Randomized Quicksort
Randomized Algorithms & Data Structures

- **Monte-Carlo algorithms**
  - Find the correct answer most of the time
  - Can usually amplify probability of success with repetitions
  - Example, Karger’s min cut (in textbook)

- **Las-Vegas algorithms**
  - Always find the correct answer, e.g. RandQuick sort (today!)
  - But the worst-case running-time guarantees are not strong (they hold in expectation or with high probability, but their goodness depends on randomness)

- **Randomized data structures**: hashing, search trees, filters, etc.
Randomized Algorithm I

Randomized Selection
Problem. Find the $k$th smallest/largest element in an unsorted array

- Recall our selection algorithm from back in our divide and conquer unit (lecture 15):

**Select** $(A, k)$:


Else:

Choose a pivot $p \leftarrow A[1, \ldots, n]$; let $r$ be the rank of $p$

$r, A_{<p}, A_{>p} \leftarrow \text{Partition}((A, p))$

If $k = r$: return $p$

Else if $k < r$: Select $(A_{<p}, k)$

Else: Select $(A_{>p}, k - r)$
Selection with a Good Pivot

- Recall: pivot is “good” if it reduced the array size by at least a constant
  - Gives a recurrence $T(n) \leq T(\alpha n) + O(n)$ for some constant $\alpha < 1$
  - Expands to a decreasing geometric series $T(n) = O(n)$
- In the deterministic algorithm, how did we find a good pivot?
  - Split array into groups of 5
  - And computed the median of group medians
  - The pivot guaranteed that $n \to 7n/10$
- **Here is a silly idea:** What if we pick the pivot uniformly at random?
  - Seems like the pivot is “usually” around the midpoint
  - What is the expected running time?
Randomized Selection

- **Problem.** Find the $k^{th}$ smallest/largest element in an unsorted array
- Recall our selection algorithm

**Select** $(A, k)$:

- Else:
  - Choose a pivot $p \leftarrow A[1, \ldots, n]$ uniformly at random; let $r$ be the rank of $p$
  - $r, A_{<p}, A_{>p} \leftarrow$ Partition($(A, p)$)
  - If $k = r$: return $p$
  - Else if $k < r$: Select $(A_{<p}, k)$
  - Else: Select $(A_{>p}, k - r)$
Analyzing Randomized Selection

• Normally, we’d write a recurrence relation for a recursive function

• A bit complicated now—input sizes of later recursive calls depend on the random choices of pivots in earlier calls

• We will use a different accounting trick for running time

• Randomized selection makes at most one recursive call each time:
  
  • Group multiple recursive call in “phases”
  
  • Sum of work done by all calls is equal to the sum of the work done in all the phases
Analyzing in Phases

- **Idea**: let a “phase” of the algorithm be the time it takes for the array size to drop by a constant factor (say \( n \rightarrow (3/4) \cdot n \))

- If array shrinks by a constant factor in each phase and linear work is done in each phase, what would be the running time?

- \( T(n) = c(n + 3n/4 + (3/4)^2n + \ldots + 1) = O(n) \)

- If we want a 1/4th, 3/4th split, what range should our pivot be in?
  - Middle half of the array (if \( n \) size array, then pivot in \([n/4, 3n/4]\))
  - What is the probability of picking such a pivot?
    - 1/2

- Phase ends as soon as we pick a pivot in the middle half
  - Expected # of recursive calls until phase ends? 2
Expected Running Time

- Let the algorithm be in phase $j$ when the size of the array is
  - At least $n \left( \frac{3}{4} \right)^{j+1}$ but not greater that $n \left( \frac{3}{4} \right)^j$
- Expected number of iterations within a phase: 2
- Let $X_j$ be the expected number of steps spent in phase $j$
- $X = X_0 + X_1 + X_2 \ldots$ be the total number of steps taken by the algorithm
- $E(X_j) = E(\# \text{ recursive calls until } j\text{th phase ends} \cdot \# \text{ steps in phase } j)$
- $E(X_j) \leq cn(3/4)^j \cdot E(\# \text{ recursive calls until } j\text{th phase ends}) = 2cn(3/4)^j$
Expected Running Time

- Let $X_j$ be the expected number of steps spent in phase $j$
- $X = X_0 + X_1 + X_2\ldots$ be the total number of steps taken by the algorithm
- $E(X_j) = E($ # of iterations until $j$th phase ends $\cdot$ # steps in phase $j)$
- $E(X_j) \leq n(3/4)^j \cdot E($ # iterations until $j$th phase ends $) = 2cn(3/4)^j$
- Now we can apply linearity of expectation:

$$E[X] = \sum_j E[X_j] \leq \sum_j 2cn \left(\frac{3}{4}\right)^j = 2cn \sum_j \left(\frac{3}{4}\right)^j$$

$$= \Theta(n)$$
Pivot Selection

- Deterministic and random both take $O(n)$ time
  - What’s the advantage of the deterministic algorithm?
  - Worst-case guarantee—the random algorithm could be very slow sometimes
  - What’s the advantage of the random algorithm?
  - Much much simpler and better constants hidden in $O()$
- Which should you use?
  - Pretty much always random
  - Question to ask yourself:
    - how often is the randomized algorithm going to be much worse than $O(n)$?
Randomized Algorithm II
Randomized QuickSort
Randomized Quicksort

• Recall deterministic Quicksort

• Depending on the choice pivot, could be $O(n^2)$

• What if we pick the pivot uniformly at random?
  • We saw in randomized selection that this leads to good pivots half of the time

Quicksort($A$):
If $|A| < 3$ : Sort($A$) directly
Else: choose a pivot element $p \leftarrow A$
  $A_{<p}, A_{>p} \leftarrow$ Partition around $p$
  Quicksort($A_{<p}$)
  Quicksort($A_{>p}$)
Randomized Quicksort

• Intuitively half the pivots will be good, half bad
• We will analyze quick sort using another accounting trick (see the textbook for example similar to selection’s approach of analyzing “phases”)
• Total work done can be split into two types:
  • Work done making recursive calls (this is a lower order term, it turns out)
  • Work partitioning the elements
• How many recursive calls in the worst case?
  • Imagine worst pivot being chosen each time
  • $O(n)$
Randomized Quicksort

- We thus need to bound the work partitioning elements
- Partitioning an array of size $n$ around a pivot $p$ takes exactly $n - 1$ comparisons
- We won't look at partitions made in each recursive call, which depend on the choice of random pivot
- **Idea:** Instead, account for the total work done by the partition step by summing up the total number of comparisons made
- Two ways to count total comparisons:
  - Look at the size of arrays across recursive calls and sum
  - Look at all pairs of elements and count total # of times they are compared (this is easier to do in this case)
Aside: Randomized Analysis

- Often multiple ways to determine a randomized algorithm’s cost
- We can split into phases, or count the cost directly. We can calculate each probability, or use linearity of expectation
- Intrinsically some “cleverness” involved in choosing the way that gets you a clean answer
- We’ll focus on problems where there’s a clear path to finding the solution (either it follows directly from the question, or we’ll revisit problems you’ve seen before). More complex problems abound if you look!
- That said, here’s a very clever way to calculate Quicksort’s running time
Counting Total Comparisons

- Just for analysis, let $B$ denote the sorted version of input array $A$, that is, $B[i]$ is the $i^{th}$ smallest element in $A$.

- Define random variable $X_{ij}$ as the number of times Quicksort compares $B[i]$ and $B[j]$.

- **Observation:** $X_{ij} = 0$ or $X_{ij} = 1$, why?
  - $B[i], B[j]$ only compared when one of them is the current pivot; pivots are excluded from future recursive calls.

- Let $T = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$ be the total number of comparisons made by randomized Quicksort.
Expected Running Time

- **Goal:** \( E[T] = E \left[ \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij} \right] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[X_{ij}] \)

- \( E[X_{ij}] = \Pr[X_{ij} = 1] \)

- When is \( X_{ij} = 1 \)? That is, when are \( B[i] \) and \( B[j] \) compared?

- Consider a particular recursive call. Let rank of pivot \( p \) be \( r \).
  - Let's think about where \( B[i], B[j] \) lie with respect to \( p \)
Expected Running Time

- **Goal:** \[ E[T] = E \left[ \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij} \right] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[X_{ij}] \]

- \[ E[X_{ij}] = \Pr[X_{ij} = 1] \]

- When is \( X_{ij} = 1 \)? That is, when are \( B[i] \) and \( B[j] \) compared?

- Consider a particular recursive call. Let rank of pivot \( p \) be \( r \).
  - Case 1. One of them is the pivot: \( r = i \) or \( r = j \)
  - Case 2. Pivot is between them: \( r > i \) and \( r < j \)
  - Case 3. Both less than the pivot: \( r > i, j \)
  - Case 4. Both greater than the pivot: \( r < i, j \)
Comparisons for Each Case

- **Case 1.** $r = i$ or $r = j$
  - $B[i]$ and $B[j]$ are compared once and one of them is excluded from all future calls

- **Case 2.** $r > i$ and $r < j$
  - $B[i]$ and $B[j]$ are both compared to the pivot but not to each other, after which they are in different recursive calls: will never be compared again

- **Case 3.** $r > i, j$ and **Case 4.** $r < i, j$
  - $B[i]$ and $B[j]$ are not compared to each other, they are both in the same subarray and may be compared in the future

- **Takeaway:** $B[i], B[j]$ are compared for the 1st time when one of them is chosen as pivot from $B[i], B[i + 1], \ldots, B[j]$ & never again
Expected Running Time

1. $\Pr[X_{ij} = 1] = \Pr(\text{one of them is picked as pivot from } B[i], B[i + 1], \ldots, B[j])$

2. $\Pr[X_{ij} = 1] = \frac{2}{j - i + 1}$

3. $E[T] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[X_{ij}] = 2 \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{j - i + 1}$
Expected Running Time

- $B[i]$ and $B[j]$ are compared iff one of them is the first pivot chosen from the range $B[i], B[i + 1], \ldots, B[j]$

- $\Pr[X_{ij} = 1] = \frac{2}{j - i + 1}$

- $E[T] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[X_{ij}] = 2 \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{j - i + 1}$

- For fixed $i$, inner sum is $\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n - i + 1} \leq \sum_{\ell=2}^{n} \frac{1}{\ell} = O(\log n)$

- Thus, expected number of comparisons is: $E[T] = O(n \log n)$
Quick Sort Summary

- Las Vegas algorithms like Quicksort and Selection are always correct and their running time guarantees hold in expectation.

- We can actually prove that the number of comparisons made by Quicksort is $O(n \log n)$ with high probability.
  
  - W.H.P. means that the probability that the running time of quicksort is more than a constant $c$ factor away from its expectation is very small (polynomially small: less than $1/n^c$ for $c \geq 1$).

- Whp bounds are called concentration bounds.

- Whp: ideal guarantees possible for a randomized algorithm.
Acknowledgments

• Some of the material in these slides are taken from
  • Shikha Singh
  • Jeff Erickson's Algorithms Book (http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf)