Sorting

1 Introduction

The sorting problem is: given a sequence of n items from an ordered domain, permute the terms 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, for example "`if(condition)statement; else statement;`" 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 example, *bubblesort* takes only linear time if the array is sorted, but in the worst case takes quadratic time.

When we write code for an algorithm, we assume items are of integer type. In many sorting algorithms, exchange of two items is needed, and can be implemented in C++ as follows.

```
void swap(int&x,int&y)
 {
  int tmp = x;
  x = y;
  y = tmp;
 }
bool sortpair(int&x,int&y)
 {
  if(y < x)
   {
    swap(x,y);
    return true;
   }
  else return false;
  kount++;
 }
```

We will also need a linked list data structure and operations insert and traverse:

```
struct node;
typedef node*list;
struct node
 {
  int item;
  list link;
 };

void insert(list&root, int newitem)
```

[1] That means, the worst case sequence of length n.

```
 {
  list newlist = new node;
  newlist->item = newitem;
  newlist->link = root;
  root = newlist;
 }

void traverse(list root)
 {
  if(root)
   {
    cout << root->item << " ";
    traverse(root->link);
   }
 }
```

We will use this implementation of binary search tree:

```
struct treenode;
typedef treenode*tree;
struct treenode
 {
  int item;
  tree left,
  tree right;
 }

void insert(tree&root,int newitem)
 {
  if(tree)
   if(newitem < tree->item) insert(tree->left,newitem)
   else insert(tree->right,newitem)
  else
   {
    root = new treenode;
    root->item = newitem;
   }
 }

void inorderwrite(tree root)
 {
  if(tree)
   {
    inorderwrite(root->left);
    cout << root-> item << endl;
```

```
    inorderwrite(root->right);
   }
 }
```

We will use these code fragments later in this document.

2 Bubblesort

Bubblesort was initially published by Edward Friend, and was subsequently rediscovered many times. Although it is (or was) commonly believed to be efficient, its worst and average case times are both quadratic. The algorithm consists of repeated passes, where at each pass adjacent elements are compared and possibly swapped. If no pairs are swapped during a pass, the array is sorted, and no more passes are needed.

```
void bubblesort()
 {
  bool sorted = false;
  for(int i = n-1; i > 0 and not sorted; i--)
   {
    sorted = true;
    for(int j = 0; j < i; j++)
     if(sortpair(A[j],A[j+1])
      sorted = false;
   }
 }
```

2.1 Shellsort

Shellsort (or Shell sort) was invented by Donald Shell in 1959. At one time the recommended sorting algorithm in some textbooks, it is still used in some applications.

Shellsort uses a decreasing sequence of integers, called *gaps*. $g_1, g_2, \ldots g_m$, where $g_1 < n$ and $g_m = 1$. Shellsort consists of one phase for each gap. If g_i is the current gap, the array A is partitioned into g_i subarrays, each an arithmetic sequence with increment g_i. The phase consists of bubblesorting each subarray; the next phase uses gap g_{i+1}. After the last phase, where the gap is 1, the array is sorted. But what is the ideal gap sequence? Shell used a decreasing sequence of powers of 2, which was subsequenly found to be the worst choice. Using a better gap sequence, the overall time complexity of Shellsort is $O(n^{\frac{3}{2}} = O(n\sqrt{n})$, a vast improvement over quadratic.

3 Selection Sort

Given a set D of n items, selection consists of repeatedly selecting the smallest item in D and moving it to an array. After n steps, the array will contain the original items of D is sorted order. Here is the array

implentation of selection sort, where the items are initally the entries of the array A is arbitrary order, and at the end the same items are in A is sorted order, and $D = A[i \ldots n-1]$ contains the remaining items.

```
for(int i = 0; i < n; i++)
 for(int j = i; j < n; j++)
  sortpair(A[i],A[j]);
```

After i iterations of the outer loop, the subarray $A[0 \ldots i-1]$ contains the smallest i items in sorted order. Write the loop invariant of the inner loop.

The time complexity of the above code is quadratic, since combined number of iterations of the loops is roughly $n^2/2$.

3.1 Heapsort

Heapsort is a form of selection sort, speeding up the inner loop by using a minheap data structure. In the simpler version, all items are placed in a heap, the deletemin is executed n times to give the items in sorted order. Here is code, where we assume that H is a minheap.

```
heapsort()
 {
  initialize(H); // H becomes an empty heap
  for(int i = 0; i < n; i++)
   insert(H,A[i]);
  for(int i = 0; i < n; i++)
   A[i] = deletemin(H);
 }
```

Insertion of an item takes $O(\log n)$ time, using bubbleup, while each deletemin also takes $O(\log n)$ time. Thus, the sorst case time complexity of heapsort is $\Theta(n \log n)$. In Section 3.2, we give a more complex version of heapsort which takes slightly more than half the time of the simple version, but still $O(n \log n)$ time in the worst case.

3.2 Faster Heapsort

Heapsort can be sped up by almost a factor of two. We use a maxheap, and introduce an initial phase, *heapify*, which rearranges the items of the initial array into a maxheap in linear time. 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 of heapify "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

Insertion Sort

Dual to selection sort, insertion sort consists of insertion of arbitrary items into a sorted structure, such as an ordered array or a binary search tree. Here is C++ code for the quadratic time array implementation of insertion sort on an arbitrary array $A[n]$.

```
void insertionsort()
 {
  for(int i = 1; i < n; i++)
   for(j = i; j >= 1 and x[j] < x[j-1]; j--)
    swap(x[j],x[j-1];
 }
```

The outer loop iterates $n-1$ times. At the start of iteration i of that loop, the subarray $A[0] \ldots A[i-1]$

is sorted, although those items may not be in their final places. During that iteration, $A[i]$ is inserted into that subarray, maintaining sorted order. After $n-1$ iterations of the outer loop, the sorted subarray is all of A, and hence we are done. Each insertion takes $O(n)$ time and there are $n-1$ insertions, hence the overall time complexity is $O(n^2)$.

3.3 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.

The time complexity of tree sort is quadratic in the worst case. $O(n \log n)$ time for tree sort, We can used 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, but Lipton's observation is not always correct. See below for my solution, given an unsorted array $A[n]$, and an empty binary search tree.

```
for(int i = 0; i < n; i++)
 insert(root,A[i]);
inorderwrite(root);
```

3.4 Treap Sort

Tree sort can be used on a balanced binary tree, such as an AVL tree, a red-black tree, or a treap. In the worst case, treap sort takes quadratic time. However, the worst case expected time is $\Theta(n \log n)$ In a research project, I used a treap to sort my data and the speedup was quite noticeable.

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 merged.

4.1 Merging

The most important subprogram of mergesort is *merge* Two sorted arrays of length p and q, respectively, can be merged into a single sorted array in $O(p+q)$ time. Similarly, sorted linked lists of length p and q can be merged tnto a sorted linked list in $O(p+q)$ time.

Merging Sorted Arrays. The following C++ code merges sorted arrays $X[p]$ and $Y[q]$ into a sorted array $A[p+q]$.

```
void merge(int X[],int p,int Y[],int q,int A[]);
 {
  int i = 0;
  int j = 0;
  while(i < p and j < q)
   if{X[i] < Y[j])
    A[i+j] = X[i++];
   else
    A[i+j] = Y[j++];
  while(i < p)
   A[i+q] = X[i++]
  while(j < q)
   A[p+j] = Y[j++]
 }
```

A drawback of array based merge is that additional workspace is need to store the merged array. Instead, we can use a linked list.

```
list merge(list k, list ell)
 {
  if(k == null) return ell;
  else if ell == null return k;
  else if(ell->item < k->item)
  return merge(ell,k);
  else
   {
    k->link = merge(k->link,ell);
    return k;
   }
 }


list mergesort(list x)
 {
  if(x and x->link)
   {
    list a = x;
    list b = x->link;
    while(b and b->link)
     {
      a = a->link;
      b = b->link->link;
     }
    list y = a->link;
```

```
    a->link = null;
    x = mergesort(x);
    y = mergesort(y);
    return merge(x,y);
   }
  else return x;
 }
```

5 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 can separate runs by commas: Z,KTUY,W,Q,FGP,L,AR,MX,N,J,DK, 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

6 Quicksort

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

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
    $x \leq$ **pivot** for all $x \in \mathcal{X}_1$
    $x \geq$ **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.

6.1 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.

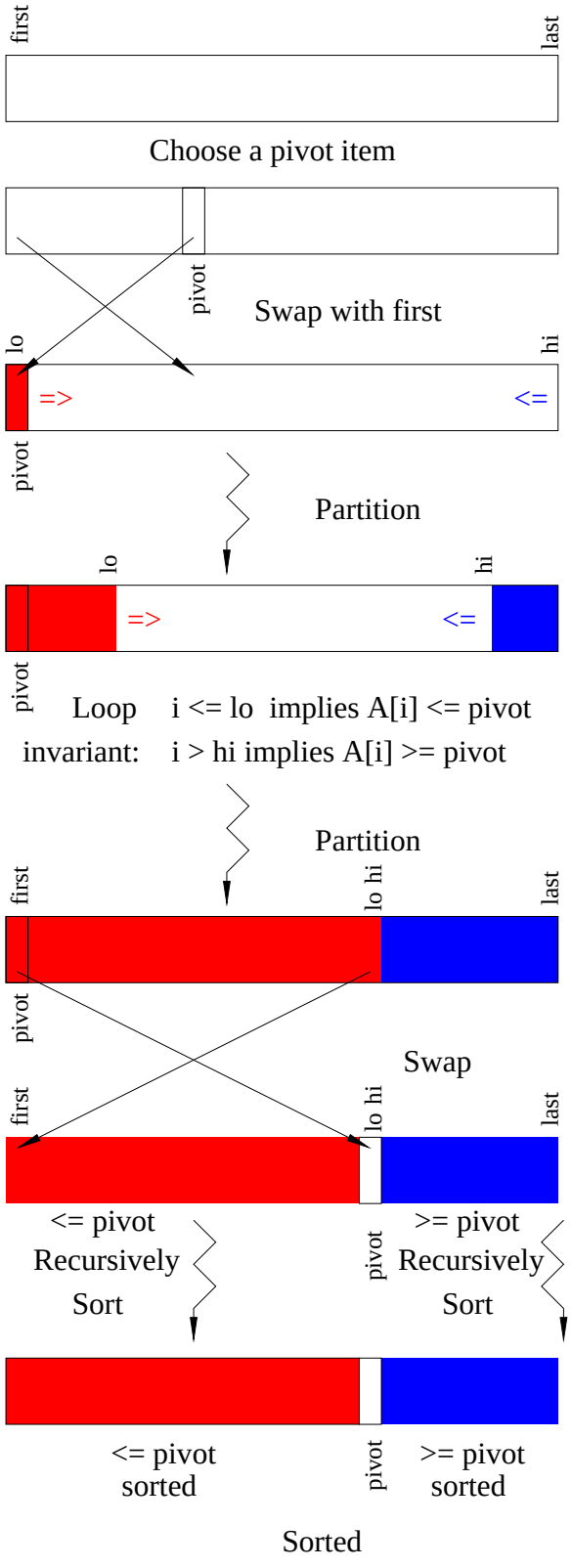
4. Pick three random items, and let the pivot be the median of those three. This is slightly better than picking a random pivot, but only by about 5% if the array is very large, and not at all better if the array has length less than 3000.

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

6.2 C++ Code for Quicksort

```
void quicksort(int first, int last)
  // sort subarray A[first .. last]
 {
  if(first < last) // otherwise done
   {
    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) // 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);
  return 1;
 }
```



7 Time and Space Complexities

Auxiliary space complexity is space used over and above the space needed to store the input and output. We do not count the space used for pointers in a linked list or binary tree.

Algorithm	Time	Auxiliary Space
Bubble Sort	$O(n^2)$	$O(1)$
Array Selection Sort	$O(n^2)$	$O(1)$
Array Insertion Sort	$O(n^2)$	$O(1)$
Array Merge Sort	$O(n \log n)$	$O(n)$
Linked List Merge Sort	$O(n \log n)$	$O(\log n)$
Heap Sort	$O(n \log n)$	$O(1)$
Quick Sort	$O(n^2)$	$O(\log n)$
Polyphase Mergesort	$O(n \log n)$	$O(n)$
Tree Sort	$O(n^2)$	$O(n)$
Randomized Algorithms		
Quick Sort Random Pivot	Expected $O(n \log n)$	$O(\log n)$
Treap Sort	Expected $O(n \log n)$	$O(n)$