Computers in Engineering COMP 208

Recursion

Michael A. Hawker

What is Recursion?

- Recursion is a programming technique that allows us to approach programming in a different style
- To solve a problem, we make an assumption that the computer is able to solve smaller instances of the same problem

What is Recursion?

- If we assume the computer can solve the smaller problems, our task is to tell it how to use these solutions to solve the larger problem
- We also have to describe how to solve the “smallest” problem (the one that doesn’t decompose) directly

Sum of Squares – Iterative

Add up the squares of all the numbers between two positive values m and n, m<=n

```
int SumSquares(int m, int n){
  int i, sum;
  sum = 0;
  for (i=m; i<=n; i++)
   sum = sum + i*i;
  return sum;
}
```

Recursive Solution

- We have to think of the problem in a new way
- Assume that we can solve smaller instances of the problem
- How do we use that information to solve the larger problem?

Recursive Solution

- What are the smaller instances?
- How do we use their solution to solve our problem?
- What is the smallest instance? We have to solve it directly.
- This instance is called the base case.

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;
}
```

Recursive Sum of Squares (b)

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

Recursive Sum of Squares (c)

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

Forgetting Base Case

Forgetting the base case can lead to an infinite sequence of calls:

```
int SumSquares(int m, int n){
  printf("Entering SumSquares/n");
  return m*m + SumSquares(m+1, n);
}
```

Visualizing Recursion

We can use call trees to visualize the computations carried out during recursive calls

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;
}
```

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

int factorial(int n){
   return power(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(n-k)*fact(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);
}
```

McNugget Numbers

- Chicken McNuggets originally came in boxes of 6, 9 or 20.
- A number is McNuggetable if it can be obtained by adding together orders of boxes of varying sizes
- We want to determine whether a given number is McNuggetable or not

McNugget Numbers

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

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) ||
  (n == 0);
}
```

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;
}
```

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 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

The Merge Sort Shell Again

Now that we have seen the use of dynamic memory allocation, let’s have another look at the mergesort 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);
}
```

Merge Sort Itself

```
void _merge_sort(int arr[], int size, int temporary[]){

  int half = size / 2;
  int i;

  if(size <= 1) return;

  _merge_sort(arr, half, temporary);
  _merge_sort(arr + half, size-half, temporary + half);
  merge(arr, half, arr + half, size - half, temporary);

  for (i=0;i<size;i++)
    arr[i] = temporary[i];
 }
```

Merging Two Sorted Lists

Sorted Array a:

781 8641 9819 14287 15229 21020 24044

Sorted Array b:

6223 11311 14153 15751 17411 21626 28948 32560 32765

Merged Array:

781

Merging Two Sorted Lists

Sorted Array a:

781 **8641** 9819 14287 15229 21020 24044

Sorted Array b:

6223 11311 14153 15751 17411 21626 28948 32560 32765

Merged Array:

781 **6223**

Merging Two Sorted Lists

Sorted Array a:

781 **8641** 9819 14287 15229 21020 24044

Sorted Array b:

6223 **11311** 14153 15751 17411 21626 28948 32560 32765

Merged Array:

781 6223 **8641**

Merging Two Sorted Lists

Sorted Array a:

```
781 8641 9819 14287 15229 21020 24044
```

Sorted Array b:

```
6223 11311 14153 15751 17411 21626
  28948 32560 32765
```

Merged Array:

```
781 6223 8641 9819
```

Merging Two Sorted Lists

Sorted Array a:

781 8641 9819 **14287** 15229 21020 24044

Sorted Array b:

6223 **11311** 14153 15751 17411 21626 28948 32560 32765

Merged Array:

781 6223 8641 9819 **11311**

Merging Two Sorted Lists

We continue in this way until one of the lists is exhausted.

Then just fill in the rest of the merged list with the remaining values.

Merging Two Sorted Lists

Sorted Array a:

```
781 8641 9819 14287 15229 21020 24044
```

Sorted Array b:

```
6223 11311 14153 15751 17411 21626
  28948 32560 32765
```

Merged Array:

```
781 6223 8641 9819 11311 14153 14287
  15229 15751 17411 21020 21626 24044
```

Merging Two Sorted Lists

Sorted Array a:

781 8641 9819 14287 15229 21020 24044

Sorted Array b:

6223 11311 14153 15751 17411 21626
28948 32560 32765

Merged Array:

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

Merge

```
void merge(int left[], int left_size, int right[],
           int right_size, int destination[]){

  int left_i = 0, right_i = 0, destination_i = 0;

  while((left_i < left_size) && (right_i < right_size))
    if(left[left_i] < right[right_i])
      destination[destination_i++] = left[left_i++];
    else
      destination[destination_i++] = right[right_i++];
  while(left_i < left_size)
    destination[destination_i++] = left[left_i++];
  while(right_i < right_size)
    destination[destination_i++] = right[right_i++];
}
```

Merge Sort (Variation)

```
static void _merge_sort(int arr[], int size, int temporary[])
{
  int half = size / 2;

  if(size <= 1) return;

  _merge_sort(arr, half, temporary);

  _merge_sort(arr + half, size - half, temporary + half);

  merge(arr, half, arr + half, size - half, temporary);

  memcpy(arr, temporary, size * sizeof (int));

  return;
}
```

What's so great about mergesort?

- Insertion sort, Selection sort, Bubble sort all take time $O(n^2)$ to sort n values.
- The call tree for mergesort shows that it takes $O(n \log n)$ time.
- For large data sets that is a tremendous improvement
- Mergesort is one of a group of very efficient sorting algorithms that are used in most applications.