CSE 421
Introduction to Algorithms
Lecture 12, Winter 2024
Dynamic Programming
Announcements

• Dynamic Programming Reading:
  – 6.1-6.2, Weighted Interval Scheduling
  – 6.4 Knapsack and Subset Sum
  – 6.6 String Alignment
    • 6.7* String Alignment in linear space
  – 6.8 Shortest Paths (again)
  – 6.9 Negative cost cycles
    • How to make an infinite amount of money
Dynamic Programming

- The most important algorithmic technique covered in CSE 421

- Key ideas
  - Express solution in terms of a polynomial number of sub problems
  - Order sub problems to avoid recomputation
Recursion vs Iteration

Factorial(n){
    if (n <= 1)
        return 1;
    else
        return n*Factorial(n-1);
}

Factorial(n){
    v = 1;
    for (i = 2; i <= n; i++)
        v = v*i
    return v;
}
Counting Rabbits

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, \ldots

\[ F_0 = 0; \quad F_1 = 1; \quad F_n = F_{n-1} + F_{n-2} \]

Fib(n){
    if (n = 0)
        return 0;
    else if (n = 1)
        return 1;
    else
        return Fib(n-1) + Fib(n-2);
}


Fibonacci with Memoization

Fib(n) {  
    if (n == 0)  
        return 0;  
    else if (n == 1)  
        return 1;  
    else  
        return Fib(n-1) + Fib(n-2);  
}
Reordering computation

Fib(n){
    int[ ] F = new [n+1]
    F[0] = 0;
    F[1] = 1;
    for (i = 2; i <= n; i++)
        F[i] = F[i-1] + F[i-2];
    return F[n];
}

Dynamic Programming

• Weighted Interval Scheduling
• Given a collection of intervals $I_1, \ldots, I_n$ with weights $w_1, \ldots, w_n$, choose a maximum weight set of non-overlapping intervals.

Intervals sorted by end time:

4

6

3

5

7

6
Optimality Condition

• $\text{Opt}[j]$ is the maximum weight independent set of intervals $I_1, I_2, \ldots, I_j$

• $\text{Opt}[j] = \max( \text{Opt}[j - 1], w_j + \text{Opt}[p[j]] )$
  – Where $p[j]$ is the index of the last interval which finishes before $I_j$ starts
Algorithm

MaxValue(j) =
    if j = 0 return 0
    else
        return max( MaxValue(j-1),
                      w_j + MaxValue(p[j]) )

Worst case run time: $2^n$
A better algorithm

M[j] initialized to -1 before the first recursive call for all j

MaxValue(j) =
  if j = 0 return 0;
  else if M[j] != -1 return M[j];
  else {
    M[j] = max(MaxValue(j-1), wj + MaxValue(p[j]));
    return M[j];
  }
Iterative Algorithm

MaxValue(n) {
    int[ ] M = new int[n+1];
    M[0] = 0;

    for (int i = 1; i <= n; i++) {
        M[j] = max(M[j-1], w_j + M[p[j]]);
    }

    return M[n];
}
Fill in the array with the Opt values

\[ \text{Opt}[j] = \max (\text{Opt}[j-1], w_j + \text{Opt}[p[j]]) \]

\[
\begin{array}{cccccc}
2 & \text{P}[I_1] = 0 \\
4 & \text{P}[I_2] = 0 \\
7 & \text{P}[I_3] = 1 \\
4 & \text{P}[I_4] = 2 \\
6 & \text{P}[I_5] = 1 \\
7 & \text{P}[I_6] = 4 \\
6 & \text{P}[I_7] = 3
\end{array}
\]
Computing the solution

\[ \text{Opt}[j] = \max (\text{Opt}[j - 1], w_j + \text{Opt}[p[j]]) \]

Record which case is used in Opt computation

\[
\begin{array}{cccccc}
2 & 4 & 7 & 4 & 6 & 7 \\
& & & & & 6 \\
& & & & & & 7 \\
& & & & & & 6 \\
\end{array}
\]
Iterative Algorithm

```java
int[] M = new int[n+1];
char[] R = new char[n+1];

M[0] = 0;
for (int j = 1; j < n+1; j++) {
    v1 = M[j-1];
    v2 = W[j] + M[P[j]];
    if (v1 > v2) {
        M[j] = v1;
        R[j] = 'A';
    } else {
        M[j] = v2;
        R[j] = 'B';
    }
}
```
Algorithm Summary

- O(n) time algorithm for finding maximum weight independent set of intervals.
- Key idea: Creating an Opt function to express optimal set of $I_1, I_2, \ldots, I_k$ in terms of optimal set of $I_1, I_2, \ldots, I_{k-1}$.
- Organize computation to avoid recomputation.
Optimal linear interpolation

Error $= \sum (y_i - ax_i - b)^2$
What is the optimal linear interpolation with three line segments?
What is the optimal linear interpolation with two line segments
What is the optimal linear interpolation with n line segments
Notation

- Points $p_1, p_2, \ldots, p_n$ ordered by x-coordinate ($p_i = (x_i, y_i)$)
- $E_{i,j}$ is the least squares error for the optimal line interpolating $p_i, \ldots, p_j$
Optimal interpolation with two segments

• Give an equation for the optimal interpolation of $p_1, \ldots, p_n$ with two line segments

• $E_{i,j}$ is the least squares error for the optimal line interpolating $p_i, \ldots, p_j$
Optimal interpolation with $k$ segments

• Optimal segmentation with three segments
  – $\text{Min}_{i,j}\{E_{1,i} + E_{i,j} + E_{j,n}\}$
  – $O(n^2)$ combinations considered

• Generalization to $k$ segments leads to considering $O(n^{k-1})$ combinations
Opt\textsubscript{k}[\textit{j}]: Minimum error approximating 
\textit{p\textsubscript{1}}…\textit{p\textsubscript{j}} with \textit{k} segments

How do you express Opt\textsubscript{k}[\textit{j}] in terms of 
Opt\textsubscript{k-1}[1],…,Opt\textsubscript{k-1}[\textit{j}]?
Optimal sub-solution property

Optimal solution with $k$ segments extends an optimal solution of $k-1$ segments on a smaller problem
Optimal multi-segment interpolation

Compute Opt[ k, j ] for 0 < k < j < n

for j := 1 to n
    Opt[ 1, j] = E_{1,j};
for k := 2 to n-1
    for j := 2 to n
        t := E_{1,j}
        for i := 1 to j -1
            t = min (t, Opt[k-1, i ] + E_{i,j})
        Opt[k, j] = t
Determining the solution

• When Opt[k,j] is computed, record the value of i that minimized the sum
• Store this value in a auxiliary array
• Use to reconstruct solution
Variable number of segments

- Segments not specified in advance
- Penalty function associated with segments
- Cost = Interpolation error + $C \times \#\text{Segments}$
Penalty cost measure

- $\text{Opt}[j] = \min(E_{1,j}, \min_i (\text{Opt}[i] + E_{i,j} + P))$