CS 91 USACO

Bronze Division

Unit 1: Introduction to Competitive Programming

LECTURE 3: COMPLETE SEARCH

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Modulo-Arithmetic
- Practice: Friday the Thirteen
- Complete Search
- Solution Space
  1. Subset
  2. Combination
  3. Permutation
  4. Pairing

Practice: Friday the Thirteen

SECTION 1

Problem Statement

- Is Friday the 13th really an unusual event?
- That is, does the 13th of the month land on a Friday less often than on any other day of the week? To answer this question, write a program that will compute the frequency that the 13th of each month lands on Saturday, Sunday, Monday, Tuesday, Wednesday, Thursday, and Friday over a given period of N years. The time period to test will be from January 1, 1900 to December 31, 1900+N-1 for a given number of years, N. N is positive and will not exceed 400.
- Note that the start year is NINETEEN HUNDRED, not NINETEEN NINETY.

Problem Statement

- There are few facts you need to know before you can solve this problem:
  - January 1, 1900 was on a Monday.
  - Thirty days has September, April, June, and November, all the rest have 31 except for February which has 28 except in leap years when it has 29.
  - Every year evenly divisible by 4 is a leap year (1992 = 4*498 so 1992 will be a leap year, but the year 1990 is not a leap year)
  - The rule above does not hold for century years. Century years divisible by 400 are leap years, all others are not. Thus, the century years 1700, 1800, 1900 and 2100 are not leap years, but 2000 is a leap year.
- Do not use any built-in date functions in your computer language.
- Don't just precompute the answers, either, please.

INPUT FORMAT (file friday.in):

- One line with the integer N.

SAMPLE INPUT:

```
20
```

OUTPUT FORMAT (friday.out):

- Seven space separated integers on one line. These integers represent the number of times the 13th falls on Saturday, Sunday, Monday, Tuesday, ..., Friday.

SAMPLE OUTPUT:

```
36 33 34 33 35 35 34
```

Nature of the Problem

- Modulo Arithmetic
- Accumulator of Remainder
- The answer can only be 0-6
- Application: Print Calendar, Determine a specific day's day, Chinese Zodiac

Main Ideas

- Count the total number of days from January 1, 1900 to a specific date, then take modulo.
- Taking results on partial sum is OK.
- Calculate the number of days for each 13^{th} day of any month between January 1, 1900 to the specific date.
- Use Histogram to tally how many Saturdays on 13, Sundays on 13, and etc.

Complete Search

SECTION 2

The Problem Solving Loop

1. Identify the problem.

2. Explore information and create ideas.

3. Select the best idea.

4. Build and test the idea.

5. Evaluate the results.

Complete Search

- Solving a problem using complete search is based on the ``Keep It Simple, Stupid'' principle. The goal of solving contest problems is to write programs that work in the time allowed, whether or not there is a faster algorithm.
- Complete search exploits the brute force, straight-forward, try-them-all method of finding the answer. This method should almost always be the first algorithm/solution you consider. If this works within time and space constraints, then do it: it's easy to code and usually easy to debug. This means you'll have more time to work on all the hard problems, where brute force doesn't work quickly enough.
- In the case of a problem with only fewer than a couple million possibilities, iterate through each one of them, and see if the answer works.

How to evaluate 'number of possibilities'

- How does one know how many evaluations of some criterion/criteria must be made to determine if brute force is reasonable? One must have an idea of the "rough number of operations" that will be required to solve the task for the highest possible input value(s).
- Such an evaluation is what the "big-O" order notation is all about. You'll see algorithms claiming to be O(N) or O(N log N) or O(N*N). This means that (roughly, in our general case) increasing the size of the input will increase the run time proportional to the argument of O().

How to evaluate 'number of possibilities'

- Thus, for an O(N) solution, doubling N will double the then number of 'operations' performed. However, for O(N*N), when N is, say, 1000 then the runtime is 1,000,000 times slower than the fundamental simple computation in the middle of the loop. This can add up quickly.
- Generally O(N*N) (or larger exponents for N) solutions start to get slow when N gets large, but some programming tasks limit N to a small enough value that N*N isn't so large as to preclude brute force.

How to evaluate 'number of possibilities'

- The most basic way to determine if your program has an O(N), O(N*N), or O(N*N*N) solution is to find the code that is surrounded by the greatest number of loops (i.e., nested most deeply) and tally the loop indexes. If the loops execute unconditionally, simply multiply the three loop sizes together:

```
for i = 1 to N
    for j = 1 to N
        for k = 1 to N
            [do some computing]
        end
    end
end
```

- Three nested loops, all size N, O(N*N*N).

How to evaluate 'number of possibilities'

- The inside computation in a typical contest problem is generally fairly quick, on the order of 10's or 100's of nanoseconds (billionths), occasionally a few microseconds (millionths). But, if N is 1000, an O(N*N*N) solution multiplies this by a billion, turning nanoseconds of inner loop computation into seconds of runtime, generally too slow.

How to evaluate 'number of possibilities'

- Recursive solutions must be thought through in parallel lines of thinking to the loop solutions above. Recursive solutions, though, incur the additional overhead of allocating stack space for local variables. Each nested recursion allocates a few or even many variables. By the time you're nesting 10,000 (est.) times deep, you're liable to exceed memory limits. Avoid having unused local variables. Calculate the maximum depth of recursion and make sure you won't use up your entire stack space.
- Note that sequential parts of a solution are dominated by the single slowest loop (recursion). Three O(N*N) loops and a single O(N*N*N) loop make for an O(N*N*N) execution speed.

Solution Space

SECTION 3

Symbol Set

- Each language will have an alphabet. The alphabet will define the word space. Word space usually can also be the solution space for some problem defined by the language.

name	R()	lgR()	characters
BINARY	2	1	01
DNA	4	2	ACTG
OCTAL	8	3	01234567
DECIMAL	10	4	0123456789
HEXADECIMAL	16	4	0123456789ABCDEF
PROTEIN	20	5	ACDEFGHIKLMNPQRSTVWY
LOWERCASE	26	5	abcdefghijklmnopqrstuvwxyz
UPPERCASE	26	5	ABCDEFGHIJKLMNOPQRSTUVWXYZ
BASE64	64	6	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdef ghijklmnopqrstuvwxyz0123456789+/
ASCII	128	7	*ASCII characters*
EXTENDED_ASCII	256	8	*extended ASCII characters*
UNICODE16	65536	16	*Unicode characters*

Demonstration Program

SYMBOLSETS.JAVA

Generating subsets

- To generate all the subset of a set = {1, 2, 3};
- You just basically need to generate an index number ranging from 0 to 2^3; (exclusive)
- Then, for each index number, you check each bit, if the bit 0 is 1, then the element 1 is included in the subset, if bit 0 is 0, then the element 1 is not included int the subset.
- The solution space size is 2^N.

Generating subsets

```
for (int b = 0; b < (1<<n); b++) {
  vector<int> v;
  for (int i = 0; i < n; i++) {
    if (b&(1<<i)) v.push_back(i+1);
  }
}
```

Demonstration Program

SUBSETS.JAVA

```java
public class Subsets
{
    static int[] a = {1, 2, 3, 4};
    public static void main(String[] args){
        ArrayList<ArrayList<Integer>> sets = new ArrayList<ArrayList<Integer>>();

        int N=3;

        for (int b=0; b<Math.pow(2, N); b++){
            ArrayList<Integer> set = new ArrayList<Integer>();
            for (int i=0; i<N; i++){
                int bi = 1;
                for (int j=0; j<i; j++) bi *=2;
                if ((b & bi)>0) set.add(a[i]);
             }
            sets.add(set);
         }
        System.out.println(sets);
    }
}
```

Iteration over DNA

- A DNA strand consist of Nucleotide types.
- There are 4 nucleotide types. Therefore, a DNA strand of length 3 will have 4^3 possible combinations. To generate all possible DNA combinations of a DNA strand of length 3. We can use Iteration of the symbol set for 3 times.

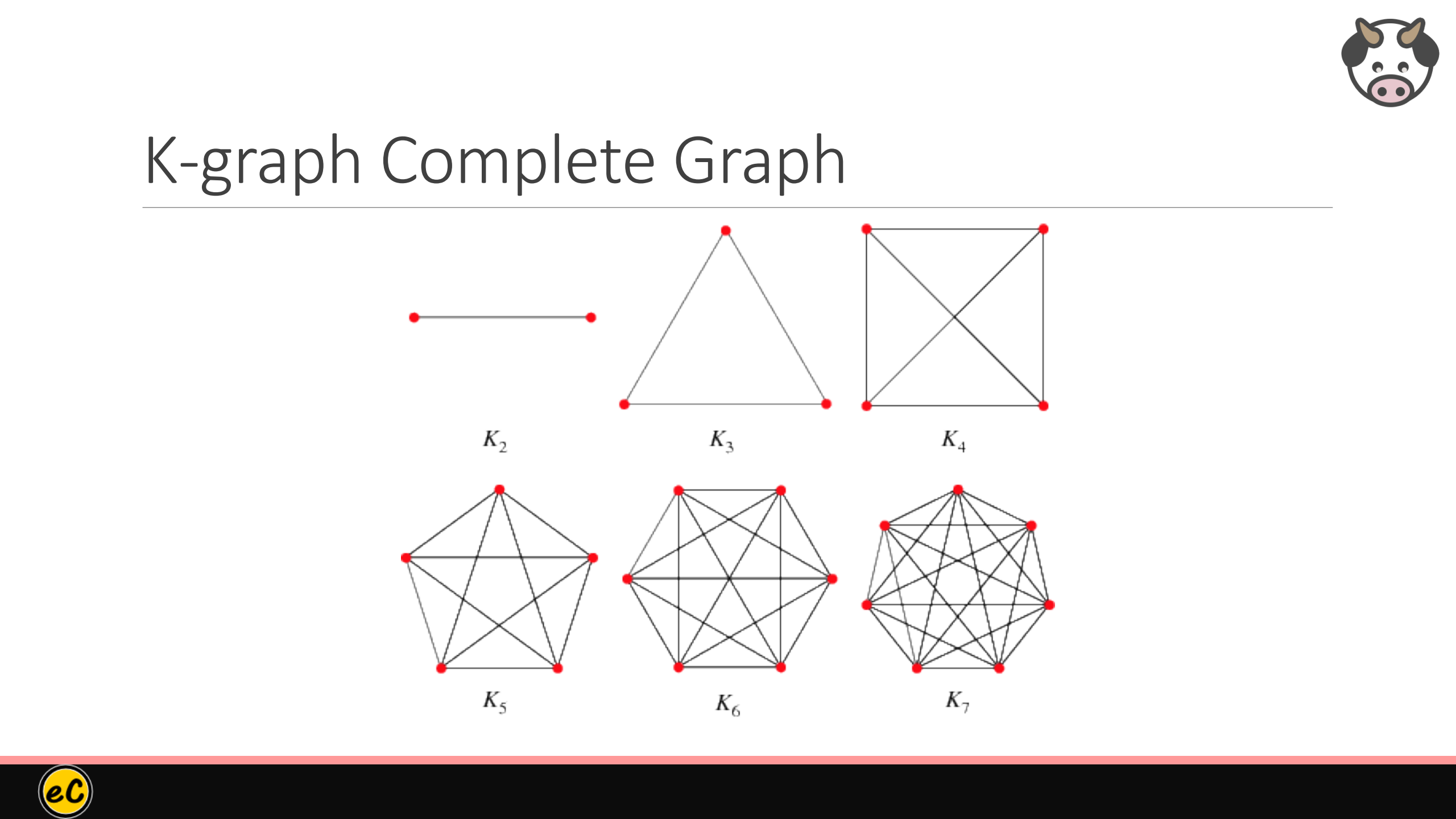
Demonstration Program

ITERATION.JAVA

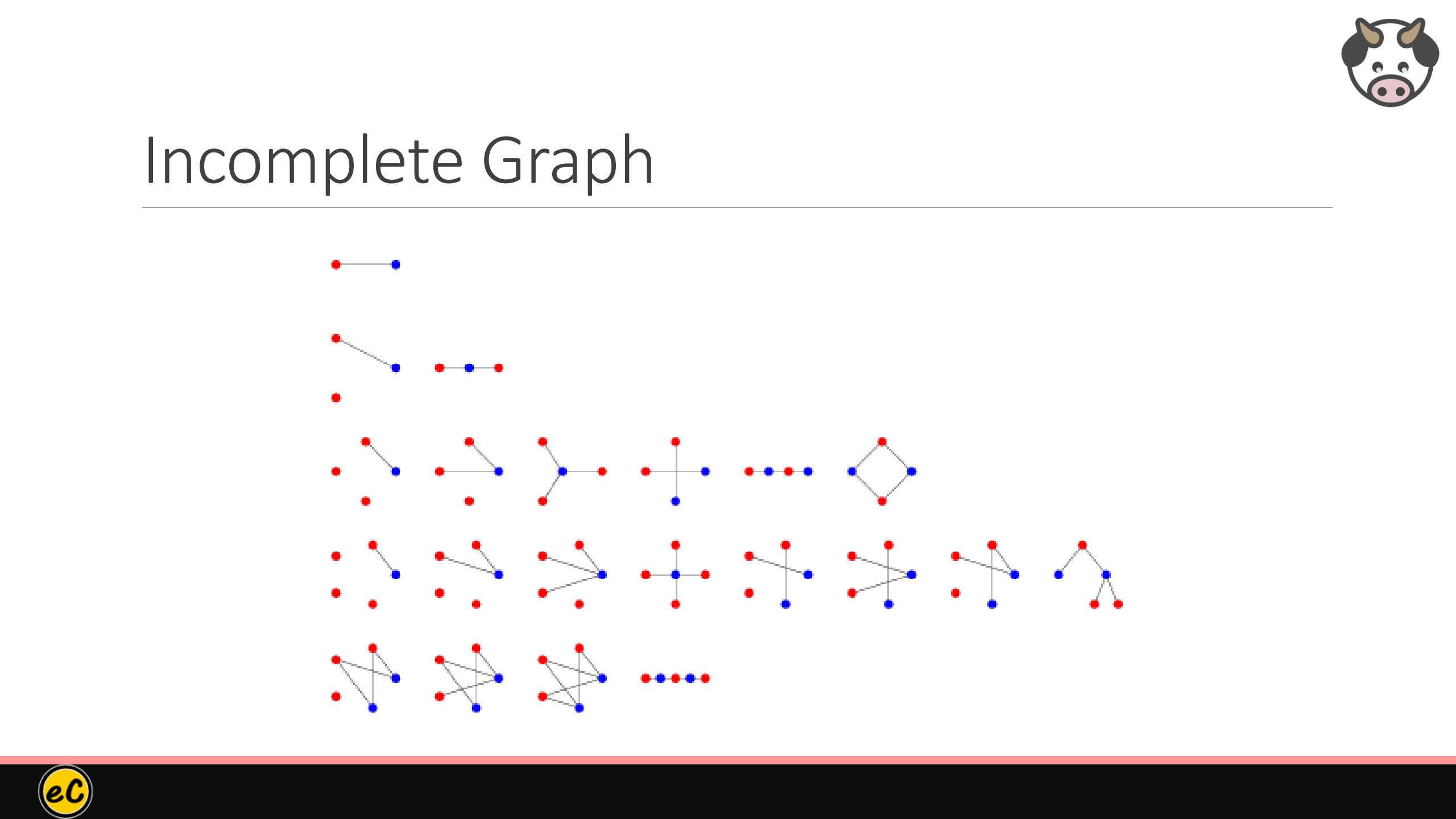
```java
import java.util.*;
public class Iteration
{
    public final static String DNA = "ACTG";

    public static void main(String[] args){
      System.out.print("\f");
      int N=3;
      int base = 1;
      for (int i=0; i<N; i++) base *=DNA.length();

      ArrayList<String> dna_set = new ArrayList<String>();
      for (int i=0; i<base; i++){
            String b = Integer.toString(i, DNA.length());
            int zero = N-b.length();
            for (int k=0; k<zero; k++) b = "0"+b; // stuff b in front
            String d = "";
            for (int j=0; j<N; j++){
                int x = Integer.parseInt(b.substring(j, j+1));
                d += DNA.charAt(x);
            }
            dna_set.add(d);
        }
      System.out.println("DNA Listing: ");
      for (String dna: dna_set){
           System.out.println(dna);
        }
    }
}
```

Combinations

SECTION 4



K-graph Complete Graph



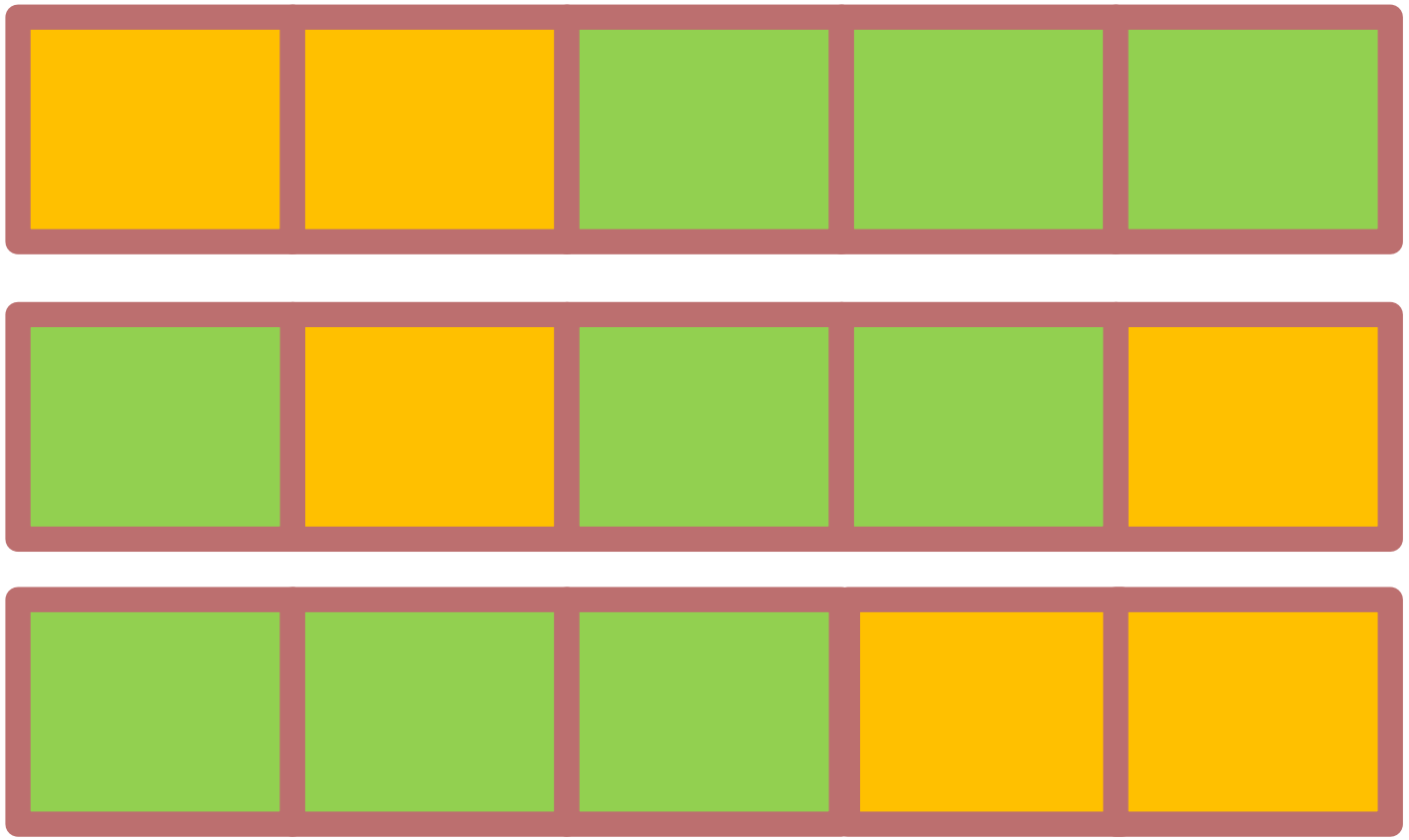
Incomplete Graph

C_2^n Combination

- Let's generate all the length 2 combination for "ABCDE"
- Use K-graph methods

```
[AB, AC, AD, AE, BC, BD, BE, CD, CE, DE]
```

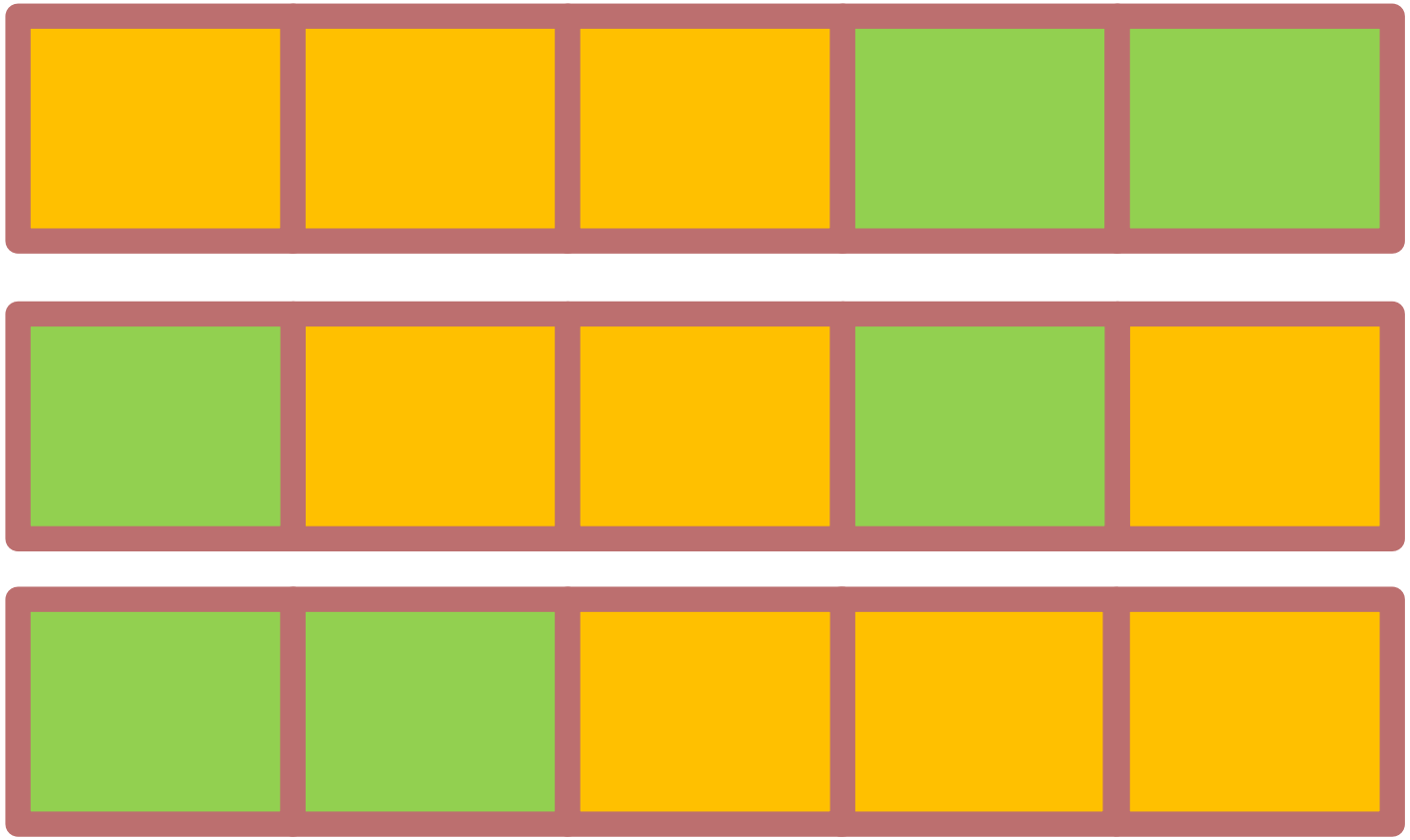
C_2^n Combination



C_2^n Combination

```
2  public class Cn2
3  {
4      static String A = "ABCDE";
5
6      public static void main(String[] args){
7          ArrayList<String> cn2 = new ArrayList<String>();
8          int N = A.length();
9          for (int i=0; i<N-1; i++){
10             for (int j=i+1; j<N; j++){
11                 String s = "";
12                 s+= A.charAt(i);
13                 s+= A.charAt(j);
14                 cn2.add(s);
15             }
16         }
17         System.out.println(cn2);
18     }
19 }
```

C_3^n Combination



C_3^n Combination

```
public static void main(String[] args){
    ArrayList<String> cn3 = new ArrayList<String>();
    int N = A.length();
    for (int i=0; i<N-2; i++){
      for (int j=i+1; j<N-1; j++){
        for (int k=j+1; k<N; k++){
          String s = "";
          s+= A.charAt(i);
          s+= A.charAt(j);
          s+= A.charAt(k);
          cn3.add(s);
        }
      }

    }
    System.out.println(cn3);
 }
```

C_2^n and C_3^n

BlueJ: Terminal Window - CompleteSearch

Options

```
[AB, AC, AD, AE, BC, BD, BE, CD, CE, DE]
[ABC, ABD, ABE, ACD, ACE, ADE, BCD, BCE, BDE, CDE]
```

Can only enter input while your programming is running

Permutations

SECTION 5

Anagram

- The Anagram of a string is all the possible permutation of a String.
- Say, we got a string of “0123”. The anagram of all of these permutations are listed like the right hand-side.

- Anagram is taught to Ryan’s Python course.

0123	2013
0132	2031
0213	2103
0231	2130
0312	2301
0321	2310
1023	3012
1032	3021
1203	3102
1230	3120
1302	3201
1320	3210

Idea:

- Recursive programming.

permute(“0123”) => “0”+ permute(“123”)

=> “1”+permute(“023”)

=> “2”+permute(“013”)

=> “3”+permute(“012”)

Put all results from sub-cases into an array list. If a string’s length is 0 or one. The only possible solution is itself. But you need to make a list of a single string. Say, [“0”]

Anagram

```
public static ArrayList<String> permute(String s){
    if (s.length()==1) {
        ArrayList<String> a = new ArrayList<String>();
        a.add(s);
        return a;
     }
    ArrayList<String> a = new ArrayList<String>();
    for (int i=0; i<s.length(); i++){
        String head = s.substring(i, i+1);  // "0", "1", "2", "3"
        ArrayList<String> b = permute(s.substring(0, i)+s.substring(i+1));
        for (int j=0; j<b.size(); j++){
            a.add(head+b.get(j));
         }
     }
    return a;
}
```

P_m^n Permutations

- The Anagram of a string is all the possible permutation of a String. The P_m^n is a shorter format of Anagram.
- Say, we got a string of “0123”.
- If we want to do P_m^n,
- We should get 01, 02, 03, 12, 13, 10, 20, 21, 23, 30, 31, 32 totally 12 possible permutations.

Permutation is a shorter format of Anagram

- So, if P is the length of a string and we pick M letters for permutation.
- Then, the recursion should stop at P-M+1.
- P_3^5 , P-M+1 = 5 – 3 + 1 = 3. When s.length() is subcase is 3, we list all the 1-character strings for remaining characters.

```
P(5, 2):
AB AC AD AE BA BC BD BE CA CB CD CE DA DB DC DE EA EB EC ED
P(5, 3):
ABC ABD ABE ACB ACD ACE ADB ADC ADE AEB
AEC AED BAC BAD BAE BCA BCD BCE BDA BDC
BDE BEA BEC BED CAB CAD CAE CBA CBD CBE
CDA CDB CDE CEA CEB CED DAB DAC DAE DBA
DBC DBE DCA DCB DCE DEA DEB DEC EAB EAC
EAD EBA EBC EBD ECA ECB ECD EDA EDB EDC
```

Pairing

SECTION 6

Pairing

- Pairing is the process to group elements into pairs. Let's assume there are even number of elements to be paired (2N).
- There will be totally, (2N-1) * (2N-3) * … * 1 possible pairings.

Pairing

- For example, there are 6 elements (ABCDEF).
- There will be totally 15 possible pairings: 5 * 3 * 1
- (A, B)(C, D)(E, F), (A, B)(C, E)(D, F), (A, B)(C, F)(D, E)
- (A, C)(B, D)(E, F), (A, C)(B, E)(D, F), (A, C)(B, F)(D, E)
- (A, D)(B, C)(E, F), (A, D)(B, E)(C, F), (A, D)(B, F)(C, D)
- (A, E)(B, C)(D, F), (A, E)(B, D)(C, F), (A, E)(B, F)(C, D)
- (A, F)(B, C)(D, E), (A, F)(B, D)(C, E), (A, F)(B, E)(C, F)

Basic Paring Algorithm

Demo Program: pairings.java

```
32  public static void pair(int[] partners){
33      int i=0;
34      while (i<alphabet.length && partners[i]!=-1) i++;
35      if (i==alphabet.length){
36          System.out.println(Arrays.toString(partners));
37          return;
38      }
39
40      for (int j=i+1; j<alphabet.length; j++){
41          if (partners[j]==-1) {
42              partners[i] = j;
43              partners[j] = i;
44              pair(partners);
45              partners[i] = -1;
46              partners[j] = -1;
47          }
48      }
49  }
```

Find the first unpaired element

If all paired, generate all paringing

Iterate through the second unpaired element

Generate pairing Strings with Alphabet

Demo Program: pairing.java

```
13  public static String generate(int[] partners, String[] alphabet){
14      Set s = new HashSet<Integer>();
15      boolean first = true;
16      String text = "";
17      for (int i=0; i<partners.length; i++){
18          if (!s.contains(i)){
19              s.add(i);
20              s.add(partners[i]);
21              if (first){
22                  text += ""+alphabet[i]+alphabet[partners[i]];
23                  first = false;
24              }
25              else {
26                  text += "-"+alphabet[i]+alphabet[partners[i]];
27              }
28          }
29      }
30      return text;
31  }
```

Generate pairing Strings with Alphabet

Demo Program: pairing.java

```
33  public static void pair(int[] partners){
34      //System.out.println(Arrays.toString(partners));
35      int i=0;
36      while (i<alphabet.length && partners[i]!=-1) i++;
37      if (i==alphabet.length){
38          //System.out.println(Arrays.toString(partners));
39          pairings.add(generate(partners, alphabet));
40          return;
41      }
42
43      for (int j=i+1; j<alphabet.length; j++){
44          if (partners[j]==-1) {
45              partners[i] = j;
46              partners[j] = i;
47              pair(partners);
48              partners[i] = -1;
49              partners[j] = -1;
50          }
51      }
52  }
```

```
AB-CD-EF
AB-CE-DF
AB-CF-DE
AC-BD-EF
AC-BE-DF
AC-BF-DE
AD-BC-EF
AD-BE-CF
AD-BF-CE
AE-BC-DF
AE-BD-CF
AE-BF-CD
AF-BC-DE
AF-BD-CE
AF-BE-CD
```