Algorithms – Ch4 - Divide & Conquer

Recurrences: as we saw in another lecture, the Divide and Conquer approach leads to Recurrence Formulae to represent its time costs:
• for low values of n (1 is typical, but we could have n larger) \( T(n) = \Theta(1) \).
• for large(r) values of n, \( T(n) = a \cdot T(n/b) + f(n) \), where \( f(n) \) represents the costs of dividing and those of recombining the solutions.

As we go, we will examine in some detail a few methods of solution.

At this point we will mostly just look at some examples of algorithms to which this point of view applies.

1. The substitution method: essentially, guess the solution and prove the guess to be correct by mathematical induction. This will work well in all cases. **Slight difficulty**: you have to guess right – and that is not easy in most cases…

2. The recursion-tree method: convert the recurrence into a tree (as in Mergesort) and add up all the costs of each level, and then add up the costs of all the levels. **Slight difficulty**: unless the costs at each level are easy to compute, you may have some difficult bounds to cook up.

3. The Master Method: there is a theorem... Basically, find out which set of hypotheses hold in your special case, and read off the conclusion. **Slight difficulty**: sometimes you will have trouble proving that the hypotheses you need (or any relevant hypotheses) are actually satisfied.

Examples of Divide and Conquer:

the Maximum Subarray problem.

Problem: given an array of n numbers, find the (a) contiguous subarray whose sum has the largest value.

Application: an unrealistic stock market game, in which you decide when to buy and sell a stock, with full knowledge of the past and future. **The restriction is** that you can perform just one buy followed by a sell. The buy and sell both occur right after the close of the market.

The interpretation of the numbers: each number represents the stock value at closing on any particular day.

Examples:

<table>
<thead>
<tr>
<th>Day</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price</td>
<td>100</td>
<td>113</td>
<td>110</td>
<td>85</td>
<td>105</td>
<td>102</td>
<td>86</td>
<td>63</td>
<td>81</td>
<td>101</td>
<td>94</td>
<td>106</td>
<td>101</td>
<td>79</td>
<td>94</td>
<td>30</td>
<td>97</td>
</tr>
<tr>
<td>Change</td>
<td>13</td>
<td>-3</td>
<td>-25</td>
<td>20</td>
<td>-3</td>
<td>-16</td>
<td>-23</td>
<td>18</td>
<td>20</td>
<td>-7</td>
<td>12</td>
<td>-5</td>
<td>-22</td>
<td>15</td>
<td>-4</td>
<td>-7</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.1**: Information about the price of stock in the Volatile Chemical Corporation after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.
Another Example: buying low and selling high, even with perfect knowledge, is not trivial:

First Solution: compute the value change of each subarray corresponding to each pair of dates, and find the maximum.

1. How many pairs of dates:
2. This belongs to the class $\Theta(n^2)$
3. The rest of the costs, although possibly constant, don’t improve the situation: $\Omega(n^2)$.

Not a pleasant prospect if we are rummaging through long time-series (Who told you it was easy to get rich??), even if you are allowed to post-date your stock options...

We are going to find an algorithm with an $o(n^2)$ running time (i.e. strictly asymptotically faster than $n^2$), which should allow us to look at longer time-series.

Transformation: Instead of the daily price, let us consider the daily change: $A[i]$ is the difference between the closing value on day $i$ and that on day $i-1$.

The problem becomes that of finding a contiguous subarray the sum of whose values is maximum.

On a first look this seems even worse: roughly the same number of intervals (one fewer, to be precise), and the requirement to add the values in the subarray rather than just computing a difference: $\Omega(n^3)$?

It is actually possible to perform the computation in $\Theta(n^2)$ time by

1. Computing all the daily changes;
2. Computing the changes over 2 days (one addition each)
3. Computing the changes over 3 days (one further addition to extend the length-2 arrays... etc... check it out. You’ll need a two-dimensional array to store the intermediate computations.

Still bad though. Can we do better??
How do we divide?

We observe that a maximum contiguous subarray $A[i...j]$ must be located as follows:
1. It lies entirely in the left half of the original array: $[low...mid]$;
2. It lies entirely in the right half of the original array: $[mid+1...high]$;
3. It straddles the midpoint of the original array: $i \leq mid < j$.

The left-recursion will return the indices and value for the largest contiguous subarray in the left half of $A[low...high]$, the right recursion will return the indices and value for the largest contiguous subarray in the left half of $A[low...high]$, and `FIND-MAX-CROSSING-SUBARRAY` will return the indices and value for the largest contiguous subarray that straddles the midpoint of $A[low...high]$.

It is now easy to choose the contiguous subarray with largest value and return its endpoints and value to the caller.

---

The total numbers of iterations for both loops is exactly $high-low+1$.

The left-recursion will return the indices and value for the largest contiguous subarray in the left half of $A[low...high]$, the right recursion will return the indices and value for the largest contiguous subarray in the left half of $A[low...high]$, and `FIND-MAX-CROSSING-SUBARRAY` will return the indices and value for the largest contiguous subarray that straddles the midpoint of $A[low...high]$.

It is now easy to choose the contiguous subarray with largest value and return its endpoints and value to the caller.
The recursive algorithm:

```java
FIND-MAXIMUM-SUBARRAY(A, low, high)
1 if high == low
2 return (low, high, A[low]) // base case: only one element
3 else mid = (low + high)/2
4 (left-low, left-high, left-sum) = FIND-MAXIMUM-SUBARRAY(A, low, mid)
5 (right-low, right-high, right-sum) = FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
6 (cross-low, cross-high, cross-sum) = FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
7 if left-sum ≥ right-sum and left-sum ≥ cross-sum
8 return (left-low, left-high, left-sum)
9 else if right-sum ≥ left-sum and right-sum ≥ cross-sum
10 return (right-low, right-high, right-sum)
11 else return (cross-low, cross-high, cross-sum)
```

We finally have:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
2T(n/2) + \Theta(n) & \text{if } n > 1.
\end{cases}
\]

The recurrence has the same form as that for MERGESORT, and thus we should expect it to have the same solution \( T(n) = \Theta(n \log n) \).

This algorithm is clearly substantially faster than any of the brute-force methods. It required some cleverness, and the programming is a little more complicated – but the payoff is large.

### Strassen’s Algorithm for Matrix Multiplication.

We start from the standard algorithm for matrix multiplication:

```java
SQUARE-MATRIX-MULTIPLY(A, B)
1 n = A.rows
2 let C be a new n \times n matrix
3 for i = 1 to n
4 for j = 1 to n
5 \( c_{ij} = 0 \)
6 for k = 1 to n
7 \( c_{ij} = c_{ij} + a_{ik} \cdot b_{kj} \)
8 return C
```

The triple nested loop clearly implies a time \( \Theta(n^3) \), which is not quite as bad as it looks since we are dealing with \( n^2 \) elements, so \( n^3 = (n^2) \cdot 3 \).
Algorithms – Ch4 - Divide & Conquer

If we are going to try some clever Divide & Conquer scheme, we could start by coming up with a non-clever one... Here it is: if \( n \) is a power of 2, we can always subdivide an \( n \times n \) matrix into 4 \( n/2 \times n/2 \) ones:

\[
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},
\]

so that we rewrite the equation \( C = A \cdot B \) in

\[
\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.
\]

Equation (4.10) corresponds to the four equations

\[
C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{12},
C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22},
C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{12},
C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}.
\]

Each of these four equations specifies two multiplications of \( n/2 \times n/2 \) matrices and the addition of their \( n/2 \times n/2 \) products. We can use these equations to create a straightforward, recursive, divide-and-conquer algorithm:

The Algorithm:

**SQUARE-MATRIX-MULTIPLY-RECURSIVE** \((A, B)\)

1. \( n = A\. \text{rows} \)
2. let \( C \) be a new \( n \times n \) matrix
3. if \( n = 1 \)
   4. \( C_{11} = a_{11} \cdot b_{11} \)
5. else partition \( A, B, \) and \( C \) as in equations (4.9)
6. \( C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11}) + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21}) \)
7. \( C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{22}) + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{22}) \)
8. \( C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11}) + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{21}) \)
9. \( C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{22}) + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{22}) \)
10. return \( C \)

More importantly, one can ask the question: do we need all 8 multiplications or can we find a clever way or coming up with fewer? We would expect a cost, probably in a larger number of additions, but additions are much cheaper and, as long as the number of additions doesn’t go up with \( n \), we can put them in the \( \Theta(n^2) \) term.

Algorithms – Ch4 - Divide & Conquer

A pretty immediate conclusion is that multiplying 2 \( n \times n \) matrices involves 8 multiplications of \( n/2 \times n/2 \) matrices and 4 additions of \( n/2 \times n/2 \) matrices.

Additions of matrices are easy: 2 \( n \times n \) matrices cost \( \Theta(n^2) \) to add, and the total cost of the 4 additions remains \( \Theta(n^2) \).

Partitioning of matrices has cost that depends on the method: copying would have a \( \Theta(n^2) \) cost (because of the \( n^2 \) elements), using indices that provide information on the original array (avoiding copying) could cost as little as \( \Theta(1) \).

The Recursion formula is:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
\Theta(1) + 8T(n/2) + \Theta(n^2) & \text{otherwise.}
\end{cases}
\]

Using the Master method (apply the theorem) we should end up with the same result we had before. We could also try \( T(n) = cn^3 \) and see if we can show an appropriate inequality remains satisfied through a mathematical induction.
Algorithms – Ch4 - Divide & Conquer

Strassen’s Method:

1. Divide the input matrices $A$ and $B$ and output matrix $C$ into $n/2 \times n/2$ submatrices, as in equation (4.9). This step takes $\Theta(1)$ time by index calculation, just as in SQUARE-MATRIX-MULTIPLY-RECURSIVE.

2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices in $\Theta(n^2)$ time.

3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products $P_1, P_2, \ldots, P_7$. Each matrix $P_i$ is $n/2 \times n/2$.

4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix $C$ by adding and subtracting various combinations of the $P_i$ matrices. We can compute all four submatrices in $\Theta(n^2)$ time.

The Substitution Method

This is the easiest method, IF YOU CAN GUESS... And we know that guessing is rarely easy...

Consider the recurrence:

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

which is similar to previous ones.

They suggest a guess $T(n) = O(n \lg n)$. We need to show that $T(n) \leq c \cdot n \lg n$ for an appropriate choice of $c > 0$ and “all large enough $n$”. We don’t worry too much about $n = 1$ (or $n$ small) for now: the base case should become clear later.

We start with the “induction hypothesis”, and keep track of the “floor”...

The Substitution Method

Induction hypothesis for the specific recurrence: assume the bound holds for all $m < n$ (in particular $m \leq \lfloor n/2 \rfloor$):

$$T(n) \leq 2(c \cdot \lfloor n/2 \rfloor \cdot \lg \lfloor n/2 \rfloor) + n \leq c \cdot n \cdot \lg \lfloor n/2 \rfloor + n$$

$$= c \cdot n \cdot \lg n - c \cdot n \cdot \lg 2 + n$$

$$= c \cdot n \cdot \lg n - cn + n$$

$$\leq c \cdot n \cdot \lg n$$ (true as long as we pick $c \geq 1$)

The problem that remains is: what about the “boundary condition” (or the “base case” for the induction)? If we start with $T(1) = 1$, the functional bound is NOT satisfied...

Solution: start the induction with a larger $n$... after all we are concerned with asymptotics, which means we get to choose when to first require the inequality to hold...

Algorithms – Ch4 - Divide & Conquer

The Recursion Formula becomes:

$$T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
\Theta(1) + 7T(n/2) + \Theta(n^2) & \text{otherwise.} 
\end{cases}$$

With solution $T(n) = \Theta(n^\omega) = \Theta(n^{2.81})$. 
The Substitution Method

What we have to ensure is that there is an \( n_0 > 0 \) and a \( c > 0 \) as in the "big Oh" definition. Let's go find them.

\[
T(1) = 1; \ T(2) \leq 2 \ T(1) + 2 = 2 + 2 = 4; \ T(3) = 2 \ T(1) + 3 \leq 5.
\]

For \( n > 3 \), the recurrence does not depend directly on \( T(1) \) and \( c \ n \ lg \ n > 0 \). Choose \( c \) large enough so that \( T(2) = 4 \leq c \ 2 \ lg \ 2 \), and \( T(3) = 5 \leq c \ 3 \ lg \ 3 \). Any choice \( c \geq 2 \) will do it.

Other problems with this method: see text pp 84-86

The Master Theorem

This is a result that tries to unify a number of possible recursion formulae.

**Theorem 4.1 (Master theorem)**

Let \( a \geq 1 \) and \( b > 1 \) be constants, let \( f(n) \) be a function, and let \( T(n) \) be defined on the nonnegative integers by the recurrence

\[
T(n) = aT(n/b) + f(n),
\]

where we interpret \( n/b \) to mean either \( [n/b] \) or \( \lfloor n/b \rfloor \). Then \( T(n) \) has the following asymptotic bounds:

1. If \( f(n) = O(n^{log_b a}) \) for some constant \( c > 0 \), then \( T(n) = \Theta(n^{log_b a}) \).
2. If \( f(n) = \Theta(n^{log_b a}) \), then \( T(n) = \Theta(n^{log_b a} \ lg n) \).
3. If \( f(n) = \Omega(n^{log_b a}) \) for some constant \( c > 0 \), and if \( af(n/b) \leq cf(n) \) for some constant \( c < 1 \) and all sufficiently large \( n \), then \( T(n) = \Theta(f(n)) \).

For this recurrence, we have \( a = 9, \ b = 3, \ f(n) = n \), and thus we have that \( n^{log_{10} 3} = n^{0.5} = \Theta(n^{0.5}) \). Since \( f(n) = O(n^{log_{10} 3}) \), where \( \epsilon = 1 \), we can apply case 1 of the master theorem and conclude that the solution is \( T(n) = \Theta(n^{0.5}) \).
Algorithms – Ch4 - Divide & Conquer

\[ T(n) = T(2n/3) + 1, \]

in which \( a = 1, b = 3/2, f(n) = 1, \) and \( n^{log_a b} = n^{log_{3/2} 1} = n^0 = 1. \) Case 2 applies, since \( f(n) = \Theta(n^{log_a b}) = \Theta(1), \) and thus the solution to the recurrence is \( T(n) = \Theta(\lg n). \)

Algorithms – Ch4 - Divide & Conquer

\[ T(n) = 3T(n/4) + n \lg n. \]

we have \( a = 3, b = 4, f(n) = n \lg n, \) and \( n^{log_a b} = n^{log_{4} 3} = O(n^{0.795}). \)

Since \( f(n) = \Omega(n^{log_a b} + \epsilon), \) where \( \epsilon = 0.2, \) case 3 applies if we can show that the regularity condition holds for \( f(n). \) For sufficently large \( n, \) we have that

\[ af(n/b) = 3(n/4) \lg(n/4) \leq (3/4)n \lg n = cf(n) \]

for \( c = 3/4. \) Consequently, by case 3, the solution to the recurrence is \( T(n) = \Theta(n \lg n). \)

The master method does not apply to the recurrence

\[ T(n) = 2T(n/2) + n \lg n, \]

even though it appears to have the proper form: \( a = 2, b = 2, f(n) = n \lg n, \) and \( n^{log_a b} = n. \) You might mistakenly think that case 3 should apply, since \( f(n) = n \lg n \) is asymptotically larger than \( n^{log_a b} = n. \) The problem is that it is not polynomially larger. The ratio \( f(n)/n^{log_a b} = (n \lg n)/n = \lg n \) is asymptotically less than \( n^\epsilon \) for any positive constant \( \epsilon. \) Consequently, the recurrence falls into the gap between case 2 and case 3. (See Exercise 4.6-2 for a solution.)

Let's use the master method to solve the recurrences we saw in Sections 4.1 and 4.2. Recurrence (4.7),

\[ T(n) = 2T(n/2) + \Theta(n), \]

characterizes the running times of the divide-and-conquer algorithm for both the maximum-subarray problem and merge sort. (As is our practice, we omit stating the base case in the recurrence.) Here, we have \( a = 2, b = 2, f(n) = \Theta(n), \) and thus we have that \( n^{log_a b} = n^{log_{2} 2} = n. \) Case 2 applies, since \( f(n) = \Theta(n), \) and so we have the solution \( T(n) = \Theta(n \lg n). \)
Recurrence (4.17),
\[ T(n) = 8T(n/2) + \Theta(n^2) , \]
describes the running time of the first divide-and-conquer algorithm that we saw for matrix multiplication. Now we have \(a = 8, b = 2,\) and \(f(n) = \Theta(n^2),\)
and so \(n^{\log_{2} 8} = n^{3} = n^3.\) Since \(n^3\) is polynomially larger than \(f(n)\) (that is, \(f(n) = O(n^{b+\epsilon})\) for \(\epsilon = 1),\) case 1 applies, and \(T(n) = \Theta(n^3).\)

Finally, consider recurrence (4.18),
\[ T(n) = 7T(n/2) + \Theta(n^2) , \]
which describes the running time of Strassen’s algorithm. Here, we have \(a = 7, b = 2, f(n) = \Theta(n^2),\) and thus \(n^{\log_{2} 7} = n^{m_{7}}.\) Rewriting \(\log_{2} 7\) as \(\log 7\) and recalling that \(2.80 < \log 7 < 2.81,\) we see that \(f(n) = O(n^{b+\epsilon})\) for \(\epsilon = 0.8.\)
Again, case 1 applies, and we have the solution \(T(n) = \Theta(n^{m_{7}}).\)