CSC373 Fall 2019
Lecture Notes (Archived)

Yuchen Wang

March 15, 2025

Contents

1 Divide & Conquer

General framework

1. Break (a large chunk of) a problem into smaller subproblems of the same type
2. Solve each subproblem recursively
3. At the end, quickly combine solutions from the subproblems and/or solve any remaining part of the original problem

1.1 Master Theorem

Useful for analyzing divide-and-conquer running time

Theorem Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence.

$$T(n) = aT(\frac{n}{b}) + f(n)$$

where we interpret $\frac{n}{b}$ to mean either $\lfloor \frac{n}{b} \rfloor$ or $\lceil \frac{n}{b} \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(\frac{b}{n}) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Note There are recurrence relations which do not fall under any of these cases (e.g. the recurrence relation $T(n) \leq T(n/5) + T(7n/10) + \mathcal{O}(n)$ from QuickSelect where the smaller instances are not of uniform size, or the recurrence relation $T(n) = \sqrt{n}T(\sqrt{n}) + n$ where a and b are not constants).

Theorem (from CSC236) Divide-and-conquer algorithms: partition problem into b roughly equal subproblems, solve, and recombine:

$$T(n) \begin{cases} k & \text{if } n \leq B \\ a_1 T(\lceil n/b \rceil) + a_2 T(\lfloor n/b \rfloor) + f(n) & \text{if } n > B \end{cases}$$

where $b, k > 0, a_1, a_2 \geq 0$, and $a = a_1 + a_2 > 0$. $f(n)$ is the cost of splitting and recombining. If f from the previous slide has $f \in \Theta(n^d)$, then

$$T(n) \in \begin{cases} \theta(n^d) & \text{if } a < b^d \\ \theta(n^d \log n) & \text{if } a = b^d \\ \theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

1.2 Counting Inversions

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

Applications

1. Voting theory
2. Collaborative filtering
3. Measuring the "sortedness" of an array
4. Sensitivity analysis of Google's ranking function
5. Rank aggregation for meta-searching on the Web
6. Nonparametric statistics (e.g., Kendall's tau distance)

Brute Force Check all $\theta(n^2)$ pairs

Divide & conquer

1. Divide: break away into two equal halves x and y
2. Conquer: count inversions in each half recursively
3. Combine:
   Solve (remaining): count inversions with one entry in x and one in y
   Merge: add all three counts

Count inversions (a, b) with $a \in A$ and $b \in B$, assuming A and B are sorted:

1. Scan A and B from left to right
2. Compare a_i and b_j

SORT-AND-COUNT (L)

IF list L has one element
  RETURN $(0, L)$.

DIVIDE the list into two halves A and B.
$(r_A, A) \leftarrow$ SORT-AND-COUNT(A).
$(r_B, B) \leftarrow$ SORT-AND-COUNT(B).
$(r_{AB}, L') \leftarrow$ MERGE-AND-COUNT(A, B).

RETURN $(r_A + r_B + r_{AB}, L')$.

3. If $a_i < b_j$, then a_i is not inverted with any element left in B
4. If $a_i > b_j$, then b_j is inverted with every element left in A
5. Append smaller element to sorted list C

How do we formally prove correctness? Induction on n is usually very helpful, allows you to assume correctness of subproblems

Running time analysis

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

Master theorem says this is $T(n) = O(n \log n)$

1.3 Closest Pair in $\mathbb{R}^2$

Problem Given n points of the form (x_i, y_i) in the plane, find the closest pair of points.

Applications

1. Basic primitive in graphics and computer vision
2. Geographic information systems, molecular modeling, air traffic control
3. Special case of nearest neighbor

Brute force running time

$$\Theta(n^2)$$

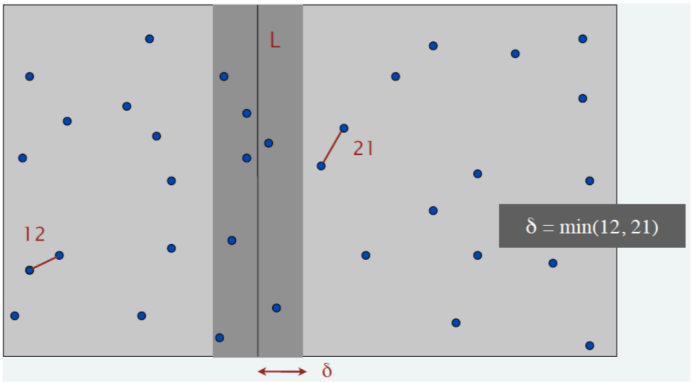
Intuition from 1D? By sorting and checking, the problem would be easily $\mathcal{O}(n \log n)$

Non-degeneracy Assumption No two points have the same x or y coordinate

Closest Pair in $\mathbb{R}^2$

1. Divide: points in equal halves by drawing a vertical line L
2. Conquer: solve each half recursively
3. Combine: find closest pair with one point on each side of L
4. Return the best of 3 solutions

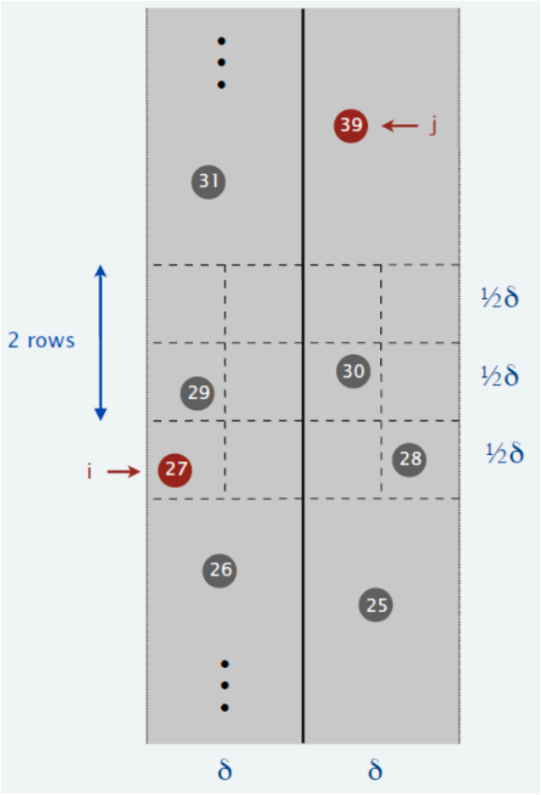
Combine: We can restrict our attention to points within ϵ of L on each side, where ϵ = best of the solutions in two halves



1. Only need to look at points within ϵ of L on each side
2. Sort points on the strip by y coordinate
3. Only need to check each point with next 11 points in sorted list

Why 11? Claim: If two points are at least 12 positions apart in the sorted list, their distance is at least ϵ.
proof:

1. No two points lie in the same $\delta/2 \times \delta/2$ box
2. Two points that are more than two rows apart are at distance at least δ



1.4 Recap: Karatsuba's Algorithm

Fast way to multiply two n digit integers x and y.

Brute force running time $\mathcal{O}(n^2)$

Algorithm

1. Divide each integer into two parts

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

$$xy = (x_1y_1) * 10^n + (x_1y_2 + x_2y_1) * 10^{\frac{n}{2}} + (x_2y_2)$$

2. Four $\frac{n}{2}$-digit can be replaced by three

$$x_1y_2 + x_2y_1 = (x_1 + x_2)(y_1 + y_2) - x_1y_1 - x_2y_2$$

Running time

$$T(n) = 3T(\frac{n}{2}) + \mathcal{O}(n)$$
$$\implies T(n) = \mathcal{O}(n^{\log_2 3})$$

1.5 Strassen's Algorithm

Generalizes Karatsuba's insight to design a fast algorithm for multiplying two $n \times n$ matrices

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

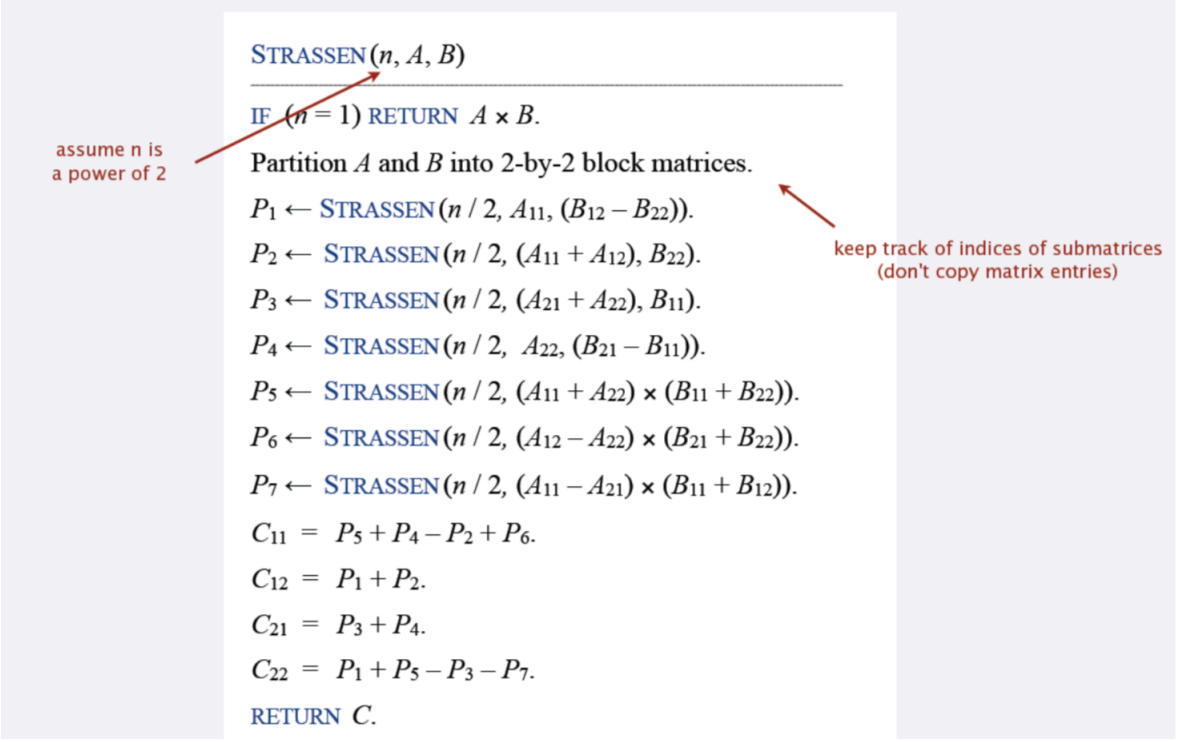
Call n the "size" of the problem. Naively, this requires 8 multiplications of size $\frac{n}{2}$, but by Strassen's insight, we can replace 8 multiplications by 7. (Assume n is a power of 2 because we need to recursively partition the matrix into 2-by-2 block matrices.)

Running time

$$T(n) = 7T(\frac{n}{2}) + \mathcal{O}(n^2)$$
$$T(n) \implies \mathcal{O}(n^{\log_2 7})$$

1.6 Median & Selection

Selection Given n comparable elements, find kth smallest.
minimum: $k = 1$
maximum: $k = n$
median: $k = \lfloor (n+1)/2 \rfloor$



STRASSEN(n, A, B)

IF $(n = 1)$ RETURN $A \times B$.

Partition A and B into 2-by-2 block matrices.

$P_1 \leftarrow$ STRASSEN$(n / 2, A_{11}, (B_{12} - B_{22}))$.

$P_2 \leftarrow$ STRASSEN$(n / 2, (A_{11} + A_{12}), B_{22})$.

$P_3 \leftarrow$ STRASSEN$(n / 2, (A_{21} + A_{22}), B_{11})$.

$P_4 \leftarrow$ STRASSEN$(n / 2, A_{22}, (B_{21} - B_{11}))$.

$P_5 \leftarrow$ STRASSEN$(n / 2, (A_{11} + A_{22}) \times (B_{11} + B_{22}))$.

$P_6 \leftarrow$ STRASSEN$(n / 2, (A_{12} - A_{22}) \times (B_{21} + B_{22}))$.

$P_7 \leftarrow$ STRASSEN$(n / 2, (A_{11} - A_{21}) \times (B_{11} + B_{12}))$.

$C_{11} = P_5 + P_4 - P_2 + P_6$.

$C_{12} = P_1 + P_2$.

$C_{21} = P_3 + P_4$.

$C_{22} = P_1 + P_5 - P_3 - P_7$.

RETURN C.

Different Implementations & their running time

1. $\mathcal{O}(n)$ compares for min or max
2. $\mathcal{O}(n \log n)$ compares by sorting
3. $\mathcal{O}(n \log k)$ compares with a binary heap

Applications

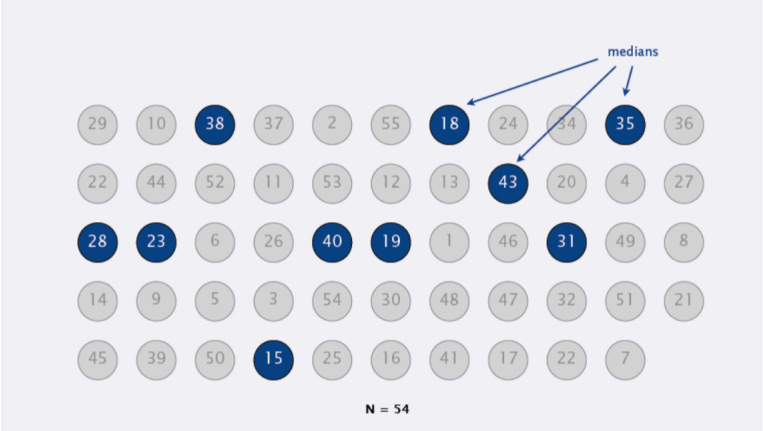
1. Order statistics
2. "top k"
3. bottleneck paths

Quick(Randomized) Select Selection is easier than sorting. Partially sort array relative to a pivot element, and look for the kth smallest in sub-array to the left or right of pivot. If partition only one element a time, then the algorithm would not be efficient. Therefore we need techniques to find a good pivot.

Look for *kth* smallest in array A[p..r]

```
QUICK-SELECT (A; p; r; k)
if p == r return A[p]                          // single element array, k must be 1.
q = QUICK-PARTITION(A; p; r)                   // A[p..q-1] <= A[q] <= A[q+1..r]
j =q-p+1                                       // k is size of p..q
if k == j return A[q]                          //  the pivot is kth smallest
elseif k < j return QUICK-SELECT(A;p;q-1; i)   // search in p..q-1
else return QUICK-SELECT(A;q+1;r;k –j)         // search in q+1..r
```

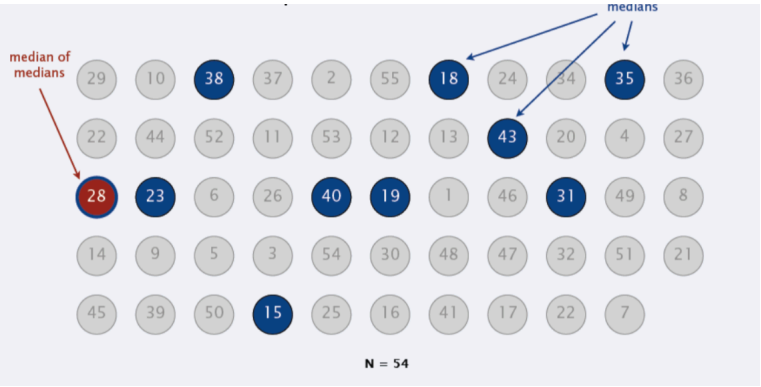
Finding a good pivot Algorithm as the following:
1. Divide n elements into $\lfloor \frac{n}{5} \rfloor$ groups of 5 elements each (plus extra)
2. Find median of each group (except extra)



3. Find median of medians recursively.
4. Use median-of-medians as pivot element.

Analysis of median-of-medians selection algorithm - 5 element case

1. At least half of 5-element medians $\leq p$
2. At least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ medians $\leq p$
3. At least $3\lfloor n/10 \rfloor$ elements $\leq p$



4. At least $3\lfloor n/10 \rfloor$ elements $\geq p$
5. QUICK-SELECT() called recursively with at most $n - 3\lfloor n/10 \rfloor$ elements
6. [not clear] $C(n)$ = max number of compares on an array of n elements.

$$C(n) \leq C(\lfloor \frac{n}{5} \rfloor) + C(n - 3\lfloor \frac{n}{10} \rfloor) + \frac{6}{5}n + n$$

$\frac{6}{5}n$: we need 6 comparisons to find the median among 5 elements
n: cost of QUICK-PARTITION()

Analysis of median-of-medians selection algorithm - general case
[not clear] Suppose we have m elements in each group.
Then we have approximately $\frac{n}{m}$ groups and $\frac{n}{2m}(1+\frac{m-1}{2})$ number of elements less than median-of-medians.
Then

$$T(n) = T(\frac{n}{m}) + T(n - \frac{n(m+1)}{4m})$$

Want

$$\frac{cn}{m} + cn > \frac{cn(m+1)}{4m}$$

$$\implies \frac{m+1}{4m} > \frac{1}{m} \implies m > 3$$

Choosing 4 involves floor and ceiling, so we choose 5 as the smallest number for which this algorithm can work.

1.7 Algorithm Design

Best algorithm for a problem is

1. Typically hard to determine (We still don't know best algorithms for multiplying two n-digit integers or two $n \times n$ matrices)

2. Usually, we design an algorithm and then analyze its running time, but some times we can do the reverse:
   E.g., if you know you want an $\mathcal{O}(n^2 \log n)$ algorithm, Master theorem suggests that you can get it by

   $$T(n) = 4T(\frac{n}{2}) + \mathcal{O}(n^2)$$

   So maybe you want to break your problem into 4 problems of size $\frac{n}{2}$ each, and then do $\mathcal{O}(n^2)$ computation to combine.

Access to input For much of this analysis, we are assuming random access to elements of input, so we are ignoring underlying data structures (e.g. doubly linked list, binary tree, etc.)

Machine operations We're only counting comparison or arithmetic operations, so we are ignoring issues like how real numbers will be represented in closest pair problem. When we get to P vs NP, representation will matter.

Size of the problem Can be any reasonable parameter of the problem. For example, for matrix multiplication, we used n as the size, while an input consists of two matrices with n^2 entries. It doesn't matter whether we call n or n^2 the size of the problem, because the actual running time of the algorithm won't change.

2 Greedy Algorithms

Outline We want to find a solution x that maximizes some objective function f, but the space of possible solutions x is too large. Since the solution x is typically composed of several parts (e.g. x may be a set, composed of its elements), instead of directly computing x,

1. Compute it one part at a time

2. Select the next part "greedily" to get maximum immediate benefit (this needs to be defined carefully for each problem)

3. May not be optimal because there is no foresight
4. But sometimes this can be optimal too!

2.1 Interval Scheduling

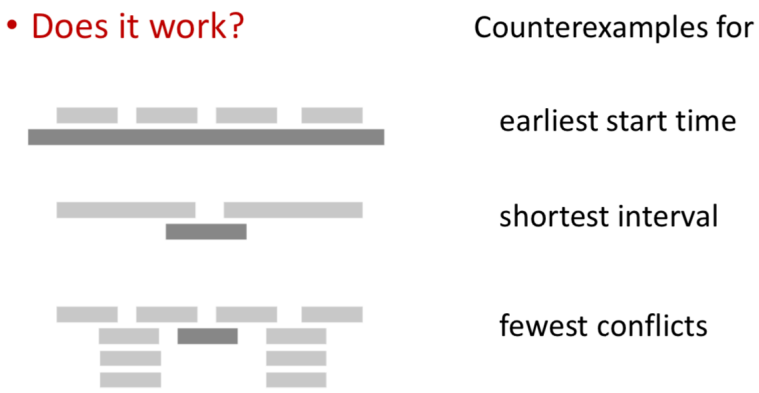
Problem Job j starts at time s_j and finishes at time f_j. Two jobs are compatible if they don't overlap.
Goal: find maximum-size subset of mutually compatible jobs

Greedy template Consider jobs in some "natural" order. Take each job if it's compatible with the ones already chosen.

Order candidates

1. Earliest start time: ascending order of s_j
2. Earliest finish time: ascending order of f_j
3. Shortest interval: ascending order of $f_j - s_j$
4. Fewest conflicts: ascending order of c_j, where c_j is the number of remaining jobs that conflict with j
5. Distance between jobs

Counterexamples For proving 1,2,4 do not work:



Implementing greedy with earliest finish time (EFT) Sort jobs by finish time. Say $f_1 \leq f_2 \leq \cdots \leq f_n$. When deciding whether job j should be included, we need to check whether it is compatible with all previously added jobs.
We only need to check if $s_j \geq f_{i^*}$, where i^* is the last added job. This is because for any job i added before i^*, $f_i \leq f_{i^*}$. So we can simply store and maintain the finish time of the last added job.

Running time $\mathcal{O}(n \log n)$
(Sorting takes $n \log n$)

Optimality of greedy with EFT Suppose for contradiction that greedy is not optimal.
Say greedy selects jobs $i_1, i_2, \ldots, i_k$ sorted by finish time.
Consider the optimal solution $j_1, j_2, \ldots, j_m$ (also sorted by finish time) which matches greedy for as long as possible. That is, we want $j_1 = i_1, \ldots, j_r = i_r$ for greatest possible r.

Both i_{r+1} and j_{r+1} were compatible with the previous selection ($i_1 = j_1, \ldots, i_r = j_r$)
Consider the solution $i_1, i_2, \ldots, i_r, i_{r+1}, j_{r+2}, \ldots, j_m$. It should still be feasible (since $f_{i_{r+1}} \leq f_{j_{r+1}}$), therefore it is still optimal, and it matches with greedy for one more step (contradiction!) ■

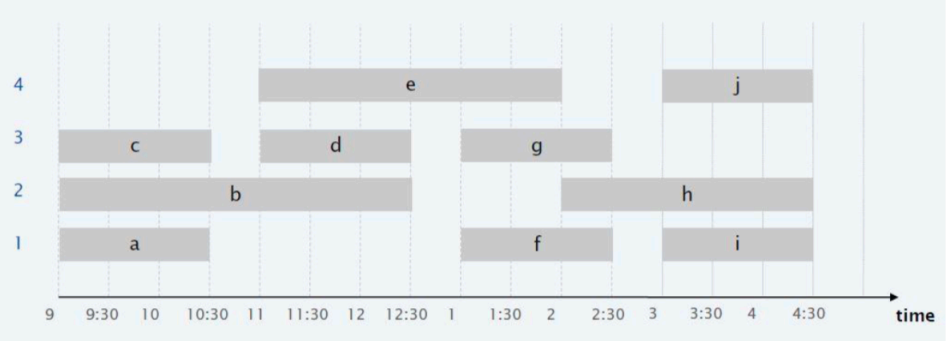
2.2 Interval Partitioning

Problem Job j starts at time s_j and finishes at time 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

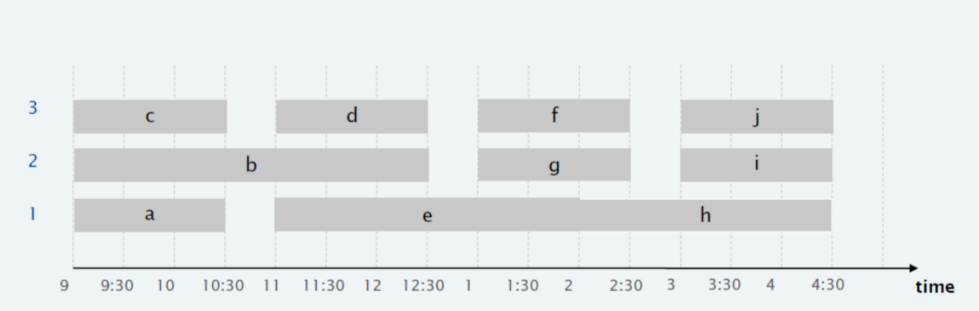
One idea Find the maximum compatible set using the previous greedy EFT algorithm, call it one partition, recurse on the remaining jobs.
$\Longrightarrow$ Doesn't work!

Think of scheduling lectures for various courses into few classrooms as possible.

- This schedule uses 4 classrooms for scheduling 10 lectures



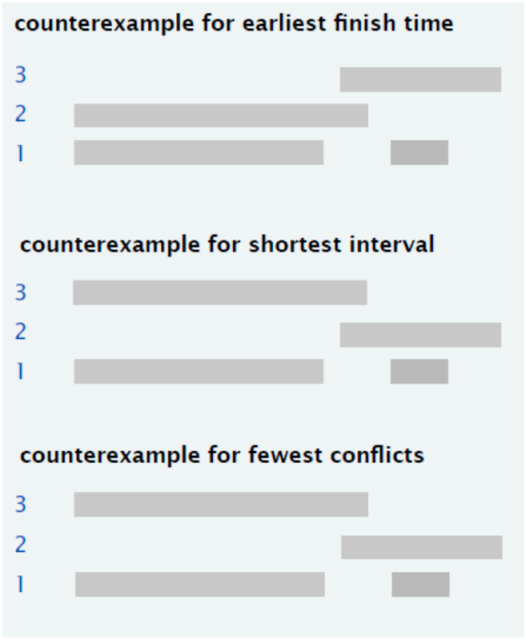
- This schedule uses 3 classrooms for scheduling 10 lectures



Greedy template Go through lectures in some "natural order", assign each lecture to a compatible classroom, and create a new classroom if the lecture conflicts with every existing classroom

Order of lectures

1. Earliest start time: ascending order of s_j
2. Earliest finish time: ascending order of f_j
3. Shortest interval: ascending order of $f_j - s_j$
4. Fewest conflicts: ascending order of c_j, where c_j is the number of remaining jobs that conflict with j



- At least when you assign each lecture to an arbitrary feasible classroom, three of these heuristics do not work.
- The fourth one works! (next slide)

Figure 1: Counterexamples

2.3 Interval Graphs

Interval scheduling and interval partitioning can be seen as graph problems.

Input

1. Graph $G = (V, E)$

2. Vertices V = jobs or lectures
3. Edge $(i, j) \in E$ if jobs i and j are incompatible

Problems

1. Interval scheduling = maximum independent set (MIS)
2. Interval partitioning = graph colouring

2.4 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 as s_j, it will finish at $f_j = s_j + t_j$.
Lateness: $l_j = \max\{0, f_j - d_j\}$
Goal: minimize the maximum lateness, $L = \max_j l_j$
Total lateness minimization is $NP - complete$

Contrast with interval scheduling

1. We can decide the start time
2. All jobs must be scheduled on a single machine

2.5 Lossless Compression

Problem We have a document that is written using n distinct labels.
Naive encoding: represent each label using $k = \log n$ bits
If the document has length m, this used $m \log n$ bits.
But what if some labels are much more frequent in the document than others? We need to assign shorter codes to more frequent letters.
To avoid conflicts, we need prefix-free encoding:
Map each label x to a bit-string $c(x)$ such that for all distinct labels x and y, $c(x)$ is not a prefix of $c(y)$. So we can read left to right, find the first point where it becomes a valid encoding, decode the label, and continue.

Formal Problem Given n symbols and their frequencies $(w_1, \ldots, w_n)$, find a prefix-free encoding with lengths $(l_1, \ldots, l_n)$ assigned to the symbols which minimizes $\sum_{i=1}^{n} w_i \cdot l_i$

2.6 Other Greedy Algorithms

1. Dijkstra's shortest path algorithm
2. Kruskal and Prim's minimum spanning tree algorithms