Chapter 4

Sorting

Sorting is the process of re-arranging a given set of objects in a specific order. It is the most frequently performed activity in data processing.

In this chapter, we will study several sorting algorithms. We assume that all the data to be sorted can be stored in memory, or sorting in place. That is, each element can be access randomly and efficiently.

Assume that the input data is stored in an array $x[1..n]$.

We shall use x_i to denote the i-th element in the array.

The C programming languages address the array starting from 0, not 1. The first element $x[0]$ is not used to store the input data in the algorithm described in this chapter, but it may be used to store useful information to make the algorithms more efficient.

For simplicity, assume that the data to be sorted are integers.

In practical applications, sorting is usually performed on a set of records. Each record contains many fields. Sorting is based on one or several fields.

We shall re-arrange the data in non-decreasing order.

4.1 Basic Sorting Algorithms

4.1.1 Insertion Sort

The strategy of insertion sort is to divide the elements in the list into sorted part and un-sorted part.

And then repeatedly insert the first un-sorted element into the sorted part.

44 | 55 12 42 94 18 06 67
44 55 | 12 42 94 18 06 67
12 44 55 | 42 94 18 06 67
⋮

Input: n, $x_1, x_2, \ldots, x_n$
Output: $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$
Method:

```
for (i = 2; i ≤ n; i = i + 1) {
  y = x_i; x_0 = y;
  j = i − 1;
  while (x_j > y) {
    x_{j+1} = x_j; j = j − 1;
  }
  x_{j+1} = y;
}
```

Figure 4.1: Insertion sort

Assume that the input data is sorted, then the inner loop will execute only once. Therefore, the time complexity is $O(n)$. This is the best case.

Assume that the input data is inverted ordered. Then the inner loop will execute i times.

Therefore,

$$T(n) = \left(\sum_{i=2}^{n} c\, i \right) + d,$$

where c is the amount of computation in the inner loop and d is the amount of computation in the rest of the algorithm.

This implies that $T(n)$ is $O(n^2)$. This is the worst case.

Binary insertion sort

In the above algorithm, elements will be inserted into the sorted part of the list. Binary search can be used to find where to insert the element.

Input: n, $x_1, x_2, \ldots, x_n$
Output: $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$
Method:

```
for (i = 2; i ≤ n; i = i + 1) {
  y = x_i;
  l = 1; r = i − 1;
  while (l ≤ r) {
    m = (l + r)/2;
    if (y < x_m) r = m − 1; else l = m + 1;
  }
  for (j = i − 1; j ≥ l; j = j − 1)
    x_{j+1} = x_j;
  x_l = y;
}
```

Figure 4.2: Binary insertion sort

At iteration i, the position of the element to be inserted can be found in the $O(\log(i-1))$ comparisons.

Therefore, the number of comparisons is $\sum_{i=2}^{n} \log(i-1)$, which is $O(n \log n)$.

However, the number of data movements is still $O(n^2)$.

4.1.2 Selection sort

The strategy of selection sort is to partition the list into sorted part and un-sorted part.

Then it repeatedly selects the smallest element in the unsorted part and swaps it with the first element in the unsorted part.

| 44 55 12 42 94 18 <u>06</u> 67
06 | 55 <u>12</u> 42 94 18 44 67
06 12 | 55 42 94 <u>18</u> 44 67
⋮

Input: n, $x_1, x_2, \ldots, x_n$
Output: $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$
Method:

```
for (i = 1; i < n; i = i + 1) {
  y = x_i; k = i;
  for (j = i + 1; j ≤ n; j = j + 1) {
    if (x_j < y) {
      y = x_j; k = j;
    }
  }
  x_k = x_i; x_i = y
}
```

Figure 4.3: Selection sort

It is easy to see that

1. The number of data movements is only linear.
2. The number of comparisons is $O(n^2)$.

4.2 Efficient Sorting Algorithms

Theorem 4.1 *Selecting the smallest element in an unsorted list of n elements requires at least $n-1$ comparisons.*

We can also show that any sorting algorithm which exchange only adjacent elements will need to do $\Omega(n^2)$ exchange of adjacent elements.

Definition 4.1 *For each pair of distinct elements x_i and x_j in the list, if $x_i > x_j$ and $i < j$, then it is called an inversion.*

1. sorted list: no inversions
2. inverted list: $\binom{n}{2}$ inversions

Theorem 4.2 *Any algorithm that sorts elements by removing at most 1 inversion after each exchange of elements must do at least $\frac{n(n-1)}{4}$ exchange of elements on average.*

This theorem can also be stated in the following way.

Theorem 4.3 *Any algorithm that sorts elements by removing at most 1 inversion after each comparison must do at least $\frac{n(n-1)}{4}$ comparisons on average.*

Therefore, bubble sort requires $O(n^2)$ time, in the worst case.

In order to design an efficient sorting algorithm, we should exchange data as far as possible and avoid repeatedly selecting the smallest (largest) element from a set of too many ($O(n)$) elements.

4.3 Quick Sort

Quick sort selects a pivot value p, and then partition the lists into two parts.

1. All elements in the first part are less than or equal to p.
2. All elements in the second part are greater than or equal to p.
   An example:

   $$\underline{44}\ 55\ 12\ 42\ 94\ 06\ \underline{18}\ 67$$
   $$18\ \underline{55}\ 12\ 42\ 94\ \underline{06}\ 44\ 67$$
   $$\underbrace{18\ 06\ 12}_{\leq 42}\ 42\ \underbrace{94\ 55\ 44\ 67}_{\geq 42}$$

   Another example:

   $$\underline{44}\ 55\ 12\ 42\ 94\ 66\ \underline{18}\ 67$$
   $$18\ \underline{55}\ 12\ \underline{42}\ 94\ 66\ 44\ 67$$
   $$\underbrace{18\ 42\ 12}_{\leq 42}\ \underbrace{55\ 94\ 55\ 44\ 67}_{\geq 42}$$

3. The algorithm then sorts the two parts recursively.

Input: n, $x_1, x_2, \ldots, x_n$
Output: $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$
Method:

```
qsort (a[], l, r)
{
  i = l; j = r;
  p = a_k where k is randomly chosen from [l, r];
  while (i ≤ j) {
    while (a_i < p) i = i + 1;
    while (p < a_j) j = j − 1;
    if (i ≤ j) {
      t = a_i; a_i = a_j; a_j = t;
      i = i + 1; j = j − 1;
    }
  }
  if (l < j) qsort (a, l, j);
  if (i < r) qsort (a, i, r);
}

{
  qsort (x, 1, n);
}
```

Figure 4.4: Quick sort

Time complexity of quick sort

$$T(n) = \begin{cases} c & \text{if } n = 1, \\ T(u) + T(v) + an + b & \text{if } n > 1. \end{cases}$$

where $1 < u, v < n$, and $u + v \leq n$.

1. If $u = 1$ or $v = 1$, then no recursive calls are required.
2. If, in every partition of a m-element array, the smaller part always contains at least cm elements for some constant c with $0 < c \leq 1/2$ (equivalently, the larger part contains at most $(1 - c)m$ elements), then $T(n)$ is $O(n \log n)$.
   Otherwise, $T(n)$ is $O(n^2)$.

Space complexity of quick sort

When a recursive call occurs, all local variables are assigned a new space.

Therefore, Quicksort requires extra space to handle recursion.

The amount of extra space required is proportional to the depth of the recursive call.

The following version of quick sort is non-recursive.

It uses stack to eliminate recursive calls.

Each entry in the stack stores two values l and r, indicating that the subarray $x[l, r]$ still needs to be sorted.

Input: n, $x_1, x_2, \ldots, x_n$
Output: $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$
Method:

```
int l[N], r[N];                    // stack
s = 1; l_s = 1; r_s = n;
while (s > 0) {
  l = l_s; r = r_s; s = s − 1;
  i = l; j = r;
  p = x_k where k is randomly chosen from [l, r];
  while (i ≤ j) {
    while (x_i < p) i = i + 1;
    while (p < x_j) j = j − 1;
    if (i ≤ j) {
      z = x_i; x_i = x_j x_j = z;
      i = i + 1; j = j − 1;
    }
  }
  if (l < j) {s = s + 1; l_s = l; r_s = j;}
  if (i < r) {s = s + 1; l_s = i; r_s = r;}
}
```

Figure 4.5: Non-recursive version of quick sort

How big the stack must be for sorting n elements? (N =?)

In the worst case,

1. One part of the partition P_1 contains only 2 elements, and the other part P_2 contains $n - 2$ elements, and
2. P_1 is pushed onto the stack before P_2.

Then the size of the stack will be $n/2$.

It turn out that the size of the stack can be small if we sort the shorter part of the unsorted array first.

Theorem 4.4 *If the smaller part of the unsorted array is to be sorted first, then the height of the stack is bounded by* $\log_2 n + 1$.

The proof is by induction on n.

The details of the proof are left as an exercise.

Therefore, the non-recursive version of quicksort can save space if the smaller parts are sorted first.

In the case of uneven partitioning, the recursive version requires a larger system stack to handle the recursive calls, regardless of whether the smaller parts are sorted first.

4.3.1 Merge Sort

The strategy of merge sort is to divide the list into two parts of equal size, sort the two parts recursively, and then merge the two sorted parts.

Input: n, $x_1, x_2, \ldots, x_n$
Output: $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$
Method:

```
mergesort (a[], l, r)
{
  if (l < r) {
    k = (l + r)/2;
    mergesort (a, l, k);
    mergesort (a, k + 1, r);
    merge (a, l, k, a, k + 1, r, b);
    copy b[l..r] to a[l..r];
  }
}

{
  mergesort (x, 1, n);
}
```

Figure 4.6: Merge sort

```
merge (a[], l_1, r_1, b[], l_2, r_2, c[])
{
  i = l_1; j = l_2; k = l_1;
  while ((i ≤ r_1)&&(j ≤ r_2)) {
    if (a_i < b_j) {
      c_k = a_i; i = i + 1;
    } else {
      c_k = b_j; j = j + 1;
    }
    k = k + 1;
  }
  while (i ≤ r_1) { c_k = a_i; i = i + 1; k = k + 1; }
  while (j ≤ r_2) { c_k = b_j; j = j + 1; k = k + 1; }
}
```

Figure 4.7: Merge two sorted lists

Analysis of Merge Sort

Let b, c and d be constants.

$$T(n) = \begin{cases} b & \text{if } n = 1, \\ 2T(n/2) + cn + d & \text{if } n > 1. \end{cases}$$

Therefore, $T(n) = O(n \log n)$.

Merge sort has a disadvantage: the algorithm requires an additional array of size n to do the merge.

There is no easy way to merge the two sorted list of size $n/2$ efficiently without additional storage of size n.

4.3.2 Heap Sort

Let the data to be sorted be $x_1, x_2, \ldots, x_n$.

We may regard the data as a rooted ordered binary tree in the following way.

> Each x_i has its left child x_{2i} and right child x_{2i+1}.

Note that, if $2i > n$ then x_i is a leaf; if $n = 2i$, then x_i has only left child, no right child.

Consider a special type of binary tree, called a heap which is defined as a rooted ordered binary tree with

$$x_i \geq \max\{x_{2i}, x_{2i+1}\}$$

for each internal node x_i.

That is, the value stored in each internal node is no less than the values stored in its left and right children.

Note that trees are not required to define a heap. We can also define a heap by using a list.

An example of a heap: 50, 24, 30, 20, 21, 18, 3, 12, 5, 6.

It is easy to see that if $x_1, x_2, \ldots, x_n$ is a heap, then x_1 is the largest element.

Heap sort first makes the input data $x_1, x_2, \ldots, x_n$ into a heap, then it swaps the values of x_1 and x_n.

Now the largest element is in its place. The algorithm then removes the last element from the input data, and repeat the procedure repeatedly until the input data is sorted.

After swapping x_1 and x_n, we need to ignore x_n and make the data into heap again.

It turn out that this can be done in $O(\log m)$ time, where m is the current number of elements in the heap.

We first describe the algorithm and then analyze its complexity.

Let n be the number of elements in the heap. The following procedure `sift` when called with parameters $a[]$, r, and n will make the sub-tree stored in an array $a[1..n]$ and rooted at r a heap

It is required that, before `sift`(a, r, n) was called, both left and right sub-trees of r must be heaps.

```
sift (a[], r, n)
{
  i = r; j = 2 * i; x = a_i;
  while (j ≤ n) {
    if (j < n)
      if (a_j < a_{j+1}) j = j + 1;
    if (x ≥ a_j) break;
    a_i = a_j; i = j; j = 2 * i;
  }
  a_i = x;
}
```

Figure 4.8: The procedure `sift` to make the sub-tree stored in an array $a[1..n]$ and rooted at r a heap.

The program to sort a sequence of n integers $x_1, x_2, \ldots, x_n$ using heap sort can be described as follows.

Input: n, $x_1, x_2, \ldots, x_n$
Output: $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$
Method:

```
  r = n/2;
  while (r > 0) {
    sift(x, r, n); r = r − 1;
  }
  m = n;
  while (m > 0) {
    y = x_1; x_1 = x_m; x_m = y;
    m = m − 1; sift(x, 1, m);
  }
```

Figure 4.9: Heap sort

Analysis of Heap Sort

Assume that the data to be sorted has n elements, it is stored in an array $a[1..n]$.

Consider the array as a binary tree as defined above.

Define the height of a node u to be $h(u) = 0$ if u is a leaf.

Otherwise let v and w be its left and right child, respectively. Define

$$h(u) = \max\{h(v), h(w)\} + 1.$$

(When u has only left child, omit $h(w)$ in the above definition.)

let the height of the tree be k, that is, $k = \lfloor \log n \rfloor + 1$.

There is only one node with height k — the root.

There are at most 2 nodes with height $k - 1$, at most 4 nodes with height $k - 2$, etc.

In general, there are at most 2^i nodes with height $k - i$, $i = 0, 1, \ldots, k$.

The time complexity of $\texttt{sift}(a[], r, n)$ is $O(h(r))$, where $h(r)$ is the height of node r.

Therefore, the time complexity of making a heap from the input data is bounded by

$$\begin{aligned}
\sum_{i=0}^{k} (k-i)2^i &= \sum_{i=0}^{k} k2^i - \sum_{i=0}^{k} i2^i \\
&= k(2^{k+1} - 1) - (k2^{k+1} - 2^{k+1} + 2) \\
&= 2^{k+1} - k - 2 = 2n - \log n - 2.
\end{aligned}$$

This is linear time.

The second while-loop in the heap sort algorithm takes $O(\log m)$ time in each iteration, where m is the number of element in the heap.

$$\sum_{m=n}^{1} \log m,$$

which is $\Omega(n \log n)$.

Therefore, the time complexity of the heap sort is $O(n \log n)$.

4.4 Lower Bound for Sorting by Comparison

The best algorithms we have studied so far have their time complexity $O(n \log n)$ in the worst case. Can the time complexity of sorting be further improved?

The following theorem shows that this is impossible if comparison of keys is the only way to determine the order of elements in the sorting.

Theorem 4.5 *Any sorting algorithm in which comparison of keys is used in determining the order of elements must do* $\Omega(n \log n)$ *comparisons.*

Proof. The proof is by constructing the decision tree. Assume that each element is distinct. After each comparison, there are only two cases, $x_i < x_j$ or $x_i > x_j$. If there is only one comparison, at most two outcomes can be distinguished. In general, k comparisons can distinguish at most 2^k outcomes. Therefore, the decision tree need to have enough height to distinguish $n!$ distinct possible outcomes.

Let the height of the decision tree be h. Then there are at most 2^h leaves. Since there must be at least $n!$ leaves,

$$2^h \geq n!.$$

Therefore,

$$h \geq \log(n!).$$

Note that

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n < n! < \sqrt{2\pi n}\left(1 + \frac{1}{12n}\right)\left(\frac{n}{e}\right)^n.$$

We conclude that

$$h \geq \log(\sqrt{2n\pi}(n/e)^n) = \frac{1}{2}\log(2n\pi) + n\left(\log n - \log(e)\right).$$

4.5 Other Sorting Algorithms

In this section, we introduce sorting algorithms without comparison of keys. This is not possible in general. When the input is drawn from a specific domain, then it may be possible to sort without comparing keys.

4.5.1 Counting Sort

Assume that the input data are drawn from $[0, m)$. That is, $0 \leq x_i < m$.

We need m counters $c[0, ..m-1]$.

Each counter $c[i]$ stores the number of occurrences of i in $x_1, x_2, \ldots, x_n$.

Input: n, $x_1, x_2, \ldots, x_n$, x_i are integers in $[0, m)$
Output: $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$
Method:

```
for (i = 0; i < m; i = i + 1) c_i = 0;
for (i = 1; i ≤ n; i = i + 1) c_{x_i} = c_{x_i} + 1;
for (i = 0; i < m; i = i + 1)
  while (c_i > 0) {
    print i; c_i = c_i − 1;
  }
```

Figure 4.10: Counting sort

Although it is in a while-loop, it is easy to check that the "print" instruction will be performed at most n times during the execution of the algorithm.

Therefore, the time complexity of counting sort is $O(m + n)$.

If m is $O(n)$, or independent of n, then we can apply counting technique to sort the input data in linear time.

4.5.2 Radix Sort

Suppose that the keys are composed of k-digits.

We may sort the data by the least significant digit first.

Then sort the data by the 2-nd least significant digit, and so on, until the most significant digit is sorted.

Radix sort for k-digit data consists of k phases. Any efficient sorting algorithm can be used in each phase.

However, when the keys are the same, the algorithm should keep the original order of the data.

Sorting algorithms with this property are called stable.

Not all sorting algorithms are stable. Modifications of unstable sorting algorithms should be done before they can be used in radix sort.

Exercises

⋆1. Proof: For the non-recursive version of quick sort as shown in Figure 4.5, if after partitioning, a smaller part of the unsorted list is sorted first, the stack size is $\log n + 1$bounds.
Hint: Assume the index of the stack starts from 1. Let (l, r) be the values stored in the k-th entry in the stack, then $r - l + 1 \leq n/2^{k-1}$.

2. Which of the sorting algorithms described in Section 4.1.1 to Section 4.3.2 are stable and which are not. Design a general method to modify an unstable sorting algorithm into a stable one.

⋆3. For each of the sorting algorithms described in Section 4.1.1 to Section 4.3.2, analyze the time complexity to sort a list of n data which is already sorted and a list which is inverted. (In quick sort, assume that the $(l + r)/2$-th element is always selected as the pivot element.)

⋆⋆4. Given 12 balls with the same diameter and color. They all have the same weight, except one of them which can be lighter or heavier. Show how to identify the ball with different weight with a balance. Any number of balls can be placed on both sides of the balance. Your goal is to design a procedure to identify the ball with different weight in 3 measurements. Can you do it with 2 measurements? Justify your answer.

⋆⋆5. Write sorting programs based on the algorithms described in this chapter and compare the performances of these algorithms by using randomly generated data of sizes 100, 500, 1000, 2500, 5000, 10000, 50000, 100000, 500000, and 1000000.

⋆6. You are given an infinite array $A[1..\infty]$ in which the first n cells contain integers in sorted order and the rest of the cells are filled with ∞. You are not given the value of n. Describe an algorithm that takes an integer x as input and finds a position in the array containing x, if such a position exists, in $O(\log n)$ time. (If you are disturbed by the fact that the array A has infinite length, assume instead that it is of length n, but that you don't know this length, and that the implementation of the array data type in your programming language returns the error message ∞ whenever elements $A[i]$ with $i > n$ ($i \geq n$ in C) are accessed.)

7. Given a sorted array of distinct integers $A[1..n]$, design a divide-and-conquer algorithm, that runs in time $O(\log n)$, to find out whether there is an index i for which $A[i] = i$.

⋆8. Show that any array of integers $x[1..n]$ can be sorted in $O(n + M)$ time, where $M = \max_i\{x_i\} - \min_i\{x_i\}$.