Chapter 2

Divide-and-Conquer

Preview

- Overview
- Lecture Notes
  - Binary Search
  - Mergesort
  - The Divide-and-Conquer Approach
  - Quick Sort (Partition Exchange Sort)
  - Strassen's Matrix Multiplication Algorithm
  - Arithmetic with Large Integers
  - Determining Thresholds
  - When Nor to Use Divide-and-Conquer
- Quick Check
- Quick Check
- Classroom Discussion
- Homework
- Keywords
- Important Links

Overview

This chapter introduces an approach to designing algorithms, **divide-and-conquer**. This approach, divides an instance of a problem into two or more smaller instances. The smaller instances are usually instances of the original problem. If solutions to the smaller instances can be obtained readily, the solution to the original instance can be obtained by combining these solutions. If the smaller instances are still too large to be solved readily, they can be divided into still smaller instances. This process of dividing the instances continues until they are so small that a solution is readily obtainable.

The divide-and-conquer approach is a **top-down** approach. The solution of a *top-level* instance of a problem is obtained by going down and obtaining solutions to smaller instances.

Lecture Notes

Binary Search

Remember the iterative version of Binary Search which we discussed in section 1.2. Here we will present a recursive version to illustrate the top-down used by divide and conquer.

Binary Search locates a key *x* in a non-decreasing sorted array by first comparing x with the middle item of the array. If they are equal, the algorithm is done. If not, the array is divided into two sub-arrays, one containing all the items to the left of the middle item and the other containing all the items to the right. If x is smaller than the middle item, this procedure is then applied to the left sub-array. Otherwise, it is applied to the right sub-array. That is, *x* is compared with the middle item of the appropriate sub-array. If they are equal, the algorithm is done. If not, the sub-array is divided in two. This procedure is repeated until *x* is found or it is determined that *x* is not in the array.

Binary Search is the simplest kind of divide-and-conquer algorithm because the instance is broken down into only one smaller instance, so there is no combination of outputs. The solution of the original instance is the solution to the smaller instance.

This recursive version of Binary Search employs that no operations are done after the recursive call (tail-recursion); it is straightforward to produce an iterative version. Indeed it is better to use iteration since iterative algorithms execute faster than recursive algorithms because no stack needs to be maintained.

In an actual implementation of a recursive routine, a new copy of any variable passed to the routine is made in each recursive call. If a variable's value does not change, the copy is unnecessary. This waste could be costly if the variable is an array. One way to circumvent this problem is to pass the variable by address. If the implementation language is C++, an array is automatically passed by address, and using *const* guarantees the array cannot be modified. Including all this information in our pseudocode expression of recursive algorithms clutters them and diminishes their clarity.

Because the recursive version of Binary Search employs **tail-recursion**, it is straightforward to produce an iterative version. A substantial amount of memory can be saved by eliminating a stack developed in the recursive calls. An iterative algorithm will execute faster than a recursive version.

Mergesort

Using **two-way merging**, we can combine two sorted arrays into one array. By repeatedly applying the merging procedure, we can sort an array. Eventually the size of the subarrays will become 1, and an array of size 1 is trivially sorted. This procedure is called **mergesort**. Mergesort has three steps.

Step 1: Divide the array into 2 sub-arrays each of n/2.

Step 2: Solve each sub-array by sorting it (use recursion till array is sufficiently small).

Step 3: Combine solutions to the sub-arrays by merging them into a single sorted array.

In Mergesort, we are concentrating on comparison. Suppose we have n keys we will not have more than n comparisons. In the worst case we will have n–1 and in the normal case we will have C<= nlog(n+1) C = O(nlogn).

An **in-place sort** is a sorting algorithm that does not use any extra space beyond that needed to store the input.

Divide-and-Conquer Approach

After studying two divide-and-conquer algorithms (Binary Search and Mergesort), we can better understand the general description of the divide-and-conquer approach. This approach involves these steps:

Step 1: Divide an instance of a problem into one or more smaller instances.

Step 2: Conquer solve each of the smaller instances (use recursion until the array is sufficiently small).

Step 3: Combine the solutions of the smaller instances to obtain the solution of the original instance.

Quicksort (Partition Exchange Sort)

The sorting algorithm we will discuss now is partition exchange sort or quicksort. Quicksort is similar to mergesort in that the sort is accomplished by dividing the array into two partitions and then sorting each partition recursively. In quicksort, the array is partitioned by placing all items smaller than some small **pivot item.** The pivot item can be any item.

After the partitioning, the order of the items in the subarrays is unspecified and is a result of how the partitioning is implemented. It is important that items smaller than the pivot item are to the left of the item, and all items larger than the pivot item are to the right of it. Quicksort is then called recursively to sort each of the two subarrays. They are partitioned, and the procedure is continued until an array with one item is reached. This array is trivially sorted.

Procedure partition works by checking each item in the array in sequence. When an item is found to be less than the pivot item, it is moved to the left side of the array.

In the good situation performance of quick sort we will get O(nlogn). The good performance comes if the pivot we are choosing comes at the median. But we will face the worst case of quick sort if the pivot comes as either the smallest element or the biggest one where we will get $O(n^2)$

Quick Check

1. The recursive version of Binary Search employs ____.
   Answer: tail-recursion.
2. Iterative algorithms execute faster than tail-recursion because no ___ needs to be maintained.
   Answer: stack
3. A sorting algorithm that does not use any extra space beyond that needed to store input is called a(n), ____.
   Answer: in-place sort
4. In quick sort all items smaller than the pivot are placed on the right of it (True or False).
   Answer: False
5. In quick sort the partitioning of the array is done by the procedure.
   Answer: partition

Strassen's Matrix Multiplication Algorithm

The best way to illustrate this method is through an example. Suppose we want to multiply two 2 * 2 matrices A and B and have their result in C as follows:

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Strassen determined some values:

$m1 = (a11 + a22)\ (b11 + b22)$
$m2 = (a21 + a22)\ b11$
$m3 = a11\ (b12 - b22)$
$m4 = a22\ (b21 - b11)$
$m5 = (a11 + a12)\ b22$
$m6 = (a21 - a11)\ (b11 + b12)$
$m7 = (a12 - a22)\ (b21 + b22)$

So the product C is given by

$$C = \begin{bmatrix} m1+m4-m5+m7 & m3+m5 \\ m2+m4 & m1+m3-m2+m6 \end{bmatrix}$$

To multiply two 2 X 2 matrices, Strassen's method requires seven multiplications and 18 additions/subtractions, whereas the straightforward method requires eight multiplication and four additions/subtractions. Because the commutativity of multiplications is not used in Strassen's formulas, those formulas pertain to larger matrices that are each divided into four submatrices.

Shmuel Winograd developed a variant of Strassen's algorithm that requires 15 additions/subtractions. Coppersmith and Winograd developed a matrix multiplication algorithm whose time complexity for the number of multiplications in in O. The constant is so large that Strassen;s algorithm is usually more efficient.

Other matrix operations such as inverting a matrix and finding the determinant of a matrix are directly related to matrix multiplication. We can readily create algorithms that are as efficient as Strassen's algorithm for matrix multiplication.

Arithmetic with Large Integers

Large integers are defined as integers whose size exceeds the computer's hardware capability of representing integers, we should use software to represent and manipulate these integers. We can accomplish this with the help of the divide-and-conquer approach.

Representation of large Integers: Addition and Other Linear Time Operations

A straightforward way to represent a large integer is to use an array of integers, in which each array slot stores one digit. To represent both positive and negative integers we need only reserve the high-order array slot for the sign. We could use 0 in that slot to represent a positive integer and 1 to represent a negative integer. It is not difficult to write linear-time algorithms for addition and subtraction, where *n* is the number of digits in the large integers. The basic operation consists of the manipulation of one decimal digit.

Multiplication of Large Integers

Here we will also use the divide and conquer algorithm to split an n-digit integer into two integers of approximately n/2 digits. So, if n is the number of digits in the integer *u*, we will split the integer into two integers, one with ceiling(*n*/2) and the other with floor(*n*/2):

$$u = x * 10^m + y$$

u has n digits
x has ceiling(n/2)
y has floor(n/2)
m = n/2

Suppose we have another n-digit integer $v = w * 10^m + z$ then:

$$u * v = xw * 10^{2m} + (xz + wy) * 10^m + yz$$

Recursively, these smaller integers can then be multiplied by dividing them into yet smaller integers.

Determining Thresholds

Recursion has a fair amount of overhead in terms of computer time. Suppose we have a sorting problem of size n we should develop a method that determines for what values of *n* it is at least as fast to call an alternative algorithm as it is to divide the instance further. These values depend on the divide-and-conquer algorithm, the alternative algorithm, and the computer on which they are implemented.

We should try to find an **optimal threshold value** (instance size) and for any instance size less than this value we should call another algorithm. An optimal threshold value does not always exist. We can still use the results of the analysis to pick a threshold value. We then modify the divide-and-conquer algorithm so that the instance is no longer divided once *n* reaches the threshold value. An alternative algorithm is called instead.

Quick Check

1. Strassen's method works best for 2 * 2 matrices (True or False).
   Answer: False
2. In Strassen's method, large matrices are usually divided into ____ sub-matrices.
   Answer: four
3. In the multiplication of large integers the division process is continued until ____ value is reached.

Answer: threshold

4. An optimal threshold value always exists (True or False).
   Answer: False

When not to use Divide-and-Conquer

Divide-and-conquer algorithms should be avoided in the following two cases:
1. An instance of size *n* is divided into two or more instances each almost of size *n*.
2. An instance of size *n* is divided into almost *n* instances of size *n/c*, where *c* is a constant.

Neither of these is acceptable for large values of *n.*

Classroom Discussion

- Is Mergesort a stable sort? Discuss
- What is the worst case and best case scenarios if we take the pivot in quick sort as the first element? Discuss

Homework

Assign Exercises 4, 16, 26, and 30.

Keywords

- **Divide-and conquer –** divide an instance of a problem into two or smaller instances.
- **Exchange Sort–**compares elements of an array and swaps those that are not in the proper positions.
- **In-place sort –** sorting algorithm that does not use any extra space beyond that needed to store the input.
- **Iteration–**the act of repeating a process with the aim of producing a desired goal or result.
- **Large-integer –** a defined data type to mean an array big enough to represent the integers in the application of interest.
- **Mergesort–**a comparison-based sorting algorithm. It divides the array into two halves, which are resorted recursively and then merged to create a sorted whole.
- **Optimal threshold value –** an instance size such that for any smaller instance it would be at least as fast to call the other algorithm as it would be to divide the instance further.
- **Procedure partition–**works by checking each item in the array. Items found to be less than the pivot item are moved to the left of the array, while items found to be higher than the pivot item are moved to the right of the array.
- **Quicksort–**a sorting technique that sequences a list by continually dividing the list and then moving the lower items to one side and the higher items to the other side.
- **Tail-recursion –** no operations are done after recursive call
- **Top-down –** to get a solution to a top-level instance of a problem go down and obtain solutions to smaller instances.
- **Two-way merging –** combining two sorted arrays into one sorted array

Important Links

- http://www.sciencenews.org/ (Science news online)
- http://www.codecodex.com/wiki/Merge_sort (Code Codex)

- http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/quickSort.htm (Quick Sort)