Divide-and-Conquer

1. Introduction

You are familiar with analysis of algorithms from previous CSCI courses, especially CSCI 36200. The idea is to evaluate the space and time efficiency of an algorithm, i.e., what its resource consumption is. The goal is to obtain some sort of expression for space or time required as a function of n, where n is a measure of the input size.

For space efficiency, only the most trivial of algorithms can execute without fully examining the input, so n is a floor on the memory space required. And there will generally be a few extra variables needed in order to carry out the algorithm. A broad categorization of space efficiency, then, is whether the space required is essentially n (n + a few extra memory locations), or whether it is some multiple of n (for example, an array that gets copied over and over as the algorithm executes).

We will primarily be interested in time efficiency. As you know, this does not mean running the algorithm on a real computer with stopwatch in hand. This would be too dependent on the speed of the processor, the programming language used, perhaps the skill of the programmer, etc. (Although the stopwatch approach is used in benchmarking to compare, say, the same algorithm on the same input data using two different processors.) Instead we identify one or more "units of work", tasks that seem essential to the operation of the algorithm, and try to determine how many times that task(s) is done. This often varies with the input data, so an attempt can be made to answer "what's the worst case? the average case? the best case?". Sometimes these answers are all the same. If they differ, the best case is no doubt over-optimistic – how often will the conditions for the "best case" appear? Similarly, the worst case may be over-pessimistic, but at least you know the upper bound. The average case analysis is often difficult for two reasons: 1) what constitutes "average input" can be hard to define and 2) the mathematics involved in an average-case analysis is usually more complex than a worst-case analysis.

How does one identify the "unit of work"? While this may depend on the algorithm, it is more often associated with the problem itself. For example, if the problem is to search a list of values for a target value, it is hard to see how this could be solved without comparing the target value against (possibly) multiple values in the list, so this comparison would be the work unit. (However, there is an exception even to this seemingly obvious result. In the very special case where the input values are monotonically increasing at a uniform rate, one could do an "interpolation" to see where the value would fall if it is in the list and check that spot.)

2. Example 1 – Iterative Selection Sort

The problem is to sort a list of distinct items that support an ordering relation into increasing sorted order. Sorting is also a problem that seems by its nature to indicate the work

units. In order to put the items in sorted order, they will have to be compared against each other and possibly moved around. A description of the iterative selection sort algorithm follows[1]:

```
SelectionSort(list L: int n)
//L is the list of size n, indexed 1 through n, to be sorted in increasing
//order

//Post: The entries of list L have been rearranged so that the items are
//sorted into increasing order.
{
   for (int position = n; position > 1; position--)
  {
      int max = max_location(L, position);
      swap(L[max], L[position]);
   }
}
```

The idea behind the algorithm is to find the maximum element in the unsorted section of the list and swap it to the back of the unsorted section, then reduce the size of the unsorted section. The sorted part of the list is created from the back to the front of the list. There are two functions used. The *swap* function works as follows:

```
swap(x,y)
{
   temp = x;
   x = y;
   y = temp;
}
```

so that each invocation of the swap function requires 3 assignment statements. Here's where the "moving around" among items in the list takes place. The *max_location* function finds the index of the maximum item in the unsorted part of the list, which is between 1 and *position*.

```
max_location(list L, int position)
{
   temp_max = 1;
   for (int i = 2; i <= position; i++)
   {
      if (L[temp_max] < L[i]
         temp_max = i;
   }
   return temp_max;
}
```

Within *max_location*, comparison of list elements occurs. Assignments might also occur to update the value of *temp_max*, but these are assignments between index counters (simple integers), not between list elements, which could possibly be lengthy records, so these assignments won't be counted as work units. Whenever *max_location* is invoked, *position* -1 comparisons are done within the for loop of *max_location*. Within the for loop of selection sort,

[1] Algorithm "descriptions" will be fairly detailed, but not compiler-ready code.

position goes from n down to 1. Therefore the number of comparisons done in running the selection sort algorithm is

$(n - 1) + (n - 2) + \ldots + 1$

and a simple induction proof shows that this sum is equal to $n(n-1)/2 = (n^2 - n)/2$. Each invocation of the selection sort for loop invokes the swap function, so $3(n - 1)$ assignments are done.

Summary: selection sort on an n-element list requires $\Theta(n^2)$ comparisons and $\Theta(n)$ assignments. One of the interesting features of selection sort is that it does the same amount of work regardless of the state of the input data (already sorted, reverse sorted, completely random, etc.)

3. Divide and Conquer

The first algorithm design technique we will study is the divide-and-conquer approach, probably already familiar to you. In this approach, the problem to be solved is divided into smaller and smaller pieces until a trivial, easily-solved base case is encountered. Then the solutions to the various sub-problems are knitted back together to eventually create a solution to the original problem as a whole.

This sounds suspiciously like the way recursion works, and the divide-and-conquer approach invariably leads to a recursive algorithm. The analysis of a recursive algorithm often involves solution of a recurrence relation. You should recall from CSCI 34000 that there are two common forms of recurrence relations for which a solution formula is known.

(i) Linear first-order recurrence relation with constant coefficients:

$S(n) = cS(n - 1) + g(n)$ for $n \geq 2$

The solution is

$$S(n) = c^{n-1}S(1) + \sum_{i=2}^{n} c^{n-i} g(i)$$

(ii) Divide-and-conquer recurrence relation:

$S(n) = cS\left(\frac{n}{2}\right) + g(n)$ for $n \geq 2$, $n = 2^m$

The solution is

$$S(n) = c^{\lg n}S(1) + \sum_{i=1}^{\lg n} c^{(\lg n)-i} g(2^i)$$

In both case (i) and case (ii), the constant coefficient "c" represents the number of subproblems the algorithm has to solve for each round of recursion while the function g(n) counts the additional operations required for each round of recursion.

For case (ii), the assumption that n is a power of 2 is necessary in that the notation S(n/2) makes no sense if n/2 is not an integer. In implementation, though, if n is not a power of 2, most programming languages use integer division (floor function) so that n/2 results in the integer $\lfloor n/2 \rfloor$. This small variation does not, in general, affect the order of magnitude of the analysis, so one can use the above solution formula whether or not n is a power of 2.

The solution formulas each contain a summation expression. Because we want a closed-form functional expression for the work required as a function of n, we have to be able to find an expression for the value of the summation; otherwise we are no better off than before. Fortunately, many such summations have a closed-form expression for their value that can be proved by induction.

4. Recursive Selection Sort

There is a recursive version of Selection Sort. Here's what it looks like:

```
RecursiveSelectionSort(list L; int j)
//recursively sorts the items from 1 to j in list L
//into increasing order
{
   if (j == 1) then
      sort is complete, write out the sorted list
   else
   {
      int max = max_location(L, j);
      swap(L[max], L[j]);
      RecursiveSelectionSort(L, j - 1)
   }
}
```

This function is initially invoked with $j = n$.

We can write a recurrence relation for the number of comparisons this algorithm performs. The trivial base case consists of a 1-element list. Using the notation C(n) for the number of comparisons to sort an n-element list, we know that

$C(1) = 0$	//no comparisons required to sort a 1-element list
$C(n) = (n - 1) + C(n - 1), n \geq 2$	//n-1 comparisons to find max_location for an //n-element list plus however much work to sort the //(n-1) element list

This is a linear first-order recurrence relation with constant coefficients, matching form (i) above where c = 1 and g(n) = n – 1. (Each round of recursion requires solving one subproblem plus an additional n – 1 operations.) From the solution formula,

$$C(n) = 1^{n-1}(0) + \sum_{i=2}^{n}(1)^{n-i}(i-1) = 0 + \sum_{i=2}^{n}(i-1) = 1 + 2 + \ldots + (n-1) = \frac{(n-1)n}{2}$$

So the number of comparisons is $\Theta(n^2)$, just as before. Also, just as before, the swap function is invoked n – 1 times, giving $\Theta(n)$ assignments.

Well, no improvement here – what happened? Although we have a recursive selection sort algorithm, it does not really incorporate the divide-and-conquer principle to any effect. Divide-and-conquer works only if you can make the smaller pieces of the problem **significantly** smaller. In the case of recursive selection sort, each new invocation of the algorithm acts on a list only one element smaller than the previous invocation. Not much progress is being made.

5. A Better Divide-and-Conquer Algorithm - Mergesort

The problem is the same as before – sorting a list L in increasing order. Mergesort cuts the list in half, recursively sorts each half, and then knits the two sorted halves back together. The recursion stops when the list has been reduced to size 1. The body of the recursive Mergesort function looks like this, and is initially invoked with low = 1 and high = n.

```
Mergesort(list L, int low, int high)
{
   if (low < high)
  { // Otherwise, no sorting is needed.
       mid = (low + high) / 2;
       mergesort(L, low, mid);
       mergesort(L, mid + 1, high)
       merge(L, low, high);
   }
}
```

We want an expression for the number of comparisons required for Mergesort to sort an n-element array. We can represent this symbolically by C(n). Comparisons are done entirely in the merge function. By the time you get to the merge function, you have an array A with two sorted halves A1 and A2. You want to build a merged, sorted array out of A1 and A2 by comparing the next two elements of each half and putting the smaller one into its proper position in the growing final array. The only problem is, you can't just put this element back into A or you've potentially written over data you need later. The simplest way to avoid this problem is to merge A1 and A2 into a new array B, then copy the results back into A. This makes Mergesort a space-inefficient algorithm, doubling the array size required.

In the final merge, each comparison results in copying one value from one of the two sorted halves A1 and A2 into the new sorted array B being created. The number of comparisons is therefore bounded above by the sum of the lengths of the two halves, which is n. (The number of comparisons is actually less then this because once one of the halves runs out of elements, no more comparisons are required.) As an upper bound, then, we can write the number of comparisons by the following recurrence relation:

C(1) = 0 //no work to sort a 1 element array

C(n) = 2*C(n/2) + n, n ≥ 2 //the number of compares to sort an n-element array is the n
//[at most] compares needed for the final merge, plus 2 times
// however many compares it takes to sort each half

This is a divide-and-conquer recurrence relation, matching form (ii) above where c = 2 and g(n) = n. So the solution is

$$C(n) = 2^{\lg n}C(1) + \sum_{i=1}^{\lg n} 2^{\lg n - i}2^i = 0 + \sum_{i=1}^{\lg n} 2^{\lg n - i}\,2^i = \sum_{i=1}^{\lg n} 2^{\lg n} = \sum_{i=1}^{\lg n} n = n \lg n$$

Again, this is a worst-case analysis, but is nonetheless a big improvement over Selection Sort. That's because Mergesort is an effective divide-and-conquer algorithm – the sublists are half the size of the original list, so a substantial reduction has occurred in the size of the problem to be solved.

6. Example 2 – Matrix Multiplication

Consider two n × n matrices A and B. Then A · B exists and is an n × n matrix. The standard matrix multiplication algorithm looks like this (elements in row i of A are multiplied by the corresponding elements in column j of B and the results added together to form $C_{i,j}$):

```
MatrixMultiplication (A, B)
//computes n × n matrix A · B for n × n matrix A, n × n matrix B
//stores result in C
{
   for i = 1 to n do
       for j = 1 to n do
            C[i, j] = 0
            for k = 1 to n do
                   C[i, j] = C[i, j] + A[i, k] * B[k, j]
            end for
        end for
   end for
   write out product matrix C
}
```

The work going on in this matrix multiplication is a series of multiplications and additions. One multiplication and one addition occur for each pass through the innermost for loop. It's easy to see from the three nested for loops that n^3 multiplications and n^3 additions are required. (The above makes for tidy code, but one could reduce the number of additions to $n^2(n - 1)$ by making k = 1 a special case where C[i, j] = A[i, 1] * B[1, j], however the result is still $\Theta(n^3)$).

7. Divide and Conquer – Strassen's Algorithm

We'll look at a different algorithm for matrix multiplication that may be a less familiar example of the divide-and-conquer approach. In applying Strassen's algorithm we assume that $n = 2^m$ for some $m \geq 0$. (Here n really does need to be a power of 2 because we are going to split the "physical" $n \times n$ matrix into four $n/2 \times n/2$ matrices. If n is not a power of 2, extra rows and columns of 0's can be added to "pad" the matrices to make the dimension a power of 2 – of course, this will increase the work involved. Because of padding, Strassen's algorithm actually applies to matrices of any compatible size, not just square matrices. Thus $A \cdot B$ can be computed if A is $n \times m$ and B is $m \times p$.)

First consider a simple case of multiplying two 2×2 matrices.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

The product

$$\mathbf{C} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix} \qquad (1)$$

can also be written (check the calculations) as

$$\mathbf{C} = \begin{bmatrix} p_1 + p_4 - p_5 + p_7 & p_3 + p_5 \\ p_2 + p_4 & p_1 + p_3 - p_2 + p_6 \end{bmatrix} \qquad (2)$$

where

$$p_1 = (a_{11} + a_{22})(b_{11} + b_{22}) \qquad p_2 = (a_{21} + a_{22})b_{11}$$
$$p_3 = a_{11}(b_{12} - b_{22}) \qquad p_4 = a_{22}(b_{21} - b_{11})$$
$$p_5 = (a_{11} + a_{12})b_{22} \qquad p_6 = (a_{21} - a_{11})(b_{11} + b_{12})$$
$$p_7 = (a_{12} - a_{22})(b_{21} + b_{22})$$

Computing the various p_i quantities requires 7 multiplications and 10 additions (counting subtractions as additions). Computing the product **C** once the p_i quantities exist takes 0 multiplications and an additional 8 additions. In total, computing **C** by this method requires 7 multiplications and 18 additions. Now take two $n \times n$ matrices **A** and **B** and partition each of them into four $n/2 \times n/2$ matrices:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix}$$

The product $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ is still given by Equation (1) using A_{ij} and B_{ij} instead of a_{ij} and b_{ij}, respectively. Therefore Equation (2) still holds, and the product **C** requires 7 (n/2 × n/2) matrix multiplications. (Matrix multiplication is not commutative, but the p_i expressions used in Equation 2 do not make any use of commutativity of multiplication.) This is an example of a divide and conquer algorithm where the work has been reduced to several instances of the same problem on a significantly reduced input size. (Although there is an additional overhead of 18 (n/2 × n/2) matrix additions.)

Let M(n) represent the number of multiplications required for a product of two n × n matrices. Using Strassen's algorithm, we can write

$M(1) = 1$ //one multiplication in the product of two 1 × 1 matrices

$M(n) = 7M\left(\frac{n}{2}\right)$ for $n \geq 2$ //7 times whatever is required to multiply (n/2 × n/2) matrices

This is a divide-and-conquer recurrence relation of the form (ii) above where c = 7 and g(n) = 0. The solution is

$$M(n) = c^{\lg n} M(1) + \sum_{i=1}^{\lg n} c^{(\lg n)-i} g(2^i) = 7^{\lg n}(1) + \sum_{i=1}^{\lg n} 7^{(\lg n)-i} 0 = 7^{\lg n}.$$

As it turns out,

$7^{\lg n} = n^{\lg 7}$

(take the lg of both sides, giving lg n * lg 7 = lg 7 * lg n, which is true). So M(n) is $7^{\lg n} = n^{\lg 7} \cong n^{2.8}$ (more accurately, $n^{2.807355}$).

Now let's consider additions and let A(n) represent the number of additions required for a product of two n × n matrices. Then

$A(1) = 0$ // no additions required to multiply two 1 × 1 matrices

$A(n) = 7A\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2$ for $n \geq 2$ //7 multiplications of (n/2 × n/2) matrices, each of

// which requires $A\left(\frac{n}{2}\right)$ additions. The product

// also requires 18 additions of (n/2 × n/2) matrices,
//each of which requires $(n/2)^2$ additions.

This is also a divide-and-conquer recurrence relation of the form (ii) above where c = 7 and $g(n) = 18\left(\frac{n}{2}\right)^2$. The solution is

$$A(n) = 7^{\lg n}(0) + \sum_{i=1}^{\lg n} 7^{\lg n - i} 18 \left(\frac{2^i}{2}\right)^2 = 7^{\lg n} \frac{18}{4} \sum_{i=1}^{\lg n} 7^{-i} 2^{2i} = 7^{\lg n} \frac{9}{2} \sum_{i=1}^{\lg n} \frac{(2^2)^i}{7^i} = 7^{\lg n} \frac{9}{2} \sum_{i=1}^{\lg n} \left(\frac{4}{7}\right)^i =$$

$$7^{\lg n} \frac{9}{2} \cdot \frac{4}{7} \left[1 + \frac{4}{7} + \left(\frac{4}{7}\right)^2 + \cdots + \left(\frac{4}{7}\right)^{\lg n - 1}\right] = \text{(using formula for the sum of terms of a}$$

$$\text{geometric sequence) } 7^{\lg n} \frac{9}{2} \cdot \frac{4}{7} \cdot \frac{7}{3} \left[1 - \left(\frac{4}{7}\right)^{\lg n}\right] = 7^{\lg n}(6)\left[1 - \frac{4^{\lg n}}{7^{\lg n}}\right] = 6 * 7^{\lg n} - 6 * 4^{\lg n} =$$

$$6n^{\lg 7} - 6n^{\lg 4} = 6n^{\lg 7} - 6n^2 \cong 6n^{2.8} - 6n^2$$

Adding the results for both multiplications and additions, we get $\Theta(n^{\lg 7}) + \Theta(n^{\lg 7}) = \Theta(n^{\lg 7}) \cong \Theta(n^{2.8})$, which is better than the $\Theta(n^3)$ of the traditional algorithm. So there is an improvement in the overall order of magnitude. However, we found the actual expressions (or close approximations), and because of the 6 coefficient in the number of additions, Strassen's algorithm isn't really an improvement until n becomes quite large. How large is a matter also of the actual implementation, any use of parallelism, etc., so there is no one clear "break-even" point.

Since Strassen's algorithm, there have been other improvements on the order of magnitude for $n \times n$ matrix multiplication:

Strassen (1969): $n^{2.807355}$
Coopersmith-Winograd (1990): $n^{2.375477}$
Stothers (2010): $n^{2.373}$
Williams (2011): $n^{2.3728642}$
LeGall (2014): $n^{2.3728639}$

But these other algorithms are seldom used because the improvement is not worth the effort unless n is so large as to be beyond current hardware capabilities; "astronomically large", as one writer put it.