CSC 6013 Algorithms and Discrete Structures   Michael Bradley

Class Notes for Week 6: Recursive algorithms and recurrence relations;

solving recurrence relations by back substitution and the master method

Read these sections from the textbook: §4.3 to 4.5

This week, we look at:

Algorithms:

- understanding recursive algorithms
- understanding recurrence relations
- determining asymptotic performance using back substitution
- determining asymptotic performance using the master method
- examples: depth first search (DFS), n-factorial, Towers of Hanoi, binary search, max element in array

Recursive Algorithms + Recurrence relation + Back-Substitution

A **recursive algorithm** is an algorithm that calls itself, usually to solve one or more smaller version(s) of the same problem.

A **recurrence relation** is a pair of equations that defines a function on the non-negative integers.

> The first equation is a **recursive equation** that specifies the value of the function for any positive integer n in terms of a computation involving the value of the same function evaluated at some smaller value of the variable.
>
> The second equation is the **stopping condition** or **base case** that specifies the value of the function for one or more specific small non-negative integers.

example 0:

Depth First Search – This is a recursive algorithm for processing/visiting every vertex in a graph exactly once.

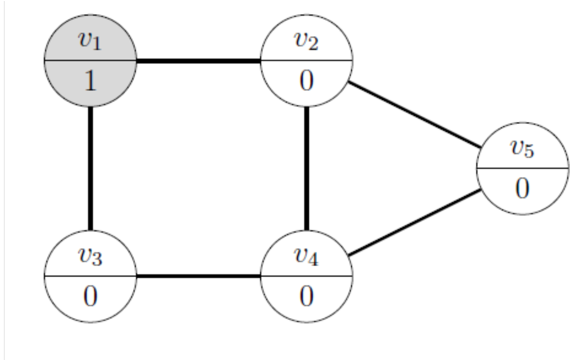
See the notes from module 5 for full details.

```
# Input: A graph G = (V, E) and a global variable, count.
# Output: A graph G = (V,' E') where E' = E and V' = V
#       except that the nodes of V' are marked with integers
#       that indicate in what order they were visited.
def DFS(G):
        global count
        count = 0

# Mark every vertex in V with 0.
for v in G:
        v.visited = 0

for v in G:
        if v.visited == 0:
                DFSVisit (v)
```

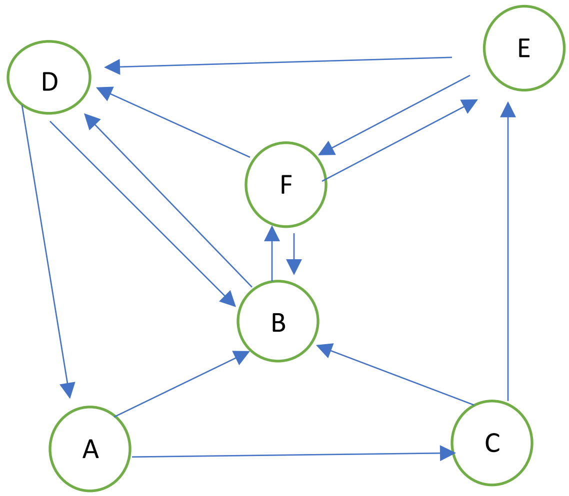
The DFSVisit algorithm is as follows:

```
# Input: v is an unvisited vertex.
# Output: All descendants of v have been visited.
def DFSVisit(v):
        count = count + 1
        v.visited = count
        for u in v.getAdjacent():
                if u.visited == 0:
                        DFSVisit(u)  #This recursive algorithm calls itself.
```

Let's look at an example trace for DFS starting at v1. (Again, full details are in the module 5 class notes.)



Second example of DFS using graph from worksheet 5. Start at vertex A.



example 1: For any non-negative integer n, n-factorial is defined as

$$n! = \begin{cases} 1 & if\ n = 0 \\ n \cdot (n-1)! & if\ n > 0 \end{cases}$$

Here is an implementation of this algorithm in python.

```
1    # Input:  n ≥ 0
2    # Output: n! is returned
3    def Factorial(n):
4        if n == 0:
5            return 1
6        else:
7            return n * Factorial(n-1)
```

If we count only the number of times the multiplication is performed, based on lines 5 and 7, we get the work for this algorithm expressed as a recurrence relation

$$T(n) = 1 + T(n-1)\ , T(0) = 0$$

Using back-substitution we can rewrite this a couple of times,

$$T(n) = 1 + T(n-1)$$

$$= 1 + [1 + T(n-2)]\ = 2 + T(n-2)$$

$$= 2 + [1 + T(n-3)] = 3 + T(n-3)$$

discover the general pattern,

$$= k + T(n-k)$$

and take it to the base case (T(0) = 0) when n – k = 0, or n = k,

$$= n + T(0)$$

$$= n$$

So, the algorithm for n-factorial is in the asymptotic class O(n).

example 2: The **Towers of Hanoi**

Input: A board with three pegs p1, p2, p3 and a stack of n disks {d1 < d2 < d3 < … < dn} stacked on the left most peg.
Output: A sequence of moves that move the disks from p1 to p3 obeying the rules:

1. Only one disk can be moved at a time.
2. Only the top most disk on any peg can be moved.
3. di cannot be placed on top of dj if i > j. (In other words, you can never place a larger disk on top of a smaller disk.)

A recursive algorithm would be:

1. Recursively move n-1 disks from p1 to p2 using p3 as an auxiliary peg.
2. Move the largest disk from p1 to p3.
3. Recursively move n-1 disks from p2 to p3 using p1 as an auxiliary peg.

If we count only the number of times a disk is moved, we get the work for this algorithm expressed as a recurrence relation

$$T(n) = T(n-1) + 1 + T(n-1)\,, T(1) = 1$$

or in other words

$$T(n) = 2T(n-1) + 1, T(1) = 1$$

Using back-substitution we can rewrite this a couple of times,

$T(n) = 1 + 2T(n-1)$

$= 1 + 2[1 + 2T(n-2)] = 1 + 2 + \ 4T(n-2)$

$= 1 + 2 + 4[1 + 2T(n-3)] = 1 + 2 + 4 + 8T(n-3)$

$= 2^0 + 2^1 + 2^2 + 2^3T(n-3)$

discover the general pattern,

$= 2^0 + 2^1 + 2^2 + \cdots + 2^{k-1} + 2^kT(n-k)$

and take it to the base case (T(1) = 1) when n – k = 1, so n - 1 = k and n – 2 = k - 1,

$= 2^0 + 2^1 + 2^2 + \cdots + 2^{n-2} + 2^{k-1}T(1)$

$= 2^0 + 2^1 + 2^2 + \cdots + 2^{n-2} + 2^{n-1}$

= $2^n - 1$

So, the algorithm for Towers of Hanoi is in the asymptotic class $O(2^n)$.

example 3: **Binary Search**

Binary search is a classic recursive algorithm that is used to search for a search key in a sorted array.

Given an array of n entries, we examine the middle slot, A[n/2], where one of three things happens:

either array entry A[n/2] matches the search key and we are done with a successful search,
or the search key is < A[n/2] and we repeat the process with the left half of the array,
or the search key is > A[n/2] and we repeat the process with the right half of the array.

In the worst case,

at each step we do one unit of work,
eliminate half of the remaining array entries,
and repeat the search on a problem that is half as big as it was one unit of work ago.

Here is one implementation of this recursive algorithm for binary search in Python code.

```
1    # Input: An array A in sorted order, end > start > 0,
2    #     and key k.
3    # Output: Index i such that A[i] = k, or None if no
4    #     match is found.
5    def BinarySearch(A, start, end, k):
6        m = math.floor((end + start)/2)
7        if start > end:
8            return None
9        elif A[m] == k:
10           return m
11       elif A[m] > k:
12           return BinarySearch(A, start, m-1, k)
13       else:
14           return BinarySearch(A, m+1, end, k)
```

If we count only the number of times we compare the search key with the middle slot in the remaining portion of the array (lines 9 and 11), we get the work for this algorithm expressed as a recurrence relation

$$T(n) = 2 + T\left(\frac{n}{2}\right), T(1) = 1$$

Back-Substitution:

$$T(n) = 2 + T\left(\frac{n}{2}\right)$$
$$= 2 + \left[2 + T\left(\frac{n}{4}\right)\right] = 4 + T\left(\frac{n}{4}\right)$$
$$= 4 + \left[2 + T\left(\frac{n}{8}\right)\right] = 6 + T\left(\frac{n}{8}\right)$$
$$= 6 + \left[2 + T\left(\frac{n}{16}\right)\right] = 8 + T\left(\frac{n}{16}\right)$$
$$= 8 + \left[2 + T\left(\frac{n}{32}\right)\right] = 10 + T\left(\frac{n}{32}\right)$$

in general, after k steps we have

$$= 2k + T\left(\frac{n}{2^k}\right)$$

This reaches the base case, T(1)=1, when $n = 2^k$, k = lg(n)

$$= 2\lg(n) + T(1)$$
$$= 2\lg(n) + 1$$

So, the binary search algorithm is in the asymptotic class $O(\lg(n))$

example 4: maximum element in an array

```
1	# Input: An array A and an integer n indicating how many elements in A to consider
2	# Output: Index i such that A[i] >= A[k] for all k<n
3	def MaxElement(A, n):
4		if n == 1:
5			return 0
6		else:
7			best = MaxElement(A, n-1)
8			if (A[n-1] > A[best]):
9				return n-1
10			else:
11				return best
```

If we count only the number of times we compare two array elements, we have one explicit comparison in line 8 plus whatever comparisons are done in the recursive call in line 7. We get the work for this algorithm expressed as a recurrence relation

$$T(n) = 1 + T(n-1), T(1) = 0$$

This is the same recurrence relation as the algorithm for n-factorial.

So, the algorithm for MaxElement is also in the asymptotic class O(n).

Now we will practice the back-substitution method with a few recurrence relations without the code or algorithm that produced them.

example 5: If the work performed by an algorithm in the worst case is given by the recurrence relation

$$T(n) = n + T(n-1) \ , T(1) = 1$$

use back substitution to determine the algorithm's asymptotic class Big-Oh.

To understand the formula in words, the work to solve a problem of any size (here designated by T(n)) is the size of the problem (n) plus the work to solve a problem one size smaller (T(n-1)). With specific number, T(10) = 10 + T(9), and T(9) = 9 + T(8), and also T(4) = 4 + T(3).

For large values of n,

$$T(n) = n + T(n-1)$$

Find an expression for T(n-1) in terms of the next smaller integer, n-2, and substitute this for T(n-1) in the above equation.

$$T(n) = n + [(n-1) + T(n-2)]$$

Very important!!!!! Note that we have (n-1) + inside the square brackets because we are substituting for T(n-1).

This simplifies (removing parentheses and gathering like terms but NOT performing the arithmetic) as

$$T(n) = n + (n-1) + T(n-2)$$

Since n-1 is large, n-2 is also large, and we can use the recurrence relation to find an expression for T(n-2) in terms of the next smaller integer, n-3, and substitute this for T(n-2) in the above equation.

$$T(n) = n + (n-1) + [(n-2) + T(n-3)]$$

which simplifies as

$$T(n) = n + (n-1) + (n-2) + T(n-3)$$

What is the pattern? For any integer k < n,

$$T(n) = n + (n-1) + (n-2) + \cdots + \big(n-(k-1)\big) + T(n-k)$$

We reach the base case, T(1) = 1, when n - k = 1, which happens when n-(k-1) = n-k+1=2

$$T(n) = n + (n-1) + (n-2) + \cdots + 2 + T(1)$$

Since T(1) = 1, the summation is all the integers from n down to 1, so

$$T(n) = n + (n-1) + (n-2) + \cdots + 2 + 1$$

From one of our summation formulas, we recognize this as

$$T(n) = \frac{n(n+1)}{2}$$

which simplifies to

$$T(n) = \frac{1}{2}n^2 + \frac{1}{2}n$$

so

$$T(n) = O(n^2)$$

----

example 6: If the work performed by an algorithm in the worst case is given by the recurrence relation

$$T(n) = 1 + 2T\left(\frac{n}{2}\right), T(1) = 1$$

use back substitution to determine the algorithm's asymptotic class Big-Oh.

To understand the formula in words, the work to solve a problem of any size (here designated by T(n)) is 1 unit of work plus twice the work to solve a problem half as big $\left(T\left(\frac{n}{2}\right)\right)$. With specific numbers, T(10) = 1 + 2T(5) , and T(64) = 1 +2 T(32), and also T(4) = 1 + 2T(2).

For large values of n,

$$T(n) = 1 + 2T\left(\frac{n}{2}\right)$$

Find an expression for $T\left(\frac{n}{2}\right)$ in terms of the next smaller problem size, n/4, and substitute this for $T\left(\frac{n}{2}\right)$ in the above equation.

$$T(n) = 1 + 2\left[1 + 2T\left(\frac{n}{4}\right)\right]$$

This simplifies (removing parentheses and gathering like terms but NOT performing much arithmetic) as

$$T(n) = 1 + 2 + 4T\left(\frac{n}{4}\right)$$

Since n is large, so are n/2 and n/4, and we can use the recurrence relation to find an expression for T(n/4) in terms of the next smaller problem size, n/8, and substitute this for T(n/4) in the above equation.

$$T(n) = 1 + 2 + 4\left[1 + 2T\left(\frac{n}{8}\right)\right]$$

which simplifies as

$$T(n) = 1 + 2 + 4 + 8T\left(\frac{n}{8}\right)$$

To help us see the pattern, we will write 8 , 4, 2, and 1 as powers of 2:

$$T(n) = 2^0 + 2^1 + 2^2 + 2^3 T\left(\frac{n}{2^3}\right)$$

What is the pattern now? For any integer k,

$$T(n) = 2^0 + 2^1 + 2^2 + \cdots + 2^{k-1} + 2^k T\left(\frac{n}{2^k}\right)$$

We reach the base case, T(1) = 1, when $\frac{n}{2^k} = 1$ or, in other words, when $n = 2^k$ or $k = \lg(n)$.

What if n is not an integer power of 2? For simplicity, we will assume that n **IS** an integer power of 2. (This is a legitimate assumption, but we will skip the high-powered mathematical theory that justifies our using this assumption.) So,

$$T(n) = 2^0 + 2^1 + 2^2 + \cdots + 2^{k-1} + 2^{\lg(n)-1} + 2^{\lg(n)} T(1)$$

Since $T(1)$ = 1, the summation is

$$T(n) = 2^0 + 2^1 + 2^2 + \cdots + 2^{k-1} + 2^{\lg(n)-1} + 2^{\lg(n)}$$

Since $2^{\lg(n)} = n$ and $2^{\lg(n)-1} = \frac{n}{2}$, our summation is

$$T(n) = 1 + 2 + 4 + \cdots + \frac{n}{2} + n$$

so $T(n) = 2n - 1 = O(n)$.

example 7: If the work performed by an algorithm in the worst case is given by the recurrence relation

$$T(n) = n^4 + T\left(\frac{n}{2}\right),T(1) = 0$$

use back substitution to determine the algorithm's asymptotic class Big-Oh.

To understand the formula in words, the work to solve a problem of any size (T(n)) is the 4th power of the problem size plus the work to solve a problem half as big $\left(T\left(\frac{n}{2}\right)\right)$. With specific number, $T(10) = 10^4 + T(5)$, and $T(64) = 64^4 + T(32)$, and also $T(4) = 4^4 + T(2)$,

For large values of n,

$$T(n) = n^4 + T\left(\frac{n}{2}\right)$$

Find an expression for $T\left(\frac{n}{2}\right)$ in terms of the next smaller problem size, n/4, and substitute this into the above equation.

$$T(n) = n^4 + \left[\left(\frac{n}{2}\right)^4 + T\left(\frac{n}{4}\right)\right]$$

This simplifies (removing parentheses and gathering like terms but NOT performing much arithmetic) as

$$T(n) = n^4 + \left(\frac{n}{2}\right)^4 + T\left(\frac{n}{4}\right)$$

Since n is large, so are n/2 and n/4, and we can use the recurrence relation to find an expression for

T(n/4) in terms of the next smaller problem size, n/8, and substitute this into the above equation.

$$T(n) = n^4 + \left(\frac{n}{2}\right)^4 + \left[\left(\frac{n}{4}\right)^4 + T\left(\frac{n}{8}\right)\right]$$

which simplifies as

$$T(n) = n^4 + \left(\frac{n}{2}\right)^4 + \left(\frac{n}{4}\right)^4 + T\left(\frac{n}{8}\right)$$

To help us see the pattern, we will write 8 , 4, 2, and 1 as powers of 2:

$$T(n) = \left(\frac{n}{2^0}\right)^4 + \left(\frac{n}{2^1}\right)^4 + \left(\frac{n}{2^2}\right)^4 + T\left(\frac{n}{2^3}\right)$$

What is the pattern now? Assuming n is an integer power of 2, for any integer $k \leq \lg(n)$,

$$T(n) = \left(\frac{n}{2^0}\right)^4 + \left(\frac{n}{2^1}\right)^4 + \left(\frac{n}{2^2}\right)^4 + \cdots + \left(\frac{n}{2^{k-1}}\right)^4 + T\left(\frac{n}{2^k}\right)$$

We reach the base case, T(1) = 1, when $\frac{n}{2^k} = 1$ or, in other words, when $n = 2^k$ or $k = \lg(n)$. So,

$$T(n) = \left(\frac{n}{2^0}\right)^4 + \left(\frac{n}{2^1}\right)^4 + \left(\frac{n}{2^2}\right)^4 + \cdots + \left(\frac{n}{2^{\lg(n)-1}}\right)^4 + T\left(\frac{n}{2^{\lg(n)}}\right)$$

There are lg(n) terms that are raised to the 4th power. Each of these terms is ≤ n^4, and the final term is T(1) = 1, so

$$T(n) \leq n^4 \cdot \lg(n) + 1$$

So, $T(n) = O(n^4 \cdot \lg(n))$.

This is a “loose” upper bound. We could improve on it if we took more time to add up the lg(n) terms. We will revisit this recurrence in example 12.

example 8: The worst-case work for a recursive algorithm is given by the recurrence relation

$T(n) = \lg(n) + T(n-1)$, $T(0) = 0$ for some constant c.

Back-Substitution:

$$T(n) = \lg(n) + T(n-1)$$
$$= \lg(n) + [\lg(n-1) + T(n-2)]$$
$$= \lg(n) + \lg(n-1) + [\lg(n-2) + T(n-3)]$$

In general, after k steps,

$$= \lg(n) + \lg(n-1) + \lg(n-2) + \cdots + \lg(n-(k-1)) + T(n-k)$$

We reach the base case, T(1)=c, when $n - k = 0$, which means $n - (k-1) = n - k + 1 = 1$

$$= \lg(n) + \lg(n-1) + \lg(n-2) + \cdots + \lg(1) + T(0)$$

We do not have a summation formula to solve this summation, but we can see that every term is $\leq \lg(n)$ and there are n terms in the sum.

So, we have a loose upper bound of

$$T(n) = n \cdot \lg(n)$$

BREAK

Master Method

A deep mathematical theory behind the scenes provides us with a three-part rule for determining the asymptotic class of an algorithm whose workload is given by a recurrence relation of the form

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

The theorem (or rule) is called the

Master Method:
For $T(n) = a \cdot T\left(\frac{n}{b}\right) + n^d$

1) if $a < b^d$ then $T(n) = O(n^d)$
2) if $a = b^d$ then $T(n) = O(n^d \cdot \lg(n))$
3) if $a > b^d$ then $T(n) = O\left(n^{\log_b a}\right)$

example 9: If For $T(n) = 4T\left(\frac{n}{3}\right) + n^3$

then *a* = 4, *b* = 3, *d* = 3

so, comparing *a* to b^d shows

$$4 < 3^3 = 27$$

so, case 1 applies and $T(n) = O(n^3)$.

----

example 10: If For $T(n) = 4T\left(\frac{n}{2}\right) + n^2$

then *a* = 4, *b* = 2, *d* = 2

so, comparing *a* to b^d shows

$$4 = 2^2$$

so, case 2 applies and $T(n) = O(n^2\lg(n))$.

----

example 11: If For $T(n) = 4T\left(\frac{n}{2}\right) + n$

then *a* = 4, *b* = 2, *d* = 1

so, comparing *a* to b^d shows

$$4 > 2^1$$

so, case 3 applies and $T(n) = O\left(n^{\log_2 4}\right) = O(n^2)$.

Let us re-do the analysis in example 6 using the Master Method.

example 12

If the work performed by an algorithm in the worst case is given by the recurrence relation

$$T(n) = 2T\left(\frac{n}{2}\right) + 1\,, T(1) = 1$$

use back substitution to determine the algorithm's asymptotic class Big-Oh.

We can ignore the base case and rewrite the "+1" as $+n^0$ to obtain

$$T(n) = 2T\left(\frac{n}{2}\right) + n^0$$

Using the master method, we have *a* = 2, *b* = 2, *d* = 0

so, comparing *a* to b^d shows

$$2 > 2^0 = 1$$

so, case 3 applies and $T(n) = O\left(n^{\log_2 2}\right) = O(n^1) = O(n)$.

Let us re-do the analysis in example 7 using the Master Method.

example 13 = example 7 again

If the work performed by an algorithm in the worst case is given by the recurrence relation

$$T(n) = T\left(\frac{n}{2}\right) + n^4\ , T(1) = 0$$

use back substitution to determine the algorithm's asymptotic class Big-Oh.

$$T(n) = T\left(\frac{n}{2}\right) + n^4$$

Using the master method, we have *a* = 1, *b* = 2, *d* = 4

so, comparing *a* to b^d shows

$$1 < 2^4 = 16$$

so, case 1 applies and $T(n) = O(n^4)$.

When we used back-substitution to find the Big-Oh class of this algorithm, we concluded and $T(n) = O(n^4 \cdot \lg(n))$ which is bigger than $O(n^4)$.

----

example 14 = example 3 (Binary Search) again

$$T(n) = T\left(\frac{n}{2}\right) + 2, T(1) = 1$$

Master Method:

$$T(n) = T\left(\frac{n}{2}\right) + 2$$

Rewrite as

$$T(n) = 1 \cdot T\left(\frac{n}{2}\right) + 2n^0$$

We have *a* = 1, *b* = 2, *d* = 0

so, comparing *a* to b^d shows

$$1 = 2^0$$

so, case 2 applies and $T(n) = O(n^0 \cdot \lg(n)) = O(\lg(n))$.

Let's end this set of notes by solving a problem by both back-substitution and the master method.

example 15: The worst-case work for a recursive algorithm is given by the recurrence relation

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n\,, T(1) = c \text{ for some constant c.}$$

Back-Substitution:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$
$$= 2\left[2T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)\right] + n = 4T\left(\frac{n}{4}\right) + n + n$$
$$= 4\left[2T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)\right] + 2n = 8T\left(\frac{n}{8}\right) + 3n$$

in general

$$= 2^k T\left(\frac{n}{2^k}\right) + kn$$

which reaches the base case, T(1)=c, when $n = 2^k$, k = lg(n) [We assume n is an integer power of 2.]

$$= 2^{\lg(n)} T\left(\frac{n}{2^{\lg(n)}}\right) + n\lg(n) \quad = \quad nT(1) + n\lg(n)$$
$$= cn + n\lg(n)$$
$$= O(\mathrm{n}\lg(n))$$

Master Method:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Rewrite as

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n^1$$

We have *a* = 2, *b* = 2, *d* = 1

so, comparing *a* to b^d shows

$$2 = 2^1$$

so, case 2 applies and $T(n) = O(n^1 \cdot \lg(n)) = O(\mathrm{n}\lg(n))$.

THE END