Slides courtesy of

- Leiserson, Indyk, Kellis, myself
Topics

• Randomized algorithms:
  – algorithms that flip coins as a basic operation
• Three examples:
  – Matrix product checker: is $AB=C$?
  – Pattern Matching
  – Quicksort:
    • Example of divide and conquer
    • Fast and practical sorting algorithm
Randomized (also called: probabilistic) Algorithms

• Algorithms that can flip coins as a basic step
  – Namely: Algorithm can toss fair coin $c$ which is either Heads or Tails with probability $\frac{1}{2}$
  $\Rightarrow$ Algorithm Can generate a random number $r$ from some range $\{1...R\}$

• Algorithms makes decisions based on $r$’s value

• On the same input, depending on the value of $r$, a randomized algorithm can behave differently
Randomized vs. Deterministic Algorithms

• On the same input, on different executions, randomized algorithms may
  – Run for a different number of steps
  – Produce different outputs

• Why? On different executions, different coins are flipped (different random numbers are used), and the value of these coins can changes the course of executions
Randomized Algorithms: Two Flavors

**Monte Carlo Algorithm:**
For every input,
- regardless of coins tossed, Algorithm always run in polynomial time
- prob(output is correct) > high

**Las Vegas Algorithm:**
For every input.
- regardless of coins tossed, algorithm is correct
- the algorithms runs in expected polynomial time.
  \(\Rightarrow\) for all but a small number of executions, the algorithm runs in polynomial time.

**Important:** The probabilities & expectations above, are over the random choices of the algorithm! (not over the input)
Why does it make sense to toss coins?

- Some problems can not be solved, even in principle, if the computer is deterministic and cannot toss coins: the problem of asynchronous agreement in distributed computing is such an example (see distributed lecture).
- For some problems, we only know exponential deterministic algorithms, whereas we can find polynomial randomized algorithms.
- For some problems, can get a significant polynomial time speedup by going from a deterministic to a randomized algorithm. Say from $\Theta(n^8)$ to $\Theta(n^3)$.

**Intuition?** Think of algorithm design as battling an adversary who is preparing an input which will slow it down. By using randomness, after the input is chosen, the adversary can not know what the algorithm will do in runtime and therefore cannot choose an input which illicit a bad running time.
Where do we get coins from

• An assumption – for this class.
• In practice:
  – Pseudo Random Number Generators
• Fundamentally, Nature?

”.. I, at any rate, am convinced that He does not throw dice.”
Letter to Max Born (4 December 1926);
Monte Carlo Algorithms (MC)

Let $n$ be the length of the input.

For every input:

- MC runs in worst case polynomial Time
- MC is correct with high probability
  - Constant probability $> \frac{1}{2}$
  - Polynomially high $> 1 - \frac{1}{n^c}$ for $c > 0$
  - Exponentially high $> 1 - \frac{1}{2^{cn}}$ for $c > 0$

- Can transform a constant probability to exponentially high but there will still be a non-0 probability of being incorrect.
6.042 Math: Review at home

• Finite Sample Spaces
• Events
• Random Variables
• Independence
• Expectation
• Linearity of expectation
Matrix Product (last time)

• Compute $\mathbf{C} = \mathbf{A} \times \mathbf{B}$
  – Simple algorithm: $\mathcal{O}(n^3)$ time
  – Multiply two $2 \times 2$ matrices using 7 mult.
    $\rightarrow \mathcal{O}(n^{2.81\ldots})$ time [Strassen’69]
  – Multiply two $70 \times 70$ matrices using $143640$ multiplications $\rightarrow \mathcal{O}(n^{2.795\ldots})$ time [Pan’78]
  – …
  – $\mathcal{O}(n^{2.376\ldots})$ [Coppersmith-Winograd]
Matrix Product Checker

- **Given:** \(n \times n\) matrices \(A, B, C\) (entries from field)
- **Goal:** check if \(A \times B = C\) or not?

- Deterministically in \(O(n^{2.376})\), simply re-multiply the matrices [Note: counting multiplication as \(O(1)\)]
- **Q:** can we do it better than multiply?

- We will see an \(O(n^2)\) algorithm that:
  - If \(AB = C\), then \(\text{Prob}[\text{output}=\text{YES}]=1\)
  - If \(AB \neq C\), then \(\text{Prob}[\text{output}=\text{YES}] \leq \frac{1}{2}\)
The algorithm [Freivald]

- Choose a random binary vector $r[1...n]$, such that $\Pr[r_i=1]=\frac{1}{2}$, independently for $i=1...n$.
- If $A(Br)=Cr$, then output ‘YES’, else output ‘NO’.

**Easy observations:**

- $O(n^2)$ time, since takes 3 matrix-vector multiplications: $Br$, $A(Br)$, $Cr$, and each product is $O(n^2)$ multiplications.
- If $AB=C$, then $A(Br)=(AB)r=Cr$ and the algorithm always outputs YES. Namely, $\text{prob(error)}=0$ in this case.
Analyzing Correctness if AB≠C

Claim: if AB≠C, then Prob[ABr≠Cr] ≥ 1/2

Proof: Let D=AB-C. Our hypothesis is thus that D ≠ 0. Clearly, there exists r such that Dr≠0, We need to show there are many r such that Dr≠0: Specifically:

the Prob[Dr ≠ 0] ≥ 1/2 for a randomly chosen r.

Idea: will show that ∀ r, either Dr≠0 or there exists an r’ such that Dr’≠0. And the mapping from r to r’ is 1-to-1 (next slide)

This will clearly establish our claim
Suppose $AB-C=D\neq 0$, and $d_{ij} \neq 0$

(continued)

Fix vector $v$ which is 0 in all its coordinates except for $v_j = 1$. Then, $(Dv)_j = d_{ij} \neq 0$ which implies $Dv \neq 0$

Now take any $r$ that can be chosen by our algorithm. If $Dr \neq 0$, we are done. Suppose $Dr = 0$. Consider $r'$ that is the same as $r$ in all coordinates except for $r'_j = r_j + v_j \mod 2$. This means that, $Dr' = D(r+v) = 0 + Dv \neq 0$

Moreover, $r$ to $r'$ is 1-to-1. Why? If $r' = r + v = r'' + v \mod 2$, then $r = r''$

Conclusion: number of $r'$ for which $Dr' \neq 0$ is $\geq$

number of $r$ for which $Dr = 0$ $\Rightarrow$ Prob[$Dr \neq 0] \geq 1/2$ QED
Two cups

- For every $r$ in cup1, there exists $r'$ in cup2 such that $r=r'$ in all but one coordinate
- And the transformation is 1-to-1

$\implies$ Cup 2 contains at least $\frac{1}{2}$ of all possible $r$'s.
Let's check understanding

• Where did we use the fact that the entries come from a field?
• What if we chose the entries of $r$ from a larger set of size $t$? What would be the error probability?
• How can we make the error probability go down from $\frac{1}{2}$ to $(\frac{1}{2})^{100}$?
  – Repeat for 100 different $r$’s and reject if $A_{Br} \neq C_r$ for any of the $r$’s, otherwise accept
Las Vegas Randomized Algorithms

- Can generate random \( r \in \{0, \ldots, R\} \) (coins)
- The running time on input \( x \), becomes a random variable \( \text{Time}(x, r) \) depending on randomness \( r \)

- **Expected Polynomial Time:**
  \[
  T(n) = E_r[\text{Time}(x, r)] \text{ for any } x \text{ of length } n \text{ is bounded by a polynomial function in } n
  
- **Always correct** (for all input, prob(error)=0)
Quicksort

• Divide and conquer algorithm but the work is in divide rather than in combine
• Different Versions:
  • Basic: good in average case (for a random input)
  • Randomized: good for all inputs in expectation
• Sorts “in place” (like insertion sort, but not like merge sort).
• Very practical (with tuning).
Idea: Divide and conquer

Quicksort an $n$-element array $A$:

**Divide:**

1. Pick a **pivot** element $x$ in $A$
2. Partition the array into sub-arrays
   
   $L$(elements $<x$), $E$ (elements $=x$), $G$ (elements $>x$)

**Conquer:** Recursively sort subarrays $L$ and $G$

**Combine:** Trivial.
How do you Choose a Pivot $x$

**Basic Quick Sort:**

$x = A[1]$

*Show:* worst case $O(n^2)$ time

$O(n\log n)$ time for average input

**Randomized Quick Sort:**

$x$ is chosen at random from the array $A$

(recursively each time a random choice)

*Show:* expected $O(n\log n)$ for all inputs

Randomness gives us more control over runtime.
How to Partition: Idea

- Partition an input sequence as follows:
  - Remove, in turn, each element $y$ from $A$ and
  - Insert $y$ into $L$, $E$ or $G$, depending on the result of the comparison with the pivot $x$
- Each insertion and removal takes $O(1)$ time
- Thus, the partition step takes $O(n)$
- To do this in place: tricky code (next 2 slides, skip in lecture)
Pseudocode for basic quicksort

\[ \text{QUICKSORT}(A, p, r) \]
\[ \text{if } p < r \]
\[ \quad \text{then } q \leftarrow \text{PARTITION}(A, p, r) \]
\[ \quad \text{QUICKSORT}(A, p, q-1) \]
\[ \quad \text{QUICKSORT}(A, q+1, r) \]

\textbf{Initial call:} \text{QUICKSORT}(A, 1, n)
Partitioning subroutine

\textsc{Partition}(A, p, r) \triangleright A[p \ldots r]

\begin{align*}
x & \leftarrow A[p] \quad \triangleright \text{pivot } = A[p] \\
i & \leftarrow p \\
\text{for } j & \leftarrow p + 1 \text{ to } r \\
\text{do if } & A[j] \leq x \\
\text{then } & i \leftarrow i + 1 \\
\text{exchange } & A[i] \leftrightarrow A[j] \\
\text{exchange } & A[p] \leftrightarrow A[i] \\
\text{return } & i
\end{align*}

\textbf{Invariant:}
\[\begin{array}{cccc}
\text{x} & \leq x & \geq x & ? \\
p & i & j & r
\end{array}\]
Analysis of quicksort

• Assume all input elements are distinct.
• In practice, there are better partitioning algorithms for when duplicate input elements may exist.
• What is the worst case running time of Basic Quicksort?
Worst-case Running Time

- The worst case for quick-sort occurs when the array is sorted or reverse sorted and the pivot is the unique minimum or maximum element.
- One of $L$ and $G$ has size $n - 1$ and the other has size 0.
- The running time is proportional to the sum:
  \[ n + (n - 1) + \ldots + 2 + 1 \]

Thus, the worst-case running time of quick-sort is $O(n^2)$.
Randomized quicksort

- Partition around a *random* element. I.e., around $A[t]$, where $t$ chosen uniformly at random from {p…r}

- We will show that the *expected* time is $O(n \log n)$ for all input arrays $A$
Analysis method #1: Indicator random variables

Let $T(n)$ = the random variable for the running time of randomized quicksort on an input of size $n$, assuming random numbers are independent.

For $k = 0, 1, \ldots, n-1$, define the indicator random variable

\[ X_k = \begin{cases} 
1 & \text{if PARTITION generates a } k : n-k-1 \text{ split,} \\
0 & \text{otherwise.} 
\end{cases} \]

\[
\begin{array}{cccccccc}
X_0 & X_1 & X_2 & X_3 & X_{k-1} & X_k & X_{k+1} & X_{n-3} & X_{n-2} & X_{n-1} \\
0 & 0 & 0 & 0 & \ldots & 0 & 1 & 0 & 0 & 0 & \ldots & 0 & 0 & 0 & 0
\end{array}
\]

$\leftarrow k \quad \rightarrow n-k-1$
Great expectations: $E[X_k], E[T(n)]$.

- Can use random variables to calculate expectations
- Expected value of indicator random variable:
  \[
  E[X_k] = 1 \cdot \Pr\{X_k=1\} + 0 \cdot \Pr\{X_k=0\}
  = 1 \cdot \left(\frac{1}{n}\right) + 0 \cdot \left(\frac{n-1}{n}\right)
  = \frac{1}{n}
  \]
- Since all splits are equally likely, assuming elements are distinct.
- Can use $E[X_k]$ to calculate $E[T(n)]$
The power and simplicity of indicator random variable

\[ T(n) = \begin{cases} 
  T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split}, \\
  T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split}, \\
  \vdots \\
  T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split}, 
\end{cases} \]

\[ = \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n)) \]

- Summarize all \( n \) cases in a single expression using \( X_k \).
- Sum selects the \( X_k \) where the split happens (\( X_k = 1 \))
Calculating expectation

\[ E[T(n)] = E \left[ \sum_{k=0}^{n-1} X_k (T(k) + T(n - k - 1) + \Theta(n)) \right] \]

Take expectations of both sides.
Calculating expectation

\[ E[T(n)] = E \left[ \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n)) \right] \]

\[ = \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + \Theta(n))] \]

Linearity of expectation.
Calculating expectation

\[ E[T(n)] = E \left[ \sum_{k=0}^{n-1} X_k (T(k) + T(n - k - 1) + \Theta(n)) \right] \]

\[ = \sum_{k=0}^{n-1} E[X_k (T(k) + T(n - k - 1) + \Theta(n))] \]

\[ = \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n - k - 1) + \Theta(n)] \]

Independence of \( X_k \) from other random choices in the recursive calls.
Calculating expectation

\[ E[T(n)] = E\left[ \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n)) \right] \]

\[ = \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + \Theta(n))] \]

\[ = \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)] \]

\[ = \frac{1}{n} \sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n} \sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n) \]

Linearity of expectation; \( E[X_k] = 1/n \).
Calculating expectation

$$E[T(n)] = E\left[ \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n)) \right]$$

$$= \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + \Theta(n))]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

$$= \frac{1}{n} \sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n} \sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n)$$

$$= \frac{2}{n} \sum_{k=1}^{n-1} E[T(k)] + \Theta(n)$$

Summations have identical terms.
Hairy recurrence

\[ E[T(n)] = \frac{2}{n} \sum_{k=2}^{n-1} E[T(k)] + \Theta(n) \]

(The \( k = 0, 1 \) terms can be absorbed in the \( \Theta(n) \).)

**Prove:** \( E[T(n)] \leq an \lg n \) for constant \( a > 0 \).

- Choose \( a \) large enough so that \( an \lg n \) dominates \( E[T(n)] \) for sufficiently small \( n \geq 2 \).

**Use fact:** \( \sum_{k=2}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \) (exercise).
Hairy recurrence

\[ E[T(n)] = 2 \frac{n-1}{n} \sum_{k=2}^{n-1} E[T(k)] + \Theta(n) \]

(The \( k = 0, 1 \) terms can be absorbed in the \( \Theta(n) \).)

**Prove:** \( E[T(n)] \leq an \lg n \) for constant \( a > 0 \).
Substitution method

\[ E[T(n)] \leq \frac{2}{n} \sum_{k=2}^{n-1} ak \lg k + \Theta(n) \]

Substitute inductive hypothesis.
Substitution method

\[ E[T(n)] \leq \frac{2}{n} \sum_{k=2}^{n-1} a k \log k + \Theta(n) \]

\[ \leq \frac{2a}{n} \left( \frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + \Theta(n) \]

Use fact.
Substitution method

\[
E[T(n)] \leq \frac{2}{n} \sum_{k=2}^{n-1} ak \log k + \Theta(n)
\]

\[
\leq \frac{2a}{n} \left( \frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + \Theta(n)
\]

\[
eq an \log n - \left( \frac{an}{4} - \Theta(n) \right)
\]

Express as \textit{desired} – \textit{residual}. 
Substitution method

\[ E[T(n)] \leq 2 \sum_{k=2}^{n-1} \frac{ak \log k}{n} + \Theta(n) \]

\[ = \frac{2a}{n} \left( \frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + \Theta(n) \]

\[ = an \log n - \left( \frac{an}{4} - \Theta(n) \right) \]

\[ \leq an \log n, \]

if \( a \) is chosen large enough so that \( an/4 \) dominates the \( \Theta(n) \).
Analysis method #1 summary

- Defined indicator random variable $X_k$, marking the partition point for $k:n-k-1$ split.
- Expressed running time $T(n)$ (rand. var.) as a function of this indicator random variable.
- Calculated the expected value of $E[T(n)]$ using properties of $E[X_k]$.
- Prove $E[T(n)] \leq an \log n$ by induction using the substitution method.

⇒ Quicksort expected running time $O(n \log n)$. 
“Paranoid” quicksort: Analysis method

• Let’s modify the algorithm to make it easier to analyze by another method:
  
  • Repeat:
    • Choose the pivot to be a random element of the array
    • Perform \textsc{Partition}
    • Until the resulting partition will cause a “good” recursive call: the sizes of both \(L\) and \(G\) are no larger than \(\frac{3}{4}(\text{size of the array})\)
    • Recurse on both sub-arrays
Expected Running Time Analysis for paranoid quick sort

- Consider a recursive call of quick-sort on a sequence of size $s$.

- **Define**
  - **Good call:** the sizes of $L$ and $G$ are each less than $3s/4$
  - **Bad call:** one of $L$ and $G$ has size greater than $3s/4$

  ![Diagram showing good and bad calls](Diagram.png)

- **Claim:** A call is good with probability $\frac{1}{2}$
- **Proof:** $1/2$ of the possible pivots cause good calls:

Analysis

• Let $T(n)$ be an upper bound on the expected running time on any array of $n$ elements

• Consider any input of size $n$

• The time needed to sort the input is bounded from the above by a sum of
  • The time needed to sort the left subarray
  • The time needed to sort the right subarray
  • The number of iterations till we get a good call * $cn$ (Cost of partition)
Expectations

Therefore:

\[ T(n) \leq \max T(i) + T(n - i) + E[\# \text{ iterations}] \cdot cn \]

where maximum is taken over \( i \in [n/4, 3n/4] \)

- We showed \( \Pr[\text{good call}] > 1/2 \Rightarrow E[\#\text{iterations}] \) is \( \leq 2 \)
- So:

\[ T(n) \leq \max T(i) + T(n - i) + 2cn \quad , i \in [n/4 \quad ,3n/4] \]
Final bound

• Use a recursion tree argument:
  • Tree depth is $\Theta(\log n)$
  • Total cost at each level is at most $2cn$
  • Overall $T(n) = O(n \log n)$