Finishing up Dijkstra's

Admin

- Assignment 3 is due Oct 8 (it says Oct 7th on the handout; that's a typo)
- Get started early!
- Reminder: if you're at home, switch to camera using the button on the top-right
- Keep up the Covid caution!
- Any other questions/comments?

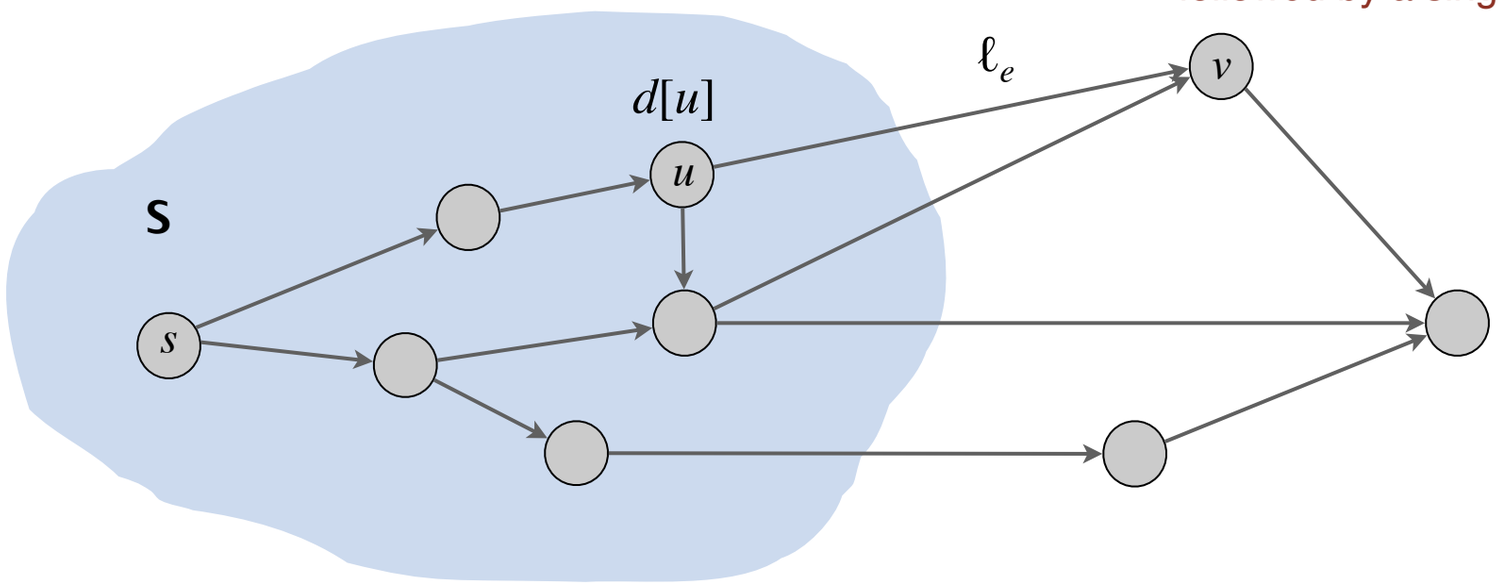
Dijkstra's Algorithm

Greedy approach. Maintain a set of explored nodes S for which algorithm has determined $d[u]$ = length of a shortest $s \rightsquigarrow u$ path.

- Initialize $S \leftarrow \{s\}$, $d[s] \leftarrow 0$.
- Repeatedly add unexplored node $v \notin S$ which minimizes

$$\min_{e=(u,v):u \in S} d[u] + \ell_e$$

the length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$



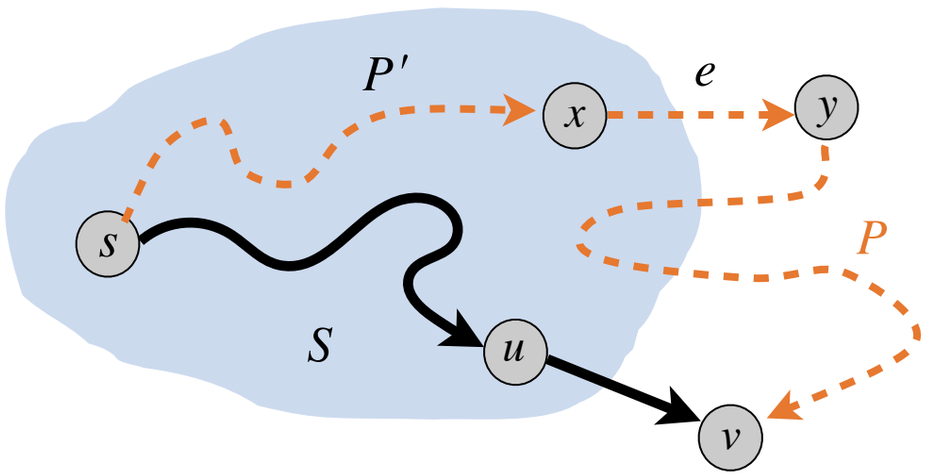
Dijkstra’s Demo

Pseudocode in Textbook

- Use it to test your understanding
- But, high-level idea is most important

Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is length of a shortest s-u path

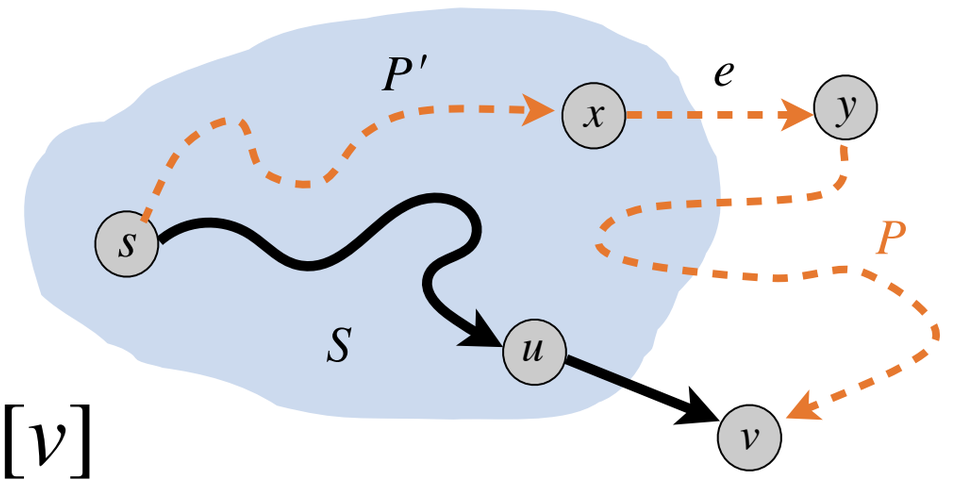


Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is length of a shortest s-u path

Proof. [By induction on $|S|$]. Base case: $|S| = 1$, $S = \{s\}$and $d[s] = 0$. Assume holds for some $k = |S| \geq 1$. Let v be next node added to S

- Consider some other s-v path P in G
- Our goal is to show $w(P) \geq d[v]$
- Let $e = (x, y)$ be the first edge along P that leaves S
- Let P' be the subpath from s to x
- $w(P') \geq d[x]$ (by inductive hypothesis)



Non-negative weights

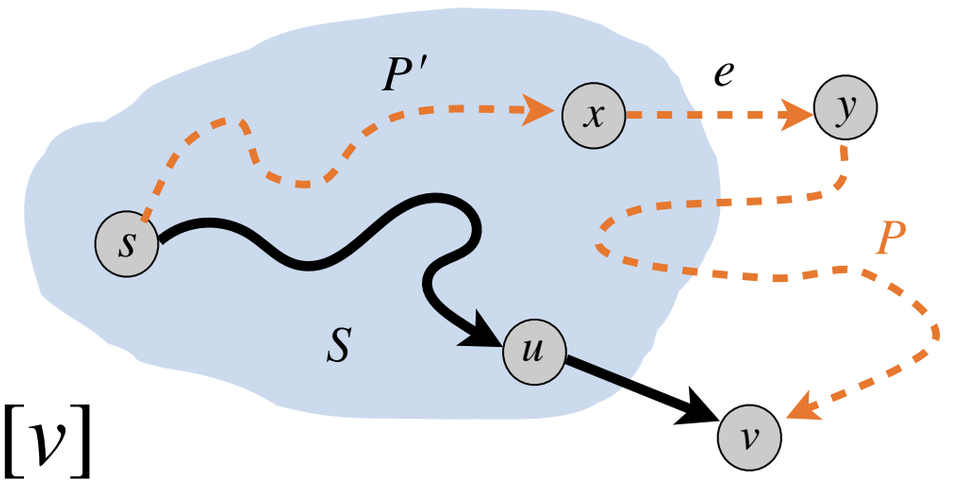
$$w(P) \geq w(P') + w_e \geq d[x] + w_e \geq d[y] \geq d[v]$$

Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is length of a shortest s-u path

Proof. [By induction on $|S|$]. Base case: $|S| = 1$, $S = \{s\}$and $d[s] = 0$. Assume holds for some $k = |S| \geq 1$. Let v be next node added to S

- Consider some other s-v path P in G
- Our goal is to show $w(P) \geq d[v]$
- Let $e = (x, y)$ be the first edge along P that leaves S
- Let P' be the subpath from s to x
- $w(P') \geq d[x]$ (by inductive hypothesis)



Inductive Hypothesis

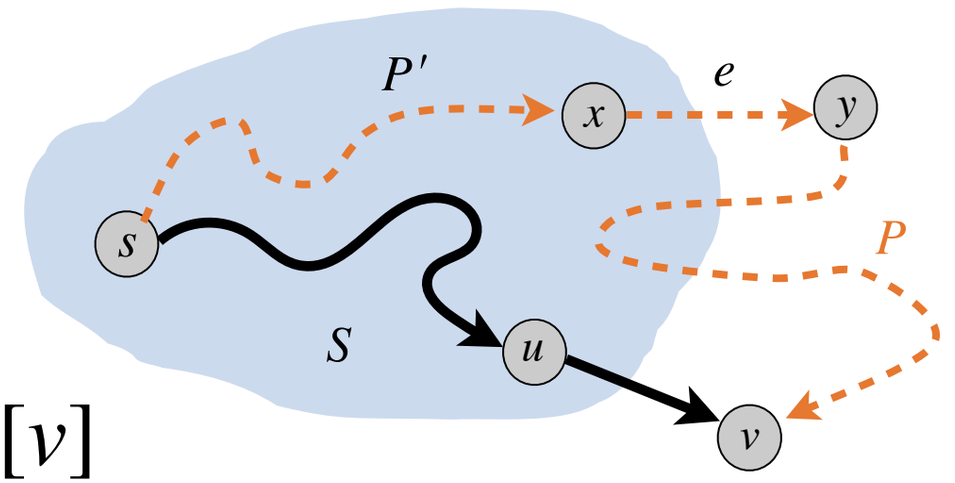
$$w(P) \geq w(P') + w_e \geq d[x] + w_e \geq d[y] \geq d[v]$$

Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is length of a shortest s-u path

Proof. [By induction on $|S|$]. Base case: $|S| = 1$, $S = \{s\}$and $d[s] = 0$. Assume holds for some $k = |S| \geq 1$. Let v be next node added to S

- Consider some other s-v path P in G
- Our goal is to show $w(P) \geq d[v]$
- Let $e = (x, y)$ be the first edge along P that leaves S
- Let P' be the subpath from s to x
- $w(P') \geq d[x]$ (by inductive hypothesis)



When x was added to S, $d[y]$ was updated

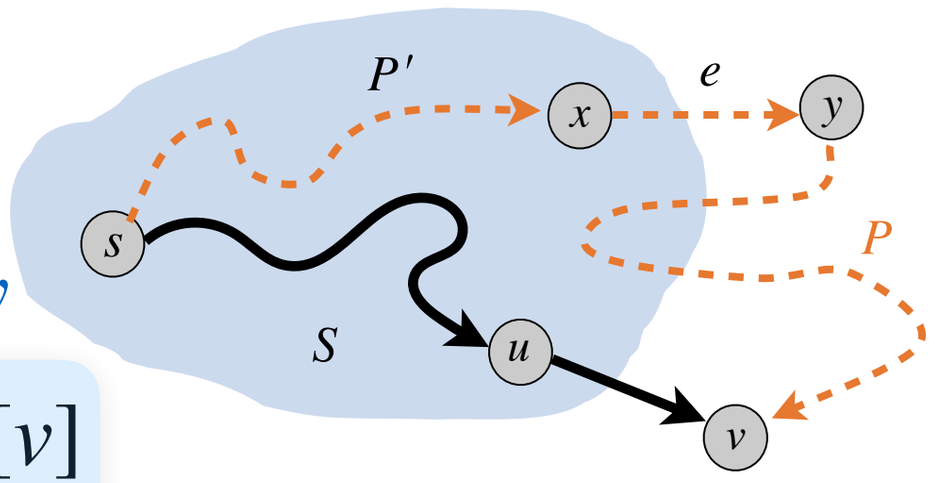
$$w(P) \geq w(P') + w_e \geq d[x] + w_e \geq d[y] \geq d[v]$$

Dijkstra's Algorithm: Correctness

Invariant. For each $u \in S$, $d[u]$ is length of a shortest s-u path

Proof. [By induction on $|S|$]. Base case: $|S| = 1$, $S = \{s\}$and $d[s] = 0$. Assume holds for some $k = |S| \geq 1$. Let v be next node added to S

- Consider some other s-v path P in G
- Our goal is to show $w(P) \geq d[v]$
- Let $e = (x, y)$ be the first edge along P that leaves S
- Let P' be the subpath from s to x
- $w(P') \geq d[x]$ (by inductive hypothesis)



Dijkstra chose to add v instead of y

$$w(P) \geq w(P') + w_e \geq d[x] + w_e \geq d[y] \geq d[v]$$

Implementation & Running Time

How can we efficiently implement Dijkstra's algorithm? We need to be able to

- Visit every neighbor of a vertex
- Maintain set of visited S and unvisited vertices $V - S$
- Maintain a tree of edges $(v, (\text{pred}[v])$
- (Delete-min) Select & delete unvisited vertex v with min d[v]
- (Decrease-key) Update $d[v]$ for unvisited vertices

Priority Queue: Delete-Min & Decrease-Key

Updating the Priority Queue

How to to update priorities (perform decrease-key) in the priority queue efficiently?

- Recall vertices are represented by $1,\ldots,n$
- Maintain an array `PQIndex[1..n]` that holds the index of each vertex v in the priority queue
- **(Decrease-min)** If we update $d[u]$ for some u, we then heapify-up from u's location in the PQ to restore heap property
- Every time we swap two heap elements, we update `PQIndex` for the two vertices

Time and Space Analysis

Space: $O(n+m)$; Running Time:

- Traversal of S (each edge visited at most once)
  - $O(n \log n + m)$
  - Why the $O(\log n)$?
    - n deleteMin operations from PQ to select next vertex $O(n \log n)$
- Construction of T: time proportional to its size:$O(n)$
- Creation of priority queue: $O(n)$
- At most one decrease-key for each edge: $O(m \log n)$

Total time: $O((n+m)\log n)$**.** This is $O(m \log n)$ if G is connected (Why?)

What About Undirected Graphs

How to solve the single-source shortest paths problem in undirected graphs with positive edge lengths?

(a) Replace each undirected edge with two antiparallel edges of same length and run Dijkstra's algorithm on the resulting digraph

(b) Modify Dijkstra's algorithms so that when it processes node u, it consider all edges incident to u (instead of edges leaving u)

(c) Either A or B

(d) Neither A nor B

Shortest Path in Linear Time

[Thorup 1999] Can solve single-source shortest paths problem in undirected graphs with positive **integer** edge lengths in $O(m)$ time.

Remark. Does not explore vertices in increasing distance from s

Undirected Single Source Shortest Paths with Positive Integer Weights in Linear Time

Mikkel Thorup
AT&T Labs—Research

The single source shortest paths problem (SSSP) is one of the classic problems in algorithmic graph theory: given a positively weighted graph G with a source vertex s, find the shortest path from s to all other vertices in the graph.

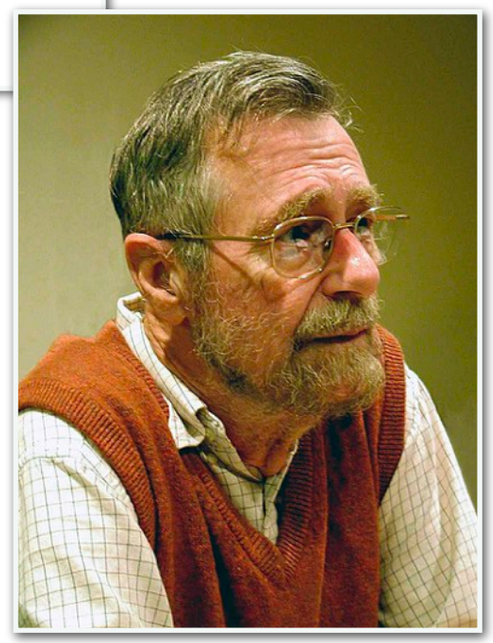
Since 1959 all theoretical developments in SSSP for general directed and undirected graphs have been based on Dijkstra's algorithm, visiting the vertices in order of increasing distance from s. Thus, any implementation of Dijkstra's algorithm sorts the vertices according to their distances from s. However, we do not know how to sort in linear time.

Here, a deterministic linear time and linear space algorithm is presented for the undirected single source shortest paths problem with positive integer weights. The algorithm avoids the sorting bottle-neck by building a hierarchical bucketing structure, identifying vertex pairs that may be visited in any order.

Edsger Dijkstra (1930-2002)

> *“What’s the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path, which I designed in about 20 minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path.”* — Edsger Dijsktra

- Shortest-path algorithm was actually discovered independently (around 1956) by a bunch of different people (read Jeff Erickson’s description and Strigler’s law in CS). *“Leyzorek-Gray-Johnson-Ladew-Meaker-Petry-SeitzDantzig-Dijkstra-Minty-Whiting-Hillier algorithm”*



Recap: Greedy Algorithms

- Scheduling non-conflicting jobs (intervals)
    - Earliest finish-times first
    - Greedy stays ahead to prove correctness
- Scheduling with deadlines to maximize lateness of jobs
    - Earliest-deadline first
    - Exchange argument to prove correctness
- Minimum spanning trees: greedily pick edges
    - Cut property: essentially a non-local exchange argument
    - Prims, Kruskals: correctness from cut property
    - Union find data structure
- Djisktra's shortest path: greedily find paths

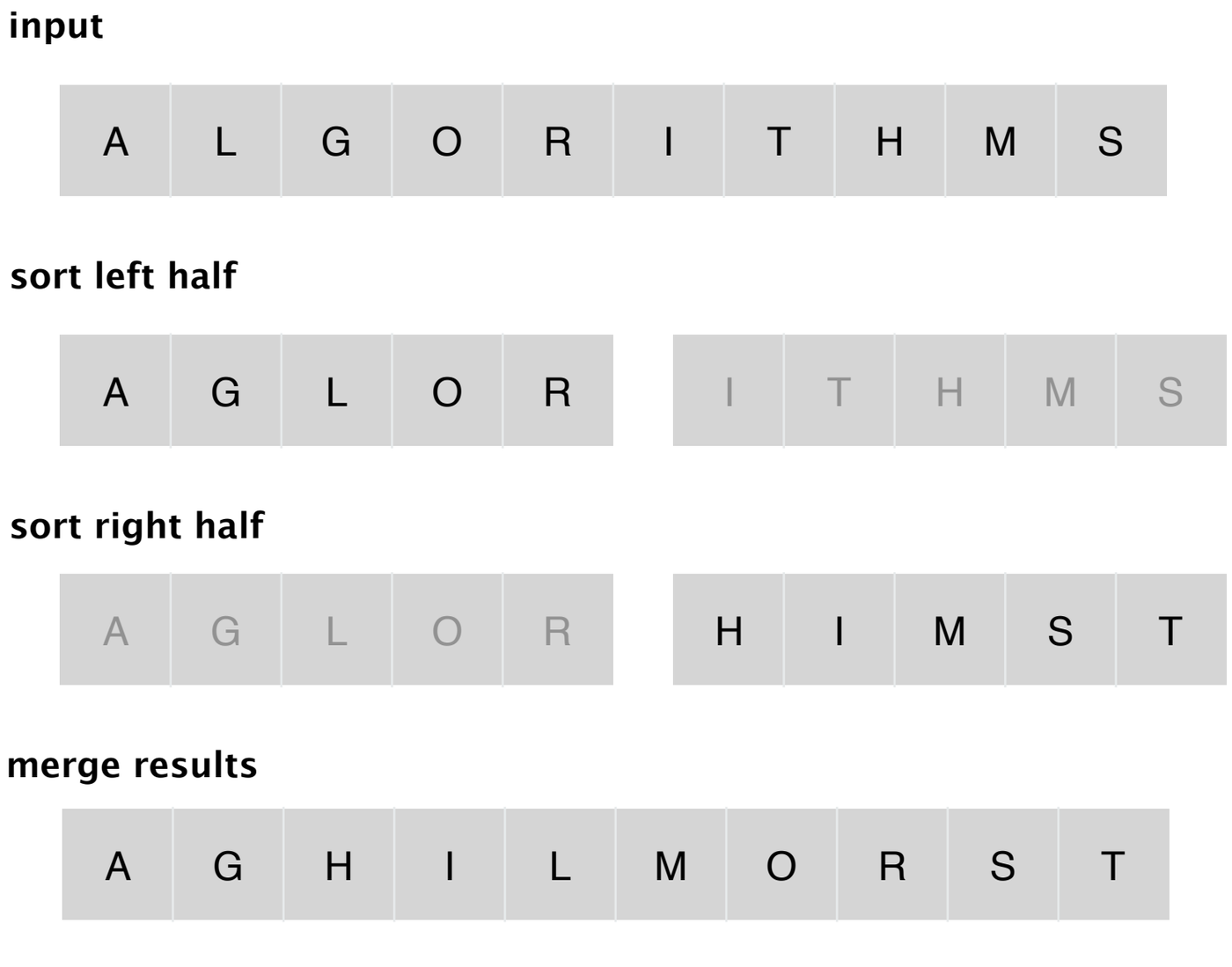
Divide and Conquer: Sorting and Recurrences

Divide & Conquer: The Pattern

- **Divide** the problem into several independent smaller instances of exactly the same problem
- **Delegate** each smaller instance to the **Recursion Fairy** (technically known as induction hypothesis)
- **Combine** the solutions for the smaller instances
  - Assume the recursion fairy correctly solves the smaller instances, how can you combine them?

Recap: Merge Sort

- Divide input array into 2 subarrays of roughly equal size
- Recursively mergesort each of the subarrays
- Merge the sorted subarrays into a single sorted array



Recap: Merge Sort

MERGE-SORT(L)

IF (list L has one element)

  RETURN L.

Divide the list into two halves A and B.

$A \leftarrow$ MERGE-SORT(A). ⟵ $T(n / 2)$

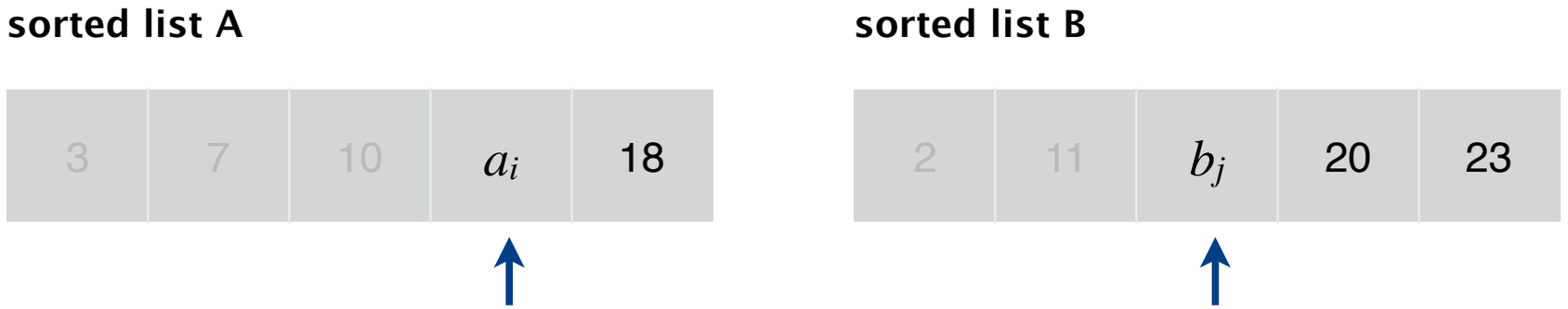
$B \leftarrow$ MERGE-SORT(B). ⟵ $T(n / 2)$

$L \leftarrow$ MERGE(A, B). ⟵ $\Theta(n)$

RETURN L.

Merge Step: $\Theta(n)$

- Scan subarrays from left to right
- Compare element by element; create new merged array



Correctness of D&C Algorithms

- **Correctness proof pattern:**
  - Natural proof by induction pattern
  - Show base case holds, rely on the recursion fairy (I mean induction hypothesis) to prove the inductive step
  - Often the crux on the proof is showing that the solutions returned by the recursive calls are **"combined"** correctly

- **Claim. (Combine step.)** Merge subroutine correctly merges two sorted subarrays $A[1,\ldots i]$ and $B[1,\ldots,j]$ where $i+j=n$.
  - Prove that for the first k iterations of the loop correctly merge A and B for $k=1$ to n.
  - **Invariant.** Merged array is sorted after every iteration and contains the k smallest elements in A or B
  - Base case: $k=1$,

Correctness of D&C Algorithms

Claim. (Combine step.) Merge subroutine correctly merges two sorted subarrays $A[1,\ldots i]$ and $B[1,\ldots,j]$ where $i+j=n$.

- Prove that for the first k iterations of the loop correctly merge A and B for $k=1$ to n.
- **Invariant.** Merged array is sorted after every iteration and contains the k smallest elements in A or B
  - Base case: $k=1$
  - Inductive step:
    - The $k+1$st smallest element must be the "next" element of A or B since they are sorted
    - In fact it is the smallest of them
    - Therefore, the merge correctly places the $k+1$st smallest element. With the inductive hypothesis, the $k+1$st iteration gives the correct output

Merge-Sort Running Time Recurrence

- Let $T(n)$ represent how long Merge Sort takes on an input of size n
- $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n)$
- **Base case:** $T(1) = 1$; often ignored
- We will ignore the floors and ceilings
    - Our methods will tolerate such sloppiness
    - For asymptotic bounds, turns out it doesn't matter

- So the recurrence simplifies to:
    - $T(n) = 2T(n/2) + O(n)$
    - The answer to this ends up being $T(n) = O(n \log n)$
    - Today we will learn different ways to derive it

Recurrences: Unfolding

Method 1. Unfolding the recurrence

- Assume $n = 2^\ell$ (that is, $\ell = \log n$)
- Because we don't care about constant factors and are only upper-bounding, we can always choose smallest power of 2 greater than that is, $n < n' = 2^\ell < 2n$

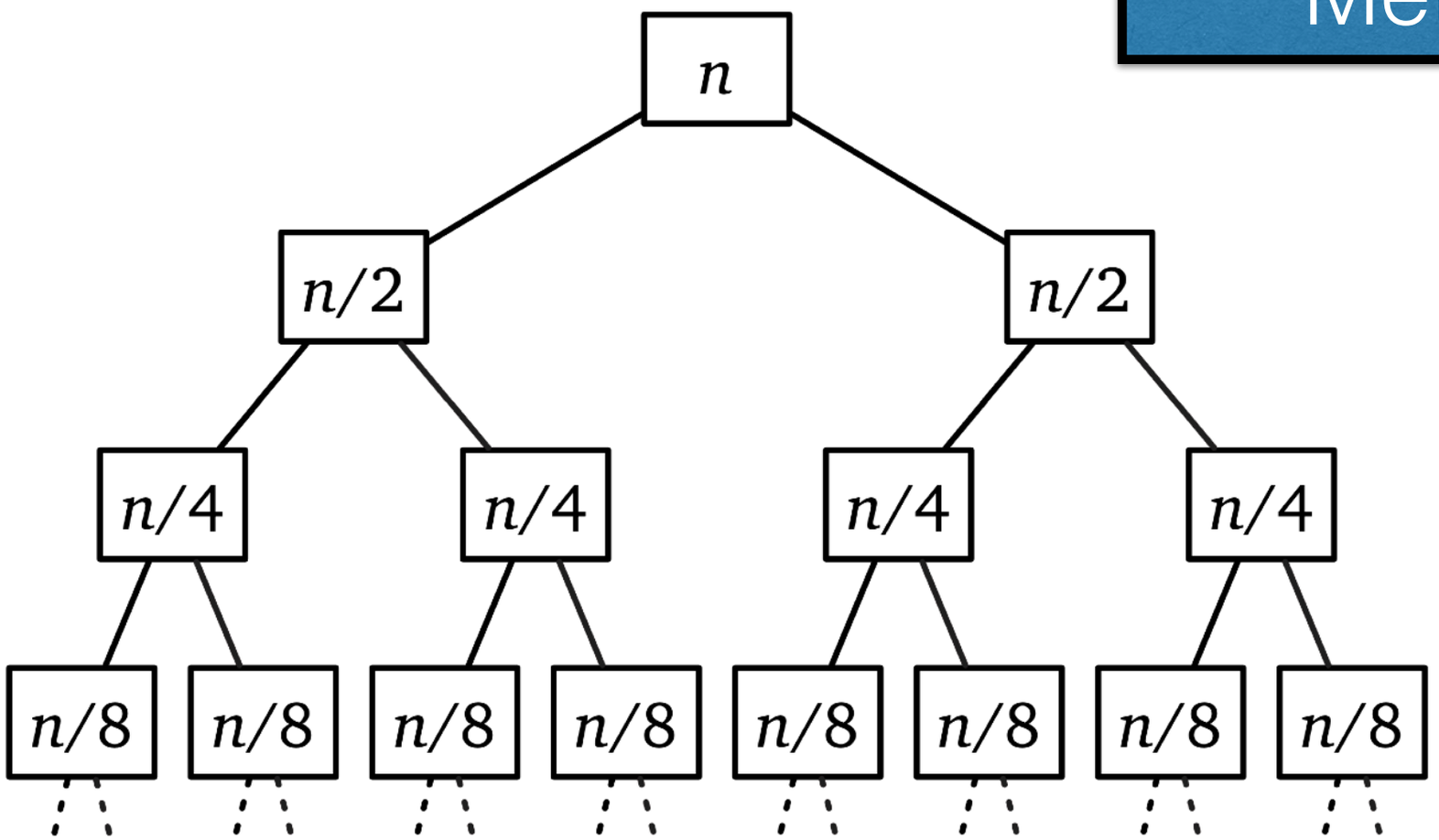
$$
\begin{aligned}
T(n) &= 2T(n/2) + cn \\
&= 2T(2^{\ell-1}) + c2^\ell \\
&= 2(2T(2^{\ell-2}) + c2^{\ell-1}) + c2^\ell = 2^2T(2^{\ell-2}) + 2 \cdot c2^\ell \\
&= 2^3T(2^{\ell-3}) + 3 \cdot 2^\ell \\
&= \ldots = 2^\ell T(2^0) + c\ell 2^\ell = O(n \log n)
\end{aligned}
$$

Recurrences: Recursion Tree

Method 2. Recursion Trees

- Work done at each level $2^i \cdot (n/2^i) = n$
- Total $\log_2 n$ levels

Recommended Method!



Recurrences: Guess & Verify

Method 3. Guess and Verify

- Eyeball recurrence and make a guess
- Verify guess using induction
- **More on this tomorrow…**

Divide & Conquer: Quicksort

- Choose a pivot element from the array
- Partition the array into two parts: left less than the pivot, right greater than the pivot
- Recursively quicksort the first and last subarrays

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Choose a pivot:	S	O	R	T	I	N	G	E	X	A	M	**P**	L
Partition:	A	G	O	E	I	N	L	M	**P**	T	X	S	R
Recurse Left:	A	E	G	I	L	M	N	O	**P**	T	X	S	R
Recurse Right:	A	E	G	I	L	M	N	O	**P**	R	S	T	X

Divide & Conquer: Quicksort

- Choose a pivot element from the array
- Partition the array into two parts: left less than the pivot, right greater than the pivot
- Recursively quicksort the first and last subarrays
- **Description.** (Divide and conquer): often the cleanest way to present is **short and clean pseudocode** with high level explanation
- **Correctness proof.** Induction and showing that partition step correctly partitions the array.

```
QUICKSORT(A[1..n]):
  if (n > 1)
      Choose a pivot element A[p]
      r ← PARTITION(A, p)
      QUICKSORT(A[1..r − 1])      ⟨⟨Recurse!⟩⟩
      QUICKSORT(A[r + 1..n])      ⟨⟨Recurse!⟩⟩
```

Quick Sort Analysis

- Partition takes $O(n)$ time
- Size of the subproblems depends pivot; let r be the rank of the pivot, then:
- $T(n) = T(r-1) + T(n-r) + O(n),\ \ T(1) = 1$
- Let us analyze some cases for r
    - **Best case:** r is the median: $r = \lfloor n/2 \rfloor$ (how fast can we compute the median?)
    - **Worst case:** $r = 1$ or $r = n$
    - **In between:** say $n/10 \leq r \leq 9n/10$
- Note in the worst-case analysis, we only consider the worst case for r. We are looking at the difference cases, just to get a sense for it.

Quick Sort: Cases

- Suppose $r = n/2$ (pivot is the median element), then
  - $T(n) = 2T(n/2) + O(n)$, $T(1) = 1$
  - We have already solved this recurrence
  - $T(n) = O(n \log n)$

- Suppose $r = 1$ or $r = n - 1$, then
  - $T(n) = T(n-1) + T(1) + 1$
  - What running time would this recurrence lead to?
  - $T(n) = \Theta(n^2)$ (notice: this is tight!)

Quick Sort: Cases

- Suppose $r = n/10$ (that is, you get a one-tenth, nine-tenths split
- $T(n) = T(n/10) + T(9n/10) + O(n)$
- Let's look at the recursion tree for this recurrence

- We get $T(n) = O(n \log n)$, in fact, we get $\Theta(n \log n)$

- In general, the following holds (we'll show it later):

- $T(n) = T(\alpha n) + T(\beta n) + O(n)$
  - If $\alpha + \beta < 1 : T(n) = O(n)$
  - If $\alpha + \beta = 1, T(n) = O(n \log n)$

Quick Sort: Theory and Practice

- We can find the **median element in** $\Theta(n)$ time
    - Using divide and conquer! we'll learn how in next lecture
- In practice, the constants hidden in the Oh notation for median finding are too large to use for sorting
- Common heuristic
    - Median of three (pick elements from the start, middle and end and take their median)
- If the pivot is chosen **uniformly at random**
    - quick sort runs in time $O(n \log n)$ in expectation and *with high probability*
    - We will prove this in the second half of the class

Acknowledgments

- Some of the material in these slides are taken from
  - Kleinberg Tardos Slides by Kevin Wayne (https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf)
  - Jeff Erickson's Algorithms Book (http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf)