1 Divide and Conquer Algorithms

One way to create fast algorithms is to devise a divide-and-conquer algorithm for a problem. Divide-and-conquer algorithms are naturally recursive algorithms and recurrence relations are the natural too needed to determine their cost. Three important divide and counquer algorithms are Mergesort, Quicksort and the FFT (Fast Fourier Transform). Let us give a simple example first.

Suppose we are given an array A of n numbers and we want to add them. For example

$$A = \boxed{1}\boxed{3}\boxed{5}\boxed{7}\boxed{9}\boxed{4}$$

The simplest way to add them is to use a loop. The following C code does this. The type `double` is for variables whose values are decimal numbers like 3.145159265.

```
1: double Add( double A[], int n ) {
2: // Add A[0]+A[1]+...+A[n-1]
3:    int i;
4:    double s = 0.0;
6:    for( i=0; i<n; i++ )
7:        s = s + A[i];
8:    return s;
9: }
```

How many additions of decimal numbers does the Add algorithm do? The answer is n because it adds decimal numbers in line 7 and line 7 is executed n times.

Another way to add the numbers in an array is to use the following divide-and-conquer algorithm. The idea is to divide the array A into halves, add the numbers of the first half to get s_1, then add the entries of the second half to get s_2, then return $s_1 + s_2$. In our example we have

$$\underset{s_1 = 9}{\boxed{1}\boxed{3}\boxed{5}} \qquad \underset{s_2 = 20}{\boxed{7}\boxed{9}\boxed{6}}$$

One reason we might want to add this way is so we can assign one processor to calculate s_1 and a second processor to calculate s_2 to speed up the computation.

The idea of a divide-and-conquer algorithm is to apply the same strategy recursively to the two halves of A. Here is C code to do this. The code assumes $n \geq 1$.

```
 1: double Add( double A[], int n ) {
 2: // Add A[0]+A[1]+...+A[n-1]
 3:    double s1,s2,*B;  int n1,n2;
 4:    if( n==1 ) return A[0];
 5:    n1 = n/2; n2 = n-n1;
 6:    s1 = Add(A,n1); // s1 = A[0]+A[1]+...+A[n1-1]
 7:    B = A + n1; // B is a subarray of A starting at n1
 8:    s2 = Add(B,n2); // s2 = A[n1]+A[n2+1]+...+A[n-1]
 9:    return s1+s2;
10: }
```

Line 7 makes B a subarray of A that starts at index n_1 of A. This means `B[0]` gets the value in `A[n1]`, and `B[1]` gets the value in `A[n1+1]`, etc. You may picture the arrays A and B as follows. where we have included subscripts for both arrays.

$$\begin{array}{ccc|ccc} A_0 & A_1 & A_2 & A_3 & A_4 & A_5 \\ \hline 1 & 3 & 5 & 7 & 9 & 4 \\ \hline B_{-3} & B_{-2} & B_{-1} & B_0 & B_1 & B_2 \end{array}$$

Let $T(n)$ be the number of additions algorithm Add does in line 9. We have $T(1) = 0$ from line 4 and $T(n) = T(n_1) + T(n_2) + 1$ from lines 6,8 and 9. If n is even then $n_1 = n_2 = n/2$ and the recurrence simplifies to

$$T(n) = 2T(n/2) + 1.$$

We will solve this recurrence later and show that $T(n) = n - 1$ additions. But first, let us give a general strategy for divide-and-conquer algorithms. If we are given a problem of size $n > 1$, the general strategy is to

Step 1: Divide the problem into $a \geq 2$ subproblems of approximately the same size, say size b. Algorithm Add divided A into $a = 2$ subproblems of size $n_1 = n/2$ and $n_2 = n - n_1$.

Step 2: Solve the subproblems recursively using the same "divide-and-conquer" approach. Algorithm Add does this in lines 6 and 8 obtaining solutions s_1 and s_2.

Step 3: Combine the results from the subproblems to obtain the final solution. Algorithm Add does this when it computes `s1 + s2` in line 9.

Let $f(n)$ be the number of operations to solve the problem using the divide-and-conquer algorithm and we let $h(n)$ be the number of operations in steps 1 and 3. To make our analysis easier, we will assume $n = b^k$ which means we can always divide n exactly into b sub-problems of size n/b. Then we have a recurrence for $f(n)$ of the form

$$f(n) = af(n/b) + h(n) \quad \text{for } n > 1. \tag{1}$$

For $n = 1$ we may assume the time to solve the problem is a constant $c \geq 0$ so that $f(1) = c$. In our Add example we have $a = 2$, $b = 2$, $c = 0$ and $h(n) = 1$ so that the recurrence is $f(n) = 2f(n/2)+1$. Theorem 10.1 below solves recurrence (1) for the case $h(n) = d$ where d is a constant. Our Theorem 10.1 generalizes slightly Theorem 10.1 in the Grimaldi text.

Theorem 10.1 Let a and b be positive integers with $b \geq 2$. Let c and d be positive real numbers. Let $n = b^k$ for $k \geq 0$ and let $f : \mathbb{Z}^+ \to \mathbb{R}$. If $f(1) = c$ and $f(n) = af(n/b) + d$ for $n > 1$ then

(1) $f(n) = d \log_b n + c$ for $a = 1$.

(2) $f(n) = d\dfrac{an^{\log_b a} - 1}{a - 1} + cn^{\log_b a}$ for $a \geq 2$.

(3) $f(n) = \frac{n-1}{a-1}d + nc$ for $a \geq 2$ and $b = a$.

Proof: For $k \geq 1$ we expand the recurrence to obtain the following equations

$$\begin{aligned}
f(n) &= af(n/b) + d \\
af(n/b) &= a(af(n/b^2) + d) = a^2 f(n/b^2) + ad \\
a^2 f(n/b^2) &= a^2(af(n/b^3) + d) = a^3 f(n/b^3) + a^2 d \\
\vdots &\leq \vdots \\
a^{k-2} f(n/b^{k-2}) &= a^{k-1} f(n/b^{k-2}) + a^{k-2} d \\
a^{k-1} f(n/b^{k-1}) &= a^k f(n/b^k) + a^{k-1} d = a^k f(1) + a^{k-1} d \\
a^k f(1) &= a^k c
\end{aligned}$$

Adding these equations and cancelling equal terms we have

$$\begin{aligned} f(n) &= a^k f(1) + (d + ad + a^2 d + \ldots + a^{k-1} d) \\ &= a^k c + (1 + a + a^2 + \ldots + a^{k-1}) d. \end{aligned}$$

Using $n = b^k$ so that $k = \log_b n$, for $a = 1$ we obtain (1) $f(n) = c + kd = c + d \log_b n$. To show (2) we use $\sum_{i=0}^{k-1} a^i = \frac{a^k - 1}{a-1}$ and $a^k = a^{\log_b n} = n^{\log_b a}$ so that

$$f(n) = a^k c + \frac{a^k - 1}{a-1} d = c n^{\log_b a} + d \frac{n^{\log_b a} - 1}{a-1}.$$

For case (3) where $a = b$ we have $\log_b a = 1$ so $f(n) = cn + \frac{n-1}{a-1} d$. This ends the proof.

Returning to our example of adding an array of n numbers where we had $T(n) = 2T(n/2) + 1$ and $T(1) = 0$. Applying Theorem 10.1 (3) with $c = 0$, $d = 1$, and $a = b = 2$ we obtain

$$T(n) = cn + \frac{n-1}{a-1} d = n - 1.$$

The choice $h(n) = d$ in Theorem 10.1 is too restrictive in general. For the Mergesort algorithm we will need to solve the recurrence $T(n) = 2T(n/2) + n - 1$ with $T(1) = 0$. In the exercises we ask you to solve the recurrence $T(n) = aT(n/b) + h(n)$ for $h(n) = An + B$ because this recurrence occurs in many important divide-and-conquer algorithms.

2 Mergesort

Suppose we are given an array A of n objects, perhaps integers, and suppose we want to sort them. Previously we saw that the Bubble sort algorithm does $n(n-1)/2$ comparisons. The Mergesort algorithm is much faster. We will show that for $n = 2^k$ for $k \geq 0$, it does at most $n \log_2 n - n + 1$ comparisons. Below is a table comparing the number of comparisons of the two algorithms.

	n	4	16	64	1024	10^6
Bubblesort	$n(n-1)/2$	6	120	2016	523776	approx 5×10^{11}
Mergesort	$n \log_2 n - n + 1$	5	49	321	9217	approx 20×10^6

For an array with $n = 1024$ entries, Bubblesort does a factor of 523776/9217=56.8 times as many comparisons as Mergesort. The power of Mergesort becomes apparent for larger n. For $n = 10^6$ Mergesort does a factor of over $25,000$ fewer comparisons! So how does the Merge sort algorithm work? Consider the array A below with $n = 7$ elements.

$A =$

17	3	12	5	1	14	10

The main idea of Mergesort is to split the array into two halves and sort them recursively. Since the size n may not be even we will use $n_1 = \lfloor n/2 \rfloor = 3$ for the first array and $n_2 = n - n_1 = 4$ for the second. So we have

$A =$

17	3	12

5	1	14	10

The next step is to sort the two sub-arrays recursively. That is, we use the Mergesort algorithm on the first n_1 elements and then the Mergesort algorithm on the remaining n_2 elements. After this is done we will have

$A =$ | 3 | 12 | 17 | | 1 | 5 | 10 | 14 |

The final step is to "merge" the two sorted arrays into a new array C of size n then copy the contents of C back into A. We give the code for Mergesort below where we have not yet defined the Merge step. In the code line 8 defines B to be the subarray of A starting at index n_1.

```
 1: void Mergesort( int A[], int n, int C[] ) {
 2: // sort A[0],A[1],...,A[n] into ascending order
 3: // C is an array of length n for working storage
 4:    int n1,n2,*B;
 5:    if( n<=1 ) return;
 6:    n1 = n/2;
 7:    n2 = n-n1;
 8:    B = A + n1;
 9:    Mergesort(A,n1,C); // sort the first half of A
10:    Mergesort(B,n2,C); // sort the second half of A
11:    Merge(A,n1,B,n2,C); // merge A and B into C
12:    for( i=0; i<n; i++ ) A[i] = C[i];
13:    return;
14: }
```

Figure 1: C code for the Mergesort algorithm

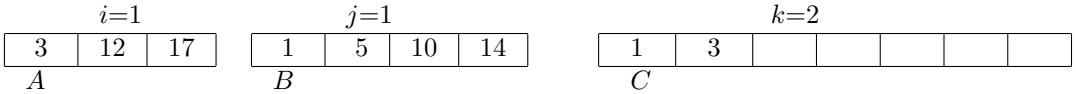
Let $C(n)$ be the number of comparisons that Mergesort does and let $M(n1, n2)$ be the number of comparisons that the, as yet, unspecified Merge algorithm does in line 11. Line 5 means $C(1) = 0$. Because of the two recursive calls in lines 9 and 10 we have the recurrence

$$C(n) = C(n_1) + C(n_2) + M(n_1, n_2). \tag{2}$$

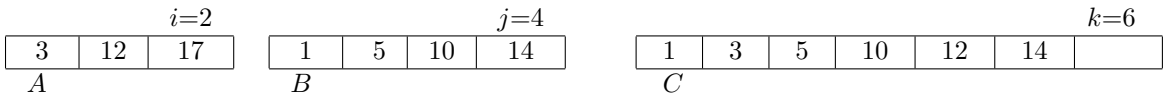
Now we need to explain how the Merge step works. To merge the two halves of A, we maintain three array indexes i, j, k. In the merge the index i tells us which element of A we are at, the index j tells us which element of the array B we are at, and k tells us the next empty slot in the array C. Schematically, at the start of the merge, we have the following picture

Now the merge begins. Since the first half of A and the second half of A are already sorted, the smallest element in A must be either A_0 or B_0. In our example, $B_0 = 1$ is the smallest element. So we copy B_0 into C_0 and add 1 to i and k so that we have the following picture.

Now we compare A_i and B_j again. Since $i = 1$ and $j = 0$ we compare `A[i]=3` with `B[j]=5`. As the smaller is 3 we copy 3 into C_k and add 1 to i and k. Now we have this picture



Repeating this we will stop the merging process when $i = n_1$ or $j = n_2$, whichever occurs first. In our example since 17 is the largest element we will reach $j = n_2 = 4$ first. And we will have



The final step is to copy any remaining elements from A or B into C. In our example, we have $i = 2$ so we copy A_2 into C_k. C code for doing a merge is presented below.

```
 1: void Merge( int * A, int n1, int * B, int n2, int *C ) {
 2: // Merge the sorted arrays A of length n1 and B of length n2 into C
 3:    int i,j,k;
 4:    i = j = k = 0;
 5:    while( i<n1 && j<n2 )
 6:       if( A[i]<B[j] ) { C[k] = A[i]; i++; k++; }
 7:       else { C[k] = B[j]; j++; k++; }
 8:    while( i<n1 ) { C[k] = A[i]; i++; k++; }
 9:    while( j<n2 ) { C[k] = B[j]; j++; k++; }
10:    return;
11: }
```

Figure 2: C code for merging two sorted arrays A and B into the array C

Our next task is to determine how many comparisons between elements of the arrays A and B the Merge algorithm does. Let $M(n_1, n_2)$ be the number of comparisons. We make the following observation. Each time the loop in lines 5–7 is executed, one element from either A or B is moved to C and one comparison is done in line 6. Since there are n_1 elements in A, and n_2 elements in B, at most $n_1 + n_2 - 1$ are moved to C, so at most $n_1 + n_2 - 1$ comparisons are done. Thus

$$M(n_1, n_2) \leq n_1 + n_2 - 1.$$

There can be fewer comparisons. For example, if all elements in A are less then B_0, then only n_1 comparisons would be done.

Now we complete the analysis of the Mergesort algorithm where $C(n)$ is the total number of comparisons. From equation (2), using $n_1 + n_2 = n$ we have

$$\begin{aligned} C(n) &= C(n_1) + C(n_2) + M(n_1, n_2) \\ &\leq C(n_1) + C(n_2) + n_1 + n_2 - 1 \\ &= C(m) + C(n - m) + (n - 1). \end{aligned}$$

Substituting $n_1 = \lfloor n/2 \rfloor$ we have

$$C(n) \leq C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil)) + (n - 1).$$

To simplify the analysis we assume $n = 2^k$ for some $k \geq 0$ so that $k = \log_2 n$ and $n_1 = n_2 = n/2$ and

$$C(n) \leq 2\,C(n/2) + n - 1.$$

Now we solve the recurrence with $C(1) = 0$. We have

$$\begin{aligned} C(n) &\leq 2\,C(n/2) + n - 1 \\ 2\,C(n/2) &\leq 2[2\,C(n/4) + (n/2 - 1)] = 4\,C(n/4) + n - 2 \\ 4\,C(n/4) &\leq 4[2\,C(n/8) + (n/4 - 1)] = 8\,C(n/8) + n - 4 \end{aligned}$$

$$\begin{aligned} \ldots &\leq \ldots \\ \frac{n}{4}C(4) &\leq \frac{n}{4}[2\,C(2)+(4-1)] = \frac{n}{2}\,C(2)+n-\frac{n}{4} \\ \frac{n}{2}C(2) &\leq \frac{n}{2}\,[2\,C(1)+(2-1)] = n\,C(1)+n-\frac{n}{2} \\ n\,C(1) &= 0 \end{aligned}$$

Adding these inequalities and cancelling equal terms we have

$$\begin{aligned} C(n) &\leq (n-1)+(n-2)+(n-4)+\ldots+(n-n/4)+(n-n/2)+0 \\ &= \sum_{i=0}^{k-1}(n-2^i) \\ &= \sum_{i=0}^{k-1} n - \sum_{i=0}^{k-1} 2^i \\ &= nk-(2^k-1) \\ &= n\log_2 n - n + 1 \end{aligned}$$

We note that this is the maximum number of comparisons that the Mergesort algorithm does. In the exercises you will consider the minimum number of comparisons which happens when the array A is already sorted. You will also consider another divide and conquer algorithm, the Fast Fourier Transform. Many consider the Faster Fourier Transform the most important algorithm discovered in the 20^{th} century. We have shown that for $n = 2^k$ the Mergesort algorithm does at most $n\log_2 n - n + 1$ comparisons.

Exercises

1 Consider the recurrence $T(n) = aT(n/b) + h(n)$ for $h(n) = An + B$ with initial value $T(1) = C$ where a, b are positive integers with $b \geq 2$ and A, B, C are real numbers with $A \geq 0$ and $C \geq 0$. Using the method in the proof of Theorem 10.1 solve for $T(n)$. Simplify the formula for the case $a = b = 2$.

2 Below is C code for the Fast Fourier Transform for the integers modulo p. The input is the array A of long integers and the output is the array B of long integers. Note, you do not need to know what algorithm FFT is doing in order to count the number of operations that it does. If you are interested, the FFT algorithm and applications of it will be presented in detail in MACM 401.

```
 1: void FFT( long A[], long n, long w, B[], long p ) {
 2: // assumes 0 <= A[i] < p and B is an array of size n and n = 2^k
 3:    long i,n2,w2,wi,s,t;
 4:    if( n==1 ) return;
 5:    n2 = n/2;
 6:    for( i=0; i<n2; i++ ) { B[i] = A[2*i]; B[n2+i] = A[2*i+1]; }
 7:    w2 = w*w % p;
 8:    FFT( B, n2, w2, A, p );
 9:    FFT( B+n2, n2, w2, A+n2, p );
10:    wi = 1;
11:    for( i=0; i<n2; i++ ) {
12:       s = B[i]; t = w*B[n2+i] % p;
13:       A[i] = (s+t) % p; A[n2+i] = (s-t) % p;
14:       wi = w*wi % p;
15:    }
16:    return;
17: }
```

Let $T(n)$ be the number of multiplications that algorithm FFT does in lines 7, 12 and 14. Give a recurrence equation for $T(n)$ and an initial value for $T(1)$ then solve for $T(n)$. You should get an answer of the form $T(n) = An\log_2 n + Bn + C$ for some constants A, B, C.

3 Recall that the recurrence for number of comparisons that the Mergesort algorithm does is $C(n) \leq 2\,C(n/2) + n - 1$ and $C(1) = 0$. If the input array A is already sorted, give a recurrence for the number of comparisons Mergesort does and solve it.

4 Consider the recurrence $f(n) = 2f(n/2) + h(n)$ with $f(1) = c$ and $h(1) > 0$. Suppose, for n even, we know that $h(n) \geq 2h(n/2)$. Here $h(n)$ represents the cost of solving some problem of size n. The assumption $h(n) \geq 2h(n/2)$ means solving a problem of size n costs at least as much as solving two problems of half the size. This is true of sorting for example. Use this to show that $f(n) \leq h(n)\log_2 n + cn$.