Lebanese University
Faculty of Science
Computer Science BS Degree

Advanced Algorithms I3341

Dr Siba Haidar & Dr Antoun Yaacoub

Course Chapters

as per the textbook

1. Introduction
2. Data Structures and Libraries
3. Problem Solving Paradigms
4. Graph
5. Mathematics
6. String Processing
7. Computational Geometry

Problem Solving Paradigms

Chapter 3

Chapter Outline

1. Complete Search = Brute Force
   a. Iterative Complete Search
   b. Recursive Complete Search
   c. Tips
2. Divide and Conquer
   a. Interesting Usages of Binary Search
3. Greedy
   a. Examples
4. Dynamic Programming
   a. DP Illustration
   b. Classical Examples
   c. Non-Classical Examples

Overview and Motivation

Overview and Motivation

- need to master problem solving paradigms
- to be able to attack a given problem with the appropriate tool
- example: 4 simple tasks
  - array A containing n ≤ 10K small integers ≤ 100K
  - A = {10, 7, 3, 5, 8, 2, 9}, n = 7
  - 1. Find largest & smallest element → 10 & 2
  - 2. Find k_{th} smallest element → for k = 2 → 3
  - 3. Find largest gap g / x, y ∈ A & g = |x − y| → 8
  - 4. Find longest increasing subsequence → {3, 5, 8, 9}

Brute Force?

- 1. Find largest & smallest element
  - → $O(n)$ Complete Search
- 2. Find kth smallest element
  - find smallest and replace with big number
  - repeat k times
  - if k = n/2 → $O(n \times n/2) \sim O(n^2)$
  - better sort then choose kth → $O(nlogn)$ → Divide and Conquer
  - better $O(n)$ → also Divide and Conquer
- 3. Find largest gap g / x, y ∈ A & g = |x – y|
  - consider every pair → $O(n^2)$
  - largest – smallest → $O(n)$ → Greedy
- 4. Find longest increasing subsequence
  - try all $O(2^n)$ possible subsequences
  - not feasible for all n ≤ 10K
  - → $O(n^2)$ DP
  - → $O(nlogn)$ Greedy

Advise

- do not just memorize solutions
- remember and internalize the thought process and problem solving strategies used

Complete Search

Complete Search

- = brute force
- = recursive backtracking

- traverse entire (or part of) search space
- during search
  - allowed to prune (not to explore) parts
- develop a Complete Search solution
  - when clearly no other algorithm available
    - ex: enumerating all permutations of {0, 1, 2, . . .,N − 1} → $O(N!)$
  - or when better algorithms overkill as input size is small
    - ex: answering Range Minimum Queries on static arrays with N ≤ 100 solvable $O(N)$
- may receive a Time Limit (TL) but no Wrong Answer (WA)
- if < time limit → implement

n	Worst AC Algorithm	Comment
$\leq [10..11]$	$O(n!), O(n^6)$	e.g. Enumerating permutations (Section 3.2)
$\leq [15..18]$	$O(2^n \times n^2)$	e.g. DP TSP (Section 3.5.2)
$\leq [18..22]$	$O(2^n \times n)$	e.g. DP with bitmask technique (Section 8.3.1)
≤ 100	$O(n^4)$	e.g. DP with 3 dimensions + $O(n)$ loop, ${}_nC_{k=4}$
≤ 400	$O(n^3)$	e.g. Floyd Warshall's (Section 4.5)
$\leq 2K$	$O(n^2 \log_2 n)$	e.g. 2-nested loops + a tree-related DS (Section 2.3)
$\leq 10K$	$O(n^2)$	e.g. Bubble/Selection/Insertion Sort (Section 2.2)
$\leq 1M$	$O(n \log_2 n)$	e.g. Merge Sort, building Segment Tree (Section 2.3)
$\leq 100M$	$O(n), O(\log_2 n), O(1)$	Most contest problem has $n \leq 1M$ (I/O bottleneck)

Table 1.4: Rule of thumb time complexities for the 'Worst AC Algorithm' for various single test-case input sizes n, assuming that your CPU can compute 100M items in 3s.

Iterative Complete Search

Two Nested Loops: UVa 725 - Division

Write a program that finds and displays all pairs of 5-digit numbers that between them use the digits 0 through 9 once each, such that the first number divided by the second is equal to an integer N, where $2 \leq N \leq 79$. That is,

$$\frac{abcde}{fghij} = N$$

where each letter represents a different digit. The first digit of one of the numerals is allowed to be zero.

Input

Each line of the input file consists of a valid integer N. An input of zero is to terminate the program.

Output

Your program have to display ALL qualifying pairs of numerals, sorted by increasing numerator (and, of course, denominator).

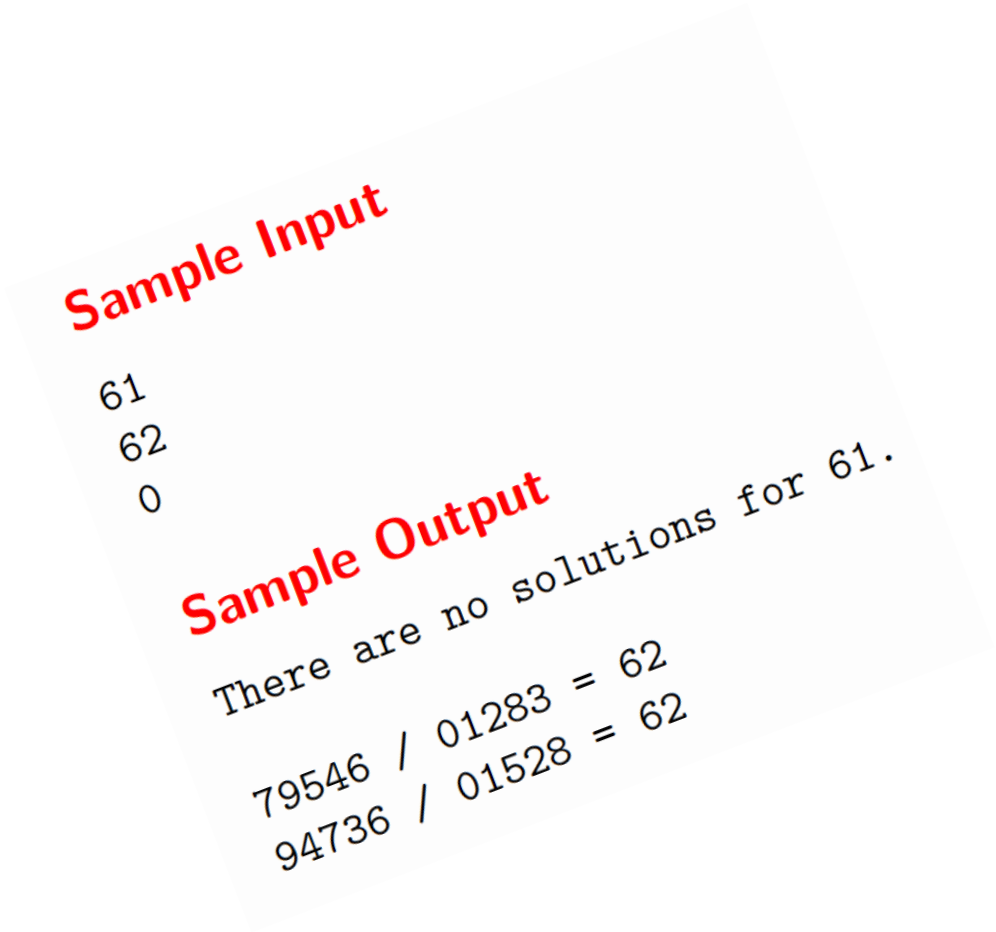
Your output should be in the following general form:

$xxxxx$ / $xxxxx$ = N
$xxxxx$ / $xxxxx$ = N
.
.

In case there are no pairs of numerals satisfying the condition, you must write '`There are no solutions for` N.'. Separate the output for two different values of N by a blank line.

Two Nested Loops: UVa 725 - Division

- Abridged problem statement:
  - find and display all pairs of 5 digit numbers
  - such that: 1st / 2nd = N
  - $abcde/fghij = N$
  - $2 \leq N \leq 79$
  - each letter = different digit
  - 1st digit allowed to be zero



Two Nested Loops: UVa 725 - Division

- $abcde/fghij = N$
- quick analysis
  - $01234 \leq fghij \leq 98765$
  - $\approx 100K$
- better bound
  - $01234 \leq fghij \leq \frac{98765}{N}$
  - $\approx 50K$ for $N = 2$
  - $N \nearrow, possibilities \searrow$
- for each $fghij$
  - $abcde = fghij \times N$
  - then check uniqueness of digits
  - doubly nested loop
  - → ≈ 50K × 10 = 500K op. / test case
  - small

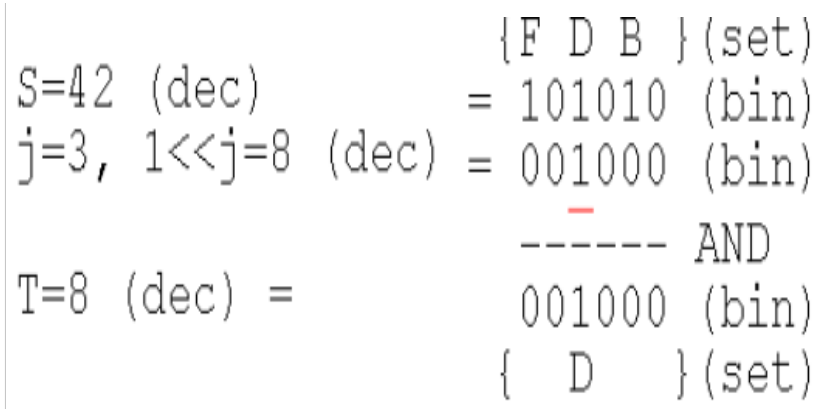
```
int main() {
  bool first = true;
  int N;
  while (scanf("%d", &N), N) {
    if (!first) printf("\n");
    first = false;
    bool noSolution = true;
    for (int fghij = 1234; fghij <= 98765/N; ++fghij)
    {
      int abcde = fghij*N;
      int tmp, used = (fghij < 10000);
      // if digit f=0, then we have to flag it
      tmp = abcde;
      while (tmp) {
        used |= 1<<(tmp%10);
        tmp /= 10;}
      tmp = fghij;
      while (tmp) {
        used |= 1<<(tmp%10);
        tmp /= 10;}
      if (used == (1<<10)-1) {
        // if all digits are used, print it
        printf("%05d / %05d = %d\n", abcde, fghij, N);
        noSolution = false;}
    }
    if (noSolution)
      printf("There are no solutions for %d.\n", N);
  }
  return 0;
}
```

BitMask next slide

format specifier to print exactly 5 digits

Bitmasks

- lightweight small sets of booleans
- int = sequence of bits
- bitwise manipulation → more efficient



- example
  - to set / turn on the j-th item (0-based indexing) of the set,
  - use the bitwise OR operation S |= (1 << j)
- https://visualgo.net/en/bitmask

Many Nested Loops: UVa 441 - Lotto

In the German Lotto you have to select 6 numbers from the set {1,2,...,49}. A popular strategy to play Lotto - although it doesn't increase your chance of winning — is to select a subset S containing k ($k > 6$) of these 49 numbers, and then play several games with choosing numbers only from S.

For example, for $k = 8$ and $S = \{1, 2, 3, 5, 8, 13, 21, 34\}$ there are 28 possible games: [1,2,3,5,8,13], [1,2,3,5,8,21], [1,2,3,5,8,34], [1,2,3,5,13,21], ..., [3,5,8,13,21,34].

Your job is to write a program that reads in the number k and the set S and then prints all possible games choosing numbers only from S.

Input

The input file will contain one or more test cases.

Each test case consists of one line containing several integers separated from each other by spaces. The first integer on the line will be the number k ($6 < k < 13$). Then k integers, specifying the set S, will follow in ascending order.

Input will be terminated by a value of zero (0) for k.

Output

For each test case, print all possible games, each game on one line.

The numbers of each game have to be sorted in ascending order and separated from each other by exactly one space. The games themselves have to be sorted lexicographically, that means sorted by the lowest number first, then by the second lowest and so on, as demonstrated in the sample output below.

The test cases have to be separated from each other by exactly one blank line. Do not put a blank line after the last test case.

Many Nested Loops: UVa 441 - Lotto

- single loop → easy
- must master nested
- Abridged problem statement:
  - Given k integers
  - 6 < k < 13
  - enumerate all possible subsets of size 6 of these integers
  - in sorted order

- 6 nested loops
- largest test case k = 12
- $c_{12}^{6} = \frac{12!}{6! \times (12-6)!} = 924$
- → small

```
int main() {
  bool first = true;
  int k;
  while (cin >> k, k) {
    if (!first) cout << endl;
    first = false;
    int S[16];
    for (int i = 0; i < k; ++i)
      scanf("%d", &S[i]);
    for (int a = 0  ; a < k-5; ++a)
      for (int b = a+1; b < k-4; ++b)
        for (int c = b+1; c < k-3; ++c)
          for (int d = c+1; d < k-2; ++d)
            for (int e = d+1; e < k-1; ++e)
              for (int f = e+1; f < k  ; ++f)
                printf("%d %d %d %d %d\n",
                S[a],S[b],S[c],S[d],S[e],S[f]);
  }
  return 0;
}
```

Loops + Pruning: UVa 11565 - Simple Equations

Let us look at a boring mathematics problem. :-)
We have three different integers, x, y and z, which satisfy the following three relations:

- $x + y + z = A$
- $xyz = B$
- $x^2 + y^2 + z^2 = C$

You are asked to **write a program** that solves for x, y and z for given values of A, B and C.

Input

The first line of the input file gives the number of test cases N ($N < 20$). Each of the following N lines gives the values of A, B and C ($1 \leq A, B, C \leq 10000$).

Output

For each test case, output the corresponding values of x, y and z. If there are many possible answers, choose the one with the least value of x. If there is a tie, output the one with the least value of y.
If there is no solution, output the line '`No solution.`' instead.

Loops + Pruning: UVa 11565 - Simple Equations

- Abridged problem statement:
  - given 3 ints A, B, C
  - $1 \le A,B,C \le 10000$
  - find 3 other distinct integers x, y, z
  - such that
    - $x + y + z = A$
    - $x \times y \times z = B$
    - $x^2 + y^2 + z^2 = C$
- 3rd equation → good start
  - if
    - C = 10000, y=1, z=2
    - → range of x is [−100 . . . 100]
  - same reasoning for y and z
- triply-nested iterative solution
  - $201 \times 201 \times 201 \approx 8M$ ops /test case

short circuit lightweight check $x \neq y \neq z$

```
// 0.150s
bool sol = false; int x, y, z;
for (x = -100; x <= 100; ++x)
  for (y = -100; y <= 100; ++y)
    for (z = -100; z <= 100; ++z)
      if ((y != x) && (z != x) && (z != y)
            && (x+y+z == A)
            && (x*y*z == B)
            && (x*x + y*y + z*z == C))
      {
        if (!sol)
          printf("%d %d %d\n", x, y, z);
          sol = true;
      }
```

Loops + Pruning: UVa 11565 - Simple Equations

- better solution
- sum =A
  - x↗, y and z ↘
- product= B
  - since $x < y < z$
    - (for ≠ solutions →choose smaller x)
  - x max can be $\approx y \approx z$
  - $x \times x \times x < x \times y \times z = B$
  - $x < \sqrt[3]{B} < 10000$
  - $x \in [-22 \cdots 22]$
- + use break | continue
- ++ need few other optimizations
  - to solve harder one:
  - UVa 11571 - Simple Equations – Extreme!!

```
// 0.000s
int main() {
  int N;
  scanf("%d", &N);
  while (N--) {
    int A, B, C;
    scanf("%d %d %d", &A, &B, &C);
    bool sol = false;
    int x, y, z;
    for (x = -22; (x <= 22) && !sol; ++x)
      if (x*x <= C)
        for (y = -100; (y <= 100) && !sol; ++y)
          if ((y != x) && (x*x + y*y <= C))
            for (z = -100;(z <= 100) && !sol; ++z)
              if ((z != x) && (z != y)
                  && (x+y+z == A)
                  && (x*y*z == B)
                  && (x*x + y*y + z*z == C))
              {
                printf("%d %d %d\n", x, y, z);
                sol = true;
              }
    if (!sol)
      printf("No solution.\n");
  }
  return 0;
}
```

Permutations: UVa 11742 - Social Constraints

Socializing can be a very complicated thing among teenagers. For example, finding a good seating arrangement in a movie theater can be a difficult task. Here is a list of constraints that could potentially apply to two individuals A and B in this situation:

- if A and B are dating, then they must sit beside each other
- if A and B are fighting, then they cannot sit beside each other
- if A and B have just broke up, then they must sit at opposite ends of the row

Teenage politics is a complicated thing meaning the constraints can get even more complicated than those listed above. However, we restrict this problem to a particular form of constraint that simply specifies a lower or upper bound on the number of seats separating two specific individuals.

The group arrives after everyone else watching the show has been seated. By some stroke of luck, there are exactly as many open seats as there are teenagers and all of these seats appear consecutively in the front row. How many possible seating arrangements satisfy the constraints?

"Under the table for two?"

Permutations: UVa 11742 - Social Constraints

- Abridged problem statement:
  - there are n movie goers
    - $0 < n \leq 8$
  - they will sit in the front row
  - in n consecutive open seats
  - there are m seating constraints
    - $0 \leq m \leq 20$
  - constraint a b c =
    - movie goer a and movie goer b
    - must be c seats apart
      - at most if c>0
      - at least if c<0
  - How many possible seating arrangements are there?
- key
  - explore all permutations
  - $O(m \times n!)$
  - largest case $20 \times 8! = 806400$ ops
- set counter = 0
  - explore $n!$ permutations
  - if perm. satisfies all m constraints
    - counter ++

```
const int MAX_n = 8;
const int MAX_m = 20;

int main() {
  int n, m;
  while (scanf("%d %d", &n, &m), (n || m)) {
    int a[MAX_m], b[MAX_m], c[MAX_m];
    for (int j = 0; j < m; ++j)
      scanf("%d %d %d", &a[j], &b[j], &c[j]);
    int p[MAX_n];
    for (int i = 0; i < n; ++i)
      p[i] = i;
    int ans = 0;
    do {
      bool all_ok = true;
      for (int j = 0; (j < m) && all_ok; ++j) {
        int pos_a = p[a[j]], pos_b = p[b[j]];
        int d_pos = abs(pos_a-pos_b);
        if (c[j] > 0) all_ok = (d_pos <= c[j]);
        else          all_ok = (d_pos >= abs(c[j]));
      }
      if (all_ok) ++ans;
    }
    while (next_permutation(p, p+n));
    printf("%d\n", ans);
  }
  return 0;
}
```

how to permute? next slide

Dr Siba Hadar & Dr Antoun Yaacoub

Permutations in Java

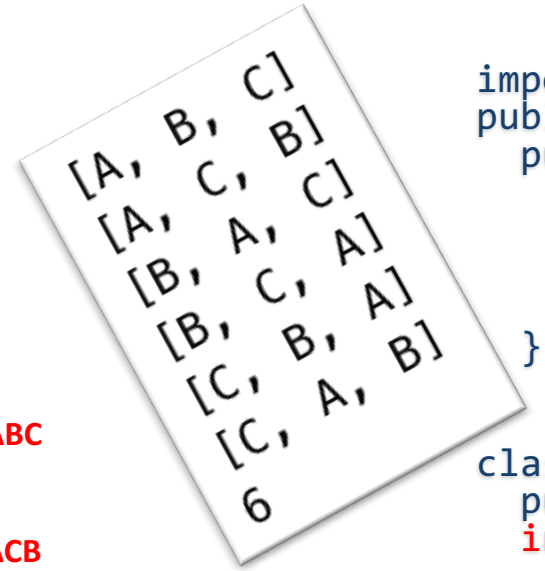
- recursive + param index n < size
  - base case index 2 (last)
    - no more permutations to do
    - so handle
  - recursive call
    - for loop
    - permute index n with all following indexes
    - recursive call for next index (n+1)
    - permute back for next iteration
- initial call n = 0

```
class Permute {
  public static <T>
  void permute (List<T> items, int n) {
    if (n == items.size() - 1) {
      // handle a permutation
    }
    else {
      for (int i = n; i < items.size(); i++) {
        Collections.swap(items, i, n);
        permute(items, n + 1);
        Collections.swap(items, i, n);
      }
    }
  }
}
```

Permutations in Java

- example permute ABC 0
- 0!=2
- for loop i=0..2

- i=0
  - swap 0 0 → ABC
  - permute ABC 1
    - 1!=2
    - for loop i=1..2
    - i=1
      - swap 1 1 → ABC
      - permute ABC 2 → base case **ABC**
      - swap 1 1 → ABC
    - i=2
      - swap 2 1 → ACB
      - permute ACB 2 → base case **ACB**
      - swap 2 1 → ABC
  - swap 0 0 → ABC
- i=1
  - swap 1 0 → BAC
  - permute BAC 1 → same as before
    - → **BAC**
    - → **BCA**
  - swap 1 0 → ABC
- i=2
  - swap 2 0 → CBA
  - permute CBA 1 → same as before
    - → **CBA**
    - → **CAB**
  - swap 2 0 → ABC




```
import java.util.*;
public class Main {
  public static void main(String[] args) {
    List<String> letters =
       Arrays.asList("A", "B", "C");
    System.out.println(
      Permute.<String>permute(letters, 0));
  }


class Permute {
  public static <T>
  int permute (List<T> items, int n) {
    if (n == items.size() - 1) {
      // handle a permutation
      System.out.println(items);
      return 1;
    }
    int c=0;
    for (int i = n; i < items.size(); i++) {
      Collections.swap(items, i, n);
      c+=permute(items, n + 1);
      Collections.swap(items, i, n);
    }
    return c;
  }
}
```

Permutations in C++

- same thing as in java
  - but
- next_permutation
  - already in std++

- example outputs
  - A B C
  - A C B
  - B A C
  - B C A
  - C A B
  - C B A
  - 6

```
// permute

#include "stdc++.h"
using namespace std;

int main() {
    char p[3]={'A','B','C'};
    int ans = 0;
    do {
        printf("%c %c %c\n", p[0],p[1],p[2]);
        ++ans;
    } while (next_permutation(p, p+3));
    printf("%d\n", ans);
    return 0;
}
```


Dr Siba Hadar & Dr Antoun Yaacoub

Subsets: UVa 12455 - Bars

Some things grow if you put them together.

We have some metallic bars, theirs length known, and, if necessary, we want to solder some of them in order to obtain another one being exactly a given length long. No bar can be cut up. Is it possible?

Input

The first line of the input contains an integer,t, $0 \leq t \leq 50$, indicating the number of test cases. For each test case, three lines appear, the first one contains a number n, $0 \leq n \leq 1000$, representing the length of the bar we want to obtain. The second line contains a number p, $1 \leq p \leq 20$, representing the number of bars we have. The third line of each test case contains p numbers, representing the length of the p bars.

Output

For each test case the output should contain a single line, consists of the string 'YES' or the string 'NO', depending on whether solution is possible or not.

Subsets: UVa 12455 - Bars

- Abridged problem statement1:
  - Given a list l containing n integers
  - $1 \leq n \leq 20$
  - is there a subset of list l that sums to another given integer X?
- try all possible subsets
  - 2^n
  - sum each subset in $O(n)$
  - check if sum = X
  - $O(n \times 2^n)$
  - largest test case
  - n = 20
  - $20 \times 2^{20} \approx 21M$
  - large
  - but still viable because…
    - bit manipulation is fast
- easy solution
  - binary representation of integers
  - $0 \cdots 2^n - 1$
  - to describe all possible subsets

```
for (i = 0; i < (1 << n); i++) {
  sum = 0;
  for (int j = 0; j < n; j++)
    if (i & (1 << j)) // j is turned on?
      sum += l[j];
  if (sum == X) break;
}
```

Recursive Complete Search

Simple Backtracking: UVa 750 - 8 Queens Chess Problem

In chess it is possible to place eight queens on the board so that no one queen can be taken by any other. Write a program that will determine all such possible arrangements for eight queens given the initial position of one of the queens.

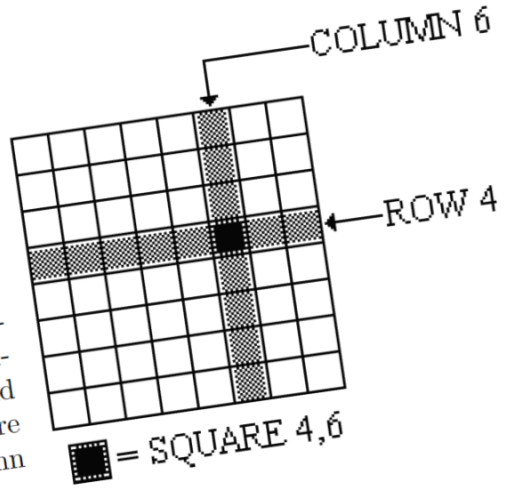
Do not attempt to write a program which evaluates every possible 8 configuration of 8 queens placed on the board. This would require 8^8 evaluations and would bring the system to its knees. There will be a reasonable run time constraint placed on your program.

Input

The first line of the input contains the number of datasets, and it's followed by a blank line.

Each dataset contains a pair of positive integers separated by a single space. The numbers represent the square on which one of the eight queens must be positioned. A valid square will be represented; it will not be necessary to validate the input.

To standardize our notation, assume that the upper left-most corner of the board is position (1,1). Rows run horizontally and the top row is row 1. Columns are vertical and column 1 is the left-most column. Any reference to a square is by row then column; thus square (4,6) means row 4, column 6.



Each dataset is separated by a blank line.

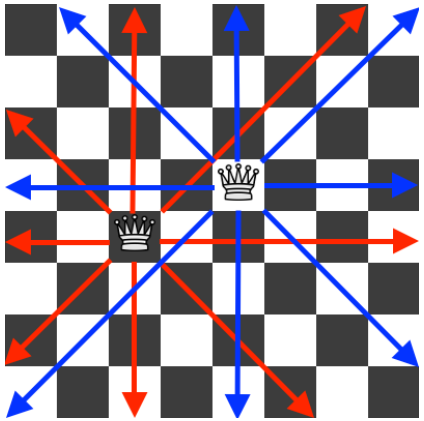
Output

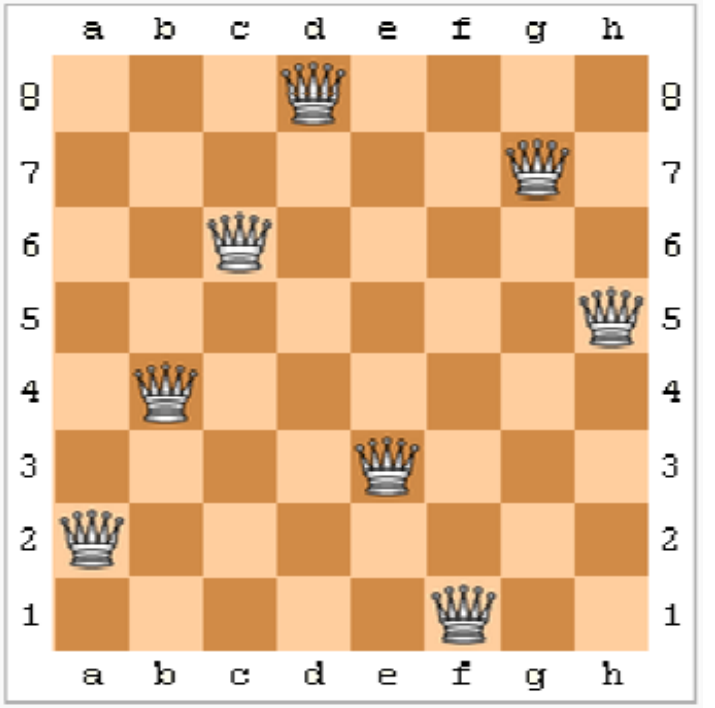
Output for each dataset will consist of a one-line-per-solution representation.

Each solution will be sequentially numbered $1 \ldots N$. Each solution will consist of 8 numbers. Each of the 8 numbers will be the ROW coordinate for that solution. The column coordinate will be indicated by the order in which the 8 numbers are printed. That is, the first number represents the ROW in which the queen is positioned in column 1; the second number represents the ROW in which the queen is positioned in column 2, and so on.

Simple Backtracking: UVa 750 - 8 Queens Chess Problem

- Abridged problem statement:
  - chess 8x8 board
  - place 8 queens / no attacks possible
  - given 1 queen position (a, b)
  - → determine
    - all such possible arrangements
    - output possibilities in lexicographic order
- naïve ways (TL)
  - choose 8 out of 64 cells…
  - $c_{64}^{8} \approx 4B$ possibilities 😐
- insight 1
  - put 1 queen in each column
  - $8^8 \approx 17M$ 🙄
- →++ easy optimizations



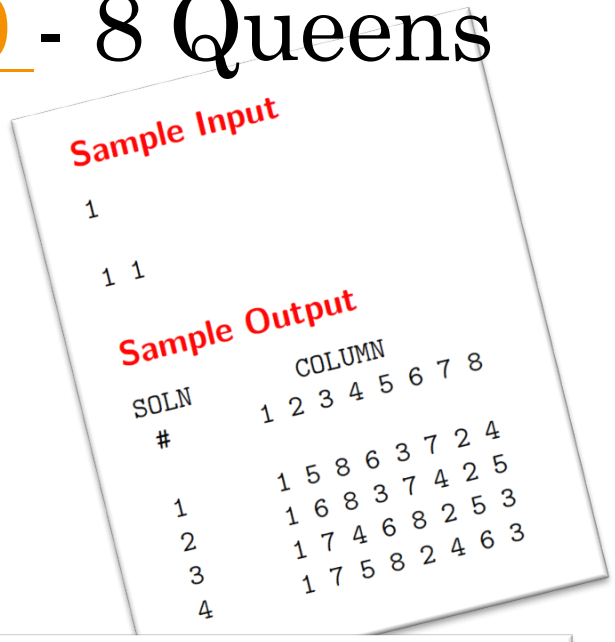


Simple Backtracking: UVa 750 - 8 Queens Chess Problem

- →++ easy optimizations:

- no share same column | row
- → find valid permutations rows positions
- $8! \approx 40K$
- fast

- example
  - row[i] = r
    - queen in column i is placed in row r
  - row = {1, 3, 5, 7, 2, 0, 6, 4}
  - Figure 3.1


Sample Input

1

1 1

Sample Output

SOLN COLUMN
1 2 3 4 5 6 7 8

1 1 5 8 6 3 7 2 4
2 1 6 8 3 7 4 2 5
3 1 7 4 6 8 2 5 3
4 1 7 5 8 2 4 6 3

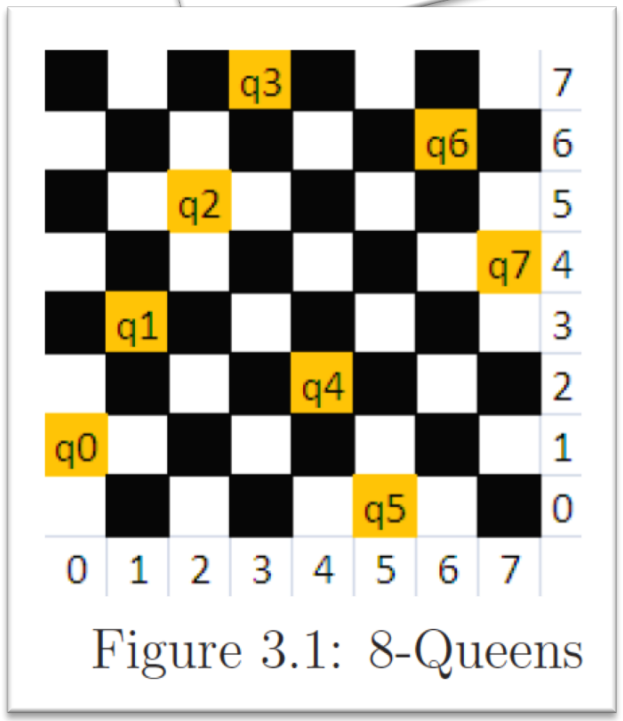


Figure 3.1: 8-Queens

Simple Backtracking: UVa 750 - 8 Queens Chess Problem

- more
- no share same diagonal
  - queen $A(i, j)$
  - queen $B(k, l)$
  - if $|i - k| \neq |j - l|$→OK
- recursive backtracking
  - places queens 1 by 1
  - in columns 0 to 7
  - observing all constraints above
- check candidate solution
  - if ∃1 queen / satisfy input constraints: row[b] == a
  - →print it
- sub O(n!) → AC verdict

```
int row[8], a, b, lineCounter;

bool canPlace(int r, int c) {
  for (int prev = 0; prev < c; ++prev)
    if ((row[prev] == r) || (abs(row[prev]-r) == abs(prev-c)))
      return false;
  return true;
}

void backtrack(int c) {
  if ((c == 8) && (row[b] == a)) {
    printf("%2d      %d", ++lineCounter, row[0]+1);
    for (int j = 1; j < 8; ++j) printf(" %d", row[j]+1);
    printf("\n");
    return;
  }

  for (int r = 0; r < 8; ++r) {
    if ((c == b) && (r != a)) continue;
    if (canPlace(r, c))
      row[c] = r, backtrack(c+1);
  }
}

int main() {
  int TC; scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &a, &b); --a; --b;
    memset(row, 0, sizeof row);
    lineCounter = 0;
    printf("SOLN       COLUMN\n");
    printf(" #      1 2 3 4 5 6 7 8\n\n");
    backtrack(0);
    if (TC) printf("\n");
  }
  return 0;
}
```

base case

early pruning

recurse only if!

notice the comma

0-based indexing

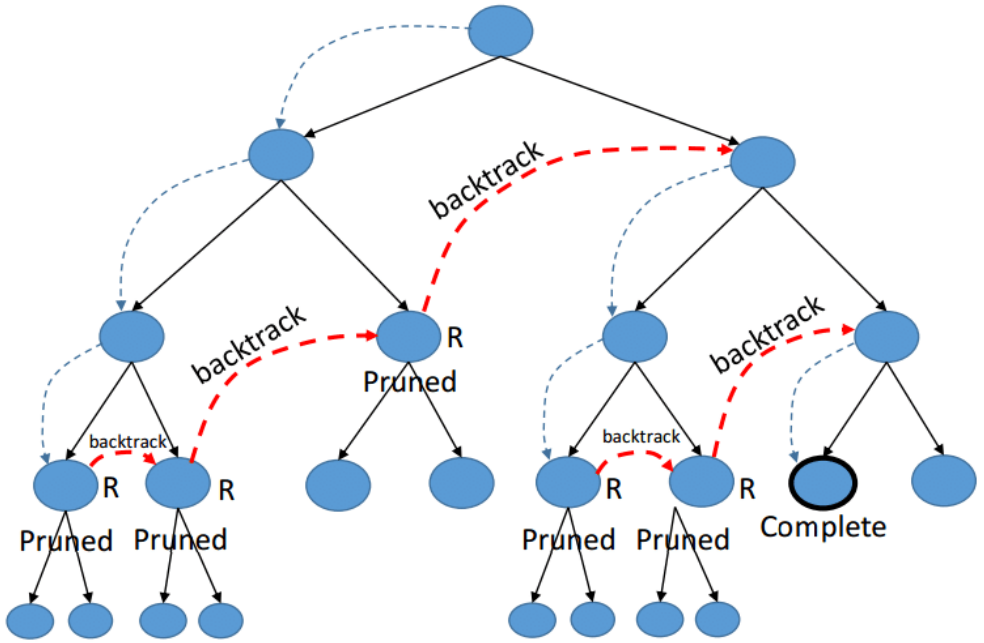
fill array with zeros

Dr Siba Hadar & Dr Antoun Yaacoub

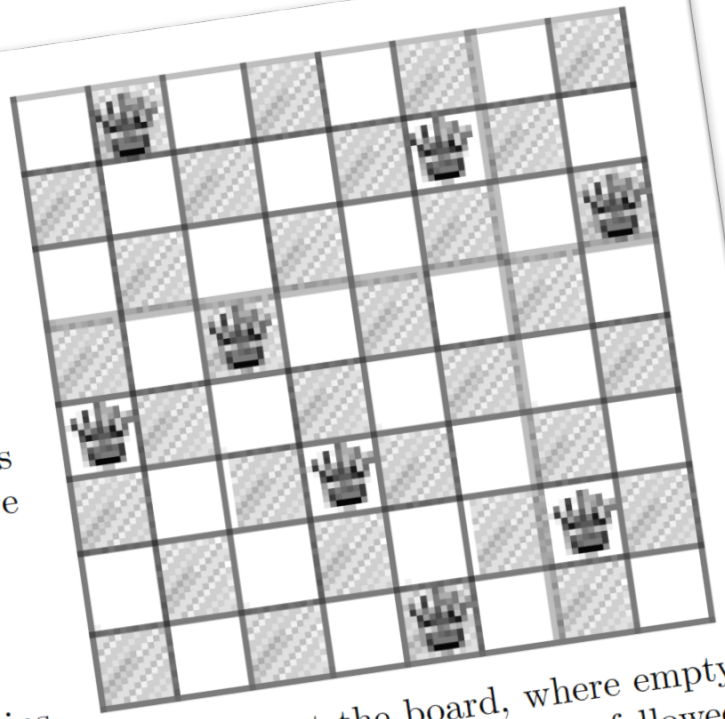
So what is Backtracking?

- general algorithm
  - for finding all (or some) solutions
  - to constraint satisfaction problems
  - incrementally builds candidates to solutions
  - + abandons a bad candidate →backtrack
- only for problems which
  - admit concept of
    - partial candidate solution
  - relatively quick test
    - valid | not valid
- faster > brute force enum
  - eliminates many candidates with a single test

```
procedure EXPLORE(node n)
    if REJECT(n) then return
    if COMPLETE(n) then
        OUTPUT(n)
    for n_i : CHILDREN(n) do EXPLORE(n_i)
```



More Challenging Backtracking: UVa 11195 - Another n-Queen Problem



I guess the n-queen problem is known by every person who has studied backtracking. In this problem you should count the number of placement of n queens on an $n * n$ board so that no two queens attack each other. To make the problem a little bit harder (easier?), there are some bad squares where queens cannot be placed. Please keep in mind that bad squares cannot be used to block queens' attack.

Even if two solutions become the same after some rotations and reflections, they are regarded as different. So there are exactly 92 solutions to the traditional 8-queen problem.

Input

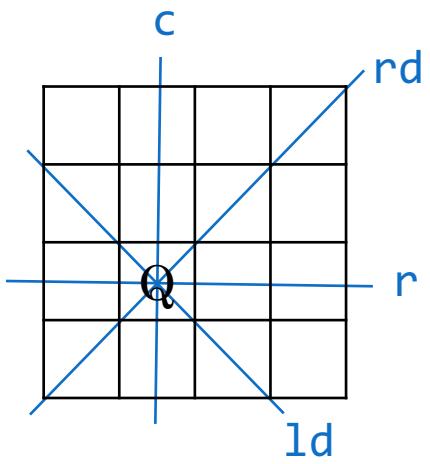
The input consists of at most 10 test cases. Each case contains one integers n $(3 \leq n \leq 15)$ in the first line. The following n lines represent the board, where empty squares are represented by dots '.', bad squares are represented by asterisks '*'. The last case is followed by a single zero, which should not be processed.

Output

For each test case, print the case number and the number of solutions.

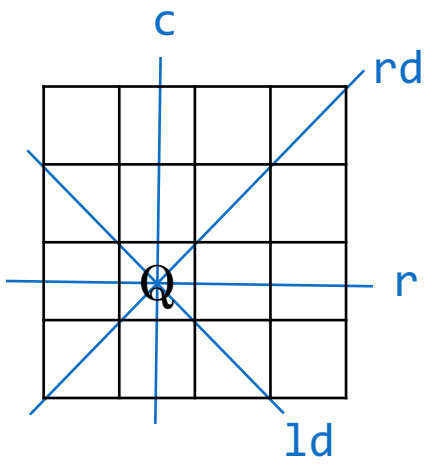
More Challenging Backtracking: UVa 11195 - Another n-Queen Problem

- Abridged problem statement:
  - given an $n \times n$ chessboard
  - 3 < n < 15
  - some cells are bad
  - how many ways to place n queens?
- recursive backtracking
  - not fast enough
    - for n = 14 and no bad cells
  - sub-O(n!)
    - OK for n = 8
    - KO for n = 14



More Challenging Backtracking: UVa 11195 - Another n-Queen Problem

- instead of canPlace
  - store info with 3 bool arrays
- initially all set to false
  - n rows (rw)
  - $2 \times n - 1$ left diagonals (ld)
  - $2 \times n - 1$ right diagonals (rd)
- $Q(r, c)$
  - $rw[r] = true$
  - all (a, b), s.t. $|r - c| = |a - b|$ occupied
    - $r - c = a - b$
      - $\rightarrow ld[r - c + n - 1] = true$
    - $r + c = a + b$
      - $\rightarrow rd[r + c] = true$
- ++ cond:
  - $board[r][c] \neq bad!$




```
rw=[   0010]
ld=[0010000]
rd=[0001000]
```

More Challenging Backtracking: UVa 11195 - Another n-Queen Problem

- instead of canPlace
  - store info with 3 bool arrays
- initially all set to false
  - n rows (rw)
  - $2 \times n - 1$ left diagonals (ld)
  - $2 \times n - 1$ right diagonals (rd)
- $Q(r, c)$
  - $rw[r] = true$
  - all (a, b), s.t. $|r - c| = |a - b|$ occupied
    - $r - c = a - b$
      - $\rightarrow ld[r - c + n - 1] = true$
    - $r + c = a + b$
      - $\rightarrow rd[r + c] = true$
- ++ cond:
  - $board[r][c] \neq bad!$

```
void backtrack(int c) {

  if (c == n) { ans++; return; }

  for (int r = 0; r < n; r++)

    if (board[r][c] != '*' && !rw[r]
        && !ld[r - c + n - 1] && !rd[r + c]) {

      rw[r] = ld[r - c + n - 1]
            = rd[r + c]
            = true;

      backtrack(c + 1);

      rw[r] = ld[r - c + n - 1]
            = rd[r + c]
            = false;
    }
}
```

global

flag off

restore because this time we want to count all cases

Dr Siba Hadar & Dr Antoun Yaacoub

Visualization

- https://visualgo.net/en/recursion

Tips

Tip 1: Filtering versus Generating

Filters

- examine candidate solutions
- choose correct
- usually 'filter' programs are written iteratively
- easier to code
- run slower
- difficult to prune search space iteratively
- do complexity analysis → decide if filter is good enough or need to create a generator

Generators

- build solutions
- prune invalid partial solutions
- usually 'generator' programs are written recursively
  - → easier to implement this way
- gives us greater flexibility for pruning search space

Tip 2: Prune Infeasible/Inferior Search Space Early

- when generating solutions using recursive backtracking
  - may encounter a partial solution that will never lead to a full solution
  - → prune the search there
  - explore other parts of search space
- rule of thumb
  - "The earlier you can prune the search space, the better."
- in other problems
  - compute the 'potential worth'
  - if < worth of current best found
  - prune search there

Tip 3: Utilize Symmetries

- some problems have symmetries
- exploit to reduce execution time
- ex
  - 8-queens problem
  - 92 solutions
  - only 12 unique
- however sometimes considering symmetries can complicate code

Tip 4: Pre-Computation a.k.a. Pre-Calculation

- generate tables or other DS to accelerate lookup
- →Pre-Computation
  - trades memory/space for time
- rarely be used for recent programming contest problems

Tip 5: Try Solving the Problem Backwards

- some contest problems look far easier
- when they are solved ‘backwards’
  - from a less obvious angle
- than when they are solved using a frontal attack
  - from the more obvious angle
- be prepared to attempt unconventional approaches to problems

Tip 6: Optimizing Your Source Code

- tricks to optimize
- understanding computer hardware and how it is organized
  - I/O
  - memory
  - cache behavior …
- → design better code

- 1) C++ >> Java
  - faster
  - Java users get extra time
- 2) IO
  - C/C++ users
    - use the faster C-style
    - scanf/printf >> cin/cout
  - Java users
    - use the faster BufferedReader / BufferedWriter classe (skeleton):

```
public class Main {
  public static void main (String args[]) throws IOException {
    Main myWork = new Main();
    myWork.Begin();
  }
  void Begin() throws IOException {
    Scanner sc = new Scanner(System.in);
    PrintWriter out = new PrintWriter(System.out);
    int tc = sc.nextInt();
    out.print("something\n");
  }
}
class Scanner {
  StringTokenizer st;
  BufferedReader br;
  public Scanner(InputStream s){
    br=new BufferedReader(new InputStreamReader(s));
  }
  public String next() throws IOException {
    while (st == null || !st.hasMoreTokens())
      st = new StringTokenizer(br.readLine());
    return st.nextToken();
  }
  public int nextInt() throws IOException {
    return Integer.parseInt(next()); }
  public long nextLong() throws IOException {
    return Long.parseLong(next()); }
  public String nextLine() throws IOException {
    return br.readLine();}
  public boolean ready() throws IOException {
    return br.ready();}
}
```

Java Skeleton

Dr Siba Hadar & Dr Antoun Yaacoub

Tip 6: Optimizing Your Source Code

- 3) in C++ STL algorithm::sort
  - quicksort>> heapsort
- 4) access 2D array in a row major fashion
- 5) Bit manipulation >> array of booleans
  - bitmask in Section 2.2
  - if > 64 bits → use the C++ STL bitset
- 6) low level data types
  - bigger array >> vector
  - 32-bit ints >> 64-bit long longs
- 7) Java
  - ArrayList>>Vector
  - StringBuilder >> StringBuffer
  - others are thread safe → not needed here
- 8) declare most DS global
  - enough memory for largest input
  - >> function arguments
  - multiple test cases → clear/reset
- 9) iterative >> recursive
  - faster
- 10) array access in loops
  - store in local variable temp = A[i]
  - faster if read only
- 11) in C/C++ macros | inline functions can reduce runtime
- 12)strings
  - C++ users: C-style character arrays >> C++ STL string
  - Java users: Be careful String objects immutable

Tip 7: Use Better Data Structures & Algorithms

- self explanatory
- >> Tip 6