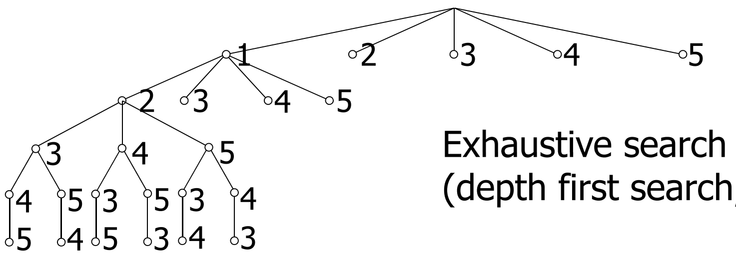
Permutation, Combination, and Related Problems - backtracking algorithms

Pei-yih Ting

Introduction

- You might need to generate permutations, combinations, partitions to enumerate all possible configurations in order to find the optimal solutions or brute-force solutions of some problems
- Procedural programming is about data processing
- To design the program:
  - figure out how the data/configuration changes
  - find the suitable representation of data in a program
  - figure out the “process” that transforms the data step by step

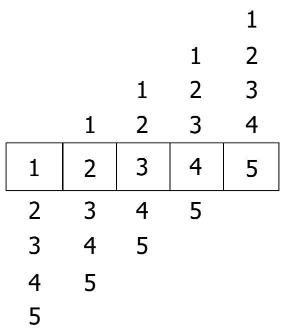
Generating Permutations



Exhaustive search (depth first search, DFS)

5! = 120 permutations

1 2 3 4 5
1 2 3 5 4
1 2 4 3 5
1 2 4 5 3
1 2 5 3 4
1 2 5 4 3
...



n-layer for loop Implementation

```
int i1, i2, i3, i4;
for (i1=1; i1<=4; i1++) {
   for (i2=1; i2<=4; i2++) {
      if (i2 == i1) continue;
      for (i3=1; i3<=4; i3++) {
         if ((i3==i2)||(i3==i1)) continue;
         for (i4=1; i4<=4; i4++) {
            if ((i4==i3)||(i4==i2)||(i4==i1)) continue;
            printf("%d %d %d %d\n", i1, i2, i3, i4);
         }
      }
   }
}
```

i1	i2	i3	i4

This is a quick implementation but **NOT scalable**.

2-layer for loop Implementation

- Consider this simplified loop without collision constraints

```
int data[4];
for (data[0]=1; data[0]<=4; data[0]++) {
   for (data[1]=1; data[1]<=4; data[1]++) {
      for (data[2]=1; data[2]<=4; data[2]++) {
         for (data[3]=1; data[3]<=4; data[3]++) {
            printf("%d %d %d %d\n", data[0],
                      data[1], data[2], data[3]);
         }
      }
   }
}
```

data

i1	i2	i3	i4
1	1	1	1
1	1	1	2
1	1	1	3
1	1	1	4
1	1	2	1
1	1	2	2
1	1	2	3
1	1	2	4
1	1	3	1

...

- It is still not scalable.

- Is there a ***scalable*** program structure that generates the same (i1, i2, i3, i4) counting-up sequence? yes

Equivalent 2-layer for loop

```
void nextIndex(int index[], int n) {
   int i;
   for (i=n-1; i>0; i--)
      if (index[i]<n)
         { index[i]++; return; }
      else
         index[i] = 1;
   index[0]++;
}
```

index

1	1	1	1
1	1	1	2
1	1	1	3
1	1	1	4
1	1	2	1
1	1	2	2
1	1	2	3
1	1	2	4
1	1	3	1

...

Scalable

```
int i, index[4], n=4;
for (i=0; i<n; i++)
   index[i] = 1;
for (; index[0]<=n; nextIndex(index, n))
   printArray(index, n);
```


2-layer for loop for Permutation

```
int index[4], n=4;
for (i=0; i<n; i++)
   index[i] = i+1;
for (; index[0]<=n; nextIndex(index, n))
   if (isValid(index, n))
      printPerm(index, n);
```

index

1	2	3	4
1	2	4	3
1	3	2	4
1	3	4	2
1	4	2	3
1	4	3	2
2	1	3	4
2	1	4	3

...

```
int isValid(int index[], int n) {
   int i, j;
   for (i=0; i<n-1; i++)
      for (j=i+1; j<n; j++)
         if (index[i]==index[j])
            return 0;
   return 1;
}
```

```
void nextIndex(int index[], int n) {
   int i;
   for (i=n-1; i>0; i--)
      if (index[i]<n)
         ...
   index[0]++;
}
```


Counting w/o invalid() check

```
int next(int perm[], int used[], int n) {
   int i, j, k;
   for (i=n-1; i>=0; i--) {
      used[perm[i]] = 0;
      do
         perm[i]++;
      while ((perm[i]<=n)&&used[perm[i]]);
      if (perm[i]<=n) {
         used[perm[i]] = 1;
         for (k=0,j=i+1; j<n; j++) {
            while (used[++k]);
            perm[j] = k, used[k] = 1;
         }
         return 1;
      }
   }
   return 0;
}
```

perm

1	2	3
1	**3**	**2**
2	**1**	**3**
2	**3**	**1**
3	**1**	**2**
3	**2**	**1**

```
int perm[10], used[11], n=3;
for (i=0; i<n; i++)
   perm[i] = i+1, used[i+1] = 1;
do
   printArray(perm, n);
while (next(perm, used, n));
```

Another Implementation

num[-1]	num[0]			num[n-1]
0	0	1	2	3
	0	1	3	2
	0	2	1	3
	0	2	3	1
	0	3	1	2
	0	3	2	1
	1	0	2	3
		...		

```
void print(int m, int num[]);
void next(int n, int num[], int it);
int main() {
   int i, n=4, num0[] = {0, 0, 1, 2, 3}, *num=num0+1;
   while (num[-1]==0) {
      print(n, num);
      i=n-1;
      do
         next(n, num, --i);
      while (i>=0&&num[i]==n);
      while (++i<n)
         num[i]=-1, next(n, num, i);
   }
   return 0;
}
```

```
void print(int n, int num[]) {
   for (int i=0; i<n; i++)
      printf("%d ", num[i]);
   printf("\n");
}
```

```
void next(int n, int num[], int it) {
   int i;
   for (num[it]++; num[it]<n; num[it]++) {
      for (i=0; i<it; i++)
         if (num[i]==num[it]) break;
      if (i==it) return;
   }
}
```

找到比 **num[it]** 大的下一個可用數字, 找不到 **num[it]** 就設為 **n**

Recursive Counting Up

data —pivot→

0	1	3	2	...

data[0] ⋯ data[m-1]

```
int main() {
   int data[10];
   countUp(4, data, 4, 0);
   return 0;
}

void countUp(int n, int data[], int m, int pivot) {
   if (pivot>=m)
      print(data, m);
   else
      for (data[pivot]=0; data[pivot]<n; data[pivot]++)
         countUp(n, data, m, pivot+1);
}
```

base

digits

the running digit

data[0] 0 1 2 3

data[pivot] 0 1 2 3

0 1 2 3 0

data[m-1] 0 1 2 3

Exhaustive search (depth first search, DFS)

0 0 0 0
0 0 0 1
0 0 0 2
0 0 0 3
0 0 1 0
...
0 0 3 3
0 1 0 0
0 1 0 1
0 1 0 2
0 1 0 3
0 1 1 0
...
0 1 3 2
0 1 3 3
0 2 0 0
...
1 0 0 0
...
3 3 3 2
3 3 3 3

Generalization of Deep *for* Loops

```
int data[4];
for (data[0]=1; data[0]<=4; data[0]++) {
   for (data[1]=1; data[1]<=4; data[1]++) {
      for (data[2]=1; data[2]<=4; data[3]++) {
         for (data[3]=1; data[3]<=4; data[3]++) {
            printf("%d %d %d %d\n", data[0],
               data[1], data[2], data[3]);
         }
      }
   }
}
```

① data[0] ② 0 ⑥ 1 ⑩ 2

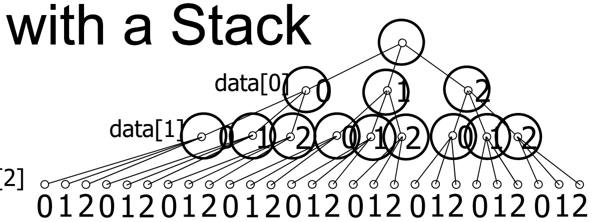
data[1] ③ ④ ⑤ 0 1 2 ⑦ ⑧ ⑨ 0 1 2 ⑪ ⑫ ⑬ 0 1 2

data[2] 012 012 012 012 012 012 012 012 012

Recursive DFS implementation

```
void countUp(int n, int data[], int m, int pivot) {
   if (pivot>=m)
      print(data, m);
   else
      for (data[pivot]=0; data[pivot]<n; data[pivot]++)
         countUp(n, data, m, pivot+1);
}
```


DFS with a Stack




```
#include <stdio.h>
int main() {
   int n=3, i, data[10], p;
   int top=1,s[10*10][2]={{-1,0}};
   while (top>0) {
      p=s[--top][0];
      if (p>=0) data[p]=s[top][1];
      if (p==n-1)
         for (i=0; i<n; i++)
            printf("%d%c", data[i], i<n-1?' ':'\n');
      else
         for (p++,i=n-1; i>=0; i--)
            s[top][0]=p, s[top][1]=i, top++;
   }
   return 0;
}
```

pop

push

p

-1 0 1 2

data

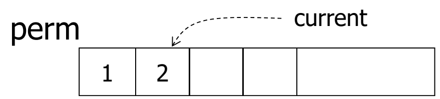
p, v

data[p]=v;

2, 0
2, 0
2, 0
2, 0
0, 0
0, 2
-0, 0

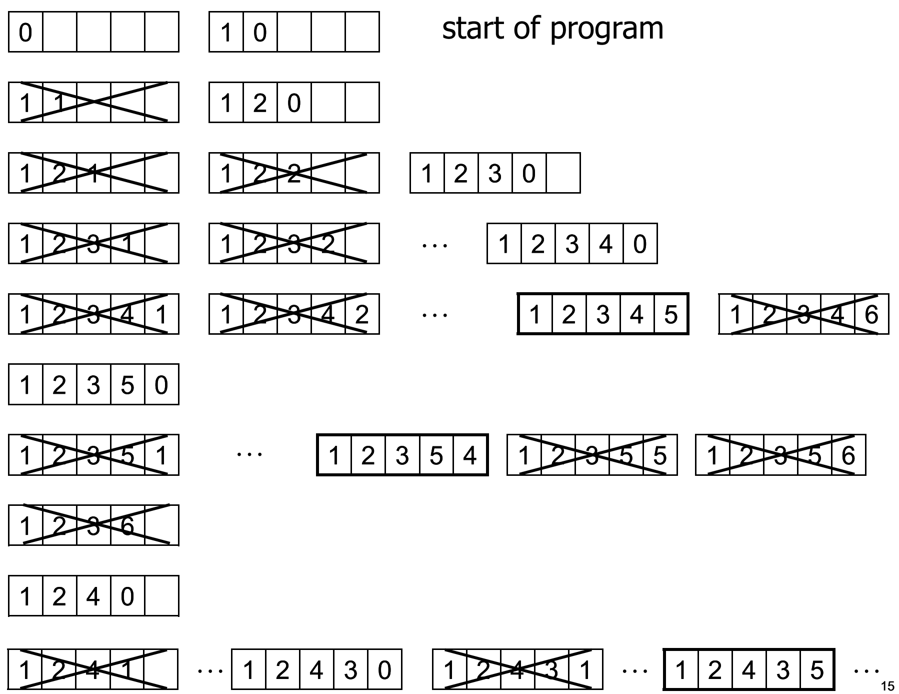
Permutation

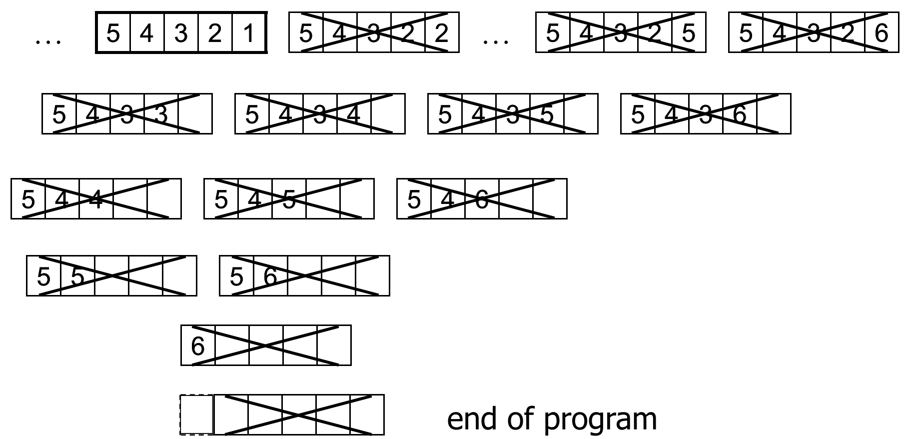
- If you view the index[] as an n-digit n-ary number (the digits set is {1, 2, …, n}), the previous program generates all possible increasing numbers (from 111…1 to nnn…n) and eliminates all invalid numbers with repeating digits on the way of counting up.
- Problem: slow as n gets to 10 or larger. n^n vs. n!
- Next, you will see another implementation. It tries to generate only valid increasing numbers with the next() function. Later, we will have some more algorithms to generate permutations.




```
#include <stdio.h>

void main()
{
    int size, perm[12] = {0}, current=0, solCount=0, i;

    printf("Please input number of elements: ");
    scanf("%d", &size);

    while (current>=0)
    {
        current += next(size, current, perm);
        if (current == size)
        {
            solCount++;
            printf("%4d: ", solCount);
            for (i=0; i<size; i++)
                printf("%d ", perm[i]);
            printf("\n");
            current = size-1;
        }
    }
    printf("Total %d permutations\n",solCount);
}
```

```
int next(int size, int pivot, int perm[]
{
    int i, collision;

    while (perm[pivot]++ < size)
    {
        collision = 0;
        for (i=0; i<pivot; i++)
            if (perm[pivot] == perm[i])
            {
                collision = 1;
                break;
            }
        if (!collision) return 1;
    }
    perm[pivot] = 0;
    return -1;
}
```



Permutation from Swapping

- **Recursive**
  - 1 + permutations of {2, 3, 4}
  - 2 + permutations of {1, 3, 4}
  - 3 + permutations of {2, 1, 4}
  - 4 + permutations of {2, 3, 1}

```
1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 3 2
1 4 2 3
2 1 3 4
2 1 4 3
  ...
3 2 1 4
3 2 4 1
  ...
4 2 3 1
4 2 1 3
  ...
4 1 2 3
```

```
for (i=0; i<n; i++) a[i] = i+1;
permutation(a, 0, n-1);
```

```
void permutation(int perm[], int start, int end) {
    if (start == end) { printPerm(perm, end+1); return; }
    for (int i=start; i<=end; i++) {
        swap(&perm[start], &perm[i]);
        permutation(perm, start+1, end);
        swap(&perm[start], &perm[i]);
    }
}
```


from Swapping (cont'd)

ZJ e446

- The output of previous program is **not** in **lexicographic order**

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

```
13: 3 2 1 4       13: 3 1 2 4
14: 3 2 4 1       14: 3 1 4 2
15: 3 1 2 4       15: 3 2 1 4
16: 3 1 4 2       16: 3 2 4 1
17: 3 4 1 2       17: 3 4 1 2
18: 3 4 2 1       18: 3 4 2 1
19: 4 2 3 1       19: 4 1 2 3
20: 4 2 1 3       20: 4 1 3 2
21: 4 3 2 1       21: 4 2 1 3
22: 4 3 1 2       22: 4 2 3 1
23: 4 1 3 2       23: 4 3 1 2
24: 4 1 2 3       24: 4 3 2 1
```

start i end

perm[] | 3 | 2 | 1 | 4 |

```
for (int i=start; i<=end; i++) {
    swap(&perm[start], &perm[i]);
    pRotate(&perm[start], &perm[i]);
    permutation(perm, start+1, end);
    swap(&perm[start], &perm[i]);
    pRotate(&perm[i], &perm[start]);
}
```

如果排列的數字允許重複?

ITSA33 problem #2

如果按順序排就可以了

```
void pRotate(int *a, int *b) {
    int tmp=*b, d=b>a?-1:1;
    while (a!=b) *b = *(b+d), b+=d;
    *a = tmp;
}
```


Permutation from Rotation

```
0 1 2 3   2 0 1 3   1 2 0 3   1 0 2 3   2 1 0 3   0 2 1 3
3 0 1 2   3 2 0 1   3 1 2 0   3 1 0 2   3 2 1 0   3 0 2 1
2 3 0 1   1 3 2 0   0 3 1 2   2 3 1 0   0 3 2 1   1 3 0 2
1 2 3 0   0 1 3 2   2 0 3 1   0 2 3 1   1 0 3 2   2 1 3 0
0 1 2 3   2 0 1 3   1 2 0 3   1 0 2 3   2 1 0 3   0 2 1 3
                    0 1 2 3                       1 0 2 3
                                                  0 1 2 3
```

```
int rotate_n_check(int size, char data[]) {
    int i;
    for (i=size; i>=2; i--) {
        rotate(i, data);
        if (data[i-1] != i-1) return 1;
    }
    return 0;
}
```

```
for (i=0; i<num; i++) digit[i] = i;
do
    printPermutation(num, digit);
while (rotate_n_check(num, digit));
```


Rotation (cont')

- Why does it work?

1-digit: 0

2-digit: 0 1, 1 0

3-digit: 0 1 2, 2 0 1, 1 2 0, 1 0 2, 2 1 0, 0 2 1

4-digit:

```
0 1 2 3
3 0 1 2
2 3 0 1
1 2 3 0
2 0 1 3
3 2 0 1
...
1 2 0 3
...
1 0 2 3
...
2 1 0 3
...
0 2 1 3
...
```

Recursive Implementation

```
void permuteFromRotation(char pdata[], int len) {
   char cdata[20];
   int i, j, tmp;
   if (len>=n) {
      pdata[n]=0; printf("%s\n", pdata);
      return;
   }
   for (i=0; i<len; i++) cdata[i] = pdata[i];
   cdata[len] = items[len];
   for (i=0; i<=len; i++) {
      permuteFromRotation(cdata, len+1);
      for (tmp=cdata[len],j=len; j>0; j--)
         cdata[j] = cdata[j-1];
      cdata[0] = tmp;
   }
}
```

```
result[0] = items[0];
permuteFromRotation(result, 1);
```

len 2

pdata[] '0' '1'

cdata[] '0' '1' '2'
'2' '0' '1' '3'
'3' '2' '0' '2'
'2' '3' '0' '1'
'1' '2' '3' '0'

items[len]

items[] '0' '1' '2' '3'

n 4

Recursive Implementation

```
char items[]="0123"; int n=4;
void permuteFromRotation(char pdata[], int len) {
   int i, j, tmp;
   if (len>=n) {
      pdata[n]=0; printf("%s\n", pdata);
      return;
   }
   pdata[len] = items[len];
   for (i=0; i<=len; i++) {
      permuteFromRotation(pdata, len+1);
      for (tmp=pdata[len],j=len; j>0; j--)
         pdata[j] = pdata[j-1];
      pdata[0] = tmp;
   }
}
```

```
result[0] = items[0];
permuteFromRotation(result, 1);
```

len 2

pdata[] '0' '1' '2'
'2' '0' '1' '3'
'3' '2' '0' '2'
'2' '3' '0' '1'
'1' '2' '3' '0'

items[len]

items[] '0' '1' '2' '3'

n 4

Permutation by Exchanging Neighbors

perm	pos	perm	pos
3 2 1 0	0 0 0	3 2 0 1	0 0 1
2 3 1 0	1 0 0	2 3 0 1	1 0 1
2 1 3 0	2 0 0	2 0 3 1	2 0 1
2 1 0 3	3 0 0	2 0 1 3	3 0 1
3 1 2 0	0 1 0	3 0 2 1	0 1 1
1 3 2 0	1 1 0	0 3 2 1	1 1 1
1 2 3 0	2 1 0	0 2 3 1	2 1 1
1 2 0 3	3 1 0	0 2 1 3	3 1 1
3 1 0 2	0 2 0	3 0 1 2	0 2 1
1 3 0 2	1 2 0	0 3 1 2	1 2 1
1 0 3 2	2 2 0	0 1 3 2	2 2 1
1 0 2 3	3 2 0	0 1 2 3	3 2 1

```
for (i=0;i<num;i++)
  perm[i]=num-1-i, position[i]=0;
do
  printPermutation(num, perm);
while (shift_n_check(num, perm, position))
```

```
int shift_n_check(int size, char perm[],
                          char pos[]) {
  for (int i=size; i>=2; i--) {
    shiftRight(i, &perm[size-i], &pos[size-i]);
    if (pos[size-i] > 0) return 1;
  }
  return 0;
}
```

```
void shiftRight(int size, char perm[],
                       char *pos){
  char tmp = perm[*pos];
  if (*pos < size-1) {
    perm[*pos] = perm[*pos+1];
    perm[++(*pos)] = tmp;
  }
  else {
    for (int i=size-2; i>=0; i--)
      perm[i+1] = perm[i];
    perm[*pos=0] = tmp;
  }
}
```


Exchanging Neighbors (cont'd)

- Why does it work?

1-digit 0

2-digit
0 1
1 0

3-digit
0 1 2
0 2 1
2 0 1
1 0 2
1 2 0
2 1 0

4-digit
0 1 2 3
0 1 3 2
0 3 1 2
3 0 1 2
0 2 1 3
0 2 3 1
...
2 0 1 3
...
1 0 2 3
...
1 2 0 3
...
2 1 0 3
...

Recursive Implementation

```
void permuteFromExchanging(char pdata[], int len) {
   char cdata[10];
   int i, j, tmp;
   if (len>=n) {
      pdata[n]=0; printf("%s\n", pdata);
      return;
   }
   for (i=0; i<len; i++) cdata[i] = pdata[i];
   cdata[len] = items[len];
   for (i=len; i>=0; i--) {
      permuteFromExchanging(cdata, len+1);
      if (i>0)
         tmp=cdata[i], cdata[i]=cdata[i-1], cdata[i-1] = tmp;
   }
}
      result[0] = items[0];
      permuteFromExchanging(result, 1);
```

len 2

pdata[] '0' '1'

cdata[] '0' '1' '2'

'0' '2' '1' '3'

'2' '0' '1' '2'

'0' '3' '1' '2'

'3' '0' '1' '2'

'0' '0' '1' '3'

'0' '0' '3' '1'

'0' '3' '0' '1'

'3' '0' '0' '1'

items[len]

items[] '0' '1' '2' '3'

n 4

Recursive Impl. w/o Extra Array

```
void permuteFromExchanging(char data[], int len) {
   int i, j, tmp;
   if (len>=n) {
      data[n]=0; printf("%s\n", data);
      return;
   }
   data[len] = items[len];
   for (i=len; i>=0; i--) {
      permuteFromExchanging(data, len+1);
      if (i>0)
         tmp=data[i], data[i]=data[i-1], data[i-1] = tmp;
   }
   for (i=0; i<len; i++) data[i] = data[i+1];
}
      result[0] = items[0];
      permuteFromExchanging(result, 1);
```

len 2

data[] '0' '1' '2'

'0' '2' '1' '3'

'2' '0' '3' '2'

'0' '3' '1' '2'

'3' '0' '1' '2'

'0' '0' '1' '3'

'0' '0' '3' '1'

'0' '3' '0' '1'

'3' '0' '0' '1'

items[len]

items[] '0' '1' '2' '3'

n 4

Generate Disjoint Partitions

- A partition of a set X is a set of non-empty subsets of X such that every element x in X is in exactly one of these subsets.
  - e.g. The set X={1, 2, 3} has five partitions:
    - { {1}, {2}, {3} }, sometimes written 1 | 2 | 3.
    - { {1, 2}, {3} }, or 12 | 3.
    - { {1, 3}, {2} }, or 13 | 2.
    - { {1}, {2, 3} }, or 1 | 23.
    - { {1, 2, 3} }, or 123
  - The total number of partitions is the Bell numbers B_n

$$B_{n+1} = \sum_{k=0}^{n} C_k^n B_k$$

e.g. $B_0 = 1$, $B_1 = 1$, $B_2 = 2$, $B_3 = 5$, $B_4 = 15$, $B_5 = 52$, $B_6 = 203$, …

$B_{20} = 51724158235372 \approx 5x10^{13}$, $B_{30} = 846749014511809332450147 \approx 8x10^{23}$,

$B_{40} = 157450588391204931289324344702531067 \approx 2x10^{35}$,

$B_{50} = 185724268771078270438257767181908917499221852770 \approx 2x10^{47}$.

Generate Partitions (cont'd)

- A simple algorithm through counting up with dynamically adjusted ceiling,
  partition / max e.g., X is a 4-element set $\{a_0,a_1,a_2,a_3\}$

1: **0,0,0,0** / 0,**1,1,1** ==> $\{a_0,a_1,a_2,a_3\}$
2: **0,0,0,1** / 0,1,1,1 ==> $\{a_0,a_1,a_2\},\{a_3\}$
3: **0,0,1,0** / 0,1,1,**2** ==> $\{a_0,a_1,a_3\},\{a_2\}$
4: **0,0,1,1** / 0,1,1,2 ==> $\{a_0,a_1\},\{a_2,a_3\}$
5: **0,0,1,2** / 0,1,1,2 ==> $\{a_0,a_1\},\{a_2\},\{a_3\}$
6: **0,1,0,0** / 0,1,**2,2** ==> $\{a_0,a_2,a_3\},\{a_1\}$
7: **0,1,0,1** / 0,1,2,2 ==> $\{a_0,a_2\},\{a_1,a_3\}$
8: **0,1,0,2** / 0,1,2,2 ==> $\{a_0,a_2\},\{a_1\},\{a_3\}$
9: **0,1,1,0** / 0,1,2,2 ==> $\{a_0,a_3\},\{a_1,a_2\}$
10: **0,1,1,1** / 0,1,2,2 ==> $\{a_0\},\{a_1,a_2,a_3\}$
11: **0,1,1,2** / 0,1,2,2 ==> $\{a_0\},\{a_1,a_2\},\{a_3\}$
12: **0,1,2,0** / 0,1,2,**3** ==> $\{a_0,a_3\},\{a_1\},\{a_2\}$
13: **0,1,2,1** / 0,1,2,3 ==> $\{a_0\},\{a_1,a_3\},\{a_2\}$
14: **0,1,2,2** / 0,1,2,3 ==> $\{a_0\},\{a_1\},\{a_2,a_3\}$
15: **0,1,2,3** / 0,1,2,3 ==> $\{a_0\},\{a_1\},\{a_2\},\{a_3\}$

Implementation with 2 arrays

0,0,1,2 label the partitions for each set elements a_i

	0	1	2	3
partition	0	0	1	2
max	0	1	1	2

each max[i] ≥ all prior partitions

when a non-tailing digit reaches its max

	0	1	2	3
partition	0	1	**2**	**0**
max	0	1	2	**3**

Implementation

```
int next(int size, int pivot, int partition[], int max[]) {
   int i, j, maxVal;
   while (pivot>0 && (partition[pivot] >= max[pivot]))
      pivot--;
   if (pivot > 0) {
      partition[pivot]++;
      for (i=pivot+1; i<size; i++) {
         partition[i] = 0;
         maxVal = 0;
         for (j=1; j<i; j++)
            if (partition[j]>maxVal)
               maxVal = partition[j];
         max[i] = maxVal+1;
      }
      return size;
   }
   return pivot;
}
```

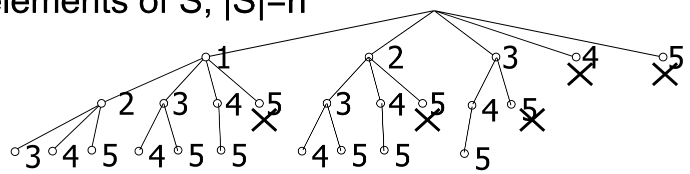
pivot

	0	1	2	3
partition	**0**	**1**	**1**	**2**
max	**0**	**1**	**2**	**2**

	0	1	2	3
partition	**0**	**1**	**2**	**0**
max	**0**	**1**	**2**	**3**

Generate (n,k)-Combinations

- (n,k)-combination of a set S is a subset of k distinct elements of S, |S|=n



C^5_3 = 10
(5,3)-combinations

1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5

		1
	1	2
1	2	3
2	3	4
3	4	5
4	5	
5		

- Extend the counting-up program and avoid those non-increasing sequences

Efficient Enumerating w/o Invalid()

```
int next(int comb[], int n, int k) {
   int i, j;
   for (i=k-1; i>=0; i--)
      if (comb[i]<n-(k-1-i)) {
         comb[i]++;
         for (j=i+1; j<k; j++)
            comb[j] = comb[j-1]+1;
         return 1;
      }
   return 0;
}
```

3 4 5

comb

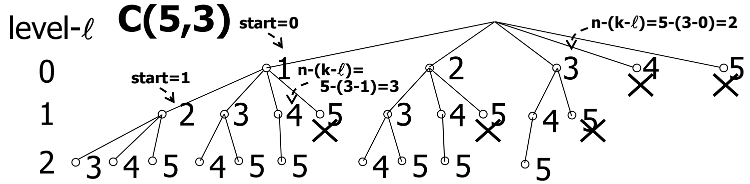
1	2	3
1	2	**4**
1	2	**5**
1	**3**	**4**
1	3	**5**
1	**4**	**5**
2	**3**	**4**
2	3	**5**
2	**4**	**5**
3	**4**	**5**

```
int comb[10], n=5, k=3;
for (i=0; i<k; i++)
   comb[i] = i+1;
do
   printArray(comb, k);
while (next(comb, n, k));
```

Output symbols

'Z'	'A'	'K'	'D'	'S'

Recursive (n,k)-Combinations



- select level-ℓ elements from seq[start]~seq[n-(k-ℓ)]
- put the result to result[ℓ]~result[k-1]
- **comb(n,k,seq,0,0,result); // DFS backtracking**

```
void comb(int n, int k, const int seq[], int start, int ℓ, int result[]) {
   if (ℓ==k) print(result, k); // result[0]~result[k-1]
   else
      for (int i=start; i<=n-(k-ℓ); i++)
         result[ℓ] = seq[i],
         comb(n, k, seq, i+1, ℓ+1, result);
}
```

e.g. bruteforce **UVa10326**
UVa11659 w/ binary search

1 3
$\underbrace{\ell\sim k-1}$
level
result

1	2	3

1 5 3 1
$\underbrace{\text{start}\sim n-(k-\ell)}$
seq

1	2	3	4	5

Generate Power Set

- The power set of a set is the collection of all subsets of S, including the empty set and S itself.
  - e.g. S = {1, 2, 3},
    2^S = {{}, {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}}
- We can extend the program for generating (n,k)-combinations to generate the whole power set.
```
int n=3, seq[]={1,2,3}, result[3], k;
for (k=0; k<=n; k++)
   comb(n, k, seq, 0, 0, result);
```


正整數 N 的加法拆解 (整數劃分)

- 每個正整數 N 都可以寫成比它小的數字和, 請把所有不重複整數加總等於 N 的方法寫出來

```
Sample Input
6

Sample Output
1 2 3
1 5
2 4
6
```

```
Sample Input
10

Sample Output
1 2 3 4
1 2 7
1 3 6
1 4 5
2 3 5
3 7
4 6
10
```

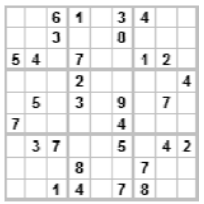
```
Sample Input
15

Sample Output
1 2 3 4 5
1 2 3 9
1 2 4 8
1 2 5 7
1 2 12
1 3 4 7
1 3 5 6
1 3 11
1 4 10
1 5 9
1 6 8
1 14
```

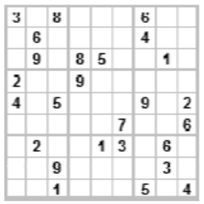
```
2 3 4 6
2 3 10
2 4 9
2 5 8
2 6 7
2 13
3 4 8
3 5 7
3 12
4 5 6
4 11
5 10
6 9
7 8
15
```

- 輸出請依照範例以一般化字典序 (lexicographic order) 排列

Sudoku

- Sudoku: In these three examples, 81 cells are divided into 9 blocks each with 9 cells (3-by-3). A player is required to fill in the blank cells such that integers in each row, each column, and each block are permutations of {1,2,3,…,9}, i.e. no duplication of numbers in each row, column, or block.





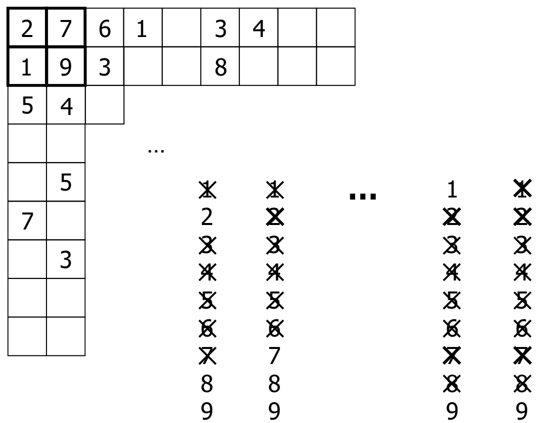
number of lines
row, column, value
row, column, value
…

```
30
0, 0, 7
0, 1, 8
0, 2, 9
0, 4, 2
0, 8, 5
1, 0, 6
1, 5, 5
1, 7, 8
…
```

Initial configuration

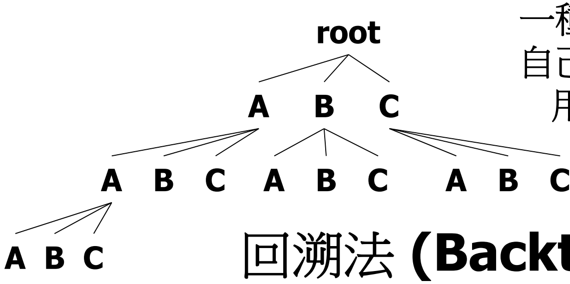
Sudoku (cont'd)

- Extension of counting



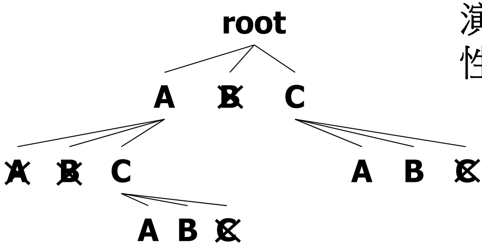
more constraints on the set of values to be filled in each cell

遞迴函式 (Recursive Function)



一種程式實作方式自己呼叫自己，常常用來窮舉所有的可能性, **DFS**

回溯法 (Backtracking)



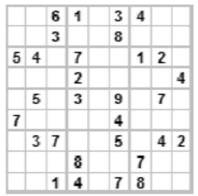
演算法**:** 窮舉所有的可能性**,** 但是一發現不行趕快回頭嘗試下一個可能的路徑**,** 常常用遞迴函式來實作**,** 也可以用迴圈實作

12-1 尋找整除數字 **12-2** 速讀

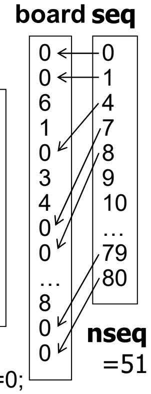
Backtracting with Iteration

- **constraints**: DFS → backtracking

```
void sudoku(int board[], int seq[], int nseq, int p) {
   while (p>=0)
      if (p>=nseq)  print_solution, p--;
      else if (board[seq[p]]>=9)
         board[seq[p--]] = 0;
      else {
         board[seq[p]]++;
         if (isValid(board, seq[p])) p++;
      }
}
```



```
30
0, 0, 7
0, 1, 8
0, 2, 9
0, 4, 2
0, 8, 5
1, 0, 6
1, 5, 5
…
```




```
int isValid(int data[], int p) {
   int i, j, r=p/9, c=p%9;
   for (i=0; i<9; i++) {
      if(i!=c && data[p]==data[r*9+i]) return 0;
      if(i!=r && data[p]==data[i*9+c]) return 0;
   }
   for (i=r/3*3; i<r/3*3+3; i++)
      for (j=c/3*3; j<c/3*3+3; j++)
         if ((i!=r||j!=c) && data[p]==data[i*9+j]) return 0;
   return 1;
}
```

```
int main() {
   int board[81]={};
   int seq[81], nseq=0;
   read constraints to board[81]
   prepare seq[81] and nseq
   sudoku(board, seq, nseq, 0);
   return 0;
}
```


Backtracking using Recursion

- without any constraint, raw **DFS** to generate 9^{81} possibilities

```
int main() {
   int data[81]; // 9x9
   sudoku(data, 0);
   return 0;
}
```

```
void print(int data[]) {
   int i, j;
   for (i=0; i<9; i++)
      for (j=0; j<9; j++)
         printf("%2d%s", data[i*9+j], j==9?"\n":"");
}
```

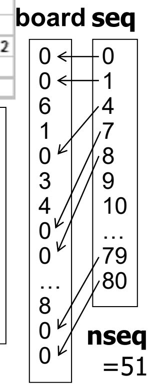
```
void sudoku(int data[], int p) {
   if (pivot>=81)
      print(data);
   else
      for (data[p]=1; data[p]<=9; data[p]++)
         sudoku(data, p+1);
}
```


Recursion (cont'd)

- **constraints**: DFS → backtracking

```
void sudoku(int board[], int seq[], int nseq, int p) {
   if (p>=nseq)
      print(board);
   else
      for (board[seq[p]]=1; board[seq[p]]<=9; board[seq[p]]++)
         if (isValid(board,seq[p]))
            sudoku(board, seq, nseq, p+1);
}
```

```
30
0, 0, 7
0, 1, 8
0, 2, 9
0, 4, 2
0, 8, 5
1, 0, 6
1, 5, 5
…
```




```
int isValid(int data[], int p) {
   int i, j, r=p/9, c=p%9;
   for (i=0; i<9; i++) {
      if(i!=c && data[p]==data[r*9+i]) return 0;
      if(i!=r && data[p]==data[i*9+c]) return 0;
   }
   for (i=r/3*3; i<r/3*3+3; i++)
      for (j=c/3*3; j<c/3*3+3; j++)
         if ((i!=r||j!=c) && data[p]==data[i*9+j])
            return 0;
   return 1;
}
```

```
int main() {
   int board[81]={};
   int seq[81], nseq=0;
   read constraints to board[81]
   prepare seq[81] and nseq
   sudoku(board, seq, nseq, 0);
   return 0;
}
```

5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Determinant of an n-by-n Matrix

- Leibniz formula

$$\mathbf{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$$

  - Determinant

$$\det(\mathbf{A}) = a_{1,1}a_{2,2}a_{3,3} + a_{1,2}a_{2,3}a_{3,1} + a_{1,3}a_{2,1}a_{3,2} - a_{1,3}a_{2,2}a_{3,1} - a_{1,2}a_{2,1}a_{3,3} - a_{1,1}a_{2,3}a_{3,2}$$

  - Permutation?

$$\det(\mathbf{A}) = a_{1,\mathbf{1}}a_{2,\mathbf{2}}a_{3,\mathbf{3}} + a_{1,\mathbf{2}}a_{2,\mathbf{3}}a_{3,\mathbf{1}} + a_{1,\mathbf{3}}a_{2,\mathbf{1}}a_{3,\mathbf{2}} - a_{1,\mathbf{3}}a_{2,\mathbf{2}}a_{3,\mathbf{1}} - a_{1,\mathbf{2}}a_{2,\mathbf{1}}a_{3,\mathbf{3}} - a_{1,\mathbf{1}}a_{2,\mathbf{3}}a_{3,\mathbf{2}}$$

  - $\text{sign}(\sigma) = \begin{cases} 1 & \text{if } \sigma \text{ has even number of misplaced pairs} \\ -1 & \text{if } \dots \quad \text{odd } \dots \end{cases}$

    e.g. sign(123)=1, sign(231)=1, sign(312)=1
    sign(321)=-1, sign(213)=-1, sign(132)=-1

Applications

1. 給定 n 個各種長度的木棒, 是否可排出正方形? 是否可排出指定長寬比例的矩形?
2. 有 m 條星狀連接的路徑, 在各路徑途中指定位置有不同金額的獎勵, 如果油料有限制, 最多只能夠走 n 公里, 請找到由中心點出發能夠取回最大獎勵的方式
3. n 個小偷竊得 m 個不同價值的物品, 怎樣才能比較公平地分贓而不至於內鬨呢?
4. n 組數字, 由每一組數字中各取一個, 總和最大的取法?
5. 尋找 nxn 魔方陣, 洛書
6. Tetromino & pentomino
7. 輸入整數N, 2≤N≤9, 印出一N位數字X, 其N個數字都不重複且mod 10^i 皆能被N整除, 例如N = 8，X=21579648、X mod 10^7=1579648、579648、79648、9648、648、48、8 皆可以被8整除且各位數字不重複出現。