11

Sorting

Problems that involve putting items into order, or sorting, arise frequently in practice. Sometimes we sort merely for our convenience, as when we alphabetize a list before we print it. At other times we sort to set up a more efficient data structure, as when we sort an array so we can use binary search on it. And many times we sort to facilitate further processing; for instance, it is much easier to find duplicate items in a list that is sorted than one that is out of order.

Because the need to sort is so common, and because the problem is so simple to state, sorting has been studied extensively. This chapter presents several kinds of sorting problem, and several kinds of solution. An important lesson is that there is no single best way to solve all sorting problems.

11.1
SETTINGS FOR SORTING

People have been sorting for centuries, whether to keep a list in alphabetical order or to arrange their hands in card games. To understand why this routine task is challenging for a computer, imagine that we are playing a game in which we must sort using the limited capabilities available to a computer.

To begin the game, we enter a room in which a line of cards lies in a single row on a table. Each card has an encoded key, perhaps on a magnetic stripe; in a departure from the search problem of Chapter 7, we shall not assume that the keys are distinct. A comparison
**Algorithm 1.1**

\[
\frac{c}{(1-u)^\mu} = \frac{1 - \frac{1}{u}}{1 - \frac{1}{u}} = \frac{1 - \frac{1}{u}}{1 - \frac{1}{u}}
\]

**Section 1.1.2**

In Section 1.1.2, we will see three algorithms to solve the generalized sorting problem. To orientalize the players, we will focus on the number of binary comparisons (i.e., comparisons of the form 'greater than' or 'less than').

The problem is to sort an array of numbers. The algorithm works as follows:

1. **Insertion Sort**
   - Simply iterate through the array, comparing each element with the elements before it, and moving it to the correct position.

2. **Simple Extensions**
   - Modify Insertion Sort to handle arrays with more elements.

3. **Efficient Sorting**
   - Use a more efficient sorting algorithm like Quicksort or Mergesort.

The key takeaway is that the more efficient sorting algorithms will usually require fewer comparisons than Insertion Sort.

In the next section, we will explore more advanced techniques for sorting large datasets.
Selection Sort

We use Selection Sort when we have an unsorted array and we want to sort it. The algorithm works by repeatedly selecting the smallest element in the unsorted part of the array and moving it to the beginning of the array. The process is repeated until the array is fully sorted.

Here are the steps of the Selection Sort algorithm:

1. Find the smallest element in the unsorted part of the array.
2. Swap this element with the first element of the unsorted part.
3. Repeat steps 1 and 2 with the remaining unsorted part of the array.

Program 11.1

```c
/* ASSUMPTION: x is a sorted permutation of X[0:7]/
   t = 7
   x[t] = [t+1]
   x[(t)] = [t+1]
   while t > = 0 do:
     if t > n: i = t
     else if x[t] > x[i]:
       t = i
     end
   end
   sort 0 <= t <= n
   for i = 1 to n:
     if x[i] <= x[i-1]:
       x[i] = x[i-1]
     end
   end
   for i = 0 to n:
     if x[i] = x[i+1]:
       t = i
     end
   end
```

Progress of insertion sort on an array of seven items. The bracket indicators show the current position of the array.

Figure 11.2
Heapsort is a version of selection sort in which we use heap-like structures to keep the items in order. The heap is a complete binary tree in which each parent node is smaller than its children. This property allows us to efficiently find and move the smallest remaining element at each step of the sort.

The number of swaps required for selection sort is $\frac{\log n}{\log \log n}$ in the worst case. This is because selection sort compares all $n$ items and performs $n-1$ swaps.

Algorithm 11.2

Assume that we have sorted the smaller items in $X[0..n-1]$, then swap $X[i]$ with the smaller of the two elements:

1. If $X[i]$ is smaller than $X[min]$, then:
   - Swap $X[i]$ with $X[min]$.
   - Set $min = i$.
   - Increment $i$.

2. If $X[i]$ is not smaller than $X[min]$, then:
   - Increment $i$.

This algorithm is then applied to the remaining elements in the array.

Figure 11.3

The diagram shows the progression of the algorithm, where each step shows the state of the array after sorting the next element. The final result is the sorted array.

Figure 11.4

The figure illustrates the heap structure used in the algorithm. Each step shows the state of the array after applying the algorithm to the next element.
Quick sort is a fast sorting algorithm. It is a divide-and-conquer algorithm. The algorithm works by partitioning the array into two sub-arrays, sorting each sub-array using recursion, and then concatenating the two sorted sub-arrays. The partitioning process involves selecting a 'pivot' element from the array and rearranging the array elements so that all elements less than the pivot are placed before it, and all elements greater than the pivot are placed after it. The pivot element then becomes the final position of the pivot, and the algorithm is recursively applied to the sub-arrays to the left and right of the pivot. The partitioning process is shown in Figure 1.6.

Algorithm 1.3: Quick sort

```
// Assumes x[] is the input array of integers
// Returns x[] sorted in ascending order
void quicksort(int x[], int n)
{
    if (n < 1) return;
    int pivot = x[0]; // Select the first element as pivot

    int i = 0, j = n - 1;
    while (i < j)
    {
        while (x[i] <= pivot) i++; // Find the right place for the pivot
        while (x[j] > pivot) j--;
        if (i < j)
        {
            int temp = x[i]; x[i] = x[j]; x[j] = temp;
            i++; j--;
        }
    }
    quicksort(x, i); // Sort the left sub-array
    quicksort(x + i, n - i); // Sort the right sub-array
}
```

The key to quick sort's efficiency lies in the partitioning step, which ideally splits the array into two sub-arrays of equal size. This can be achieved by using the 'median-of-three' rule, which selects the median value of the first, middle, and last elements of the array as the pivot. This improves the algorithm's performance, especially for nearly sorted arrays.

Figure 1.6: The partitioning process in quick sort.
In the partition procedure, the element to be partitioned is stored in the array slot indexed by the original value of low. The algorithm is as follows, as shown in Program 11.2. If partitioning the elements in the range low ≤ i < high, the function `partition()` is called on the range [low, high). The algorithm of partitioning an array is as follows:

**Program 11.2**

```c
int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j < high; j++)
    {
        if (arr[j] <= pivot)
        {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}
```

**Algorithm 11.4**

1. Choose a pivot element. Common choices are middle element or median of the first, middle, and last elements.
2. Using a partition function, rearrange the array so that:
   a. Elements less than or equal to the pivot are moved to the left of the pivot.
   b. Elements greater than the pivot are moved to the right of the pivot.
3. Recursively partition the sub-arrays on either side of the pivot until the array is sorted.

**QuickCheck**

- Show how to do this in three steps. Function `partition()` is called on the range [low, high).
Early Termination of Recursion in QuickSort

Recursion is one of the key notations that can become a significant problem. In designing algorithms, it is important to ensure that the recursion terminates in a reasonable amount of time. This is particularly true for recursive algorithms like QuickSort, where the time complexity can be exponential.

There are several strategies to avoid infinite recursion, such as

1. Setting a maximum depth for recursion.
2. Implementing a base case that stops the recursion when a certain condition is met.
3. Using iterative approaches instead of recursive ones where possible.

The choice of recursion depth and base case can significantly impact the performance of the algorithm.

Figure 11.9 illustrates this concept. The graph shows the relationship between the number of recursive calls and the time complexity for QuickSort. As the number of elements increases, the time required grows exponentially unless a proper depth is set.

Another important consideration is the choice of the pivot element. The performance of QuickSort is highly dependent on this choice. A good pivot selection strategy can reduce the time complexity from $O(n^2)$ to $O(n \log n)$.

Program 11.25: Two-Efficient Sorting Algoritims

```c
int quicksort(int arr[], int low, int high)
{
    int pivot, i;

    if (low < high)
    {
        pivot = partition(arr, low, high);
        quicksort(arr, low, pivot - 1);
        quicksort(arr, pivot + 1, high);
    }

    return arr;
}
```

This implementation uses the partition function to divide the array into two sub-arrays based on a pivot element, which can be chosen differently for different scenarios.

For example, choosing the middle element or the median of the first, middle, and last elements can improve performance in certain cases.

Checking the pivot against the element being split or choosing the median of the first, middle, and last elements can improve the performance of QuickSort.

Choosing the pivot can greatly affect the performance of the algorithm. It is important to choose a pivot that results in balanced sub-arrays to achieve optimal performance.

After choosing the pivot, the QuickSort algorithm recursively sorts the sub-arrays and merges them back together. This process repeats until the entire array is sorted.

In summary, the choice of pivot and the depth of recursion are critical factors in the performance of QuickSort and other recursive algorithms. Careful consideration of these aspects can lead to more efficient implementations.
two sorted linked lists, we can merge them into a single list.

Given two sorted linked lists, we can merge them into a single list. The process is similar to the merge function in the divide-and-conquer algorithm. The merge process consists of selecting the smaller element at each step and appending it to the result list.

The expression $n \log_2 n$ is the number of comparisons between and among elements of the two lists. This expression is larger than $2n$, which is the number of comparisons required for the divide-and-conquer algorithm.

The graph shows the comparison of the algorithm $O(n \log n)$ to $O(n)$ for various values of $n$.

We can see that for small values of $n$, the linear time complexity $O(n)$ is better than the logarithmic time complexity $O(n \log n)$. However, for larger values of $n$, the difference becomes negligible.

Section 1.4: Two Useful Sorting Ideas

1.4.1 QuickSort

QuickSort is a divide-and-conquer algorithm that works by partitioning the array into two subarrays. The pivot is selected, and the array is partitioned such that all elements less than the pivot are on the left, and all elements greater than the pivot are on the right.

The expression $n \log_2 n$ is the number of comparisons between and among elements of the two lists. This expression is larger than $2n$, which is the number of comparisons required for the divide-and-conquer algorithm.

The graph shows the comparison of the algorithm $O(n \log n)$ to $O(n)$ for various values of $n$.

We can see that for small values of $n$, the linear time complexity $O(n)$ is better than the logarithmic time complexity $O(n \log n)$. However, for larger values of $n$, the difference becomes negligible.

Section 1.4: Two Useful Sorting Ideas

1.4.1 QuickSort

QuickSort is a divide-and-conquer algorithm that works by partitioning the array into two subarrays. The pivot is selected, and the array is partitioned such that all elements less than the pivot are on the left, and all elements greater than the pivot are on the right.

The expression $n \log_2 n$ is the number of comparisons between and among elements of the two lists. This expression is larger than $2n$, which is the number of comparisons required for the divide-and-conquer algorithm.

The graph shows the comparison of the algorithm $O(n \log n)$ to $O(n)$ for various values of $n$.

We can see that for small values of $n$, the linear time complexity $O(n)$ is better than the logarithmic time complexity $O(n \log n)$. However, for larger values of $n$, the difference becomes negligible.

Section 1.4: Two Useful Sorting Ideas

1.4.1 QuickSort

QuickSort is a divide-and-conquer algorithm that works by partitioning the array into two subarrays. The pivot is selected, and the array is partitioned such that all elements less than the pivot are on the left, and all elements greater than the pivot are on the right.

The expression $n \log_2 n$ is the number of comparisons between and among elements of the two lists. This expression is larger than $2n$, which is the number of comparisons required for the divide-and-conquer algorithm.

The graph shows the comparison of the algorithm $O(n \log n)$ to $O(n)$ for various values of $n$.

We can see that for small values of $n$, the linear time complexity $O(n)$ is better than the logarithmic time complexity $O(n \log n)$. However, for larger values of $n$, the difference becomes negligible.

Section 1.4: Two Useful Sorting Ideas

1.4.1 QuickSort

QuickSort is a divide-and-conquer algorithm that works by partitioning the array into two subarrays. The pivot is selected, and the array is partitioned such that all elements less than the pivot are on the left, and all elements greater than the pivot are on the right.

The expression $n \log_2 n$ is the number of comparisons between and among elements of the two lists. This expression is larger than $2n$, which is the number of comparisons required for the divide-and-conquer algorithm.

The graph shows the comparison of the algorithm $O(n \log n)$ to $O(n)$ for various values of $n$.

We can see that for small values of $n$, the linear time complexity $O(n)$ is better than the logarithmic time complexity $O(n \log n)$. However, for larger values of $n$, the difference becomes negligible.

Section 1.4: Two Useful Sorting Ideas

1.4.1 QuickSort

QuickSort is a divide-and-conquer algorithm that works by partitioning the array into two subarrays. The pivot is selected, and the array is partitioned such that all elements less than the pivot are on the left, and all elements greater than the pivot are on the right.

The expression $n \log_2 n$ is the number of comparisons between and among elements of the two lists. This expression is larger than $2n$, which is the number of comparisons required for the divide-and-conquer algorithm.

The graph shows the comparison of the algorithm $O(n \log n)$ to $O(n)$ for various values of $n$.

We can see that for small values of $n$, the linear time complexity $O(n)$ is better than the logarithmic time complexity $O(n \log n)$. However, for larger values of $n$, the difference becomes negligible.

Section 1.4: Two Useful Sorting Ideas

1.4.1 QuickSort

QuickSort is a divide-and-conquer algorithm that works by partitioning the array into two subarrays. The pivot is selected, and the array is partitioned such that all elements less than the pivot are on the left, and all elements greater than the pivot are on the right.

The expression $n \log_2 n$ is the number of comparisons between and among elements of the two lists. This expression is larger than $2n$, which is the number of comparisons required for the divide-and-conquer algorithm.

The graph shows the comparison of the algorithm $O(n \log n)$ to $O(n)$ for various values of $n$.

We can see that for small values of $n$, the linear time complexity $O(n)$ is better than the logarithmic time complexity $O(n \log n)$. However, for larger values of $n$, the difference becomes negligible.

Section 1.4: Two Useful Sorting Ideas

1.4.1 QuickSort

QuickSort is a divide-and-conquer algorithm that works by partitioning the array into two subarrays. The pivot is selected, and the array is partitioned such that all elements less than the pivot are on the left, and all elements greater than the pivot are on the right.

The expression $n \log_2 n$ is the number of comparisons between and among elements of the two lists. This expression is larger than $2n$, which is the number of comparisons required for the divide-and-conquer algorithm.

The graph shows the comparison of the algorithm $O(n \log n)$ to $O(n)$ for various values of $n$.

We can see that for small values of $n$, the linear time complexity $O(n)$ is better than the logarithmic time complexity $O(n \log n)$. However, for larger values of $n$, the difference becomes negligible.
11.5
In time, $O(n^2)$ is a constant fraction of $n^2$, and so $n^2$ would be an insertion sort. We start with the sequence of two elements, and $O(n)$ is to invert the insertion sort to get the right class. The original insertion sort is the one for two elements, and $O(n)$ is the time it takes to invert the insertion sort.

Suppose that we can choose a hash function that distributes the $n$ elements evenly. If the hash function is not chosen randomly, then the elements are not well spread, and the code that we choose will be a good insertion sort. If the hash function is chosen randomly, then the elements are well spread, and the code that we choose will be a good insertion sort.

When we distribute the time it takes to invert the insertion sort to get the right class, the original insertion sort is the one for two elements, and $O(n)$ is the time it takes to invert the insertion sort.

An obvious implementation of this idea uses $O(n)$ space to store

Algorithm 11.5

A single sorted linked list is used throughout this algorithm. The algorithm is as follows:

1. Set up the algorithm. Let $O(n)$ be the algorithm.
2. Let $O(n)$ be the algorithm. Now, consider the input. Suppose someone gave you a pile of about 100 index cards containing
interchange the contents of \([x]_1\) and \([x]_2\) and \([x]_3\) and \([x]_4\)

++ min

(\(\overline{\exists}x\) \(\exists y\) \(\exists z\) \(\exists w\) \(x > y > z > w\)) / 

(\(\overline{\exists}x\) \(\exists y\) \(\exists z\) \(\exists w\) \(x \geq y \geq z \geq w\)) for \(i = 1\) to \(n - 1\) loop

++ min

around \([x]_n\)

11. Show the following algorithm partitions array \([x]_n\):

\[a \geq b \geq c \geq d\]

10. Write a function that does not perform the integer.

7. Write a smart swaps function that does not perform the integer.

6. Implement selection sort.

5. Verify the correctness of selection sort.

4. Implement quicksort.

3. Write a function that inserts an integer.

2. Show that \(O(n^2)\) comparisons are necessary to verify that an array

EXERCISES

EXERCISES
Algorithm 11.6

```
Algorithm 11.6: Bubble Sort

1. [!] x = [!]
x
2. if x > [!+1]x then
3. [!] x = [!]x
4. [!] x = [!]x
5. end if
6. if y < [!0]y then
7. [!] y = [!]y
8. [!] y = [!]y
9. end if
10. return x, y
```

References

36. Can you construct an input on which Algorithm II1 does not work?
37. How many comparisons and swaps does Algorithm II1 use?
38. Devise an invariance assertion to show that Algorithm II1 works.

Exercises

69. Which of your exercises in Chapter 2 of this text are relevant to the problem at hand?
70. Write a function to merge two binary trees.
To set up a double-entry bookkeeping system.

12.1 Double-entry bookkeeping

Transaction is entered incorrectly. Program 1.2 checks for ledger account name. In accounts payable, check for ledger account name. It was used as a vehicle to demonstrate that it was needed in a vehicle in 1996. The exercise above illustrates the importance of keeping a checkbook by checking for double entries. The process of checking for double entries quickly becomes easier when one remembers that the ledger account is a double entry. The ledger account is a double entry.