Today

• Randomized algorithms: algorithms that flip coins
  – Matrix product checker: is $AB = C$?
  – Quicksort:
    • Example of divide and conquer
    • Fast and practical sorting algorithm
    • Other applications on Wednesday
Randomized Algorithms

• Algorithms that make random decisions
• That is:
  – Can generate a random number \( x \) from some range \( \{1\ldots R\} \)
  – Make decisions based on the value of \( x \)
• Why would it make sense?
Two cups, one coin

- If you always choose a fixed cup, the adversary will put the coin in the other one, so the expected payoff = $0
- If you choose a random cup, the expected payoff = $0.5
Randomized Algorithms

- Two basic types:
  - Typically fast (but sometimes slow): Las Vegas
  - Typically correct (but sometimes output garbage): Monte Carlo

- The probabilities are defined by the random numbers of the algorithm! (not by random choices of the problem instance)
Matrix Product

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

• Given: $n \times n$ matrices $A, B, C$
• Goal: is $A \times B = C$?
• We will see an $O(n^2)$ algorithm that:
  – If answer=$YES$, then $\Pr[\text{output}=\text{YES}]=1$
  – If answer=$NO$, then $\Pr[\text{output}=\text{YES}] \leq \frac{1}{2}$
The algorithm

• Algorithm:
  – Choose a random binary vector \( x[1…n] \), such that \( \Pr[x_i=1]=\frac{1}{2} \), \( i=1…n \)
  – Check if \( ABx=Cx \)

• Does it run in \( O(n^2) \) time?
  – YES, because \( ABx = A(Bx) \)
Correctness

• Let $D = AB$, need to check if $D = C$

• What if $D = C$ ?
  – Then $Dx = Cx$, so the output is YES

• What if $D \neq C$ ?
  – Presumably there exists $x$ such that $Dx \neq Cx$
  – We need to show there are many such $x$
D ≠ C
Vector product

- Consider vectors \( d \neq c \) (say, \( d_i \neq c_i \))
- Choose a random binary \( x \)
- We have \( dx = cx \) iff \( (d-c)x = 0 \)
- \( \Pr[(d-c)x = 0] = ? \)

\[
\begin{aligned}
\text{(d-c):} & & d_1-c_1 & & d_2-c_2 & & \ldots & & d_i-c_i & & \ldots & & d_n-c_n \\
\text{x:} & & x_1 & & x_2 & & \ldots & & x_i & & \ldots & & x_n
\end{aligned}
\]

\[
= \sum_{j \neq i} (d_j-c_j)x_j + (d_i-c_i)x_i
\]
Analysis, ctd.

- If \( x_i = 0 \), then \( (c-d)x = S_1 \)
- If \( x_i = 1 \), then \( (c-d)x = S_2 \neq S_1 \)
- So, \( \geq 1 \) of the choices gives \( (c-d)x \neq 0 \)

\[ \rightarrow \Pr[cx=dx] \leq \frac{1}{2} \]
Matrix Product Checker

• Is $A \times B = C$?
• We have an algorithm that:
  – If answer=YES, then $\Pr[\text{output}=\text{YES}]=1$
  – If answer=NO, then $\Pr[\text{output}=\text{YES}] \leq \frac{1}{2}$
• What if we want to reduce $\frac{1}{2}$ to $\frac{1}{4}$?
  – Run the algorithm twice, using independent random numbers
  – Output YES only if both runs say YES
• Analysis:
  – If answer=YES, then $\Pr[\text{output}_1=\text{YES}, \text{output}_2=\text{YES}]=1$
  – If answer=NO, then
    $\Pr[\text{output}=\text{YES}] = \Pr[\text{output}_1=\text{YES}, \text{output}_2=\text{YES}]
     = \Pr[\text{output}_1=\text{YES}] \times \Pr[\text{output}_2=\text{YES}]
     \leq \frac{1}{4}$
Quicksort

- Divide-and-conquer algorithm.
- Sorts “in place” (like insertion sort, but not like merge sort).
- Very practical (with tuning).
- Can be viewed as a randomized Las Vegas algorithm
Divide and conquer

Quicksort an $n$-element array:

1. **Divide:** Partition the array into two subarrays around a pivot $x$ such that elements in lower subarray $\leq x \leq$ elements in upper subarray.

   $\leq x \quad x \quad \geq x$

2. **Conquer:** Recursively sort the two subarrays.

3. **Combine:** Trivial.

   **Key:** Linear-time partitioning subroutine.
Pseudocode for quicksort

\[\text{QUICKSORT}(A, p, r)\]

\[
\begin{align*}
\text{if } & \ p < r \\
\text{then } & \ q \leftarrow \text{PARTITION}(A, p, r) \\
\text{QUICKSORT}(A, p, q-1) \\
\text{QUICKSORT}(A, q+1, r)
\end{align*}
\]

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

\[
\text{PARTITION}(A, p, r) \leadsto A[p \ldots r]
\]

\[
x \leftarrow A[p]\quad \leadsto \text{pivot} = A[p]
\]

\[
i \leftarrow p
\]

\[
\text{for } j \leftarrow p + 1 \text{ to } r \\
\text{do if } A[j] \leq x \\
\text{then } i \leftarrow i + 1
\]

\[
\text{exchange } A[i] \leftrightarrow A[j]
\]

\[
\text{exchange } A[p] \leftrightarrow A[i]
\]

\[
\text{return } i
\]

**Invariant:**

\[
\begin{array}{ccccc}
x & \leq x & \geq x & ? \\
p & i & j & r
\end{array}
\]
Example of partitioning

\[ \begin{align*}
6 & \quad 10 & \quad 13 & \quad 5 & \quad 8 & \quad 3 & \quad 2 & \quad 11 \\
\end{align*} \]

\( i \quad j \)
Example of partitioning

\[ \begin{array}{cccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
\end{array} \]

\[ i \rightarrow j \]
Example of partitioning

\[
\begin{array}{cccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
\end{array}
\]

\[i \quad \rightarrow \quad j\]
Example of partitioning

\[ \begin{array}{cccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
6 & 5 & 13 & 10 & 8 & 3 & 2 & 11 \\
\end{array} \]

\[ i \quad j \]
Example of partitioning

\[
\begin{array}{cccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
6 & 5 & 13 & 10 & 8 & 3 & 2 & 11
\end{array}
\]
Example of partitioning

\begin{figure}
\begin{center}
\begin{tabular}{cccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
6 & 5 & 13 & 10 & 8 & 3 & 2 & 11 \\
\end{tabular}
\end{center}
\end{figure}
Example of partitioning
Example of partitioning
Example of partitioning
Example of partitioning

\[
\begin{array}{cccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
6 & 5 & 13 & 10 & 8 & 3 & 2 & 11 \\
6 & 5 & 3 & 10 & 8 & 13 & 2 & 11 \\
6 & 5 & 3 & 2 & 8 & 13 & 10 & 11 \\
\end{array}
\]

\[i \quad \quad \quad j\]
Example of partitioning

\[
\begin{array}{cccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
6 & 5 & 13 & 10 & 8 & 3 & 2 & 11 \\
6 & 5 & 3 & 10 & 8 & 13 & 2 & 11 \\
6 & 5 & 3 & 2 & 8 & 13 & 10 & 11
\end{array}
\]
Example of partitioning
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 Quicksort?
Worst-case of quicksort

• Input sorted or reverse sorted.
• Partition around min or max element.
• One side of partition always has no elements.

\[ T(n) = T(0) + T(n - 1) + \Theta(n) \]
\[ = \Theta(1) + T(n - 1) + \Theta(n) \]
\[ = T(n - 1) + \Theta(n) \]
\[ = \Theta(n^2) \text{ (arithmetic series)} \]
Worst-case recursion tree

\[ T(n) = T(0) + T(n-1) + cn \]
Worst-case recursion tree

\[ T(n) = T(0) + T(n-1) + cn \]
Worst-case recursion tree

\[ T(n) = T(0) + T(n-1) + cn \]
Worst-case recursion tree

\[ T(n) = T(0) + T(n-1) + cn \]
Worst-case recursion tree

\[ T(n) = T(0) + T(n-1) + cn \]
Worst-case recursion tree

\[ T(n) = T(0) + T(n-1) + cn \]

\[ \Theta \left( \sum_{k=1}^{n} k \right) = \Theta(n^2) \]
Worst-case recursion tree

\[ T(n) = T(0) + T(n-1) + cn \]

\[ \Theta(h) = \Theta(n) \]

\[ \Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2) \]
Nice-case analysis

If we’re lucky, PARTITION splits the array evenly:

\[ T(n) = 2T(n/2) + \Theta(n) \]
\[ = \Theta(n \lg n) \quad \text{(same as merge sort)} \]

What if the split is always \( \frac{1}{10} : \frac{9}{10} \) ?

\[ T(n) = T\left(\frac{1}{10}n\right) + T\left(\frac{9}{10}n\right) + \Theta(n) \]
Analysis of nice case

$T(n)$
Analysis of nice case

\[ T\left(\frac{1}{10} n\right) \quad cn \quad T\left(\frac{9}{10} n\right) \]
Analysis of nice case

\[ cn \]

\[ \frac{1}{10} cn \quad T\left(\frac{1}{100} n\right) T\left(\frac{9}{100} n\right) \]

\[ \frac{9}{10} cn \quad T\left(\frac{9}{100} n\right) T\left(\frac{81}{100} n\right) \]
Analysis of nice case

\[ cn \]

\[ \frac{1}{10} \]  
\[ \frac{9}{100} \]  
\[ \Theta(1) \]

\[ \frac{1}{100} \]  
\[ \frac{9}{100} \]  
\[ \Theta(1) \]

\[ \log_{10/9} n \]  
\[ \frac{81}{100} \]  
\[ \Theta(1) \]
Analysis of nice case

\[ \Theta(1) \]

\[ cn \log_{10} n \leq T(n) \leq cn \log_{10/9} n + O(n) \]
Randomized quicksort

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

• We will show that the *expected* time is $O(n \log n)$
“Paranoid” quicksort

- Will modify the algorithm to make it easier to analyze:
  - Repeat:
    - Choose the pivot to be a random element of the array
    - Perform PARTITION
  - Until the resulting split is “lucky”, i.e., not worse than $1/10 : 9/10$
  - Recurse on both sub-arrays
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 until we get a lucky split, times $cn$. 
Expectations

# partitions = # iterations until lucky split of at most 1/10:9/10

• By linearity of expectation:

\[ T(n) \leq \max T(i) + T(n - i) + E[\# partitions] \cdot cn \]

where maximum is taken over \( i \in [n/10, 9n/10] \)

• We will show that \( E[\# partitions] \) is \( \leq 10/8 \)

• Therefore:

\[ T(n) \leq \max T(i) + T(n - i) + 2cn, i \in [n/10, 9n/10] \]
Final bound

• Can use the recursion tree argument:
  • Tree depth is $\Theta(\log n)$
  • Total expected work at each level is at most $\frac{10}{8} cn$
  • The total expected time is $O(n \log n)$
Lucky partitions

• The probability that a random pivot induces lucky partition is at least \( \frac{8}{10} \)
  (we are not lucky if the pivot happens to be among the smallest/largest \( \frac{n}{10} \) elements)

• If we flip a coin, with heads prob. \( p = \frac{8}{10} \), the expected waiting time for the first head is \( \frac{1}{p} = \frac{10}{8} \)
Quicksort in practice

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort can benefit substantially from code tuning.
- Quicksort behaves well even with caching and virtual memory.
- Quicksort is great!
More intuition

Suppose we alternate lucky, unlucky, lucky, unlucky, lucky, ….

\[ L(n) = 2U(n/2) + \Theta(n) \quad \text{lucky} \]
\[ U(n) = L(n - 1) + \Theta(n) \quad \text{unlucky} \]

Solving:

\[
L(n) = 2(L(n/2 - 1) + \Theta(n/2)) + \Theta(n) \\
= 2L(n/2 - 1) + \Theta(n) \\
= \Theta(n \log n) \quad \text{Lucky!}
\]

How can we make sure we are usually lucky?
Randomized quicksort analysis

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

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

$E[X_k] = \Pr\{X_k = 1\} = 1/n$, since all splits are equally likely, assuming elements are distinct.
Analysis (continued)

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

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

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

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.
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

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

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

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

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

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

Summations have identical terms.
Hairy recurrence

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

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

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

- Choose \( a \) large enough so that \( a n \lg n \) 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).
Substitution method

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

Substitute inductive hypothesis.
Substitution method

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

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

Use fact.
Substitution method

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

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

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

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

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

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

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

\[ \leq an \log n , \]

if \( a \) is chosen large enough so that \( an/4 \) dominates the \( \Theta(n) \).
• Assume

Running time = $O(n)$ for $n$ elements.
Randomized Algorithms

• Algorithms that make decisions based on random coin flips.
• Can “fool” the adversary.
• The running time (or even correctness) is a random variable; we measure the expected running time.
• We assume all random choices are independent.
• This is not the average case!