ITDO 5014 ADSA
Module 5

String Matching

ITDO 5014 ADSA

Module 5 : String Matching (7 Hrs)

1. Introduction
2. The naïve string matching algorithm
3. Rabin Karp algorithm
4. Boyer Moore algorithm
5. Knuth-Morris-Pratt algorithm (KMP)
6. Longest common subsequence(LCS)

   Analysis of All algorithms and

   problem solving.

Self-learning Topics: Implementation of Robin Karp algorithm, KMP algorithm and LCS.

Q1. Explain diffetrent string matching algorithms. (10M) (2*)

Q2 Define Knuth Morris Pratt algo for string matching. Write a function to implement the algorithm. (10M)

Q3 Write a note on Knuth Morris Pratt pattern matching. (3*)

Q4. Explain longest Common Subsequence (LCS) with suitable example. (3*)

Q5. What is longest Common Subsequence (LCS) problem ? Find LCS for the following string.

X = ACB AED Y = ABC ABE

Q6. Find LCS for the following string.

P = (100101101101) Q = (0110)

1.Naive algorithm for Pattern Searching :

Given a text : txt[0..n-1] and a pattern : pat[0..m-1],

write a function search (char pat[], char txt[])
that prints all occurrences of pat[] in txt[] where n > m.

Examples:

Input: txt[] = "THIS IS A TEA POT"
pat[] = "TEA"
Output: Pattern found at index 10

Input: txt[] = "AABAACAADAABAABA"
pat[] = "AABA"
Output: Pattern found at index 0
Pattern found at index 9
Pattern found at index 12

Text : A A B A A C A A D A A B A A B A

Pattern : A A B A

A A B A   A A B A

A A B A A C A A D A A B A A B A

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

A A B A

Pattern Found at 0, 9 and 12

Pattern searching is an important problem in computer science.

When we do search for a string in notepad/word file or browser or database,

pattern searching algorithms are used to show the search results.

Naive Pattern Searching:

Slide the pattern over text one by one and check for a match.

If a match is found, then slides by 1 again to check for subsequent matches.

```
// C program for Naive Pattern Searching
algorithm
 void search(char* pat, char* txt)
{            int M = strlen(pat);
             int N = strlen(txt);

   /* A loop to slide pat[] one by one */
   for (int i = 0; i <= N - M; i++)          O(n–m +1)
   {          int j;
 // For current index i, check for pattern match
        for (j = 0; j < M; j++)                    O (m)
            if (txt[i + j] != pat[j])
                break;
        if (j == M)       // if pat[0...M-1] = txt[i, i+1,
                          //  ...i+M-1]
          printf("Pattern found at index %d \n", i);
   }
}
```

```
/* Driver program to test above function */

int main()
{                    012345678910
   char txt[] = "AABAACAADAABAAABAA";
   char pat[]= "AABA";

   search(pat, txt);

   return 0;
}
```

Output:
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13

Best case :

The best case occurs when the first character of the pattern is not present in text at all.

txt[] = "AABCCAADDEE";
pat[] = "FAA";
The number of comparisons in best case is O(n).

Worst case :

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

1) When all characters of the text and pattern are same.

txt[] = "AAAAAAAAAAAAAAAAAA";
pat[] = "AAAAA";

2) Worst case also occurs when only the last character is different.

txt[] = "AAAAAAAAAAAAAAAAAB";
pat[] = "AAAAB";
The number of comparisons in the worst case is O(m*(n-m+1)).

Although strings which have repeated characters are not likely to appear in English text, they may well occur in other applications (for example, in binary texts).

The KMP matching algorithm improves the worst case to O(n).

2. Rabin-Karp Algorithm

String Matching

Introduction

- **Rabin-Karp algorithm** is a **string-matching algorithm** created by Richard M. Karp and Michael O. Rabin that uses **hashing** to find any one of a **set of pattern strings** in a text.
- It moves from one ‘window’ to another to check hash value without checking all characters of all cases. When the **hash value** of the **window** is **matched**, then only it **checks each character**.
- This makes algorithm more efficient.

- NAÏVE ALGO : ➔
    - **txt[] = "AAAAAAAAAAAAAAAAAA";**
    - **pat[] = "AAAAA";**

- **2) Worst case also occurs when only the last character is different.**
    - **txt[] = "AAAAAAAAAAAAAAAAAB";**
    - **pat[] = "AAAAB";**
-

Pseudo Code

Step 1: Take input text ‘T’.
Let ‘n’ be the length of T.

txt[] = "AAAAAAAAAAAAAAAAAB";
pat[] = "AAAAB";

Step 2: Let ‘P’ be a pattern (substring) of T.
Let ‘m’ be the length of P.

Step 3: Compute hashing using a suitable hash function **(mod hash function preferred)**
Choose a **random prime number** ‘q’ and find **P mod q**

Step 4: for i=0 to n-m

if hash value matches do:

if all characters of window **match**,
then **valid hit**

if all characters of window **don’t match**,
then **spurious hit**

Example

Q. Apply Rabin-Karp algorithm to Find whether the input text contains the given pattern (234) or not.

Step 1: T =

3	**1**	**2**	**3**	**4**	**8**	**6**	**2**

Length of input text (n) = 8

for i=0 to n-m
if hash value matches do:
if all characters of window **match**,
then **valid hit**
if all characters of window **don't match**,
then **spurious hit**

Step 2: Suppose we want to match a pattern **P=234**
Length of pattern (m) = 3
Hence, **each window is of size = 3**

Step 3: Compute hashing. Choose any random prime number q.
Let q=13

$\therefore$ P mod q = 234 mod 13 = 0
$\therefore$ H = 0

Example

Window 1:

3	**1**	**2**	**3**	**4**	**8**	**6**	**2**

312 mod 13 = 0   Hash value matches.

Now check for each character of the window.

312 ≠ 234. **Spurious Hit.**

Window 2:

3	**1**	**2**	**3**	**4**	**8**	**6**	**2**

123 mod 13 = 6

Hash value doesn't matches.

for i=0 to n-m
    if hash value matches do:
        if all characters of window **match**,
            then **valid hit**

        if all characters of window **don't match**,
            then **spurious hit**

Example

Window 3:

3	**1**	**2**	**3**	**4**	**8**	**6**	**2**

234 mod 13 = 0
Hash value matches. Now check for each character of the window.
234 = 234. **Valid Hit**. Pattern found at position 3

Window 4:

3	**1**	**2**	**3**	**4**	**8**	**6**	**2**

348 mod 13 = 10
Hash value doesn’t matches.

Example

Window 5:

3	**1**	**2**	**3**	**4**	**8**	**6**	**2**

486 mod 13 = 5
Hash value doesn’t matches.

Window 6:

3	**1**	**2**	**3**	**4**	**8**	**6**	**2**

862 mod 13 = 4
Hash value doesn’t matches.

Q. Apply Rabin-Karp algorithm to Find whether the input text contains the given pattern (NST) or not.

A	**N**	**S**	**T**	**Z**	**X**	**N**	**S**	**T**	**U**

Step 1: T =

Length of input text (n) = 10

Step 2: Suppose we want to match a pattern **P=NST**
Length of pattern (m) = 3
Hence, **each window is of size = 3**

Step 3: Now since the input text is an alphabetical string in this case,
we can use either **ASCII codes** or **(A:1 – Z:26)** system.
Compute hashing using a suitable hash function.
N(14) + S(19) + T(20) = 53
Choose any random prime number ‘q’.
Let q=17

$\therefore$ 53 mod 17 = 2

Example

Window 1:

A	**N**	**S**	**T**	**Z**	**X**	**N**	**S**	**T**	**U**

A(1) + N(14) + S(19) mod 17 = 34 mod 17 = 0
Hash value doesn’t matches.

Window 2:

A	**N**	**S**	**T**	**Z**	**X**	**N**	**S**	**T**	**U**

N(14) + S(19) + T(20) = 53 mod 17 = 2
Hash value matches. Now check for each character of the window.
53 = 53. **Valid Hit**. Pattern found at position 2

Example

Window 3:

A	N	S	T	Z	X	N	S	T	U

S(19) + T(20) + Z(26) mod 17 = 65 mod 17 = 14
Hash value doesn’t matches.

Window 4:

A	N	S	T	Z	X	N	S	T	U

T(20) + Z(26) + X(24) = 70 mod 17 = 2
Hash value matches. Now check for each character of the window.
70 ≠ 53. **Spurious Hit**.

Example

Window 5:

A	N	S	T	Z	X	N	S	T	U

Z(26) + X(24) + N(14) mod 17 = 64 mod 17 = 13
Hash value doesn’t matches.

Window 6:

A	N	S	T	Z	X	N	S	T	U

X(24) + N(14) + S(19) = 57 mod 17 = 6
Hash value doesn’t matches.

Example

Window 7:

A	N	S	T	Z	X	N	S	T	U

N(14) + S(19) + T(20) mod 17 = 53 mod 17 = 2
Hash value matches. Now check for each character of the window.
53 = 53. **Valid Hit**. Pattern found at position 7

Window 8:

A	N	S	T	Z	X	N	S	T	U

S(19) + T(20) + U(21) = 60 mod 17 = 9
Hash value doesn’t matches.

UT2

Rabin Karp Algo

Pattern = 624

Let q = 13   p mod q = 624 mod 13 = 0
⇒ h = 0

2	8	6	2	4	7	0

① 286 mod 13 = 0

286 ≠ 624 ⇒ Spurious Hit

② 862 mod 13 = 4

⇒ Hash values do not match.

③ $\boxed{624 \bmod 13 = 0}$

624 (pattern) = 624 ⇒ Valid Hit ⇒ Pattern found at index 3

④ 247 mod 13 = 0

624 ≠ 247 ⇒ Spurious Hit

⑤ 470 mod 13 = 2

⇒ Hash values do not match

———o———

Algorithm

```
Rabin_Karp (T, P)
{
n = T.length                          // length of input text
m = P.length                          // length of pattern
h^p = Hash ( P[…] )                   // P mod q
h^T = Hash ( T[…] )                   // T mod q

for i=0 to n-m          O(n+m)
{
    if (h^p == h^T)                   // if hash values are equal
    {
        if (P [0 … m-1] == T [i+0… i+m-1])   // if pattern and window characters are equal
            {
                print “Pattern found at position i+1”
            }
    }
}
}
```

Time Complexity & Applications

Time Complexity:

The time complexity of Rabin-Karp algorithm is **O(n+m)** where,
n = length of input text
m = length of pattern

Applications:

(i) Plagiarism detection
(ii) Text processing
(iii) Bioinformatics
(iv) Compression

4.Longest Common Subsequence (LCS)

- **LCS: The longest common subsequence (LCS) problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences).**
- **Subsequences are not required to occupy consecutive positions within the original sequences.**

Given two sequences X and Y, a sequence Z is said to be a common subsequence of X and Y, if Z is a subsequence of both X and Y. For example, if

X={ A,C,B,D,E,G,C,E,D,B,G} and

Y={B,E,G,C,F,E,U,B,K }

then a common subsequences of X and Y could be

Z1={ B,E,E,B}

Z2 = { B,E,G,C,E,B }

The longest common subsequence of X and Y is Z2 = {B,E,G,C,E,B } .

- **Consider two strings:**

str1 = “abcdaf”

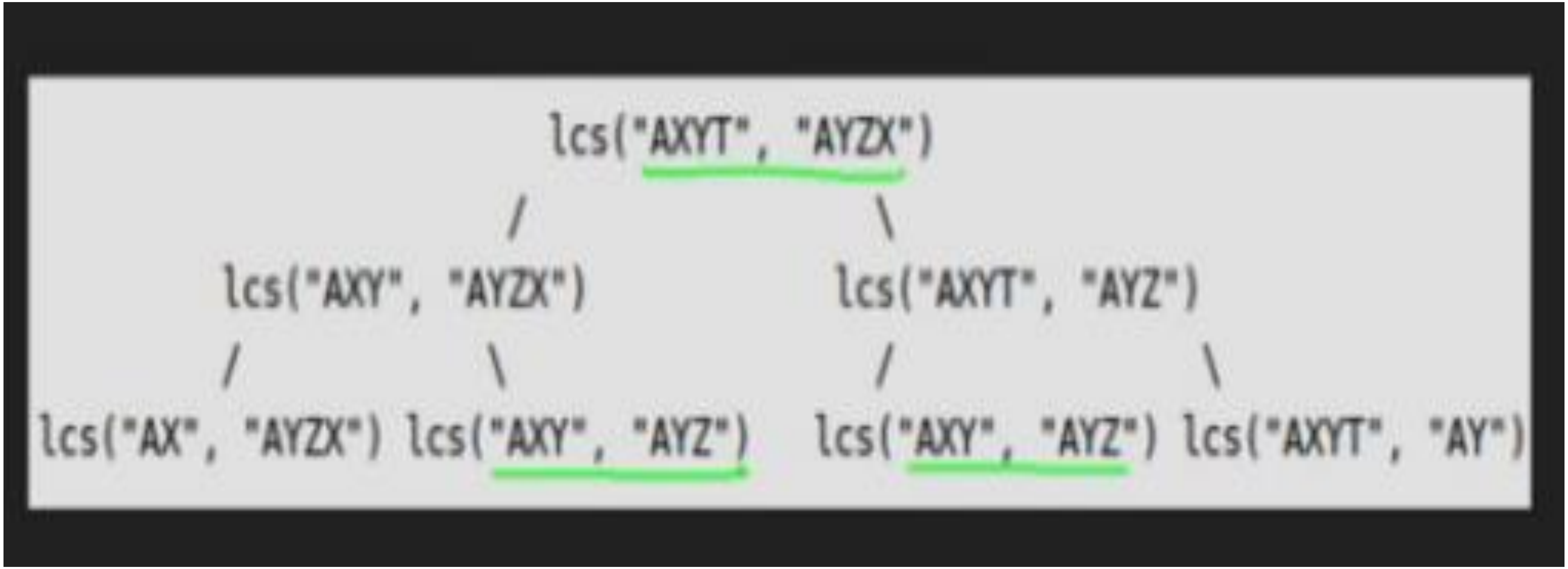
str2 =“acbcf”

Now here we can say that :

longest common subsequence will be : abcf

and its length = 4

- **If we use divide and conquer approach to solve LCS problem then:**



- **So to solve this problem programmatically we use dynamic programming technique.**

- By using dynamic programming technique for LCS, arrange both the strings in following manner:

str1 →		A	B	C	D	A	F
str2 ↓	0	0	0	0	0	0	0
a	0						
c	0						
b	0						
c	0						
f	0						

Dynamic Programming

Approach:

- If the last characters match:
  - LCS[i][j] = LCS[i-1][j-1] + 1
- If the last characters don't match:
  - LCS[i][j] = max(LCS[i-1][j] , LCS[i][j-1])

- By using dynamic programming technique for LCS, arrange both the strings in following manner:

str1 →		A	B	C	D	A	F
str2 ↓	0	0	0	0	0	0	0
a	0						
c	0						
b	0						
c	0						
f	0						

Dynamic programming approach for LCS:

if X[i] = Y[j] (If last characters match i.e.)

LCS[i][j] := LCS[i-1][j-1] + 1

else   (If last characters don’t match i.e.)

LCS[i][j] := max(LCS[i-1][j], LCS[i][j-1])

- Step 1: Consider only 1st character from both the strings, and find its LCS. Str1 = **A B C D A F** **Str2 = A C B C F**

Str1		**A**					
Str2	**0**	**0**					
A	**0**	**0+1** $\boxed{\mathbf{1}}$					

if X[i] = Y[j]   (If last characters match i.e.)

LCS[i][j] := LCS[i-1][j-1] + 1

else   (If last characters don't match i.e.)

LCS[i][j] := max(LCS[i-1][j], LCS[i][j-1])

- Now consider 2 characters from str1 and single character from str2.

Str1	j	A	B				
i str2	0	0	0				
A	0	1	max 1				

Str1		A	B	C			
str2	0	0	0	0			
A	0	1	1	max 1			

Str1		A	B	C	D	A	F
str2	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1

if X[i] = Y[j] (If last characters match i.e.)

LCS[i][j] := LCS[i-1][j-1] + 1

else (If last characters don't match i.e.)

LCS[i][j] := max(LCS[i-1][j], LCS[i][j-1])

if X[i] = Y[j] LCS[i][j] := LCS[i-1][j-1] + 1 else LCS[i][j] := max(LCS[i-1][j], LCS[i][j-1])

Str1		A	B	C	D	A	F
str2	0	0	0	0	0	0	0
A	0	1	1	0+1 1	1	1	1
C	0	1	1	$\boxed{2}$			

Str1		A	B	C	D	A	F
str2	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
C	0	1	1	2	max 2		

Str1		A	B	C	D	A	F
str2	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
C	0	1 max	1	2	2	2	2
B	0	1					

Str1		A	B	C	D	A	F
str2	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
C	0	1	1	2	2	2	2
B	0	1	1+1 $\boxed{2}$				

Str1		A	B	C	D	A	F
str2	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
C	0	1	1	2	2	2	2
B	0	1	2	0+1 2	2	2	2
C	0	1	2	3			

Str1		A	B	C	D	A	F
str2	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
C	0	1	1	2	2	2	2
B	0	1	2	2	2	2	2
C	0	1	2	3	3	3	3

Str1		A	B	C	D	A	F
str2	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
C	0	1	1	2	2	2	2
B	0	1	2	2	2	2	2
C	0	1	2	3	3	3	3
F	0	1	2	3	3	3	0+1 4

if X[i] = Y[j] (If last characters match i.e.)

LCS[i][j] := LCS[i-1][j-1] + 1

else (If last characters don’t match i.e.)

LCS[i][j] := max(LCS[i-1][j], LCS[i][j-1])

if X[i] = Y[j]   LCS[i][j] := LCS[i-1][j-1] + 1   else   LCS[i][j] := max(LCS[i-1][j], LCS[i][j-1])

Final Matrix:

Str1		A	B	C	D	A	F
str2	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
C	0	1	1	2	2	2	2
B	0	1	2	2	2	2	2
C	0	1	2	3	3	3	3
F	0	1	2	3	3	3	4

Total length = 4

F, C, B, A

Longest Common sequence:
A,B,C,F

Algorithm

```
function LCSLength(X[1..m], Y[1..n])
  C = array(0..m, 0..n)
  for i := 0..m              O(m)
    C[i][0] = 0
  for j := 0..n              O(n)
    C[0][j] = 0
  for i := 1..m              O(m*n)
     for j := 1..n
        if X[i] = Y[j]
           C[i][j] := C[i-1][j-1] + 1
        else
           C[i][j] := max(C[i][j-1], C[i-1][j])
  return C[m][n]
```

Complexity

- Time Complexity for LCS using dynamic programming is:

$T(n) = m(loop1) + n(loop2) + (m*n)(loop3)$

$T(n) = m + n + (mn)$

$T(n) = O(mn)$

if X[i] = Y[j]   LCS[i][j] := LCS[i-1][j-1] + 1   else   LCS[i][j] := max(LCS[i-1][j], LCS[i][j-1])

Solve Following Example:

1. X=BACDB, Y=BDCB

2. Str1=bcacbcab, Str2= bccabcc

if X[i] = Y[j] LCS[i][j] := LCS[i-1][j-1] + 1 else LCS[i][j] := max(LCS[i-1][j], LCS[i][j-1])

Solve Following Example:

1. X=BACDB, Y=BDCB

		0	1	2	3	4 $=n$
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
$m=5$	B	0	1	2	2	3

X = BACDB

Y = BDCB

LCS = BDB

LCS = BCB

if X[i] = Y[j]   LCS[i][j] := LCS[i-1][j-1] + 1   else   LCS[i][j] := max(LCS[i-1][j], LCS[i][j-1])

Solve Following Example:

1. X=BACDB, Y=BDCB

2. Str1=bcacbcab, Str2= bccabcc

		0	1	2	3	4 $=n$
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
$m=5$	B	0	1	2	2	3

X = BACDB

Y = BDCB

LCS = BDB

LCS = BCB

String 1 = bcacbcab; String 2 = bccabcc

		b	c	a	c	b	c	a	b
	0	0	0	0	0	0	0	0	0
b	0	1	1	1	1	1	1	1	1
c	0	1	2	2	2	2	2	2	2
c	0	1	2	2	3	3	3	3	3
a	0	1	2	3	3	3	3	4	4
b	0	1	2	3	3	4	4	4	5
c	0	1	2	3	4	4	5	5	5
c	0	1	2	3	4	4	5	5	5

Here Maximum Common Subsequence between String1 and String2 is of length 5.

MODULE 6

1. Genetic Algorithms

Genetic Algorithms :

Genetic algorithms (GAs) are used mainly for optimization problems for which the exact algorithms are of very low efficiency.

GAs search for good solutions to a problem from among a (large) number of possible solutions. The current set of possible solutions is used to generate a new set of possible solution.

They start with an initial set of possible solutions, and at each step they do the following:

Evaluate the current set of solutions (current generation).

Choose the best of them to serve as "parents" for the new generation, and construct the new generation.

The loop runs until :

-a specified condition becomes true,
e.g. a solution is found that satisfies the criteria for a "good" solution, or
-the number of iterations exceeds a given value, or
-no improvement has been recorded when evaluating the solutions.

Key issues here are:

How large to be the size of a population so that there is sufficient diversity?
Usually the size is determined through trial-and-error experiments.

How to represent the solutions so that the representations can be manipulated to obtain a new solution?

One approach is to represent the solutions as strings of characters (could be binary strings) and to use various types of "crossover" (explained below) to obtain a new set of solutions.

The strings are usually called "chromosomes"

how to evaluate a solution?.

The function used to evaluate a solution is called "fitness function" and it depends on the nature of the problem.

How to manipulate the representations in order to construct a new solution?

The method that is characteristic of GAs is to combine two or more representations by taking substrings of each of them to construct a new solution.

This operation is called "crossover".

How to choose the individual solutions that will serve as parents for the new generation?

The process of choosing parents is called "selection".

Various methods have been experimente.

The choice is dependent on the nature of the problem and the chosen representation.

One method is to choose parents with a probability proportional to their fitness.

This method is called "roulette wheel selection".

How to avoid convergence to a set of equal solutions?

The approach here is to change randomly the representation of a solution.
If the representation is a bit string we can flip bits. This operation is called "mutation".

Using the terminology above, we can outline the algorithms to obtain one generation:

compute the fitness of each chromosome
select parents based on the fitness value and a selection strategy
perform crossover to obtain new chromosomes
perform mutation on the new chromosomes (with fixed or random probability)

Examples:

The knapsack problem solved with GAs
The traveling salesman problem

Thank You