CSC373 Notes

Max Xu

January 23, 2026

Contents

§1 Day 1: Intro (Jan 06, 2026)

This was taken off the slides from past years.

§1.1 About this Class

This class is about designing algorithms to solve problems.

We will be:

(i) Designing fast algorithms
  - Divide and conquer
  - Greedy algorithms
  - Dynamic programming
  - Network flow
  - Linear programming

(ii) Proving no fast algorithms are likely possible
  - Reductions and NP-completeness

(iii) Solving problems where no fast algorithms are possible
  - Approximation algorithms
  - Randomized algorithms

When we analyze an algorithm, we do correctness and running-time proofs.

§2 Day 2: Intro Redux (Jan 08, 2026)

This course is now about the thought process behind solutions of problems. We use the **RAM Computational Model**.

A proof is a convincing argument:

- Convince your TA for marks
- Convince employer that your program does what it claim it does
- Convince yourself that you're not producing word salad

Sometimes, formal verification is used for mission-critical applications, where unit tests may not have sufficient coverage. We use semi-formal proofs in this course, to prove specific results (as opposed to more general ones, like in math).

§2.1 Divide & Conquer

The general framework is to:

(i) Break a problem into two smaller subproblems of the same type

(ii) Solve each problem recursively and independently

(iii) Quickly combine solutions from subproblems to form a solution to a bigger part of the problem

'Quick/cheap' means that the step count is in $O(f(n))$ where f is a polynomial.

Recurrence relations are often encountered while analyzing the running time of divide-and-conquer algorithms. We take the master theorem from (CLRS) for granted, a general result about the asymptotic behavior of certain types of recurrences.

Theorem 2.1 (CLRS Master Theorem)

Let $a \geq 1$, $b > 1$. Have $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence $T(n) = aT(n/b) + f(n)$, where n/b is interpreted as $\lceil \frac{n}{b} \rceil$ or $\lfloor \frac{n}{b} \rfloor$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) \in O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) \in \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log_2 n)$
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) \in \Theta(f(n))$

n/b describes the size of the subproblems, and $f(n)$ describes the step count required to merge/divide the subproblems to form a solution of size n. The Master Theorem handles the leaf-heavy, balanced, and root-heavy case in that order.

Example 2.2

Some problems that can be solved using a divide-and-conquer approach include:

- Counting inversions in an array
- Closest pair in $\mathbb{R}^2$, with non-degeneracy assumption

Algorithms considered divide-and-conquer include:

- Karatsuba's Algorithm
- Strassen's Algorithm

Some problems that

There was also a brief discussion about galactic algorithms.

§3 Day 3: Greedy Algorithms I (Jan 13, 2026)

Greedy algorithms have the following outline:

Goal Find a solution x involving a objective function f (finding maxima/minima)

Challenge It is not feasible to check the entire solution space

Observation Decompose x into its parts (being individual decisions), make the choice that maximize the 'immediate benefit' (e.g. maximize change in f).

The correctness proof needs to show that the choices made greedily are in fact optimal. The greedy (partial) solution after j iterations can be extended to an optimal solutions, for each j.

Problem 3.1 (Task Scheduling)

Suppose you have a set of jobs J. Each job j starts at time s_j and ends at time f_j. Two jobs i, j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ don't overlap. Our task is to find the maximum size subset of J of pairwise compatible jobs.

First, we describe our algorithm. Let $n = |J|$. Initialize our partial solution $P = \emptyset$.

(i) Sort jobs by finish time, giving us $f_1 \leq \cdots \leq f_n$. Iterate through our sorted jobs from lowest finish time to highest finishing time.

(ii) For some particular job j, check if it starts after the last job in P, i^*

(iii) If $s_j \geq f_{i^*}$ add it to our partial solution P.

(iv) Go back to step (ii), until no jobs are left

Quickly verify that our algorithm produces output that is pairwise compatible. $(*)$

It remains to show that our algorithm is optimal. There are 2 approaches, contradiction and induction, though they are equivalent.

Contradiction Suppose for contradiction that our algorithm doesn't produce the optimal solution, instead giving $I = i_1, \cdots, i_k$ sorted by finish time.

Since n is finite, an optimal solution $J = j_1, \cdots, j_m$ exists (meaning $m > k$), that matches our algorithm's greedy solution for the largest possible contiguous chunk of indices from the beginning. Let r be the last index, where $i_1 = j_i, i_2 = j_2, \cdots, i_r = j_r$ (r defaults to 0).

We may then replace j_{r+1} with i_{r+1} in J, creating a new solution of size m, J'. WTS J' is a pairwise compatible solution of size m.

- Show $f_{i_r} \leq s_{i_{r+1}}$.
  This follows from $(*)$.
- Show $f_{i_{r+1}} \leq s_{j_{r+2}}$.
  Our algorithm has the property that i_{r+1} is the first compatible job in the remaining set of jobs, sorted by finishing time. In other words,

$f_{i_{r+1}} \leq f_{j_{r+1}} \leq s_{j_{r+2}}$, with the last inequality following from J being pairwise compatible.

This shows that J' is pairwise compatible, with size equal to that of J, hence it is optimal. Yet J' and I are equal up till the $r + 1$st index, contradicting our original claim.

Induction The induction case uses a very similar argument, which argues that every partial solution from our algorithm is a subset of a optimal solution, where optimality once again stems from our algorithm's choice of the next compatible task with the earliest finishing time.

Both methods ultimately make the same claim: the greedy choice at step j is always part of an optimal solution. Induction proves this through a sequence of steps, while contradiction proves it by showing that any supposed 'non-optimal' greedy choice could still be extended to reach the optimum.

§4 Day 4: Greedy Algorithms II (Jan 22, 2026)

Outline for finding optimal greedy algorithms:

(i) List heuristics that sound reasonable

(ii) Test each of your heuristics on a test case, analyze their shortcomings

(iii) Convince yourself that one is optimal, and explain why it succeeds

Usually our greedy algorithm only finds a optimal solution. There may be more optimal solutions that our greedy algorithm cannot find.

Problem 4.1 (Minimizing Lateness)

Suppose you have a single machine and j jobs, each requiring t_j units of time and are due by d_j. Your task is to choose the start times for each job $s_j \geq 0$, such that

$$\max_j \{0, s_j + t_j - d_j\}$$

is minimized. The machine can only do one task at a time.

Sort the tasks by deadline in non-decreasing order. Define a 'inversion'[1], which is a pair (i, j), where $d_i < d_j$, yet $s_j < s_i$. Begin by stating some observations:

- There is an optimal schedule with no idle time between tasks. $(*)$
- If a schedule with no idle time between tasks has at least one inversion, it has an inversion between consecutive tasks: i.e. exists j, $(j, j+1)$ is an inversion.
- Removing the aforementioned inversion by scheduling task j at s_{j+1}, and task $j+1$ at $s_{j+1} + t_j$ does not increase lateness, and also reduces the number of inversions by 1. $(**)$

Now we show our algorithm is optimal.

Induction We will prove by induction on n, the problem size.

Base Case: There is nothing to do.

Inductive Step: A viable induction hypothesis is there always exists an optimal schedule for problems with n jobs, with no idle time between tasks $(*)$ and no inversions $(**)$, and that our algorithm produces this schedule.

Contradiction We use the minimal counterexample strategy. Suppose there's an optimal schedule with the fewest inversions. Clearly, the number of inversions is nonzero, since we can 'reduce' that case to something our algorithm will output using $(*)$. We then use a standard well-ordering argument using $(**)$ to remove inversions, concluding that the set of counterexamples is empty.

[1] tangentially related to the inversion encountered in sorting

Problem 4.2 (Lossless Compression)

Given n symbols and their frequencies/weights $(w_1, \ldots, w_n)$, find a prefix-free encoding with lengths $(\ell_1, \ldots, \ell_n)$ assigned to the symbols which minimizes $\sum_{i=1}^{n} w_i \cdot \ell_i$, the size of the compressed document.

This problem is motivated by restrictions on file transfer speed. Lossless compression should reduce the file size, and result in no loss of information. We now motivate the notion of prefix-free encoding:

Example 4.3

Suppose you encode a as 0, b as 1, and c as 01. When encountering 01 in a transmission, it is unclear whether the intended message is ab or c. Realize this ambiguity arises from the fact that the encoding of a is a prefix for c.

The **Huffman Coding** technique generates a prefix-free encoding for any set of n symbols, by building a binary tree. The symbols will be stored in the leaves (hence the internal nodes' symbol field are meaningless), and the encoding will be the sequence of binary digits (0 for left, 1 for right) needed to reach the corresponding leaf for a symbol from the root. The choice of 0 and 1 for left and right does not matter.

- Initialize a min-priority queue by w by adding (x, w_x)
- While there's at least 2 nodes in the queue, pop 2 nodes from the priority queue, and set their parent to a new node with weight $w_x + w_y$. Add this node back to the priority queue. The choice between left right child also doesn't matter.

Notice that the choice of the 2 nodes with the smallest w field makes the algorithm greedy. We now prove that this is prefix free.

Contradiction For contradiction, suppose s_1 and s_2 are both symbols, and the encoding for s_1, is a prefix of that of s_2. The node obtained by following that for s_1 in the tree is an ancestor of that of s_2. But this contradicts our tree invariant, that all symbols are stored in leaves.

See that prefix-free encoding generates a tree T. Define $W(T) = \sum_{i=1}^{n} w_i \cdot \ell_i$, where ℓ_i is the length of the path from the root to the leaf.

We state some observations/lemmas:

- If $w_x < w_y$, then $\ell_x \geq \ell_y$ in any optimal tree.
- For each pair (x, w_x), (y, w_y) that Huffman Coding combines, there exists an optimal tree in which they are siblings. $(*)$

Now, we prove optimality.

Induction We prove by induction on the size of the alphabet.

Base Case: There is nothing to do $(n = 1)$.

Induction Step: Our induction hypothesis our algorithm produces optimal trees for alphabets of smaller size.

Let (x, w_x) and (y, w_y) be the symbols first assigned a parent by our algorithm (thus at maximum depth). Consider the alphabet, with x and y removed, and the additional symbol xy, with $w_{xy} = w_x + w_y$. Our algorithm produces an optimal tree H' by the induction hypothesis.

We now take said tree, and add (x, w_x) and (y, w_y) as the children of (xy, w_{xy}), giving a new tree H. We show that this tree is optimal. Realize $W(H) = W(H') + w_x + w_y$. By $(*)$, there exists an optimal tree T where x and y are siblings. Joining them together to form (xy, w_{xy}) once again, get a tree T' with $W(T') + w_x + w_y = W(T)$. Re-arranging, see that $W(T) = W(H)$. This shows optimality of our algorithms' tree H.