Maggie Johnson

Handout #16
CS109

Recurrence Relations
Analysis of Recursive Algorithms

Key topics:

* Recurrence Relations
* Solving Recurrence Relations
* The Towers of Hanoi
* Analyzing Recursive Subprograms

Recurrence Relations

Recall that a recurrence relation is the second part of a recursive definition:

> A recursive or inductive definition has the following parts:
>
> 1) the initial condition or basis which defines the first (or first few) elements of the sequence;
> 2) an inductive step in which later terms in the sequence are defined in terms of earlier terms. This is called a recurrence relation.

A **recurrence relation** for a sequence $a_1, a_2, a_3,...$ is a formula that relates each term a_k to certain of its predecessors $a_{k-1}, a_{k-2}, \ldots, a_{k-i}$, where i is a fixed integer and k is any integer greater than or equal to i. The initial conditions for such a recurrence relation specify the values of $a_1, a_2, a_3,\ldots a_{i-1}$.

When we introduced the idea of a recursive definition, we were talking about the many ways of defining a sequence of values (enumerate them, come up with a formula or give a recursive definition). Suppose you have an enumerated sequence that satisfies a given recursive definition (or recurrence relation). It is frequently very useful to have the formula as well as these other two definitions, especially if you need to determine a very large member of the sequence. Such an explicit formula is called a *solution* to the recurrence relation. If the member of the sequence can be calculated using a fixed number of elementary operations, we say it is a *closed form formula*. Solving recurrence relations is the key to analysing recursive subprograms.

We will begin by learning how to model problems using recurrence relations.

Example 1

a) Make a list of all bit strings of length 0, 1, 2, & 3 that do not contain the bit pattern 11. How many such strings are there?
b) Find the number of strings of length ten that do not contain the pattern 11.

a) One way to solve this problem is enumeration:

length 0: empty string	1
length 1: 0, 1	2
length 2: 00, 01, 10, 11	3
length 3: 000, 001, 010, 011, 100, 101, 110, 111	5

b) To do this for strings of length ten would be ridiculous because there are 1024 possible strings. There must be a better way...

Suppose the number of bit strings of length less than some integer k that do not contain the 11 pattern is known (it just so happens that we do know it for 0, 1, 2, 3). To find the number of strings of length k that do not contain the bit pattern 11, we can try to express strings that have a certain property in terms of shorter strings with the same property.

Consider the set of all bit strings of length k that do not contain 11. Any string in the set begins with a 0 or a 1. If the string begins with a 0, the remaining k-1 characters can be any sequence of 0's and 1's except the pattern 11 cannot appear. If the string begins with 1, then the second character must be a 0; the remaining k-2 characters can then be any sequence of 0's and 1's that does not contain the pattern 11. So, the set of all strings of length k that do not contain the pattern 11 can be partitioned as follows:

set of all bit strings of form 0 - - - - k-1 bits without 11	set of all bit strings of form 10 - - - k-2 bits without 11

The number of elements in the entire set then equals the sum of the elements of the two subsets:

the number of strings of length k that do not contain 11	=	the number of strings of length k-1 that do not contain 11	+	the number of strings of length k - 2 that do not contain 11

Or, in other words (as a recurrence relation):

1) $s_0 = 1 \qquad s_1 = 2$
2) $s_k = s_{k-1} + s_{k-2}$

If we calculate this recurrence to s_{10}, we get 144.

Now try this one: A single pair of rabbits (male and female, of course) is born at the beginning of a year. Assume the following interesting conditions:

1) Rabbit pairs are not fertile during the first month of life, but thereafter give birth to one new male/female pair at the end of *every* month.
2) No deaths ever occur (not even from exhaustion...)

Find a recurrence relation that can be used to tell us how many rabbits we will have at the end of one year.

Many problems in the biological, management and social sciences lead to sequences which satisfy recurrence relations (as we just saw). Here is the famous *Lancaster Equations of Combat*:

Two armies engage in combat. Each army counts the number of soldiers still in combat at the end of the day. Let a_0 and b_0 denote the number of soldiers in the first and second armies, respectively, before combat begins, and let a_n and b_n denote the number of men in the two armies at the end of the n^{th} day. Thus, $a_{n-1} - a_n$ represents the number of soldiers *lost* by the first army on the n^{th} day. Similarly, $b_{n-1} - b_n$ represents the number of soldiers lost by the second army on the n^{th} day.

Suppose it is known that the decrease in the number of soldiers in each army is proportional to the number of soldiers in the other army at the beginning of each day. Thus, we have constants A and B such that $a_{n-1} - a_n = A * b_{n-1}$ and $b_{n-1} - b_n = B * a_{n-1}$. These constants measure the effectiveness of the weapons of the different armies.

Solving Recurrence Relations

Now we need to look at how to find a closed form formula for a recurrence relation. There are a few different methods. The most intuitive is called the "guess" method, where you enumerate some elements of the sequence, look for a pattern and guess at a formula. Then, you use induction to prove it is correct.

Example 2

Consider the following sequence:

1) $a_0 = 1$
2) $a_k = a_{k-1} + 2$

We want to find a formula that can be used to calculate any member of this sequence without having to calculate previous members; this is what we mean when we say we want to find a closed form formula. Having such a formula saves a lot of time, and they will come in handy in just a little while. First, enumerate some of the elements:

$a_0 = 1$
$a_1 = a_1 + 2 = 1 + 2 = 3$
$a_2 = a_2 + 2 = 3 + 2 = 5$
$a_3 = a_3 + 2 = 5 + 2 = 7$ this looks familiar...
$a_n = 1 + 2n$

Then, we have to inductively prove that our guess is correct. P(n) denotes the following:

if $a_1, a_2, a_3, \ldots, a_n$ is the sequence defined by

1) $a_0 = 1$
2) $a_k = a_{k-1} + 2$

then, $a_n = 1 + 2n$ for all $n >= 1$

i) base case: prove that P(0) is true: $a_0 = 1 + 2 * 0 = 1$.

ii) induction hypothesis is to assume P(k): $a_k = 1 + 2k$ and show P(k+1): $a_{k+1} = 1 + 2(k + 1)$.

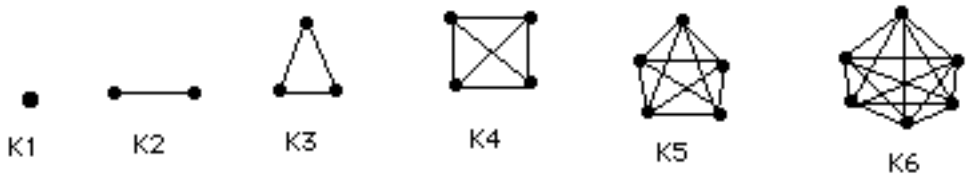
PROOF:

$a_{k+1} = a_{(k-1+1)} + 2$	from the recurrence relation (this is a "given")
$a_{k+1} = a_k + 2$	algebra
$a_{k+1} = 1+2k + 2$	substitution from inductive hypothesis
$a_{k+1} = 1+2(k+1)$	algebra

P(k+1) is true when P(k) is true, and therefore P(n) is true for all natural numbers.

Another magical application of induction!

Let K_n be a complete graph on n vertices, i.e., between any two of the n vertices lies an edge.



Define a recurrence relation for the number of edges in K_n, and then solve this using the "guess" method.

Another method of solving recurrence relations is called *repeated substitution*. It works like this: you take the recurrence relation and use it to enumerate elements of the sequence back from k. To find the next element back from k, we substitute k-1 in for k in the recurrence relation. In the previous example ($a_k = a_{k-1} + 2$), we get

$$a_{k-1} = a_{k-2} + 2$$

Then, we substitute this value for a_{k-1} back in the original recurrence relation equation. We do this because we want to find an equation for a_k, so if we enumerate back from k, but substitute those equations in the formula for a_k, we will hopefully find a pattern. What we are looking for is a lot of different representations of the formula for a_k based on previous elements in the sequence. So,

$$a_k = a_{k-1} + 2 = (a_{k-2} + 2) + 2 = a_{k-2} + 4$$

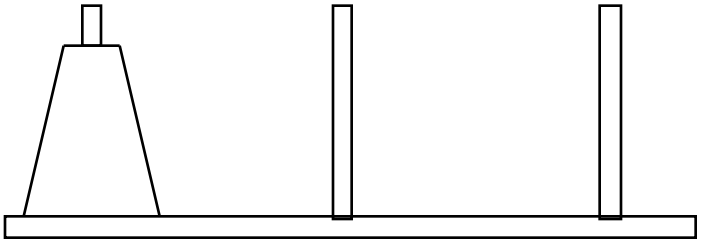
Then, we substitute k-2 for k in the original equation to get $a_{k-2} = a_{k-3} + 2$ and then substitute this value for a_{k-2} in the previous expression:

$$a_k = a_{k-2} + 4 = (a_{k-3} + 2) + 4 = a_{k-3} + 6$$
$$a_k = a_{k-3} + 6 = (a_{k-4} + 2) + 6 = a_{k-4} + 8$$
$$a_k = a_{k-4} + 8\ldots.$$

The pattern is obvious: $a_k = a_{k-i} + 2i$. We need to get rid of some of these variables though. Specifically, we need to get rid of the a_k on the right side of the equation to get a closed form formula; i.e., one where we do not have to evaluate previous members of the sequence to get the one we want. Since we have proven using induction that the formula works for all i, we can set i equal to any value we want to simplify this formula. Set i = k. We get: $a_k = a_0 + 2k$ or $a_k = 1 + 2k$.

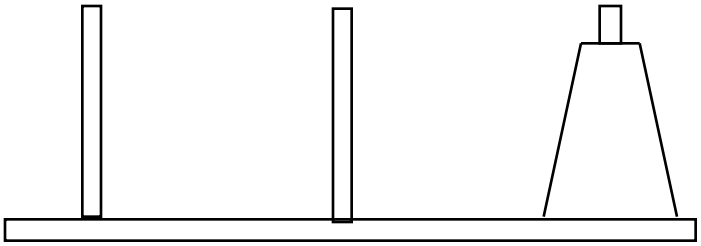
Example 3

The Towers of Hanoi was a popular puzzle in the 19th century and in certain versions of CS106. The game consists of three pegs mounted on a board together with disks of different sizes. Initially, these disks are placed on the first peg in order of size with the largest on the bottom..



The object of the game is to move all the disks from the first peg to the third but:

1) Only one disk may be moved at a time (specifically, the top disk on any peg).
2) At no time can a larger disk be placed on a smaller one.



One solution is to move the top k-1 disks from the first peg to the second using the third as an auxiliary holding spot. Then, move the bottom disk from the first to the third. Finally, move the top k-1 disks from the second to the third peg using the first as an auxiliary holding spot. This, needless to say, suggests a recursive solution (given below in pseudocode):

```
Tower(k, Beg, Aux, End) {
        if k = 1
                move disk from Beg to End;
        else {
                Tower(k-1, Beg, End, Aux);
                move disk from Beg to End;
                Tower(k-1, Aux, Beg, End) }
}
```

Let's see if we can use what we just learned about modeling and solving recurrence relations to better understand this problem. If H_n denotes the number of moves to solve the Tower of Hanoi puzzle with n disks, we will try to find a recurrence relation for the sequence, and the solution to the recurrence relation. This way we will be able to tell how many moves it will take to solve a Towers of Hanoi problem of any size; something I am sure you have always wondered.

Moving the n-1 disks from the first peg to the second requires H_{n-1} moves. Then, use one move to transfer the bottom disk from the first peg to the last. Finally, the n-1 disks are moved from the middle peg to the last. So,

$$H_n = 2H_{n-1} + 1$$

The base cases are $H_0 = 0$ and $H_1 = 1$. This tells us a lot but we would have to go through a lot of recursive applications of the rule to determine the number of moves for a specific n. We need a closed form formula where there is no H_n on the right side of the equation. To do this, we will try repeated substitution:

$$\begin{aligned} H_n &= 2H_{n-1} + 1 \\ &= 2(2H_{n-2} + 1) + 1 = 2^2 * H_{n-2} + 2 + 1 \\ &= 2^2 (2H_{n-3} + 1) + 2 + 1 = 2^3 * H_{n-3} + 2^2 + 2 + 1 \end{aligned}$$

The pattern is:

$$H_n = 2^i * H_{n-i} + 2^{i-1} + 2^{i-2} + \ldots + 2^2 + 2^1 + 2^0$$

We can set n = i (we will prove this pattern is correct using induction in a minute).

$$H_n = 2^i H_0 + 2^{n-1} + 2^{n-2} + \ldots + 2^2 + 2 + 1$$

$H_0 = 0$ so the formula is $2^{n-1} + 2^{n-2} + \ldots + 2^2 + 2 + 1$. This is a well known summation formula that can easily be proven by induction to equal $2^n - 1$. Therefore, the number of moves required for n disks is $2^n - 1$.

We need to prove this formula is correct using induction. An inductive proof is *always* an essential part of a guess or repeated substitution process. P(n) denotes the number of moves required to solve the Towers of Hanoi with n disks is $H_n = 2^n - 1$.

i) base case: prove that P(0) is true: $2^0 - 1 = 0 = H_0$; P(1) = $2^1 - 1 = 1 = H_1$.

ii) induction hypothesis is to assume P(k): $H_k = 2^k - 1$ and show P(k+1):

$H_{k+1} = 2^{k+1} - 1$.

PROOF:

We know from the recurrence relation ($H_n = 2H_{n-1} + 1$) that $H_{k+1} = 2 * H_k + 1$. We use the inductive hypothesis to substitute for H_k. This gives us

$$\begin{aligned} H_{k+1} &= 2 * (2^k - 1) + 1 \\ &= 2^{k+1} - 2 + 1 \\ &= 2^{k+1} - 1 \end{aligned}$$

P(k+1) is true when P(k) is true, and therefore P(n) is true for all natural numbers.

The ancient myth of this puzzle tells that there is a tower in Hanoi where monks are transferring 64 gold disks from one peg to another according to the rules of the puzzle. They take one second to move one disk. The myth says that the world will end when they finish the puzzle.

$2^{64} - 1 = 1{,}446{,}744{,}073{,}709{,}551{,}615$ seconds

So this will take about 500 billion years to solve the puzzle.

Suppose that in addition to the requirement that a larger disk can never be placed on a smaller disk, one can only move a disk from one pole to an adjacent pole. Let a_n = the minimum number of moves needed to transfer a tower of n disks from pole A to pole C.

Find a_1, a_2, a_3.
Define a recurrence relation expressing a_k in terms of a_{k-1} for all integers $k >= 2$.

Analyzing Recursive Programs

`Does a recursive factorial function run in O(n!) time?`

Solving recurrence relations is the key to analyzing recursive programs. We need a formula that represents the number of recursive calls and the work done in each call as the program converges on the base case. We will then use this formula to determine an upper bound on the program's worst-case performance. In terms of the analysis of algorithms, a recurrence relation has the following definition:

> A **recurrence relation** expresses the running time of a recursive algorithm. This includes how many recursive calls are generated at each level of the recursion, how much of the problem is solved by each recursive call, and how much work is done at each level.

We have some unknown running time for function F that is defined in terms of F's argument n (since this controls the number of recursive calls). We call this $T_F(n)$. The value of $T_F(n)$ is established by an inductive definition based on the size of argument n. There are two cases:

1) n is sufficiently small that no recursive calls are made. This case corresponds to a base case in an inductive definition of $T_F(n)$.
2) n is sufficiently large that recursive calls are made. However, we assume that whatever recursive calls F makes, they will be made with smaller arguments. This case corresponds to an inductive step in the definition of $T_F(n)$.

The recurrence relation for $T_F(n)$ is derived by examining the code of F and defining the running times for the two cases stated above. We then solve the recurrence relation to derive a Big-Oh expression.

Example 4

```
int fact( int n)
{
        1)      if (n <= 1)
        2)              return(1);
                else
        3)              return(n * fact(n-1));
}
```

We will use T(n) to represent the unknown running time of this function with n as the size of the argument. For the base case of the inductive definition of T(n), we will take n to equal 1, since no recursive calls are made in this case. When n = 1, we execute only lines 1 and 2, each of which has a constant running time which we will just designate as some time value a.

When n > 1, the condition of line 1 is false so we execute lines 1 and 3. Line 1 takes some constant time value b and line 3 takes T(n-1) since we are making a recursive call. If the running time of fact = T(n), then the running time of a call to T with (n-1) as the argument must be T(n-1).

base case: T(1) = a
induction: T(n) = b + T(n-1) for n > 1

Now we must solve this recurrence relation, i.e., come up with a closed form formula. We can try just enumerating a few to see if there is a pattern:

T(1) = a
T(2) = a + b
T(3) = b + T(2) = b + (a+b) = a + 2b
T(4) = b + T(3) = b + (a+2b) = a + 3b
T(n) = a + (n-1)b for all n >= 1

To be complete, we need to do an inductive proof: P(n) denotes the running time of fact is T(n) = a + (n-1)b. (the recurrence relation is T(1) = a; T(n) = b + T(n-1)).

i) base case: prove that P(1) is true; T(1) = a + 0*b = a; the base case of our inductive definition states that T(1) = a.

ii) induction hypothesis is to assume P(k): T(k) = a + (k-1)b and show P(k+1): T(k+1) = a + kb.

PROOF:

We know from the recurrence relation that T(k+1) = b + T(k). We use the inductive hypothesis to substitute for T(k). This gives us

T(k+1) = b + T(k)
= b + a + (k-1)b
= b + a + kb - b
= a + kb

P(k+1) is true when P(k) is true, and therefore P(n) is true for all natural numbers.

Now that we know the closed form formula for this recurrence relation, we use this formula to determine the complexity. T(n) = a + (n-1)b = a + bn - b has a complexity of O(n). This makes sense: all it says is that to compute n!, we make n calls to fact, each of which requires O(1) time.

An important point is that categories of recursive functions will all have the same recurrence relations, and therefore the same complexity. Any recursive function with the same basic form as fact will also be O(n). For example:

```
int Sum(int n)
{
        if (n == 1)
                return(1);
        else
                return(n + sum(n - 1));
}
```

Example 5

Consider a recursive selection sort algorithm:

```
        void SelectionSort(int A[], int i, int n)
        {
                int j, small, temp;

1)              if (i < n) {
2)                      small = i;
3)                      for (j=i+1, j <= n, j++) {
4)                              if (A[j] < A[small])
5)                                      small = j; }
6)                      temp = A[small];
7)                      A[small] = A[i];
8)                      A[i] = temp;
9)                      SelectionSort(A, i+1, n)
                }
        }
```

Parameter n indicates the size of the array; parameter i tells us how much of the array is left to sort, specifically A[i..n]. So, a call to SelectionSort(A,1,n) will sort the entire array through recursive calls.

We now develop a recurrence relation. Note that the size of the array to be sorted is equal to n-i+1. We will call this value (the size of the array) m. There is 1 base case (m=1); in this case, only line 1 is executed taking some constant amount of time which we will call a. Note that m=0 is not a base case because the recursion converges down to a list with one element; when i=n, m = 1.

The inductive case is for m > 1: this is when recursive calls are made. Lines 2, 6, 7, and 8 each take a constant amount of time. The for loop of lines 3, 4 and 5 will execute m times (where m = n-i+1). So a recursive call to this function will be dominated by the time it takes to execute the for loop m times which we shall designate O(m). The time for the recursive call of line 9 is T(m-1). So, the inductive definition of recursive SelectionSort:

$$T(1) = a$$
$$T(m) = T(m-1) + O(m)$$

To solve this recurrence relation, we first get rid of the Big-Oh expression by substituting the definition of Big-Oh: (f(n) = O(g(n)) if f(n) <= C * g(n)) so we can substitute C*m for O(m):

$$T(1) = a$$
$$T(m) = T(m-1) + C*m$$

Now, we can either try repeated substitutions or just enumerate a few and look for a pattern. Let's try repeated substitution:

$$\begin{aligned} T(m) &= T(m-1) + C*m \\ &= T(m-2) + 2Cm - C && \text{because } T(m-1) = T(m-2) + C(m-1) \\ &= T(m-3) + 3Cm - 3C && \text{because } T(m-2) = T(m-3) + C(m-2) \\ &= T(m-4) + 4Cm - 6C && \text{because } T(m-3) = T(m-4) + C(m-3) \\ &= T(m-5) + 5Cm - 10C && \text{because } T(m-4) = T(m-5) + C(m-4) \\ &\ldots \\ &= T(m-j) + jCm - (j(j-1)/2)C \end{aligned}$$

To get a closed form formula we let j = m-1. We do this because our base case is T(1). If we were to continue the repeated substitution down to the last possible substitution, we want to stop at T(1) because T(0) is really not the base case that the recursion converges on. (Note that we have to do an inductive proof later to allow us to do this substitution, but for now):

$$\begin{aligned} T(m) &= T(1) + m^2C - Cm - ((m-1)(m-2)/2)C \\ &= a + m^2C - Cm - (m^2C - 3Cm + 2C)/2 \\ &= a + (2m^2C - 2Cm - m^2C + 3Cm - 2C)/2 \\ &= a + m^2C + Cm - 2C \\ &= a + C(m-1)(m+2)/2 \end{aligned}$$

We need to verify this by doing an inductive proof, but we will leave that as an exercise for the reader.

So, finally we have a closed form formula $T(m) = a + C(m-1)(m+2)/2$. The complexity of this formula is $O(m^2)$ which is the same complexity for iterative selection sort, so doing it recursively did not save us any time. (In fact, the recursive solution is less desirable because of the overhead associated with recursion.) The above recurrence relation occurs with any recursive subprogram that has a single loop of some form, and then does a recursive call.

Example 6

MergeSort is a very efficient sorting algorithm that uses a "Divide and Conquer" strategy, i.e., the problem of sorting a list is reduced to the problem of sorting two smaller lists.. Suppose that you have a sorted array A with r elements and another sorted array B with s elements. Merging is an operation that combines the elements of A with the elements of B into one large sorted array C with r+s elements. For the following example, we will assume that we have a merge function that performs this task in O(n) time.

MergeSort is based on this merging algorithm. What happens is we set up a recursive function that divides a list into two halves; we then sort the two halves and merge them together using the algorithm above. The question is how do you sort the two halves? We sort them by using Mergesort, of course, i.e., by calling it recursively. The pseudocode for MergeSort:

```
MergeSort(L);

1)      if (length of L > 1) { then
2)              split list into 1st part, second part
3)              MergeSort(first part) ;
4)              MergeSort(second part);
5)              Merge first part & second part into sorted list;
        }
```

Note: we will also assume that we have a Split function that runs in O(n).

The recursive calls continue dividing the list into halves until each half contains only one element; obviously a list of one element is sorted. The algorithm then merges these smaller halves into larger sorted halves until we have one large sorted list. For example, if the list contains the following integers:

7 9 2 4 3 6 1 8

The first thing that happens is we split the list into 2 parts: (MS #0 = original call)

7 9 2 4 3 6 1 8

Then we call Mergesort on the first part which again splits the list into two parts: (MS #1)

7 9 2 4

We call Mergesort again on the first part which splits the list into two parts: (MS #2)

7 9

When we call MergeSort the next time, the if condition fails and we return up a level of recursion to call #2. Now we execute the next line in call #2: MergeSort(second part) which executes on the list containing 9. Again the if condition fails, and we return back up to call #2. We execute the next line which Merges the lists containing 7 and 9. This merges these 2 "sorted" lists together. We have completed recursive call #2. Now we return up to recursive call #1 and find we are on the list containing 2 and 4. We execute the next line in recursive call #1 and call MergeSort(second part) on this list and end up with separate lists of 2 and 4. We end up merging these two sorted lists into one list containing 2 and 4 just as we did with the 7 and 9. Once that work is complete, we return to recursive call #1 and execute the Merge on the two lists each with two elements. The first part of the list is sorted: 2 4 7 9.

We return to recursive call #0 (the original call) and execute the next line: MergeSort(second part). This sends us through another cycle of recursion just like what we did for the first part of the list. When we have finished, we have a second sorted list: 1 3 6 8.

The last line of call #0 Merges these two lists of four elements into one sorted list.

To analyse the complexity of MergeSort, we need to define a recurrence relation. The base case is when we have a list of 1 element and only line 1 of the function is executed. Thus, the base case is constant time, O(1).

If the test of line 1 fails, we must execute lines 2-5. The time spent in this function when the length of the list > 1 is the sum of the following:

1) O(1) for the test on line 1
2) O(n) for the split function call on line 2
3) T(n/2) for recursive call on line 3
4) T(n/2) for recursive call on line 4
5) O(n) for the merge function call on line 5

If we drop the O(1)'s and apply the summation rule, we get $2T(n/2) + O(n)$. If we substitute constants for the Big-Oh notation:

$T(1) = a$
$T(n) = 2T(n/2) + bn$

To solve this recurrence relation, we will enumerate a few values. We will stick to n's that are powers of two so things divide evenly:

T(2)	= 2T(1) + 2b	=	2a + 2b
T(4)	= 2T(2) + 4b = 2(2a + 2b) + 4b	=	4a + 8b
T(8)	= 2T(4) + 8b = 2(4a + 8b) + 8b	=	8a + 24b
T(16)	= 2T(8) + 16b = 2(8a + 24b) + 16b	=	16a + 64b

There is obviously a pattern but it is not as easy to represent as the others have been. Note the following relationships:

value of n:	2	4	8	16
coeff of b:	2	8	24	64
ratio:	1	2	3	4

So, it appears that the coefficient of b is n times another factor that grows by 1 each time n doubles. The ratio is $\log_2 n$ because $\log_2 2 = 1$, $\log_2 4 = 2$, $\log_2 8 = 3$, etc. Our "guess" for the solution of this recurrence relation is $T(n) = an + bn \log_2 n$.

We could have used repeated substitution in which case we would have the following formula: $T(n) = 2^i T(n/2^i) + ibn$. If we let $i = \log_2 n$, we end up with $n*T(1) + bn \log_2 n = an + bn \log_2 n$ (because $2^{\log_2 n} = n$). Of course, we have to do an inductive proof to finish (refer to Aho, Ullman in the bibliography, p. 142 for the proof). Finally, the complexity of $an + bn \log_2 n$ is $O(n \log_2 n)$ which is considerably faster than selection sort. You will often see this recurrence relation in "divide & conquer" sorts.

Final Word on Recurrence Relations

We have presented some of the more important recurrence relations that you will find in analysing recursive subprograms. There are others that you may come across, many of which are not quite as easy to solve as the ones presented here. In those cases, the technique of repeated substitution is limited because a pattern may just not be obvious. For example, consider the function for generating Fibonacci numbers:

```
int fib(int n) {
        if ((n == 1) || (n == 0))
                return(1);
        else
                return(fib(n-1) + fib(n-2));
}
```

We can write the recurrence relation as follows:

$T(n) = a$
$T(n) = T(n\text{-}1) + T(n\text{-}2) + b$

But watch what happens when we begin expanding the right hand side:

$T(n) = T(n\text{-}1) + T(n\text{-}2) + b$
$T(n-1) = T(n\text{-}2) + T(n\text{-}3) + b$
$T(n) = 2T(n\text{-}2) + T(n\text{-}3) + 2b$
$T(n-2) = T(n\text{-}3) + T(n\text{-}4) + b$
$T(n) = 3T(n\text{-}3) + 2T(n\text{-}4) + 4b$
$T(n-3) = T(n\text{-}4) + T(n\text{-}5) + b$
$T(n) = 5T(n\text{-}4) + 3T(n\text{-}5) + 7b$
…

Clearly, this approach is getting us nowhere. The recurrence above is a special kind of beast known as an homogeneous second order linear difference equation with constant coefficients (*wow!*), and there is a technique for solving such things. If you want to know more about the pure math end of solving recurrence relations, refer to Rosen, Chapter 5.

Bibliography

A. Aho, J. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Reading, MA: Addison-Wesley, 1974.

A. Aho, J.D. Ullman, *Foundations of Computer Science*, New York: W.H. Freeman, 1992.

T. Cormen, C. Leiserson, R. Rivest, *Introduction to Algorithms*, New York: McGraw-Hill, 1991.

S. Epp, *Discrete Mathematics with Applications*, Belmont, CA: Wadsworth, 1990.

R. McEliece, R. Ash, C. Ash, *Introduction to Discrete Mathematics*, New York: Random House, 1989.

J. Mott, A Kandel, T. Baker, *Discrete Mathematics for Computer Scientists*, Reston, VA: Reston Publishing, 1983.

K. Rosen, *Discrete Mathematics and its Applications 2nd Ed.*, New York: McGraw-Hill, 1991.

Historical Notes

Aho, Hopcroft and Ullman advocated the use of recurrence relations to describe the running time of recursive algorithms. As mentioned in a previous handout, Eduard Lucas (1842-1891) created the Towers of Hanoi game in 1883. It became a very popular puzzle in the late 19th century (he made up the myth too). Selection sort was first described in an article by Daniel Goldenberg in 1952 ("Time Analyses of Various Methods of Sorting Data" in an MIT Computer Lab memo (M-1680)) where it came in second to last place (Mergesort was the fastest and bubble was the slowest). The first Mergesort was devised by John Von Neumann for the EDVAC in 1945.