

- 
1.  $\sum_{i=1}^n \log(i)^2 = \Omega(n \log(n))$  \_\_\_\_\_  True  False
- $\sum_{i=1}^n \log(i)^2 = O(n \log(n)^3)$  \_\_\_\_\_  True  False
- $2^{100} = O(1)$ ,  $\frac{1}{n} = O(1)$ ,  $2^n = O(3^n)$ , and  $\sum_{i=1}^n i^3 = \Theta(n^4)$  \_\_\_\_\_  True  False
- If  $\alpha = 1$ , then  $\alpha^0 + \alpha^1 + \alpha^2 + \dots + \alpha^n = \Omega(1)$  \_\_\_\_\_  True  False
- Suppose  $T(0) = 1$  and  $T(n) = n + T(n - 1)$  for  $n > 0$ .  
Then  $T(n) = \Omega(n^2)$ . \_\_\_\_\_  True  False
- Suppose  $T(0) = 1$  and  $T(n) = n^2 + T(n/2)$  for  $n > 0$ .  
Then  $T(n) = \Theta(n^2 \log n)$ . \_\_\_\_\_  True  False
- Suppose  $T(0) = 0$  and  $T(n) = n + 2T(n/2)$  for  $n > 0$ .  
Then  $T(n) = O(n)$ . \_\_\_\_\_  True  False
- Suppose  $T(0) = 1$  and  $T(n) = 1 + 3T(n/4)$  for  $n > 0$ .  
The recursion tree for  $T$  has depth  $\Theta(\log n)$ . \_\_\_\_\_  True  False
- Suppose  $T(0) = 1$  and  $T(n) = 1 + 2T(n - 2)$  for  $n > 0$ .  
The recursion tree for  $T$  has  $O(2^n)$  nodes. \_\_\_\_\_  True  False
- For any algorithm whose worst-case running time is  $O(n^2)$ , the algorithm must run in time  $\Omega(n^2)$  on all inputs. \_\_\_\_\_  True  False
- 

2. What is the big-O running time of the following subroutine, as a function of  $n$ ?

```
int mystery4(int n) {  
    int x = 0;  
    for (int i = 0; i < n * n * n ; ++i)  
        for (int j = i; j < n; ++j)  
            ++x;  
    return x;  
}
```

The running time is proportional to

$$\begin{aligned}\sum_{i=0}^{n^3-1} \max(1, n-i) &= \sum_{i=0}^n n-i + \sum_{i=n+1}^{n^3-1} 1 \\ &= \Theta(n^2) + \Theta(n^3) \\ &= \Theta(n^3)\end{aligned}$$

Give pseudo-code for an algorithm that runs in  $O(n)$  time and computes the same value as `mystery4(n)`.

```
int mystery4(int n) {
    int x = 0;
    for (int i = 0; i <= n ; ++i)
        x += n-i;
    return x;
}
```

- 
3. Here is a routine for computing Fibonacci numbers. It is the standard recursive routine, augmented to cache previously computed values, but *it only caches values for odd n*:

```
int fib(int n) {
    static map<int, int> cache;
    if (n <= 1) return n;
    if (n % 2 == 0) return fib(n-1) + fib(n-2);    // if n is even, recurse
    // if n is odd, use cache
    if (cache.find(n) == cache.end())              // if not in cache, recurse
        cache[n] = fib(n-1) + fib(n-2);
    return cache[n];
}
```

- (a) Draw a complete recursion tree (showing the structure of recursive calls) for `fib(9)`.  
(b) What is the big-O running time of this routine? Explain your answer.

Here's the tree for  $\text{fib}(15)$ ...

The running time is  $\Theta(n^2)$ .

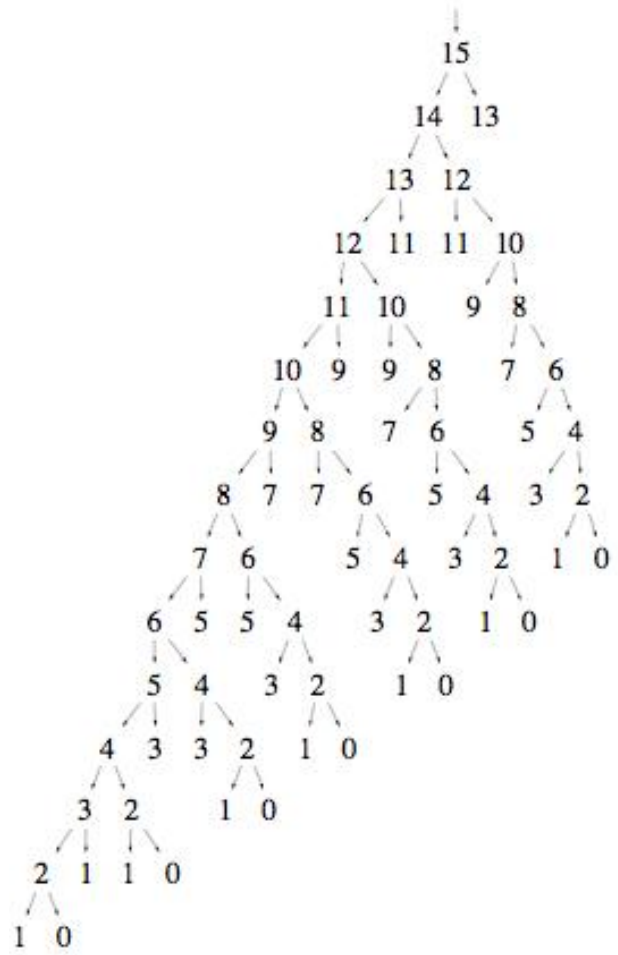
The left spine of the recursion tree has  $n$  nodes:  $n, n-1, n-2, \dots, 1$ . Each node on the left spine corresponds to the first time  $\text{fib}(i)$  is called, for  $i = n, n-1, n-2, \dots, 1$ .

Each of those calls to  $\text{fib}$  for odd  $i$  results in  $\text{fib}(i)$  being cached.

After that, subsequent calls to  $\text{fib}(i)$  return immediately if  $i$  is odd. But if  $i$  is even, then  $\text{fib}(i)$  calls  $\text{fib}(i-2)$ , which calls  $\text{fib}(i-4)$ , which calls  $\text{fib}(i-6)$ , and so on down to  $\text{fib}(0)$ .

So  $\text{fib}(i)$  for  $i$  even takes  $\Theta(i)$  time.

So total time is proportional to  $\Theta(n)$  for the left spine, then  $\sum_{i=2,4,6,\dots,n} \Theta(i) = \Theta(n^2)$  for the calls to  $\text{fib}(i)$  where  $i$  is even.



4. (a) Give a high-level but precise description of an efficient algorithm for the following problem:

Given a directed acyclic graph  $G$  and three vertices  $S, W, T$ , compute the number of paths from  $S$  to  $T$  that go through  $W$ .

A linear-time algorithm is possible.

- (b) What is the worst-case running time of your algorithm in terms of the number,  $m$ , of edges and the number,  $n$ , of vertices of  $G$ ?

- (c) Explain why your algorithm is correct.

(a)

1. Compute the number  $a$  of paths from  $S$  to  $W$ .
2. Compute the number  $b$  of paths from  $W$  to  $T$ .
3. Return the product  $ab$ .

To compute the number of paths from a node  $x$  to node  $y$ , use the algorithm described in class: topologically sort the vertices, then compute the number  $p[v]$  of paths from  $x$  to each node  $v$  using the recurrence  $p[v] = \sum_{w:(w,v) \in E} p[w]$ .

(b)  $O(n + m)$ . The latter algorithm to count paths takes  $O(n + m)$  time (topological sorting is done by DFS (linear time), then to compute the  $p[v]$  values takes time proportional to  $n + m$ ). Our algorithm just runs that one twice.

(c) The number of paths from  $S$  to  $T$  through  $W$  is the product  $ab$ , because for each path  $P_1$  from  $S$  to  $W$  and each path  $P_2$  from  $W$  to  $T$ , there is a path  $P = P_1, P_2$  from  $S$  to  $T$ . Conversely, every path from  $S$  to  $T$  through  $W$  has this form.

---

5. Longest common subsequence:

|   | s | u | p | e | r | c | a | l | a | f | r | a | g |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | 0 |   |   |   |   |   | 1 |   | . |   |   | . |   |
| m | 0 |   |   |   |   |   | 1 |   |   |   |   |   |   |
| a | 0 |   |   |   |   |   | 1 |   | 2 |   |   | . |   |
| n | 0 |   |   |   |   |   | 1 |   | 2 |   |   |   |   |
| a | 0 |   |   |   |   |   | 1 |   | 2 |   |   | 3 |   |
| p | 0 |   | 1 |   |   |   |   |   | 2 |   |   | 3 |   |
| l | 0 |   | 1 |   |   |   |   | 2 |   |   |   | 3 |   |
| a | 0 |   | 1 |   |   |   | 2 |   | 3 |   |   | . |   |
| n | 0 |   | 1 |   |   |   | 2 |   | 3 |   |   |   |   |

(a) Fill in the table above for the longest common subsequence algorithm. You need not fill in any cell whose value is the same as the value of the cell to its left. Note: dots denote pairs of common letters.

(b) Given sequences of length  $n$  and  $m$ , what is the big-O running time of this algorithm?

$O(nm)$

---

6. Recall the second algorithm we considered for greatest common divisor:

```

gcd2(i, j)
  if i == 1 or j == 1: return 1
  if i == j: return i
  if i < j: return gcd2(i, j-i)
  else: return gcd2(i-j, j)

```

Prove that the worst-case running time of this algorithm is  $\Omega(i)$ .

For any  $i$ ,  $\text{gcd2}(i, 2)$  takes  $\Omega(i)$  time: it calls  $\text{gcd2}(i - 2, 2)$ , which calls  $\text{gcd2}(i - 4, 2)$ , which calls  $\text{gcd2}(i - 6, 2)$ , and so on, down to  $\text{gcd2}(1, 2)$  or  $\text{gcd2}(2, 2)$ .

Thus, for any  $i$ , the worst-case running time of the algorithm is at least  $\Omega(i)$ .