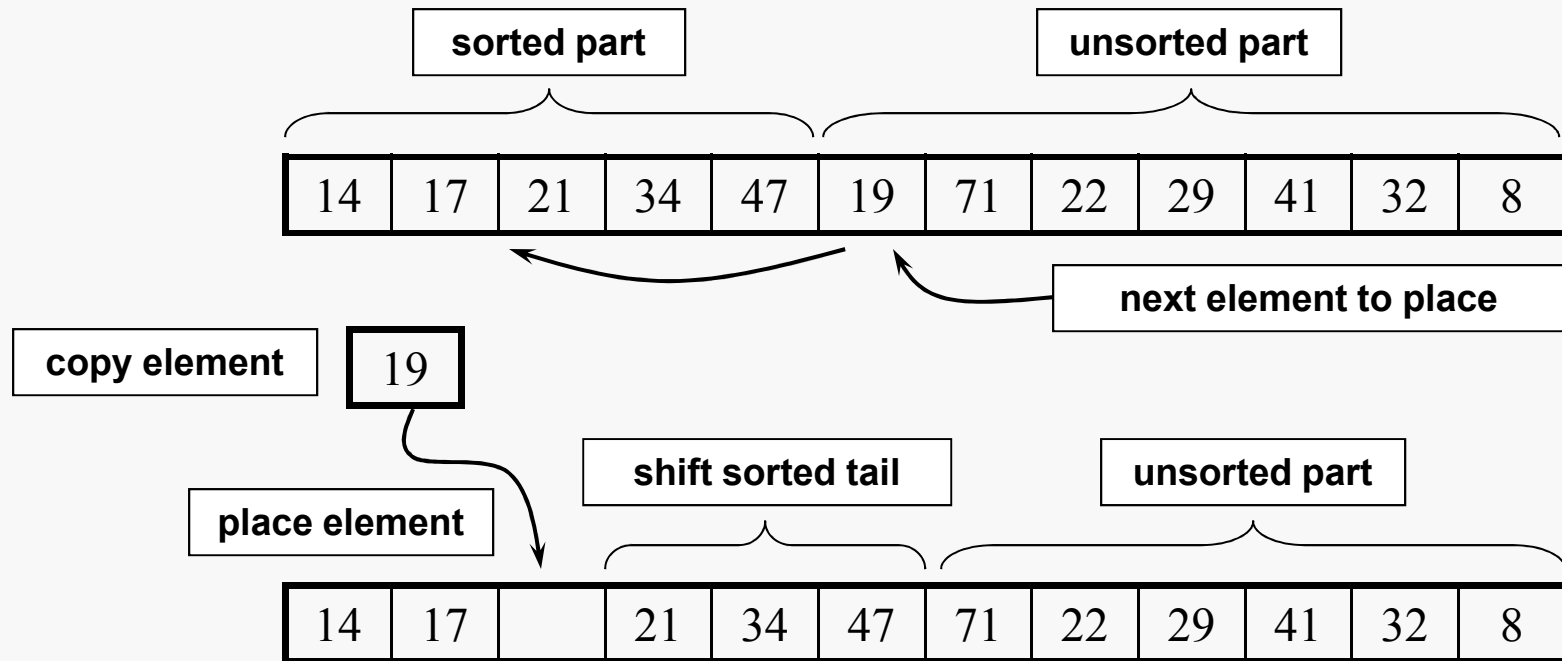Insertion Sort:

Insertion sort closely resembles the insertion function for a sorted list.

For a contiguous list, the primary costs are the comparisons to determine which part of the sorted portion must be shifted, and the assignments needed to accomplish that shifting of the sorted tail.

Assuming a list of N elements, Insertion Sort requires:

Average case: $N^2/4 + \Theta(N)$ comparisons and $N^2/4 + \Theta(N)$ assignments

Consider the element which is initially at the Kth position and suppose it winds up at position j, where j can be anything from 1 to K. A final position of j will require $K - j + 1$ comparisons. Therefore, on average, the number of comparisons to place the Kth element is:

$$\frac{1}{K}\sum_{j=1}^{K}(K - j + 1) = \frac{1}{K}\left[K^2 - \frac{K(K+1)}{2} + K\right] = \frac{K+1}{2}$$

The average total cost for insertion sort on a list of N elements is thus:

$$\sum_{k=2}^{N}\left(\frac{K+1}{2}\right) = \sum_{k=1}^{N-1}\left(\frac{K+2}{2}\right) = \frac{1}{2}\left[\frac{(N+1)(N+2)}{2}\right] = \frac{1}{4}N^2 + \frac{3}{4}N - 1$$

(…continued…)

The analysis for assignments is similar, differing only in that if the element in the Kth position winds up at position j, where j is between 1 to $K - 1$ inclusive, then the number of assignments is $K - j + 2$. The case where the element does not move is special, in that no assignments take place.

Proceeding as before, the average total number of assignments satisfies:

$$\sum_{k=2}^{N}\left(\frac{K+3}{2}-\frac{2}{K}\right) < \sum_{k=1}^{N-1}\left(\frac{K+4}{2}\right) = \frac{1}{4}N^2 + \frac{7}{4}N + 3$$

Worst case: $N^2/2 + \Theta(N)$ comparisons and $N^2/2 + \Theta(N)$ assignments

> **QTP: when will the worst case be achieved?**

Best case: $N - 1$ comparisons and no assignments (list is pre-sorted)

Before considering how to improve on Insertion Sort, consider the question:

How fast is it possible to sort?

Now, "fast" here must refer to algorithmic complexity, not time. We will consider the number of comparisons of elements a sorting algorithm must make in order to fully sort a list.

Note that this is an extremely broad issue since we seek an answer of the form: any sorting algorithm, no matter how it works, must, on average, perform at least $\Theta(f(N))$ comparisons when sorting a list of N elements.

Thus, we cannot simply consider any particular sorting algorithm…

A bit of combinatorics (the mathematics of counting)…

>   Given a collection of N distinct objects, the number of different ways to line them up in a row is N!.

Thus, a sorting algorithm must, in the worst case, determine the correct ordering among N! possible results.

If the algorithm compares two elements, the result of the comparison eliminates certain orderings as the final answer, and directs the "search" to the remaining, possible orderings.

We may represent the process of comparison sorting with a binary tree…

A comparison tree is a binary tree in which each internal node represents the comparison of two particular elements of a set, and the edges represent the two possible outcomes of that comparison.

For example, given the set A = {a, b, c} a sorting algorithm will begin by comparing two of the elements (it doesn't matter which two, so we'll choose arbitrarily):

$$a \leq b \leq c$$
$$a \leq c \leq b$$
$$c \leq a \leq b$$

T

a < b?

F

$$b \leq a \leq c$$
$$b \leq c \leq a$$
$$c \leq b \leq a$$
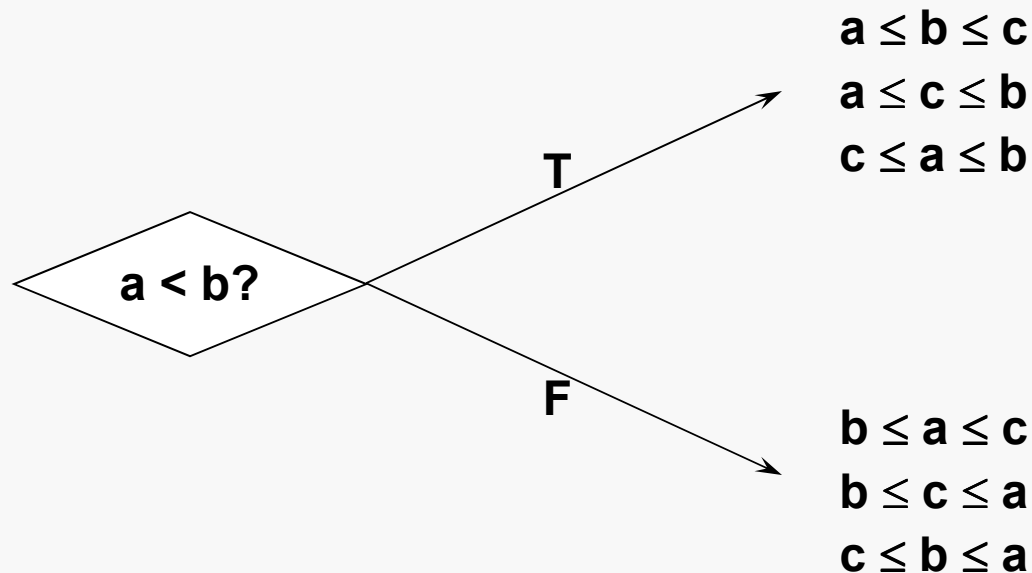
For example, given the set A = {a, b, c} a sorting algorithm will begin by comparing two of the elements (it doesn't matter which two, so we'll choose arbitrarily):
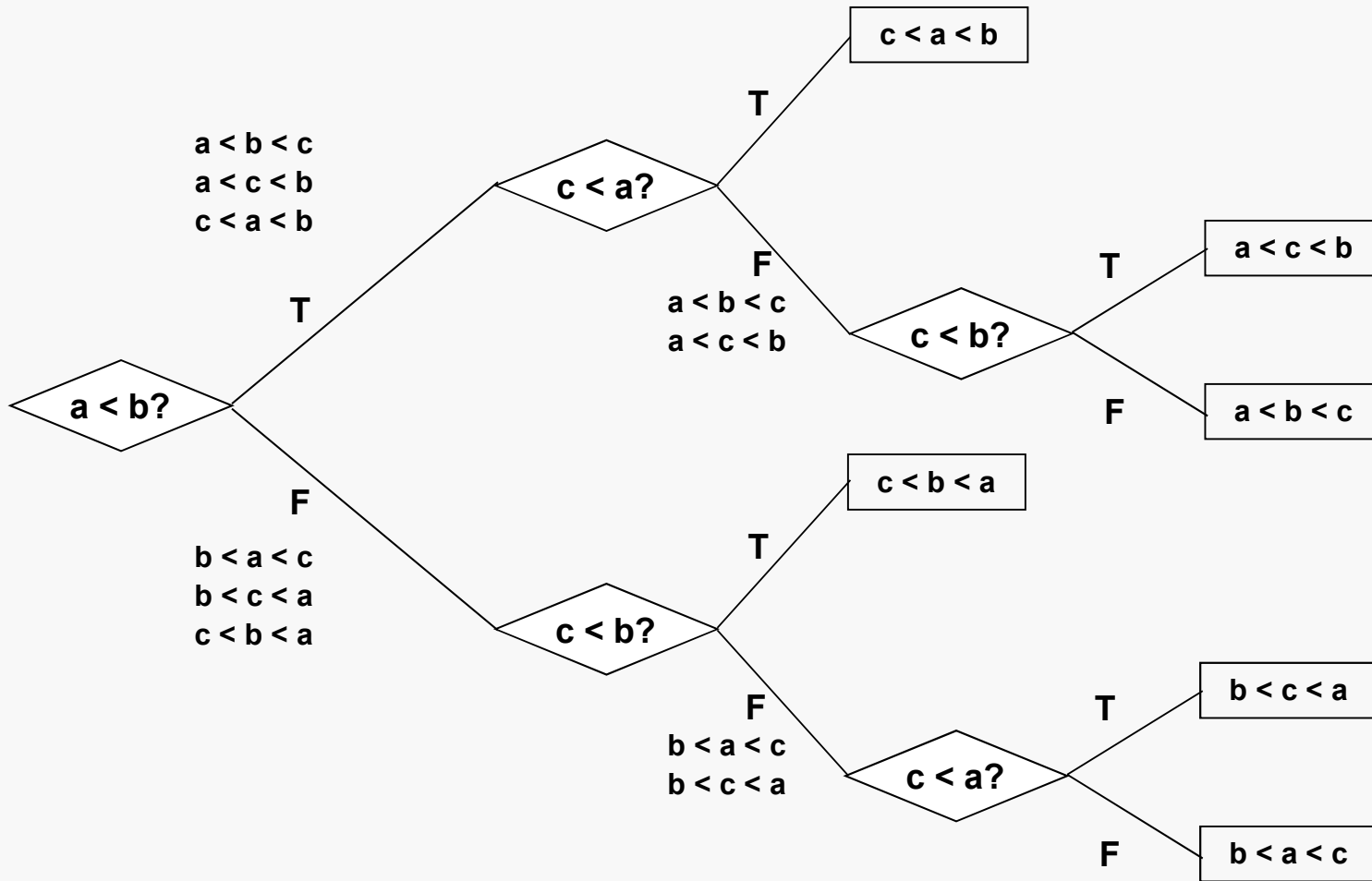
Theorem: Any algorithm that sorts a list of N entries by use of key comparisons must, in its worst case, perform at least $\lceil \log(N!) \rceil$ comparisons of keys, and, in the average case, it must perform at least $\log(N!)$ comparisons of keys.

proof: The operation of the algorithm can be represented by a comparison tree. That tree must have N! leaves, since there are N! possible answers.

We proved earlier that the number of levels in a binary tree with N! leaves must have at least $1 + \lceil \log(N!) \rceil$. If the answer corresponds to a leaf in the bottom level of the comparison tree, the algorithm must traverse at least $\lceil \log(N!) \rceil$ internal nodes, and so must do at least $\lceil \log(N!) \rceil$ comparisons.

That proves the worst case lower bound.

The proof of the average case lower bound is tediously similar.

QED

The average case lower bound, log(N!) comparisons, is somewhat ugly. We can simplify it by applying a result known as Stirling's Formula:

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e} \right)^n \left[1 + \frac{1}{12n} + O\left(\frac{1}{n^2} \right) \right]$$

or, converting to logarithmic form:

$$\ln(n!) \approx \left(n + \frac{1}{2} \right) \ln n - n + \frac{1}{2} \ln(2\pi) + \frac{1}{12n} + O\left(\frac{1}{n^2} \right)$$

Using Stirling's Formula, and changing bases, we have that:

$$\log(n!) \approx \left(n + \frac{1}{2}\right)\log n - \log(e)n + \frac{1}{2}\log(2\pi) + \frac{\log(e)}{12n}$$

$$\approx \left(n + \frac{1}{2}\right)\left(\log n - \frac{3}{2}\right) + 2$$

$$= n\log n - \frac{3}{2}n + \frac{1}{2}\log n + \frac{5}{4}$$

So, log(n!) is Θ(n log n).

For most practical purposes, only the first term matters here, so you will often see the assertion that the lower bound for comparisons in sorting is n log n.

The process just shown is known as Shell Sort (after its originator: Donald Shell, 1959), and may be summed up as follows:

- partition the list into discontiguous sublists whose elements are some step size, h, apart.

- sort each sublist by applying Insertion Sort (or …)

- decrease the value of h and repeat the first two steps, stopping after a pass in which h is 1.

The end result will be a sorted list because the final pass (with h being 1) is simply an application of Insertion Sort to the entire list.

What is an optimal sequence of values for the step size h? No one knows…

What is the performance analysis for Shell Sort? No one knows… in general.

Knuth has shown that if two particular step sizes are used, then Shell Sort takes $O(N^{3/2})$ time (versus $O(N^2)$ for simple Insertion Sort).

$$\left(\frac{16N}{\pi}\right)^{1/3} \text{ and } 1$$

Good mixing of the sublists can be provided by choosing the step sizes by the rule:

$$\begin{cases} h_1 = 1 \\ h_{i+1} = 3h_i + 1 \text{ for i} \geq 1 \end{cases}$$

Empirically, for large values of N, Shell Sort appears to be about $O(N^{5/4})$ if the step size scheme given above is used.

Shell Sort represents a "divide-and-conquer" approach to the problem.

That is, we break a large problem into smaller parts (which are presumably more manageable), handle each part, and then somehow recombine the separate results to achieve a final solution.

In Shell Sort, the recombination is achieved by decreasing the step size to 1, and physically keeping the sublists within the original list structure. Note that Shell Sort is better suited to a contiguous list than a linked list.

We will now consider a somewhat similar algorithm that retains the divide and conquer strategy, but which is better suited to linked lists.

In Merge Sort, we chop the list into two (or more) sublists which are as nearly equal in size as we can achieve, sort each sublist separately, and then carefully merge the resulting sorted sublists to achieve a final sorting of the original list.
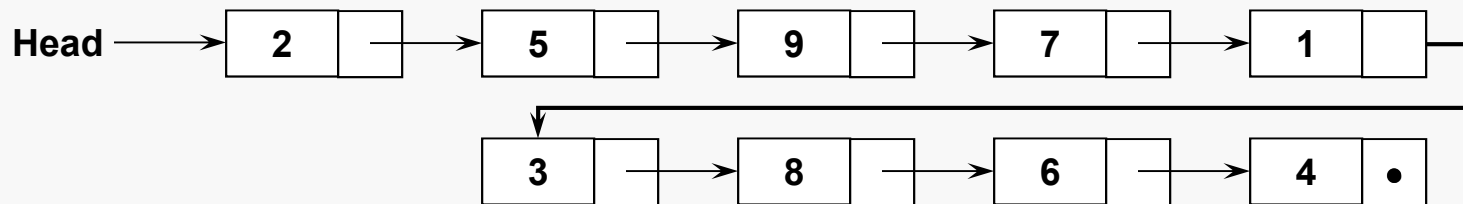
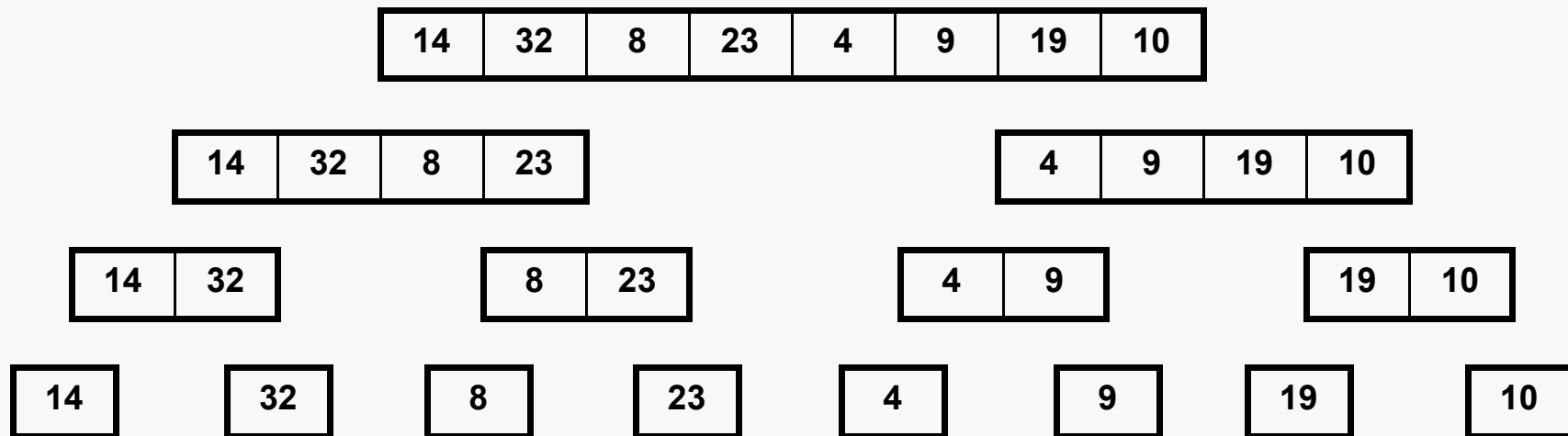Merging two sorted lists into a single sorted list is relatively trivial:
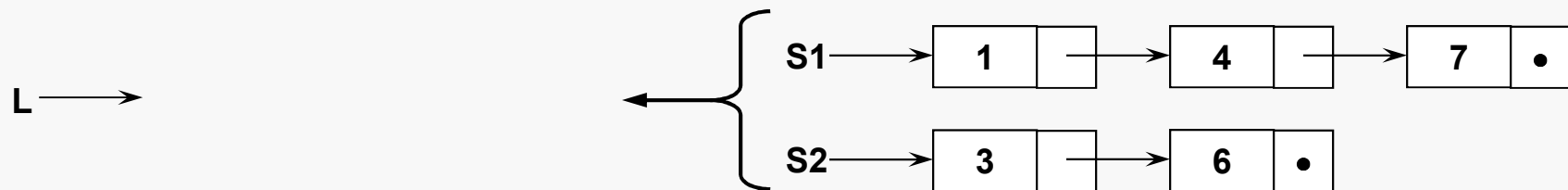
At a high level, the implementation involves two activities, partitioning and merging, each represented by a corresponding function.

The number of partitioning steps equals the number of merge steps, partitioning taking place during the recursive descent and merging during the recursive ascent as the calls back out.

Consider partitioning for a list of 8 elements:

| 14 | 32 | 8 | 23 | 4 | 9 | 19 | 10 |

| 14 | 32 | 8 | 23 | | 4 | 9 | 19 | 10 |

| 14 | 32 | | 8 | 23 | | 4 | 9 | | 19 | 10 |

| 14 | | 32 | | 8 | | 23 | | 4 | | 9 | | 19 | | 10 |

The maximum number of levels is $\lceil \log N \rceil$.

All the element comparisons take place during the merge phase. Logically, we may consider this as if the algorithm re-merged each level before proceeding to the next:

| 14 | | 32 | | 8 | | 23 | | 4 | | 9 | | 19 | | 10 |

| 14 | 32 | | 8 | 23 | | 4 | 9 | | 10 | 19 |

| 8 | 14 | 23 | 32 | | 4 | 9 | 10 | 19 |

| 4 | 8 | 9 | 10 | 14 | 19 | 23 | 32 |
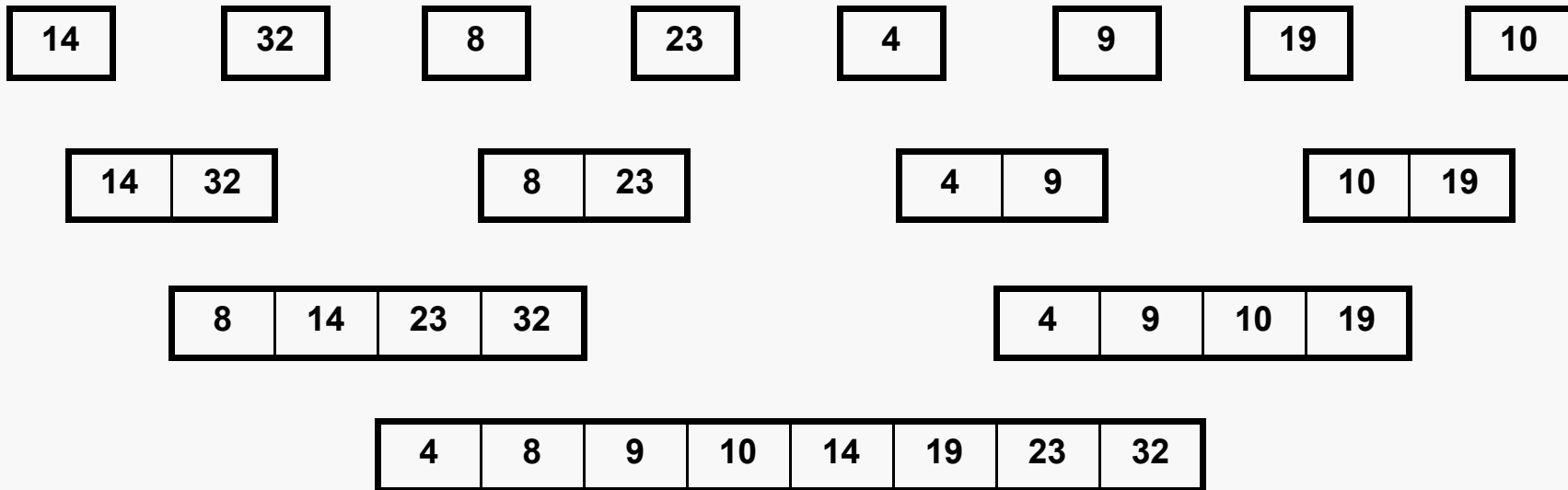
So merging the sublists involves $\lceil \log N \rceil$ passes. On each pass, each list element is used in (at most) one comparison, so the number of element comparisons per pass is N. Hence, the number of comparisons for Merge Sort is $\Theta(N \lceil \log N \rceil)$.

Merge Sort comes very close to the theoretical optimum number of comparisons. A closer analysis shows that for a list of N elements, the average number of element comparisons using Merge Sort is actually:

$$\Theta(N \log N) - 1.1583N + 1$$

Recall that the theoretical minimum is:

$$N \log N - 1.44N + \Theta(1)$$

For a linked list, Merge Sort is the sorting algorithm of choice, providing nearly optimal comparisons and requiring NO element assignments (although there is a considerable amount of pointer manipulation), and requiring NO significant additional storage.

For a contiguous list, Merge Sort would require either using $\Theta(N)$ additional storage for the sublists or using a considerably complex algorithm to achieve the merge with a small amount of additional storage.

A list can be sorted by first building it into a heap, and then iteratively deleting the root node from the heap until the heap is empty. If the deleted roots are stored in reverse order in an array they will be sorted in ascending order (if a max heap is used).

```cpp
template <typename T>
void HeapSort(T* List, unsigned int Size) {

    HeapT<T> toSort(List, Size);
    toSort.BuildHeap();

    unsigned int Idx = Size - 1;
    while ( !toSort.isEmpty() ) {
        List[Idx] = toSort.RemoveRoot();
        Idx--;
    }
}
```

Recalling the earlier analysis of building a heap, level k of a full and complete binary tree will contain 2^k nodes, and that those nodes are k levels below the root level.

So, when the root is deleted the maximum number of levels the swapped node can sift down is the number of the level from which that node was swapped.

Thus, in the worst case, for deleting all the roots…

$$\text{Comparisons} = 2\sum_{k=1}^{d-1} k2^k = 4\sum_{k=1}^{d-1} k2^{k-1} = 4\left[(d-2)2^{d-1}+1\right]$$

$$= 2N\lceil \log N \rceil + 2\lceil \log N \rceil - 4N$$

As usual, with Heap Sort, this would entail half as many element swaps.

Adding in the cost of building the heap from our earlier analysis,

$$\text{Total Comparisons} = \left(2N - 2\lceil \log N \rceil\right) + \left(2N\lceil \log N \rceil + 2\lceil \log N \rceil - 4N\right)$$
$$= 2N\lceil \log N \rceil - 2N$$

and...

$$\text{Total Swaps} = N\lceil \log N \rceil - N$$

So, in the worst case, Heap Sort is $\Theta(N \log N)$ in both swaps and comparisons.

The average case analysis is apparently so difficult no one has managed it rigorously. However, empirical analysis indicates that Heap Sort does not do much better, on average, than in the worst case.

QuickSort is conceptually similar to MergeSort in that it is a divide-and-conquer approach, involving successive partitioning of the list to be sorted.

The differences lie in the manner in which the partitioning is done, and that the sublists are maintained in proper relative order so no merging is required.

QuickSort is a naturally recursive algorithm, each step of which involves:

- pick a pivot value for the current sublist

- partition the sublist into two sublists, the left containing only values less than the pivot value and the right containing only values greater than or equal to the pivot value

- recursively process each of the resulting sublists

As they say, the devil is in the details…

As with Merge Sort, we can view QuickSort as consisting of a sequence of steps, each involving pivot selection and then partitioning of each nontrivial sublist.

What is the total partitioning cost at each step?

As implemented here, each partitioning step isolates a copy of the pivot value and that value is not subsequently moved, so the number of elements that are subject to partitioning decreases at each step. However, it is relatively clear that the total cost of partitioning at any step is no worse than $\Theta(N)$.

But, how many steps must be performed?

Ah, there's the rub… that depends upon how well the pivot values work.

For simplicity in analysis, we will make the following assumptions:

- the key values to be partitioned are $1\ldots N$

- $C(N)$ is the number of comparisons made by QuickSort when applied to a list of length N

- $S(N)$ is the number of swaps made by QuickSort when applied to a list of length N

If partitioning produces one sublist of length r and one of $N - r - 1$, then:

$$C(N) = N - 1 + C(r) + C(N - r - 1)$$

since the initial partition will require $N - 1$ element comparisons.

The formula derived here is known as a recurrence relation.

For a given value of r, it is possible to solve for $C(N)\ldots$

Suppose $r = 0$; i. e., that QuickSort produces one empty sublist and merely splits off the pivot value from the remaining $N - 1$ elements. Then we have:

$C(1) = 0$

$C(2) = 1 + C(1)$ $= 1$

$C(3) = 2 + C(2)$ $= 2 + 1$

$C(4) = 3 + C(3)$ $= 3 + 2 + 1$

. . .

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

                                       $= 0.5N^2 - 0.5N$

This is the worst case, and is as bad as the worst case for selection sort.

Similar analysis shows that the number of swaps in this case is:

$$S(N) = 0.5N^2 + 1.5N - 1$$

which is $\Theta(1.5N^2)$ element assignments, even worse than insertion sort.

For the average case, we will consider all possible results of the partitioning phase and compute the average of those. We assume that all possible orderings of the list elements are equally likely to be the correct sorted order, and let p denote the pivot value chosen.

Thus, after partitioning, recalling we assume the key values are 1…N, the values 1, … p – 1 are to the left of p, and the values p + 1, … N are to the right of p.

Let *S(N)* be the average number of swaps made by QuickSort on a list of length N, and *S(N, p)* be the average number of swaps if the value p is chosen as the initial pivot.

The partition implementation given here will perform p – 1 swaps within the loop, and two more outside the loop. Therefore…

$$S(N, p) = p + 1 + S(p - 1) + S(N - p)$$

Now there are N possible choices for p (1…N), and so if we sum this expression from p = 1 to p = N and divide by N we have:

$$S(N) = \frac{N}{2} + \frac{3}{2} + \frac{2}{N}\left[S(0) + S(1) + \cdots + S(N-1)\right] \qquad \text{(A)}$$

That's another recurrence relation… we may solve this by playing a clever trick; first note that if QuickSort were applied to a list of N – 1 elements, we'd have:

$$S(N - 1) = \frac{N-1}{2} + \frac{3}{2} + \frac{2}{N-1}\left[S(0) + S(1) + \cdots + S(N-2)\right] \qquad \text{(B)}$$

If we multiply both sides of equation (A) by N, and multiply both sides of equation (B) by N − 1, and then subtract, we can obtain:

$$NS(N) - (N-1)S(N-1) = N + 1 + 2S(N-1) \qquad \text{(C)}$$

Rearranging terms, we get:

$$\frac{S(N)}{N+1} = \frac{S(N-1)}{N} + \frac{1}{N} \qquad \text{(D)}$$

A closed-form solution for this recurrence relation can be guessed by writing down the first few terms of the sequence. Doing so, we obtain…

$$\frac{S(N)}{N+1} = \frac{S(2)}{3} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{N} \tag{E}$$

Now, it can be shown that for all $N \geq 1$,

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{N} = \ln N + O(1)$$

Applying that fact to equation (E), we can show that: $\dfrac{S(N)}{N+1} = \ln N + O(1)$

Algebra and a log base conversion yield:

$$S(N) = N \ln N + O(N) \approx 0.69 N \log N + O(N)$$

A similar analysis for comparisons starts with the relationship:

$$C(N, p) = N - 1 + C(p - 1) + C(N - p)$$

This eventually yields:

$$C(N) = 2N \ln N + O(N) \approx 1.39 N \log N + O(N)$$

So, on average, the given implementation of QuickSort is $\Theta(N \log N)$ in both swaps and comparisons.

QuickSort can be improved by

- switching to a nonrecursive sort algorithm on sublists that are relatively short. The common rule of thumb is to switch to insertion sort if the sublist contains fewer than 10 elements.

- eliminating the recursion altogether. However, the resulting implementation is considerably more difficult to understand (and therefore to verify) than the recursive version.

- making a more careful choice of the pivot value than was done in the given implementation.

- improving the partition algorithm to reduce the number of swaps. It is possible to reduce the number of swaps during partitioning to about 1/3 of those used by the given implementation.

The earlier sorting algorithms are clearly only suitable for short lists, where the total time will be negligible anyway.

While MergeSort can be adapted for contiguous lists, and QuickSort for linked lists, each adaptation induces considerable loss of efficiency and/or clarity; therefore, MergeSort and QuickSort cannot really be considered competitors.

HeapSort is inferior to the average case of QuickSort, but not by much. In the worst cases, QuickSort is among the worst sorting algorithms discussed here. From that perspective, HeapSort is attractive as a hedge against the possible worst behavior of QuickSort.

However, in practice QuickSort rarely degenerates to its worst case and is almost always $\Theta(N \log N)$.

Theoretical analysis tells us that no sorting algorithm can require fewer comparisons than $\Theta(N \log N)$, in its average case, right?

Well, actually no… the derivation of that result assumes that the algorithm works by comparing key values.

It is possible to devise sorting algorithms that do not compare the elements of the list to each other. In that case, the earlier result does not apply.

However, such algorithms require special assumptions regarding the type and/or range of the key values to be sorted.

Assume we need to sort a list of integers in the range 0-99:

| 34 | 16 | 83 | 76 | 40 | 72 | 38 | 80 | 89 | 87 |

Given an array of 10 linked lists (bins) for storage, make a pass through the list of integers, and place each into the bin that matches its 1's digit.

Then, make a second pass, taking each bin in order, and append each integer to the bin that matches its 10's digit.

```
Bin
 0:    40    80
 1:
 2:    72
 3:    83
 4:    34
 5:
 6:    16    76
 7:    87
 8:    38
 9:    89
```

```
Bin
 0:
 1:    16
 2:
 3:    34    38
 4:    40
 5:
 6:
 7:    72    76
 8:    80    83    87    89
 9:
```

Now if you just read the bins, in order, the elements will appear in ascending order. Each pass takes $\Theta(N)$ work, and the number of passes is just the number of digits in the largest integer in the original list.

That beats N log(N), by a considerable margin, but only in a somewhat special case.

```
Bin
 0:
 1:     16
 2:
 3:     34     38
 4:     40
 5:
 6:
 7:     72     76
 8:     80     83     87     89
 9:
```

Note the number of <u>bins</u> is determined by the number of possible values in each position, while the number of <u>passes</u> is determined by the maximum length of the data elements.