CS 5321:
Advanced Algorithms –
Probabilistic Analysis &
Randomized Algorithms
Ali Ebnesasir
Department of Computer Science
Michigan Technological University

Acknowledgement

• Eric Torng
• Moon Jung Chung
• Charles Ofria

Outline

• Tools from probability theory
  – Example: indicator random variables, expected value, linearity of expectation
• Probabilistic (average-case) analysis
  – Example: Hiring problem
• Randomized algorithms
  – The role of pseudo random generators
    – Example: hiring problem, birthday paradox
• Online hiring problem
• Quick sort
Tools for Probabilistic Analysis - I

- Random variable
  - Represent a function from a finite or countably infinite sample space to real numbers
- Indicator random variable
  - Represents the occurrence of an event
- Let \( I \) be the indicator random variable for an event \( E \). Then,
  - \( I = 1 \) if \( E \) occurs
  - \( I = 0 \) if \( E \) does not occur
- Expected value of a random variable represents the average (mean) of the values it takes
  - Example: earn $3 for each head (H), lose $2 for each tail (T)
  - What is the expected value of a random variable \( X \) representing the profit?

\[
E[X] = 6 \cdot Pr\{2 \text{ H's}\} + 1 \cdot Pr\{1 \text{ H}, 1 \text{ T}\} - 4 \cdot Pr\{2 \text{ T's}\} = 1
\]

Tools for Probabilistic Analysis - II

- If a random variable \( X \) represents a composite of many random events, then
  - Define a collection of “indicator” variables \( X_i \) that focus on individual events; typically \( X = \sum X_i \)
- Linearity of expectations
  - Let \( X, Y, \) and \( Z \) be random variables s.t. \( X = Y + Z \)
- Recurrence Relations

Probabilistic (Average-Case) Analysis

- Algorithm is deterministic
  - For a fixed input, it will run the same every time
- Analysis Technique
  - Assume a probability distribution for your inputs
  - Analyze item of interest over probability distribution
- Caveats
  - Specific inputs may have much worse performance
  - If distribution is wrong, analysis may give misleading pictures
Example: Hiring Problem

- **Input**
  - A sequence of n candidates for a position
  - Each has a distinct quality rating that we can determine in an interview
- **Algorithm**
  - `best = 0;`
  - For k = 1 to n // total number of candidates
    - If candidate(k) is better than `best`, then
      - hire(k) and `best = k;`
- **Cost:**
  - Number of hires
- **Worst-case cost is n**

Probabilistic Analysis of the Hiring Problem

- **Assume a probability distribution**
  - Each of the n! permutations is equally likely, i.e., uniform random permutations
- **Analyze item of interest over probability distribution**
  - Define random variables
    - Let X = random variable corresponding to # of hires
    - Let X_i = “indicator variable” that i’th interviewed candidate is hired
      - Value 0 if not hired, 1 if hired
    - X = \[ \sum_{i=1}^{n} X_i \]
    - E[X] = ?
  - Explain why:
    - E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = ?

Randomization

- **Instead of assuming that the input is ordered randomly, randomly order the input**
  - For a fixed input, it will run differently depending on the result of random “coin tosses”
- **Hiring problem:**
  - Randomly permute the list of candidates
  - `best = 0;`
  - For k = 1 to n
    - If candidate(k) is better than `best`, then
      - hire(k) and `best = k;`
  - Each execution is likely to differ from its previous execution
  - No particular input elicits its worst-case behavior

What is the expected hiring cost?
Randomization –
Permute-By-Sorting

- How do we generate a uniform random permutation on the input?
- Permute-By-Sorting
  - For i = 1 to n
    do P[i] = RANDOM(1, n)  // RANDOM generates distinct priorities
  - Sort input using array P as the array of keys
- Does it generate a uniform random permutation where the probability of each permutation is 1/n!?
- What is the probability of generating a list where each element i receives the i-th smallest priority?

Randomization – In-Place

- In-Place-Randomization
  - For i = 1 to n
    swap inputElement[i] with inputElement[RANDOM(i,n)]
- Does this algorithm generate a uniform random permutation?

The Birthday Paradox - I

- How many people must be in a room before there is a 50% chance that two of them have the same birthday?
- Index the people from 1 to k
- n: the number of days in a year
- What is the probability of i’s birthday and j’s being on a specific day d?
- What is the probability of i’s birthday and j’s being the same?
The Birthday Paradox - II

• How about the probability of at least 2 people having the same birthday?
• $1 - \text{probability of all birthdays are different}$
• $A_i$: the event that person $i$’s birthday is different from $j$’s, for all $j < i$
• $B_k = \cap_{i=1}^{k} A_i = A_k \cap B_{k-1}$
• $\Pr[B_k] = \Pr[B_{k-1}] \cdot \Pr[A_k | B_{k-1}]$, where $\Pr[A_k | B_{k-1}] = (n-k+1) / n$
• Why?

The Birthday Paradox - III

• Iteratively apply the recurrence $\Pr[B_k] = \Pr[B_{k-1}] \cdot \Pr[A_k | B_{k-1}]$
• $\Pr[B_k] = \Pr[B_{k-2}] \cdot \Pr[A_{k-1} | B_{k-2}] \cdot \Pr[A_k | B_{k-1}]$
• $\cdots$
• $= \Pr[B_1] \cdot (1 - 1/n) \cdot (1 - 2/n) \cdots (1 - (k-1)/n)$
• Having $(1+x) \leq e^x$ gives us $\Pr[B_k] \leq e^{-k(k-1)/2n}$
• Since we need $\Pr[B_k] \leq 1/2$, we should have $-k(k-1)/2n \leq \ln(1/2)$
• For $n = 365$, we have $k \geq 23$

The Birthday Paradox – Alternative Analysis

• Indicator random variables: simpler but approximate analysis
  – $X_{ij} = 1$ if person $i$ and $j$ have the same birthday
  – $X_{ij} = 0$ if person $i$ and $j$ have the same birthday
• $E[X_{ij}] = \Pr[\text{person } i \text{ and } j \text{ have the same birthday}]$
  = $1/n$
• Let $X$ represent the number of pairs having the same birthday. Thus,
• $X = \sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij}$
• $E[X] = \sum_{i=1}^{k} \sum_{j=i+1}^{k} E[X_{ij}]$
• $E[X] = (k)(1/n) = k(k-1)/2n$
• For $n = 365$, we have $k \geq 28$
Indicator Random Variables vs. Probability Analysis

- **Pros**
  - Simplify the probabilistic analysis

- **Cons**
  - Provide an approximate cost

Online Hiring Problem

- **Input**
  - A sequence of $n$ candidates for a position
  - Each has a distinct quality rating that we can determine in an interview
    - We know total ranking of interviewed candidates, but not with respect to candidates left to interview
  - We can hire only once

  - **OnLineHire(k,n)**
    - $best = 0$
    - For $j = 1$ to $k$
      - If candidate($j$) is better than best, $best = j$
    - For $j = k+1$ to $n$
      - If candidate($j$) is better than best, hire($j$) and return;
    - return $n$

- **Questions:**
  - What is the probability we hire the best qualified candidate given $k$?
  - What is the best value of $k$ to maximize above probability?

Online Hiring Analysis

- Let $S$ be the probability that we successfully hire the best qualified candidate
- Let $S_i$ be the probability that the best-qualified candidate is the $i$-th one
- Then, we have $Pr(S) = \sum_{i=k+1}^{n} Pr(S_i)$; Why?
- Let $M(j) =$ the maximum score amongst candidates $1$ through $j$
- What needs to happen for $S_i$ to be true?
  - Best candidate is in position $i$: $B_i$
  - No candidate in positions $k+1$ through $i-1$ are hired: $O_i$
  - These two quantities are independent, so we can multiply their probabilities to get $S_i$
Computing S

- \( B_i = \frac{1}{n} \)
- \( O_i = \frac{k}{i(i-1)} \)
- \( S_i = \frac{k}{i(i-1)} \)
- \( S = \sum_{i > k} S_i = \frac{k}{n} \sum_{i > k} \frac{1}{i(i-1)} \) is probability of success
- \( \frac{k}{n} (H_n - H_k) \): roughly \( \frac{k}{n} (\ln n - \ln k) \)
- Maximized when \( k = n/e \)
- Leads to probability of success of \( 1/e \)

Quicksort Algorithm

- Overview
  - Choose a pivot element
  - Partition elements to be sorted based on partition element
  - Recursively sort smaller and larger elements

Quicksort Walkthrough

17  12  6  23  19  8  5  10
6  8  5  10  17  12  23  19
5  6  8  17  12  19  23
6  8  12  17  23
6  17
5  6  8  10  12  17  19  23
Pseudocode

Quicksort(A, low, high) {
    if (low < high) {
        pivotLocation = Partition(A, low, high);
        Quicksort(A, low, pivotLocation - 1);
        Quicksort(A, pivotLocation+1, high);
    }
}

Pseudocode

int Partition(A, low, high) {
    pivot = A[high];
    leftwall = low-1;
    for i = low to high-1 {
        if A[i] < pivot then {
            leftwall = leftwall+1;
            swap(A[i], A[leftwall]);
        }
        swap(A[high], A[leftwall+1]);
    }
    return leftwall+1;
}

Worst Case for Quicksort
Average Case for Quicksort?

Intuitive Average Case Analysis

Anywhere in the middle half is a decent partition

\[
\left(\frac{3}{4}\right)^h n = 1 \implies n = \left(\frac{4}{3}\right)^h \quad \text{// h is the number of partitions}
\]

\[
\log(n) = h \log\left(\frac{4}{3}\right)
\]

\[
h = \frac{\log(n)}{\log\left(\frac{4}{3}\right)} < 2 \log(n)
\]

How many steps?

At most \(2\log(n)\) decent partitions suffices to sort an array of \(n\) elements.

But if we just take arbitrary pivot points, how often will they, in fact, be decent?

Since any number ranked between \(n/4\) and \(3n/4\) would make a decent pivot, half the pivots on average are decent.

Therefore, on average, we will need \(2 \times 2\log(n) = 4\log(n)\) partitions to guarantee sorting.
Formal Average-Case Analysis

- Let $X$ denote the random variable that represents the total number of comparisons performed.
- Let indicator variable $X_{ij} = \begin{cases} 0 & \text{if not compared}, \\ 1 & \text{if compared} \end{cases}$.
- $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$
- $E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$

Computing $E[X_{ij}]$

- $E[X_{ij}] = \text{probability that i and j are compared}$
- Observation:
  - All comparisons are between a pivot element and another element.
  - If an item $k$ is chosen as pivot where $i < k < j$, then items $i$ and $j$ will not be compared.
- $E[X_{ij}] = \sum_{k=1}^{n-i} \frac{2}{k+1} \leq \sum_{i=1}^{n-1} 2 H_{n-i+1}$
  $\leq 2 (n-1) H_n$

Computing $E[X]$

- $E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$
  $= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$
  $= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$
  $= 2 (n-1) H_n$
  $= 2 (n-1) H_n$
Alternative Average-Case Analysis

- Let $T(n)$ denote the average time required to sort an $n$-element array
  - “Average” assuming each permutation equally likely
- Write a recurrence relation for $T(n)$
  - $T(n) =$
- Using induction, we can then prove that $T(n) = \Theta(n \log n)$
  - Requires some simplification and summation manipulation

Avoiding the Worst-Case

- Understanding quicksort’s worst-case
- Methods for avoiding it
  - Pivot strategies
  - Randomization

Understanding the worst case

A B D F H J K
A B D F H J
A B D F H
A B D F
A B D
A B
A

The worst case occurs in a likely case for many applications.
Pivot Strategies

• Use the middle Element of the sub-array as the pivot.
• Use the median element (first, middle, last) to make sure to avoid any kind of pre-sorting.

What is the worst-case performance for these pivot selection mechanisms?

Randomized Quicksort

• Make chance of worst-case run time equally small for all inputs
• Methods
  – Choose pivot element randomly from range [low..high]
  – Initially permute the array