* [Skip navigation](#content)
* [Skip to search form](#q)

# **November Column: Welcome to Algorithms in a Nutshell**

By [George T. Heineman](http://www.oreillynet.com/pub/au/3414)  
October 25, 2008 ([1ed](http://broadcast.oreilly.com/2008/10/welcome-to-algorithms-in-a-nut.html)) | Revised January 3 2016 (2ed)

**Overview**

This is the first of a series of monthly columns in the blog associated with the [**Algorithms in a Nutshell**](http://oreilly.com/catalog/9780596516246/)**, 1ed** book, published October 2008 by [**O'Reilly Media, Inc**](http://www.oreilly.com/). In preparation for the second edition, we have updated the text of these blogs to correspond to the new source code layout.

In this column we show how to download and install the **Algorithm Development Kit (ADK)**, the code repository associated with the book. You will be able to download the latest ADK version from the github [**repository associated with the book**](https://github.com/heineman/algorithms-nutshell-2ed). Clone this repository onto your local file system; we refer to this directory as $ADKHOME. Read the accompanying README.txt file for instructions on building the ADK; for the rest of this column we assume you have a properly built ADK on your computing platform.

**ADK Contents**

The code repository contains full implementations of algorithms from the following domains. The list is numbered by chapter number within the book.

1. **Sorting**: Insertion Sort, Quicksort, Selection Sort, Heap Sort, Bucket Sort. Merge Sort
2. **Searching**: Sequential Search, Binary Array Search, Hash-based Search, Binary Search Tree, Bloom Filter, AVL Tree
3. **Graph Algorithms**: Depth-first Search, Breadth-first Search, Dijkstra's Algorithm, Bellman-Ford Algorithm, Floyd-Warshall Algorithm, Prim's Algorithm
4. **Path Finding in AI**: Depth-first Search, Breadth-first Search, A\*Search, Minimax, NegMax, AlphaBeta
5. **Network Flow Algorithms**: Ford-Fulkerson, Edmonds-Karp
6. **Computational Geometry**: Convex Hull Scan, LineSweep, Fortune Sweep
7. **Spatial Structures: KD-Tree, Quadtree, R-Tree,** Nearest Neighbor Query, Range Queries
8. **Emerging Algorithms:** Knapsack, Multithreaded Quicksort, Probabilistic Estimation

In this column we will show how to use some of the **Sorting** algorithms implemented in C. Subsequent columns will investigate each of the remaining chapters listed above. Note that we provide implementation using a variety of programming languages, including Java, Python, C, and C++.

**Retrieving ADK 2.0 and build instructions**

The code samples are found on the github repository (<https://github.com/heineman/algorithms-nutshell-2ed.git>). Retrieve as follows:

% git clone <https://github.com/heineman/algorithms-nutshell-2ed.git>

The above will create a working directory "algorithms-nutshell-2ed". You need to have a working JUnit 4.0 compatible installation; find details on how to do this from <http://junit.org>. It is expected that there will be a sibling "junit4.0" directory to the above "algorithms-nutshell-2ed" directory, which will contain the required Jar files, in this case: **hamcrest-core-1.3.jar**, **junit-4.12.jar** orwhatever are the latest release versions available.

% cd $ADKHOME

% ls

algorithms-nutshell-2ed junit4.0

% bash no\_ant.sh

% cd Code

% make

% cd ..

% cd Blogs

% bash no\_ant.sh

If you have trouble completing this build, read the installation instructions for further details.

**Retrieving and building November code samples**

All Blog code samples are found on the github repository (<https://github.com/heineman/algorithms-nutshell-2ed.git>) in the Blogs/ project. You must have built the ADK before attempting any of these Blog examples.

To build the Blog code for this month, there is a makefile; here is how it would be built on Linux:

% cd $ADKHOME/Blogs/src/algs/blog/welcome

% make

Within this directory is an Excel spreadsheet NovemberData.xlsx that contains a sample of the benchmark runs found in this column.

**API Interface for sorting routines**

The following snippet presents the API of a routine that sorts in place the collection of elements pointed to by vals[0] .. vals[numElements-1].

/\*\*

\* Interface to sort the pointer-based information found in vals. Each element vals[i] points to an element in

\* the collection to be sorted.

\*  
 \* @param vals Array of pointer-based information in memory

\* @param numElements Number of elements in vals

\* @param cmp function to compare two elements

\*/

extern void sortPointers (void \*\*vals, int numElements, int (\*cmp) (void \*a1, void \*a2));

You may have read that Quicksort is the fastest algorithm to use for sorting elements. In **Chapter 4** of [**Algorithms in a Nutshell**](http://oreilly.com/catalog/9780596516246/) we compare numerous sorting algorithms under various scenarios to show the situations under which the individual algorithms perform the best. Let's see how Quicksort performs on a small collection, namely the line-by-line contents of the 12,041 line, 451-page bill H.R. 1424, more commonly known as [**The Emergency Economic Stabilization Act of 2008**](http://banking.senate.gov/public/_files/latestversionAYO08C32_xml.pdf).

As a baseline measure, let's time how long it takes for the *sort* utility to sort the file. This is not entirely a fair comparison since *sort* is a complex program that allows users to configure numerous ways to sort. In any event, here are the results:

% time sort BillText.txt > /dev/null

0.008u 0.000s 0:00.00 0.0% 0+0k 0+0io 0pf+0w

About 8/1000ths of a second.

Now let's look at the code written for this column. The **sample.c** program loads up the contents of the input text file (provided by the user via a command line argument) and constructs an array of pointers to the strings representing each line of input. Once constructed, the sortPointers routine is invoked, and all values are summarily printed to *stdout* in sorted order, according to the *strcmp* routine. Using this driver, we construct several executables which we now evaluate. The Makefile shows how each of these executables is built. All sorting implementations in the ADK use the same interface which makes it easy to experiment with each one. The following code presents the full implementation of HeapSort, for example:

#include "report.h"

/\*\* Heapify the subarray ar[0,max). \*/

static void heapify (void \*\*ar,

int(\*cmp)(const void \*,const void \*),

int idx, int max) {

int left = 2\*idx + 1;

int right = 2\*idx + 2;

int largest;

/\* Find largest element of A[idx], A[left], and A[right]. \*/

if (left < max && cmp (ar[left], ar[idx]) > 0) {

largest = left;

} else {

largest = idx;

}

if (right < max && cmp(ar[right], ar[largest]) > 0) {

largest = right;

}

/\* If largest is not already parent, swap and propagate. \*/

if (largest != idx) {

void \*tmp;

#ifdef COUNT

ADD\_SWAP;

#endif /\* COUNT \*/

tmp = ar[idx];

ar[idx] = ar[largest];

ar[largest] = tmp;

heapify(ar, cmp, largest, max);

}

}

/\*\* Build heap within array by repeatedly invoking heapify. \*/

static void buildHeap (void \*\*ar,

int(\*cmp)(const void \*,const void \*),

int n) {

int i;

for (i = n/2-1; i>=0; i--) {

heapify (ar, cmp, i, n);

}

}

/\*\* Sort the array using Heap Sort implementation. \*/

void sortPointers (void \*\*ar, int n,

int(\*cmp)(const void \*,const void \*)) {

int i;

buildHeap (ar, cmp, n);

for (i = n-1; i >= 1; i--) {

void \*tmp;

#ifdef COUNT

ADD\_SWAP;

#endif /\* COUNT \*/

tmp = ar[0];

ar[0] = ar[i];

ar[i] = tmp;

heapify (ar, cmp, 0, i);

}

}

The above code can be found in $ADKHOME/Code/Sorting/PointerBased/straight\_HeapSort.c.

Run the **sample** executable (which uses Quicksort) using the Linux *time* utility

% time ./sample BillText.txt > /dev/null

0.008u 0.000s 0:00.01 0.0% 0+0k 0+0io 0pf+0w

It takes about 8/1000th of a second. It appears that our Quicksort implementation is five times as fast. However, the *time* utility is not sufficient to capture fine-grained timing measurements. The **timing.c** driver shows how to compute more accurate timing measurements. From this driver we construct executables for the same algorithms.

% ./quicksort BillText.txt > /dev/null

0.003773 seconds

If you run this twenty times and discard the best and worst performing runs, you might find that the average behavior is something like 0.0037±.0001 seconds (**Note: Your mileage mary vary**). How does how Quicksort perform against how Bucket Sort on the same data?

% ./bucketsort BillText.txt > /dev/null

0.009579 seconds

Running twenty trials, eliminating the best and worst performers, results in an average time of 0.0097±0.0001 seconds. On this data set, a Bucket Sort (using 65,536 bins) performs three times as slowly.

What about Heap Sort? Twenty sample runs (of ./heapsort BillText.txt > /dev/null) yield the following timing measurements: 0.0050±.0001 seconds. This performance average is much closer to the Quicksort run, but still not as efficient. To sum up:

|  |  |
| --- | --- |
| **Algorithm** | **Time (in seconds) to sort 12,041 lines from H.R.1424 bill** |
| Quicksort | 0.00377 |
| Heap Sort | 0.00500 |
| Bucket Sort | 0.00958 |

**Reasons for difference**

To compute some "under the hood" measurements on these various sorting algorithms, you will need to recompile the ADK with some debugging information. Change to the $ADKHOME/Code/Sorting/PointerBased directory and edit the Makefile there to uncomment out the following line, to set the COUNT compiler directive.

COUNTCOMPARE = -DCOUNT

Within the $ADKHOME/Code/Sorting/PointerBased directory, type:

% make clean

% make

to properly recompile the sorting algorithms to compute additional information about (a) the number of times two elements were compared; and (b) the number of times two elements were swapped. Now return to $ADKHOME/Blogs/src/algs/blog/welcome and edit its Makefile in a similar fashion, to ensure that COUNTCOMPARE is set to -DCOUNT. Now type:

% make clean

% make

to complete the build of the executables. Now rerun each executable once, ignoring the actual execution time since now we focus on the reported information (which is written to *stderr*):

% ./quicksort BillText.txt > /dev/null

0.003991 seconds

278237 comparisons, 203246 swaps

% ./heapsort BillText.txt > /dev/null

0.005157 seconds

285591 comparisons, 150061 swaps

% ./bucketsort BillText.txt > /dev/null

0.010317 seconds

766933 comparisons, 756554 swaps

As you can see, the primary factor that impacts the performance is the number of comparisons between elements. Indeed, while Heap Sort swaps 37% fewer items than Quicksort, Heap Sort still is about 10% slower. The difference is inherent in the mechanics of the algorithm. Note, too, that Bucket Sort requires nearly three times as many comparisons as Quicksort (and about four times as many element swaps) on this input set. For this input data, the clear winner is Quicksort.

**Warning: At this point you should go back to the** $ADKHOME/Code/Sorting/PointerBased **directory and modify its** Makefile **to eliminate the counting measurements for swaps (and type** make clean; make**) otherwise the ADK will perform slower for future measurements. While you are at it, make the same modification to the** Makefile **in the** $ADKHOME/Blogs/src/algs/blog/welcome **directory and type** make clean; make**.**

In the book we compare the sorting algorithms on random string permutations of the 26 letters of the alphabet. To view these results on 12,041 random 26-letter strings, type the following command within the $ADKHOME/Blogs/src/algs/blog/welcomedirectory.

% make compare

This will use the ADK measurement tools to run 20 random trials over the data and generate timing results. The results are shown in the following table:

|  |  |
| --- | --- |
| **Algorithm** | **Time (in seconds) to sort 12,041 permutations of alphabet** |
| Bucket Sort | 0.0016 |
| Quicksort | 0.0028 |
| Heap Sort | 0.0049 |

Hmm. On this input data, Bucket Sort performs best. Why is this the case? One observation is that this Bucket Sort now uses 17,576 bins instead of 65,536, but that by itself is not the only story. The reason is that the randomized nature of this last trial ensures that the 12,041 randomized strings are evenly distributed among the 17,576 bins. If you inspect the data as found in the H.R.1424 bill, you will see a non-standard distribution: there are only 537 unique two-character prefixes of each line, which means that although there are 65,536 bins, only .8% of them are used. Even worse, 25% of the lines are placed into 12 bins with over 200 string lines in each bin. The bin with the most lines has 524 (this is the bin assigned for all lines that begin with the letters "th", a common prefix for words in the English language!).

**One more class of input, for good measure**

All good things come in threes. What if we used the sorting methods over 12,041 random English words drawn from a dictionary containing 160,136 words of two to thirteen letters? Type the following command within the $ADKHOME/Blogs/src/algs/blog/welcomedirectory.

% make wordCompare

|  |  |
| --- | --- |
| **Algorithm** | **Time (in seconds) to sort 12,041 random English words** |
| Bucket Sort | 0.0023 (a 44% increase) |
| Quicksort | 0.0029 (a 4% increase) |
| Heap Sort | 0.0050 (a 4% increase) |

Bucket Sort still manages to be the fastest performance, although it shows the highest percentage growth over the sorting times over the permutations of the alphabet. Will this trend continue? In a word, no. In the following experiment we sort 13-times as many random English words. Type the following command within the $ADKHOME/Blogs/src/algs/blog/welcomedirectory.

% make dictionaryCompare

Quicksort is clearly the most efficient and it also showed the smallest multiplicative growth.

|  |  |
| --- | --- |
| **Algorithm** | **Time (in seconds) to sort 156,533 random English words** |
| Quicksort | 0.0626 (39 times slower) |
| Heap Sort | 0.1333 (48 times slower) |
| Bucket Sort | 0.1638 (32 times slower) |

We provide the above comparison of sorting algorithms on three different classes of input data to stress one of the fundamental principles of working with algorithms: You must know your input data. Rarely will you find a single algorithm that works "in all cases", regardless of input. In chapter 4, we present analysis and benchmark results for a number of sorting algorithms to show their individual strengths and weaknesses. We encourage you to carry out your own empirical evaluations.

**Final Note: Optimization**

Given the existing sortPointers interface, there is bound to be some overhead simply in passing a function pointer to be used for comparing two elements. Can this performance penalty be eliminated? First, run the following code within the $ADKHOME/Blogs/src/algs/blog/welcomedirectory.

% ./modified\_quicksort BillText.txt > /dev/null

Review the **modified\_baseQsort.c** code to see how the modified code eliminates this function parameter to produce an executable that performs (over 20 trials as before) with an average of 0.0038±.0001 seconds, essentially with no improvement. It seems as if the optimizing compiler directive "-O3" is sufficiently powerful to ensure that there is no noticeable difference in implementations. Such a result should strengthen the argument for developing general purpose code libraries, rather than spending (wasting?) time in super-optimizing code for efficiency gains.

**Next Column**

In next Month's December column, we will further investigate algorithms from **Chapter 5: Searching**. Until next time, we hope you take the opportunity to investigate the numerous algorithms in the [**Algorithms in a Nutshell**](http://oreilly.com/catalog/9780596516246/) book as well as to explore the examples provided in the ADK.

[**Algorithms in a Nutshell**](http://oreilly.com/catalog/9780596516246/)  
*George T. Heineman, Gary Pollice, Stanley Selkow*