University of Washington

Department of Computer Science and Engineering

CSE 373

Problem Set 1 Solution

1. (1 point) Exercise 2.1.

   In ascending order, the functions are $2/n$, 37, $\sqrt{n}$, n, $n\log(\log n)$, $n\log n$, $\log(n^2)$, $n\log^2 n$, $n^{1.5}$, n^2, $n^2\log n$, n^3, $2^{n/2}$, 2^n.

   Note that $n\log n$ and $n\log(n^2)$ grow at the same rate, since $n\log(n^2) = 2n\log n$.

2. (3 points) Exercise 2.2, parts A, C, and D.

   Part A. True, by rule 1(a) in the book.

   Part C. False; let $T_1(n) = n^2$ and $T_2(n) = n$, so $f(n) = n^2$.

   Part D. False; same counter-example as in Part C.

3. (6 points) Exercise 2.6.

   (1) $O(n)$; (2) $O(n^2)$; (3) $O(n^3)$; (4) $O(n^2)$; (5) $O(n^5)$; (6) $O(n^4)$.

   Loops (1) through (4) are pretty straightforward.

   In Loop (5), note that the inner two loops are just the same as in (4) except that we have $i * i$ instead of n. Thus, these two loops are $O((i * i)^2) = O(i^4)$. Since the outer loop executes n times, the overall complexity is $O(n^5)$.

   In Loop (6), the tricky part is the `if` statement. Note that $j \bmod i = 0$ is true exactly when j is a multiple of i; i.e., when $j = i, 2i, 3i, \ldots$. Between 1 and i^2, this happens exactly i times. Hence, even though the `if` is executed i^2 times, the `for` is carried out only i of those times. Thus, the overall complexity is $O(n^4)$, not $O(n^5)$.

4. (2 points) Iterative Fibonacci sequence.

```
function fibonacci(n : int) : int
  var fminus1, fminus2 fib, i

begin
  fminus2 := 0;
  fminus1 := 1;

  if n = 0 then
    return 0;
```

```
  else if n = 1 then
    return 1;

  else begin
    for i := 2 to n begin
      fib := fminus1 + fminus2;
      (* Update previous two values *)
      fminus2 = fminus1;
      fminus1 = fib;
    end

    return fib;
  end
end
```

5. (4 points) FIND

   Yes, FIND is supported by linked lists. It is $O(n)$, where n is the length of the list. In the worst case, you have to traverse the entire list (if the item you want is at the very end).

   The advantage of arrays over linked lists is that with an array you have "random access" to any of the elements. Hence, assuming the array is sorted, the FIND can be done in $O(\log n)$ time using binary search, as opposed to $O(n)$ with linked lists. A similar speedup cannot be obtained with linked lists because each element of the list must be accessed through the previous element, and hence in order to get to the last element, we have to traverse the whole list.

   Note that it *is* possible to implement binary search with linked lists — but it is *more* inefficient than linear search!

6. (4 points) Binary vs. Sequential Search

   Larry's list does not change frequently. However, he calls his friends often. In terms of the functions he uses, this translates into lots of FINDs and not many INSERTs and DELETEs. He can therefore benefit from using binary search by keeping his information in a sorted array. He will get the benefit of the $O(\log n)$ search time but not suffer too much for the $O(n)$ INSERT and DELETE times that are expected with arrays.

   Sam, on the other hand, has a very dynamic list. However, he does not access it frequently. Thus, if he keeps his information in a linked list, he will benefit from the $O(1)$ INSERT operation. In the rare cases he does need a number, he can use sequential search. This will not be too inefficient in his case because it is very likely the person he is calling is near the beginning of his list, so he will rarely hit the worst-case of having to scan the entire list. Note that we need not worry too much about DELETE. We can either assume he never deletes entries (the older ones get pushed to the end of the list) or that he periodically just drops the part of the list that is beyond a certain point.