McGill

COMP 208 – Computers in Engineering

COMP 208
Computers in Engineering

Lecture 18

Jun Wang
School of Computer Science
McGill University
Fall 2007

Recursive Sum of Squares (a)

```
int SumSquares(int m, int n){
  if (m<n)
   return m*m + SumSquares(m+1, n);
  else
   return m*m;
}
```

Denote by f(m, n) the sum of squares of numbers from m to n, then,

$$f(m, n) = \begin{cases} m*m, & \text{if } m >= n \\ m*m + f(m+1, n), & \text{otherwise} \end{cases}$$

example

$$f(m, n) = \begin{cases} m*m, & \text{if } m >= n \\ m*m + f(m+1, n), & \text{otherwise} \end{cases}$$

m=2, n=5

f(2,5) = 2*2 + f(3, 5)

→ = 3*3 + f(4, 5)

→ = 4*4 + f(5, 5)

→ = 5*5

base case

Factorial

The factorial function is a classic example of the use of recursion

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

$$\text{fact}(n) = \begin{cases} 1, & \text{if } n < 1 \\ 1*2*3...*n, & \text{otherwise} \end{cases}$$

therefore, $$\text{fact}(n) = \begin{cases} 1, & \text{if } n < 1 \\ n * \text{fact}(n-1), & \text{otherwise} \end{cases}$$

Divide and Conquer

- Many problems can be solved efficiently by
  1. Splitting them in half
  2. Recursively solving the two subproblems
  3. Combining the solutions to solve the original problem
- This often involves adding extra parameters to keep track of the subproblems
- Too keep to the original problem specification we often create a “shell” program

Factorial Again
Divide and Conquer

```
int product(int m, int n)
{
   int mid;
   if (m<n){
        mid = (m+n)/2;
        return product(m,mid) * product(mid+1,n);
   }
   else
        return m;
}

int factorial(int n)
{
   return product(1,n);
}
```

Choosing k objects from n

- The number of ways in which k objects can be chosen from n is given by:

$$C_{n,k} = \frac{n!}{(n-k)!k!}$$

- We can write a function that computes this using the factorial function

Comb (n, k)

```
int comb(int n, int k)
{
   if (k > n) return 0;
   return fact(n)/(fact(k)*fact(n-k));
}

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

An Alternative Approach

- Computing factorials of even fairly small numbers can cause overflow
- An alternative algorithm uses a recurrence relation:

$$C_{n,k} = 1, \quad k = 1 \; or \; k = n$$

$$C_{n,k} = C_{n-1,k} + C_{n-1,k-1}, \quad 1 < k < n$$

A Recursive Comb(n,k)

```
int comb(int n, int k)
{
  if (k > n) return 0;
  if ((k == n) || (k == 0)) return 1;
  return comb(n-1,k) + comb(n-1,k-1);
}
```

No factorial involved

McNugget Numbers

Chicken McNuggets originally came in boxes of 6, 9 or 20.

A number is McNugget if it can be obtained by adding together orders of boxes of varying sizes

We want to determine whether a given number is McNugget or not

McNugget Numbers

```
int is_mc_nugget(int n){
   if (n == 0) return 1;
   if (n < 6) return 0;
   if ((n >= 20) && is_mc_nugget(n-20))
     return 1;
   else if ((n >= 9) && is_mc_nugget(n-9))
          return 1;
   else if ((n >= 6) && is_mc_nugget(n-6))
          return 1;
   else return 0;
}
```

2 base cases

McNugget Numbers

A clever solution:

```
int is_mc_nugget(int n)
{
  return
      ((n >= 20) && is_mc_nugget(n-20)) ||
      ((n >=  9) && is_mc_nugget(n- 9)) ||
      ((n >=  6) && is_mc_nugget(n- 6)) ||
      (n == 20) || (n == 9) || (n == 6);
}
```

Binary Search

In binary search, we check the middle element and then search the first or second half of the array, a smaller instance of the same problem

That looks pretty recursive!

The base case occurs when the array is empty

Recursive Binary Search

```
int recursive_binary_search
        (int val, int arr[], int left, int right){
  int mid = (left + right) / 2;

  if(left>right)
   return -1;
  if(arr[mid] > val)
   return recursive_binary_search(val,arr,left,mid-1);
  else if(arr[mid] < val)
   return recursive_binary_search(val,arr,mid+1,right);
  else
   return mid;
}
```

```
int main()
{
  int a[] = {3,5,7,8,9};
  int idx = recursive_binary_search(5, a, 0, 4);
}
```

Recursive Sorting

Many sorting algorithms can be developed recursively

If we assume we can sort small arrays, we can use that information to sort larger arrays

Let's reexamine bubble sort from a recursive point of view

Recursive Bubble Sort

- We begin by comparing pairs of values, rearranging those that are out of order
- The result is that the smallest value is in the first position of the array
- We can then recursively sort the rest of the array
- The base case occurs when the array only has one value left

Recursive Bubble Sort

```
void recursive_bubble_sort(int arr[], int size){
  int i;

  if(size <= 1) return;
  for(i = size - 1; i; --i)
    if(arr[i] < arr[i - 1])
      swap(&arr[i], &arr[i - 1]);

  recursive_bubble_sort(arr + 1, size - 1);

}
```

Recursive Selection Sort

- We select the smallest value and put it at the front of the array (by swapping)
- That leaves us with a smaller array to sort recursively
- The base case occurs when the array has one value

Recursive Selection Sort

```
void recursive_select_sort(int arr[], int size){

  int index_of_small;

  if(size <= 1) return;
  index_of_small = find_smallest(arr, size);
  swap(arr, arr + index_of_small);

  recursive_select_sort(arr + 1, size - 1);

}
```

Recursive find_min

```
int recursive_find_min(int a[], int size)
{
   if(size == 1)
      return 0;
   else {
      int idx = recursive_find_min(a, size-1);
      if(a[idx] <= a[size-1])
         return idx;
      else
         return size-1;
   }
}
```

Recursive Insertion Sort

- We first sort the small array that includes all but the last value
- Then insert this value in the proper position
- The base case occurs when the array has one value

Recursive Insertion Sort

```
void recursive_insertion_sort(int arr[],
                              int size){
  int i;

  if(size <= 1) return;
  recursive_insertion_sort(arr, size - 1);
  for(i = size - 1; i; --i)
    if(arr[i] < arr[i - 1])
      swap(&arr[i], &arr[i - 1]);
    else
      break;

}
```

Towers of Hanoi

```
#include <stdio.h>
void toh(int,int,int,int);
int main() {
   int n;
   scanf ("%d",&n);
   toh(n, 1, 2, 3);
   return 0;
}
void toh (int n, int a, int b, int c) {
   if (n>0) {
        toh (n-1, a, c, b);
        printf ("Move a disk from %d to %d\n", a, b);
        toh (n-1, c, b, a);
   }
}
```

Sorting by Merging

Array a: 9819 21020 14287 15229 781 8641 24044 14153
15751 32765 21626 28948 17411 11311 32560 6223 14466

Sort first half (recursively):
9819 21020 14287 15229 781 8641 24044 14153
Sorted:
781 8641 9819 14153 14287 15229 21020 24044

Sort second half (recursively):
15751 32765 21626 28948 17411 11311 32560 6223 14466
Sorted:
6223 11311 14466 15751 17411 21626 28948 32560 32765

Now **merge** the two sorted halves:
Sorted Array a: 781 6223 8641 9819 11311 14153 14287
14466 15229 15751 17411 21020 21626 24044 28948 32560
32765

Algorithm

- This is a recursive algorithm.
- The base case occurs when there are zero or one values to sort. Then the array is already sorted.
- The general case involves splitting the array, sorting the two halves and merging.

Temporary Storage

- To merge the two halves that have already been sorted we use a temporary array
- We create this array when needed
- We free the storage used when we are done

The Outer "Shell"

```
void merge_sort(int arr[], int size){

  // Allocate the temporary array.
  int *temporary = (int *) malloc(size * sizeof (int));

  // Start the recursive sort.
  _merge_sort(arr, size, temporary);

  // Free the allocated array.
  free(temporary);

  return;
}
```

Dynamic Memory Allocation

When a variable declaration is executed the C compiler allocates memory for an object of the specified type

A C program also has access to a large chunk of memory called the **free store**

Dynamic memory allocation enables us to allocate blocks of memory of any size **within** the program, not just when declaring variables

This memory can be released when no longer needed

These are useful for creating dynamic arrays and dynamic data structures such as linked lists (which we do not cover in this course)

malloc

- The **malloc** function removes a specified number of contiguous memory bytes from the free store and returns a pointer to this block

- It is defined by:

```
void *malloc(number_of_bytes)
```

- This function is defined in **stdlib.h**
- The argument type must be an **`unsigned integer`**.

malloc

- `malloc(n)` returns a pointer of type **`void *`** that is the start in memory of a block of n bytes
- If memory cannot be allocated a NULL pointer is returned.
- This pointer can be converted to any type.

sizeof

- How do we know how many bytes of storage we need to hold a data object?
- `Sizeof()` can be used to find the size of any data type, variable or structure
- Even if you know the actual size you want, programs are more portable if you use `sizeof()`

sizeof

- To reserve a block of memory capable of holding 100 integers, we can write:

```
int *ip;
ip = (int *) malloc(100*sizeof(int));
```

- The duality between pointers and arrays allows us to treat the reserved memory like an array.

Deallocating Memory

- Suppose a large program calls mergesort many times in the course of a computation involving large arrays
- Each time a new block of memory is allocated but after the call, it is no longer accessible. That memory is called **garbage**
- Programs that generate garbage are said to have a **memory leak**

Deallocating Memory

- Memory leaks can lead to severe deterioration in performance and eventually to program failure
- Some languages (such as Java) automatically check for blocks of memory that cannot be accessed and return them to the free store
- This is called automatic garbage collection
- In C the programmer is responsible to make sure that garbage is not left behind

`free()`

- The function **`free()`** takes a pointer to an object as its value and frees the memory that pointer refers to

- DANGER: Make sure the pointer is not NULL or you can cause a spectacular crash