Problem Solving:

Backtracking

April 2019

Honguk Woo

Algorithm paradigms

- General approaches to construction of efficient solutions to problems
  - Can be used as a reference to design a new algorithm to problems

  - We learned *"Divide and Conquer"*
    - Divide a problem instance into smaller sub-instances of the same problem, solve these recursively, and then put solutions together to a solution of the given instance
    - *e.g, Binary Search, Mergesort, Qsort …*
    - Relevant to many algorithms that can be implemented by using recursion

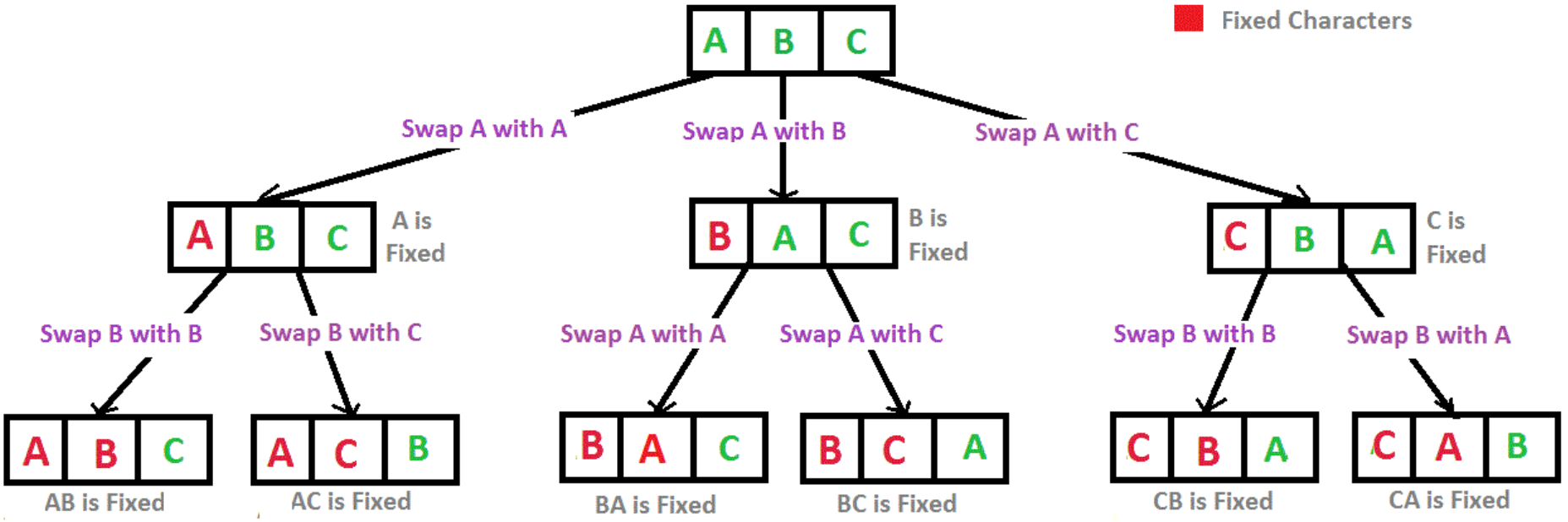
  - *There are many other algorithm paradigms*
    - ***Bruce forth***
    - ***Backtracking***
    - *Dynamic Programming*
    - *Greedy*

Backtracking

- Brute Force (Exhaustive Search)
  - a general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement
    - e.g., sequential search
  - Recursion can be used to implement Brute force algorithms
    - e.g., “ABC” string permutation

- Backtracking (Recursion with Pruning)
  - a general algorithmic technique that considers searching every possible combination
  - the search process can be pruned to avoid considering cases that don't look promising
  - *Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, that incrementally builds candidates to the solutions, and abandons each partial candidate (“backtracks”) as soon as it determines that the candidate cannot possibly be completed to a valid solution. (definition from wikipedia)*

Permutations of “ABC” string




```
void permute(char *a, int l, int r) {
   if (l == r) printf("%s\n", a);
   else {
       for (i = l; i <= r; i++) {
          swap((a+l), (a+i));
          permute(a, l+1, r);
          swap((a+l), (a+i));
       }
   }
}
```

```
char str[] = “abc”;
Permute(str, 0, strlen(str) -1);
```


https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/

Q : N-Queens Problem

- Given a chess board having *n* ×*n* cells, we need to place *n* queens in such a way that no queen is attacked by any other queen. A queen can attack horizontally, vertically and diagonally
  (meaning that **no two queens can be in the same row, column, diagonal**)

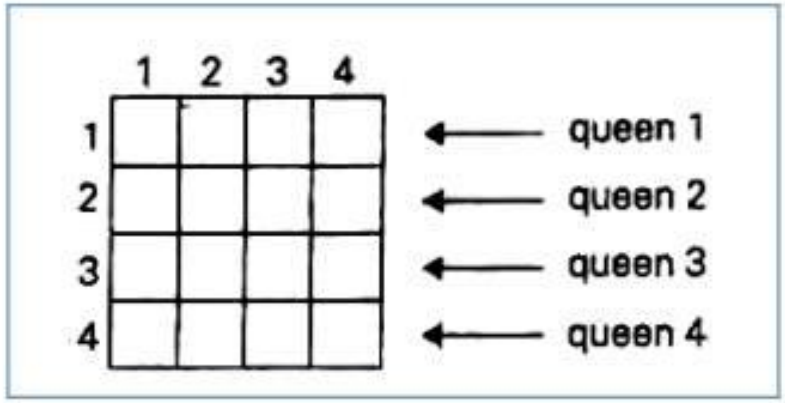
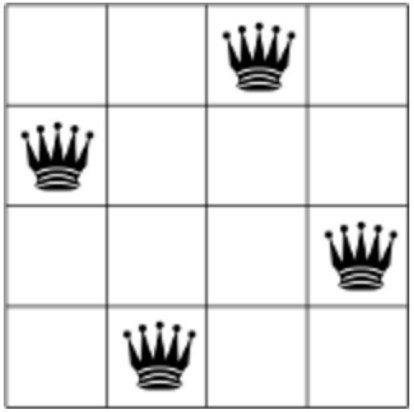


Fig: Board for the Four-queens problem

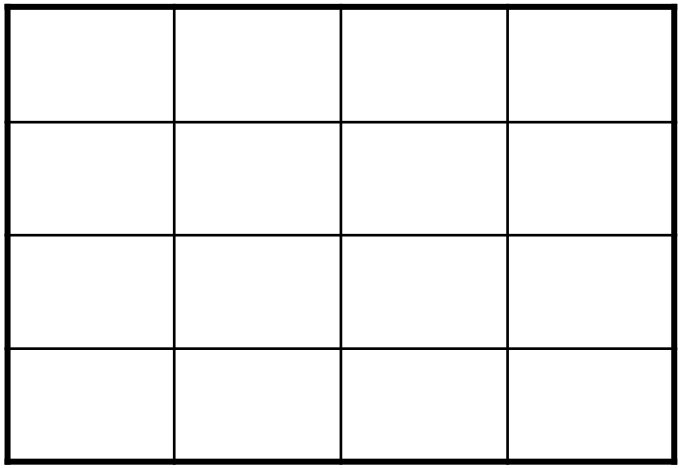


Solution for N-Queens

- no solution for n < 4

- n=4 case
  - list all case systematically
  - test each case if it is a solution
  - ${}_{16}C_4$ cases – n^2Cn cases

  - better way?



N-Queens : Efficiency

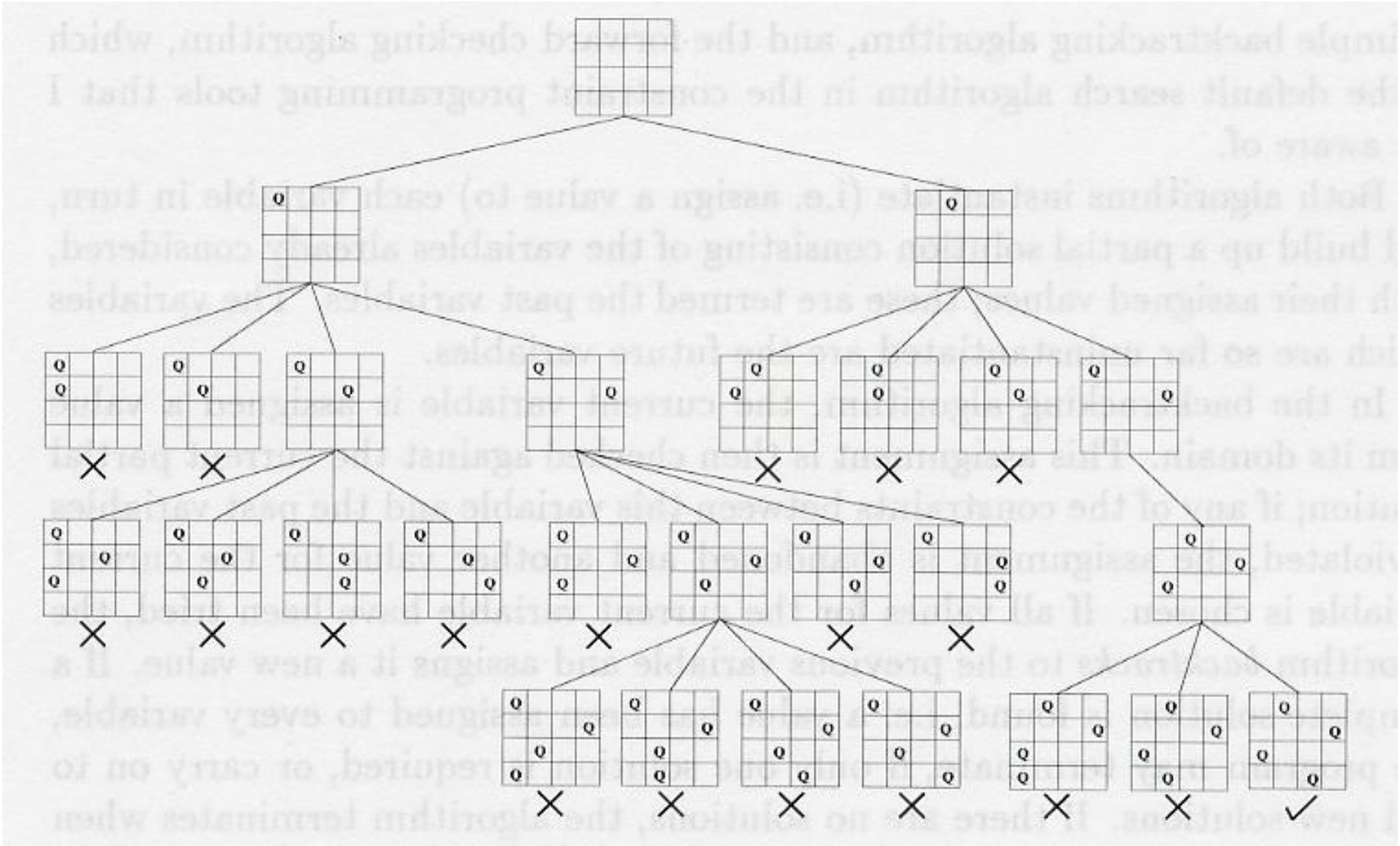
- Solution space of the n-queens problem
  - there are n^2 cells (n X n board)
  - each cell either has a queen (TRUE) or not (FALSE)
  - total combinations 2^n
    - 8-queens problem ~ $1.84 * 10^{19}$
- In case of 8-queens
  - If we think that each queen can be at any place
    - the 1st queen = 64 cases
    - the 2nd queen = 64 cases
    - :
    - the 8th queen = 64 cases
    - TOTAL **$64^8 = 2.81 * 10^{14}$ cases**
  - If we think that each queen can be at any place but at a different row
    - the 1st queen = 8 cases
    - the 2nd queen = 8 cases
    - :
    - the 8th queen = 8 cases
    - TOTAL **$8^8 = 1.67 * 10^7$ cases**

Backtracking general structure

- Solution set : vector $a = (a_1, a_2, \ldots, a_n)$

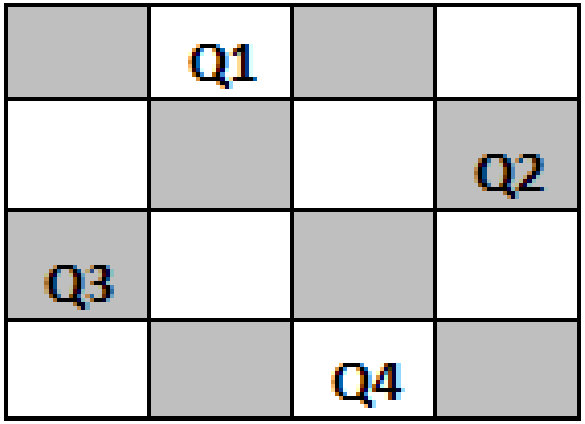
1. Given a partial solution $a = (a_1, \ldots, a_k)$
2. if it is a solution, do something
3. else check if partial solution a can be extendible to some complete solution
4. if yes, add a_{k+1} to the solution vector, and recur

N-Queens : backtracking

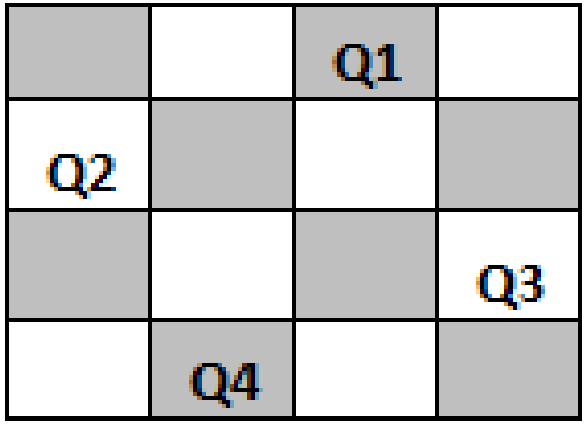


N-Queens : safe configurations

- [2, 4, 1, 3]
- [3, 1, 4, 2]



Solution 1



Solution 2

(not backtracking) Bruce forth

```
#include <stdio.h>
#include <stdlib.h>

int sol = 0;

int isSafe(int a[], int r){
  for(int i=0; i<r; i++) {
    for(int j = 0; j<r; j++) {
      if (i != j) {
        if (a[i] == a[j] ) return 0;
        else if (abs(i-j) == abs(a[i]-a[j])) return 0;
      }
    }
  }
  return 1;
}


void bt(int a[], int l, int r){
  if (l >= r) {
    sol++;
    if (isSafe(a, r) > 0) {
      printf("%d : ", sol);
      for (int i = 0; i < r; i++)
        printf("%d", a[i]);
      printf("\n");
    }
  } else {
    for(int i=1; i<=r; i++){
      a[l] = i;
      bt(a, l+1, r);
    }
  }
}
```

8-queens

```
int main() {
  int s[] = {1, 1, 1, 1, 1, 1, 1, 1};
  bt(s, 0, 8);
  return 0;
}
```

hong@Ubuntu-V:~/myLec/2019.s/backtracking$ time ./bt2
1 : 15863724
2 : 16837425
3 : 17468253
4 : 17582463
5 : 24683175
;
;
90 : 82531746
91 : 83162574
92 : 84136275

real	0m1.224s
user	0m0.517s
sys	0m0.000s

backtracking

```
include <stdio.h>
#include <stdlib.h>

int sol = 0;

int isSafeYet(int a[], int last){
  for(int i=0; i<=last; i++) {
    for(int j = 0; j<=last; j++) {
      if (i != j) {
        if (a[i] == a[j] ) return 0;
        else if (abs(i-j) == abs(a[i]-a[j])) return 0;
      }
    }
  }
  return 1;
}


void bt(int a[], int l, int r){
  if (l >= r) {
    sol++;
    printf("%d : ", sol);
    for (int i = 0; i < r; i++)
      printf("%d", a[i]);
    printf("\n");
  }
  else {
      for(int i=1; i<=r; i++){
        a[l] = i;
        if ( isSafeYet(a, l) > 0)
          bt(a, l+1, r);
      }
  }
}
```

8-queens

```
int main() {
  int s[] = {1, 1, 1, 1, 1, 1, 1, 1};
  bt(s, 0, 8);
  return 0;
}
```

```
hong@Ubuntu-V:~/myLec/2019.s/backtracking$ time ./bt3
1 : 15863724
2 : 16837425
3 : 17468253
4 : 17582463
5 : 24683175
6 : 25713864
;
;
90 : 82531746
91 : 83162574
92 : 84136275

real    0m0.006s
user    0m0.000s
sys     0m0.003s
```

General code structure for Backtracking

1. Given a partial solution $a = (a_1, \ldots, a_k)$
2. if it is a solution, do something
3. else check if partial solution a can be extendible to some complete solution
4. if yes, add a_{k+1} to the solution vector, and recur

```
void backtrack(int a[], int k, int input) {
  int c[MAXCAND];
  int ncand;
  int i;

  if(is_a_solution(a, k, input))
    process_solution(a, k, input);
  else {
    k++;
    construct_candidates(a, k, input, c,
&ncand);
    for(i=0; i<ncand; i++){
      a[k]=c[i];
      backtrack(a, k, input);
      if (finished) return;
    }
  }
}
```

Q : All subsets

Print out all subsets of {1, 2, 3, 4}

```
#include <stdio.h>
#define MAXCAND 20
int finished = 0;

int is_a_solution(int a[], int k, int n){
  return (k==n);
}

void process_solution(int a[], int k, int n) {
  int i;
  printf("{");
  for(i=1; i<=k; i++)
    if(a[i] == 1) printf(" %d", i);
  printf(" }\n");
}

void construct_candidates(int a[], int k, int n, int
c[], int *nc) {
  c[0] = 1;
  c[1] = 0;
  *nc = 2;
}


int main() {
  int a[100];
  backtrack(a, 0, 4);
  return 0;
}
```

```
void backtrack(int a[], int k, int input) {
  int c[MAXCAND];
  int ncand;
  int i;

  if(is_a_solution(a, k, input))
    process_solution(a, k, input);
  else {
    k++;
    construct_candidates(a, k, input, c, &ncand);
    for(i=0; i<ncand; i++){
      a[k]=c[i];
      backtrack(a, k, input);
      if (finished) return;
    }
  }
}
```

```
{1234}
{123}
{124}
{12}
{134}
{13}
{14}
{1}
{234}
{23}
{24}
{2}
{34}
{3}
{4}
{}
```

Again, N-Queens

```
int is_a_solution(int a[], int k, int n){
  return (k==n);
}

void process_solution(int a[], int k, int n) {
  sol++;
  printf("%d : ", sol);
  for (int i = 1; i <= k; i++)
    printf("%d", a[i]);
  printf("\n");
}

void construct_candidates(int a[], int k, int n, int c[],
int *nc) {
  *nc = 0;
  int issafe = 1; // true
  for(int i = 1; i <= n; i++) {  // a[k] = i
    issafe = 1;
    for(int j = 1; j < k; j++) {
      if ((a[j] == i) || (abs(j-k) == abs(a[j]-i))) {
        issafe = 0; // false
        break;
      }
    }
    if(issafe) {
      c[*nc] = i;
      *nc = *nc+1;
    }
  }
}

int main() {
  int a[100] = {0};
  backtrack(a, 0, 8);
  return 0;
}
```

```
void backtrack(int a[], int k, int input) {
  int c[MAXCAND];
  int ncand;
  int i;

  if(is_a_solution(a, k, input))
    process_solution(a, k, input);
  else {
    k++;
    construct_candidates(a, k, input, c,
&ncand);
    for(i=0; i<ncand; i++){
      a[k]=c[i];
      backtrack(a, k, input);
      if (finished) return;
    }
  }
}
```

Q : subset sum problem

Given a set of positive integers, and a value *sum S*,
find out if there exist a subset in array whose sum is equal to given *sum S.*

Example:
int[] A = { 3, 2, 7, 1}, S = 6

Output: True, subset is (3, 2, 1}

A tug of war is being arranged for the office picnic. The picnickers must be fairly divided into two teams. Every person must be on one team or the other, the number of people on the two teams must not differ by more than one, and the total weight of the people on each team should be as nearly equal as possible.

Sample Input

```
1

3
100
90
200
```

Sample Output

```
190 200
```

Discussion : Tug of War

Given a set of *n* integers, divide the set in two subsets of *n/2* sizes each such that the difference of the sum of two subsets is as minimum as possible. If n is even, then sizes of two subsets must be strictly n/2 and if n is odd, then size of one subset must be (n-1)/2 and size of other subset must be (n+1)/2.

For example, let given set be {3, 4, 5, -3, 100, 1, 89, 54, 23, 20}, the size of set is 10.
Output for this set should be {4, 100, 1, 23, 20} and {3, 5, -3, 89, 54}. Both output subsets are of size 5 and sum of elements in both subsets is same (148 and 148).

Let us consider another example where n is odd.
Let given set be {23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4}.
The output subsets should be {45, -34, 12, 98, -1} and {23, 0, -99, 4, 189, 4}.
The sums of elements in two subsets are 120 and 121 respectively.

https://www.geeksforgeeks.org/tug-of-war/

Discussion : 15-Puzzle

Given puzzle

Target puzzle

The only legal operation is to exchange the missing tile with one of its neighbors (**L**eft, **R**ight, **U**p, **D**own)

Input : Given puzzle

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

Output: the move sequence to solve the puzzle

```
LLLDRDRDR
```