Randomized Algorithms

Part Two

Outline for Today

- **Quicksort**
  - Can we speed up sorting using randomness?
- **Indicator Variables**
  - A powerful and versatile technique in randomized algorithms.
- **Randomized Max-Cut**
  - Approximating **NP**-hard problems with randomized algorithms.

Quicksort

Quicksort

- **Quicksort** is as follows:
  - If the sequence has 0 elements, it is sorted.
  - Otherwise, choose a pivot and run a partitioning step to put it into the proper place.
  - Recursively apply quicksort to the elements strictly to the left and right of the pivot.

Initial Observations

- Like the partition-based selection algorithms, quicksort's behavior depends on the choice of pivot.
- **Really good case:** Always pick the median element as the pivot:

$$T(0) = \Theta(1)$$
$$T(n) = 2T(\lfloor n / 2 \rfloor) + \Theta(n)$$

$$\mathbf{T(n) = \Theta(n \log n)}$$

Initial Observations

- Like the partition-based selection algorithms, quicksort's behavior depends on the choice of pivot.
- **Really bad case:** Always pick the min or max element as the pivot:

$$T(0) = \Theta(1)$$
$$T(n) = T(n - 1) + \Theta(n)$$

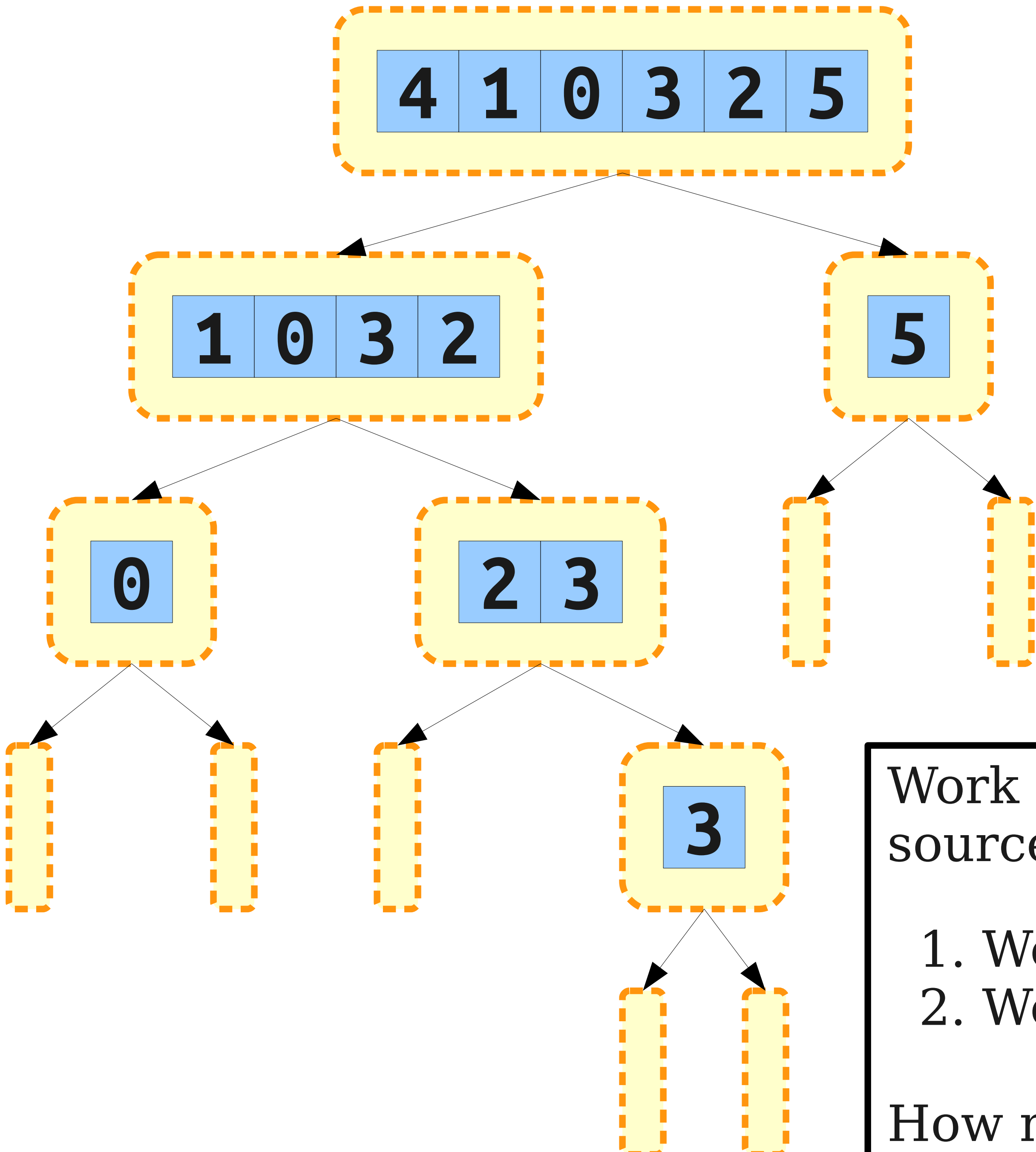
$$\mathbf{T(\mathit{n}) = \Theta(\mathit{n}^2)}$$

Choosing Random Pivots

- As with quickselect, we can ask this question: what happens if you pick pivots purely at random?
- This is called **randomized quicksort**.
- Question: What is the expected runtime of randomized quicksort?

Accounting Tricks

- As with quickselect, we will *not* try to analyze quicksort by writing out a recurrence relation.
- Instead, we will try to account for the work done by the algorithm in a different but equivalent method.
- This will keep the math a *lot* simpler.



Work done comes from two sources:

1. Work making recursive calls
2. Work partitioning elements.

How much work is from each source?

Counting Recursive Calls

- When the input array has size $n > 0$, quicksort will
  - Choose a pivot.
  - Recurse on the array formed from all elements before the pivot.
  - Recurse on the array formed from all elements after the pivot.
- Given this information, can we bound the total number of recursive calls the algorithm will make?

Counting Recursive Calls

- Begin with an array of n elements.
- Each recursive call deletes one element from the array and recursively processes the remaining subarrays.
- Therefore, there will be n recursive calls on nonempty subarrays.
- Therefore, can be at most $n + 1$ leaf nodes with calls on arrays of size 0.
- Would expect $2n + 1 =$ **$\Theta(n)$** recursive calls regardless of how the recursion plays out.

Counting Recursive Calls

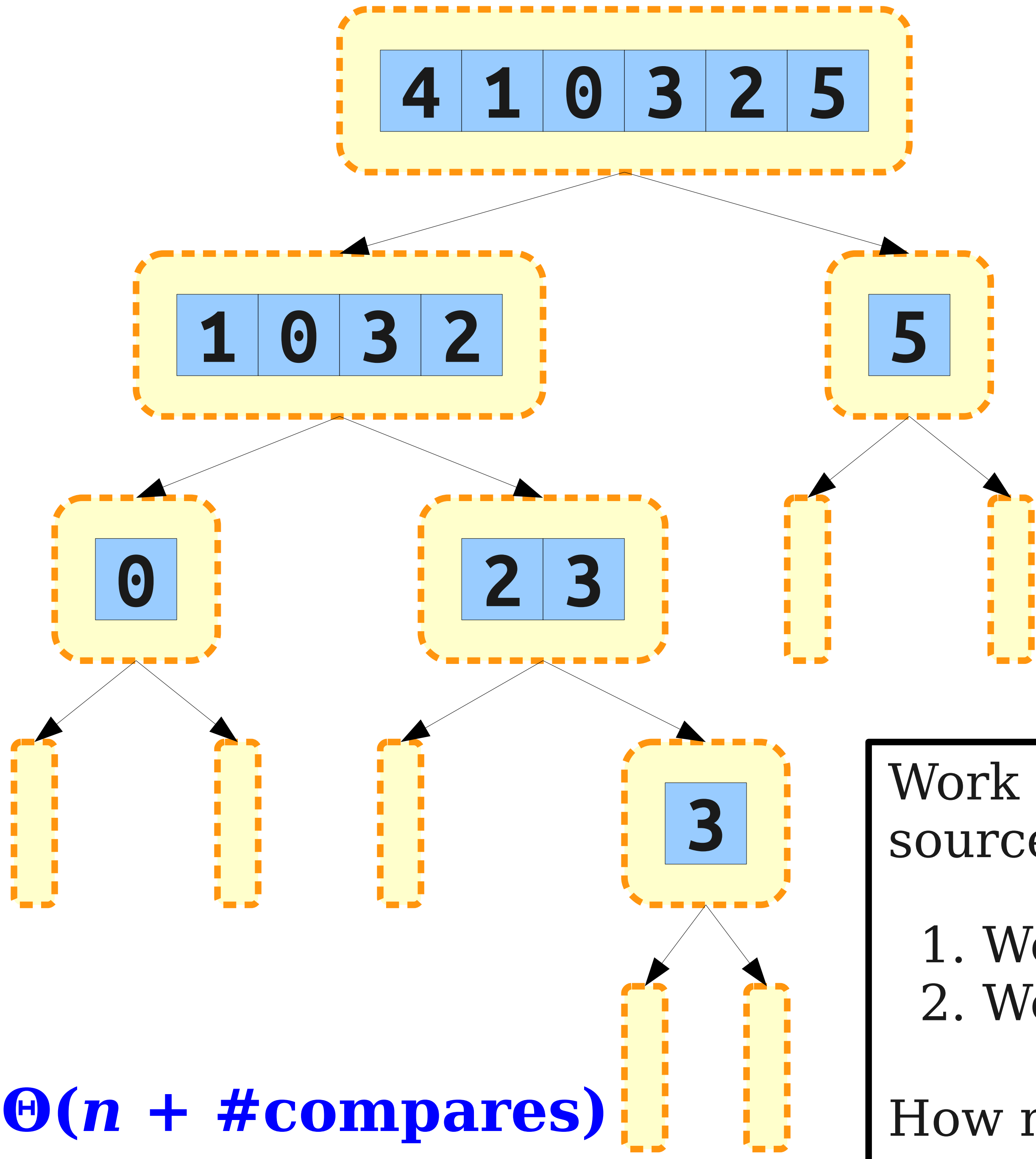
Theorem: On any input of size n, quicksort will make exactly $2n + 1$ total recursive calls.

Proof: By induction. As a base case, the claim is true when $n = 0$ since just one call is made.

Assume the claim is true for $0 \leq n' < n$. Then quicksort will split the input apart into a piece of size k and a piece of size $n - k - 1$. The first piece leads to at most $2k + 1$ calls and the second to $2n - 2k - 2 + 1 = 2n - 2k - 1$ calls. This gives a total of $2n$ calls, and adding in the initial call yields a total of $2n + 1$ calls. ■

Counting Partition Work

- From before: running partition on an array of size n takes time $\Theta(n)$.
- More precisely: running partition on an array of size n can be done making exactly $n - 1$ comparisons.
- **Idea:** Account for the total work done by the partition step by summing up the total number of comparisons made.
- Will only be off by $\Theta(n)$ (the -1 term from n calls to partition); can fix later.



$\Theta(n + \#\text{compares})$

Work done comes from two sources:

1. Work making recursive calls
2. Work partitioning elements.

How much work is from each source?

Counting Comparisons

- One way to count up total number of comparisons: Look at the sizes of all subarrays across all recursive calls and sum up across those.
- Another way to count up total number of comparisons: Look at all pairs of elements and count how many times each of those pairs was compared.
- Account “vertically” rather than “horizontally”

Return of the Random Variables

- Let's denote by v_i the ith largest value of the array to sort, using 1-indexing.
  - For now, assume no duplicates.
- Let C_{ij} be a random variable equal to the number of times v_i and v_j are compared.
- The total number of comparisons made, denoted by the random variable X, is

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} C_{ij}$$

Expecting the Unexpected

- The expected number of comparisons made is E[X], which is

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n} \sum_{j=i+1}^{n} C_{ij}\right] \\ &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[C_{ij}] \end{aligned}$$

(Isn't linearity of expectation great?)

When Compares Happen

- We need to find a formula for $\text{E}[C_{ij}]$, the number of times v_i and v_j are compared.
- Some facts about partition:
  - All $n - 1$ elements other than the pivot are compared against the pivot.
  - No other elements are compared.
- Therefore, v_i and v_j are compared only when v_i or v_j is a pivot in a partitioning step.

When Compares Happen

- **Claim:** If v_i and v_j are compared once, they are never compared again.
- Suppose v_i and v_j are compared. Then either v_i or v_j is a pivot in a partition step.
- The pivot is never included in either subarray in a recursive call.
- Consequently, this is the only time that v_i and v_j will be compared.

Defining C_{ij}

- We can now give a more rigorous definition of C_{ij}:

$$C_{ij} = \begin{cases} 1 & \text{if } v_i \text{ and } v_j \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$$

- Given this, $\mathrm{E}[C_{ij}]$ is given by

$$\begin{aligned} \mathrm{E}[C_{ij}] &= 0 \cdot P(C_{ij}=0) + 1 \cdot P(C_{ij}=1) \\ &= P(C_{ij}=1) \\ &= P(v_i \text{ and } v_j \text{ are compared}) \end{aligned}$$

Our Expected Value

- Using the fact that

$$E[C_{ij}] = P(v_i \text{ and } v_j \text{ are compared})$$

we have

$$\begin{aligned} E[X] &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[C_{ij}] \\ &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} P(v_i \text{ and } v_j \text{ are compared}) \end{aligned}$$

- Amazingly, this reduces to a sum of probabilities!

Indicator Random Variables

- An **indicator random variable** is a random variable of the form

$$X = \begin{cases} 1 & \text{if event } \mathcal{E} \text{ occurs} \\ 0 & \text{otherwise} \end{cases}$$

- For an indicator random variable X with underlying event $\mathcal{E}$, $\mathrm{E}[X] = P(\mathcal{E})$.
- This interacts very nicely with linearity of expectation, as you just saw.
- We will use indicator random variables extensively when studying randomized algorithms.

What is the probability v_i and v_j are compared?

Comparing Elements

- ***Claim:*** v_i and v_j are compared iff v_i or v_j is the first pivot chosen from v_i, v_{i+1}, v_{i+2}, …, v_{j-1}, v_j.
- ***Proof Sketch:*** v_i and v_j are together in the same array as long as no pivots from this range are chosen. As soon as a pivot is chosen from here, they are separated. They are only compared iff v_i or v_j is the chosen pivot.
- ***Corollary:***

$$P(v_i \text{ and } v_j \text{ are compared}) = 2 / (j - i + 1)$$

Plugging and Chugging

$$\mathrm{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} P(v_i \text{ and } v_j \text{ are compared})$$

$$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Let $\boldsymbol{k = j - i}$. Then $k + i = j$, so we can just the loop bounds as

$$i + 1 \leq j \leq n$$

$$i + 1 \leq k + i \leq n$$

$$\boldsymbol{1 \leq k \leq n - i}$$

Plugging and Chugging

$$
\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} P(v_i \text{ and } v_j \text{ are compared}) \\
&= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&\leq \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{2}{k+1} \\
&= n \sum_{k=1}^{n} \frac{2}{k+1} = 2n \sum_{k=1}^{n} \frac{1}{k+1} \leq 2n \sum_{k=1}^{n} \frac{1}{k}
\end{aligned}
$$

Harmonic Numbers

- The nth **harmonic number**, denoted H_n, is defined as

$$H_n = \sum_{i=1}^{n} \frac{1}{i}$$

- Some values:
  - $H_0 = 0$
  - $H_1 = 1$
  - $H_2 = 3/2$

$H_3 = 11 / 6$

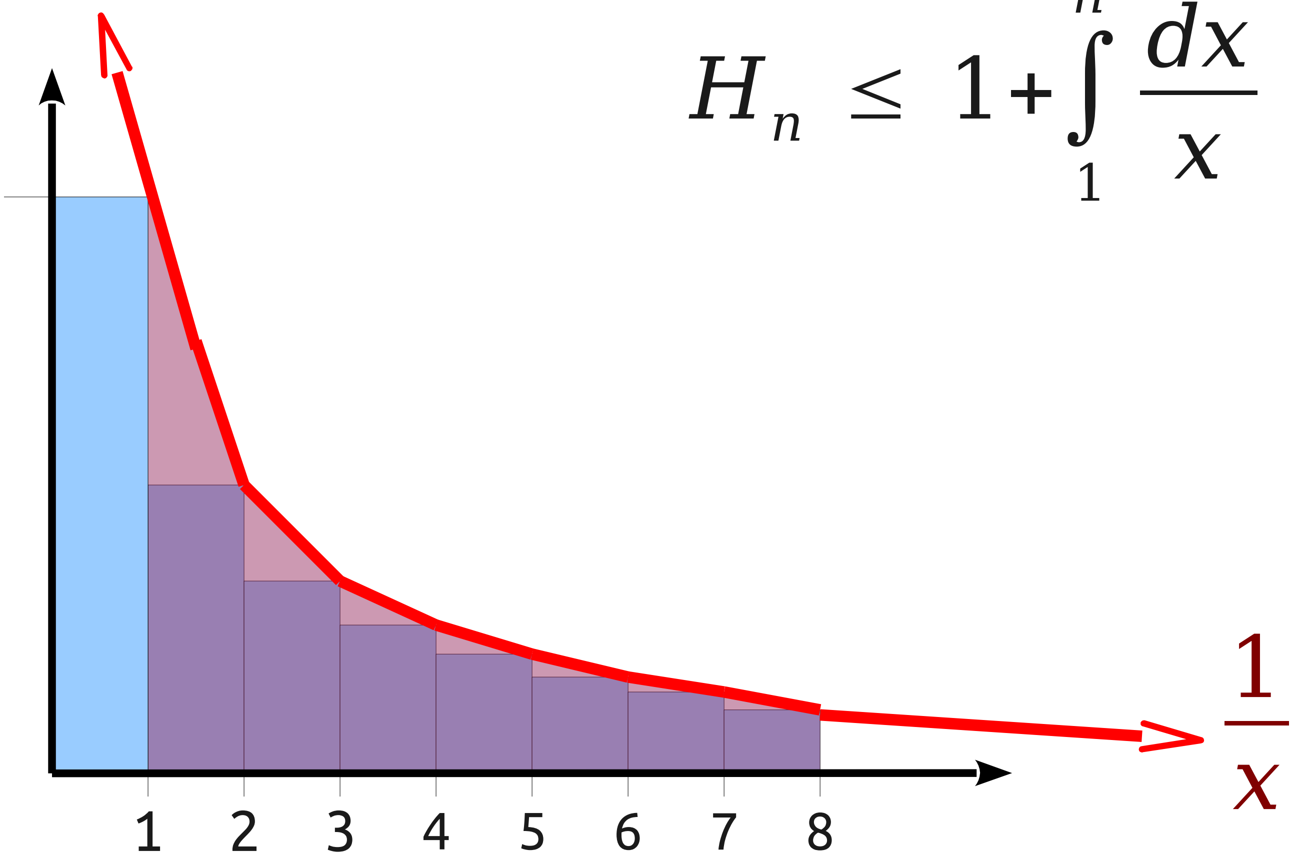
$H_4 = 25 / 12$

$H_5 = 137 / 60$

Mathematical Harmony

- ***Theorem:*** $H_n = \Theta(\log n)$
- ***Proof Idea:***

$$H_n \leq 1 + \int_1^n \frac{dx}{x} = \ln n + 1$$



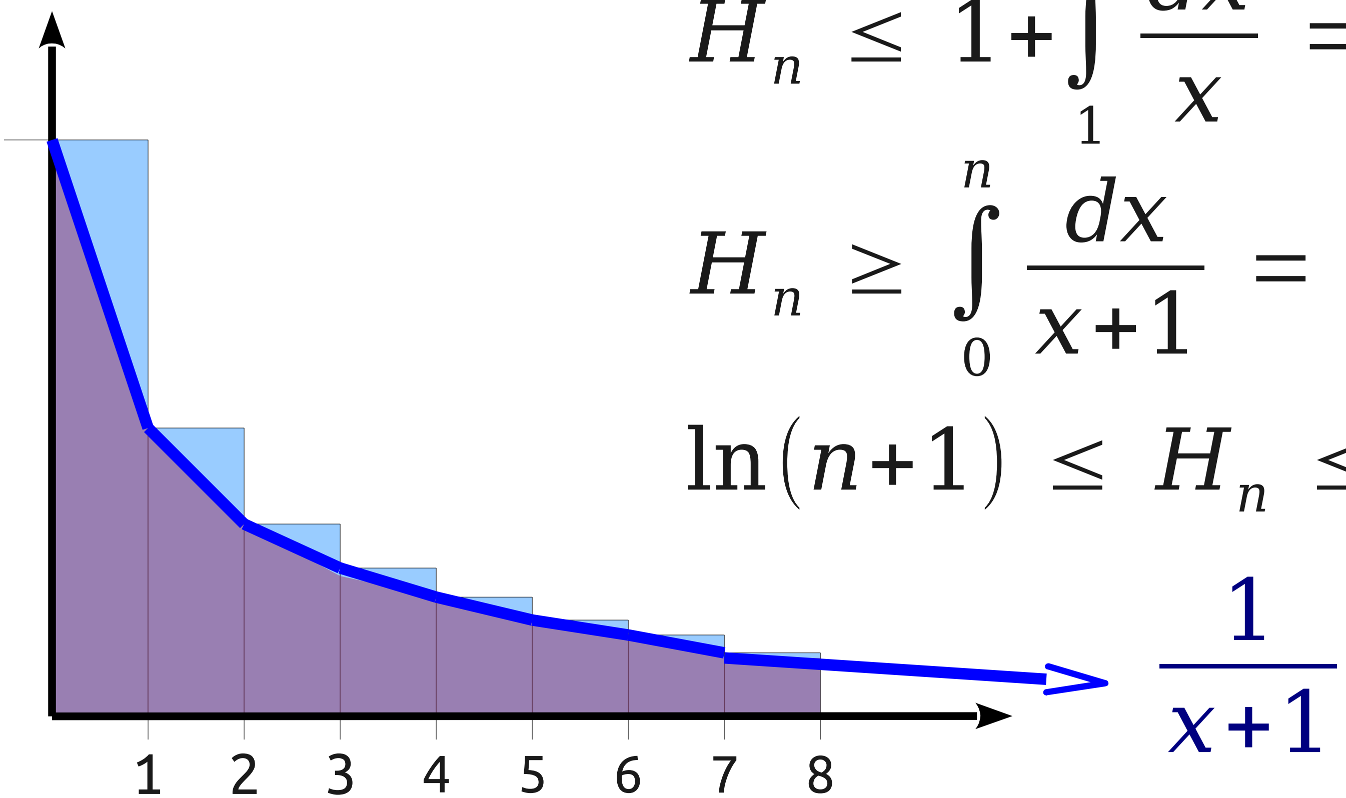
Mathematical Harmony

- ***Theorem:*** $H_n = \Theta(\log n)$
- ***Proof Idea:***

$$H_n \leq 1 + \int_1^n \frac{dx}{x} = \ln n + 1$$

$$H_n \geq \int_0^n \frac{dx}{x+1} = \ln(n+1)$$

$$\ln(n+1) \leq H_n \leq \ln n + 1$$



The Finishing Touches

$$
\begin{aligned}
\mathrm{E}[X] &\le 2n\sum_{k=1}^{n}\frac{1}{k} \\
&= 2n\cdot H_n \\
&= 2n\cdot\Theta(\log n) \\
&= \mathrm{O}(n\log n)
\end{aligned}
$$

Why This Matters

- We have just shown that the runtime of randomized quicksort is, on expectation, O(n log n).
- To do so, we needed to use two new mathematical techniques:
  - Indicator random variables.
  - Bounding summations by integrals.
- We will use the first of these techniques more extensively over the next few days.

Introsort

- As with quickselect, quicksort still has a pathological $\Theta(n^2)$ case, though it's unlikely.
- Quicksort is, on average, faster than heapsort.
- The **introsort** algorithm addresses this:
  - Run quicksort, tracking the recursion depth.
  - If it exceeds some limit, switch to heapsort.
- Given good pivots, runs just as fast as quicksort.
- Given bad pivots, is only marginally worse than heapsort.
- Guarantees $O(n \log n)$ behavior.

A Different Algorithm: **Max-Cut**

Global Cuts

- Given an undirected graph $G = (V, E)$, a **cut** in G is a pair $(S, V - S)$ of two sets S and $V - S$ that split the nodes into two groups.
- The **size** or **cost** of a cut, denoted by $c(S, V - S)$, is the number of edges with one endpoint in S and one in $V - S$.
- A **global min cut** is a cut in G with the least total cost. A **global max cut** is a cut in G with maximum total cost.

Global Cuts

- Interestingly:
  - There are many polynomial-time algorithms known for global min-cut.
  - Global max-cut is **NP**-hard and no polynomial-time algorithms are known for it.
- Today, we'll see an algorithm for approximating global max-cut.
- On Friday, we'll see a randomized algorithm for finding a global min-cut.

Approximating Max-Cut

- For a maximization problem, an **α-approximation algorithm** is an algorithm that produces a value that is within a factor of α of the true value.
- A 0.5-approximation to max-cut would produce a cut whose size is at least 50% the size of the true largest cut.
- Our goal will be to find a randomized approximation algorithm for max-cut.

A Really Simple Algorithm

- Here is our algorithm:
  - For each node, toss a fair coin.
  - If it lands heads, place the node into one part of the cut.
  - If it lands tails, place the node into the other part of the cut.

Analyzing the Algorithm

- On expectation, how large of a cut will this algorithm find?
- For each edge e, C_e be an indicator random variable where

$$C_e = \begin{cases} 1 & \text{if } e \text{ crosses the cut} \\ 0 & \text{otherwise} \end{cases}$$

- Then the number of edges X crossing the cut will be given by

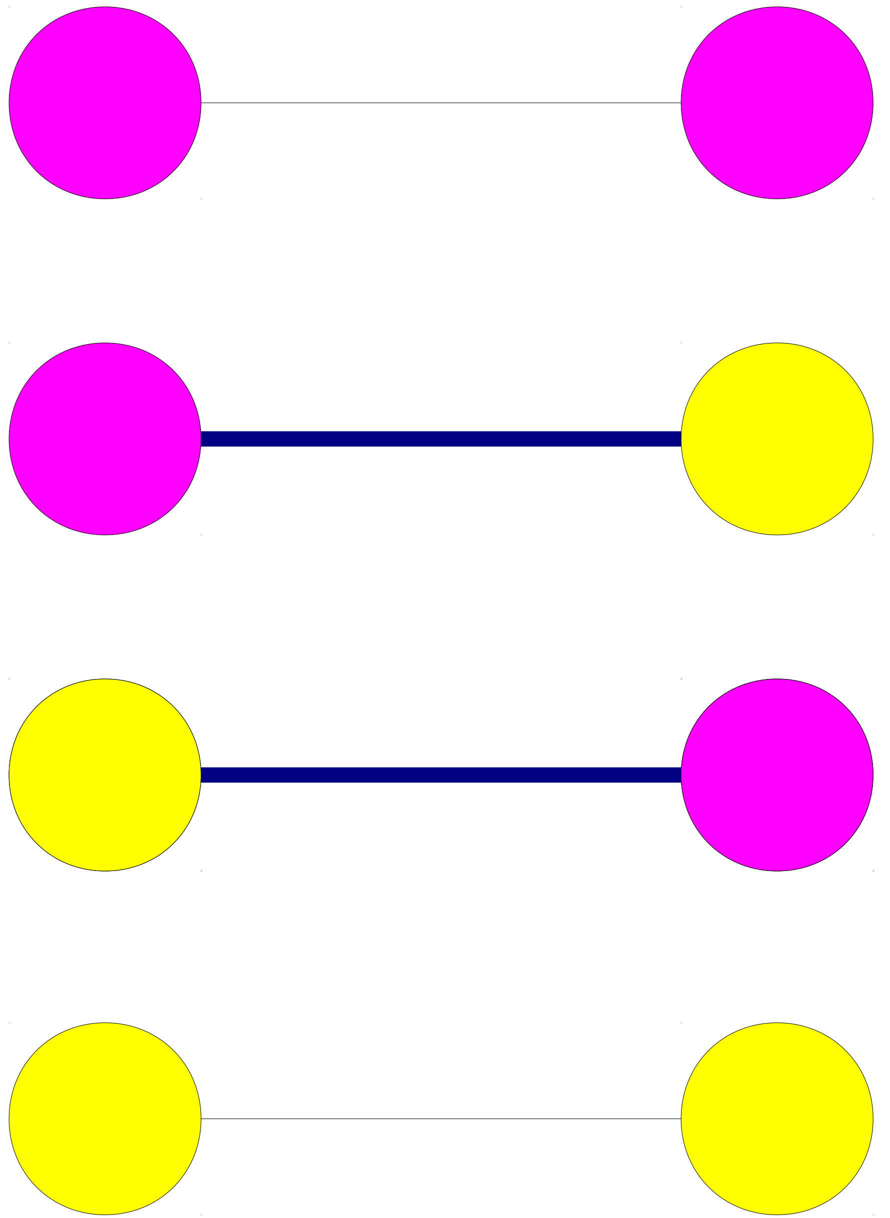
$$X = \sum_{e \in E} C_e$$

What Did You Expect?

- The expected number of edges crossing the cut is given by $\mathrm{E}[X]$.
- This is

$$
\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{e \in E} C_e\right] \\
&= \sum_{e \in E} \mathrm{E}[C_e] \\
&= \sum_{e \in E} P(e \text{ crosses the cut})
\end{aligned}
$$

Four Possibilities



That Was Unexpected

- The expected number of edges crossing the cut is given by $\mathrm{E}[X]$.
- This is

$$\begin{aligned}\mathrm{E}[X] &= \sum_{e \in E} P(e \text{ crosses the cut}) \\ &= \sum_{e \in E} \frac{1}{2} \\ &= \frac{m}{2}\end{aligned}$$

- All cuts have size $\leq m$, so this is always within a factor of two of optimal!

Randomized Approximation Algorithms

- This algorithm is a randomized 0.5-approximation to max-cut.
- The algorithm runs in time O(n).
- It's **NP**-hard to find a true maximum cut, but it's not at all hard to (on expectation) find a cut that has size at least half that of the maximum cut!

Improving the Odds

- Running our algorithm will, on expectation, produce a cut with size m / 2.
- However, we don't know the actual probability that our cut has this size.
- We can use a standard technique to amplify the probability of success.

Do it Again

- Since any *individual* run of the algorithm might not produce a large cut, we could try this approach:
  - Run the algorithm k times.
  - Return the largest cut found.
- Goal: Show that with the right choice of k, this returns a large cut with high probability.
  - Specifically: Will show we get a cut of size $m / 4$ with high probability.
- Runtime is $O((m + n)k)$: k rounds of doing $O(m + n)$ work (n to build the cut, m to determine the size.)

More Probabilities

- Let X_1, X_2, …, X_k be random variables corresponding to the sizes of the cuts found by each run of the algorithm.
- Let $\mathcal{E}$ be the event that our algorithm produces a cut of size less than $m / 4$. Then

$$\mathcal{E} = \bigcap_{i=1}^{k} \left(X_i \le \frac{m}{4} \right)$$

- Since all X_i variables are independent, we have

$$P(\mathcal{E}) = P\left(\bigcap_{i=1}^{k} \left(X_i \le \frac{m}{4} \right) \right) = \prod_{i=1}^{k} P\left(X_i \le \frac{m}{4} \right)$$

A Simplification

- Let $Y_1, Y_2, \ldots, Y_k$ be random variables defined as follows:

$$Y_i = m - X_i$$

- Then

$$P(\mathcal{E}) = \prod_{i=1}^{k} P\left(X_i \leq \frac{m}{4}\right) = \prod_{i=1}^{k} P\left(Y_i \geq \frac{3m}{4}\right)$$

- What now?

Markov's Inequality

- **Markov's Inequality** states that for any nonnegative random variable X, that

$$P(X \geq c) \leq \frac{\mathrm{E}[X]}{c}$$

- Equivalently:

$$P(X \geq c\,\mathrm{E}[X]) \leq \frac{1}{c}$$

- This holds for any random variable X.
- Can often get tighter bounds if we know something about the distribution of X.

Markov to the Rescue

- Let $Y_1, Y_2, \ldots, Y_k$ be random variables defined as follows:

$$Y_i = m - X_i$$

- Then

$$\mathrm{E}[Y_i] = m - \mathrm{E}[X_i] = m - m / 2 = m / 2$$

- Then

$$\begin{aligned} P(\mathcal{E}) &= \prod_{i=1}^{k} P\left(Y_i \geq \frac{3m}{4}\right) \leq \prod_{i=1}^{k} \frac{\mathrm{E}[Y_i]}{3m/4} \\ &= \prod_{i=1}^{k} \frac{m/2}{3m/4} = \prod_{i=1}^{k} 2/3 = \left(\frac{2}{3}\right)^k \end{aligned}$$

The Finishing Touches

- If we run the algorithm k times and take the maximum cut we find, then the probability that we *don't* get m / 4 edges or more is at most $(2 / 3)^k$.
- The probability we *do* get at least m / 4 edges is at least $1 - (2 / 3)^k$.
- If we set $k = \log_{3/2} m$, the probability we get at least m / 4 edges is **$1 - 1 / m$**.
- There is a randomized, **$O((m + n) \log m)$**-time algorithm that finds a **(0.25)-approximation** to max-cut with probability **$1 - 1 / m$**.

Why This Works

- Given a randomized algorithm that has a probability p of success, we can amplify that probability significantly by repeating the algorithm multiple times.
- This technique is used extensively in randomized algorithms; we'll see another example of this on Friday.

Next Time

- Karger's Algorithm
- Finding a Global Min-Cut
- Applications of Global Min-Cut