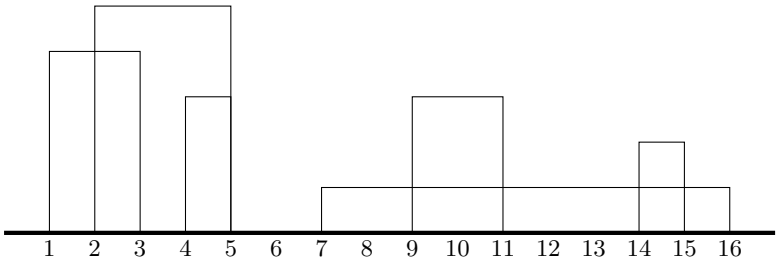
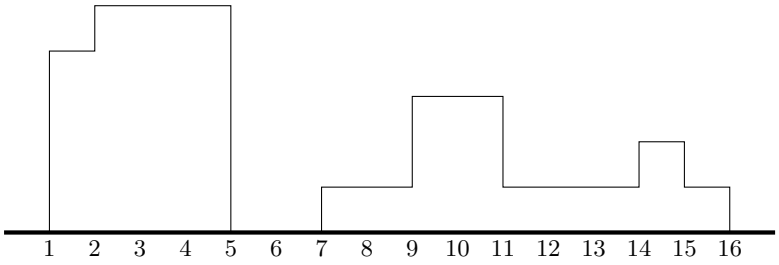
ASSIGNMENT 3: Solutions CMSC 451 (Spring 2016)

1. *Drawing a skyline.*

Suppose you are given a description of rectangular, possibly overlapping buildings and you would like to draw the corresponding skyline with no overlapping lines. More concretely, a *skyline* is a list of tuples, each consisting of alternating x coordinates and the heights connecting them. For ease of reading, we place bars over the entries representing heights. For example, the skyline $(1, \overline{4}, 3), (2, \overline{5}, 5), (4, \overline{3}, 5), (7, \overline{1}, 16), (9, \overline{3}, 11), (14, \overline{2}, 15)$ can be drawn as follows:



The desired output in this case is described by the drawing



with the corresponding skyline $(1, \overline{4}, 2, \overline{5}, 5), (7, \overline{1}, 9, \overline{3}, 11, \overline{1}, 14, \overline{2}, 15, \overline{1}, 16)$. Design a divide-and-conquer algorithm that, when given a skyline for n separate buildings (i.e., a list of n 3-tuples, not necessarily occurring in any particular order), produces the corresponding skyline of the drawing with overlapping lines removed. Prove that your algorithm is correct and analyze its running time. For full credit, your algorithm should run in time $O(n \log n)$.

Solution: The skyline can be found by a natural divide-and-conquer algorithm similar to merge sort. Let B denote the list of input buildings, where each element of B is a 3-tuple representing a building. The following algorithm finds the skyline of buildings $B[low], B[low+1], \ldots, B[high]$:

FindSkyline$(B, low, high)$:

If $(low = high)$ // single building
    Return $B[low]$
$mid = \lfloor (low + high)/2 \rfloor$
$sk1 =$ FindSkyline(B, low, mid)
$sk2 =$ FindSkyline$(B, mid + 1, high)$
Return Merge$(sk1, sk2)$

The Merge subroutine to merge two skylines is very similar to that for merge sort. In particular, the x coordinates of the two skylines are merged as per usual to maintain the sorted order. The height of the resulting skyline between two given x coordinates is computed as the maximum of the height specified by the two skylines. Algorithm 1 shows the pseudocode.

Merge($sk1, sk2$*)*

$sk \leftarrow \emptyset$;

Let $sk1 = (a_1, h_1, a_2, h_2, \ldots, a_p)$ and $sk2 = (b_1, l_1, b_2, l_2, \ldots, b_q)$ where a_i, b_i indicate the x coordinates and h_i, l_i indicate the height of the skyline between two adjacent x coordinates;

Let $h_p = h_q = curr\_l = curr\_h = 0$;

Initialize $i = j = k = 1$;

while $i \leq p$ *and* $j \leq q$ **do**

  if $a_i < b_j$ **then**

    $c_k = a_i$;

    $g_k = \max(curr\_l, h_i)$;

    $curr\_h = h_i$;

    $i++, k++$;

  else

    $c_k = b_j$;

    $g_k = \max(curr\_h, l_j)$;

    $curr\_l = l_j$;

    $j++, k++$;

while $i \leq p$ **do**

  $c_k = a_i, g_k = h_i$;

  $i++, k++$;

while $j \leq q$ **do**

  $c_k = b_j, g_k = l_j$;

  $j++, k++$;

$sk \leftarrow (c_1, g_1, \ldots, c_k, g_k)$

Algorithm 1: Subroutine for merging two skylines

Proof of Correctness: Given two skylines $sk1$ and $sk2$ where the x coordinates are in sorted order, it can be easily verified that the Merge subroutine correctly yields the skyline of all the buildings in $sk1$ and $sk2$. Note that at any given x coordinate, the height of the resulting skyline is the maximum of the height of the two skylines at that coordinate. Consequently, any building that that is spanned by either $sk1$ of $sk2$ must also be spanned by the new skyline.

Running Time: Since the Merge subroutine scans through each skyline once and performs only constant amount of work per step, the Merge subroutine runs in time $O(n)$. Thus, we have the following recurrence for the running time $T(n)$ of the FindSkyline algorithm:

$$T(n) = 2T(n/2) + O(n).$$

Solving this recurrence yields $T(n) = O(n \log n)$.

2. *Fast Fourier transform details.*

   (a) The *discrete Fourier transform* modulo d is the $d \times d$ matrix with entries $F_{j,k} = \frac{1}{\sqrt{d}}\omega_d^{jk}$ for $j, k \in \{0, 1, \ldots, d-1\}$, where $\omega_d = e^{2\pi i/d}$. Show that the discrete Fourier transform is *unitary*, i.e., its conjugate transpose is its inverse.

   Solution:

   Claim: $\sum_{a=0}^{d-1} \omega_d^{ab} = \begin{cases} d & \text{if } b = 0 \\ 0 & \text{if } b = 1, 2, \ldots, d-1 \end{cases}$

   Proof. If $b = 0$, we have $\omega_d^{ab} = 1$ and the claim follows. On the other hand, if $b \neq 0$, let

   $$X = \sum_{a=0}^{d-1} \omega_d^{ab} = 1 + \omega_d^b + \omega_d^{2b} + \omega_d^{3b} + \ldots + \omega_d^{(d-1)b}$$

   But since $\omega_d^d = 1$, we have

   $$\begin{aligned} X &= \omega_d^b + \omega_d^{2b} + \omega_d^{3b} + \ldots + \omega_d^{(d-1)b} + \omega_d^{db} \\ &= \omega_d^b \left(1 + \omega_d^b + \omega_d^{2b} + \omega_d^{3b} + \ldots + \omega_d^{(d-1)b}\right) = \omega_d^b X. \end{aligned}$$

   Since $\omega_d^b \neq 1$, this implies that $X = 0$. □

   Note that one can also do the proof for the case $b \neq 0$ using the well-known formula for geometric summation.

   Let F^* denote the conjugate transpose of matrix F and let $A = FF^*$. We have

   $$A_{j,k} = \sum_{l=0}^{d-1} F_{j,l} F^*_{l,k} = \sum_{l=0}^{d-1} \frac{1}{d} \omega_d^{jl} \omega_d^{-kl} = \frac{1}{d} \sum_{l=0}^{d-1} \omega_d^{l(j-k)}.$$

   Applying the above claim, we get the desired result, namely

   $$A_{j,k} = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{if } j \neq k. \end{cases}$$

(b) In class, we discussed a polynomial multiplication algorithm based on the fast Fourier transform. This algorithm takes as input the coefficients of two polynomials of degree $n-1$ and outputs the coefficients of their product, which is a polynomial of degree $2n-2$. Write pseudocode for this algorithm, and show that your implementation runs in time $O(n \log n)$.

Solution: Let $a = (a_0, a_1, \ldots, a_{n-1})$ and $b = (b_0, b_1, \ldots, b_{n-1})$ denote the coefficients of the two input polynomials $A(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ and $B(x) = b_0 + b_1 x + \cdots + b_{n-1}x^{n-1}$, respectively.

If n is not a power of 2, then let k be the smallest integer such that $2n > n' = 2^k > n$. We now consider A and B to be polynomials of degree $n'-1$ by setting the coefficients $a_j = b_j = 0$ for all $n \leq j \leq n'-1$. As a result, for the rest of this algorithm, we can assume without loss of generality that n is a power of 2.

Algorithm 2 shows the pseudocode for a divide-and-conquer algorithm that evaluates a polynomial at the d^{th} roots of unity ω_d^j for $j \in \{0, 1, \ldots, d\}$, where d is a power of 2. Algorithm 2 satisfies the recurrence $T(n) = 2T(n/2) + O(n)$ and thus runs in time $O(n \log n)$.

Evaluate(a, d): *// a is the vector of coefficients of polynomial $A(x)$ of degree at most $d-1$*

- **if** $n == 1$ **then**
  - $A(1) = A(-1) = a_0$
- **else**
  - $a_{even} = (a_0, a_2, \ldots, a_{d-2})$;
  - $a_{odd} = (a_1, a_3, \ldots, a_{d-1})$;
  - $(A_{even}(\omega_{d/2}^0), \ldots, A_{even}(\omega_{d/2}^{d/2-1})) = Evaluate(a_{even}, d/2)$;
  - $(A_{odd}(\omega_{d/2}^0), \ldots, A_{odd}(\omega_{d/2}^{d/2-1})) = Evaluate(a_{odd}, d/2)$;
  - **for** $j = 0, 1, \ldots, d-1$ **do**
    - $A(\omega_d^j) = A_{even}(\omega_{d/2}^j) + \omega_d^j A_{odd}(\omega_{d/2}^j)$;
- **return** $(A(\omega_d^0), \ldots, A(\omega_d^{2n-1}))$

Algorithm 2: Evaluating a polynomial at roots of unity

Algorithm 3 then shows the pseudocode to compute the coefficients of the product of polynomials $A(x)$ and $B(x)$. Since Algorithm 3 makes three calls to *Evaluate*() and performs only $O(n)$ work otherwise, the total time complexity remains $O(n \log n)$.

Multiply(a, b): *// a and b are coefficients of polynomials $A(x)$ and $B(x)$ of degree $n-1$, padded with zeros to have degree $2n-1$*

- $(A(\omega_{2n}^0), \ldots, A(\omega_{2n}^{2n-1})) = Evaluate(a, 2n)$;
- $(B(\omega_{2n}^0), \ldots, B(\omega_{2n}^{2n-1})) = Evaluate(b, 2n)$;
- **for** $j = 0, 1, \ldots, 2n-1$ **do**
  - $d_j = C(\omega_{2n}^j) = A(\omega_{2n}^j)B(\omega_{2n}^j)$
- $(D(\omega_{2n}^0), \ldots, D(\omega_{2n}^{2n-1})) = Evaluate(d, 2n)$;
- **for** $j = 0, 1, \ldots, 2n-2$ **do**
  - $c_j = \frac{1}{2n} D(\omega_{2n}^{2n-j})$
- **return** c

Algorithm 3: Computing the coefficients of the product of two polynomials

Observe that Algorithm 3 performs polynomial interpolation by applying polynomial eval-

uation (using the FFT) and then reordering the output to effectively replace ω_{2n} by its complex conjugate.

3. *Maximum ordered ratio in linear time.*

   Suppose you are given as input a sequence of positive numbers $a_1, a_2, \ldots, a_n$ with $n \geq 2$. Your goal is to find the largest ratio between two of these numbers where the numerator occurs after the denominator in the sequence. In other words, you would like to compute

   $$\max\left\{\frac{a_i}{a_j} : i, j \in \{1, 2, \ldots, n\} \text{ with } i > j\right\}.$$

   Use dynamic programming to design a linear-time algorithm for this problem. Prove that your algorithm is correct and that its running time is $O(n)$.

   Solution: For $i \geq 2$, let $OPT[i]$ denote the maximum ordered ratio in the first i numbers $a_1, a_2, \ldots, a_i$.

   We now have two cases: either (i) the i^{th} number a_i is the numerator of the maximum ordered ratio, or (ii) it is not the numerator. If a_i is the numerator, then to get the maximum ordered ratio, the denominator must be $\min_{j<i} a_j$. Consequently, we get the following recurrence:

   $$OPT[i] = \max\left\{OPT[i-1], \frac{a_i}{\min_{j<i} a_j}\right\}.$$

   Since we can maintain $\min_{j<i} a_j$ in constant time for every i, we can compute $OPT[i]$ for all indices i using constant time per index to obtain a total running time of $O(n)$. The detailed pseudocode follows.

   Maximum Ordered Ratio(a):

   $OPT[2] = \frac{a_2}{a_1}$
   $temp = \min(a_1, a_2)$
   For $i = 3, 4, \ldots, n$
   $\quad OPT[i] = \max\left\{OPT[i-1], \frac{a_i}{temp}\right\}$
   $\quad temp = \min(temp, a_i)$
   Return $OPT[n]$

4. *Commuting to optimize profit.*

   Suppose you run a company with two offices, one in Washington and the other in San Francisco. Each week, you must choose where to work, and your choice will affect your profit: in week i, you will make DC_i dollars if you work in Washington or SF_i dollars if you work in San Francisco. While you clearly would like to work in the location with the higher profit, each flight from Washington to San Francisco (or vice versa) costs \$1000. Given the lists $\mathrm{DC}_1, \ldots, \mathrm{DC}_n$ and $\mathrm{SF}_1, \ldots, \mathrm{SF}_n$, your goal is to find a schedule that maximizes your total profit, taking transportation costs into account. Since you live in Washington, your schedule must start and end there.

   (a) An obvious greedy strategy is to always work in the office with the higher profit. Give a (minimal) example showing that this strategy is not always optimal.

   Solution: Consider a small instance with only one week: let $DC_1 = 1000$ and $SF_1 = 2000$. The greedy algorithm will choose to work in SF as it yields higher profit; however, once we

take transportation costs into account, working in SF yields a total profit of 0 (since you must fly from DC to SF and back). On the other hand, working in DC yields a profit of 1000.

(b) Design a polynomial-time algorithm that finds the maximum total profit. Justify your algorithm's correctness and establish its running time. To get full credit for your solution, you should make your running time as small as possible.

Solution: Figure 1 illustrates the problem statement. Solid edges have a cost of 1000 each while dashed edges have a cost of 0. The objective is to find the most profitable path from "start" to "end" where the profit of a path is sum of profits of vertices in the path minus the cost of edges.

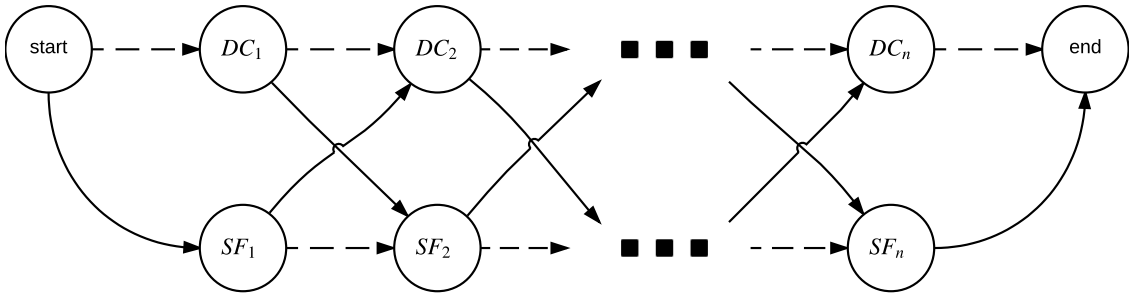


Figure 1: Commuting to maximize profit

Let $A[i]$ denote the profit of the best path that ends at node DC_i and let $B[i]$ denote the profit of the best path that ends at node SF_i. The following recurrence relations follow from the problem statement:

$$\begin{aligned} A[1] &= DC_1 \\ B[1] &= SF_1 - 1000 \\ A[i] &= \max(A[i-1] + DC_i, B[i-1] - 1000 + DC_i) \\ B[i] &= \max(B[i-1] + SF_i, A[i-1] - 1000 + SF_i). \end{aligned}$$

Since the path must end at DC, the profit of the best path is $\max(A[n], B[n] - 1000)$. The algorithm simply evaluates the recursion iteratively and then outputs $\max(A[n], B[n] - 1000)$.

Running Time: Since we perform only constant amount of work to compute each $A[i]$ and $B[i]$, the total running time is $O(n)$.