CSE 417 Algorithms and Complexity

Lecture 19, Winter 2023

Dynamic Programming, Part II

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

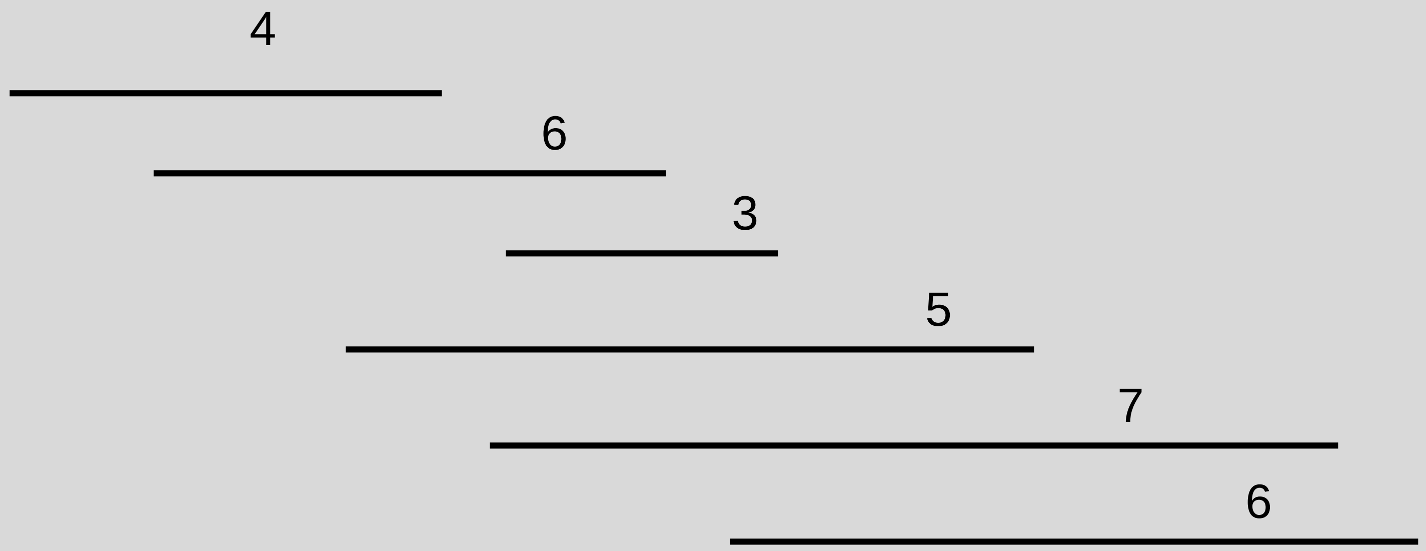
Key Ideas for Dynamic Programming

- Give a recursive solution for the problem in terms of optimizing and objective function
- Order subproblems to avoid duplicate computation
- Determine the elements that form the solution

Intervals sorted by end time

Weighted Interval Scheduling

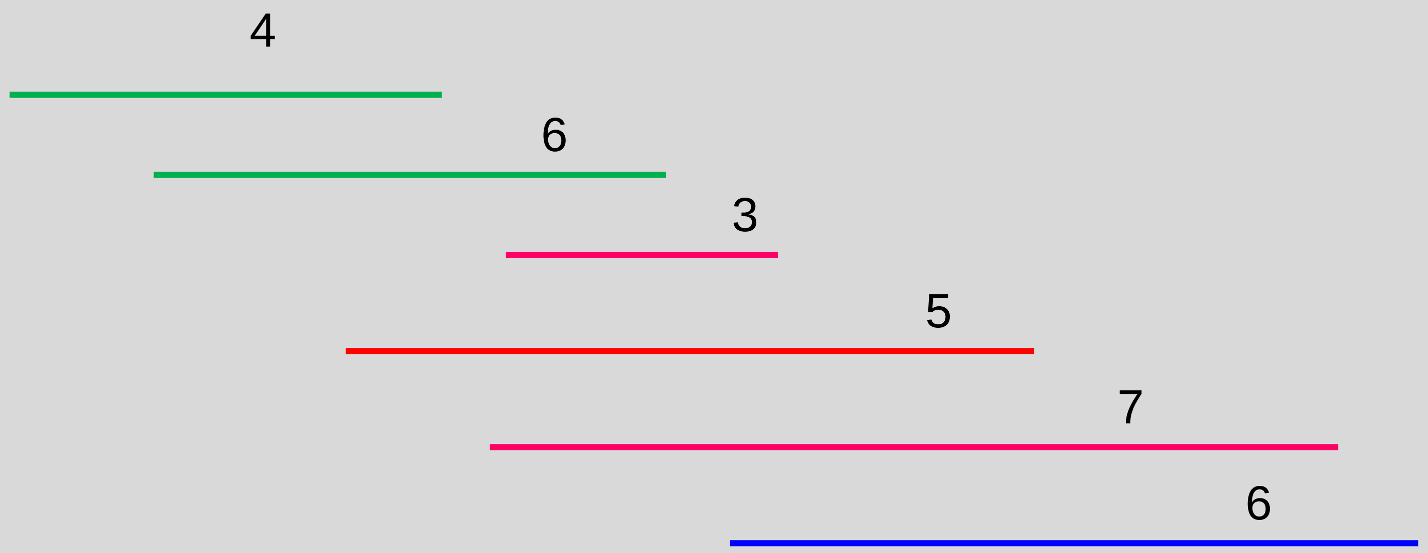
- Given a collection of intervals $I_1, \ldots, I_n$ with weights $w_1, \ldots, w_n$, choose a maximum weight set of non-overlapping intervals



Intervals sorted by end time

Recursive Solution

Express the solution to a problem of size n in terms of solutions to problems of size k, where k < n



Intervals sorted by end time

Optimality Condition

- Opt[j] is the maximum weight independent set of intervals $I_1, I_2, \ldots, I_j$
- Opt[j] = max(Opt[j – 1], w_j + Opt[p[j]])
  - Where p[j] is the index of the last interval which finishes before I_j starts
- Convert to iterative algorithm to compute: Opt[1], Opt[2], Opt[3],…, Opt[n-1], Opt[n]

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];
}
```

How many different ways can I walk to work?

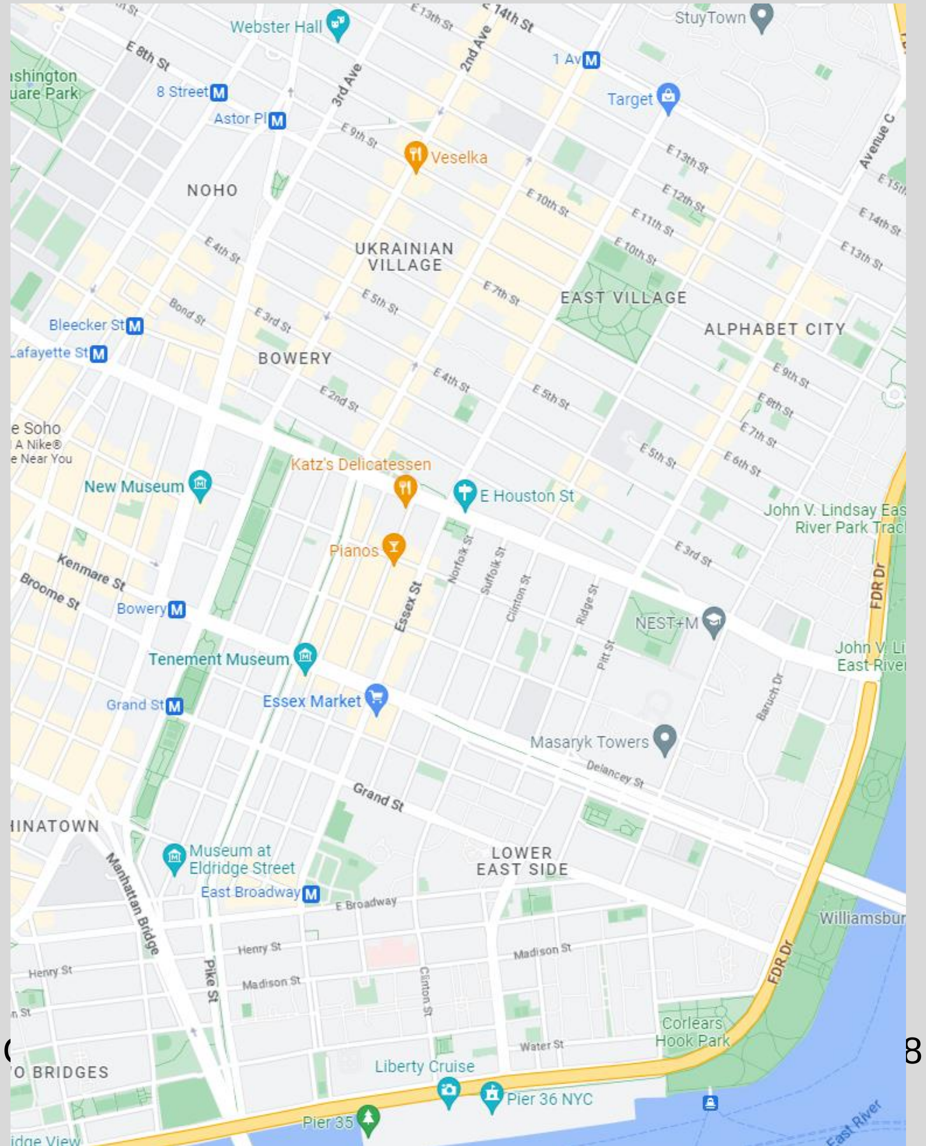
Only taking “efficient” routes

Make the problem discrete

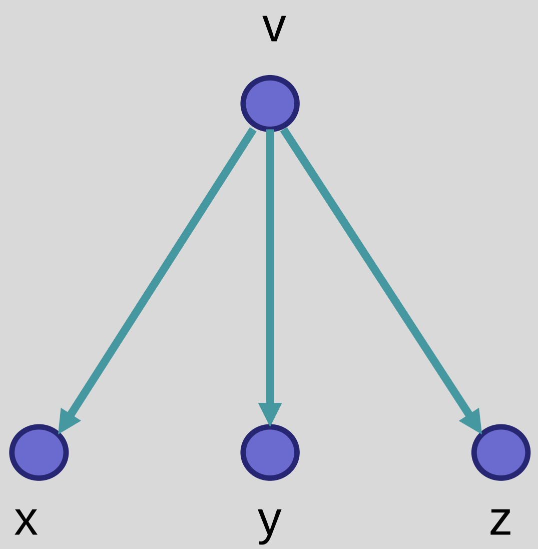
Directed Graph model: Intersections and streets

Assume the graph is a directed acyclic graph (DAG)

Problem: compute the number of paths from vertex h to vertex w



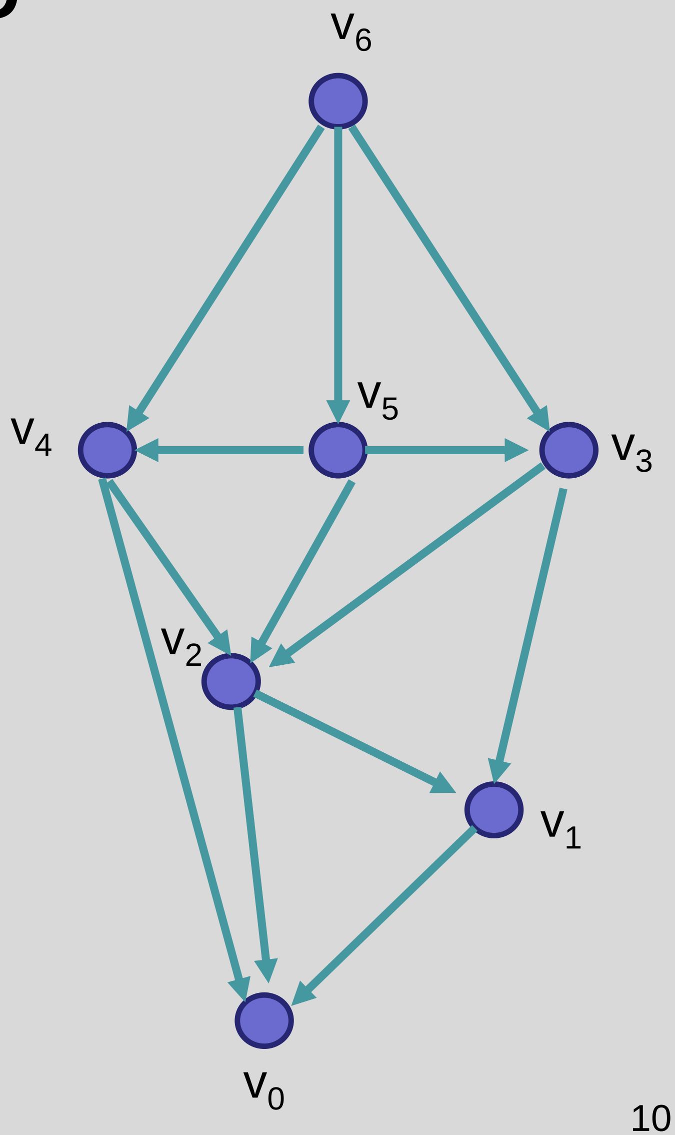
P[v]: Number of paths from v to v_0



How do you compute P[v] if you know P[x], P[y], and P[z]?

Recursive Algorithm

```
PC(v){
    if (v == $v_0$)
        return 1;
    count = 0;
    foreach (w in $N^+(v)$){
        count = count + PC(w);
    }
    return count;
}
```



Ordering the vertices

How do you order the vertices of a DAG such that if there is an edge from v to w, w comes before v in the ordering?

Path Counting

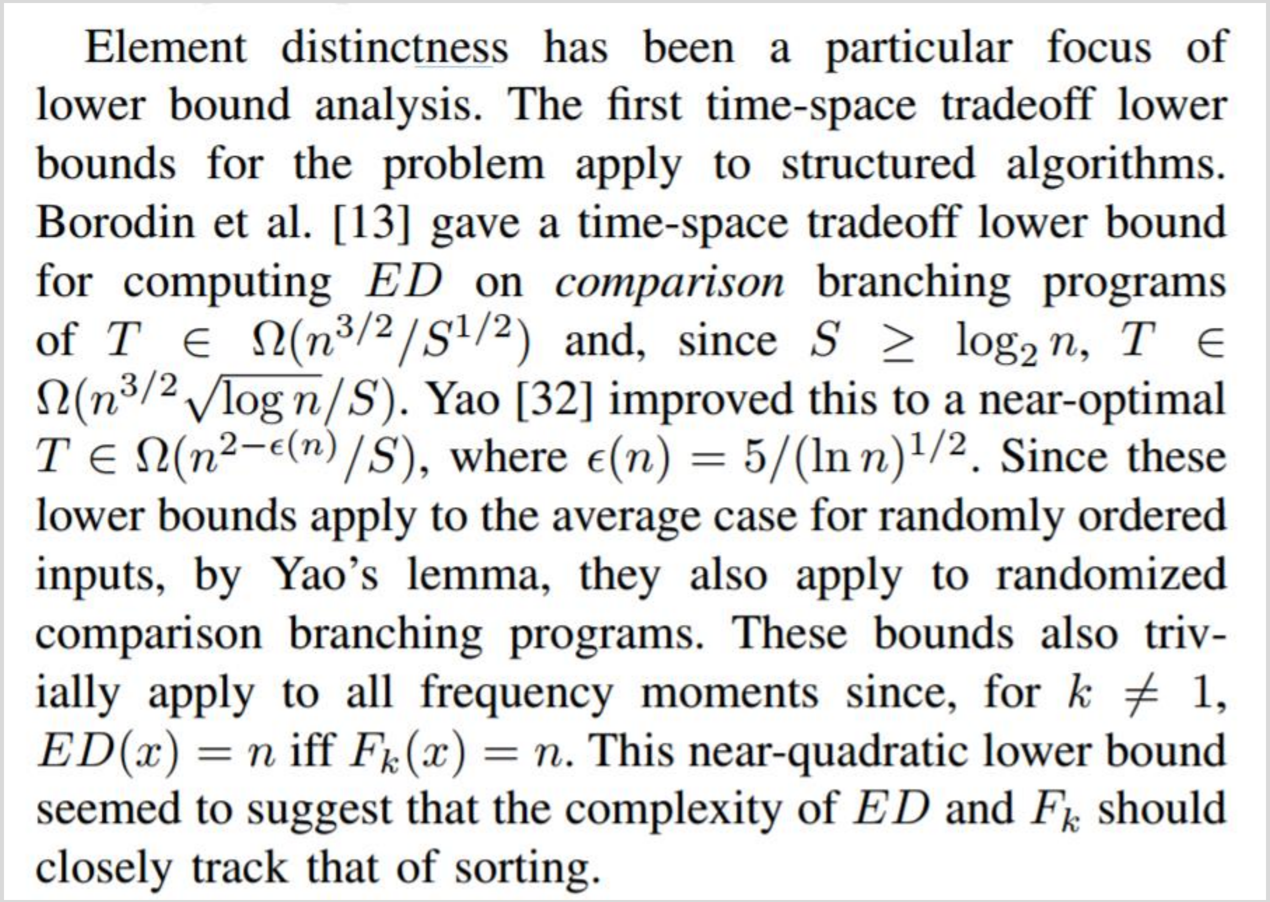
G=(V,E) is an n node directed acyclic graph, with v_{n-1}, v_{n-2}, . . ., v_1, v_0 a topological order of the vertices. An array is computed giving the number of paths from each vertex to v_0.

```
CountPaths(G, P){
    P[0] = 1;
    for (i = 1 to n-1){
        P[i] = 0;
        foreach (w in $N^+(v_i)$){
        P[i] = P[i] + P[w];
    }
}
```

Typesetting

- Layout text on a page to optimize readability and aesthetic measures
- Skilled profession replaced by computing
- Goal – give text a uniform appearance which is primarily done by choosing line breaks to balance white space
  - Interword spacing can stretch or shrink
  - Hyphenation is sometimes available

Optimal line breaking



Element distinctness has been a particular focus of lower bound analysis. The first time-space tradeoff lower bounds for the problem apply to structured algorithms. Borodin et al. [13] gave a time-space tradeoff lower bound for computing ED on *comparison* branching programs of $T \in \Omega(n^{3/2}/S^{1/2})$ and, since $S \geq \log_2 n$, $T \in \Omega(n^{3/2}\sqrt{\log n}/S)$. Yao [32] improved this to a near-optimal $T \in \Omega(n^{2-\epsilon(n)}/S)$, where $\epsilon(n) = 5/(\ln n)^{1/2}$. Since these lower bounds apply to the average case for randomly ordered inputs, by Yao's lemma, they also apply to randomized comparison branching programs. These bounds also trivially apply to all frequency moments since, for $k \neq 1$, $ED(x) = n$ iff $F_k(x) = n$. This near-quadratic lower bound seemed to suggest that the complexity of ED and F_k should closely track that of sorting.

Optimal Line Breaking

- Words have length w_i, line length L
- Penalty related to white space or overflow of the line
  - Quadratic measure often used
- Pen(i, j): Penalty for putting w_i, w_{i+1},…,w_j on the same line
- Opt[m]: minimum penalty for ending a line with w_m

The quick brown
fox jumped over
the   lazy   dog.

The quick brown
fox   jumped
over the lazy dog.

Pen(“The quick brown”) = 1
Pen(“fox jumped over”) = 2
Pen(“fox jumped”) = 8
Pen(“the lazy dog”) = 6
Pen(“over the lazy dog.”) = 4

Pen(i, j): Penalty for putting w_i, w_{i+1},…,w_j on the same line

Optimal Line Breaking

Optimal score for ending a line with w_m

$$Opt[m] = \min_i \{ Opt[i] + Pen(i+1,m) \} \text{ for } 0 < i < m$$

For words $w_1, w_2, \ldots, w_n$, we compute Opt[n] to find the optimal layout

Optimal Line Breaking

```
Opt[0] = 0;
for m = 1 to n {
    Find i that minimizes Opt [ i ] + Pen(i+1,m);
    Opt[m] = Opt [ i ] + Pen(i+1,m);
    Pred[m] = i;
}
```