CMPSC 465 Data Structures & Algorithms
Fall 2021 Antonio Blanca and David Koslicki

HW 2

1. (20 pts.) Problem 1

(a) This is false. A counterexample is $f(n) = 2n$ and $g(n) = n$. $f(n)$ is $O(g(n))$ in this case because we can find constants $c = 3$ and $n_0 = 1$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. However, since $2^{f(n)} = 2^{2n}$ and $2^{g(n)} = 2^n$, $\lim_{n\to\infty} \frac{2^{2n}}{2^n} = \infty \neq 0$. So, $2^{f(n)}$ grows faster than $2^{g(n)}$ asymptotically. Thus, the statement is false.

(b) This is true. Proof: As $f(n)$ is $O(g(n))$, there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. Then, $0^2 \leq f(n)^2 \leq c^2 g(n)^2$ for all $n \geq n_0$. Let $d = c^2$. We have $0 \leq f(n)^2 \leq dg(n)^2$ for all $n \geq n_0$. So, we've found constants d and n_0 that satisfy the definition of Big-O notation. Thus, $f(n)^2$ is $O(g(n)^2)$.

(c) This is false. A counterexample is $f(n) = 2n$ and $g(n) = n$. $f(n)$ is $O(g(n))$ in this case because we can find constants $c = 3$ and $n_0 = 1$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. Let $h(n)$ be $0.5n$. $h(n)$ is $O(g(n))$ in this case because we can find constants $c = 3$ and $n_0 = 1$ such that $0 \leq h(n) \leq cg(n)$ for all $n \geq n_0$. However, since $2^{f(n)} = 2^{2n}$ and $2^{h(n)} = 2^{0.5n}$, $\lim_{n\to\infty} \frac{2^{2n}}{2^{0.5n}} = \infty \neq 0$. So, $2^{f(n)}$ grows asymptotically faster than $2^{h(n)}$, which is an example of $2^{O(g(n))}$. Thus, the statement is false.

(d) This is false. A counterexample is $f(n) = 2$ and $g(n) = 1$. $f(n)$ is $O(g(n))$ in this case because we can find constants $c = 3$ and $n_0 = 1$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. However, $\log_2 f(n) = 1$ and $\log_2 g(n) = 0$. As a result, no positive constants c and n_0 can satisfy that $0 \leq \log_2 f(n) \leq c \log_2 g(n)$ for all $n \geq n_0$ because $\log_2 f(n) > c \log_2 g(n) = 0$ no matter which value c takes. So, the statement is false.

(e) This is false. A counterexample is $f(n) = 0.5$ and $g(n) = 1$. $\log_2 f(n)$ is $O(\log_2 g(n))$ in this case because we can find constants $c = 1$ and $n_0 = 1$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. However, $\log_2 f(n) = -1$. As a result, $f(n)\log_2 f(n) = -0.5$, and $0 \leq f(n)\log_2 f(n)$ can never be satisfied. So, the statement is false.

2. (20 pts.) Problem 2

(a) We can observe that the turn i with the coefficient a_i has $\log_2 i$ multiplications. Overall, there are $\sum_{i=1}^{n} \log_2 i = \log_2(n!)$ multiplications, thus $O(n \log n)$ (from worksheet 1 problem 2-i, $\log(n!) = \Theta(n \log n)$). There are n additions, thus $O(n)$.

(b) If we consider the value of z *after* each iteration we obtain

$$\begin{array}{rcl}
i = n-1 & \rightarrow & z = a_n x_0 + a_{n-1} \\
i = n-2 & \rightarrow & z = a_n x_0^2 + a_{n-1} x_0 + a_{n-2} \\
i = n-3 & \rightarrow & z = a_n x_0^3 + a_{n-1} x_0^2 + a_{n-2} x_0 + a_{n-3} \\
\cdots & & \\
i = 0 & \rightarrow & z = a_n x_0^n + a_{n-1} x_0^{n-1} + \cdots + a_1 x_0 + a_0,
\end{array}$$

To describe in more detail, since the coefficients a_i are added to z in order from n to 0, the term with coefficient a_i multiplies with x_0 a total of i times. So, we have the desired polynomial $a_0 + a_1 x_0 + a_2 x_0^2 + \cdots + a_n x_0^n$ at the end.

(c) Every iteration of the for loop uses one multiplication and one addition, so the routine uses n additions and n multiplications.

3. (20 pts.) Problem 3

Let $M(n,m)$ be the time it takes to multiply a n bit number by a m bit number. The while loop in algorithm 2 takes $y-1$ iterations, and in each iteration we have to compute $z \cdot x$. Note that the multiplication of an n_1 bit number by an n_2 bit number results in a number with at most n_1+n_2 bits. Therefore, at i^{th} iteration of while loop, z has $O(in)$ number of bits (before multiplication), therefore the cost of multiplication at i^{th} iteration is $O(M(ni,n))$, and since there are $y-1$ iterations, the total running time would be:

$$\sum_{i=1}^{y-1} O(M(ni,n)) \tag{1}$$

(a) For this part, note that $M(ni,n) = O(n^2 i)$. As a result we have:

$$\sum_{i=1}^{y-1} O(M(ni,n)) = \sum_{i=1}^{y-1} O(in^2) = O(n^2)\sum_{i=1}^{y-1} i = O(n^2)\frac{(y-1)y}{2} = O(n^2y^2) \tag{2}$$

(b) For this part, note that $M(ni,n) = O(ni\log(ni))$, since $ni > n$. Therefore we have:

$$\begin{aligned}
\sum_{i=1}^{y-1} O(M(ni,n)) &= \sum_{i=1}^{y-1} O(ni\log(in)) && (3)\\
&= O(n)\sum_{i=1}^{y-1} i\log(in) && (4)\\
&= O(n\log n)\sum_{i=1}^{y-1} i + O(n)\sum_{i=1}^{y-1} i\log i && (5)\\
&= O(y^2 n\log n) + O(ny^2\log y) && (6)\\
&= O(n^2y^2) && (7)
\end{aligned}$$

Note: This problem turned out to be harder than we anticipated, so everyone will get the maximum score (10 pts).

4. (20 pts.) Problem 4

(a) For any 2×2 matrices X and Y:

$$XY = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix}\begin{pmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{pmatrix} = \begin{pmatrix} x_{11}y_{11}+x_{12}y_{21} & x_{11}y_{12}+x_{12}y_{22} \\ x_{21}y_{11}+x_{22}y_{21} & x_{21}y_{12}+x_{22}y_{22} \end{pmatrix}$$

This shows that every entry of XY is the addition of two products of the entries of the original matrices. Hence every entry can be computed in 2 multiplications and one addition. The whole matrix can be calculated in 8 multiplications and 4 additions.

(b) Let $A' = XA$, where A is an arbitrary 2×2 matrix, and $X = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Since entries of A' are only the sum of at most two entries of A, then we can say that number of bits of A' entries are at most one bit more than number of bits in A. Therefore, each time we multiply a matrix by X, the number of bits of each entry only increased by at most one bit, so we can conclude that X^i has as most i bits, which is $O(n)$, since $i < n$.

(c) In each call of `matrix`(X,n), we will go from n to $\frac{n}{2}$. So, it takes $\lfloor \log_2 n \rfloor$ recursive calls for the algorithm to end, and to return the output. Also, in each call, we have to do either $Z \cdot Z$ or $Z \cdot Z \cdot X$. Now, note that in i^{th} iteration or i^{th} recursive call we have $Z = X^{\frac{n}{2^i}}$, which means that entries of Z at i^{th} recursive call has at most $\frac{n}{2^i}$ bits (According to part b). Now note that for computing either $Z \cdot Z$ or $Z \cdot Z \cdot X$, we have constant number of multiplication, and additions between numbers with at most $\frac{n}{2^i}$ bits, which takes $O(M(\frac{n}{2^i})) < O(M(n))$ time. Therefore, the total run time can be written as follows:

$$\sum_{i=1}^{\lfloor \log_2 n \rfloor} O(M(\frac{n}{2^i})) = O(M(n) \log n) \tag{8}$$

5. (20 pts.) Problem 5

(a) The answer is at least 2^h and at most $2^{h+1} - 1$. This is because a complete binary tree of height $h-1$ has $\sum_{i=0}^{h-1} 2^i = 2^h - 1$ elements, and the number of elements in a heap of depth h is strictly larger than the number of vertices in a complete binary tree of height $h-1$ and less than (or equal) the number of nodes in a complete binary tree of height h.

(b) The array is not a Min heap. The node containing 26 is at position 9 of the array, so its parent is at position 4, which contains 35. This violates the Min Heap Property.

(c) Consider the min heap with n vertices where the root and every other node contains the number 2. Suppose now that 1 is inserted to the first available position at the lowest level of the heap. That is, $A[i] = 2$ for $0 \leq i \leq n-1$ and $A[n] = 1$. Since 1 is the minimum element of the heap, when Heapify-UP is called from position n, the node containing 1 must be swapped through each level of the heap until it is the new root node. Since the heap has height $\lfloor \log n \rfloor$, Heapify-UP has worst-case time $\Omega(\log n)$.

(d) Recall that heapsort works by first building a Min heap. It then uses the Min heap to produce a sorted array by repeatedly swapping the root with the last element and calling Heapify-Down from the root. If the array is already sorted in increasing order, then Build-Heap will call Heapify-Down $O(n)$ times but no swaps will occur. Hence, Build-Heap would take $O(n)$ time in this case. (Recall that in lectures we showed that Build-Heap would take $O(n \log n)$, but it was also mentioned that in fact Build-Heap takes $O(n)$ in the worst case.) However, heapsort will still take $O(n \log n)$. This is because each time we swap the root of the heap with the last element, we have to call Heapify-Down which will make $O(\log n)$ swaps.

If the array is sorted in decreasing order, it can also be similarly checked that mergesort will also take $\Theta(n \log n)$ time. Build-Heap takes $O(n)$, but Heapify-Down is called $O(n)$ times.

Rubric:

Problem 1, total points 20

(a) 4 points.
1 point: correct conclusion (statement is false)
1.5 points: a counterexample that makes sense
1.5 points: an explanation on why the provided counterexample shows the statement is false

(b) 4 points.
1 point: correct conclusion (statement is true)
3 points: a proof that reasons by making an association with the definition of big-O notation

(c) 4 points.
1 point: correct conclusion (statement is true)
3 points: a proof that reasons by making an association with the definition of big-O notation

(d) 4 points.
1 point: correct conclusion (statement is true)
1 point: make an association with the definition of big-O for $f(n)$ and $g(n)$ in proof
1 point: make an association with the definition of big-O for $g(n)$ and $O(g(n))$ in proof
1 point: how the proof reaches to the conclusion makes sense

(e) 4 points.
1 point: correct conclusion (statement is false)
1.5 points: a counterexample that makes sense
1.5 points: an explanation on why the provided counterexample shows the statement is false

Problem 2, total points 20

(a) 5 points.
1.5 points: correct answer for the number of sums
1.5 points: correct answer for the number of multiplications
2 points: the explanations make sense
Note: answers with exact numbers only or in big-O notations only are both acceptable, as long as the provided exact numbers or the big-O notations are correct.

(b) 10 points.
3 points: provide a proof
7 points: describe the pattern of the loop in the proof

(c) 5 points.
1.5 points: correct answer for the number of sums
1.5 points: correct answer for the number of multiplications
2 points: the explanations make sense
Note: answers with exact numbers only or in big-O notations only are both acceptable, as long as the provided exact numbers or the big-O notations are correct.

Problem 3, 20 pts

(a) part a is worth 10 pts. 6 pts for showing the runtime for each iteration of while loop (note that the i^{th} iteration of algorithm 2 takes $O(i \cdot n^2)$), and 4 pts for showing the total runtime of algorithm (the total runtime is $O(n^2y^2)$).

(b) part b is worth 10 pts. Assign 10 pts to every student.

Problem 4, 20 pts

(a) part a is worth 3 pts. Computing the multiplication of two matrices correctly is worth 2 pts, even if the final answer is not correct.

(b) part b is worth 5 pts.

(c) part c is worth 12 pts. 3 pts, for showing that there are $O(\log_2 n)$ recursive call, and 9 pts for showing the runtime for each recursive call which is $O(M(\frac{n}{2^i}))$. However, $O(M(n))$ is also accepted for the runtime of each recursive call.

Problem 5, 20 pts.

(a) 5 points for this part, if at least 2^h is pointed, 1.5 pts, if at most $2^{h+1}-1$ is pointed, 1.5 pts, reasonable explanation 2 pts

(b) answer as No, 2.5 pts, denote that element 26 is in the wrong place, and it should be swapped with 35, 2.5 pts

(c) any clear and reasonable explanation should be 5 pts

(d) increasing order, running time is $O(n \log n)$, 1 pts, reasonable explanation 1.5 pts, decreasing order, running time is $O(n \log n)$, 1 pts, reasonable explanation 1.5 pts