An Introduction to Programming through C++

with

Manoj Prabhakaran

Lecture 16

Recursion (ctd.)

Based on material developed by Prof. Abhiram G. Ranade

We saw

- Drawing fractals 
- Permutations
- GCD (and Bézout's Identity)

Today

- Tail Recursion
  - Example: Binary Search, GCD
- Divide-and-Conquer
  - Example: Merge Sort
- Memo-ization
  - Example: Fibonacci sequence

Searching in an Array

- Given an array of (say) integers, check for the absence/presence of various numbers in the array (return an index, or -1)

```
int search(float A[], float query, int size);
```

- Simplest idea:

```
for (int i=0; i<size; i++)
  if (A[i]==query) return i;
return -1;
```

- Can we do better? Yes, if the array is sorted, *much better*!

Binary Search

- Consider searching for a number, say 3.14 in an array of sorted numbers
- Let us look at the middle first!

- We can eliminate half of the array from consideration now!
- To search in the rest of the array, we *recurse*!
  - Look at the middle of the remaining array
  - Eliminate half again
- Until a match found (a middle element equals the target) or all of the array eliminated

Binary Search

```
int srch (const float A[], const float& target, int left, int right) {
    if (right < left) return -1;              // empty array: target not found
    int mid = (left+right)/2;
    if (target < A[mid])
      return srch(A,target,left,mid-1);       // recurse to the left of mid
    else if (A[mid] < target)
      return srch(A,target,mid+1,right);      // recurse to the right of mid
    else // target == A[mid]
      return mid;                             // found!
}
```

Example: (Integer) Square Root

- Find ⌊ √x ⌋ using binary search!
- Search in the implicit "sorted array" { 0, 1, ⋯ , x }

```
int isqrt(int x, int left=0, int right=0) {
    if(right==0) right = x; // set range [0..x] if called without a range
    if (right < left)       // we just passed the square-root
        return right;       // right has the smaller number
    int mid = (left+right)/2;
    int sqr = mid*mid;      // to check √x <=> mid, will check x <=> sqr
    if (x < sqr)
        return isqrt(x,left,mid-1);
    else if (sqr < x)
        return isqrt(x,mid+1,right);
    else // sqr == x
        return mid;
}
```

Example: Square Root

- Find √x using binary search (to a desired accuracy)
- Search in the implicit "sorted array" of real numbers!

Adds one bit of precision in each iteration. Faster methods exist.

```
double sqrt(double x, double left=0, double right=0) {
    const double delta = 1e-12;
    if(right==0) right = x;     // set range to [0,x] if no range given
    if (right < left + delta)   // small enough interval
        return left;            // left has the smaller number
    double mid = (left+right)/2; double sqr = mid*mid;
    if (x < sqr)
        return sqrt(x,left,mid);
    else if (x > sqr)
        return sqrt(x,mid,right);
    else // sqr == x
        return mid;
}
```

Tail Recursion

- In many cases of recursion, the return value is the same as what the recursive call returns
- Such a recursion can be rewritten as a loop
- Avoids overheads of function calls
  - Like adding a new frame to the function-call stack, copying arguments and return values
- Compilers may often do such rewriting automatically

```
resultType f(inputType x) {

  if (baseCase(x))
    return handleBaseCase(x);

  x = smallerProblem(x);

  return f(x);
}
```

```
resultType f(inputType x) {
  while (!baseCase(x)) {
    x = smallerProblem(x);
  }
  return handleBaseCase(x);
}
```

Tail Recursion

```
!( right < left
   || target == A[mid] )
```

```
int srch (float A[], float target, int left, int right) {
  if (right < left)
    return -1;
  int mid = (left+right)/2;
  if (target == A[mid])
      return mid;

  if (target < A[mid])
    right = mid-1;
  else
    left = mid+1;

  return srch(A,target,left,right);
}
```

```
int srch_loop(float A[], float target, int n) {
  int left = 0, right = n-1, mid;
  while (right >= left
           && target != A[mid=(left+right)/2]) {
    if(target < A[mid]) right = mid-1;
    else left = mid+1;
  }
  return (right < left) ? -1 : mid;
}
```

```
resultType f(inputType x) {
  while (!baseCase(x)) {
    x = smallerProblem(x);
  }
  return handleBaseCase(x);
}
```

Tail Recursion

```
int gcd (int a, int b) {
  if(b==0)
    return abs(a);
  return gcd(b,a%b);
}
```

```
int gcd_loop(int a, int b) {
  while (b!=0) {
    // update (a,b) to (b,a%b)
    std::swap(a,b); b %= a;
  }
  return abs(a);
}
```

```
resultType f(inputType x) {

  if (baseCase(x))
    return handleBaseCase(x);

  x = smallerProblem(x);

  return f(x);
}
```

```
resultType f(inputType x) {
  while (!baseCase(x)) {
    x = smallerProblem(x);
  }
  return handleBaseCase(x);
}
```

Divide-and-Conquer

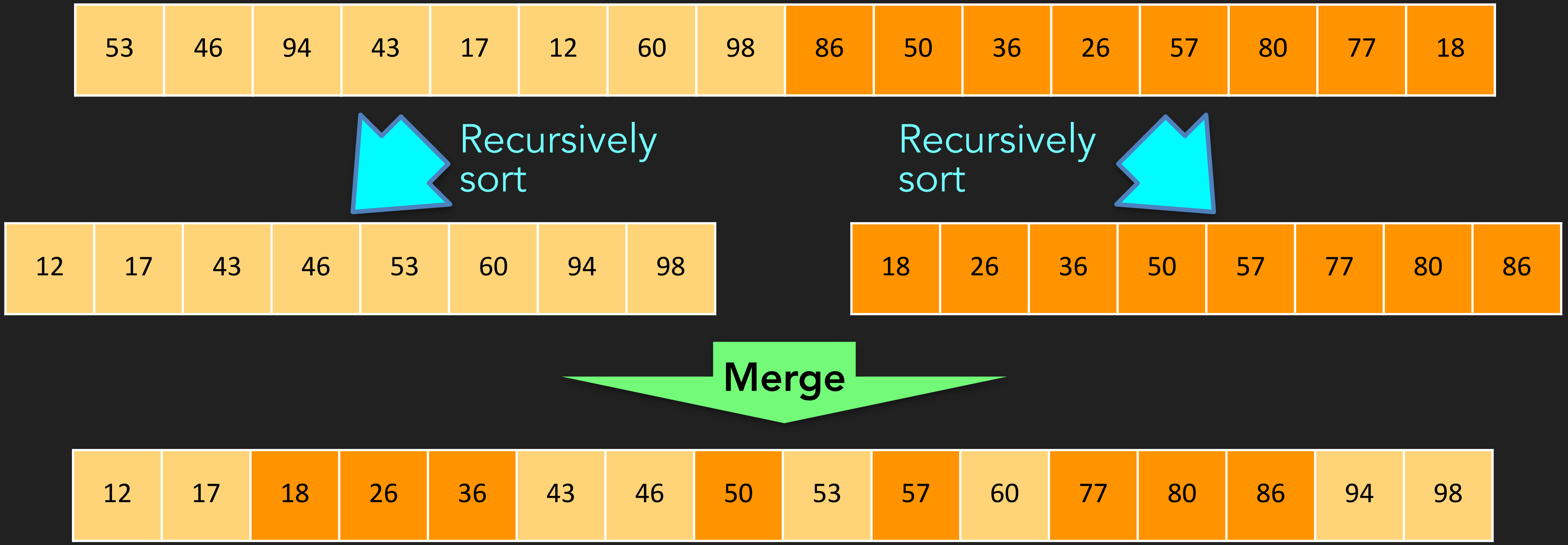
- Some *algorithms* use a Divide-and-Conquer strategy
  (a.k.a. Divide-Conquer-and-Combine)
  - A problem instance is divided into two or more smaller problems
  - The smaller instances are solved recursively
  - The results are then combined to get the result for the original instance
- Implicit in drawing fractals:

- Another example: Merge Sort

```
resultType f(inputType x) {

  if (baseCase(x))
    return handleBaseCase(x);

  inputType x1, x2;
  Divide(x,x1,x2);

  resultType y1, y2;
  y1 = f(x1); y2 = f(x2);

  return Combine(y1,y2);
}
```

Merge Sort

- Split into two (almost) equal halves, and recursively sort each half
- Merge the two sorted arrays into a single sorted array



Merge Sort

```
// merge X[left..mid] and X[mid+1..right] into Y[left..right], where mid = (left+right)/2
void merge(const int X[], int Y[], int left, int right) {
    int mid = (left+right)/2, L = left, R = mid+1;   // L,R: next indices of left/right halves
    for(int i=left; i <= right; ++i) {
        if(L <= mid && (R > right || X[L] <= X[R])) Y[i] = X[L++];    // copy from left
        else Y[i] = X[R++];                                             // copy from right
    }
}
```

12	17	43	46	53	60	94	98

18	26	36	50	57	77	80	86

Merge

12	17	18	26	36	43	46	50	53	57	60	77	80	86	94	98

Merge Sort

Output will be in the array out[ ] in indices left,...,right. A temporary array scratch[ ] passed as input (since its size is not known at compile-time).

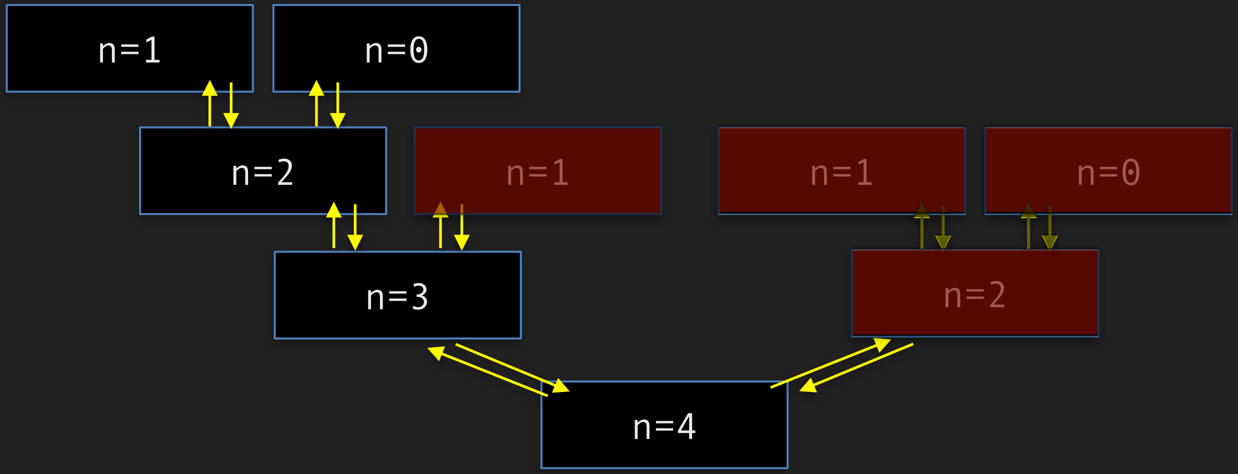
```
#include <cassert>
void sort (const int in[], int out[],
           int left, int right,
                     int scratch[]) {
  assert(left <= right);
  if (left==right) {
    out[left] = in[left];
    return;
  }
  int mid = (left+right)/2;
  sort(in,scratch,left,mid,out);
  sort(in,scratch,mid+1,right,out);

  merge(scratch,out,left,right);
}
```

```
resultType f(inputType x) {

  if (baseCase(x))
    return handleBaseCase(x);

  inputType x1, x2;
  Divide(x,x1,x2);

  resultType y1, y2;
  y1 = f(x1); y2 = f(x2);

  return Combine(y1,y2);
}
```

Recall: Fibonacci Sequence

```
int Fibonacci(unsigned int n) {
  if(n==0) return 0;
  if(n==1) return 1;
  return Fibonacci(n-1) + Fibonacci(n-2);
}
```

} Base cases



A very inefficient implementation!

Because it is evaluated on the same input again and again

Why not just store the results of earlier evaluations?

Memoization

Memo-ization

Demo

- A simple, <u>generic</u> way to drastically improve the efficiency of many recursive implementations
  - <u>Generic</u>: works without looking into the specifics of the algorithm

```
int memoFib(unsigned n, Memo& memo) {
  if memo.has(n) return memo.get(n);
  int ans = (n<=1)? n : memoFib(n-1,memo) + memoFib(n-2,memo);
  memo.add(n,ans);
  return ans;
}
```

  - For Fibonacci, it would have been enough to have the largest two values memo-ized

A Few New Things (A Preview)

```
struct Memo {
    static const int NMAX = 90;
    bool filled[NMAX] = {};
    unsigned long long memo[NMAX];

    bool has(unsigned i) { return (i<NMAX) && filled[i];
    unsigned long long get(unsigned i);
    void add(unsigned i, unsigned long long val) {
        if(i>=NMAX)
          throw std::invalid_argument("Out of memo-ization range!");
        memo[i] = val; filled[i] = true;
    }
};
```

A static member in a struct:
There will be only one copy for all the instances.
Can be accessed without instances, e.g., as Memo::NMAX

Members can be initialised in the struct declaration

Throwing an "exception". Will cause the program to exit.
Later: How to handle an exception without exiting.

Today

- Tail Recursion
  - Example: Binary Search, GCD
  - Can be turned into loops
    - Your compiler will often do this for you
- Divide-and-Conquer
  - Example: Merge Sort
- Memo-ization
  - Example: Fibonacci sequence
- Along the way, a peek at Exceptions: throw