Wang XD, Wu YJ. An improved HEAPSORT algorithm with $n\log n - 0.788928n$ comparisons in the worst case. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 22(6): 898~903 Nov. 2007

An Improved HEAPSORT Algorithm with $n\log n - 0.788928n$ Comparisons in the Worst Case

Xiao-Dong Wang (王晓东) and Ying-Jie Wu (吴英杰)

College of Mathematics and Computer Science, Fuzhou University, Fuzhou 350002, China

E-mail: wangxd@fzu.edu.cn; yjwu@fzu.edu.cn

Received September 30, 2006; revised May 8, 2007.

Abstract A new variant of HEAPSORT is presented in this paper. The algorithm is not an internal sorting algorithm in the strong sense, since extra storage for n integers is necessary. The basic idea of the new algorithm is similar to the classical sorting algorithm HEAPSORT, but the algorithm rebuilds the heap in another way. The basic idea of the new algorithm is it uses only one comparison at each node. The new algorithm shift walks down a path in the heap until a leaf is reached. The request of placing the element in the root immediately to its destination is relaxed. The new algorithm requires about $n\log n - 0.788928n$ comparisons in the worst case and $n\log n - n$ comparisons on the average which is only about $0.4n$ more than necessary. It beats on average even the clever variants of QUICKSORT, if n is not very small. The difference between the worst case and the best case indicates that there is still room for improvement of the new algorithm by constructing heap more carefully.

Keywords data structures, analysis of algorithms, heaps, HEAPSORT

1 Introduction

Sorting is one of the most fundamental problems in computer science. In this paper only general and sequential sorting algorithms are studied. All results should be compared with the simple lower bound[1]

$$\begin{aligned}\log(n!) &= n\log n - n\log e + \theta(\log n)\\ &\approx n\log n - 1.442695n,\end{aligned}$$

for the worst and average case numbers of comparisons of general sorting algorithms. With respect to this lower bound, sorting by merging and sorting by insertion and binary search are very efficient. HEAPSORT[2,3] needs $2n\log n$ comparisons. HEAPSORT is in almost all cases less efficient than QUICKSORT. All versions of QUICKSORT are inefficient in the worst case but efficient in the average case.

Let $H(n) = 1+1/2+\cdots+1/n$ be the n-th harmonic number, $Q(n)$ the average number of comparisons of QUICKSORT, and $CQ(n)$ the average number of comparisons of the best-of-three variant of QUICKSORT called CLEVER-QUICKSORT. Then[4]

$$Q(n) = 2(n+1)H(n) - 4n \approx 1.386n\log n - 2.846n$$

and for $n \geqslant 6$

$$CQ(n) \approx 1.188n\log n - 2.255n.$$

Because of these results HEAPSORT has been considered for a long time only for theoretical reasons. Carlsson[5] presented a new variant of HEAPSORT whose average and worst-case complexity is $n\log n + \theta(n\log\log n)$. This algorithm does not beat CLEVER QUICKSORT on average for $n \leqslant 10^{16}$. Another variant of HEAPSORT is called BOTTOM-UP-HEAPSORT[4,6]. The worst-case number of comparisons of the algorithm is about $1.5n\log n - 0.4n$[4].

In this paper a new variant of HEAPSORT algorithm is presented. The new algorithm is not an internal sorting algorithm in the strong sense, since extra storage for n integers is necessary. The basic idea of the new algorithm is similar to the classical sorting algorithm HEAPSORT, but the algorithm rebuilds the heap in another way. The idea of the new algorithm is it uses only one comparison at each node. With one comparison we can decide which child of the node just considered contains the larger element and this child is promoted directly to the position of its parent. In this way, the algorithm shift walks down a path in the heap until a leaf is reached. In the new algorithm, the request of placing the element in the root immediately to its destination is relaxed. Therefore the algorithm saves the comparisons for rebuilding the heap.

In Section 2 we present the algorithm and discuss some details of its implementation. In Section 3 we prove that the worst-case number of comparisons

Short Paper

Supported by the Natural Science Foundation of Fujian under Grant No. A0510008.

of the new algorithm is remarkably small: it can be bounded by $n \log n - 0.788928n$. In Section 4 we estimate the average complexity of the new algorithm. The number of comparisons of the new algorithm is about $n \log n - n$ on average. The new algorithm is compared with other practical sorting algorithms. We finish the paper with conclusions in Section 5.

2 New Algorithm

Let $a[1..n]$ be an array of n elements of a key and some information associated with this key. This array is a (maximum) heap if, for all $i \in \{2, \ldots, n\}$, the key of element $a[i/2]$ is larger than or equal to that of element $a[i]$. That is, a heap is a pointer free representation of a binary tree, where the elements stored are partially ordered according to their keys. Element $a[1]$ with the largest key is stored at the root. Elements $a[i/2], a[2i]$ and $a[2i+1]$ (if they exist) are respectively stored at the parent, the left child and the right child of the node at which element $a[i]$ is stored. If a node has no children then the node is a leaf, otherwise the node is an internal node.

HEAPSORT is a classical sorting algorithm that is described in almost all algorithmic textbooks. Generally the HEAPSORT algorithm can be divided into two phases: the heap creation phase and the selection phase. HEAPSORT sorts the given elements in ascending order with respect to their keys as follows:

```
Input: Array a[1..n] of n elements.
Output: The elements in a[1..n] in sorted order.
void HEAPSORT()
{ buildheap();
  for (int i = n; i > 1; i − −) {
    swap(a[1], a[i]);
    reheap(i − 1);
  }
}
```

In the heap creation phase, the algorithm buildheap rearranges the input array $a[1..n]$ into a heap. In the selection phase, the algorithm reheap starts at the root and stops if the element at the node just considered is not smaller than the elements at its two children. Otherwise it interchanges the element at the node just considered with the larger element of the two children elements and considers the corresponding child. Algorithm reheap needs two comparisons at each node in order to compute the maximum of the three elements at the node and its two children.

The basic idea of the new algorithm is similar to HEAPSORT, but the algorithm reheap in another way. The new heapsort algorithm RANK_HEAPSORT can be described as follows:

```
void RANK_HEAPSORT()
{ buildheap();
  for (int i = n; i > 1; i − −) shift(i);
   rearrange();
  }
}
```

The core of the new algorithm is the algorithm shift. The algorithm reheap uses two comparisons at each node, while the algorithm shift uses only one comparison at each node. With one comparison we can decide which child of the node just considered contains the larger element and this child is promoted directly to the position of its parent. In this way, the algorithm shift walks down a path in the heap until a leaf is reached. This path will be called special path, and the last element on this path will be called special leaf. When the special leaf $a[k]$ is reached in the algorithm shift(i), the elements in $a[0]$ and $a[k]$ are swapped and the rank of the element in $a[k]$ is now i. To record the rank of the element in $a[k]$, an extra array $rank[1..n]$ is used. In the new algorithm, the request of placing the element in $a[0]$ immediately to its destination $a[i]$ is relaxed. The element in $a[0]$ is placed in $a[k]$ instead and its rank is stored in $rank[k]$. The values of $rank[i]$, $i \in \{1, \ldots, n\}$, are initialed with value 0. Once the value of $rank[i]$, $i \in \{1, \ldots, n\}$, is set to a value larger than 1, then the element in $a[i]$ is no longer an element of the current heap. In this way, the array $rank[1..n]$ is also used to indicate the elements of the current heap. The value of $rank[0]$ is set to $n + 1$ and hence the element in $a[0]$ is always not an element of the current heap.

In the classical algorithm HEAPSORT, elements $a[i/2], a[2i]$ and $a[2i+1]$ (if they exist) are respectively stored at the parent, the left child and the right child of the node at which element $a[i]$ is stored. In our new algorithm RANK_HEAPSORT, the relations are no longer hold. The unranked elements of the array still have the property that the key of an element is larger than or equal to that of its children. That is $a[rank[i/2]] \geqslant a[rank[i]]$ for all i. We call unranked elements of this array a pseudo heap.

The algorithm shift can be described as follows:

```
void shift(int index)
{ int i = 0;
  while(internal(i)){
    i = maxchild(i); a[i/2] = a[i];}
  a[i] = a[0]; rank[i] = index;
}
```

In the algorithm, the function internal(i) is used to test whether the node $a[i]$ is an internal node or not.

```
bool internal(int i)
{ int k = 2 * i;
  return (k ⩽ n) && !rank[k])||(k < n)
  && (!rank[k + 1]);
}
```

The function maxchild(i) returns the index of the child of the node $a[i]$ with the larger key. Obviously, only when the node $a[i]$ has both its left and right children the function maxchild(i) performs one comparison.

```
int maxchild(int i)
{ int k = 2 * i;
  bool left = (k ⩽ n) && (!rank[k]),
  right = (k < n) && (!rank[k + 1]);
  if (!left||left && right &&
    a[k] < a[k + 1])k + +;
  return k;
}
```

After $n-1$ calls of function shift, all elements get their right ranks. The algorithm rearrange rearranges the elements in an ascending order according to their ranks as follows.

```
void rearrange()
{ for (int i = 2; i ⩽ n; i + +)
  while(rank[i]! = i) {
    swap(a[i], a[rank[i]]);
    swap(rank[i], rank[rank[i]]); }
}
```

The correctness of the new algorithm is obvious, since after each call of function shift the elements remain unranked forms a pseudo heap.

3 Worst Case Analysis

Let the number of comparisons of the new algorithm for n elements be $T(n) = build(n) + sort(n)$, where $build(n)$ is the number of comparisons of the heap creation phase and $sort(n)$ is the number of comparisons of the selection phase.

Let $f(n)$ denote the sum of the depths of the pseudo heap considered during the selection phase.

Lemma 1. $f(n) = (n+1)\lfloor \log n \rfloor - 2^{\lfloor \log n \rfloor + 1} + 2$.

Proof. In a pseudo heap, the depth of the element $a[i]$ equals $\lfloor \log i \rfloor$. The sum of all $\lfloor \log i \rfloor$, $2 \leqslant i \leqslant n$, equals

$$\begin{aligned}&\sum_{1 \leqslant i \leqslant \lfloor \log n \rfloor - 1} i2^i + \lfloor \log n \rfloor (n - 2^{\lfloor \log n \rfloor} + 1)\\ &= (\lfloor \log n \rfloor - 2)2^{\lfloor \log n \rfloor} + 2\\ &\quad + n\lfloor \log n \rfloor - \lfloor \log n \rfloor 2^{\lfloor \log n \rfloor} + \lfloor \log n \rfloor\\ &= (n+1)\lfloor \log n \rfloor - 2^{\lfloor \log n \rfloor + 1} + 2.\end{aligned}$$

□

We estimate $sort(n)$ in the worst case first.

Theorem 1. *In the worst case, $sort(n)$ is bounded by*

$$f(n) - \lfloor n/2 \rfloor \leqslant (n+1)\log n - 2.413928n + 3.$$

Proof. The algorithm shift uses only one comparison at each node with two children. During the selection phase, each path form $a[1]$ to $a[i]$, $2 \leqslant i \leqslant n$, is searched as a special path once. If every node in the special path has two children, then the number of comparisons is equal to the depth of the special path. The sum of the depths of the pseudo heap considered during the selection phase is $f(n)$. From the beginning of the selection phase, there are exactly $\lceil (n-1)/2 \rceil$ nodes with two children. It is clear that these nodes will eventually become single child nodes in at least one special path. Consequently, there are at least $\lfloor n/2 \rfloor$ single child nodes on the special paths during the selection phase. Therefore $sort(n)$ is bounded above by $f(n) - \lfloor n/2 \rfloor$.

For all integers n, there is an x in [0, 1] with

$$\begin{aligned}f(n) - \lfloor n/2 \rfloor &= (n+1)\lfloor \log n \rfloor - 2^{\lfloor \log n \rfloor + 1} + 2 - \lfloor n/2 \rfloor\\ &= (n+1)(\log n - x) - 2^{\log n - x + 1}\\ &\quad + 2 - \lfloor n/2 \rfloor\\ &= (n+1)\log n - (x + 2^{1-x})n\\ &\quad + 2 - \lfloor n/2 \rfloor - x.\end{aligned}$$

It is easy to see that the function $g(x) = x + 2^{1-x}$ takes its maximum 2 for $x = 0, 1$, and $g(x)$ takes its minimum for $x = \log(2 \ln 2) = 0.471234$, where

$$g(\log(2 \ln 2) = \log(2 \ln 2) + (\ln 2)^{-1} \geqslant 1.913928$$

Therefore in the worst case,

$$\begin{aligned}&sort(n) \leqslant f(n) - \lfloor n/2 \rfloor\\ &\leqslant (n+1)\log n - 1.913928n - 0.5n + 2 - x + \delta\\ &\leqslant (n+1)\log n - 2.413928n + 3.\end{aligned}$$

□

Previously, the fastest algorithm for building heaps was due to Gonnet and Munro[7]. This algorithm takes about $1.625n + o(n)$ comparisons in the worst case. If buildheap() uses this algorithm, then in the worst case $T(n)$, the number of comparisons of the new algorithm for n elements is about $n \log n - 0.788928n$.

The number of element moves during the selection phase is obviously

$$f(n) \leqslant (n+1)\log n - 1.913928n + 2.$$

4 Experiments

Theorem 1 is of course also an upper bound, on the average case complexity of the new algorithm. Our worst-case analysis in Section 3 implies that the new algorithm can save comparisons only because the single child nodes on the special paths during the selection phase can be reduced. The number of the single child nodes on the special paths during the selection phase can be reduced substantially in the best case.

Theorem 2. *In the best case, sort(n), the number of comparisons of the new algorithm during the selection phase can be as large as $f(n)/2$ when $n = 2^k - 1$, and $k \geqslant 1$.*

Proof. The idea for constructing a best case instance of the algorithm is balancing. Let $best(i)$ be the number of comparisons required at node i in the best case. Then it can be proved by induction that for any integer $i \leqslant \lfloor (n-1)/2 \rfloor$ we have,

$$\begin{aligned} best(i) &\geqslant \min\{size(2i), size(2i+1)\} \\ &\quad + best(2i) + best(2i+1) \\ &= size(2i+1) + best(2i) + best(2i+1). \end{aligned}$$

where $size(i)$ is denoted as the size of the sub-tree rooted at node i.

From the above recursive inequality we can conclude that $best(1)$, the number of comparisons of the new algorithm during the selection phase can be bounded below by $\sum_{i=1}^{\lfloor (n-1)/2 \rfloor} size(2i+1)$.

For any integer $j > 0$, let $n0(j)$ be the number of zeros in the binary representation of j, and $n1(j)$ be the number of ones in the binary representation of j. If the binary representation of j is viewed as the path label from root of the heap to the node j, then a travel from root to the node j goes into right exactly $n1(j)-1$ times and left exactly $n0(j)$ times. Therefore the contribution of node j to the sum $\sum_{i=1}^{\lfloor (n-1)/2 \rfloor} size(2i+1)$ is $n1(j) - 1$. That is,

$$\sum_{i=1}^{\lfloor (n-1)/2 \rfloor} size(2i+1) = \sum_{i=1}^{n} (n1(i) - 1) = \sum_{i=1}^{n} n1(i) - n.$$

For integer set $S(n) = \{1, 2, \ldots, n\}$, the following algorithm constructs a special heap achieving the best case.

```
void best(int i, int x)
{ a[i] = x;
  if (size(i) > 1){
  if (size(2 * i + 1) > 0) best(2 * i + 1, x - 1);
  best(2 * i, x - size(2 * i + 1) - 1); }
}
```

The call best$(1, n)$ creates the heap with $\sum_{i=1}^{\lfloor (n-1)/2 \rfloor} size(2i+1)$ comparisons during the selection phase. Therefore we can conclude that in the best case, the number of comparisons of the new algorithm during the selection phase is

$$\sum_{i=1}^{\lfloor (n-1)/2 \rfloor} size(2i+1) = \sum_{i=1}^{n} n1(i) - n.$$

Since for any integer j, $n0(j) + n1(j) = \lfloor \log j \rfloor + 1$, we have,

$$\sum_{i=1}^{n} (n0(i) + n1(i)) = \sum_{i=1}^{n} (\lfloor \log i \rfloor + 1) = f(n) + n.$$

It can be easily seen that for any integer n, $\sum_{i=1}^{n} n0(i) \geqslant \sum_{i=1}^{n} n1(i) - n$. The equality holds when $n = 2^k - 1$, $k = 1, 2, \ldots$.

Therefore

$$f(n) + n = \sum_{i=1}^{n} n0(i) + \sum_{i=1}^{n} n1(i) \geqslant 2 \sum_{i=1}^{n} n1(i) - n.$$

That is,

$$\sum_{i=1}^{n} n1(i) - n \leqslant \frac{f(n)}{2} \approx \frac{1}{2}(n+1)\log n - 0.95696n.$$

The equality holds when $n = 2^k - 1$, $k = 1, 2, \ldots$. □

We cannot compute the average case complexity exactly, since the new algorithm does not construct random heaps during the selection phase. Table 1 presents data of our implementation of the new algorithm on some random sets of floating point data.

Table 1. Number of Comparisons (in the best case, worst case and on random data averaged over 30 runs)

n	100	1 000	10 000	100 000	1 000 000
Best Case	219	3 938	54 613	715 030	8 884 999
Worst Case	430	7 487	108 631	1 418 946	17 451 445
Average Case	405	7 317	106 963	1 402 241	17 336 960

The presented data reveals an average number of comparisons of the new algorithm during the selection phase is about $n \log n - 2.55n$.

The McDiarmid and Reed's variant of BOTTOM-UP-HEAPSORT algorithm[8] uses, on average, about $1.52n$ comparisons to build a heap. If buildheap() uses this algorithm, then the average number of comparisons of the new algorithm for n elements can be $n \log n - n$.

Table 2. Number of Comparisons of Different Algorithms on Random Data Averaged over 30 Runs

Comparisons	100	1 000	10 000	100 000	1 000 000
QUICKSORT	728	11 831	161 165	2 078 841	25 632 211
CLEVER-QUICKSORT	648	10 325	143 138	1 825 988	22 290 858
BOTTOM-UP-HEAPSORT	693	10 303	136 511	1 698 252	20 281 354
MDR-HEAPSORT	655	9 883	132 311	1 656 355	19 863 034
WEAK-HEAPSORT	615	9 511	128 565	1 618 686	19 487 753
RELAXED-WEAK-HEAPSORT	571	8 971	123 547	1 568 623	18 948 426
RANK-HEAPSORT	568	8 963	123 403	1 567 563	18 928 516

Given a uniform distribution of all permutations of the input array, QUICKSORT[9] reveals an average number of at most $1.386n \log n - 2.846n + O(\log n)$ comparisons. In CLEVER-QUICKSORT, the median-of-three variant of QUICKSORT[7], this value is approximately

$$1.188n \log n - 2.255n + O(\log n).$$

BOTTOM-UP-HEAPSORT[4,6] is a variant of HEAPSORT with $1.5n \log n + \theta(n)$ key comparisons in the worst case. The idea is to search the path to the leaf independently to the place for the root element to sink. Since the expected depth is high, this path is traversed bottom up. The average number of comparisons in BOTTOM-UP-HEAPSORT is about $n \log n + O(n)$. A further refinement MDR-HEAPSORT performs less than $n \log n + 1.1n$ comparisons in the worst case[6]. WEAK-HEAPSORT proposed by Dutton[10] uses less than $n \log n + 0.086013n$ comparisons. RELAXED-WEAK-HEAPSORT is a WEAK-HEAPSORT[11] variant which consumes at most $O(n \log n)$ extra bits and executes exactly $nk - 2^k + 1$ comparisons in the best, worst, and average cases if $k = \lceil \log n \rceil$.

Our new algorithm is compared with the 6 practical sorting algorithms above. In Sections 3 and 4, we consider the selection phase of the new algorithm. Let the number of comparisons of the new algorithm for n elements be $T(n) = \mathit{build}(n) + \mathit{sort}(n)$, where $\mathit{build}(n)$ is the number of comparisons of the heap creation phase and $\mathit{sort}(n)$ is the number of comparisons of the selection phase. Different heap construction algorithms will infect the total time required by the new algorithm. The classical algorithm of Floyd for building heaps from the bottom- up yields an upper bound of $2n$ comparisons. The McDiarmid and Reed's variant of BOTTOM-UP-HEAPSORT algorithm[8,12~15] uses, on average, about $1.52n$ comparisons to build a heap. If buildheap() uses this algorithm, then the average number of comparisons of the new algorithm for n elements can be $n \log n - n$.

Table 2 presents data of our implementation of various performance sorting algorithms on some random sets of floating point data. The $O(n \log n)$ bits for storing the index in RELAXED WEAK-HEAPSORT matches the amount of n integers in $[1, \ldots, n]$ needed for storing rank in RANK-HEAPSORT.

5 Conclusions

We have presented a new variant of HEAPSORT algorithm. The new algorithm requires about $n \log n - 0.788928n$ comparisons in the worst case and $n \log n - n$ comparisons on the average which is only about $0.4n$ more than necessary. Reinhardt[16] shows that MERGESORT can be designed in-place with $n \log n - 1.3n + O(\log n)$ comparisons in the worst case. However, for practical purposes the algorithm is too slow. The big difference between $\mathit{sort}(n)$ in the worst case and in the best case indicates that there is still room for improvement of the new algorithm by constructing heap more carefully. Concerning the layered memory and cache structure within a modern personal computer, a better prediction requires meticulous analysis, which, in turn, can lead to more efficient algorithms.

References

[1] Knuth D E. The Art of Computer Programming — Sorting and Searching. 2nd Edition, New York: Addison Wesley, 1998.

[2] Floyd R W. Algorithm 245: Treesort 3. *Communications of the ACM*, 1964, 7(4): 701.

[3] Williams J W J. Algorithm 232: HEAPSORT. *Communications of the ACM*, 1964, 7(4): 347~348.

[4] Wegener I. BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT, beating on an average, QUICKSORT (if n is not very small). *Theoretical Computer Science*, 1993, 118(1): 81~98.

[5] Carlsson S. A variant of HEAPSORT with almost optimal number of comparisons. *Information Processing Letters*, 1987, 24(3): 247~250.

[6] Wegener I. The worst case complexity of McDiarmid and Reed's variant of BOTTOM-UP HEAPSORT is less than $n \log n + 1.1n$. *Information and Computation*, 1992, 97(1): 86~96.

[7] Gonnet G H, Munro J I. Heaps on heaps. *SIAM Journal on Computing*, 1986, 15(6): 964~971.

[8] McDiarmid C J H, Reed B A. Building heaps fast. *Journal of Algorithms*, 1989, 10(3): 352~365.

[9] Hoare C A R. Quicksort. *Computer Journal*, 5(1): 10~15.

[10] Dutton R D. Weak heap sort. *BIT*, 1993, 33(3): 372~381.

[11] Edelkamp S, Stiegeler P. Implementing HEAPSORT with $n\log n-0.9n$ and QUICKSORT with $n\log n+0.2n$ comparisons. *ACM Journal of Experimental Algorithmics (JEA)*, 2002, 7(1): 1∼20.
[12] Cantone D, Cincotti G. QuickHeapsort, an efficient mix of classical sorting algorithms. *Theoretical Computer Science*, 2002, 285(1): 25∼42.
[13] Carlsson S, Chen J. The complexity of heaps. In *Proc. the Third Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM*, Philadelphia, PA, October 1992, pp.393∼402.
[14] Ding Y, Weiss M A. Best case lower bounds for Heapsort. *Computing*, 1992, 49(1): 1∼9.
[15] Z Li, Bruce A Reed. Heap building bounds. *LNCS*, 2005, 3608(1): 14∼23.
[16] Reinhardt K. Sorting in place with a worst case complexity of $n\log n-1.3n+O(\log n)$ comparisons and $\varepsilon n\log n+O(1)$ transports. *LNCS*, 1992, 650(6): 489∼499.

Xiao-Dong Wang is a professor and chair of the College of Mathematics and Computer Science, Fuzhou University, P.R. China. His research interests are in design, analysis, and computational evaluation of algorithms and data structures for solving large-scale problems motivated from information assurance and security, the Internet, information visualization, and geometric computing, parallel and distributed algorithms, and complexity theory. His books within the last three years are "The Design and Analysis of Algorithms" (2005), "The Design and Experiments of Algorithms" (2006) and "Data Structures in C" (2007). Professor Wang is a senior member of China Computer Federation and chair of Fujian Computer Federation of China.

Ying-Jie Wu received his B.S. and M.S. degrees in computer science from Fuzhou University in 2001 and 2004, respectively. Now he is a lecturer at the College of Mathematics and Computer Science, Fuzhou University. He is currently a Ph.D. candidate in computer science at Southeast University. His research interests are in algorithms design and data mining.

Appendix A

A.1 C Code for McDiarmid/Reed's Heap Building Algorithm.

```
void build()
{
  for(int i = n/2; i ⩾ 1; i − −)
     mcdiarmid_reed(i);
}

void mcdiarmid_reed(int i)
{
  bubble_up(i, trickle_dowm(i));
}
```

//trickle down an empty position to the bottom of the heap.

```
int trickle_dowm(int i)
{
  a[0] = a[i];
  i∗ = 2;
  while(i ⩽ n){
    if(i < n && a[i] < a[i + 1])i + +;
    a[i/2] = a[i];
    i∗ = 2;
    count++;
  }
  a[i/2] = a[0];
  return i/2;
}
```

//bubble up a[c] to its correct position.

```
void HEAP::bubble_up(int i, int c)
{
  a[0] = a[c];
  while(c > i && a[0] > a[c/2]){
    a[c] = a[c/2]; c/ = 2; count++;
  }
  a[c] = a[0];
}
```

A2. Min-Max Intervals, and Statistical Variance of Experimental Data.

	N	Mean	Std Dev	Minimum	Maximum
Heap 1	102	52.509 803 9	29.764 336 8	1.000 000 0	101.000 000 0
Heap 2	1 002	497.967 065 9	286.983 435 8	2.000 000 0	1 001.00
Heap 3	10 002	5 013.61	2 877.60	2.000 000 0	10 001.00
Heap 4	100 002	50 070.39	28 847.73	1.000 000 0	100 001.00
Heap 5	1 000 002	424 925.49	92 216.30	1.000 000 0	1 000 001.00