described $(1,1)$. The second cheapest fare would be made from the head of one list and the second element of another, and hence would be either $(1,2)$ or $(2,1)$. Then it gets more complicated. The third cheapest could either be the unused pair above or $(1,3)$ or $(3,1)$. Indeed it would have been $(3,1)$ in the example above if the third fare of X had been \$120.

"Tell me," I asked. "Do we have time to sort the two respective lists of fares in increasing order?"

"Don't have to." the leader replied. "They come out in sorted order from the database."

Good news. That meant there was a natural order to the pair values. We never need to evaluate the pairs $(i+1,j)$ or $(i,j+1)$ before (i,j), because they clearly define more expensive fares.

"Got it!," I said. "We will keep track of index pairs in a priority queue, with the sum of the fare costs as the key for the pair. Initially we put only pair $(1,1)$ on the queue. If it proves it is not feasible, we put its two successors on—namely $(1,2)$ and $(2,1)$. In general, we enqueue pairs $(i+1,j)$ and $(i,j+1)$ after evaluating/rejecting pair (i,j). We will get through all the pairs in the right order if we do so."

The gang caught on quickly. "Sure. But what about duplicates? We will construct pair (x,y) two different ways, both when expanding $(x-1,y)$ and $(x,y-1)$."

"You are right. We need an extra data structure to guard against duplicates. The simplest might be a hash table to tell us whether a given pair exists in the priority queue before we insert a duplicate. In fact, we will never have more than n active pairs in our data structure, since there can only be one pair for each distinct value of the first coordinate."

And so it went. Our approach naturally generalizes to itineraries with more than two legs, (a complexity which grows with the number of legs). The best-first evaluation inherent in our priority queue enabled the system to stop as soon as it found the provably cheapest fare. This proved to be fast enough to provide interactive response to the user. That said, I haven't noticed my travel tickets getting any cheaper.

4.5 Mergesort: Sorting by Divide-and-Conquer

Recursive algorithms reduce large problems into smaller ones. A recursive approach to sorting involves partitioning the elements into two groups, sorting each of the smaller problems recursively, and then interleaving the two sorted lists to totally order the elements. This algorithm is called *mergesort*, recognizing the importance of the interleaving operation:

```
Mergesort(A[1,n])
    Merge( MergeSort(A[1, ⌊n/2⌋]), MergeSort(A[⌊n/2⌋ + 1, n]) )
```

The basis case of the recursion occurs when the subarray to be sorted consists of a single element, so no rearrangement is possible. A trace of the execution of

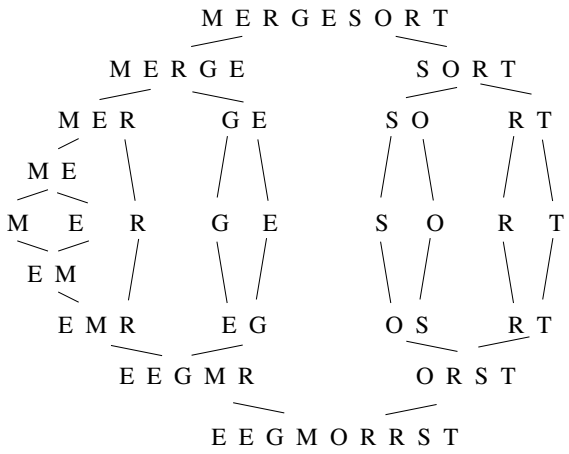

Figure 4.4: Animation of mergesort in action

mergesort is given in Figure 4.4. Picture the action as it happens during an in-order traversal of the top tree, with the array-state transformations reported in the bottom, reflected tree.

The efficiency of mergesort depends upon how efficiently we combine the two sorted halves into a single sorted list. We could concatenate them into one list and call heapsort or some other sorting algorithm to do it, but that would just destroy all the work spent sorting our component lists.

Instead we can *merge* the two lists together. Observe that the smallest overall item in two lists sorted in increasing order (as above) must sit at the top of one of the two lists. This smallest element can be removed, leaving two sorted lists behind—one slightly shorter than before. The second smallest item overall must be atop one of these lists. Repeating this operation until both lists are empty merges two sorted lists (with a total of n elements between them) into one, using at most $n-1$ comparisons or $O(n)$ total work.

What is the total running time of mergesort? It helps to think about how much work is done at each level of the execution tree. If we assume for simplicity that n is a power of two, the kth level consists of all the 2^k calls to `mergesort` processing subranges of $n/2^k$ elements.

The work done on the $(k=0)$th level involves merging two sorted lists, each of size $n/2$, for a total of at most $n-1$ comparisons. The work done on the $(k=1)$th level involves merging two pairs of sorted lists, each of size $n/4$, for a total of at most $n-2$ comparisons. In general, the work done on the kth level involves merging 2^k pairs sorted list, each of size $n/2^{k+1}$, for a total of at most $n-2^k$ comparisons. *Linear work is done merging all the elements on each level.* Each of the n elements

appears in exactly one subproblem on each level. The most expensive case (in terms of comparsions) is actually the top level.

The number of elements in a subproblem gets halved at each level. Thus the number of times we can halve n until we get to 1 is $\lceil \lg_2 n \rceil$. Because the recursion goes $\lg n$ levels deep, and a linear amount of work is done per level, mergesort takes $O(n \log n)$ time in the worst case.

Mergesort is a great algorithm for sorting linked lists, because it does not rely on random access to elements as does heapsort or quicksort. Its primary disadvantage is the need for an auxilliary buffer when sorting arrays. It is easy to merge two sorted linked lists without using any extra space, by just rearranging the pointers. However, to merge two sorted arrays (or portions of an array), we need use a third array to store the result of the merge to avoid stepping on the component arrays. Consider merging $\{4, 5, 6\}$ with $\{1, 2, 3\}$, packed from left to right in a single array. Without a buffer, we would overwrite the elements of the top half during merging and lose them.

Mergesort is a classic divide-and-conquer algorithm. We are ahead of the game whenever we can break one large problem into two smaller problems, because the smaller problems are easier to solve. The trick is taking advantage of the two partial solutions to construct a solution of the full problem, as we did with the merge operation.

Implementation

The divide-and-conquer `mergesort` routine follows naturally from the pseudocode:

```
mergesort(item_type s[], int low, int high)
{
        int i;                  /* counter */
        int middle;             /* index of middle element */

        if (low < high) {
                middle = (low+high)/2;
                mergesort(s,low,middle);
                mergesort(s,middle+1,high);
                merge(s, low, middle, high);
        }
}
```

More challenging turns out to be the details of how the merging is done. The problem is that we must put our merged array somewhere. To avoid losing an element by overwriting it in the course of the merge, we first copy each subarray to a separate queue and merge these elements back into the array. In particular:

```
merge(item_type s[], int low, int middle, int high)
{
  int i;                   /* counter */
  queue buffer1, buffer2; /* buffers to hold elements for merging */

  init_queue(&buffer1);
  init_queue(&buffer2);

  for (i=low; i<=middle; i++) enqueue(&buffer1,s[i]);
  for (i=middle+1; i<=high; i++) enqueue(&buffer2,s[i]);

  i = low;
  while (!(empty_queue(&buffer1) || empty_queue(&buffer2))) {
          if (headq(&buffer1) <= headq(&buffer2))
                  s[i++] = dequeue(&buffer1);
          else
                  s[i++] = dequeue(&buffer2);
  }

  while (!empty_queue(&buffer1)) s[i++] = dequeue(&buffer1);
  while (!empty_queue(&buffer2)) s[i++] = dequeue(&buffer2);
}
```

4.6 Quicksort: Sorting by Randomization

Suppose we select a random item p from the n items we seek to sort. *Quicksort* (shown in action in Figure 4.5) separates the $n-1$ other items into two piles: a low pile containing all the elements that appear before p in sorted order and a high pile containing all the elements that appear after p in sorted order. Low and high denote the array positions we place the respective piles, leaving a single slot between them for p.

Such partitioning buys us two things. First, the pivot element p ends up in the exact array position it will reside in the the final sorted order. Second, after partitioning no element flops to the other side in the final sorted order. *Thus we can now sort the elements to the left and the right of the pivot independently!* This gives us a recursive sorting algorithm, since we can use the partitioning approach to sort each subproblem. The algorithm must be correct since each element ultimately ends up in the proper position:

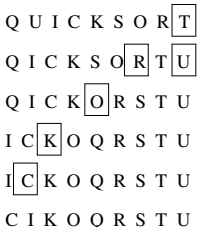

Figure 4.5: Animation of quicksort in action

```
quicksort(item_type s[], int l, int h)
{
        int p;                  /* index of partition */

        if ((h-l)>0) {
                p = partition(s,l,h);
                quicksort(s,l,p-1);
                quicksort(s,p+1,h);
        }
}
```

We can partition the array in one linear scan for a particular pivot element by maintaining three sections of the array: less than the pivot (to the left of `firsthigh`), greater than or equal to the pivot (between `firsthigh` and `i`), and unexplored (to the right of `i`), as implemented below:

```
int partition(item_type s[], int l, int h)
{
        int i;                  /* counter */
        int p;                  /* pivot element index */
        int firsthigh;          /* divider position for pivot element */

        p = h;
        firsthigh = l;
        for (i=l; i<h; i++)
                if (s[i] < s[p]) {
                        swap(&s[i],&s[firsthigh]);
                        firsthigh ++;
                }
        swap(&s[p],&s[firsthigh]);

        return(firsthigh);
}
```

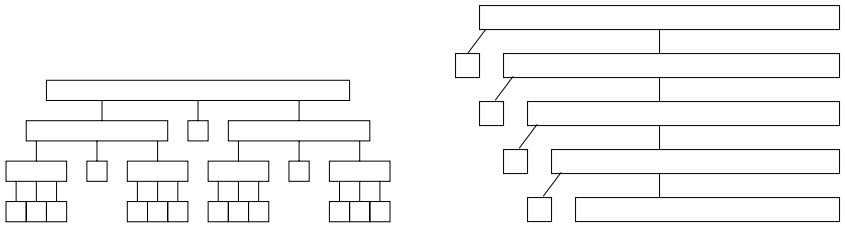

Figure 4.6: The best-case (l) and worst-case (r) recursion trees for quicksort

Since the partitioning step consists of at most n swaps, it takes linear time in the number of keys. But how long does the entire quicksort take? As with mergesort, quicksort builds a recursion tree of nested subranges of the n-element array. As with mergesort, quicksort spends linear time processing (now `partition`ing instead of `merge`ing) the elements in each subarray on each level. As with mergesort, quicksort runs in $O(n \cdot h)$ time, where h is the height of the recursion tree.

The difficulty is that the height of the tree depends upon where the pivot element ends up in each partition. If we get very lucky and *happen* to repeatedly pick the median element as our pivot, the subproblems are always half the size of the previous level. The height represents the number of times we can halve n until we get down to 1, or at most $\lceil \lg_2 n \rceil$. This happy situation is shown in Figure 4.6(l), and corresponds to the best case of quicksort.

Now suppose we consistently get unlucky, and our pivot element always splits the array as unequally as possible. This implies that the pivot element is always the biggest or smallest element in the sub-array. After this pivot settles into its position, we are left with one subproblem of size $n - 1$. We spent linear work and reduced the size of our problem by one measly element, as shown in Figure 4.6(r). It takes a tree of height $n - 1$ to chop our array down to one element per level, for a worst case time of $\Theta(n^2)$.

Thus, the worst case for quicksort is worse than heapsort or mergesort. To justify its name, quicksort had better be good in the average case. Understanding why requires some intuition about random sampling.

4.6.1 Intuition: The Expected Case for Quicksort

The expected performance of quicksort depends upon the height of the partition tree constructed by random pivot elements at each step. Mergesort ran in $O(n \log n)$ time because we split the keys into two equal halves, sorted them recursively, and then merged the halves in linear time. Thus, whenever our pivot element is near the center of the sorted array (i.e. , the pivot is close to the median element), we get a good split and realize the same performance as mergesort.

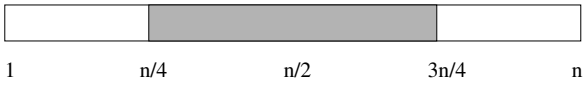

Figure 4.7: Half the time, the pivot is close to the median element

I will give an intuitive explanation of why quicksort is $O(n \log n)$ in the average case. How likely is it that a randomly selected pivot is a good one? The best possible selection for the pivot would be the median key, because exactly half of elements would end up left, and half the elements right, of the pivot. Unfortunately, we only have a probability of $1/n$ of randomly selecting the median as pivot, which is quite small.

Suppose a key is a *good enough* pivot if it lies is in the center half of the sorted space of keys—i.e. , those ranked from $n/4$ to $3n/4$ in the space of all keys to be sorted. Such *good enough* pivot elements are quite plentiful, since half the elements lie closer to the middle than one of the two ends (see Figure 4.7). Thus, on each selection we will pick a *good enough* pivot with probability of $1/2$.

Can you flip a coin so it comes up tails each time? Not without cheating. If you flip a fair coin n times, it will come out heads about half the time. Let heads denote the chance of picking a *good enough* pivot.

The worst possible *good enough* pivot leaves the bigger half of the space partition with $3n/4$ items. What is the height h_g of a quicksort partition tree constructed repeatedly from the worst-possible *good enough* pivot? The deepest path through this tree passes through partitions of size $n, (3/4)n, (3/4)^2 n, \ldots$, down to 1. How many times can we multiply n by 3/4 until it gets down to 1?

$$(3/4)^{h_g} n = 1 \Rightarrow n = (4/3)^{h_g}$$

so $h_g = \log_{4/3} n$.

But only half of all randomly selected pivots will be *good enough*. The rest we classify as *bad*. The worst of these bad pivots will do essentially nothing to reduce the partition size along the deepest path. The deepest path from the root through a typical randomly-constructed quicksort partition tree will pass through roughly equal numbers of good-enough and bad pivots. Since the expected number of good splits and bad splits is the same, the bad splits can only double the height of the tree, so $h \approx 2h_g = 2 \log_{4/3} n$, which is clearly $\Theta(\log n)$.

On average, random quicksort partition trees (and by analogy, binary search trees under random insertion) are very good. More careful analysis shows the average height after n insertions is approximately $2 \ln n$. Since $2 \ln n \approx 1.386 \lg_2 n$, this is only 39% taller than a perfectly balanced binary tree. Since quicksort does $O(n)$ work partitioning on each level, the average time is $O(n \log n)$. If we are *extremely* unlucky and our randomly selected elements always are among the largest or smallest element in the array, quicksort turns into selection sort and runs in $O(n^2)$. However, the odds against this are vanishingly small.