Randomized Algorithms I
6.046 Lecture 4

Prof. Ronitt Rubinfeld

Warning: see lecture notes for more details
(some slides borrowed from previous versions of
Demaine, Leiserson, Indyk, Kellis, Goldwasser)
Our focus this week…

- **Randomized algorithms:**
  - algorithms that can flip coins
Why randomness?

• Solve new problems?
• Faster? More practical?
• Simpler?
The model:
Randomized (also called: probabilistic) Algorithms

- Can flip coins (pick random bit) as a basic step
  - Toss fair coin:
    - Heads or Tails with probability $\frac{1}{2}$
  - Often need tosses of “dice”:
    - Can generate a random number $r$ from range $\{1\ldots R\}$

- Algorithms make decisions based on $r$’s value
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);
Randomized vs. Deterministic Algorithms

- On the same input, on different executions, randomized algorithms may
  - Run for a different number of steps
  - Produce different outputs
First example:

Quicksort

Quicksort

- Divide and conquer algorithm
  - work is in divide, not in combine
- Sorts “in place”
  - Doesn’t need extra space
  - like insertion sort, but not like merge sort
- Very practical (with tuning).
Quicksort Versions:

- **Basic:**
  good for most inputs (may be bad for others)

- **Randomized:**
  good for all inputs in expectation

- **Deterministic:**
  good for all inputs in worst case
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 $> x$)

![Partitioning Array]

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

**Combine:** Trivial.
Partitioning subroutine

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

\[
x \leftarrow A[p] \quad \triangleright \text{pivot} = A[p]
\]
\[
i \leftarrow p
\]
\[
\text{for } j \leftarrow p + 1 \text{ to } r
\]
\[
\text{do if } A[j] \preceq x
\]
\[
\text{then } i \leftarrow i + 1
\]
\[
\text{exchange } A[i]
\]
\[
\leftrightarrow A[j]
\]
\[
\text{exchange } A[p] \leftrightarrow A[i]
\]

**Invariant:**

\[
\begin{array}{cccc}
\text{x} & \mathbf{\preceq} & \text{x} & \text{3 x} & \text{?} \\
p & i & j & r
\end{array}
\]
Example (see blackboard/notes)
Worst-case Running Time

- When 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
- Running time proportional to sum
  \[ n + (n - 1) + \ldots + 2 + 1 \]

```plaintext
depth     time
0         n
1         n - 1
...       ...
n - 1     1
```
Why is quicksort good?
Good cases:

• Lucky case: partition array evenly

• Pretty lucky case: partition into 1/10:9/10 split

• Alternate lucky/unlucky cases
How can we find a good pivot element?

• True on “average”

• Pick a **RANDOM** pivot

• Pick the **median** element
Randomized quicksort

• Partition around a *random* element.
  • Use $A[t]$, where $t$ chosen uniformly at random from \{p…r\}
• Will show that *expected* time is $O(n \log n)$ for all input arrays $A$
How many comparisons?

See notes…
How many comparisons?

Expected number of comparisons is $O(n \log n)$
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**: 
  $T(n) = E_r[\text{Time}(x, r)]$ for any $x$ of length $n$ is bounded by a polynomial function in $n$

- **Always correct** (for all input, prob(error) = 0)
Randomized Algorithms: Two Flavors

**Monte Carlo Algorithm:**
For every input,

- regardless of coins tossed, Algorithm *always runs in polynomial time*
- prob(output is correct) > high

**Las Vegas Algorithm:**
For every input.

- the algorithms runs in expected polynomial time.
  \[\Rightarrow\] for all but a small number of executions, the algorithm runs in polynomial time).
- regardless of coins tossed, algorithm is **correct**

Important: The probabilities & expectations above, are over the random choices of the algorithm! (not over the input)
Trick question:

• Is quicksort Monte Carlo or Las Vegas?
Summary

• Basic deterministic algorithm is good for most inputs
• Randomized algorithm is usually good for all inputs
• Deterministic algorithm is always good for all inputs
  – Why use randomized?
Quicksort in practice:

- Very fast (typically 2x faster than mergesort)
- Uses memory well
- Simple to implement