Washington State University
School of Electrical Engineering and Computer Science
Spring 2022

CptS 223 Advanced Data Structures in C++
Homework 1 - Solution
Due: January 19, 2022 (11:59pm pacific time)

General Instructions: Put your answers to the following problems into a PDF document and upload the document as your submission for Homework 1 for the course CptS 223 Pullman on the Canvas system by the above deadline.

1. For not shoveling the snow off your sidewalk, the city fined you \$2 for the first day. Each subsequent day, until you shovel the snow, the fine is squared (i.e., the fine progresses as follows: \$2, \$4, \$16, \$256, \$65,536, . . .).
   a. What would be the fine on day *N*? Show your work.
   b. How many days would it take for the fine to reach *D* dollars? Hint: Use floor or ceiling. Show your work.

   Solution:
   a. $F(N) = 2^{2^{(N-1)}}$. The sequence of fines follows the pattern:

Day	Fine
1	2
2	2^2
3	$(2^2)^2$
4	$((2^2)^2)^2$
N	$2^{2^{(N-1)}}$

   b. Solving for N in part (a):

$$F = 2^{2^{(N-1)}}$$
$$log_2 F = 2^{(N-1)}$$
$$log_2(log_2 F) = N - 1$$
$$N = 1 + log_2(log_2 F)$$

However, if we compute this for a fine that's not in the sequence, e.g., \$1000, then we get a fractional day, e.g., 4.32. In this example, the fine would not reach \$1000 until the next day 5. So, we need to take the ceiling of the log term:

$$N = 1 + \lceil log_2(log_2 F) \rceil$$

2. Use mathematical induction to prove your formula for problem 1a.

   Solution: Let F(N) be the fine on day N.

   Step 1: Base Case (N=1, Fine=2): F(1) = $2^{2^{(1-1)}}$ = 2. Thus, works for base case.

   Step 2: Assume F(N) = $2^{2^{(N-1)}}$ works for all N ≤ k.

   Step 3: Show formula works for (k+1) days. We know from the definition of the fine, that the fine F(k+1) on day (k+1) is the square of the fine F(k) on day k. From Step 2, we can assume our formula works for k days; therefore, the fine on day (k+1) is the square of $2^{2^{(k-1)}}$. Thus, $F(k+1) = (2^{2^{(k-1)}})^2 = 2^{2 \cdot 2^{(k-1)}} = 2^{2^k} = 2^{2^{((k+1)-1)}}$, which is our formula for N=k+1. Thus, the formula is proven for all N.

3. Write an efficient iterative (i.e., loop-based) algorithm *Fibonacci(n)* that returns the *nth* Fibonacci number. Your algorithm may only use a constant amount of memory (i.e., no auxiliary array). Argue that the running time *T(n)* of the algorithm is linear in *n*, i.e., $T(n) \leq cn$ for some constant *c*.

   Solution: The following pseudocode implements an efficient, iterative algorithm for computing Fibonacci(n). The algorithm executes a for-loop n times, and the operations performed in the loop are constant time; therefore, the running time of the algorithm is linear in n.

```
Fibonacci(n)
     if (n <= 1)
     then F = 1
     else
          F0 = 1
          F1 = 1
          for i = 2 to n
               F = F0 + F1
               F0 = F1
               F1 = F
     return F
```

4. Consider the function *IndexEqual(A,i,j)* that returns true if there exists an index x ($i \leq x \leq j$) such that A[x] = x; otherwise, returns false. You may assume *A* is a sorted integer array in which every element is unique.
   a. Write an efficient recursive algorithm for *IndexEqual(A,i,j)*.
   b. What is the situation resulting in the best-case running time of your function, and give an expression for that running time?
   c. What is the situation resulting in the worst-case running time of your function, and give an expression for that running time in terms of *n*, where *n=j-i+1*?

Solution:

a. Below is the pseudocode for IndexEqual(A,i,j). Note that the efficiency comes from being able to ignore half the array each time. An approach that traverses the entire array is not efficient.

```
IndexEqual(A,i,j)
     if (i <= j)
     then k = floor ((i+j)/2)
          if (A[k] == k)
          then return true
          else if (A[k] < k)
               then return IndexEqual(A,k+1,j)
               else return IndexEqual(A,i,k-1)
     else return false
```

b. The best case occurs when the index-equal element is at the mid-point of the array; that is, the algorithm will execute only the first four lines and return true. These are all constant-time operations, so the best-case running time is constant, i.e., T(n) = c.

c. The worst case occurs when no index-equal element exists, in which case the algorithm continues to call itself recursively on half of the array until (i > j), i.e., we have halved the array down to nothing. So, the running time T(N) consists of the running time on half of the array size T(N/2), and the rest of the processing is constant time. Thus, $T(N) = T(N/2) + c \cong \log_2 N$. You can also argue based on class discussion that the algorithm will terminate when we halve the array enough times to reduce it to one (or zero) elements, which is ($\log_2 N$) times.