Chapter 2

Recurrence Relations and Divide and Conquer Algorithms

2.1 Recursive Definition

Define a class of closely related objects in terms of the objects themselves.

Basis: Simple objects are defined
Inductive step: Larger objects are defined in terms of smaller ones.

Example:

$f(n) = n! = 1 \times 2 \times 3 \cdots n$

Basis: $1! = 1$
Induction: $n! = n \times (n-1)!$

Generates 1, 1, 2, 6, 24, 120, 720, 5040,...

```
int factorial (int n)
{
  if (n == 0)
      return 1;
  else
      return   n * factorial (n-1);
}
```

Example

f(n) = 1 if n = 0
f(n) = f(n-1) + 1/f(n-1) if n > 0
Generates 1, 2, 5/2, 29/10, 941/290, ...
A recurrence relation: f is defined in terms of itself.

- Recursive definition undesirable feature: To determine an element $s_i, 1 \le i \le n$ of $S = \{s_1, s_2, \cdots, s_n\}$, we must first compute the values of some or all the previous elements $s_1, s_2, \cdots, s_{i-1}$. This is computationally intensive.

2. 2 Anatomy of a Recursive Call

Let $f(n) = x^n$

f(n) = 1 if n =0
$f(n) = x \times x^{n-1}$ if n > 0

```
float power (float x, int n)      // assumes n is non-negative
{
  if (! n)
     return 1;                // if ( n == 0)
  else
     return  x * power(x, n-1);
}
```

The calls to the function for n = 4 is as follows;

call 1 $x^4 = x * x^3$

call 2 $x * x^2$
call 3 x * x
call 4 $x * x^0 = x * 1 = x$
call 5 1

or alternatively

```
call 1  power(x, 4)
call 2          power(x, 3)
call 3                power(x, 2)
call 4                      power(x, 1)
call 5                            power(x, 0)
call 5                            1
call 4                      x
call 3                x * x
call 2          x * x * x
call 1     x * x * x * x
```

The system keeps track of all calls on its run-time stack. Each line of code is assigned a number by the system and if a line is a function call, then the number is the return address. The address is used by the system to remember where to resume execution after the function has completed statement

```
main( )
{ . . .
  y = power( 5.6, 2);      // 136
  . . .
}
```

A trace of recursive calls is

```
call 1      power(5.6, 2)
call 2          power(5.6, 1)
call 3              power(5.6, 0)
call 3              1
call 2          5.6
call 1      31.36
```

When the function is invoked for the first time, four items are pushed onto the run-time stack:

- return address 136,
- actual parameters 5.6 and 2
- one location reserved for the value returned by power( ).

Non-recursive implementation:

```
float NonRecPower( float x, int n)
{
float result = 1;

    if (n > 0)
        for (result = x; n > 1; --n)
            result *= x;
    return result;
}
```

However, the recursive version:

- is intuitive since it is similar to the original definition, and original structure of the definition is retained.
- increases program readability.
- the code generally is shorter and more compact.

2.3 Solving Recurrence Relations

Needed for algorithm complexity. We associate with each function f( ) in a program, an unknown complexity C(n), where n is the size of the function's argument.

```
int factorial (int n)
{
  if (n == 0)          // 1
     return 1;         // 2
  else
     return  n * factorial (n-1);    // 3
}
```

Basis : C(n), n = 0 and the call to factorial executes lines 1 and 2, which needs a constant (call it o) operation. When n > 0, line 3 is executed which takes a constant time equal to b for multiplication plus C(n-1) for the recursive call to factorial. Hence for n >0, the complexity is b + C(n-1), and we can define C(n) by the following recurrence relation:

basis: $C(1) = a$
induction $C(n) = b + C(n-1)$ for $n > 1$

Apply the induction rule successively:

$C(2) = b + C(1) = a + b$

$C(3) = b + C(2) = a + 2b$
$C(4) = b + C(3) = a + 3b$

Find a pattern and determine the solution

$C(n) = a + (n-1)b$ for all $n >= 1$.

- Computing sample values then **finding a pattern** for the solution and finally proving that our solution is correct by an induction proof, is a common method of dealing with recurrences.

- **Repeated substitutions** is another method of solution. Since the recurrence relation holds true for all $n > 1$, then substitute for n the values n-1, n-2, ... 2, to get the set of equations. For the above example:

$$C(n) = b + C(n-1) \quad (1)$$
$$C(n-1) = b + C(n-2) \quad (2)$$
$$C(n-2) = b + C(n-3) \quad (3)$$
$$\ldots$$
$$C(2) = b + C(1) \quad (4)$$

Next substitute for C(n-1) from (2) into 1 to get

$$C(n) = 2b + C(n-2) \quad (5)$$

Now substitute (3) into (5) to get

$$C(n) = 3b + C(n-3)$$

Proceed, each time replacing C(n-i) by b + C(n-i-1), i =1, 2, . . ., etc until we get down to C(1). At this point, we have

$$C(n) = (n-1)b + C(1) = (n-1)b + a = nb + a - b$$

Note that since a and b are constants, the running of the factorial function is $\Theta(n)$.

We can generalize the above two main techniques for solving the recurrence relations to more complex problems, as follows.

2.4 Recurrence of the form:

$$C(1) = a$$
$$C(n) = C(n-1) + g(n) \quad \text{for } n > 1 \qquad (6)$$

$$\begin{aligned} C(n) &= C(n-1) + g(n) \\ &= C(n-2) + g(n-1) + g(n) \\ &= C(n-3) + g(n-2) + g(n-1) + g(n) \\ &\;\;\ldots \\ &= C(n-i) + g(n-i+1) + g(n-i+2) + \ldots + g(n-1) + g(n) \end{aligned}$$

Therefore

$$C(n) = C(n-i) + \sum_{j=0}^{i-1} g(n-j) \qquad (7)$$

Pick i = n-1, which gives n-i = 1 (base case).

$$C(n) = C(1) + \sum_{j=0}^{n-2} g(n-j) = a + \sum_{j=0}^{n-2} g(n-j)$$

- For factorial function, g(n) = b for all n,

$$C(n) = a + \sum_{j=0}^{n-2} b = a + (n-1)b$$

- Another special case of (6) is

  $C(1) = a$
  $C(n) = C(n-1) + bn \qquad \text{for } n > 1$

  Here $g(n) = nb$, and from (7) we have

  $$C(n) = a + \sum_{j=0}^{n-2} (n-j)b = a + nb + (n-1)b + (n-2)b + \ldots + 2b$$
  $$= a - b + b\, n(n+1)/2$$

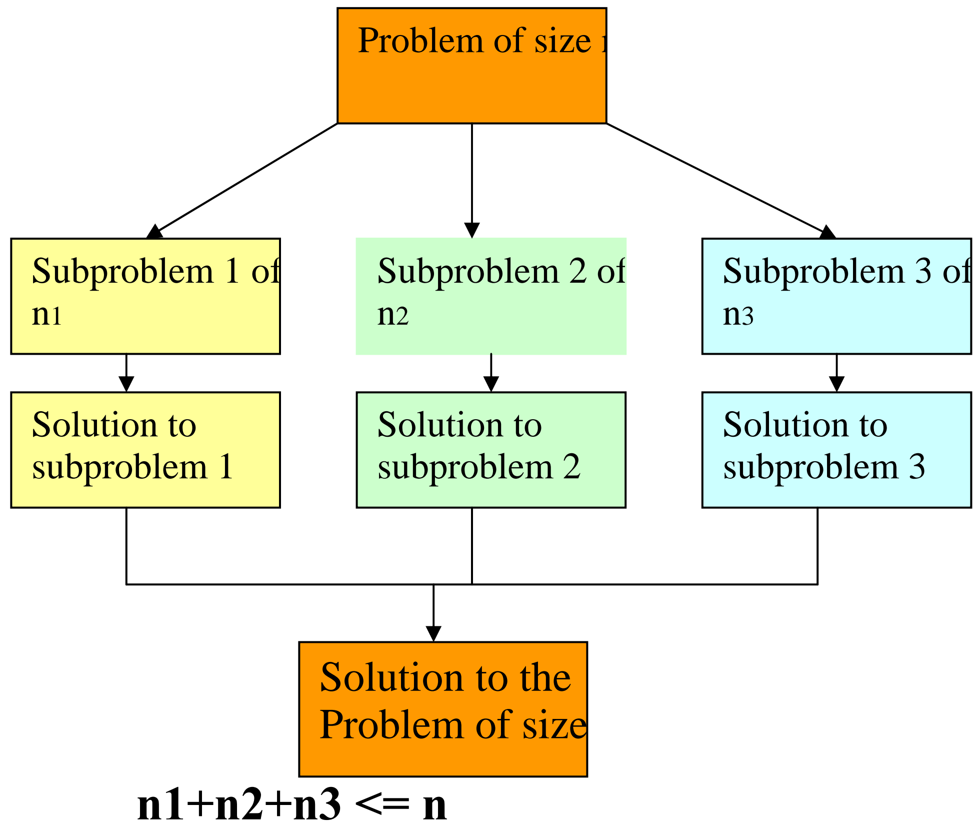
  and the complexity $C(n)$ is $\Theta(n^2)$.

Divide & Conquer Algorithms

Divide and conquer strategies are the basis of some the powerful algorithms.

1. Divide a problem of size n into several instances of the same problem with smaller sizes.
2. Solve the smaller size problem

3. Combine solutions to get solution of original problem



Note 1: A special case - the problem is divided into b sub-problems but only a subset of these smaller needs to be solved and merged. Example Binary search

Note 2: DCA are not always efficient. Example computing the sum of n numbers by DCA will result will result same number of basic operations since

$$a_0 + a_1 + \cdots + a_{n-1} = (a_0 + a_1 + \cdots + a_{n/2-1}) + (a_{n/2} + a_{n/2+1} + \cdots + a_{n-1})$$

The elements are recursively divided into two parts until the sub sum size is one, then they are added.

However, in many problems, DCA can result in considerable computation advantage.

Recurrence Relation of the Form

$C(n) = C(n/a) + b, C(1) =1$

assume that n is a power of a, i.e. $n = a^k$**, where k is an integer. Repeated substitution gives**

$$C(n) = C(n/a) + b$$

$$C(n/a) = C(n/a^2) + b$$

$$C(n) = C(n/a^2) + 2b$$

and after k steps;

$$C(n) = C(n/a^k) + kb$$

$$= 1 + b \log_a n$$

Example: Binary Search

```
// Binary search for key in an ordered array A
int BinarySerach (A [], int low, int hi, key)
{
    int mid;
    if (low > hi) return (-1) // stopping condition
    else {
        mid = (low+hi)/2;
        if (key == A[mid])   return mid;  // key found,
// array index returned
        else if (key < A[mid])
```

```
            return  BinarySearch(A, low, mid –1, key);
            else  return  BinarySearch(A, mid +1, hi, key);
        }
}
```

The basic operation is comparison and addition/division. If the cost of these for one recursive call is one unit (of time), then we have

$$C(n) = C(n/2) + 1$$

And the solution (16) now becomes

$$C(n) = \lg n + 1$$

where lg stands for log to the base of 2.

Example: Exponentiation

We want to write a function to compute x^n**. The obvious algorithm consists of a for loop to carry out n-1 multiplication. For example to find** x^5**:**
$x^5 = ((((x*x)*x)*x)*x)$

The complexity (number of multiplication) is C(n) = n-1. An alternative algorithm is to use recursion, as follows:

```
int pow(int x, int n)
{
   if (n == 0)   return 1;              // 1
   if (n == 1)   return x;              // 2
   if (n%2 == 0)
```

```
            return( pow (x*x, n/2));        // 3
    else
            return ( pow (x*x, n/2) * x);         // 4
}
```

The first couple of lines handle the base case. Otherwise if n is even, we have

$x^n = (x^{n/2})(x^{n/2})$

and if n is odd

$x^n = (x^{n/2})(x^{n/2})(x)$

For example to compute x^{31}**, the algorithm does the following 8 multiplications:**

$x^{31} = (x^{15}) * (x^{15}) * x$

$x^{15} = (x^7) * (x^7) * x$

$x^7 = (x^3) * (x^3) * x$

$x^3 = (x) * (x) * x$

To determine the running time complexity, let us consider the case where n is
a power of 2, that is $n = 2^k$**, where k is a positive integer. In this case, only lines 2 and 3 get executed (line 1 is executed only once). Let running time C(n) be**
the number of multiplication, then we can write

$$C(n) = C(n/2) + 1$$

The solution to this recurrence relation is given by (18) C(n) = lg n+1 . In general, when n is not a power of 2, both lines 3 and 4 get executed in the worst case giving C(n) = 2 lg n (Why?). We conclude that the recursive implementation of exponentiation is logarithmic (i.e. it is $\Theta(\lg n)$ **) whereas non-**

recursive implementation is linear (i.e. $\Theta(n)$). This result, however, must not lead us to think that recursive solution is always more efficient, as we will see in the following section.

Example: Majority Element

The majority element in an array A of size n is an element that appears more than n/2 times.

Example: 3,3,4,2,4,4,2,4,4 majority element is 4, 3,3,4,2,4,4,2,4 has no majority element.

When the number of elements in the array is always even:

- Find a candidate majority element using the following steps.

1. Form an array B half the size of A (i.e. n/2).
2. Compare A[0] with A[1], and if A[0] = A[1], copy one of them into B[0].
3. Continue Step 3 with A[2], A[3], etc. until all entries of the array A are processed, and if necessary copied into B.
4. Recursively find a candidate for B, that is create an array C of size
equal to half the size of B (i.e. n/4). If B[0] = B[1], then copy B[0] into C[0], check B[2] and B[3], and if they are equal copy B[2] in C[1], and continue.
5. Continue creating arrays D, E, etc as above, stop when a candidate element is found.
6. If the candidate element is found, use a sequential search through A to determine if this candidate is actually the majority element.

In the worst case, after making n/2 comparisons between adjacent array elements, the size of the resulting array is halved.

$$C(n) = C(n/2) + n/2; \quad C(1) = 1$$

4.1 General Divide and Conquer Recurrence Relation

A problem of size n, is divided into b >= 2 problems of size n/b with a >= 1 of these sub-problems needing to be solved.

Let g(n) = cost of dividing the problem in smaller problems and then combining the solutions

$$C(n) = aC(n/b) + g(n)$$

$$\begin{aligned} C(n) &= a[aC(n/b^2) + g(n/b)] + g(n) = a^2C(n/b^2) + a\,g(n/b) + g(n) \\ &= a^3C(n/b^3) + a^2g(n/b^2) + ag(n/b) + a^0g(n) \\ &= \ldots \\ &= a^kC(n/b^k) + a^{k-1}g(n/b^{k-1}) + a^{k-2}g(n/b^{k-2}) + \cdots + a^0g(n) \end{aligned}$$

Assume $n = b^k$, **or** $k = \log_b(n)$ **where k is a positive integer**

$$C(n) = a^{\log_b(n)}C(1) + \sum_{j=0}^{\log_b(n)-1} a^j g(n/b^j)$$

Note: $a^{\log_b(n)} = n^{\log_b(a)}$

$$C(n) = n^{\log_b(a)}C(1) + \sum_{j=0}^{\log_b(n)-1} a^j g(n/b^j)]$$

Master Theorem: If $g(n)$ is $\Theta(n^d)$ **where** $d \geq 0$,

$$C(n) \text{ is } \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_b n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

4.2 Example: Fake-Coin Problem

Among n coins one is lighter. A scale compares any two set of coins, and tells us which side is lighter or if they are equal.

1. **Divide the coins into two piles of $\lfloor n/2 \rfloor$ coins, leaving one coin out if n is odd.**
2. **Put the two pile on the scale. If they weigh the same the coin put aside is the fake one, otherwise proceed with the lighter pile.**

$$C(n) = C(n/2) + 1 \quad ; \qquad C(1) = 0$$

$$C(n) = aC(n/b) + g(n)$$

$$C(n) = n^{\log_b(a)}C(1) + \sum_{j=0}^{\log_b(n)-1} a^j g(n/b^j)$$

Here a =1, b =2, g(n) = 1,

$$C(n) = \sum_{j=0}^{\log_b(n)-1} 1 = \log_b n$$

Note: This recurrence relation is similar to the binary search equation.

Note: If $g(n)$ is $\Theta(n^d)$ where $d \geq 0$, Master Theorem:

$$C(n) \text{ is } \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_b n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

d = 0, a = 1, b =2 so the second case and $g(n)$ is $\Theta(\log n)$.

4. 3 Example– Recursive DCA Summation:

$$a_0 + a_1 + \cdots + a_{n-1} = (a_0 + a_1 + \cdots + a_{n/2-1}) + (a_{n/2} + a_{n/2+1} + \cdots + a_{n-1})$$

The number of additions made

$$C(n) = 2\,C(n/2) + 1$$
$$C(1) = 1$$

$$C(n) = aC(n/b) + g(n)$$

$$C(n) = n^{\log_b(a)} C(1) + \sum_{j=0}^{\log_b(n)-1} a^j g(n/b^j)$$

a = 2, b =2, g(n)=1. The solution is

$$C(n) = n + \sum_{j=0}^{\log_2(n)-1} 2^j$$

Since

$$\sum_{j=0}^{p} \alpha^j = \frac{\alpha^{p+1} - 1}{\alpha - 1} = 2^{\lg n} - 1 = n - 1$$

$$C(n) = 2n - 1$$

Note: $C(n) = \Theta(n)$. Thus in this case DCA requires the same number of additions as the brute force method which uses n additions. However, the recursive algorithm has the added cost of recursive calls.

Exercise: Solve C(n) = 2C(n/2) + lg n; C(1)=1.

Note from Chapter 1: $\sum_{j=0}^{p} j2^j = (p-1)2^{p+1} + 2.$

2.4 Recursive versus Non-Recursive Functions

We consider two examples as follows:

Example: Exponentiation

Compute x^n. The obvious algorithm consists of a for loop to carry out n-1 multiplication. For example to find x^5:

x^5**= ((((x*x)*x)*x)*x)**

The complexity (number of multiplication) is C(n) = n-1.

Alternatively

```
int pow(int x, int n)
{
   if (n == 0)   return 1;               //   base case
   if (n == 1)   return x;               // base case
   if (n%2 == 0)
            return( pow (x*x, n/2));        // 3
   else
            return ( pow (x*x, n/2) * x);        // 4
}
```

If n is even,

$x^n = (x^{n/2})(x^{n/2})$

and if n is odd

$x^n = (x^{n/2})(x^{n/2})(x)$

For example to compute x^{31}**, the algorithm does the following 8 multiplications:**

$x^{31} = (x^{15}) * (x^{15}) * x$

$x^{15} = (x^7) * (x^7) * x$

$x^7 = (x^3) * (x^3) * x$

$x^3 = (x) * (x) * x$

Let $n = 2^k$**. In this case, only lines 2 and 3 get executed (line 1 is executed only once). Let C(n) = number of multiplications**

C(n) = C(n/2) + 1

Solution is is C(n) = lg n (show this!).

In general, when n is not a power of 2, both lines 3 and 4 get executed in the worst case giving C(n) = 2 lg n.

Note: Recursive implementation is O(lg n)).

Non-recursive implementation is linear (i.e. O(n)).

Example: Fibonacci numbers

The Finonacci numbers are $F_0, F_1, F_2, \cdots, F_n$ **where** $F_0 = F_1 = 1$**, and** $F_{i+1} = F_i + F_{i-1}$**. A recursive algorithm to compute these numbers is**

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

If C(n) is the work needed to compute the Fobonacci numbers up to n, then we have

C(n) = C(n-1) + C(n-2)

It was mentioned in Chapter 1 that the solution to an equation of the above form satisfies

$C(n) \le a^n$ for all $n \ge 1$

where a = 5/3. As a result, C(n) is $O(a^n)$, and the complexity is exponential in n.

Exercise:
(a)Show that the exact solution to the Fibonacci recurrence relation is

$$C(n) = \frac{1}{\sqrt{5}}\left(b^n - c^n\right)$$

where $b = \frac{1+\sqrt{5}}{2}$ and $c = \frac{1-\sqrt{5}}{2}$**.**
(b) Use the result in (a) to show that $C(n)$ is $O(b^n)$

Non-recursive algorithm

```
int fibonacci ( int n)
{
        int i, last, next_to_last, answer;

        if ( n<= 1)  return 1;
        last = next_to_last = 1;
        for( i = 2; i <= n; i++)
        {
                answer = last + next_to_last;
                next_to_last = last;
                last = answer;
        }
        return answer;
}
```

- **only one "for" loop which iterates n times, the above algorithm is linear in n, i.e. O(n).**
-
- **Note: To compute fib(n), there is one call to fib(n-1) and fib(n-2). However, since fib(n-1) recursively makes a call to fib(n-2) and fib(n-3), there are actually two separate calls to compute fib(n-2). If we trace the algorithm, we can see that fib(n-3) is computed three times, fib(n-4) is computed five times, fib(n-5) is computed eight times, and so on. The growth of redundant calculations is explosive.**