Problem Set 2 Solutions

Problem 2-1. Egyptian Pyramid Architecture

(a) For this problem, Khufru just needs to merge two valid skylines. A valid skyline is effectively a list of points in two dimensions sorted by their X positions with non-negative Y positions. The following is one possible approach at a solution.

At a high level, we want to be able to scan our skylines together, from left to right, and generate the new resulting skyline. We do this by first making the skylines easier to compare, then resolving all the places where the skylines intersect. The resulting skyline is simply the skyline formed by taking the higher segments.

![Diagram of Egyptian Pyramid skyline]

**Figure 1:** Egyptian Pyramid skyline

There are three steps therefore. See Figure 1 while reading this for visualizations:

- First we make sure that we can easily compare the skylines. To do this, we generate two new skylines that look exactly the same, but that have small line chunks that line up on the x-axis. So, if there is a line from X positions 3 to 7 on one skyline, and in the other skyline we have a line that ends at X position 4, we turn our 3 to 7 line into two lines, one that goes from 3 to 4 and one from 4 to 7 (that have the same angle and look like the same line). If we do this for both skylines, then the skylines will end up being two valid skylines formed
by small line segments that are easily comparable. If the line segments do not intersect, then we can always select the higher segment to get our skyline. If they do though, we need to do more.

* We once again create two new skylines. These skylines will be exactly the same as the old skylines, except with additional points inserted where two lines intersect. Once we do this, we have two skylines formed as a bunch of small lines that do not intersect.

* Now we just generate a resulting skyline by consistently picking the higher line in the two skylines. We know these are just a bunch of non-intersecting segments, so we just pick the higher one for our final output.

The above was the high-level description. Now here is the pseudo-code:

First, we need some convenience functions.

**INTERSECT-P** decides if two lines intersect. It requires that the left X coordinates and the right X coordinates are shared by the two lines. It therefore only cares for the y points of the two lines. Note that two equal line segments are not considered intersecting.

```
INTERSECT-P(x_1, y_1, x_1, y_1, x_2, y_2, x_2, y_2)
1    if ((y_1 > y_2) and (y_1 > y_2)) or ((y_1 < y_2) and (y_1 > y_2))
2        then return True
3        else return False
```

The above clearly terminates in running time $O(1)$, and works because the only way for an intersection to happen is for the left side to start higher and the right side to be lower or vice versa.

**INTERSECTION** returns the $(x, y)$ position of where the intersection actually (you must submit two valid lines that intersect, and whose $x_1 = x_2$, and $x_1 = x_2$).

```
INTERSECTION(x_1, y_1, x_1, y_2, x_2, y_2, x_2, y_2)
1    slope1 ← ((y_1 - y_1)/(x_1 - x_1))
2    off1 ← (y_1)/(slope1 * x_1)
3    slope2 ← ((y_2 - y_2)/(x_2 - x_2))
4    off2 ← (y_2)/(slope2 * x_2)
5    X ← (off1 - off2)/(slope2 - slope1)
6    Y ← (slope1 * X) + off1
7    return (X, Y)
```

The above works (it’s basic algebra). Its running time is also $O(1)$.

**PREDICTION** attempts to compute the y value of a point on a line formed by two points $(x_i, y_i), (x_r, y_r)$. 
PREDICTION\((x_l, y_l, x_r, y_r, X)\)
1 \(\text{slope} \leftarrow (y_r - y_l)/(x_r - x_l)\)
2 \(\text{off} \leftarrow (y_l/(\text{slope} \times x_l))\)
3 \(Y = (\text{slope} \times X) + \text{off}\)
4 \text{return} \((X, Y)\)

Again, basic algebra suggests the above works, and it runs in \(O(1)\) time.

\text{HIGHER} will tell us which line segment is higher. It assumes that the \(X\) coordinates are lined up, and the lines do not intersect. It returns the \((X, Y)\) pair that is the left side of the higher line. If the two lines are always equal, it will just return the left point of the first line.

\text{HIGHER}\((x_{1l}, y_{1l}, x_{1r}, y_{1r}, x_{2l}, y_{2l}, x_{2r}, y_{2r})\)
1 \text{if } \((y_{1l} < y_{2l}) \text{ or } (y_{1r} < y_{2r}) \) \text{ then return } \((x_{2l}, y_{2l})\)
2 \text{else}
3 \text{return} \((x_{1l}, y_{1l})\)

Again the above obviously does what’s described since the lines can’t intersect. It runs in \(O(1)\).

With the above functions we can now construct our three pass system. The first pseudo-code is the algorithm \text{SKYLINE\_MERGE} which is the algorithm asked for in the problem set:

\text{SKYLINE\_MERGE}\((S_1, S_2)\)
1 \((N_1, N_2) \leftarrow \text{SKYLINE\_MERGE\_PHASE1}\((S_1, S_2)\)\)
2 \((C_1, C_2) \leftarrow \text{SKYLINE\_MERGE\_PHASE2}\((N_1, N_2)\)\)
3 \text{return} \text{SKYLINE\_MERGE\_PHASE3}\((C_1, C_2)\)

Where each of these phases were described above. The pseudo-code for each phase follows here:

Phase 1 aligns all the existing \(X\)-coordinates. To make life easier, we insert two points \((0, 0)\) at the beginning and \((Z, 0)\) at the end (where \(Z\) is the larger final \(X\) coordinate in the two skylines) if they are not there.
SKYLINE MERGE PHASE 1(S1, S2)

1 \( N_1[0] \leftarrow (0,0) \)
2 \( N_2[0] \leftarrow (0,0) \)
3 \( i_1 \leftarrow 1 \)
4 \( i_2 \leftarrow 1 \)
5 \( j \leftarrow 2 \)
6 \( done \leftarrow False \)
7 \[[\text{Comment: lines 7 to 16 make sure they start at (0,0) and end at the same X position}]\]
8 \( \text{if } S_1[1] = (0,0) \)
9 \( \text{then } i_1 \leftarrow i_1 + 1 \)
10 \( \text{if } S_2[1] = (0,0) \)
11 \( \text{then } i_2 = i_2 + 1 \)
12 \( Z \leftarrow \text{MAX}(X - \text{COORD}(S_1[length(S_1)]), X - \text{COORD}(S_2[length(S_2)])) \)
13 \( \text{if } Z \neq (X - \text{COORD}(S_1[length(S_1)])) \)
14 \( \text{then } S_1[length(S_1) + 1] \leftarrow (Z, 0) \)
15 \( \text{if } Z \neq (X - \text{COORD}(S_2[length(S_2)])) \)
16 \( \text{then } S_2[length(S_2) + 1] \leftarrow (Z, 0) \)
17 \[[\text{Comment: Now for the actual work of phase 1}]\]
18 \( \text{while } (X - \text{COORD}(N_1[j])) \neq Z \)
19 \( \text{do if } (X - \text{COORD}(S_1[i_1])) = (X - \text{COORD}(S_2[i_2])) \)
20 \( \text{then} \)
21 \( N_1[j] \leftarrow S_1[i_1] \)
22 \( N_2[j] \leftarrow S_2[i_2] \)
23 \( i_1 \leftarrow i_1 + 1 \)
24 \( i_2 \leftarrow i_2 + 1 \)
25 \( \text{else if } (X - \text{COORD}(S_1[i_1])) < (X - \text{COORD}(S_2[i_2])) \)
26 \( \text{then} \)
27 \( N_1[j] \leftarrow S_1[i_1] \)
28 \( N_2[j] \leftarrow \text{PREDICTION}(N_2[j - 1], S_2[i_2], X - \text{COORD}(S1[i_1])) \)
29 \( i_1 \leftarrow i_1 + 1 \)
30 \( \text{else} \)
31 \( N_2[j] \leftarrow S_2[i_2] \)
32 \( N_1[j] \leftarrow \text{PREDICTION}(N_1[j - 1], S_1[i_1], X - \text{COORD}(S2[i_2])) \)
33 \( i_2 \leftarrow i_2 + 1 \)
34 \( j \leftarrow j + 1 \)
35 \( \text{return } (N_1, N_2) \)

The above was phase 1. It's a rather obfuscated way to implement a scan from left to right (on the X axis) where we copy points over to a new list, and generate additional points if necessary to get all the X coordinates to line up.

Here is phase 2:
SKYLINE_MERGE_PHASE2($N_1, N_2$)
1 $C_1 \leftarrow$ empty array of points
2 $C_2 \leftarrow$ empty array of points
3 $j \leftarrow 1$
4 $i \leftarrow 1$
5 $C_1[i] \leftarrow N_1[j]$
6 $C_2[i] \leftarrow N_2[j]$
7 $j \leftarrow j + 1$
8 $Z \leftarrow (N_1[length(N_1)])$
9
10 while ($C_1[i] \neq Z$)
11 \hspace{1em} do if (not INTERSECT-P($C_1[i], N_1[j], C_2[i], N_2[j]$))
12 \hspace{2em} then
13 \hspace{3em} $i \leftarrow i + 1$
14 \hspace{3em} $C_1[i] \leftarrow N_1[j]$
15 \hspace{3em} $C_2[i] \leftarrow N_2[j]$
16 \hspace{3em} $j \leftarrow j + 1$
17 \hspace{2em} else
18 \hspace{3em} $p \leftarrow$ INTERSECTION($C_1[i], N_1[j], C_2[i], N_2[j]$)
19 \hspace{3em} $i \leftarrow i + 1$
20 \hspace{3em} $C_1[i] \leftarrow p$
21 \hspace{3em} $C_2[i] \leftarrow p$
22 \hspace{1em} return ($C_1, C_2$)

And voila, phase 2 is done. Now we can sort of see why Khufu had difficulties. phase2 takes a bunch of segments that are all lined up, but that may intersect each other, and creates a new set of even MORE segments that represent the same skylines, but that wherever we had an intersection, now we have two segments, one ending at the intersection point, and one starting at it.

Finally, we get to the shortest phase, phase 3:

SKYLINE_MERGE_PHASE3($C_1, C_2$)
1 \hspace{1em} result \leftarrow empty array of points
2 \hspace{1em} for $i \leftarrow 1$ to $(length(C_1) - 1)$
3 \hspace{2em} do
4 \hspace{3em} result[$i$] = HIGHER($C_1[i], C_1[i + 1], C_2[i], C_2[i + 1]$)
5 \hspace{2em} result[$i$] = $C_1[i]$
6 \hspace{1em} return result

Phase 3 just resolves the skylines.

Now that we’ve provided all the pseudo-code, we can analyze the algorithm we made. We argue it works based on the following. The first phase generates two skylines that are exactly the same appearance as the original two skylines (just more points are added, but they are all on the skylines, so the skyline view doesn’t change). The second phase ALSO does the same (adding more points onto the skyline without
changing with the skyline looks like). The third phase is the decision phase which takes two skylines that look like the original skylines, but that are formed by line segments that are 'lined up' on the X-axis and that do NOT intersect. At this point, all phase 3 does is pick the higher one, which is what the resulting skyline is.

Arguing that it runs in linear time is a little harder. We will analyze each phase separately.

- phase 1 at the WORST case inserts a new point into skyline 2 for every point in skyline 1, and vice versa. Therefore, at WORST we take our input of size $n$, and make it into an input of size $2n$. We do this by scanning the data from left to right and running constant time operations on it. Therefore, it takes $O(n)$ operations.

Alternatively, you can look at the actual source code for the analysis: At every pass in the loop, we increment either $i_1$ or $i_2$, or both. We know that if we reach the final point in either array, we’ve reached the final point in both (we go up based on X-axis positions, and we’ve ensure that the final point in both skylines are at the same X coordinate). Therefore, we will not go through the loop more than the sum of the length of the two input skylines, which is again $O(n)$. All the operations inside the loop are $O(1)$, which results in an $O(n)$ result for phase 1.

- phase 2 Also runs in $O(n)$. We start with two skylines that 'line up' on the X-axis. We then insert additional points when the line segments intersect. There are $n$ line segments in each skyline, but only segments that line up can intersect. So, we have a possible maximum of $n$ intersections to add. So, again, phase 2 takes two skylines of length $n$ and makes two skylines of length at most $2n$ using $O(1)$ operations, by scanning from left to right. This again takes $O(n)$.

Instead of that, one can again look at the code, although it’s a little harder to argue this. Basically, the loop terminates when we’ve copied the last element out. To get to the last element, $j$ must be increased to the last value. However, $j$ only increases in the first case (when the lines don’t intersect). However, one can argue that the first case must occur at least once for each time the second case happens. The reason is that if there are two lined-up segments that intersect (the reason we went to case 2), and we then examine the resulting segments (after we break up the segments into shorter segments that go up to the intersection point) we can’t have another intersection (or the original line segments were not straight lines). Therefore, case 1 must happen at LEAST half the time. In the worst case, it only happens half the time, which means the loop is run $2n$ times. Every operation in the loop is $O(1)$. Therefore, phase two is $O(n)$.

- finally, phase 3 is also $O(n)$. It is given as input lots of lined-up non-intersecting line segments, and just spits out the 'higher' line segments. This is a scan from left to right on at most $4n$ data, which takes $4n$ time, which is $O(n)$.

Alternatively, by looking at the code, we terminate when i points at the last element, and we increase i on every pass of the loop. All operations in the loop are $O(1)$, and so the code is $O(n)$. 

Handout 16: Problem Set 2 Solutions
So, we have an algorithm that runs in time $O(n) + O(n) + O(n)$ which is of course $O(n)$. We can even try to argue that one can’t do asymptotically better, since we need to scan through all inputs (length $n$) AND write an output (length at least $n$), and this can’t be done faster than $O(n)$ (although arguing this was not expected for this problem).

So far, we have avoided Khufru’s fate.

(b) Now that we have the `SKYLINE MerGE` algorithm, we need to create an algorithm for figuring out a skyline given a set of pyramids. We will use a divide-and-conquer approach.

The idea is simple. Given $n$ pyramids, we will split up our set of pyramids in half, and try to figure out the skyline formed by each half. Then, when we have two skylines, we will call `skyline_merge` to merge the two skylines. To resolve the two half skylines, we will call ourselves recursively down to one pyramid, whose skyline is obvious.

Algorithm `RECURSIVE SKYLINE` takes a set of pyramids $S$, and does what we described above. $|S|$ is the number of items in $S$. We assume that $S[x]$ is three points representing pyramid $x$, which also looks like a skyline.

```plaintext
RECURSIVE SKYLINE(S, start, end)
1  if (start = end)
2    then return (S[start])
3  else
4    split ← floor((end - start) / 2)
5    skyline1 ← RECURSIVE SKYLINE(S, start, start + split)
6    skyline2 ← RECURSIVE SKYLINE(S, start + split + 1, end)
7    return SKYLINE MERGE(skyline1, skyline2)
```

First we need to argue that this algorithm does what it should do. We could do induction on the number of pyramids. However, we’re only arguing, not trying to prove, so the high-level argument is simple. We know we can generate a skyline for a single pyramid (it’s just the pyramid points). We know we have a way to combine two pyramid skylines. The above algorithm basically takes the set of pyramids, and generates skylines out of all of them individually. Then it combines the skylines ‘pairwise’ over and over until it gets one skyline. It’s clear that any ordering of the pairwise combination will yield the same skyline, so the above algorithm must work assuming our merge function works.

For our runtime analysis, we simply write out the recurrence equation:

$$T(n) = 2T(n/2) + O(n)$$  \hspace{1cm} (1)

The solution to that recurrence is $T(n) = O(n \log n)$.

If one wants to be extra careful, one can analyze the relationship between the number
of pyramids and the number of points (1 to 3), so really \( n \) pyramids gives us \( 3n \) points for the skyline, but \( 3n \log 3n \) is still \( O(n \log n) \).

(c) The idea in this problem is to perform a reduction. That is, show that if we have a black-box that can solve the skyline problem, we can create a computer than can solve sorting. In this case, we have a black-box that gives us a skyline in time \( O(n) \), and we want to be able to use it to sort in \( O(n) \). This means that whatever else we do apart from using the box can’t take longer than \( O(n) \). So running quicksort on the input is a no-no.

It turns out that there is a very simple solution to this problem. Given an input \( A = < a_1, a_2, \ldots, a_n > \), we generate a set of pyramids \( P = < p_1, p_2, \ldots, p_n > \). We use the following idea; each pyramid \( p_i \) represents the number \( a_i \). Specifically, the pyramid is made of the three coordinates \((a_i - 0.1, 0)\), \((a_i, 1)\), \((a_i + 0.1, 0)\). Basically, we turn our input into a bunch of very skinny pyramids that don’t intersect. Since we don’t have any repeats of integers, we don’t have to worry about two overlapping pyramids. After we feed this input to the skyline, it will spit out a skyline of skinny pyramids, but the representation of this skyline is a bunch of points sorted by their \( X \)-axis. We only need to now extract all points whose \( y \)-coordinate is 1, and we’ve successfully sorted the input.

We now need to argue that this doesn’t increase the running time. Specifically, apart from using the black-box, all our additional work is \( O(n) \). Well, converting an input of \( n \) integers to the pyramids takes around \( 3n \) (we have to write out 3 points for each integer input). Feeding the \( 3n \) points to the black box also takes \( 3n \). We then take the resulting skyline (that is around \( 3n \) points) and scan through all the values from left to right. Every time we see a \( y \)-coordinate of 1, we print something. This also takes \( 3n \). So, in all, this runs in \( O(n) \) time.

Yay.

**Problem 2-2.** Verifying Matrix Multiplication

(a) **Complexity:** Computing the dot product of two \( n \)-ary vectors \( v_1 \) and \( v_2 \) can be done in \( \Theta(n) \) operations in the obvious way (and results in a scalar). Simply compute \( \sum_{i=1}^{n} v_1[i]v_2[i] \) by multiplying the \( i \)th elements in turn and summing the products.

Computing the product of an \( n \times n \) matrix \( M \) and an \( n \)-ary vector \( v \) can be done \( \Theta(n^2) \) multiplications and additions (and results in an \( n \)-ary vector). To do so, compute the \( n \) individual dot products of the rows of \( M \) with \( v \).

Deciding the equality of two \( n \)-ary vectors can be done in \( \Theta(n) \) comparisons but comparing each pair of corresponding elements in turn.

Thus, it immediately follows that we can compute \( Cx \) in \( \Theta(n^2) \). We note that \((A \times B)x = (A \times (B \times x))\). Computing the latter term requires 2 matrix-vector multiplications.

Our implementation is as follows:

1. Select \( n \)-ary vector \( x \) uniformly at random from \( \{0, 1\}^n \).
2. Compute $C x$.
3. Compute $(A \times (B \times x))$
4. If the result vectors are equal, output YES, otherwise output NO.

The whole operation as described takes $\Theta(n) + \Theta(n^2) + 2\Theta(n^2) + \Theta(n) = \Theta(n^2)$ operations, as required.

(b) **Correctness I:** If $A \times B = C$, then $(A \times B)x = (A \times (B \times x)) = C x$ for any $x$. So, yes, the algorithm will always output YES for this case.

(c) **Correctness II:** Assume that $A \times B \neq C$. We want to compute $Pr[(A \times (B \times x)) = C x]$ over the choices of random $n$-ary binary vectors.

$$Pr[(A \times (B \times x)) = C x] = Pr[(A \times B) \times x = C x] = Pr[((A \times B) \times x) - C x = 0] = Pr[((A \times B) - C)x = 0]$$

Let $M$ be the nonzero matrix $((A \times B) - C)$. (We know $M$ is nonzero by our assumption.) Without loss of generality let $r$ be the first nonzero row of $M$.

$$Pr[((A \times B) - C)x = 0] = Pr[Mx = 0] \leq Pr[rx = 0]$$

Thus, it is sufficient to show that the probability that the dot product of a uniformly random binary vector and arbitrary nonzero vector is not zero is at least $1/2$. Note that our probability space is the choice of $x$. So it is sufficient to show that for all $r$, at least $1/2$ of all possible choices of $x$ result in a nonzero dot product $rx$.

**Theorem 1** For every nonzero $n$-ary vector $r$ and $n$-ary binary vector $x$ s.t. $rx = 0$, there exists a distinct $n$-ary binary vector $x'$ for which $rx \neq 0$.

**Proof** (by construction) We construct a mapping from every $x$ that yields a zero dot product with $r$ to a distinct $x'$ that yields a nonzero dot product with $r$. Thus, at most $1/2$ of all choices of $x$ can yield a zero dot product. Let $r[j]$ be the first nonzero element of $r$. There are two cases.

If $x[j]$ is zero, then setting $x'[j]$ to one adds $r[j]$ to the dot product and thus yields a nonzero result as needed. That is, select $x'$ s.t. $\forall i(1 \leq i \leq n, j \neq i) : x'[i] = x[i]$ and $x'[j] = 1$. Computing $rx' rx' = rx + r[j] = 0 + r[j] = r[j] \neq 0$ as needed.

If $x[j]$ is one, then setting $x'[j]$ to zero subtracts $r[j]$ from the dot product and thus yields a nonzero result as needed. That is, select $x'$ s.t. $\forall i(1 \leq i \leq n, j \neq i) : x'[i] = x[i]$ and $x'[j] = 0$. Computing $rx' rx' = rx - r[j] = 0 - r[j] = -r[j] \neq 0$ as needed.
(d) To determine correctly, with probability $1 - (1/2)^k$ whether a given matrix $C$ output by the the matrix multiplication chip on input matrices $A$ and $B$ is correct, independently select $k$ random vectors $x_1, x_2, \ldots, x_k$. For each vector $x_i$, check if $(A \times B)x_i = Cx_i$ by the method described above in Problem 2-2. If all $k$ checks output YES then output YES. Otherwise, if any check outputs NO, output NO.

The algorithm could be wrong by either reporting a false negative (output NO when the multiplication is in fact correct) or by reporting a false positive (output YES when the multiplication is in fact wrong).

The probability of the former error is the probability that any instance of the single vector check reports NO when the multiplication is, in fact, wrong. Above in Problem 2-2 we argued that the probability of any instance reporting a false negative is 0. So, by the union bound, the probability of the whole algorithm reporting a false negative is 0.

In order for the whole algorithm to report a false positive, it must compute $k$ individual false positives. Above in Problem 2-2 we argued that the probability of any instance reporting a false positive was at most 1/2. Since each vector is chosen independently, the probability that any single check of the multiplication reports a false positive is independent of the probability that any other single check of the multiplication reports a false positive. Thus the probability of a false positive from the whole algorithm is the product of the probability that each single check of the multiplication reports a false positive. Thus,

$$Pr[\text{false positive of whole}] = \prod_{i=1}^{k} Pr[\text{false positive in instance } i] \leq (1/2)^k.$$  

So the probability of correctness for the whole algorithm is

$$1 - Pr[\text{error}] \geq 1 - ((1/2)^k + 0) = 1 - (1/2)^k$$

as required.

The running time of the algorithm is the sum of the running times of the $k$ instances of the single vector checks. Specifically $k \times \Theta(n^2) = \Theta(k \times n^2)$.

---

**Problem 2-3.** Know-it-all

Given an array $A$ of $n$ integers in the range $[0..k]$ we preprocess them to answer later range queries as follows. $B$ is an array of size $k$, such that $B[i]$ will denote
RANGE\_PREPROCESS(A, B)
1   for i ← 0 to k
2       do
3           B[i] ← 0
4   j ← 0
5   while j ≠ (length(A) - 1)
6       do
8           j ← j + 1
9   i ← 1
10  while i ≠ (k + 1)
11     do
12        B[i] ← B[i] + B[i - 1]
13       i ← i + 1
14  return B

RANGE\_LOOKUP(B, a, b)
1  if (a = 0)
2     then
3        return B[b]
4   else
5      return B[b] - B[a - 1]

Run time The initialization and final loops take Θ(k) steps. The middle loop takes Θ(n) steps. So the whole preprocessing phase takes Θ(n + k).
Lookup is constant time, as needed.

Correctness We assume the correctness of the initialization loop. For the other two loops we identify the guard, show an invariant and conclude the loop establishes the desired post condition. We use the post condition of the final preprocessing loop to show the correctness of the lookup procedure.

Guard 1: j ≠ (length(A) - 1)
Invariant 1: \( \forall i(0 \leq i < k) \) B[i] is # of integers = i in A[0...j - 1].

Proof (by induction on the number of iterations j)

1.Base Case (j = 0) The facts that B is zero everywhere and the range 0... - 1 is empty are enough to establish the invariant.

2.Inductive Case (j > 0) By the inductive hypothesis we know that \( \forall i(0 \leq i < k) \) B[i] is # of integers = i in A[0...j - 1]. We need to show the equality now holds over the
sub-array $A[0 \ldots (j + 1) - 1]$. But there is only one number in $A[0 \ldots (j + 1) - 1]$ not in $A[0 \ldots j] - 1$. That number is $A[j]$. However in line 7 we add increment $B[A[j]]$ and that reestablishes the invariant.

**Post 1:** $\forall i (0 \leq i < k) \ B[i]$ is # of integers $= i$ in $A$.

It is easy to see that the negation of Guard 1 and Invariant 1 establish Post 1.

**Guard 2:** $i \neq k + 1$

**Invariant 2:** $\forall p (0 \leq p < i) \ B[p]$ is # of integers $\leq p$ in $A$.

**Proof** (by induction on the number of iterations $i$)

1. **Base Case** ($i = 0$) We know from Post 1 that $B[0]$ contains the # of integers $= 0$ in $A$. Since $A$ only contains integers in the range $[0..k]$ this is exactly the number of integers $\leq 0$ in $A$, as needed.

2. **Inductive Case** ($i > 0$) By the inductive hypothesis we know that $\forall p (0 \leq p < i) \ B[p]$ is # of integers $\leq p$ in $A$. We need to show the equality now holds when $p$ ranges from 0 to $i + 1$. The only new number in the range is $i + 1$. In line 12 we add $B[(i + 1)]$ to $B[(i + 1) - 1]$. By the I.H. we know that $B[i]$ contains the # of integers $\leq i$ in $A$. By Post 1, we know that $B[i + 1]$ contains the # of integers $= i + 1$ in $A$. So storing that sum in $B[i + 1]$ reestablishes the invariant.

**Post 2:** $\forall p (0 \leq p \leq k) \ B[p]$ = # of integers $\leq p$ in $A$. It is easy to see that the negation of Guard 2 and Invariant 2 establish Post 2.

**Lookup:** The number of integers in the range $[a \ldots b]$ is just the number of integers $\leq b$ minus the number of integers $< a$. The latter number is 0 if $a = 0$ or the number of numbers $\leq a - 1$, O.W. By Post 2 we know that $B[b]$ and $B[a]$ have exactly these two quantities stored.

**Problem 2-4.** Paranoid Quicksort

(a) To compute the running time of paranoid quicksort, we will analyze the recursion trees generated by the algorithm. We will show the expected running time is $\Theta(n \log n)$.

First, notice that in order for the algorithm to recurse, the smaller part of a partition must contain at least $1/5$ of the elements being partitioned. Likewise, the larger part can contain no more than $4/5$ of the elements. Therefore, at level $i$ of the recursion tree, the number of elements in the largest partition is at most $(4/5)^i n$. Thus, the depth of the recursion tree is at most $\log_{5/4} n$. Since every leaf of the tree is a partition containing a single element, the recursion tree has $n$ leaves. So the depth of the recursion tree is at least $\log_2 n$. Thus, the recursion tree of paranoid quick sort is always $\Theta(\log n)$ deep.
With this in mind, our analysis proceeds as follows.

\[
E[\text{total running time}] = E \left[ \sum_{\text{nodes in tree}} (\text{running time at node}) \right]
\]

\[
= \sum_{\text{nodes in tree}} E[\text{running time at node}]
\]

\[
= \sum_{\text{nodes in tree}} E[(\text{size of partition})(\text{number of partitions})]
\]

\[
= \sum_{\text{nodes in tree}} E[(\text{size of partition})] E[(\text{number of partitions})]
\]

\[
= \sum_{\text{nodes in tree}} E[(\text{size of partition})] (5/3)
\]

\[
= (5/3) E \left[ \sum_{\text{nodes in tree}} (\text{size of partition}) \right]
\]

\[
= (5/3) E \left[ \sum_{\text{levels}} \sum_{\text{nodes in level}} (\text{size of partition}) \right]
\]

\[
= (5/3) E \left[ \sum_{\text{levels}} n \right]
\]

\[
= (5/3) E[n \Theta(\log n)]
\]

\[
= (5/3) \Theta(n \log n)
\]

\[
= \Theta(n \log n)
\]

The first equality substitutes an equivalent definition of total running time. The second equality follows by linearity of expectation. Each partition takes time linear in the number of elements to be partitioned. The running time at a node is the product of the size of the sub-array to be partitioned and the number of partitions chosen until a “good” one is found, giving the third equality. The size of a partition depends only on the random selections made at nodes above the current node in the recursion tree. The number of partitions done at a node depends only on random choices made at this node. So the two random variables are independent. Thus, in this case, the expectation of the product is the product of the expectations, giving the fourth equality. The probability \( p \) of a “good” split on any given partition is 3/5 since it is the chance that element selected in from the middle three quintiles of the sub-array. Since each partition is an independent trial, the expected number of trials until success is 1/p or 5/3. Finally, we rearrange constant terms, count the number and size of nodes in the tree, and use linearity of expectations to conclude that the expected running time of paranoid quicksort is \( \Theta(n \log n) \).

(b) Argue that this algorithm will correctly sort the input in time \( c(n \log n) \) for some constant \( c \) with probability greater than \( 1 - \frac{1}{n} \).

We do an analysis extremely similar to the one done in lecture for \textsc{Randomized-Quicksort}. First of all, since partitioning (luckily or unluckily) an array of size \( k \) takes \( k \) units of time, we can “charge” one unit to each element of the array. The total running time \( T \) can thus be expressed as \( T = \sum_e T(e) \) where \( e \) denotes any
element of the array and \( T(e) \) denotes the number of times we tried (successfully or unsuccessfully) to partition an array containing \( e \). Let’s focus on one element \( e \) and consider all the nodes of the recursion tree in which \( e \) appears. At each of these nodes, we will be trying a few (unlucky) partitions before we succeed in partitioning the node; the probability of a lucky partition being \( 3/5 \) and all these trials are independent. Therefore, the analysis done in lecture shows that for \( C \) large enough, we have that

\[
Pr[T(e) \geq C \log_{5/4} n] \leq \frac{1}{n^2}.
\]

This implies that

\[
Pr[T(e) \geq C \log_{5/4} n \text{ for some } e] \leq \frac{1}{n},
\]

or in other words that

\[
Pr[T(e) \leq C \log_{5/4} n \text{ for all } e] \geq 1 - \frac{1}{n},
\]

implying that

\[
Pr[T \leq Cn \log_{5/4} n] \geq 1 - \frac{1}{n}.
\]