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
  - Friday (6.096): Application in computational biology
    - Discovering common motifs in DNA sequences

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. $\rightarrow O(n^{2.81...})$ time [Strassen'69]
  - Multiply two $70 \times 70$ matrices using $143640$ multiplications $\rightarrow O(n^{2.795...})$ time [Pan'78]
  - ...
  - $O(n^{2.376...})$ [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 $x$

Vector product

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

$$(d-c): d_1 - c_1 \quad d_2 - c_2 \quad \ldots \quad d_n - c_n$$

$x$: $x_1 \quad x_2 \quad \ldots \quad x_n \quad \ldots \quad x_n$

$$= \sum_{j=1}^{n} (d_j - c_j)x_j + (d_i - c_i)x_i$$

D=≠C

- If $x=0$, then $(c-d)x=S_1$
- If $x=1$, then $(c-d)x=S_2 \neq S_1$
- So, $\geq 1$ of the choices gives $(c-d)x \neq 0$
  $\rightarrow \Pr[\text{output}=\text{YES}] \leq \frac{1}{2}$

Analysis, ctd.

Randomized Algorithms, Quicksort
Matrix Product Checker

- Is $A \cdot B = C$?
- 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
    $\Pr[\text{output}=\text{YES}] = \Pr[\text{output}_1=\text{YES}, \text{output}_2=\text{YES}] = \Pr[\text{output}_1=\text{YES}] \cdot \Pr[\text{output}_2=\text{YES}] \leq \frac{1}{4}$

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.

2. Conquer: Recursively sort the two subarrays.

Key: Linear-time partitioning subroutine.

Pseudocode for quicksort

```
QUICKSORT(A, p, r)
if p < r then
    q ← PARTITION(A, p, r)
    QUICKSORT(A, p, q–1)
    QUICKSORT(A, q+1, r)
Initial call: QUICKSORT(A, 1, n)
```

Partitioning subroutine

```
PARTITION(A, p, r) → A[p .. r]
    x ← A[p]
    i ← p
    for j ← p + 1 to r do if A[j] ≤ x then
        i ← i + 1
    return i
```

Invariant:

```
\begin{array}{c}
\text{x} \\
p \\
i \\
j \\
r
\end{array}
```

Example of partitioning

```
\begin{array}{c}
6 \\
10 \\
13 \\
5 \\
8 \\
3 \\
2 \\
11
\end{array}
```

\[i \quad j\]
Randomized Algorithms, Quicksort
Example of partitioning

Example of partitioning

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?

\[ \geq x \]
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**

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

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

\[h = n\]

**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 \(\frac{1}{10} : \frac{9}{10}\)?

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

- By linearity of expectation:

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

where maximum is taken over \( i \in [n/10,9n/10] \).
- We will show that \( E[\#\text{partitions}] \) is \( \leq 10/8 \).
- Therefore:

\[
T(n) \leq \max T(i) + T(n-i) + 2cn, i \in [n/10,9n/10]
\]
6.046 Introduction to Algorithms, Lecture 4

Randomized Algorithms, Quicksort

Final bound

- Can use the recursion tree argument:
  - Tree depth is $\Theta(\log n)$
  - Total expected work at each level is at most $10/8 \cdot cn$
  - The total expected time is $O(n \log n)$

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 $1/p = 10/8$

More intuition

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

$L(n) = 2U(n/2) + \Theta(n)$ lucky

$U(n) = L(n - 1) + \Theta(n)$ unlucky

Solving:

$L(n) = 2(L(n/2 - 1) + \Theta(n/2)) + \Theta(n)$

$= 2L(n/2 - 1) + \Theta(n)$

$= \Theta(n \log n)$ Lucky!

How can we make sure we are usually lucky?

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!

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!