

An efficient implementation of Blum, Floyd, Pratt, Rivest, and Tarjan's worst-case linear selection algorithm

Derrick Coetzee, webmaster@moonflare.com

January 22, 2004

Contents

1	Introduction	1
2	The Average-Case Linear, Fast Algorithm	3
3	The Worst-Case Linear Algorithm	5
4	The Key: A Fast Median-of-5 Function	7
5	The partition function	9
6	Driver and Benchmarking	10
7	Conclusions and Possible Improvements	13
8	Indexes	13
8.1	Code Chunks	13
8.2	Identifiers	13

1 Introduction

In 1973, Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan wrote a paper entitled *Time bounds for selection* which explored the problem of selecting the k th smallest element in an array, and demonstrated an explicit algorithm for solving it in worst-case $O(n)$ time, using only comparisons. This algorithm has been republished in many other works, and this implementation is based on the description given in Cormen, Leiserson, Rivest, and Stein's important textbook *Introduction to Algorithms*.

In practice, the algorithm is typically not used, nor should be in its naïve form, because there is a much simpler algorithm, a close relative of quicksort,

which solves the algorithm in very fast average-case time, although, like quick-sort, it can potentially take a very long time. This algorithm is also demonstrated here, for comparison. I have made several small improvements to the worst-case linear algorithm, however, that close the gap.

This code is written in C++, but in a procedural fashion; it is essentially C with some sugar on top. The source file in outline is:

```
2  <select.cpp 2>≡
    <header files 4b>

    <using directives 4c>

    <median5 7>

    <partition 9>

    <select 5>

    <selectRandom 3>

    <driver and benchmark code 10a>
```

2 The Average-Case Linear, Fast Algorithm

Let's recall the algorithm for quicksort:

```
quicksort(a, begin, end)
  if length a > 1 then
    locate a pivot a[i]
    p := partition(a, begin, end, i)
    quicksort(a, begin, p-1)
    quicksort(a, p, end)
```

Here, `partition` is a function that finds the final sorted position of `a[i]`, and moves all elements less than `a[i]` to indexes below this position, and all elements greater than or equal to `a[i]` to indexes above this position. It does this in linear time, and it returns the new index of the pivot. As long as each partition has some significant proportion of the elements in it, the two recursive calls have considerably less work to do, and quicksort takes $O(n \log n)$ time.

Now let's look at `selectRandom`. Instead of returning the position of the k th smallest element, this function permutes the array to move it into position k :

```
3  <selectRandom 3>≡ (2)
    template <typename T>
    void selectRandom(T a[], int size, int k) {
      <select base case 4a>
      int pivotIdx = partition(a, size, rand() % size);
      if (k != pivotIdx) {
        if (k < pivotIdx) {
          selectRandom(a, pivotIdx, k);
        } else /* if (k > pivotIdx) */ {
          selectRandom(a + pivotIdx + 1, size - pivotIdx - 1, k - pivotIdx - 1);
        }
      }
    }
}
```

Defines:

`selectRandom`, used in chunk 11.

Uses `partition` 9.

Like a typical randomized quicksort, we perform a partition around a randomly chosen pivot element. However, instead of recursing on both partitions, we already know which one will contain the k th smallest element, because only the `pivotIdx` smallest elements are found below the pivot after the partition, and the others above.

If $k = \text{pivotIdx}$, then we are done, since, as noted above, `partition` places the pivot into its correct sorted position. Like most functions in this program, the function is templated over the type of the elements in the array.

As in quicksort, this algorithm becomes needlessly inefficient for the sub-problems involving very small lists, and so we simply use insertion sort for such cases, moving the k th smallest element to position k for all k :

```
4a  <select base case 4a>≡ (3 5)
      if (size < 5) {
          for (int i=0; i<size; i++)
              for (int j=i+1; j<size; j++)
                  if (a[j] < a[i])
                      swap(a[i], a[j]);
          return;
      }
```

The swap function here is a template function from the C++ standard library and will be used again. It requires a header file and a using directive:

```
4b  <header files 4b>≡ (2) 10c▷
      #include <algorithm>

4c  <using directives 4c>≡ (2)
      using std::swap;
```

3 The Worst-Case Linear Algorithm

Despite its speed, if the random number generator collaborates with the program generating the array in some mystical event, the above algorithm can become quadratic. For example, it might always choose the smallest value as a pivot. The algorithm described in this section is provably worst-case linear, but generally at least twice as slow in practice. Like `selectRandom`, `select` permutes the array to place the k th largest value in `a[k]`.

```

5  <select 5>≡ (2)
    template <typename T>
    void select(T a[], int size, int k) {
        <select base case 4a>

        <select: find a pivot 6>

        int newMOMIdx = partition(a, size, MOMIdx);
        if (k != newMOMIdx) {
            if (k < newMOMIdx) {
                select(a, newMOMIdx, k);
            } else /* if (k > newMOMIdx) */ {
                select(a + newMOMIdx + 1, size - newMOMIdx - 1, k - newMOMIdx - 1);
            }
        }
    }

```

Defines:

`select`, used in chunks 6, 11, and 12a.

Uses `MOMIdx` 6 and `partition` 9.

From just the above, the function is nearly identical to `selectRandom`. The difference here is, instead of relying on fate, we seek to find a good pivot, `a[MOMIdx]`, which will always make each partition at least $3/10$ the size of the array. To do this, we first divide the array up into small groups of 5 elements each. We then use the `median5` function to find the median for each, and we move these medians into a block of values at the beginning of the array:

```
6  <select: find a pivot 6>≡ (5)
    int groupNum = 0;
    int* group = a;
    for( ; groupNum*5 <= size-5; group += 5, groupNum++) {
        swap(group[median5(group)], a[groupNum]);
    }
    int numMedians = size/5;
    // Index of median of medians
    int MOMIdx = numMedians/2;
    select(a, numMedians, MOMIdx);
```

Defines:

- `group`, never used.
- `groupNum`, never used.
- `MOMIdx`, used in chunk 5.
- `numMedians`, never used.

Uses `median5` 7 and `select` 5.

Putting the medians in a block simplifies further operations and increases their locality of reference. Having done this, we use `select` itself to identify the median of these median-of-5's, and this will be our pivot.

To show that it is in fact a good pivot, note that there if there are n elements in a , there are about $n/5$ median-of-5 values, and half of them, or $n/10$ elements, are less than the median of them. Moreover, for every one of these elements, our original array contains 2 elements less than it, and so in total there are at least $2(n/10) + n/10 = 3n/10$ elements less than our pivot choice, or 30% of the elements.

To verify that the runtime is linear, we can write a recursion relation for the worst-case running time:

$$T(1) = O(1).$$

$$T(n) = O(n) + T(n/5) + T(7n/10).$$

Informally, it suffices to note that $1/5 + 7/10 = 9/10 < 1$, and whenever we have $T(n) = O(n) + T(cn)$ for $0 < c < 1$, we in fact have $T(n) = O(n)$.

4 The Key: A Fast Median-of-5 Function

If we profile the program, we quickly discover that the function which locates the median of 5 values is the bottleneck, called a huge number of times and taking about half the execution time. Ideally, this function would be written in clever assembly to avoid any unnecessary memory operations and exploit special instructions. To preserve portability, however, I merely wrote it in a way that encourages `gcc` to allocate registers efficiently.

First, we load the five values into five local variables, which allows the compiler to put most of them in registers:

```
7  <median5 7>≡ (2) 8a>
    template <typename T>
    int median5(T* a) {
        int a0 = a[0];
        int a1 = a[1];
        int a2 = a[2];
        int a3 = a[3];
        int a4 = a[4];
```

Defines:

`median5`, used in chunk 6.

Next, we literally perform insertion sort on the five registers, swapping register values until `a0` through `a4` are in order. Because the memory is barely touched, this is fast:

```
8a  <median5 7>+≡ (2) <7 8b>
    if (a1 < a0)
        swap(a0, a1);
    if (a2 < a0)
        swap(a0, a2);
    if (a3 < a0)
        swap(a0, a3);
    if (a4 < a0)
        swap(a0, a4);

    if (a2 < a1)
        swap(a1, a2);
    if (a3 < a1)
        swap(a1, a3);
    if (a4 < a1)
        swap(a1, a4);

    if (a3 < a2)
        swap(a2, a3);
    if (a4 < a2)
        swap(a2, a4);
```

Although we now have the median value, we still need the median index. Although we could have watched the median index throughout the computation, on my machine the extra registers required for this caused a great deal of spilling and slowdown. Instead, we simply compare the median value to each of the original array elements, returning the index that matches:

```
8b  <median5 7>+≡ (2) <8a>
    if (a2 == a[0])
        return 0;
    if (a2 == a[1])
        return 1;
    if (a2 == a[2])
        return 2;
    if (a2 == a[3])
        return 3;
    // else if (a2 == a[4])
        return 4;
}
```

This algorithm runs very quickly in practice. On my 3.3GhZ Pentium 4 machine with profiling enabled, each call takes about half a microsecond.

5 The partition function

The partition algorithm is the same one used in quicksort, essentially as described in *Introduction to Algorithms*. It takes the most time second only to `median5`, about 33% of the run-time, and for the simpler algorithm takes nearly all of its time.

Partition functions by positioning two pointers at the beginning of the array, the load pointer and the store pointer. The load pointer advances through the array, finding all values less than the pivot value, and swapping them into the initial segment of the array. The store pointer keeps track of where the next one goes, and its final position is where the pivot's final position is. The pivot is kept out of the way during the computation by moving it to the end of the array.

```
9  <partition 9>≡ (2)
    template <typename T>
    int partition(T a[], int size, int pivot) {
        int pivotValue = a[pivot];
        swap(a[pivot], a[size-1]);
        int storePos = 0;
        for(int loadPos=0; loadPos < size-1; loadPos++) {
            if (a[loadPos] < pivotValue) {
                swap(a[loadPos], a[storePos]);
                storePos++;
            }
        }
        swap(a[storePos], a[size-1]);
        return storePos;
    }
```

Defines:

`partition`, used in chunks 3 and 5.

It should be easy to convince yourself that when this process is complete, only values less than the pivot precede it, and only greater than or equal values follow it.

6 Driver and Benchmarking

I performed timings of both algorithms, as well as compared them to the more obvious approach of sorting and then indexing the desired element. The driver code in outline is:

```
10a <driver and benchmark code 10a>≡ (2)
    <makeRandomArray 10b>
    <main 11>
```

The function `makeRandomArray` simply fills in an array with random results of `rand()`. It always seeds with a constant seed before beginning so that all three algorithms work on exactly the same list:

```
10b <makeRandomArray 10b>≡ (10a)
    const int SEED = 352302;

    void makeRandomArray(int* a, int size) {
        srand(SEED);
        for (int i=0; i<size; i++) {
            a[i] = rand();
        }
    }
```

Defines:

```
    makeRandomArray, used in chunk 11.
    SEED, never used.
```

The random number generation functions require `stdlib.h`:

```
10c <header files 4b>+≡ (2) <4b 12b>
    #include <stdlib.h>
```

The program takes a single argument, the size of the array to test. It times and stores the result of each of the three methods, resetting the array to the same list of values before each one:

```

11  <main 11>≡ (10a) 12a▶
    int main(int argc, char* argv[]) {
        int size = atoi(argv[1]);
        int* a = new int[size];

        makeRandomArray(a, size);
        clock_t selectBegin = clock();
        select(a, size, size/2);
        clock_t selectEnd = clock();
        int selectResult = a[size/2];

        makeRandomArray(a, size);
        clock_t selectRandomBegin = clock();
        selectRandom(a, size, size/2);
        clock_t selectRandomEnd = clock();
        int selectRandomResult = a[size/2];

        makeRandomArray(a, size);
        clock_t sortBegin = clock();
        std::sort(a, a+size);
        clock_t sortEnd = clock();
        int sortResult = a[size/2];

        delete[] a;

```

Defines:

`main`, never used.

Uses `makeRandomArray` 10b, `select` 5, and `selectRandom` 3.

At the end, we make sure the methods give consistent results, and print the timings in milliseconds for comparison:

```
12a  <main 11>+≡ (10a) <11
      using std::cout;
      using std::endl;

      if (!(selectResult == selectRandomResult &&
            selectRandomResult == sortResult)) {
          cout << "Inconsistent result" << endl;
      }

      cout << "Select time (ms): " << double(selectEnd-selectBegin)*1000/CLOCKS_PER_SEC << endl;
      cout << "Randomized select time (ms): " << double(selectRandomEnd-selectRandomBegin)*1000/CLOCKS_PER_SEC << endl;
      cout << "Sort time (ms): " << double(sortEnd-sortBegin)*1000/CLOCKS_PER_SEC << endl;

      return 0;
    }

```

Uses `select 5`.

The I/O here requires another header:

```
12b  <header files 4b>+≡ (2) <10c
      #include <iostream>

```

7 Conclusions and Possible Improvements

Despite the optimizations, the simpler algorithm typically ran at least twice as quickly as the more sophisticated one with the better worst-case bound. I would advise that the simpler one be used in applications.

Hybrid ideas are possible. We do a lot of work, including a number of recursive calls, to obtain the pivot in `select`, but this may be excessive. We might limit the call depth to some value k , like 2 or 3, and then return a random element. This destroys worst-case linear time, but is more likely to be linear than the simpler algorithm, and is faster.

8 Indexes

8.1 Code Chunks

<driver and benchmark code 10a>
 <header files 4b>
 <main 11>
 <makeRandomArray 10b>
 <median5 7>
 <partition 9>
 <select 5>
 <select base case 4a>
 <select.cpp 2>
 <select: find a pivot 6>
 <selectRandom 3>
 <using directives 4c>

8.2 Identifiers

group: 6
 groupNum: 6
 main: 11
 makeRandomArray: 10b, 11
 median5: 6, 7
 MOMIdx: 5, 6
 numMedians: 6
 partition: 3, 5, 9
 SEED: 10b
 select: 5, 6, 11, 12a
 selectRandom: 3, 11