Living on the Edge: The limits of Integer Computation

A Naive Algorithm for Combinations

Consider the following obvious algorithm for computing ${}^nC_k$

$$ {}^nC_k = \begin{cases} \perp & \text{if } n < 0 \vee k < 0 \vee k > n \\ 1 & \text{if } k = 0 \vee k = n \\ \dfrac{n!}{(n-k)!k!} & \end{cases} \tag{1} $$

Note that the second case defines ${}^0C_0 = 1$.

There are obvious problems with this algorithm on a real computer. For $k > 14$, $k!$ is always greater than the maximum integer the machine will allow with 32 bits. So even though ${}^nC_k$ might be a small number e.g. ${}^{20}C_{19} = 20$, the computation will never give us the result because of overflow obtained in the computation of 20! or 19!. Hence it is necessary to use alternative means to do such a computation. In the following algorithms we concentrate on the last clause for computation and assume that the technical completeness requirements are the same as in (1).

Alternative algorithm 1.

A little analysis shows that we may prove by induction that

$$ {}^nC_k = {}^{n-1}C_{k-1} + {}^{n-1}C_k \tag{2} $$

In fact there is a simple intuitive argument to prove (2). Assume you have a set S of n objects ($n > 0$). Consider the collection A of all subsets of S containing k objects. Clearly $|A| = {}^nC_k$. Consider any object $x \in S$. Partition A into two collections B and C where B is the collection of those sets in A each of which contains x, and C is the collection of those sets in A which *do not* contain x.

Clearly, B and C are disjoint collections and $A = B \cup C$. Hence $|A| = |B| + |C|$. Further,

$$ \begin{aligned} |B| &= {}^{n-1}C_{k-1} \quad \text{and} \\ |C| &= {}^{n-1}C_k \end{aligned} $$

from which (2) follows.

However the obvious algorithm defined by (2) may also not work very close to the limits of the computer memory, since it involves two recursions. It is likely to be very slow because of the non-linear recursion involved. But it is very safe with respect to overflows because ultimately its computation works out to be a sum of ${}^nC_k$ additions of the number 1. But because of the non-linear recursion involved in it, it might run out of memory storage.

We consider yet another algorithm based on obvious identities.

Alternative algorithm 2.

For $0 < k \leq n$ we know from school mathematics that the computation in (1) could be simplified by cancelling out common factors. Hence

$$^nC_k = \frac{n(n-1)\cdots(n-k+1)}{k!} \tag{3}$$

and

$$^nC_k = \frac{n(n-1)\cdots(k+1)}{(n-k)!} \tag{4}$$

A simple counting shows that (3) requires the multiplication of k numbers each in the numerator and denominatore and a single division. Similarly (4 requires $n-k$ multiplications instead. Clearly, then we may use the relative values of the two quantities k and $n-k$ to determine how to proceed. We then have the following algorithm.

$$^nC_k = \begin{cases} \dfrac{n(n-1)...(n-k+1)}{k!} & \text{if } k \leq n-k \\ \dfrac{n(n-1)...(k+1)}{(n-k)!} & \text{else} \end{cases} \tag{5}$$

Depending on the values of n and k, algorithm (5) chooses the one with the smaller number of multiplications.

However, this may still not be satisfactory if the division is performed after <u>*all*</u> the multiplications have been done. In general, integer multiplications could increase the product well beyond the integer limits of the machine.

Before we proceed with further improvements, we notice that both clauses of (5) are special cases of a function "*quotient of products*" defined as follows:

$$qop(m,l) = \frac{m(m-1)\cdots(m-l+1)}{l!} \tag{6}$$

where $qop(m,l)$ is the product of l consecutive integers starting from m down to $m-l+1$ divided by $l!$ (which itself is also the product of l consecutive integers starting from 1 up to l).

Algorithm (5) may then be rewritten as

$$^nC_k = \begin{cases} qop(n,k) & \text{if } k \leq n-k \\ qop(n,n-k) & \text{else} \end{cases} \tag{7}$$

The problem of improving algorithm 5 then reduces to the problem of computing *qop* in such a fashion as to provent overflows whenever possible.

Algorithm for *qop*

One possibility is to design the algorithm for *qop* by trying to ensure that each intermediate result is always as small a positive integer as possible. This involves keeping the number of multiplications small and at the same time doing *exact* integer division whenever possible in order to keep the intermediate result small.

We may use the following results from elementary number theory.

Theorem 1 *In any set of n consecutive integers ($n > 0$) there exists a multiple of k for every k, $0 < k \leq n$.*

The proof of the above theorem is obvious. Since $0 < k \leq n$ and there are n consecutive integers, the remainders of the n integers when divided by k, would yield at least one number with remainder 0.

An even more powerful theorem is the following.

Theorem 2 *The product of n consecutive positive integers is divisible by $n!$.*

The proof of the theorem is a generalization of the following lemma.

Lemma 1 *Let A be a set of n ($n \geq 2$) consecutive positive integers and let $k, m \in \{1..n\}$ such that $k \neq m$. Then $km \mid \prod A$.*

Proof. Without loss of generality we may assume $1 \leq k < m \leq n$. Clearly there are $\lfloor n/k \rfloor \geq 1$ multiples of k in A and $\lfloor n/m \rfloor \geq 1$ multiples of m in A. If there are distinct numbers $p, q \in A$ such that $k \mid p$ and $m \mid q$, there is nothing to prove.

Suppose $\lfloor n/k \rfloor = 1 = \lfloor n/m \rfloor$ and let $q \in A$ be the only multiple of both k and m. Let $gcd(k, m) = d \geq 1$. Then $k = da$ and $m = db$ for some $a, b \geq 1$ and $a \neq b$. Obviously $1 \leq d \leq k < m \leq n$ $a \leq k$, $b \leq m$ and $a < b$ since $k < m$. Further there are at least $\lfloor n/d \rfloor$ multiples of d in A and $\lfloor n/d \rfloor \geq b > a \geq 1$. Hence $\lfloor n/d \rfloor \geq 2$ i.e. there are at least two multiples of d in A, of which q is clearly one. Let $p \in A - \{q\}$ be another multiple of d. From the claim below it follows that $km \mid \prod A$.

Claim. $\underline{km \mid pq}$.

Proof of claim. Since $k \mid q$, $q = ki$ for some i and hence $q = dai$. Similarly since $m \mid q$, $q = mj$ for some j and so $j = q/m = \dfrac{ai}{b}$ which is a positive integer. $d \mid p$, so there must be some e such that $p = de$. Therefore $p(q/m) = deai/b = kei/b$ which must be a positive integer. Hence $k \mid p(q/m)$. Therfore $km \mid pq$. □.

An initial refinement of *qop* is the following.

$$qop(m, l) = \begin{cases} \bot & \text{if } l > m \vee l \leq 0 \vee m \leq 0 \\ 1 & \text{else if } m = 0 \vee l = 0 \\ qop\_iter(m - l + 1, m, 1, l, 1) & \text{else} \end{cases} \tag{8}$$

where $qop\_iter(bn, en, bd, ed, p)$ is a function which computes the value of $qop(m, l)$ by an algorithm which keeps the value of the intermediate result p small, positive and ensures exact integer division. In particular, p is an intermediate result of the quotient of products such that the numbers still to be multiplied in the numerator are those in the interval $[bn, en]$ and the numbers that still need to be divided (in the denominator) are those in the interval $[bd, ed]$. In effect we ensure that

$$m - l + 1 \leq bn \leq en \leq m$$

and

$$1 \leq bd \leq ed \leq l$$

and that the result of all other multiplications and divisions is given by the value p.

The algorithm for qop_iter then goes as follows.

$$qop\_iter(bn, en, bd, ed, p) = \begin{cases} p & \text{if } bd > ed \wedge bn > en \\ p \times en & \text{else if } bd > ed \wedge bn = en \\ p \texttt{ div } ed & \text{else if } bd = ed \wedge bn > en \\ qop\_iter(bn, en, bd, ed - 1, p \texttt{ div } ed) & \text{else if } ed \mid p \\ qop\_iter(bn, en, bd + 1, ed, p \texttt{ div } bd) & \text{else if } bd \mid p \\ qop\_iter(bn + 1, en, bd, ed, p \times bn) & \text{else} \end{cases} \tag{9}$$

Notice that in each case of division we ensure that integer division is performed only if the dividend is an integer multiple of the divisor. This is true of the third case p `div` ed too, because of theorem 2.

Assignment

Translate the algorithms (1), (2) and (5) into technically complete programs and test them out at the limits of integer computation on the machine. In particular, find non-trivial values values of n and k which exercise all the cases of the algorithm and

1. show by examples that within the limits of integer computation all the three programs yield the same values when there is no overflow.
2. find values of n and k for which algorithm (1) overflows whereas algorithms (2) and (5) yield identical integer values.
3. find values of n and k for which algorithms (5) and (1) overflow whereas algorithm (2) yields an integer value. This might happen (even though we are trying to keep the product as small as possible) because integer multiplication can cause overflows sooner than integer addition.
4. Can you find values of n and k for which algorithms (1) and (2) overflow whereas algorithm (5) yields an integer value?