**DISCLAIMER:** These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

Goal for this week is to discuss dynamic programs with more complicated states. We already encountered some of these when discussing knapsack last week, in the form of 2-D knapsack, or shortest path with added knapsack-like capacity constraints. The interactions between states in what we’ll discuss now are more direct, in that they correspond to intervals of a line segment.

The most classical of these is probably optimal binary search tree.

**Problem 2.1.** Given a sequence of numbers 1…n, as well as the frequency that item i will be accessed, $f[i]$, build a binary search tree to minimize the total cost of all accesses. That is, minimize

$$\sum_{1 \leq i \leq n} \text{depth}[i] \times f[i]$$

where $\text{depth}[i]$ is the depth of node $i$ in this search tree.

To solve this, consider where the root of the binary search tree, $p$ is. It will divide 1…n into two sub-intervals: those to the left of $p$ and those to the right. Note that the cost of accessing everything in the left subtree is their distance to the left root, plus 1 due to $p$. Same for the right subtree.

That means that the ‘additional’ cost due to having $p$ is the total sum of the frequency of all the nodes. Furthermore, it in turn suffices to have the optimum search trees in both the left and the right subtrees.

This leads to the dynamic programming state $OPT[l][r]$ which gives the optimum total cost of a binary tree with frequency counts $f[l…r]$. The answer is given by $OPT[1][n]$, and the base case is

$$OPT[i][i] = f[i]$$

or more generally

$$OPT[l][r] = 0 \quad \text{if } j < i.$$

The second one has the advantage of not having to deal with empty intervals as special cases.

The transition is then enumerating where the root is:

$$OPT[l][r] = \sum_{l \leq i \leq r} f[i] + \min_{l \leq p \leq r} OPT[l][p - 1] + OPT[p + 1][r].$$
There is also the issue of in which order to access the dynamic program states: state \([l \ldots r]\) depends on all states \([\hat{l}, \hat{r}]\) with \(\hat{l} \geq l\) and \(\hat{r} \leq r\). In particular, it’s a bad idea to access the states in increasing order of both \(l\) and \(r\). Instead, one can access the states in decreasing order of \(l\), but increasing order of \(r\).

This ‘fit a tree to a sequence’ example also generalizes to the parsing of context free grammars. Such grammars are related to programming languages, and consist a set of rules of the form of

\[ A \rightarrow s \]

where \(s\) is a string of alphabets.

For example, if the starting symbol is \(S\), and the two rules are

\[ S \rightarrow \epsilon S \rightarrow 0S1, \]

where \(\epsilon\) represents the empty string, we can generate 0011 via the sequence

\[ S \rightarrow 0S1 \rightarrow 00S11 \rightarrow 0011. \]

**Problem 2.2.** Check if a string \(A[1 \ldots n]\) can be generated from a given context free grammar.

For this, note that the string generated from a single symbol is always a contiguous substring of the final outcome. So we obtain dynamic programming states of the form of

\[ DP[A][l][r] = \text{whether } x[l \ldots r] \text{ can be generated via the starting symbol } A. \]

Then for a rule of the form of \(A \rightarrow BC\), the transition is

\[ DP[A][l][r] = \bigvee_{l \leq k \leq r} (DP[B][l][k] \land DP[C][k + 1][r]). \]

Furthermore, for the more general case of multi-way splits, we can either define states based on how far into the rules we’ve matched, or directly convert the grammar to one where each rule only has two RHS symbols.


We start with some of the more direct (aka. unprincipled) way to get such interval based dynamic programs. Specifically, combinatorics problems where the remaining state is always a single interval.

**Problem 2.3.** You start at 0 on the number line. There are \(n\) stashes of sugar, each located at \(x_i\), and provide you with \(d_i\) units of energy. Find in \(O(n^2)\) time the maximum total distance that you can move, consuming each stash at most once.

This problem is actually available at [https://dmoj.ca/problem/tle16c6s3](https://dmoj.ca/problem/tle16c6s3). A similar one, [https://dmoj.ca/problem/ncco2d1p1](https://dmoj.ca/problem/ncco2d1p1), was picked for problem set 2 instead.

The key observation to this problem is that if you pass by an uncollected stash, you may as well collect it.
That means that if we sort the stashes by location, the set of ones that we’ve collected is always of the form of \( x[i \ldots j] \) for some \( i \leq j \).

Of course, when at once of these locations, we want to maximize the amount of distance that can still be traveled. So the dynamic programming state are:

\[
MaxEnergy[i][j][0], MaxEnergy[i][j][1],
\]

which give the max energy remaining when having picked up everything in interval \( x[i \ldots j] \), and are currently standing on the left (\( x[i] \)) or right (\( x[j] \)) endpoints respectively.

The transition involves 2 possibilities per state: continue moving in the current direction, or ‘flip over’ to the other end of the traversed list.

This then gives the intervals of stashes that could be picked up. To obtain the maximum total energy / distance travelled, note that that amount equals to the sum of stashes picked up. So we enumerate again through the intervals that are possible, and pick the one with the maximum amount.

**Problem 2.4.** There is a sequence of numbers. Two players alternate take numbers from this sequence: they can only be taken from either end. Both players want to maximize the sum of numbers they take, and play optimally. Calculate the optimal amount that player 1 can win.

This problem belongs to the category of adversarial search: each player wants to pick a strategy that maximizes their payoff. Note that the total score of the players, given a state of the game, is always the same: the sum of the numbers remaining. So maximizing ones winning is equivalent to minimizing the winning of the other player.

To turn this into a dynamic program, observe that the remaining items is always a sub-interval of the original input. This leads to the dynamic programming states

\[
OPT[i][j] = \text{maximum sum achievable if a player starts with } x[i \ldots j]
\]

where we assume the input sequence is \( x[1 \ldots n] \).

The base case is

\[
OPT[i][i] = x[i]
\]

and the solution is found in \( OPT[1][n] \). So all that remains is the transition: here the possibilities are that we can hand the other player \([i \ldots j - 1]\) or \([i + 1 \ldots j]\). Among these two possibilities, we want to minimize what the other player gets. So we have

\[
OPT[i][j] = \sum_{i \leq k \leq j} x[k] - \min\{OPT[i][j - 1], OPT[i + 1][j]\}.
\]

This gives an \( O(n^2) \) time algorithm with \( O(n^2) \) memory. With a little more effort, one can show that the memory can also be reduced to \( O(n) \).
This problem can actually be solved in $O(n \log n)$ time using a relatively simple greedy algorithm. More on it can be found at https://www.mimuw.edu.pl/~idziaszek/termity/termity.pdf and https://szkopul.edu.pl/problemset/problem/EqNMzomALKTC6kJ0Vi_atW/site/?key=statement.

Problem 2.5. Given a set of matrices $A_1, \ldots, A_n$ with dimensions $m_0 \times m_1$, $m_1 \times m_2$, $\ldots$, $m_{n-1} \times m_n$, compute $A_1 \times A_2 \times \cdots \times A_n$ with minimum cost. The cost of multiplying an $x \times y$ matrix by a $y \times z$ matrix is exactly $xyz$.

The goal is to multiply these matrices all together, leading to one of size $m_0 \times m_n$. The cost is dependent on ordering though: if we multiply three matrices of sizes $1 \times 100$, $100 \times 10$, $10 \times 30$, multiplying the first two costs 1000, which then produces a $1 \times 10$ matrix to be multiplied against a $10 \times 30$ matrix, which takes cost 300. First multiplying the $100 \times 10$ matrix against the $10 \times 30$ matrix costs 30000, much more than otherwise.

In this case with $n = 3$, we can simply pick the least expensive multiplication. However, this doesn’t work once we’re at 4 matrices: see the example in the text book.

Note that the dimensions of the resulting matrix from multiplying an $m \times n$ matrix and $n \times p$ matrix has dimensions $m \times p$. Similarly, multiplying $A \times B \times C$ where matrix $A$ is $m \times n$, matrix $B$ is $n \times p$, and matrix $C$ is $p \times q$, results in an $m \times q$ matrix.

Accordingly, the product $A_i \times A_{i+1} \times \cdots \times A_j$ will have dimension $m_{i-1} \times m_j$ because $A_i$ has dimension $m_{i-1} \times m_i$ and $A_j$ has dimension $m_{j-1} \times m_j$.

Suppose the last multiplication that occurred was $(A_1 \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_n)$, which is to say that we first computed $A_1 \times \cdots \times A_k$ and $A_{k+1} \times \cdots \times A_n$, then multiplied the result. The cost of the final operation would be $m_0 \cdot m_k \cdot m_n$ because $A_1 \times \cdots \times A_k$ has dimension $m_0 \times m_k$ and $A_{k+1} \times \cdots \times A_n$ has dimension $m_k \times m_n$. To compute the full cost then of the entire multiplication, we would need to know the cost of computing $A_1 \times \cdots \times A_k$ and $A_{k+1} \times \cdots \times A_n$. These can both be considered subproblems of our original problem. As the final multiplication must take some form $(A_1 \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_n)$ for $k$ between 1 and $n - 1$, if we knew the optimal cost for computing $A_1 \times \cdots \times A_k$ and $A_{k+1} \times \cdots \times A_n$ for each $k$ between 1 and $n - 1$, then we could take the minimum combination and that would be the minimum cost of computing the full product. This is suggestive that the subproblems we should be considering are that of minimally computing $A_i \times A_{i+1} \times \cdots \times A_j$ for each $i \leq j$.

1. Consider the subproblem of minimally computing $A_i \times A_{i+1} \times \cdots \times A_j$ and let $C(i, j)$ be the minimal cost of computing $A_i \times A_{i+1} \times \cdots \times A_j$.

2. Base case: $C(i, i) = 0$ for all $i$.

3. Subproblems: $C(i, j)$ for all $i \leq j$
4. Transition:

\[ C(i, j) = \min_{i \leq k < j} \{C(i, k) + C(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j\}. \]

The value \(m_{i-1} \cdot m_k \cdot m_j\) is from assuming we have \(A_i \times \cdots \times A_k\) which has dimension \(m_{i-1} \times m_k\), and \(A_{k+1} \times \cdots \times A_j\) which has dimension \(m_k \times m_j\) and we need to compute \((A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)\). The cost of this computation must then be \(m_{i-1} \cdot m_k \cdot m_j\).

**Chain Matrix Multiplication**

1. \(C(i, i) = 0\) for all \(i\)

2. For \(s = 1\) to \(n - 1\)

   (a) For \(i = 1\) to \(n - s\)

   i. \(j = i + s\)

   ii. \(C(i, j) = \min_{i \leq k < j} \{C(i, k) + C(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j\}\)

3. Return \(C(1, n)\).

Each transition takes \(O(n)\) time because it is the maximum over at most \(n\) numbers, each of which takes \(O(1)\) time to compute. The number of subproblems is \(O(n^2)\). The full running time is \(O(n^3)\).