Solutions to Homework 3: Dynamic Programming and More

Solution 1: The final M and H matrices are shown in Fig. 1 along with the multiplication order.

- $M[1,3]$: The choices are $M[1,1] + M[2,3] + 2 \cdot 2 \cdot 3 = 42$ or $M[1,2] + M[2,3] + 2 \cdot 5 \cdot 3 = 50$. The first is better, and we set $M[1,3] = 29$ and $H[1,3] = 1$.
- $M[2,4]$: The choices are $M[2,2] + M[3,4] + 2 \cdot 5 \cdot 1 = 25$ or $M[2,3] + M[4,4] + 2 \cdot 3 \cdot 1 = 36$. The first is better, and we set $M[2,4] = 25$ and $H[2,4] = 2$.
- $M[1,4]$: The choices are $M[1,1] + M[2,4] + 2 \cdot 2 \cdot 1 = 29$, $M[1,2] + M[3,4] + 2 \cdot 5 \cdot 1 = 45$, or $M[1,3] + M[4,4] + 2 \cdot 3 \cdot 1 = 48$. The first is the best, and we set $M[1,4] = 29$ and $H[1,4] = 1$.

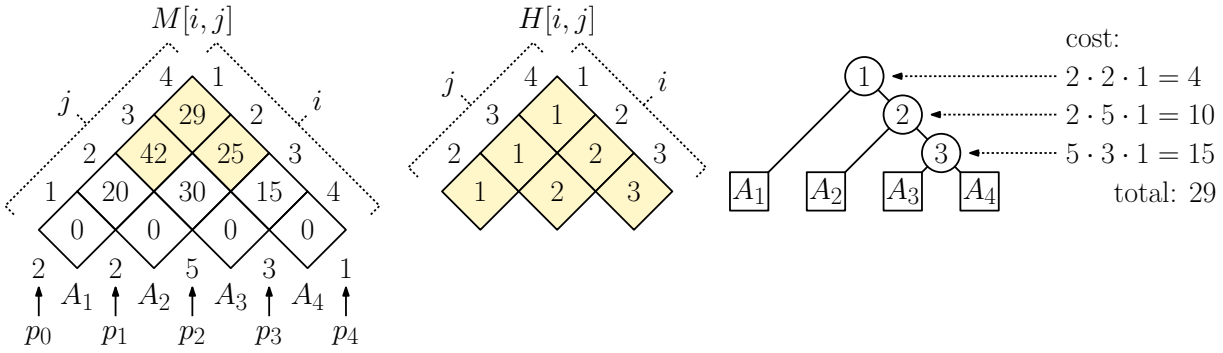


Figure 1: Chain-matrix multiplication.

To get the final multiplication order we see that $H[1,4] = 1$, so we multiply $A_1(A_2 \cdot A_3 \cdot A_4)$. We continue with $H[2,4] = 2$, so this leaves $A_1(A_2(A_3 \cdot A_4))$. The tree is shown in the figure.

Solution 2: For $0 \le j \le n$, let $\text{MP}(j)$ denote the smallest achievable max-penalty for typesetting the first j words. Ultimately, we want to compute $\text{MP}(n)$. For the basis case we have $\text{MP}(0) = 0$, since there is nothing to lay out and hence there is no penalty involved. Let us assume we have access to a utility function $\text{len}(i,j)$, which, for $1 \le i \le j \le n$, returns the sum of word lengths $\sum_{k=1}^{j} w_k$. (See the challenge problem for how this can be done.)

(a) To compute $\text{MP}(j)$ observe that w_j will be the last word on the last line of the layout, but what is the first word of this line? It will be some word w_i, where $1 \le i \le j$ and $\text{len}(i,j) \le L$. Assuming this, the penalty associated with this line will be $L - \text{len}(i,j)$. Assuming that we lay out the remaining words w_1 through w_{i-1} in the best possible manner, the remaining penalty is $\text{MP}(i-1)$. (Observe that the principle of optimality holds here.) The overall penalty is the maximum of these quantities, that is, $\max(L - \text{len}(i,j), \text{MP}(i-1))$. Among the available options, we select the one that produces the lowest value. Thus, we have

$$\text{MP}(j) \;=\; \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{\substack{1 \le i \le j \\ \text{len}(i,j) \le L}} \max(L - \text{len}(i,j), \text{MP}(i-1)) & \text{otherwise.} \end{cases}$$

This solution involves a 1-parameter function, but requires a loop to determine the best split. Another approach (which yields the same running time, but takes more space) is based on a 2-parameter function. For $1 \leq i \leq j \leq n$, let $\text{MP}'(i, j)$ denote the smallest achievable max-penalty for typesetting the first j words, under the assumption that the last line starts *on or before* w_i. The final answer will be $\text{MP}'(n, n)$. To avoid dealing with cases where the total word length exceeds the line length, let us define

$$\text{penalty}(i, j) \;=\; \begin{cases} L - \text{len}(i, j) & \text{if } \text{len}(i, j) \leq L \\ \infty & \text{otherwise} \end{cases}$$

For the basis case, observe that if $i = 1$, then we are putting all the words on a single line, and the overall penalty is $\text{penalty}(i, j)$. Otherwise, $i \geq 2$. Either the last line starts with w_i, in which case the penalty for this last line is $\text{penalty}(i, j)$. The remaining subproblem is to typeset the first $i - 1$ words, which is $\text{MP}'(i - 1, i - 1)$. The overall max-penalty is the maximum of these two. Otherwise, the last line starts earlier than w_i (or equivalently, on or before w_{i-1}), in which case the overall penalty is given by $\text{MP}'(i - 1, j)$. As always, we take the better of these two options.

$$\text{MP}'(i, j) \;=\; \begin{cases} \text{penalty}(i, j) & \text{if } i = 1 \\ \min(\max(\text{penalty}(i, j), \text{MP}'(i - 1, i - 1)), \text{MP}'(i - 1, j)) & \text{otherwise.} \end{cases}$$

Note that once $\text{len}(i, j)$ exceeds L, $\text{MP}'(i, j)$ will be ∞, thus if we were to implement this, we could add this additional check to avoid unnecessary recursive function calls.

(b) We present a memoized implementation in the code block below. (A bottom-up implementation is also quite straightforward). We assume that the array `pred[1..n]` has been precomputed, and we have access to the function $\text{len}(i, j)$. The values are stored in the array $\text{MP}[0..n]$, which is initialized to -1. The initial call is `max-penalty(n)`, which computes the minimum-penalty segmentation of all n words.

Memoized Typesetting with Max Penalty

```
max-penalty(j) {                                  // max-penalty typesetting
    if (MP[j] == -1) {                            // undefined?
        if (j == 0) {                             // basis case
            MP[0] = 0
        } else {
            MP[j] = infinity
            for (i = j downto 1)   {              // try all possible splits
                if (len(i,j) > L) break           // too many words for this line
                thisPenalty = L - len(i,j)        // penalty for last line
                prevPenalty = max-penalty(i-1)    // penalty for prev lines
                MP[j] = min(max(thisPenalty, prevPenalty))
            }
        }
    }
    return MP[j]                                  // return the max penalty
}
```

Clearly, there are $n+1$ values MP[0..n] to be computed, and each one involves minimizing over $O(n)$ possibilities. If we can compute len(i, j) runs in constant time, the overall time is $O(n^2)$.

(c) Let's assume that the array MP[0..n] has been computed in part (b). We know that the last line contains words w_j through w_n, where pred$[n] \le j \le n$. This implies that the previous lines contain the first $j-1$ words, and the penalty for these lines is MP$[j-1]$. Thus, the minimum overall penalty (ignoring the last line) is:

$$\min_{\text{pred}[n] \le j \le n-1} \text{MP}[j-1].$$

Assuming MP has been computed, this takes additional $O(n)$ time to execute. So, the overall time is $O(n^2)$.

Solution 3:

(a) We claim that given n defects, there will be $n+1$ subchips. The proof by (strong) induction on n. The basis ($n = 0$) is trivial (zero defects and one chip). The first cut eliminates one defect. Suppose there are n_1 and n_2 defects remaining in the interiors of the two resulting subrectangles. We have $n_1 + n_2 = n - 1$. By the induction hypothesis, cutting these results in $n_1 + 1$ and $n_2 + 1$ subchips, respectively. Combining these, we have a total of

$$(n_1 + 1) + (n_2 + 1) \;=\; (n_1 + n_2) + 2 \;=\; (n-1) + 2 \;=\; n + 1$$

subchips, as desired.

(b) The subproblems are the possible rectangles within the original chip. Let's assume that we have sorted the x-coordinates of the points in ascending order $x_1 \le \cdots \le x_n$ and the same for the y-coordinates $y_1 \le \cdots \le y_n$. Let's add two additional coordinates to each list to cover the sides of the enclosing square by defining $x_0 = y_0 = 0$ and $x_{n+1} = y_{n+1} = L$ (see Fig. 2(a)). As a convenience, let us assume we have access to a geometry query defectCount(i, i', j, j'), which returns a count of defects in the interior of the rectangle $[x_i, x_{i'}] \times [y_j, y_{j'}]$.

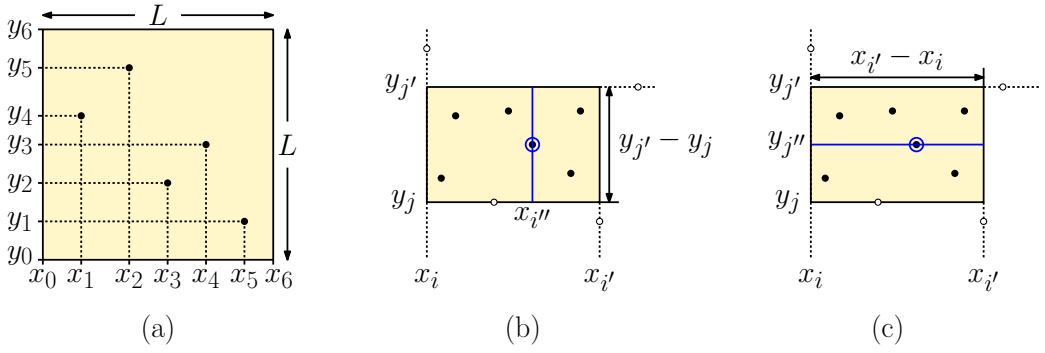


Figure 2: Chip cutting for $n = 5$.

Given $0 \le i < i' \le n+1$, and $0 \le j < j' \le n+1$, define $C(i, i', j, j')$ to be the minimum total cost of any hierarchical cutting of the rectangle $[x_i, x_{i'}] \times [y_j, y_{j'}]$. Ultimately, our objective is to cut the entire chip, that is, $C(0, n+1, 0, n+1)$.

In order to compute $C(i, i', j, j')$ we first observe the basis case that if this subrectangle has no defects in its interior, that is, $\mathsf{defectCount}(i, i', j, j') = 0$, then it does not need to be cut, and hence its cost is zero. (Another possible basis case would be when $i' = i + 1$ or $j' = j'$, but we assert that before we get to this point, we will encounter the defect-count basis case, since such a rectangle cannot contain any defects in its interior.)

Otherwise, we will need to apply either a vertical or horizontal cut.

- We can make a vertical cut between x_i and $x_{i'}$, by considering all cuts through $x_{i''}$, where $i < i'' < i'$. This generates a cutting cost of $y_{j'} - y_j$ and yields two subrectangles whose total cost we compute recursively as $C(i, i'', j, j') + C(i'', i', j, j')$ (see Fig. 2(b)). Thus, we have

$$\mathsf{cost}_x(i, i', j, j') \;=\; (y_{j'} - y_j) + \min_{i<i''<i'} C(i, i'', j, j') + C(i'', i', j, j').$$

- Otherwise, we make a horizontal cut between y_j and $y_{j'}$, by considering all cuts through $y_{j''}$, where $j < j'' < j'$. This generates a cutting cost of $x_{i''} - x_i$ and yields two subrectangles whose total cost we compute recursively as $C(i, i', j, j'') + C(i, i', j'', j')$ (see Fig. 2(c)).

$$\mathsf{cost}_y(i, i', j, j') \;=\; (x_{i'} - x_i) + \min_{j<j''<j'} C(i, i', j, j'') + C(i, i', j'', j').$$

Observe that the principle of optimality holds, which means that when cutting the two subrectangles, we should do so in a manner that minimizes the total cutting costs. Combining these, we have the following recursive rule:[1]

$$C(i, i', j, j') \;=\; \begin{cases} 0 & \text{if } \mathsf{defectCount}(i, i', j, j') = 0 \\ \min \begin{pmatrix} \mathsf{cost}_x(i, i', j, j') \\ \mathsf{cost}_y(i, i', j, j') \end{pmatrix} & \text{(otherwise).} \end{cases}$$

The overall cost is $C(0, n+1, 0, n+1)$, which covers the entire rectangle $[x_0, x_{n+1}] \times [y_0, y_{n+1}] = [0, L] \times [0, L]$.

(c) We present a memoized solution. Each call to `min-split(i, i', j, j')` computes the value of $C(i, i', j, j')$ (if it has not already been computed) and then saves this value in the 4-dimensional array C. Initially, all the entries are set to -1 to indicate that they are undefined. The initial call is `min-split(0, n+1, 0, n+1)`. This is presented in the code block below.

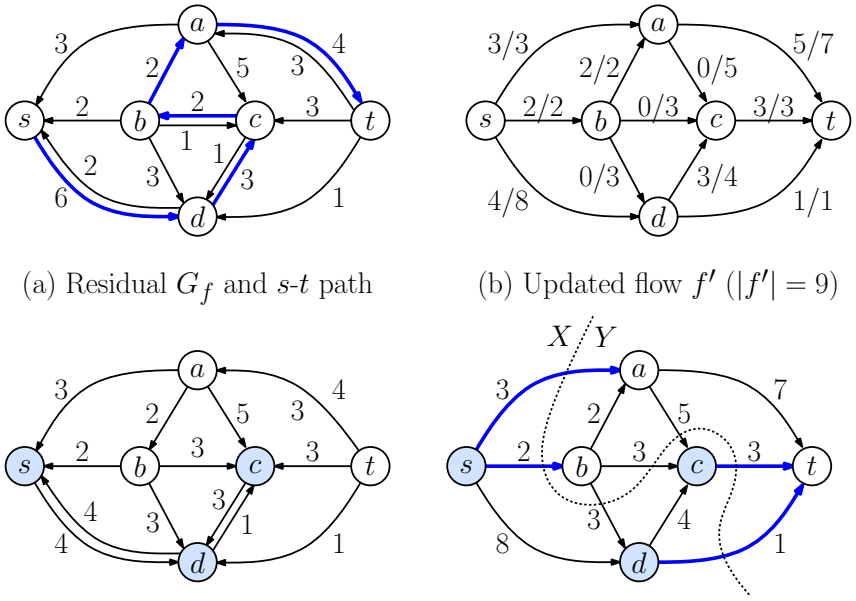
(d) There are $(n + 2)^4 = O(n^4)$ entries in array C. (A more refined analysis reveals that we only access roughly $(n + 2)^4/4$ of the entries, since $i \le i'$ and $j \le j'$, but this does not affect the asymptotic values.) The number of iterations through each of the for-loops is at most n, and therefore the overall running time is $O(n^5)$.

[1] There is something obviously wasteful about this formulation. When making the vertical cut at $x_{i''}$, we did not bother to check that the point having this x-coordinate even lies within the rectangle $[x_i, x_{i'}] \times [y_j, y_{j'}]$! (The same applies for $y_{j''}$.) It turns out the sloppiness does not adversely affect the correctness nor the asymptotic running time. To see why, observe first that any such cut will not be helpful in forming the optimum solution, since the cut does not pass through a defect, and hence it does not reduce the number of defects. It can also be shown that, even if we consider only points lying within the rectangle, the running time will be $O(n^5)$, only with a smaller constant. (I believe that the constant factor will be about $5! = 120$.) Thus, while it is wasteful from a practical perspective, if we are wearing our "theoretician pants", we just don't care.

```
min-split(i, i', j, j') {                          // computes C(i, i', j, j')
    if (C[i, i', j, j'] == -1) {                   // not yet defined?
        if (defectCount(i, i', j, j') == 0)        // no more defects
            C[i, i', j, j'] = 0
        else {                                     // need to split
            best = +infinity
            for (i'' = i to i')                    // try all vertical cuts
                best = min(best, (y[j'] - y[j]) +
                  min-split(i, i'', j, j') + min-split(i'', i', j, j'))
            for (j'' = j to j')                    // try all horizontal cuts
                best = min(best, (x[i'] - x[i]) +
                  min-split(i, i', j, j'') + min-split(i', i, j'', j'))
            C[i, i', j, j'] = best                 // take the best of all
        }
    return C[i, i', j, j']
}
```

Solution 4:

(a) The residual network G_f is shown in Fig. 3(a).



(a) Residual G_f and s-t path

(b) Updated flow f' ($|f'| = 9$)

(c) Updated residual $G_{f'}$ (s, d, and c are reachable)

(d) Cut of capacity 9

Figure 3: Solution to Problem 1.

(b) There is only one s-t path, and it is highlighted with heavy blue edges in Fig. 3(a). The maximum flow that can be pushed through this path is 2.

(c) The updated flow is shown in Fig. 3(b) and the updated residual network $G_{f'}$ is shown in Fig. 3(c).

(d) There is no path from s to t in the $G_{f'}$. (The vertices that are reachable from s are shaded.) It follows that this flow is maximum. Its value is the sum of the flow values out of s, which is $3 + 2 + 4 = 9$.

(e) The flow from (c) was already maximum, so the residual graph is the same as for part (c).

(f) As shown in the Min-Cut/Max-Flow Theorem, a minimum cut (X, Y) results by setting X to the nodes of the residual reachable from s and setting Y to the rest. This yields the cut $(X, Y) = (\{s, c, d\}, \{a, b, t\})$, as shown in Fig. 3(d). The capacity of the cut is equal to the sum of capacities of edges crossing from the X side to the Y side, which is $3 + 2 + 3 + 1 = 9$, which matches $|f'|$. (Note that the edges from Y to X play no role in the cut's capacity.)

Solution 5:

(a) We convert a vertex-capacitated network $G = (V, E)$ into an equivalent edge-capacitated network $G' = (V', E')$ as follows. First, we split each $u \in V$ vertex other than s into t vertex into a pair of vertices $u', u'' \in V'$, which are connected by a "mini-edge". This mini-edge has a capacity equal to the vertex capacity. Let s'' and t' denote the source and sink vertices in G'. Next, for each edge (u, v) in the original graph, create an edge (u'', v') of infinite capacity in your new network (see Fig. 4(b)).

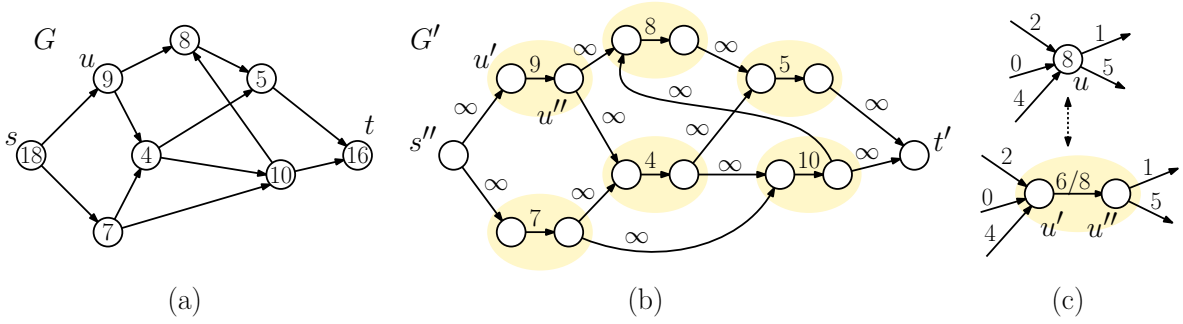


Figure 4: Solution to Problem 5(a).

To establish correctness, we show that given any flow f in G there exists a flow of equal value in G'. To go from G to G', we just copy the flow values from each edge (u, v) to the corresponding edge (u'', v'). We set the flow on the mini-edge to the sum of flows on the incoming edges. By flow conservation this must match the sum of flows on the outgoing edges (see Fig. 4(c)). The capacity on the mini-edge enforces the vertex capacity. To go in the other direction, we copy the flows on the edges (u'', v') to the original edge (u, v). Because of the capacity constraint on the mini-edges, the flow through each vertex satisfies the vertex capacities.

(b) We convert an edge-capacitated network $G = (V, E)$ into an equivalent vertex-capacitated network $G' = (V', E')$ as follows. We split each edge into two edges by adding a "mini-vertex" in the middle. We set the capacity of the mini-vertex to the capacity of the edge. We set the capacities of the original vertices to ∞ (see Fig. 5(b)).

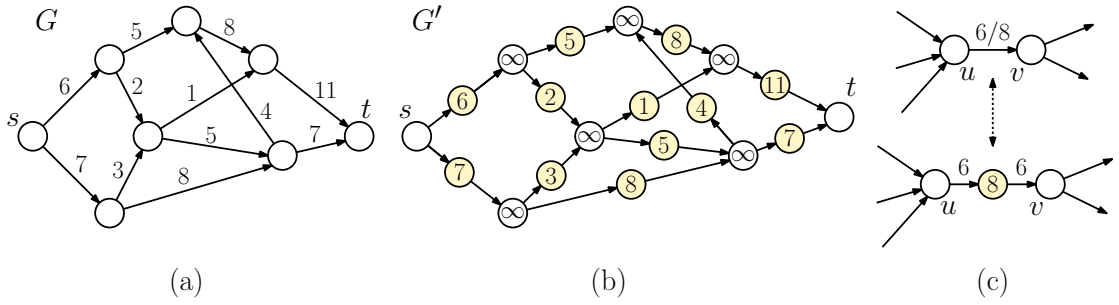


Figure 5: Solution to Problem 5(b).

To establish correctness, we show that given any flow f in G there exists a flow of equal value in G'. To go from G to G', we just copy the flow value on each edge (u, v) to the two halves of the split edge. We set the flow on the two edges to be the same as the flow in the original edge (see Fig. 5(c)). The capacity on the mini-vertex enforces the edge capacity constraint. To go in the other direction, observe that the flows on the two copies of each split edge are equal, and we copy the flow to the original edge (u, v).

Solution to Challenge Problem 1: This employs a standard trick, called *prefix sums*. Compute an array $P[0..n]$, where $P[i]$ is equal to the *ith prefix sum*, that is, $P[i] = \sum_{i=1}^{i} w_i$. This can be computed in $O(n)$ time by setting $P[0] \leftarrow 0$ and for $2 \leq i \leq n$, $P[i] \leftarrow P[i-1] + w_i$. Clearly, it can be stored in $O(n)$ space. To compute $\text{len}(i, j)$ we return $P[j] - P[i-1]$, since

$$\text{len}(i, j) \;=\; \sum_{k=i}^{j} w_k \;=\; \sum_{k=1}^{j} w_k - \sum_{k=1}^{i-1} w_k \;=\; P[j] - P[i-1].$$

Solution to Challenge Problem 2: The solution to this problem is similar in spirit to the the first challenge problem. For each pair of indices $[i, j]$, we compute the block sum of the entries whose indices are less than or equal to both i and j. For $0 \leq i, j \leq n$, define

$$P[i, j] \;=\; \sum_{i''=1}^{i} \sum_{j''=1}^{j} M[i'', j''].$$

This array can be computed in $O(n^2)$ in the same incremental manner as in Challenge Problem 1. Now, we can compute any block sum for the subarray $[i, i'] \times [j, j']$ by taking

$$\text{blockSum}(i, i', j, j') \;=\; P[i', j'] - P[i, j'] - P[i', j] + P[i, j]$$

(see Fig. 6). To see why this works, observe that $P[i', j']$ sums all the entries in the submatrix whose lower-right corner is $[i', j']$, from which we subtract the submatrices to its left and above with $P[i', j]$ and $P[i, j']$, respectively. However, this doubly subtracts the elements of the submatrix whose lower-right corner is $[i, j]$, so to compensate we add in $P[i, j]$.

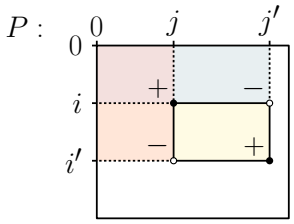


Figure 6: Block sum in a matrix.