Divide and Conquer (Dynamic Programming)

Raymond Klefstad, Ph.D.

Introduction

- Divide-and-Conquer Link
  - partition problem into disjoint sub-problems
  - solve the sub-problems recursively
  - combine their solutions to solve original problem
- Dynamic Programming Link
  - applies when sub-problems overlap
    - when sub-problems share sub-sub-problems
  - “Programming” refers to “tabular method” like “Linear Programming”
  - compute solutions to sub-problems once and save them in a table
- Recursion must be mastered first

Recursion

- Recursion occurs when a function calls itself Link
- Usually starts with case analysis
  - Handle base cases, e.g., N==0, an empty list or string
  - Handle general cases, e.g., N>0, non-empty list or string
  - Beware infinite recursion (like infinite looping)
- Example: sum up the first N positive numbers

```
int sum( int N ) {
    if ( N <= 0 ) return 0;
    else return N + sum(N - 1);
}
```

Factorial

- N! = N * N-1 * N-2 … * 1

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

- N!   re-write using the ternary conditional operator

```
int factorial( int N ) {
    return N == 0 ? 1 : N * factorial(N - 1);
}
```

Review of C-strings

- C-strings excellent for recursion
- Null terminated array of characters
  - E.g., char s[] = “Hello”; // s has 6 chars and last is ‘\0’
- C-arrays can be processed using pointer arithmetic
  - ***s*** is current character, e.g., ‘H’
  - **s+1** is rest of string, e.g., “ello”
- Can traverse a string in two ways

```
for ( int i=0;  s[i] != '\0';  ++i ) putchar(s[i]);

for ( char *p = s;  *p != '\0';  ++p ) putchar(*p)
```

- C-string parameters are often declared as char *

```
char * strcpy( char * dest, char * src );
```

How To Think About Recursion

- I want to write a function that computes the length of a c-string named s
- Assume I already have such a function, but I can't call it directly on my parameter s
- Suppose I know the length of the rest of this string s+1
  - I just add one to that length
- What is my base case?
  - The empty string? Yes   What is its length?   Zero
- Let's write a few and compare iterative to recursive

String Length: strlen()

```
char s[] = “ABC”;
int i = strlen(s);   // should be 3

int strlen( char * s ) {
        int len = 0;
        for ( int i=0; s[i] != ‘\0’; ++i )
                ++len;
        return len;
}

int strlen( char * s ) {
       return *s == ‘\0’ ? 0 : 1 + strlen( s + 1 );
}
```

Find char in String: strchr()

```
char s[100] = “ABC”;
char *p = strchr(s, ‘B’);   // p will be “BC”

char * strchr( char * src, char c ) {
      for ( ;  *src != c;  ++src  )
            if ( !*src ) return 0;
      return src;
}

char * strchr( char * src, char c ) {
      return *src == c  ?  src  :  !*src  ?  0  :  strchr( src + 1, c );
}
```

String copy: strcpy()

```
char s[] = “ABC”,  t[4];
char *p = strcpy(t, s);   // p will be “ABC” pointing to array t

char * strcpy( char * dest, char * src ) {
        char * ret = dest;
        while ( *dest++ = *src++ )
                ;
        return ret;
}

char * strcpy( char * dest, char * src ) {
        *dest = *src;
        if ( *src ) strcpy( dest + 1, src + 1 );
        return dest;
}
```

Append to End: strcat()

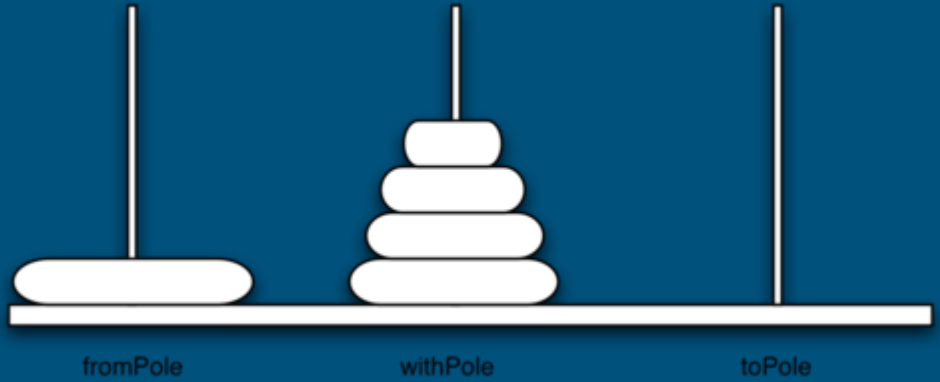
```
char s[100] = “ABC”;
strcat(s, “DEF”);   // s will be ABCDEF

char * strcat( char * dest, char * src ) {
        char * ret = dest;
        while ( *dest++ = *src++ )
                ;
        return ret;
}

char * strcat( char * dest, char * src ) {
        if ( !*dest ) strcpy( dest, src );
        else strcat( dest + 1, src );
        return dest;
}
```

Divide & Conquer Examples

- Problems that can be divided, solved, and merged are often ideal for recursion
  - Draw a line on a pen plotter
  - Sorting data within an array: Quick Sort, Merge Sort
  - Traversing a binary search tree
  - Searching a hierarchical directory structure
  - Solving towers of Hanoi puzzle

Towers of Hanoi

```
void solve(int N, char * from, char * to, char * with) {
    if ( N >= 1 ) {
        solve(N-1, from, with, to);
        print("move top disk from ", from, " to ", to);
        solve(N-1, with, to, from);
    }
}
```



Performance

- Recursion adds function call overhead
  - Compared to loops
- Compilers can optimize some types of recursion
- Tail recursion
  - When there is one recursive call, e.g., linked lists, c-strings, numbers
  - Can be converted to looping
- Complete recursion or Total recursion
  - Two or more calls
    - E.g., Sorts, Trees, Graphics
  - More difficult to optimize, requires explicit stack, rarely done
- But what about stack depth? O(N), O(lg N)

Recursion Summary

- Recursion is when a function is defined by calling itself
- Often leads to concise, elegant solutions
- Must beware of infinite recursion
- Potential function call overhead may be removed by some optimizing compilers

Dynamic Programming

- an improvement to divide and conquer
- often applied to optimization problems
- may have many solutions, but seeking optimal
- 4 steps:
  1. Characterize structure of an optimal solution
  2. Recursively define value of an optimal solution
  3. Compute value of an optimal solution bottom-up
  4. Construct solution from computed information
- use memoization to avoid duplicated computation

Memoization

- Memoization save computed values for reuse Link
- Often combined with recursion
  - If value is memoized, return saved value
  - Else, compute new value, save it, return it
- Example: sum up the first N positive numbers

```
int sum( int N ) {
    if ( N <= 0 )  return 0;
    else return N + sum( N - 1 );
}
```

Memoization Example

```
#define M 1000

int sum( int N ) {
    static int A[ M ] = { 0 };

    if ( N < 0 )
        return 0;
    if ( A[ N ] == 0 )
        A[ N ] = N + sum( N - 1 );
    return A[ N ];
}
```

```
int main()
{
    int N = 30;
    for ( int i = 0; i < N; ++i )
        cout << i << '=' << sum(i) << endl;
    cout << endl;
    return 0;
}
```

Fibonacci Numbers

- Fibonacci number Link
- 0,1,1,2,3,5,8,13,21,34,55,89,144,...

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

T(n) = O(1.6180^n) = O(2^N)

Fib Bottom-up

```
int fib( int N ) {
  int f[N+1];
  f[0] = 0;   f[1] = 1;
  for ( int i = 2; i <= N; ++i )  {
      f[ i ] = f[ i-1 ] + f[ i-2 ];
  }
  return f[N];
}
```

Fib Top-down

```
int fib( int N ) {
  static int f[ N+1 ] = {0};
  if ( N <= 0 ) return 0;
  else if ( N == 1 ) return 1;
  if ( f[ N ] == 0 )  f[ N ] = fib( N-1 ) + fib( N-2 );
  return f[ N ];
}
```

Dynamic Programming

- 4 steps:
  1. Characterize structure of an optimal solution
  2. Recursively define value of an optimal solution
  3. Compute value of an optimal solution bottom-up
  4. Construct solution from computed information
- use memoization to avoid duplicated computation

DP Example: LCS

- given two strings: S of length N, and T of length M
- find Longest Common Subsequence
  - the longest sequence of characters that appear left-to-right
  - but not necessarily in a contiguous block

  S = ABAZDC

  T = BACBAD
- LCS has length 4 and is the string ABAD
- useful for genomics and text editor display update
- naive algorithm is exponential

Solving LCS with DP

- sub-problem: look at LCS for prefix of S and prefix of T
  - overall pairs of prefixes
- to simplify, find only length of LCS
  - later we'll modify to extract the answer
- let LCS[i][j] be length of LCS of S[0..i] with T[0..j]
- Can we solve LCS[i][j] using LCS’s of smaller problems? Yes!

Recursive Cases

- Case 1: S[i] != T[j]   no match
  - Desired subsequence must ignore either S[i] or T[j]
  - LCS[i][j] = max(LCS[i – 1][j], LCS[i][j – 1])
- Case 2: S[i] == T[j]   match
  - LCS of S[0..i] and T[0..j] is one longer...
  - LCS[i][j]   = 1   + LCS[i – 1][j – 1]
-

Computation of the Table

```
for ( int i = 0; i < N; ++i )  // S
    for (int j = 0; j < M; ++j)  // T
        T[ i ][ j ] = case analysis from previous slide, is O(1)
```

The Table

S \ T	B	A	C	B	A	D
A	0	1	1	1	1	1
B	1	1	1	2	2	2
A	1	2	2	2	3	3
Z	1	2	2	2	3	3
D	1	2	2	2	3	4
C	1	2	3	3	3	4

final answer is in the lower right-hand corner: 4

What is the Sequence?

- must be extracted from our table
- walk backwards through table, starting at lower-right corner
- If cell directly above or directly to right is equal to value
  - move to that cell (if both are, choose either one)
- if both are less than value
  - move diagonally up and left (case 2)
  - and output the associated matching character

Extracting the Answer

S \ T	B	A	C	B	A	D
A	0	1	1	1	1	1
B	1	1	1	2	2	2
A	1	2	2	2	3	3
Z	1	2	2	2	3	3
D	1	2	2	2	3	4
C	1	2	3	3	3	4

- this will print answer in reverse, DABA

Time Complexity

- exhaustive search is exponential
- our dynamic programming algorithm is O(N * M)