CS 325 Recitation Session 1

Chirag Kaudan

April 2025

1 Introduction

- Introduction: what are recitation sessions?
- Survey of how many people are confused on lecture topics so far
- Problem solving for dynamic programming, more examples for experience
- High level discussion over the principles of dynamic programming using a concrete problem as an example

2 Weighted Interval Scheduling

Basic Setup: We have a set of n requests labeled $\{1, 2, \ldots, n\}$ where the ith request corresponds to an interval from s_i, the start time, to f_i, the finish time, of that request. Additionally, each request i has a certain value v_i.

Definition 1. *Two requests are compatible if they do not overlap. A set of requests is compatible if no two of them overlap.*

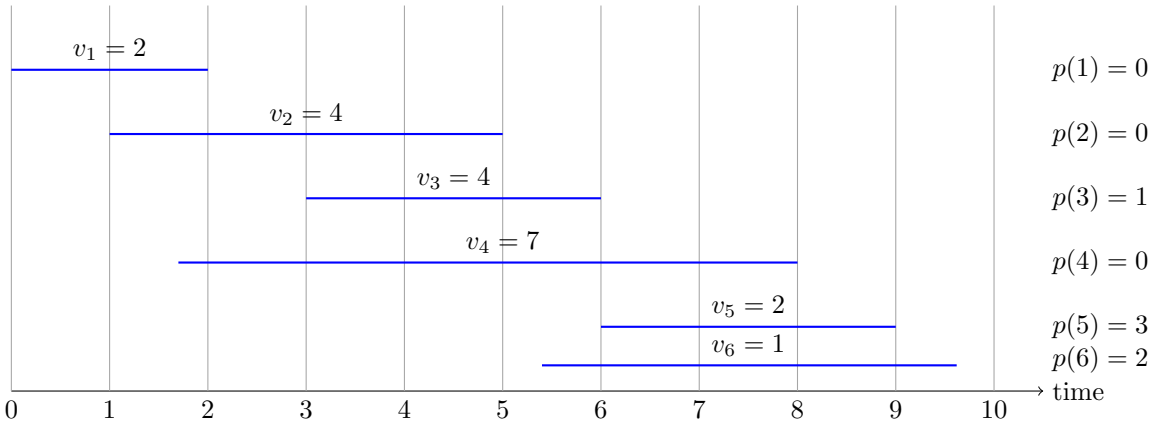
> **Problem Statement:** Given a set of n requests labeled $\{1, 2, \ldots, n\}$ with the start times, finish times, and values for each request, find a compatible subset S of intervals with maximum weight $\sum_{i \in S} v_i$.

We will suppose the requests are sorted in order of nondecreasing finish time: $f_1 \leq f_2 \leq \cdots \leq f_n$ and we say that a request i comes *before* a request j if $i < j$ in this ordering. We introduce some more notation to make it easier to talk about the intervals.

Definition 2. *For an interval j, we define $p(j)$ to be the largest interval i such that $f_i < s_j$. That is, $p(j) = i$ is the rightmost interval that ends before j begins. Define $p(j) = 0$ if no interval before j is disjoint from j.*

2.1 Example of Setup/Definitions

Example of the setup and $p(j)$



2.2 Designing Recurrence

- Give them some time to think about the problem, potential approaches, etc.
- Introduce recursive thinking; consider an optimal solution $\mathcal{O}$ for the given input $\{1, 2, \ldots, n\}$ and times/values
- Either interval $n \in \mathcal{O}$ or not...encourage them to think about this binary choice
- If $n \in \mathcal{O}$ then no interval after $p(n)$ (i.e. no interval in the set $\{p(n), p(n)+1, \ldots, n-1, n\}$) can also be in $\mathcal{O}$ by the definition of $p(n)$.
- Further, $\mathcal{O}$ must contain within it an optimal solution to the set of interval $\{1, 2, \ldots, p(n)\}$. **Why?**
- Now if $n \notin \mathcal{O}$, then $\mathcal{O}$ is simply equal to the optimal solution to the set $\{1, 2, \ldots, n-1\}$. **Why?**
- So in either case, we can get optimal solution $\mathcal{O}$ by looking at optimal solutions for smaller problems of the form $\{1, 2, \ldots, j\}$; this is a recurrence
- Let's introduce some definitions to refer to these subproblems

  Definition 3. *For j between 1 and n, let $\mathcal{O}_j$ be the optimal solution to the problem consisting of the requests $\{1, 2, \ldots, j\}$ and let $\mathrm{OPT}(j)$ denote the value of this solution. We define $\mathrm{OPT}(0) = 0$ since that is the optimal over the empty set of intervals*

- The optimal solution we are after is exactly $\mathcal{O}_n$ with value $\mathrm{OPT}(n)$

The reasoning above generalizes not just to n but for any j; for $\mathcal{O}_j$ on the set $\{1, 2, \ldots, j\}$, this reasoning suggests that if $j \in \mathcal{O}_j$, then $\mathrm{OPT}(j) = v_j + \mathrm{OPT}(p(j))$. And if $j \notin \mathcal{O}_j$, then $\mathrm{OPT}(j) = \mathrm{OPT}(j-1)$. These are exactly the two choices we have (binary choice), so

$$\mathrm{OPT}(j) = \max\{v_j + \mathrm{OPT}(p(j)), \mathrm{OPT}(j-1)\}$$

But how do we actually decide if some request j belongs to the optimal solution $\mathcal{O}_j$? This is not difficult:

Request j belongs to an optimal solution on the set $\{1, 2, \ldots, j\}$ if and only if

$$v_j + \mathrm{OPT}(p(j)) \geq \mathrm{OPT}(j-1)$$

Just this observation is powerful; it directly gives us a recursive algorithm for computing $\mathrm{OPT}(n)$, assuming that we have already sorted the intervals by finishing time and computed the $p(j)$ for each j. Perhaps

you can see that sorting and computing the $p(j)$ values can be done quickly, in $\Theta(n \log n)$ time (exercise for the reader).

Algorithm 1: ComputeOpt(j)

```
if j = 0 then
    return 0
else
    return max{v_j + ComputeOpt(p(j)), ComputeOpt(j − 1)}
```

Briefly ask them to prove the correctness of this algorithm (should be intuitively obvious, formally requires induction and that is useful to see that the order in which subproblems are solved is important)

Congratulations, we have a correct algorithm to solve the problem that used recursive thinking to come up with a solution to. But this is a terribly slow algorithm; it is exponential in the worst case

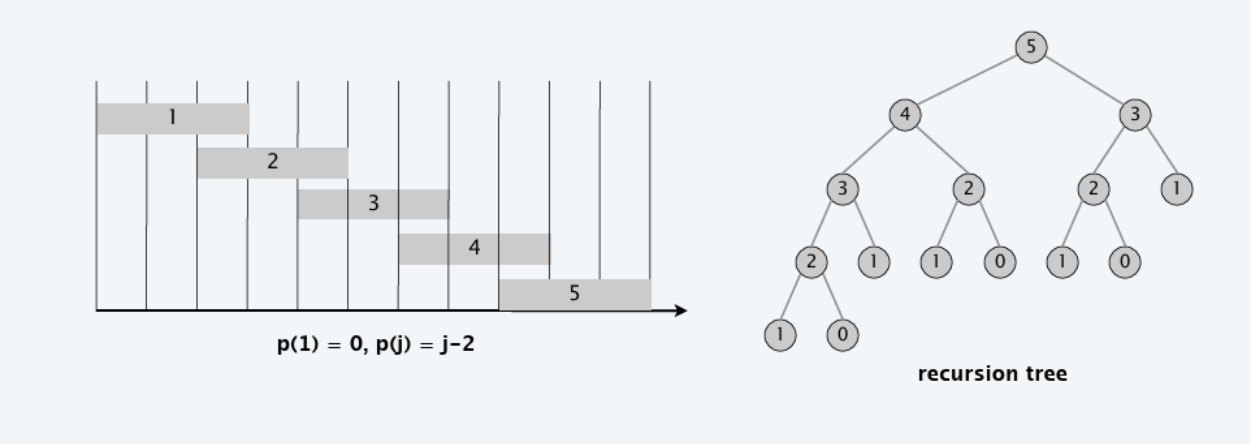


Figure 1: Diagram from Kleinberg and Tardos' Algorithm Design showing an undesirable tree of recursive subcalls, we don't want this many subcalls if we want to solve things quickly!

This grows like the Fibonacci numbers (should have seen in class); specifically, the number of vertices in this recursive call tree is $\Omega(2^n)$

2.3 Memoization

But the following observation explains why working through an exponential number of subproblems is strange:

Observation: There are only $n+1$ distinct subproblems involved here; precisely the subproblems $\text{OPT}(0), \text{OPT}(1), \ldots, \text{OPT}_n$. If we saved the answer to each subproblem in some globally accessible place the first time we computed it, we could simply use this pre-computed value to avoid recalculating it from scratch when it is called in the future. This technique is called memoization.

We can implement memoization fairly easily to turn our current recursive algorithm into a memoized recursive algorithm, which we will see is much more efficient. The idea is to use an array $M[0 \ldots n]$ to store values of $\text{OPT}(j)$ in $M[j]$ the first time you compute its value.

Algorithm 2: MComputeOpt(j)

```
if j = 0 then
    return 0
if M[j] is not empty then
    return M[j]
else
    Define M[j] = max{v_j + MComputeOpt(p(j)), MComputeOpt(j − 1)}
    return M[j]
```

Clearly this algorithm is correct, since it is nearly the same algorithm as ComputeOpt, but the running time is way better than exponential.

Theorem 1. *The running time of* MComputeOpt(n) *is* $O(n)$*, assuming the input is sorted by finishing times already.*

Proof. The running time for a call to MComputeOpt is $O(1)$ if we are excluding any recursive calls it may make. Thus, the total running time of MComputeOpt is bounded by a constant times the number of calls issued by MComputeOpt. Consider the number of entries of M that are not empty. Initially, this number is 0, but every time the algorithm uses the recurrence, where it issues two recursive calls to MComputeOpt, it fills an entry of the table. Since there are only $n + 1$ entries of the table, the algorithm can make at most $O(n)$ calls to MComputeOpt. Therefore, the running time of MComputeOpt(n) is $O(n)$. □

2.4 Bottom Up/Iterative Version

The algorithm we have above is perfectly desirable; it runs in linear time and uses linear space, but notice that it is recursive and many people prefer having iterative algorithms. We can talk about the advantages and disadvantages of the memoized recursive/top down approach versus the bottom up/iterative approach, but you should see both.

The main reason is conceptual: thinking about dynamic programming as iterating over subproblems is sometimes more natural and general.

The main thing to realize about the algorithm we developed is that the array M is the key. If we have this array, we're done; $M[n]$ is our final answer and the above formula guarantee that M is populated with the correct optimal values.

> **Observation:** The key to our dynamic programming solution is really this array $M[0 \dots n]$. We can directly compute the values of this array iteratively, without memoized recursion.

Algorithm 3: IterativeComputeOpt

```
Create array M[0 ... n]
Set M[0] = 0
for i from 1 to n do
    M[i] = max{v_i + M[p(i)], M[i − 1]}
```

One advantage is that the running time analysis becomes even easier; we need not talk about the number of recursive calls in the procedure anymore. This algorithm explicitly runs for n iterations, spending constant time per each iteration, so it's running time is $\Theta(n)$.

Some people find memoized recursion more natural, and unlike the iterative version, you won't necessarily end up solving all subproblems with memoized recursion, only the ones that are necessary. But some cons are that recursive calls use up the stack on your hardware, and that you cannot forget about old, unused states, unlike in bottom-up. Often times, we can save the space requirements

2.5 Recovering Actual Solution In Addition to its Value

What if we wanted the actual set of intervals, and not just the value of the maximum compatible interval selection? How could we go about finding it?

- **Idea 1:** Naively maintain another array S where $S[i]$ actually stores the optimal solution set of intervals from $\{1, 2, \ldots, i\}$.
- This is bad because it takes $O(n)$ time to write this set out every time instead of the $O(1)$ constant time to update an entry of the table M
- **Idea 2:** Do not explicitly maintain S, but recover the optimal solution through values in the M array after the optimal value has been computed

We know from a previous observation that request j belongs to an optimal solution on the set $\{1, 2, \ldots, j\}$ if and only if

$$v_j + \text{OPT}(p(j)) \geq \text{OPT}(j-1)$$

So after the algorithm terminates, we can run through the table M and for each entry of the table, we can determine which option lead to the value in that entry. We therefore know which intervals are chosen by the optimal solution.

Algorithm 4: FindSolution(j)

Input : The array M of optimal values to subproblems
if $j = 0$ **then**
  Output nothing
Let S be the set of intervals in an optimal solution
if $v_j + M[p(j)] \geq M[j-1]$ **then**
  Output j along with the set returned by FindSolution(p(j))
else
  Output the set returned by FindSolution($j-1$)

And by a similar argument, we can show that FindSolution runs in $O(n)$ time.

2.6 Principles of Dynamic Programming: High Level Discussion Using W.I.S as an Example

It is this bottom up, iterative approach that captures the spirit of dynamic programming most explicitly.

Ingredients for dynamic programming:

- A collection of subproblems related to the original problem
- Should be small number of them (polynomial number of them)
- There is some natural ordering on the subproblems from "smallest" to "largest" and an easy recursive formula to compute the value of subproblems given the values of small subproblems

There is real subtlety in designing dynamic programming algorithms. Sometimes it helps conceptually to think about a recursive formula and then find the set of subproblems that are required to unravel that recurrence, but other times it helps more to first think about a natural set of subproblems and then find a recurrence between them. Fortunately, many of the dynamic programming problems you'll see in this course do not get so much more trickier than this, so this worked through example and informal guidelines should serve you well!

3 Ordering Subproblems and D.P DAGs

Section is much less detailed, written for TA as notes during recitation

- Importance of solving subproblems in particular order; associated DAG with every DP problem
- Subproblems correspond to conceptual vertices, dependencies between subproblems correspond to conceptual (directed) edges
- Facts about DAGs (they might not be familiar with basic properties of DAGs)
- Talk about shortest paths in DAGs, relationship to LIS