**CHAPTER 2**

**Algorithm Analysis**

**2.1** 2/*N*, 37, , *N*, *N* log log *N*, *N* log *N*, *N* log(*N*2), *N* log2 *N*, *N*1.5, *N*2, *N*2 log *N*, *N*3, 2*N*/2, 2*N*.

*N* log *N* and *N* log (*N*2) grow at the same rate.

**2.2 (a)** True.

**(b)** False. A counterexample is *T*1(*N*) = 2*N*, *T*2(*N*) = *N*, and *f* (*N*) = *N*.

**(c)** False. A counterexample is *T*1(*N*) = *N*2, *T*2(*N*) = *N*, and *f* (*N*) = *N*2.

**(d)** False. The same counterexample as in part (c) applies.

**2.3** We claim that *N* log *N* is the slower growing function. To see this, suppose otherwise. Then,  would grow slower than log *N*. Taking logs of both sides, we find that, under this assumption,  grows slower than log log *N*. But the first expression simplifies to  If *L* = log *N*, then we are claiming that  grows slower than log *L*, or equivalently, that ε2*L* grows slower than log2 *L*. But we know that log2 *L* = *o*(*L*), so the original assumption is false, proving the claim.

**2.4** Clearly,  if *k*1 < *k*2, so we need to worry only about positive integers. The claim is clearly true for *k* = 0 and *k* = 1. Suppose it is true for *k* < *i*. Then, by L’Hôpital’s rule,



The second limit is zero by the inductive hypothesis, proving the claim.

**2.5** Let *f*(*N*) = 1 when *N* is even, and *N* when *N* is odd. Likewise, let *g*(*N*) = 1 when *N* is odd, and *N* when *N* is even. Then the ratio *f*(*N*)/*g*(*N*) oscillates between 0 and *inf*.

**2.6 (a)** 

**(b)** *O*(log log *D*)

**2.7** For all these programs, the following analysis will agree with a simulation:

**(I)**The running time is *O*(*N*).

**(II)**The running time is *O*(*N*2).

**(III)**The running time is *O*(*N*3).

**(IV)**The running time is *O*(*N*2).

**(V)***j* can be as large as *i*2, which could be as large as *N*2. *k* can be as large as *j*, which is *N*2. The running time is thus proportional to *N*⋅*N*2⋅*N*2, which is *O*(*N*5).

**(VI)**The *if* statement is executed at most *N*3 times, by previous arguments, but it is true only *O*(*N*2) times (because it is true exactly *i* times for each *i*). Thus the innermost loop is only executed *O*(*N*2) times. Each time through, it takes *O*(*j*2) = *O*(*N*2) time, for a total of *O*(*N*4). This is an example where multiplying loop sizes can occasionally give an overestimate.

**2.8 (a)** It should be clear that all algorithms generate only legal permutations. The first two algorithms have tests to guarantee no duplicates; the third algorithm works by shuffling an array that initially has no duplicates, so none can occur. It is also clear that the first two algorithms are completely random, and that each permutation is equally likely. The third algorithm, due to R. Floyd, is not as obvious; the correctness can be proved by induction. See J. Bentley, “Programming Pearls,” *Communications of the ACM* 30 (1987), 754–757. Note that if the second line of algorithm 3 is replaced with the statement

swap References( a[i], a[ ran dint( 0, n–1 ) ] );

then not all permutations are equally likely. To see this, notice that for *N* = 3, there are 27 equally likely ways of performing the three swaps, depending on the three random integers. Since there are only 6 permutations, and 6 does not evenly divide 27, each permutation cannot possibly be equally represented.

**(b)** For the first algorithm, the time to decide if a random number to be placed in *a*[*i*] has not been used earlier is *O*(*i*). The expected number of random numbers that need to be tried is *N*/(*N* – *i*). This is obtained as follows: *i* of the *N* numbers would be duplicates. Thus the probability of success is (*N* – *i*)/*N*. Thus the expected number of independent trials is *N*/(*N* – *i*). The time bound is thus



The second algorithm saves a factor of *i* for each random number, and thus reduces the time bound to *O*(*N* log *N*) on average. The third algorithm is clearly linear.

**(c,d)** The running times should agree with the preceding analysis if the machine has enough memory. If not, the third algorithm will not seem linear because of a drastic increase for large *N*.

**2.9** Algorithm 1 at 10,000 is about 38 minutes and at 100,000 is about 26 days. Algorithms 1–4 at 1 million are approximately: 72 years, 4 hours, 0.7 seconds, and 0.03 seconds respectively. These calculations assume a machine with enough memory to hold the entire array.

**2.10 (a)** *O*(*N*)

**(b)** *O*(*N*2)

**(c)** The answer depends on how many digits past the decimal point are computed. Each digit costs *O*(*N*).

**2.11 (a)** Five times as long, or 2.5 ms.

**(b)** Slightly more than five times as long.

**(c)** 25 times as long, or 12.5 ms.

**(d)** 125 times as long, or 62.5 ms.

**2.12 (a)** 12000 times as large a problem, or input size 1,200,000.

**(b)** input size of approximately 425,000.

**(c) ** times as large a problem, or input size 10,954.

**(d)** 120001/3 times as large a problem, or input size 2,289.

**2.13 (a)** *O*(*N*2).

**(b)** *O*(*N* log *N*).

**2.14**

a) Below are the values of poly at each iteration

poly = 0

poly = 3\*0 + 4 = 4

poly = 3\*4 + 8 = 20

poly = 3\*20 + 1 = 61

poly = 3\*61 + 2 = 185

b) = *a*N*xN*+ *a*N-1*xN-1*+ … + *a*1*x* + *a*0 =(x…(*x*(*x*(*a*N) + *a*N-1) + *a*N-2) … *a*1) + *a*0

c) O(*N*)

2.15

**bool indexIsValue( int A[], int low, int high)**

**{**

**if (low>high) return false;**

**else**

**{**

**int mid = (low+high)/2;**

**if (A[mid] == mid) return true;**

**else if (A[mid] > mid) return indexIsValue(A,low, mid-1);**

**else return indexIsValue(A,mid+1,high);**

**}**

**}// O(lg(n))**

2.16

**int gcd( int a, int b)**

**{**

**if (b == 0) return a;**

**else if (a%2 && b%2) // both odd**

**return gcd((a+b)/2, (a-b)/2);**

**else if (a%2 && !b%2) // b is even, a is odd**

**return gcd(a,b/2);**

**else if (!a%2 && b%2) // a is even, b is odd**

**return gcd(a/2, b);**

**else**

**return 2\*gcd(a/2, b/2); // both even**

**}**

2.17

a)

/\*\*

\* Linear-time minimum contiguous subsequence sum algorithm.

\*/

int minSubSum( const vector<int> & a )

{

int minSum = a[0], thisSum = 0;

for( int j = 1; j < a.size( ); ++j )

{

thisSum += a[ j ];

if( thisSum < minSum )

minSum = thisSum;

else if( thisSum > 0 )

thisSum = 0;

}

b)

/\*\*

\* Quadratic time minimum positive contiguous subsequence sum algorithm.

\*/

int minPosSubSum( const vector<int> & a )

{

vector<int> partialSum;

int minPosSum = -1;

int sum;

partialSum.push\_back(a[0]);

if (a[0] > 0)

minPosSum = a[0];

for (auto i = 1; i < a.size(); i++)

{

partialSum.push\_back(a[i] + a[i-1]);

if (a[i] > 0 && minPosSum == -1)

minPosSum = a[i];

else if (a[i] > 0 && a[i] < minPosSum)

minPosSum = a[i];

}

for (auto i = 0 ; i < a.size(); i++)

for (auto j = i+1; j < a.size(); j++)

{

sum = partialSum[j] - partialSum[i];

if (sum > 0 && sum < minPosSum)

minPosSum = sum;

}

return minPosSum;

}// O(n + n^2) = O(n^2)

//A O(nlg(n)) algorithm exists but uses an advanced data structure

c)

/\*\*

\* linear time maximum contiguous subsequence product algorithm.

\*/

double maxSubProd( const vector<double> & a )

{

double maxVal = 1, maxPos = 1, maxNeg = 0;

for (auto i = 0; i < a.size(); i++)

{

if (a[i] > 0)

{

maxPos = maxPos\*a[i];

maxNeg = maxNeg\*a[i];

}

else if (a[i] < 0)

{

maxNeg = -maxPos\*a[i];

maxPos = -maxNeg\*a[i];// maxNeg == 0 will cause an empty positive suffix

}

else

{

maxPos =1;

maxNeg = 0;

}

if (maxPos < 1) maxPos = 1;

if (maxPos > maxVal) maxVal = maxPos;

}

return maxVal;

} // O(n)

2.18

/\*

bisection method

assumes (f(low)\*f(high) < 0 and only 1 zero between low and high)

termination ensured by choosing a tolerance (tol).

\*/

#include<iostream>

using namespace std;

const double tol = 0.00001;

double f(double x)

{

return x\*x - x;

}

double bisection( double (\*f)(double), double low, double high)

{

double mid = (high + low)/2.;

double value;

if ((high - low) < tol)

return (mid);

value = f(mid);

if (value\*f(low) < 0)

return bisection(f, low, mid);

else

return bisection(f, mid, high);

}

int main()

{

cout<<bisection(f, .1 , 10)<<endl;

return 0;

}

2.19

struct MaxSeq

{

int value;

int startIndex;

int endIndex;

MaxSeq operator + (const MaxSeq & rhs) const

{

MaxSeq sum;

sum.value = value + rhs.value;

sum.startIndex = min(startIndex, rhs.startIndex);

sum.endIndex = max(endIndex, rhs.endIndex);

return sum;

}

};

MaxSeq max(const MaxSeq & a, const MaxSeq & b)

{

if (a.value > b.value)

return a;

else

return b;

}

/\*\*

\* Cubic maximum contiguous subsequence sum algorithm.

\*/

MaxSeq maxSubSum1( const vector<int> & a )

{

MaxSeq maxSum;

maxSum.value = 0;

for( int i = 0; i < a.size( ); ++i )

for( int j = i; j < a.size( ); ++j )

{

int thisSum = 0;

for( int k = i; k <= j; ++k )

thisSum += a[ k ];

if( thisSum > maxSum.value )

{

maxSum.value = thisSum;

maxSum.startIndex = i;

maxSum.endIndex = j;

}

}

return maxSum;

}

/\*\*

\* Quadratic maximum contiguous subsequence sum algorithm.

\*/

MaxSeq maxSubSum2( const vector<int> & a )

{

MaxSeq maxSum;

maxSum.value = 0;

for( int i = 0; i < a.size( ); ++i )

{

int thisSum = 0;

for( int j = i; j < a.size( ); ++j )

{

thisSum += a[ j ];

if( thisSum > maxSum.value )

{

maxSum.value = thisSum;

maxSum.startIndex = i;

maxSum.endIndex = j;

}

}

return maxSum;

}

/\*\*

\* Recursive maximum contiguous subsequence sum algorithm.

\* Finds maximum sum in subarray spanning a[left..right].

\* Does not attempt to maintain actual best sequence.

\*/

MaxSeq maxSumRec( const vector<int> & a, int left, int right )

{

MaxSeq maxSum;

maxSum.value = a[left];

maxSum.endIndex = right;

maxSum.startIndex = left;

if( left == right ) // Base case

{

if( a[ left ] > 0 )

return maxSum;

else

{ maxSum.value = 0;

return maxSum;

}

}

int center = ( left + right ) / 2;

MaxSeq maxLeftSum = maxSumRec( a, left, center );

MaxSeq maxRightSum = maxSumRec( a, center + 1, right );

MaxSeq maxLeftBorderSum, leftBorderSum;

maxLeftBorderSum.value = leftBorderSum.value = 0;

leftBorderSum.startIndex = left;

leftBorderSum.endIndex = center;

for( int i = center; i >= left; --i )

{

leftBorderSum.value += a[ i ];

if( leftBorderSum.value > maxLeftBorderSum.value )

maxLeftBorderSum = leftBorderSum;

}

MaxSeq maxRightBorderSum, rightBorderSum;

maxRightBorderSum.value = rightBorderSum.value = 0;

rightBorderSum.startIndex = center+1;

rightBorderSum.endIndex = right;

for( int j = center + 1; j <= right; ++j )

{

rightBorderSum.value += a[ j ];

if( rightBorderSum.value > maxRightBorderSum.value )

maxRightBorderSum = rightBorderSum;

}

return max( max(maxLeftSum, maxRightSum),

maxLeftBorderSum + maxRightBorderSum );

}

/\*\*

\* Linear-time maximum contiguous subsequence sum algorithm.

\*/

MaxSeq maxSubSum4( const vector<int> & a )

{

MaxSeq maxSum, thisSum;

maxSum.value = thisSum.value = 0;

maxSum.startIndex = maxSum.endIndex = 0;

thisSum.startIndex = 0;

for( int j = 0; j < a.size( ); ++j )

{

thisSum.value += a[ j ];

thisSum.endIndex = j;

if( thisSum.value > maxSum.value )

maxSum = thisSum;

else if( thisSum.value < 0 )

{

thisSum.value = 0;

thisSum.startIndex = j;

}

}

return maxSum;

}

**2.20** **(a)** Test to see if *N* is an odd number (or 2) and is not divisible by 

**(b) ** assuming that all divisions count for one unit of time.

**(c)** *B* = *O*(log *N*).

**(d)** *O*(2*B*/2).

**(e)** If a 20-bit number can be tested in time *T*, then a 40-bit number would require about *T*2 time.

**(f)** *B* is the better measure because it more accurately represents the *size* of the input.

**2.21** The running time is proportional to *N* times the sum of the reciprocals of the primes less than *N*. This is *O*(*N* log log *N*). See Knuth, Volume 2.

**2.22** Compute *X*2, *X*4, *X*8, *X*10, *X*20, *X*40, *X*60, and *X*62.

**2.23** Maintain an array that can be filled in a for loop. The array will contain *X*, *X*2, *X*4, up to  The binary representation of *N* (which can be obtained by testing even or odd and then dividing by 2, until all bits are examined) can be used to multiply the appropriate entries of the array.

**2.24** For *N* = 0 or *N* = 1, the number of multiplies is zero. If *b*(*N*) is the number of ones in the binary representation of *N*, then if *N* > 1, the number of multiplies used is



**2.25 (a)** *A*.

**(b)** *B*.

**(c)** The information given is not sufficient to determine an answer. We have only worst-case bounds.

**(d)** Yes.

**2.26 (a)** Recursion is unnecessary if there are two or fewer elements.

**(b)** One way to do this is to note that if the first *N* – 1 elements have a majority, then the last element cannot change this. Otherwise, the last element could be a majority. Thus if *N* is odd, ignore the last element. Run the algorithm as before. If no majority element emerges, then return the *Nth* element as a candidate.

**(c)** The running time is *O*(*N*), and satisfies *T*(*N*) = *T*(*N*/2) + *O*(*N*).

**(d)** One copy of the original needs to be saved. After this, the *B* array, and indeed the recursion, can be avoided by placing each *Bi* in the *A* array. The difference is that the original recursive strategy implies that *O*(log *N*) arrays are used; this guarantees only two copies.

**2.27** Start from the top-right corner. With a comparison, either a match is found, we go left, or we go down. Therefore, the number of comparisons is linear.

**2.28 (a, c)** Find the two largest numbers in the array.

**(b, d)** Similar solutions; (b) is described here. The maximum difference is at least zero (*i* ≡ *j*), so that can be the initial value of the answer to beat. At any point in the algorithm, we have the current value *j*, and the current low point *i*. If *a*[*j*] – *a*[*i*] is larger than the current best, update the best difference. If *a*[*j*] is less than *a*[*i*], reset the current low point to *i*. Start with *i* at index 0, *j* at index 0. *j* just scans the array, so the running time is *O*(*N*).

**2.29** Otherwise, we could perform operations in parallel by cleverly encoding several integers into one. For instance, if A = 001, B = 101, C = 111, D = 100, we could add A and B at the same time as C and D by adding 00A00C + 00B00D. We could extend this to add *N* pairs of numbers at once in unit cost.

2.30

a) There are RC squares to start a search. For each starting square there are 8 directions in which to create words to look up. Since the max length of a word is 10 letters, there are possible 80RC words. Each of these needs to be looked up in the word list. So the running time is O(RCW)

b) If the word list is sorted, one can use a binary search to loop up potential words and the running time becomes O(RC lg (W)).

**2.31** No. If *low* = 1, *high* = 2, then *mid* = 1, and the recursive call does not make progress.

2.32

/\*\*

\* Performs the standard binary search using two comparisons per level.

\* Returns index where item is found or -1 if not found.

\*/

template <typename Comparable>

int binarySearch( const vector<Comparable> & a, const Comparable & x )

{

int low = 0, high = a.size( ) - 1;

while( low < high )

{

int mid = ( low + high ) / 2;

if( a[ mid ] <= x )

low = mid + 1;

else

high = mid - 1;

}

if (low > high)

return NOT\_FOUND; // NOT\_FOUND is defined as -1

else

return (a[low] == x);

}

**2.33** No. As in Exercise 2.31, no progress is made.

**2.34** See my textbook *Data Structures and Problem Solving using Java* for an explanation.