Practice Quiz 1 Solutions

- Do not open this quiz booklet until you are directed to do so.
- This quiz starts at 2:35 P.M. and ends at 3:55 P.M. It contains **3** problems, some with multiple parts. The quiz contains 13 pages, including this one. You have 80 minutes to earn 100 points.
- This quiz is closed book. You may use one handwritten $8\frac{1}{2}'' \times 11''$ crib sheet. No calculators are permitted.
- When the quiz begins, write your name on every page of this quiz booklet in the space provided.
- Write your solutions in the space provided. If you need more space, write on the back of the sheet containing the problem. Do not put part of the answer to one problem on the back of the sheet for another problem, since the pages will be separated for grading.
- Do not spend too much time on any problem. Read them all through first and attack them in the order that allows you to make the most progress.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- Good luck!

Problem 1. Integer Multiplication [15 points]

The following algorithm multiplies two nonnegative integers A and B:

```
MULT(A, B)
P ← 0
while A ≠ 0
     do if A mod 2 = 1
          then P ← P + B
        A ← ⌊A/2⌋
        B ← 2B
return P
```

Let $A^{(0)}$, $B^{(0)}$, and $P^{(0)}$ be the values of A, B, and P, respectively, immediately before the loop executes, and for $k \geq 1$, let $A^{(k)}$, $B^{(k)}$, and $P^{(k)}$ be the values of these variables immediately after the kth iteration of the loop. Give a loop invariant that can be used to prove the correctness of the algorithm. *You need **not** actually prove correctness.*

Answer:

$$A^{(0)}B^{(0)} = A^{(k)}B^{(k)} + P^{(k)}.$$

Notes: Almost everyone had an idea of what an invariant should look like. About one-third of the students got the invariant right. Many folks gave a recursive invariant that did not involve the product that the algorithm is trying to compute. This invariant can not be used directly together with the negation of loop-test to establish the post-condition.

Some people believed that the invariant should be true at every point inside the loop. They did not understand that the invariant must be true at the beginning of every iteration of the loop. That is, assuming it was true at the beginning of iteration i, the body of the loop re-establishes the invariant, so that it is true at the beginning of iteration $i + 1$.

Problem 2. True or False, and Justify [50 points] (10 parts)

Circle **T** or **F** for each of the following statements to indicate whether the statement is true or false, respectively. If the statement is correct, briefly state why. If the statement is wrong, explain why. The more content you provide in your justification, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

T F The solution to the recurrence

$$T(n) = 100\,T(n/99) + \lg(n!)$$

is $T(n) = \Theta(n \lg n)$.

False. Observe that $\lg(n!) = \Theta(n \lg n) = O(n^{\lg_{99} 100 - 0.002})$. Therefore, by Master Theorem case 1, the solution is $\Theta(n^{\lg_{99} 100})$.

Notes: Many students wrongly thought this was case 2. Some even identified this as case 3. Among those who correctly spotted this as being case 1, the majority were not aware that, in order to prove that $\delta > 1 + \epsilon$ for some $\epsilon > 0$, it is *not* sufficient to show that $\delta > 1$ (no points were deducted for this oversight, however).

T F Radix sort works correctly even if insertion sort is used as its auxiliary sort instead of counting sort.

True. The correctness of radix sort requires the auxiliary sort to be stable. Insertion sort as presented in this course is stable.

Notes: Most students identified correctly that we needed a stable sort, however a significant portion visualized an unstable version of insertion sort and thus gave the wrong answer.

T F If bucket sort is implemented by using heapsort to sort the individual buckets, instead of by using insertion sort as in the normal algorithm, then the worst-case running time of bucket sort is reduced to $\Theta(n \lg n)$.

True. Bucket sort was presented in Recitation 4. Given an input of n numbers from a specified range, bucket sort divides the range into n intervals and associates one bucket with each interval. It distributes the items into the buckets, which takes $\Theta(n)$ time. It then sorts the items in each bucket using an auxiliary sort. It finally concatenates the sorted bucket lists together in $\Theta(n)$ time.

Let b_i be the number of items that fall into bucket i, for $i = 1, \ldots, n$. Let $f(n)$ be the running time of the auxiliary sort used to sort the items in each bucket. The total running time $T(n)$ of bucket sort is then

$$T(n) = \Theta(n) + \Theta\left(\sum_{i=1}^{n} f(b_i)\right).$$

We argued in recitation that the worst case is when all of the items fall into one bucket. In this case $T(n) = \Theta(n) + \Theta(f(n))$. If insertion sort is used as the auxiliary sort, then $T(n) = \Theta(n^2)$. If heap sort is used, then $T(n) = \Theta(n \lg n)$.

Notes: Many people did not give the full running time formula for $T(n)$ or did not argue that $T(n)$ is dominated by the time to sort the items in the buckets.

Some folks argued that bucket sort runs in expected $\Theta(n)$ time (assuming a uniform distribution over the inputs), which is true and irrelevant.

T F An adversary can present an input of n distinct numbers to RANDOMIZED-SELECT that will force it to run in $\Omega(n^2)$ time.

False. RANDOMIZED-SELECT was the first selection algorithm presented in Lecture 7. It is a randomized algorithm. The running time of a randomized algorithm is a random variable whose value for a particular run of the algorithm is determined by the random numbers the algorithm uses for that run. The running time of RANDOMIZED-SELECT does not does not depend its input (if all numbers are distinct). No adversary has control over which random numbers the algorithm will use, and no adversary can determine which random numbers the algorithm will use. Therefore, although it is true that RANDOMIZED-SELECT runs in $\Theta(n^2)$ time in the worst case, no adversary can force this behavior.

Notes: Some people thought that RANDOMIZED-SELECT was a randomized version of the deterministic SELECT algorithm (the second algorithm presented in Lecture 7). We guess that this was unfortunate inference from the pattern seemingly set by RANDOMIZED-PARTITION and RANDOMIZED-QUICKSORT.

T F The information-theoretic (decision-tree) lower bound on comparison sorting can be used to prove that the number of comparisons needed to build a heap of n elements is $\Omega(n \lg n)$ in the worst case.

False. The procedure BUILD-HEAP runs in $\Theta(n)$ time. BUILD-HEAP was presented in Lecture 5, is given in CLR, and was part of Problem 3 of Problem Set 2.

An acceptable justification is that building a heap of n elements does not sort them, so the sorting lower bound does not apply.

Notes: Some people felt compelled to empty the heap after building it. That is, they argued that HEAPSORT requires $\Omega(n \lg n)$ comparisons, which is true and irrelevant.

Others argued that the information-theoretic lower bound does not apply to the worst case running time of a sorting algorithm, which is false and irrelevant.

Still others pointed out that the information-theoretic lower bound says that asymptotically *at least* $n \lg n$ comparisons are required; it does not say how many comparisons are actually used by a particular algorithm on a worst case input. This is quite right and quite irrelevant. We can only guess that these people misread $\Omega(n \lg n)$ as $O(n \lg n)$.

T F Sorting 6 elements with a comparison sort requires at least 10 comparisons in the worst case.

True. As shown in Lecture 6, the number of leaves of a decision tree which sorts 6 elements is $6!$ and the height of the tree is at least $\lg(6!)$. Since $6! = 720$ and $2^9 = 512$ and $2^{10} = 1024$, we have $9 < \lg(6!) < 10$. Thus at least 10 comparisons are needed.

Notes: This was identical to the problem given on the practice quiz except that the particular numbers were changed.

Some folks had difficulty computing powers of 2 correctly. Starting today make sure you have the first 11 powers of 2 memorized: $0{:}1,\ 1{:}2,\ 2{:}4,\ 3{:}8,\ 4{:}16,\ 5{:}32,$
$6{:}64,\ 7{:}128,\ 8{:}256,\ 9{:}512,\ 10{:}1024.$

Some people calculated a lower bound on the height of the decision tree using the function $n \lg n$. This is not a good strategy since this function is only asymptotically equivalent to $\lg(n!)$. Most discovered that $6 \lg 6$ is at least 12, some going as far as 18, or as stated by more than one person, "6×2.5ish = 18ish" (in fact $\lg 6 \approx 2.58$). Of these, many concluded that indeed at least 10 comparisons are needed because at least 12 comparisons are needed. However, the rest of the folks asserted that it is not true that at least 10 comparisons, precisely because at least 12 are needed. These folks were gently reminded that $10 < 12$.

T F The sum of the smallest $\sqrt{n}$ elements in an unsorted array of n distinct numbers can be found in $O(n)$ time.

True.

1. Find the $\sqrt{n}$*th* smallest element, r, using the SELECT algorithm.
2. Partition the array around r.
3. Sum the first $\sqrt{n}$ elements of the array.

Step 1 takes $\Theta(n)$ time, Step 2 takes $\Theta(n)$ time, and Step 3 takes $\Theta(\sqrt{n})$ time. Hence this gives a $\Theta(n)$ algorithm for summing up the smallest $\sqrt{n}$ elements in the given array.

Notes: People either nailed this question or didn't. Some used RADIX-SORT to sort the elements in $\Theta(n)$ time, which can't be done since we don't know the range of the numbers.

Others tried to prove that it couldn't be done by giving an algorithm that didn't work. This only proves that *they* couldn't do it. A favorite was to find each of the $\sqrt{n}$ smallest elements in $\Theta(n)$ time (via SELECT or a linear scan through the array), for a total running time of $\Theta(n^{3/2})$.

T F The collection $\mathcal{H} = \{h_1, h_2, h_3\}$ of hash functions is universal, where the three hash functions map the universe $\{A, B, C, D\}$ of keys into the range $\{0, 1, 2\}$ according to the following table:

x	$h_1(x)$	$h_2(x)$	$h_3(x)$
A	1	0	2
B	0	1	2
C	0	0	0
D	1	1	0

True. A hash family $\mathcal{H}$ that maps a universe of keys U into m slots is *universal* if for each pair of distinct keys $x, y \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(x) = h(y)$ is exactly $|\mathcal{H}|/m$. In this problem, $|\mathcal{H}| = 3$ and $m = 3$. Therefore, for any pair of the four distinct keys, exactly 1 hash function should make them collide. By consulting the table above, we have:

$$\begin{array}{lll} h(A) = h(B) & \text{only for } h_3 & \text{mapping into slot } 2 \\ h(A) = h(C) & \text{only for } h_2 & \text{mapping into slot } 0 \\ h(A) = h(D) & \text{only for } h_1 & \text{mapping into slot } 1 \\ h(B) = h(C) & \text{only for } h_1 & \text{mapping into slot } 0 \\ h(B) = h(D) & \text{only for } h_2 & \text{mapping into slot } 1 \\ h(C) = h(D) & \text{only for } h_3 & \text{mapping into slot } 0 \end{array}$$

Notes: Many folks misinterpreted the definition of a universal family of hash functions. They thought that for every $h \in \mathcal{H}$, the probability that $h(x) = h(y)$ for x and y chosen uniformly at random is $|\mathcal{H}|/m$.

T F Let X be an indicator random variable such that $\Pr\{X = 1\} = 1/2$. Then, we have $\mathrm{E}[X(1-X)] = 1/4$.

False. $\mathrm{E}[X(1-X)] = 0$, which can be seen in many ways:

- $X(1-X) = 0$ if $X = 1$ or $X = 0$.
- $\mathrm{E}[X(1-X)] = \mathrm{E}[X - X^2] = \mathrm{E}[X] - \mathrm{E}[X^2] = \mathrm{E}[X] - \mathrm{E}[X] = 0$.
- $\mathrm{E}[X(1-X)] = \sum_x x(1-x)\Pr\{X = x\} = 0(1-0)\Pr\{X = 0\} + 1(1-1)\Pr\{X = 1\} = 0$.

Notes: Many people mistakenly asserted the indepedence of non-independent variables. This took the form of

$$\mathrm{E}[X(1-X)] = \mathrm{E}[X]\,\mathrm{E}[1-X] = (1/2)(1/2) = 1/4$$

which is false because X and $1 - X$ are not independent, or

$$\mathrm{E}\left[X - X^2\right] = \mathrm{E}[X] - \mathrm{E}[X]\,\mathrm{E}[X] = 1/2 - 1/4 = 1/4$$

which is false because X and X are not independent.

Other folks did not remember rules of expectation, saying such things as, "$\mathrm{E}[X(1-X)] = \mathrm{E}[X] \cdot \mathrm{E}[1-X]$ by linearity of expectation." Many did not do sanity checks on their answers, which is particularly easy for indicator random variables, which only take on two values. Those who did noticed that $X(1-X) = 0$ for both $X = 0$ and $X = 1$.

T F Suppose that a 3-input sorting network correctly sorts the sequences $\langle 2, 3, 8\rangle$, $\langle 3, 8, 2\rangle$, and $\langle 8, 2, 3\rangle$. Then, it also correctly sorts all sequences of 3 numbers. (*Hint: Apply threshold functions to the sequence elements.*)

True. If we apply a threshold function which flips from 0 to 1 at $x = +\infty, 8,$ and 3 respectively, we will get four $0-1$ sequences for each of the correctly sorted sequences. Then, if all the possible $0-1$ length 3 sequences are accounted for, we can say that the sorting network will correctly sort *any* sequence of 3 numbers, by the $0-1$ principle. The following table shows the sequences obtained by applying the threshold function at each of the above-mentioned points to each of the three sequences:

original sequence	:	2 3 8	3 8 2	8 2 3
threshold at $+\infty$	:	0 0 0	0 0 0	0 0 0
threshold at 8	:	0 0 1	0 1 0	1 0 0
threshold at 3	:	0 1 1	1 1 0	1 0 1
threshold at 2	:	1 1 1	1 1 1	1 1 1

All possible sequences of $0, 1$ appear, therefore we conclude that the sorting network correctly sorts any three numbers.

Note that any comparator network correctly sorts the sequence composed of all zeros and that composed of all ones. Therefore, we could have avoided the first and last thresholds.

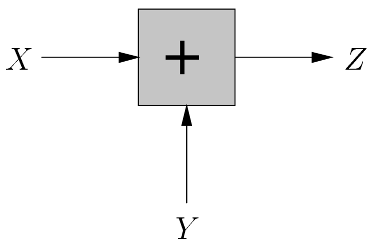
Notes: People did much better on this question than expected. Those who knew the 0-1 Principle generally got it right, even though the problem required some nontrivial thinking. Those who got it wrong did not seem to know the 0-1 Principle. Only a few seemed to know the 0-1 Principle yet were unable to figure out the solution.

Problem 3. Pop Count [35 points] (5 parts) Some computers provide a ***population-count***, or ***pop-count***, instruction POPC that determines the total number of bits in a word that are "1." Specifically, if an n-bit word is $a = \langle a_0 a_1 \cdots a_{n-1} \rangle$, then

$$\text{POPC}(a) = \sum_{i=0}^{n-1} a_i \ .$$

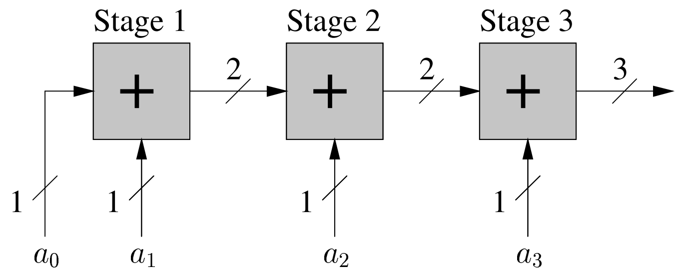
For example, $\text{POPC}(\langle 01000101 \rangle) = 3$.

This problem explores how to implement the POPC instruction as a circuit. The basic building block we shall use is an ADDER component that takes in two binary values X and Y and produces their sum $Z = X + Y$:



Since wiring is an important contributor to the expense of an implementation, we shall attempt to minimize the number of wires required to connect the inputs, components, and output of a pop-count circuit. A cable of w wires can convey values in the range from 0 to $2^w - 1$, inclusive.

Professor Blaise has invented a pop-count circuit, composed of $n - 1$ ADDER's, to implement POPC on an n-bit word $\langle a_0 a_1 \cdots a_{n-1} \rangle$. Here is the professor's circuit for $n = 4$:



Each stage i of this circuit adds the bit a_i into a running sum, which it forwards to stage $i+1$. Each cable of wires in the figure is labeled with the number of its constituent wires.

(a) Argue that the stage-i ADDER requires $\Theta(\lg i)$ output wires.

Answer: The output of the stage-i adder must carry the addition of $i + 1$ bits, which is a value of at most $i + 1$ since all the bits in the sequence $\langle a_0, a_1, \cdots, a_i \rangle$ may be 1. As was stated in the problem, a cable of w wires can carry numbers up to $2^w - 1$. We therefore have the following equality and need to solve for w:

$$\begin{aligned} i + 1 &= 2^w - 1 \\ i + 2 &= 2^w \\ \lg(i + 2) &= w \end{aligned}$$

Since $\lg(i + 2)$ describes the number of wires needed to carry at most the value $i + 1$, we must take the ceiling of it. Therefore,

$$w = \lceil \lg(i + 2) \rceil = \Theta(\lg i)$$

wires are needed to carry the output of the stage-i adder.

Notes: Most folks correctly argued the $\Theta(\lg i)$ bound. Very few calculated the exact number of wires required.

(b) For an input word with $n = 4$ bits, the total number of wires in the professor's pop-count circuit is $W(n) = 11$ wires. Explain briefly why the recurrence

$$W(n) = W(n-1) + \Theta(\lg n)$$

accurately describes the total number of wires in an n-input circuit. Give a good asymptotic lower (big-Ω) bound for $W(n)$, and briefly justify your answer.

Answer: Assume that, in order to add the first $n-1$ bits ($\langle a_0, a_1, \cdots, a_{n-2}\rangle$), the circuit uses $W(n-1)$ wires total in its $n-2$ stages. Then, in order to add in the last bit, we add one adder to the circuit, whose one input is the output of the last adder of the $n-1$ circuit. The other input to the new stage-$n-1$ adder is the last bit a_{n-1}. From part **(a)** we know that this new adder will require $\Theta(\lg(n-1)) = \Theta(\lg n)$ output wires. The recurrence that describes the total number of wires then becomes

$$\begin{aligned}
W(n) &= W(n-1) + \Theta(\lg n) + 1 \\
&= W(n-1) + \Theta(\lg(n) + 1) \\
&= W(n-1) + \Theta(\lg n).
\end{aligned}$$

The solution to this recurrence is obtained via *iteration*. Iteration gives us a Θ bound which in turn gives us an Ω bound.

$$\begin{aligned}
W(n) &= W(n-1) + \Theta(\lg n) \\
&= W(n-1) + \Theta(\lg(n-1)) + \Theta(\lg(n)) \\
&= W(n-2) + \Theta(\lg(n-2)) + \Theta(\lg(n-1)) + \Theta(\lg(n)) \\
&\ldots \\
&= W(1) + \Theta(\lg(2)) + \Theta(\lg(n-2)) + \Theta(\lg(n-1)) + \Theta(\lg(n)) \\
&= 1 + \Theta(\lg(2) + .. + \lg(n-2) + \lg(n-1) + \lg(n)) \\
&= 1 + \Theta(\lg((n)(n-1)(n-2)..(2)(1))) \\
&= 1 + \Theta(\lg(n!)) \\
&= 1 + \Theta(n \lg n) \\
&= \Theta(n \lg n)
\end{aligned}$$

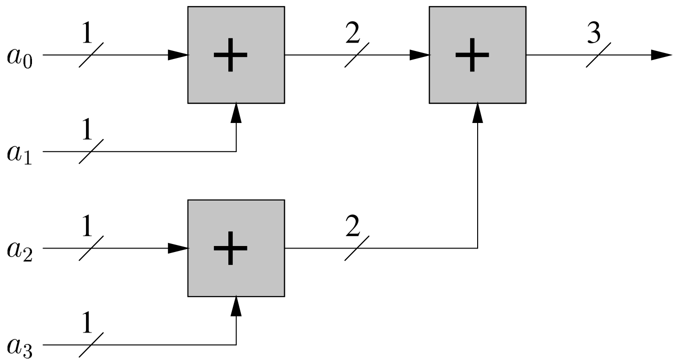
We therefore conclude that $W(n) = \Omega(n \lg n)$.

Notes: A common error was to use a recursion tree, saying that each level of the tree is $\Theta(\lg n)$. This is wrong. The work in every level is *not* $\Theta(\lg n)$; it is $O(\lg n)$. Having started off on the wrong foot, the erroneous proof states that, since there are n levels, the solution is $W(n) = O(n \lg n)$ which *does not* mean that $W(n) = \Omega(n \lg n)$.

A similar common mistake was to iterate the recurrence and state that each one of the logs is *at most* (meaning $O(\cdot)$) $\lg n$, but instead write down $\Theta(\lg n)$.

A divide-and-conquer pop-count circuit operating on an n-bit word $\langle a_0 a_1 \cdots a_{n-1} \rangle$ can be constructed by using an ADDER component to combine the recursively computed pop-count of the first $n/2$ bits with the recursively computed pop-count of the last $n/2$ bits.

(c) Draw a picture of such a divide-and-conquer pop-count circuit on 4 inputs. Label the number of wires comprising each cable, as was done for Professor Blaise's circuit. How many wires does the divide-and-conquer circuit require for $n = 4$?



Answer: This circuit has $1 + 1 + 1 + 1 + 2 + 2 + 3 = 11$ **(eleven)** wires.

Notes: Many folks drew exactly the circuit above, correctly labelled all the wires, and then miscounted the number of wires. Counts ranged from 7 to 13.

Some people drew the circuit recursively or else tried to pipeline the circuit. Credit was given if the solution could be deciphered.

(d) Give a recurrence that describes the total number $W(n)$ of wires required by the divide-and-conquer pop-count circuit for an n-bit word. Give a good asymptotic upper (big-O) bound on $W(n)$, and briefly justify your answer.

Answer: The last stage of the divide-and-conquer pop-count circuit adds the output of two sized $n/2$ divide-and-conquer pop-count adders. That sum requires $\Theta(\lg n)$ wires to carry it (see part **(a)**). Therefore, the recurrence for the number of wires is

$$W(n) = 2W(n/2) + \Theta(\lg n).$$

The Master theorem case 1 gives the solution to this recurrence where

$$f(n) = \lg n = O(n^{\log_b a - \epsilon}) = O(n^{\log_2 2 - \epsilon}) = O(n)$$

for (say) $\epsilon = 1/2$. Thus,

$$W(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

Finally, we conclude our upper bound. $W(n) = \Theta(n) \Rightarrow W(n) = O(n)$.

(e) Does Professor Blaise work at MIT or Harvard? Why?

Answer: It's not clear where Blaise works, but it is clear that he doesn't know the material from 6.046. His iterative adder uses asymptotically more wires than the divide-and-conquer adder. In addition, his adder requires n stages to add while the divide-and-conquer adder has depth $\lg n$. Clearly, one of his graduate students saw the professor's initial design, suggested the improvement, and submitted the problem for use on the 6.046 quiz. [Answers along these lines were awarded one extra credit point.]