• 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.

• Main topics:
  – Schedule changes: Test 1 moved to next week, Test 2 moved to (two weeks) after fall break.
  – Office hours location changes (aka. rush B): check Klaus 2108, Klaus 2100, and area between theory labs in that order.
  – This lecture: introduction to dynamic programming.
  – Example problems: paths on grid, longest increasing subsequence.

1 Recursive Search

Consider the following problem: you start at row 1, column 1 of a \( n \) row, \( m \) column grid. The only valid moves are walking downwards and rightwards. How many different paths can one take to get to \((n, m)\)?

This can be solved by enumeration. Specifically, we can write a brute force search program as follows:

```c
NumWays(i, j) {
    if (i == n AND j == m) then RETURN 1;
    result = 0;
    if (i < n) result = result + NumWays(i + 1, j);
    if (j < m) result = result + NumWays(i, j + 1);
    RETURN result;
}
```

This program recurses on the two possible choices at each step. It does one recursive call for each step of every path.

With a bit more math, one can check that the number of paths is \( O(2^{n+m}) \), so the total running time can be bounded by \( O((n + m)2^{n+m}) \).\(^1\)

\(^1\)this is a crude bound, but even tight bounds are still exponential when \( n \) is close to \( m \).
2 Search States

To speed this program up, note that if we’re now at \((i, j)\), how we got there doesn’t matter.

It can also be solved faster by counting the number of ways that one can get to \((i, j)\). That is, this recursive search has a state that’s much smaller than the search space. In the case of counting paths, we have:

- State is the location where we’re currently at, for which there is at most \(O(nm)\) of them.
- Search space is the number of paths, which is exponential in \(\min\{n, m\}\).

So we can take advantage of this by storing the answer of a state that we’ve already computed. This can be done either implicitly (by using e.g. a hash table to return the return value of \(\text{NumWays}(i, j)\)), or by explicitly define a state:

\[
\text{NumWays}(i, j) \overset{\text{def}}{=} \text{Number of ways of getting from } (i, j) \text{ to } (n, m) \quad (1)
\]

The first approach of implicitly remembering function signatures is known as memorization. The second approach is more motivated by sequences and series, and is the origin of the term ‘dynamic programming’. This came about because each step of the recurrence may not be a closed form involving other terms. In our case we have (ignoring boundary cases):

\[
\text{NumWays}(i, j) = \text{NumWays}(i + 1) + \text{NumWays}(i, j + 1), \quad (2)
\]

which is a closed form recurrence.

However, we can easily get to a problem where this is not the case: suppose the grid has obstacles where we can’t get through. Then for the grids where we can’t visit, say \(S\), we have to set

\[
\text{NumWays}(i, j) \leftarrow 0 \quad \forall (i, j) \in S.
\]

For example, on a 3 rows by 4 columns grid with an obstacle in row 2, column 2, the NumWays array looks like:

\[
\begin{array}{cccc}
8 & 6 & 6 & 2 \\
2 & 0 & 4 & 2 \\
2 & 2 & 2 & 1 \\
\end{array}
\]

3 Dynamic Programming

To get into a situation where we want to make even more decisions, suppose instead of counting the number of paths, each cell in the grid has a score, and we want to maximize our total score. The best score is still ‘history independent’: it’s only a function of the current location \((i, j)\). So we can define the state \(\text{BestScore}(i, j)\) to be the max sum of
scores starting from \((i, j)\) and ending at \((n, m)\), giving the transition (ignoring boundaries again):

\[
BestScore(i, j) = Score(i, j) + \max \{ Score(i + 1) + Score(i, j + 1) \}. \tag{3}
\]

This transition is ‘dynamic’ in that at each step a different decision can be made based on the values of the two subsequent scores. This name is still a bit strange, and for me it only started to make sense when contrasted with other terms such as linear programming (which is in Section 7 of the text, and will be covered after). Nonetheless, dynamic programming has proven to be a sufficiently useful paradigm for us to systematically formalize it. Its components are:

1. The states: these are essentially the ‘search signatures’. Here it is \(BestScore(i, j)\), the max score that one could get when starting at \((i, j)\) going to \((n, m)\).

2. The base case, this the termination condition. Here it’s \(BestScore(n, m) = Score[n][m]\).

3. The transition function, which is above.

As an example, on a 2 rows by 3 with values:

\[
\begin{array}{ccc}
0 & 5 & 7 \\
8 & 2 & 1
\end{array}
\]

the \(BestScore\) array that we get is:

\[
\begin{array}{ccc}
13 & 13 & 8 \\
11 & 3 & 1
\end{array}
\]

When designing a dynamic program, the transition function is the majority of the work: it describes how the problem decouples, and what are the states. The states can often be inferred from the transition function, while the base case is in turn obtainable from inspecting the states.

One other important component of a dynamic program is the order in which we visit the states. Note that to compute \(BestScore(i, j)\), we need to know the values of \(BestScore(i + 1, j)\) and \(BestScore(i, j + 1)\). At a first glance this leads to a chicken-and-egg problem: we need the results of states to compute more results of states. An approach such as memorization more or less hides this under the rug, but the iterative approach explicitly requires us to think about how the states are ordered.

Thankfully, in most cases there is a ‘size’ of the problem that gives a clear picture of what the states should be. For both the path counting, and max score path problem, we can just walk ‘backwards’ on the table: up the rows, and right-to-left on each row,. Then at each point we spend \(O(1)\) evaluating the transition function, giving \(O(nm)\) total.
4 Longest Increasing Subsequence

For one more example, consider the longest increasing subsequence (LIS) problem: given a sequence

\[ A[1] \ldots A[n], \]

find the longest subsequence, \( i_1 < i_2 \ldots i_k \) such that

\[ A[i_j] < A[i_{j+1}] \quad \forall 1 \leq j < k. \]

A search program simply tries all next elements that start somewhere.

\[
\text{LIS(i)} \{ \text{\textbackslash returns longest increasing subsequence starting at i} \\
\text{result = 1;} \\
\text{FOR j = i + 1...n} \\
\quad \text{if (A[i] < A[j])} \\
\quad \quad \text{result = max(result, LIS(j) + 1);} \\
\text{RETURN result;}
\}
\]

Here the \( LIS(j) + 1 \) takes into account that the sequence also uses \( A[i] \). Calling this function on all starting locations and taking max then gives the result.

Note once again that the result of \( LIS(i) \) is dependent only on \( i \): only the value of \( A[i] \), rather than the choices of entries before, affects the length of the rest of the sequence. So we can create a dynamic program with:

1. State: \( LIS[i] \): length of longest increasing subsequence starting at \( i \).
2. Base case: \( LIS[i] \geq 1 \).
3. Transition:

\[
LIS[i] = 1 + \left( \max_{j > i, A[i] < A[j]} LIS[j] \right).
\]

Finally, this transition can be evaluated by ordering the states backwards: we first compute \( LIS[n] \), then \( LIS[n - 1] \), and etc. Each of these takes \( O(n) \) to compute, giving a total of \( O(n^2) \).

As an example, on the input sequence

\[ 8, 3, 9, 4, 6, \]

the \( LIS \) array at the end gives:

\[ 2, 3, 1, 2, 1. \]