COMP 271 Design and Analysis of Algorithms
2003 Spring Semester
Solutions to Question Bank Number 1 (Selected Problems)

Answer 1. The proof is by induction on n, the limit of the summation. For the basis case we consider the smallest legal value of n, namely 1. We have

$$\sum_{i=1}^{1} i(i-1) = 0 = \frac{1(1-1)(1+1)}{3},$$

as desired. For the induction step, we will assume that the formula holds for all the values $1, 2, \ldots, n-1$, then show that it holds for n. The standard method is to get rid of the last term of the sum, use the induction hypothesis to apply the formula to the the sum consisting of the first $n-1$ terms, and then add the last term back in again and simplify.

$$\begin{aligned}
\sum_{i=1}^{n} i(i-1) &= \left(\sum_{i=1}^{n-1} i(i-1)\right) + n(n-1) \\
&= \frac{(n-1)((n-1)-1)((n-1)+1)}{3} + n(n-1) \qquad \text{(by ind. hyp.)} \\
&= \frac{(n-1)(n-2)n}{3} + n(n-1) = \frac{(n-2)n(n-1) + 3n(n-1)}{3} \\
&= \frac{(n-2+3)n(n-1)}{3} = \frac{n(n-1)(n+1)}{3},
\end{aligned}$$

as desired.

Answer 2.

(a) True. Since $T_1(n) = O(f(n))$ and $T_2(n) = O(f(n))$, it follows from the definition that there exist constants $c_1, c_2 > 0$ and positive integers n_1, n_2 such that $T_1(n) \le c_1 f(n)$ for $n \ge n_1$ and $T_2(n) \le c_2 f(n)$ for $n \ge n_2$. This implies that, $T_1(n) + T_2(n) \le (c_1 + c_2) f(n)$ for $n \ge \max(n_1, n_2)$. Thus, $T_1(n) + T_2(n) = O(f(n))$.

(b) False. For a counterexample to the claim, let $T_1(n) = n^2, T_2(n) = n, f(n) = n^2$. Then $T_1(n) = O(f(n))$ and $T_2(n) = O(f(n))$ but $\frac{T_1(n)}{T_2(n)} = n \ne O(1)$.

(c) False. We can use the same counterexample as in part (b). Note that $T_1(n) \ne O(T_2(n))$

Answer 3.

	A	Relation:	B
(a)	$n^3 + n\log n$	Ω, Θ, O	$n^3 + n^2 \log n$
(b)	$\log\sqrt{n}$	Ω	$\sqrt{\log n}$
(c)	$n \log_3 n$	Ω, Θ, O	$n \log_4 n$
(d)	2^n	Ω	$2^{n/2}$
(e)	$\log(2^n)$	Ω, Θ, O	$\log(3^n)$

Notes:

(a) Both are $\Theta(n^3)$, the lower order terms can be ignored. Note that if $A(n) = \Theta(B(n))$, then automatically $A(n) = O(B(n))$ and $A(n) = \Omega(B(n))$.

(b) After simplifying, A is $(1/2)\lg n$, and B is $\sqrt{\log n}$. Substituting $m = \log n$, we can see ratio A/B grows as $m/2\sqrt{m} = \sqrt{m}/2$ which tends to infinity as n (and hence m) tends to infinity.

(c) Log base conversion only introduces a constant factor.

(d) The ratio is $2^n/2^{n/2} = (2)^{n/2}$ which goes to infinity in the limit.

(e) After simplifying these are $n \lg 2$ and $n \lg 3$, both of which are $\Theta(n)$.

Answer 4.

(a) $T(n) = O(n)$.

(b) $T(n) = O(\log n)$.

(c) $T(n) = O(n)$.

(d) $T(n) = O(n)$.

(e) $T(n) = O(n \log n)$.

(f) $T(n) = O(n^2)$.

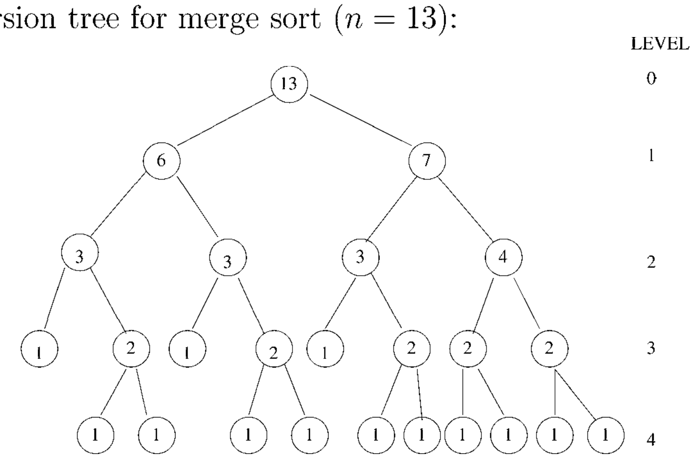
Answer 5.

The recurrence for the number of comparisons is:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1. \end{aligned}$$

(Note that if you use the following recurrence for the running time: $T(1) = 1; T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n$, you will obtain slightly different results.)

(a) Recursion tree for merge sort ($n = 13$):



(b) There are 5 levels in the recursion tree.

(c) Number of comparisons at levels 0, 1, 2 and 3 are 12, 11, 9 and 5, respectively.

(d) The total number of comparisons is 37.

(e) For general n, the number of levels is $1 + \log n$, the number of comparisons at each level is $O(n)$, and the total number of comparisons is $O(n \log n)$.

Answer 6. For any value of n, $\max(f(n), g(n))$ is either equal to $f(n)$ or equal to $g(n)$. Therefore, for all n,

$$\max(f(n), g(n)) \leq f(n) + g(n).$$

Using $c = 1$ and $n_0 = 1$ in the big-oh definition, it follows that

$$\max(f(n), g(n)) = O(f(n) + g(n)).$$

Also, for all n,

$$\max(f(n), g(n)) \geq f(n)$$

and

$$\max(f(n), g(n)) \geq g(n).$$

Adding we have

$$2 \times \max(f(n), g(n)) \geq f(n) + g(n).$$

Therefore,

$$\max(f(n), g(n)) \geq \frac{1}{2}(f(n) + g(n)).$$

Using $c = 1/2$ and $n_0 = 1$ in the Omega definition, it folows that

$$\max(f(n), g(n)) = \Omega(f(n) + g(n)).$$

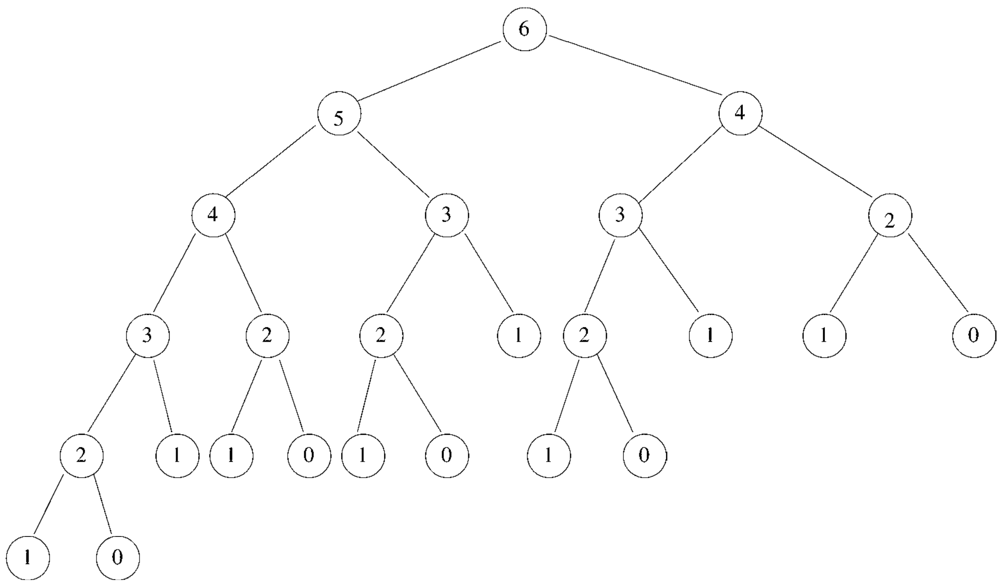
Since $\max(f(n), g(n)) = O(f(n) + g(n))$ and $\max(f(n), g(n)) = \Omega(f(n) + g(n))$, it implies that $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

Answer 8.

(a) It computes the Fibonacci numbers, which are defined by the following recurrence relation:

$$
\begin{aligned}
F(0) &= F(1) = 1 \\
F(n) &= F(n-1) + F(n-2) \qquad \text{if } n > 1
\end{aligned}
$$

(b)



The recursion tree is shown in the figure. It is easy to see that `unknown[i]` is executed once for $i = 5$, twice for $i = 4$, three times for $i = 3$, five times for $i = 2$, eight times for $i = 1$, and five times for $i = 0$.

(c) 12 additions are performed to compute `unknown(6)`.

(d) Let $T(n)$ denote the time taken to compute `unknown(n)`. Then the recurrence relation for $T(n)$ is:

$$
\begin{aligned}
T(0) &= T(1) = 1 \\
T(n) &= T(n-1) + T(n-2) + 1 \qquad \text{if } n > 1
\end{aligned}
$$

(e) We claim that $T(n) \geq c(1.5)^n$ for some constant c. Without knowing what c is, we proceed with the proof by induction. For the basis case, we need to check for both $n = 0$ and $n = 1$. Note that $T(0) = 1 \geq c \cdot (1.5)^0$, for $c \leq 1$, and $T(1) = 1 \geq c \cdot (1.5)^1$, for $c \leq 2/3$. So let us choose $c = 2/3$. For the induction step, we assume the induction hypothesis that for all $0 \leq k < n$, $T(k) \geq c(1.5)^k$, and then we show that the $T(n) \geq c(1.5)^n$. If we apply the definition of T and the induction hypothesis and simplify we get:

$$
\begin{aligned}
T(n) &= T(n-1)+T(n-2)+1 \geq \frac{2}{3}(1.5)^{n-1}+\frac{2}{3}(1.5)^{n-2}+1 \\
&\geq \frac{2}{3}(1.5)^{n-2}(1.5+1)+1 \\
&\geq \frac{2}{3}(1.5)^{n-2}(2.5)+1 \\
&\geq \frac{2}{3}(1.5)^{n-2}(1.5)^2+1 \\
&\geq \frac{2}{3}(1.5)^{n}+1 \\
&\geq \frac{2}{3}(1.5)^{n},
\end{aligned}
$$

which completes the induction proof. It follows that $T(n) = \Omega(1.5^n)$.

(f) Note that the recurrence given for $T(n)$ also applies to the number of additions. Hence the number of additions performed to compute `unknown(100)` $\geq (2/3)(1.5)^{100}$. Since the computer can perform a million additions each second, it takes $\geq (2/3)(1.5)^{100}/10^6$ seconds. This simplifies to $\geq (2.71)10^{11}$ seconds or more than 86 centuries!

(g)
```
float unknown(int n)
{
  F[0] = F[1] = 0;
  for i = 2 to n {
     F[n] = F[n-1] + F[n-2]
  }
  return(F[n]);
}
```

This program takes $O(n)$ time to compute `unknown(n)`. In the recursive program, the same values are computed *repeatedly* (see part(b)). But in the new program, we do not compute the same values again and again; instead each value $F[i]$ is computed *exactly once* and *stored* for future reference.