

COSC329 Assignment on Combinatorial Generation

Implement the following algorithms for permutation generation in C and confirm the outputs are correct for $n=4$. Then remove the output statement and measure the computing time for each method with $n=10, 11, \text{ or } 12$, depending on the speed of your computer. If the computer warrants, you can choose larger values of n .

Performance measurement project

- (1) Algorithm 5
- (2) Algorithm 6
- (3) Algorithm 7
- (4) Algorithm 8
- (5) Algorithm 9
- (6) Algorithm X. See attachment for this algorithm.
- (7) Algorithm H. See attachment for this algorithm.
- (8) Sudoku project. Replace the permutation algorithm in the Sudoku program attached with any efficient permutation algorithm, and also improve the overall structure for speed up. Measure the computing time with your own Sudoku example, and compare the time with that of the attached program with the same Sudoku.

Present the following in hardcopy

- A. Source lists (1) - (8) and outputs for $n=4$ for (1) • (7)
- B. Time measurements with $n=10, 11, \text{ or } 12$ for all of the above seven methods (1) • (7)
- C. Picture of the Sudoku, its solution and time measurements for project (8).
- D. Discussions on the performances of those methods and possible suggestions for improvements. Improvement is required for at least one of (1) • (7) and also for (8). Documentations for your improvements are required.

Items A-C are rather straightforward. Item D will make a difference in the grade.

Due: 5:00 pm, 5 June 2009. Drop due: 5:00 pm, 12 June 2009.

Worth 20%

In addition, you are required to submit electronically `prog1.c`, `€`, `prog8.c`, and `ff.dat`, corresponding to the above (1), `€`, (8). For electronic submission, follow the guideline below.

These programs must be able to be run with `•gcc`, and `•a.out`, . The user is supposed to give only n . For (8), a Sudoku must be provided in file `•, ff.dat`, . In my example, it took about 4 seconds. You should choose a moderately hard Sudoku for your experiment. For documentation style, an example is attached. For Sudoku, you can expand my documentation.

There are many Sudoku programs on the web, which may be more efficient than my program. For the purpose of the assignment, however, you are required to work on the attached program.

Grading policy. A = 30, B = 10, C = 10, D = 50 in percentage

Documentation of Algorithm 5

```

var used: array[1..100] of Boolean;
var a: array[1..100] of integer;
procedure perm(k);
  begin
    if k<=n then begin
      for i:=1 to n do
        if not used[i] then begin
          a[k]:=i;
          used[i]:=true;
          perm(k+1);
          used[i]:=false
        end
      else output(a)
    end;
  begin {main program}
    for i:=1 to n do used[i]:=false;
    perm(1)
  end.

```

Variables: n: size of permutations
i: item i

Procedures: perm: the main recursive procedure with parameter k

Arrays: a: container of permutations,
used : Boolean array to indicate if item i is used or not.

This algorithm generates permutations in lexicographic order. The procedure `perm`, is a recursive one. The parameter k controls the position of the container array `a`. In `perm`, item i is tested for $i=1, \dots, n$. If i is not used, `a[k]` is set to i, `used[i]` is set to true, and `perm` calls itself recursively with parameter k+1. The meaning of `used`, is to tell the positions downstream that item i is in use. After the call `perm(k+1)`, `used[i]` is reset to false. As the items are tested in increasing order at each position, obviously the order in which permutations are generated is lexicographic.

After `used`, is initialized to false for all i, the main program calls `perm`, at position 1. If the parameter k becomes n+1, the current permutation in `a`, is output. When we measure the computation time with large n, this output statement should be removed.

The algorithm is designed based on the simple idea of generating all n-ary strings, and removing non-eligible strings. The actual computation is slightly better than this, since the string `1 1 1`, for example, will cut further recursive calls. See the following for n=3.

1 1 1	1 3 1	2 2 1	3 1 1	3 3 1
1 1 2	1 3 2 ok	2 2 2	3 1 2 ok	3 3 2
1 1 3	1 3 3	2 2 3	3 1 3	3 3 3
1 2 1	2 1 1	2 3 1 ok	3 2 1 ok	
1 2 2	2 1 2	2 3 2	3 2 2	
1 2 3 ok	2 1 3 ok	2 3 3	3 2 3	

Algorithm X

```

int i,j,k,n;
int a[100], d[100];
out(){int i;
  for(i=1;i<=n;i++)printf("%d ",a[i]);
  getchar();
}
init(){int i; for(i=1;i<=n;i++)d[i]=i; }
p(int k){
int i;
  if(k<=n){
  for(i=1;i<=n-k+1;i++){
    a[k]=d[i]; d[i]=d[n-k+1]; // line 1
    p(k+1); // line 2
    d[i]=a[k]; // line 3
  }
  } else out();
}
main(){
  scanf("%d", &n); getchar();
  init(); p(1);
}

```

Output for n=4

1 4 3 2 This algorithm permutes based on the list d of available items. Parameter k
 1 4 2 3 controls the position of the container array a. At line 1, it consumes d[i] and
 1 2 4 3 bring the last element d[n-k+1] to d[i], and then recurs for position k+1
 1 2 3 4 at line 2. After that at line 3 a[k] is returned to d[i].

1 3 4 2

1 3 2 4 The snapshot of d at the beginning of each p(k).

2 1 3 4 The first few steps started from (k, i)=(1, 1) leading to the first output.

	k=1	k=2	k=3	k=4
2 4 1 3	d = 1 2 3 4	d = 4 2 3 4	d = 3 2 3 4	d = 2 2 3 4
2 4 3 1	< >	< >	< >	< >

2 3 1 4

2 3 4 1 The range for i is given by < >.

3 1 4 2

3 1 2 4 A few steps started from (k, i)=(1, 2)

3 2 1 4

	k=1	k=2	k=3	k=4
3 2 4 1	d = 1 2 3 4	d = 1 4 3 4	d = 3 4 3 4	d = 4 4 3 4
3 4 1 2	< >	< >	< >	< >

3 4 2 1

4 1 3 2

4 1 2 3 Proof by backward induction on k.

4 2 1 3 Basis. k=n. True.

4 2 3 1 Induction step. Suppose p(k+1) generates all permutations of d[1..n-k]
 4 3 1 2 in a[k+1..n]. p(k) puts d[1..n-k+1] one by one to a[k], replace d[1] by
 4 3 2 1 d[n-k+1], and call p(k+1). After that it recovers a[k] to d[1]. Thus p(k)
 generates permutations of d[1..n-(k-1)] in a[(k-1)+1..n].

Heap's algorithm (Algorithm H)

```
program ex(input,output);
var i,n:integer;
    a:array[1..10] of integer;
    p:array[0..10] of integer;
procedure out;
var k:integer;
begin
    for k:=1 to n do write(a[k]:6);
    readln;
end;
procedure swap(i,j:integer);
var w:integer;
begin
    w:=a[i]; a[i]:=a[j]; a[j]:=w
end;
procedure perm(n:integer);
var k:integer;
begin
    for k:=1 to n-1 do begin
        if n>2 then perm(n-1);
        if odd(n) then swap(1,n) else swap(k,n);
        out
    end;
    if n>2 then perm(n-1)
end;
begin
    write('input n ');
    readln(n);
    for i:=1 to n do a[i]:=i;
    out;
    perm(n);
    readln
end.
```

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

Lemma. Procedure perm(n) converts (1, 2, ..., n) into

(n, 2, 3, ..., n-1, 1) if n is odd

(n-2, n-1, 2, 3, ..., n-3, n, 1) if n is even

Proof. We can confirm the lemma for n up to 4 from the algorithm directly. Suppose n is greater than or equal to 5. Proof is by induction. Suppose the lemma is true for n.

Suppose n is odd and n+1 is even. We trace the changes from (1, 2, ..., n, n+1) after each perm(n) and swap(k, n+1).

(1, 2, ..., n, n+1)	perm	(n, 2, 3, ..., 1, n+1)	swap(1, n+1)
(n+1, 2, 3, ..., 1, n)	perm	(1, 2, 3, ..., n+1, n)	swap(2, n+1)
(1, n, 3, ..., n-1, n+1, 2)	perm	(n+1, n, 3, ..., n-1, 1, 2)	swap(3, n+1)
(n+1, n, 2, ..., n-1, 1, 3)	perm	(1, n, 2, ..., n-1, n+1, 3)	swap(4, n+1)
(1, n, 2, 3, ..., n-1, n+1, 4)	perm	(n+1, n, 2, 3, ..., n-1, 1, 4)	swap(5, n+1)
.....			
		(n+1, n, 2, 3, ..., 1, n-1)	swap(n, n+1)
(n+1, n, 2, 3, ..., n-2, n-1, 1)	perm	(n-1, n, 2, 3, ..., n-3, n-2, n+1, 1)	

Suppose n is even and n+1 is odd. We trace in a similar fashion with perm(n) and swap(1, n+1). In the following, the inner (a, ..., b) means the consecutive list of integers from a to b. If a>b, this is empty.

Symbols p for perm and s for swap

(1, 2,, n-2, n-1, n, n+1)	p	(n-2, n-1, (2, 3,, n-3), n, 1, n+1)	s
(n+1, n-1, (2,, n-3), n, 1, n-2)	p	(n-3, n, n-1, (2,, n-4), 1, n+1, n-2)	s
(n-2, n, n-1, 2,, n-4, 1, n+1, n-3)	p	(n-4, 1, n, n-1, ..., n-5, n+1, n-2, n-3)	s
.....			
(3, (6,, n-2), n+1, 1, n-1, 5, 4, 2)	p	(n-1, (5, ..., n-2), n+1, 1, n, 4, 3, 2)	s
(2, (5,, n-2), n+1, 1, n, 4, 3, n-1)	p	(n, (4,, n-2), n+1, 1, 3, 2, n-1)	s
(n-1, (4,, n-2), n+1, 1, 3, 2, n)	p	(1, (3, 4,, n-2), n+1, 2, n-1, n)	s
(n, (3, 4,, n-2), n+1, 2, n-1, 1)	p	(n+1, (2, 3,, n-1), n, 1)	

Theorem. Heaps' algorithm correctly generates permutations.

Proof. By induction on n. Again we confirm theorem for n up to 4 directly. Suppose n is greater than or equal to 5. Suppose theorem is true for n-1. We observe from the lemma that a[n] gets the value of 1, ..., n, each only once. By inductive hypothesis, perm(n-1) generates all permutations of the rest. Thus all permutations of 1, ..., n are generated in array a.

Outline of the Sudoku Program

Sudoku is to fill vacant positions in a 9 by 9 board with digits from 1 to 9 so that there is no conflict. Here conflict means there is no two same digits in columns, rows or blocks of size 3 by 3. We call those digits items to distinguish them from other numbers.

Example. We call the 9 by 9 array board.

```

-----
|   |   |   ||   |   |   || 4 | 6 |   | level 1
-----
|   | 1 |   ||   |   |   ||   |   |   | level 2
-----
|   | 9 | 4 || 1 |   | 2 || 5 |   |   |
+++++
| 6 | 5 |   || 7 |   |   ||   |   |   |
-----
|   | 2 | 3 ||   | 8 |   || 7 | 5 |   |
-----
|   |   |   ||   |   | 5 ||   | 2 | 9 |
+++++
|   |   | 7 || 2 |   | 3 || 9 | 4 |   |
-----
|   |   |   ||   |   |   ||   | 3 |   |
-----
|   | 8 | 2 ||   |   |   ||   |   |   | level 9
-----

```

This Sudoku is from the Press
May 4, 2009

The difficulty of a Sudoku is
roughly measured by the number
of vacant positions.
In this example, there are $81 - 27 =$
55 vacant positions.

The top level structure of the algorithm is a tree search from level 1 to level 9. The search goes through all permutations of candidates at each level. For example, at level 1, we have candidate items $\text{cand} = (1, 2, 3, 5, 7, 8, 9)$ to be placed at positions $\text{pos}(1) = (1, 2, 3, 5, 7, 8, 9)$. We generate all permutations of cand , and the list of candidates is placed in the list $\text{cand-list}(1)$. That is

$\text{cand-list}(1) = ((1, 2, 3, 5, 7, 8, 9), (1, 2, 3, 5, 7, 8, 9), \dots, (9, 8, 7, 5, 3, 2, 1))$
The size, $\text{size}(1)$, is $7! = 5040$. $\text{pos}(1) = (1, 2, 3, 4, 5, 6, 9)$

Take one permutation. Items in the permutation is placed at positions given by $\text{pos}(1)$. For each placement we check conflicts and if there is no conflict, we call the search recursively for the next level. An abstract form follows:

```

procedure search(i){
  if(i<=9) {
    Save row i of board to b1;
    for(k=1; k<=size(i)!; k++) // size(i) is the size of cand-list(i)
      Place the k-th permutation of cand-list(i) to positions in pos(i);
      if (no conflict) search(i+1);
    Recover b1 to row i of board;
  } else output(board)
}

```

```
}
```

Actual implementation is as follows:

We generate all permutations of integers (1, ..., size(i)), and store those in array permutation[i][k][*], where * means for a permutation list, and k means the k-th permutation. cand[i][*] is fixed and items in cand are accessed by integers in the permutation. Example. cand[1][*] = (1, 3, 7) and permutation[1][6][*] = (3, 2, 1). Then (7, 3, 1) are generated to be placed at vacant positions on row 1.

To generate all permutations Algorithm 6 is used. The output statement is replaced by the action to store the generated permutation to array •permutation, . In the assignment, you are required to replace the permutation algorithm by some other permutation algorithm.

Conflict check is done for vertical direction and blocks. Vertical check is done upwards and downwards from row i. Block check is done when we finish placements of candidates at row i where i is a multiple of 3. We could do this earlier so that we can save unnecessary searches.

Block check is done by sorting the nine items in the block into array •store, , and check neighbouring items for equality.

```
#include <stdio.h>
long long int counter, tt;
int permutation[10][400000][10];
int n; int a[100];
int b[10][10]; // Board for sudoku
int cand[10][10]; // cand[i][*]: candidates for row i
int size[10]; // size[i]: size of the i-th candidate list
int pos[10][10]; // pos[i][*]: candidate positions for row i
/** Check if there is duplicate in array "store" of size n=9 */
int sortcheck(int store[]){int i, j, k, t, min;
    t=0;
    for(i=1;i<=8; i++){
        min=store[i]; j=i;
        for(k=i+1;k<=9;k++)if(store[k]<min){ min=store[k]; j=k;}
        store[j]=store[i]; store[i]=min;
    }
    for(i=1;i<=8;i++)if(store[i]==store[i+1]){t=1; return t;}
    return t;
}
/** Store items in block b[i..i+2][j..j+2] into "store" */
/** Then check if there is duplicate by calling "sortcheck" */
int blockcheck(int bound){int i,j,k,l,t,ii;
    int store[10]; t=0;
    for(i=1;i<=bound;i=i+3)
    for(j=1;j<=9;j=j+3){
        ii=0;
        for(k=i;k<=i+2;k++)for(l=j;l<=j+2;l++)
            { ii++; store[ii]=b[k][l]; }
    }
}
```

```

    if(sortcheck(store)==1){t=1; return t;}
}
return t;
}
out1(){int i, j; // Solution is output
printf("Solution time= %lld millisec\n", (clock()-tt)/1000);
for(i=1;i<=9;i++){
    for(j=1;j<=9;j++)printf("%d ", b[i][j]);
    printf("\n");
}
}
int factorial(int n){int i, s; //n! is computed
s=1; for(i=1;i<=n;i++)s=i*s; return s;
}

/** a permutation is output for row i as the counter-th permutation */
out(int i, int a[], int n){
int j;
    counter++;
    for(j=1;j<=n;j++)permutation[i][counter][j]= a[j];
}
/** Permutation generation */
swap(int a[], int i, int j){
    int w;
    w=a[i]; a[i]=a[j]; a[j]=w;
}
reverse(int a[], int k, int n){
    int i;
    for(i=k;i<=(k+n-1)/2;i++)swap(a, i, n-i+k);
}
int minimum(int a[], int k, int n){
    int i,j,temp;
    j=0; temp=99;
    for(i=k;i<=n;i++){
        if ((a[i]<temp) && (a[i]>a[k-1])){ temp=a[i]; j=i; }
    }
    return j;
}
perm(int i, int a[], int k, int n){
int ii,j;
    for(ii=k;ii<=n;ii++){
        perm(i, a, k+1, n);
        if(ii<n){
            reverse(a, k+1, n);
            j=minimum(a, k+1, n);
            swap(a, k,j); out(i, a, n);
        }
    }
}
}
}

```

```

permmain(int i, int n){int j; /* n is size of permutation, 9 in our case */
    int a[10];
    for(j=1;j<=n;j++)a[j]=j;
    out(i, a, n);
    perm(i, a, 1, n);
}
int find_cand(int cand[], int pos[], int i){int j, k;
    int count=0;
    int member, c;
    for(k=1;k<=n;k++){
        member=0;
        for(j=1;j<=n;j++) if(b[i][j]==k){member=member+1; } /* Check k */
        if(member==0) {count++; cand[count]=k; } /* No occurrence of k */
    }
    c=0;
    for(j=1;j<=n;j++)if(b[i][j]==0){c++; pos[c]=j;} /* Record vacant positions */
    return c;
}
int check(int i, int k){
    int ii, j, t, p[10], test[10];
    t=0;
    for(j=1;j<=size[i];j++)p[j]=permutation[i][k][j]; /* k-th permutation */
    for(j=1;j<=size[i];j++)test[j]=cand[i][p[j]]; /* array •test, is checked with array b */
    /*** check upward ***/
    for(ii=i-1;ii>=1;ii--){
        for(j=1;j<=size[i];j++)
            if(test[j]==b[ii][pos[i][j]]){t=1; return t;}
    }
    /*** check downward ***/
    for(ii=i+1;ii<=9;ii++){
        for(j=1;j<=size[i];j++)
            if(test[j]==b[ii][pos[i][j]]){t=1; return t;}
    }
    if(t==0) for(j=1;j<=size[i];j++)b[i][pos[i][j]]=test[j];
    /*** block check ***/
    if((i/3)*3==i){t=blockcheck(i); if(t==1)return t;}
    return t;
}
search(int i){
    int j, k; int b1[10];
    for(j=1;j<=9;j++)b1[j]=b[i][j]; /* Save row i to b1 */
    if(i<=9){
        for(k=1;k<=factorial(size[i]);k++){ // k-th permutation in row i
            if(check(i, k)==0)search(i+1);
        }
        for(j=1;j<=9;j++)b[i][j]=b1[j]; /* Recover b1 to row I */
    }else out1();
}
main(){int i, j, k;

```

```

char *filename[10];
FILE *in_file;
int tempcand[10], temppos[10]; int tempsize;
printf("Input file name ");
scanf("%s", filename);
in_file=fopen(filename, "r");
n=9;
for (i=1;i<=n;i++) {
    for (j=1;j<=n;j++) fscanf(in_file, "%d ",&b[i][j]);
    fscanf(in_file, "\n");
}
printf("b=\n");
for (i=1;i<=n;i++) {
    for (j=1;j<=n;j++) printf("%d ",b[i][j]);
    printf("\n");
}
for(i=1;i<=9;i++){
    tempsize=find_cand(tempcand, temppos, i);
    size[i]=tempsize;
    for (j=1;j<=tempsize;j++) { cand[i][j]=tempcand[j];}
    for (j=1;j<=tempsize;j++) { pos[i][j]=temppos[j];}
}
for(i=1;i<=9;i++){
    counter=0;
    permmain(i, size[i]);
}
tt=clock();
search(1);
printf("\n");
for(k=1;k<=9;k++){
    for(i=1;i<=9;i++)printf("%d ",b[k][i]);
    printf("\n");
}
printf("total time = %d millisec\n", (clock()-tt)/1000);
}

```