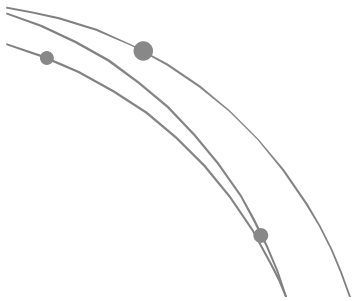
Enumeration and DFS

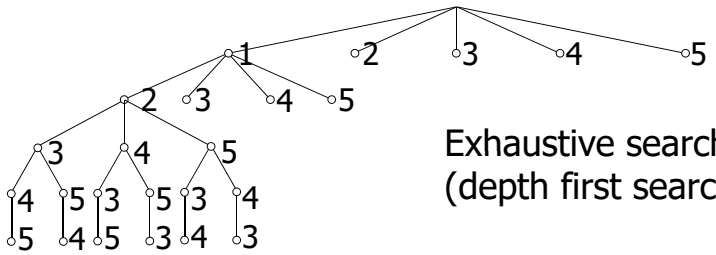


Pei-yih Ting

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

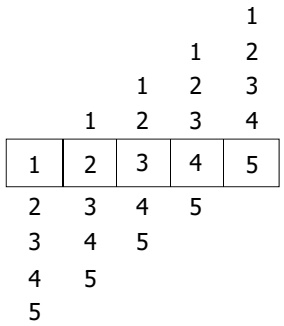
Enumerating Permutations



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**.

n-layer for loop Implementation

- Consider this **simplified** problem 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 next(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]++;
}

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

Scalable

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

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
...			

2-layer for loop for Permutation

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

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;
}
```

數字不會重複

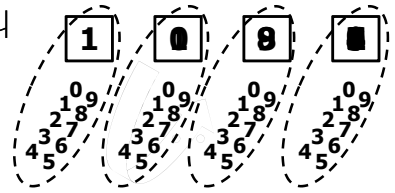
index

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
...			

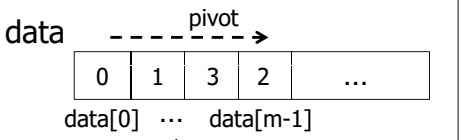
```
int index[4], n=4;
for (i=0; i<n; i++)
  index[i] = i+1;
for (; index[0]<=n; next(index,n))
  if (isValid(index, n))
    print(index, n);
```


Recursive

- 電影院找座位時, 詢問前一排的人他是第幾排
  - 如他是第 n 排, 這一排就是 n+1 排
  - 如果他不知道的話, 他自己會問前一排的人
    - 每個人只負責一點點事情
    - 想像每一次遞迴的函式呼叫是請不同的人依照同一個程序來做事情, 只是分配到的問題稍微變小一點而已
- 數 n 位數字的時候, 每次函式呼叫只負責數一位數字
  - 管理可以使用的數字, 由裡面第一個數字開始依序使用
  - 每次使用一個新的數字後, 找一個人數右邊那一位數 (遞迴函式呼叫)
  - 右邊的人數完 (return) 以後, 把數字加 1, 重複前一個動作



Recursive Counting Up




```
01 int main() {
02     int data[10];
03     countUp(4, data, 4, 0);
04     return 0;
05 }
06
07 void countUp(int b, int data[], int m, int pivot) {
08     if (pivot>=m)
09         print(data, m);
10     else
11         for (data[pivot]=0; data[pivot]<b; data[pivot]++)
12             countUp(b, data, m, pivot+1);
13 }
```

base

digits

data[0]

data[pivot]

data[m-1]

the running digit

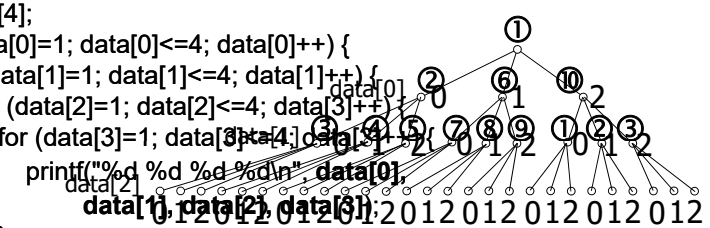
Exhaustive search (depth first search, DFS)

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

Ex. ZeroJudge b511 換銅板

Generalization of Deep *for* Loops

```
01 int data[4];
02 for (data[0]=1; data[0]<=4; data[0]++) {
03     for (data[1]=1; data[1]<=4; data[1]++) {
04         for (data[2]=1; data[2]<=4; data[3]++) {
05             for (data[3]=1; data[3]<=4; data[3]++) {
06                 printf("%d %d %d %d\n", data[0],
07                     data[1], data[2], data[3]);
08             }
09         }
10     }
11 }
```



Recursive DFS implementation

```
07 void countUp(int base, int data[], int m, int pivot) {
08     if (pivot>=m)
09         print(data, m);
10     else
11         for (data[pivot]=0; data[pivot]<base; data[pivot]++)
12             countUp(base, data, m, pivot+1);
13 }
```


Counting Up with Validity Check

```
01 void perm(int n, int data[], int m, int p) {
02     int i;
03     if (p>=m)
04         for (i=0; i<m; i++)
05             printf("%d%c", data[i], " \n"[i==m-1]);
06     else
07         for (data[p]=0; data[p]<n; data[p]++) {
08             for (i=0; i<p; i++)
09                 if (data[i]==data[p]) break;
10             if (i==p)
11                 perm(n, data, m, p+1);
12         }
13 }
```

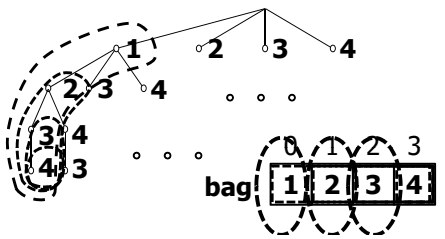
n=4, m=2			
0 1	1 0	2 0	3 0
0 2	1 2	2 1	3 1
0 3	1 3	2 3	3 2

n=4, m=4
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
1 0 3 2
1 2 0 3
1 2 3 0
. . .
3 2 1 0

```
01 int main() {
02     int data[10], n=4, m=4;
03     perm(n, data, m, 0);
04     return 0;
05 }
```


Efficient?

- Generate sequences with **constraints** – avoiding digits used previously **by comparison** ⇒ heavier burden for lower levels
- How about using **a bag of digits**: take one out and pass the bag with remaining digits on to lower levels



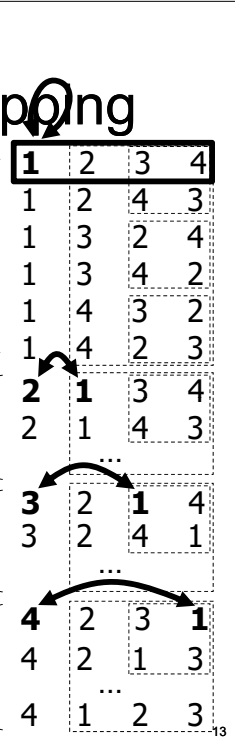

```
void main() {
    int bag[]={1,2,3,4};
    for (int i=0; i<4; i++) {
        swap(bag,0,i);
        for (int i=1; i<4; i++) {
            swap(bag,1,i);
            for (int i=2; i<4; i++) {
                swap(bag,2,i);
                print(bag,4);
                swap(bag,2,i);
            }
            swap(bag,1,i);
        }
        swap(bag,0,i);
    }
}
```

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}

```
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)

- 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
6: 1 4 2 3
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
12: 2 4 1 3
```

```
5: 1 4 2 3
6: 1 4 3 2
11: 2 4 1 3
12: 2 4 3 1
```

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

如果排列的數字允許重複?

ITSA33 problem #2

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

tmp: 3

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

start a b b end

perm[] 3 2 1 4

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

如果按順序排列可以了

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
```

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

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

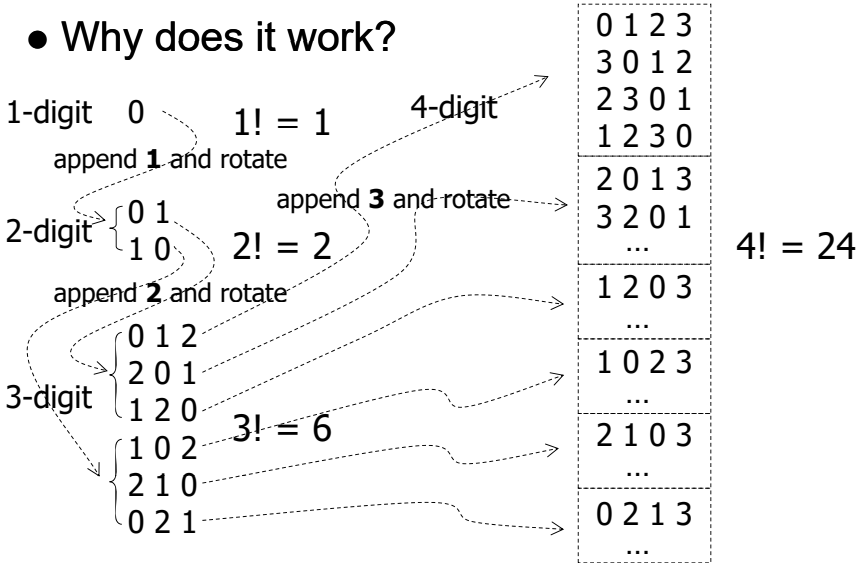
```
void rotate(int n, char x[]) {
    char tmp = x[n-1];
    for (int i=n-1; i>0; i--)
        x[i] = x[i-1];
    x[0] = tmp;
}
```

tmp: 3

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

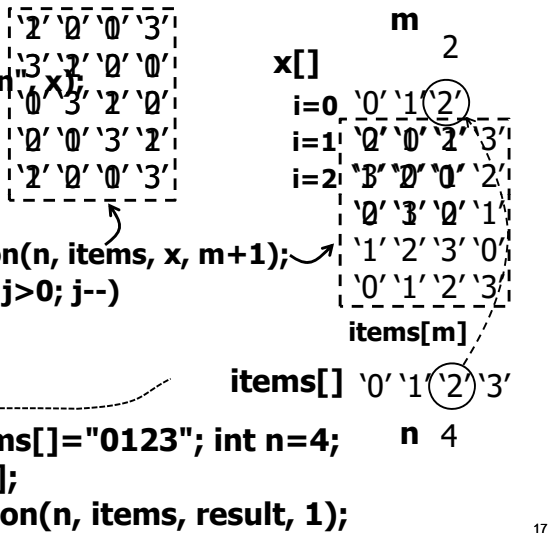
Permutation from Rotation (cont'd)

- Why does it work?



Recursive Implementation

```
void permuteFromRotation(int n, char items[], char x[], int m) {
   int i, j, tmp;
   if (m>=n) {
      x[n]=0; printf("%s\n", x);
      return;
   }
   x[m] = items[m];
   for (i=0; i<=m; i++) {
      permuteFromRotation(n, items, x, m+1);
      for (tmp=x[m],j=m; j>0; j--)
         x[j] =  x[j-1];
      x[0] = tmp;
   }
}
   char result[10], items[]="0123"; int n=4;
   result[0] = items[0];
   permuteFromRotation(n, items, result, 1);
```



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

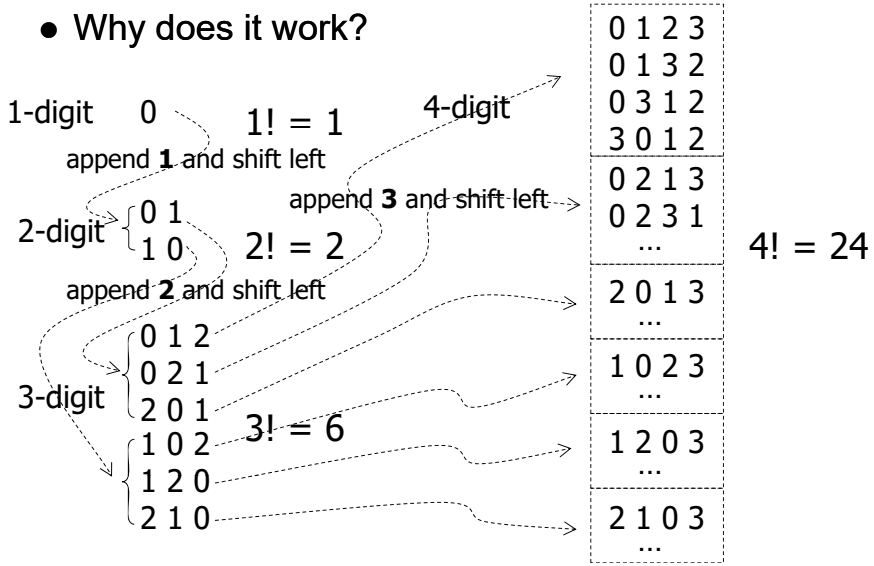
```
for (i=0;i<num;i++)
  perm[i]=num-1-i, pos[i]=0;
do
  print (num, perm);
while (shift_n_check(num, perm, pos));
```

```
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;
  }
}
```

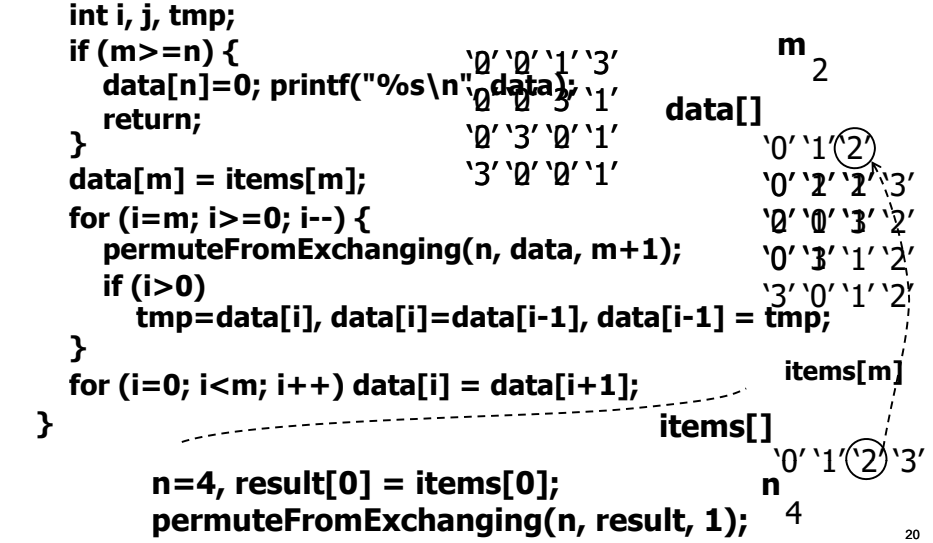

Exchanging Neighbors (cont'd)

- Why does it work?



Recursive Implementation

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



Generate Set Partitions

- A **parti** … **X** s.t. e …
  - e.g. …
    1. {…
    2. {…
    3. {…
    4. {…
    5. {…

		partitions			B_0=1
n=1	{1}	{1}			B_1=1
n=2	{1,2}	{1,2}	{1}{2}		B_2=2
n=3	{1,2,3}	{1,2} {1}{2} {3}	{2}{1,3} {1}{2,3}	{1,2,3}	B_3=?
		$B_{2\text{-}0}$	$B_{2\text{-}1}$	$B_{2\text{-}2}$	B_3=5

…ets of …sets.

…String …ngle

15
15 20 27 37 52
52

- The total number of partitions of a set of size **n** is the **Bell numbers** B_n

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

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** ≈ $5x10^{13}$, B_{30} = **846749014511809332450147** ≈ $8x10^{23}$,

B_{40} = **1574505883912049312893243447025310 67** ≈ $2x10^{35}$,

B_{50} = **185724268771078270438257767181908917499221852770** ≈ $2x10^{47}$.

Generate Set Partitions - Iterative

- A simple algorithm like counting up with dynamically adjusted ceiling,

RGS

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

increase all tailing max's when an RGS element reaches its max

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

Iterative Implementation

```
01 int next(int size, int pivot, int partition[], int max[]) {
02     int i, j, maxVal;
03     while (pivot>0 && (partition[pivot] >= max[pivot]))
04         pivot--;
05     if (pivot > 0) {
06         partition[pivot]++;
07         for (i=pivot+1; i<size; i++) {
08             partition[i] = 0;
09             maxVal = 0;
10             for (j=1; j<i; j++)
11                 if (partition[j]>maxVal)
12                     maxVal = partition[j];
13             max[i] = maxVal+1;
14         }
15         return size;
16     }
17     return pivot;
18 }
```

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

Recursive DFS Implementation

```
int main() {
    int n, rgs[10];
    scanf("%d", &n);
    rgs[0]=0, RGS(n, rgs, 1, 1);
    return 0;
}

void RGS(int n, int rgs[], int p, int m) {
    static int count=0;
    if (p>=n) {
        printf("%6d: ", count++);
        for (int i=0; i<n; i++)
            printf("%d%c", rgs[i], "\n "[i<n-1]);
    }
    else
        for (rgs[p]=0; rgs[p]<=m; rgs[p]++)
            RGS(n, rgs, p+1, m+(rgs[p]==m) );
}
```

p n=4

rgs[]	0	1	2	3
	0	1	2	

m	0	1	2

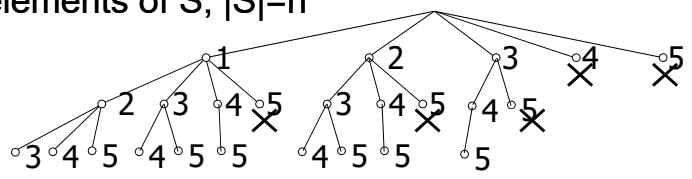
m_p = 1+max **rgs**[i] (i<**p**) = max(1+**rgs**[i]) (i<**p**)

m_{p+1} = max(1+**rgs**[i]) (i<**p+1**) = max(m_p, 1+**rgs**[**p**])

rgs[**p**]<=m_p

Generate (n,k)-Combinations

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



$C_3^5 = 10$
(5,3)-combinations

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

	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

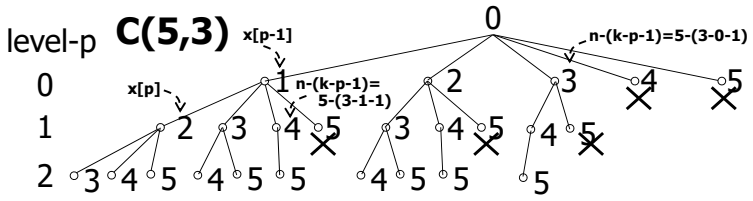
```
int comb[10], n=5, k=3;
for (i=0; i<k; i++)
   comb[i] = i+1;
do
   print(comb, k);
while (next(comb, n, k));
```

Output symbols

'Z'	'A'	'K'	'D'	'S'
1	2	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**

Recursive (n,k)-Combinations




```
void comb(int n, int k, int x[], int p) {
  if (p>=k)
    for (int i=0; i<k; i++)
      printf("%d%c", x[i],” \n”[i==k-1]);
  else
    for (x[p]=x[p-1]+1; x[p]<n-(k-p-2); x[p]++)
      comb(n, k, x, p+1);
}
```

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

```
int n, k, x[20]={0};
scanf("%d%d", &n, &k);
comb(n,k,x+1,0);
```


Generate Power Set

- The power set 2^S of a set **S** 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 directly extend the program for generating (n,k)-combinations to generate the whole power set

```
int n=3, seq[]={0,1,2,3}, k;
for (k=0; k<=n; k++)
   comb(n, k, seq+1, 0);
```

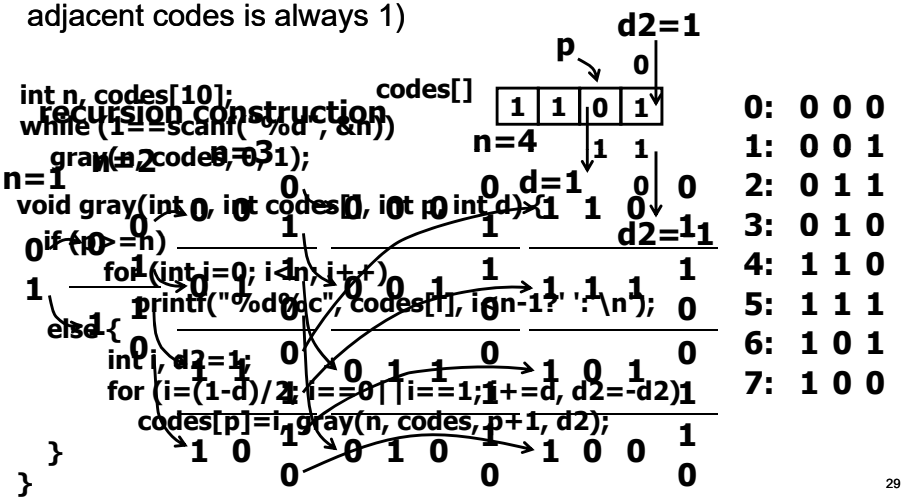
0	0	0
0	0	1
0	0	2
0	0	3

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

- the 2nd method is to count with S'=S ∪ {0}, only allow 0 to duplicate, and treat 0 as null element, e.g. S' = {0,1,2,3}
- A 3rd method is to use the binary representation of an integer from 0 to $2^{|S|}$-1: ex. S={1,2,3}, 5=$(101)_2$ means {1,3}

Gray Code Generation

- **Gray code** is also known as the **reflected binary code:** single-distance codes (the Hamming distance between adjacent codes is always 1)



Gray Code (Cont'd)

- **Gray code** is also known as the **reflected binary code:** single-distance codes (the Hamming distance between adjacent codes is always 1)

```
int i, n, codes[10];
while (1==scanf("%d", &n)) {
   for (i=0; i<n; i++) codes[i]=0;
   gray(n, codes, 0);
}
void gray(int n, int codes[], int p) {
  if (p>=n)
        for (int i=0; i<n; i++)
           printf("%d%c", codes[i], i<n-1?' ':'\n');
  else
        for (int i=0,j=codes[p] ; i<2; i++,j++ )
           codes[p]=j%2, gray(n, codes, p+1);
}
```

p, codes[], i, j

0: 0 0 0
1: 0 0 1
2: 0 1 1
3: 0 1 0
4: 1 1 0
5: 1 1 1
6: 1 0 1
7: 1 0 0

or gray(n, x, p+1), x[p]=(x[p]+1)%2, gray(n, x, p+1);

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	Sample Output
6	1 2 3 1 5 2 4 6

Sample Input	Sample Output
10	1 2 3 4 1 2 7 1 3 6 1 4 5 1 9 2 3 5 2 8 3 7 4 6 10

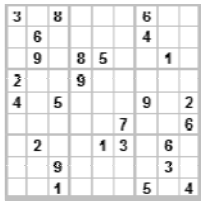
Sample Input	Sample Output
15	1 2 3 4 5 1 2 3 9 1 2 4 8 1 2 5 7 1 2 12 1 3 4 7 1 3 5 6 1 3 11 1 4 10 1 5 9 1 6 8 1 14

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.



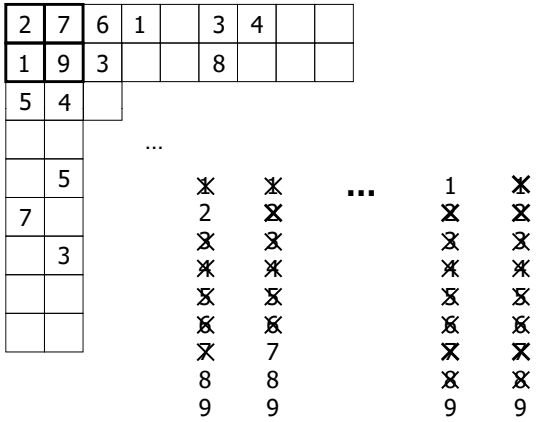
number of lines
row, column, value
row, column, value
…

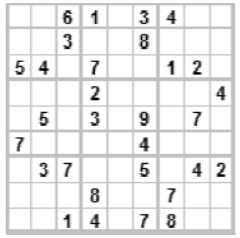
30
0, 0, 7
0, 1, 8
0, 2, 9
0, 4, 2
0, 8, 5
1, 0, 6
1, 5, 5
1, 7, 8
…

Initial configuration

Sudoku (cont'd)

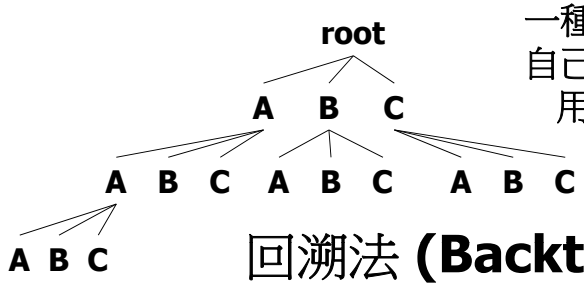
- Extension of counting





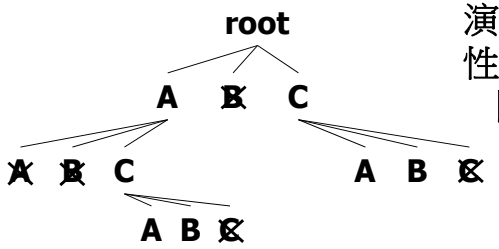
more constraints on the set of values to be filled in each cell

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)



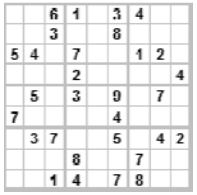
演算法: 窮舉所有的可能
性, 但是一發現不行趕快
回頭嘗試下一個可能
的路徑, 常常用遞迴
函式來實作, 也可以
用迴圈實作

12-1 尋找整除數字 **12-2** 速讀

Backtracting with Iteration

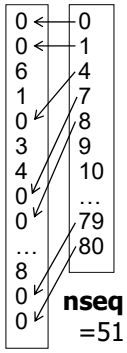
- **constraints**: DFS → backtracking

```
01 void sudoku(int board[], int seq[], int nseq, int p) {
02     while (p>=0)
03         if (p>=nseq) print_solution(), p--;
04         else if (board[seq[p]]>=9)
05             board[seq[p--]] = 0;
06         else {
07             board[seq[p]]++;
08             if (isValid(board, seq[p])) p++;
09         }
10 }
```



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

board seq



nseq =51

```
01 int isValid(int data[], int p) {
02     int i, j, r=p/9, c=p%9;
03     for (i=0; i<9; i++) {
04         if(i!=c && data[p]==data[r*9+i]) return 0;
05         if(i!=r && data[p]==data[i*9+c]) return 0;
06     }
07     for (i=r/3*3; i<r/3*3+3; i++)
08         for (j=c/3*3; j<c/3*3+3; j++)
09             if ((i!=r||j!=c) && data[p]==data[i*9+j]) return 0;
11     return 1;
12 }
```

```
01 int main() {
02     int board[81]={};
03     int seq[81], nseq=0;
04     read constraints to board[81]
05     prepare seq[81] and nseq
06     sudoku(board, seq, nseq, 0);
07     return 0;
08 }
```


Backtracking using Recursion

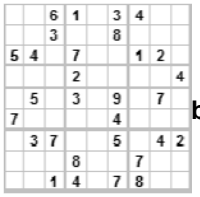
- without any constraint, raw **DFS** to generate 9^{81} possibilities

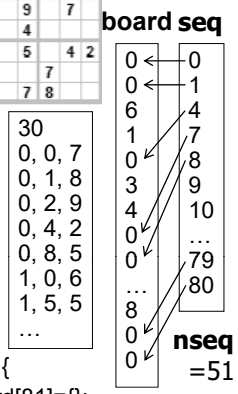
```
01 int main() {
02     int data[81]; // 9x9
03     sudoku(data, 0);
04     return 0;
05 }
```

```
01 void print(int data[]) {
02     int i, j;
03     for (i=0; i<9; i++)
04         for (j=0; j<9; j++)
05             printf("%2d%s", data[i*9+j], j==9?"\n":"");
06 }
```

```
01 void sudoku(int data[], int p) {
02     if (pivot>=81)
03         print(data);
04     else
05         for (data[p]=1; data[p]<=9; data[p]++)
06             sudoku(data, p+1);
07 }
```

Recursion (cont'd)



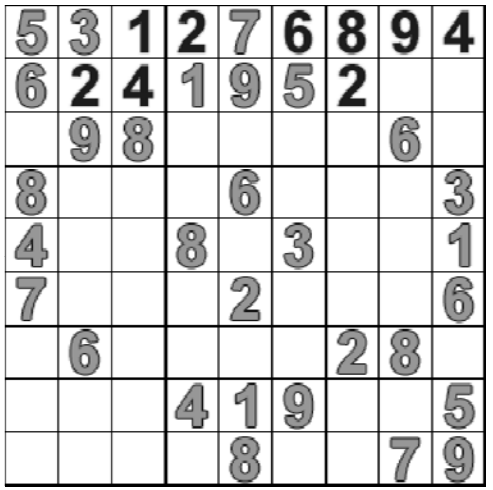


- **constraints**: DFS → backtracking

```
01 void sudoku(int board[], int seq[], int nseq, int p) {
02     if (p>=nseq)
03         print(board);
04     else
05         for (board[seq[p]]=1; board[seq[p]]<=9; board[seq[p]]++)
06             if (isValid(board,seq[p]))
07                 sudoku(board, seq, nseq, p+1);
08 }
```

```
01 int isValid(int data[], int p) {
02     int i, j, r=p/9, c=p%9;
03     for (i=0; i<9; i++) {
04         if(i!=c && data[p]==data[r*9+i]) return 0;
05         if(i!=r && data[p]==data[i*9+c]) return 0;
06     }
07     for (i=r/3*3; i<r/3*3+3; i++)
08         for (j=c/3*3; j<c/3*3+3; j++)
09             if ((i!=r||j!=c) && data[p]==data[i*9+j])
10                 return 0;
11     return 1;
12 }
```

```
01 int main() {
02     int board[81]={};
03     int seq[81], nseq=0;
04     read constraints to board[81]
05     prepare seq[81] and nseq
06     sudoku(board, seq, nseq, 0);
07     return 0;
08 }
```

Determinant of an n-by-n Matrix

- Leibniz formula

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

  - 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?

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

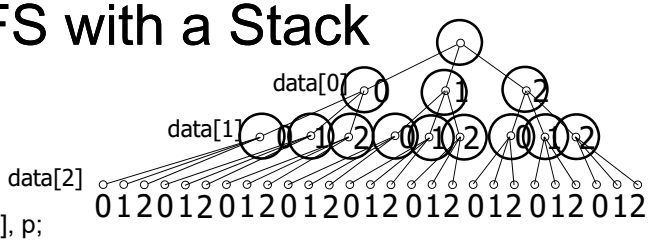
  - $\text{sign}(\sigma) = \begin{cases} 1 & \text{if } \sigma \text{ has even number of misplaced pairs} \\ -1 & \text{if … odd …} \end{cases}$

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。

DFS with a Stack




```
01 #include <stdio.h>
02 int main() {
03     int n=3, i, data[10], p;
04     int top=1,s[10*10][2]={{-1,0}};
05     while (top>0) {
06         p=s[--top][0];
07         if (p>=0) data[p]=s[top][1];
08         if (p==n-1)
09             for (i=0; i<n; i++)
10                 printf("%d%c", data[i], i<n-1?' ':'\n');
11         else
12             for (p++,i=n-1; i>=0; i--)
13                 s[top][0]=p, s[top++][1]=i;
14     }
15     return 0;
16 }
```

pop

push

p, v

data[p]=v;

p

	-1	0	1	2
data				

2, 0
2, 1
2, 2
2, 0
1, 1
1, 2
0, 0