Randomized Algorithms: Introduction

- Approximate Median
- Selection
- Quicksort

Algorithmic design patterns.

- Greedy.
- Divide-and-conquer.
- Dynamic programming.
- Network flow.
- Randomization.

Randomized Algorithms. A randomized algorithm is an algorithm whose working not only depends on the input but also on certain random choices made by the algorithm.

Assumption. We have a random number generator Random(a, b) that generates for two integers a, b with a < b an integer r with a $a \le r \le b$ uniformly at random. We assume that Random(a, b) runs in O(1) time or precisely fair coin flip is done in unit time.

Why randomize? Can lead to simplest, fastest, or only known algorithm for a particular problem.

Ex. Symmetry breaking protocols, graph algorithms, quicksort, hashing, load balancing, Monte Carlo integration, cryptography.

Randomized Approximate Median

Randomized Approximate Median

Input. A set S of n numbers. Assume for simplicity that all numbers are distinct.

Rank. The rank of a number x in S is 1 plus the number of elements in S that are smaller than x.

Median. A median of S is a number of rank $\lfloor (n+1)/2 \rfloor$.

Approximate Median. A δ-approximate median is an element of rank k with $\left(\frac{1}{2} - \delta\right)(n+1) \le k \le \left(\frac{1}{2} + \delta\right)(n+1)$ for some given constant $0 \le \delta \le \frac{1}{2}$.

Problem. Report a δ-approximate median

Algorithm 1

```
ApproxMedian1(S, δ)
    r = Random(1,n)
    x* = S[r]
    k = 1
    for  i = 1 to n do
         if S[i] < x* then
              k = k+1
    if (1/2 − δ)(n + 1) ≤ k ≤ (1/2 + δ)(n + 1)  then
       return x*
     else
       return "error"
```

Running time. O(n)
Success probability. $\frac{\left(\frac{1}{2}+\delta\right)(n+1)-\left(\frac{1}{2}-\delta\right)(n+1)}{n} \approx 2\delta$

Ex. For $\delta = \frac{1}{4}$, the success probability is $\frac{1}{2}$ and for $\delta = \frac{1}{10}$ where we are looking for an element that is closer to the median, the success probability is getting worse.

Algorithm 2

```
ApproxMedian2(S, δ,c)
    j = 1
    repeat
        result = ApproxMedian1(S, δ)
        j = j+1
    until (result ≠ error) or (j = c+1)
    return result
```

Running time. O(cn)

Success probability. $1-(1-2\delta)^c$

Ex. For $\delta = \frac{1}{4}$ and c=10, we get a $\frac{1}{4}$-approximate median with success rate 99.9%. And For $\delta = \frac{1}{10}$ and c=10, we get a $\frac{1}{10}$-approximate median with success rate 89.2%.

Algorithm 3

```
ApproxMedian3(S, δ)
    repeat
        result = ApproxMedian1(S, δ)
    until result ≠ error
    return result
```

Success probability. 1

Running time.

E(running time of ApproxMedian3)=E((#calls to ApproxMedian1).O(n)) =O(n). E(#calls to ApproxMedian1)=O(n). $(1/2\delta)$=O(n/δ)

Remark. when we will talk about "expected running time" we actually mean "worst-case expected running time" (for different inputs the expected running time may be different —this is not the case in the ApproxMedian3).

Running Time

Deterministic Algorithms.

- $T_{\text{worst-case}}(n) = \max_{|X|=n} T(X)$
- $T_{\text{best-case}}(n) = \min_{|X|=n} T(X)$
- $T_{\text{average-case}}(n) = E_{|X|=n}(T(X)) = \sum T(x).\Pr(X = x)$

Randomized Algorithms.

- $T_{\text{worst-case expected}}(n) = \max_{|X|=n} E(T(X))$

Monte Carlo vs. Las Vegas Algorithms

Monte Carlo algorithm. Guaranteed to run in poly-time, likely to find correct answer.
Ex: ApproxMedian1

Las Vegas algorithm. Guaranteed to find correct answer, likely to run in poly-time.
Ex: ApproxMedian3

	Running time	Correctness
Las Vegas Algorithm	probabilistic	certain
Monte Carlo Algorithm	certain	probabilistic

Remark. ApproxMedian2 is mixture: the random choices both impact the running time and the correctness. Sometimes this is also called a Monte Carlo algorithm.

Remark. Can always convert a Las Vegas algorithm into Monte Carlo, but no known method to convert the other way.

Randomized Selection

Randomized Selection

Selection. Given a set S of n distinct elements and an integer i, we want to find the element of rank i in S

```
Selection(S,i)
   if |S| = 1 return the only element of S

   choose a splitter aj ∈ S uniformly at random
   foreach (a ∈ S) {
      if      (a < aj) put a in S-
      else if (a > aj) put a in S+
   }
   k = |S-|
   if k = i-1 then return aj
   else if k > i-1 then
        Selection(S-,i)
   else
        Selection(S+,i-k-1)
```

Randomized Selection: Analysis

Running time.

- [Best case.] Select the median element as the splitter: Selection makes $\Theta(n)$ comparisons ($T_{best}(n)=O(n)+T_{best}(n/2)$).
- [Worst case.] Select the smallest element as the splitter: Selection makes $\Theta(n^2)$ comparisons ($T_{worst}(n)=O(n)+T_{worst}(n-1)$).

Randomize. Protect against worst case by choosing splitter at random.

Intuition. If we always select an element that is bigger than 25% of the elements and smaller than 25% of the elements, then Selection makes $\Theta(n)$ comparisons.

Randomized Selection: Analysis

Running time.

$T_{exp}(n) = O(n) + \sum_{j=1}^{n} \Pr($element of rank j is splitter$)\cdot T_{exp}(\max(j-1, n-j))$

$= O(n) + 1/\mathrm{n} \sum_{j=1}^{n} \mathrm{T_{exp}}(\max(\mathrm{j}, \mathrm{n}-\mathrm{j}))$

It can be shown than $T_{exp}(n) = \mathrm{O(n)}$

Easier method. With probability 1/2 we recurse on at most 3n/4 elements. So

$$T_{exp}(n) \leq O(n) + \frac{1}{2} T_{exp}(3n/4) + \frac{1}{2} T_{exp}(n-1)$$

This recurrence is pretty easy to solve by induction.

Randomized Quicksort

Quicksort

Sorting. Given a set of n distinct elements S, rearrange them in ascending order.

```
RandomizedQuicksort(S) {
   if |S| = 0 return

   choose a splitter aᵢ ∈ S uniformly at random
   foreach (a ∈ S) {
      if      (a < aᵢ) put a in S⁻
      else if (a > aᵢ) put a in S⁺
   }
   RandomizedQuicksort(S⁻)
   output aᵢ
   RandomizedQuicksort(S⁺)
}
```

Quicksort: Analysis

Running time.

- [Best case.] Select the median element as the splitter: quicksort makes $\Theta(n \log n)$ comparisons.
- [Worst case.] Select the smallest element as the splitter: quicksort makes $\Theta(n^2)$ comparisons.

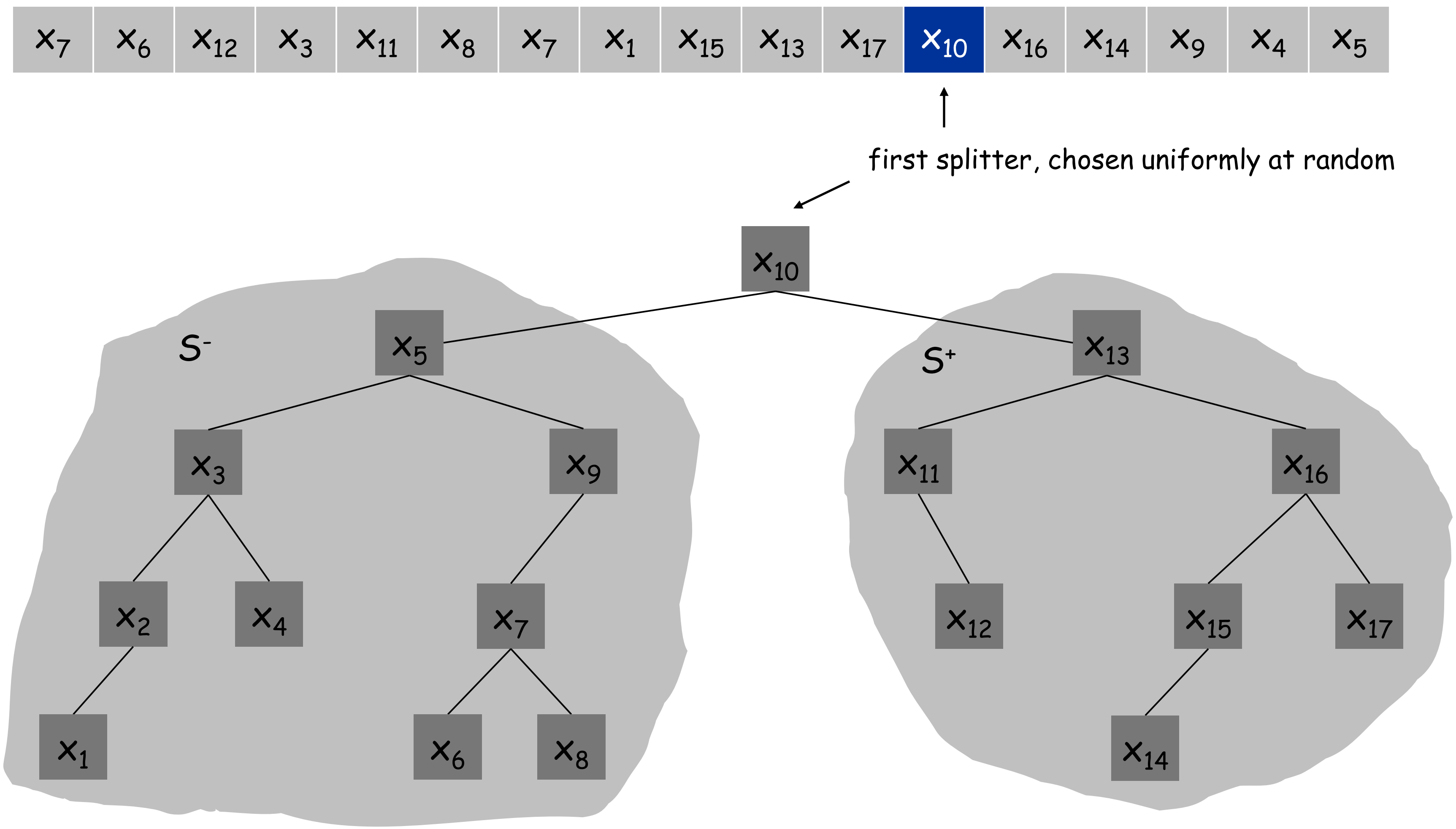
Randomize. Protect against worst case by choosing splitter at random.

Intuition. If we always select an element that is bigger than 25% of the elements and smaller than 25% of the elements, then quicksort makes $\Theta(n \log n)$ comparisons.

Notation. Label elements so that $x_1 < x_2 < \ldots < x_n$.

Quicksort: BST Representation of Splitters

BST representation. Draw recursive BST of splitters.

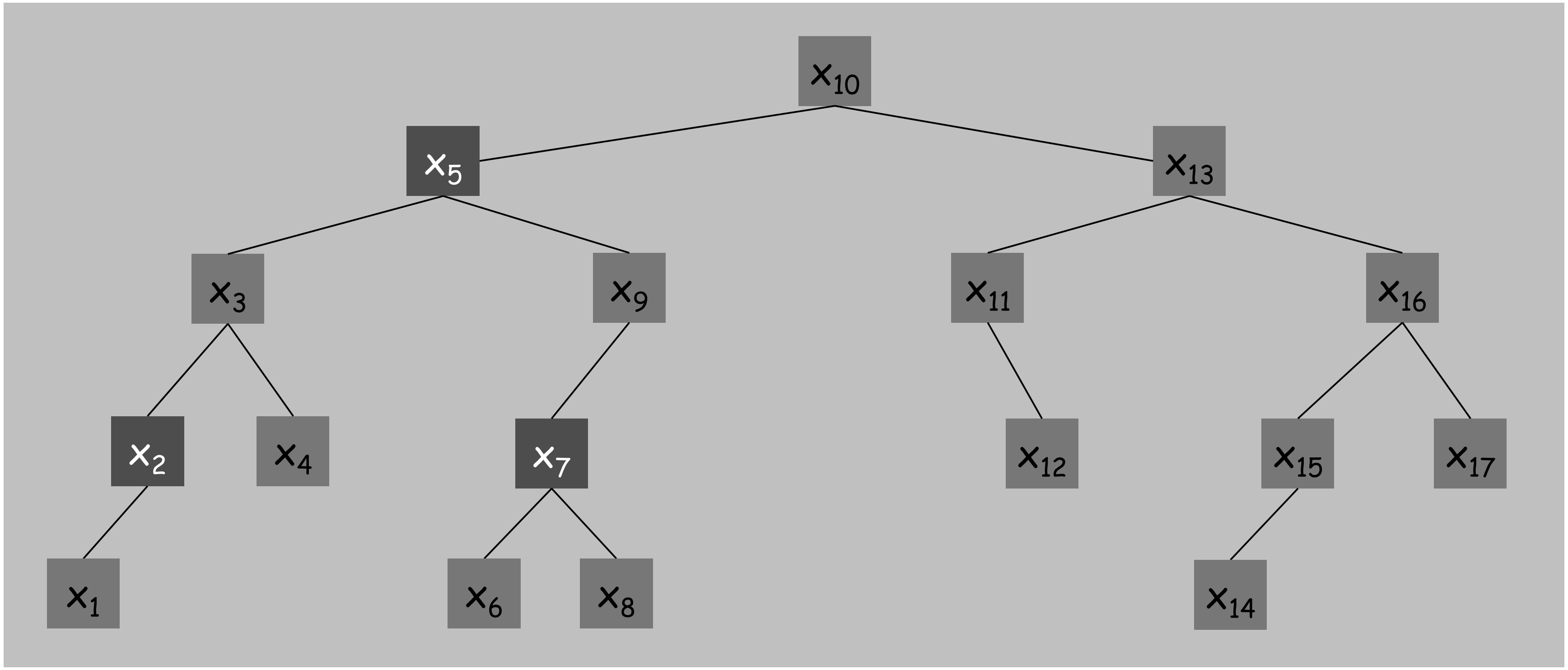


Quicksort: BST Representation of Splitters

Observation. Element only compared with its ancestors and descendants.

- x_2 and x_7 are compared if their lca = x_2 or x_7.
- x_2 and x_7 are not compared if their lca = x_3 or x_4 or x_5 or x_6.

Claim. Pr[x_i and x_j are compared] = $2 / |j - i + 1|$.



Quicksort: BST Representation of Splitters

Claim Proof.

- Consider $S_{ij} = \{x_i, \ldots, x_j\}$
- If splitter does not belong to S_{ij}, all elements of S_{ij} stay together and no comparison is made between x_i and x_j.
- This continues until at some point one of the elements in S_{ij} is chosen as the splitter.
- If x_i or x_j is selected to be the splitter, x_i and x_j are compared. Otherwise, x_i and x_j are never compared.
- Since each element of S_{ij} has equal probability of being chosen as splitter, we therefore find

$$\Pr[x_i \text{ and } x_j \text{ are compared}] = 2 / |j - i + 1|.$$

Quicksort: Expected Number of Comparisons

Theorem. Expected # of comparisons is O(n log n).

Pf.

- $X_{ij} = 1$ if x_i and x_j are compared. Otherwise, $X_{ij} = 0$
- $X = \sum X_{ij}$ is the #comparisons and $E(X) = \sum E(X_{ij})$

$$\sum_{1 \le i < j \le n} E(X_{ij}) = \sum_{1 \le i < j \le n} \frac{2}{j-i+1} = 2\sum_{i=1}^{n} \sum_{j=2}^{i} \frac{1}{j} \le 2n \sum_{j=1}^{n} \frac{1}{j} \approx 2n \int_{x=1}^{n} \frac{1}{x} dx = 2n \ln n$$

Ex. If n = 1 million, the probability that randomized quicksort takes less than 4n ln n comparisons is at least 99.94%.

> **Chebyshev's inequality.** $\Pr[|X - \mu| \ge k\sigma] \le 1 / k^2$.

Quicksort: Another Approach

Use approximate median. Instead of picking the pivot uniformly at random from S, we could also insist in picking a good pivot. An easy way to do this is to use algorithm ApproxMedian3 to find a (1/4)-approximate median. Now the expected running time is bounded by

E[(running time of ApproxMedian3 with δ = 1/4) + (time for recursive calls)] = O(n)+E[time for recursive calls]

$$T_{exp}(n) \leq O(n) + T_{exp}(3n/4) + T_{exp}(n/4)$$

Then,

$$T_{exp}(n) = O(n \ \log n)$$

.

References

References

- Lecture notes of advanced algorithms by Mark de berg
- The slides were prepared by Kevin Wayne. The slides are distributed by Pearson Addison-Wesley.