CS / MCS 401 Homework 3 grader solutions

assignment due July 6, 2016
written by Jānis Lazovskis
maximum points: 33

Some questions from CLRS. Questions marked with an asterisk $*$ were not graded.

1 *Use the definition of a binomial random variable X and the definition of the expected value $E[X] = \sum_{all\ x} x \cdot P(x)$ to prove $E[X] = np$. Do not use indicator random variables.*

Let X be a binomial random variable with probability p of success, and suppose that X is evaluated n times. Recall the probability of k successes over n trials for X is $\binom{n}{k}p^k(1-p)^{n-k}$. The expected value of X is then

$$\begin{aligned}
E[X] &= \sum_{\text{all } x} x \cdot P(x) && \text{(definition)} \\
&= \sum_{k=0}^{n} k \cdot P(k \text{ successes}) && \text{(since } n \text{ trials, renaming } x \text{ to } k\text{)} \\
&= \sum_{k=0}^{n} k \binom{n}{k} p^k (1-p)^{n-k} && \text{(definition)} \\
&= \sum_{k=1}^{n} k \cdot \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k} && \text{(definition, first term is 0)} \\
&= \sum_{k=1}^{n} np \cdot \frac{(n-1)!}{(k-1)!(n-k)!} p^{k-1} (1-p)^{n-k} && \text{(reducing and factoring)} \\
&= np \sum_{k=1}^{n} \binom{n-1}{k-1} p^{k-1} (1-p)^{n-k} && \text{(rearranging)} \\
&= np \sum_{k=0}^{n-1} \binom{n-1}{k} p^{k} (1-p)^{(n-1)-k} && \text{(reindexing)} \\
&= np \cdot (p + (1-p))^{n-1} && \text{(binomial theorem)} \\
&= np. && \text{(simplifying)}
\end{aligned}$$

■

2 *Given X is a binomial random variable, use indicator random vairables to prove $E[X] = np$.*

Let X_k, for $k = 0, 1, \ldots, n$ be the indicator random variable evaluating to 1 if there are k successes, and evaluating to 0 otherwise. By Lemma 5.1 (on page 118 of CLRS), we have that $E[X_k] = p$, so then

$$E[X] = E\left[\sum_{k=0}^{n} X_k\right] = \sum_{k=0}^{n} E[X_k] = \sum_{k=0}^{n} p = np.$$

■

5.2-3 *(p.122)* *Use indicator random variables to compute the expected value of the sum of n dice.*

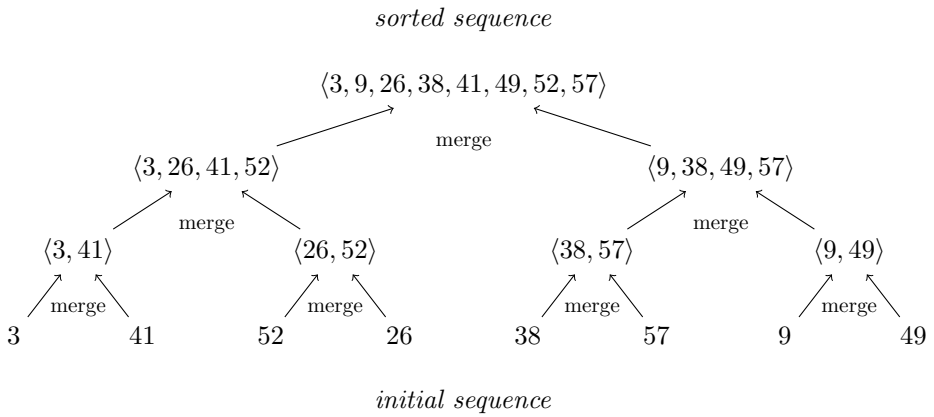
For each die, let X_k, for $k = 1, 2, 3, 4, 5, 6$ be the indicator random variable evaluating to 1 if the number k is rolled, and evaluating to 0 otherwise. Let X be the random variable evaluating to the number rolled on the die. It is clear that $E[X_k] = 1/6$ for all k, assuming the dice are fair. For every roll, the expected value is

$$E[X] = 1 \cdot E[X_1] + \cdots + 6 \cdot E[X_6] = \frac{1 + \cdots + 6}{6} = \frac{21}{6} = \frac{7}{2}.$$

Hence the expected value for n rolls is $E[nX] = nE[X] = n\frac{7}{2}$. ■

2.3-1 *(p.37)* *Using Figure 2.4 as a model, illustrate the operation of merge sort on the array* $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

We follow the diagram in Figure 2.4 with the given sequence.



■

2.3-2 *(p.37)* *Rewrite the* MERGE *procedure so that it does not use sentinels, instead stopping once either array* L *or* R *has had all its elements copied back to* A *and then copying the remainder of the other array back into* A.

The following is one version of such a rewrite. The main difference is the removal of lines 8 and 9, and the adjustment of the **for** loop on line 12.

```
1   n_1 = q − p + 1
2   n_2 = r − q
3   let L[1, ..., n_1] and R[1, ..., n_2] be new arrays
4   for i = 1 to n_1
5       L[i] = A[p + i − 1]
6   for j = 1 to n_2
7       R[j] = A[q + j]
8   i = 1
9   j = 1
10  k = p
11  while i ≠ n_1 + 1 and j ≠ n_2 + 1
12      if L[i] ⩽ R[j]
13          A[k] = L[i]
14          i = i + 1
15      else
16          A[k] = R[j]
17          j = j + 1
18      k = k + 1
19  if i = n_1 + 1 and j ≠ n_1 + 1
20      A[k, ..., r] = R[j, ..., n_2]
21  else if j = n_2 + 1 and i ≠ n_2 + 1
22      A[k, ..., r] = L[i, ..., n_1]
```

Note that L and R may be the same length, so to not raise an error, an **else if** command has to be used instead of **else**. The difference is that **else** following an **if** is evaluated if the **if** fails, whereas to condition of **else if** following an **if** is checked before evaluated. We could have used two **if** commands, but the second one will be checked even if the first one evaluates to true, increasing running time. ■

2.3-3 *(p.39)* *Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence*

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n \log_2(n)$.

In the base case $n = 2$ we see that $T(2) = 2$ by what is given, and $T(2) = 2 \cdot \log_2(2) = 2$ by what is expected. Hence the base case holds. For the inductive case, when $n = 2^k$ for $k \geqslant 2$, we have that

$$T(2^k) = 2T(2^{k-1}) + 2^k = 2 \cdot 2^{k-1} \cdot \log_2(2^{k-1}) + 2^k = 2^k(k-1) + 2^k = k2^k = n \log_2(n),$$

as desired. ■

2.3-6 *(p.39)* *Observe that the **while** loop of lines 5-7 of the* INSERTION-SORT *procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1 \,..\, j-1]$. Can we use a binary search (see Exercise 2.3-5) instead to improve the overall worst-case running time of insertion sort to $\Theta(n \log_2(n))$?*

No, because even if we use binary search, the search must be done for every element in the list, so the number of times line 5 is executed is still $\sum_{j=2}^{n} s_j$. While it may be true that $s_j \leqslant t_j$, the sum is still up to n, so we will get a quadratic factor of n^2 in the running time calculation. ■

2.4 a,b,c *(p.41)* *Let $A[1 \,..\, n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an **inversion** of A.*

a. *List the five inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$.*

b. *What array with elements from the set $\{1, 2, \ldots, n\}$ has the most inversions? How many does it have?*

c. *What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.*

a. The five inversions of A are $(1, 5)$, $(2, 5)$, $(3, 4)$, $(3, 5)$, and $(4, 5)$.

b. The reverse ordered array $\langle n, n-1, \ldots, 2, 1 \rangle$ has the most inversions, since every pair (i, j) with $i < j$ is an inversion. The number of inversions are $(n-1) + (n-2) + \cdots + 2 + 1 = \sum_{k=1}^{n-1} k = n(n-1)/2$.

c. For each j, the number of inversions (i, j) is one less than t_j from line 5 in the code of insertion sort. Since the only lines to contribute an n^2 term are ones with t_j in the sums, if there are (for example) $\log_2(n)$ inversions, then the running time of inerstion sort will be $O(n \log_2(n))$. However, since terms contributing n to the running time come from lines not containing t_j, no matter how small the number of inversions, the running time will never be less than $O(n)$. ■

6 *Find the best-case and worst-case time for the Bubble-Sort and Better-Bubble-Sort algorithms given in class slides. The slides are posted on Piazza. Include the algorithms with your work.*

First we analyze the algorithms, similarly to how it is done in the book.

	BUBBLE-SORT	*cost*	*times*
1	**for** $m = A.length$ **downto** 2	c_1	n
2	**for** $i = 1$ **to** $m - 1$	c_2	$\sum_{m=2}^{n} m$
3	**if** $A[i] > A[i+1]$	c_3	$\sum_{m=2}^{n} m - 1$
4	$swap(A[i], A[i+1])$	c_4	$\sum_{m=2}^{n} m - 1$

	BETTER-BUBBLE-SORT	*cost*	*times*
1	**for** $m = A.length$ **downto** 2	c_1	n
2	$sorted = true$	c_2	$n - 1$
3	**for** $i = 1$ **to** $m - 1$	c_3	$\sum_{m=2}^{n} m$
4	**if** $A[i] > A[i+1]$	c_4	$\sum_{m=2}^{n} m - 1$
5	$swap(A[i], A[i+1])$	c_5	$\sum_{m=2}^{n} m - 1$
6	$sorted = false$	c_6	$\sum_{m=2}^{n} m - 1$
7	**if** $sorted = true$	c_7	$n - 1$
8	**return**	c_8	1

For bubble-sort, the best case is if the **if** loop is never evaluated, so we get

$$c_1 n + c_2 \sum_{m=2}^{n} m + c_3 \sum_{m=2}^{n} (m-1) = c_1 n + c_2 \left(\frac{n(n+1)}{2} - 1 \right) + c_3 \left(\frac{n(n-1)}{2} \right) = O(n^2).$$

The worst case happens if **if** is true every time, and line 4 is evaluated every time. Then the running time is

$$c_1 n + c_2 \sum_{m=2}^{n} m + c_3 \sum_{m=2}^{n} m - 1 + c_4 \sum_{m=2}^{n} (m-1) = c_1 n + c_2 \left(\frac{n(n+1)}{2} - 1 \right) + c_3 \left(\frac{n(n-1)}{2} \right) + c_4 \left(\frac{n(n-1)}{2} \right) = O(n^2).$$

For better-bubble-sort, the best case is if we hit line 8 as soon as possible (in the first go through the **for** loop in line 1) and the **if** in line 4 is never evaluated. That would be a running time of

$$c_1 + c_2 + c_3 n + c_4 (n-1) + c_7 + c_8 = O(n).$$

The worst case is similar to the worst case for bubble-sort, in fact it must be at least $O(n^2)$, since the code for bubble-sort is contained within the code for better-bubble-sort. Since none of the times are cubic in n, it must be that the running time for the worst case is also $O(n^2)$. ■

$*$ **7** *Find the best-case, worst-case and average-case running tines for Insertion-Sort. Show all work and explain any assumptions that you use. Include the algorithm with your work.*

8 *Prove that* $\lfloor -x \rfloor = -\lceil x \rceil$ *and that* $\lceil -x \rceil = -\lfloor x \rfloor$.

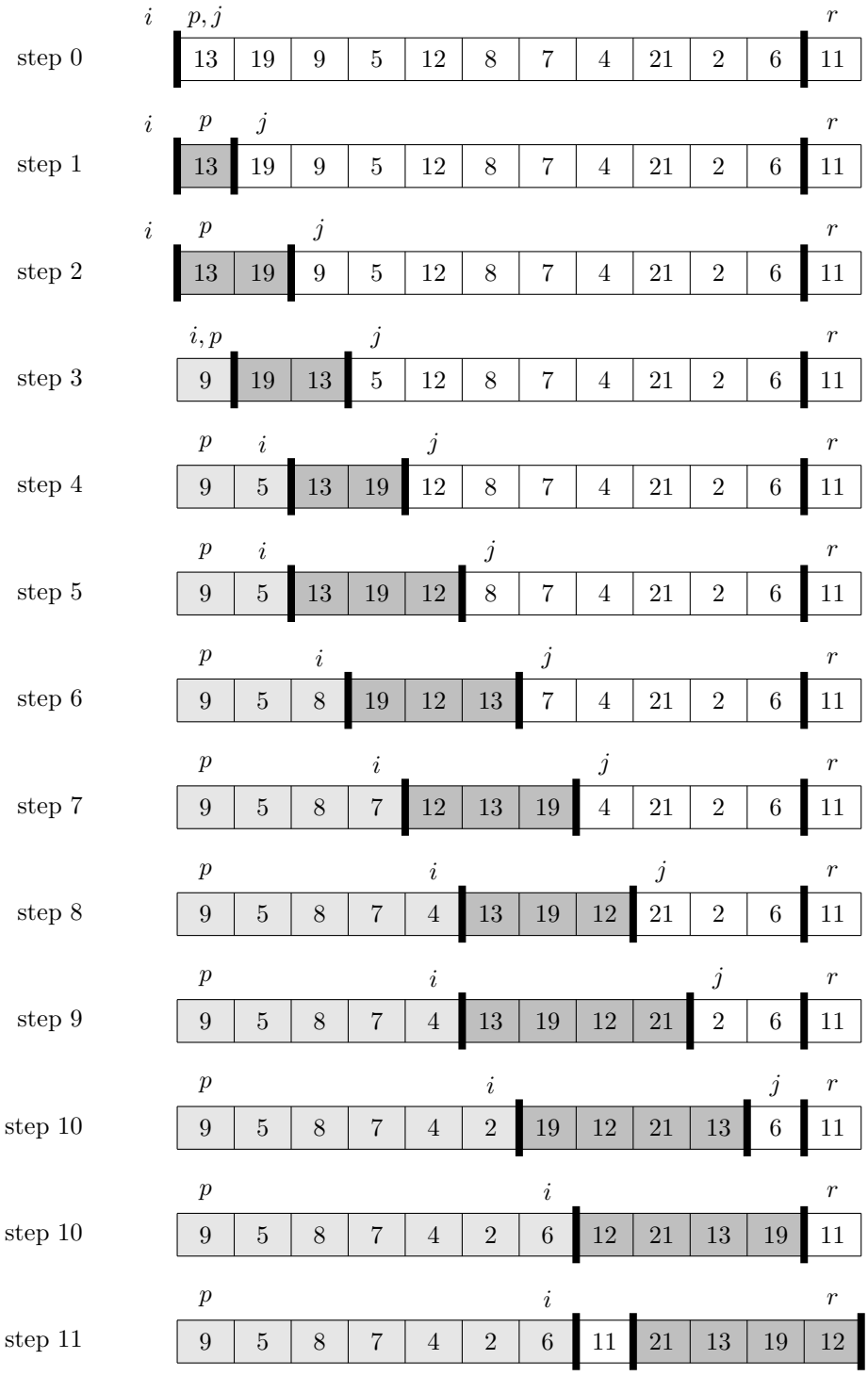
From the definition of the floor and ceiling functions, we have that

$$-x - 1 < \lfloor -x \rfloor \leqslant -x \qquad \text{and} \qquad x \leqslant \lceil x \rceil < x + 1 \iff -x \geqslant -\lceil x \rceil > -x - 1.$$

Since there is exactly one integer in the interval $(-x-1, -x]$, and both $\lfloor -x \rfloor$ and $-\lceil x \rceil$ are integers in this range, it must be that $\lfloor -x \rfloor = -\lceil x \rceil$. An analogous argumment works for the second part. ■

7.1-1 *(p.173)* *Using Figure 7.1 as a model, illustrate the operation of* PARTITION *on the array* $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$.

We follow Figure 7.1 with the given array. Step 0 has the array in the given order, and step 11 has the array in the final order.



The procedure returns the index value $i + 1 = p + 7$. ■

7.1-2 *(p.174)* *What value of q does* PARTITION *return when all elements in the array $A[p\,..\,r]$ have the same value? Modify* PARTITION *so that $q = \lfloor (p+r)/2 \rfloor$ when all elements in the array $A[p\,..\,r]$ have the same value.*

If all the elements in the array have the same value, then PARTITION will return the index r, because the **if** condition on line 4 will always be true, and so the value of i will be increased $r-p$ times, and since i starts out at $p-1$, we have that $p-1+r-p = r-1$. Out of the **for** loop the value of $i+1 = r-1+1 = r$ is returned.

Here is one way of modifying the PARTITION algorithm - by keeping a variable that is changed only if a different value from $A[r]$ is encountered.

```
 1  x = A[r]
 2  i = p − 1
 3  same = true
 4  for j = p to r − 1
 5      if A[j] ⩽ x
 6          i = i + 1
 7          exchange A[i] with A[j]
 8      if A[j] < x
 9          same = false
10  exchange A[i + 1] with A[r]
11  if same
12      return ⌊(p + r)/2⌋
13  else
14      return i + 1
```

■

7.1-3 *(p.174)* *Give a brief argument that the running time of* PARTITION *on a subarray of size n is $\Theta(n)$.*

Every **for** and **while** loop (usually) increases running time by a factor of n or $\log(n)$ (either way, a non-constant function of n), and in PARTITION we only have one **for** loop. The line on which the **for** loop appears is executed $r-p$ times, which is the length of the subarray being considered. The lines outside the **for** loop are executed eactly once and the lines inside the **for** loop are also executed at most $r-p-1 = n-1$ times. This means the running time is $a+bn = \Theta(n)$ for some appropriate constants a and b. ■

7.1-4 *(p.174)* *How would you modify* QUICKSORT *to sort into nonincreasing order?*

Here is one way of modifying the QUICKSORT algorithm - by switching the inequality in line 5 of PARTITION the other way:

```
5    if A[j] ⩽ x        →        5    if A[j] > x
```

Leaving the rest of the code alone, this puts all values greater than $A[r]$ on the left of $A[q]$ at the end, and all values less than or equal to $A[r]$ on the right of $A[q]$. Another method (which only changes QUICKSORT and not PARTITION) is swapping the two groups after PARTITION is done with them:

```
1  if p < r
2      q = PARTITION(A, p, r)
3      exchange A[p, . . . , q − 1] with A[q + 1, . . . , r]
4      q = r − (q − p)
5      QUICKSORT(A, p, q − 1)
6      QUICKSORT(A, q + 1, r)
```

Note it is not necessary to swap the parts after the last execution of the nested QUICKSORT, because then there is only one element in the subarray, so there is nothing to swap. ■

7-3 *(p.187)* *An alternative analysis of the running time of randomized quicksort focuses on the expected running time of each individual recursive call to* RANDOMIZED-QUICKSORT*, rather than on the number of comparisons performed.*

a. *Argue that, given an array of size n, the probability that any particular element is chosen as the pivot is $1/n$. Use this to define indicator random variables $X_i = I\{i\text{th smallest element is chosen as the pivot}\}$. What is $E[X_i]$?*

b. *Let $T(n)$ be a random variable denoting the running time of quicksort on an array of size n. Argue that*

$$E[T(n)] = E\left[\sum_{q=1}^{n} X_q(T(q-1) + T(n-q) + \Theta(n))\right].$$

c. *Show that we can rewrite equation (7.5) as*

$$E[T(n)] = \frac{2}{n}\sum_{q=2}^{n-1} E[T(q)] + \Theta(n).$$

d. *Show that*

$$\sum_{k=2}^{n-1} k\log_2(k) \leqslant \frac{1}{2}n^2\log_2(n) - \frac{1}{8}n^2.$$

Hint: Split then summation into two parts, one for $k = 2, 3, \ldots, \lceil n/2\rceil - 1$ and one for $k = \lceil n/2\rceil, \ldots, n-1$.

e. *Using the bound from equation (7.7), show that the recurrence in equation (7.6) has the solution $E[T(n)] = \Theta(n\log_2(n))$. Hint: Show, by substitution, that $E[T(n)] \leqslant an\log_2(n)$ for sufficiently large n and for some positive constant a.*

a. Since the pivot is chosen at random from the array, every element has the same probability of being chosen. This means each element has probability $1/n$ of being chosen. By Lemma 5.1, we know that for an indicator variable X_A evaluating to 1 if event A occurs and 0 otherwise, $E[X] = \Pr(A)$. Since the ith smallest element of the array is just one specific element, we have that $E[X_i] = 1/n$ for all i.

b. Previously we argued that PARTITION has running time $\Theta(n)$. If index q is chosen as the pivot, then we immediately see that $E[T(n)] = E[\Theta(n) + T(q-1) + T(n-q)]$ by the code of QUICKSORT. Since the indicator vairbale X_q is 1 only if q is chosen as the pivot, we may cover all the possibilities by summing over all q and multiplying each term by X_q, giving the desired form.

c. Equation (7.5) is the equation from part **b.** above, and simplifying it is just algebra.

$$\begin{aligned}
E[T(n)] &= E\left[\sum_{q=1}^{n} X_q(T(q-1) + T(n-q) + \Theta(n))\right] && \text{(given)}\\
&= \sum_{q=1}^{n} E\left[X_q(T(q-1) + T(n-q) + \Theta(n))\right] && \text{(linearity of } E\text{)}\\
&= \sum_{q=1}^{n} E\left[X_q\right] E\left[T(q-1) + T(n-q) + \Theta(n))\right] && \text{(independence of } X_q \text{ and } T(n)\text{)}\\
&= \frac{1}{n}\sum_{q=1}^{n} E\left[T(q-1) + T(n-q) + \Theta(n))\right] && \text{(part } \textbf{a.} \text{ above)}\\
&= \frac{1}{n}\sum_{q=1}^{n} E[\Theta(n)] + \frac{1}{n}\sum_{q=1}^{n}\left(E\left[T(q-1)\right] + E\left[T(n-q)\right]\right) && \text{(linearity of } \Sigma \text{ and } E\text{)}\\
&= \frac{n}{n}\cdot\Theta(n) + \frac{1}{n}\sum_{q=1}^{n}\left(E\left[T(q-1)\right] + E\left[T(n-q)\right]\right) && (\Theta \text{ is not a random variable})\\
&= \Theta(n) + \frac{2}{n}\sum_{q=1}^{n} E\left[T(q-1)\right] && (\text{since } q-1 = n-(n-(q-1)))\\
&= \Theta(n) + \frac{2E[T(0)]}{n} + \frac{2E[T(1)]}{n} + \frac{2}{n}\sum_{q=3}^{n} E\left[T(q-1)\right]
\end{aligned}$$

$$= \Theta(n) + \frac{2}{n}\sum_{q=3}^{n} E\left[T(q-1)\right] \qquad (T(0),\, T(1) \text{ have linear running time})$$

$$= \Theta(n) + \frac{2}{n}\sum_{q=2}^{n-1} E\left[T(q)\right] \qquad (\text{reindexing})$$

In the second line from the bottom the running times $T(0)$ and $T(1)$ are absorbed into the $\Theta(n)$, and as following the algorithm we do approximately n executions.

d. This part is also just algebra. We follow the hint to get

$$\begin{aligned}
\sum_{k=2}^{n-1} k\log_2(k) &= \sum_{k=2}^{\lceil n/2\rceil-1} k\log_2(k) + \sum_{k=\lceil n/2\rceil}^{n-1} k\log_2(k) && \text{(taking the hint)}\\
&\leqslant \sum_{k=2}^{\lceil n/2\rceil-1} k\log_2(n/2) + \sum_{k=\lceil n/2\rceil}^{n-1} k\log_2(n) && \text{(bounding above)}\\
&= \log_2(n/2)\sum_{k=2}^{\lceil n/2\rceil-1} k + \log_2(n)\sum_{k=\lceil n/2\rceil}^{n-1} k && \text{(factoring)}\\
&= \log_2(n)\sum_{k=2}^{n-1} k - \log_2(2)\sum_{k=2}^{\lceil n/2\rceil-1} k && \text{(laws of logarithms)}\\
&= \log_2(n)\left(\frac{n(n-1)}{2} - 1\right) - \left(\frac{\left\lceil \frac{n}{2}\right\rceil\left(\left\lceil \frac{n}{2}\right\rceil - 1\right)}{2} - 1\right) && \text{(sum formula)}\\
&\leqslant \log_2(n)\left(\frac{n(n-1)}{2} - 1\right) - \left(\frac{\frac{n}{2}\left(\frac{n}{2} - 1\right)}{2} - 1\right) && \text{(bounding above)}\\
&= \log_2(n)\cdot\frac{n^2-n-2}{2} - \frac{n^2-2n-8}{8}\\
&= \frac{1}{2}n^2\log_2(n) - \frac{1}{8}n^2 - \left(\frac{n}{2}\log_2(n) + \log_2(n) + \frac{n}{4} + 1\right)\\
&\leqslant \frac{1}{2}n^2\log_2(n) - \frac{1}{8}n^2.
\end{aligned}$$

Note in the second-to-last line the last term is positive for all positive integers n, so adding it simply bounds the original expression from above.

e. Applying substitution, we guess that $E[T(n)] \leqslant an\log_2(n)$ for sufficiently large n and for some positive constant a. Plugging this into the results from parts **c.** and **d.** above, we get

$$\begin{aligned}
E[T(n)] &= \frac{2}{n}\sum_{q=2}^{n-1} E[T(q)] + \Theta(n) && \text{(part } \mathbf{c.} \text{ above)}\\
&= \frac{2}{n}\sum_{q=2}^{n-1} aq\log_2(q) + \Theta(n) && \text{(guessing the result)}\\
&\leqslant \frac{2a}{n}\left(\frac{1}{2}n^2\log_2(n) - \frac{1}{8}n^2\right) + \Theta(n) && \text{(part } \mathbf{d.} \text{ above)}\\
&= an\log_2(n) - \frac{a}{4}n + \Theta(n)\\
&\leqslant an\log_2(n) && \text{(for } a \text{ large enough)}\\
&= \Theta(n\log_2(n)).
\end{aligned}$$

Note that $\Theta(n) \not\subset \Theta(n\log_2(n))$, but by making a large enough, the term $-an/4$ cancels the effect of any constant function $bn \in \Theta(n)$. ■

∗ **11** *Find a big-O for* QUICKSORT *for the following three cases:*

a. *a pre-sorted array;*

b. *a reverse pre-sorted array;*

c. *an array with all equal elements.*

Consider a situation where the recursive calls to QUICKSORT *alternate good case, bad case, good case, bad case, etc. Under what conditions could this possibly happen?*