Week 15 - Friday

COMP 4500

Last time

- What did we talk about last time?
- Review of first third of course
  - Big oh
  - Calculating running time
  - Graph basics
  - Greedy algorithms
    - Interval scheduling
    - Shortest path
    - Minimum spanning tree
    - Huffman codes

Questions?

Assignment 7

Logical warmup

- People with assorted eye colors live on an island. They are all perfect logicians: If a conclusion can be logically deduced, they will do it instantly. No one knows their own eye color. Every night at midnight, a ferry stops at the island. Any islanders who have figured out the color of their own eyes must leave the island, and the rest stay. Everyone can see everyone else at all times and keeps a count of the number of people they see with each eye color (excluding themselves), but they cannot otherwise communicate. Everyone on the island knows these rules.

- On this island there are 100 blue-eyed people, 100 brown-eyed people, and the Guru (she happens to have green eyes). So any given blue-eyed person can see 100 people with brown eyes and 99 people with blue eyes (and one with green), but that does not tell him his own eye color; as far as he knows the totals could be 101 brown and 99 blue. Or 100 brown, 99 blue, and he could have red eyes.

- The Guru is allowed to speak once, at noon, on one day in all their endless years on the island. Standing before the islanders, she says the following:

  "I can see someone who has blue eyes."

- Who leaves the island, and on what night?

Review

Final exam

- Final exam:
  - Friday, April 24, 2026
  - 10:15 a.m. – 12:15 p.m.
- It will mostly be short answer
- There will be diagrams
- There might be a matching problem
- There will likely be a (simple) proof
- It will be 50% longer than previous exams, but you will have 100% more time

Recurrence Relations

Divide and conquer

- **Divide and conquer** algorithms are ones in which we divide a problem into parts and recursively solve each part
- Then, we do some work to combine the solutions to each part into a final solution
- Divide and conquer algorithms are often simple
- However, their running time can be challenging to compute because recursion is involved

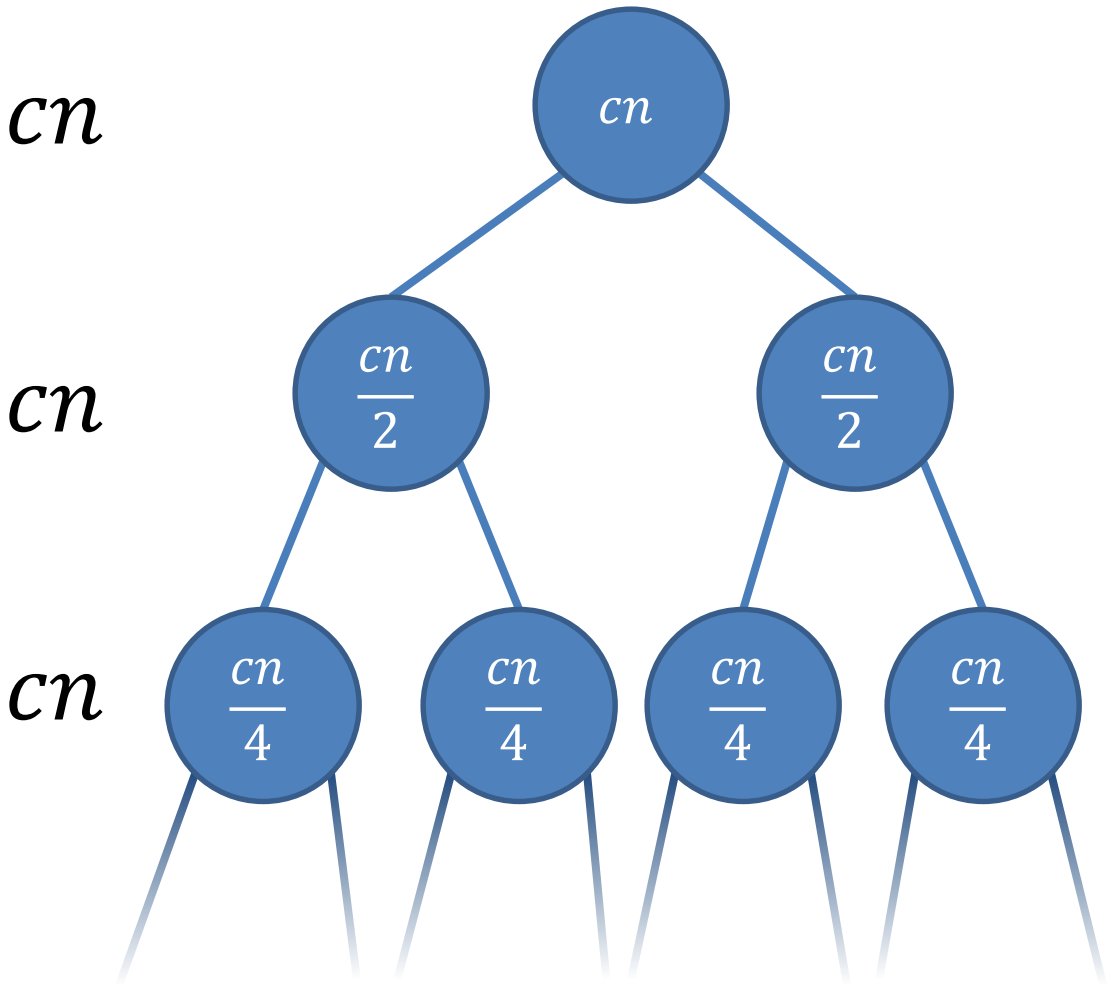
Mergesort algorithm

- If there are two elements in the array or fewer then
  - Make sure they're in order
- Else
  - Divide list into two halves
  - Recursively merge sort the two halves
  - Merge the two sorted halves together into the final list

Time for mergesort

- The algorithm is simple, but recursive
- We'll use $T(n)$ to describe the total running time recursively
  - $T(n) \leq c, \qquad n \leq 2$
  - $T(n) \leq 2T\left(\frac{n}{2}\right) + cn, \qquad n > 2$
- Is it really the same constant c for both?
  - No, but it's an inequality, so we just take the bigger one

Intuition about mergesort recursion



- Each time, the recursion cuts the work in half while doubling the number of problems
  - The total work at each level is thus always cn
- To go from n to 2, we have to cut the size in half $(\log_2 n) - 1$ times

Recursively defined sequences

- Defining a sequence recursively as with Mergesort is called a **recurrence relation**
- The **initial conditions** give the starting point
- Example:
  - Initial conditions
    - $T(0) = 1$
    - $T(1) = 2$
  - Recurrence relation
    - $T(k) = T(k-1) + 3T(k-2) + k$, for all integers $k \geq 2$
  - Find $T(2)$, $T(3)$, and $T(4)$

Finding explicit formulas by iteration

- We want to be able to turn recurrence relations into explicit formulas whenever possible
- Often, the simplest way is to find these formulas by **iteration**
- The technique of iteration relies on writing out many expansions of the recursive sequence and looking for patterns
- That's it

Employing outside formulas

- Intelligent pattern matching gets you a long way
- However, it is sometimes necessary to substitute in some known formula to simplify a series of terms
- Recall
  - Geometric series: $1 + r + r^2 + \ldots + r^n = \frac{r^{n+1} - 1}{r - 1}$
  - Arithmetic series: $1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$

Further recurrence relations

- We have seen that recurrence relations of the form $T(n) \leq 2T\left(\frac{n}{2}\right) + cn$ are bounded by $O(n \log n)$
- What about $T(n) \leq qT\left(\frac{n}{2}\right) + cn$ where q is bigger than 2 (more than two sub-problems)?
- There will still be $\log_2 n$ levels of recursion
- However, there will **not** be a consistent cn amount of work at each level

Converting to summation

- In general, it's

$$T(n) \leq \sum_{j=0}^{\log_2 n-1} \left(\frac{q}{2}\right)^j cn = cn \sum_{j=0}^{\log_2 n-1} \left(\frac{q}{2}\right)^j$$

- This is a geometric series, where $r = \frac{q}{2}$

$$T(n) \leq cn\left(\frac{r^{\log_2 n} - 1}{r - 1}\right) \leq cn\left(\frac{r^{\log_2 n}}{r - 1}\right)$$

Final bound

$$T(n) \leq cn\left(\frac{r^{\log_2 n}-1}{r-1}\right) \leq cn\left(\frac{r^{\log_2 n}}{r-1}\right)$$

- Since $r-1$ is a constant, we can pull it out
- $T(n) \leq \left(\frac{c}{r-1}\right) n r^{\log_2 n}$
- For $a > 1$ and $b > 1$, $a^{\log b} = b^{\log a}$, thus $r^{\log_2 n} = n^{\log_2 r} = n^{\log_2(q/2)} = n^{(\log_2 q)-1}$
- $T(n) \leq \left(\frac{c}{r-1}\right) n \cdot n^{(\log_2 q)-1} \leq \left(\frac{c}{r-1}\right) n^{\log_2 q}$ which is $O\left(n^{\log_2 q}\right)$

Counting Inversions

Ranking similarity

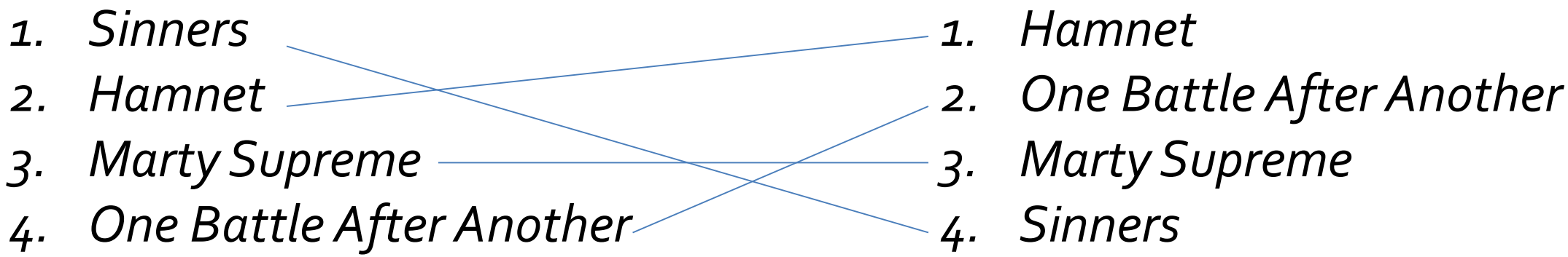
- What if we wanted to measure the similarity of one ranking to another ranking?
- **Inversions** are pairs of elements that are out of order in one ranking with respect to the other
- Formally, for indices $i < j$, there's an inversion if ranking $r_i > r_j$

Minimum and maximum inversions

- If two rankings are the same, they would have no inversions
- If two rankings were sorted in opposite directions, they would have $\binom{n}{2} = \frac{n!}{(n-2)!2!} = \frac{n(n-1)}{2}$ inversions

Visualization of inversions

- You can visualize inversions as the number of line segments crossings if you match up items in one list with the other



- A total of 4 inversions

Can we do better than $O(n^2)$?

- Of course!
- We can borrow from the Mergesort algorithm
- Divide the problem in half
- Then, we will get the number of inversions in the first half and in the second half
- Are we done?
  - No, we also have to count the inversions between the first half and the second half
  - Those are exactly those elements in the first half that are bigger than elements from the second half
  - We can find those during the merge process

Merge-and-Count(A, B)

- Maintain a $Current$ pointer into each list, initialized to point to the front elements
- Set $Count = 0$
- While both lists have elements
  - Let a_i and b_j be the elements pointed to by the $Current$ pointer
  - Append the smaller one to the output list
  - If b_j is smaller then
    - Increment $Count$ by the number of elements left in A
  - Advance the $Current$ pointer in the list that had the smaller element

Sort-and-Count(L)

- If the list has one element then
  - Return 0 inversions and the list L
- Else
  - Divide the list into two halves:
    - A has the first $\lceil \frac{n}{2} \rceil$ elements
    - B has the remaining $\lfloor \frac{n}{2} \rfloor$ elements
  - $(inversions_A, A)$ = Sort-and-Count(A)
  - $(inversions_B, B)$ = Sort-and-Count(B)
  - $(inversions, L)$ = Merge-and-Count(A, B)
  - Return $inversions + inversions_A + inversions_B$ and sorted list L

Running time

- Since Merge-and-Count is bounded by $O(n)$, the running time for Sort-and-Count is clearly:
  - $T(1) \leq c$
  - $T(n) \leq 2T\left(\frac{n}{2}\right) + cn$, for $n \geq 2$
- By the same analysis as for Mergesort, $T(n)$ is $O(n \log n)$

Closest Pair of Points

Closest pair of points

- Imagine you have a set of points in a 2D plane
- How do you find the pair of points that's closest?
- This is a fundamental problem in the area of **computational geometry**
- As usual, you could look at all pairs of points

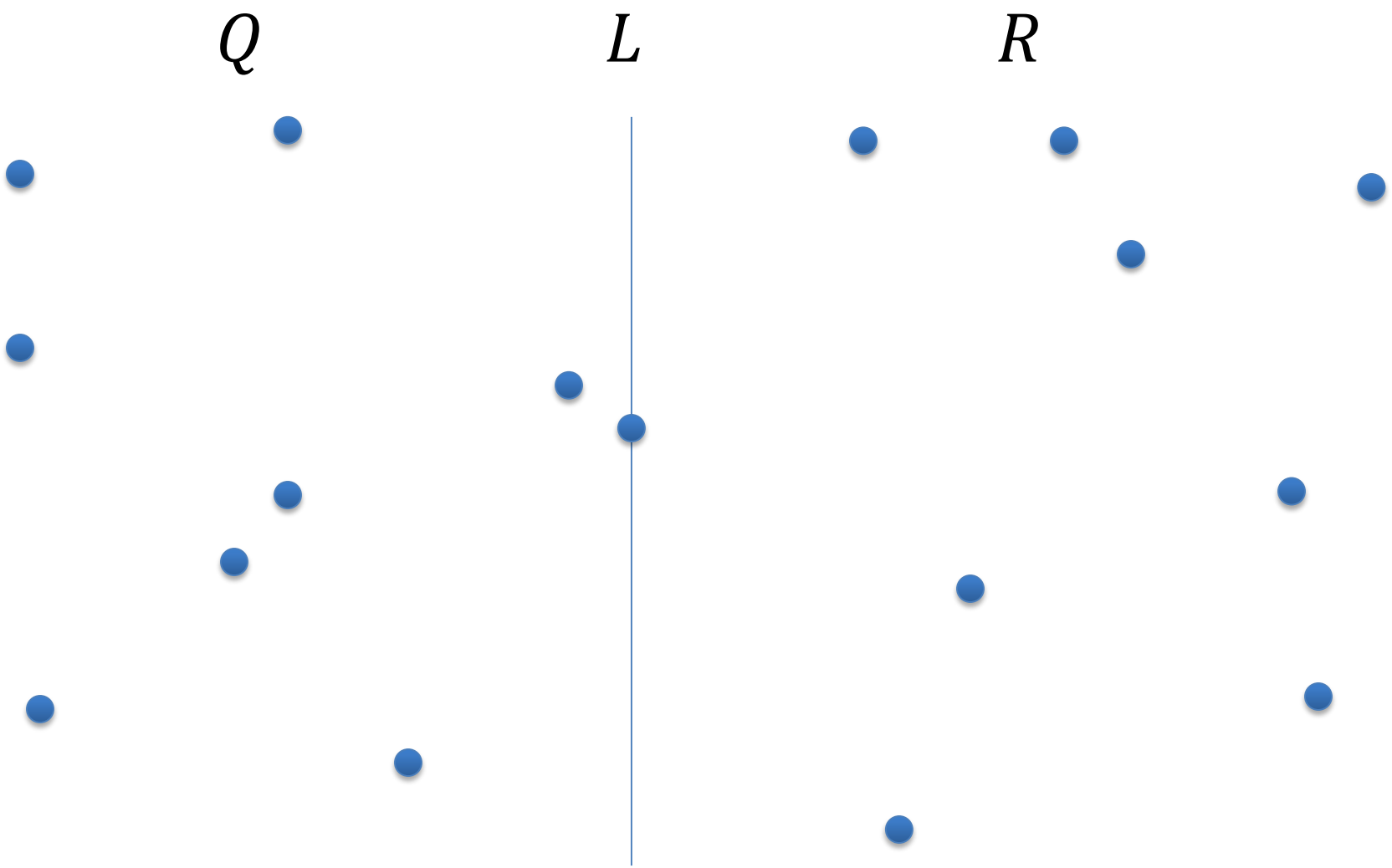
Designing the algorithm

- To make things simpler, we assume that no two points have the same x-coordinate or y-coordinate
- Think about a one-dimensional approach:
  - Sort the list by x-value
  - The two closest points must be next to each other in the list

Divide

- Since the name of the chapter is divide and conquer, that's what we do
- First, sort all of the points by increasing x-values, calling this list P_x
- Then, sort all of the points by increasing y-values, calling this list P_y
- Find the median point in P_x and drop a line through it, dividing the points into those with smaller x (set Q) and larger x (set R)
- Recursively find the closest pair of points on the left side and the closest pair of points on the right side

Divide points



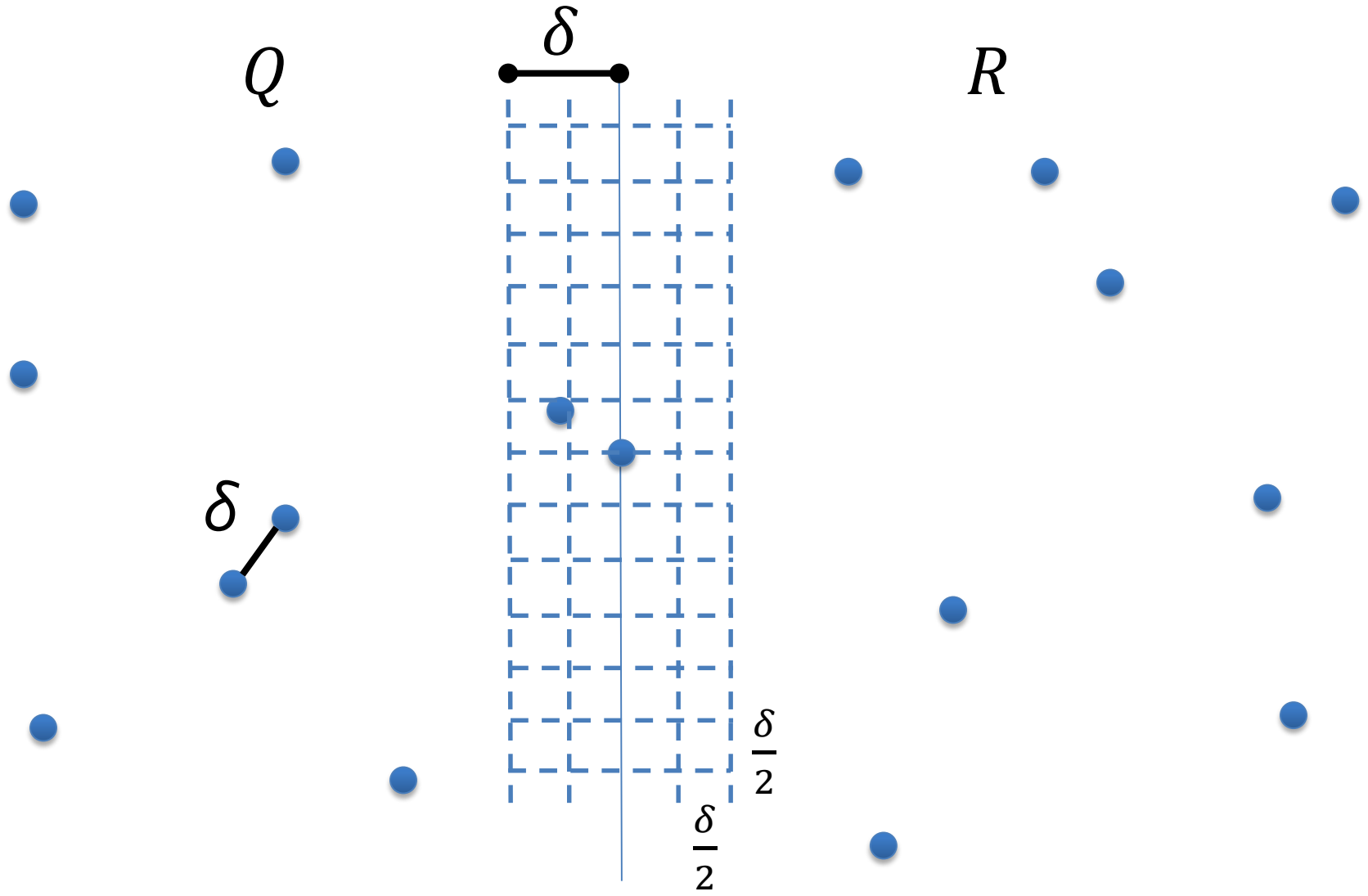
… and …

- We have ~~magically~~ recursively found the closest pair of points in Q and the closest pair in R
  - Between those two pairs, let's say the closest has distance δ
- But what if the closest pair straddles L, with one point in Q and the other in R?
- We do a linear scan of P_y, the list of points sorted by y values, making a new y-sorted list of points S_y whose x-coordinate is within δ of L

… conquer!

- We scan through the list S_y
- For each element, we compute the distance between it and the next 15 elements
- We find the closest distance
- If the closest distance is smaller than δ, that's the true closest pair
- Otherwise, we use the smaller of the pairs from Q and R

Divide points



Running time

- Pre-processing:
  - Sort the points by x: $O(n \log n)$
  - Sort the points by y: $O(n \log n)$
- Recursion:
  - If there are three or fewer points, find the closest pair by comparing all pairs
  - Otherwise, divide into sets Q and R: $O(n)$ time
  - Make lists Q_x, Q_y, R_x, and R_y, giving the points in Q and R sorted by x and y, respectively: $O(n)$ time
  - Construct S_y: $O(n)$ time
  - For every point in S_y (of which there can only be n), compute the distance to the next 15 points: $O(n)$
- $T(n) \leq 2T\left(\frac{n}{2}\right) + cn$ which is $O(n \log n)$

Integer Multiplication

We need a trick

- We want $x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{\frac{n}{2}} + x_0y_0$
- What if we compute
  - $a = (x_1 + x_0) \cdot (y_1 + y_0)$
    $= x_1y_1 + x_0y_1 + x_1y_0 + x_0y_0$
  - $b = x_1y_1$
  - $c = x_0y_0$
- Then, $b \cdot 2^n + (a - b - c) \cdot 2^{\frac{n}{2}} + c =$
- $x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{\frac{n}{2}} + x_0y_0$

Running time

- We do two additions before the multiplies: O(***n***)
- We do three recursive multiplies of ***n***/2-bit numbers
- We do two additions and two subtractions after the multiplies: O(***n***)
- $T(n) \leq 3T\left(\frac{n}{2}\right) + cn$
- Which is $O\left(n^{\log_2 3}\right) \approx O(n^{1.59})$, which is better!

Master Theorem

Basic form of the Master Theorem

- For recursion that on a problem size n that:
  - Makes a recursive calls
  - Divides the total work by b for each recursive call
  - Does $f(n)$ non-recursive work at each call
- Its running time can be given in the following form, suitable for use in the Master Theorem:

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

where $a \geq 1$ and $b > 1$

Case 1

- If $f(n)$ is $O\left(n^{\log_b(a)-\epsilon}\right)$ for some constant $\epsilon > 0$, then

$$T(n) \text{ is } \Theta\left(n^{\log_b(a)}\right)$$

Case 2

- If $f(n)$ is $\Theta\left(n^{\log_b(a)} \log^k n\right)$ for some constant $k \geq 0$, then

$$T(n) \text{ is } \Theta\left(n^{\log_b(a)} \log^{k+1} n\right)$$

Case 3

- If $f(n)$ is $\Omega\left(n^{\log_b(a)+\epsilon}\right)$ for some constant $\epsilon > 0$, and if

$$af\left(\frac{n}{b}\right) \leq cf(n)$$

for some constant $c < 1$ and sufficiently large n, then

$$T(n) \text{ is } \Theta(f(n))$$

Example Problems

Recursive sequence example

- $g_k = g_{k-1} + k$ for all integers $k \geq 1$
- $g_0 = 7$

- Give an explicit formula for this recurrence relation
- **Hint:** Use the method of iteration

Sample master theorem problem

- $T(n) = 64T\left(\frac{n}{4}\right) + 12n^2$

- Sometimes it helps to think about how I create questions:
  - Generate a recurrence relation that fits Case 1
  - Generate a recurrence relation that fits Case 2
  - Generate a recurrence relation that fits Case 3

Dynamic Programming

Weighted interval scheduling

- The **weighted interval scheduling** problem extends interval scheduling by attaching a weight (usually a real number) to each request
- Now the goal is not to maximize the **number** of requests served but the total **weight**
- Our greedy approach is worthless, since some high value requests might be tossed out
- We could try all possible subsets of requests, but there are **exponential** of those
- **Dynamic programming** will allow us to save parts of optimal answers and combine them efficiently

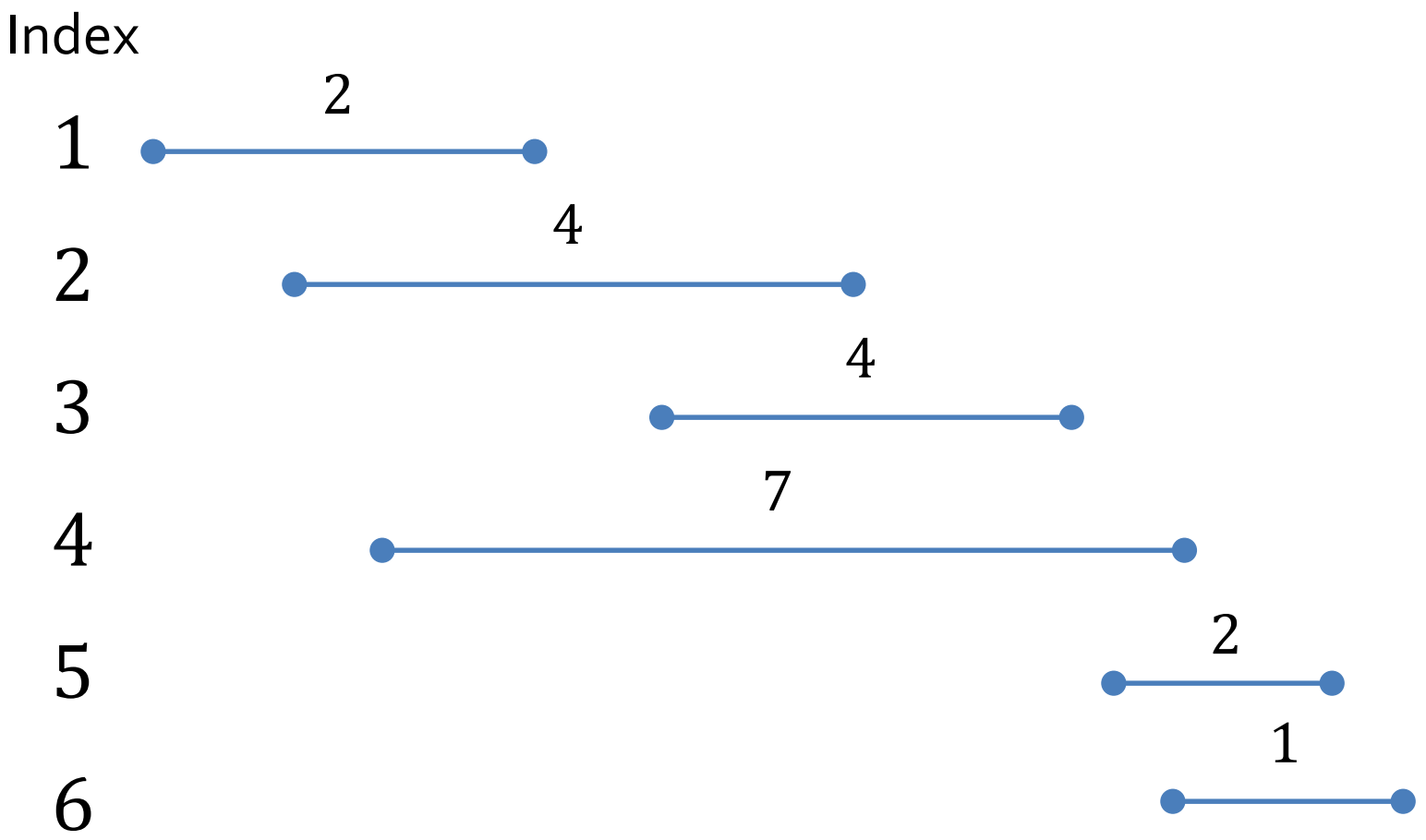
Notation

- We have n requests labeled $1, 2, \ldots, n$
- Request i has a start time s_i and a finish time f_i
- Request i has a value v_i
- Two intervals are **compatible** if they don't overlap

Designing the algorithm

- Let's go back to our intuition from the unweighted problem
- Imagine that the requests are sorted by finish time so that $f_1 \leq f_2 \leq \ldots \leq f_n$
- We say that request i comes before request j if $i < j$, giving a natural left-to-right order
- For any request j, let $p(j)$ be the largest index $i < j$ such that request i ends before j begins
  - If there is no such request, then $p(j) = 0$

$p(j)$ examples



$$p(1) = 0$$

$$p(2) = 0$$

$$p(3) = 1$$

$$p(4) = 0$$

$$p(5) = 3$$

$$p(6) = 3$$

Iterative solution to find value of weighted interval scheduling

- Iterative-Compute-Opt
  - $M[0] = 0$
  - For $j = 1$ up to n
    - $M[j] = \max(v_j + M[p(j)], M[j-1])$

- Algorithm is O(n)

Algorithm for solution

- Find-Solution(j, M)
  - If $j = 0$ then
    - Output nothing
  - Else if $v_j + M[p(j)] \geq M[j - 1]$ then
    - Output j together with the result of Find-Solution($p(j)$)
  - Else
    - Output the result of Find-Solution($j - 1$)
- Algorithm is $O(n)$

Why is this dynamic programming?

- The key element that separates dynamic programming from divide-and-conquer is that you have to *keep* the answers to subproblems around
- It's not simply a one-and-done situation
- Based on which intervals overlap with which other intervals, it's hard to predict when you'll need an earlier $M[j]$ value
- Thus, dynamic programming can often give us polynomial algorithms *but* with linear (and sometimes even larger) space requirements

Informal guidelines

- Weighted interval scheduling follows a set of informal guidelines that are essentially universal in dynamic programming solutions:
  1. There are only a polynomial number of subproblems
  2. The solution to the original problem can easily be computed from (or is one of) the solutions to the subproblems
  3. There is a natural ordering of subproblems from "smallest" to "largest"
  4. There is an easy-to-compute recurrence that lets us compute the solution to a subproblem from the solutions of smaller subproblems

Subset sum

- Let's say that we have a series of n jobs that we can run on a single machine
- Each job i takes time w_i
- We must finish all jobs before time W
- We want to keep the machine as busy as possible, working on jobs until as close to W as we can

A new recurrence

- If job n is not in the optimal set,

$$\mathrm{OPT}(n, W) = \mathrm{OPT}(n - 1, W)$$

- If job n is in the optimal set,

$$\mathrm{OPT}(n, W) = w_n + \mathrm{OPT}(n - 1, W - w_n)$$

- We can make the full recurrence for all possible weight values:
  - If $w < w_i$, then

$$\mathrm{OPT}(i, w) = \mathrm{OPT}(i - 1, w)$$

  - Otherwise,

$$\mathrm{OPT}(i, w) = \max(\mathrm{OPT}(i - 1, w), w_i + \mathrm{OPT}(i - 1, w - w_i))$$

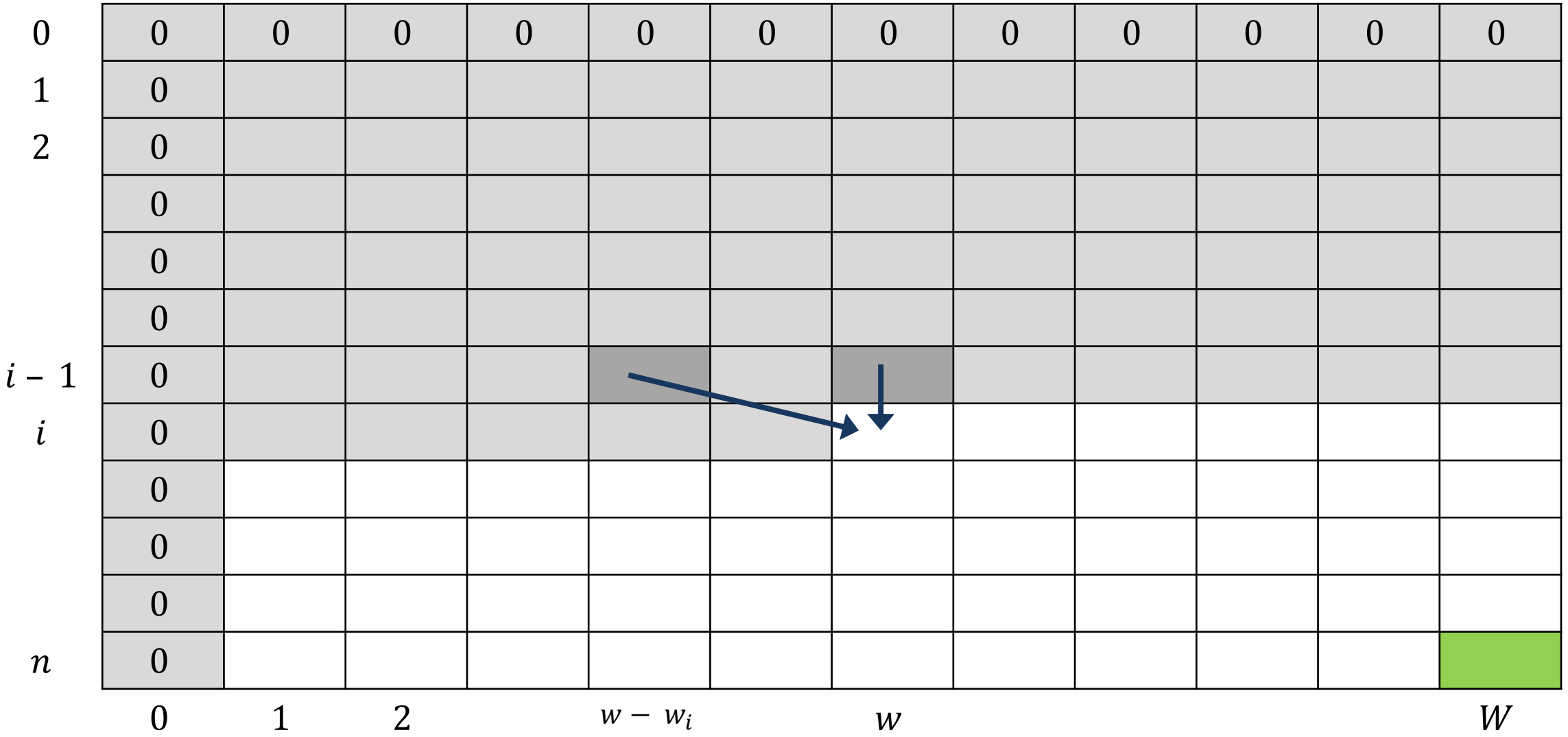
Subset-Sum(n, W)

- Create 2D array $M[0 \ldots n][0 \ldots W]$
- For w from 1 to W
  - Initialize $M[0][w] = 0$
- For i from 1 to n
  - For w from 0 to W
    - If $w < w_i$, then
      - $\mathrm{OPT}(i, w) = \mathrm{OPT}(i - 1, w)$
    - Else
      - $\mathrm{OPT}(i, w) = \max(\mathrm{OPT}(i - 1, w), w_i + \mathrm{OPT}(i - 1, w - w_i))$
- Return $M[n][W]$

What does that look like?

- We're building a big 2D array
- Its size is nW
  - n is the number of items
  - W is the maximum weight
  - Actually, it's got one more row and one more column, just to make things easier
- The book makes this array with row 0 at the bottom
- I've never seen anyone else do that
- I'm going to put row 0 at the top

Table M of OPT values



Running time

- The algorithm has a simple nested loop
  - The outer loop runs $n + 1$ times
  - The inner loop runs $W + 1$ times
- The total running time is $O(nW)$
- The space needed is also $O(nW)$
- Note that this time is not polynomial in terms of n
- It's polynomial in n and W, but W is the maximum weight
  - Which could be huge!
- We call running times like this **pseudo-polynomial**
- Things are fine if W is similar to n, but it could be huge!

Subset sum example

- Weights: $1, 4, 8, 3, 6$
- Maximum: 15
- Create the table to find all of the optimal values that include items $1, 2, \ldots, i$ for every possible weight w up to 15

Table to fill in

i	w_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1																
2	4																
3	8																
4	3																
5	6																

Knapsack

- The knapsack problem is a classic problem that extends subset sum a little
- As before, there is a maximum capacity W and each item has a weight w_i
- Each item also has a value v_i
- The goal is to maximize the value of objects collected without exceeding the capacity
- … like Indiana Jones trying to put the most valuable objects from a tomb into his limited-capacity knapsack

An easy extension

- The knapsack problem is really the same problem, except that we are concerned with maximum **value** instead of maximum **weight**
- We need only to update the recurrence to keep the maximum value:
  - If $w < w_i$, then
    $$\mathrm{OPT}(i, w) = \mathrm{OPT}(i - 1, w)$$
  - Otherwise,
    $\mathrm{OPT}(i, w) = \max(\mathrm{OPT}(i - 1, w), v_i + \mathrm{OPT}(i - 1, w - w_i))$

Knapsack example

- Items (w_i, vi):
  - $(6, 20)$
  - $(4, 10)$
  - $(3, 9)$
  - $(2, 5)$
- Maximum weight: 8
- Create the table to find all of the optimal values that include items $1, 2, \ldots, i$ for every possible weight w up to 8

Fill in the table

i	w_i	v_i	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0	0	0
1	6	20									
2	4	10									
3	3	9									
4	2	5									

Alignment

- An alignment is a list of matches between characters in strings X and Y that doesn't cross
- Consider:
  - `stop-`
  - `-tops`
- This alignment is $(2,1)$, $(3,2)$, $(4,3)$

Alignment cost

- Some optimal alignment will have the lowest cost
- Cost:
  - Gap penalty $\delta > 0$, for every gap
  - Mismatch cost α_{pq} for aligning p with q
    - α_{pp} is presumably 0 but does not have to be
  - Total cost is the sum of the gap penalties and mismatch costs

Formulating the recurrence

- Let $\mathrm{OPT}(i, j)$ be the minimum cost of an alignment of the first i characters in X to the first j characters in Y
- In case 1, we would have to pay a matching cost of matching the character at i to j
- In cases 2 and 3, you will pay a gap penalty

$$\mathrm{OPT}(i,j) = \min \begin{cases} \alpha_{x_i y_j} + \mathrm{OPT}(i-1, j-1) \\ \delta + \mathrm{OPT}(i-1, j) \\ \delta + \mathrm{OPT}(i, j-1) \end{cases}$$

Now what?

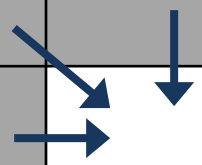
- We do our usual thing
- Build up a table of values with $m + 1$ rows and $n + 1$ columns
- In row 0, column i has value $i\delta$ to build up strings from the empty string
- In column 0, row i has value $i\delta$ to build up strings from the empty string
- The other entries (i, j) can be computed from $(i - 1, j - 1)$, $(i - 1, j)$, $(i, j - 1)$

Alignment(X, Y)

- Create array $A[0 \ldots m][0 \ldots n]$
- For i from 0 to m
  - Set $A[i][0] = i\delta$
- For j from 0 to n
  - Set $A[0][j] = j\delta$
- For i from 1 to m
  - For j from 1 to n
    - Set $A[i][j] = \min(\alpha_{x_i y_j} + A[i-1][j-1], \delta + A[i-1][j], \delta + A[i][j-1])$
- Return $A[m][n]$

Table A of OPT values

0	0	δ	2δ	…	$(j-1)\delta$	$j\delta$	…	$n\delta$
1	δ							
2	2δ							
	…							
$i-1$	$(i-1)\delta$							
i	$i\delta$							
	…							
m	$m\delta$							
	0	1	2	…	$j-1$	j	…	n



Sequence alignment example

- Find the minimum cost to align:
  - "tarnish"
  - "strength"
- The cost of an insertion (or deletion) δ is 1
- The cost of replacing any letter with a different letter is 1
- The cost of "replacing" any letter with itself is 0

Fill in the table

			t	a	r	n	i	s	h
	0	1	2	3	4	5	6	7
s	1							
t	2							
r	3							
e	4							
n	5							
g	6							
t	7							
h	8							

Ticket Out the Door

Upcoming

Next time…

- Review final third of the course
  - Flow networks
  - NP-completeness
  - Approximation algorithms

Reminders

- Finish Assignment 7
  - **Due Wednesday by midnight**
- Review chapters 7, 8, and 11
- Final exam:
  - Friday, April 24, 2026
  - 10:15 a.m. – 12:15 p.m.