Divide-and-Conquer Algorithms - Mathematical Model

Divide-and-conquer algorithms are usually recursive in structure. They call themselves recursively with the smaller in size original problems. Steps at each level of recursion

- Divide the problem into the a number of subproblems
- Conquer the subproblems
- Combine the subproblems solutions to get the solution of the original problem

We will consider divided and conquer algorithms represented by mathematical equations called also recurrence relations:

$T(n) = aT(n/b) + D(n) + C(n)$ if $n > c$
$T(n) = \Theta(1)$ if $n \leq c$

a is the number of subproblems
n/b is the size of the subproblem and $b > 1$
$D(n)$ is the time to divide the subproblems
$C(n)$ is the time to combine the solution of subproblems into solution of the original problem.

Solving Recurrences

There are three methods for obtaining asymptotic solutions of recurrences:

- guessing a form of the solution and then using mathematical induction to verify solution.
- iterating a recurrence (may need a lot of algebra, symbolic tools like Maple or Mathematica may be handy). The recurrence is expanded and expressed as a summation. Techniques for evaluating summations can then be used to provide bound on the solution.
- using a "cookbook" method (the so called master method). Solutions of three main cases for recurrence equations are provided.

Recurrence Relation for Merge Sort

Merge sort uses the divide-and-conquer approach.

Divide: Divide the n-element input sequence to be sorted into two subsequences of $n/2$ elements each.

Conquer: Sort the two subsequences recursively using the merge sort.

Combine: Merge the two sorted subsequences to produce the sorted sequence.

In merge sort we have: $a = 2$, $b = 2$, $D(n) + C(n) = \Theta(n)$, where $C(n) = \Theta(n)$ (merging two $n/2$ element subarrays) and $D(n) = \Theta(1)$ (computes the middle of the subarray).

$T(n) = 2T(n/2) + \Theta(n)$ and $T(1) = 0$

Merge Sort - Pseudocode

Here Lower $\leq$ Upper, and they denote lower and upper indices of the subarray to be sorted. We need a copy of Source to make sorting. In Destination we get a sorted sequence. The sorting itself is done in Merge.

```
define IndexLimit 1000
typedef /* type used for sorting */ ElementType;
typedef ElementType ElementArray[IndexLimit];
void MergeSort(ElementArray Source, ElementArray Destination,
               int Lower, int Upper)
{  int Mid;
   if (Lower < Upper) {
      Mid = (Lower + Upper)/2;
      MergeSort(Destination, Source, Lower, Mid);
      MergeSort(Destination, Source, Mid + 1, Upper);
      Merge(Source, Destination, Lower, Mid, Upper);
   }
} /* end of MergeSort */
main() {
  ElementArray ACopy;
  int k, n;
  ...
  for (k = 0; k < n; k++)
    ACopy[k] = A[k];
    /* there are n elements to sort in the array A */
```

The Pseudocode for the Merge Operation

```
void Merge(ElementArray Source, ElementArray Destination,
           int Lower, int Mid, int Upper)
{ int s1, s2, d;
    s1 = Lower;  s2 = Mid + 1; d = Lower;
    do {   /* comparisons */
        if (Source[s1] < Source[s2]) {
           Destination[d] = Source[s1];
           s1++; }
        else {
           Destination[d] = Source[s2];
           s2++; };
        d++;
    } while (s1 <= Mid && s2 <= Upper);
```

Merge Operation (cont.)

```
/* move what is left of remaining list */
  if (s1 > Mid)
    do {
      Destination[d] = Source[s2];
      s2++;  d++;
    } while (s2 <= Upper);
  else
    do {
     Destination[d] = Source[s1];
      s1++;  d++;
    } while (s1 <= Mid);
} /* end of Merge */
```

Verification of Solution Using Structural Induction

The merge sort requires $\Theta(n \log_2 n)$ operations (comparisons). There is no best-case or worst-case for the merge sort. To get this result we need to solve the following equation:

$$T(n) = 2T(n/2) + \Theta(n), \qquad \text{if } n > 1,$$

with $T(1) = \Theta(1)$. It turns out that the solution is $\Theta(n \log_2 n)$. This can be proved using mathematical induction.
The price paid for using the merge sort is an increased memory space because it needs a duplicate copy of the array being sorted. In many situations it can be impractical (especially if n is large).
The induction method is used to justify that the guess solution for merge sort is correct.
According to the definition of the big-O notation it should shown that exits positive constant c and n_0 such for any $n \geq n_0$ $T(n) \leq cn \log_2 n$. Assume that $n = 2^k$

Proof by Structural Induction

Base case: Take $n = 2$ so $T(2) = 2T(1) + 2 = 4$ and $4 \leq 2c$ so $c \geq 2$.
Inductive step: Assume for $\frac{n}{2}$, there exists a positive constant c that $T(\frac{n}{2}) \leq c\frac{n}{2}\log_2(\frac{n}{2})$
Prove: that for any $n \geq \frac{n}{2}$, $T(n) \leq c * n \log_2 n$.
Proof:
$T(n) = 2T(\frac{n}{2}) + n \leq 2cn\log_2(\frac{n}{2}) + n = cn(\log_2(n) - 1) + n = cn\log_2 n - n(1 - c) \leq cn\log_2 n$ for $c \geq 1$.
Combine the basis and the inductive steps, we can conclude that $c \geq 2$.
We can prove that the lower bound Ω is also of the order of $n\log_2 n$ for any positive constant $c \leq 1$.

Example of Verification of Solution

Consider

$$T(n) = 2T(n/2) + n^3 \qquad \text{if } n > 1,$$

Guess: $T(n) = O(n^3)$
induction: assume that $T(k) \leq ck^3$, and prove $T(n) \leq cn^3$ for $n \geq k$

$$\begin{aligned} T(n) &= 2T(n/2) + n^3 \\ &\leq 2c(n/2)^3 + n^3 \\ &\leq cn^3/4 + n^3 \\ &\leq (c/4+1)n^3 \\ &\leq cn^3. \end{aligned}$$

assuming that $c/4 + 1 \leq c$, which is equivalent to $c \geq 4/3$.

A Wrong Guess

If we guess (wrongly): $T(n) = O(n)$, then by induction we assume that $T(k) \leq ck$ and $T(n) \leq cn$ for any n.

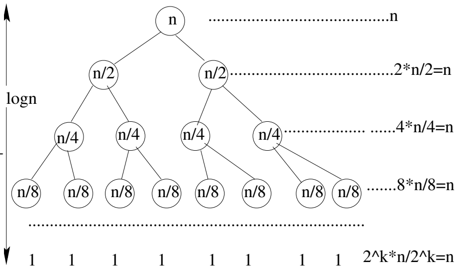
$$\begin{aligned} T(n) &= 2T(n/2) + n^3 \\ &\leq 2c(n/2) + n^3 \\ &\leq cn + n^3 \\ &= (c + n^2)n \\ &\leq (?)cn. \end{aligned}$$

The induction proof is incorrect because the value of c depends now on n (and it is not longer constant).

Iterating Method and Recursive Tree for Merge Sort.

$T(n) = 2T(\frac{n}{2}) + n$ and $T(1) = 1$
Replace $T(\frac{n}{2})$ by $2T(\frac{n}{4}) + \frac{n}{2}$
$T(n) = n + 2 * \frac{n}{2} + 4T(\frac{n}{4})$
$= 2n + 4T(\frac{n}{4})$
After the k steps $T(n) = kn + 2^k T(1)$

and $k = \log_2 n$.



The Master Method

The Master methods provides an asymptotic bound depending on comparison of the polynomial power of $f(n)$ with some powers of n. This is given in the following theorem.

Theorem Let $a \geq 1$ and $b > 1$ be constants, $f(n)$ be a function, and $T(n)$ be defined (on nonnegative integers) by:

$$T(n) = aT(n/b) + f(n)$$

Then $T(n)$ can be bounded asymptotically as follows:
$T(n) = \Theta(n^{\log_b a})$ if $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$
$T(n) = \Theta(n^{\log_b(a)} \log n)$ if $f(n) = \Theta(n^{\log_b a})$
$T(n) = \Theta(f(n))$ if $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

The Master Method for Merge Sort

The Merge sort recurrence relation

$$T(n) = 2T(n/2) + \Theta(n)$$

Using the master method we have $a = 2$, $b = 2$, $f(n) = \Theta(n)$. Then $\log_2 2 = 1$ and $n^1 = n$. We have case 2 of the master method. The solution is of order $T(n) = n \log_2 n$

The Master Method - Examples

Examples

1. Let

$$T(n) = T(n/2) + \Theta(1)$$

Using the master method we have $a = 1$, $b = 2$, $f(n) = \Theta(1)$. Then $\log_2 1 = 0$ and $n^0 = 1$. We have case 2 of the master method. The solution is $T(n) = \log_2 n$.

2. Let

$$T(n) = 3T(n/3) + n^2$$

Here $a = 3$, $b = 3$ and $\log_3 3 = 1$. Since $f(n) = n^2$ and $2 > 1$ we get the case 3. We need to check the regularity condition:

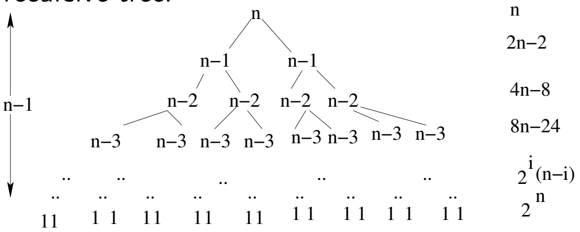
$$3(n/3)^2 \leq cn^2 \qquad \text{for some } c < 1.$$

It is enough to take $c \geq 1/3$, for instance $c = 0.5$. The solution is $T(n) = \Theta(n^2)$.

Examples

Example

Solve the recurrence $T(n) = 2T(n-1) + n$ and $T(1) = 1$ using a recursive tree.



$$
\begin{aligned}
T(n) &= 2(2T(n-2) + (n-1) + n \\
&= 4T(n-2) + 2(n-1) + n \\
&= \cdots = \\
&= \sum_{i=0}^{n-1} 2^i (n-i) \\
&= 2^n \sum^{n} i(1/2)^i \leq 2^{n+1}.
\end{aligned}
$$

Multiplication of n-bit integers

Let X and Y be two n-bit integers (assume that $n = 2^k$).

$$X = 2^{n/2}X_L + X_R$$

$$Y = 2^{n/2}Y_L + Y_R$$

X =	X_L	X_R
Y =	Y_L	Y_R

$$\begin{aligned}
XY &= (2^{n/2}X_L + X_R)(2^{n/2}Y_L + Y_R) \\
&= 2^n X_L Y_L + 2^{n/2} X_L Y_R + 2^{n/2} X_R Y_L + X_R Y_R \\
&= 2^n X_L Y_L + 2^{n/2}(X_L Y_R + X_R Y_L) + X_R Y_R
\end{aligned}$$

There are four intermediate $n/2$-bit multiplications and three intermediate $n/2$-bit additions with running time $O(n)$. That gives the following recurrence relation: $T(n) = 4T(n/2) + O(n)$ with a solution $T(n) = O(n^2)$.

But multiplication of the n-bit integers X and Y can be done with only three intermediate $n/2$-bit multiplications.
Notice that

$$X_L Y_R + X_R Y_L = (X_L + X_R)(Y_L + Y_R) - X_L Y_L - X_R Y_R$$

and then

$$\begin{aligned} XY &= 2^n X_L Y_L + 2^{n/2} \left((X_L + X_R)(Y_L + Y_R) - X_L Y_L - X_R Y_R\right) + X_R Y_R \\ &= (2^n - 2^{n/2}) X_L Y_L + 2^{n/2} (X_L + X_R)(Y_L + Y_R) - (2^{n/2} - 1) X_R Y_R \end{aligned}$$

(cont.)

This is a recursive algorithm for this multiplication problem. Notice that multiplication by a power of 2 is done by shifting bits (and this can be done fast on any computer).

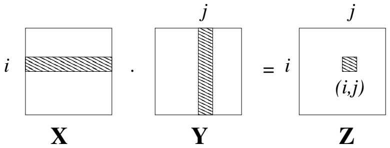
$multiply(X, Y)$
input n–bit positive integers X and Y
if $n = 1$ return $X * Y$
X_L, X_R= leftmost $\lceil n/2 \rceil$, rightmost $\lfloor n/2 \rfloor$ bits of X
Y_L, Y_R= leftmost $\lceil n/2 \rceil$, rightmost $\lfloor n/2 \rfloor$ bits of Y
$p_1 = multiply(X_L, Y_L)$
$p_2 = multiply(X_R, Y_R)$
$p_3 = multiply(X_L + X_R, Y_L + Y_R)$
return $p_1 * 2^n + (p_3 - p_1 - p_2) * 2^{n/2} + p_2$

This is the recurrence relation: $T(n) = 3T(n/2) + O(n)$ with a solution $T(n) = \Theta(n^{\log_2 3}) \cong O(n^{1.59})$.

Matrix Multiplication

The product of two $n \times n$ matrices X and Y is a third matrix $Z = XY$, with (i,j)th entry

$$Z_{i,j} = \sum_{k=1}^{n} X_{i,k} Y_{k,j}.$$



There are n^2 entries and calculation of each of them takes $O(n)$ arithmetic operations and this makes the algorithm to execute $n^2 O(n) = O(n^3)$ operations.

In 1969, the German mathematician Volker Strassen announced more efficient algorithm, based on the divide-and-conquer strategy.

(cont.)

For matrix multiplications, it is easy to break them into subproblems using a block-wise approach – each matrix can be represented with four blocks, each of size $n/2 \times n/2$ (for simplicity, assume that n is a power-of-2 integer).

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix},$$

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

We apply the divide-and-conquer strategy to compute a product of XY. We recursively compute eight size-$n/2$ products: $AE, BG, AF, BH, CE, DG, CF, DH$, and then perform $O(n^2)$ additions. The behavior of this algorithm is described by this formula $T(n) = 8T(n/2) + O(n^2)$, which gives the running time as $O(n^3)$.

(cont.)

Strassen showed that at each step the product can be computed with only seven multiplications using some clever algebraic tricks

$P_1 = A(F - H)$ $\quad P_5 = (A + D)(E + H)$

$P_2 = (A + B)H$ $\quad P_6 = (B - D)(G + H)$

$P_3 = (C + D)E$ $\quad P_7 = (A - C)(E + F)$

$P_4 = D(G - E)$

and $XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$.

The algorithm is represented by the recurrence relation, $T(n) = 7T(n/2) + O(n^2)$. Then, by the master theorem, $T(n) = O(n^{\log_2 7}) \cong O(n^{2.81})$ operations, which is smaller than $O(n^3)$ for large n. (Note that the constant in $O(n^{\log_2 7})$ can be large.)