Combinations in C++, Part 2

Shao Voon Wong, 9 Apr 2009 CPOL

4.18 (29 votes)

Rate:

Introduce 4 new algorithms on finding combinations

- **Download source - 58.4 KB**
- **Download benchmark source - 61.75 KB**
- **Download benchmark application - 35.78 KB**

Introduction

There are 4 new algorithms present in this article. The first one is about speed optimization by using integers as array indexes (I'll explain that later). The second one is about concurrent programming (now with multi-core CPUs becoming mainstream in the near future, you can cut the time to find a lot of combinations by half and more). The third and fourth ones deal with finding combinations with repetitions.

In my articles, I place a strong emphasis to explain the techniques as clearly as possible. It is of utmost importance for you, the reader, to understand the explanations. Because once you understand them, you need not know how I do it; you can implement it yourself, even in your favourite computer language. You can implement it yourself, or in an iterative way? Recursive way? Your way? Heck! Your implementation could even be faster than mine, who knows?

Source Code Version

For the source code changes, all the classes and functions fall under the `stdcomb` namespace and they are given a name called "Combination Library" and a version (1.5.0), so that the users know which is the latest version, as there are a few copies of source code floating around the Internet. Always get the latest version, as it contains new features and/or bug-fixes. You can always be sure that CodeProject has the most up-to-date version.

Another Way of Finding the Combinations

Combination is the way of picking a different unique smaller sequence from a bigger sequence, without regard to the ordering (positions) of the elements (in the smaller sequence). This article teaches you how to find combinations. For every technique presented in this article, I will explain it as well as I can before going on to teach you how to use the source code.

The Notations Used in this Article

n : n is the larger sequence from which r sequence is picked

r : r is the smaller sequence picked from n sequence.

c : c is the formula for the total number of possible combinations of r picked from n distinct objects : n! / (r! (n-r)!)

The ! postfix means factorial.

The Technique for Finding Combinations Without Repetitions

I added this section because I realized the explanation in the original article is not easy for the readers to visualise the underlying pattern. Let me explain the new approach now: the shifting technique.

To find combinations, we will use a technique which involves shifting the last combination element from the left to the right.

Find Combinations of 1 from a Sequence of 5

Let us first find combinations of 1 from a sequence of 5 numbers {0,1,2,3,4}. Please note that the red box is the combination element.

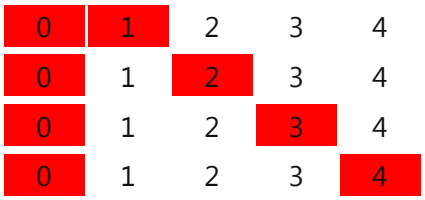
0	1	2	3	4
0	**1**	2	3	4
0	1	**2**	3	4
0	1	2	**3**	4
0	1	2	3	**4**

Total 5 combinations are generated.

```
0
1
2
3
4
```

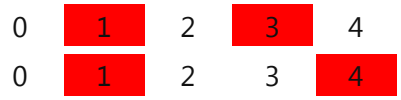
Find Combinations of 2 from a Sequence of 5

For the first example, it is pretty easy, isn't it. Let us go on to find combinations of 2 from a sequence of 5 numbers {0,1,2,3,4}. Please note the red boxes are the combination elements.



Oops, we can't shift the last element anymore. Now we shift the first element and bring back the last element to its right side. Shown below.

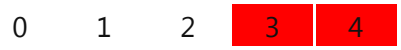
For the next 2 combinations, we continue to shift the last element.



Again, we cannot shift anymore and shift the first element and bring the last element to its side.

Shift last element as usual.

Shift the first element and bring the last to its side.



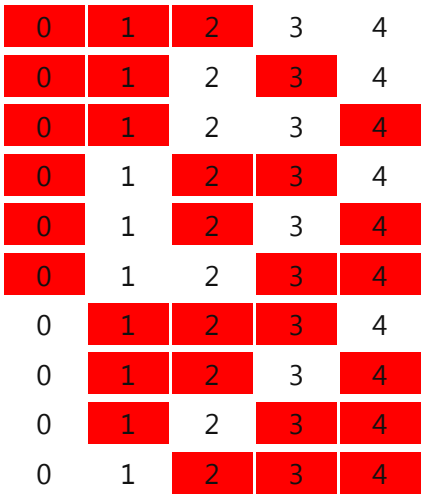
Oops, we can shift neither the first nor last element anymore. This is the end of all the combinations generated.

Total 10 combinations are generated.

```
01
02
03
04
12
13
14
23
24
34
```

Find Combinations of 3 from a Sequence of 5

Let us go on to find combinations of 3 from a sequence of 5 numbers {0,1,2,3,4}. Please note the red boxes are the combination elements. But for this, I will not explain. Observe the pattern.



Total 10 combinations are generated.

```
012
013
014
023
024
034
123
124
134
```

```
234
```

Steps to Find Combinations

Of course, I can go on to demonstrate finding combinations of 4 elements out of 5 and so on. But I don't. I trust that you can extrapolate the technique for finding different combinations from different sequences.

Let us define the steps for the shifting pattern we used to find all combinations.

- Initially, All the element(s) must be on the leftmost of the array.
- Shift the last red box until it cannot shift anymore, shift the next rightmost red box (if there is one) and bring back the last element to the right side of it. This is the new combination.
- Continue to shift the last element to the right.
- It comes to a point where the last 2 elements cannot shift any longer, then shift the 3^{rd} one (from right) and bring the last 2 to its side consecutively.
- Previous point rephrased for all abitrary cases - ('i' here stands for number). It comes to a point where the last i elements cannot shift any longer, then shift the i+1 element (from right) and bring the last i elements to its side consecutively.
- Continue until all the elements cannot shift any longer, all the combinations have been generated.

Optimised Version: Index Combination

Is there any way we can optimise the technique used in `next_combination()` function? The answer is yes but we have to give up the genericness of the `next_combination()`.

Now, the `next_combination()` compares the element, which it is going to shift, with other elements in the n sequence to know its current position in the n sequence. Let us see if we can get rid of this finding operation.

Take a look at this combination (Take 2 out of 5):

```
01234
 x x
```

If the 2 elements in the r sequence are integers which store its index position in n sequence, then the finding is not needed. And with this method, the combination returned is a combination of the indexes in the n sequence, is no longer a combination of objects. The timing of finding combinations from a sequence of any custom objects is the same, O(1), because integer index is used.

The == operator is not needed to be defined for this method to work; next_combination needs it to be defined unless you used the predicate version. Other requirements remain the same. Anyway, finding combinations of objects is actually the same as finding combinations of integers. And integer variables should be faster than class objects because of the object overhead. So we should use this!

This technique is implemented in the `CIdxComb` class. Here is an example on how to use the `CIdxComb` class.

```
#include <iostream>
#include <string>
#include <vector>
#include "IndexCombination.h"

using namespace std;
using namespace stdcomb;
void foo()
{
   CIdxComb cb;
   cb.SetSizes(5,3);
   // n sequence
   vector<string> vsAnimal;
   vsAnimal.push_back( "Elephant" );
   vsAnimal.push_back( "Deer" );
   vsAnimal.push_back( "Cat" );
   vsAnimal.push_back( "Bear" );
   vsAnimal.push_back( "Ape" );
   // r sequence
   vector<unsigned int> vi(3);
   vi[0] = 0;
   vi[1] = 1;
   vi[2] = 2;

   cout<< vsAnimal[ vi[0] ] << " "
      << vsAnimal[ vi[1] ] << " "
      << vsAnimal[ vi[2] ] << "\n";

   int Total = 1;
   while ( cb.GetNextComb( vi ) )
   {
      // Do whatever processing you want
      // ...

      cout<< vsAnimal[ vi[0] ] << " "
```

```
        << vsAnimal[ vi[1] ] << " "
        << vsAnimal[ vi[2] ] << endl;

      ++Total;
   }

   cout<< "\nTotal : " << Total << endl;
}
```

Benchmark Between next_combination() and CIdxComb

Below is a benchmark result of `next_combination()` and `CIdxComb` for finding all the combinations of r sequence of 6 from n sequence of 45, 10 times,
using `QueryPerformanceCounter()`, on a Intel P4 2.66Ghz CPU and 1GB of DDR2 RAM PC.

`next_combination` - 1925.76 milliseconds
`CIdxComb` - 149.608 milliseconds

This benchmark program is included in the source!

There is a very strange occurrance that `CIdxComb` runs as slow or slower than `next_combination` in some benchmarks in debug build. I don't know why. But I think this difference is due to how iterators (which`next_combination()` used) and array subscript index (which `CIdxComb` used) are handled in `std::vector`. The above result is from a release build without any optimization.

A Helper Algorithm for Concurrency

Let us consider a case that you need to generate lots of combinations, say 100 billion. You have a dual-core CPU PC, so naturally you would want to maximize the dual core by having some concurrent processing (mult-threading). You can do that, if you can tell the 1^{st} thread to calculate combinations starting from the 1^{st}combination and the 2^{nd} thread start from 50 billionth combination number. Each thread will find 50 billion combinations.

The problem is you do not know what combination is at 50 billionth! Let's say you do not have a dual-core CPU PC or SMP PC, you have 4 computers, so how about this: you could tell the 1^{st} computer to calculate from the $1^{st}25^{th}$ billion, the 2^{nd} computer the 2^{nd} 25^{th} billion and so on. The missing link here is a way to calculate a combination based on a position in the total number of combinations so

that `CIdxComb` or`next_combination()` can continue on finding combination starting from that one.

Fortunately, your hero here, which is me, has figured out a way to do this. The easiest way to explain this method is to walk though a simple example with me.

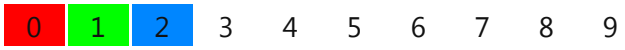
The Technique of Finding Combination, Given its Index

Let us find out the 92nd combination of 3 out of sequence of 10. Take note in zero-based counting, it would be number 91. In total, there are 120 combinations.

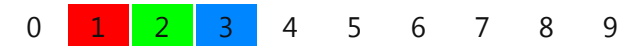
Below are the positions of the combinations which are placed side by side. I am going to show you how I derive the positions later, so right now you take it that they are correct. 1st element is represented by the red box, 2ndthe green box and 3rd the blue box.

Let us find the position of the red box first.

0th



36th



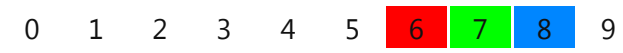
64th

85th

100th

110th

116th

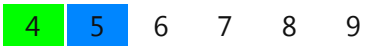


119^{th}

Since the one we are going to find is 91^{st}, it falls between 85^{th} and 100^{th}, so the 1^{st} element of the 91^{st}combination is 3.

To find the 2^{nd} element (green box), we must subtract 85 from 91 and the result is 6^{th} combination.

0^{th}



5^{th}

9^{th}

12^{th}

14^{th}

Since 6^{th} falls between 5^{th} and 9^{th}, the second element is 5.

Before we find the 3^{rd} element, we must subtract 5 from 6 whose result is 1^{st} combination.

0^{th}

1^{st}

2^{nd}

3rd

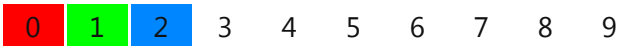
From the 1st combination above, we can see that the 3rd element is 7.

Now that all the 3 elements have been found, we have the 91st combination shown below

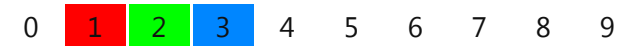
91th

Let us get back to the problem of knowing the position, given a combination whose elements are side by side.

0th



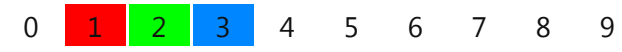
36th



How do I know the second combination shown above is number 36th? Take a look below:

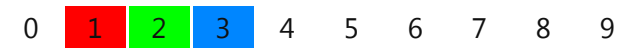
0th

36th



The combination is 36th, because the total number of combinations of 2 out of 9 is 36. Since this is zero-based, the next combination is the 36th. Let me give you another example!

36th



64th

The combination is 64th, because the total number of combinations of 2 out of 8 is 28 and add 28 to 36, you will get 64.

Combinadic

Combinadic is the method I have just described. You may want to read more about it in the Combinadic wikibecause you will find that my explanation and method are different from what is written there. There is a reason for this because I rediscovered this method myself and while writing this section, I found out about Combinadic. Rest assured that my method and Combinadic is actually the same method, merely a different way to get the answer.

A bit of trivia here: I was quite upset that I am not the first one who discovered this method till I stopped writing this article for a year. Well, my reason for inventing this algorithm was to find a method to split the work of finding combinations into different work loads so as to speed up the work on many processors. I found two algorithms, Combinadic and Factoradic, but Combinadic was the first one I found. Later, I found out the algorithm can be modified to find permutation, given its position. I also found out that I am not the first one who discovered Factoradic.

You may have noticed the `CFindCombByIdx` operates on numbers only and the contents of the first combination vector always start from zero and in running numbers. Unlike `next_combination()` which operates on objects by comparing, `CFindCombByIdx` operates on zero based running numbers. What you will get from its results are indexes. For example, you get a result of {0,2,1} out of {0,1,2,3,4}, it means the 1st element is index 0 of array (of objects) you originally want to find combination, 2nd element refers to index 2 of the array and 3rd element is index 1 of the array. Finding combinations of numbers is much easier and faster than finding objects.`CFindCombByIdx` uses a big integer class because this algorithm uses factorials to find the combination and factorials are usually very large numbers. Since it is a templated class and you know the factorials used are not going to exceeded 64 bit, you can use `__int64` type instead. The is the output of the above example code.

Example Code

```
CFindCombByIdx< CFindTotalComb<BigInteger>,
```

```
    BigInteger > findcomb;

const unsigned int nComb = 3;
const unsigned int nSet = 10;

// Intialize the vector with size nComb
vector<unsigned int> vec(nComb);

// vector returned by FindCombByIdx is the combination at 91.
findcomb.FindCombByIdx( nSet, nComb, 91, vec );
```

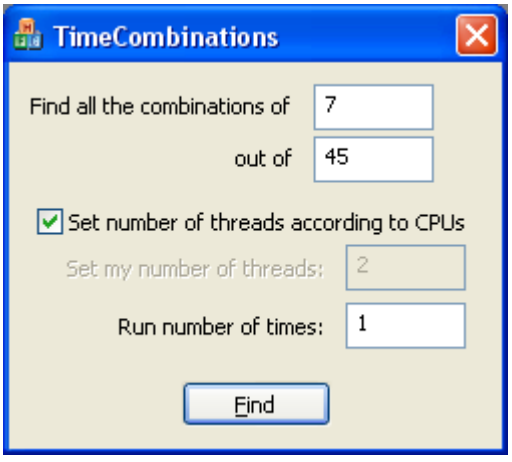
Below is the timing results of `next_combination`, `CIdxComb` and `CFindCombByIdx` finding combinations of 6 out of 20 elements which is 38760 combinations in total.

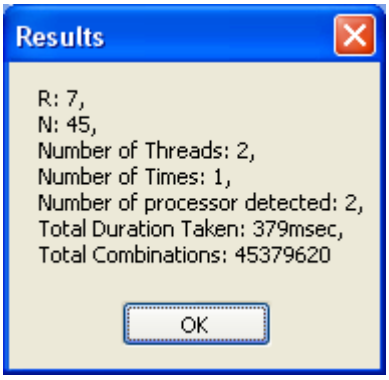
```
time taken for next_combination: 5.04 milliseconds
time taken for CIdxComb: 0.88 milliseconds
time taken for CFindCombByIdx:  5886.43 milliseconds
```

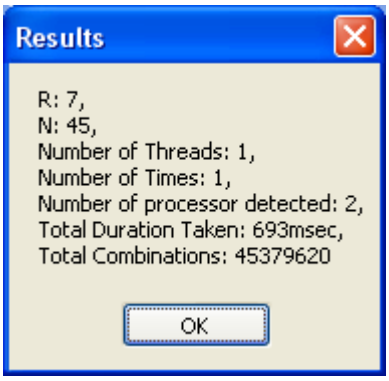
As you can see, `CFindCombByIdx` is very slow compared to the other 2 algorithms but we are only going to use`CFindCombByIdx` to find the n^{th} combination we want and use `CIdxComb` for finding the rest of the consecutive combinations.

A Benchmark Program for your PC

I have included a demo program called `TimeCombinations` for you to benchmark your PC. It is nice to know how much time has been reduced if you have a dual-core or quad-core PC. The program does not store the combination anywhere and discards them after every computation. If you have a uniprocessor PC, then you have not much use for this benchmark program. Below are the screenshots of the program.







Finding Combinations from a Set with Repeated Elements

Finding combinations from a set with repeated elements is almost the same as finding combinations from a set with no repeated elements: the shifting technique is used and the set needs to be sorted first before applying this technique. It is just that there are some special situations you need to take care of.

Let me used a example of finding combinations of 3 out of 8 {0,1,2,3,3,3,4,5}, the number 3 is repeated 3 times.

With this combination now, the green box is in the 4th column which is 3.

0 1 2 3 3 3 4 5

If you get this next combination, you need to shift the green box to the last 3, which is the 6th column, before returning this combination result so that the next combination would not be repeated and correct.

0 1 2 3 3 3 4 5

Another situation to take care of is as below.

If you use the combination shifting technique, the next combination would be as below:

You need to shift the green box and the blue box to the 5th and 6th column respectively so that the next combination would be correct.

There is all you need to know and take care of when finding combinations from a set with repeated elements.

Sample Code for Finding Combinations from a Set with Repeated Elements

Below is the sample. The name of the class used to find the combinations is called `CCombFromRepSet`. As the code is straightforward and commented, I shall not explain the working of the code.

```
#include <vector>
#include <iostream>
#include "CombFromRepSet.h"

using namespace std;
using namespace stdcomb;

int main()
{
   // Initialize the set
   vector<unsigned int> svi;
   svi.push_back(0); // 0
   svi.push_back(1); // 1
   svi.push_back(2); // 2
   svi.push_back(3); // 3
   svi.push_back(3); // 4
   svi.push_back(3); // 5
   svi.push_back(4); // 6
   svi.push_back(5); // 7

   // Object to find the combinations from set with repeated elements
```

```
   CCombFromRepSet cfrs;
   cfrs.SetRepeatSetInfo( svi );

   // Set the size of Set and number elements of the combination
   const unsigned int SET = 8;
   const unsigned int COMB = 3;
   cfrs.SetSizes( SET, COMB );

   // Initialize the first combination vector
   vector<unsigned int> vi;
   for( unsigned int j=0; j<COMB; ++j )
      vi.push_back( j );

   // Set the first combination
   cfrs.SetFirstComb( vi );

   // Display the first combination
   int Cnt=0;
   {
      cout<<Cnt<<")";
      ++Cnt;
      for( unsigned int i=0; i<vi.size(); ++i)
      {
         cout<<svi[vi[i]]<<",";
      }
      cout<<endl;
   }

   // Find and display the subsequent combinations
   while( cfrs.GetNextComb( vi ) )
   {
      cout<<Cnt<<")";
      ++Cnt;
      for( unsigned int i=0; i<vi.size(); ++i)
      {
         cout<<svi[vi[i]]<<",";
      }
      cout<<endl;
   }

   system( "pause" );

   return 0;
}
```

This is the output of the sample code:

```
0)0,1,2,
1)0,1,3,
2)0,1,4,
3)0,1,5,
4)0,2,3,
5)0,2,4,
6)0,2,5,
7)0,3,3,
8)0,3,4,
9)0,3,5,
10)0,4,5,
11)1,2,3,
12)1,2,4,
13)1,2,5,
14)1,3,3,
15)1,3,4,
16)1,3,5,
17)1,4,5,
18)2,3,3,
19)2,3,4,
20)2,3,5,
21)2,4,5,
22)3,3,3,
23)3,3,4,
24)3,3,5,
25)3,4,5,
```

Finding Repeated Combinations from a Set with No Repeated Elements

Now we have come to the last algorithm in the article: finding repeated combinations from a set with no repeated elements! Let me re-iterate again: Combination is the way of picking a different unique smaller sequence from a bigger sequence, without regard to the ordering (positions) of the elements (in the smaller sequence), meaning {0,0,1}, {0,1,0} and {1,0,0} are actually the same combination because they all contain one 1 and two zeros.

The formula for calculating the total number of repeated combinations: (n + r - 1)!/(r!.(n-1)!)

Let me explain, by finding combinations of 3 out 5 {0,1,2,3,4}:

```
0,0,0 -> the 1st combination
0,0,1
0,0,2
0,0,3
0,0,4
0,1,1 -> this combination is 0,1,1, not 0,1,0 because we already have 0,0,1.
0,1,2
0,1,3
0,1,4
0,2,2 -> this combination is 0,2,2, not 0,2,0 because we already have 0,0,2.
0,2,3
.
.
0,4,4
1,1,1 -> this combination is 1,1,1, not 1,0,0 because we already have 0,0,1.
1,1,2
1,1,3
1,1,4
1,2,2 -> this combination is 1,2,2, not 1,2,0 because we already have 0,1,2.
.
.
4,4,4 -> Last combination
```

As a rule of thumb, when we crossover to the next level in the leftmost column (as you may have noticed from the above explanation), the numbers in rightmost columns always follow the crossover column. For example, the next combination of 0,4,4 is 1,1,1, the first column crossover from 0 to 1, the second and third column follows the first column. Let's give you another example, the next combination of 0,0,4 is 0,1,1, the second column crossover from 0 to 1, third column follows the second column. Remember, when crossover, the minimum number in the rightmost columns after the crossover column follows the crossover column.

Example Code for Finding Combinations from a Set with Repeated Elements

Below is the sample. The name of the function used to find the combinations is called "`CombWithRep`". As the code is straightforward and commented, I shall not explain the working of the code.

```
#include <iostream>
```

```cpp
#include <vector>
#include <string>
#include "CombWithRep.h"

using namespace std;
using namespace stdcomb;

int main()
{
   const int SET = 5;
   const int COMB = 3;

   // Initialize the first combination vector to zeros
   std::vector<unsigned int> vi( COMB, 0 );

   // Display the first combination
   int Cnt=0;
   {
      cout<<Cnt<<")";
      ++Cnt;
      for( int j=0; j<COMB; ++j )
      {
         cout<< vi[j] << ",";
      }
      cout<<endl;
   }
   // Find and display the subsequent combinations
   while( CombWithRep( SET, COMB, vi ) )
   {
      cout<<Cnt<<")";
      for( int j=0; j<COMB; ++j )
      {
         cout<< vi[j] << ",";
      }
      cout<<endl;
      ++Cnt;
   }

   cout<<endl;

   system( "pause" );

   return 0;
}
```

This is the output of the example code:

```
0)0,0,0,
1)0,0,1,
2)0,0,2,
3)0,0,3,
4)0,0,4,
5)0,1,1,
6)0,1,2,
7)0,1,3,
8)0,1,4,
9)0,2,2,
10)0,2,3,
11)0,2,4,
12)0,3,3,
13)0,3,4,
14)0,4,4,
15)1,1,1,
16)1,1,2,
17)1,1,3,
18)1,1,4,
19)1,2,2,
20)1,2,3,
21)1,2,4,
22)1,3,3,
23)1,3,4,
24)1,4,4,
25)2,2,2,
26)2,2,3,
27)2,2,4,
28)2,3,3,
29)2,3,4,
30)2,4,4,
31)3,3,3,
32)3,3,4,
33)3,4,4,
34)4,4,4,
```

Conclusion

We have come to the end of the article. We have covered an optimized algorithm to find non-repeated combinations with integers, an algorithm to find non-repeated combinations, given their index so as to split the work into different work tasks for

different processors, an optimized algorithm to finding combinations from a set with repeated elements and lastly, finding repeated combinations. I hope you have found my article useful and easy to understand. I love to hear your feedback, good or bad so that I can improve my article! Thanks!

References

- Combinatorics
- Combinations
- Combinadic
- Generating the mth Lexicographical Element of a Mathematical Combination
- Combinations in C++

History

- 7 April 2009 - Updated benchmark source and application
- 12 March 2009 - Changed the source code to use C++ Big Integer Library written by Matt McCutchen, so as to remove the need to download and build Crypto++, and also to reduce download sizes.
- 17 Nov 2007 - First release

License

About the Author

Shao Voon Wong

Software Developer
Singapore