CSE 417 Algorithms and Complexity

Lecture 19, Autumn 2023
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

• Homework 7
Dynamic Programming

• The most important algorithmic technique covered in CSE 417

• 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, ...  

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

Fib(n){  
    if (n = 0)  
        return 0;  
    else if (n = 1)  
        return 1;  
    else  
        return Fib(n-1) + Fib(n-2);  
}
\[ F_6 \]
\[ F_5 \]
\[ F_4 \]
\[ F_3 \]
\[ F_2 \]
\[ F_1 \]
\[ F_0 \]
\[ F_1 \]
\[ F_0 \]
\[ F_1 \]
\[ F_0 \]
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
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), w_j + 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];
}

11/13/2023       CSE 417
Fill in the array with the Opt values

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11/13/2023

CSE 417
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
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(n²) time algorithm for finding maximum weight independent set of intervals

• Key idea: Creating an Opt function to express optimal set of I₁, I₂, . . . Iₖ in terms of optimal set of I₁, I₂, . . . Iₖ₋₁

• Organize computation to avoid recomputation