Lecture Notes for COSC329

Lecture on Combinatorial Generation

1. Introduction

We consider the problem of generating all member of a certain class of combinatorial objects. These include binary strings, permutations, combinations, parenthesis strings, etc. Generation algorithms for those objects have practical applications. Binary strings, for example, can be used to test the behaviour of a logical circuit involving an n-bit register. The set of permutations can be used to obtain the shortest circuit tour of a salesperson for the travelling salesperson's problem. The order of generation is important. Two representative orders are lexicographic order and minimal change order. The former is clear from its name. The latter defines some order in which we go from one object to the next with a very few changes, one or two. In addition to the correctness of algorithms, our major concern is the time necessary to generate all the objects.

2. Binary Strings

Let us generate n bit binary strings in lexicographic order for n=4 as follows:

0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1

Let B(n) be the set of n bit binary strings arranged in lexicographic order. Then the set can be characterized as follows:

$$B(n+1) = \begin{array}{l} 0B(n) \\ 1B(n), \end{array}$$

where 0B(n) is the set of n+1 bit strings obtained by attaching 0 to the beginning to all string of B(n). 1B(n) is similar. The important point here is that those set are ordered using the order in B(n). This observation leads to the following recursive algorithm. Array a is to hold combinatorial objects. The output statement outputs array elements in reverse order, that is, i=n, ..., 1..

Algorithm 1

```
procedure binary(n);
   begin
      if n>0 then begin
           a[n]:=0; binary(n-1);
           a[n]:=1; binary(n-1)
      end else output(a)
    end;
    begin {main program}
      binary(n)
    end.
```

As we see from the above example, we have many changes form object to object occasionally, such as from 0111 to 1000, that is, n, changes. Based on this observation we give the following iterative algorithm for binary strings.

Algorithm 2

```
for i:=1 to n do a[i]:=0;
   repeat
     output(a);
     i:=0;
     while a[i+1]=1 do
        a[i+1]:=0;
        i:=i+1;
     end;
     a[i+1]:=1
   until i=n.
```

Let us analyze the computing time of these algorithms. We measure the time by the number of assignment statements executed on array a. Then for Algorithms 1, we have the following recurrence equation. Note that we exclude the time for output.

$T(1) = 2$
$T(n) = 2T(n-1) + 2,$ for $n>1$.

Theorem 1. $T(n) = 2^{(n+1)} - 2$. ($2^n = 2^n$)
Proof. By induction. Basis is true for n=1. Induction step. Assume the theorem is true for n-1. Then we have

$$T(n) = 2T(n-1) + 2 = 2*(2^n - 2) + 2 = 2^{(n + 1)} - 2$$

The analysis of Algorithm 2 is similar. Since we have 2^n binary strings, we see that the average time for one string is O(1). This time is sometimes called O(1) amortized time to distinguish from the average time based on the randomness of the input data, such as that for quicksort.

Next we devise algorithms for binary strings with minimal changes. The following is the binary reflected Gray code, invented by Gray in the 1930s with a patent from the US Government. See the following for n=4.

0 0 0 0
0 0 0 1
0 0 1 1
0 0 1 0
0 1 1 0
0 1 1 1
0 1 0 1
0 1 0 0
1 1 0 0
1 1 0 1
1 1 1 1
1 1 1 0
1 0 1 0
1 0 1 1
1 0 0 1
1 0 0 0

The general structure is given below. G(n) is the Gray code of length n

$$G(1) = \{0, 1\}$$
$$G(n) = 0G(n-1)$$
$$\quad\quad\quad 1G'(n-1),$$

where G'(n) is the Gray code of length n arranged in reverse order as follows:

$$G'(1) = \{1, 0\}$$
$$G'(n) = 1G(n-1)$$
$$\quad\quad\quad 0G'(n-1).$$

Theorem 2. In the Gray code G(n), we can go from a string to the next with a bit change.
Proof. By induction. Basis of n=1 is obvious. Induction step. Suppose the theorem is true for n-1. Then within 0G(n-1) and 1G(n-1), we can go with a bit change. Now the last of G(n-1) is the same as the first of G'(n-1). Thus we can cross the boundary between 0G(n-1) and 1G'(n-1) with a bit change. We can prove a similar property for G'(n).

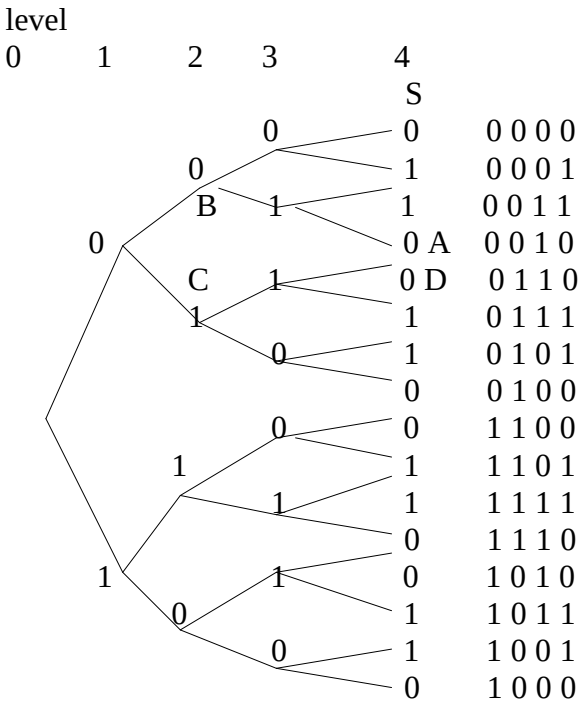
A recursive algorithm for the Gray code is given below. Gray1 and Gray2 correspond to G and G'.

Algorithm 3. Recursive Gray code

```
procedure Gray1(n);
begin
   if n>0 then begin
      a[n]:=0; Gray1(n-1);
      a[n]:=1; Gray2(n-1)
   end else output(a)
end;
procedure Gray2(n);
begin
   if n>0 then begin
      a[n]:=1; Gray1(n-1);
      a[n]:=0; Gray2(n-1)
   end else output(a)
end;
begin {main program}
  Gray1(n)
end.
```

Theorem 3. We have the same analysis for Gray1 and Gray2 as that for binary strings in lexicographic order in Theorem 1. Thus the amortized time for one string is O(1).
Proof. Hint. Set up recurrence for T1(n) and T2(n) for Gray1(n) and Gray2(n) as the numbers of assignment executions on “a”.

Although the above algorithm generates one string in O(1) amortized time, it sometimes spends up to O(n) time from string to string. In the following iterative algorithm, we spend O(1) worst case time from string to string. To how it works, we expand the process of generation for n=4 into in the following tree form.



We generate bit strings by traversing the above tree starting from S. The general move of “up”, “cross”, and “down” is demonstrated by the move (...,A,B,C,D, ...). The move A ->B is guided by the array “up”. The algorithm follows.

Algorithm 4. Iterative Gray code

```
program ex(input,output);
var i,n:integer;
   a,d,up:array[0..100] of integer;
procedure out;
var i:integer;
begin
  for i:=1 to n do write(a[i]:2);
  writeln
end;
begin
  readln(n);
  for i:=1 to n do a[i]:=0;
  for i:=1 to n do d[i]:=1;
  for i:=0 to n do up[i]:=i;
  repeat
    out;
    i:=up[n];
    up[n]:=n;
    a[i]:=a[i]+d[i];
     up[i]:=up[i-1];
     up[i-1]:=i-1;
     d[i]:=-d[i]
  until i=0;
end.
```

Exercise. Design an algorithm for ternary strings with minimal changes, in both recursive and iterative forms.

3. Permutations

There are very many algorithms for permutation generation. We start with observing the process in lexicographic order.

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

Let S be the set of items to be permuted. In the above S = {1, 2, 3, 4}.
A general recursive framework is given below.

```
procedure perm(n);
   begin
      if n>=0 then begin
        for each x in S in increasing order do begin
           delete x from S;
           perm(n-1);
           return x to S
        end
       else output(a)
   end;
   begin {main program}
      perm(n)
   end.
```

There are many ways to implement the set S. The following is not an efficient way.

Algorithm 5

```
var used: array[1..100] of Boolean;
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.
```

This algorithm takes O(n^n) time instead of O(n!) time, meaning the amortized time for one permutation can not be O(1). What is its amortized time?

The following is a more efficient method. This algorithm prepares the set of available items in increasing order after each recursive call. Reverse(k+1) is to reverse (a[k+1], ..., a[n]). Since the order was in decreasing order after each recursive call, we have this sequence in increasing order after “reverse”. After that we swap a[k] and a[j] where a[j] is the minimum in the sequence.

Algorithm 6

```
var a: array[1..100] of integer;
procedure swap(i, j);
var w:integer;
begin  w:=a[i]; a[i]:=a[j]; a[j]:=w;end;
function minimum(k);
var i,j,temp:integer;
begin
  j:=0; temp:=99;
  for i:=k to n do
    if (a[i]<temp) and (a[i]>a[k-1]) then begin
      temp:=a[i]; j:=i
    end;
  minimum:=j;
end;
procedure reverse(k);
var i:integer;
begin  for i:=k to (k+n-1) div 2 do swap(i, n-i+k);end;
porcedure perm(k);
    begin
        if k<=n-1 then
           for i:=k to n do begin
                perm(k+1);
                if i<>n then begin
                   reverse(k+1);
                   j:=minimum(k+1)
                   swap(k,j); output(a)
                end
            end
end;
begin {main program}
   read(n);
   for i:=1 to n do a[i]:=i;
   output(a);
   perm(1)
end.
```

The next is the iterative version of the above algorithm.

Algorithm 7.

```
{Declarations variables and procedures are the same as before}
begin {main program}
   readln(n);
```

```
for i:=1 to n do a[i]:=i;
out;
repeat
  i:=n;
  while a[i-1]>a[i] do i:=i-1;
  i:=i-1;
  if i>0 then begin
    reverse(i+1);
    j:=minimum(i+1);
    swap(i,j);
    out
  end
until i=0;
```

In the while loop, the algorithm scans the sequence from left to right while it is monotone increasing. It sets i to the first point after the peak. Then it reverses the increasing sequence, and swap a[i] and a[j] where a[j] is minimum among the reversed portion.

We analyze Algorithms 6 and 7 by counting the executions in the assignment statements on array a in the procedure reverse. Other times are proportional to it. We express this number by T(n). Then we have the following recurrence.

$$T(1) = 0$$

$$T(n) = nT(n-1) + n(n-1)$$

Theorem 4. T(n) = O(n!). Thus the amortized time for one permutation is O(1).
Proof. We first prove the following by induction.

$$T(n) = \sum_{i=1}^{n-1} n(n-1) \ldots (n-i) \quad \text{for } n > 1, \quad T(1) = 0$$

Basis follows from the definition. Induction step. Assume the theorem is true for n-1. Then we have

$$T(n) = .n \sum_{i=1}^{n-2} (n-1)(n-1-1) \ldots (n-1-i) + n(n-1)$$

$$= \sum_{i=1}^{n-1} n(n-1) \ldots (n-i)$$

From this we see

$$T(n) = n!(1 + 1/2! + \ldots + 1/(n-2)!) \leq (e-1)*n!,$$

where e = 2.719..., and we use the fact that $e = 1 + 1/1! + 1/2! + 1/3! + ...$

4. The Johnson-Trotter Algorithm

This method swaps items from permutations to permutation. in the following way. We take an example of n up to 4. Let P(n) be the set of permutations.

P(1) = {1}	P(2) = 1 2	p(3) = 1 2 3
	2 1	1 3 2
		3 1 2
		3 2 1
		2 3 1
		2 1 3

P(4) =	1 2 3 4	3 1 2 4	2 3 1 4
	1 2 4 3	3 1 4 2	2 3 4 1
	1 4 2 3	3 4 1 2	2 4 3 1
	4 1 2 3	4 3 1 2	4 2 3 1
	4 1 3 2	4 3 2 1	4 2 1 3
	1 4 3 2	3 4 2 1	2 4 1 3
	1 3 4 2	3 2 4 1	2 1 4 3
	1 3 2 4	3 2 1 4	2 1 3 4

The definition of order is recursive. The item “n” moves from right to left in the first permutation in P(n-1), and from left to right in the next permutation, etc

We first make the following nested structure.

```
Start from {1, 2, 3, 4}
d[2]:=-1; d[3]:=-1; d[4]:=-1;
for i2:=1 to 2 do begin
   for i3:=1 to 3 do begin
      for i4:=1 to 4 do
         if i4<4 then move 4 by d[4].
      d[4]:=-d[4];
      if i3<3 then move 3 by d[3]
   end;
   d[3]:=-d[3];
   if i2<2 then move 2 by d[2]
end;
d[2]:=-d[2]
```

As we can not have a variable number of nested loops, we have the following recursive algorithm that simulate the above algorithm. In the algorithm, 4 is generalized into n. The call nest(k) corresponds to the k-th loop. The array element p[i] holds the position of item i.

Algorithm 8. Recursive Johnson

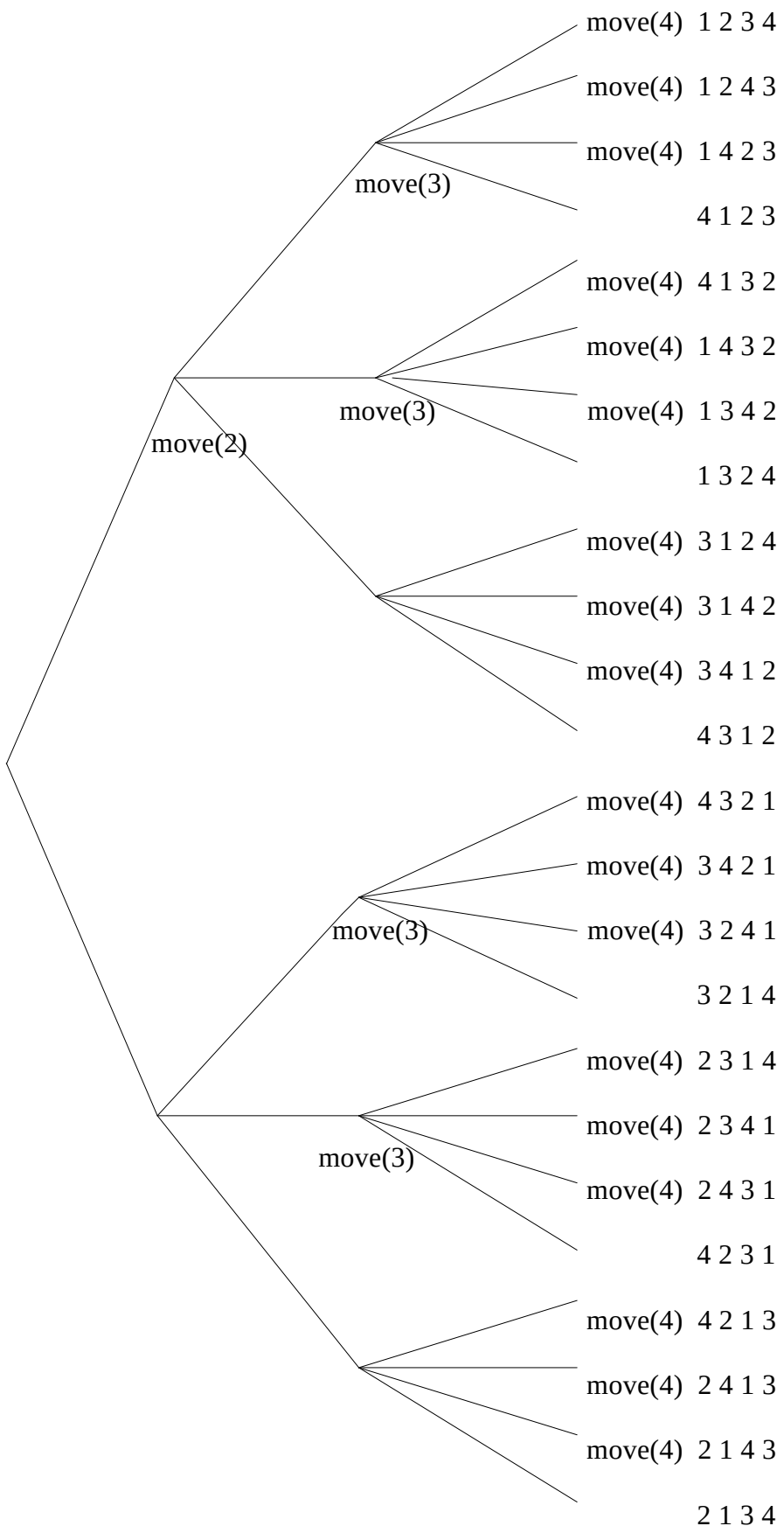
```
program ex(input,output);
var i,j,k,m,n:integer;
   a,d,p:array[1..100] of integer;
procedure out;
var i:integer;
begin
  for i:=1 to n do write(a[i]:2);
  writeln
end;
procedure move(x:integer);
var w:integer;
begin
  w:=a[p[x]+d[x]]; a[p[x]+d[x]]:=x; a[p[x]]:=w;
  p[w]:=p[x]; p[x]:=p[x]+d[x];
end;
procedure nest(k:integer);
var i:integer;
begin
  if k<=n then begin
    for i:=1 to k do begin
      nest(k+1);
      if i<k then begin move(k); out end;
    end;
    d[k]:=-d[k]
  end
end;
begin
  readln(n);
  for i:=1 to n do begin a[i]:=i; p[i]:=i; d[i]:=-1 end;
  out;
  nest(2)
end.
```

Theorem 5. The amortized time for one permutation by Algorithm 8 is O(1).
Proof. The time spent from two adjacent permutations (..., n) to (..., n), or from (n, ...) to (n, ...), where n does not move is O(n). There are (n-1)! such occasions. The time from permutation to permutation where n moves is obviously O(1). Thus the total time is

$$O(n(n-1)!) + O(n!) = O(n!).$$

Hence the amortized time for one permutation is O(n!)/n! = O(1).

To convert the above recursive algorithm into an iterative one, we analyze the tree structure of the recursive calls in the following.



When we go to a last branch, we update the value of array “up”, so that we can navigate to go up to an appropriate node. The core part of the iterative algorithm follows.

Algorithm 9. Iterative Johnson

```
procedure move(x:integer);
var w:integer;
begin
  c[x]:=c[x]+1;
  w:=a[p[x]+d[x]]; a[p[x]+d[x]]:=x; a[p[x]]:=w;
  p[w]:=p[x]; p[x]:=p[x]+d[x]; out
end;
begin
  readln(n);
  for i:=1 to n do a[i]:=i;
  for i:=1 to n do c[i]:=0;
  for i:=1 to n do d[i]:=-1;
  for i:=1 to n do p[i]:=i;
  for i:=0 to n do up[i]:=i;
  out;
  repeat
    i:=up[n];
    up[n]:=n;
    if i>1 then move(i);
    if c[i]=i-1 then begin
      up[i]:=up[i-1];
      up[i-1]:=i-1;
      d[i]:=-d[i];
      c[i]:=0
    end
  until i=1;
end.
```

Theorem 6. The worst case time from permutation to permutation of Algorithm 9 is O(1).
Proof. Obvious.

6. Combinations

We consider the problem of generating combinations of n elements out of r elements. As usual, we start with an example of n=4 and r=6 in the following. The order is lexicographic.

```
1 2 3 4
1 2 3 5
1 2 3 6
1 2 4 5
1 2 4 6
1 2 5 6
1 3 4 5
1 3 4 6
1 3 5 6
1 4 5 6
2 3 4 5
2 3 4 6
2 3 5 6
2 4 5 6
3 4 5 6
```

We have the following obvious nested loop structure for the generation.

Algorithm 10.

```
for i1:=1 to 3 do
   for i2:=i1+1 to 4 do
      for i3:=i2+1 to 5 do
         for i4:=i3+1 to 6 do
            writeln(i1,i2,i3,i4)
```

We translate this into the following recursive algorithm.

Algorithm 11. Recursive algorithm for combinations

```
procedure nest(k, j);
begin
   if i<=n then begin
      for i:=j to r-n+k do begin
         a[k]:=i;
         nest(k+1, i+1);
      end
   else output(a)
end;
begin
   read(n, r);
   nest(1, 1)
end.
```

This algorithm is converted to an iterative one in the following.

Algorithm 12.

```
readln(n,r);
a[0]:=-1;
for i:=1 to n do a[i]:=i;
j:=1;
while j<>0 do begin
  out;
  j:=n;
  while a[j]=r-n+j do j:=j-1;
  a[j]:=a[j]+1;
  for i:=j+1 to n do a[i]:=a[i-1]+1
end;
```

Let us analyze the move from 1 4 5 6 to 2 3 4 5 in the example. The inner while loop scans from the end. While the value at the j-th position is the maximum possible at the position, we keep going to the left. In this example we stop at the 1st position with the value of 1. Then we add 1 to 1 and make the increasing sequence to the right. In this case we create 3 4 5 to the right of 2.

Theorem 7. The amortized time for one combination by Algorithm 11 or 12 is O(1) for most of n and r, i.e., except for n = r - o(r).
Proof. We measure the time by the number of updates from combination to combination. We have a change by "a[j]:=a[j]+1" in the above algorithm always. This number accounts for C(r, n) where C(r, n) = r!/(n!(r-n)!) is a binomial number, the number of combinations when n items are taken out of r items. When a[n]=r, this statement is executed once more to reset a[n] to some value. When a[n-1] = r-1, this is executed once more to reset a[n-1] to some value, ... etc. This leads to the following measurement of cost, T(r, n).

$$T(r, n) = C(r, n) + C(r-1, n-1) + \ldots + C(r-n, 0) = C(r+1, n).$$

Dividing this value by C(r, n) yields

$$T(r, n)/C(r, n) = (r+1)/(r+1-n),$$

which the amortised time for one combination.

7. Combinations in vector form

We consider the problem of generating combinations in a binary vector form, that is, generation all subsets of size k out of the set of size n. We take the following example of k=4 and n=6 in lexicographic order.

```
0 0 1 1 1 1
0 1 0 1 1 1
0 1 1 0 1 1
0 1 1 1 0 1
0 1 1 1 1 0
1 0 0 1 1 1
1 0 1 0 1 1
1 0 1 1 0 1
1 0 1 1 1 0
1 1 0 0 1 1
1 1 0 1 0 1
1 1 0 1 1 0
1 1 1 0 0 1
1 1 1 0 1 0
1 1 1 1 0 0
```

Let B(n, k) be the set of those sequences, i.e., n=6 and k=4 in the example. Then a general scheme looks like

```
B(n, k)  =   0B(n-1, k)
             1B(n-1, k-1).
```

A recursive algorithm follows.

Algorithm 13.

```
program ex(input,output);
var i,j,k,m,n:integer;
   a:array[0..100] of integer;
procedure binary(i, k:integer);
begin
  if (k>=0) and (i<=n-k) then begin
    a[i]:=0;
    binary(i+1, k);
    a[i]:=1;
    if k>0 then binary(i+1,k-1);
  end
  else out
end;
begin
  write('input k, n '); readln(k,n);
  for i:=n-k+1 to n do a[i]:=1;
  binary(1,k)
end.
```

The procedure binary(i, k) is to generate the sequences from the i-th position to the n-th position with k 1's. An iterative version is left as an exercise. The amortized time for one vector is O(1) for most values of k and n.

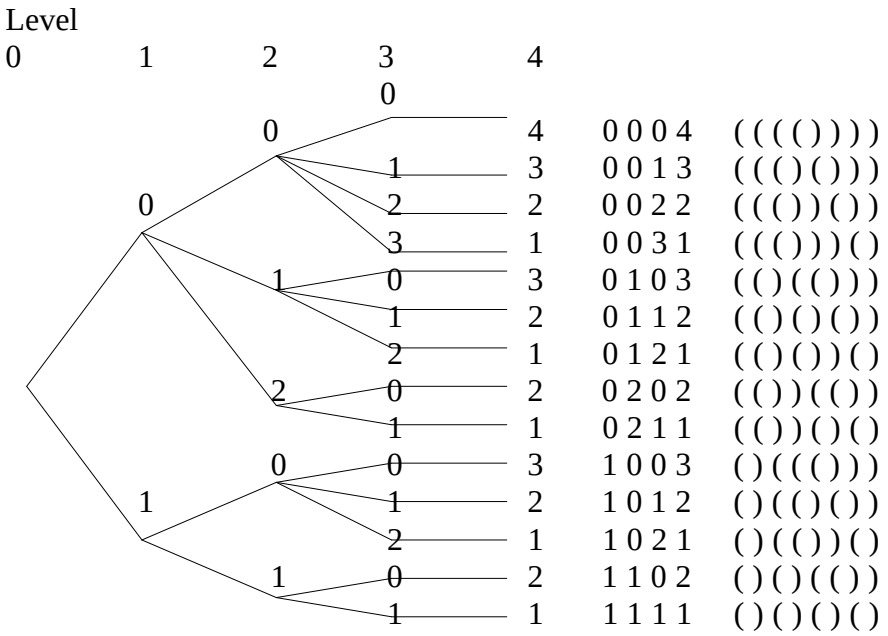
Notice that the last sequence in 0B(n-1, k) and 1B(n-1, k-1) are different at two positions. Thus if we modify the order of B by

$$B(n, k) = 0B(n-1, k)$$
$$\quad\quad\quad 1B'(n-1, k-1),$$

where B'(n,k) is the reversed set of B(n, k), then we can go from a sequence to the next with two changes. An iterative version of this approach establishes O(1) worst case time. Details are omitted. In this case we use O(r) space. There is another algorithm that establishes O(1) worst case time from combination to combination with O(n) space, that is, combinations are given in an array of size n. See attached paper.

8. Parenthesis strings

We consider the problem of generating well-formed parenthesis strings of n pairs. We go with an example as usual with n=4.



As we see in the above there are 14 sequences for n=4. At the left hand side are corresponding characteristic sequences a[1], ..., a[n] defined in the following.

a[i] is the number of right parentheses between the i-th left parenthesis and the (i+1)-th left parenthesis.

Confirm that the above characteristic sequences are correct. The definition of well-formedness is now defined by a[1] + ... + a[i] $\leq$ i and a[1] + ... + a[n] = n.

If we define s[i] = a[1] + ... + a[i] for i=1, ..., n, and s[0]=0, the above condition can be restated as follows:

The value of a[i] can change over the range of 0, 1, ..., i - s[i] and a[n] = n-s[n-1].

This observation leads to the following recursive algorithm for generation parenthesis strings in lexicographic order, where the order is defined in terms of the lexicographic order of the characteristic sequences.

Algorithm 14. Parenthesis strings

```
program ex(input,output);
var i,j,k,m,n:integer;
   a:array[0..100] of integer;
   q:array[0..100] of char;
procedure out;
var i:integer;
begin
  for i:=1 to n do write(a[i]:2);
  for i:=1 to 2*n do write(q[i]:2);
  readln
end;
procedure paren(i,s:integer);
var i,t:integer;
begin
  if i<=n-1 then begin
    for k:=0 to i-s do begin
      a[i]:=k;
      t:=s+a[i];
      q[t+i+1]:='(';
      paren(i+1,t);
      q[t+i+1]:=')'
    end
  end
  else if i=n then begin a[i]:=n-s; out end
end;
begin
  readln(n);
  for i:=1 to 2*n do q[i]:=')';
  paren(0,0);
end.
```

The correctness of this algorithm is seen by observing the changes of k at level i=2, for example, in the previous figure. When the accumulated sum of s=a[1]=0, k changes as k=0, 1, 2. For each value of k, we set q[s+k+i+1] to “(“ and call the procedure recursively. After that, we set q[s+k+i+1] to “)”, and increase k by 1. The general situation is described in the following.

```
1 2 3        s+i
(  ) ( ... ) ...( (        0th paren
(  ) ( ... ) ...( ) (      1st paren
(  ) ( ... ) ...( ) ) (   2nd paren
```

The number of parenthesis strings can be calculated from the following table. We first start from binomial numbers. We count the number of paths from coordinate (0, 0) to coordinate (i, j), denoted by B(i, j). Obviously we have

B(i, 0) = 1, for i = 1, 2, ..., B(0, j) = 1, for j=1, 2, ...
B(i, j) = B(i, j-1) + B(i-1, j)

	0	1	2	3	4	5
0	*	1	1	1	1	1
1	1	2	3	4	5	6
2	1	3	6	10	15	21
3	1	4	10	20	35	56
4	1	5	15	35	70	126
5	1	6	21	56	126	252

The value of B(i, j) defines the binomial number C(i+j, i) = C(i+j, j).

Now the number H(n) of parenthesis strings of length n is calculated by the number of paths from co-ordinate (0, 0) to co-ordinate (n, n) without crossing the diagonal as shown below.

		0	1	2	3	4	5
0		*	1	1	1	1	1
1	#	0	1	2	3	4	5
2		0	0	2	5	9	14
3		0	0	0	5	14	28
4		0	0	0	0	14	42
5		0	0	0	0	0	42

	-1	0	1	2	3	4	5
0		*	*	*			
1	#			*			
2	#			*			
3	#	#	#	*			
4				*	*	*	
5						*	*

The number H(n) is the number X of paths from (0, 0) to (n, n) minus the number Y of paths that visit in violation of parenthesis strings the co-ordinate (i+1, i) for i between 0 and n-1, both ends inclusive. As we saw before, X = C(2n, n). To estimate Y, take up an arbitrary path that satisfies the condition for Y and let the first co-ordinate of violating visit be (i+1, i). Let us flip the portion of the path from (0, 0) to (i+1, i) symmetrically over the diagonal (-1, 0), (1, 0), ..., (i+1, i). See the portion shown by "#" in the above figure. Then the original path and this modified path correspond one to one. Thus the number Y is the number of paths from (-1, 0) to (n, n), which is C(2n, n-1). Thus H(n) is given by

H(n) = C(2n, n) - C(2n, n-1) = C(2n, n)/(n+1).

This number H(n) is called the n-th Catalan number.

Theorem 8. The amortized time for one parenthesis string by Algorithm 14 is O(1).
Proof. Observe in the figure of the tree of recursive calls that the number of nodes at level i is C(2i, i). From Stirling's approximation formula n! ≅ (√ 2π n)(n^n)(e^(-n)), we have

$C(2n, n) = O(2^{(2n)}/(n\sqrt{n}) = O(4^n)$.

The time spent from a node to the next sibling node is proportional to the path length from the node to the leaves. Thus from the following lemma, we have the result.

Lemma. For a general recursive algorithm, suppose we expand the execution into a tree of recursive calls where each call produce at most c calls and the depth of tree is n. Suppose the time spent from a node to the next sibling node is proportional to the path length form the node to the leaf level. Suppose also the lengths from all nodes at the same level to the leaf level are equal. Then the total time is $O(c^n)$, and hence the amortized time for each node is $O(1)$.
Proof. Ignoring the constant factor, the time is given by

$$T = c(n-1) + (c^2)(n-2) + \ldots, (c^{(n-1)}) + c^n$$

$$= n(c + c^2 + \ldots c^{(n-1)}) - (c + 2(c^2) + \ldots + (n-1)(c^{(n-1)}) + c^n$$

$$= nc(c^{(n-1)}-1)/(c-1) - ((n-1)c^n - (c^n - 1)/(c -1)) + c^n$$

$$= O(c^n).$$

On the other hand there are N nodes where

$$N = c + c^2 + \ldots c^n = c(c^{(n+1)} -1) = O(c^n).$$

Thus we have

$$T/N = O(1)$$

There exist algorithms that can generate parenthesis strings by swapping two parentheses at a time, and also there exist algorithms that can generate parenthesis strings in $O(1)$ worst case time per string. See attached paper.