COMP 446 / 546 ALGORITHM DESIGN AND ANALYSIS

LECTURE 3 RANDOMIZED ALGORITHMS

ALPTEKİN KÜPÇÜ

Based on slides of Shafi Goldwasser and Cevdet Aykanat

RANDOMIZED QUICKSORT

- **Average-case assumptions:**
  - All permutations are equally likely
  - Cannot always expect to hold

- **Alternative to assuming a distribution: Impose a distribution**
  - Partition around a random pivot

RANDOMIZED ALGORITHMS

- **Randomization typically useful when**
  - There are many ways that algorithm can proceed
  - But, it is difficult to determine a way that is guaranteed to be good.
  - Many good alternatives; simply choose one randomly

- **Running time is independent of input**
- **No spesific input causes worst-case behavior**
- **Worst case determined only by the output of the random number generator**
- **No adversary can force worst-case behavior**

RANDOMIZED QUICKSORT

R-QUICKSORT (A,p,r)

```
if p<r then
        q ← R-PARTITION (A,p,r)
        R-QUICKSORT (A,p,q)
        R-QUICKSORT (A,q+1,r)
```

R-PARTITION (A,p,r)

```
s ← RANDOM (p,r)
swap A[ p ] ↔ A[ s ]
return Hoare-PARTITION (A,p,r)
```

We will show that the expected running time is O(n log n) for all input arrays A

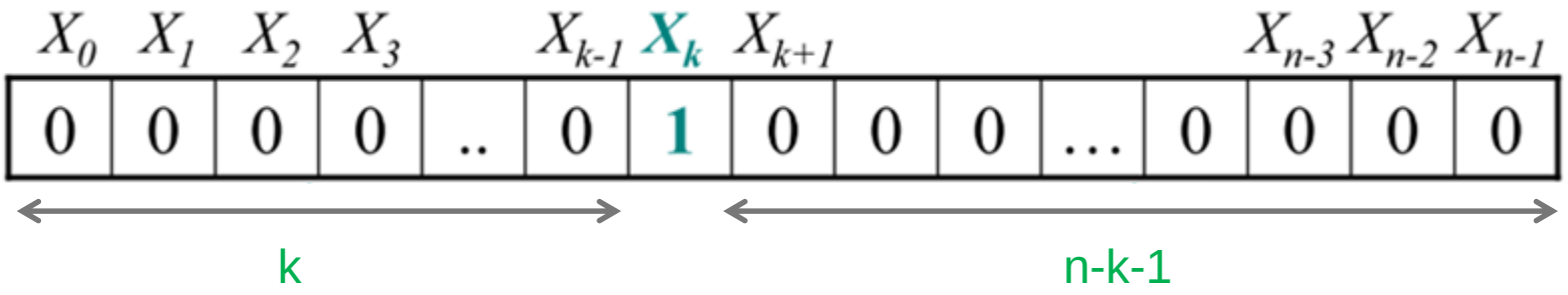
ANALYSIS METHOD #1: SUMMARY

- **Define indicator random variable X_k**
  - X_k marks the partition point for $k:(n-k-1)$ split.
- **Express expected running time $T(n)$ as a funtion of this indicator random variable.**
- **Calculate the expected value of E[$T(n)$] using properties of E[X_k].**
- **Prove E[$T(n)$] ≤ $c\ n$ log n by induction using the substitution method.**
- **Overall, this shows expected running time of Randomized Quicksort is O(n log n).**

ANALYSIS METHOD #1: INDICATOR RANDOM VARIABLES

- For k= 0,1,……., n-1, define the indicator random variable

X_k = { 1 if Partition generates a k:(n-k-1) split
        { 0 otherwise



ANALYSIS METHOD #1: INDICATOR RANDOM VARIABLES

$$T(n) = \begin{cases} T(0) + T(n-1) + cn & \text{if } 0{:}n-1 \text{ split} \\ T(1) + T(n-2) + cn & \text{if } 1{:}n-2 \text{ split} \\ \dots & \\ T(n-1) + T(0) + cn & \text{if } n-1{:}0 \text{ split} \end{cases}$$

$$T(n) = \sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + cn\right)$$

Only one X_k value is 1, all others are 0.

ANALYSIS METHOD #1: INDICATOR RANDOM VARIABLES

- **Expected value of indicator random variable X_k:**

$$E[X_k] = 1*Pr[X_k = 1] + 0*Pr[X_k = 0]$$
$$= 1*(1/n) + 0*((n-1)/n)$$
$$= 1/n$$

- **Since all splits are equally likely, assuming elements of the input are distinct.**

ANALYSIS METHOD #1: CALCULATING EXPECTATION

E[T(n)] $= \mathbf{E}[\sum_{k=0}^{n-1} X_k \, (T(k) + T(n-k-1) + \, cn)]$

ANALYSIS METHOD #1: CALCULATING EXPECTATION

$$\mathrm{E}[\,T(n)\,] = \mathrm{E}\left[\sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + cn\right)\right]$$

$$= \sum_{k=0}^{n-1} \mathrm{E}\left[\, X_k \left(T(k) + T(n-k-1) + cn\right)\,\right]$$

Linearity of expectation

ANALYSIS METHOD #1: CALCULATING EXPECTATION

$$E[\,T(n)\,] = \mathbf{E}\left[\sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + cn\right)\right]$$

$$= \sum_{k=0}^{n-1} \mathbf{E}\left[\, X_k \left(T(k) + T(n-k-1) + cn\right)\,\right]$$

$$= \sum_{k=0}^{n-1} \mathbf{E}[\, X_k\,] \,.\, \mathbf{E}\left[\, T(k) + T(n-k-1) + cn\,\right]$$

Independence of X_k from other random variables.

ANALYSIS METHOD #1: CALCULATING EXPECTATION

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + cn\right)\right]$$

$$= \sum_{k=0}^{n-1} \mathrm{E}\left[X_k \left(T(k) + T(n-k-1) + cn\right)\right]$$

$$= \sum_{k=0}^{n-1} \mathrm{E}[X_k] \cdot \mathrm{E}\left[T(k) + T(n-k-1) + cn\right]$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} \mathrm{E}[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} \mathrm{E}[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} cn$$

$\mathrm{E}[X_k] = 1/n$
and linearity of expectation

ANALYSIS METHOD #1: CALCULATING EXPECTATION

$$\text{E[T(n)]} = \mathbf{E}[\sum_{k=0}^{n-1} X_k\,(T(k) + T(n-k-1) + cn)]$$

$$= \sum_{k=0}^{n-1} \mathbf{E}[\, X_k\,(T(k) + T(n-k-1) + cn)\,]$$

$$= \sum_{k=0}^{n-1} \mathbf{E}[\, X_k\,] \,.\, \mathbf{E}[\, T(k) + T(n-k-1) + cn\,]$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} \mathbf{E}[\, T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} \mathbf{E}[\, T(n-k-1)\,] + \frac{1}{n}\sum_{k=0}^{n-1} cn$$

$$= \frac{2}{n}\sum_{k=0}^{n-1} \mathbf{E}[\, T(k)\,] + \Theta(n)$$

Both summations have identical terms and 1/n * n * cn = Θ(n)

ANALYSIS METHOD #1: CALCULATING EXPECTATION

$$\mathbf{E[\ T(n)\ ]} = \frac{2}{n}\sum_{k=0}^{n-1}\mathbf{E}[\ T(k)\ ] + \Theta(n)$$

ANALYSIS METHOD #1: CALCULATING EXPECTATION

$$\text{E[T(n)]} = \frac{2}{n} \sum_{k=0}^{n-1} \mathbf{E}[\, T(k) \,] + \Theta(n)$$

$$= \frac{2}{n} \sum_{k=2}^{n-1} \mathbf{E}[\, T(k) \,] + \Theta(n)$$

Simplify by absorbing k=0 and k=1 terms in Θ(n)

ANALYSIS METHOD #1: CALCULATING EXPECTATION

$$\text{E[T(n)]} = \frac{2}{n} \sum_{k=0}^{n-1} \mathbf{E}[\, T(k) \,] + \Theta(n)$$

$$= \frac{2}{n} \sum_{k=2}^{n-1} \mathbf{E}[\, T(k) \,] + \Theta(n)$$

Prove: **E[T(n)] ≤ c n log n**

for some constant c

ANALYSIS METHOD #1: SUBSTITUTION METHOD

E[T(n)] $\leq \frac{2}{n} \sum_{k=2}^{n-1} c\, k \log k + \Theta(n)$

Substitute inductive hypothesis

ANALYSIS METHOD #1: SUBSTITUTION METHOD

E[T(n)] $\leq \frac{2}{n} \sum_{k=2}^{n-1} c\, k \log k + \Theta(n)$

$= \frac{2c}{n} \left(\frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + \Theta(n)$

Use Fact: $\sum_{k=2}^{n-1} k \log k \leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2$

ANALYSIS METHOD #1: SUBSTITUTION METHOD

$$E[T(n)] \leq \frac{2}{n} \sum_{k=2}^{n-1} c\, k \log k + \Theta(n)$$

$$= \frac{2c}{n} \left(\frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + \Theta(n)$$

$$= c\, n \log n - \left(\frac{cn}{4} - \Theta(n) \right)$$

Express as desired minus residual

ANALYSIS METHOD #1: SUBSTITUTION METHOD

E[T(n)] $\leq \frac{2}{n} \sum_{k=2}^{n-1} c\, k \log k + \Theta(n)$

$= \frac{2c}{n} \left(\frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + \Theta(n)$

$= c\, n \log n - \left(\frac{cn}{4} - \Theta(n) \right)$

$\leq c\, n \log n$

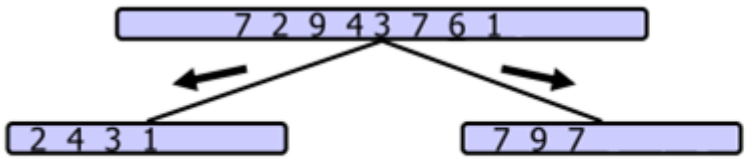
If *c* is chosen large enough *cn/4* dominates the $\Theta(n)$

ANALYSIS METHOD #2: PARANOID QUICKSORT

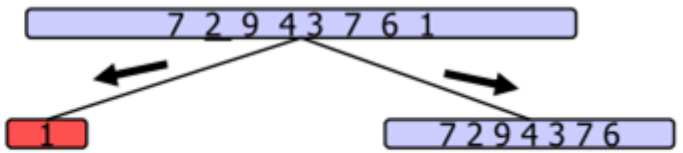
- **Repeat:**
  - Choose a random pivot
  - Perform PARTITION
  - Repeat until the resulting partition is "good"
    - The sizes of both L (less-than partition) and G (greater-than partition) are $\leq \frac{3}{4}n$
- **Recursively sort both sub-arrays**

ANALYSIS METHOD #2: PARANOID QUICKSORT

- **Consider a recursive call of paranoid quicksort on a sequence of size s.**
  - Good: The sizes of both L and G are ≤ ¾s
  - Bad: One of L or G has size ≥ ¾s



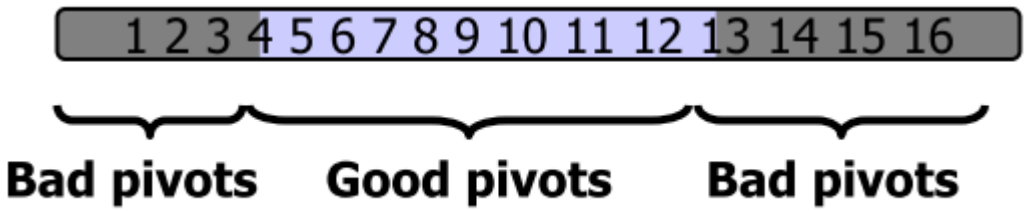
Good call



Bad call

ANALYSIS METHOD #2: PARANOID QUICKSORT

- **Claim:** A call is good with probability ½
- **Proof:** ½ of the possible pivots cause good calls:



ANALYSIS METHOD #2: PARANOID QUICKSORT

- **Let *T(n)* be the expected running time of paranoid quicksort.**
- **Partitioning takes #iterations*cn time**
- **T(n) = T(i) + T(n-i) + E[#iterations]*cn**
  - **i** $\in$ [n/4, 3n/4]
- **We showed**
  - Pr[good call] ≥ 1/2 => E[# iterations] ≤ 2

ANALYSIS METHOD #2: PARANOID QUICKSORT

- **Use a recursion tree argument:**
  - Tree depth is $\Theta(\log n)$
    - with log base 4/3
    - does not change asymptotics
- **Total cost at each level is at most $2cn$**
- **Overall expected runtime $T(n) = O(n \log n)$**

- **Randomized Quicksort still has $O(n^2)$ worst-case complexity, yet no particular input causes the worst-case!**

HIRING ASSISTANT PROBLEM

- **Scenario**
  - You are using an employment agency to hire a new office assistant.
  - The agency sends you one candidate each day.
  - You interview the candidate and must immediately decide whether or not to hire that person.
    - If you hire, you must also fire your current office assistant — even if it's someone you have recently hired.
  - Cost to interview is c_i per candidate (interview fee paid to agency).
  - Cost to hire is c_h per candidate (includes cost to fire current office assistant + hiring fee paid to agency).
- **Assume $c_h >> c_i$**

HIRE-ASSISTANT ALGORITHM

- **Rules**
  - Always hire the best candidate seen so far
  - Fire current office assistant if current candidate is better
  - Always have an assistant
    - Must hire the first candidate

```
HIRE-ASSISTANT (n)
        best = 0                        // candidate 0 is a least-qualified dummy candidate
        for i = 1 to n
                interview candidate i
                if candidate i is better than candidate best
                        best = i
                        hire candidate i
```

- **Goal: Determine what the price of this strategy will be.**

HIRE-ASSISTANT ANALYSIS

- **If there are *n* candidates, and we hire *m* of them, the total cost is $O(nc_i + mc_h)$**
- **Have to pay nc_i to interview, no matter how many we hire.**
- **Thus, focus on analyzing the hiring cost mc_h**
- **mc_h varies with each run**
  - It depends on the order in which we interview the candidates.
  - In the worst case, we hire all candidates.
  - This happens if each one is better than all who came before. In other words, if the candidates appear in increasing order of quality.
  - If we hire all, then the cost is $O(nc_i + nc_h) = O(nc_h)$

RANDOMIZED-HIRE-ASSISTANT

- **Change the scenario:**
  - The employment agency sends us a list of all candidates in advance.
  - On each day, we randomly choose a candidate from the list to interview (but considering only those we have not yet interviewed).
  - Instead of relying on the candidates being presented to us in a random order, we take control of the process and enforce a random order.

RANDOMIZED-HIRE-ASSISTANT (n)

randomly permute the list of candidates

HIRE-ASSISTANT (n)

RANDOMIZED-HIRE-ASSISTANT ANALYSIS

- **Let X be a random variable denotes the number of times we hire a new office assistant.**
- **Define indicator random variables $X_1,...,X_n$**
  - X_i is 1 if candidate i is hired, 0 otherwise.
  - $E[X_i]$ = Pr[i is hired] = Pr[i is the best candidate so far]
  - Assumption that the candidates arrive in random order means any one of these first i candidates is equally likely to be the best one so far.
  - Pr[i is the best candidate so far] = $1/i$
  - $X = X_1+...+X_n$

RANDOMIZED-HIRE-ASSISTANT ANALYSIS

$E[X] = E[\sum_{i=1}^{n} X_i]$

$= \sum_{i=1}^{n} E[X_i]$

$= \sum_{i=1}^{n} 1/i$

$= \ln n + O(1)$ (equation (A.7) : the sum is a harmonic series)

The expected hiring cost is $O(\log n\, c_h)$
Worst-case is still $O(nc_h)$
But now the agency cannot force the worst-case!

HOW TO RANDOMIZE A LIST?

- **Input: an array A[1..n]**
- **Output: Uniformly permuted array**
  - All n! permutations can be the output with equal probability

RANDOMIZE-IN-PLACE (A,n)

```
for i = 1 to n
    swap A[i] ↔ A[RANDOM(i,n)]
```

- **Read the book on correctness of this algorithm.**
  - Must make sure all n! permutations can be output and are equally likely to be output.
  - Makes a nice quiz question.

RANDOMIZED ALGORITHMS

- **Randomized algorithms do not change the worst-case**
- **But the average-case can sometimes be made as good as the best-case by randomization**
- **Or the possibility of worst-case input being supplied may be minimized (depends on the workload/distribution)**
- **Randomized algorithms may produce a different output each time they are executed given the same input**
- **Or may take different number of steps for the same input**
- **Therefore, randomized algorithms have expected running times**
- **Even though randomized algorithms do not change the worst-case, they may be easier to design. Some problems have randomized solutions that run much faster than deterministic solutions. (e.g., Primality Testing)**

FORMALITY ON RANDOMIZED ALGORITHMS

- **Also called Probabilistic Algorithms**
- **Algorithms that can flip coins as a basic step**
  - Namely: Algorithm can toss fair coin c which is either Heads or Tails with probability ½
  - Using this, algorithm can generate a random number r from some range {1…R}
  - Algorithm makes decisions based on the value of r
- **Where do we get coins from?**
  - An assumption – for this class.
  - In practice: Pseudo Random Number Generators
  - Fundamentally: Nature?

WHY USE RANDOMIZATION?

- Some problems cannot be solved, even in principle, if the computer is deterministic and cannot toss coins.
- For some problems, we only know exponential deterministic algorithms, whereas we can find polynomial randomized algorithms. (e.g., Polynomial Identity Testing)
- For some problems, can get a significant polynomial time speedup by going from a deterministic to a randomized algorithm. Say from $\Theta(n^8)$ to $\Theta(n^3)$.
- Intuition? Think of algorithm design as battling an adversary who is preparing an input which will slow it down. By using randomness, after the input is chosen, the adversary can not know what the algorithm will do in runtime and therefore cannot choose an input which causes a bad running time.

http://en.wikipedia.org/wiki/Randomized_algorithm#Where_randomness_helps

RANDOMIZED ALGORITHMS: TWO FLAVORS

- **Monte Carlo Algorithm:**
  - For every input
    - Regardless of coins tossed, algorithm always runs in polynomial time
    - Pr[output is correct] is high

- **Las Vegas Algorithm:**
  - For every input
    - Regardless of coins tossed, algorithm is always correct
    - The algorithm runs in expected polynomial time
      - Thus, for all but a small number of executions, the algorithm runs in polynomial time.

- The probabilities and expectations above are over the random choices of the algorithm! (not over the input)

MONTE CARLO ALGORITHMS

- **For every input of size n, algorithm:**
  - runs in worst-case polynomial time
  - is correct with high probability

MONTE CARLO ALGORITHMS

- **For every input of size n, algorithm:**
  - runs in worst-case polynomial time
  - is correct with high probability
    - Constant probability: $> {}^{1}/_{2}$

MONTE CARLO ALGORITHMS

- **For every input of size n, algorithm:**
  - runs in worst-case polynomial time
  - is correct with high probability
    - Constant probability: $> {}^{1}/_{2}$
    - Polynomial high probability: $> 1 - {}^{1}/_{n^{c}}$ for constant c > 0

MONTE CARLO ALGORITHMS

- **For every input of size n, algorithm:**
  - runs in worst-case polynomial time
  - is correct with high probability
    - Constant probability: $> {}^{1}/_{2}$
    - Polynomial high probability: $> 1 - {}^{1}/_{n^{c}}$ for constant c > 0
    - Exponential high probability: $> 1 - {}^{1}/_{2^{n^{c}}}$ for c > 0

MONTE CARLO ALGORITHMS

- **For every input of size n, algorithm:**
  - runs in worst-case polynomial time
  - is correct with high probability
    - Constant probability: $> 1/2$
    - Polynomial high probability: $> 1 - 1/n^c$ for constant $c > 0$
    - Exponential high probability: $> 1 - 1/2^{n^c}$ for $c > 0$
- **Can transform a constant probability to exponentially high probability by running the algorithm multiple times**
  - But there will still be a non-zero probability of being incorrect.
  - e.g., Miller-Rabin Primality Test

LAS VEGAS ALGORITHMS

- **Running time not only depends on the input size, but also the random coins.**
  - e.g., on input x and random coins r, represent running time as a function $T(x,r)$ of x and r
- **Algorithm runs in expected polynomial time:**
  - $T(n) = E[\ T(x,r)\ ]$ for any input x of length n
  - $T(n)$ is bounded by a polynomial function in n
- **Algorithm is always correct.**
- **Examples: Randomized Quicksort, Hire-Assistant**