Sorting

1 Introduction

The sorting problem is: given a sequence of n items from an ordered domain, permute the items into an increasing sequence. Duplicates are permitted, so 2,3,3,5,6 is an increasing sequence. A sorting algorithm is called *coparison-based* if each branch point of the algorithm is a comparison between two of the original items. If $\mathcal{A}$ is any comparison-based algorithm for sorting a sequence of length n, the number of comparisons during an execution of $\mathcal{A}$ is at least $\log_2 n! \approx n \log_2 n$ in the worst case.[1]

For each of the sorting algorithm described below, we assume that we are given a sequence $x_1, x_2, \ldots x_n$.

2 Selection Sort

Let $\mathcal{X}$ be the set $\{x_1, x_2, \ldots x_n\}$. Pseudocode for selection sort:

For each i from 1 to n
  Let y_i be the smallest member of $\mathcal{X}$. (Search step)
  Delete y_i from $\mathcal{X}$.

The sequence $y_1, y_2, \ldots y_n$ is the output. The time complexity of selection sort depends on how the search step is implemented. If linear search is used, selection sort takes $O(n^2)$ time.

3 Insertion Sort

The traditional version is to maintain an ordered array A, which is initially empty. Here is the pseudocode:

For each i from 1 to n
  Insert x_i into $A = \{a_j\}$ such that A remains ordered.
A is the output.

Insertion of an item into an ordered array takes linear time. To insert an item x, traverse A until you find an element $a_j \geq x$. Then:
For all $k \geq i$ in reverse order
  $a_{k+1} = a_k$
$a_j = x$

This version of selection sort takes $O(n^2)$ time.

[1] That means, the worst case sequence of length n. For example, *bubblesort* is very fast in some cases, such as if the sequence is already sorted, but is much slower in many other cases. The worst case time is defined to be the maximum time, taken over all $n!$ permutations of the input.

4 Mergesort

Mergesort and quicksort are the two standard *divide-and-conquer* sorting algorithms. For each of these, the data are divided into two smaller sequences, each of which is (recursively) sorted; the two sorted sequences are then combined.

4.1 Merging

Suppose $\mathcal{X} = x_1, \ldots x_n$ and $\mathcal{Y} = y_1, \ldots y_m$ are ordered. We merge $\mathcal{X}$ and $\mathcal{Y}$ into an ordered sequence $\mathcal{Z} = z_1 \ldots z_{n+m}$ as follows:

```
i = 1
j = 1
For k from 1 to n + m
    While i ≤ n and j ≤ m
        If x_i < y_j
            z_k = x_i
            i++
        Else
            z_k = y_j
            j++
        k++
    If j = m + 1 copy the remaining items of X into Z
    Else copy the remaining items of Y into Z
Z is the output.
```

We now give the pseudocode for mergesort of $\mathcal{X} = x_1, \ldots x_n$:

```
If n = 1, X is already sorted and we are done.
Else
    X1 = x_1 ... x_{n/2}
    X2 = x_{n/2+1} ... x_n
    Recursively sort X1, with output Y1
    Recursively sort X2, with output Y2
    Merge Y1 and Y2 with output Z
Z is the output.
```

The time complexity of mergesort is $\Theta(n \log n)$.

4.2 Example Computation

```
ATIHESVL
ATIH ESVL
AT IH ES VL
AT HI ES LV
AHIT ELSV
AEHILSTV
```

5 Quicksort

Pseudocode for quicksort. The input is a sequence $\mathcal{X} = x_1 \ldots x_n$.

If $n \leq 1$, $\mathcal{X}$ is already sorted and we are done.
Else
  Pick **pivot** $\in \mathcal{X}$.
  Partition $\mathcal{X}\backslash\{\textbf{pivot}\}$ into two sets $\mathcal{X}_1$ and $\mathcal{X}_2$ such that
    $x \leq \textbf{pivot}$ for all $x \in \mathcal{X}_1$
    $x \geq \textbf{pivot}$ for all $x \in \mathcal{X}_2$
  Recursively sort $\mathcal{X}_1$, with output $\mathcal{Y}_1$
  Recursively sort $\mathcal{X}_2$, with output $\mathcal{Y}_2$
  $\mathcal{Z} = \mathcal{Y}_1\{\textbf{pivot}\}\mathcal{Y}_2$ (concatenation)
$\mathcal{Z}$ is the ouput.

5.1 Problems with Quicksort

The time complexity of quicksort is $\Omega(n \log n)$, but is $O(n^2)$ in the worst case. In mergesort, the list is divided into two nearly equal sublists, but in quicksort, with a bad pivot, they might not be even close to equal. In the first version of quicksort that I read, the first item was chosen to be the pivot, which makes the overall running time quadratic if the list is sorted. However it is chosen, the pivot must first be swapped into the first position before partition begins. The ideal choice of the pivot is the iedian, but finding that median deterministically takes so much time that quicksort would no longer be "quick."

There are a number of stragies used in practice in an effort to avoid this problem. Here are some of them.

1. Pick the pivot to be the middle item in the array. This is very easy to do, and almost always leads to $O(n \log n)$ time. In the C++ code given below, the middle item is chosen for the pivot.

2. Pick the the median of the first, last, and middle items in the array. This is a common suggestion, but the improvement, if any, is slight.

3. Pick a random item as the pivot. Of course, your code is no longer deterministic. Most computers have a built-in pseudo-random number generator. With this choice, the probability of taking quadratic time is vanishingly small, and the expected time is $\Theta(n \log n)$. Expected time can be decreased somewhat by picking the pivot to be the median of three randomly chosen items.

4. Pick three random items, and let the pivot be the median of those three. This is slightly better than picking a random pivot, but only by about 5% if the array is very large, and not at all better if the array has length less than 3000.

5. Ignore the problem: just pick the first item. This is dangerous, since the array might already be sorted, or almost sorted.

C++ Code for Quicksort

Assume there is a given array $A[n]$, for $n >= 1$, of some ordered type. We will assign int to be the type of A.

```
void swap(int&x,int&y)
 {
  int temp = x;
  x = y;
  y = temp;
 }

void quicksort(int first, int last)    // input condition: first <= last
  // sorts the subarray A[first .. last]
 {
  if(first < last) // otherwise there is only one entry
   {
    int mid = (first+last)/2;
    swap(A[first],A[mid]);
    int pivot = A[first];
    int lo = first;
    int hi = last;
     // loop invariant holds
    while(lo < hi) // the partition loop
     {
       // loop invariant holds
      while(A[lo+1] < pivot)lo++;
      while(A[hi] > pivot)hi--;
      if(lo+1 < hi)
       {
        swap(A[lo+1],A[hi]);
        lo++;
        hi--;
       }
      else if(lo+1 == hi) hi--;
     }
     // loop invariant holds
    swap(A[first],A[lo]);
    // now A[lo] = pivot
    if(first < lo) quicksort(first,lo-1);
    if(lo+1 < last) quicksort(lo+1,last);
   }
 }
```

```
int main()
 {
  quicksort(0,N-1);
  cout << endl;
  return 1;
 }
```

6 Polyphase Mergesort

Despite its name, polyphase mergesort is not a form of mergesort. In a sequence, we define a *run* to be a maximal ordered subsequence. For example, the sequence AYUGWIBCESOPNF separated into runs is AY U GW I BCDES OP N F.

The algorithm works as follows. In the first phase, deal the runs of the sequence out into two files.
We separate those files into runs, merge the first runs of the files 1 and 2, write into file 3, the second runs of files 1 and 2 into file 4, the third The time complexity of polyphase mergesort is $O(n \log n)$. Each phase takes $\Theta(n)$ time, and there are $O(\log n)$ phases, since, if there are at most k runs in each file after a phase, there at most $k/2$ runs in each file after the next phase. For example starting with ZKTUYWQFGPLARMXNJDK, we write to files f1 and f2 on odd-numbered phases and to f3 and f4 on even-numbered phases.
first phase
f1: ZWFGPARNDK
f2: KTUYQLMXJ
second phase
f3: KTUYZFGLMPXN
f4: QWAJRDK
third phase
f1: KQTUWYZDKN
f2: AFGJLMPRX
fourth phase
f3: AFGJLMPRTUWXYZ
f4: DKN
fifth and last phase
f1: ADFGJKLMNPRTUWXYZ
f2: empty

7 Tree Sort

Tree sort is a fast form of insertion sort. We insert the items of $\mathcal{X}$ one at a time into a search structure, such as a binary search tree, and then create an ordered array by traversing the search struction in alphabetic order, which is *inorder* for a binary search tree.

Since a binary search tree has very bad worst case performance, the time complexity of tree sort quadratic in the worst case. In order to guarantee $O(n \log n)$ time for tree sort, we must use a balanced binary search tree, or some other choice of search structure where insertion takes $O(\log n)$ time. However, according to Lipton, there is no need to worry about that in practice.

However, Lipton's observation does not always hold in practice, as I discovered in my research. See below for my solution.

8 Treap Sort

This is the same as tree sort, except that we use a treap, which is a more sophisticated kind of binary search tree. Construction of a treap requires generating random numbers, and in the worst case, treap sort takes quadratic time. However, for a large example, the probability of that is comparable to the probability of being hit by two asteroids simultaneously. In my research project, I used a treap to sort my data and the speedup with quite noticeable.

9 Heap Sort

Heap sort is a sophisticated form of selection sort, and takes $\Theta(n \log n)$ time. Although asymptotically optimal, heapsort generally takes longer than other $O(n \log n)$-time sorting algorithms.

The first phase, called *heapify*, is to place the items of $\mathcal{X}$ into a maxheap. The unsophisticated version of heapify is to start with an empty heap and then insert the items one at a time. Each insertion takes $O(\log n)$ time, and so this version of heapify takes $O(n \log n)$ time. However, there is a more sophisticaed verson of heapify which takes $\Theta(n)$ time. As a consequence, the time of heapify is cut almost in half. I call this method "bottom-up bubbledown." Here is an example. As usual, we implement the heap as an almost complete binary tree stored in an array in level order. Suppose our file is THJESYIFWZGPZBRN. We simply start with an array containing those items, then execute bubbledown at each position, with decreasing indices starting from the middle. Here are the steps of heapify. (Remember, it's a maxheap.)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
T	H	J	E	S	Y	I	F	W	Z	G	P	X	B	R	N	
T	H	J	E	S	Y	I	N	W	Z	G	P	X	B	R	F	bubbledown(F)
T	H	J	E	S	Y	R	N	W	Z	G	P	X	B	I	F	bubbledown(I)
T	H	J	E	S	Y	R	N	W	Z	G	P	X	B	I	F	bubbledown(Y) nothing happens
T	H	J	E	Z	Y	R	N	W	S	G	P	X	B	I	F	bubbledown(S)
T	H	J	W	Z	Y	R	N	E	S	G	P	X	B	I	F	bubbledown(E)
T	H	Y	W	Z	J	R	N	E	S	G	P	X	B	I	F	bubbledown(J)
T	H	Y	W	Z	X	R	N	E	S	G	P	J	B	I	F	bubbledown(J), cont
T	Z	Y	W	H	X	R	N	E	S	G	P	J	B	I	F	bubbledown(H)
T	Z	Y	W	S	X	R	N	E	H	G	P	J	B	I	F	bubbledown(H), cont
Z	T	Y	W	S	X	R	N	E	H	G	P	J	B	I	F	bubbledown(T)
Z	W	Y	T	S	X	R	N	E	H	G	P	J	B	I	F	bubbledown(T), cont

Heap order is achieved.

The second, and longer, phase is the selection sequence. If we used the above example, it would be too long, so I'll start with something smaller.

During the second phase, the array has two parts. The left, shrinking, part, is the maxheap, while the right, growing part (shown with bold letters) is the part already sorted. Each iteration consists of the swap of the maximum item in position 1 to the item in the last position of the heap, followed by restoration of heap order. The item that was in position 1 becomes the newest item in the sorted portion, which grows by one. The heap is decremented. Then the item in position 1, which is out of place, bubbles down until heap order is restored.

After n iterations, the heap is empty and the sorted portion is the entire array.

1	2	3	4	5	6	7	8	9	
H	U	B	J	E	R	P	W	Q	
H	U	B	W	E	R	P	J	Q	bubbledown(J)
H	U	R	W	E	B	P	J	Q	bubbledown(B)
H	W	R	U	E	B	P	J	Q	bubbledown(U)
W	H	R	U	E	B	P	J	Q	bubbledown(H)
W	U	R	H	E	B	P	J	Q	bubbledown(H), cont
W	U	R	Q	E	B	P	J	H	bubbledown(H), cont
W	U	R	Q	E	B	P	J	H	bubbledown(H) heap order
H	U	R	Q	E	B	P	J	**W**	swap W and H
U	H	R	Q	E	B	P	J	**W**	bubbledown(H)
U	Q	R	H	E	B	P	J	**W**	bubbledown(H), cont
U	Q	R	J	E	B	P	H	**W**	bubbledown(H), cont
H	Q	R	J	E	B	P	**U**	**W**	swap U and H
R	Q	H	J	E	B	P	**U**	**W**	bubbledown(H)
H	Q	P	J	E	B	**R**	**U**	**W**	swap R and H
Q	H	P	J	E	B	**R**	**U**	**W**	bubbledown(H)
Q	J	P	H	E	B	**R**	**U**	**W**	bubbledown(H), cont
B	J	P	H	E	**Q**	**R**	**U**	**W**	swap Q and B
P	J	B	H	E	**Q**	**R**	**U**	**W**	bubbledown(B)
E	J	B	H	**P**	**Q**	**R**	**U**	**W**	swap P and E
J	E	B	H	**P**	**Q**	**R**	**U**	**W**	bubbledown E
J	H	B	E	**P**	**Q**	**R**	**U**	**W**	bubbledown E, cont
E	H	B	**J**	**P**	**Q**	**R**	**U**	**W**	swap E and J
H	E	B	**J**	**P**	**Q**	**R**	**U**	**W**	bubbledown(E)
B	E	**H**	**J**	**P**	**Q**	**R**	**U**	**W**	swap H and B
E	B	**H**	**J**	**P**	**Q**	**R**	**U**	**W**	bubbledown(B)
B	**E**	**H**	**J**	**P**	**Q**	**R**	**U**	**W**	swap E and B
B	**E**	**H**	**J**	**P**	**Q**	**R**	**U**	**W**	sorted