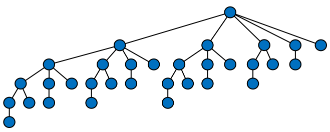
Lecture 11. Randomized Algorithms

CpSc 8400: Algorithms and Data Structures

Brian C. Dean



School of Computing
Clemson University
Spring, 2016

Randomized Algorithms

- Randomized algorithms offer many advantages:
  - Simplicity (e.g., randomly-balanced BSTs).
  - Malicious adversary often cannot force worst-case performance.
  - Efficiency (e.g., randomized quicksort vs. deterministic O(n log n) quicksort)
  - In some cases, randomization even gives us more power than we have with deterministic algorithms!
- … and some disadvantages:
  - May be more difficult to debug.
  - Analysis requires some background in probability.
  - We need a good source of randomness…

2

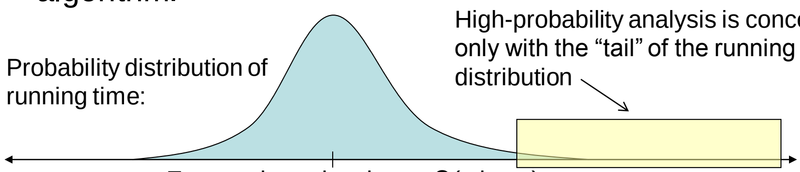
Randomized Algorithms

- Two types of randomized algorithms:
  - **Las Vegas**. Always correct but running time may vary depending random choices made by algorithm.
  - **Monte Carlo**. Deterministic running time but may make mistakes.
- We can convert any Las Vegas algorithm into a Monte Carlo algorithm by stopping it early if it fails to run fast enough.
  - So Las Vegas often viewed as the “better” of the 2 types.
- Can convert Monte Carlo to Las Vegas if we can check correctness of a solution: keep running until we get a correct solution.

3

Analyzing Randomized Algorithms

- With a Monte Carlo algorithm, we typically want to prove that the output is correct with high probability.
- With Las Vegas algorithms, we typically want to show that a certain running time bound holds with high probability.
  - Another popular analysis goal (which we’ll talk about soon), is to compute the “expected” running time of the algorithm.



4

Average-Case Analysis Versus Randomized Algorithms

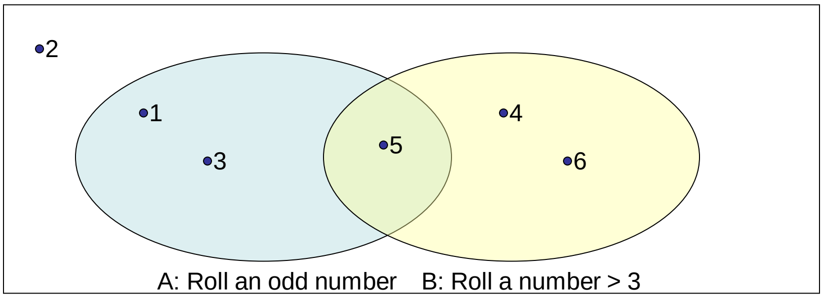
- Recall that we usually focus on **worst-case** performance of algorithms.
- Sometimes (especially when worst-case inputs due not often occur in practice), it makes sense also to consider **average-case** performance – the expected running time when given a random input (according to some probability distribution that mimics what you expect in practice).
- However, note that this is **completely different** from analyzing the (worst-case) expected running time of a randomized algorithm.
  - Randomness in input vs. randomness in algorithm.

Basic Probability Theory

- A random trial is described by a set of **outcomes**, each with an associated **probability**.
- Probabilities of all outcomes sum to 1.
- An **event** is a set of outcomes, and the probability of an event is the sum of the probabilities of these outcomes.
- Example: Let E be the event that we roll two dice and they sum to 4.
  - This includes the outcomes (1, 3), (2, 2), and (3, 1) each with probability 1/36, so Pr[E] = 3/36 = 1/12.
- Pr[~E] = 1 – Pr[E].

Intersections and Unions of Events

- If A and B are events, then
  - A ∩ B is the event that both occur
  - A U B is the event that either A or B occur, or both.
- Pr[A U B] = Pr[A] + Pr[B] – Pr[A ∩ B], as we can see from **Venn diagram** below (for rolling a 6-sided die).



Back to Randomized Quicksort…

- If we select pivot elements that result in "balanced partitions" (e.g., the median), quicksort runs in Θ(n log n) time.
- If we select pivot elements poorly, quicksort could run in $\Theta(n^2)$ time.
- **Claim**: Randomized quicksort (where we select pivot elements at random) runs in O(n log n) time with high probability.
  - But how do we prove this…?
  - And what does "with high probability" mean?

“With High Probability”

- A Monte Carlo randomized algorithm produces correct output **with high probability** if

  Pr[failure] ≤ $1 / n^c$

  where c can be chosen to be any constant we wish, by adjusting the hidden constant in the O(…) running time of the algorithm.
- A Las Vegas randomized algorithm runs in O(X) time **with high probability** if

  Pr[fails to run in O(X) time] ≤ $1 / n^c$

  where c can be chosen to be any constant we wish, by adjusting the hidden constant in the O(X) running time.

9

Intersection of Events and Independence

- **Pr**[A ∩ B] = **Pr**[A] **Pr**[B] if and only if events A and B are **independent**.
- Two events are independent if knowledge of the occurrence of one event has no impact on the probability of the occurrences of the other event.
- Example: Boosting success probability via repeated trials.
  - Take a Monte Carlo algorithm that fails with probability ≤ ½ (and suppose we can detect when it fails).
  - If we run it k times, the probability of failure drops to ≤ $1/2^k$.
  - If we run it k=c log n times, the probability of failure drops to ≤ $1/n^c$ (i.e., the algorithm succeeds with high probability)

10

Reducing Problem Size by a Constant Fraction per Iteration

- Suppose we have an algorithm for which:
  - We start with a problem of size n.
  - In every iteration, the effective size of the problem is reduced to a constant fraction of its original size.
- Then, it's easy to see that our algorithm must perform only O(log n) iterations.
- Prototypical example: binary search.
  - Each iteration reduces problem to ½ of its original size.

11

The Randomized Reduction Lemma

- Suppose we have an algorithm for which:
  - We start with a problem of size n.
  - In every iteration, the effective size of the problem is reduced to a constant fraction of its original size **with some constant probability.**
- Then, our algorithm performs only O(log n) iterations **with high probability**.
- We call this the **randomized reduction lemma**, and it will be one of the main tools we use for analyzing randomized algorithms and data structures.

12

Example: Randomized Binary Search

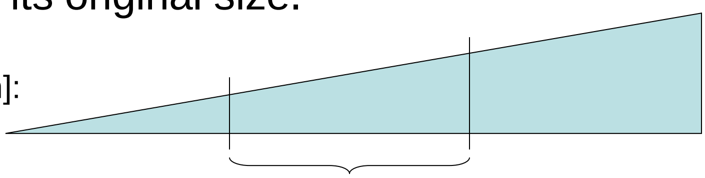
- Suppose we're searching a sorted array A[1…n] for an element of value X.
- In a standard binary search, we compare A[n/2] and X, then recursively search either A[1…n/2] or A[n/2…n].
- Suppose instead of A[n/2], we compare X against a random element and then recursively search left or right as before…

13

Example: Randomized Binary Search

- **Claim**: Randomized binary search runs in O(log n) time with high probability.
- **Simple proof**: If we happen to choose a "pivot" element A[j] where n/3 ≤ j ≤ 2n/3 (and this happens with probability ≥ 1/3), then our problem will be reduced to ≤ 2/3 of its original size.

A[1..n]:



14

The Union Bound

- Recall that **Pr**[A U B] = **Pr**[A] + **Pr**[B] – **Pr**[A ∩ B].
- It typically suffices just to use the rough upper bound **Pr**[A U B] ≤ **Pr**[A] + **Pr**[B].
- For multiple events $E_1 \ldots E_k$, this gives us what is known as the **union bound** or **Boole's inequality**:

  Pr[E_1 U E_2 U … U E_k] ≤ **Pr**[E_1] + … + **Pr**[E_k].
- For example:
  - Suppose each of 50 parts in a complex machine fails with probability ≤ 1/100
  - Then **Pr**[entire machine fails] ≤ 50(1/100) = ½.

15

The Union Bound

- If there are n bad events that can happen, and each happens with probability ≤ p, then the probability **any** bad event happens is at most np.

  I.e, for events $E_1 \ldots E_n$, **Pr**[U_i E_i] ≤ Σ_i **Pr**[E_i]
- This meshes particularly well with our definition of "with high probability": If some property holds for a generic input element w.h.p., then it also holds for each of our n input elements w.h.p.
- **Example:** If randomized quicksort spends only O(log n) on a generic input element w.h.p, then its total running time is O(n log n) w.h.p.

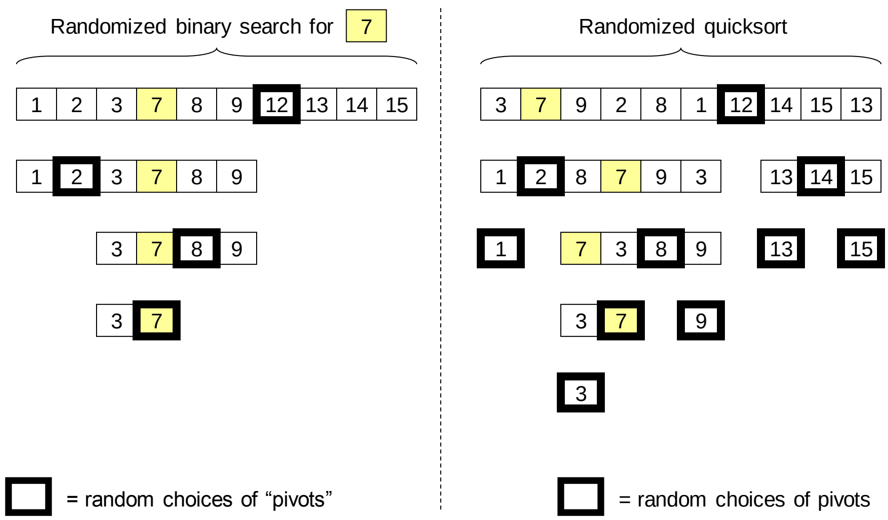
16

A Prototypical "High Probability" Analysis…

- **Step 1:** Consider some generic input element e. Show that randomized quicksort spends O(log n) work on e w.h.p.
  - Usually easy with the randomized reduction lemma.
- **Step 2:** The union bound automatically allows us to extend this result to show that randomized quicksort spends O(log n) work on **each** of its input elements w.h.p.
- So total running time is O(n log n) w.h.p.

17

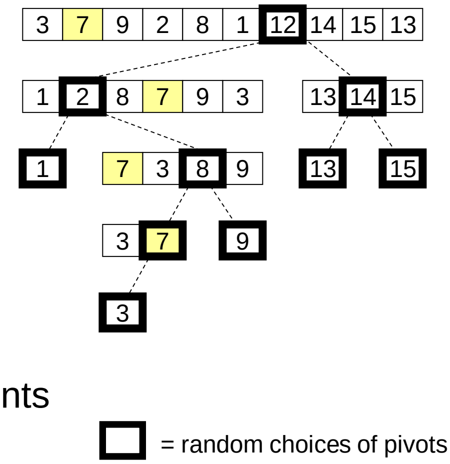
Randomized Reduction and Randomized Quicksort



18

Quicksort and BST Construction

- There is a direct analog between quicksort and the process of building a BST!
- Hence, both have the same running times.
- Randomized quicksort is analogous to building a BST by inserting elements in random order.



19

Randomly-Built BSTs

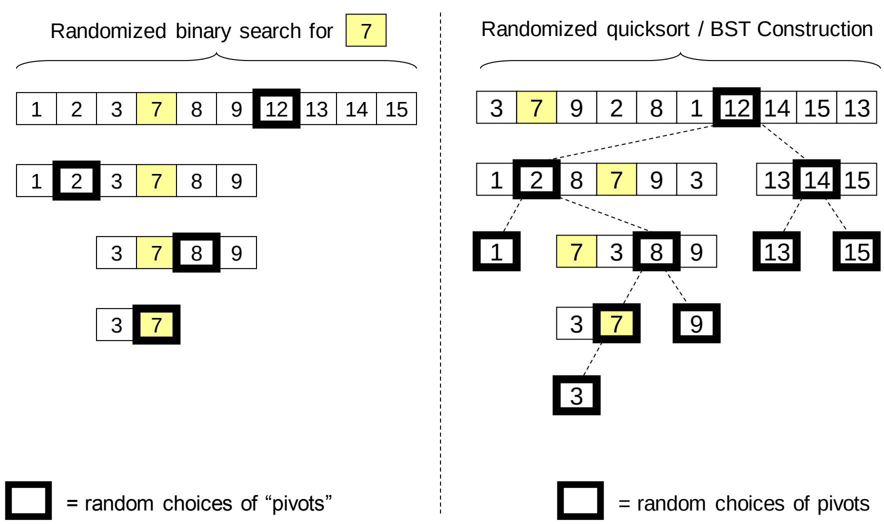
- **Theorem:** if we build a BST on n elements by inserting them in <u>random</u> order, then with high probability each call to insert will take O(log n) time.
- Equivalently, with high probability:
  - Each element will have depth O(log n).
  - The entire tree will have depth O(log n).
  - The entire tree will take O(n log n) time to build.
- **Corollary:** randomized quicksort runs in O(n log n) time with high probability!

20

Randomly-Built BSTs

- **Theorem:** if we build a BST on n elements by inserting them in <u>random</u> order, then with high probability each call to insert will take O(log n) time.
- Equivalently, with high probability:
  - Each element will have depth O(log n).
  - The entire tree will have depth O(log n).
  - The entire tree will take O(n log n) time to build.
- **Corollary:** randomized quicksort runs in O(n log n) time with high probability!

Randomized Quicksort, Binary Search, and BST Construction



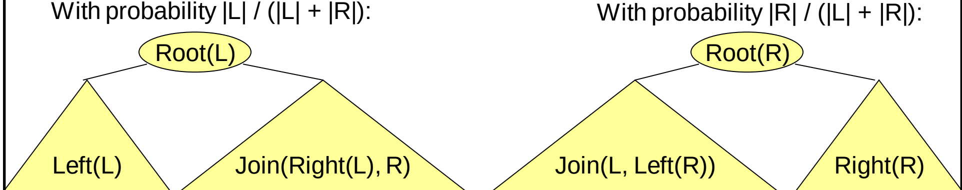
Maintaining Randomness

- If we build a BST at random on n elements, then with high probability it will be balanced.
- However, subsequent calls to insert and delete might cause the tree to become unbalanced.
- Remarkably, we can fix this by doing some carefully chosen random rotations after each insert and delete so the tree is always in a state that is "as if it was just randomly built from scratch".
  - More precisely: within each subtree, each element is equally likely to be at the root.
- This gives us a simple randomized mechanism for keeping a BST balanced with high probability.

23

Randomly-Balanced BSTs

- To insert an element e into an (n – 1)-element tree:
  - With probability 1/n, insert e at the root (insert as usual, then rotate up to root).
  - Otherwise (with probability 1 – 1/n), recursively insert into the left or right subtree of the root,
- To delete an element e, replace e with the a **randomized join** of e's two subtrees L and R:



24