Theoretical Computer Science 134 (1994) 559–565
Elsevier

Note

Asymptotic optimal HEAPSORT algorithm

Gu Xunrang and Zhu Yuzhang

Department of Computer Science, Shanghai University of Science and Technology, Shanghai 201800, People's Republic of China

Communicated by M. Nivat
Received June 1992

Abstract

Gu, X. and Y. Zhu, Asymptotic optimal HEAPSORT algorithm, Theoretical Computer Science 134 (1994) 559–565.

Heapsort algorithm HEAPSORT runs in a higher efficiency way. It has been improved to reduce the constant factor of the complexity. An asymptotic optimal heapsort algorithm is given in this paper. When the efficiency becomes the lowest, the constant factor of its complexity will not be more than $\frac{4}{3}$.

1. Introduction

Heapsort algorithm HEAPSORT was created by Williams in the 1960s, and it was improved by Floyd. It requires $O(n)$ comparisons to set up the original heap, and requires $O(n \log n)$ [3] comparisons to rearrange the heap. (All logarithms in this paper are to the base 2.) Its worst case time complexity is $T(n) = 2n \log n + O(n)$.

It has been theoretically proved that any algorithm for sorting n elements requires at least $\log(n!)$ comparisons. And we know,

$$\log n! \approx n \log n - 1.44\, n + \log \sqrt{n} + 1.325.$$

Correspondence to: Gu Xunrang, Department of Computer Science, Shanghai University of Science and Technology, Shanghai 201800, People's Republic of China.

SSDI 0304-3975(94)00113-W

Some people modified this algorithm on rearranging the heap and the complexity reduces to $\frac{4}{3}n\log n+O(n)$ [4], and $\frac{7}{6}n\log n+O(n)$ [5]. We notice that when rearranging the heap, the root is removed and it becomes vacant. By comparing its leftson with the rightson once, the larger one can move up one level. Repeat this action until the vacant node appears at $\frac{2}{3}$ of the height of the current heap. Therefore, the algorithm will have the asymptotic optimal performance if the procedure runs recursively on the progressively increasing fraction value of the height of the current subheap.

2. Definition

If the binary tree T is a heap [4], the subtree whose root can be any node in T is called the subheap of T.

It is clear that the subheap whose root is the root of T is the heap itself.

3. Algorithm

3.1. The way to design the algorithm

$$S(num)=\frac{2^{num}}{2^{num}-1} \quad (num\geqslant 1)$$

num can take the value $\ldots, 6, 5, 4, 3, 2, 1$; accordingly. $S(num)$ has the value $\ldots, \frac{32}{63}, \frac{16}{31}, \frac{8}{15}, \frac{4}{7}, \frac{2}{3}, 1$.

The value of *num* is ought to be taken large enough. For sorting problem on n elements, $num=\lfloor \log(\log n) \rfloor$. Once *num* is set, it will remain unchanging in the whole process of heapsort.

The process of rearranging the heap is implemented recursively. When rearranging the current subheap with height h, by one comparison the leftson with the rightson, the larger can move up one level. Repeat this process until it reaches at $S(num)h$ of the current subheap. Suppose at this point the current vacant node is i. The next step is to compare the element at leaf node, which is outside the current heap and to be sorted, with the element at the father's node of the vacant node i. If the latter is smaller, the former is shifted to node i, and by one comparison with its father's several times it will move up to the proper position of the current subheap. If the latter is greater, the recursive procedure will be carried out. When it reaches at the maximum depth, the element at leaf node which is outside the current heap and to be sorted will be shifted to the current vacant node, and it will move down along certain path.

3.2. Algorithm

(1) Set up the original heap [1.4]

```
procedure HEAPFIFY (i, j);
  {Arrange elements A[i]−A[j] of array A into a heap }
  if i is not a leaf
    and a son of i contains an element which is larger than i
    then begin
           Let k be a son of i with the larger element;
           interchange A[i] and A[k];
           HEAPFIFY (k, j)
           end;

procedure BUILDHEAP;
  for i:=⌊n/2⌋ step −1 until 1 do HEAPFIFY (i, n);
```

(2) Rearranging the heap

```
procedure RE-UPORDOWN (i, num, j);
  begin
    h2:=⌊(2^(num−1)/(2^num−1)) h1⌋;
    count := 1;
    while count ⩽h2 do
      {The process of comparing the leftson with the rightson once
      and the larger moving up one level stops at height 2^(num−1)/(2^num−1)
      of the current subheap.}
        begin
          Let r be a son of i with the larger element;
          A[i]:=A[r];
          i:=r;
          count:=count+1
        end;
      if A[j+1]⩾A[⌊i/2⌋]
        then {A[j+1] moves up to the proper position of the current subheap}
          begin
            A[i]:=A[j+1];
            while A[i]>A[⌊i/2⌋] do
              begin
                interchange A[i] and A[⌊i/2⌋];
                i:=⌊i/2⌋
              end
          end
      else if num>1
        then begin
```

```
                h1:=h1−h2; {Get the height of the subheap.}
                RE-UPORDOWN(i, num−1, j)
                  {Rearranging the current subheap recursively.}
              end
          else begin
                A[i]:=A[j+1];
                HEAPFIFY(i, j)
              end
  end;
```

(3) Heap sorting

Input: array of elements $A[i]$ $(1 \leqslant i \leqslant n)$ to be sorted. We define the dummy element $A[0]$, which is large enough, in case it is out of bound during the comparisons.

Output: the sorted array A.

```
procedure ASYMPTOTIC-OPTIMAL-HEAPSORT;
  begin
    BUILDHEAP;
    Find the value of num; {say num=⌊log(log n)⌋}
    for j:=n step −1 until 2 do
      begin
        temp:=A[1];
        h1:=⌊log(j−1)⌋; {Get the height of the current heap}
        i:=1; {Starting from the first node}
        RE-UPORDOWN (i, num, j−1);
        A[j]:=temp
      end
  end;
```

4. Analysis of the complexity

We consider the elements ranging from node i to node j. Suppose h is the height of the subheap whose root is i. Therefore we have the following lemma.

Lemma. *The number of comparisons that the recursive procedure takes is*

$$T(num, h) = 2S(num)h.$$

Proof. The process of rearranging the heap is correct. This can be proved by induction on the recursion depth.

When rearranging the current subheap, first by one comparison the leftson with the rightson of the vacant node, the larger can move up one level. Repeat this process until

it reaches $S(num)h$ of the current subheap and totally it needs $S(num)h$ comparisons. The next step is to compare the element $A[j+1]$ at leaf node, which is outside the current heap and to be sorted, with the element at the father's node of the vacant node i. At the worst case, either $A[j+1]$ moves up to the root of the current subheap and it takes $S(num)h$ comparisons, or $A[j+1]$ is searched for the position in the subheap whose root is i and the height is $(1-S(num))h$. So we have the following recursion equation.

$$\begin{cases} T(num, h)=\begin{cases}2S(num)h \\ S(num)h+T(num-1, (1-S(num))h)\end{cases} \\ T(1, h)=2h \end{cases}$$

$S(num)=2^{num-1}/(2^{num}-1)$, if the recursion depth is k, so $0\leqslant k\leqslant num-1$.

(1) When $k=0$, we directly have $T(num, h)=2S(num)h$.

(2) Now we consider $0<k<num-1$.

If $A[j+1]$ moves up in the current subheap, at the worst case,

$$\begin{aligned} T(num, h)&=S(num)h+S(num-1)(1-S(num))h+\cdots \\ &\quad +S(num-k+1)(1-S(num))(1-S(num-1))\cdots(1-S(num-k+2))h \\ &\quad +2S(num-k)(1-S(num))(1-S(num-1))\cdots(1-S(num-k+1))h \\ &=\frac{2^{num-1}}{2^{num}-1}h+\frac{2^{num-2}}{2^{num-1}-1}\cdot\frac{2^{num-1}-1}{2^{num}-1}h+\cdots \\ &\quad +\frac{2^{num-k}}{2^{num-k+1}-1}\cdot\frac{2^{num-1}-1}{2^{num}-1}\cdot\frac{2^{num-2}-1}{2^{num-1}-1}\cdot\cdots\cdot\frac{2^{num-k+1}-1}{2^{num-k+2}-1}h \\ &\quad +2\cdot\frac{2^{num-k-1}}{2^{num-k}-1}\cdot\frac{2^{num-1}-1}{2^{num}-1}\cdot\frac{2^{num-2}-1}{2^{num-1}-1}\cdot\cdots\cdot\frac{2^{num-k}-1}{2^{num-k+1}-1}h \\ &=\frac{2^{num-1}}{2^{num}-1}h+\frac{2^{num-2}}{2^{num}-1}h+\cdots+\frac{2^{num-1}}{2^{num}-1}h+2\cdot\frac{2^{num-k-1}}{2^{num}-1}h \\ &=\frac{h}{2^{num}-1}[2^{num-1}+2^{num-1}+\cdots+2^{num-k}+2^1\cdot 2^{num-k-1}] \\ &=\frac{h}{2^{num}-1}2^{num} \\ &=2S(num)h. \end{aligned}$$

If $A[j+1]$ moves down in the current subheap, the recursion depth will be increased by 1.

(3) When $k=num-1$, the recursion depth becomes the top value, at the worst case,

$$\begin{aligned}
T(num, h) &= S(num)h + S(num-1)(1-S(num))h \\
&\quad + S(num-2)((1-S(num))(1-S(num-1))h \\
&\quad + \cdots + S(2)(1-S(num))(1-S(num-1))\cdots \cdot \tfrac{7}{15} \cdot \tfrac{3}{7}k \\
&\quad + 2S(1)(1-S(num))(1-S(num-1))\cdots \cdot \tfrac{7}{15} \cdot \tfrac{3}{7} \cdot \tfrac{1}{3}h \\
&= \frac{2^{num-1}}{2^{num}-1}h + \frac{2^{num-2}}{2^{num-1}-1} \cdot \frac{2^{num-1}-1}{2^{num}-1}h \\
&\quad + \frac{2^{num-3}}{2^{num-2}-1} \cdot \frac{2^{num-1}-1}{2^{num}-1} \cdot \frac{2^{num-2}-1}{2^{num-1}-1}h \\
&\quad + \cdots + \frac{2}{3} \cdot \frac{2^{num-1}-1}{2^{num}-1} \cdot \frac{2^{num-2}-1}{2^{num-1}-1} \cdot \cdots \cdot \frac{7}{15} \cdot \frac{3}{7}h \\
&\quad + 2 \cdot \frac{2^{num-1}-1}{2^{num}-1} \cdot \frac{2^{num-2}-1}{2^{num-1}-1} \cdot \cdots \cdot \frac{7}{15} \cdot \frac{3}{7} \cdot \frac{1}{3}h \\
&= \frac{h}{2^{num}-1}[2^{num-1}+2^{num-2}++2^{num-3}+\cdots+2^1+2^1] \\
&= \frac{h}{2^{num}-1}2^{num} \\
&= 2S(num)h. \qquad \square
\end{aligned}$$

Theorem. *The worst case time complexity of the algorithm ASYMPTOTIC-OPTIMAL -HEAPSORT is*

$$T(n) = \frac{2^{num}}{2^{num}-1} n \log n + O(n).$$

Proof. The correctness of the heapsort algorithm can be proved by induction on the number of times that the "FOR" loop has been executed.

The complexity of the algorithm consists of two parts:

(1) Setting up the original heap by calling BUILDHEAP.

It takes time $O(n)$ [2, 4]

(2) The time it requires for the FOR loop to perform. Since the height of the heap with $j-1$ elements is $h=\lfloor \log(j-1) \rfloor$, so according to the above lemma, we have

$$\begin{aligned}
T'(n) &= \sum_{2 \leqslant j \leqslant n} T(num, \lfloor \log(j-1) \rfloor) = \sum_{2 \leqslant j \leqslant n} [2S(num) \cdot \lfloor \log(j-1) \rfloor] \\
&= 2S(num) \sum_{2 \leqslant j \leqslant n} \lfloor \log(j-1) \rfloor \\
&= 2S(num) n \log n = \frac{2^{num}}{2^{num}-1} n \log n
\end{aligned}$$

The total complexity of ASYMPTOTIC-OPTIMAL-HEAPSORT is

$$T(n)=\frac{2^{num}}{2^{num}-1}n\log n+O(n). \qquad \square$$

5. Discussion

(1) In fact, when rearranging the current heap. once the recursive procedure is executed, $A[j+1]$ and $A[\lfloor i/2 \rfloor]$ compares once, totally it requires *num* comparisons at most. Hence, heap sorting will have another $n \cdot num$ (n times *num*) comparisons. Normally, $num=\lfloor \log(\log n) \rfloor$, so that,

$$T(n)=\frac{2^{num}}{2^{num}-1}n\log n+O(n\log(\log n))+O(n).$$

(2) When $n\to\infty$, $\log(\log n)\to\infty$, i.e., $num\to\infty$. We notice that

$$\lim_{num\to\infty}\frac{2^{num}}{2^{num}-1}=1,$$

hence $T(n)=n\log n+O(n\log(\log n))+O(n)$. So we can conclude that the algorithm is of asymptotic optimal performance.

(3) The condition to call the recursive procedure RE-UPORDOWN to rearrange the heap is $num>1$, i.e. $\lfloor \log(\log n) \rfloor>1$, i.e., $n\geqslant 16$. It is advisable to use the revised heapsort algorithm [4] if the number of elements to be sorted is less than sixteen.

(4)When $n\geqslant 16$, $\lfloor \log(\log n) \rfloor\geqslant 2$, i.e., $num\geqslant 2$, but

$$\operatorname*{Min}_{num>2}\{S(num)\}=\tfrac{2}{3},$$

so from the revised heapsort algorithm, we know that the constant factor of the complexity is not more than $\frac{4}{3}$, and it depends on the *num* value in $(1, \frac{4}{3}]$.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1975).

[2] S. Baase, *Computer Algorithms: Introduction to Design and Analysis* (Addison-Wesley, Reading, MA, 1978).

[3] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms* (Computer Science Press, 1978).

[4] X. Gu and Y. Zhu, A new HEAPSORT algorithm and the analysis of its complexity, *Computer J.* **33** (3) (1990) 281–282.

[5] I. Wegener, A simple modification of Xunrang and Yuzhang's heapsort variant improving its complexity significantly, FB Informatik, University of Dortmund, Germany, 1992.