University of Waterloo
Systems Design Engineering

Fibonacci Sequence and Kth Order Statistic

Microsoft Inc.
1 Microsoft Way, Redmond, WA, 98052

Prepared by
Jeyavithushan Jeyaloganathan
ID 20348912
3A Department of Systems Design Engineering
September 10th, 2012

Jeyavithushan Jeyaloganathan
124 Homedale Drive.
Scarborough, Ontario
M1V 1M2

September 10th, 2012

Dr. Paul Fieguth, Department Chair
Department of Systems Design Engineering
Engineering 5, 6th Floor
University of Waterloo
200 University Avenue West
Waterloo, Ontario N2L 3G1

Dr. Paul Fieguth

This report, entitled "Fibonacci Sequence and Kth Order Statistic" was prepared as my 2A Work Report for Microsoft. This is my fourth work term report and serves as a makeup to the 2A work term report I failed due to lateness. The main aim of this paper is to analyze existing algorithms that solve for the Fibonacci Sequence and the Kth Order Statistic. These are common problems in the computer science and engineering field in which non-optimal solutions are consistently used.

Microsoft is a company based in Redmond, Washington that has a wide portfolio of software and hardware products – the primary one being its Windows operating system.

The Windows Phone 8 camera section, in which I was a program manager, is managed by Ken Crocker and is primarily involved designing the camera experience for the next release of the Windows Phone OS.

This report was written entirely by me and has not received any previous academic credit at this or any other institution. I received no other assistance.

Sincerely,
Jeyavithushan Jeyaloganathan
ID 20348912

Abstract

This paper analyzes the existing algorithms to generate the Fibonacci numbers and to solve for the Kth Order statistic. Further insights into these algorithms are made based on the result of the analytical and run-time analysis. Critical criteria determining the success of the algorithm include average and worst case time complexity, space complexity, run time with both small and large n(datasets/ nth term). The results of the various analyses such as run-time analysis point toward the median of medians quick selection algorithm as being the fastest Kth Order Statistic selector and the matrix square multiplication algorithm as being the fastest Fibonacci number generator. In terms of run-time analysis, these algorithms preformed exceedingly well and verified the analytical analysis. It was determined on the other hand that the most commonly used recursive solution to the Fibonacci sequence is extremely slow and can cause crashes. Another inference tells us that while the quicksort and max heap kth order statistic algorithms weren't the fastest, due to the fact they are fairly basic algorithms and easy to code they are good algorithms to use if coding time and skill is a consideration. Delineated recommendations for future analysis include using double long as the type variable for the Fibonacci sequence algorithms to prevent overflows and using a more comprehensive code profiler with a more precise timer to analyze the run-times of the algorithms in order to gain more insight into them.

Table of Contents

Table of Figures & Tables

1.0 Introduction

1.1 Overview and Scope

This paper analyzes from a holistic perspective, multiple methods to solve two important but basic problems in computer science.

- **How do you find the Kth order statistic?**
- **How can you generate the nth term in the Fibonacci sequence?**

It provides an in depth analysis of the various existing algorithms to these problems and attempts to delineate the optimum solutions based on a variety of criteria including the space and time complexities of the algorithm as well as its average and worst-case behaviours.

1.2 Motivation and Significance

1.2.1 Fibonacci Sequence

In mathematics, an important sequence is the Fibonacci series.

It is a sequence numbers defined by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2}$$

$$where\ n \in N$$

That is, F_n is the nth term of the Fibonacci sequence where n is a number that belongs in the set of all Natural numbers. The first terms of the Fibonacci sequence are

$$F_0 = 0$$

$$F_1 = 1$$

These are referred to as the seed values and from them; the rest of the sequence can be generated.

Table 1 Fibonacci Sequence

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9
0	1	1	2	3	5	8	13	21	34

Refer to Appendix A for calculations

The Fibonacci sequence occurs unexpectedly in various mathematical theories such as in the sum of shallow diagonals in Pascal's triangle [1].

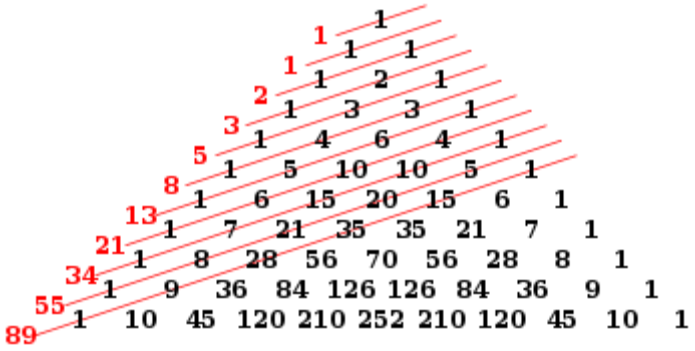


Figure 1 Fibonacci in Pascal's Triangle [1]

It also appears in biological and natural settings such as branching trees, arrangement of leaves on a stem and the fruitlets of a pineapple [2]. The Fibonacci numbers are important in computer science; more specifically they are useful in the run-time analysis of Euclid's algorithm to determine the greatest common divisor of two integers. Additionally, they are also used in real world modeling simulations and computer graphics emulating nature.

Because of the sequence's importance, it is necessary to know the optimal method to generate it in order to save time and space.

1.2.2 Kth Largest/Smallest Element

In computer science a common problem lies in determining the Kth largest/smallest element in a random set of numbers, typically in an array. K is defined as a natural number ranging from 0 to N-1 where N is the size of the array.

Finding the Kth order statistic is a problem that occurs in many fields of computer science and mathematics such as in digital image processing and statistics & probability. It is important in developing a distribution as well as finding the mean, minimum and maximum of any set of data [3].

Additionally when working with large datasets, it becomes apparent that translating a vast volume of information into a human readable format is very difficult unless a pivot (K) is selected where K is either the most expensive, largest, smallest etc. item in the data set.

Because it is a commonly recurring problem, solving it in the most efficient and effective manner appropriate to the situation is of great interest in order to save time, space and money.

2.0 Analysis

This section contains the proposed methodology & evaluation schema for the different solutions to the Fibonacci and Kth order statistic problems.

2.1 Methodology

The methodology outlines the procedure and criteria that will be used to evaluate the algorithms for the two problems.

2.1.1 Evaluation Criteria

The algorithms will be evaluated based on 5 main evaluation criterion.

Average case run time analysis

The average case time complexity for the algorithm. This will be determined analytically.

Worst case run time analysis.

The worst case time complexity for the algorithm. This will we be determined analytically.

Space Complexity

The space required by the algorithm. This will be determined analytically.

Run time for small term/dataset

The run time in milliseconds of the algorithms for finding the 10^{th} Fibonacci number and finding the 25^{th} smallest order statistic in a set of 100 elements.

Run time for large term/dataset

The run time in milliseconds of the algorithms for finding the 45^{th} Fibonacci number as well as the 2500^{th} smallest order statistic in a set of 10000 elements.

2.1.2 Evaluation Schema

Using the evaluation criteria, each algorithm will be ranked against each other in each category.

Table 2 Fibonacci Evaluation Schema

Fibonacci Method	Average Case	Worst Case	Space Complexity	Time for 10^{th} term (milliseconds)	Time for 45^{th} term(milliseconds)
1					
2					
3					
4					
5					

Table 3 Kth Order Statistic Evaluation Schema

Kth Order Statistic Algorithm	Average Case	Worst Case	Space Complexity	Time for 25^{th} term in set of 100(milliseconds)	Time for 250^{th} term in set of 1000(milliseconds)
1					
2					
3					
4					
5					

Notice that there are no weightings in the schemas. This is intentional as in practice the efficiency and effectiveness of an algorithm is situational. Therefore it is better to collect raw data in the analysis and draw conclusions about the algorithms usefulness only when considering specific situations.

2.2 Analytical Algorithm Analysis

This section describes 5 solutions for each problem. The specific pseudo codes for the algorithms were obtained from the indicated websites in citations [4] [5]. A working implementation of the described algorithms written in C++ can be found in Appendix A.

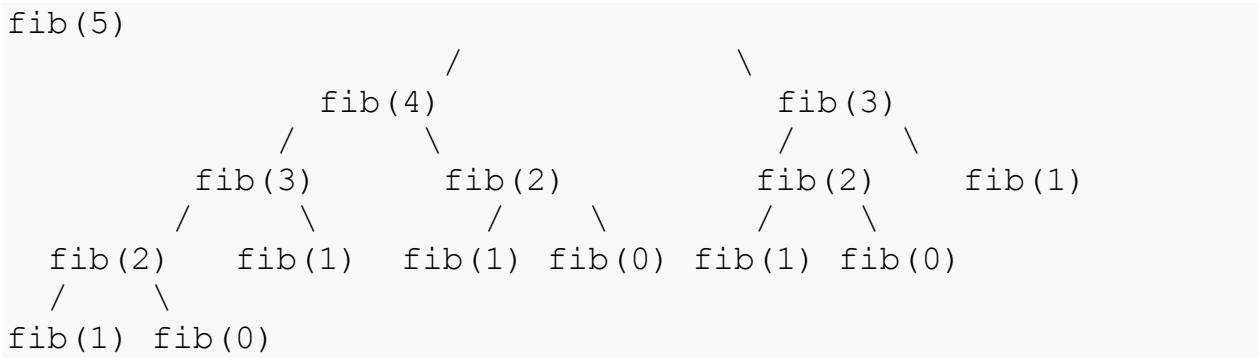
2.2.1 Fibonacci Sequence 1

```
int fib (int n)
{
if (n <= 2) return 1
else return fib(n-1) + fib(n-2)
}
```

The first algorithm uses the basic recursive approach to find the nth Fibonacci number. The recurrence relation is given by

$$T(n) = T(n-1) + T(n-2)$$

In recursive algorithms, the recurrence relation represents the time complexity. In this case the relation translates to an exponential time algorithm$\approx O(1.6^n)$. It is observed that this algorithm does a lot of repeated work. It is easier to see by following the recursion tree for the 5th Fibonacci number.



The space complexity is constant as the algorithm uses no extra space besides the memory in the call stack.

2.2.2 Fibonacci Sequence 2

```
int fib(int n)
    {
    int f[n+1];
    f[1] = f[2] = 1;
    for (int i = 3; i <= n; i++)
        f[i] = f[i-1] + f[i-2];
    return f[n];
    }
```

In order to avoid recomputation of the sub problems of algorithm 1, a dynamic programming approach can be taken that takes advantage of the repeating and overlapping sub structure of the Fibonacci Sequence. This algorithm solves each sub problem once and stores it for later lookup in order to avoid recomputation.

Since an array of size n is created, the space complexity of this algorithm is O(n).
The time complexity is O(n) as well. This is due to the for loop.

2.2.3 Fibonacci Sequence 3

```
int fib(int n)
    {
    int a = 1, b = 1;
    for (int i = 3; i <= n; i++) {
        int c = a + b;
        a = b;
        b = c;
    }
    return b;
    }
```

This algorithm is a modified version of algorithm 2. It optimizes it by using two rotating variables to store the previous Fibonacci numbers for calculation. This allows it to have a space complexity of O(1) but sacrifices on speed. Even though it is still linear time, it is slower than algorithm two because it takes roughly 4n lines of operations to compute the Fibonacci number.

2.2.4 Fibonacci Sequence 4

```
int fib(int n)
    {
    int M[2][2] = {{1,0},{0,1}}
    for (int i = 1; i < n; i++)
        M = M * {{1,1},{1,0}}
    return M[0][0];
    }
```

Another algorithm takes advantage of matrices and matrix multiplication to compute the Fibonacci sequence since it can be represented as the following:

$$\begin{matrix}[\,1\;1\,]\,n & [\,F(n+1)\;F(n)\;\;] \\ [\,1\;0\,] = & [\,F(n)\;\;F(n-1)\,]\end{matrix}$$

This takes O(n) time and uses O(n) space although it is still slower than algorithm 2 and 3 because of the speed of the matrix multiplication.

2.2.5 Fibonacci Sequence 5

```
int M[2][2] = {{1,0}{0,1}}

    int fib(int n)
    {
    matpow(n-1);
    return M[0][0];
    }

    void matpow(int n)
    {
    if (n > 1) {
        matpow(n/2);
        M = M*M;
    }
    if (n is odd) M = M*{{1,1}{1,0}}
    }
```

A unique method of multiplication that finds the nth power of a number by squaring the (n/2)th power allows the algorithm to be improved drastically. The recurrence relation for this special method of multiplication is given by

$$T(n) = O(1) + T(n/2)$$

This solves to a time complexity of O(logn). This means that when n is a large number such as 1 million, logn is about 20 in log base 2. It also has a space complexity of O(1).

2.2.6 Kth Order Statistic 1

1) Modify Bubblesort such that the limit for the outer loop is K.

2) The last k elements of the array in step 1 are the greatest.

This algorithm modifies Bubble sort to find the kth largest element. Its time complexity is O(nk) as it only goes through the array n times for the first k elements. The space complexity is O(1).

2.2.7 Kth Order Statistic 2

1) Store the first k elements in a temporary array called temp[0..k-1].

2) For each element *i* in the initial array arr[k] to arr[n-1]

    3) Find the smallest element in temp[] and let it be *min*.

    4) If *i* is greater than this minimum, the remove the minimum from temp[] and insert *i*.

5) The final k elements of *temp* are the greatest ones.

This algorithm is inspired by partial sorting. It creates two partitions of the array and then replaces the sub array of size k with the (n-k)th greatest elements. The time complexity is O((n-k)*k) since it tries to find the minimum in the sub array n-k times. The space complexity is O(k) because of the temp array needed.

2.2.8 Kth Order Statistic 3

1) Sort the elements in the array.

2) List out the final kth elements in the array.

Sorting using the fastest method in practice – quicksort takes O(nlogn) time on average. Printing out the k elements in the array takes k time so the run time complexity is O(nlogn + k). The space complexity is O(1).

2.2.9 Kth Order Statistic 4

1) Build a Max Heap tree in O(n)

2) Extract the max k times to get k maximum elements from the Max Heap O(klogn)

Adding up the two time complexities indicate that the overall time complexity of this algorithm is O(n+klogn). The space complexity is also O(1) since no extra space is needed.

2.2.10 Kth Order Statistic 5

1) Divide the set of elements into groups of 5.

2) Find the median of these group (quicksort).

3) Find the median of these medians.

4) Use Hoare's quicksort partition algorithm around this median.

5) Check if the median in its correct spot is the Kth order statistic. If not repeat steps 1-4 on the left side if the median is larger and the right side if it is smaller.

6) If it is the correct, then you have your Kth sorted elements.

This algorithm modifies Hoare's quicksort algorithm by selecting a better pivot for it using the median of medians algorithm that was developed by Blum, Floydd, Pratt, and Tarjan in their 1973 paper "Time bounds for Selection" [6]

The pivot selection is vital as it is what allows the algorithm to run in linear time. The recurrence relation for this algorithm reduces from

$$T(n) \leq T(n/5) + T(7 \cdot n/10) + O(n)$$

To

$$T(n) \leq c \cdot n \cdot (1 + (9/10) + (9/10)^2 + \cdots) \in O(n)$$

Showing that it runs in O(n) time with O(1) space. Note: For the purpose of this analysis, a professor of Stanford's code was used [7].

2.3 Run time Algorithm Analysis

The Fibonacci and Kth order statistic algorithms were written in C++ using Bloodshed Dev C++ and were compiled and executed on an i7 sandybridge quadcore 2630QM processor @ 2.00gHZ with 6 gigabytes of Ram.

2.31 Fibonacci Sequence

The algorithms for determining the Fibonacci numbers were compiled and their runtime was analyzed using the clock() function in the C++ standard library. The run times were collected for two different parameters:

- 10^{th} Fibonacci Term
- 45^{th} Fibonacci Term

Table 4 Run Time Analysis Fibonacci

Fibonacci Algorithms	Time to determine 10^{th} term in milliseconds	Time to determine 45^{th} term in milliseconds
1	0	Hanging
2	0	0
3	0	0
4	0	0
5	0	0

2.3.2 Kth Order Statistic

The algorithms for determining the Kth order statistic were analyzed using two different size sets of data and two different Kth order statistics.

- 25^{th} Largest in a set of 100 elements
- 2500^{th} Largest in a set of 10000 elements

Table 5 Run Time Anlaysis Kth Order Statistic

Kth Order Statistic Algorithms	**Time to determine 25^{th} Largest element in milliseconds**	**Time to determine 2500^{th} largest element in milliseconds**
1	117	10035
2	7	2400
3	1	16
4	4	14
5	0	11

2.4 Results

Combining the results of the analysis and tabulating the data a larger picture of the algorithms can be formed.

Table 6 Fibonacci Algorithm Results

Fibonacci Method	Average Case	Worst Case	Space Complexity	Time to determine 10th term in milliseconds	Time to determine 45th term in milliseconds
1	$O(1.6^n)$	$O(1.6^n)$	O(1)	0	Hanging
2	O(n)	O(n)	O(n)	0	0
3	O(n)	O(n)	O(1)	0	0
4	O(n)	O(n)	O(n)	0	0
5	O(logn)	O(logn)	O(1)	0	0

Table 7 Kth Order Algorithm Results

Kth Order Statistic Algorithms	Average Case	Worst Case	Space Complexity	Time to determine 25th Largest element in milliseconds	Time to determine 2500th largest element in milliseconds
1	O(nk)	O(nk)	O(1)	117	10035
2	O(n-k)*k	O(n-k)*k	O(k)	7	2400
3	O(nlogn+k)	O(n^2+k)	O(1)	1	16
4	O(klogn+n)	O(klogn+n)	O(1)	4	14
5	O(n)	O(n^2)	O(1)	0	11

The zeroes in the results mean that the task was essentially completed in 0 milliseconds. Additionally, hanging means that the task never finished. Most likely the hanging occurred because the call stack was overloaded because of the numerous sub problems in the Fibonacci sequence. When looking at the 5th algorithm for the Kth Order Statistic, it is important to note that the worst case rarely occurs. An example of this is where the Kth Order statistic is the last element in the array.

3.0 Conclusions

Reflecting on the results of the analytical and run time analysis, certain conclusions can be drawn.

3.1 All algorithms can be used to calculate Fibonacci numbers under the 10^{th} term

Since all the algorithms tested, completed the task of finding the 10^{th} Fibonacci number in essentially 0 milliseconds it can be safely said that any of them can be used in practice.

3.2 Most algorithms can be used to calculate the Fibonacci numbers up to the 45^{th} term

All the algorithms except for the 1rst one can be used to calculate the Fibonacci numbers up to the 45^{th} one. The 1rst one can't be used because its recursive nature overloads the call stack and causes the process to hang.

3.3 The best algorithm to calculate the Fibonacci number is the 5th one

Given the analytical analysis of the 5^{th} algorithm, it is plane to see that it is the most efficient when it comes to calculating the Fibonacci numbers because of its worst case run time of O(logn) and its space complexity of O(1). Leveraging the matrix form of the Fibonacci sequence and the power of square multiplication rules allows a logarithmic run time that supersedes all other algorithms. The only drawback of it is that it is slightly more difficult to code than the others.

3.4 The quicksort and max heap Kth Order Statistic algorithms are easy to code and very fast

The nature of the quicksort and the max heap algorithm allow it to be coded very easily by anyone with familiarity with basic data structures. They both function very fast for both large and small sets of data.

3.5 Partial bubble sorting and creating a temp array are very slow algorithms

Compared to the rest of the algorithms, the partial bubble sorting and replacing the min in a temp array with the next greatest element are very slow and seem to be slower on an order of 10x. Using these algorithms in any project would certainly slow it down considerably.

3.6 The median of medians quick selection algorithm is linear time

The fifth Kth Order statistic algorithm is a very fast algorithm. Normally the quick select algorithm is avoided because of its worst case run time if a bad pivot is selected but the median of medians algorithm allows for a better selection of pivot almost guaranteeing a linear run time with the exception of the kth element being found at the very edges of the array. It is a very fast algorithm but it is difficult to code without understanding its intricacies.

4.0 Recommendations

4.1 When writing the Fibonacci Sequence Algorithms use Long Double

The usage of the INT type variable prevented the computation of the Fibonacci Sequence over the 50^{th} term because of the maximum integer number which is 2147483647. In order to avoid this limitation, use the LONG DOUBLE type variable.

4.2 A more precise code profile/ run-time analyzer would allow further insight into run times of the algorithms

Looking at the Fibonacci sequence, it is very difficult to identify which algorithm is superior in the case of the smaller Fibonacci terms because of the precision of the clock() used to profile the code. A more precise timer would allow one to differentiate the run times leading to more insight such as which algorithm is faster in practice.

Appendix A

Calculating the Fibonacci Numbers

0,1

$0 + 1 = 1$

$1 + 1 = 2$

$2 + 1 = 3$

$2 + 3 = 5$

$3 + 5 = 8$

$5 + 8 = 13$

$8 + 13 = 21$

$13 + 21 = 34$

Generating Random Data Set

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   int num;
   ofstream outFile;
   outFile.open("input.txt", fstream::trunc);
   for (int i = 0; i < 10000; i++) {
      num = rand()%10000;
      outFile<<num<<" ";
   }
}
```

Kth Order Statistic Algorithms 1-4 [5]

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

void readFile(vector<int> &input);
void readFile (vector<int> &input) {
   int j;
   ifstream myFile("input.txt");
   while (!myFile.eof()) {
      myFile >> j;
      input.push_back(j);
   }
   input.pop_back();
}

void bubbleSort(vector<int> &vec, int k);
void bubbleSort(vector<int> &vec, int k) {

      for(int i = 0; i < vec.size()-k;i++){
                  for(int j = i+1; j < vec.size(); j++)
                  {
                  if(vec[i] > vec[j])
                        std::swap(vec[i],vec[j]);
                  }
   }
}
```

```
void qsort( vector<int> &array, int leftbound , int rightbound );
void qsort( vector<int> &array, int leftbound , int rightbound )
{
   if ((rightbound - leftbound) >= 2)
   {
      int pivot = array[leftbound + (rand() % (rightbound - leftbound))];
      int leftposition    = leftbound;
      int rightposition   = rightbound - 1;

      while ( leftposition < rightposition )
      {
         while ((array[leftposition] <= pivot) && (leftposition < rightposition))
         {   ++leftposition;
         }
         while ((array[rightposition] > pivot ) && (leftposition < rightposition))
         {   --rightposition;
         }
         std::swap(array[leftposition], array[rightposition]);
      }
      // At least the pivot point will be on the left hand side.
      // This will also be the largest value. So move the leftposition back
      // to remove all the pivot points.
      while(((leftposition-1) > leftbound) && (array[leftposition-1] == pivot))
      {   --leftposition;
      }
      qsort(array, leftbound, leftposition-1);    // leftposition is one past the end of the left
      qsort(array, rightposition+1, rightbound);  // Thus at the start of the right.
   }
}

int min(vector<int> &vec);
```

```
int min(vector<int> &vec){
    int min = vec[0];
    int location = 0;
    for (int i = 1; i<vec.size(); i++){
        if (vec[i]<min){
        min = vec[i];
        location = i;
        }
    }
    return location;
}

void tempArray (vector<int> &vec, int k);
void tempArray (vector<int> &vec, int k){
    vector<int> temp;

    for (int i = 0; i < k; i ++)
    temp.push_back(vec[i]);

    for (int i = k; i<vec.size(); i++){
        int location = min(temp);
        if (temp[location]<vec[i])
            swap(temp[location],vec[i]);
    }

    //for (int i = 0, j = temp.size()-1; i<k; i ++, j--)
    //cout << temp[j]<< endl;
    return;
}

void maxHeap(vector<int> &input, int k);
```

```
void maxHeap(vector<int> &input, int k){

    make_heap(input.begin(),input.end());
    cout<<input.front()<<endl;
    for (int i = 0; i<k-1; i++){
    pop_heap(input.begin(),input.end()-i);
    //cout<<input.front()<<endl;
    }
}


int main () {
    int k = 2500;
    vector<int> input;
    readFile(input);
    float t1 = clock();
    //bubbleSort(input,k);
    //qsort(input, 0, input.size());
    //tempArray(input, k);
    //maxHeap(input, k);
    float t2 = clock();
    float diff = (t2-t1)/CLOCKS_PER_SEC;
    cout << diff << " sec" << endl;

    //for (int i = 0, j = input.size()-1; i<k; i ++, j--)
    //cout << input[j]<< endl;

    system("pause");
}
```

Kth Order Statistic Algorithms 5 [7]

```
/*****************************************************************
 * File: Selection.hh
 * Author: Keith Schwarz (htiek@cs.stanford.edu)
 *
 * An implementation of the linear-time order selection algorithm
 * (median of medians) invented by Blum, Floyd, Pratt, Rivest,
 * and Tarjan.  This classic algorithm takes as input an array
 * and an index, then repositions the elements in the array so
 * that the nth smallest element is in the correct index, all
 * smaller elements are to the left, and all larger elements are
 * to the right.
 *
 * The basic idea behind the algorithm is to repeatedly use the
 * partition algorithm to split the input into three ranges.
 * Partitioning works by picking an arbitrary element of the
 * range, then rearranging the elements of the array so that
 * the element is in its correct position in sorted order, the
 * elements to the left of the element are no bigger than the
 * element, and the elements to the right are no larger than
 * the element.  For example:
 *
 * ------------0++++++++++++++++++++++++++
 *     ^      ^           ^
 *     |      |           |
 *     |      |           +--- Bigger elements
 *     |      +------------------ The element itself
 *     +------------------------- Smaller elements
 *
 * Once we have done this step, there are three cases.  First,
 * if the pivot element is now in the index we're looking for,
```

```
* we can just return it.  Otherwise, if the element is to the
* left, we recursively search the right side.  Finally, if the
* element is to the right, we recursively search the right
* side.
*
* This basic approach works marvelously well provided that every
* time we perform a partition, we split the elements into roughly
* even groups.  If this happens, we'll end up throwing away a
* large amount of the input at each step and we get a nice linear
* runtime.  However, we can run into trouble if our choice of
* pivots for the partition step are bad and generate lopsided
* splits.  This is akin to the behavior of quicksort - given
* bad pivots, even a normally fast algorithm can degenerate to
* quadratic behavior.
*
* To avoid this sort of problem, the median-of-medians algorithm
* uses some clever recursion to ensure a good pivot choice.  In
* particular, it works by breaking the input up into groups of
* five elements, sorting each of those blocks, then taking their
* medians.  It then recursively invokes the algorithm on those
* medians to choose the median of those medians, then uses that
* median as the pivot.  It can be shown that this choice of
* pivot ends up splitting the input no worse than 70%/30%.
* Using the formal definition of big-O, this can be shown to
* guarantee O(n) runtime in the worst case.
*
* The STL provides a variant of this algorithm called
* nth_element which uses the partitioning scheme without the
* clever choice of median.  On expectation it runs in O(n),
* but can degenerate to O(n^2) on bad choices of pivot.
*/
```

```
#include <functional>
#include <iterator>
#include <algorithm>
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

/**
 * Function: void Selection(RandomIterator begin, RandomIterator middle,
 *                          RandomIterator end);
 * Usage: Selection(v.begin(), v.begin() + 5, v.end());
 * ------------------------------------------------------------
 * Rearranges the elements of the range such that the element
 * at position middle is the element that would be there if the
 * elements were sorted, the elements [begin, middle) are all
 * no greater than middle, and the elements [middle + 1, end)
 * are all no less than middle.
 */
template <typename RandomIterator>
void Selection(RandomIterator begin, RandomIterator middle, RandomIterator end);

/**
 * Function: void Selection(RandomIterator begin, RandomIterator middle,
 *                          RandomIterator end, Comparator comp);
 * Usage: Selection(v.begin(), v.begin() + 5, v.end());
 * ------------------------------------------------------------
 * Rearranges the elements of the range such that the element
```

```
 * at position middle is the element that would be there if the
 * elements were sorted, the elements [begin, middle) are all
 * no greater than middle, and the elements [middle + 1, end)
 * are all no less than middle.  Comparisons are done according
 * to comp, which should be a strict weak ordering.
 */
template <typename RandomIterator, typename Comparator>
void Selection(RandomIterator begin, RandomIterator middle, RandomIterator end,
               Comparator comp);

/* * * * * Implementation Detail Below This Point * * * * */
namespace selection_detail {
  /**
   * Function: Partition(RandomIterator begin, RandomIterator end,
   *                     Comparator comp);
   * Usage: Partition(begin, end, comp);
   * ------------------------------------------------------------
   * Applies the partition algorithm to the range [begin, end),
   * assuming that the pivot element is pointed at by begin.
   * Comparisons are performed using comp.  Returns an iterator
   * to the final position of the pivot element.
   */
  template <typename RandomIterator, typename Comparator>
  RandomIterator Partition(RandomIterator begin, RandomIterator end,
                           Comparator comp) {
    /* To handle strange edge cases, check whether the range is
     * empty or contains exactly one element.
     */
    if (begin == end || begin + 1 == end)
      return begin; // Pivot element, if it exists, is in the front.
```

```
/* The following algorithm for doing an in-place partition is
 * one of the most efficient partitioning algorithms.  It works
 * by maintaining two pointers, one on the left-hand side of
 * the range and one on the right-hand side, and marching them
 * inward until each one encounters a mismatch.  Once the
 * mismatch is found, the mismatched elements are swapped and
 * the process continues.  When the two endpoints meet, we have
 * found the ultimate location of the pivot.
 *
 * This algorithm was originally developed by C.A.R. Hoare,
 * who also invented quicksort.
 */
RandomIterator lhs = begin + 1;
RandomIterator rhs = end - 1;
while (true) {
  /* Keep marching the right-hand side inward until we encounter
   * an element that's too small to be on the left or we hit the
   * left-hand pointer.
   */
  while (lhs < rhs && !comp(*rhs, *begin))
    --rhs;
  /* Keep marching the left-hand side forward until we encounter
   * a the right-hand side or an element that's too big to be
   * on the left-hand side.
   */
  while (lhs < rhs && comp(*lhs, *begin))
    ++lhs;

  /* Now, if the two pointers have hit one another, we've found
   * the crossover point and are done.
   */
```

```
      if (lhs == rhs) break;

      /* Otherwise, exchange the elements pointed at by rhs and lhs. */
      std::iter_swap(lhs, rhs);
    }
    /* When we have reached this point, the two iterators have crossed
     * and we have the partition point.  However, there is one more edge
     * case to consider.  If the pivot element is the smallest element
     * in the range, then the two pointers will cross over on the first
     * step.  In this case, we don't want to exchange the pivot element
     * and the crossover point.
     */
    if (comp(*begin, *lhs))
      return begin;

    /* Otherwise, exchange the pivot and crossover, then return the
     * crossover.
     */
    std::iter_swap(begin, lhs);
    return lhs;
  }
}

/* Actual implementation of the Selection algorithm. */
template <typename RandomIterator, typename Comparator>
void Selection(RandomIterator begin, RandomIterator middle, RandomIterator end,
               Comparator comp) {
  /* Edge case checking - if there are no elements or exactly one element,
   * we're done.
   */
  if (begin == end || begin + 1 == end)
```

```
  return;

/* The first step in this algorithm is to break the input up into groups of
 * five elements, sort each, and then recursively invoke the function to get
 * the median of those medians.  For simplicity, we'll do this by sorting
 * each block of five, then moving the median to the front of the list.
 * From there, we can recursively invoke Selection by just delineating the
 * front of the list.
 */
RandomIterator nextMedianPosition = begin;

/* Due to idiosyncrasies of the STL, while it's technically permissible to
 * walk an iterator an arbitrary number of steps off the end of a container,
 * it's not supported in most compilers.  Consequently, we'll compute the
 * position of the last block and iterate until we hit it.
 */
RandomIterator lastBlockStart = begin + ((end - begin) / 5) * 5;
for (RandomIterator blockStart = begin; blockStart != lastBlockStart;
     blockStart += 5, ++nextMedianPosition) {
  /* Sort the block. */
  std::sort(blockStart, blockStart + 5, comp);

  /* Move the median to the next open median slot. */
  std::iter_swap(nextMedianPosition, blockStart + 2);
}

/* Now, if the last block isn't empty (i.e. lastBlockStart isn't
 * the end iterator), sort it as well and move the median to the
 * front.
 */
if (lastBlockStart != end) {
```

```
  std::sort(lastBlockStart, end);
  std::iter_swap(nextMedianPosition, lastBlockStart + (end - lastBlockStart) / 2);

  /* Bump this value so we include it in the recursion. */
  ++nextMedianPosition;
}

/* Recursively invoke the function on the medians to get the median of medians. */
RandomIterator medianOfMedians = begin + (nextMedianPosition - begin) / 2;
Selection(begin, medianOfMedians, nextMedianPosition);

/* Swap the new median down to the front and invoke partition to get the
 * crossover point.
 */
std::iter_swap(medianOfMedians, begin);
RandomIterator crossoverPoint = selection_detail::Partition(begin, end, comp);

/* If the crossover point is the element we're looking for, that's great!
 * We're done.
 */
if (crossoverPoint == middle) return;

/* Otherwise, if the crossover point is to the right of the middle, recursively
 * invoke the function on the lower half.
 */
else if (crossoverPoint > middle)
  Selection(begin, middle, crossoverPoint, comp);

/* Otherwise, the crossover point is to the left.  Recursively invoke the function
 * on the upper half.
 */
```

```
  else
    Selection(crossoverPoint, middle, end, comp);
}

/* The version of this function without a comparator just uses std::less as a
 * comparator.
 */
template <typename RandomIterator>
void Selection(RandomIterator begin, RandomIterator middle, RandomIterator end) {
  /* The final parameter to this function is a bit tricky, but it means
   * "look up the type of the element being iterated over, then pass it
   * in as a parameter to the std::less comparator type."
   */
  return Selection(begin, middle, end,
                   std::less<typename std::iterator_traits<RandomIterator>::value_type>());
}

void readFile(vector<int> &input);
void readFile (vector<int> &input) {

    int j;
    ifstream myFile("input.txt");
    while (!myFile.eof()) {
        myFile >> j;
        input.push_back(j);
    }
}

int main() {

    vector<int> input;
```

```
    readFile(input);
    input.pop_back();

    float t1 = clock();
    Selection(input.begin(),input.begin()+2500,input.end());
    float diff = (t2-t1)/CLOCKS_PER_SEC;

    cout << diff << " sec" << endl;



    //for (int i = 0; i<input.size();i++){
        //cout << input[i]<<endl;
    //}
 system("pause");
}
```

Fibonacci Algorithms 1-5 [4]

```
#include<stdio.h>
#include<iostream>
#include<algorithm>

using namespace std;



int  fib1(int n)
{
   if ( n <= 1 )
      return n;
   return fib1(n-1) + fib1(n-2);
}

int  fib2(int n)
{
  /* Declare an array to store fibonacci numbers. */
  int  f[n+1];
  int i;

  /* 0th and 1st number of the series are 0 and 1*/
  f[0] = 0;
  f[1] = 1;

  for (i = 2; i <= n; i++)
  {
      /* Add the previous 2 numbers in the series
         and store it */
      f[i] = f[i-1] + f[i-2];
  }
```

```
  return f[n];
}

int  fib3(int n)
{
  int  a = 0, b = 1, c, i;
  if( n == 0)
    return a;
  for (i = 2; i <= n; i++)
  {
     c = a + b;
     a = b;
     b = c;
  }
  return b;
}

/* Helper function that multiplies 2 matricies F and M of size 2*2, and
  puts the multiplication result back to F[][] */
void multiply(int  F[2][2], int  M[2][2]);

/* Helper function that calculates F[][] raise to the power n and puts the
  result in F[][]
  Note that this function is desinged only for fib() and won't work as general
  power function */
void power(int  F[2][2], int n);

int  fib4(int n)
{
  int  F[2][2] = {{1,1},{1,0}};
```

```
  if(n == 0)
     return 0;
  power(F, n-1);

  return F[0][0];
}

void multiply(int  F[2][2], int  M[2][2])
{
  int  x =  F[0][0]*M[0][0] + F[0][1]*M[1][0];
  int  y =  F[0][0]*M[0][1] + F[0][1]*M[1][1];
  int  z =  F[1][0]*M[0][0] + F[1][1]*M[1][0];
  int  w =  F[1][0]*M[0][1] + F[1][1]*M[1][1];

  F[0][0] = x;
  F[0][1] = y;
  F[1][0] = z;
  F[1][1] = w;
}

void power(int  F[2][2], int n)
{
  int  i;
  int  M[2][2] = {{1,1},{1,0}};

  // n - 1 times multiply the matrix to {{1,0},{0,1}}
  for ( i = 2; i <= n; i++ )
     multiply(F, M);
}

void multiply1(int  F[2][2], int  M[2][2]);
```

```
void power1(int  F[2][2], int n);

/* function that returns nth Fibonacci number */
int  fib5(int n)
{
  int  F[2][2] = {{1,1},{1,0}};
  if(n == 0)
    return 0;
  power1(F, n-1);
  return F[0][0];
}

/* Optimized version of power() in method 4 */
void power1(int  F[2][2], int n)
{
  if( n == 0 || n == 1)
      return;
  int  M[2][2] = {{1,1},{1,0}};

  power1(F, n/2);
  multiply1(F, F);

  if( n%2 != 0 )
     multiply1(F, M);
}

void multiply1(int  F[2][2], int  M[2][2])
{
  int  x =  F[0][0]*M[0][0] + F[0][1]*M[1][0];
  int  y =  F[0][0]*M[0][1] + F[0][1]*M[1][1];
```

```
  int  z =  F[1][0]*M[0][0] + F[1][1]*M[1][0];
  int  w =  F[1][0]*M[0][1] + F[1][1]*M[1][1];

  F[0][0] = x;
  F[0][1] = y;
  F[1][0] = z;
  F[1][1] = w;
}


int main ()
{
  int n = 45;
  int  f = 0;
  float t1;
  float t2;
  float diff;

  t1 = clock();
  f = fib1(n);
  t2 = clock();
  diff = (t2-t1)/CLOCKS_PER_SEC;
  printf("%d ", f);
  cout << "Fib1 took "<< diff<<" sec"<<endl;

  t1 = clock();
  f = fib2(n);
  t2 = clock();
  diff = (t2-t1)/CLOCKS_PER_SEC;
  printf("%d ", f);
  cout << "Fib2 took "<< diff<<" sec"<<endl;
```

```
  t1 = clock();
  f = fib3(n);
  t2 = clock();
  diff = (t2-t1)/CLOCKS_PER_SEC;
  printf("%d ", f);
  cout << "Fib3 took "<< diff<<" sec"<<endl;

  t1 = clock();
  f = fib4(n);
  t2 = clock();
  diff = (t2-t1)/CLOCKS_PER_SEC;
  printf("%d ", f);
  cout << "Fib4 took "<< diff<<" sec"<<endl;

  t1 = clock();
  f = fib5(n);
  t2 = clock();
  diff = (t2-t1)/CLOCKS_PER_SEC;
  printf("%d ", f);
  cout << "Fib5 took "<< diff<<" sec"<<endl;


  getchar();
  return 0;
}
```

Bibliography

[1] Wikipedia, "Fibonacci Number," Wikimedia, 15 May 2012. [Online]. Available: http://en.wikipedia.org/wiki/Fibonacci_number. [Accessed 1 September 2012].

[2] J. Jones and W. Wilson, Science, an Incomplete Education ISBN 978-0-7394-7582-9 P.544, Ballentine Books, 2006.

[3] I. Pittas, "Order Statistics in Digital Processing," *IEEE*, vol. ECE, no. 572, 2011.

[4] Stein, "Design and Analsysis of Algorithms," 6 January 1996. [Online]. Available: http://www.ics.uci.edu/~eppstein/161/960109.html. [Accessed 10 September 2012].

[5] Geek4u, "Kth Largest Element," GeeksforGeeks, 2012. [Online]. Available: http://www.geeksforgeeks.org/archives/2392. [Accessed 10 September 2012].

[6] Wikipedia, "Selection Algorithm," Wikimedia, August 2012. [Online]. Available: http://en.wikipedia.org/wiki/Selection_algorithm. [Accessed 10 September 2012].

[7] K. Schwarz, "Selection.hh," Standford, 2010. [Online]. Available: http://www.keithschwarz.com/interesting/code/median-of-medians/Selection.hh.html. [Accessed 10 September 2012].