CSC373 Notes

Last updated January 20, 2024

1 Proofs

Why do particular algorithms run quickly? Why is it that traveling salesman takes forever, but something like sorting is quick? This course gives you the tools to help you design algorithms that are fast for solving these problems, and to give you what the features are that leads to something being fast or not.

We're trying to figure out whether an algorithm is efficient. There are two different types of efficiency:

- Weakly polynomial (the number)
- Strongly polynomial (no. of bits)

When can we find an example that runs in polynomial time and the structures of algorithms that end up running in polynomial time?

We have:

- Greedy
- Divide and conquer
- Network flow
- Dynamic programming
- Linear programming

These approaches are some of the most common approaches used to find examples for these algorithms that run in polynomial time.

We're also going to show example of problems we strongly suspect that do not run in polynomial time (NP hard or NP complete) – often by reducing them to instances of other NP-hard problems. For example, Super Mario Brothers is NP complete (you can build levels, and the level is beatable IFF the corresponding decision problem is correct). We'll see more fun examples along the way.

One of the things we're going to see is that some variants of problems are extremely easy, but when we change it a bit, it becomes extremely hard. Why?

Linear programming can be done in polynomial time, but if you change it into integer linear programming, all of a sudden, it becomes almost impossible to solve.

Randomized algorithms give us strategies where in case we get an NP problem, we argue how bad a random guess at a solution would be relative to the best case. In some cases, we can prove that the random solution isn't that bad.

This course is theoretical in nature. We'll drill down into the simplest problems.

1.1 How do I Structure Proofs?

Reduce everything down to a well-established proof structure. Putting the logic in a well-structured way makes errors easier to catch and makes it easier to be confident that your ideas are right. Use these structures:

- Induction, all forms
- Contradictions
  - Assume your claim isn't true, come up with an impossible statement
- Not of the two above
  - Beware, this is very error prone

How do I set up an inductive proof? Of the two common proof techniques, inductive proofs are easier to set up.

- Break the problem into a number of steps, $s(i)$
- Show the induction hypothesis holds for the base case $s(0)$, which is extremely easy to show
  - Show an array that contains one item is sorted
- Show that $s(i) \Rightarrow s(i+1)$

When you're proving something with induction, you should begin by drawing a picture. Figure out the structure of the problem before trying to prove it. When you see problems, you won't know how to solve them right away, but drawing some figures and

going through small examples makes the structure of the algorithm clearer. It makes it more obvious to set up the induction hypothesis, but it's rare to see it right away. **Don't try to do the prof without understanding the structure of the problem.**

1.2 Bubble sort Inductive Proof

Prove that one iteration of bubble sort puts the lowest element in the right.

How does this end up working? Induction on the array length

BASE CASE: $n = 1$. Our list is already sorted.

INDUCTION STEP:

Assume bubble sort will put the lowest number on the right for an array with $n - 1$ items.

Say we have a list A with n items and run our first iteration of bubblesort on $A[0 \ldots n-1]$. We'll end up with $\left[- \quad - \quad x\right][y]$

There are two cases:

Case 1: $x \leq y$: swap x and y, our last element would be x which is the smallest

Case 2: Do nothing, nothing happens, the right most element is the smallest

As we've covered all possible cases, we know that the induction step holds.

So by induction, our argument here works.

1.3 Bubble Sort Contradiction Proof

Assume that the opposite of the hypothesis was true.

Prove that one iteration of bubble sort puts the lowest element in the right.

Show that if the opposite were true then the assumptions of the problem would be violated.

The opposite is: (One iteration of bubble sort ran AND the lowest element isn't on the right)

Proof goes like this:

- Assume bubble sort fails and position i is first from right with $a[i] \leq a[i+1]$ (where $i = len(A) - 2$)
  - Here, we **assume a case of failure, that this was the result after the swap**
- If $a[i] \leq a[i+1]$, a swap should've happened but because we said that bubble sort failed, we don't swap. But we assumed that bubble sort must have run

I could really rephrase that as:

- Lowest element on the right $\Rightarrow$ One iteration of bubble sort couldn't have run

2 Divide and Conquer

2.1 Merge Sort

You know how it works. At least. Some preliminaries:

- Merging two sorted lists is $\mathcal{O}(n)$

Task: sort a list of integers $[1, 3, 4, 2]$

Merge sort works like this:

- Split A into L and R
- Call Mergesort on L and R
- Merge the two back together

I'm taking a problem, breaking them up into two, then I combine them back together.

2.1.1 Proving the Mergesort Algorithm

With induction: $P(n)$ means that Mergesort works for a list L of that size.

Base case: $P(2)$, there is no recursion, and we can very easily deduce that the list would be sorted.

Inductive step: For k that is a power of 2, assume $\forall i \in \{2, 4, \ldots, k\}, P(i)$. Prove $P(2k)$.

The list of size $2k$ will be split into two, so we will get two sorted lists coming in. Now, what we need to do is argue that merging two sublists works.

So really, here's our proof for the merge algorithm:

$$\text{Merge ran} \Rightarrow \text{Merged list is sorted}$$

Let's prove the contrapositive:

$$\neg\text{Merged list is sorted} \Rightarrow \neg\text{Merge ran}$$

Now, let's look at the merging algorithm again, and for this one concrete example: suppose that `merge` does this.

$$[4, 5][6, 7] \downarrow$$
$$[4, 6, 5, 7] = S$$

`Merge` couldn't have done this. Let's look at the first location on the list that isn't correctly sorted: at index 1, 6.

Because the list isn't sorted, $\exists i,\ S[i] > S[j],\ \ j > i$. If we assume this, we end up going through the list element by element and compare the two results. When we're breaking up the two list, we sort them and iterate over their positions.

This argument goes: if we actually run merge, we would get $[4, 5, 6, 7]$. Moreover, for $[4, 6, 5, 7]$ to be put as the output, $S[j] = 5 > S[i] = 6$, which is impossible according to the merge algorithm. This means that the merge algorithm couldn't have ran.

2.2 Master Theorem

I've seen it before. It's an amazingly powerful result. What it is, us that it gives you the asymptotic scaling for the solution of a recurrence relation. This is hugely important for divide and conquer algorithms, as they always have a recursive structure to it.

The cost of Mergesort is:

$$T(n) \leq 2\,T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

It goes by this:

Let $a \geq 1,\ b > 1$ be constants, $T(n)$ be defined on $\mathbb{Z}^{\geq 0}$, and $\frac{n}{b}$ can be $\left\lceil \frac{n}{b} \right\rceil$, and let $T(n)$ be:

$$T(n) \leq aT\left(\frac{n}{b}\right) + f(n)$$

Let $d = \log_b(a)$.

1. If $f(n) \in \mathcal{O}\left(n^{d-\varepsilon}\right)$ for some $\varepsilon > 0,\ T(n) = \Theta\left(n^d\right)$
2. If $f(n) \in \Theta\left(n^d \log^k(n)\right)$ for some $k \geq 0$, $T(n) = \Theta\left(n^d \log^{k+1} n\right)$
3. If $f(n) \in \Omega\left(n^{d+\varepsilon}\right)$ for some $\varepsilon > 0$, $T(n) = \Theta\left(f(n)\right)$

For case 2: Ideally, you want k as close to 0; 0 gives us the least upper bound. Picking a higher k gives a looser upper bound which we probably don't want.

For Mergesort, we have $a = 2,\ b = 2 \Rightarrow d = \log_2(2) = 1$.

In this case, we know that $f(n) \in \mathcal{O}(n)$. Case 1 can't be true: $f(n) \notin \mathcal{O}\left(n^{1-\varepsilon}\right)$. Yet, case 2 work.

Case 2: $f(n) \in \Theta\left(n^1 \log^0 n\right) = \Theta(n) \Rightarrow T(n) \in \Theta\left(n^d \log^{k+1} n\right)$

WARNING: If you're using a specific master theorem, let the TAs know which one you're using.

2.3 Counting Inversions

Given an array a of length m, count the number of pairs (i, j) such that $i < j$ but $a[i] > a[j]$.

An inversion is a pair of elements opposite in the sorted order: e.g. [1, **3**, **2**, 6, **9**, **8**]. A list that is sorted in the reverse order would have around n^2 inversions.

The brute-force algorithm would take $\mathcal{O}\left(n^2\right)$ time.

Let's divide and conquer count inversions on the left, count inversions on the right, and then count the inversions that are between the left and the right.

Sorting kills all information about the inversions, so we want to store the output. That's why this algorithm has two outputs.

Here's how it works, SORT-AND-COUNT which returns (result, sorted list):

- Take a list L
- In the case of a base case, return $(0, L)$
- Break it in half. A, B
- Apply SORT-AND-COUNT, getting $\left(\underset{\text{NUM INVERSIONS}}{r_A}, A\right)$, (r_B, B) back
- Merge it back using the steps below

How do we count inversions (a, b) such that $a \in A, b \in B$? And how do we not merge sort over and over again?

We start with two halves, and we combine our inversion counting and sorting:

- Scan A and B LTR
- Compare a_i, b_j
- If $a_i \leq b_j$, do your merge step like usual
- Otherwise, if $a_i > b_j$, add 1 to the inversion count and do the merge step like usual

Meaning our runtime formula:

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$
$$\Rightarrow \mathcal{O}\left(n\log(n)\right)$$

If we just relied on Mergesort and done this iteratively, this cost would be $n\log(n)$ so the second bit of the master theorem, we couldn't do $k = 0$ and we would have a squared algorithm.

2.4 Closest Pair in R2

Given n points of the form (x_i, y_i) in the plane, find the closest pair of points. Assume that if I extract all x and y from all points I won't have duplicates.

Brute force: $\Theta\left(n^2\right)$. We'll have to look at every possible pair in the data. $\binom{n}{2} \in \mathcal{O}\left(n^2\right)$

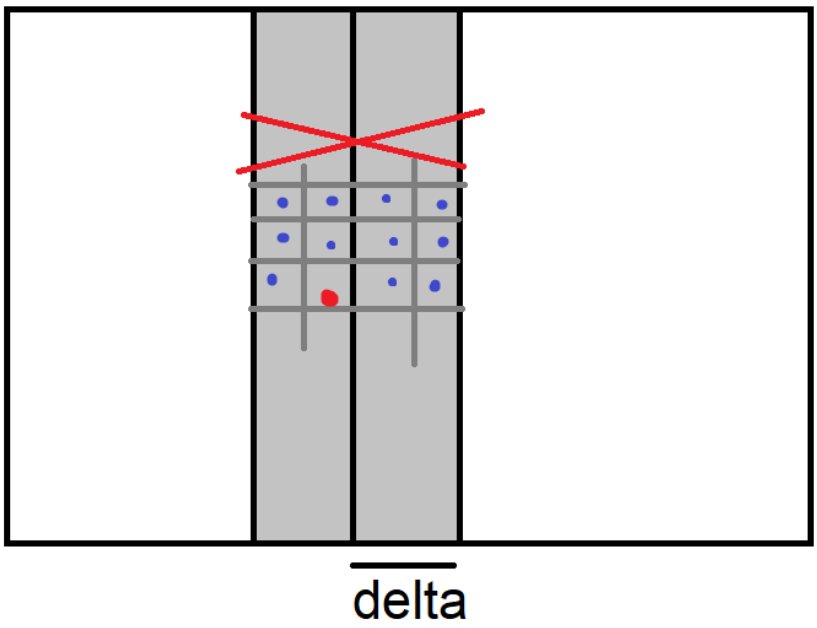
Divide and conquer what we do, is draw a line partway through the data such that half of the data is on the left, and half of the data is on the right. What I'll do, is I'll find the nearest pair on both sides then merge.

However, there's a big complication: merging is very hard.

What we need to do for the combine step, is to go through and check all points in the cross step. However:

Actually, we shouldn't think about doing our comparisons for everything. Reason why, is that:

δ is the lower of the shortest distance between two points on the left, on the right



We can save ourselves a bit of trouble and restrict our attention to our values in the δ region. Yet, we could have a situation where the large fraction of our points could lie inside this region.

Actually, something cool ends up happening. When you look at the sorted list of points, the maximum no. of points you need to compare on the opposite side is a constant. We only need to look at **11** points (I could skip work if the points are on the same side – the POINT of this is that it's a constant no. of points).

Let's break everything up.

Claim: If two points are at least 12 positions apart in the sorted list, their distance is at least δ

Proof: No points lie in the same $\frac{\delta}{2} \times \frac{\delta}{2}$ box

- If two of them were in the same cell, the maximum distance between the two corners is $\frac{\delta}{2}\sqrt{2} = \frac{\delta}{\sqrt{2}} < \delta$
- If two of them happen to show up in the same cell, the distance must be less than δ, but δ is the smallest distance between the two

So, what are the total comparisons that we need to do?

Merging takes $\mathcal{O}(n)$.

So, running time analysis.

- Finding points on strip: $\mathcal{O}(n)$
- Sorting by y coord: $\mathcal{O}\left(n \log(n)\right)$
- Testing against 11 points: $\mathcal{O}(n)$

Runtime:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + \mathcal{O}\left(n \log(n)\right)$$

Master theorem tells us that the cost is $\mathcal{O}\left(n \log^2(n)\right)$

By using divide-and-conquer, we can speed up our algorithm by a lot.

We can improve this to $\mathcal{O}\left(n \log(n)\right)$ by doing a simple sort by y-coordinates at the start.

2.5 Karatsuba's Algorithm

Divide each integer into two parts:

$$x = x_1 \cdot 10^{\frac{n}{2}} + x_2,\ y = y_1 \times 10^{\frac{n}{2}} + y_2$$

$$xy = (x_1 y_1)\, 10^n + (x_1 y_2 + x_2 y_1)\, 10^{\frac{n}{2}} + (x_2 y_2)$$

For $\frac{n}{2}$-digit multiplications can be replaced by three:

$$x_1 y_2 + x_2 y_1 = (x_1 + x_2)\,(y_1 + y_2) - x_1 y_1 - x_2 y_2$$

We can reuse multiplications that we've already done.

Runtime:

$$T(n) \leq 3T\left(\frac{n}{2}\right) + O(n) \Rightarrow T(n) = \mathcal{O}\left(n^{\log_2(3)}\right)$$

Most of these multiplication algorithms have great scaling but they have horrible constant factors. Karatsuba is an exception.

2.6 Strassen's Algorithm

Instead of multiplying numbers, let's look at multiplying matrices.

Imagine we have two "block" matrices:

$$C = A \cdot B = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Naively, this requires two multiplications of size $\frac{n}{2}$.

However, some of these multiplications, we can memorize and replicate after doing a clever substitution. By doing that, you would replace the 8 multiplications you would normally do with 7:

$$T(n) \leq 7T\left(\frac{n}{2}\right) + \mathcal{O}\left(n^2\right) \Rightarrow T(n) = \mathcal{O}\left(n^{\log_2(7)}\right)$$

The lower bound for matrix multiplication is $\Omega\left(n^2\right)$, but in practice it's $\mathcal{O}\left(n^{2.37}\right)$. It's interesting because the gap between n^2 and what we have right now isn't known. The problem with finding an optimum is really theoretically interesting.

2.7 Kth Smallest / Quick Select

Sorting it, find kth element, or using a heap isn't that quick. It won't be better than $n \log(n)$. Good news – selection is easier than sorting.

`QuickSelect` goes like follows:

- Find a pivot p
- Divide A into two sub-arrays: A_{less} = elements $\leq p$, A_{more} = elements $> p$
- If $|A_{\text{less}}| \geq k$, return kth smallest in A_{less}, else find $(k - |A_{\text{less}}|)$-th smallest element in A_{more}

Problem? If the pivot is close to the min or the max, we would get $T(n) \in \mathcal{O}\left(n^2\right)$. Best to find a pivot that evenly distributes the two elements.

So, are we going to select a pivot randomly? We want to do things deterministically. Select the average? That could be subject to outliers. Best solution?

2.7.1 Median of Medians

- Divide n elements into $\frac{n}{5}$ groups of 5 each
- Find the median of each group
  - Takes $\mathcal{O}(n)$ to here
- Find the median of $\frac{n}{5}$ medians = p^*
  - Takes $T\left(\frac{n}{5}\right)$, **this has to be done recursively**
- Create A_{less} and A_{more} according to p^*
  - Takes $\mathcal{O}(n)$
  - $\frac{n}{10}$ of the $\frac{n}{5}$ medians are $\leq p^*$
  - For each such median, there are *at least* 3 elements $\leq p^*$
  - $\Rightarrow$ There can be at most $\frac{7n}{10}$ elements that can be $\geq p^*$
  - In the other direction, there can be at most $\frac{7n}{10}$ elements that can be $\leq p^*$
- **Our work here is done. But, reiterating the last point from Quick Select:** Run selection on one of A_{less} or A_{more}, however you did it before
  - Takes $T\left(\frac{7n}{10}\right)$

We get that

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + \mathcal{O}(n)$$

Note that $\frac{n}{5} + \frac{7n}{10} = \frac{9n}{10}$. Only a fraction of n, using a similar analysis to the one in the master theorem, $T(n) = \mathcal{O}(n)$

By the way, you don't need to memorize the specific algorithms – you just need to know how to use these basic ideas. It may be easier or harder than the memorization challenges.

You will be tested on the ability to use these tools.

3 Greedy Algorithms

Find a solution x maximizing or minimizing an objective function f. The challenge is when the space of possible solutions x is too large. Does it find the best solution possible, and how much time does it take to find the best solution possible?

Here's our approach:

- **Compute it one part at a time**
  - For each part, they will always pick the best solution right now **without regard for the future**
- Select the next part greedily to get the most immediate benefit (this needs to be defined carefully for each problem)
- Guarantees polynomial time
- Need to prove that this will always return an optimal solution despite having no foresight

Greedy algorithms don't always give the optimal solution! We'll play with some heuristics that we can prove its optimal, and then we can prove that the heuristic is in fact optimal.

Beware:

- Most of the time, greedy algorithms, will find one of many equivalently optimal solutions. Greedy algorithms will return one of the optimal solutions.

3.1 When should I use it?

When **all of the** following hold:

- Optimal substructure
  - Solving one problem contains optimal solutions to all other subproblems
  - An optimal solution at one level can be thought of adding onto an optimal solution to a smaller level.
  - Otherwise, greedy algorithms won't give you the best option.
- Greedy choice property
  - Among all choices, the greedy solution performs best

3.2 Interval Scheduling

Given some intervals (job j starts as s_j and finishes at f_j), two jobs are compatible in the same way how UofT timetables don't conflict. Find the maximum-size subset of mutually compatible jobs.

How do we do it?

- Always pick the task that finishes first.

Here comes the earliest finishing time heuristic.

For scheduling-type problems, there is a set of typical heuristics that could end up working well. Go through each heuristic and come up with a counterexample. If you come up with one, abandon strategy and try another.

Look for the one you can't prove is suboptimal, then prove that it is optimal.

What are the heuristics? Choose in the order of

- Earliest start time
  - Not – if the earliest task takes the entire day, blocking everything else out
- **Earliest finishing time (the best)**
- Shortest interval
  - Which happens to rule everything else out: two other tasks that one ends at noon, one starts at noon, yet the shortest job overlaps noon
- Fewest conflicts
  - Which happens to block everything else out; we can cram a bunch of other conflicts

You would struggle looking for a contradiction with earliest finishing time. You would switch over and take a look at the finishing time as the heuristic.

Before we improve its optimality, let's look at how long it will take for it to run.

To do earliest finishing time, we need to compute the earliest finishing time. The easiest way to do that is by sorting. To find the one that has the earliest initial starting time, just sort the list. Sorting takes $\mathcal{O}\left(n \log(n)\right)$. Tiebreaks do not matter.

We then start picking tasks according to that list. Look at all tasks that end later than our current task in the same order and choose the earliest compatible one. You might think this will take $\mathcal{O}\left(n^2\right)$ time, but no, because we only need to compare all future-ending intervals with our previous interval, it's constant time in total.

3.2.1 Interval Scheduling Proof of Optimality By Contradiction

- Assume what we're proving is false
- Derive an inconsistency

For a contradiction, **assume that greedy is optimal**

Suppose that greedy selects jobs $i_1, \ldots, i_k$ sorted by finish time

Consider an optimal solution $j_1, j_2, \ldots, j_m$ which matches greedy for as many indices as possible from the start (we want $j_1 = i_1, \ldots, j_r = i_r$ for the greatest possible r)

Both i_{r+1} and j_{r+1} must be compatible with the previous selection (the Greedy algorithm can never choose something invalid; so is the optimal solution)

What are we going to do with this?

Consider a new solution $i_1, i_2, \ldots, i_r, \mathbf{i_{r+1}}, j_{r+2}, \ldots, j_m$

We have replaced j_{r+1} with i_{r+1} in our reference optimal solution.

- A conflict is impossible: $j_r = i_r$ and greedy cannot select anything that conflicts
- i_{r+1}.finish $\leq$ j_{r+1}.finish. If that wasn't the case, greedy would've chosen j_{r+1} instead.
- j_{r+1}.finish $\leq j_{r+2}$.start, so, the right side of i_{r+1} is still compatible.

This results in a contradiction: the new solution matches greedy for $r + 1$ intervals. This means that greedy is optimal, **as any step of it can be extended into an optimal strategy**.

Key fact:

Can be extended into an optimal strategy in any step $\Rightarrow$ Optimal

3.2.2 Proof of Optimality by Induction

Induction gives you a lot of scaffolding and gives you a structure to work off from. The structure of inductive proof gives you guides of what you should do.

Let S_j be the **subset of jobs picked by greedy after considering the first j jobs in the increasing order of finish time.**

- Define $S_0 = \emptyset$.

We call a partial solution *promising* if there is a way to extend it to an optimal solution by picking some subject of jobs $j + 1, \ldots, n$ (indices of jobs sorted by finish time)

- $\exists T \subseteq \{j+1, \ldots, n\}$ such that $O_j = S_j \cup T$ is optimal

Our inductive claim: $\forall t \in \{0, 1, \ldots, n\}$, S_t is promising.

If S_n is promising, then it must be optimal as there are no more jobs to consider.

Base case:

For $t = 0$, $S_0 = \emptyset$ is promising, because any optimal solution can extend this set.

Induction hypothesis: Suppose claim holds for $t = j - 1$ and optimal solution O_{j-1} extends S_{j-1}. WTP: O_j extends S_{j-1} with no issues

Induction step: At $t = j$, we have two possibilities.

On considering the jth job:

If greedy **did not** select job j:

- j must conflict with some job in S_{j-1}
- Since $S_{j-1} \subseteq O_{j-1}$, O_{j-1} cannot include job j either
- $O_j = O_{j-1}$ also extends $S_j = S_{j-1}$ (transitivity of the = sign)

If greedy **did** select job j:

- $S_j = S_{j-1} \cup \{j\}$
- Consider the earliest job r in $O_{j-1} \setminus S_{j-1}$
- Consider O_j obtained by replacing r with j in O_{j-1}
- What happens if $r \neq j$? We still need to show that there are no conflicts.
  - We know greedy can't select anything with conflicts, so there are no conflicts
  - Because greedy selects jobs in ascending order of finish time, job j's finish time must be **before** job r's finish time
- We know that O_j extends S_j with no issues (really obvious just read the above again).

This means that $\forall j \in \{0, \ldots, n\}$, S_j is promising.

3.3 What are the Arguments Saying?

Both proof methods make the same claim:

- The greedy solution after j iterations can be extended into an optimal solution $\forall j$

The same key argument is used:

- If the greedy solution after j iterations can be extended to an optimal solution, then the greedy solution after $j + 1$ iterations can be extended to an optimal solution as well

The difference:

- Induction: this is the induction step
- Contradiction: we take the greatest j for which the greedy solution can be extended to an optimal solution (where anything ahead **we assumed for a contradiction cannot**), and we derive a contradiction by saying that it can (by extending the greedy solution after $j + 1$ iterations).

3.3.1 Another Flavor of Greedy Proofs

Greedy stays ahead

Let $i_1, \ldots, i_k$ be the greedy solution sorted by finish time

Let $j_1, \ldots, j_m$ be an optimal solution sorted by finish time

Claim: $f_{i_r} \leq f_{j_r}$ $\forall r$

Proof:

Base case: $f_{i_1} \leq f_{j_1}$ because greedy sorts in finish time and will always initially choose the lecture that finishes the earliest.

Inductive step: Assume $f_{i_l} \leq f_{j_l}$ (inductive hypothesis). Show $f_{i_{l+1}} \leq f_{j_{l+1}}$.

By the greedy algorithm, i_{l+1} is the lecture that finishes the earliest that is compatible with f_{i_l}. Can $f_{i_{l+1}} > f_{j_{l+1}}$? No, as if it were, greedy would've selected $f_{j_{l+1}}$ (contradiction spotted). So, this means that our claim is proven.

Why does the claim imply Greedy is optimal?

Suppose greedy is not optimal ($k < m$).

By the claim, $f_{i_k} \leq f_{j_k}$.

However, $s_{j_{k+1}} \geq f_{j_k}$. But then, $s_{j_{k+1}} \geq f_{i_k}$, so $s_{j_{k+1}}$ must have been considered by the greedy algorithm. This is our contradiction.

3.4 Interval Partitioning / Scheduling Lectures In Rooms

Jobs j starts at time s_j and finishes at f_j. Two jobs are compatible if they don't overlap. Goal: group jobs into fewest partitions such that jobs in the same partition are compatible.

So, what heuristic should we use? Schedule, considering *this* ordering first:

- Finish time
- Shortest interval
- Fewest conflicts
- **Start time**
  - Start time is the one that works; everything else has counterexamples

So, the algorithm goes like this:

Input: a set of n lectures

- Sort lectures by start time
- $d = 0$ (number of allocated classrooms)
- For $j = 1$ to n:
  - If lecture j is compatible with some classroom (doesn't matter which one)
    - Schedule it in that classroom
  - Else:
    - Allocate a new classroom $d + 1$

        * Schedule lecture j there
        * $d = d + 1$
- Return the schedule

3.4.1 Runtime

Key step to check if a lecture is compatible with some classroom:

- Store classrooms in a priority queue, key = latest finish time of any lecture in the classroom

If lecture j compatible with some classroom?

- Same as, is $s_j \geq$ latest finish time of that classroom
  - Yes: add it, increase minimum key of that classroom to f_j
  - No: Create a new classroom, add lecture j, set key to f_j
- $\mathcal{O}(n)$ priority queue operations, $\mathcal{O}\left(n \log(n)\right)$ time

3.4.2 Proof of Optimality

This is a different style of proof than the one before. We're going to prove a lower bound: that it is not possible to do interval partitioning with fewer than some number of classrooms. Then, we need to show that our algorithm matches that number.

First, prove that there is no way I can do better than k (the depth). Then, I need to show that greedy will always give me k (the depth) classrooms. If greedy achieves that and there is no way to do better than that, then greedy is optimal.

Proof of optimality (lower bound) – I CAN'T DO BETTER THAN DEPTH:

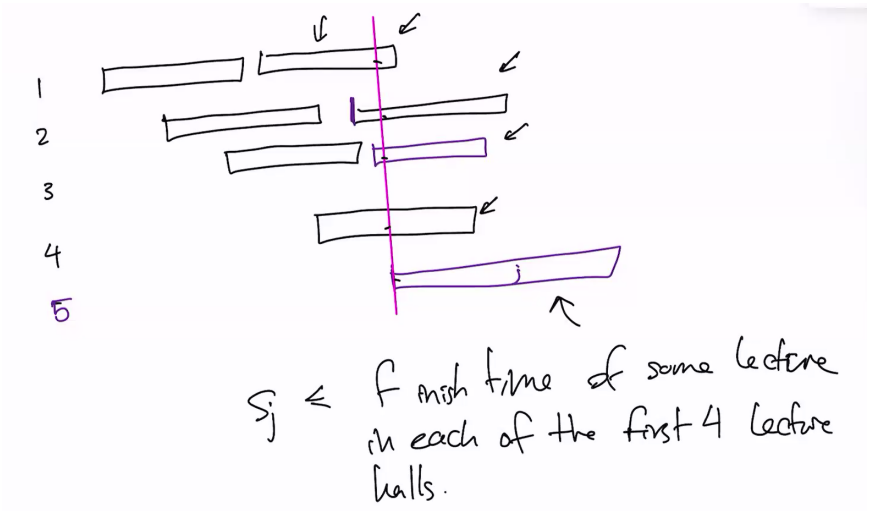
- WTS: d = Classrooms needed by greedy $\geq$ depth
  - Where depth = maximum no. of lectures running at any time
  - Job i runs in $[s_i, f_i)$

  - The rationale, is that if you have 4 classes running at the same time, you at least will have to pack them into four different classrooms

Now, we want to claim our greedy algorithm uses only these many classrooms

Proof of optimality (upper bound) – I CAN'T DO WORSE THAN DEPTH:

- Let d = no. classrooms used by greedy. Show that $d \leq$ depth.
- Classroom d was opened because there was a lecture j which was incompatible with some lecture already scheduled in each of $d - 1$ other classrooms.
  - All these d lectures end after s_j.
  - Since we sorted by the start time, they all start at or before s_j.
- $\Rightarrow$ So, at time s_j, we have d mutually overlapping (conflicting) lectures
  - **We're showing that greedy can also detect the minimum depth (max no. of concurrently conflicting lectures).** (With our lower bound, we can get rid of the word "minimum")
- Hence, depth $\geq d$ = no. classrooms used by greedy. In other words, **the depth (highest no. of concurrently conflicting lectures) is at most the no. of classrooms used by greedy.**
  - When I opened the dth classroom, the depth at that point was d. The depth cannot be smaller than d.
  - From the lower bound proof: the depth cannot be larger than d, as if it were so the classroom would've already been made for that.



Overall, we showed that depth $\geq d$ and $d \geq$ depth, then it can only be concluded that d = depth and greedy only uses as many classrooms as the depth.

3.5 Minimizing Lateness

Problem:

- We have a single machine
- Each job j requires t_j units of time and is due by time d_j
- If it's scheduled to start at s_j, it will finish at $f_j = s_j + t_j$
- Lateness: $l_j = \max\left(0,\ f_j - d_j\right)$
- Goal: figure out the ordering of the tasks that minimize the **maximum** lateness of any of my tasks: $L = \max_j l_j$. *This is not minimizing the sum of all task lateness!*

Contrast with interval scheduling:

- We decide the start time
- There are soft deadlines

Let's look at the greedy template:

- Shortest processing time: t_j ascending

  - Counter: $(t, d) : (1, 100), (10, 10)$ – opt: 0, this heuristic: 1

- Earliest deadline first: d_j ascending
  - It's this one
- Smallest slack first: $d_j - t_j$ ascending
  - Counter: $(t, d) : (1, 2, s = 1), (10, 10, s = 0)$ – opt: 1, this heuristic: 11