Problem Set 2 Solutions

Problem 2-1. Multiplying the Klingon way

(a) The pseudocode for \textsc{WorfMultiply} is shown below:

\begin{verbatim}
WorfMultiply(a, b)
left ← a
right ← b
mult ← 0
while (left ≥ 1) do
    if (left ≡ 1 mod 2) then
        mult ← mult + right
    left ← left div 2
    right ← right * 2
return mult
\end{verbatim}

(b) The algorithm uses the following identity to compute the product:

\[ a \ast b = (a_0b + a_12b + a_22^2b + \ldots + a_{n-1}2^{n-1}b) \]

This insight lets us “guess” the following loop invariant for our algorithm:

\[ mult = a \ast b - left \ast right \]

The invariant is clearly true before the loop is entered. Assume the invariant holds before some iteration and let \( mult', left', right' \) be the variables after the iteration is over.

Case I: \( (left \equiv 1 \mod 2) \)
In this case \( left' = (left - 1)/2 \Rightarrow right' = right \ast 2 mult' = mult + right \)
So, \( ab - left' \ast right' = ab - left \ast right + right = mult + right = mult' \)

Case II: \( (left \equiv 0 \mod 2) \)
In this case \( left' = left/2 \Rightarrow right' = right \ast 2 mult' = mult \)
So, \( ab - left' \ast right' = ab - left \ast right = mult = mult' \)

This proves the invariant. The algorithm terminates because \( left \) decreases strictly at the end of each iteration and must fall to 0 eventually. At the end of the loop, \( left = 0 \); this together with the invariant implies that \( mult = a \ast b \) which completes the proof of correctness for the algorithm.
Problem 2-2. More multiplication

(a) In the worst case, \( b \) could be as large as \( 2^n - 1 \). Since addition takes \( \Theta(n) \) bitwise operations, the overall worst case time complexity of this algorithm is \( \Theta(n2^n) \). Clearly, this algorithm is not practically feasible.

(b) In algorithm WORFMULTIPLY, the while loop is executed \( n \) times. The costliest operation inside the while loop is the addition which takes \( \Theta(n) \) bit operations (the multiplication and division by 2 can be achieved by left and right shifts respectively). Thus the overall time complexity of Worf’s algorithm is \( \Theta(n^2) \).

(c) The divide and conquer technique for this problem is similar to polynomial multiplication. Divide \( a \) and \( b \) into \( n/2 \) bit numbers, \( a_1, a_0 \) and \( b_1, b_0 \) respectively. Compute the products \( m_1 = (a_0 + a_1)(b_0 + b_1), m_2 = a_1b_1, m_3 = a_0b_0 \) recursively and then merge them using the identity \( ab = m_22^n + (m_1 - m_2 - m_3)2^{n/2} + m_3 \).

The recurrence for the running time is \( T(n) = 3T(n/2) + \Theta(n) \) which yields \( T(n) = \Theta(n^{\log_23}) \).

Problem 2-3. Searching an unsorted array

(a) For the deterministic algorithm, number of comparisons required in the worst case is \( n - k + 1 \) if \( k \geq 1 \) and \( n \) if \( k = 0 \).

(b) \( \bullet \) \( k \geq 1 \)

The randomized algorithm looks at successively smaller intervals of elements till it finds \( x \). Let \( X \) be a random variable denoting the number of elements searched till \( x \) was found. Then,

\[
E[X] = \sum_{i=1}^{n-k+1} \Pr[X \geq i] = \sum_{i=1}^{n-k+1} \frac{n-k}{i-1} \binom{n-k}{i-1} = \sum_{i=1}^{n-k+1} \frac{\binom{n-i+1}{k}}{\binom{n}{k}} = \frac{n+1}{k+1}
\]

Note that the second step follows from the fact that there are \( \binom{n-k}{i-1} \) ways of choosing the first \( i - 1 \) elements and the number of ways in which none of the desired copies shows up is \( \binom{n-k}{i-1} \). The rest of the steps are just algebraic manipulation.

Thus, the expected running time is \( \frac{n+1}{k+1} \).

Alternatively, one can use another equivalent expression for the expectation, namely,

\[
E[X] = \sum_{i=1}^{n-k+1} i \Pr[X = i] = \sum_{i=1}^{n-k+1} i \binom{n-k}{i-1} \frac{k}{n-i+1} = k \sum_{i=1}^{n-k+1} \frac{i}{n-i+1} \binom{n-i+1}{k}
\]
Handout 12: Problem Set 2 Solutions

The last sum again evaluates to \( \frac{n+1}{k+1} \) (Why?).

- \( k = 0 \)

Clearly, the randomized algorithm will have to look at all the elements. Therefore, the running time in this case is \( n \).

Note that if \( k = \omega(1) \), then the randomized algorithm is asymptotically better.

Problem 2-4. The skyline problem

One \( O(n \lg n) \) solution to the skyline problem is a simple divide-and-conquer algorithm. Divide the city arbitrarily into two subcities, each with \( n/2 \) buildings, and recursively find a skyline for each subcity. Finally, merge the two subskylines in linear time. For the base case of the recursion, we simply produce a skyline with zero height everywhere except between the left and right coordinates of the building.

Once we decide to use divide and conquer, the only difficulty left is how to merge two skylines in linear time. The solution is reminiscent of the \textsc{merge} procedure used by \textsc{merge-sort}. Assume that the \( x \)-coordinates of a skyline \( A \) are stored in array \( A_x \) and the corresponding heights in \( A_h \). Call the two subskylines to be merged \( A \) and \( B \), and let \( C \) be the new merged skyline which we are trying to produce. Scan \( A_x \) and \( B_x \) from left to right, comparing \( x \) values. At each step, focus on the array with the smaller \( x \) value. Imagine that at some step, the smaller \( x \) value resides in \( A \). Then the current height in \( C_h \) is set to the max of the current height in \( A_h \) and the previous height in \( B_h \).

The pseudocode for \textsc{skyline-merge}(\( A, n, B, m \)) which merges two subskyline arrays \( A \) and \( B \), of lengths \( n \) and \( m \) respectively, in \( \Theta(n + m) \) time, is shown on the next page. This pseudocode uses the boundary condition that \( C_h[0] \) is not equal to any other value. This convention is only to make the behavior of line 17 well defined the first time through the loop.

A formal proof of correctness with a loop invariant would be somewhat clumsy for this code. Therefore, we give a more informal correctness argument. Notice that after every iteration of the loop, \( C_h[k-1] \) contains a height which starts at \( x = C_x[k-1] \) and whose endpoint is not yet determined. Furthermore, \( i \) and \( j \) have the smallest values possible, subject to the constraint that the \( x \)-coordinates \( A_x[i] \) and \( B_x[j] \) are both greater than or equal to \( C_x[k-1] \). Finally, \( A_x[i] \) and \( B_x[j] \) both represent \( x \)-coordinates at which the height of their respective subskylines change. It follows that the height of the skyline we are producing will be \( C_h[k-1] \) until \( x \geq \min(A[i], B[j]) \). At that point, we must consider changing the height in the skyline we are producing. If \( A_x[i] = B_x[j] \) then it should be obvious that the new height in \( C \) should be \( \max(A_h[i], B_h[j]) \). Otherwise, assume without loss of generality that \( A_x[i] < B_x[j] \). In this case, the height of skyline \( A \) at \( x = A_x[i] \) changes to \( A_h[i] \), while the height of skyline \( B \) at \( x = A_x[i] \) is still \( B_x[j-1] \). Thus, the new height in \( C \) at \( x = A[i] \) should be \( \max(A_h[i], B_h[j-1]) \).
\textbf{Skyline-Merge}(A, n, B, m)

1. \texttt{i} \leftarrow 1
2. \texttt{j} \leftarrow 1
3. \texttt{k} \leftarrow 1
4. \textbf{while} (i \leq n) \textbf{or} (j \leq m) \textbf{do}
5. \quad \texttt{x} \leftarrow \text{min}(A_x[i], B_x[j])
6. \quad \textbf{if} A[i] < B[j] \textbf{then}
7. \quad \quad \texttt{max} \leftarrow \text{max}(A_h[i], B_h[j - 1])
8. \quad \quad \texttt{i} \leftarrow \texttt{i} + 1
9. \quad \textbf{else}
10. \quad \quad \textbf{if} A[i] > B[j] \textbf{then}
11. \quad \quad \quad \texttt{max} \leftarrow \text{max}(A_h[i - 1], B_h[j])
12. \quad \quad \quad \texttt{j} \leftarrow \texttt{j} + 1
13. \quad \quad \textbf{else}
14. \quad \quad \quad \texttt{max} \leftarrow \text{max}(A_h[i], B_h[j])
15. \quad \quad \quad \texttt{i} \leftarrow \texttt{i} + 1
16. \quad \quad \quad \texttt{j} \leftarrow \texttt{j} + 1
17. \quad \textbf{if} \texttt{max} \neq C_h[k - 1] \textbf{then} \quad \triangleright \text{This prevents us from placing a redundant height}
18. \quad \quad \texttt{C}_x[k] \leftarrow \texttt{x} \quad \triangleright \text{in } C \text{ by “changing” to a new height which}
19. \quad \quad \texttt{C}_h[k] \leftarrow \text{max} \quad \triangleright \text{happens to be the current height in } C.
20. \texttt{k} \leftarrow \texttt{k} + 1
21. \textbf{return } C

Note that this argument assumes that the x-coordinates in skylines \( A \) and \( B \) are in sorted order. In other words, \( A_x[1] \leq A_x[2] \leq \ldots \leq A_x[n] \), and \( B_x[1] \leq B_x[2] \leq \ldots \leq B_x[m] \). This assumption is valid because \( C \) will have this property (if \( A \) and \( B \) do) when \textbf{Skyline-Merge} terminates, and this requirement is obviously met in the base case of a city containing exactly one building.

The recurrence for this algorithm is \( T(n) = 2T(n/2) + \Theta(n) \) and the solution to this familiar recurrence is \( T(n) = \Theta(n \lg n) \).