Mark Nelson

Programming, mostly.

Search

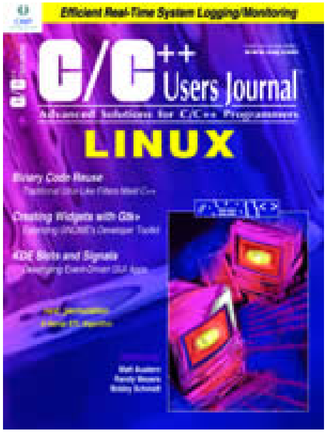
- Home
- About Mark Nelson
- Archives
- Liberal Code Use Policy

C++ Algorithms: next_permutation()

« XML Schema Validation in the Microsoft Universe
Building a URL Scanner With Java 1.3 »

Posted in March 1st, 2002
by Mark Nelson in Magazine Articles

Like 12 1



C/C++ Users Journal

March, 2002

Note: Thanks to Shawn McGee for pointing out an error in Figure 1. The print edition of this article in C/C++ Users Journal had an unfortunate extra line!

My daughter's math teacher at Hockaday School in Dallas wants his sixth-grade students to enjoy their class. He's fond of sending home interesting problems that are meant to be both entertaining and enriching. As most parents probably know, this can only mean trouble!

Last week Mr. Bourek sent home a worksheet containing a set of variations on the traditional magic square. Students were given various shapes, such as triangles, stars, and so on, and asked to fill in a set of consecutive numbers at the vertices. The goal was to come up with an arrangement of numbers such that various rows, columns, and diagonals all added up to a given sum.

Kaitlin worked her way through most of the problems in fairly quick order. But the shape shown in Figure 1 managed to stump her.

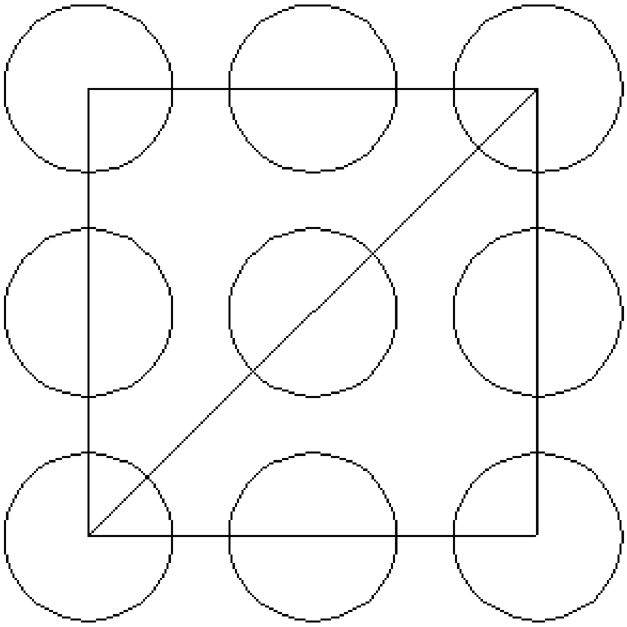


Figure 1 - Sum = 17

The problem was simple enough. All she had to do was place the numbers 1 through 9 in the nine positions of the figure so that the sum of all the straight lines was 17. Although Kate was able to knock the other problems out quickly, this one was still unsolved after fifteen minutes or so; well past the normal sixth-grade attention span.

Even worse, after another 10 minutes of my help we were no closer to a solution. That's when I decided it was time for a brute force approach. I remembered that the standard C++ library had a handy function, **`next_permutation()`**, that would let me iterate through all the possible arrangements of the figure with just a couple of lines of code. All I had to do was check the five different sums for each permutation and I'd have the answer in no time.

The resulting program is shown in Listing 1, and its output is given below:

```
100706 : 3 5 9 8 2 1 6 4 7
114154 : 3 8 6 5 2 4 9 1 7
246489 : 7 1 9 4 2 5 6 8 3
259937 : 7 4 6 1 2 8 9 5 3
362880 permutations were tested
```

A little quick sketching will show you that the four solutions are simply rotations and mirror images of the one true solution. You can also see that randomly putting down numbers makes the odds almost 100,000:1 against finding a solution. Not quite as bad as the lottery, but it clearly shows that random guessing isn't going to work. (My daughter asked me to give her the center position only, upon which she solved the rest of it in roughly 30 seconds.)

PLAIN TEXT

C++:

```
#include <iostream>
#include <algorithm>
using namespace std;
```

```
main()
{
    int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    int count = 0;
    do
    {
        if ( ( a[0] + a[1] + a[2] ) == 17 &&
             ( a[0] + a[3] + a[6] ) == 17 &&
             ( a[2] + a[5] + a[8] ) == 17 &&
             ( a[2] + a[4] + a[6] ) == 17 &&
             ( a[6] + a[7] + a[8] ) == 17 )
        {
            cout <<count << " : ";
            for ( int i = 0 ; i <9 ; i++ )
                cout <<a[ i ] <<" ";
            cout <<"\n";
        }
        count++;
    } while ( next_permutation( a, a + 9 ) );
    cout <<count <<" permutations were tested\n";
    return 0;
}
```

Listing 1 - Magic.cpp

Magic Permutations

From this program you can see that `next_permutation()` is a handy function to have in the C++ library. In my case it meant the difference between writing an impulse program versus fiddling around with pencil and paper for another hour. What really makes `next_permuation()` interesting to me is the fact that it can generate permutations without keeping any additional information beyond the sequence being juggled. I can generate a permutation, go off and do whatever I like with it, even write the numbers out to a file and save them for later. Regardless of what I do, `next_permuation()` will always be happy to generate the next set in the series given only the previous one as input.

Just writing a function to generate permutations isn't particularly hard. One easy way to tackle the problem is with a recursive approach. If the string you want to permute is *n* characters long, you execute a loop that makes one pass per character in the string. Each time through the loop you remove character *i* from the string, and keep it as a prefix. You then print out all the permutations of the remaining substring concatenated with the prefix. How do you get the list of permutations of the substring? By recursively calling the permutation function. The only additional piece of logic you need to include is the test to see if a substring is only one character long. If it is, you don't need to call the permutation function, because you already have the only permutation of the string.

For example, to print the permutations of "abc", you will first strip off the "a" character, and then get the permutations of "bc". To get those permutations, you will first strip off the "b" character, and get a resulting permutation list of "c". You then strip off the "c" character, and get a resulting permutation of "b". The results when combined with the prefix character of "a" give strings "abc" and "acb".

You then repeat the process for prefix "b" and substring "ac", then for prefix "c" and substring "ab".

Using the `string` class in the C++ standard library makes it fairly easy to implement this logic. Listing 2 shows `permute.cpp` which implements this algorithm relatively faithfully.

PLAIN TEXT

C++:

```
#include <iostream>
#include <string>
using namespace std;

void permute( string prefix, string s )
{
    if ( s.size() <= 1  )
        cout <<prefix <<s <<"\n";
    else
        for ( char *p = s.begin(); p <s.end(); p++ )
        {
            char c = *p;
            s.erase( p );
            permute( prefix + c, s );
            s.insert( p, c );
        }
}

main()
{
    permute( "", "12345" );
    return 0;
}
```

Listing 2 - Permute.cpp

This approach to generating permutations is okay, but its recursive nature makes it unattractive for use in a library. To use this in a library we would have to employ a function pointer that would be invoked from deep inside the chain of function calls. That would work, but it's definitely not the nicest way to do it.

`next_permutation()` manages to avoid this trouble by using a simple algorithm that can sequentially generate all the permutations of a sequence (in the same order as the algorithm I described above) without maintaining any internal state information. The first time I saw this code was in the original STL published by Alexander Stepanov and Ming Lee at Hewlett-Packard. The original code is shown in Listing 3.

PLAIN TEXT

C++:

```
template <class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last) {
    if (first == last) return false;
    BidirectionalIterator i = first;
    ++i;
    if (i == last) return false;
    i = last;
    --i;

    for(;;) {
        BidirectionalIterator ii = i--;
        if (*i <*ii) {
            BidirectionalIterator j = last;
            while (!(*i <*--j));
            iter_swap(i, j);
            reverse(ii, last);
            return true;
```

```
        }
        if (i == first) {
            reverse(first, last);
            return false;
        }
    }
}
```

Listing 3 - Function *next_permutation()* from the STL

Using this function is simple. You call it repetitively, asking it to permute a given sequence. If you start with a sequence in ascending order, `next_permutation()` will work its way through all possible permutations of the sequence, eventually returning a value of false when there are no more permutations left.

Internals of next_permutation()

You don't need to be an STL expert to understand this code, but if you've never been exposed to this new part of the C++ standard library, there are a few things you need to know.

First, iterators (and the `BidirectionalIterator` type used here) are an STL abstraction of pointers. When looking at this code you can mentally think of the iterators as pointers. The permutation sequence is defined by iterators `first` and `last`. `first` points to the first element in the sequence, while `last` points one past the last element.

The code shown in Listing 3 also uses two other STL functions. `iter_swap()` swaps the values ponted to by its two arguments. And `reverse()` simply reverses the sequence defined by its two arguments. By convention of course, the first argument points to the start of the sequence to be reversed, and the last argument points one past the end of the sequence.

To help illustrate the workings of this algorithm, I've included a listing of a permutation sequence in Figure 2. It contains all 120 permutations of a five digit sequence. With that output example, plus Listing 3, it is fairly easy to see how this code works.

The function first does a cursory check for sequences of length 0 or 1, and returns `false` if it finds either. Naturally, sequences of those lengths only have one permutation, so they must always return false.

After passing those tests, the algorithm goes into a search loop. It starts at the end of the sequence and works its way towards the front, looking for two consecutive members of the sequence where member n is less than member n+1. These members are pointed to by iterators `i` and `ii` respectively. If it doesn't find two values that pass this test, it means all permutations have been generated. You can see this is the case in Figure 2 for the very last value, '54321'.

```
12345 12354 12435 12453 12534 12543 13245 13254 13425 13452
13524 13542 14235 14253 14325 14352 14523 14532 15234 15243
15324 15342 15423 15432 21345 21354 21435 21453 21534 21543
23145 23154 23415 23451 23514 23541 24135 24153 24315 24351
24513 24531 25134 25143 25314 25341 25413 25431 31245 31254
31425 31452 31524 31542 32145 32154 32415 32451 32514 32541
34125 34152 34215 34251 34512 34521 35124 35142 35214 35241
35412 35421 41235 41253 41325 41352 41523 41532 42135 42153
42315 42351 42513 42531 43125 43152 43215 43251 43512 43521
45123 45132 45213 45231 45312 45321 51234 51243 51324 51342
```

```
51423 51432 52134 52143 52314 52341 52413 52431 53124 53142
53214 53241 53412 53421 54123 54132 54213 54231 54312 54321
```

Figure 2 - A sequence generated by next_permutation()

Once iterators `i` and `ii` have been properly located, there are still a few more steps left. The next step is to again start searching from the end of the sequence for the first member that is greater than or equal to the member pointed to by `i`. Because of the previous search for `i` and `ii`, we know that at worst the search will end at `ii`, but it might end earlier. Once this member is located, it is pointed to by iterator `j`.

Once these three iterators are located, there are only two more simple steps. First, a call is made to **`iter_swap( i, j )`**. This simply swaps the members pointed to by `i` and `j`. Finally, a call is made to **`reverse( ii, last )`**. This has the effect of reversing the sequence that starts at `ii` and ends at the end of the sequence.

The end result is a routine that is short, simple, and runs in linear time. You really can't ask for much more than that.

Walk through an example

For a quick look at the algorithm in action, consider what happens when you call **`next_permutation( "23541" )`**. After passing through the initial size tests, the algorithm will search for suitable values for iterators *i* and *ii*. (Remember that you are searching from the end of the sequence for the first adjacent pair where the value pointed to by *i* is less than the value pointed to by `ii`, and `i` is one less than `ii`.) The first pair of values that meet the test are seen when `i` points to 3 and `ii` points to 5. After that, a second search starts from the end for the first value of `j` where `j` points to a greater value than that pointed to by `i`. This is seen when `j` points to 4.

Once the three iterators are set, there are only two tasks left to perform. The first is to call **`iter_swap( i, j )`**, which swaps the values pointed to by the iterators `i` and `j`. After you do this, you are left with the modified sequence "24531". The last step is to call **`reverse( ii, last )`**, which reverses the sequence starting at `ii` and finishing at the end of the sequence. This yields "24135". Examining Figure 2 shows that the result demonstrated here does agree with the output of the program.

An additional charming attribute

The algorithm shown here has one additional feature that is quite useful. It properly generates permutations when some of the members of the input sequence have identical values. For example, when I generate all the permutations of "ABCDE", I will get 120 unique character sequences. But when I generate all the permutations of "AAABB", I only get 10. This is because there are 6 different identical permutations of "AAA", and 2 identical permutations of "BB".

When I run this input set through a set of calls to **`next_permutation()`**, I see the correct output:

```
AAABB AABAB AABBA ABAAB ABABA ABBAA BAAAB BAABA BABAA BBAAA
```

This might have you scratching your head a bit. How does the algorithm know that there are 6 identical permutations of "AAA"? The recursive implementation of a permutation generator I showed in Listing 2 treats the permutations of "AAABB" just as it does "ABCDE", obligingly printing out 120 different sequences. It doesn't know or care that there are a huge number of identical permutations in the output sequence.

It's easy to see why the brute force code in Listing 2 doesn't notice the duplicates. It never pays any attention to the contents of the string that it is permuting. It couldn't possibly notice that there were duplicates. It just merrily swaps characters without paying any attention to their value.

The STL algorithm, on the other hand, actually performs comparisons of the elements that it is interchanging, and uses their relative values to determine what interchanging will be done. In the example from the last section, you saw that an input of "24531" will generate a next permutation of "24135". What if the string had a pair of duplicates, as in "24431"? If the algorithm were ignorant of character values, the next permutation would undoubtedly be "24134".

In the early case, iterators **`i`** and **`ii`** were initially set to offsets of 1 and 2 within the string. But in this case, since the value pointed to by **`i`** must be less than the value pointed to by **`ii`**, the two iterators have to be decremented to positions 0 and 1. **`j`** would again point to position 3.

The subsequent swap operation yields "34421", and the reverse function produces a final result of "31244". Remember that the algorithm works by progressively bubbling the larger values of the string into position 0, you can see that this permutation has already jumped well ahead of the permutation of "24531" on its way to completion. Thus, the algorithm "knows" how to deal with duplicate values.

Conclusion

The addition of the STL to the C++ Standard Library gave us a nice grab bag of functions that automate many routine tasks. **`next_permuation()`** turned out to be just what I needed to solve a sixth grade math problem. It might be time for you to look through the declarations in the `algorithm` header file to see what else standards committee laid on our doorstep.

41 users commented in " C++ Algorithms: next_permutation() "

Follow-up comment rss or Leave a Trackback

on July 3rd, 2006 at 11:35 am, Mark said:

Thomas Draper wrote me a while back and offered up some similar code that implements next_combination(). I've posted a copy here. I'll ask him to post a follow-up comment to explain in his own words a little mroe about it.

on March 15th, 2007 at 10:07 am, Jan Simonffy said:

I am looking any program /at internet/ for search of all possible permutations of numbers /for examle - for number 84, or 45,.../.
Send me, pleace some informations about this program. Thank you.
With sincerely : simonffy

on March 15th, 2007 at 10:58 am, Mark said:

I don't know what to send you - I kind of tried to put all the information in this article. what else do you need?

on July 25th, 2007 at 6:43 am, Paul said:

i need a c program / example / algorithm where it will display all permutations of a set number of letters a specified number of times, eg

the set letters are: char myletters[3] = {a,b,c}
specified number of times is: int mysize = 2

i need it to output something like this :

{aa}, {ba}, {ca} {ab} {bb} {cb} {ac} {bc} {cc}

this is kinda similar to what you've shown here but i need it to allow duplicates, eg {aa}

this sould always produce (sizeof(myletters))^mysize combinations, in this example it would be 3^2=9 combinations.

do you know how i could implement this? i would be very grateful if you could help. thank you.

on July 25th, 2007 at 11:21 am, [Mark]() said:

Hi Paul,

Instead of next_permutation(), you need next_combination(), which unfortunately is not part of the standard library. However, this article comes up with a pretty good implementation of it:

http://www.codeguru.com/cpp/cpp/algorithms/combinations/article.php/c5117/

As for your requirement of needing multiples, I guess you could just create dupes of your data set, so if you are taking groups of three, have your input vector look like this:

"ABCABCABC"

on July 25th, 2007 at 3:05 pm, Paul said:

Thank you for your help and quick response. I've managed to get it done now. :o)

on August 14th, 2007 at 4:53 pm, goten said:

hi, im sorry, but my english is not so good, and i can't understand how can i get the next permutation, not if there is a permutation, that is what the next_permutation() function does.

for example:

```
#include
#include
```

```
using namespace std;

int main(){
    int numbers[]= {1,2,3,4,5},
    long n=0;
    while(next_permutation(numbers, numbers + 5)){
/*
what should i suposed to put in here to generate the permutation,
i think there is a function, but im new on C  , i thinked that
next_permutation() generated permutation, but i have seen that it
function return false or true if there is a permutation, just next the one i have

*/
}
 return 0;
}
```

sorry if i have not get the article in a right way u.u.

on August 14th, 2007 at 5:50 pm, Mark said:

>i can't understand how can i get the next permutation

I did my best job describing exactly that in the article and source code, so either you haven't read it or I failed. Let's assume I failed.

when you call next_permutation(numbers, number 5), the actual array is permuted - the contents of the array will now contain the next permutation of the five numbers.

When next_permutation() returns false, it means you have generated the last permutation - the next time you call it, you will be back at the start, which is the state where all the values in the input array are in sorted order.

Hope that answers your question.

on August 15th, 2007 at 8:01 am, goten said:

thanks, i just felt like a stupid xD, because i wrote wrong a variable in a program that i was making, and that was the cause of i can't to see the permutation of the array =S. im sorry for my fault, but thanks because i got what's the way that next_permutation() works.

thanks again =D

on August 26th, 2007 at 11:58 am, jonathan said:

I'm trying to permute a char array with using next_permutation. std::next_permutation works just fine, but if I copy-paste the Listing 3 next_permutation code into my source and try to use it, it only keeps swapping the two last characters in the array?!

I'd need to keep a few other arrays updated while permuting, and I'd like to add a few lines into the next_permutation code.

Any ideas why the Listing 3 code might not work?

on August 26th, 2007 at 12:31 pm, [Mark]() said:

Jonathan,

The code in Listing 3 is a sample from a specific version of the STL. Since your compiler ships with an existing set of STL algorithms, you don't want to use Listing 3 - there may be incompatibilities, and there will certainly be naming conflicts. Just use std::next_permutation.

Is there some reason why you don't want to?

If you want to modify next_permutation (not a good idea IMO) you can copy the implementation from your compiler's library, rename it, and make our own.

on August 26th, 2007 at 12:50 pm, [Mark]() said:

Ishtiaq,

Listing 2 shows you exactly what to do. Be sure to read the article.

on August 27th, 2007 at 9:50 am, jonathan said:

The reason I wanted to meddle with the next_permutation code was that I'm working on something where the permutating speed is essential. I need to update the new positions of the elements that have moved since the last permutation (rows and cols in a matrix).

I figured the best place to do that is inside the next_permutation code right where the elements are swapped and/or reversed. It keeps me from having to go through the whole permuted array to see which elements have moved.

Looks like the implementation of next_permutation in my compilers (djgpp) library uses only a single "

on August 27th, 2007 at 9:53 am, jonathan said:

..uses only a single "

on August 27th, 2007 at 9:56 am, jonathan said:

Oh right, can't post a "<".

...uses only a single "<" to compare the iterators, instead of "<<" like in Listing 3.

Thanks for the fast reply and a great blog. :)

on August 27th, 2007 at 10:58 am, Mark said:

I think you've identified a bug in the listing, probably caused by the same HTML problems you were having in your comment :-)

I'm going to fix it.

on August 30th, 2007 at 9:52 am, bob hofacker said:

Where is &lt defined

(s.size() <= 1)

on August 30th, 2007 at 10:17 am, Mark said:

ah bob, you know how the wordpress editor likes to play tricks with your escaped html codes... this one is fixed, at least.

on September 17th, 2007 at 7:48 am, Gokul said:

Nice article, Mark.
But I was just wondering if you have come across any algorithms that list permutations such that the mirror images of strings already listed are not listed again. As in, while permuting 1234, my list must only include : 1234,1243,1324,1342,1423,1432,2134,2143,2413,3124 & 3214.

on September 17th, 2007 at 8:06 am, Mark said:

@Gokul:

It would be pretty easy to do what you want this way.

create a hash table called forbidden_values, and only add a permutation to your output set if it doesn't appear in forbidden_values.

As each legal permutation is added to the output set, add its mirror image to forbidden_values.

Sounds like a homework problem.

on September 17th, 2007 at 1:32 pm, Gokul said:

Yes Mark, I know I can do that. But here I am talking about permuting about 25 objects(that's the size of the problem I am dealing with). Hence, storing the permutations is pretty much ruled out.I need something exactly like the next_permutation() which generates all the possible permutations, keeping out the mirror images(n!/2 for the case of n objects) with as minimal memory requirements as possible.

on September 17th, 2007 at 1:53 pm, Mark said:

@Gokul:

I don't understand why storing the permutations is ruled out. That doesn't make any sense. You could store 25 easily, you could store 25,000 easily.

In any case, I'm sorry, but I don't know of a specific algorithm for dealing with this.

Good luck with your homework.

on September 23rd, 2007 at 2:41 pm, Chris said:

@Mark:

I think Gokul is talking about permutations of 25 objects, of which there are 25!, so storing them is out of the question.

@Gokul:

next_permutation() lists permutations in order: 1234

on September 23rd, 2007 at 2:44 pm, Chris said:

(darn, the less-than bug got me as well)

@Mark:

I think Gokul is talking about permutations of 25 objects, of which there are 25!, so storing them is out of the question.

@Gokul:

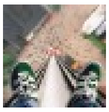
next_permutation() lists permutations in order: 1234 is smaller than 1243 (both interpreted as numbers), so 1234 is listed before 1243. Therefore, all you have to do is use next_permutation() and before adding a permutation to your output set, make sure the reversed permutation (viewed as a number) is NOT smaller than the permutation itself. If it is, its "mirror image" must already be in the output set.

For this to work you'll have to define an ordering relation between *permutations* of your objects that's "compatible" with the relation defined between individual objects. One that does make it look like a numeral system.

on September 23rd, 2007 at 2:52 pm, Mark said:

@Chris:

Good thinking, thanks, I didn't understand the question. Easy solution!



on November 9th, 2007 at 6:58 pm, Andrés said:

Mark,

This article has been of invaluable usage for me. I'm using what you exposed here for a brute force alphametics solver :).

Thanks a lot for writing this.

Greetings!
Andrés

on January 18th, 2008 at 11:04 am, Hugo Mills said:

I just turned this article up looking for an explanation of the algorithm, which is just what I wanted.

However, I note that it's possible to solve the original problem by brute force with a pen and a (small) piece of paper in a few minutes:

Label the 9 positions as a, b, c, ..., i. Then a+b+c+d+e+f+g+h+i = 45 (the numbers 1-9). Note also that if you add up all of the lines, you get:

2a + b + 3c + d + e + f + 3g + h + 2i = 5*17 = 85

Subtract one from the other, and you get:

a + 2c + 2g + i = 40

Observe further that the figure is symmetrical in both diagonals, so a and i are interchangable, and c and g are interchangable. Finally note that a and i must either both be even or both be odd. With this information, you can fully enumerate all possible combinations of a, c, g, and i quite quickly:

PLAIN TEXT

CODE:

```
a i c+g c g
1 3 18  9 9
1 5 17  9 8
1 7 16  9 7
1 9 15  9 6
```

```
           8 7
3 5 16     9 7
3 7 15     9 6
           8 7
3 9 14     9 5
           8 6
5 7 14     9 5
           8 6
5 9 13     9 4
           8 5
           7 6
7 9 12     9 3
           8 4
           7 5
2 4 17     9 8
2 6 16     9 7
2 8 15     9 6
           8 7
4 6 15     9 6
           8 7
4 8 14     9 5
           8 6
6 8 13     9 4
           8 5
           7 6
```

Many of these can be ditched immediately as duplicating numbers. Of the rest, simply computing the value that d must take will eliminate most of the remainder. After that, the full square can be computed on the remaining few until one of the solutions is found.

on January 18th, 2008 at 12:22 pm, Mark said:

@Hugo:

To me the real trick on a constraint problem is finding a deterministic path to the answer. You got pretty close with a system of algebraic equations, and I'm sure you could have finished it off by adding some more equations and then doing some more simplification. Well done!

on February 4th, 2008 at 7:22 am, Mark said:

Sean offered up the following alternative algorithm:

PLAIN TEXT

C:

```c
#include <stdio.h>

#define N 20    // number of items
#define R 10    // number of choices

int next_combination(int c[])
{
    int i = 0;
    while ( i <R-1 && c[i+1]==c[i]+1) // for each bump
```

```
        c[i] = i++;                  // fall back
    return N - ++c[i];               // push forward and verify
}

void main()
{
    // create initial combination (0..R-1)
    int c[R];
    for (int i=0; i<R; i++)
        c[i] = i;

    // display all possible combinations
    do
    {
        for (int j=0; j<R; j++)
            printf("%d ", c[j]);
        printf("\n");
    }
    while (next_combination(c));
}
```

on June 22nd, 2008 at 6:25 am, xyzzy said:

Thanks for the article. It would be still better if you provide an explanation/proof as to why the algorithm works.i.e. why first sorting the array and running the algo on the array n! times(consider unique elements) gives all the possible combinations.

on July 26th, 2009 at 12:49 pm, [Izzy88]() said:

Thanks alot for the article, it's great. I am currently working on a class project (I'm a first year studying Multimedia) that requires us to write a permutation program in C++ iteratively as well as recursively. I like the next_permutation() built in function, but I think that I will learn more about it if I actually wrote my own functions. Do you have any algorithms that you think would help my cause?


on July 26th, 2009 at 12:55 pm, [Mark Nelson]() said:

@Izzy88:

I don't know where the algorithm came from, but you can pretty much figure out next_permutation by examining the source code. A recursive version of a permutation function is trivial, the iterative version is not quite as obvious and requires a little bit of work.

- Mark

on September 12th, 2009 at 11:13 pm, gaston770 said:

http://code.google.com/codejam/contest/dashboard?c=186264#s=p1

This Google Code Jam 2009 problem solves in less than 30 lines with next_permutation.. I love it :D, Great post dear Mark. ;)

PLAIN TEXT

C:

```
int main (void) {
    int T;
    string N;
    cin>> T;
    for (int count = 1; count <= T; count++) {
        cin>> N;
        if (!next_permutation (N.begin(), N.end())) {
            int xx = 0;
            while (N[xx++] == '0');
            swap (N[--xx],N[0]);
            N.insert (N.begin(),'0');
            swap (N[0], N[1]);
        }
        cout <<"Case #" <<count << ": "<<N <<endl;
    }
    return 0;
```

on April 8th, 2010 at 8:05 am, Sven Forstmann said:

Well, lets say you have 120 permutations and you want permutation number 60. Using STL you woule have to walk through all of them step by step - wont you ?

So I've written an implementation that gives you any permutation immideately, without recursion:

PLAIN TEXT

C:

```
#include <string>

int main(int,char**)
{
    std::string default_str = "12345";

    int perm=1, digits=default_str.size();
    for (int i=1;i<=digits;perm*=i++);
    for (int a=0;a<perm;a++)
    {
        std::string avail=default_str;

        for (int b=digits,div=perm;b>0; b--)
        {
            div/=b;
            int index = (a/div)%b;
            printf("%c", avail[index] );
            avail.erase(index,1) ;
        }
        printf("\n");
    }
    printf("permutations:%d\n",perm);
    while(1);
}
```

(c) Sven Forstmann

on October 29th, 2010 at 4:33 pm, [Quick Facts]() said:

You you could change the webpage subject title C++ Algorithms: next_permutation() to more catching for your content you create. I enjoyed the the writing still.

on August 13th, 2011 at 10:19 am, Thomas Nygreen said:

Nice article!

The original problem can be solved by first listing the seven possible combinations of numbers in the range 1–9 such that no numbers are repeated and the sum is 17:

PLAIN TEXT

CODE:

```
i:   {1, 7, 9}
ii:  {2, 6, 9}
iii: {2, 7, 8}
iv:  {3, 5, 9}
v:   {3, 6, 8}
vi:  {4, 5, 8}
vii: {4, 6, 7}
```

Exactly five of these must be included in the solution. Let's label the positions a–i, as Hugo did, and start with the diagonal c-e-g. Note that both c and g must occur in three different combinations, the candidates being 6, 7, 8 and 9. Using the fact that the figure is symmetric, let c < g. Then we have c = 6 or c = 7.

As there are no combinations containing both 5 and 6 or both 5 and 7, the only positions left for 5 are d and h. Again we use symmetry, and let d = 5. This in turn means a must be 3 or 4, and g must be 8 or 9.

There is only one combination containing 1, so 1 cannot be in any corner, and there are no combinations with both 8 and 9, so h and i cannot be 9. That leaves only e and h as possible positions for 1. Now if we were to let e = 1, we would get c = 7, g = 9, a = 3, and b would need to be 7, which is already taken by c. Therefore h = 1.

Then follows g = 9, i = 7, a = 3, c = 6, b = 8, f = 4 and e = 2.

 on August 14th, 2011 at 12:06 pm, [Mark Nelson]() said:

@Thomas:

Nice approach. On problems like this I have a tendency to go for the brute force first. A more elegant solution like yours is always more satisfying though.

- Mark

on September 17th, 2011 at 8:29 am, SPEEDY said:

I love to play with variants of this kind of problems.
I am thinking if the program is required not to produce duplicate output strings when there might be some repeated characters in the input string. For example, "good_dog" is fed into the program!

on November 14th, 2011 at 9:41 am, SPEEDY said:

1. The lazybone method: Just store all permutations generated in a map that is a hash table, and check for the first time generated or not by a hash or so called a dictionary !

2. If there's no handy hash, just generate an ordered list of numbers somewhat like an odometer with input chars ordered in ASCII is fine!

on November 16th, 2011 at 7:43 am, Refrences « Belbesy M. Adel said:

[...] Generator | next_permutation() | next_permutation() 2 | Algorithms Questions GA_googleAddAttr("AdOpt", "1"); GA_googleAddAttr("Origin", "other"); [...]

on December 26th, 2011 at 11:20 am, Prashant said:

Superb...I just loved it...:)
Was looking for a similar stuff for a few days. I have implemented a recursive version but for distinct values, was trying to get it adapted for repeated values. But this algorithm is really awesome...thanks...:)

Leave A Reply

You can insert source code in your comment without fear of too much mangling by tagging it to use the iG:Syntax Hiliter plugin. A typical usage of this plugin will look like this:

```
[c]
int main()
{
printf( "Hello, world!\n" );
return 1;
}
[/c]
```

Note that tags are enclosed in square brackets, not angle brackets. Tags currently supported by this plugin are: as (ActionScript), asp, c, cpp, csharp, css, delphi, html, java, js, mysql, perl, python, ruby, smarty, sql, vb, vbnet, xml, code (Generic).

If you post your comment and you aren't happy with the way it looks, I will do everything I can to edit it to your satisfaction.

Username (*required)

Email Address (*private)

Website (*optional)

Post My Comment

Links From Google

Popular Posts

- LZW Data Compression
- Fast String Searching With Suffix Trees
- Data Compression with the Burrows-Wheeler Transform
- Arithmetic Coding + Statistical Modeling = Data Compression
- C++ Algorithms: next_permutation()
- zlib - Looking the Gift Code in the Mouth
- Priority Queues and the STL
- Building the Convex Hull
- C++ Hash Table Memoization: Simplifying Dynamic Programming
- LZW Revisited
- Hash Functions for C++ Unordered Containers
- The Million Random Digit Challenge Revisited

Recent Comments

- **Ernst** in The Million Random Digit Challenge …
- **Ernst** in The Million Random Digit Challenge …
- **to do or learn …** in Fast String Searching With Suffix T…
- **Milcho** in The Million Random Digit Challenge …
- **Mark Nelson** in The Million Random Digit Challenge …
- **Milcho** in The Million Random Digit Challenge …
- **Tobbi** in The Million Random Digit Challenge …
- **lzw算法** in LZW Data Compression
- **Ernst** in The Million Random Digit Challenge …
- **Ernst** in The Million Random Digit Challenge …

Feeds

Main Feed
Main Comment Feed
This Article's Comment Feed

Categories

Security Cisco Puzzles VoIP Standards Uncategorized Hackery Humor Networking Serial Communications Work Culture Scams Linux Graphics Video Writing Audio Mathematics Snarkiness Web Articles People Business C/C++ Programming Complaining Computer Science Magazine Articles Data Compression

Recent Entries

- C++11: Range-based for and auto
- Visual Studio 11 and Modern C++
- C++ – Where's the Hate?
- Streambuf Iterators Are a Big Help
- I'm In the Money
- Mark's Travel Guide to New Zealand

- A Visit With Tim Bell
- Streams or Iterators?
- Automating Putty
- Sendmail on Linux – the Easy Way

My Books

- The Data Compression Book
- Serial Communications: A C++ Developer's Guide, 1st. ed.
- Serial Communications: A C++ Developer's Guide, 2nd ed.
- C++ Programmer's Guide to the Standard Template Library
- Developing Cisco IP Phone Services: A Cisco AVVID Solution

Archives

Select Month

©2002 Mark Nelson
Powered by WordPress | Talian designed by VA4Business, Virtual Assistance for Business who's blog can be found at Steve Arun's Virtual Marketing Blog