Today

- Randomized algorithms: algorithms that flip coins
- Two examples:
  - 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 decisions based on random coin flips.
  - Can generate a random number $x$ from some range $\{1 \ldots R\}$
  - Make decisions based on the value of $x$
- Can “fool” the adversary.
- Difference between average case and expected case
  - Probabilities defined by random numbers of algorithm
  - Not random variation of the problem instance

Matrix Product Checker

Monte Carlo randomized algorithm
(always fast, sometimes wrong)

Randomized Algorithms

- Two basic types:
  - Always right (but sometimes slow): Las Vegas
  - Always fast (but sometimes incorrect): Monte Carlo
- The running time (or even correctness) is a random variable; we measure the expected running time.
  - Today, we’ll see an example of each type

Two cups, one coin

- Deterministic algorithm:
  - If you always choose a fixed cup, the adversary will put the coin in the other one, so the expected payoff $= 0$
- Randomized algorithm:
  - If you choose a random cup, the expected payoff $= 0.5$
Matrix Product (recall)

- Compute $C = A \times B$
  - Simple algorithm: $O(n^3)$ time
  - Multiply two $2 \times 2$ matrices using 7 mult.
    $\rightarrow O(n^{2.31\ldots})$ time [Strassen’69]
  - Multiply two $70 \times 70$ matrices using 143640 multiplications $\rightarrow O(n^{2.795\ldots})$ time [Pan’78]
  - ...
  - $O(n^{2.376\ldots})$ [Coppersmith-Winograd’90]
  - Theoretical limit is $O(n^2)$, but not reached yet

Matrix Product Checker

- Given: $n \times n$ matrices $A, B, C$
- Goal: is $A \times B = C$?

Only needs to check if the product is right. Can we do it faster than actually repeating the product?

- 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 $x[1\ldots n]$, such that $\Pr[x_i = 1] = \frac{1}{2}$, $i=1\ldots n$
  - 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 some $x$ s.t. $Dx \neq Cx$
  - We need to show there are many such $x$’s i.e. with $\Pr[\text{output} = \text{YES}]$ results in $Dx \neq Cx$.

Vector product

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

$$(d-c) = \begin{bmatrix} d_1-c_1 & d_2-c_2 & \ldots & d_k-c_k \end{bmatrix}
\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \end{bmatrix}
= \sum_{i \neq j} (d_i-c_j)x_j + (d_i-c_i)x_i.$$
What have we shown?

- If $C=AB$, the output is YES
- If $C\neq AB$, the output is NO with $P \geq \frac{1}{2}$.

- We can run matrix product checker repeatedly to bring this probability down:
  - $k$ iteration, probability of error: $P < 1/(2^k)$

Matrix Product Checker

- Is $A \times B = C$?
- We have an algorithm that:
  - If answer= YES, then $Pr[output= YES] = 1$
  - If answer= NO, then $Pr[output= YES] \leq \frac{1}{2}$
- 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 $Pr[output_1= YES, output_2= YES ] = 1$
  - If answer= NO, then
    $Pr[output= YES] = Pr[output_1= YES, output_2= YES]$
    $= Pr[output_1= YES]*Pr[output_2= YES]$
    $\leq \frac{1}{4}$

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.

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

3. **Combine:** Trivial.

**Key:** Linear-time partitioning subroutine.

---

Pseudocode for quicksort

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

---

Partitioning subroutine

\[
\text{PARTITION}(A, p, r) \Rightarrow A[p \ldots r] \\
\quad x \leftarrow A[p] \Rightarrow \text{pivot} = A[p] \\
\quad i \leftarrow p \\
\quad \text{for } j \leftarrow p + 1 \text{ to } r \\
\quad \quad \text{do if } A[j] \leq x \text{ then } i \leftarrow i + 1 \\
\quad \quad \quad \text{exchange } A[i] \leftrightarrow A[j] \\
\quad \quad \text{exchange } A[p] \leftrightarrow A[i] \\
\quad \text{return } i
\]

**Invariant:**

\[
\begin{array}{cccc}
\quad & \leq x & \geq x & ? \\
p & i & j & r
\end{array}
\]

---

Example of partitioning

---

Randomized quicksort

- Partition around a **random** element. I.e., around \(A[t]\), where \(t\) chosen uniformly at random from \(\{p \ldots r\}\)
  - First choose random element
  - Switch it with the first element
  - It becomes the pivot
  - Run algorithm as before
- We will show that the **expected** time is \(O(n \log n)\)

---

First some intuition

Balanced vs. imbalanced partitions for quicksort
Worst-case of quicksort

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

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

Worst-case recursion tree

\[ T(n) = T(0) + T(n-1) + cn \]
\[ \Theta(1) \quad c(n-1) \]
\[ \Theta(1) \quad c(n-2) \quad \Theta(1) \quad \ldots \]
\[ h = n \]

Nice-case analysis

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

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

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

\[ T(n) = T(\frac{1}{10} n) + T(\frac{9}{10} n) + \Theta(n) \]

Analysis of nice case

\[ cn \quad \log_{10} cn \quad \frac{9}{10} cn \quad \frac{81}{100} cn \quad \Theta(1) \quad \ldots \]

\[ cn \log_{10} n \leq T(n) \leq cn \log_{10} n + O(n) \]

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?
Analysis method #1: “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

- Therefore:

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

  where maximum is taken over $i \in [n/10,9n/10]$
- We will show that $E[\#\ partitions]$ is $\leq 10/8$
- Therefore:

  $$T(n) \leq \max T(i) + T(n-i) + 10/8 \cdot cn, i \in [n/10,9n/10]$$

Final bound

- Can use the recursion tree argument:
  - Tree depth is $\Theta(\log n)$
  - Total cost at each level is at most $10/8 \cdot cn$
  - Overall $T(n)=O(n \log n)$

Lucky partitions

- The probability that a random pivot induces lucky partition is at least $8/10$
  (we are not lucky if the pivot happens to be among the smallest/largest $n/10$ elements)
- If we flip a coin, with heads prob. $p=8/10$, the expected waiting time for the first head is $1/p = 10/8$

Quicksort analysis #2

Substitution method with Indicator random variables
Analysis method #2: Indicator random variables

Let $T(n)$ be 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

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

Using Random Variables to Calculate Expectations: $E[X_k], E[T(n)]$.

- Can use random variables to calculate expectations.
- Expected value of indicator random variable:

$$E[X_k] = 1 \cdot \Pr\{X_k=1\} + 0 \cdot \Pr\{X_k=0\} = 1 \cdot \frac{1}{n} + 0 \cdot \left(\frac{n-1}{n}\right) = \frac{1}{n}$$

- Since all splits are equally likely, assuming elements are distinct.
- Can use $E[X_k]$ to calculate $E[T(n)]$

Calculating expectation

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

Use linearity of expectation.

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

Expect indep of $X_k$ from other rand. choices in recurs calls.

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

Summations have identical terms.

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

Solving recurrence by substitution method

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

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

Inductive hypothesis: $E[T(n)] \leq an \log n$ for const. $a > 0$.
- Choose $a$ large enough so that $an \log n$ dominates $E[T(n)]$ for sufficiently small $n \geq 2$.

Substitution method

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

Use fact.

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

Express as desired - residual.

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

Choose $a$ large enough so $an/4$ dominates $\Theta(n)$.

$$\leq an \log n$$
**Analysis method #2 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 $E[X_k]$ to obtain recursion.
- Solved recursion with inductive hypothesis $E[T(n)] \leq an \log n$ using substitution method

➤ QuickSort expected running time $O(n \log n)$.

**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!