CSC 222: Object-Oriented Programming

Spring 2012

recursion & sorting

- recursive algorithms
- base case, recursive case
- silly examples: fibonacci, GCD
- real examples: merge sort, quick sort
- recursion vs. iteration
- recursion & efficiency

O(N log N) sorts

there are sorting algorithms that do better than insertion & selection sorts

merge sort & quick sort are commonly used O(N log N) sorts

- recall from sequential vs. binary search examples:
  when N is large, log N is much smaller than N
- thus, when N is large, N log N is much smaller than N^2

N	N log N	N^2
1,000	10,000	1,000,000
2,000	22,000	4,000,000
4,000	48,000	16,000,000
8,000	104,000	64,000,000
16,000	224,000	256,000,000
32,000	480,000	1,024,000,000

they are both recursive algorithms
i.e., each breaks the list into pieces, calls itself to sort the smaller pieces, and combines the results

Recursion

a *recursive algorithm* is one that refers to itself when solving a problem

- to solve a problem, break into smaller instances of problem, solve & combine
- recursion can be a powerful design & problem-solving technique
  examples: binary search, merge sort, hierarchical data structures, …

classic (but silly) examples:

Fibonacci numbers:
1st Fibonacci number = 1
2nd Fibonacci number = 1
Nth Fibonacci number = (N-1)th Fibonacci number + (N-2)th Fibonacci number

Euclid's algorithm to find the Greatest Common Divisor (GCD) of a and b (a ≥ b)

- if a % b == 0, the GCD(a, b) = b
- otherwise, GCD(a, b) = GCD(b, a % b)

3

Recursive methods

```
/**
 * Computes Nth Fibonacci number.
 *  @param N sequence index
 *  @returns Nth Fibonacci number
 */
public int fibonacci(int N)
{
  if (N <= 2) {
    return 1;
  }
  else {
    return fibonacci(N-1) +
           fibonacci(N-2);
  }
}
```

```
/**
 * Computes Greatest Common Denominator.
 *  @param a a positive integer
 *  @param b positive integer (a >= b)
 *  @returns GCD of a and b
 */
public int GCD(int a, int b)
{
  if (a % b == 0) {
    return b;
  }
  else {
    return GCD(b, a%b);
  }
}
```

these are classic examples, but pretty STUPID

- both can be easily implemented using iteration (i.e., loops)
- recursive approach to Fibonacci has huge redundancy

we will look at better examples later, but first analyze these simple ones

4

Understanding recursion

every recursive definition has 2 parts:

BASE CASE(S): case(s) so simple that they can be solved directly

RECURSIVE CASE(S): more complex – make use of recursion to solve *smaller* subproblems & combine into a solution to the larger problem

```
int fibonacci(int N)
{
  if (N <= 2) {    // BASE CASE
    return 1;
  }
  else {          // RECURSIVE CASE
    return fibonacci(N-1) +
           fibonacci(N-2);
  }
}
```

```
int GCD(int a, int b)
{
  if (a % b == 0) { // BASE CASE
    return b;
  }
  else {             // RECURSIVE
    return GCD(b, a%b);
  }
}
```

to verify that a recursive definition works:

- convince yourself that the base case(s) are handled correctly
- ASSUME RECURSIVE CALLS WORK ON SMALLER PROBLEMS, then convince yourself that the results from the recursive calls are combined to solve the whole

5

Avoiding infinite(?) recursion

to avoid infinite recursion:

- must have at least 1 base case (to terminate the recursive sequence)
- each recursive call must get *closer* to a base case

```
int fibonacci(int N)
{
  if (N <= 2) {    // BASE CASE
    return 1;
  }
  else {          // RECURSIVE CASE
    return fibonacci(N-1) +
           fibonacci(N-2);
  }
}
```

```
int GCD(int a, int b)
{
  if (a % b == 0) { // BASE CASE
    return b;
  }
  else {             // RECURSIVE
    return GCD(b, a%b);
  }
}
```

with each recursive call, the number is getting smaller → closer to base case (≤ 2)

with each recursive call, a & b are getting smaller → closer to base case (a % b == 0)

6

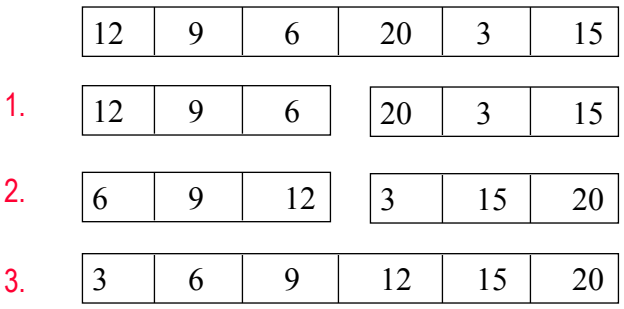
Merge sort

a better example of recursion is merge sort

BASE CASE: to sort a list of 0 or 1 item, DO NOTHING!

RECURSIVE CASE:

1. Divide the list in half
2. Recursively sort each half using merge sort
3. Merge the two sorted halves together



7

Merging two sorted lists

merging two lists can be done in a single pass

- since sorted, need only compare values at front of each, select smallest
- requires additional list structure to store merged items

```
public <T extends Comparable<? super T>> void merge(ArrayList<T> items, int low, int high) {
    ArrayList<T> copy = new ArrayList<T>();

    int size = high-low+1;
    int middle = (low+high+1)/2;
    int front1 = low;
    int front2 = middle;
    for (int i = 0; i < size; i++) {
        if (front2 > high ||
            (front1 < middle && items.get(front1).compareTo(items.get(front2)) < 0)) {
            copy.add(items.get(front1));
            front1++;
        }
        else {
            copy.add(items.get(front2));
            front2++;
        }
    }

    for (int k = 0; k < size; k++) {
        items.set(low+k, copy.get(k));
    }
}
```

8

Merge sort

once merge has been written, merge sort is simple

- for recursion to work, need to be able to specify range to be sorted
- initially, want to sort the entire range of the list (index 0 to list size – 1)
- recursive call sorts left half (start to middle) & right half (middle to end)
- . . .

```
private <T extends Comparable<? super T>> void mergeSort(ArrayList<T> items, int low, int high) {
    if (low < high) {
        int middle = (low + high)/2;
        mergeSort(items, low, middle);
        mergeSort(items, middle+1, high);
        merge(items, low, high);
    }
}

public <T extends Comparable<? super T>> void mergeSort(ArrayList<T> items) {
    mergeSort(items, 0, items.size()-1);
}
```

note: private helper method does the recursion; public method calls the helper with appropriate inputs

9

Big-Oh revisited

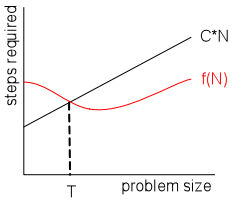
intuitively: an algorithm is O(f(N)) if the # of steps involved in solving a problem of size N has f(N) as the dominant term

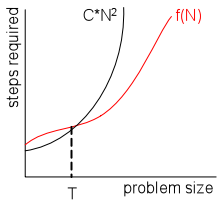
O(N):	5N	3N + 2	N/2 – 20
O(N^2):	N^2	N^2 + 100	$10N^2$ – 5N + 100

…

more formally: an algorithm is O(f(N)) if, *after some point*, the # of steps can be bounded from above by a scaled f(N) function

O(N): if number of steps can eventually be bounded by a line
O(N^2): if number of steps can eventually be bounded by a quadratic
…

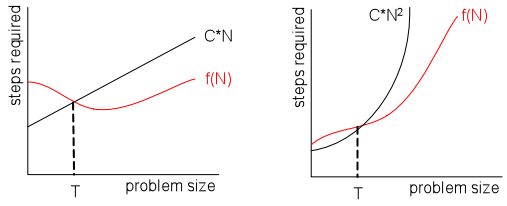




10

Technically speaking…

an algorithm is O(f(N)) if there exists a positive constant C & non-negative integer T such that for all N ≥ T, # of steps required ≤ C*f(N)



for example, insertion sort:

N(N-1)/2 shifts + N inserts + overhead = ($N^2/2 + N/2 + X$) steps

if we consider N ≥ X (i.e., let T = X), then

($N^2/2 + N/2 + X$) ≤ ($N^2/2 + N^2/2 + N^2$) = $2N^2 = CN^2$ (where C = 2) ➔ $O(N^2)$

11

Recursive analysis of a recursive algorithm

cost of sorting N items = cost of sorting left half (N/2 items) +
cost of sorting right half (N/2 items) +
cost of merging (N items)

more succinctly: Cost(N) = 2*Cost(N/2) + C_1*N

Cost(N) = 2*Cost(N/2) + C_1*N	can unwind Cost(N/2)
= 2*(2*Cost(N/4) + C_2*N/2) + C_1*N	
= 4*Cost(N/4) + (C_1 + C_2)*N	can unwind Cost(N/4)
= 4*(2*Cost(N/8) + C_3*N/4) + (C_1 + C_2)*N	
= 8*Cost(N/8) + (C_1 + C_2 + C_3)*N	can continue unwinding
= …	
= N*Cost(1) + (C_1 + C_2/2 + C_3/4 + … + C_{logN}/N)*N	
= (C_0 + C_1 + C_2 + C_3 + … + C_{logN})*N	where C_0 = Cost(1)
≤ (max(C_0, C_1,…, C_{logN})*log N) * N	
= C * N log N	where C = max(C_0, C_1,…, C_{logN})
➔ O(N log N)	

12

`Dictionary` revisited

recall most recent version of `Dictionary`

- inserts each new word in order (i.e., insertion sort) & utilizes binary search
  ➔searching is fast (binary search), but adding is slow
- N adds + N searches: N*O(N) + N*O(log N) = $O(N^2)$ + O(N log N) = $O(N^2)$

if you are going to do lots of adds in between searches:

- simply add each item at the end → O(1)
- before the first search, must sort – could use merge sort
- N adds + sort + N searches: N*O(1) + O(N log N) + N*log(N) = O(N log N)

`Collections` class contains a `sort` method that implements *quick sort*

- in practice, quick sort is a faster O(N log N) sort than merge sort

1. picks a *pivot* element from the list (can do this at random or be smarter)
2. partitions the list so that all items ≤ pivot are to left, al items > pivot are to right
3. recursively (quick) sorts the partitions

13

Modified `Dictionary` class

the `isSorted` field keeps track of whether the list is sorted (i.e., no `addWord`s have been performed since last `findWord`)

we could do a little more work in `addWord` to avoid unnecessary sorts

note: gives better performance if N adds are followed by N searches

what if the adds & searches alternate?

```
public class Dictionary2 {
    private ArrayList<String> words;
    private boolean isSorted;

    public Dictionary2() {
        this.words = new ArrayList<String>();
        this.isSorted = true;
    }

    public Dictionary2(String fileName) {
        this();      // note: once constructor can call another
        try {
            Scanner infile = new Scanner(new File(fileName));
            while (infile.hasNext()) {
                String nextWord = infile.next();
                this.addWord(nextWord);
            }
            infile.close();
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("No such file: " + fileName);
        }
    }

    public boolean addWord(String newWord) {
        this.words.add(newWord.toLowerCase());
        this.isSorted = false;
        return true;
    }

    public boolean findWord(String desiredWord) {
        if (!this.isSorted) {
            Collections.sort(this.words);
            this.isSorted = true;
        }
        return Collections.binarySearch(this.words,
                                desiredWord.toLowerCase()) >= 0;
    }

    . . .
}
```

14

Dictionary2 timings

```
import java.util.Random;

public class TimeDictionary {
  public static int timeAdds(int numValues) {
    Dictionary2 dict = new Dictionary2();
          Random randomizer = new Random();

    long startTime = System.currentTimeMillis();
    for (int i = 0; i < numValues; i++) {
      String word = "0000000000" + randomizer.nextInt();
      dict.addWord(word.substring(word.length()-10));
    }

    for (int i = 0; i < numValues; i++) {
        dict.findWord("zzz");
    }
    long endTime = System.currentTimeMillis();

    return (int)(endTime-startTime);
  }
}
```

N adds followed by N searches:

- Dictionary1 used insertion sort & binary search
- $O(N^2) + O(N \log N) \rightarrow O(N^2)$
- Dictionary2 uses add-at-end, quick sort before first search, then binary search
- $O(N) + O(N \log N) + O(N \log N) \rightarrow O(N \log N)$

# items (N)	Dictionary1 (msec)	Dictionary2 (msec)
100,000	2121	176
200,000	8021	386
400,000	31216	864

15

Recursion vs. iteration

it wouldn't be difficult to code `fibonacci` and `GCD` without recursion

```
public int fibonacci(int N) {
  int previous = 1;
  int current = 1;
  for (int i = 3; i <= N; i++) {
    int newCurrent = current + previous;
    previous = current;
    current = newCurrent;
  }
  return current;
}
```

```
public int GCD(int a, int b) {
  while (a % b != 0) {
    int temp = b;
    b = a % b;
    a = temp;
  }
  return b;
}
```

in theory, any recursive algorithm can be rewritten iteratively (using a loop)

- but sometimes, a recursive definition is MUCH clearer & MUCH easier to write
  e.g., merge sort

16

Recursion & efficiency

there is some overhead cost associated with recursion

```
public int fibonacci(int N) {
  int previous = 1;
  int current = 1;
  for (int i = 3; i <= N; i++) {
    int newCurrent = current + previous;
    previous = current;
    current = newCurrent;
  }
  return current;
}
```

```
public int GCD(int a, int b) {
  while (a % b != 0) {
    int temp = b;
    b = a % b;
    a = temp;
  }
  return b;
}
```

- with recursive version: each refinement requires a method call
  involves saving current execution state, allocating memory for the method instance, allocating and initializing parameters, returning value, …
- with iterative version: each refinement involves a loop iteration + assignments

the cost of recursion is relatively small, so usually no noticeable difference

- in practical terms, there is a limit to how deep recursion can go
  e.g., can't calculate the 10 millionth fibonacci number
- in the rare case that recursive depth can be large (> 1,000), consider iteration

17

Recursion & redundancy

in the case of GCD, there is only a minor efficiency difference

- number of recursive calls = number of loop iterations

this is not always the case → efficiency can be significantly different

(due to different underlying algorithms)

consider the recursive `fibonacci` method:

```
                          fibonacci(5)
           fibonacci(4)        +           fibonacci(3)
   fibonacci(3)  +  fibonacci(2)     fibonacci(2) + fibonacci(1)
fibonacci(2) + fibonacci(1)
```

- there is a SIGNIFICANT amount of redundancy in the recursive version
  number of recursive calls > number of loop iterations (by an exponential amount!)
- recursive version is MUCH slower than the iterative one
  in fact, it bogs down on relatively small values of N

18