Sorting

Table of Contents

Introduction
Bubblesort
Selection Sort
Duplex Selection Sort
Duplex Selection Sort (cont)
Comparison Order Analysis
Comparison Order Analysis (cont)
Time Analysis
Time Analysis (cont)
Minimal Comparison
Comparison Tree Argument
Minimal Order: Comparison Sorting
Quicksort: partitioning
Quicksort: find pivot
Quicksort
Iterative Quicksort
Iterative Quicksort Code
Quicksort Efficiency
Sorting Considerations

Introduction

Definitions and Terminology:
- Internal sorts use Arrays
- External sorts use Files
- Ascending Order:
  † Low to High
- Descending Order:
  † High to Low
- Stable Sort:
  † Maintains the relative order of equal elements, in situ.
  † Desirable if list is almost sorted or if items with equal values are to also be ordered on a secondary field.

Program efficiency
- Overall program efficiency may depend entirely upon sorting algorithm => clarity must be sacrificed for speed.

Sorting Algorithm Analysis
- Performed upon the “overriding” operation in the algorithm:
  † Comparisons
  † Swaps

```c
void swap( int& x, int& y ) {
    int tmp = x;
    x = y;
    y = tmp;
}
```
**Bubble Sort**

**Behavior**
- *Bubble* elements down (up) to their location in the sorted order.
- Bubble is $O(\cdot)$

**Graphical Trace**

```
for (i = 0; i < n-1; i++)
for (j = n-1; j > i; j--)
if (A[j] < A[j-1])
    swap(A[j], A[j-1]);
```

**Selection Sort**

**Behavior**
- In the $i^{th}$ pass, select the element with the lowest value among $A[i]$, ..., $A[n-1]$, & swap it with $A[i]$.
- SelectionSort is $O(\cdot)$

**Graphical Trace**

```
for (Begin = 0; Begin < n-1; Begin++) {
    idxOfMin = Begin;
    MinSoFar = A[Begin];
    for (Look = Begin + 1; Look < n; j++) {
        if (A[Look] < MinSoFar) {
            MinSoFar = A[Look];
            idxOfMin = Look;
        }
        swap(A[Begin], A[idxOfMin]);
    }
}
```
Missing content
Comparison Order Analysis

Comparison $O(DuplexSelectSort)$

- Outer Loop: WHILE $i$ loop
  
  loop limits shifted limits
  $= 0 \ldots N/2-1 = 1 \ldots N/2$
  
  Assume subset of array to sort is from $1 \ldots N$
  $= 1 \ldots N/2$ (i.e. start ... finish )

- Inner Loop: FOR $j$ loop

  Pass ($i$) loop limits shifted limits

  1st $1 \ldots N$ $= 1 \ldots N$
  2nd $2 \ldots N - 1$ $= 1 \ldots N - 2$
  3rd $3 \ldots N - 2$ $= 1 \ldots N - 4$

  \[ \cdots \]

  ith iteration $i \ldots N - (i-1)$ $= 1 \ldots N - (i-1) \times 2$

  \[ \cdots \]

  $N/2-1$ $= N/2-1 \ldots N - (N/2-1-1)$
  $= N/2-1 \ldots N/2+2 = 1 \ldots 4$
  $= 1 \ldots N - (i-1) \times 2$

  $N/2$ $= N/2 \ldots N - (N/2-1)$
  $= N/2 \ldots N/2 + 1 = 1 \ldots 2$

Comparison Order Analysis (cont)

Worst Case:

- 2 Comparisons on each element

\[
\sum_{i=1}^{N/2} \sum_{j=i}^{N-(i-1)} 2 = \sum_{i=1}^{N/2} \left( \sum_{j=1}^{N-2(i-1)} 2 \right)
\]

expanding yields

\[
= 2 \sum_{i=1}^{N/2} \left[ N - 2(i - 1) \right]
\]

summing the arithmetic sequence yields

\[
= 2 \left[ \frac{1}{2} \left( \frac{N}{2} \right) \left[ N + 2 \right] \right]
\]

\[
= \frac{N^2}{2} + N = O(N^2)
\]
Time Analysis

Assume:
- Start = 1    Finish = N

Analysis
- Initialization of High and Low = 2
- While Loop
  i = loop index
  5 = While condition + Assignments

\[ \sum_{i=1}^{\frac{N}{2}} [5 + \text{inner}_\text{for} + \text{swap} + \text{incr}_\text{dec}] \]

For Loop
- j = minmax index

\[ \sum_{j=1}^{N-(i-1)} 5 = (\text{For } + 2 \text{ IF conditions} + 2 \text{ assignments}) \]

\[ T(N) = \sum_{j=1}^{N-(i-1)} 5 \]

\[ T(N) = 5 \sum_{j=1}^{N-(i-1)} 1 \]

- Swap Interference
  = 8 = 2 IF + 6 Assignment

Incr & Dec = 2

Time Analysis (cont)

\[ T(N) = 2 + \sum_{i=1}^{N/2} \left[ 5 + 5 \sum_{j=1}^{N-2(i-1)} 1 + 8 + 2 \right] \]

\[ = 2 + \sum_{i=1}^{N/2} [15 + 5(N - 2[i - 1])] \]

\[ = 2 + \sum_{i=1}^{N/2} 25 + \sum_{i=1}^{N/2} 5N - \sum_{i=1}^{N/2} 10i \]

\[ = 2 + \frac{25}{2} N + \frac{5}{2} N^2 - \frac{10}{2} \left[ \frac{N}{2} \left( \frac{N}{2} + 1 \right) \right] \]

\[ = \frac{5}{4} N^2 + 10N + 2 \]

\[ \therefore T(N) \in O(N^2) \]
Minimal Comparisons

Comparison-Based Sorting
- Algorithms which compare element values to each other
- What is the minimum number of comparisons, required to sort N elements using a comparison-based sort?

Comparison Tree
- **Binary** tree (hierarchical graph ≤ 2 branches per node) which contains comparisons between 2 elements at each non-leaf node & containing element orderings at its leaf (terminal) nodes.
- Comparison Tree for 3 Elements
- Any of the 3 elements (a, b, c) could be first in the final order. Thus there are 3 distinct ways the final sorted order could start.
- After choosing the first element, there are two possible selections for the next sorted element.
- After choosing the first two elements there is only 1 remaining selection for the last element.
- Therefore selecting the first element one of 3 ways, the second element one of 2 ways and the last element 1 way, there are 6 possible final sorted orderings = $3 \times 2 \times 1 = 3!$

Comparison Tree Argument

Order Tree for sorting N Elements
- Any of the N elements (1 ... N) could be first in the final order. Thus there are N distinct ways the final sorted order could start.
- After choosing the first element, there are N-1 possible selections for the next sorted element.
- After choosing the first two elements there N-2 possible selections for the next sorted element, etc.
- Therefore selecting the first element one of N ways, the second element one of N-1 ways, etc., there are $N \times (N-1) \times (N-2) \times \ldots \times 2 \times 1$ possible final sorted orderings which = $N!$

General Comparison Tree Sorting
- The comparison tree for N elements must have $N!$ leaf nodes. Each leaf node contains one of the possible orderings of all of the N elements.
- Consider the previous comparison tree for 3 elements, all of the leaf nodes are at a depth of either 2 or $3 > \lceil \log_2 3! \rceil$
- The comparison tree for 4 elements must contain $4! = 24$ leaf nodes, all of which would be at a depth of either 4 or $5 > \lceil \log_2 4! \rceil$

**floor** $y = \lfloor x \rfloor$, $y$ is the largest integer such that $y \leq x$. 
### Minimal Order: Comparison Sorting

**General Comparison Trees**
- The comparison tree for N elements must contain N! leaf nodes, all of which would be at a depth > ⌊log₂ N!⌋.
- The minimal number of comparisons required to sort a specific (unsorted) ordering is equal to the depth from the root to a leaf.

- Since the depth of all leaf nodes is > ⌊log₂ N!⌋ in a comparison tree, the minimal number of comparisons to sort a specific initial ordering of N elements is > ⌊log₂ N!⌋.
- Stirling’s Approximation for log₂(N!) can be used to determine a lower bound for log₂(N!) which is NlogN.
- No comparison based sorting algorithm can sort faster than O(N log N).

### Quicksort: partitioning

**Algorithm**
- Select an element in the array, (key value), as the pivot key.
- Divide the array into two partitions: a left partition containing elements < the pivot key and a right partition containing elements ≥ the pivot key.

**Trace**

Start with i and j pointing to the first & last elements, respectively.
Select the pivot (3):  

\[
\begin{array}{cccccccccc}
3 & 1 & 4 & 1 & 5 & 9 & 2 & 6 & 5 & 3 \\
\hline
i & j
\end{array}
\]

Swap the end elements, then move l, r inwards.

\[
\begin{array}{cccccccccc}
3 & 1 & 4 & 1 & 5 & 9 & 2 & 6 & 5 & 3 \\
\hline
L & j & R
\end{array}
\]

Swap, and repeat:

\[
\begin{array}{cccccccccc}
2 & 1 & 4 & 1 & 5 & 9 & 3 & 6 & 5 & 3 \\
\hline
L & R
\end{array}
\]

Swap, and repeat:

\[
\begin{array}{cccccccccc}
2 & 1 & 1 & 4 & 5 & 9 & 3 & 6 & 5 & 3 \\
\hline
R & L
\end{array}
\]

**Partition Function:**

```c
int Partition(raytype A[], int i, int j, keytype pivot) {
    int left, right;
    left = i; right = j;
    do {
        swapElems( A[left] , A[right] );
        while (A[left].key < pivot)  l++;
        while (A[right].key >= pivot)  r--;
    } while (right > left);
    return (left);
}
```
Quicksort: find pivot

Pivoting
- Partitioning test requires at least 1 key with a value < that of the pivot, and 1 key value ≥ to that of the pivot.
- Therefore, pick the greater of the first two distinct values (if any).

```c
int FindPivot(const raytype A[], int i, int j) {
    keytype firstkey;  //value of first key found
    int pivot;  //pivot index
    int k;  //run right looking for another key
    
    firstkey = A[i].key;
    //return -1 if different keys are not found
    pivot = MISSING;
    k = i + 1;
    //scan for different key
    while ( (k <= j) && (pivot == -1) )
        if ( A[k].key > firstkey )  //select key
            pivot = k
        else if ( A[k].key < firstkey)
            pivot = i
        else
            k++;
    return pivot;
}
```

Improving FindPivot
- Try and pick a pivot such that the list is split into equal size sublists, (a speedup that should cut the number of partition steps to about 2/3 that of picking the first element for the pivot).
- Choose the middle (median) of the first 3 elements.
- Pick k elements at random from the list, sort them & use the median.
- There is a trade-off between reduced number of partitions & time to pick the pivot as k grows.

Quicksort Function (recursive)

```c
const int  MISSING = -1;

void QuickSort( raytype A[], int i, int j ) {
    keytype pivotKey;
    int pivotIndex;
    int k;  //index of partition >= pivot
    
    pivotIndex = FindPivot( A, i, j );
    if (pivotIndex != MISSING) {
        pivotKey = A[ pivotIndex ].key;
        k = Partition( A, i, j, pivotKey );
        QuickSort( A, i, k-1 );
        QuickSort( A, k, j );
    }
}
```

Average Case
- quicksort is based upon the intuition that swaps, (moves), should be performed over large distances to be most effective
- quicksort's average running time is faster than any currently known O(nlog₂n) internal sorting algorithms (by a constant factor).
- For very small n (e.g., n ≤ 16) a simple O(n²) algorithm is actually faster than Quicksort.
- When the sublist is small, use another sorting algorithm.

Worst Case
- In the worst case, every partition might split the list of size j - i + 1 into a list with 1 element, and a list with j - i elements.
- A partition is split into sublists of size 1 & j-i when one of the first two items in the sublist is the largest item in the sublist which is chosen by findpivot.
- When will this worst case partitioning always occur?
Iterative Quicksort

Iterative Conversion
- Iterative implementation requires using a stack to store the partition boundaries remaining to be sorted.
- Assume a stack implementation of elements consisting of two integers:

```c
typedef struct {
    int low,
    hi;
} ItemType;
```

Partitioning
- At the end of any given partition, only one subpartition need be stacked.
- The second subpartition (equated to the second recursive call), need not be stacked since it is immediately used for the next subpartitioning.

Stacking
- The order of the recursive calls, (i.e., the sorting of the subpartitions) may be made in any order.
- Stacking the larger subpartition assures that the size of the stack is minimized, since the smaller subpartition will be further divided less times than the larger subpartition.

Iterative Quicksort Code

```c
Quicksort Function (iterative)
void QuickSort( raytype A[], int i, int j ) {
    // sort the array from i ... j
    keytype pivotKey;
    int pivotIndex, tmpBnd;
    int k; //index of partition >= pivot
    ItemType parts;
    Stack subParts;

    InitializeStack( subParts );
    parts.low = i;   parts.hi = j;
    Push ( parts , subParts );
    while ( !EmptyStack( subParts ) ) {
        Pop( subParts , parts );
        while ( parts.hi > parts.low ) {
            pivotIndex = FindPivot( A, parts.low, parts.hi );
            if (pivotIndex != -1) {
                pivotKey = A[ pivotIndex].key;
                k = Partition( A, parts.low, parts.hi , pivotKey );
                // push the larger subpartition
                if ( (k-parts.low)  > (parts.hi-k) ) { //stk low part
                    tmpBnd = parts.hi;
                    parts.hi = k-1;
                    Push ( parts , subParts );
                    parts.low = k; //set current part to upper part
                    parts.hi = tmpBnd;
                } //end if
                else { // stack upper (larger) part
                    tmpBnd = parts.low;
                    parts.low = k;
                    Push ( parts , subParts );
                    parts.hi = tmpBnd; //set current part to lower part
                    parts.hi = k-1;
                } // end else
            } // end if
        } // end while
        parts.hi = parts.low;
    } // end while
}
```

Computer Science Dept Va Tech August, 1998
Intro Data Structures & SE
©1995-1998 Barnette ND, McQuain WD
Quicksort Efficiency

Graphical Trace

Working

Working

Working

Minor Improvements

- All function calls should be replaced by inline code to avoid function overhead.
- Current partition bounds should be held in register variables.

Sorting Considerations

General

- Swap pointers instead of copying records

```
record 2
record 1
record 2
record 1
```

- Carefully investigate the average data arrangement in order to select the optimal sorting algorithm.
- No one algorithm works the best in all cases.

Special

- If sort key (member) consists of consecutive (unique) N integers they can be easily mapped onto the range 0 .. N-1 & sorted.
- If the n elements are initially in array A, then:

```c
raytype A, B;
for ( i = 0; i < n; i++ )
```

- Takes $O(n)$ time.
- Requires exactly 1 record with each key value!
- Special case of Bin Sorting. (If integers are not consecutive, but within a reasonable range, used bit flags can be used to denote empty array slots.)