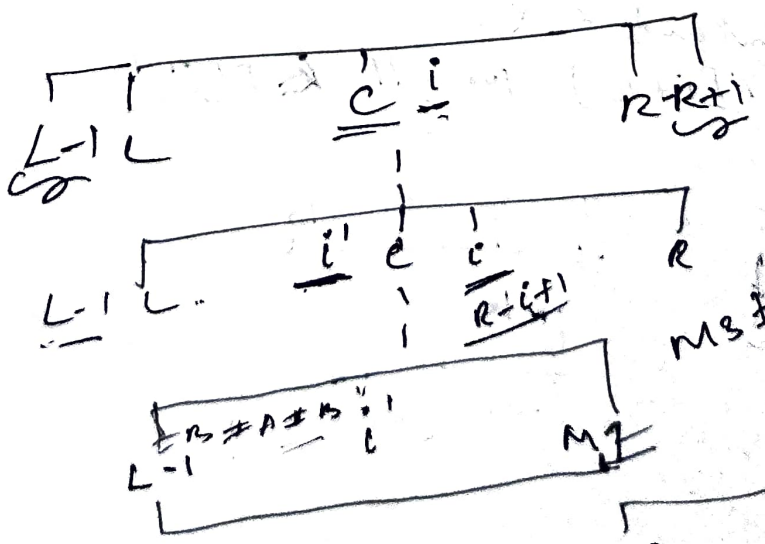
3) Mirror M_1 wrt C ✓

$M2 = 2C - M1 = 24 - 18 = 6$

$T[M2] = T[M1]$

4) $M3 = 2i - M2 = 28$

C



$M3 \neq M2$

the value

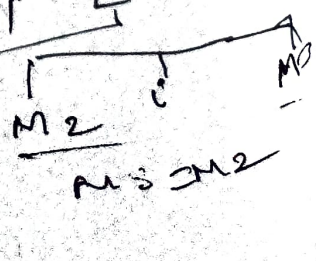
$R - i + 1$

$M3 = M2$

$M2 = M1$

$M3 = M1$

$R+1 \quad L-1$



$M3 = M2$

0 1 2 3
c b b d

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

7 - 4 = 0
5 - 3 = 2
3 - 2 = 1
1 - 1 = 0

ZigZag Conversion:

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this

3 Rows

PAYPALISHIRING

0 P0		4 A1	8 H2	12 N3
1 A4	3 P5	5 L6	7 S7	9 I8	11 I9	13 G10
2 Y11		16 I12		10 R13	

PAHNAPLSIIGYIR

Interval = 2n - 2

Step = Interval - 2i

PAYPALISHIRING

0 P0		6 I1		12 N2
1 A3	5 L4	7 S5	11 T6	13 G7
2 Y8	4 A9	8 H10	10 R11	
3 P12		9 I13		

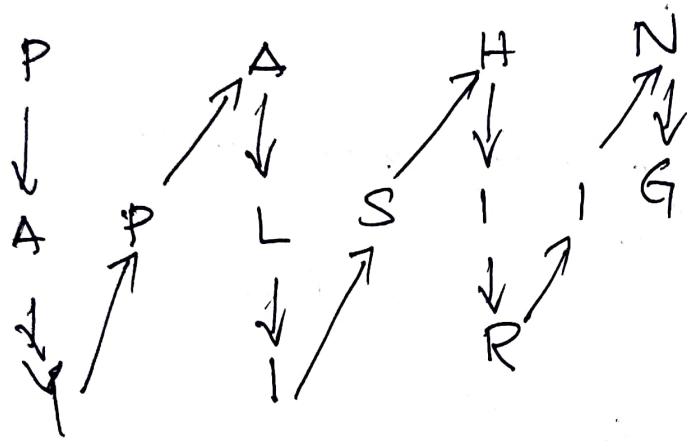
PINALSIGYAHRPI

Interval = 2×4 - 2 = 6
Step = 6 - 2i
6 - 8 = -2

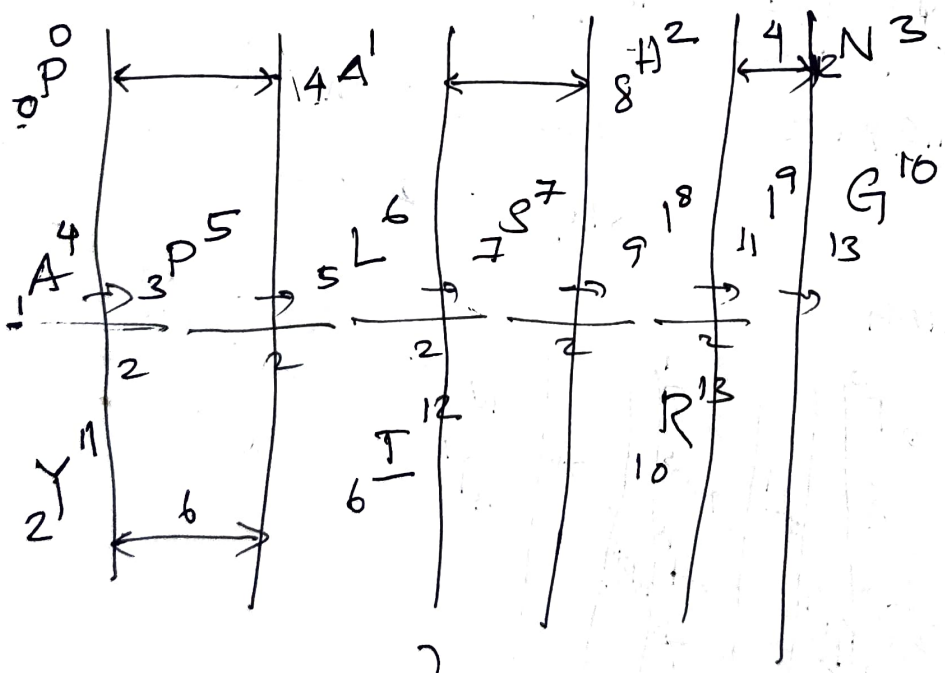
for(each row){
for(each col){
if(mid)
display mid.

PAYPALISHIRING

0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	A	Y	P	A	L	I	S	H	I	R	I	N	G



0	1	2	3	4	5	6	7	8	9	10	11	12
P	A	H	N	A	P	L	S	I	I	G	Y	I



for (number of rows)
for (each row)

Interval = 2 × numberofrow − 2
6 − 2 = 4

step = Interval − 2 × i
4 − 2 = 2

Regular Expression matching

Dynamic Programming - Bottom Up Approach

Conditions

if s.charAt(idx) == p.charAt(idx)
return s.charAt(idx-1) == p.charAt(idx-1)

if ~~s.charAt(idx)~~ p.charAt(idx) : '.'
return s.charAt(idx-1) == p.charAt(idx-1)

if p.charAt(idx) == '*'
when s.charAt(idx) ≠ p.charAt(idx-1)
Could be 0 occurence of the
char preceding *
s.charAt(idx) == p.charAt(idx-2)
when s.charAt(idx) == p.charAt(idx-1)
(or) p.charAt(idx-1) = '.'

Length of the two dimensional boolean array
(T[i][j]) will be ~~equal to~~ one more than
length of the string/text and pattern

Empty Text

Empty pattern

		0	1	2	3	4	5	6
		∅	x	a	*	b	.	c
0		T(0,0) T	F	F	F	F	F	F
1	x	F	T(1,1) T	F	T	F	F	F
2	a	F	F	T	T	F	F	F
3	a	F	F	F	T	F	F	F
4	b	F	F	F	F	T	F	F
5	y	F	F	F	F	F	T	F
6	c	F	F	F	F	F	F	T

(i-1)

∅ a*

x a *

x

x a * b . c

x
a
b
y
c

Steps :

~~for(int~~

boolean T[][] = new T[S.length+1][P.length+1]

for (int i=0; i<=S.length+1; i++) {

~~T[0][i] =~~

if(P[i-1] == '*') {

T[0][i] = T[0]

x (a*)

x

		0	1	2	3	4	5	6
			x	a	*	b	.	c
0		T	F	F	F	F	F	F
1	x	F	T	F	T	F	F	F
2	a	F	F	T	T	F	F	F
3	a	F	F	T	T	F	F	F
4	a	F	F	T	T	F	F	F
5	a	F	F	T	T	F	F	F
6	b	F	F	F	F	T	F	F
7	y	F	F	F	F	F	T	F
8	c	F	F	F	F	F	F	T

3. If no diagonal true just increment row+1, when the pattern char is a and the previous text char and curr text char matches

1) There has to be atleast one match in a row

2) Find if there's a diagonal mat i.e. in upper row+1, i+1

Summary RegEx Matching Dynamic Programming

x a * b . c
x a b y c
x a a b y c

1) Start matching from the end

2) If the charAt(Pattern) == charAt(Text)
match (Pattern-1, Text-1)

3) If the charAt(pattern) == '.' then
match (Pattern-1, Text-1) because
↳ this does not match no point
then they are in proceeding

4) If the charAt (Pattern) == '*' then
the following conditions (* means 0 or max occurences)
match 0 occurences

x a * b c

i) x c b c (✗) } 0 (zero) occurence of a
ii) x b c (✓)

If pattern is at * and assume b + b has gotten matched

Condition (i) leaves us at c
pattern -1 does not match
pattern -2 does not match
c ≠ x

Condition (ii) leaves us at x
pattern -1 does not match
pattern -2 matches 0 occurence

[or max occurence x a * b c
x a a c b c

When pattern is at *
match pattern-1 + text
which is a match

Have a 2-D boolean array

First dimension is rows the represent string text

Second dim is a/m that represent pattern

0	1	2	3	4	5
x	a	*	b	.	c

0	1	2	3	4
x	a	b	y	c

~~T[c][c]~~ T[5][4] = true

~~T[5][~~

T[4][3]

0 = x

T[0][0] = ~~x~~t T[0][1] = F T[0][2] = T T[0][3] = F

T[0][2-2]

T[0][4] = F

2-D Array

boolean T[][] = new T[s.length+1][p.length+1]

This is to account for s or p to be empty

Now you've built the 2D array:

Iterate over row ~~by row~~

Iterate over col

		0	1	2	3	4	5	6
			x	a	*	b	.	c
0		T	F	F	F	F	F	F
1	x	F	T	F	T	F	F	F
2	a	F	F	T	T	F	F	F
3	a	F	F	T	T	F	F	F
4	a	F	F	T	T	F	F	F
5	a	F	F	T	T	F	F	F
6	b	F	F	F	F	T	F	F
7	y	F	F	F	F	F	T	F
8	c	F	F	F	F	F	F	T

Wild card matching - Dynamic Programming

Given an input string (S) and a pattern (P). implement wild card pattern matching with support for '?' and '*' where

1) '?' Matches any single character
2) '*' Matches any sequence of characters including the empty sequence

The matching should cover the entire input string (not partial)

S="aa" P="a" false
S="aa" P="*" true
S="cb" P="?a" false
S="acdcb" P="a*c?b" false

		empty pattern					
		0	1	2	3	4	5
		""	a	*	c	?	b
empty string 0	""	T	F	F	F	F	F
1	a	F	T	T	F	F	F
2	c	F	F	T	T	F	F
3	d	F	F	T	F	T	F
4	c	F	F	T	T	F	F
5	b	F	F	T	F	T	T

	0	1
		*
0		T
1	a	T
2	b	T
3	c	T

Suppose

pattern = a***b***c

We need to condense the * ⇒ a*b*c

```
boolean encounteredAsterisk = true

for (int i=0; i<p.length; i++) {
    if (p[i] == '*') {
        encounteredAsterisk =
    if (encounteredAsterisk) {
        if (p[i] == '*') {
            encounteredAsterisk = true;
```

a***b***c ⇒ a*b*c

```
int writeIndex = 0;
boolean isFirst = true;
for (int i=0; i< pattern.length; i++) {
    if (pattern[i] == '*') {
        if (isFirst) {
            pattern[writeIndex++] = pattern[i];
            isFirst = false;
        } else {
            pattern[writeIndex++] = pattern[i];
            isFirst = true;
        }
    }
```

patte

P[]:-

a	*	*	*	b	*	*	c

```
boolean encounteredAsterisk = true
int asteriskAtIdx
for (int i=0; i<p.length; i++) {
    if (P[i] == '*') {
        encounteredAsterisk = true;
```

Wild Card matching - Notes:-

Case 1 :- * A pattern with one asterisk should match 0 or more sequences. The string can be either null or anything.

Case 2 :- Pattern cannot be null. only null string matches.

Case 3 :- a*****b***c?e
⇒ is equivalent to a*b*c?e

```
boolean isFirst = true
int writeIndex = 0
for (int i=0; i<p.length; i++) {
    if (P[i] == '*') {
        if (isFirst) {
            P[writeIndex++] = P[i]
            isFirst = false
        }
        else {
            P[writeIndex++] = P[i];
            isFirst = true;
        }
    }
```

Case 4:- What if the first pattern char is '*'

T[0][1] = true

when P[i] == * and writeIndex == 0

T[0][1] = true

when string is null and char at first position is *

Remember 1 is the original 0th position of pattern and text respectively. We added 0th position to account for null pattern & text respectively

Case 5:- T[0][0] = true

```
for (int k=1; k=T.length; k++) {
    T[k][0] = false
for (int j=1; j<T.length; j++) {
    for (int l=1; l<T[j].length; l++) {
        if (P[l-1] == '*') {
            T[j][l] = T[j-1][l] || T[j][l-1]
        } else if (P[l-1] == '?' || P[l-1] == S[j-1])
            T[j][l] = T[j-1][l-1]
```