CS200: Recurrence Relations and the Master Theorem

Rosen Ch. 8.1 - 8.3

Recurrence Relations: An Overview

- What is a recurrence?
  - **A recursively defined sequence …**
  - **… defined by a recurrence relation**
- Example
  - Arithmetic progression: $a, a{+}d, a{+}2d, \ldots, a{+}nd$
    - $a_0 = a$
    - $a_n = a_{n-1} + d$

Formal Definition

A recurrence relation for the sequence $\{a_n\}$ is an equation that expresses a_n in terms of one of more of the previous terms of the sequence, namely, $a_0, a_1, \ldots a_{n-1}$, for all integers n with $n \geq n_0$ where n_0 is a nonnegative integer.

- A Sequence is called a solution of a Recurrence relation + Initial conditions ("*base case*"), if its terms satisfy the recurrence relation

- Example: $a_n = a_{n-1} + 2, \;\; a_1 = 1$

a_1?, a_2? a_3?

solution?

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

Compound Interest

- You deposit $10,000 in a savings account that yields 10% yearly interest. How much money will you have after 1,2, … years? (b is balance, r is rate)

$$b_n = b_{n-1} + rb_{n-1} = (1+r)^n b_0$$

$$b_0 = 10,000$$

$$r = 0.1$$

Modeling with Recurrence

- Suppose that the number of bacteria in a colony triples every hour
  - Set up a recurrence relation for the number of bacteria after n hours have elapsed.
  - 100 bacteria are used to begin a new colony.

Recursively defined functions and recurrence relations

- **A recursive function**

  $f(0) = a$ (base case)

  $f(n) = f(n\text{-}1) + d$ for $n > 0$ (recursive step)

- The above recursively defined function generates the sequence

  $a_0 = a$

  $a_n = a_{n\text{-}1} + d$

- A recurrence relation **produces a sequence**, an application of a recursive function produces **a value from the sequence**

How to Approach Recursive Relations

Recursive Functions ⟷ **Sequence of Values**

$f(0) = 0$ (base case)
$f(n) = f(n\text{-}1) + 2$ for $n > 0$
(recursive part)

$f(0) = 0$
$f(1) = f(0)+2 = 2$
$f(2) = f(1)+2 = 4$
$f(3) = f(2) + 2 = 6$

↓

Closed Form?(solution, explicit formula)

Find a recursive function

- Give a recursive definition of f(n)=a^n, where *a* is a nonzero real number and *n* is a nonnegative integer.

  f(0) = 1,
  f(n) = a * f(n-1)

- Give a recursive definition of factorial f(n) = n!

  f(0) = 1
  f(n) = n* f(n-1)

- Rosen Chapter 5 example 3-2 pp. 346

Solving recurrence relations

Solve $a_0 = 2;\ a_n = 3a_{n-1},\ n > 0$

(1) What is the recursive function?

(2) What is the sequence of values?

a

Hint: Solve by repeated substitution, recognize a pattern, check your outcome

- $a_0 = 2;\ a_1=3(2)=6;\ a_2=3(a_1)=3(3(2));\ a_3=\ldots$

Connection to Complexity… Divide-and-Conquer

Basic idea:

Take large problem and **divide** it into **smaller problems** until problem is trivial, then **combine** parts to make solution.

Recurrence relation for the number of steps required:

$$f(n) = a\, f(n / b) + g(n)$$

n/b : the size of the sub-problems solved

a : number of sub-problems

$g(n)$: steps necessary to split sub-problems and combine solutions to sub-problems

Example: Binary Search

```
public int binSearch (int myArray[], int first,
                                    int last, int value) {
   // returns the index of value or -1 if not in the array
   int index;
   if (first > last) { index = -1; }
   else {
      int mid = (first + last)/2;
      if (value == myArray[mid]) { index = mid; }
      else if (value < myArray[mid]) {
              index = binSearch(myArray, first, mid-1, value);
      }
          else {
          index = binSearch(myArray, mid+1, last, value);
      }
   }
   return index;
}
```

What are a, b, and g(n)?

$$f(n) = a \cdot f(n/b) + g(n)$$

Estimating big-O (Master Theorem)

Let f be an increasing function that satisfies

$$f(n) = a \cdot f(n/b) + c \cdot n^d$$

whenever $n = b^k$, where k is a positive integer, $a \geq 1$, b is an integer > 1, and c and d are real numbers with c positive and d nonnegative. Then

$$f(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Section 8.3 in Rosen
Proved using induction

Binary Search using the Master Theorem

For binary search

$f(n) = a\, f(n / b) + c\, .n^d$

$\quad = 1\, f(n / 2) + c$

$$f(n) = \begin{cases} O\left(n^d\right) & \text{if } a < b^d \\ O\left(n^d \log n\right) & \text{if } a = b^d \\ O\left(n^{\log_b a}\right) & \text{if } a > b^d \end{cases}$$

Therefore, $d = 0$ *(to make* n^d *a constant)*, $b = 2$, $a = 1$.

$b^d = 2^0 = 1$

It satisfies the second condition of the Master theorem.

So, $f(n) = O(n^d \log_2 n) = O(n^0 \log_2 n) = \mathbf{O(\log_2 n)}$

Complexity of MergeSort with Master Theorem

```
public void mergesort(Comparable[] theArray, int first, int last){
    // Sorts the items in an array into ascending order.
    // Precondition: theArray[first..last] is an array.
    // Postcondition: theArray[first..last] is a sorted permutation
    if (first < last) {
            int mid = (first + last) / 2;  // midpoint of the array
            mergesort(theArray, first, mid);
            mergesort(theArray, mid + 1, last);
            merge(theArray, first, mid, last);
    }// if first >= last, there is nothing to do
}
```

- $M(n)$ is the number of operations performed by mergeSort on an array of size n
- M(0)=M(1) = 1 $M(n) = 2M(n/2) + c.n$ WHY + n ?

the cost of merging two arrays of size n/2 into one of size n

Complexity of MergeSort

Master theorem

$M(n) = 2M(n/2) + c.n$

for the mergesort algorithm

$$f(n) = a\,f(n/b) + c.n^d$$
$$= 2\,f(n/2) + c.n^1$$

$$f(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Notice that c does not play a role(big O)

$d = 1,\ b = 2,\ a = 2.$ *Therefore* $b^d = 2^1 = 2$

It satisfies the second condition of the Master theorem.

So, $f(n) = O(n^d log_2 n)$

$= O(n^1 log_2 n)$

$= \mathbf{O(nlog_2 n)}$

Best Case QuickSort Recurrence

$$f(n) = a \cdot f(n/b) + cn^d$$

Best case: assume perfect division in equal sized partitions

- a=
- b=
- c=
- d=
- O(?)

$$f(n) = \begin{cases} O\left(n^d\right) & \text{if } a < b^d \\ O\left(n^d \log n\right) & \text{if } a = b^d \\ O\left(n^{\log_b a}\right) & \text{if } a > b^d \end{cases}$$

Worst Case: n + (n-1) + … +3 + 2+ 1= $O(n^2)$

CS320 Excursion: Tractability

- A problem that is solvable using an algorithm with **polynomial worst-case complexity** is called **tractable**.
- If estimation has high degree or if the coefficients are extremely large, the algorithm may take an extremely long time to solve the problem.

Intractable vs Unsolvable problems

- If the problem *cannot be solved* using an algorithm with worst-case polynomial time complexity, such problems are called **intractable**. Have you seen such problems?

- If it can be shown that no algorithm exists for solving them, such problems are called **unsolvable**.

Hanoi

```
// pegs are numbers, via is computed
// number of moves are counted
// empty base case
public void hanoi(int n, int from, int to){
        if (n>0) {
                int via = 6 - from - to;
                hanoi(n-1,from, via);
                System.out.println("move disk " + n +
                                        " from " + from + " to " + to);
                count++;
                hanoi(n-1,via,to);
        }
}
```

Recurrence for number of moves?

Solution?

How did we prove this earlier?

Permutations

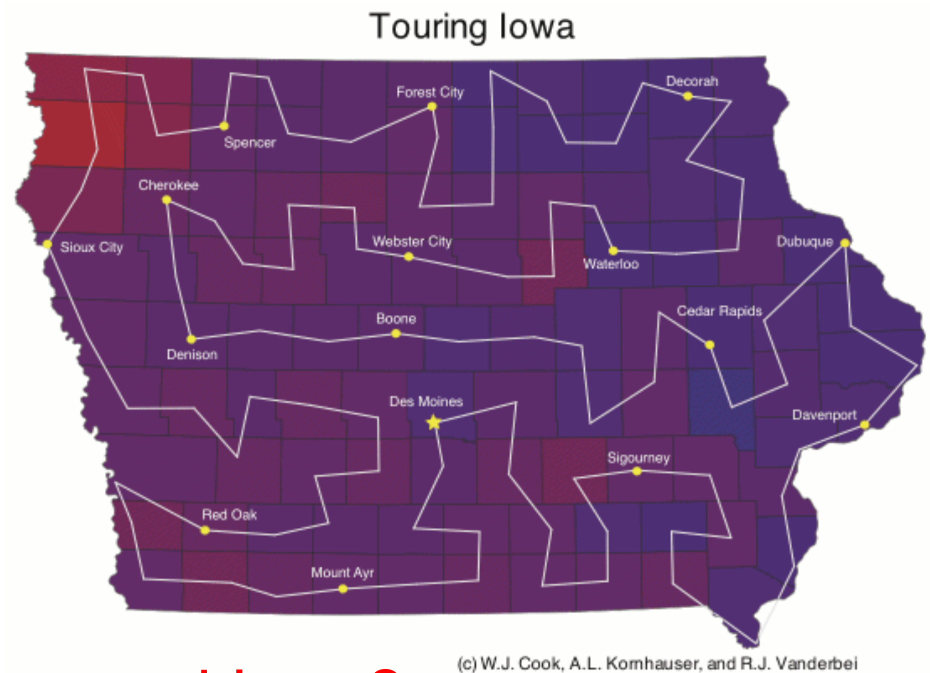
```
public void permute(int from) {
 if (from == P.length-1) {// suffix size one, nothing to permute
          System.out.println(Arrays.toString(P));
 else {   // put every item in first place and recur
           for (int i=from; i<P.length;i++) {
             swapP(from,i); // put i in first position of suffix
             permute(from+1); // permute the rest
             swapP(from,i); // PUT IT BACK
           }
         }
}
```

complexity? number of permutations? recurrence relation?

Interesting Intractable Problems

- Boolean Satisfiability **2^n**

(A v ~B v C) ^ (~A v C v ~D) ^ (B v ~C v D)



- TSP **$n!$**

- only solution:
  trial and error

how many options for these problems?