Sorting

1 Introduction

Insertion sort is the sorting algorithm that splits an array into a sorted and an unsorted region, and repeatedly picks the lowest index element of the unsorted region and inserts it into the proper position in the sorted region, as shown in Figure 1.

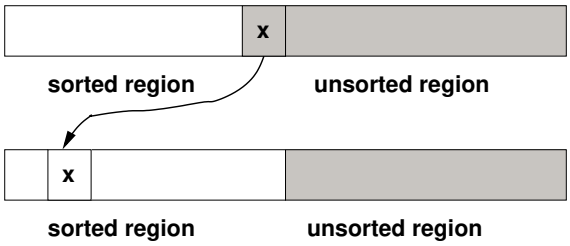

Figure 1: Insertion sort modeled as an array with a sorted and unsorted region. Each iteration moves the lowest-index value in the unsorted region into its position in the sorted region, which is initially of size 1.

The process starts at the second position and stops when the rightmost element has been inserted, thereby forcing the size of the unsorted region to zero.

Insertion sort belongs to a class of sorting algorithms that sort by comparing keys to adjacent keys and swapping the items until they end up in the correct position. Another sort like this is bubble sort. Both of these sorts move items very slowly through the array, forcing them to move one position at a time until they reach their final destination. It stands to reason that the number of data moves would be excessive for the work accomplished. After all, there must be smarter ways to put elements into the correct position.

The insertion sort algorithm is below.

```
for ( int i = 1; i < a.size(); i++) {
    tmp = a[i];
    for ( j = i; j >= 1 && tmp < a[j-1]; j = j-1)
        a[j] = a[j-1];
    a[j] = tmp;
}
```

You can verify that it does what I described. The step of assigning to tmp and then copying the value into its final position is a form of efficient swap. An ordinary swap takes three data moves; this reduces the swap to just one per item compared, plus the moves at the beginning and end of the loop.

The worst case number of comparisons and data moves is $O(N^2)$ for an array of N elements. The best case is $\Omega(N)$ data moves and $\Omega(N)$ comparisons. We can prove that the average number of comparisons and data moves is $\Omega(N^2)$.

2 A Lower Bound for Simple Sorting Algorithms

An inversion in an array is an ordered pair (i, j) such that $i < j$ but $a[i] > a[j]$. An exchange of adjacent elements removes one inversion from an array. Insertion sort needs to remove all inversions by swapping, so if there are m inversions, m swaps will be necessary.

Theorem 1. *The average number of inversions in an array of n distinct elements is n(n-1)/4.*

Proof. Let L be any list and L_R be its reverse. Pick any pair of elements $(i, j) \in L$ with $i < j$. There are $n(n-1)/2$ ways to pick such pairs. This pair is an inversion in exactly one of L or L_R. This implies that L and L_R combined have exactly $n(n-1)/2$ inversions. The set of all $n!$ permutations of n distinct elements can be divided into two disjoint sets containing lists and their reverses. In other words, half of all of these lists are the reverses of the others. This implies that the average number of inversions per list over the entire set of permutations is $n(n-1)/4$. □

Theorem 2. *Any algorithm that sorts by exchanging adjacent elements requires* $\Omega(n^2)$ *time on average.*

Proof. The average number of inversions is initially $n(n-1)/4$. Each swap reduces the number of inversions by 1, and an array is sorted if and only if it has 0 inversions, so $n(n-1)/4$ swaps are required. □

3 Shell Sort

Shell sort was invented by Donald Shell. It is like a sequence of insertion sorts carried out over varying distances in the array and has the advantage that in the early passes, it moves data items close to where they belong by swapping distant elements with each other. Consider the original insertion sort modified so that the gap between adjacent elements can be a number besides 1:

Listing 1: A Generalized Insertion Sort Algorithm

```
int gap = 1;
for ( int i = gap; i < a.size(); i++) {
    tmp = a[i];
    for ( j = i; j >= gap && tmp < a[j-gap]; j = j-gap)
        a[j] = a[j-gap];
    a[j] = tmp;
}
```

Now suppose we let the variable `gap` start with a large value and get smaller with successive passes. Each pass is a modified form of insertion sort on each of a set of interleaved sequences in the array. When the `gap` is h, it is h insertion sorts on each of the h sequences

$0, h, 2h, 3h, \ldots, k_0 h,$

$1, 1+h, 1+2h, 1+3h, \ldots, 1+k_1 h$

$2, 2+h, 2+2h, 2+3h, \ldots, 2+k_2 h$

$\ldots$

$h-1, h-1+h, h-1+2h, \ldots, h-1+k_{h-1} h$

where k_i is the largest number such that $i + k_i h < n$. For example, if the array size is 15, and the gap is $h = 5$, then each of the following subsequences of indices in the array, which we will call ***slices***, will be insertion-sorted independently:

```
slice 0:  0    5    10
slice 1:  1    6    11
slice 2:  2    7    12
slice 3:  3    8    13
slice 4:  4    9    14
```

For each fixed value of the gap, h, the sort starts at array element $a[h]$ and inserts it in the lower sorted region of the slice, then it picks $a[h+1]$ and inserts it in the sorted region of its slice, and then $a[h+2]$ is inserted, and so on, until $a[n-1]$ is inserted into its slice's sorted region. For each element $a[i]$, the sorted region of the slice is the set of array elements at indices i, $i-h$, $i-2h$, ... and so on. When the gap is h, we say that the array has been ***h-sorted***. It is obviously not sorted, because the different slices can have different values, but each slice is sorted. Of course when the gap $h = 1$ the array is fully sorted. The following table shows an array of 15 elements before and after a 5-sort of the array.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Original Sequence:	81	94	11	96	12	35	17	95	28	58	41	75	15	65	7
After 5-sort:	35	17	11	28	7	41	75	15	65	12	81	94	95	96	58

The intuition is that the large initial gap sorts move items much closer to where they have to be, and then the successively smaller gaps move them closer to their final positions.

In Shell's original algorithm the sequence of gaps was $n/2$, $n/4$, $n/8$,..., 1. This proved to be a poor choice of gaps because the worst case running time was no better than ordinary insertion sort, as is proved below.

Listing 2: Shell's Original Algorithm

```
for ( int gap = a.size()/2; gap > 0; gap /=2)
    for ( int i = gap; i < a.size(); i++) {
        tmp = a[i];
        for ( j = i; j >= gap && tmp < a[j-gap]; j = j-gap)
            a[j] = a[j-gap];
        a[j] = tmp;
}
```

3.1 Analysis of Shell Sort Using Shell's Original Increments

Lemma 3. *The running time of Shell Sort when the increment is h_k is $O(n^2/h_k)$.*

Proof. When the increment is h_k, there are h_k insertion sorts of n/h_k keys. An insertion sort of m elements requires in the worst case $O(m^2)$ steps. Therefore, when the increment is h_k the total number of steps is

$$h_k \cdot O\left(\frac{n^2}{h_k^2}\right) = O\left(\frac{n^2}{h_k}\right)$$

□

Theorem 4. *The worst case for Shell Sort, using Shell's original increment sequence, is* $\Theta(n^2)$.

Proof. The proof has two parts, one that the running time has a lower bound that is asymptotic to n^2, and one that it has an upper bound of n^2.

Proof of Lower Bound.

Consider any array of size $n = 2^m$, where m may be any positive integer. There are infinitely many of these, so this will establish asymptotic behavior. Let the $n/2$ largest numbers be in the odd-numbered positions of the array and let the $n/2$ smallest numbers be in the even numbered positions. When n is a power of 2, halving the gap, which is initially n, in each pass except the last, results in an even number. Therefore, in all passes of the algorithm until the very last pass, the gaps are even numbers, which implies that all of the smallest numbers must remain in even-numbered positions and all of the largest numbers will remain in the odd-numbered positions. When it is time to do the last pass, all of the $n/2$ smallest numbers are sorted in the indices 0, 2, 4, 6, 8, ..., and the $n/2$ largest numbers are sorted and in indices 1, 3, 5, 7, and so on. This implies that the j^{th} smallest number is in position $2(j-1)$ (e.g., the second is in $2 \cdot 2 - 2 = 2$ and the third is in $2 \cdot 3 - 2 = 4$) and must be moved to position $j-1$ when $h = 1$. Therefore, this number must be moved$(2j-2)-(j-1) = j-1$ positions. Moving all $n/2$ smallest numbers to their correct positions when $h = 1$ requires

$$\sum_{j=1}^{n/2}(j-1) = \sum_{j=0}^{(n/2)-1} j = \frac{(n/2-1)(n/2)}{2} = \Theta(n^2)$$

steps. This proves that the running time is at least $\Theta(n^2)$.

Proof of Upper Bound.

By Lemma 3, the running time of the pass when the increment is h_k is $O(n^2/h_k)$. Suppose there are t passes of the sort. If we sum over all passes, we have for the total running time,

$$O\left(\sum_{k=1}^{t}\frac{n^2}{h_k}\right) = O\left(n^2\sum_{k=1}^{t}\frac{1}{h_k}\right) = O\left(n^2\sum_{k=1}^{t}\frac{1}{2^k}\right) = O(n^2)$$

because, as we proved in Chapter 1,

$$\sum_{k=1}^{t}\frac{1}{2^k} = 2 - \frac{1}{2^{t-1}} - \frac{t}{2^t} < 2$$

and this shows that n^2 is an asymptotic upper bound as well. It follows that the worst case is $\Theta(n^2)$. □

3.2 Other Increment Schemes

Better choices of gap sequences were found after Shell's initial publication of the algorithm. Before looking at some of the good ones, consider an example using a simple sequence such as 5, 3, 1.

Example

This uses the three gaps, 5, 3, 1 on an arbitrary array whose initial state is on the first row in the table below.

	0	1	2	3	4	5	6	7	8	9	10	11	12
Original Sequence:	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort:	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort:	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort:	11	12	15	17	28	35	41	58	75	81	94	95	96

Hibbard proposed using the sequence

$$h_1 = 1,\ h_2 = 3,\ h_3 = 7,\ h_4 = 15, \ldots,\ h_t = 2^t - 1$$

which is simply the numbers one less than powers of 2. Such a sequence, 1, 3, 7, 15, 31, 63, 127,..., has the property that no two adjacent terms have a common factor: if they had a common factor p, then their difference would also be divisible by p[1]. But the difference between successive terms is $(2^{k+1}–1)–(2^k–1) = (2^{k+1}–2^k) = 2^k$ which is divisible by powers of 2 alone. Hence the only number that can divide successive terms is a power of 2. Since all of the terms are odd numbers, none are divisible by 2, so they have no common factor.

Successive terms are related by the recurrence

$$h_{k+1} = 2h_k + 1$$

Sedgewick proposed a few different sequences, such as this one, defined recursively:

$$h_1 = 1,\ h_{k+1} = 3h_k + 1$$

which generates the sequence

$$1, 4, 13, 40, 121, 364, \ldots$$

This sequence has a worst case running time of $\Theta(n^{4/3})$. Hibbard's increment sequence turns out to be an improvement over Shell's original increment sequence.

Theorem 5. *The worst case running time of Shell Sort using Hibbard's sequence,* $h_1 = 1$, $h_2 = 3$, $h_3 = 7$, $h_4 = 15$, ..., $h_t = 2^t - 1$, *is* $\Theta(n^{3/2})$

Proof. Because this theorem states the bound using $\Theta()$ notation, we need to show that the running time grows no faster than $n^{3/2}$ and also that it grows no slower than it. The first part requires showing that $n^{3/2}$ is an asymptotic upper bound on the worst case running time. The second part requires showing that it is an asymptotic lower bound on the worst case running time.

[1] If a and b are two integers with a common factor p, then there are two integers r and q such that $a = qp$ and $b = rp$. Then $a - b = qp - rp = (q - r)p$, showing that the difference has this same common factor.

The asymptotic upper bound alone is proved here. Proving the lower bound requires showing that there is some input array such that the algorithm's running time will be proportional to $n^{3/2}$. As the upper bound shows it is at most $n^{3/2}$ and the lower bound shows at least one array requires $n^{3/2}$ time, this establishes that it is $\Theta(n^{3/2})$.

Lemma 3 establishes that the running time of a pass when the increment is h_k is $O(n^2/h_k)$. We will use this fact for all increments h_k such that $h_k > n^{1/2}$. This is not a tight enough upper bound for the smaller increments, and we need to work harder to get a tighter bound for them. The general idea will be to show that by the time the increments are that small, most elements do not need to be moved very far.

So we turn to the case when $h_k \leq n^{1/2}$. By the time that the array is h_k-sorted, it has already been h_{k+1}-sorted and h_{k+2}-sorted. Now consider the array elements at positions p and $p-i$ for all $i \leq p$. If i is a multiple of h_{k+1} or h_{k+2}, then $a[p-i] < a[p]$ because these two elements were part of the same slice, either for increment h_{k+1} or increment h_{k+2}, and so they were sorted with respect to each other. More generally, suppose that i is a non-negative linear combination of h_{k+1} and h_{k+2}, i.e., $i = c_1 h_{k+1} + c_2 h_{k+2}$, for some non-negative integers c_1 and c_2. Then

$$p - i = p - (c_1 h_{k+1} + c_2 h_{k+2}) = p - c_1 h_{k+1} - c_2 h_{k+2}$$

and

$$(p-i) + c_2 h_{k+2} = p - c_1 h_{k+1}$$

which implies that after the h_{k+2}-sort, $a[p-i] < a[p - c_1 h_{k+1}]$ because they are part of the same h_{k+2}-slice. Similarly, after the h_{k+1}-sort, $a[p - c_1 h_{k+1}] < a[p]$ because they are part of the same h_{k+1}-slice and $p - c_1 h_{k+1} < p$. Thus, $a[p-i] < a[p - c_1 h_{k+1}] < a[p]$.

Since $h_{k+2} = 2h_{k+1}$ by the definition of Shell's increment scheme, h_{k+2} and h_{k+1} are relatively prime. (If not, they have a common factor >1 and so their difference $h_{k+2} - h_{k+1} = 2h_{k+1} + 1 - h_{k+1} = h_{k+1} + 1$ would have a common factor with each of them, and in particular h_{k+1} and $h_{k+1} + 1$ would have a common factor, which is impossible.) Because h_{k+2} and h_{k+1} are relatively prime, their greatest common divisor (gcd) is 1. An established theorem of number theory is that if $c = \gcd(x, y)$ then there are integers a and b such that $c = ax + by$. When x and y are relatively prime, this implies that there exist a, b such that $1 = ax + by$, which further implies that every integer can be expressed as a linear combination of x and y. A stronger result is that all integers at least as large as $(x-1)(y-1)$ can be expressed as non-negative linear combinations of x and y. Let

$$\begin{aligned} m &\geq (h_{k+2} - 1)(h_{k+1} - 1) \\ &= (2h_{k+1} + 1 - 1)(2h_k + 1 - 1) \\ &= 2(h_{k+1})(2h_k) \\ &= 4h_k(2h_k + 1) = 8h_k^2 + 4h_k \end{aligned}$$

.

By the preceding statement, m can be expressed in the form $c_1 h_{k+1} + c_2 h_{k+2}$ for non-negative integers c_1 and c_2. From the preceding discussion, we can also conclude that $a[p-m] < a[p]$. What does this mean? For any number $i \geq 8h_k^2 + 4h_k$, $a[p-i] < a[p]$. Or stated the other way, $a[p]$ is definitely larger than all elements in its h_k-slice below $a[p - (8h_k^2 + 4h_k)]$, so the furthest it would have to be moved is $8h_k^2 + 4h_k$ cells down. Therefore, during the h_k-sort, no element has to be

moved more than $8h_k - 4$ times (since each moves in h_k increments each time.) There are at most $n - h_k$ such positions, and so the total number of comparisons in this pass is $O(nh_k)$.

How many increments h_k are smaller than $n^{1/2}$? Roughly half of them, because if we approximate n as a power of 2, say 2^t, then $\sqrt{n} = 2^{t/2}$ and the increments 1, 3, 7, ..., $2^{t/2} - 1$ are half of the total number of increments. For simplicity, assume t is an even number. Then the total running time of Shell Sort is

$$\begin{aligned} & O\left(\sum_{k=1}^{t/2} nh_k + \sum_{k=t/2+1}^{t} \frac{n^2}{h_k}\right) \\ = \; & O\left(n\sum_{k=1}^{t/2} h_k + n^2 \sum_{k=t/2+1}^{t} \frac{1}{h_k}\right) \end{aligned}$$

Now we have

$$\begin{aligned} n\sum_{k=1}^{t/2} h_k &= n\left(1+3+7+\cdots+2^{t/2}-1\right) \\ &< n\left(1+4+8+\cdots+2^{t/2}\right) \\ &= n\sum_{k=1}^{t/2} 2^k \\ &= n\cdot(2^{t/2+1}-1) \\ &< n\cdot 2\sqrt{n} \\ &= 2n^{3/2} \end{aligned}$$

which shows that the first sum is $O(n \cdot n^{1/2}) = O(n^{3/2})$. Since $h_{t/2+1} = \Theta(n^{1/2})$, the second sum can be written as

$$\begin{aligned} n^2 \sum_{k=t/2+1}^{t} \frac{1}{h_k} &= n^2\left(\frac{1}{2^{t/2+1}-1} + \frac{1}{2^{t/2+2}-1} + \cdots + \frac{1}{2^t-1}\right) \\ &= n^2\left(\frac{1}{2\sqrt{n}-1} + \frac{1}{4\sqrt{n}-1} + \cdots + \frac{1}{2^{t/2}\sqrt{n}-1}\right) \\ &\approx n^2 \cdot \frac{1}{\sqrt{n}}\left(\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{t/2}}\right) \\ &< 2n^{3/2} \end{aligned}$$

which shows that the second sum is also $O(n^{3/2})$. This shows that the upper bound in the worst case running time using Hibbard's sequence is $O(n^{3/2})$. The lower bound proof, i.e., that there is a sequence that achieves it is an exercise. □

Many other sequences have been studied. Studies have shown that the sequence defined by

$$(h_i, h_{i+1}) = (9\cdot(4^{i-1} - 2^{i-1}) + 1,\ 4^{i+1} - 6\cdot 2^i + 1) \quad i = 1, 2, \ldots$$

which generates 1, , 5, 19, 41, 109, ... has $O(n^{4/3})$ worst case running time (Sedgewick, 1986). The notation above means that the formula generates pairs of gaps.

Example

This shows how Shell sort works on an array using the gap sequence 13,4,1:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
After 13-sort	A	E	O	R	T	I	N	G	E	X	A	M	P	L	S
After 4-sort	A	E	A	G	E	I	N	M	P	L	O	R	T	X	S
After 1-sort	A	A	E	E	G	I	L	M	N	O	P	R	S	T	X

4 Heap Sort

The idea underlying heapsort is to convert the array to be sorted into a ***maximum heap*** in $O(n)$ steps, and to use this heap to sort the data efficiently. A maximum heap is a heap in which the heap order property is reversed – each node is *larger* than its children, not smaller. Having converted the array into a maximum heap, the algorithm repeatedly swaps the maximum element with the one at the end of the working array, decrements the size of the working array by one element afterward. This way, the next swap is with the element that immediately precedes the one just swapped.

4.1 The Algorithm

There are less efficient ways to do this than is described here. This algorithm builds the maximum heap, and then pretends to delete the maximum element by swapping it with the element in the last position, and decrementing the variable that defines where the heap ends in the array, effectively treating the largest element as not being part of the heap anymore. Because arrays are usually 0-based, the code is different from the code in the description of heaps in the chapter on priority queues. In particular, the 0 index of the array will be the root element of the heap. This does not pose a major problem, as it simply means we need to change the definition of the leftchild, rightchild, and parent functions to the following:

```
leftchild(i)   = 2*i + 1;
rightchild(i)  = 2*i + 2;
parent(i)      = (i-1)/2;
```

Proof that the above changes are correct is left as an exercise.

We define the following macros to make the code a bit clearer and easier to maintain:

```
#define LEFT(i)    2*(i)+1
#define RIGHT(i)   2*(i)+2
#define PARENT(i)  ((i)-1)/2
```

The `PercolateDown` function must be changed slightly from the one described in *Chapter 6, Priority Queues* for a few reasons:

- Because it needs to prevent an element from being dropped "too far down" the heap, it needs a parameter that acts like a sentinel within the heap. This is the function of the third parameter, `last`.
- Because we are creating a maximum heap rather than a minimum heap, the comparisons need to be reversed.
- Because it is part of a sorting algorithm, rather than a method of a class, it needs a parameter to store a reference to the data to be sorted. We assume that the data is a vector of Comparables and make the first parameter of type `vector<Comparable>`.

```
#define LEFT(i)    2*(i)+1
/*
 * @param int [in] last is the size of the array
 * @param int [in] hole is the index of the array element to be percolated
     down
 */
void PercolateDown( vector<Comparable> & array,  int hole, int last )
{
    int        child;
    Comparable temp = array[ hole ];

    while ( LEFT(hole) < last ) {
        child = LEFT(hole);
        if ( child < last -1 && array[child] < array[child+1] )
            child++;
        if( temp < array[ child ])
            array[ hole ] = array[ child ];
        else
            break;
        hole = child;
    }
    array[ hole ] = temp;
}
```

Using the above-defined `PercolateDown`, the heapsort algorithm is

```
void heapsort(vector<Comparable & a)
{
    // Build the heap using the same algorithm described in Chapter 6
    int N = A.size();
    for (int i = (N/2)-1; i >= 0; i--)
        // percolate down in a max heap stopping if we reach N-1
        PercolateDown (A, i, N );

    // A is now a heap
    // Now repeatedly swap the max element with the last element in the
       heap
    for ( int j = N - 1; j > 0; j--) {
        swap(A[0], A[j]);       // assume a swap function exists
        PercolateDown(A, 0, j);
    }
}
```

4.2 Analysis

In Chapter 6 we proved that building the heap takes $O(n)$ steps (to be precise, $2n$ steps.) In the second stage of the algorithm, the swap takes constant time, but the `PercolateDown` step, in the worst case, will require a number of comparisons and moves in proportion to the current height of the heap. The j^{th} iteration uses at most $2 \cdot \log(N - j + 1)$ comparisons. We have

$$\sum_{j=1}^{N-1} \log(N - j + 1) = \sum_{j=2}^{N} \log(j) = \log N!$$

It can be proved that $\log(N!) = \Theta(N \log\text{N})$[2], so heapsort in the worst case has a running time that is $\Theta(N \log N)$. A more accurate analysis will show that the most number of comparisons is $2N \log N - O(N)$.

Experiments have shown that heapsort is consistent and averages only slightly fewer comparisons than its worst case. The following theorem has been proved, but the proof is omitted here.

Theorem 6. *The average number of comparisons performed to heapsort a random permutation of n distinct items is $2n \log n$–$O(n \log \log n)$.*

5 Quicksort

Quicksort is the fastest known sorting algorithm when implemented correctly, meaning that the non-recursive version should be used and the longer partition should be stacked rather than the shorter one. Quicksort has an average running time of $\Theta(N \log N)$ but a worst case performance of $O(N^2)$. The reason that quicksort is considered the fastest algorithm for sorting when used carefully, is that a good design can make the probability of achieving the worst case negligible.

Here we review the algorithm and analyze its performance.

5.1 The Algorithm

Basic idea:

Let S represent the set of elements to be sorted, i.e., S is an array. Let quicksort*(S)* be defined as the following sequence of steps:

1. If the number of elements in S is 0 or 1, then return, because it is already sorted.
2. Pick any element v in S. Call this element v the ***pivot***.
3. Partition (S–v) into two disjoint sets $S_1 = \{x \in S \mid x \leq v\}$ and $S_2 = \{x \in S \mid x \geq v\}$ with some elements equal to v in $S_1$and others in S_2
4. Return quicksort(S_1) followed by v followed by quicksort(S_2).

[2]Replace the summation by the integral as an approximation to verify this.

Picking a Pivot

A poor choice of pivot will cause one set to be very small and the other to be very large. The result will be that the algorithm achieves its $O(n^2)$ worst case behavior. If the pivot is smaller than all other keys or larger than all other keys this will happen. We want a pivot that is the median element without the trouble of finding the median(because that is too expensive.) One could pick the pivot randomly, but then there is no guarantee about what it will be, and the cost of the random number generation buys little savings in the end.

A good compromise between safety and performance is to pick three elements and take their median. Doing this almost assuredly eliminates the possibility of quadratic behavior. A good choice is to choose the first, last, and middle elements of the array.

Partitioning

The best way to partition is to push the pivot element out of the way by placing it at the beginning or end of the array. Having done that, all of the implementations do basically the same thing, except for how they handle elements equal to the pivot.

The general idea is to advance `i` and `j` pointers towards the middle swapping elements larger than the pivot until `i` and `j` cross. The following example shows an array with the initial placements of the `i` and `j` pointers and the pivot, after it has been moved to the last position in the array. In each step, first `j` travels down the array looking for a culprit that doesn't belong, and then `i` travels up the array doing the same thing. When they each stop, the elements are swapped and they each advance one element. In the following example, the lowest index element is not less than or equal to the pivot, as would be true if a smarter way of choosing the pivot were used.

8	1	4	9	0	3	5	2	7	6
i								**j**	**pivot**
8	1	4	9	0	3	5	2	7	6
i							**j**		
2	1	4	9	0	3	5	8	7	6
	i					**j**			
2	1	4	9	0	3	5	8	7	6
			i			**j**			
2	1	4	5	0	3	9	8	7	6
				i	**j**				
2	1	4	5	0	3	9	8	7	6
					j	**i**			

It stops at this point and the pivot is swapped, so the final array is

```
2  1  4  5  0  3  6  8  7  9
```

The real issue is handling elements that are equal to the pivot. The safest way to handle elements equal to the pivot is to swap them as if they were smaller or larger, otherwise the algorithm can degrade to the worst case.

A recursive version of quicksort that uses this strategy is in the listing below. The function `median3` is listed below `quicksort`.

```
C median3( vector<C> & a, int first, int last);

void quicksort( T A[], int left, int right)
{
    // make sure array has at least ten elements:
    if ( left + 10 <=  right ) {
        // pick three values, sort them, and put the pivot into A[right],
        // the smallest of the three into A[left] and the largest into A[
           right-1].
        // assume that the function choose_pivot() does all of this.
        T pivot  = median3(A, left, right);

        // A[left] <= pivot <= A[right-1] and pivot is A[right]

        // now partition
        int i = left;
        int j = right-1;
        bool done = false;

        while ( ! done ) {
            while( A[++i] < pivot ) { }   // advance i
            while ( pivot < A[--j]) { }   // advance j

            //  A[j] <= pivot <= A[i]
            if ( i < j )
                swap( A[i], A[j] );
            else
                done = true;
        }
        // now j <= i and A[i] >= pivot and A[j] <= pivot
        // so we can swap the pivot with A[i]
        swap( A[i], A[right] );


        // Now the pivot is between the two sets and in A[i]
        // quicksort the left set:
        quicksort(A, left, i-1);

        // quicksort the right set:
        quicksort(A,i+1, right);
    }
    else {
        // A is too small to quicksort efficiently
        insertion_sort(A,left,right);
    }
}
```

Notes.

1. That `i` starts at `left+1` is intentional. The `median3` function rearranges the array so that `a[left]` $\leq$ `pivot`, so it is not necessary to examine it.

2. A symmetric statement is true of `j`: `median3` places the pivot in `a[right-1]` so `j` can start

by comparing `pivot` to `a[right-2]`.

3. Neither `i` nor `j` can exceed the array bounds because of the above reasoning. The ends act as sentinels for `i` and `j`.

```
/*
 * @post a[first] <= a[(first+last)/2] <= a[last-1] && a[last] contains
    the
 *       median of the original values in locations a[first], a[last], and
 *       a[(first+last)/2].
 */
C median3( vector<C> & a, int first, int last)
{
    int middle = (first + last)/2;
    C temp;
    if ( a[first] < a[middle] )
        if ( a[middle] < a[last] ) {
            // first < mid < last
            temp = a[last-1];
            a[last-1] = a[last];
            a[last] = a[middle]; // middle is pivot
            a[middle] = temp;
        }
        else if (a[last] < a[first] ) {
            // last < first < middle
            temp = a[last-1];
            a[last-1] = a[middle];
            a[middle] = temp;
            temp = a[last];
            a[last] = a[first];  // first is pivot
            a[first] = temp;
        }
        else {
            // first < last < middle
            temp = a[last-1];
            a[last-1] = a[middle];
            a[middle] = temp;
             // last is pivot
        }
    else if ( a[middle] < a[last] ) {
        if (  a[last] < a[first] ) {
            // middle < last < first
            temp = a[last-1];
            a[last-1] = a[first];
            a[first] = a[middle];
            a[middle] = temp;  // last is pivot
         }
        else {
            // middle < first < last
            temp = a[last-1];
            a[last-1] = a[last];
            a[last] = a[first];
            a[first] = a[middle];  // first is pivot
            a[middle] = temp;
```

```
        }
    }
    else {
            // last < middle < first
            temp = a[last-1];
            a[last-1] = a[first];
            a[first] = a[last];
            a[last] = a[middle];  // middle is pivot
            a[middle] = temp;
    }
    return a[last];
}
```

5.2 A Non-recursive Quicksort Algorithm

Although function call overhead is not what it used to be, the recursion in quicksort reduces its performance because of the time to create and maintain stack frames. A non-recursive quicksort can depend on a user-defined stack and avoid the runtime library's overhead. The version below refers to generic pop and push operations. The function `median3` is the one defined earlier.

Listing 3: Non-recursive Quicksort

```
typedef int C;

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

struct Pair
{
    Pair( int x, int y): l(x), r(y) {}
    int l,r;

};


void quicksort( vector< C > &a,  int last)
{
    stack<Pair> s;
    int left,right;
    int i, j;

    Pair  temp(0,last);
    s.push(temp);
    while ( ! s.empty() ) {
        temp = s.top();
        s.pop();
        left = temp.l;
        right = temp.r;
        while ( left < right ) { /* or left < (right - MINSIZE) */
```

```
            C pivot = median3(a, left, right);

            // partition step
            i = left;
            j = right-1;
            while ( true )  {
                while ( a[++i] < pivot ) { }
                while ( pivot < a[--j] ) { }
                if ( i < j )
                    swap( a[i], a[j] );
                else
                    break;
            }
            // move pivot into position in middle
            swap( a[i], a[right] );

            // pick larger of two regions to push on stack, and do the
            // smaller within the loop. This guarantees that the stack
               will
            // never have more than log_2 (n) pairs of parameters.
            if ( (right-i) > (i-left) ) {
                if ( right > i )
                    s.push(Pair(i+1,right));
                right = i-1;
            }
            else {
                if ( i > left )
                    s.push(Pair(left,i-1));
                left = i+1;
            }
        }
        // could do the insertion sort here if right-left < MINSIZE
    }
}
```

This version is designed so that the larger of the two choices of partition is always put on the stack, and the smaller is processed immediately. By doing this we prevent the stack from ever having more than log n frames. If we did not so this, the worst case stack would be $O(n)$ frames.

5.3 Analysis of Quicksort

A recurrence relation describes the total number of comparisons. Letting $T(n)$ denote the number of comparisons with an array of size n, and let i denote the number of elements in the lower part of the partition, S_1. $T(1)$ is some constant, say a. Then

$$T(n) = T(i) + T(n - i–1) + cn$$

Worst Case

The worst case is when the partition always creates one array of size 0 and the other of size $n - 1$. The recurrence in this case is

$$T(n) = T(n-1) + cn$$

for $n > 1$, which implies that

$$\begin{aligned} T(n) &= T(n-2) + cn + c(n-1) \\ &= T(n-3) + c(n+n-1+n-2) \\ &= \cdots \\ &= T(1) + c(n+n-1+n-2+\ldots+1) \\ &= a + \frac{cn(n+1)}{2} \in \Theta(n^2) \end{aligned}$$

Average Case

The average case analysis assumes that all sizes for S_1 are equally likely. We now let $T(n)$ represent the average running time for an array of size n. It follows that $T(n)$ is the average of the running times for all possible sizes of S_1, which are each of probability $1/n$ because S_1can be of size 0, 1, 2, $\ldots, n-1$ with equal probability (the pivot is excluded from S_1.) Therefore the average running time for an array of size n is the sum of the average running times of arrays of size j and those of size $n-j-1$ for all $j = 0 \ldots n-1$, divided by n, plus the cost of the partitioning step for the array of size n:

$$\begin{aligned} T(n) &= \frac{1}{n}\sum_{j=0}^{n-1}(T(j) + T(n-j-1)) + cn \\ &= \frac{2}{n}\sum_{j=0}^{n-1} T(j) + cn \qquad iff \\ nT(n) &= 2\sum_{j=0}^{n-1} T(j) + cn^2 \qquad iff \\ (n-1)T(n-1) &= 2\sum_{j=0}^{n-2} T(j) + c(n-1)^2 \end{aligned}$$

Now subtract the last two equalities and get

$$\begin{aligned} nT(n) - (n-1)T(n-1) &= 2T(n-1) + c(n^2 - (n-1)^2) \\ &= 2T(n-1) + 2cn - c \end{aligned}$$

Rearranging and dropping the constant c,

$$nT(n) = (n+1)T(n-1) + 2cn$$

Dividing by $n(n+1)$, we get a formula that can be telescoped by successive adding:

$$\begin{aligned}
\frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{2c}{n+1} \\
\frac{T(n-1)}{n} &= \frac{T(n-2)}{n-1} + \frac{2c}{n} \\
\frac{T(n-2)}{n-1} &= \frac{T(n-3)}{n-2} + \frac{2c}{n-1} \\
&\vdots \\
\frac{T(2)}{3} &= \frac{T(1)}{2} + \frac{2c}{3}
\end{aligned}$$

Adding these equations and then subtracting the common terms from both sides yields the following equation:

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c\sum_{i=3}^{n+1}\frac{1}{i} \approx \frac{T(1)}{2} + 2c\log(n+1) \in \Theta(\log n)$$

The last sum is actually $2c\log_e(n+1) + \gamma - 3/2$ where $\gamma \approx 0.577$ and is known as Euler's constant. Multiplying by $n+1$ on both sides gives the result

$$T(n) \in O(n\log n)$$

6 A Lower Bound for Sorting

The question is, can we ever find a sorting algorithm better than the $O(n\log n)$ algorithms we have seen so far, or is it theoretically impossible to achieve this?

The answer is that if our sorting algorithm sorts by making binary comparisons, then the worst case number of comparisons is $\Omega(n\log n)$. The proof is based on the following argument:

Any sorting algorithm that uses only comparisons requires $\lceil\log(n!)\rceil$ comparisons in the worst case and $\log(n!)$ on average. To prove this we use a decision tree.

6.1 Decision Trees

A ***decision tree*** is a tree representation of an algorithm that solves a problem by a sequence of successive decisions. In a decision tree, the nodes represent logical assertions and an edge from a node to a child node is labeled by a proposition. A binary decision tree is based on binary decisions. Because a binary decision tree is a binary tree, we can use reasoning about binary trees to arrive at a theorem about algorithms that sort using binary comparisons. Figure 2 illustrates a decision tree that represents the comparisons that would sort three distinct numbers. We need some preliminary lemmas.

Lemma 7. *A binary tree of height d has at most 2^d leaves.*

Proof. We can prove this by induction on the height of the tree. If $d = 0$, the tree consists of a root, and it has $1 = 2^0$ leaves. If $d > 0$, assume it is true for $d-1$. The tree must have a root and two subtrees of height at most $d-1$, which by assumption have at most 2^{d-1} leaves each. Since the root is not a leaf, the tree has at most $2 \cdot 2^{d-1} = 2^d$ leaves. □

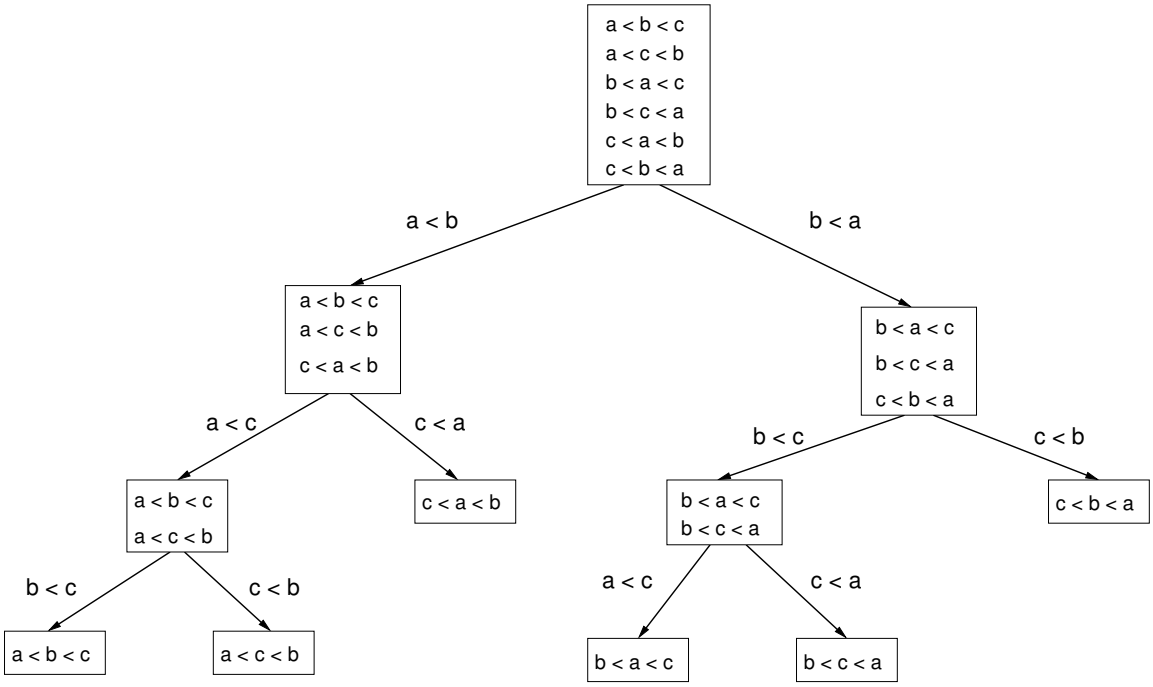

Figure 2: A decision tree that represents the decisions to sort three numbers.

Lemma 8. *If a binary tree has n leaves then it must have height at least $\lceil \log n \rceil$.*

Proof. Suppose a tree with n leaves has height less than $\lceil \log n \rceil$. There are two cases: n is not a power of 2 and n is a power of 2.

If n is not a power of 2, its height is at most $\lfloor \log n \rfloor$ because that is the largest integer less than $\lceil \log n \rceil$ in this case. But the preceding lemma states that if the tree had height $\lfloor \log n \rfloor$, it would have at most $2^{\lfloor \log n \rfloor} < n$ leaves which contradicts the premise that the tree has n leaves.

If n is a power of 2, then its height is at most $\log(n) - 1$ because that would be the largest integer less than $\lceil \log n \rceil$. From the preceding lemma, the tree would have at most $2^{\log n - 1} = n/2 < n$ leaves, which contradicts the fact that the tree has n leaves. Thus the tree must have height at least $\lceil \log n \rceil$. □

Lemma 9. *Any sorting algorithm that uses only binary comparisons between elements requires at least $\lceil \log n! \rceil$ comparisons for an array of size n.*

Proof. There are $n!$ leaves in a decision tree to sort n elements because there are $n!$ different possible outcomes. By Lemma 8, the height of this tree is at least $\lceil \log n! \rceil$. □

Theorem 10. *Any sorting algorithm that uses only comparisons between elements requires $\Omega(n \log n)$ comparisons.*

Proof. We show that $\log(n!)$ is $\Omega(n \log n)$.

$$\begin{aligned}
\log(n!) &= \log n + \log(n-1) + \log(n-2) + \cdots \log 2 + \log 1 \\
&\geq \log n + \log(n-1) + \log(n-2) + \cdots \log(n/2) \\
&\geq \frac{n}{2} \cdot \log \frac{n}{2} \\
&\geq \frac{n}{2} \cdot (\log n - 1) \\
&\geq \frac{n}{2} \cdot \log n - \frac{n}{2} \\
&= \Omega(n \log n)
\end{aligned}$$

This shows that we cannot do better than $\Omega(n \log n)$ if we are limited to binary comparisons. There are sorts that do not use them, such as radix sort and bucket sort. □