Andrew Westlake
k582r363
CS560 Design and Analysis of Algorithms
April 23rd, 2019
Analysis of Quicksort

Analysis of Quicksort

Quicksort is an excellent sorting algorithm in general and it is especially useful for programmers since it doesn't require the allocation of any extra memory. The sorting happens in place, and as long as swapping elements is non-trivial, should be a good choice for any data structure that allows comparison. That being said, the raw form of Quicksort suffers from some fatal flaws around the edge cases.

Quicksort fails to sort in O(N log N) time when the input is already sorted. The raw algorithm does not choose the pivot element in an effective way, so it ends up running without any early-outs resulting in O(N^2) time.

For this reason, Randomized Quicksort was developed. Instead of choosing the pivot element in a deterministic way, Randomized Quicksort instead chooses a random element as the pivot. This allows the algorithm to achieve O(N log N) time even when the array is sorted beforehand.

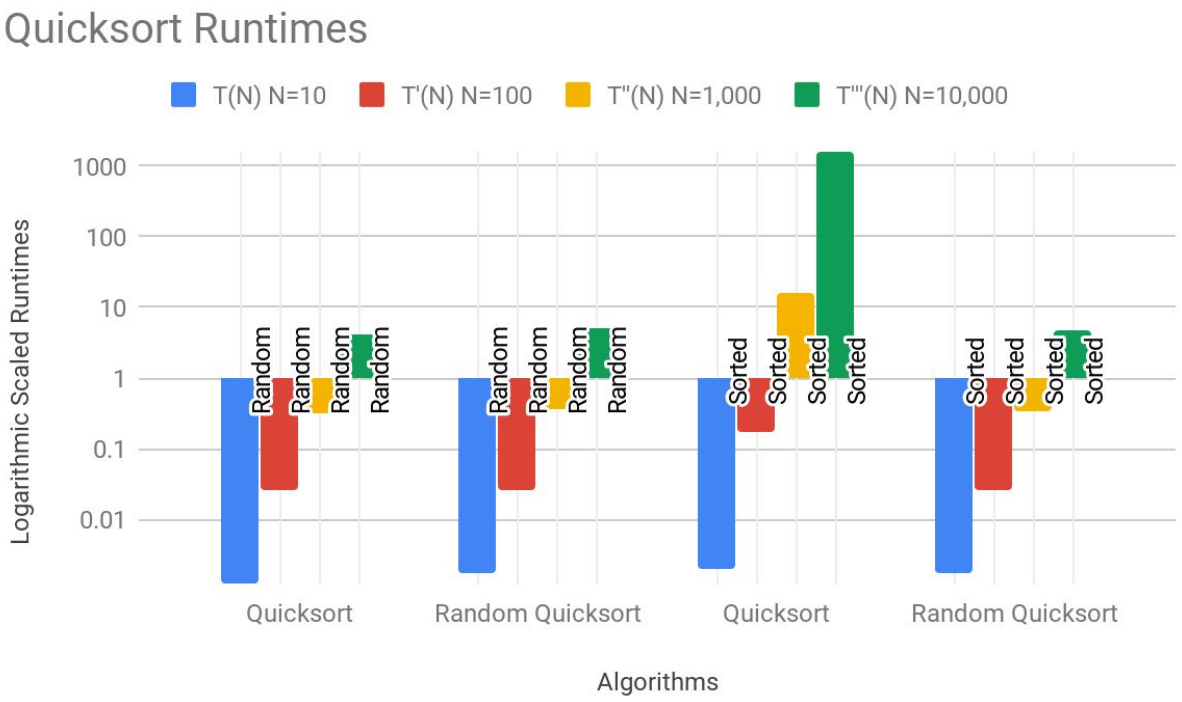
It's important to note that Randomized Quicksort by no means eliminates the edge cases, it just redistributes them. It is still just as possible to get O(N^2) time with Randomized Quicksort, however, you would have to be very unlucky for this to happen. This algorithm is an important improvement because in real-world programs, the input array is much more likely to be sorted than be in just the right configuration to trigger Randomized Quicksort's worst case.

As a demonstration, running the benchmarks for the relevant cases yields the following results:

Algorithm	Input Array	Theoretical Bound	T(N) ms	T'(N) ms	T''(N) ms	T'''(N) ms
Quicksort	Random	O(N log N)	0.001211	0.026524	0.321730	4.279060
Random Quicksort	Random	O(N log N)	0.001731	0.025310	0.371036	5.204660
Quicksort	Sorted	O(N^2)	0.002061	0.172965	16.50790	1608.250
Random Quicksort	Sorted	O(N log N)	0.001708	0.025970	0.332871	4.720000

Now, these results can be difficult to visualize with standard scales because O(N^2) runtime blows up significantly faster than O(N log N) for larger values of N. In order to

show these results in a meaningful way, I made the vertical axis a logarithmic scale. As a result, the runtimes are rebased around the median value and displayed as 100x the median value or 1000x the median value. This scale may not fully convey how stark the difference is between O(N^2) and O(N log N), but it allows us to compare the differences a bit easier.



As you can see, Quicksort and Random Quicksort perform about the same when the input is random, but we see the worst of Quicksort once the input is sorted. Randomized Quicksort runs slightly slower than Quicksort because of the overhead of pseudo-random number generation, but as you can see, that overhead is clearly preferable when the input is already sorted.

The performance difference of Quicksort with sorted input when N = 10 is negligible, but we begin to see a trend around N = 100. The performace at N = 100 is noticeably worse than the others. This trend continues all the way to N = 10,000 where the performance is significantly worse than the others. To put this in perspective, Quicksort is 341x worse than Randomized Quicksort at N = 10,000.

From these results we can name Randomized Quicksort the winner for most practical applications.

The code is included in this document for reference

```hpp
// include/sort.hpp
#pragma once

#include <vector>
#include <random>
#include <algorithm>

#include "utils.hpp"

extern void quicksort(IntVec& integers);
extern void random_quicksort(
   std::default_random_engine& generator, IntVec& integers
);
```

```cpp
// src/sort.cpp

#include "sort.hpp"

#include <algorithm>

static size_t partition(IntVec& integers, size_t p, size_t r) {
   int& x = integers.at(r - 1);
   size_t i = p - 1;

   for (auto j = p; j < r - 1; j++) {
       if (integers.at(j) <= x) {
           i += 1;
           std::swap(integers.at(i), integers.at(j));
       }
   }
   std::swap(integers.at(i + 1), integers.at(r - 1));

   return i + 1;
}

static void quicksort_impl(IntVec& integers, size_t p, size_t r) {
   if (p < r) {
       size_t q = partition(integers, p, r);

       quicksort_impl(integers, p, q);
       quicksort_impl(integers, q + 1, r);
   }
}

void quicksort(IntVec& integers) {
   quicksort_impl(integers, 0, integers.size());
}

static size_t random_partition(
   std::default_random_engine& generator, IntVec& integers, size_t p, size_t r
) {
   // Swap a random element
   std::uniform_int_distribution<size_t> distribution(p, r - 1);
   auto index = distribution(generator);
   std::swap(integers[r - 1], integers[index]);

   return partition(integers, p, r);
}

static void random_quicksort_impl(
   std::default_random_engine& generator, IntVec& integers, size_t p, size_t r
) {
   if (p < r) {
```

```
        size_t q = random_partition(generator, integers, p, r);

        random_quicksort_impl(generator, integers, p, q);
        random_quicksort_impl(generator, integers, q + 1, r);
    }
}

void random_quicksort(std::default_random_engine& generator, IntVec& integers) {
    random_quicksort_impl(generator, integers, 0, integers.size());
}
```

```cpp
// include/utils.hpp
#pragma once

#include <algorithm>
#include <cctype>
#include <chrono>
#include <functional>
#include <iostream>
#include <random>
#include <string>
#include <vector>

using Clock = std::chrono::high_resolution_clock;
using Milliseconds = std::chrono::duration<double, std::milli>;

using IntVec = std::vector<int>;
using SortFunction = std::function<void(IntVec&)>;

struct SortResults {
   SortResults();

   IntVec sorted;
   Clock::duration runtime;
};

extern IntVec generate_random_integers(
   std::default_random_engine& generator,
   size_t n
);
extern IntVec generate_increasing_integers(size_t n, size_t x);

extern SortResults run_sort(const SortFunction& sort_fn, IntVec integers);

extern void print_integers(std::ostream& file, const IntVec& integers);

// String trim utilities
(https://stackoverflow.com/questions/216823/whats-the-best-way-to-trim-stdstring)

// trim from start (in place)
static inline void ltrim(std::string &s) {
   s.erase(s.begin(), std::find_if(s.begin(), s.end(),
           std::not1(std::ptr_fun<int, int>(std::isspace))));
}

// trim from end (in place)
static inline void rtrim(std::string &s) {
   s.erase(std::find_if(s.rbegin(), s.rend(),
           std::not1(std::ptr_fun<int, int>(std::isspace))).base(), s.end());
}
```

```
// trim from both ends (in place)
static inline void trim(std::string &s) {
   ltrim(s);
   rtrim(s);
}
```

```cpp
// src/utils.cpp

#include "utils.hpp"

#include <chrono>

SortResults::SortResults() :
    runtime(0)
{
}

IntVec generate_random_integers(
    std::default_random_engine& generator, size_t n
) {
    constexpr int MAX_RANGE = 10000;

    IntVec integers;
    std::uniform_int_distribution<int> distribution(0, MAX_RANGE);

    for (size_t i = 0; i < n; i++) {
        integers.push_back(distribution(generator));
    }

    return integers;
}
IntVec generate_increasing_integers(size_t n, size_t x) {
    IntVec integers;

    if (n != 0) {
        integers.push_back(n + x);
        for (size_t i = 1; i < n; i++) {
            integers.push_back(n + (i + 1) * x);
        }
    }

    return integers;
}

SortResults run_sort(
    const std::function<void(IntVec&)>& sort_fn,
    IntVec integers
) {
    SortResults results;

    auto start = Clock::now();
    sort_fn(integers);
    auto end = Clock::now();

    results.sorted = std::move(integers);
```

```
    results.runtime = end - start;

    return results;
}

void print_integers(std::ostream& file, const IntVec& integers) {
    file << "[";

    for (int i = 0; i < (integers.size() - 1); i++) {
        file << integers[i] << ", ";
    }

    file << integers.back() << "]" << std::endl;
}
```

```cpp
// src/main.cpp

#include <chrono>
#include <fstream>
#include <iostream>
#include <random>
#include <stdexcept>
#include <string>
#include <vector>

#include "utils.hpp"
#include "sort.hpp"

struct Results {
   IntVec original;

   SortResults quicksort;
   SortResults random_quicksort;
};

Results run(std::default_random_engine& generator, IntVec original_integers) {
   Results results;

   results.quicksort = run_sort(
       [](IntVec& integers) {
           quicksort(integers);
       },
       original_integers
   );

   results.random_quicksort = run_sort(
       [&generator](IntVec& integers) {
           random_quicksort(generator, integers);
       },
       original_integers
   );

   results.original = std::move(original_integers);

   return results;
}

void write_sort_results(std::ostream& output, const SortResults& results) {
   print_integers(output, results.sorted);
   output
       << "Runtime(ms): " <<
std::chrono::duration_cast<Milliseconds>(results.runtime).count()
       << '\n';
}
```

```
void write_results(std::ostream& output, const Results& results) {
   output << "Original: \n";
   print_integers(output, results.original);
   output << '\n';

   output << "Quicksort Results: \n";
   write_sort_results(output, results.quicksort);
   output << '\n';

   output << "Randomized Quicksort Results: \n";
   write_sort_results(output, results.random_quicksort);
   output << '\n';
}

bool prompt_increasing() {
   while (true) {
       std::string line;
       std::cout << "Generate increasing integers? [y/n] ";
       std::getline(std::cin, line);
       trim(line);

       if (line == "y") {
           return true;
       } else if (line == "n") {
           return false;
       } else {
           std::cerr << "Invalid input - must be [y/n]" << std::endl;
       }
   }
}

size_t prompt_n() {
   while (true) {
       try {
           std::string line;
           std::cout << "Enter N: " << std::flush;
           std::getline(std::cin, line);

           size_t n = std::stoul(line);

           return n;
       } catch (const std::invalid_argument& e) {
           std::cerr << "Unable to convert N to an integer\n" << std::endl;
       }
   }
}

size_t prompt_x() {
   while (true) {
```

```
        try {
            std::string line;
            std::cout << "Enter X: " << std::flush;
            std::getline(std::cin, line);

            size_t x = std::stoul(line);

            return x;
        } catch (const std::invalid_argument& e) {
            std::cerr << "Unable to convert X to an integer\n" << std::endl;
        }
    }
}

void run_randomized(std::default_random_engine& generator) {
    size_t n = prompt_n();

    auto integers = generate_random_integers(generator, n);
    auto results = run(generator, integers);

    {
        std::ofstream output_file("output.txt");
        write_results(output_file, results);
    }

    std::cout
        << "The results have been written to output.txt"
        << std::endl;
}

void run_increasing(std::default_random_engine& generator) {
    size_t n = prompt_n();
    size_t x = prompt_x();

    auto integers = generate_increasing_integers(n, x);
    auto results = run(generator, integers);

    {
        std::ofstream output_file("output.txt");
        write_results(output_file, results);
    }

    std::cout
        << "The results have been written to output.txt"
        << std::endl;
}

int main(int argc, char** argv) {
    std::default_random_engine generator(
        std::chrono::system_clock::now().time_since_epoch().count()
```

```
    );

    bool is_increasing = prompt_increasing();

    if (is_increasing) {
        run_increasing(generator);
    } else {
        run_randomized(generator);
    }

    return 0;
}
```