# Quicksort Report

**Author:** Benjamin Friedman

## Table of Contents

## Background

Quicksort is a sorting algorithm that was developed in 1959 by Tony Hoare and was officially published in 1961. It takes a divide-and-conquer approach in which a single pivot is selected from an array, and every element less than the pivot goes to the left of the pivot and every element greater than the pivot goes to the right of the pivot. The pivot is now considered sorted. Then, the halves are sorted meaning the subarray's on either side of the pivot are sorted by recursively calling quicksort on them. Quicksort's worst-case performance is $O(n^2)$ but this can easily be avoided with a good pivot selection. On average, it's performance is $\Theta(nlgn)$ and in practice it sorts faster than other comparison-based sorting algorithms, even fast ones that are also $\Theta(nlgn)$ like mergesort and heapsort. It's also an in-place sorting algorithm which means the items get sorted in place in the array and no other arrays need to get created. This gives it a second advantage over non-in-place sorting algorithms like mergesort because it is more efficient in how much memory it uses. Due to its speed and memory efficiency, it is often used as the go-to sorting algorithm in many computer programs. The pseudocode for the most basic implementation of quicksort is below:

```
QUICKSORT(A, p, r)
    if p < r
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q - 1)
        QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
    x = A[r]
    i = p - 1
    for j = p to r - 1
        if A[j] ≤ x
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    return i + 1
```

As seen above, an array *A* with a starting and ending index *p* and *r* is passed to quicksort. On the first function call, this would be QUICKSORT(A, 1, A.length). Then, the partition procedure is called which selects a pivot and everything less than the pivot goes to the left of the pivot and everything greater than the pivot goes to the right of the pivot. The partition procedure returns the index of the pivot and stores it in *q*. Now that the partition, A[q], is sorted, quicksort is then called on the subarrays to either half. The recursion stops when quicksort is called and *p* is less than or equal to *r*. When all recursive function calls have been completed, the array is sorted.

# Recurrence Analysis

The recurrence for the running time quicksort is

$T(n) = T(q - 1) + T(n - q) + \Theta(n)$.

The $\Theta(n)$ is the running time of the partition procedure, and $T(q - 1)$ and $T(n - q)$ is the running time of the left and right subarrays respectively.

*Worst Case Runtime*

The worst case runtime for quicksort resulting in the slowest running time is when there is worst-case partitioning. Worst-case partitioning is when the subproblem sizes for the left and right subarrays are n - 1 elements and 0 elements. This situation could occur if, for example, the pivot selected is always the rightmost element which could happen if all of the elements in the array are the same or if the array is already sorted in ascending or descending order. The recurrence for the running time of quicksort in this case is

$T(n) = T(n - 1) + T(0) + \Theta(n)$

Since T(0) takes no time, this can be simplified to

$T(n) = T(n - 1) + \Theta(n)$

$T(n) = T(n - 1) + cn$, $c$ some positive constant

Ultimately, this becomes $T(n) = \Theta(n^2)$. The math for this is below followed by the recurrence tree.

$T(n) = cn + c(n - 1) + c(n - 2) + \ldots. 2c + c$

$T(n) = c(n + (n - 1) + (n - 2) + \ldots. 2 + 1)$ …. this is $c(1 + 2 + \ldots n)$

$T(n) = c\frac{n(n+1)}{2} = \frac{cn^2}{2} + \frac{cn}{2} = \Theta(n^2)$

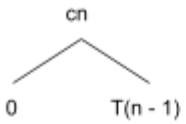
So, $T(n) = \Theta(n^2)$

The formula $\frac{n(n+1)}{2}$ comes from the sum formula for an arithmetic series.

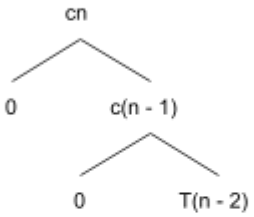
$\sum_{k=1}^{n} k = 1 + 2 + \ldots n = \frac{n(n+1)}{2}$

*Recursion Tree For Quicksort Worst-Case Partitioning*

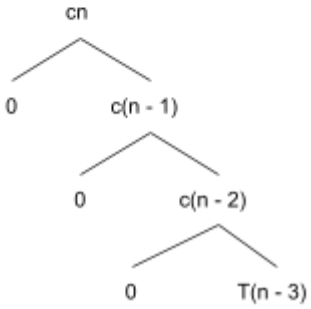
T(n) = T(n - 1) + T(0)+ cn



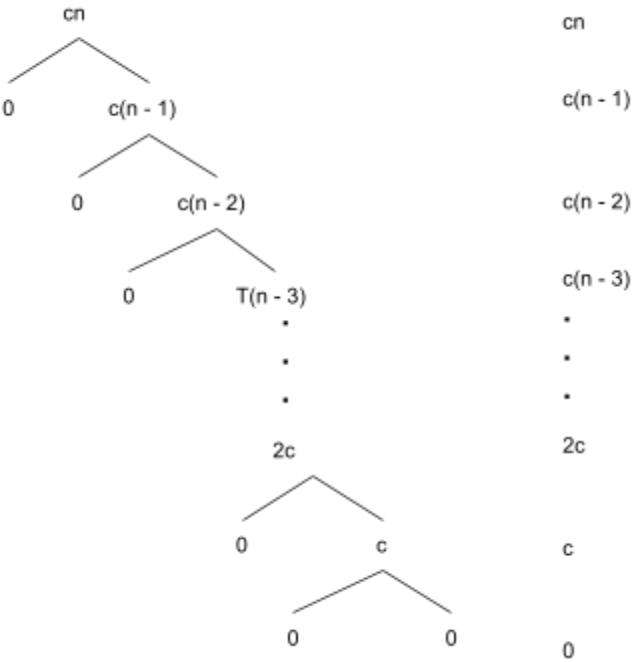

T(n - 1) = T(n - 2) + T(0) + cn




T(n - 2) = T(n - 3) + T(0) + cn




T(1) = T(0) + T(0) + c



*Best Case Runtime*

To achieve the best results for quick sort, balanced partitioning must be achieved. There is a theorem that states

For any α where $0 < \alpha \leq ½$, then $T(n) = T(\alpha n) + T((1 - \alpha)n) + \Theta(n) = \Theta(nlgn)$

The α is selected to produce a proportional split.

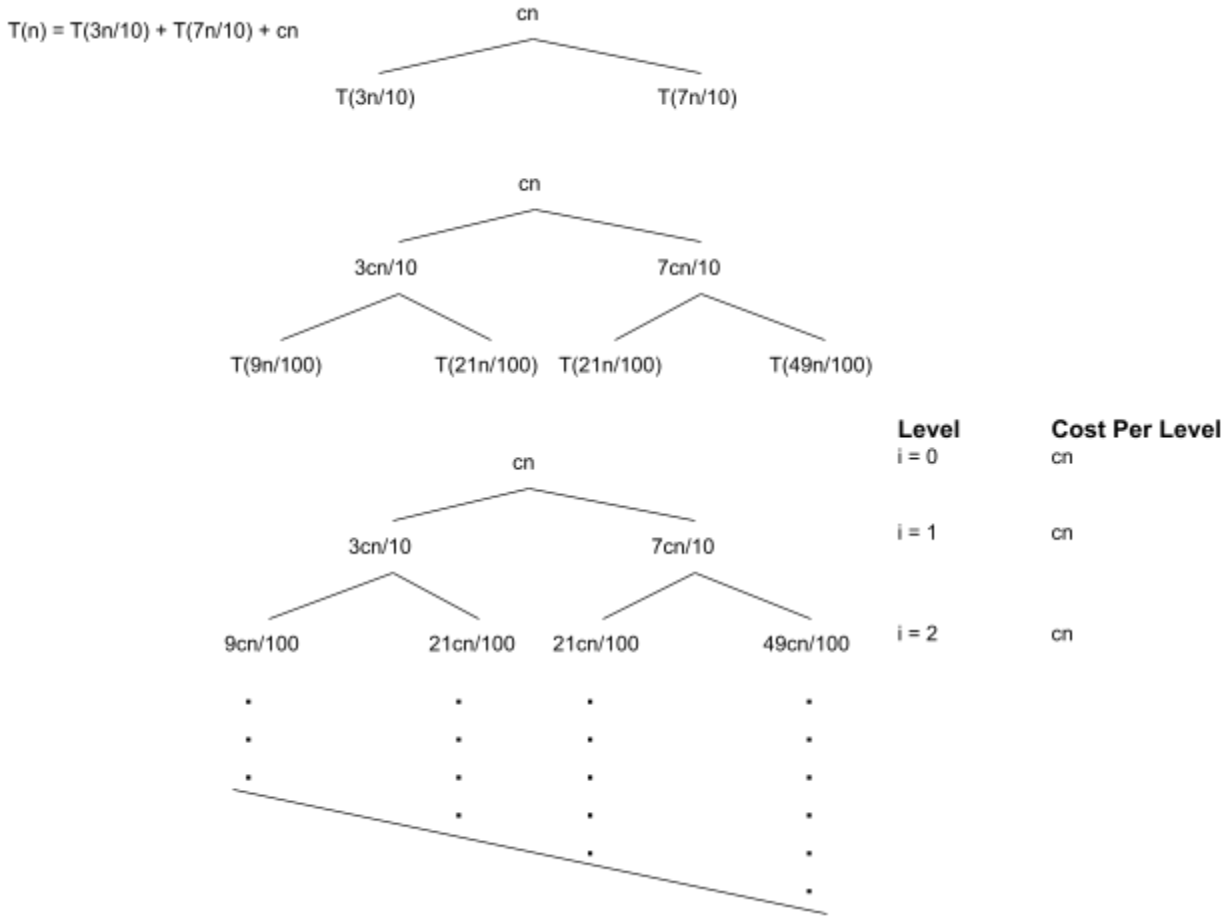
The following example, where $\alpha = \frac{3}{10}$, will demonstrate how Θ(nlgn) is achieved using a proportionals split.

For example, if $\alpha = \frac{3}{10}$ then the proportional split would result in

$$T(n) = T(\tfrac{3}{10}n) + T(\tfrac{7}{10}n) + \Theta(n) = \Theta(nlgn)$$

$T(n) = T(7n/10) + T(3n/10) + \Theta(n)$
$T(n) = T(3n/10) + T(7n/10) + cn$
$T(3n/10) = T(9n/100) + T(21n/100) + 3cn/10$
$T(7n/10) = T(21n/100) + T(49n/100) + 7n/10$

$T(9n/100) = T(27n/1000) + T(63n/1000) + 9cn/100$
$T(21n/100) = T(63n/1000) + T(147n/1000) + 21cn/100$
$T(49n/100) = T(147n/1000) + T(343n/1000) + 49cn/1000$



As seen in the recursion tree on the previous page, the leftmost node will reach the base case sooner than the nodes to the right of it, and the rightmost node will reach the base case last. Before the leftmost node reaches the base case, the cost of each level is cn since the sum of each node at the level number denoted by *i* results in cn. After the leftmost node reaches the base case, the cost of each level is less than cn

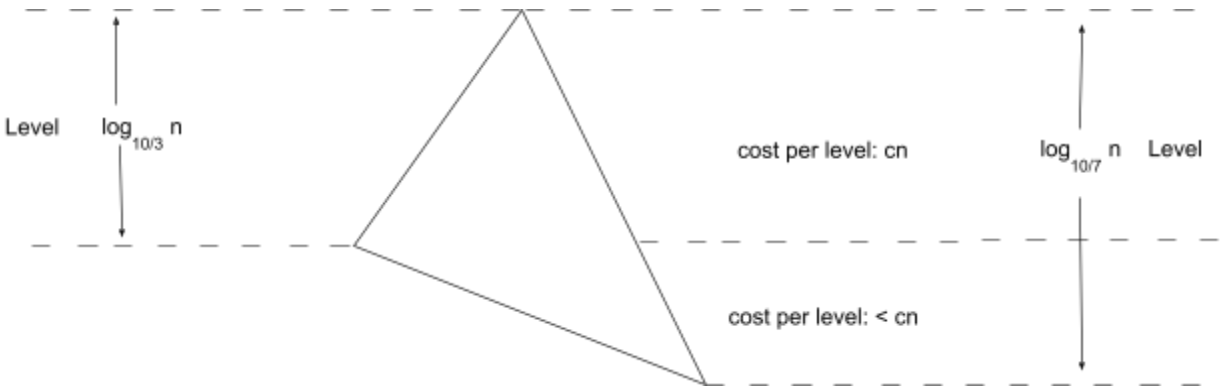
cn = cn
3cn/10 + 7cn/10 = cn
9cn/100 + 21cn/100 + 21cn/100 + 49cn/100 = cn

The problem size of the leftmost nodes follow the pattern $\left(\frac{3}{10}\right)^i n$ and when the base case is reached, the problem size is 1. There will be *i* levels with a cost of cn, where the value of *i* depends on n as seen in the equation below.

$\left(\frac{3}{10}\right)^i n = 1$ $\quad n = \left(\frac{10}{3}\right)^i$ $\quad \log_{10/3} n = i$

The problem size of the rightmost nodes follow the pattern $\left(\frac{7}{10}\right)^j n$ and when the base case is reached, the problem size is 1. There will be *j* levels in total, with some of them having a cost of cn and some of them having a cost of less than cn, where the value of *j* depends on n as seen in the equation below.

$\left(\frac{7}{10}\right)^j n = 1$ $\quad n = \left(\frac{10}{7}\right)^j$ $\quad \log_{10/7} n = j$




The runtime will be greater than $c\text{nlog}_{10/3}\text{n}$ because there are $\log_{10/3}\text{n}$ levels with a cost of cn. The runtime will be less than $c\text{nlog}_{10/7}\text{n}$ because there are $\log_{10/7}\text{n}$ levels in total, but only $c\text{nlog}_{10/3}\text{n}$ levels have a cost of cn, and the remaining $c\text{nlog}_{10/7}\text{n}$ - $\text{cnlog}_{10/3}\text{n}$ levels will have a cost that is less than cn. This provides a lower and upper bound for T(n)

$T(n) \geq c\text{nlog}_{10/3}(n)$ so $T(n) = \Omega(n\log_{10/3}n)$
$T(n) \leq c\text{nlog}_{10/7}(n)$ so $T(n) = O(n\log_{10/7}n)$

Since $T(n) = \Omega(n\log_{10/3}n)$ and $T(n) = O(n\log_{10/7}n)$, then $T(n) = \Theta(n\lg n)$

## Balanced Partitioning

So, ultimately quicksort's performance depends on consistently having good partitioning. Minimizing the occurrence of worst case partitioning can be done by selecting a good pivot. Though there are numerous strategies for this, a common strategy is to randomly select a pivot each time. The pseudocode on the previous page could be modified to implement this.

```
RANDOMIZED-QUICKSORT(A, p, r)
    if p < r
        q = RANDOMIZED-PARTITION(A, p, r)
        RANDOMIZED-QUICKSORT(A, p, q - 1)
        RANDOMIZED-QUICKSORT(A, q + 1, r)
```

```
RANDOMIZED-PARTITION(A, p, r)
    i = RANDOM(p, r)
    exchange A[r] with A[i]
    return PARTITION(A, p, r)
```

```
PARTITION(A, p, r)
    x = A[r]
    i = p - 1
    for j = p to r - 1
        if A[j] ≤ x
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    return i + 1
```

Note that RANDOM is just a function which returns a random number between *p* and *r*.

It is possible for randomized pivot selection to lead to the worst case performance in that it has a non-zero probability but the probability is so low it doesn't realistically happen in practice. For example, for an array of 100 numbers the worst case scenario odds are:

$$\frac{2}{100} \cdot \frac{2}{99} \cdot \frac{2}{98} \cdot \frac{2}{97} \cdots = \frac{2^{100}}{100!}$$

The numerator is 2 because every time there are two possible worst case indexes (the first and the last). The denominator decreases by 1 because every time a new pivot is selected the size of the subarray being sorted has decreased by 1 due to the previous number being sorted.

$2^{100}$ is a very large number and is approximately $1.3 \times 10^{30}$.
However, 100! is much larger and is approximately $9.3 \times 10^{157}$.
As a fraction this is $\frac{1.3 \times 10^{30}}{9.3 \times 10^{157}}$.

If you ignore the mantissas since they're negligible then this is the fraction $\frac{10^{30}}{10^{157}} = \frac{1}{10^{127}}$. This is an incredibly low probability. Additionally, the probability decreases as the amount of numbers being sorted increases. For example, if the size of the array increases from 100 to 150 which isn't that much, the probability becomes $\frac{1}{10^{202}}$. Now imagine if there were 10,000 or 1,000,000 numbers being sorted. The probability would continue to become astronomically smaller.

## Code Demonstration

The output of a program demonstrating how randomized quicksort works along with the code is shown below.

First, the original array is printed.
Then, each call to QUICKSORT is shown with the following:

- The values of *p* and *r* are shown.
- The array as it is currently sorted before partitioning, denoted by A.
- The subarray specific to that recursive call as it is currently sorted before partitioning, denoted by SA.
- How the subarray changes, if at all, throughout the partitioning process, denoted by PA.
- The index *q* returned by the PARTITION function and how the array and subarray are sorted after partitioning.

*Example Output For Randomized Quicksort*

```
Original Array
7  4  12 9  1  20 6  8


QUICKSORT CALL - p: 1  r: 8
BEFORE PARTITIONING
A  - idx: 1  2  3  4  5  6  7  8
          7  4  12 9  1  20 6  8
SA - idx: 1  2  3  4  5  6  7  8
          7  4  12 9  1  20 6  8
DURING PARTITIONING
PA   idx: 1  2  3  4  5  6  7  8   swapped idx 8 and 2
          7  8  12 9  1  20 6  4
PA   idx: 1  2  3  4  5  6  7  8   swapped idx 1 and 5
          1  8  12 9  7  20 6  4
PA   idx: 1  2  3  4  5  6  7  8   swapped idx 2 and 8
          1  4  12 9  7  20 6  8
AFTER PARTITIONING - q: 2
A  - idx: 1  2  3  4  5  6  7  8
          1  4  12 9  7  20 6  8
SA - idx: 1  2  3  4  5  6  7  8
          1  4  12 9  7  20 6  8



QUICKSORT CALL - p: 3  r: 8
BEFORE PARTITIONING
A  - idx: 1  2  3  4  5  6  7  8
          1  4  12 9  7  20 6  8
SA - idx: 3  4  5  6  7  8
          12 9  7  20 6  8
DURING PARTITIONING
PA   idx: 3  4  5  6  7  8   swapped idx 3 and 5
          7  9  12 20 6  8
PA   idx: 3  4  5  6  7  8   swapped idx 4 and 7
          7  6  12 20 9  8
PA   idx: 3  4  5  6  7  8   swapped idx 5 and 8
          7  6  8  20 9  12
AFTER PARTITIONING - q: 5
A  - idx: 1  2  3  4  5  6  7  8
          1  4  7  6  8  20 9  12
```

```
QUICKSORT CALL - p: 3  r: 4
BEFORE PARTITIONING
A  - idx: 1  2  3  4  5  6  7  8
          1  4  7  6  8  20 9  12
SA - idx: 3  4
          7  6
DURING PARTITIONING
PA   idx: 3  4   swapped idx 4 and 3
          6  7
AFTER PARTITIONING - q: 4
A  - idx: 1  2  3  4  5  6  7  8
          1  4  6  7  8  20 9  12
SA - idx: 3  4
          6  7



QUICKSORT CALL - p: 6  r: 8
BEFORE PARTITIONING
A  - idx: 1  2  3  4  5  6  7  8
          1  4  6  7  8  20 9  12
SA - idx: 6  7  8
          20 9  12
DURING PARTITIONING
PA   idx: 6  7  8   swapped idx 8 and 7
          20 12 9
PA   idx: 6  7  8   swapped idx 6 and 8
          9  12 20
AFTER PARTITIONING - q: 6
A  - idx: 1  2  3  4  5  6  7  8
          1  4  6  7  8  9  12 20
SA - idx: 6  7  8
          9  12 20



QUICKSORT CALL - p: 7  r: 8
BEFORE PARTITIONING
A  - idx: 1  2  3  4  5  6  7  8
          1  4  6  7  8  9  12 20
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h.>

void randomized_quicksort(int* A, int p, int r);
int randomized_partition(int* A, int p, int r);
int random(int p, int r);    // returns a random number in range [p, r]
int partition(int* A, int p, int r);
void swap(int* A, int idx1, int idx2);
void printArrayCustom(int* A, int p, int r, char* arrayType, int i1, int i2);
void printArray(int* A, int p, int r);
int gA[] = { -1, 7, 4, 12, 9, 1, 20, 6, 8 };
int sizeA = sizeof(gA) / sizeof(*gA) - 1;

int main(void) {
	srand(time(NULL));
	printf("Original Array\n");
	printArray(gA, 1, sizeA);

	randomized_quicksort(gA, 1, sizeA);

	printf("\n\n\nFinal Array\n");
	printArray(gA, 1, sizeA);
	return 0;
}

void randomized_quicksort(int* A, int p, int r) {
	if (p < r) {
		printf("\n\n\nQUICKSORT CALL - p: %d  r: %d\nBEFORE PARTITIONING\n", p, r);
		printArrayCustom(gA, 1, sizeA, "global", -1, -1);
		printf("\n");
		printArrayCustom(A, p, r, "sub", -1, -1);
		printf("\nDURING PARTITIONING\n");
		int q = randomized_partition(A, p, r);

		printf("AFTER PARTITIONING - q: %d\n", q);
		printArrayCustom(gA, 1, sizeA, "global", -1, -1);
		printf("\n");
		printArrayCustom(A, p, r, "sub", -1, -1);
		printf("\n");

		randomized_quicksort(A, p, q - 1);
		randomized_quicksort(A, q + 1, r);
	}
}
```

```
int randomized_partition(int* A, int p, int r) {
	int i = random(p, r);
	swap(A, r, i);
	if (r != i) {
		printArrayCustom(A, p, r, "partition", r, i);
		printf("\n");
	}
	return partition(A, p, r);
}


int random(int p, int r) {
	return p + rand( ) % (r - p + 1);
}


int partition(int* A, int p, int r) {
	int x = A[r];
	int i = p - 1;
	for (int j = p; j < r; j++) {
		if (A[j] <= x) {
			i++;
			swap(A, i, j);
			if (i != j) {
				printArrayCustom(A, p, r, "partition", i, j);
				printf("\n");
			}
		}
	}
	swap(A, i + 1, r);
	if (i + 1 != r) {
		printArrayCustom(A, p, r, "partition", i + 1, r);
		printf("\n");
	}
	return i + 1;
}

void swap(int* A, int idx1, int idx2) {
	if (idx1 != idx2)
		A[idx1] ^= A[idx2] ^= A[idx1] ^= A[idx2];
}
```

```
void printArrayCustom(int* A, int p, int r, char* arrayType, int i1, int i2) {
	if (!strcmp(arrayType, "global"))
		printf("A  - idx: ");
	else if (!strcmp(arrayType, "sub"))
		printf("SA - idx: ");
	else if (!strcmp(arrayType, "partition"))
		printf("%s", "PA   idx: ");


	for (int i = p; i <= r; i++)
		printf("%-3d", i);

	if (!strcmp(arrayType, "partition"))
		printf(" swapped idx %d and %d", i1, i2);


	printf("\n");
	if (!strcmp(arrayType, "global"))
		printf("          ");
	else if (!strcmp(arrayType, "sub"))
		printf("          ");
	else
		printf("%s", "          ");
	for (int i = p; i <= r; i++)
		printf("%-3d", A[i]);
}

void printArray(int* A, int p, int r) {
	for (int i = p; i <= r; i++)
		printf("%-3d", A[i]);
}
```

# Application

## Application 1: General Purpose Sorting In Programming Languages

Due to quicksort's performance superiority as a sorting algorithm, certain programming languages provide users with built-in functions to sort via quicksort. In C, there is a C standard library function called *qsort* that sorts data using the quicksort algorithm [1]. It is polymorphic and sorts data using a comparison function supplied by the user. The documentation of the function can be seen below from the C documentation [2]:

```
void qsort(void *ptr, size_t count, size_t size, int (*comp)(const void *, const void *));
```

- ptr - pointer to the array to sort
- count - number of elements in the array
- size - size of each element in the array in bytes
- comp - comparison function which returns a negative integer value if the first argument is less than the second, a positive integer value if the first argument is greater than the second and zero if the arguments are equivalent. The signature of the comparison function should be equivalent to the following:

  ```
  int cmp(const void *a, const void *b);
  ```

  The function must not modify the objects passed to it and must return consistent results when called for the same objects, regardless of their positions in the array.

*Description:* Sorts the given array pointed to by ptr in ascending order. The array contains count elements of size bytes. Function pointed to by comp is used for object comparison.

The first ever C version of *qsort* was implemented in Version 6 Unix with a previous version having been in Version 3 Unix as an assembler subroutine [1]. The function was ultimately standardized in C89 (ANSI C), and since *qsort's* original development and standardization, research has been done to create newer and improved versions. For example, upon noticing that *qsort* would have an $O(n^2)$ runtime with some simple inputs, AT&T Bell Lab researchers Jon L. Bentley and M. Douglas McIlroy set out to develop an improved version of *qsort*. They developed a version that was faster "compared to existing library sorts…typically about twice as fast - clearer, and more robust under nonrandom inputs" [3]. They published their findings in a paper in 1993, and the superior performance of their *qsort* version can be seen in the table below where they compared the sorting time of their algorithm with various data types to the Seventh Edition and Berkley *qsort* versions which were the two main versions at the time. They found that their new *qsort* version is "strictly faster than the Berkley function, which is in turn faster than the Seventh Edition function" and that the "running-time improvements vary with both machine and data type, from twenty percent to a factor of twelve".

Table II

| Type | VAX 8550 | | | MIPS R3000 | | |
|---|---|---|---|---|---|---|
| | 7th Edition | Berkeley | New | 7th Edition | Berkeley | New |
| integer | 1·25 | 0·80 | 0·60 | 0·75 | 0·41 | 0·11 |
| float | 1·39 | 0·89 | 0·70 | 0·73 | 0·43 | 0·14 |
| double | 1·78 | 1·23 | 0·91 | 1·33 | 0·78 | 0·19 |
| record | 3·24 | 2·01 | 0·92 | 3·10 | 1·75 | 0·26 |
| pointer | 2·48 | 2·10 | 1·73 | 1·09 | 0·73 | 0·41 |
| string | 3·89 | 2·82 | 1·67 | 3·41 | 1·83 | 0·37 |

The evolution of *qsort* only continued after that, and as of now other versions that have been developed like *qsort_r*, *qsort_s*, and *qsort_b*, each designed for a specific purpose [1]. For example, one issue with *qsort* was that passing in extra parameters, such as a second comparison function whose comparison of two values was achieved by comparing their difference with another value, could only be done via global variables which led to issues in particular programming circumstances, and the *qsort_r* provided a modified version of *qsort* to provide a solution to this.

But it is not just C that uses the quicksort algorithm. In fact, many of the other most popular and widely used programming languages have quicksort built into the language somewhere. C++ provides the C++ version of *qsort* in the C++ standard library also called *qsort* but it is in the std namespace so it is officially *std::qsort* [4]. Similarly, the Java programming language uses the quicksort sorting algorithm in some of the sorting methods built into the language. For example, the *Arrays.sort( )* method uses a dual-pivot quicksort on arrays of primitive data types [5]. Ultimately, it can be seen that quicksort is such a powerful sorting algorithm that its application is not limited to one specific thing but rather it's actually built into some of the most popular and commonly used programming languages. In any C, C++, or Java software application (or any other application in another programming language that also uses quicksort in its built-in sorting methods), there is a good chance that quicksort is being used if sorting is necessary. Undoubtedly, this means there are countless software programs in the world - past, present, and future - where quicksort is used.

## Application 2: Cache-Friendly Sorting Algorithm

The study of algorithms is actually a branch of mathematics that is independent of computers. As a result of this, the pure mathematical analysis of quicksort (or any other algorithm) does not take into account the other factors that affect the performance of algorithms when they actually get implemented in computers and software. One of these is cache performance. Quicksort is beneficial not only because it has a fast running time from a purely mathematical perspective, but it is considered a cache-friendly algorithm that is good for cache performance in a computer. Cache performance is often something that needs to be taken into consideration when writing software especially in lower level C and C++ programs. As a result of this, quicksort becomes even more appealing when cache-performance needs to be taken into consideration, and research has been done into how to modify it and other algorithms to maximize cache performance. A study done in 1999 by researchers Anthony LaMarca and Richard Ladner set out to "investigate the effect that caches have on the performance of sorting algorithms both experimentally and analytically" in order to "address the performance problems that high cache miss penalties introduce" [6]. The main motivation behind this study was that "since the introduction of caches, main memory has continued to grow slower relative to processor cycle times", and ultimately "cache miss penalties have grown to the point where good overall performance cannot be achieved without good cache performance". Ultimately, they explored "the potential performance gains that cache-conscious design offers in understanding and improving the performance of four popular sorting algorithms" including quicksort, and found that for three of the algorithms studied, including quicksort, "the improvement in cache performance leads to a reduction in total execution time". Variations of quicksort tuned for memory and cache optimization were studied, and they concluded that "the effects of caching are extremely important and need to be considered if good performance is a goal", and that for two of the variations of the sorting algorithms they studied including a variation of quicksort, "they both incur very few cache misses, which renders their overall performance far less sensitive to cache miss penalties than the others. As a result, these algorithms can be expected to outperform the others as relative cache miss penalties continue to increase". The graph on the following page demonstrates some of the results of their study including instructions per key, cache misses per key, and time (cycles per key) for the variations of the various sorting algorithms studied.

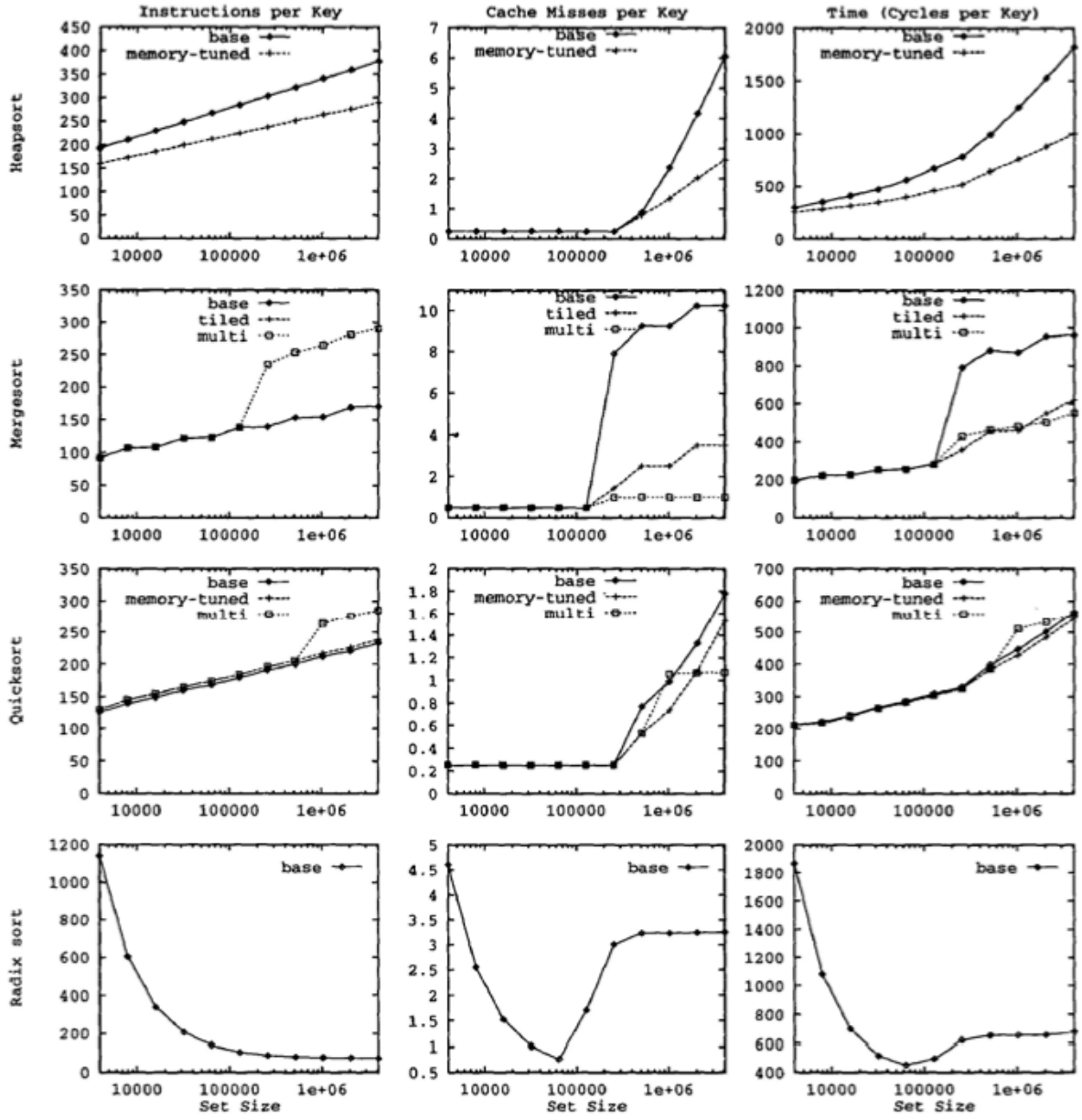



Figure 1: Performance of heapsort, mergesort, quicksort and radix sort on sets between 4,000 and 4,096,000 keys. The first column of graphs shows instruction counts per key, the second column shows cache misses per key and the third column shows execution times per key. Executions were run on a DEC Alphastation 250 and simulated cache capacity is 2 megabytes with a 32 byte block size.

## Conclusion

In conclusion, quicksort is one of the widely and most popularly used sorting algorithms in programming languages due to its superior performance and runtime. It has a time complexity of Θ(nlgn) and in practice sorts faster on average than other comparison-based sorting algorithms like heapsort or mergesort that also have time complexities of Θ(nlgn). It also has other advantages such as being cache-friendly and an in-place sorting algorithm which doesn't require building other arrays like in mergesort. Since it is not possible for any comparison-based sorting algorithm to be faster than Θ(nlgn), quicksort will continue to remain the fastest, general purpose sorting algorithm. Though there may be algorithms that are more advantageous in specific circumstances, in general quicksort is the go-to sorting algorithm when a program requires sorting in many cases. It is built into programming languages, will continue to be in the future, and will continue to have a broad application to software in helping to make applications as fast as possible.

# References

[1] “qsort”, Available: https://en.wikipedia.org/wiki/Qsort. [Accessed April 7, 2022]

[2] “qsort”, Available: https://en.cppreference.com/w/c/algorithm/qsort. [Accessed April 7, 2022].

[3] J. Bentley, M. McIlroy, “*Engineering a Sort Function”*, in *Software - Practice and Experience*, Vol. 23(11), pp. 1249-1265, 1993.

[4] “std::qsort”, Available: https://en.cppreference.com/w/cpp/algorithm/qsort. [Accessed April 7, 2022].

[5] baeldung, “Sorting in Java”, Available: https://www.baeldung.com/java-sorting. [Accessed April 7, 2022].

[6] A. LaMarca, R. Ladner, “The Influence of Caches on the Performance of Sorting”, in *Journal of Algorithms*, Vol. 31(1), pp.. 66-104, 1999.