Advanced Algorithms, Homework 3

Nathan Stouffer Kevin Browder

due: 17 September 2020

CSCI 432 Problem 3-1

Collaborators: *n/a*

Work in a group of size ≥ 2. Explain your strategy for working in a group.

Answer Our strategy for working in a group: we meet early in the week and work over the problems on scratch paper very informally. Doing this early will give us some extra time if there are any very difficult/time consuming problems Once we are satisfied with our work we do the LaTeX separately on our own time and use Git to collaborate on it. We then meet again before the assignment is due and go over the finished document and make sure everything looks good and we didn't make any mistakes when doing the LaTeX.

CSCI 432 Problem 3-2

Collaborators: *Nathan Stouffer and Kevin Browder*

Your group should make at least five contributions to the Piazza board. A contribution can be either asking a relevant question, responding to another student's question, responding to an instructor's question, or choosing a question from Chapter 1 and attempting to solve it, then describing where you get stuck in answering it.

Answer Our groups contributions are:

1. (3-4, Kevin Browder, and 7:13pm-9/14/2020).

   As we learned in CS Theory a subset sum problem, which doesn't have a weight array is NP-Complete. Does the weight component of this problem change that? We think that it does because computing a maximum seems like it requires computing all the options to know for sure that we have the maximum.

2. (3-3, Nathan Stouffer, 9:34am-9/15/2020).

   Hello everyone,

   Some languages (like java, c, c++ ...) allow any boolean condition to be a test for whether the loop should continue running and also allow any incrementing function. However, the examples that I have seen in this class seem to have a for loop running over a set of integers. Does the real-ram model allow for for loops that have a condition like end ¿ beg with some specified incrementing function?

   The reason I ask is because of hw problem 3-3. The question asks that we write an algorithm for binary search using a for loop instead of recursion. If we must run over a set of integers, we can compute an upper bound on how many iterations the for loop will run so it is possible to create a correct for loop. I also think proving termination would be easier when running over a set of integers. However, the first option would be able to exit the for-loop before reaching the upper bound of iterations. This would not improve worst case run time but it would affect the average case run time. Let me know what we are allowed to do with this model of computation.

   Another idea would be to break out the loop if a certain condition is true. Is this allowable with our model of computation?

3. (Problem 3-4 (response to a question from Dr. Fasy), Nathan Stouffer, and 10:30am-9/15/2020).

   I can see how an answer to W would provide an answer to S. If we set all the weights equal to 0, then W is pretty much just S because we only need to find one subset of X that sums to T and it is guaranteed to be the maximum weight. So I agree that W is at least as difficult as S.

   If I remember things correctly, to show that W is NP-C, we would then want to show that W is a member of NP. In the theory course, that meant giving some Nondeterministic Polynomial Decider for W. But that was when everything was a decision problem. I'm not sure how things change when we switch to the real-ram model/are trying to optimize something. If giving some nondeterministic polynomial machine that solve W suffices to show that W is a member of NP, then the following machines might do that.

   SUBSETWEIGHT:
   Suppose we are given X, W, T.
   1. nondeterministically choose X' a subset of X

2. if (sum the values of X' equals T)
3. weight = weight of X'
4. return weight
5. else
6. return -inf

MAXSUBSETWEIGHT:
Suppose we are given X, W, T
1. weights = SUBSETWEIGHT(X, W, T)
2. return max weights

I'm not really sure if all of this is allowable/fits under the span of the real-ram model. But it's what came to mind.

4. (Problem 3-6 (Response to another students question), Kevin Browder, and 12:12pm-9/17/2020).
I walked through the recursive calls like we did on Tuesday in class and used the Verbatim package in Latex for the indentation and to make it look at little nicer.

5. (Problem 3-3 (response to another student), Nathan Stouffer, and 2:53pm-9/17/2020).
It seems like you can do whatever makes the most sense to you. As Dalton said, the way Prof Fasy mentioned it in class was to use the while loop. But your original suggestion of running a for loop log n times works just as well. It also makes proving termination quite easy, although the invariant is slightly harder to prove.

   Whatever makes the most sense to you is the best way to go about the problem.

CSCI 432 Problem 3-3

Collaborators: *Nathan Stouffer and Kevin Browder*

Give the algorithm for binary search, using a for loop and no recursion.

1. Describe the problem in your own words, including describing what the input and output is.
2. Describe, in paragraph form, the algorithm you propose.
3. Provide this algorithm in the algorithm environment.
4. Use a decrementing function to prove that the loop terminates.
5. What is the loop invariant? Provide the proof.

Answer

1. The problem that binary search solves is returning the location of a target value in an array. The input to the problem must be a sorted array of comparable items paired with a target value of the same type. The output will either by the index of the target value or -1 to flag that the target is not in the array.
2. Our algorithm assumes that the array is sorted. First, we set the index to -1 (assuming that the value is not in the array). Then we iterate $\lceil \log_2(len(A)) \rceil$ times, splitting the array in half each time. We are able to drop the half of the array that does not contain the target value. We are guaranteed to reach the target item within the loop because there are less than $\lceil \log_2(len(A)) \rceil$ splits in half.
3. Here is the algorithm.

```
BinarySearch(A[1..n], targ)
1.  beg ← 1
2.  end ← n
3.  indx ← -1 // assume value is not in array
4.  for 1..⌈log2(len(A))⌉
5.        mid ← ⌊(beg + end)/2⌋
6.        if (A[mid] = targ)
7.              indx ← mid
8.        if (A[mid] > targ)
9.              end ← mid - 1
10.       if (A[mid] < targ)
11.             beg ← mid + 1
12. return indx
```

4. This algorithm at its most basic level only iterates through the array. Every time it loops the array is cut in half. A array cannot be infinite because each index in an array is assigned to a memory location. There is no such thing as infinite memory so therefore the length of an array is finite. Since we are cutting the array in half every iteration the loop will run at most $log_2(len(A))$ times and since the length of A is finite the algorithm must terminate.

5. Our loop invariant is that $arr[beg] \leq targ \leq arr[end]$ (so long as $targ$ is actually in the range of the array). Prior to the for loop, we are know that our loop invariant is true because every element in the array is between $arr[beg]$ and $arr[end]$. Then at each step in the for loop, we either do not edit beg/end or we select new values for beg and end to keep this statement true. So we keep narrowing in on $targ$ and the loop invariant stays true at each step in the for loop.

CSCI 432 Problem 3-4

Collaborators: *Nathan Stouffer and Kevin Browder*

Chapter 2, Problem 1b (Generalized SubsetSum).

1. Describe the problem in your own words, including describing what the input and output is.
2. Describe, in paragraph form, the algorithm you propose.
3. Provide this algorithm in the algorithm environment.
4. What is the runtime of your algorithm?
5. Prove partial correctness (that if your algorithm terminates, it is correct).

Answer

1. For the Generalized SubsetSum problem, our input has two parts. First, we have two equally sized arrays X and W containing positive integers paired with another positive integer T. Each value of X has a corresponding weight in W that can be found at the same index (ie weight for $X[i]$ can be found at $W[i]$). The second part of the input is an array called I (which has length $len(X)$ and begins consisting entirely of 0s) paired with an index i (which starts at 1). If it exists, our task is to find the subset of X that sums to T with the heaviest weight. If no subset of X sums to T, we will return $-\infty$. Our output will either be $-\infty$ or a positve integer (the weight of the heaviest subset that sums to T).
2. At a high level, our algorithm computes the sum of every subset of X. If the sum of the subset is T, then the weight is compared with the other subsets that sum to T. The subset with the highest weight is chosen.

   The array I contains activations for whether to include the i^{th} element of X in the subset (0 means don't include, 1 means include). The index i denotes the first index of I that has not been yet been called.
3. Here is the algorithm.

```
SubsetSum(X, W, T, I, i)
1.  if (i > len(I))
2.      return −∞
3.  if (Sum(X, I) > T)
4.      return −∞
5.  if (Sum(X, I) = T)
6.      return Weight(W, I)
7.  skip ← SubsetSum(X, W, T, I, i + 1)
8.  I[i] ← 1
9.  take ← SubsetSum(X, W, T, I, i + 1)
10. return max(skip, take)

Sum(X, I)
1.  sum ← 0
2.  for i in 1..len(X)
3.      sum ← sum + X[i] ∗ I[i]
4.  return sum

Weight(W, I)
1.  weight ← 0
2.  for i in 1..len(W)
3.      weight ← weight + W[i] ∗ I[i]
4.  return weight
```

4. Towards giving the runtime of our algorithm, we start by giving the run times of Sum and Weight. Let n be the length of X (which matches the lengths of W and I). Both Sum and Weight run in $\Theta(n)$ (since they have constant time operations that run n times). Now we give the worst case recurrence relation for SubsetSum: $T(n) = \Theta(n) + \Theta(n) + T(n-1) + O(1) + T(n-1) = 2 * T(n-1) + \Theta(n)$. We can then say that $T(n) = O(2^n)$.

5. First we set up the recursive invariant: The initialization step is checking to see if i is greater than the length of I or if the sum of X and I. If either of these are true we return $-\infty$. The maintenance step is checking if the sum of X and I. This means that every time we make the recursive call we move up the X array. Using strong induction, we have the base case were the sum of X and I is equal to T. Our inductive hypothesis: We assume that the recursive invariant holds true for all previous recursive steps. As our inductive step, since we assume the inductive hypothesis, we then check both variations of I (0, 1). So, by induction, we have proven that the algorithm is correct.

CSCI 432 Problem 3-5

Collaborators: *Nathan Stouffer and Kevin Browder*

Describe two different data structures that you can use to store a graph. Please give a complete description (i.e., a response of "an array" will not suffice).

Answer

1. Array: Use a two dimensional array of nodes to create an adjacency matrix. Edges are represented by non-zero values in the array. The size of the matrix is a square where the width and height are the number of nodes.
   Example: 0–1–2
   Adjacency Matrix:
   $\{\{0, 1, 0\}$
   $\{1, 0, 1\}$
   $\{0, 1, 0\}\}$

2. Adjacency List: Use an array of lists where each index of the array is a vertex. At each index in the array (array[i]) there is a list of nodes that are adjacent to the ith node.
   Example: 0–1–2
   {[1],
   [[0], [2]],
   [1]}

3. Relational Database: Have a node table and an edge table. The node table has a node ID and the edges table consists of two rows, A and B. Which contain the start and end node ID's of an edge respectively. This works best for directed graphs. Example: 0–1–2

ID	Name	Feature
0	Node 0	A
1	Node 1	A
2	Node 2	B

A	B
0	1
1	0
1	2
2	1

CSCI 432 Problem 3-6

Collaborators: *Nathan Stouffer and Kevin Browder*

Walk through the exponential time Longest Increasing Subsequence (LIS) algorithm on page 108 for the input: $[1, 7, 6, 11, 3, 11]$.

Walk through the algorithm using the Dynamic Programming algorithm present in Section 3.6.1

Answer

1. The LIS is 3.

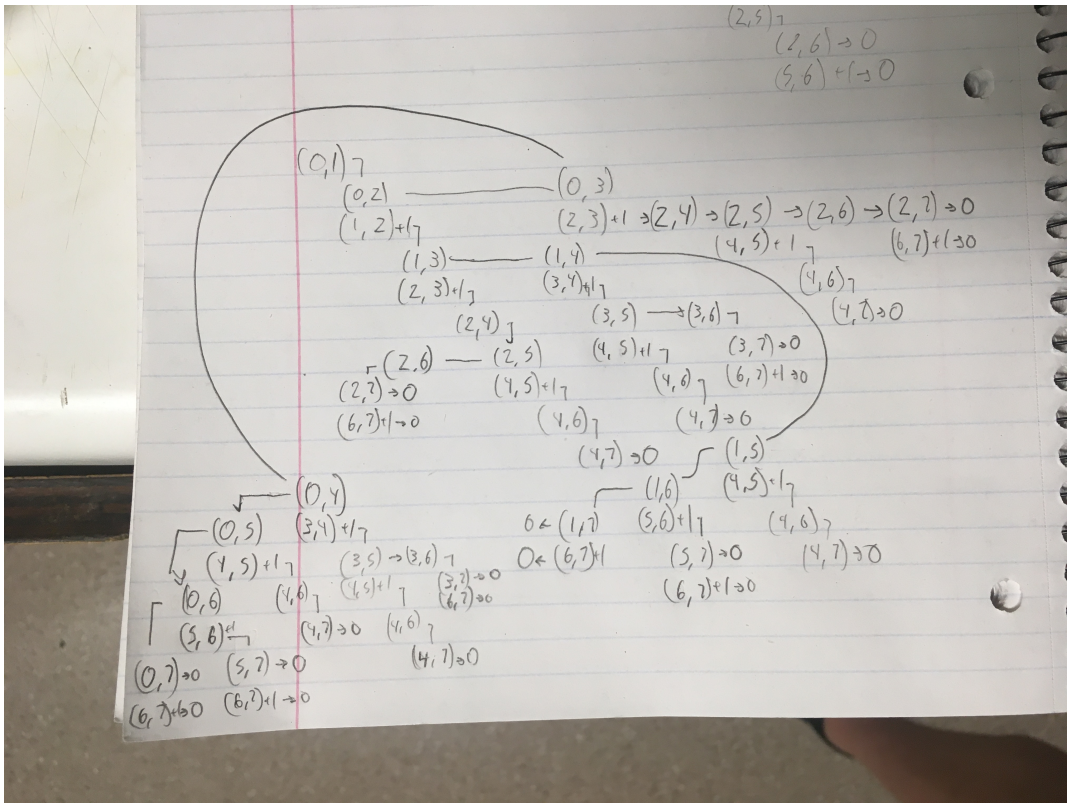

Figure 1: Problem 6 Part 1

2. Here is the output for the dynamic programming solution

{1,1,1,1,1,1}
{1,2,1,1,1,1}
{1,2,2,1,1,1}
{1,2,2,2,1,1}
{1,2,2,3,1,1}
{1,2,2,3,2,2}
{1,2,2,3,2,3}

CSCI 432 Problem 3-7

Collaborators: *Nathan Stouffer and Kevin Browder*

What is the closed form of the following recurrence relations? Use the master theorem to justify your answers:

1. $T(n) = 16T(n/4) + \Theta(n)$
2. $T(n) = 2T(n/2) + n \log n$
3. $T(n) = 6T(n/3) + n^2 \log n$
4. $T(n) = 4T(n/2) + n^2$
5. $T(n) = 9T(n/3) + n$

Note: we assume that $T(1) = \Theta(1)$ whenever it is not explicitly given.

Answer

1. Our recurrence relation is $T(n) = 16T(n/4)+\Theta(n)$. In the context of the master theorem, $f(n) = \Theta(n)$ and $n^{\log_b a} = n^{\log_4 16} = n^2$. We claim this fits case 1 of the master theorem. To show this, we must show that (for some $\epsilon > 0$) $f(n) = O(n^{\log_b a - \epsilon}) \iff \Theta(n) = O(n^{2-\epsilon})$.

   Choose $\epsilon = 1$ and then we have $\Theta(n) = O(n)$ which is true if and only if $O(n) = O(n)$ and $n = O(n)$ (this is by definition of big-theta). The first equality is trivially true and take $n_0 = 1$ and $c = 1$ to show that the second equality holds.

   Thus, $T(n)$ satisfies case 1 of the master theorem and $T(n) = \Theta(n^2)$.

2. Our recurrence relation is $T(n) = 2T(n/2) + n \log n$. In the context of the master theorem, $f(n) = n \log n$ and $n^{\log_b a} = n^{\log_2 2} = n^1 = n$. We claim this fits case 3 of the master theorem. To show this, we must show that $f(n) = \Omega(n^{\log_b a + \epsilon})$ for $\epsilon > 0$ and $af(n/b) \leq c_1 f(n)$ for some $c_1 > 1$.

   We first show that $f(n) = \Omega(n^{\log_b a+\epsilon}) \iff n \log n = \Omega(n^{1+\epsilon}) \iff n^{1+\epsilon} = O(n \log n)$. Choose $\epsilon = 0.1$ So we must show $n^{1.1} = O(n \log n)$, that is $n^{1.1} \leq cn \log n \iff n^{0.1} \leq c \log n \iff n^{0.1} \leq \log n^c$. Choose $c = 1$ and then it is true that $n \log n = \Omega(n^{1.1})$.

   We now must show that $2(n/2)\log(n/2) \leq c_1 n \log n \iff \log(n/2) \leq \log n^{c_1} \iff n/2 \leq n^{c_1}$ then just choose $c_1 = 2$ and the statement is true.

   Thus, $T(n)$ satisfies case 3 of the master theorem and $T(n) = \Theta(n \log n)$.

3. Our recurrence relation is $T(n) = 6T(n/3) + n^2 \log n$. In the context of the master theorem, $f(n) = n^2 \log n$ and $n^{\log_b a} = n^{\log_3 6}$ (note that $\log_3 6 \in (1, 2)$). We claim this fits case 3 of the master theorem. To show this, we must show that $f(n) = \Omega(n^{\log_b a+\epsilon})$ for $\epsilon > 0$ and $af(n/b) \leq c_1 f(n)$ for some $c_1 > 1$.

We first show that $f(n) = \Omega(n^{\log_b a+\epsilon}) \iff n^2 \log n = \Omega(n^{\log_3 6+\epsilon}) \iff n^{\log_3 6+\epsilon} = O(n^2 \log n)$. Choose ϵ such that $\log_3 6 + \epsilon = 2$. So we must show $n^2 = O(n^2 \log n)$. Choose $n_0 = 2$ and $c_0 = 1$, then we verify that $n^2 \leq 1n^2 \log n$ for all $n \geq 2$. Consider, $n^2 \leq 1n^2 \log n \iff 1 \leq \log n$ which is certainly true for for all $n \geq 2$.

We must now show that $6f(n/3) \leq c_1 f(n)$ for some c_1 and sufficiently large n. Choose $c_1 = 2$.

$$6f(n/3) \leq c_1 f(n) \iff 6(n/3)^2 \log(n/3) \leq 2n^2 \log(n) \iff (2/3)\log(n/3) \leq 2\log(n)$$

Using properties of logarithms, we can equivalently say that $\log((n/3)^{2/3}) \leq \log(n^2)$ which is true exactly when $(n/3)^{2/3} \leq n^2 \iff n/3 \leq n^3 \iff 1/3 \leq n^2$ which is true for $n \geq 1/\sqrt{3}$.

Therefore, $T(n)$ satisfies case 3 of the master theorem and $T(n) = \Theta(n^2 \log n)$.

4. Our recurrence relation is $T(n) = 4T(n/2) + n^2$. In the context of the master theorem, $f(n) = n^2$ and $n^{\log_b a} = n^{\log_2 4} = n^2$. We claim this satisfies case 2 of the master theorem. To show this, we must show that $f(n) = n^2 = \Theta(n^{\log_b a}) = \Theta(n^2)$.

   To show that $n^2 = \Theta(n^2)$, we must show that $n^2 = O(n^2)$ (showing the other case is symmetric so there is only the condition). Choose $n_0 = 1$ and $c = 1$, then $n^2 \leq n^2$ for all $n > 1$ and we conclude that $n^2 = O(n^2)$.

   So $T(n)$ satisfies case 2 of the master theorem and $T(n) = \Theta(n^2 \log n)$.

5. Our recurrence relation is $T(n) = 9T(n/3) + n$. In the context of the master theorem, $f(n) = n$ and $n^{\log_b a} = n^{\log_3 9} = n^2$. We claim this satisfies case 1 of the master theorem. To show this, we must show that $f(n) = n = O(n^{\log_b a-\epsilon}) = O(n^{2-\epsilon})$ for some $\epsilon > 0$.

   Let $\epsilon = 1$, then we must show that $n = O(n)$. Choose $c = 1$ and $n_0 = 1$. Then $n \leq n \iff 1 \leq 1$ which is certainly true for all $n \geq 1 = n_0$. So $T(n)$ satisfies case 1 of the master theorem and $T(n) = \Theta(n^2)$.

CSCI 432 Problem 3-8

Collaborators: *Nathan Stouffer and Kevin Browder*

The skyline problem: You are in Camden, NJ waiting for the ferry across the river to get into Philadelphia, and are looking at the skyline. You take a photo, and notice that each building has the silhouette of a rectangle. Suppose you represent each building b as a triple $(x_b^{(1)}, x_b^{(2)}, y_b)$, where the building can be seen from $x_b^{(1)}$ to $x_b^{(2)}$ horizontally and has a height of y_b. Let `rect(b)` be the set of points inside this rectangle (including the boundary). Let `buildings` be a set of n such triples representing buildings. Design an algorithm that takes `buildings` as input, and returns the skyline, where the skyline is a sequence of (x, y) coordinates defining $\cup_{b \in \texttt{buildings}} \texttt{rect}(b)$. The output should start with $(\min_b x_b^{(1)}, 0)$ and end with $(\max_b x_b^{(2)}, 0)$.

1. Describe the problem in your own words, including describing what the input and output is.
2. Describe, in paragraph form, the algorithm you propose.
3. Provide this algorithm in the algorithm environment.
4. What is the runtime of your algorithm? If you do not know, either give the tightest bounds you know, or provide a decrementing function to show that it does terminate.
5. Prove partial correctness (that if your algorithm terminates, it is correct).

Answer

1. As input, we expect a set of triples that represent the buildings of the skyline. Each contains the inclusive bounds for the range of x values (these are the first two values in the triple) that the building occupies in the sky and the height of the building as the third element. We index each of these triples as arrays. The problem is to return a sequence of xy pairs that represent the skyline by drawing straight lines between consecutive pairs of points. We will return the sequence of xy pairs as a list.
2. Our algorithm makes use of a few subroutines. First there are two sorting functions that we do not define because sorting is a well explored problem that we know operates in $\Theta(n \log n)$ time. These two functions are SortByX1 and SortByX2. Both functions expect a set of triples as their input and return a sorted list of the same triples (the key for sorting depends on which function is called).

   Aside from the Skyline algorith, we also define two helper functions Height and NextX. Height is a function that expects a sorted (by $x_b^{(1)}$) list of triples and a query value x. It returns that largest height of the buildings at that x value (excluding all $x_b^{(2)}$ values). The intention is to find the height of the skyline going forward from x. NextX expects a sorted (by $x_b^{(1)}$) list of triples, the largest $x_b^{(2)}$, as well as a pair (x, y) where x is some value and y is the corresponding height of the skyline at x. NextX returns the smallest x' larger than x where $\text{Height}(x') > y$.

   Now we describe Skyline. Skyline takes in a set of triples representing buildings and returns a set of points representing the skyline of the buildings. Skyline will start by sorting the triples according to the left endpoint of each triple and then selecting the smallest left endpoint as our starting point.

The algorithm then enters a while loop that runs until the largest right endpoint has been reached. At each iteration, the next x value is selected and a horizontal line is drawn from the last point. Then a vertical line is drawn up to the height at the new x. After the while loop terminates, the largest right endpoint is added to the path defining the skyline.

3. Here is our algorithm.

```
SKYLINE(buildings)
1.  sorted ← SORTBYX1(buildings)        // buildings sorted by x_b^(1)
2.  lastx ← SORTBYX2(buildings)[len(buildings)]        // get the largest x_b^(2)
3.  x ← buildings[1][1]
4.  y ← HEIGHT(sorted, 0)
5.  skyline ← (x, 0) ∪ (x, y)        // list to store the skyline (we use union to represent appending)
6.  while (x < lastx)
7.      newx ← NEXTX(sorted, lastx, x, y)        // get the x at the next intersection in the skyline
8.      skyline ← skyline ∪ (newx, y)        // draw a horizontal line
9.      newy ← HEIGHT(sorted, newx)        // get the y value for the skyline at newx
10.     skyline ← skyline ∪ (newx, newy)        // draw a vertical line
11.     x ← newx
12.     y ← newy
13. skyline ← ∪ (lastx, 0) // since the loop has terminated, go straight down
14. return skyline


HEIGHT(sorted, x)
1.  height ← 0
2.  for i in 1..len(sorted)
3.      if (sorted[i][1] ≤ x < sorted[i][2])
4.          if (sorted[i][3] > height)
5.              height ← sorted[i][3]


NEXTX(sorted, lastx, x, y)
1.  next ← lastx
2.  for i in 1..len(sorted)
3.      candidate ← sorted[i][1]
4.      if (x ≤ candidate)        // check if building starts after x
5.          if (candidate < next)        // check if buiding is closer than current next
6.              if (HEIGHT(sorted, candidate) > HEIGHT(sorted, next))        // check if candidate is taller
7.                  next ← candidate
8.  return candidate
```

4. We now give the runtime of our algorithm. Let n be the number of triples in the input. Each sorting step takes $\Theta(n \log n)$ time. Then the while loop runs a maximum of $2n$ times (2 for each building if there is no overlap). Each of the helper functions HEIGHT and NEXTX run in $O(n)$ time. Thus the total run time is $T(n) = 2\Theta(n \log n) + 2n + O(n) = O(n \log n)$

5. We now prove partial correctness. Assume that our algorithm terminates. NEXTX searches the set of buildings for the next x value that changes the skyline. Then, SKYLINE just asks NEXTX for the next value of x where the skyline changes height and draws lines accordingly. Thus the algorithm sketches out the skyline as it runs.