Bank Conflicts in Cache Tags

Alan H. Karp
Sidney Hantler (IBM)
HPL-94-102
November 1994

Cache, Bank conflict

We would like to know the sustained performance of a cache system with interleaved, multi-ported cache tags when it is presented with more address references than it has ports to handle. This report describes a program used to count the number of bank conflicts in the cache tags and an analysis of a formula for computing the probabilities directly.

1 Introduction

Because main memories are large and quite far from the processor, there are many ways to minimize the impact of conflicts in interleaved memories. For example, most such memories continue to accept requests while servicing a bank conflict; flow control turns off requests only when queues get too full. There have been numerous studies of the performance of interleaved memory systems on both RISC and CISC machines.

We are interested in a different problem, the performance of a cache with interleaved tags in a VLIW machine. Caches are much smaller and much closer to the processor. The only way to keep up with memory requests is to make things as simple as possible. In particular, we would like to have a cache that never has to delay the processor when it holds the requested data. In a VLIW machine, which can make more than one address request per cycle, we would like to provide one port to the cache tags for every address request. Unfortunately, we can't always have enough ports so we may have to deal with conflicts.

Simplicity dictates against having queues and general flow control, so we will assume that the cache tags stall the machine as soon as a conflict is detected. In particular, assume that the cache tags are interleaved B ways with P ports each and the memory unit will send A addresses on each cycle. If more than P but fewer than $2P$ addresses arrive at any bank, the machine will stall for a cycle; if more than $2P$ but fewer than $3P$ arrive, the machine will stall for 2 cycles, *etc.*

We will assume that the addresses fall into the banks at random. Of course, real programs don't address memory at random, but we can achieve the same effect by using a random hash when placing the data into the cache.

It is straightforward to enumerate all possible configurations of A addresses falling into B banks. This enumeration is not even very time consuming since we are dealing with relatively small numbers for typical caches. Current circuit densities force us to interleave a typical cache 4 to 16 ways. The most memory intensive operation normally supported on high performance machines is a SAXPY[2] which does 2 loads and 1 store per memory unit per cycle. Typical VLIW or super-scalar machines have between 1 and 4 memory units so that $3 \leq A \leq 12$.

The smallest case only involves some 2,048 possible configurations; the largest some 256 million. The smaller cases take only a few seconds on a modern workstation to enumerate, but the largest ones can take an excessive amount of time. Hence, we present here both the enumeration and a derivation of a simple formula to evaluate the number of conflicts.

First, we'll describe how to find the number of conflicts by enumeration. Next, we'll describe the analysis used to derive a general solution. Finally, we'll present the numbers showing the performance for various configurations. Appendices contain descriptions of the program that enumerates all cases for particular cache configurations and the program that implements the general solution.

2 Enumeration

There are two non-analytic ways to find how many references the busiest bank must handle for a given number of banks and addresses. The first is simulation by something like a Monte Carlo method which can only approximate the solution. The second approach is enumeration which can give exact counts of the possible cases. Enumeration is practical only when the number of cases to be considered is modest. For the parameters of interest, enumeration is fine.

One way to formulate the enumeration problem is using the "Stars and Bars" approach[1]. For B banks, we divide the number line into B segments which requires $B-1$ "Bars". We enumerate the cases by putting these bars down in all possible combinations with A "Stars".

If we treat the "Bars" as ones and the "Stars" as zeros, we see the enumeration consists of picking all those integers with a binary representation containing exactly $B-1$ ones and A zeros. The number of cases is quite manageable even for modestly large values of B and A; there are only about 54,000 such numbers with 15 ones and 6 zeros out of some 2,000,000 integers with 21 bits. The program that does the counts is described in Appendix A.

3 Analysis

Assume we have a cache with B banks of tags, each bank having P ports. Assume also that we simultaneously present this cache with A addresses. As long as $A \leq P$, there will never be a need to stall the machine. The interesting case is when $A > P$. If $P < A < 2P$, it will take up to 2 cycles to service the memory requests; up to 3 cycles if $2P < A < 3P$, *etc.* Hence, we need to know the number of times the busiest bank gets $1 \leq k \leq A$ requests. This section explains how we arrived at a formula for $f(B, A, k)$, the number of times the busiest B bank must service exactly k requests when presented with A random addresses.

First, look at $k = A$. There are exactly B ways to put these k references into the B banks; all A references fall into any of the B banks. If $k = A-1$, we can put all the addresses in the same bank, or $A-1$ addresses in one bank and the remaining one in another bank. The number of times this happens is $B(B-1)$. The first term comes from the ways that $A-1$ items can go into B slots; the second from the number of ways that 1 item can go into $B-1$ slots.

Things rapidly get more complicated. Basically, we end up enumerating the number of ways we can add up integers to make A. Fortunately, the next most complicated case leads us to a nice solution. When $k = A-3$, we can put the remaining 3 addresses into one of the remaining banks, 2 in one bank and the other in a third bank, or all 3 into different banks. In other words, having placed the k references into one of the B banks, we are left with the problem of placing $A-k$ reference in $B-1$ banks with no more than k references per bank. Since this second problem formulation is exactly the one we started with, we are let to a recursive formula.

Before presenting the equation, there is one more case that needs to be considered. So far we have been considering cases with $A < 2k$. If $A = mk$, we have one more situation to worry about. We can have exactly 1 bank with k references, two such banks, and so on up to m such banks. In each case, the remaining references are distributed among the remaining banks using an identical formulation. So, when we have j occurrences of k references to put into B banks, we have the "B choose j" problem, the solution to which is the binomial coefficient

$$\begin{pmatrix} B \\ j \end{pmatrix} = \frac{B!}{j!(B-j)!}. \tag{1}$$

The formula which gives the number of occurrences with the busiest of B banks having m references when the memory is presented with A simultaneous references is

$$f(B,A,m) = \sum_{k=1}^{\lfloor c/m \rfloor} \begin{pmatrix} B \\ k \end{pmatrix} \sum_{j=1}^{m-1} f(B-k, A-km, j), \tag{2}$$

with

$$\begin{array}{rcll} f(B,A,1) & = & \begin{pmatrix} B \\ A \end{pmatrix} & \forall B \geq A \geq 1 \\ & = & 0 & \forall B < A \\ f(B,0,1) & = & 1 & \forall B \geq 1 \\ f(B,0,m) & = & 0 & m > 1. \end{array} \tag{3}$$

The program that evaluates $f(B,A,m)$ is described in Appendix B. To see how this formula works, consider the case of $B = 8$, $A = 6$, and $m = 3$. We get

$$
\begin{aligned}
f(8,6,3) &= \sum_{k=1}^{2} \binom{8}{k} \sum_{j=1}^{2} f(8-k, 6-3k, j) \\
&= \binom{8}{1} [f(7,3,1) + f(7,3,2)] + \\
&\quad \binom{8}{2} [f(6,0,1) + f(6,0,2)] \\
f(7,3,2) &= \binom{7}{1} f(6,1,1) \\
f(8,6,3) &= \binom{8}{1} \binom{7}{3} + \binom{8}{1} \binom{7}{1} \binom{6}{1} + \binom{8}{2} \\
&= 644.
\end{aligned}
\tag{4}
$$

In other words, there are exactly 644 conflicts out of all possible ways of distributing 6 addresses to 8 banks. The total number of possible assignments is simply

$$
C(B,A) = \sum_{m=1}^{A} f(B,A,m). \tag{5}
$$

Since $C(8,6) = 1,716$, the busiest bank must handle 3 references 38% of the time.

4 Performance

We can now estimate the performance of our cache subsystem. The percentage of occurrences of the busiest of B banks having m references when presented with A addresses is

$$
p(B,A,m) = \frac{f(B,A,m)}{C(B,A)}. \tag{6}
$$

We can now compute the average time to respond to the A addresses when we have P ports as

$$
t(B,A) = \sum_{k=1}^{m} p(B,A,m) \left\lceil \frac{k}{P} \right\rceil \tag{7}
$$

Table 1 shows the results for 8, 16, and 32 banks presented with 6 addresses. Table 2 shows that this configuration handles 6 references in 1.58 cycles with 8 banks and in 1.24 cycles with 16 banks.

Table 1: Percent times busiest bank has m references when given 6 addresses.

m	$B = 8$	$B = 16$	$B = 32$
1	0.016317	0.147475	0.389796
2	0.440559	0.614035	0.528049
3	0.375291	0.198290	0.074887
4	0.130536	0.035383	0.006827
5	0.032634	0.004423	0.000427
6	0.004662	0.000295	0.000014

Table 2: Average number of cycles to service 6 addresses.

P	$B = 8$	$B = 16$	$B = 32$
1	2.736597	2.135928	1.700081
2	1.580420	1.243108	1.082595
3	1.167832	1.040100	1.007268
4	1.037296	1.004718	1.000440
5	1.004662	1.000295	1.000014

It's a good thing we derived Equation 2. Even the case with 16 banks takes almost a minute to enumerate on a modern workstation. Enumeration is totally impractical in the third case which has over 2,000,000 occurrences of 31 ones and 6 zeros.

5 References

[1] William Feller. *An Introduction to Probability Theory and its Applications*, volume 2 of *Wiley Mathematical Statistics Series*. John Wiley, New York, 1950.

[2] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5:308–329, 1979.

[3] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, London, 1992.

These appendices are `nuweb` descriptions of the two programs used for the calculations. The first is a simple enumeration scheme; the second embodies Equation 2.

A Enumeration Program

This section is a `nuweb` implemenation of the program that enumerates all cases for the given input parameters.

First, I'll put a brief explanation in the code.

"bankenum.c" 7a ≡

```
/*
   Enumerate number of bank conflicts - bankenum(b,s)

   b = number of banks
   s = number of simultaneous references
*/
◇
```

File defined by scraps 7abc, 8, 9abc, 10, 11, 12ab, 13, 14a.

I may want to see the bit patterns. A macro variable can be used to turn this printing on and off.

"bankenum.c" 7b ≡

```
#undef SEEBITS
◇
```

File defined by scraps 7abc, 8, 9abc, 10, 11, 12ab, 13, 14a.

Now, the standard start with paramenters coming in.

"bankenum.c" 7c ≡

```
main(argc,argv)
      int argc;
      char *argv[];
◇
```

File defined by scraps 7abc, 8, 9abc, 10, 11, 12ab, 13, 14a.

The program starts with some declarations.

Table 3: Variables used in bankenum.

Variable	Type	Definition
b	Input	Number of banks
s	Input	Number of addresses requested
i, j	Local	Loop counters
k	Local	Largest integer considered
n	Local	Remaining bits in integer
nb	Local	Number of bars
ns	Local	Number of stars
p	Local	Number of banks
ss	Local	Number of consecutive 0s
mss	Local	Max consecutive 0s for this i
t	Local	Low order bit of n
cases	Local	Number of integers with s 0s
total	Local	Number of terms in counts
counts	Local	Number of times get i 0s
countu	Local	Number of times get max i 0s
temp	Local	Provisional counts
pcts	Local	counts/total
pctu	Local	countu/cases
sum	Local	Used for running percentages
bits	Opt.	Bit patters for printing
time	Out	Ave. time per s refs

"bankenum.c" 8 ≡

```
{
  int b, s;              /* Input parameters */
  int c, d, i, j, k, n, nb, ns, p, ss, mss, t;
  int cases = 0, total = 0;
  int counts[20], countu[20], temp[20];
  float pcts[20], pctu[20], sum, time = 0;
  char bits[20];
◇
```

File defined by scraps 7abc, 8, 9abc, 10, 11, 12ab, 13, 14a.

OK, it's time to check the input parameters.

"bankenum.c" 9a ≡

```
/*
   Read and check parameters
*/
  if ( argc < 3 ) {
    printf("Not enough input parameters.\n");
    return 1;
  }
  if ( argc > 3 ) {
    printf("Too mainy input parameters.\n");
    return 2;
  }
  b = atoi(argv[1]);
  s = atoi(argv[2]);
  printf("%d banks, %d addresses\n",b,s);
◇
```

File defined by scraps 7abc, 8, 9abc, 10, 11, 12ab, 13, 14a.

I have two global counters. The first, `counts[i]`, counts the total number of times that exactly `i` addresses appear in a bank. The second, `countu[i]` counts the number of integers that have exactly `i` consecutive zeros, *i.e.* the number of cases with exactly `i` zeros in an interval.

"bankenum.c" 9b ≡

```
/*
   Initialize global counters
*/
  for ( i = 0; i < s; i++ ) counts[i] = countu[i] = 0;
◇
```

File defined by scraps 7abc, 8, 9abc, 10, 11, 12ab, 13, 14a.

This outer loop runs through all the integers between 0 and 2^{b+s-1}.

"bankenum.c" 9c ≡

```
/*
   For each index in range count stars and bars
*/
  k = 1<<(b-1+s);
  for ( i = 0; i < k; i++ ) {
◇
```

File defined by scraps 7abc, 8, 9abc, 10, 11, 12ab, 13, 14a.

The inner loop runs over bits in the current integer counting the number of ones and the number of zeros. It also keeps track of the number of consecutive zeros, recording the number of each in a temporary array. The array is temporary because we don't know if the counts are valid until we see that the integer has exactly $B-1$ bars and exactly A stars.

"bankenum.c" 10 $\equiv$

```
/*
    Initialize for this integer
*/
    for ( j = 0; j <= s; j++ ) temp[j] = 0;
    nb = ns  = ss = mss = 0;
    n = i + k;
◇
```

File defined by scraps 7abc, 8, 9abc, 10, 11, 12ab, 13, 14a.

This inner loop counts the number of ones, the number of zeros, and the number of consecutive zeros. The last count is held in the temporary array.

"bankenum.c" 11 ≡

```
/*
   Examine bits in this number one by one
*/
    for ( j = 0; j < (b+s); j++ ) {
      t = n & 1;
      bits[j] = t;
      if ( t == 1 ) nb ++;
      else          ns ++;
/*
   Stop after too many bars or stars
*/
      if ( nb <= b && ns <= s ) {
        if ( t == 1 ) {
          mss = (mss>ss) ? mss : ss;
          temp[ss]++;
          ss = 0;
        }
        else ss++;
      }
      else
        break;
      n = n >> 1;
    }
◇
```

File defined by scraps 7abc, 8, 9abc, 10, 11, 12ab, 13, 14a.

If the number of ones and the number of zeros are correct, I can record the results. I'll also print the bit patterns and individual counts if requested.

"bankenum.c" 12a ≡

```
/*
   Collect stats for valid cases
*/
    if ( nb <= b && ns <= s ) {
      cases++;
#ifdef SEEBITS
      for ( j = 1; j < (b+s); j++ )
        printf("%d",bits[b+s-1-j]);
#endif
      for ( j = 0; j <= s; j++) {
        if ( temp[j] > 0 ) d = j;
        counts[j] += temp[j];
#ifdef SEEBITS
        printf(" %d",temp[j]);
#endif
      }
      countu[d]++;
#ifdef SEEBITS
      printf(" %d %d %d %d\n",i,nb,ns,mss);
#endif
    }
  }
◇
```

File defined by scraps 7abc, 8, 9abc, 10, 11, 12ab, 13, 14a.

Now I'll print the counts. First the count of all occurrences, then the count of number of integers with having a specified count.

"bankenum.c" 12b ≡

```
  printf("Counts:\n");
  for ( i = 0; i <= s; i++ ) {
    total += counts[i];
    printf(" %8d",counts[i]);
  }
  printf("\n");
  for ( i = 0; i <= s; i++ )
    printf(" %8d",countu[i]);
◇
```

File defined by scraps 7abc, 8, 9abc, 10, 11, 12ab, 13, 14a.

The percentages of each occurrence are probably most interesting fact since I can calculate the performance of the memory system using this data. The running percentages show the fraction of occurrences that have the indicated maximum or below.

"bankenum.c" 13 ≡

```
/*
   Percentages
*/
  printf("\nPercentages:\n");
  for ( i = 0; i <= s; i++ ) {
    pcts[i] = ((float)counts[i])/((float)total);
    printf(" %8.6f",pcts[i]);
  }
  printf("\n");
  for ( i = 0; i <= s; i++ ) {
    pctu[i]=((float)countu[i])/((float)cases);
    printf(" %8.6f",pctu[i]);
  }
/*
   Running Percentages
*/
  printf("\nRunning Percentages:\n");
  sum = 0;
  for ( i = 0; i <= s; i++ ) {
    sum += pcts[i];
    printf(" %8.6f",sum);
  }
  printf("\n");
  sum = 0;
  for ( i = 0; i <= s; i++ ) {
    sum += pctu[i];
    printf(" %8.6f",sum);
  }
  printf("\n");
◇
```

File defined by scraps 7abc, 8, 9abc, 10, 11, 12ab, 13, 14a.

The average time to satisfy the incoming requests can be calculated from the number of ports on each bank, `p` in the inputs, and the percentages just printed. Note that this is the only place `p` is used.

"bankenum.c" 14a ≡

```
/*
   Timing
*/
  printf("Cycles per case for various numbers of banks for %d cases.\n",cases);
  for ( p = 1; p < s; p++ ) {
    time = 0;
    for ( i = 1; i <= s; i++ ) time += pctu[i]*((i+p-1)/p);
    printf("%d %f\n",p,time);
  }
}
◇
```

File defined by scraps 7abc, 8, 9abc, 10, 11, 12ab, 13, 14a.

B Recursion Program

This section is a nuweb implemenation of the program that evaluates the recursion formula in Equation 2.

First, I'll put a brief explanation in the code.

"bankeval.c" 14b ≡

```
/*
   Evaluate number of bank conflicts - bankeval(b,s)

   b = number of banks
   s = number of simultaneous references
*/
◇
```

File defined by scraps 14bc, 15, 16abc, 17, 18abc, 19ab, 20abc.

Now, the standard start with paramenters coming in.

"bankeval.c" 14c ≡

```
main(argc,argv)
     int argc;
     char *argv[];
◇
```

File defined by scraps 14bc, 15, 16abc, 17, 18abc, 19ab, 20abc.

The program starts with some declarations.

Table 4: Variables used by `bankeval`.

Variable	Type	Definition
`b`	Input	Number of banks
`s`	Input	Number of addresses requested
`cases`	Local	Number of integers with s 0s
`count`	Local	Number of times get max i 0s
`i, m`	Local	Loop counter
`p`	Local	Number of ports per bank
`sum`	Local	Used for running percentages
`pct`	Out	count/cases
`time`	Out	Ave. time per s refs

The program starts with some declarations.

"bankeval.c" 15 ≡

```
{
  int b, p, s;          /* Input parameters */
  int i, m;
  int cases = 0, fsumi = 0, fsumo = 0;
  int count[20];
  float pct[20], sum, time = 0;
◇
```

File defined by scraps 14bc, 15, 16abc, 17, 18abc, 19ab, 20abc.

OK, it's time to check the input parameters.

"bankeval.c" 16a ≡

```
/*
   Read and check parameters
*/
  if ( argc < 3 ) {
    printf("Not enough input parameters.\n");
    return 1;
  }
  if ( argc > 3 ) {
    printf("Too mainy input parameters.\n");
    return 2;
  }
  b = atoi(argv[1]);
  s = atoi(argv[2]);
  printf("%d banks, %d ports, %d addresses\n",b,p,s);
◇
```

File defined by scraps 14bc, 15, 16abc, 17, 18abc, 19ab, 20abc.

The count for each possible maximum is defined as a recursive function.

"bankeval.c" 16b ≡

```
/*
   The outer loop is over all possible counts per bank
*/
  cases = 0;
  for ( m = 1; m <= s; m++ ) {
    count[m] = f(b,s,m);
    cases += count[m];
  }
◇
```

File defined by scraps 14bc, 15, 16abc, 17, 18abc, 19ab, 20abc.

Now I'll print the counts. First the count of all occurrences, then the count of number of integers with having a specified count.

"bankeval.c" 16c ≡

```
  printf("Counts:\n");
  for ( i = 0; i <= s; i++ )
    printf(" %8d",count[i]);
◇
```

File defined by scraps 14bc, 15, 16abc, 17, 18abc, 19ab, 20abc.

The percentages of each occurrence are probably most interesting fact since I can calculate the performance of the memory system using this data. The running percentages show the fraction of occurrences have the indicated value or below.

"bankeval.c" 17 ≡

```
/*
   Percentages
*/
  printf("\nPercentages:\n");
  for ( i = 0; i <= s; i++ ) {
    pct[i]=((float)count[i])/((float)cases);
    printf(" %8.6f",pct[i]);
  }
  printf("\n");
/*
   Running Percentages
*/
  printf("Running Percentages:\n");
  sum = 0;
  for ( i = 0; i <= s; i++ ) {
    sum += pct[i];
    printf(" %8.6f",sum);
  }
  printf("\n");
◇
```

File defined by scraps 14bc, 15, 16abc, 17, 18abc, 19ab, 20abc.

The average time to satisfy the incoming requests can be calculated from the number of ports on each bank, `p` in the inputs, and the percentages just printed. Note that this is the only place `p` is used. I'll save time by evaluating all reasonable values for `p`.

"bankeval.c" 18a ≡

```
/*
   Timing
*/
  printf("Cycles per case for various numbers of banks for %d cases.\n",cases);
  for ( p = 1; p < s; p++ ) {
    time = 0;
    for ( i = 1; i <= s; i++ ) time += pct[i]*((i+p-1)/p);
    printf("%d %f\n",p,time);
  }
}
```

◇

File defined by scraps 14bc, 15, 16abc, 17, 18abc, 19ab, 20abc.

That's the end of the main routine. Here is the function that evaluates Equation 2.

"bankeval.c" 18b ≡

```
/*
   Recursive evaluation of

                 (s)
    sum(k=1:c/m) ( ) sum(j=1:m-1) f(s-k,c-km,j)
                 (m)
*/
int f(s,c,m)
      int s, c, m;
{
```

◇

File defined by scraps 14bc, 15, 16abc, 17, 18abc, 19ab, 20abc.

First, some declarations.

"bankeval.c" 18c ≡

```
  int j, k, sumi, sumo;
  float pct, time;
```

◇

File defined by scraps 14bc, 15, 16abc, 17, 18abc, 19ab, 20abc.

The first thing to do is put in those cases that end the recurrence. The general rule for $f(x, y, z)$ is that

Table 5: Variables used by `f(s,c,m)`.

Variable	Type	Definition
`s`	Input	Number of servers
`c`	Input	Number of clients
`m`	Input	Max clients per server
`j, k`	Local	Loop counters
`sumi`	Local	Inner loop accumulator
`sumo`	Local	Outer loop accumulator
`pct`	Out	count/cases
`time`	Out	Ave. time per s refs

$$\begin{aligned} f(s,c,1) &= \binom{s}{c} & \forall s \geq c \geq 1 \\ &= 0 & \forall s < c \\ f(s,0,1) &= 1 & \forall s \geq 1 \\ f(s,0,m) &= 0 & m > 1. \end{aligned} \tag{8}$$

"bankeval.c" 19a ≡

```
    if ( m == 1 ) {
      if ( c == 0 ) return 1;
      if ( s >= c ) return binomial(s,c);
      else          return 0;
    }
    if ( m > 1 && c == 0 ) return 0;
◇
```

File defined by scraps 14bc, 15, 16abc, 17, 18abc, 19ab, 20abc.

Now, let's look at the other cases. The outer loop runs over the number of ways that a maximum of m can occur.

"bankeval.c" 19b ≡

```
  /*
     Sum over number of ways max of m can occur
  */
    sumo = 0;
    for ( k = 1; k <= c/m; k++ ) {
◇
```

File defined by scraps 14bc, 15, 16abc, 17, 18abc, 19ab, 20abc.

The inner loop runs over all maxima less than m.

"bankeval.c" 20a ≡

```
/*
   Sum over all maxima less than m
*/
     sumi = 0;
     for ( j = 1; j < m; j++)
       sumi += f(s-k,c-k*m,j);
◇
```

File defined by scraps 14bc, 15, 16abc, 17, 18abc, 19ab, 20abc.

Finally, we do the outer sum and return.

"bankeval.c" 20b ≡

```
     sumo += binomial(s,k) * sumi;
  }
  return sumo;
}
◇
```

File defined by scraps 14bc, 15, 16abc, 17, 18abc, 19ab, 20abc.

We need to define the binomial coefficient. This routine uses the recursive formula from Equation (6.1.7) from *Numerical Recipes*[3]. This program returns 0 when $n < k$.

"bankeval.c" 20c ≡

```
/*
   Binomial coefficient by recursion
*/
int binomial(n,k)
     int n, k;
{
  if ( k >  n ) return 0;
  if ( k == 0 || k == n ) return 1;
  if ( k == 1 ) return n;
  else return ((n-k+1)*binomial(n,k-1))/k;
}
◇
```

File defined by scraps 14bc, 15, 16abc, 17, 18abc, 19ab, 20abc.