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

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

```plaintext
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, …

F₀ = 0;  F₁ = 1;  Fₙ = Fₙ₋₁ + Fₙ₋₂

```plaintext
Fib(n){
  if (n = 0)
    return 0;
  else if (n = 1)
    return 1;
  else
    return Fib(n-1) + Fib(n-2);
}
```
Fibonacci with Memoization

```java
Fib(n)
if (n == 0)
    return 0;
else if (n == 1)
    return 1;
else
    return Fib(n-1) + Fib(n-2);
```

Reordering computation

```java
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 I₁, ..., Iₙ with weights w₁, ..., wₙ, choose a maximum weight set of non-overlapping intervals

Optimality Condition

- Opt[j] is the maximum weight independent set of intervals I₁, I₂, ..., Iᵢ
- Opt[j] = max( Opt[j-1], wᵢ + Opt[p[j]])
  - Where p[j] is the index of the last interval which finishes before Iᵢ starts

Algorithm

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

Worst case run time: 2ⁿ

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), wᵢ + 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[i] = max(M[i-1], w_i + M[p[i]]);
    }
    return M[n];
}

Fill in the array with the Opt values

Opt[j] = max (Opt[j - 1], w_j + Opt[p[j]])

Computing the solution

Opt[j] = max (Opt[j - 1], w_j + Opt[p[j]])

Record which case is used in Opt computation

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, ..., I_k in terms of optimal set of I_1, I_2, ..., 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
- With two line segments
- 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:

- \( n \) line segments
  - \( \text{Give an equation for the optimal interpolation of } p_1, \ldots, p_n \text{ with two line segments} \)
  - \( E_{i,j} \) 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
  - \( \min_{i,j} \{ E_{1,j} + E_{i,j} + E_{j,n} \} \)
  - \( O(n^2) \) combinations considered
- Generalization to \( k \) segments leads to considering \( O(n^{k-1}) \) combinations
Opt$_k$[j] : Minimum error approximating $p_1...p_j$ with $k$ segments

How do you express Opt$_k$[j] in terms of Opt$_{k-1}$[1],...,Opt$_{k-1}$[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 := $E_{i,j}$
        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 x #Segments

Penalty cost measure

• Opt[j] = min($E_{1,j}$, min$_i$(Opt[i] + $E_{i,j}$ + P))