

## Midterm Exam Review

This handout contains some problems that cover material covered on the midterm exam. Reviewing these problems might (but might not!) help you with problems on the midterm. The Spring'01 midterm consisted of problems 1–4 and *CLRS* 9.3-7/*CLR* 10.3-7 (Problem 3 on Fall'01 PS5).

### Problem 1: Asymptotic Notation

Consider the function  $f(n) = n^{(2+\sin(n))}$ . For each of the following functions  $g_i$ , use "yes" or "no" to indicate whether  $g_i$  is  $o(f)$ ,  $O(f)$ ,  $\Theta(f)$ ,  $\Omega(f)$ , or  $\omega(f)$ . For full credit, you should justify each answer. In many cases, a single analysis can justify several answers. *Hint*: Draw a picture of  $f$ .

	$o(f)$	$O(f)$	$\Theta(f)$	$\Omega(f)$	$\omega(f)$
$g_1(n) = 2$					
$g_2(n) = n + \sin(n)$					
$g_3(n) = n^2$					
$g_4(n) = n^3$					

### Problem 2: Coin Flipping

(This problem is based on *CLR* Exercise 6.2-4.)

Given a biased coin with probability  $p$  ( $0 < p < 1$ ) of getting a head on every flip, develop an algorithm that uses this unfair coin to simulate a flip of a fair coin (one that has 0.5 probability of a head on any flip). Do this via the following parts:

- a. Assume that you are given an `Unfair-Flip()` function that simulates flipping the biased coin by returning H with probability  $p$  and T with probability  $(1 - p)$ . Define a function `Fair-Flip()` that returns H or T with equal probability. Give explicit pseudocode for your function.

*Hints/notes:*

- It is possible to write a solution in just a few lines of code.
  - It helps to flip the coin multiple times.
  - Do not adopt a strategy that attempts to determine the value of  $p$ !
  - It may to think in terms of the paper-tearing exercise we did in class.
- b. Argue that your definition of `Fair-Flip()` is correct – i.e., that it returns each of H and T with probability 0.5.
- c. Calculate (as a function of  $p$ ) the expected number of times that your algorithm flips the biased coin to simulate a single fair coin flip.

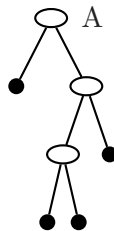
### Problem 3: Sorting Identical Elements

For each of the following array sorting algorithms, give the running time of the algorithm on an array of  $n$  identical elements. (I.e.,  $\text{leq}(\mathbf{a}, \mathbf{b})$  and  $\text{leq}(\mathbf{b}, \mathbf{a})$  are both true for every pair of elements in the array.)

- Insertion Sort
- Selection Sort
- Merge Sort
- Quick Sort (using Lomuto Partition)
- Quick Sort (using Two-Finger Partition)
- Tree Sort (in the binary search tree insertion, assume that elements  $\text{leq}$  the root are inserted in the left subtree, and those  $\text{gt}$  the root are inserted in the right subtree.)

### Problem 4: A Balancing Act

This problem involves some binary tree algorithms. For the purposes of this problem, a binary tree is either (1) a leaf or (2) a (valueless) node with left and right subtrees. For example, the following figure depicts a binary tree A in which nodes are represented by white ovals and leaves are represented as black circles.



We have the following functions for manipulating binary trees:

#### **Leaf()**

Returns a new leaf.

#### **Leaf?( $t$ )**

Returns true if  $t$  is a leaf, and false otherwise.

#### **Node( $l, r$ )**

Returns a new node whose left and right subtrees are  $l$  and  $r$ , respectively.

#### **Left( $t$ )**

Returns the left subtree of a node  $t$ .

#### **Right( $t$ )**

Returns the right subtree of a node  $t$ .

For example, the sample tree A depicted above is constructed by the following expression:

```
Node(Leaf(),
     Node(Node(Leaf(), Leaf()),
          Leaf()))
```

The **height** of a binary tree is the length of the longest path of Left and Rights from the root of the tree to a leaf. For example, the height of A is 3, the height of Left(A) is 0, and the height of Right(A) is 2.

A binary tree is **balanced** if, for every node in the tree, the height of its left and right subtrees differ by no more than 1. For example, Right(A) is balanced, but A is not (because the heights of the left and right subtrees of its root node differ by 2.)

Consider the following algorithm for determining whether a binary tree is balanced:

```
Balanced?(T)
  if Leaf?(T) then
    return true
  else
    return (|Height(Left(T)) - Height(Right(T))| ≤ 1
            and Balanced?(Left(T))
            and Balanced?(Right(T))
            )
```

In the above algorithm, assume that:

- Height(T) correctly calculates the height of a tree with  $n$  leaves in  $\Theta(n)$  time;
- | n | gives the absolute value of the number  $n$ ;
- $E_1$  and  $E_2$  evaluates both boolean expressions  $E_1$  and  $E_2$  and returns true if both are true and false if at least one is false.

a. Prove that **Balanced?(T)** correctly determines whether a tree T is balanced. That is, prove the following two facts:

1. **Balanced?(T)** returns true whenever T is balanced.
2. **Balanced?(T)** returns false whenever T is not balanced.

Use induction as your proof strategy.

b. Describe the shape of tree on which **Balanced?** exhibits its asymptotic best-case running time as a function of the number  $n$  of leaves in the tree. Give the recurrence equation for this case, and solve the recurrence equation. Express your solution using  $\Theta$  notation.

c. Describe the shape of tree on which **Balanced?** exhibits its asymptotic worst-case running time as a function of the number  $n$  of leaves in the tree. Give the recurrence equation for this case, and solve the recurrence equation. Express your solution using  $\Theta$  notation.

d. Redo parts b and c under the assumption that the two instances of **and** in **Balanced?** are replaced with the short-circuit boolean conjunction operator **&&**. That is,  $E_1$  **&&**  $E_2$  first evaluates  $E_1$ . If the value of  $E_1$  is true, the result of  $E_2$  is returned. But if the value of  $E_1$  is false, a false value is immediately returned without evaluating  $E_2$ . *Note:* the shapes of trees for the best and worst cases are not necessarily the same in this part as in parts b and c.

e. Below is an algorithm that determines both the height and balance of a tree in a single pass over a tree. It returns a pair (h, b) where h is the height of the tree and b is a boolean indicating whether the tree is balanced.

```

Height&Balanced?(T)
  if Leaf?(T) then
    return (0, true)
  else
    (hleft, bleft) ← Height&Balanced?(Left(T))
    (hright, bright) ← Height&Balanced?(Right(T))
    return (1 + max(hleft, hright),
            (|hleft - hright| ≤ 1 and bleft and bright)
            )

```

Using  $\Theta$ -notation, give the asymptotic worst-case running time of `Height&Balanced?` on a binary tree with  $n$  leaves. Justify your answer.

### Problem 5: Inversions

(This problem is inspired by *CLR* Problem 1-3.)

Consider the following Haskell `inversions` function:

```

inversions [] = 0
inversions (x:xs) = (invs x xs) + (inversions xs)

invs y [] = 0
invs y (z:zs) = (if y > z then 1 else 0) + (invs y zs)

```

- a. Give a concise English specification of the `inversions` function. I.e., given a list of integers `ints`, what is the meaning of the number returned by `inversions ints`?
- b. Give the value of each of the following expressions:
  - `inversions [4,3,2,1]`
  - `inversions [1,2,3,4]`
  - `inversions [1..100]` (note: `[1..100]` denotes the list of integers from 1 to 100, inclusive)
  - `inversions (reverse [1..100])`
  - `inversions [5,6,7,8,1,2,3,4]`
- c. What is the worst-case running time of `inversions` on a list of length  $n$ ? Use a recurrence equation to justify your answer.
- d. What is the best-case running time of `inversions` on a list of length  $n$ ? Use a recurrence equation to justify your answer.
- e. Write an alternative implementation of `inversions` that takes worst-case time  $\Theta(n \cdot \lg(n))$  when called on a list of length  $n$ . Show that your alternative implementation has the desired running time. *Hint*: modify the merge sort algorithm.

### Problem 6: Bubble Sort

Below is a Haskell program that expresses a classic sorting algorithm known as “bubble sort”:

```

bsort ws =
  let (zs, changed) = bubble ws
  in if changed
     then bsort zs
     else zs

bubble [] = ([], False)
bubble [x] = ([x], False)
bubble (x:xs) =
  let (y:ys, changed) = bubble xs
  in if x > y
     then (y:x:ys, True)
     else (x:y:ys, changed)

```

- a. Briefly explain in English (1) how `bsort` works and (2) why it terminates.
- b. What is the worst-case running time of `bsort` on a list of length  $n$ ?
- c. What is the best-case running time of `bsort` on a list of length  $n$ ?
- d. Using induction, prove that the Haskell implementation of `bubble` given above satisfies the following specification:

**(bubble1)** If  $\text{sorted}(as)$  then `bubble as = (as, False)`

**(bubble2)** If `bubble as = (cs, b)`, then  $\text{elts}(as) = \text{elts}(cs)$

**(bubble3)** If `bubble as = (cs, True)`, then the number of inversions in  $cs$  is strictly less than the number of inversions in  $as$ . (Consult the previous problem for the notion of inversion.)

- e. Assuming that `bubble` satisfies the specification in part (a), use the method of loop invariants to prove that `bsort` is a correct sorting algorithm.

### Problem 7: Building Binary Search Trees

Suppose that `insert v t` inserts the value  $v$  in the binary search tree  $t$ . Here is a Haskell function that builds a binary search tree from a list of values:

```

buildBST [] = Leaf
buildBST (v:vs) = insert v (buildBST vs)

```

- a. What is the worst-case running time of `buildBST` on a list of length  $n$ ? Use a recurrence equation to justify your answer.
- b. What is the best-case running time of `buildBST` on a list of length  $n$ ? Use a recurrence equation to justify your answer.
- c. Describe the relationship between building a binary tree for a list of  $n$  elements and performing quicksort on a list of  $n$  elements.
- d. **[B]:** based on your answer to part (c), what is the average-case running time of `buildBST` on a list of length  $n$ ?

- e. Design an alternative implementation of `buildBST` that takes worst-case time  $\Theta(n \cdot \lg(n))$ . You may use any of the algorithms we have studied in the course as black boxes.
- f. Argue that it is impossible to build a binary search tree for a list of  $n$  elements in worst-case time  $\Theta(n)$ .

### Problem 8: Range Selection

Given a list  $xs$  of  $n$  distinct integers, and two integers  $i$  and  $j$  such that  $1 \leq i \leq n$  and  $1 \leq j \leq n$ , the function `selectRange( $i, j, xs$ )` returns a list of every element in  $xs$  whose rank  $k$  satisfies  $i \leq k \leq j$ . The order of the elements in the resulting list does not matter; in particular, the resulting list does *not* have to be sorted.

Each subproblem below mentions constraints on the values in  $xs$ . Your goal in each of the subproblems is to design an algorithm for `selectRange` that runs in worst-case time  $\Theta(n)$  subject to the given constraints. In each case, the asymptotic worst-case running time should be *independent* of the particular values of  $i$  and  $j$ . For example, both `selectRange(5,5, $xs$ )` and `selectRange(1, $n$ , $xs$ )` should run in time  $\Theta(n)$ .

All elements of  $xs$  are in the range  $[a..b]$ , where  $b - a + 1 \in O(n)$ .

All elements of  $xs$  are in the range  $[a..b]$ , where  $b - a + 1 \in O(n^2)$ .

All elements of  $xs$  are integers in an unknown range.

### Problem 9: The Powers of Two

Figure 1 contains six Java class methods, all of which compute the value of 2 raised to the  $n$ th power. For each function, give the following:

- A recurrence equation that approximates the worst-case running time  $T(n)$  of the routine as a function of the input  $n$ . In all cases, you may assume that  $T(n) = 0$  for  $n < 1$  and that the overhead of the divide and glue operations within the body of `twoi` has cost that is a constant (for convenience, assume that the constant is 1).
- A solution to the recurrence equation expressed in  $\Theta$  notation.

You may find the following facts helpful in some of your analyses, especially for `two6`.

- $\log_c(a) + \log_c(b) = \log_c(a * b)$
- $\log_c(a) - \log_c(b) = \log_c(a/b)$
- $n! = n * (n - 1) * (n - 2) * \dots * 3 * 2 * 1$
- $\log_c(n!) \in \Theta(n \cdot \log_c(n))$
- if  $f \in O(g)$  but  $f \notin \Theta(g)$ , then  $\Theta(f + g) = \Theta(g)$ .

```

public static int two1 (int n) {
    if (n == 0) {
        return 1;
    } else {
        return 2 * two1(n - 1);
    }
}

public static int two2 (int n) {
    if (n == 0) {
        return 1;
    } else {
        return two2(n - 1) + two2(n - 1);
    }
}

public static int two3 (int n) {
    if (n == 0) {
        return 1;
    } else {
        return two3(n - 1) + two1(n - 1); // Note call to two1.
    }
}

public static int two4 (int n) {
// To simplify analysis, assume that the input to two4 is a power of 2.
    if (n == 0) {
        return 1;
    } else if (n % 2 == 0) {
        int x = two4(n / 2);
        return x * x;
    } else {
        return 2 * two4(n - 1);
    }
}

public static int two5 (int n) {
// To simplify analysis, assume that the input to two5 is a power of 2.
    if (n == 0) {
        return 1;
    } else if (n % 2 == 0) {
        two5(n / 2) * two5(n / 2)
    } else {
        return 2 * two5(n - 1);
    }
}

public static int two6 (int n) {
    if (n == 0) {
        return 1;
    } else {
        return two6(n - 1) + two4(n - 1) // Note call to two4.
    }
}

```

Figure 1: Six Java class methods implementing a function that raises 2 to the  $n$ th power.