Chapter 7

Introduction to Computational Complexity: The Sorting Problem

Preview

- Overview
- Lecture Notes
  - Computational Complexity
  - Insertion Sort and Selection Sort
  - Lower Bounds for Algorithms that Remove at Most One Inversion per Comparison
- Quick Check
  - Mergesort Revisited
  - Quicksort Revisited
  - Heapsort
    - Heaps and Basic Heap Routines
    - An Implementation of Heap sort
  - Comparison of Mergesort, Quicksort, and Heapsort
- Quick Check
  - Lower Bounds for Sorting Only by Comparison of Keys
    - Decision Trees for Sorting Algorithms
    - Lower Bounds for Worst-Case Behavior
    - Lower Bounds for Average-Case Behavior
  - Sorting by Distribution (Radix Sort)
- Quick Check
- Classroom Discussion
- Homework

- Keywords
- Important Links

Overview

This chapter introduces some approaches to attacking a problem. One is to try to develop a more efficient algorithm for the problem. The other is to try to prove that a more efficient algorithm is not possible. Once we have such a proof, we know that we should quit trying to obtain a faster algorithm.

Lecture Notes

Computational Complexity

Computational complexity, a field that runs hand-in-hand with algorithm design and analysis, is the study of all possible algorithms that can solve a given problem. A **computational complexity analysis** tries to determine a lower bound on the efficiency of all algorithms for a given problem. This does not mean that it must be possible to create an algorithm with the time complexity for the problem. It means only that is impossible to create one that is better than time complexity. In general, our goal for a given problem is to determine a lower bound of (f (n)) and develop a Υ(f (n)) algorithm for the problem. Once we have done this, we know that, except for improving the constant, we cannot improve on the algorithm any further.

The investigation can proceed in two directions. On one hand, we can try to find a more efficient algorithm using algorithm design methodology, while on the other hand, we can try to obtain a greater lower bound using computational complexity analysis.

We introduce computational complexity analysis by studying the Sorting problem. Several algorithms have been devised that solve the problem. By studying and comparing the algorithms, we can gain insight into how to choose among several algorithms for the same problem and how to improve a different algorithm. The problem of sorting is one of the few problems for which we have been successful in developing algorithms whose time complexities are about as good as our lower bound.

The class of sorting algorithms for which we have obtained algorithms about as good as our lower bound includes all algorithms that sort only by comparison of **keys**. Algorithms that **sort only by comparisons of keys** can compare two keys to determine which is larger, and can copy keys, but can do no other operations on them.

We analyze the algorithms in terms of the number of comparisons of keys and the number of assignments of records. We also analyze how much extra space the algorithms require besides the space needed to store the input. When the extra space is a constant, the algorithm is called an **in-place sort**. We assume we are always sorting in nondecreasing order.

Insertion Sort and Selection Sort

An **insertion sort algorithm** is an algorithm that sorts by inserting records in an existing sorted array. Assume that the keys in the first $i - 1$ slots are sorted. Let x be the value on the ith slot. Compare the x sequence with the key in the $(i - 1)$ st slot, the one in the $(i - 2)$ nd slot, etc, until a key found is smaller than x. Let j be the slot where that key is located. Move the key in slots $j + 1$ through $(i - 1)$ to slots $j + 2$ through i, and insert x in the (j+1)st slot. Repeat this process for i = 2 through $i = n$. Algorithm 7.1 illustrates this.

Table 7.1 summarizes the results concerning **Exchange Sort** and **Insertion Sort**. We can see from the table that, in terms of comparison of keys, Insertion Sort always performs at least as well as Exchange Sort and usually performs better. **Selection Sort** is a slight modification of Exchange Sort, and it removes one of the disadvantages of Exchange Sort. Selection Sort simply keeps track of the index of the current smallest key among the keys in the ith through nth slots. After determining that record, it

exchanges it with the record for the *i*th slot. Because three assignments are needed to do an exchange, *the every- case time complexity of the number of assignments of records done by Selection Sort* is:

$T(n) = 3(n - 1)$.

Exchange Sort sometimes does better than Selection Sort. Insertion Sort always performs at least as well as Selection Sort. However, Selection Sort's time complexity in terms of assignment of records is linear, whereas Insertion Sort's is quadratic. Linear time is much faster than quadratic time when *n* is large. In practice, none of these algorithms is practical for extremely large instances because all of them are quadratic-time in both the average case and the worst case.

Lower Bounds for Algorithms that Remove at Most One Inversion per Comparison

After each comparison, Insertion Sort either does nothing or moves the key in the *j*th slot to the (j+1)st slot. By moving each key in the *j*th slot up one slot, we have remedied the fact that *x* should come before that key. This is all we have done. All sorting algorithms that sort only by comparison of keys, and accomplish such a limited amount of rearranging after each comparison, require at least quadratic time. We obtain our results under the assumption that the keys to be sorted are distinct. The worst-case bound still holds true with this restriction removed because a lower bound o the worst-case performance from some sets of inputs is also a lower bound when all inputs are considered.

In general, we are concerned with sorting *n* distinct keys that come from any ordered set. We can assume that the keys are simply the positive integers 1,2,…, *n* and so on. A **permutation** is a rearrangement of elements where none are lost, added, or changed. See the text for an example of six permutations of integers 1, 2, and 3 and a discussion of Theorem 7.1.

Quick Check

1. Computational complexity is the study of all possible algorithms that can solve a given problem. (True or False)
   Answer: true
2. Insertion sort inserts records in an existing ____ array.
   Answer: sorted
3. ____ sort works by looking at entire array.
   Answer: Selection
4. An inversion in a permutation is a pair (ki, kj). (True or False)
   Answer: True

Mergesort Revisited

Mergesort was introduced before, but we will revise it and then we will introduce some improvements to it. The basic idea of Mergesort, is that we have a list of numbers we break it into two halves of approximately equal size and break each one into two halves until we get one number then we return and merge it with the preceding number.

As mentioned in the proof of Theorem 7.1, algorithms that remove at most one inversion after each comparison will do at least $n(n - 1)/2$ comparisons when the input is in reverse order. Figure 7.2 shows that when the subarrays are merged, the comparisons remove more than one inversion.

We can improve the basic Mergesort algorithm in three ways. One is a dynamic programming version of Merge sort, another is a linked version, and the third is a more complex merge algorithm.

For the first improvement, you do not need to divide the array until singletons were reached. Rather you could simply start with singletons, merge the singletons into groups of two, then into groups of four, and so on until the array was sorted. We can write an iterative version of Mergesort that uses this method, and we can avoid the overhead of the stack operations needed to implement recursion. This is a dynamic programming approach to Mergesort. Refer to the text for a more detailed discussion of the algorithm.

The second improvement of Mergesort is a linked version of the algorithm. The Sorting problem involves sorting of records according to the values of their keys. If the records are large, the amount of extra space used by Mergesort can be considerable. We can reduce the extra space by adding a link field to each record. We then sort the records into a sorted linked list by adjusting the links rather than by moving the records. We will not need to create an extra array of records. In addition we will have time savings because the time required to adjust links is less than that needed to move large records. Refer to Figure 7.3 for an illustration.

The third improvement of Mergesort is a more complex merge algorithm that is presented in Huang and Langston (1988).

Quicksort Revisited

Quicksort has an advantage over Mergesort in that no extra array is needed. It is still not an in-place sort because, while the algorithm was sorting the first subarray, the first and last indices of the other subarray need to be stored in the stack of activation records. Unlike Mergesort, we have no guarantee that the array will always be split in the middle. In the worst case, *partition* may repeatedly split the array into an empty subarray on the right and a subarray with one less item on the left. In this way, n–1 pairs of indices will end up being stacked, which means that the worst-case extra space usage is in Y (n). It is possible to modify Quicksort so that the extra space usage is at most lg n.

To improve Quicksort we can reduce the extra space usage of Quicksort in five different ways. First, in procedure quick sort, we determine which subarray is larger and always stack that one while the other is sorted. Second, there is a version of partition that cuts the average number of assignments of records significantly. Third, each of the recursive calls in procedure Quicksort causes *low*, *high*, and *pivot* point to be stacked. A good deal of the pushing and popping is unnecessary. In the first recursive call to quick sort, only the values of pivot point and high need to be saved on the stack. In the second recursive call, nothing needs to be saved. Fourth, recursive algorithms such as Quicksort can be improved by determining a threshold value at which the algorithm calls an iterative algorithm instead of dividing an instance further. Finally, Quicksort algorithm is least efficient when the input array is already sorted. The closer the input array is to being sorted, the closer we are to this worst-case performance. Therefore, if we believe that the array may be close to already being sorted, we can improve the performance by not always choosing the first item as the pivot item.

Of course, we have no reason to believe that there is any particular structure in the input array, choosing any item for the pivot item is, on average, just as good as choosing any other item. In this case, all we really gain by taking the median is the guarantee that one of the subarrays will not be empty,

Heapsort

Heapsort is an in-place Y (n lg n)algorithm. We will first review heaps and describe basic heap routines needed for sorting using heaps.

Heaps and Basic Heap Routines

The depth of a node of a tree is the number of edges in the unique oath from the root to that node. The depth d of a tree is the maximum depth of all the nodes in the tree, and a leaf in a tree is any node with no children. An **internal node** in any tree is any node that has at least one child. A **complete binary tree** must meet the following conditions:

- All internal nodes have two children.
- All leaves have depth d.

An **essentially complete binary tree** must meet these conditions:

- It is a complete binary tree down to a depth of $d - 1$.
- All nodes with depth d are as far to the left as possible.

A **heap** is an essentially complete binary tree such that:

1. The values stored at the nodes come from an ordered set.
2. The value stored at each node is greater than or equal to the values stored at its children. This is called the **heap property**.

Suppose that we have somehow arranged the keys that are to be sorted in a heap. If we repeatedly remove the key stored at the root while maintaining the heap property, the keys will be removed in nonincreasing sequence. If, while removing them, we place them in an array starting with the *n*th slot and going down to the first slot, they will be sorted in nonincreasing sequence in the array. After removing the key in the root, we can restore the heap property by replacing the key at the root with the key stored at the bottom node, deleting the bottom node, and calling a procedure *siftdown* tht "sifts" the key now at the root down the heap until the heap property is restored. The sifting is accomplished by initially comparing the key at the root with the larger keys at the children of the root. If the key at the root is smaller, the keys are exchanged. This process is repeated down the tree until the key at a node is not smaller than the larger of the keys at its children. Refer to the text for the high-level pseudocode.

The only task remaining is to arrange the keys in a heap in the first place. We can transform an essentially binary tree into a heap by repeatedly calling *siftdown* to perform the following operations:

1. Transform all subtrees whose roots have depth d – 1 into heaps.
2. Transform all subtrees whose roots have depth d – 2 into heaps.
3. Transform the entire tree into a heap.

See the text for details about the high-level pseudocode used in the sort.

This Heapsort algorithm does not appear to be an in-place sort. In other words, we need extra space for the heap. We can implement a heap using an array. We can show that the same array that stores the input can be used to implement the heap, and we never simultaneously need the same array slot for more than one purpose.

An Implementation of Heap sort

We can represent a complete binary tree in an array by storing the root in the first array slot, the root's left and right children in the second and third slots, respectively, the left and right children of the root's left child in the fourth and fifth array slots, and so on. The array representation of the heap Figure 7.5 is shown in Figure 7.8. Notice that the index of the left child of a node is twice that of the node, and the index of the left child of the node is 1 greater than twice that of the node. Recall that in the high-level pseudocode for Heapsort, we required that the keys initially be in an essentially complete binary tree. If we place the keys in an array in an arbitrary order, they will be structured in some essentially complete binary tree. Refer to the text for the low-level pseudocode used in the representation.

We can now give an algorithm for Heapsort. The algorithm assumes that they keys to be sorted are already in *H.S.* This automatically structures them in an essentially binary tree according to the representation in Figure 7.8. After the essentially complete binary tree is made into a heap, the keys are deleted from the heap starting with the *n*th array slot and going down to the first array slot. Because they are placed in sorted sequence in the output array in that same order, we can use *H.S.* as the output array with no possibility of overwriting a key in the heap. Refer to the text for a discussion of the in-place algorithm.

Comparison of Mergesort, Quicksort, and Heapsort

Table 7.2 summarizes the results of using Mergesort, Quicksort, and Heapsort. Quicksort is usually preferred to Heapsort in terms of keys and assignment of records and also in terms of extra space usage. Quicksort is usually preferred to Mergesort as well, even though Quicksort does slightly more comparisons of keys on average. However, the linked implementation of Mergesort eliminates almost all the disadvantages of Mergesort. The only disadvantage remaining is the additional space used for Y (*n*) links.

Quick Check

1. Using dynamic programming Mergesort become worst. (True or False)
   Answer: False
2. Quicksort is a ___exchange sort
   Answer: partition
3. In heaps, the value stored at each node is greater than or equal to the values stored at its children. This is called the____.
   Answer: heap property
4. If we have very large data sets, Heapsort performs badly. (True or False)
   Answer: False

Lower Bounds for Sorting Only by Comparison of Keys

We have developed Y (*n* lg *n*) sorting algorithms, which represent substantial improvement over quadratic-time algorithms. A good question is whether we can develop sorting algorithms whose time complexities are of an even better order. We show that, as long as we limit ourselves to sorting only by comparisons of keys, such algorithms are not possible.

Although our results still hold if we consider probabilistic sorting algorithms, we obtain the results for deterministic sorting algorithms. We obtain our results under the assumption that the *n* keys are distinct.

Decision Trees for Sorting Algorithms

A **decision tree** is a tree that is constructed in such a way that a decision must be made at each node about which node to visit next. A decision tree is called **valid** for sorting *n* keys if, for each permutation, of the *n* keys, there is a path from the root to a leaf that sorts that permutation. For example, the decision tree in Figure 7.11 is valid for sorting three keys, but it would no longer be valid if we removed any branch of the tree. We say that a decision tree is **pruned** if every leaf can be reached from the root by making a consistent sequence of decisions. To every deterministic algorithm for sorting *n* keys there corresponds a pruned, valid decision tree.

Lower Bounds for Worst-Case Behavior

To obtain a bound for the worst-case number of comparison keys, we must refer to Lemma 7.2: The worst-case number of comparisons of keys done by a decision tree is equal to its depth.

For Lemmas 7.1 and 7.2, we need only find a lower bound on the depth of a binary tree containing *n*! leaves to obtain our lower bound for the worst-case behavior.

Lower Bound for Average Behavior

We obtain our results under the assumption that all possible permutations are equally likely to be the input.

If the pruned, valid, binary decision tree corresponding to a deterministic sorting algorithm for sorting *n* distinct keys contains any comparison nodes with only one child, we can replace each such node by its child and prune the child to obtain a decision tree that sorts using no more comparisons than did the original tree. Every non-leaf in the new tree will contain exactly two children. A binary tree in which every non-leaf contains exactly two children is called a **2-tree**.

The **external path length (EPL)** of a tree is the total length of all the paths from the root to the leaves. The number of comparisons done by a decision tree to reach a leaf is the length of the path to the leaf. Therefore, the EPL of a decision tree is the total number of comparisons done by the decision tree to sort all possible inputs. See the text for a discussion of the applicable lemmas.

Sorting by Distribution (Radix Sort)

Any algorithm that sorts only by comparisons of keys can be no better than Y (*n* lg *n*). if we know nothing about the keys except that they are from an ordered set, we have no choice but to sort by comparing the keys. When we have more information, we can consider other sorting algorithms.

If we know that the keys are nonnegative integers represented in base 10. Assuming that they all have the same number of digits, we can first distribute them into distinct piles based on the values of the leftmost digits. Each pile can then be distributed into distinct piles based on the values of the third digits from the left, and so on. This proves is called **sorting by distribution**.

One difficulty with this procedure is that we need a variable number of piles. If we allocate 10 piles, we inspect the digits from right to left, and we always place a key in the pile corresponding to the digit currently being inspected. The keys still end up sorted if we obey this rule: On each pass, if two keys are to be placed in the same pile, the key coming from the left pile (in the previous pass) is placed to the left of the other keys.

This sorting method precedes computers. It is called a **radix sort** because the information used to sort the keys is in a particular radix (base). The radix can be any number base, or we can use letters of the alphabet. The number of piles is the same as the radix.

Because the number of keys in a particular pile changes with each pass, a good way to implement the algorithm is to use linked lists. After each pass, the keys are removed from the lists (piles) by coalescing them into one master-linked list. They are ordered in that list according to the lists (piles) from which they were removed. In the next pass, the master list is traversed from the beginning, and each key is placed at the end of the list (pile) to which it belongs in that pass. In that way, the rule just given is obeyed.

Quick Check

1. A decision tree such that for each permutation of the n keys, there is a path from the root to a leaf is called ____.
   Answer: valid
2. If the worst-case number of comparisons done by a decision tree is equal to its depth then we obtain a lower bound for worst-case behavior. (True or False)
   Answer: True
3. In a certain sorting algorithm, if the keys are distributed into piles it is called ____.
   Answer: Sorting by distribution

Classroom Discussion

- Compare all the discussed sorting algorithms in term of comparison of keys and assignment of records.

- Discuss two instances where Quicksort is more appropriate.

Homework

Assign Exercises 3, 11, 26, and 35.

Keywords

- **2-tree** – a binary tree in which every non-leaf contains exactly two children.
- **Complete binary tree –** a binary tree that satisfies the following conditions:
  1. All internal nodes have two children.
  2. All leaves have depth *d*.
- **Computational complexity analysis** – is the study of all possible algorithms that can solve a given problem.
- **Computational complexity analysis –** tries to determine the lower bound on the efficiency of all algorithms for a given problem.
- **Decision tree –** a tree such that at each node a decision must be made as to which node to visit next.
- **Essentially complete binary tree –** is a binary tree that satisfies the following conditions:
  1. It is a complete binary tree down to a depth of d - 1.
  2. The nodes with depth d are as far to the left as possible.
- **External path length (EPL) –** of a tree is the total length of all paths from the root to the leaves.
- **Heap –** Is an essentially complete binary tree such that
  1. The values stored at the nodes come from an ordered set.
  2. The value stored at each node is greater than or equal to the values stored at its children. This is the **heap property**.
- **Heapsort** – a sort algorithm that builds a heap and then repeatedly extracts the maximum item.
- **In-place sort –** a sort algorithm in which the sorted items occupy the same storage as the original ones.
- **Internal node –** in a tree is any node that has at least one child (any node that is not a leaf).
- **Insertion Sort –** a sort algorithm in which the sorted items occupy the same storage place as the original ones.
- **Key –** an element of an ordered set.
- **Lemma –** a proven statement that is used as a stepping-stone to proving another statement.
- **Lower band –** a given running time of an algorithm and it is possible to create a better one.
- **Min-EPL –** the minimum of the EPL of 2-trees containing m leaves.
- **Permutation –** a rearrangement of items where none are lost, added, or changed.
- **Pruned –** every leaf can be reached from the root by making a consistent sequence of decisions.
- **Radix sort** – a sorting algorithm that sorts integers by processing individual digits, by comparing individual digits sharing the same significant position.
- **Selection Sort –** any sorting algorithm that selects records in order and puts them in their proper positions.
- **Sorting by distribution –** a sorting method that works by placing the elements into categories and then putting the elements in the categories back in the sequence in sorted order.
- **Sort only by comparisons of keys** – algorithms can compare two keys to determine which is larger, and can copy keys, but cannot do other operations on them.
- **Sorting task –** rearranging records so that they are in order according to the values of the keys
- **Transpose –** of a permutation is the reverse of it. For example, the transpose of [3, 2, 4, 1, 5] is [5, 1, 4, 2, 3].
- **Valid decision tree –** a decision tree is valid for sorting n keys if, for each permutation of the *n* keys, there is a path from the root to a leaf that sorts that permutation.

Important Links

- http://cs.smith.edu/~thiebaut/java/sort/demo.html (Demonstration of Sorting Algorithms)
- www.nist.gov/dads (Dictionary of Algorithms and Data Structures)