LEC18: APPLICATIONS OF DYNAMIC PROGRAMMING

CSE 373 Analysis of Algorithms
Fall 2016
Instructor: Prof. Sael Lee

Lecture slide courtesy of Prof. Steven Skiena
There are three steps involved in solving a problem by dynamic programming:

1. Formulate the answer as a recurrence relation or recursive algorithm.
2. Show that the number of different parameter values taken on by your recurrence is bounded by a (hopefully small) polynomial.
3. Specify an order of evaluation for the recurrence so the partial results you need are always available when you need them.
Develop an algorithm to find the longest monotonically increasing subsequence within a sequence of \( n \) numbers.

*Difference between increasing sequence and run*

+ **run**: elements must be physical neighbors of each other.
  - EX> Given \( S = \{2, 4, 3, 5, 1, 7, 6, 9, 8\} \),
  - There are four longest increasing runs of length 2: (2, 4), (3, 5), (1, 7), and (6, 9).
  - Finding the longest increasing run in a numerical sequence is straightforward

+ **longest increasing subsequence** (LIS) of \( S \) does not require neighborhood.
  - \( S \) has eight longest increasing subsequence in \( S \) of length 5, including \( \{2, 3, 5, 6, 8\} \).
  - Finding the longest increasing subsequence is considerably trickier.
To find the right recurrence, ask what information about the first \( n - 1 \) elements of \( S \) would help you to find the answer for the entire sequence.

- The length of the LIS in \( s_1, s_2, \ldots, s_{n-1} \) seems a useful thing to know.
- In addition, we need to know the length of the longest sequence that \( s_n \) will extend.

Define \( l_i \) to be the length of the longest sequence ending with \( s_i \).
The LIS containing the $n$th number will be formed by appending it to the longest increasing sequence to the left of $n$ that ends on a number smaller than $s_n$.

The following recurrence computes $l_i$: \[ l_i = \max_{0<j<i} l_j + 1, \text{ where } (s_j < s_i) \]
\[ l_0 = 0 \]

Goal Cell: The length of the LIS of the entire permutation is given by $\max_{0\leq i\leq n} l_i$, since the winning sequence will have to end somewhere.
Table associated with our previous example:

\[ S = \{2, 4, 3, 5, 1, 7, 6, 9, 8\} \]

<table>
<thead>
<tr>
<th>Sequence ( s_i )</th>
<th>2</th>
<th>4</th>
<th>3</th>
<th>5</th>
<th>1</th>
<th>7</th>
<th>6</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length ( l_i )</td>
<td>1</td>
<td>2</td>
<td></td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Predecessor ( p_i )</td>
<td>–</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>–</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Auxiliary information: index \( p_i \) of the element that appears immediately before \( s_i \) in the longest increasing sequence ending at \( s_i \).

Reconstruction: Start from the last value of the longest sequence and follow the pointers to the other items in the sequence.
Each one of the $n$ values of $l_i$ is computed by comparing $s_i$ against (up to) $i-1 \leq n$ values to the left of it,

so this analysis gives a total of $O(n^2)$ time.
APP2: THE PARTITION PROBLEM

- **Problem:** Integer Partition without Rearrangement
- **Input:** An arrangement $S$ of nonnegative numbers $\{s_1, \ldots, s_n\}$ and an integer $k$.
- **Output:** Partition $S$ into $k$ or fewer ranges, to minimize the maximum sum over all the ranges, without reordering any of the numbers.

Example: three workers are given the task of scanning through a shelf of books in search of a given piece of information. What is the fairest way to divide the workload (i.e. Sum # of pages in the partitions are even): 100 200 300 400 500 | 600 700 | 800 900
Notice that the $k$th partition starts right after we placed the $(k-1)$st divider.

Where can we place this last divider? Between the $i$th and $(i + 1)$st elements for some $i$, where $1 \leq i \leq n$.

Let $M[n, k]$ be the minimum possible cost over all partitionings of $\{s_1, \ldots, s_n\}$ into $k$ ranges, where the cost of a partition is the largest sum of elements in one of its parts.
What is the cost of this? The total cost will be the larger of two quantities—

1. the cost of the last partition \( \sum_{j=i+1}^{n} s_j \), and
2. the cost of the largest partition formed to the left of \( i \).

See the recursion?

\[
M[n, k] = \min_{1 \leq i \leq n} \max(m[i, k - 1], \sum_{j=i+1}^{n} s_j)
\]
BOUNDARY CONDITIONS

The smallest reasonable value of the

- first argument is \( n = 1 \) (first partition consists of a single element)
  \[ M[1, k] = s_1, \text{ for all } k > 0 \] and,

- second argument is \( k = 1 \) (we do not partition \( S \) at all).

\[ M[n, 1] = \sum_{i=1}^{n} s_i \]
When we store the partial results, total of $k \cdot n$ cells exist in the table.

How much time does it take to compute the result $M[n, k]$?

- find the minimum of $n'$ quantities each of which is the maximum of the table lookup and a sum of at most $n$ elements
- $\rightarrow$ at most $n^2$ time per box

Total recurrence can be computed in $O(kn^3)$ time
```
partition(int s[], int n, int k)
{
    int m[MAXN+1][MAXK+1];    /* DP table for values */
    int d[MAXN+1][MAXK+1];    /* DP table for dividers */
    int p[MAXN+1];    /* prefix sums array */
    int cost;    /* test split cost */
    int i,j,x;    /* counters */

    p[0] = 0;    /* construct prefix sums */
    for (i=1; i<=n; i++) p[i]=p[i-1]+s[i];

    for (i=1; i<=n; i++) m[i][1] = p[i];    /* initialize boundaries */
    for (j=1; j<=k; j++) m[1][j] = s[1];

    for (i=2; i<=n; i++) /* evaluate main recurrence */
        for (j=2; j<=k; j++) {
            m[i][j] = MAXINT;
            for (x=1; x<=(i-1); x++) {
                cost = max(m[x][j-1], p[i]-p[x]);
                if (m[i][j] > cost) {
                    m[i][j] = cost;
                    d[i][j] = x;
                }
            }
        }
    reconstruct_partition(s,d,n,k);    /* print book partition */
}
```

We keep track of prefix sums $p[i] = \sum_{k=1}^{i} s_k$ for faster run time since $\sum_{k=i}^{j} s_k = p[k] - p[j]$

Since $\sum_{k=i}^{j} s_k = p[k] - p[j]$ Enables us to evaluate the recurrence in linear time per cell, yielding an $O(kn^2)$ algorithm.
**RECONSTRUCTING ACTUAL PARTITION**

- Final value of $M(n,k)$ will be the cost of the largest range in the optimal partition.
- Matrix $D$ is used to reconstruct the optimal partition by working backward from $D[n, k]$ and add a divider at each specified position.

```c
void reconstruct_partition(int s[], int d[MAXN+1][MAXK+1], int n, int k) {
    if (k == 1)
        print_books(s, 1, n);
    else {
        reconstruct_partition(s, d, d[n][k], k-1);
        print_books(s, d[n][k]+1, n);
    }
}

void print_books(int s[], int start, int end) {
    int i; /* counter */
    for (i = start; i <= end; i++) printf(" %d ", s[i]);
    printf("\n");
}
```
Partitioning
\{1, 1, 1, 1, 1, 1, 1, 1, 1\}
into \\{\{1, 1\}, \{1, 1, 1\}, \{1, 1, 1\}\}

Notice that final value of \(M(n, k)\) is the cost of the largest range in the optimal partition.
PARSING CONTEXT-FREE GRAMMARS

• Learning it in your compiler class.
A triangulation of a polygon $P = \{v_1, \ldots, v_n, v_1\}$ is a set of nonintersecting diagonals that partitions the polygon into triangles.

The weight of a triangulation is the sum of the lengths of its diagonals.

We seek to find its minimum weight triangulation for a given polygon $p$. 
Observe that every edge of the input polygon must be involved in exactly one triangle. Turning this edge \((i,j)\) into a triangle means identifying the third vertex, \(k\).

Let \(T[i, j]\) be the cost of triangulating from vertex \(v_i\) to vertex \(v_j\), ignoring the length of the chord \(d_{ij}\) from \(v_i\) to \(v_j\).

\[
T[i, j] = \min_{i+1 \leq k \leq j-1} (T[i, k] + T[k, j] + d_{ik} + d_{kj})
\]

Basis: when \(i\) and \(j\) are immediate neighbors, as \(T[i, i+1] = 0\).
Evaluation an proceed in terms of the gap size from $i$ to $j$:

Minimum-Weight-Triangulation($P$)
for $i = 1$ to $n - 1$ do $T[i, i + 1] = 0$
for gap = 2 to $n - 1$
    for $i = 1$ to $n - \text{gap}$ do
        $j = i + \text{gap}$
        $T[i, j] = \min_{i+1 \leq k \leq j-1} (T[i, k] + T[k, j] + d_{ik} + d_{kj})$
return $T[1, n]$

There are $\binom{n}{2}$ values of $T$, each of which takes $O(j - i)$ time if we evaluate the sections in order of increasing size.

Since $j - i = O(n)$, complete evaluation takes $O(n^3)$ time and $O(n^2)$ space.
Dynamic programming doesn’t always work.

**Working example:**

- **Problem:** Longest *Simple* Path
- **Input:** A weighted graph G, with specified start and end vertices s and t.
- **Output:** What is the **most expensive** path from s to t that does not visit any vertex more than once?
WHEN ARE DP ALGORITHMS CORRECT?

- Suppose we define $LP[i, j]$ as a function denoting the length of the longest simple path from $i$ to $j$.

- Note that the longest simple path from $i$ to $j$ had to visit some vertex $x$ right before reaching $j$.
  - Thus, the last edge visited must be of the form $(x, j)$.

- Recurrence relation: the length of the longest path, where $c(x, j)$ is the cost/weight of edge $(x, j)$:
  \[ LP[i, j] = \max_{(x, j) \in E} LP[i, x] + c(x, j) \]

- Can you see the problem?
  - Does not enforce simplicity (we are not allowed to visit any vertex more than once)
  - No evaluation order: It is not clear what the smaller subprograms are.
Dynamic programming can be applied to any problem that observes the *principle of optimality*. Partial solutions can be optimally extended with regard to the state after the partial solution, instead of the specifics of the partial solution itself.

- Future decisions are made based on the consequences of previous decisions, not the actual decisions themselves.
- Problems do not satisfy the principle of optimality when the specifics of the operations matter, as opposed to just the cost of the operations.

Example: in deciding whether to extend an approximate string matching by a substitution, insertion, or deletion, we did not need to know which sequence of operations had been performed to date.
WHEN ARE DP ALGORITHMS EFFICIENT?

- Running time of DP is a function of following:
  - (1) number of partial solutions we must keep track of, and
  - (2) how long it take to evaluate each partial solution.

- The partial solutions should be completely described by specifying the stopping places in the input.

  + Once the order is fixed, there are relatively few possible stopping places or states, so we get efficient algorithms.
When the objects are not firmly ordered, we get an exponential number of possible partial solutions.

EX> Suppose the state of our partial solution is entire path $P$ taken from the start to end vertex.

$$LP[i, j, P + x] = \max_{(x, j) \in E, x, j \notin P} LP[i, x, P] + c(x, j)$$

This is Correct but not efficient:

+ The path $P$ consists of an ordered sequence of up to $n - 3$ vertices. There can be up to $(n - 3)!$ such paths!