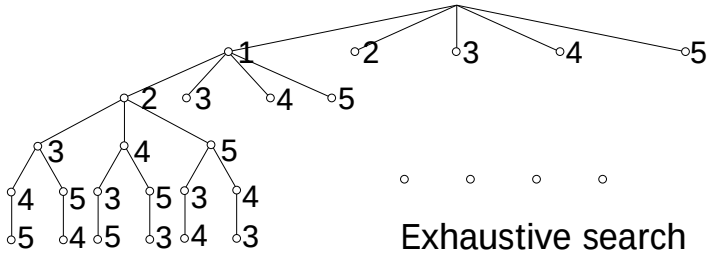
Permutation, Combination, and Related Problems

Pei-yih Ting

Introduction

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

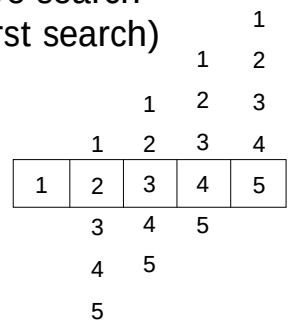
Generating Permutations



Exhaustive search (depth first search)

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

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
...			

- Is there other ***scalable*** program structure to generate the same (i1, i2, i3, i4) up-counting 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]++;
}
```

Scalable

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

```
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);
```

```
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;
}
```

index

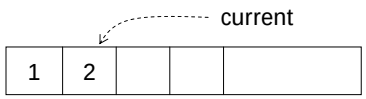
1	2	3	4
1	2	4	3
1	3	2	4
1	3	4	2
1	4	2	3
1	4	3	2
2	1	3	4
2	1	4	3
...			

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

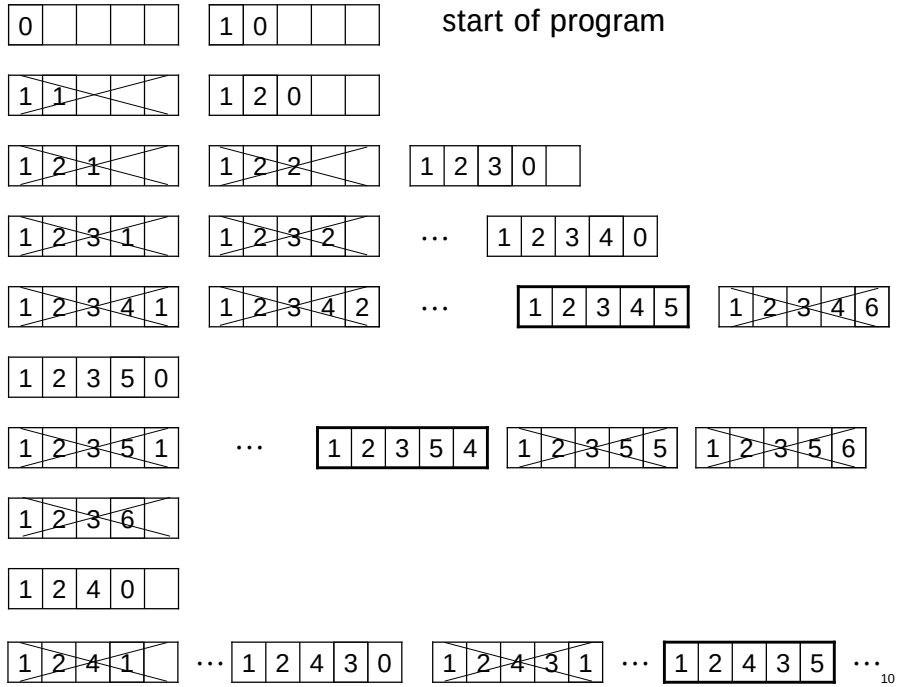

Permutation

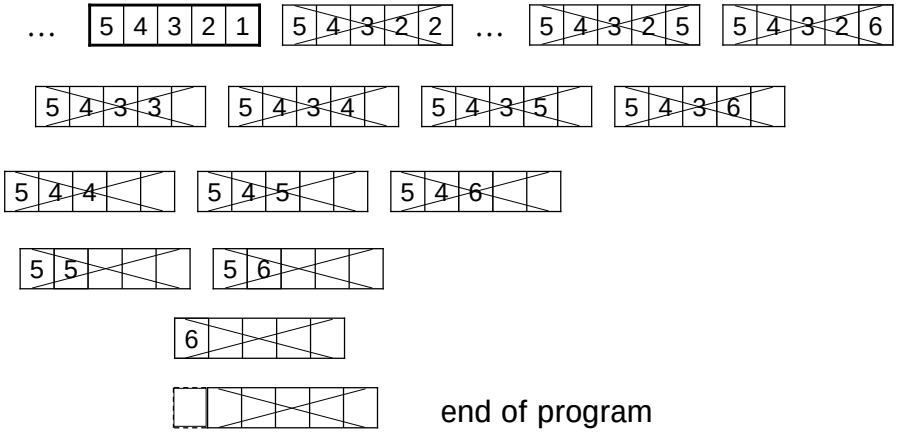
- If you view the index[] as an n-digit n-ary number (the digits set is {1, 2, …, n}), the previous program generates all possible increasing numbers (from 111…1 to nnn…n) and eliminates all invalid numbers with repeating digits on the way of counting up.
- In the following, you will see a variation very close to the previous program. The difference is that it tries to generate only valid increasing numbers with the next() function.

```
01 #include <stdio.h>
02
03 void main()
04 {
05    int size, perm[12] = {0}, current=0, solCount=0, i;
06
07    printf("Please input number of elements: ");
08    scanf("%d", &size);
09
10    while (current>=0)
11    {
12       current += next(size, current, perm);
13       if (current == size)
14       {
15          solCount++;
16          printf("%4d: ", solCount);
17          for (i=0; i<size; i++)
18             printf("%d ", perm[i]);
19          printf("\n");
20          current = size-1;
21       }
22    }
23    printf("Total %d permutations\n",solCount);
24 }
```

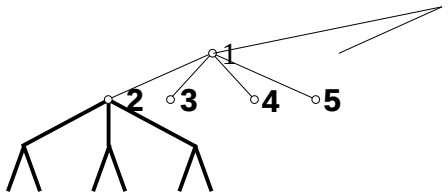



```
26 int next(int size, int pivot, int perm[]
27 {
28    int i, collision;
29
30    while (perm[pivot]++ < size)
31    {
32       collision = 0;
33       for (i=0; i<pivot; i++)
34          if (perm[pivot] == perm[i])
35          {
36             collision = 1;
37             break;
38          }
39       if (!collision) return 1;
40    }
41    perm[pivot] = 0;
42    return -1;
43 }
```




```
01 #include <stdio.h>
02 void next(int size, int pivot, int perm[])
03
04 void main()
05 {
06    int size, perm[12] = {0};
07    printf("Please input number "
08           "of elements: ");
09    scanf("%d", &size);
10    next(size, 0, perm);
11 }
12
13 void next(int size, int pivot, int perm[])
14 {
15    int i, collision;
16    static int count=0;
```



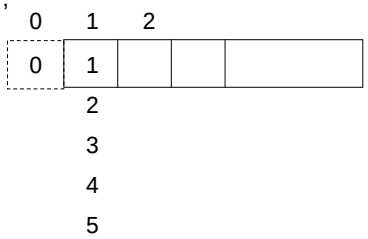

```
17    perm[pivot] = 0;
18    while (perm[pivot]++ < size)
19    {
20       collision = 0;
21       for (i=0; i<pivot; i++)
22          if (perm[pivot] == perm[i])
23          {
24             collision = 1;
25             break;
26          }
27       if (!collision)
28       {
29             if (pivot+1 < size)
30                next(size, pivot+1, perm);
31             else
32             {
33                printf("%4d: ", ++count);
34                for (i=0; i<size; i++)
35                   printf("%d ", perm[i]);
36                printf("\n");
37             }
38       }
39    }
40 }
```

Recursive Implementation

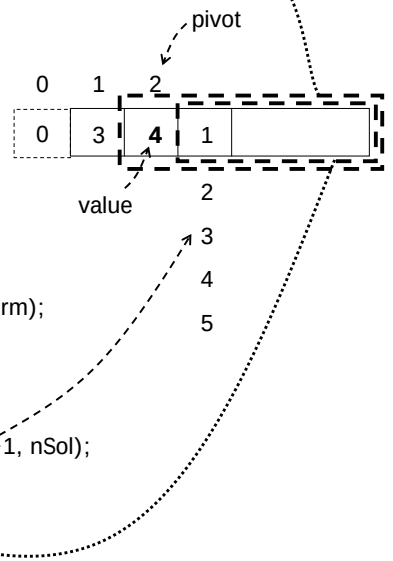
Another Recursive Implementation

```
09 void main()
10 {
11     int i, size, nSol=0;
12     int *perm;
13
14     do
15     {
16         printf("Please input N (N>=3): ");
17         scanf("%d", &size);
18     }
19     while (size<3);
20
21     perm = (int *) malloc((size+1)*sizeof(int));
22     for (i=0; i<=size; i++)
23         perm[i] = 0;
24
25     permutation(size, perm, 0, 0, &nSol);
26     printf("\n# of solutions = %d\n", nSol);
27
28     free(perm);
29     system("pause");
30 }
```

```
32 void printPerm(int iSol, int size, int *perm
33 {
34     int i;
35     printf("%d:", iSol);
36     for (i=1; i<=size; i++)
37         printf("%3d", perm[i]);
38     printf("\n");
39 }
40
41 int collision(int pivot, int *perm, int value)
42 {
43     int i;
44     for (i=1; i<=pivot; i++)
45         if (perm[i] == value)
46             return 1;
47     return 0;
48 }
```



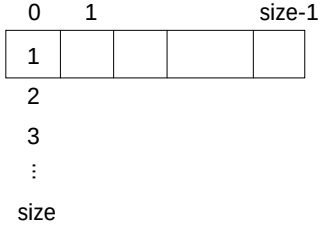

```
50 void permutation(int size, int perm[], int pivot, int value, int *nSol)
51 {
52     int i;
53
54     if (collision(pivot-1, perm, value))
55         return;
56
57     perm[pivot] = value;
58     if (pivot==size)
59     {
60         ++(*nSol);
61         if (*nSol % 100000 == 0) printf(".");
62         if (size<=6) printPerm(*nSol, size, perm);
63         return;
64     }
65
66     for (i=0; i<size; i++)
67         permutation(size, perm, pivot+1, i+1, nSol);
68     return;
69 }
```



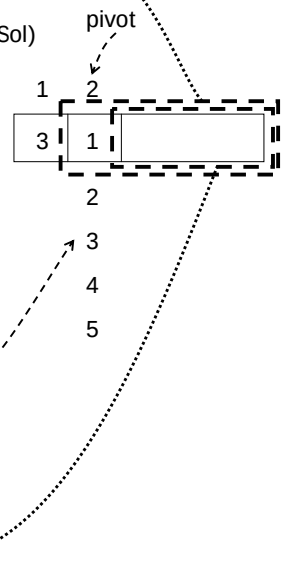
Slightly Simplified Implementation

```
09 void main()
10 {
11     int i, size, nSol=0;
12     int *perm;
13
14     do
15     {
16         printf("Please input N (N>=3): ");
17         scanf("%d", &size);
18     }
19     while (size<3);
20
21     perm = (int *) malloc(size*sizeof(int));
22
23     permutation(size, perm, 0, &nSol);
24     printf("\n# of solutions = %d\n", nSol);
25
26     free(perm);
27     system("pause");
28 }
```

```
30 void printPerm(int iSol, int size, int *perm
31 {
32     int i;
33     printf("%d:", iSol);
34     for (i=0; i<size; i++)
35         printf("%3d", perm[i]);
36     printf("\n");
37 }
38
39 int collision(int pivot, int *perm, int value)
40 {
41     int i;
42     for (i=0; i<=pivot; i++)
43         if (perm[i] == value)
44             return 1;
45     return 0;
46 }
```




```
48 void permutation(int size, int *perm, int pivot, int *nSol)
49 {
50     int i;
51
52     if (pivot >= size)
53     {
54         ++(*nSol);
55         if (*nSol % 100000 == 0) printf(".");
56         if (size<=6) printPerm(*nSol, size, perm);
57         return;
58     }
59
60     for (i=0; i<size; i++)
61         if (!collision(pivot-1, perm, i+1))
62         {
63             perm[pivot] = i+1;
64             permutation(size, perm, pivot+1, nSol);
65         }
66 }
```

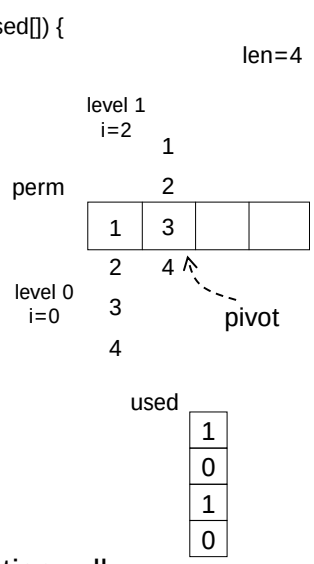


Another Recursive Implementation

```
void enumPerm(int pivot, int perm[],int len, int used[]) {
    if (pivot == len) printPerm(perm, len);
    for (int i=0;i<len;i++)
        if (!used[i]) {
            used[i] = 1;
            perm[pivot] = i+1;
            enumPerm(pivot+1,perm,len, used);
            used[i]=0;
        }
}

int main()
{
      int len=4, used[10], perm[10];
      for (i=0; i<10; i++) used[i] = 0;

      enumPerm(0, perm, len, used);
}
```



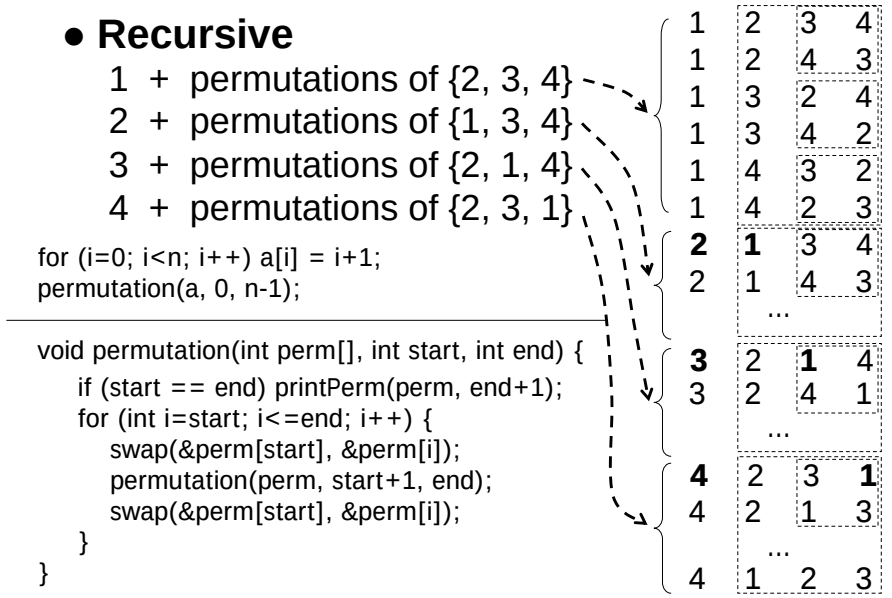
len+1 recursive function calls

Permutation from Swapping (1/4)

- **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}

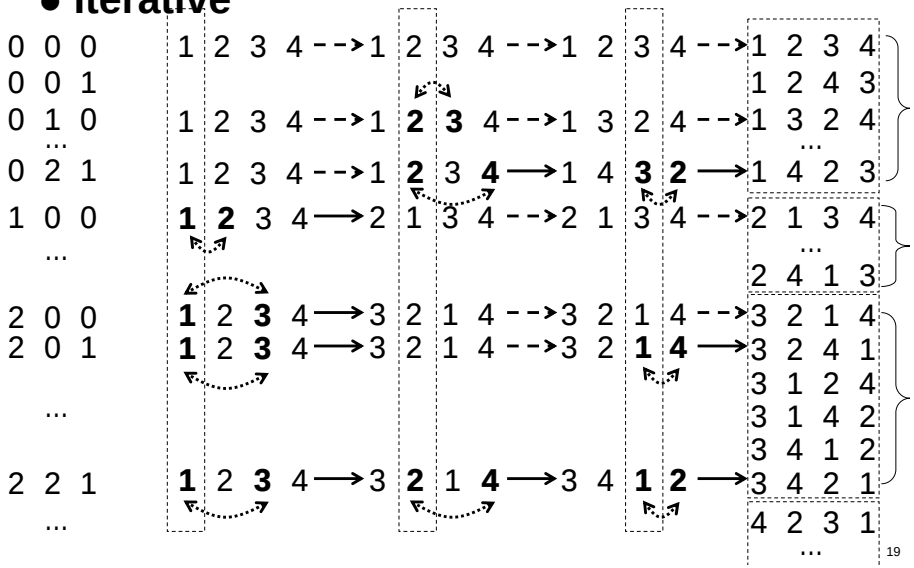
```
for (i=0; i<n; i++) a[i] = i+1;
permutation(a, 0, n-1);
```

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



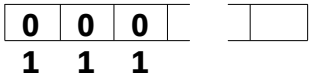
Permutation from Swapping (2/4)

- **Iterative**



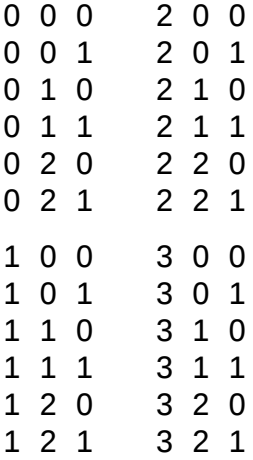
Permutation from Swapping (3/4)

- counting up




```
int next(int counter[], int size) {
    int i;
    for (i=size-1; i>=0; i--) {
        counter[i]++;
        if (counter[i]<=size-i)
            return 1;
        counter[i] = 0;
    }
    return 0;
}
```

Initially, 0 0 -1



Permutation from Swapping (4/4)

```
int main(void) {
    int i, n, perm[10], counter[9];
    printf("請輸入n 值(n<=10) ");
    scanf("%d", &n);
    for (i=0; i<n-2; i++) counter[i] = 0; counter[n-2] = -1;
    while (next(counter, n-1)) {
        for (i=0; i<n; i++) perm[i] = i+1;
        for (i=0; i<n-1; i++)
            if (counter[i])
                swap(&perm[i], &perm[i+counter[i]]);
        printPerm(perm, n);
    }
    system("pause");
    return 0;
}
```


Permutation from Rotation

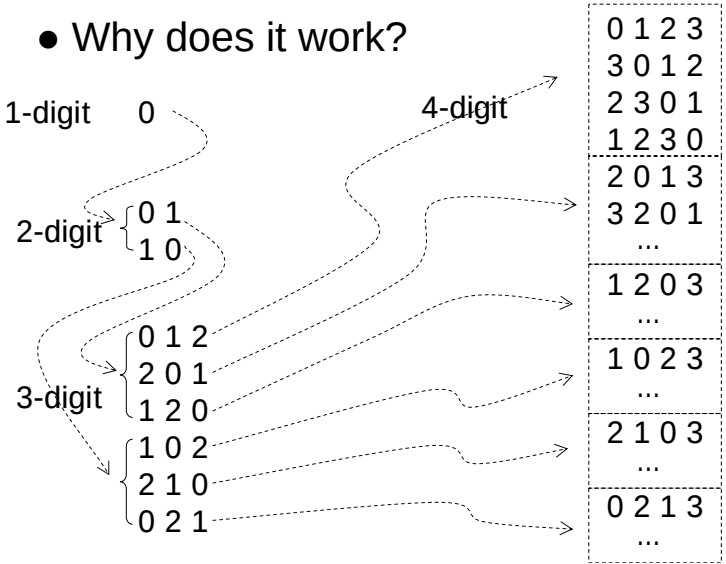
```
0 1 2 3   2 0 1 3   1 2 0 3   1 0 2 3   2 1 0 3   0 2 1 3
3 0 1 2   3 2 0 1   3 1 2 0   3 1 0 2   3 2 1 0   3 0 2 1
2 3 0 1   1 3 2 0   0 3 1 2   2 3 1 0   0 3 2 1   1 3 0 2
1 2 3 0   0 1 3 2   2 0 3 1   0 2 3 1   1 0 3 2   2 1 3 0
 0 1 2 3   2 0 1 3   1 2 0 3   1 0 2 3   2 1 0 3   0 2 1 3
     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



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

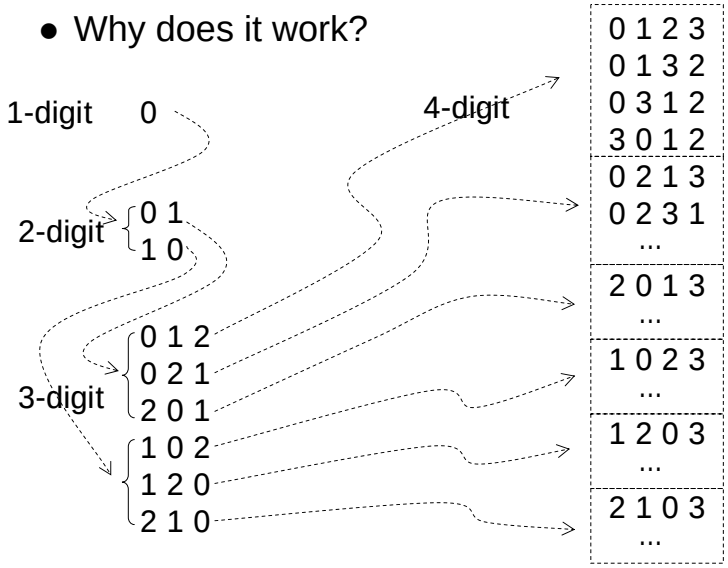
```
int shift_n_check(int size, char perm[],
                                char pos[]) {
  for (int i=size; i>=2; i--) {
    shiftRight(i, &perm[size-i], &pos[size-i]);
    if (pos[size-i] > 0) return 1;
  }
  return 0;
}
```

```
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))
```

```
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?



Generate Partitions

- A partition of a set X is a set of nonempty 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. $B_0 = 1$, $B_1 = 1$, $B_2 = 2$, $B_3 = 5$, $B_4 = 15$, $B_5 = 52$, $B_6 = 203$, …

$B_{20} = 51724158235372 \approx 5\text{x}10^{13}$, $B_{30} = 846749014511809332450147 \approx 8\text{x}10^{23}$,

$B_{40} = 157450588391204931289324344702531067 \approx 2\text{x}10^{35}$,

$B_{50} = 185724268771078270438257767181908917499221852770 \approx 2\text{x}10^{47}$.

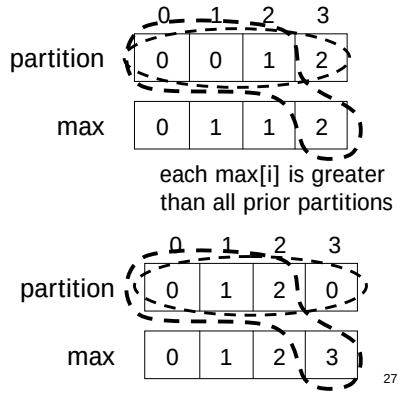
Generate Partitions (cont'd)

- A simple algorithm through counting up with dynamic 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 ==> $\{a_0\},\{a_1\},\{a_2\},\{a_3\}$

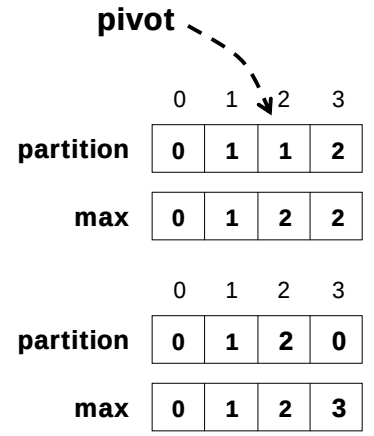
Implementation with 2 arrays

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



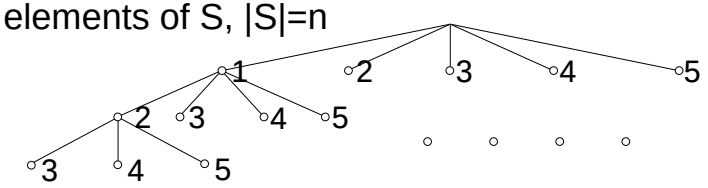
Implementation

```
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 }
```



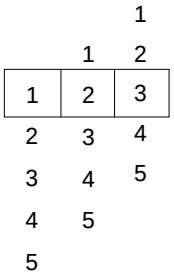
Generate (n,k)-Combinations

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



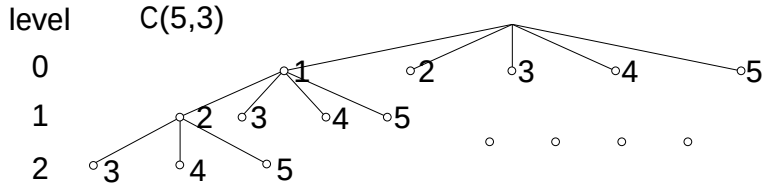
$C_3^5 = 10$
(5,3)-combinations

1	2	3
1	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



- Extend the counting-up program and skip those non-increasing sequences by dynamically decreasing candidate sets

Recursive (n,k)-Combinations



- select k-level elements from seq[start]~seq[n-1] and put the result to result[level]~result[k-1]

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

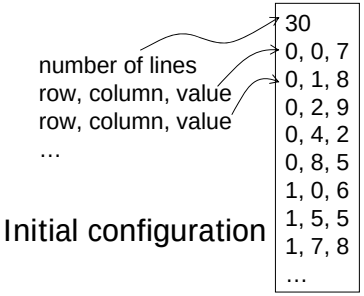
Generate Power Set

- The power set of a set is the collection of all subsets of S, including the empty set and S itself.
  - e.g. S = {x, y, z},
    2^S = {{}, {x}, {y}, {z}, {x,y}, {x,z}, {y,z}, {x,y,z}}

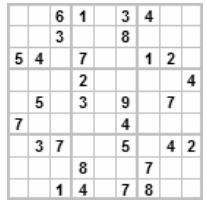
- We can extend the program for generating (n,k)-combinations to generate the whole power set.

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.



Initial configuration



Sudoku (cont'd)

- Extension of permutation generation

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

...

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

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

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

Determinant of an n-by-n Matrix

- Laplace 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?

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

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

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。