IM 1003: Computer Programming
Recursion, searching, and sorting

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Outline

- **Recursion**
- Searching
- Sorting

Recursive functions

- A function is **recursive** if it invokes itself (directly or indirectly).
- The technique of writing recursive functions is noted as **recursion**.
- Why recursion?
  - Many problems can be solved by dividing the original problem into several smaller pieces of **subproblems**.
  - Typically one or some subproblems are quite similar to the original problem.
  - Recursion are sometimes intuitive for implementing this strategy.

Recursive functions: finding the max

- As an example, suppose we want to find the maximum number in an array $A[1..n]$ (which means A is of size n).
  - Strategy 1: Write a loop to compare each number to the current maximum.
  - Strategy 2: First find the maximum of $A[1..(n-1)]$, then compare that number with $A[n]$.
- In strategy 2, we divide the original task into two subtasks:
  - Find the maximum of $A[1..(n-1)]$.
  - Compare that number with $A[n]$.
- Let's visualize this strategy!
- While subtask 2 is **simple**, subtask 1 is **similar** to the original task.
  - In particular, it can be solved with the **same** strategy!

Recursive functions: finding the max

- Let's try to implement strategy 2.
- First, I know I need to write a function whose header is:

```
double max (double array[], int len);
```

  - This function returns the maximum among **array** elements 1 to **len**.
  - I **want** this to happen, though at this moment I do not know how.
- Now let's implement strategy 2:
  - If the function really works, subtask 1 will be completed by invoking

```
double subMax = max (array, len - 1);
```

  - Subtask 2 is done by comparing **subMax** and **array[len - 1]**.

Recursive functions: finding the max

- The (wrong) complete implementation is

```
double max (double array[], int len)
{
  double subMax = max (array, len - 1);
  if (array[len - 1] > subMax)
    return array[len - 1];
  else
    return subMax;
}
```

- What will happen if we really invoke this function?

Recursive functions: finding the max

- If we really invoke this function, the program will not terminate!
  - In particular, even when the argument len is 1 in an invocation, we will still try to invoke max (array, 0), which does not make sense.
- When we are facing an array whose size is 1, the task should be solved without using the strategy.
  - It can be solved trivially: That single number is the maximum!
- With this, we can add a **stopping condition** into our function.

Recursive functions: finding the max

- The correct complete implementation is:
- Example "**08_01_max**".
- Note that both **else** can be removed. Why?

```
double max (double array[], int len)
{
  if (len == 1) // stopping condition
    return array[0];
  else
  {
    // recursive call
    double subMax = max (array, len - 1);
    if (array[len - 1] > subMax)
      return array[len - 1];
    else
      return subMax;
  }
}
```

Recursive functions: factorial

- How to write a function that computes the factorial of n?
  - Strategy 1: Write a loop to multiply 1, 2, …, and n.
  - Strategy 2: First calculate the factorial of $n - 1$, and then multiply it with n.

```
int factorial (int n)
{
  if (n == 1) // stopping condition
    return 1;
  else
    // recursive call
    return factorial (n - 1) * n;
}
```


Recursive functions: factorial

- When we invoke this function with argument 4:
- `factorial(4)`

```
= factorial(3) * 4
= (factorial(2) * 3) * 4
= ((factorial(1) * 2) * 3) * 4
= ((1 * 2) * 3) * 4
= (2 * 3) * 4
= 6 * 4
= 24
```


Some remarks

- There must be a **stopping condition** in a recursive function. Otherwise, the program won't stop.
- In many cases, a recursive strategy can also be implemented by a repetitive strategy.
  - E.g., writing a loop for finding a maximum and factorial.
- Compared with an equivalent iterative function, a recursive implementation is usually **simpler** and **easier to understand**.
- However, it generally uses **more memory spaces** and is **more time-consuming**.
  - Recall that invoking functions has some cost.

Some more examples

- Write a recursive function to find the nth Fibonacci number.
  - The Fibonacci sequence is 1, 1, 2, 3, 5, 8, 13, 21, …. Each number is the sum of the two proceeding numbers.
  - Finding the nth number can be done if we know the $(n - 1)$th and $(n - 2)$th numbers.

```
int fib (int n)
{
  if (n == 1)
    return 1;
  else if (n == 2)
    return 1;
  else // two recursive calls
    return (fib (n-1) + fib (n-2));
}
```

Some more examples

- Write a recursive function to compute the greatest common divisor of two integers.

```
int gcd (int p, int q)
{
  if (p == 0)
    return q;
  else if (q == 0)
    return p;
  else
  {
    int r = p % q;
    return gcd (q, r);
  }
}
```


Complexity issue of recursion

- In some cases, recursion is efficient enough.
  - E.g., finding the maximum, factorial, and greatest common divisor.
- In some cases, however, recursion can be very **inefficient**!
  - E.g., Fibonacci.
- Let's compare the efficiency of two different implementations of the Fibonacci problem.
  - Example "**08_02_fibonacci**".
- Why the recursive implementation is inefficient?

Power of recursion

- Though recursion is sometimes inefficient, typically implementation is easier.
- When the efficiency is not a big issue, writing recursion for problem solving is a good idea.
- Let's consider the classic example "Hanoi Tower".
  - Example "**08_03_hanoi**".
  - Is there a good way of solving the Hanoi Tower problem with repetition?

Outline

- Recursion
- **Searching**
- Sorting

Searching

- One fundamental task in computation is to **search** for an element.
  - We want to determine whether an element exists in a set.
  - If yes, we want to locate that element.
  - E.g., looking for a string in an article.
- Here we will discuss how to search for an integer in an one-dimensional array.
- Whether the array is **sorted** makes a big difference.

Searching

- Consider an integer array $A[1..n]$ and a possible element p.
- How to determine whether p exists in A?
- If so, where is it?
  - Assume that we only need to find one p even if there are multiple.
- Suppose the array is unsorted.
- One of the most straightforward way is to apply a **linear search**.
  - Compare each element with p **one by one**, from the first to the last.
  - Whenever we find a match, report its location.
  - Conclude p does not exist if we end up with nothing.
- The number of instructions we need to execute is roughly proportional to n, the array size.

Binary search

- What if the array is sorted?
- Certainly we may still apply the linear search.
- However, we may improve the efficiency by implementing a **binary search**.
  - First, we compare p with the median m (e.g., $A[(n + 1) / 2]$ if n is odd).
  - If p equals m, bingo!
  - If $p < m$, we know p must exist in **the first half** of A if it exists.
  - If $p > m$, we know p must exist in **the second half** of A if it exists.
  - For the latter two cases, we will continue searching in the **subarray**. Seems to be familiar with something... ?
- Example "**08_04_binarySearch**".

Linear search vs. binary search

- In binary search, the number of instructions to be executed is roughly proportional to... what?

- So binary search is much more efficient than linear search!
  - The difference is huge is the array is large.
  - However, binary search is possible only if the array is sorted.
  - Is it worthwhile to sort an array before we search it?
- Binary search can also be implemented with repetition.
  - Is it natural to do so?

Outline

- Recursion
- Searching
- **Sorting**

Sorting

- Given a one-dimensional integer array A of size n, how to sort it?
- There are many different ways, and here we want to introduce two well-known methods:
  - Insertion sort.
  - Merge sort.

Insertion sort

- Given numbers 6, 9, 3, 4, and 7, how would you sort them?
- Imagine that you are playing poker:
  - First put the first number 6 aside.
  - Take the second number 9 and compare it with 6. Because 9 > 6, put 9 to the right of 6.
  - Take the third number 3 and compare it with the **sorted list** (6, 9). Because 3 < 6, put 3 to the left of 6.
  - Take the fourth number 4 and compare it with the sorted list (3, 6, 9). Because 3 < 4 < 6, **insert** 4 in between 3 and 6.
  - Take the fifth number 7 and compare it with the sorted list (3, 4, 6, 9). Because 6 < 7 < 9, insert 7 in between 6 and 9.
  - The result is (3, 4, 6, 7, 9).

Insertion sort

- The key is to maintain a sorted list.
- Then for each number in the unsorted list, insert it into the proper location so that the sorted list **remains sorted**.
- How would you implement the insertion sort?
  - Recursion or repetition?
  - If recursion, why?
- Example "**08_05_insertionSort**".
- Roughly how many instructions do we need for insertion sort?
- Does binary search help?

Mergesort (Merge sort)

- Insertion sort is **simple** and fast!
  - Not really "fast", but faster than many similar sorting algorithm.
  - Because its idea and implementation is simple, it is faster than most algorithms when the array size is **small**.
- Interestingly, there is another sorting algorithm:
  - Its idea is somewhat similar to insertion sort.
  - But it is significantly faster!
- This algorithm is called **mergesort**.

Mergesort (Merge sort)

- Recall that in an insertion sort, we need to insert one number into a sorted list for many times.
- A key observation is that "inserting" **another sorted list** of size k into a sorted list (so such "inserting" is actually "**merging**") can be faster than inserting k separate numbers one by one!
- Given an unsorted array, we will:
  - First split the array into two parts, the first half and second half.
  - Then sort each subarray.
  - Finally, merge these two subarrays.
- Mergesort is perfect for recursion!

Mergesort (Merge sort)

- Interestingly, insertion sort is a special way of running mergesort.
  - Not splitting the array into two halves but split it into $A[1..n-1]$ and $A[n]$.
- Once we use the "smart split", the **efficiency** is improved a lot!
  - Insertion sort: Roughly proportional to n^2.
  - Merge sort: Roughly proportional to $n \log n$.
- A simple observation can make a huge difference!