Announcements

- Homework #3 will be posted today
  - The homework is due by 11:30 p.m. next week
- There is a quiz today!
  - 15 minutes at the end of the class
  - Materials from weeks 1, 2, and 3 lectures
  - **Closed** books and notes
  - No electronic devices (cell phones, laptops, etc.)

Matrix Multiplication

- Multiplying matrices
  - Suppose $A = (a_{ij})$ and $B = (b_{ij})$ are square n x n matrices
  - Then, if $C = A \cdot B$,

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

  - Must compute n^2 matrix entries, and each is the sum of n values

Matrix Multiplication

The pseudo-code for square matrix multiplication

```
squareMatrixMultiply(A, B)
  n = A.rows
  let C be a new n x n matrix
  for i = 1 to n
    for j = 1 to n
      c_ij = 0
      for k = 1 to n
        c_ij = c_ij + (a_ik · b_kj)
  return C
```

- The first `for` loop computes the entries of each row `i`
- The second `for` loop computes the entries of each column `j` → c_{ij}

Matrix Multiplication

- What is the time complexity for this algorithm?
  - There are 3 nested `for` loops
  - Each loop gets iterated n times

$$T(n) = c * n * n * n = \Theta(n^3)$$

  - Hence, matrix multiplication is increasingly costly as the value of n gets larger
  - A better technique which uses Strassen's algorithm runs in $\Theta(n^{\log_2 7})$ time
    - $\log_2 7 \approx 2.81$ so the time complexity is about $\Theta(n^{2.81})$

Strassen's Algorithm Matrix Multiplication

- For the product C of two 2 x 2 matrices, A and B

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

  - Strassen's algorithm says if

$$\begin{aligned} m_1 &= (a_{11} + a_{22}) \cdot (b_{11} + b_{22}) \\ m_2 &= (a_{21} + a_{22}) \cdot b_{11} \\ m_3 &= a_{11} \cdot (b_{12} - b_{22}) \\ m_4 &= a_{22} \cdot (b_{21} - b_{11}) \\ m_5 &= (a_{11} + a_{12}) \cdot b_{22} \\ m_6 &= (a_{21} - a_{11}) \cdot (b_{11} + b_{12}) \\ m_7 &= (a_{12} - a_{22}) \cdot (b_{21} + b_{22}) \end{aligned}$$

Strassen's Algorithm Matrix Multiplication

- Then, the product C is given by

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

- Strassen's algorithm partitions large matrices into sub-matrices, assuming that n is a power of 2 (i.e., $n = 2^k$)

$$\left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right] = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \times \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right]$$

- Each partition contains a sub-matrix of size $\frac{n}{2} \times \frac{n}{2}$
- Then, we compute $M_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$ all the way to M_7, as show earlier. Finally, determine the product C.

Strassen's Algorithm Matrix Multiplication

- Analyzing the algorithm
  - The base case, when $n = 1$, $T(1) = \Theta(1)$
  - The recursive case, when $n > 1$, each sub-matrix of size $\frac{n}{2} \times \frac{n}{2}$ is used. Since the algorithm is called 7 times (m_1 through m_7),

$$T(n) = 7T\left(\tfrac{n}{2}\right) + \Theta(n^2)$$

  - Combining the 2 cases, we get

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 7T\left(\tfrac{n}{2}\right) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

  - This is equivalent to

$$T(n) = n^{\log_2 7} \approx n^{2.81} \in \Theta(n^{2.81})$$

The Master Method

- Provides method for solving recurrences of the form

$$T(n) = aT\left(\tfrac{n}{b}\right) + f(n)$$

  where $a \geq 1$ and $b > 1$ are constants, and $f(n)$ is an asymptotically positive function
  - Divide a problem of size n into a subproblems, each of size ${}^{n}/_{b}$
  - The a subproblems are solved recursively, each in $T\left(\frac{n}{b}\right)$
  - The function $f(n)$ represents the costs of dividing the problem and combining the results of the subproblems

The Master Method

- The Master Theorem
  - Let $a \geq 1$ and $b > 1$ be constants, and $f(n)$ be a function. Given
    $$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$
  - Then $T(n)$ has the following asymptotic bounds:
    - If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then
      $$T(n) = \Theta(n^{\log_b a})$$
    - If $f(n) = \Theta(n^{\log_b a})$, then
      $$T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)$$
    - If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$, then
      $$T(n) = \Theta(f(n))$$

The Master Method

- The Master Theorem
  - In each case, $f(n)$ is compared with $n^{\log_b a}$
  - The larger of the two determines the solution to the recurrence
    - In the first case, $n^{\log_b a}$ is larger, so the solution is
      $$T(n) = \Theta(n^{\log_b a})$$

      $f(n)$ must be polynomially smaller than $n^{\log_b a}$; then, we can use case 1

The Master Method

- Using the Master Method
  - Simply determine which case (if any) of the master theorem applies
    - Example 1
      $$T(n) = 9T\left(\frac{n}{3}\right) + n$$

      Here $a = 9$, $b = 3$, and $f(n) = n$, so
      $$n^{\log_b a} = n^{\log_3 9} = n^2$$
      $$T(n) = \Theta\left(n^{\log_3 9}\right) = \Theta(n^2)$$

      Since $f(n) = O(n^{\log_3 9 - \epsilon})$, where $\epsilon = 1$, is polynomially smaller, we can use case 1

The Master Method

- Using the Master Method
    - Example 2
      $$T(n) = T\left(\frac{2n}{3}\right) + 1$$

      Here $a = 1$, $b = \frac{3}{2}$, and $f(n) = 1$, so
      $$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$$
      $$T(n) = \Theta\left(n^{\log_b a} \cdot \log_2 n\right) = \Theta(1 \cdot \log_2 n) = \Theta(\log_2 n)$$

      Since $f(n) = \Theta\left(n^{\log_b a}\right) = \Theta(1)$, we can use case 2

The Master Method

- Using the Master Method
  - Apply this to the Merge Sort and the Max-Subarray algorithms

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Here $a = 2$, $b = 2$, and $f(n) = \Theta(n)$, so

$$n^{\log_b a} = n^{\log_2 2} = n$$

Since $f(n) = \Theta(n)$, we can use case 2

$$T(n) = \Theta\left(n^{\log_b a} \cdot \log_2 n\right) = \Theta(n \cdot \log_2 n)$$

The Master Method

- Using the Master Method
  - Apply this to the Strassen's algorithms

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$$

Here $a = 7$, $b = 2$, and $f(n) = \Theta(n^2)$, so

$$n^{\log_b a} = n^{\log_2 7}$$

Since $f(n) = O(n^{\log_2 7 - \epsilon})$, where $\epsilon = 0.81$, is polynomially smaller, we can use case 1

$$T(n) = \Theta\left(n^{\log_2 7}\right) \in \Theta(n^{2.81})$$

The Expected Value

- The expected value (average) example
  - Suppose there are 4 students with heights 67, 68, 72, and 74 inches

$$Average\ height = \frac{67 + 68 + 72 + 74}{4} = 70.25\ inches$$

  - Suppose there are 100 students with height distribution:

% of Students	Height	% of Students	Height
25	67	35	72
30	68	10	74

$$\begin{aligned} Average\ height &= 67(0.25) + 68(0.3) + 72(0.35) + 74(0.1) \\ &= 69.75\ inches \end{aligned}$$

Also referred to as a weighted average value.

The Expected Value

- Suppose we have a probability space with the sample space

$$\{e_1, e_2, e_3, \ldots, e_n\}$$

and each outcome e_i has a real number $f(e_i)$, ***random variable***, associated with it.

- The ***expected value***, or average, of $f(e_i)$ is given by

$$f(e_1)p(e_1) + f(e_2)p(e_2) + \cdots + f(e_n)p(e_n)$$

also called ***chance variable*** or ***stochastic variable***

The Hiring Problem

- Need to hire a new office assistant
- Interview 1 candidate each day
- Decide either to hire the person or not
- Have to pay employment agency some fee to interview an applicant
- To hire a new person, you must fire the current office assistant and pay large hiring fees to the agency
- Commit to hire the best possible person

The Hiring Problem

- Pseudocode for Hire Assistant
  - Candidates for the job are numbered 1 through n
  - After interviewing candidate i, determine whether he/she is the best so far
  - Initialize with a dummy candidate 0, that is least qualified

```
hireAssistant(n)
best = 0
for i = 1 to n
  interview candidate i
  if candidate i is better than best
    best = i
    hire candidate i
```

The Hiring Problem

- The cost model is not the running time
- Focus on the cost for interviewing and hiring
  - Similar analytical techniques as for running time
  - Counting number of times certain basic operations are executed
  - c_i = cost of interviewing (low)
  - c_h = cost of hiring (high)
  - m = number of people hired
  - Total cost = $O(c_i n + c_h m)$

The Hiring Problem

- Represents a model for a common computational paradigm
- Often need to find the max or min value in a sequence
  - Examine each element of the sequence
  - Maintain a current "winner"
- Worst case $\rightarrow$ Hire every candidate
  - Occurs if all candidates come in strictly higher quality (hire n times) $\rightarrow$ Total cost of $O(c_h n)$

Probabilistic Analysis

- The use of probability to analyze problems
- Most common → analyze the running time of an algorithm
- Sometimes → use to analyze hiring cost
- Use knowledge of the distribution of the inputs
  - Average running time over all possible inputs → **average-case running time**

Probabilistic Analysis

- For the hiring problem:
  - Assume that applicants come in random order
  - There is a total order on the candidates
    - Can rank each candidate with a unique number from 1 to through n
    - Use rank(i) to denote the rank of applicant i
    - Higher rank → Better qualified
    - Thus, the ordered list ⟨rank(1), rank(2), … , rank(n)⟩ is a permutation of the list ⟨1, 2, … , n⟩ of applicants
- **Uniform random permutation** – each of the possible $n!$ permutations has equal probability

Probabilistic Analysis

- Probabilistic analysis → Need to look at distribution of inputs
  - Usually know very little about this distribution
  - Also may not be able to model it computationally
- Making the behavior of part of the algorithm random allows you to use probability and randomness to design and analyze algorithm

Randomized Algorithms

- For the hiring problem, there is no way to know whether the candidates are sent randomly
  - So, implement control over the order for interview
  - Get the list of all candidates in advance
  - Randomly select an applicant for each day
  - This way, we ensure that the order is random

Randomized Algorithms

- **Randomized Algorithm** → Combine the input with values produced by a random-number generator (e.g., a `random` method)
  - Call `random(a,b)` gives a random integer between `a` and `b`, inclusive
    - `random(2,5)` returns either 2, 3, 4, or 5 (each with probability of $^1/_4$)
    - Subsequent number returned is independent of the previous calls
- Running time of a randomized algorithm is referred to as an **expected running time**
  - Here, the algorithm itself makes the random choices

Indicator Random Variables

- ***Indicator Random Variables*** provide a convenient method for probabilities → expectations conversion
  - For an event A in a sample space S, such variable can be defined as

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs} \\ 0 & \text{if } A \text{ does not occur} \end{cases}$$

  - For example, flipping a coin
    - Sample space is $S=\{H, T\}$
    - Probability: $pr\{H\} = pr\{T\} = {}^1/_2$

Indicator Random Variables

- Define an indicator random variable X_H for coin coming up head

$$\begin{aligned} X_H &= I\{H\} \\ &= \begin{cases} 1 & \text{if } H \text{ occurs} \\ 0 & \text{if } T \text{ occurs} \end{cases} \end{aligned}$$

- The expected number of heads from one coin flip is

$$\begin{aligned} E[X_H] &= E[I\{H\}] \\ &= 1 \cdot pr\{H\} + 0 \cdot pr\{T\} \\ &= 1 \cdot \left(\tfrac{1}{2}\right) + 0 \cdot \left(\tfrac{1}{2}\right) \\ &= {}^1/_2 \end{aligned}$$

Indicator Random Variables

- Thus, the expected value of an indicator random variable associated with an event A is equal to the probability that A occurs
  - If $X_A = I\{A\}$, then $E[X_A] = pr\{A\}$
- Let $X_i = I\{\text{the } i^{th} \text{ flip results in the event } H\}$, then

X = total number of heads in the n coin flips

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

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

Indicator Random Variables

- Computation of $E[X]$ gives

$$E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i]$$
$$= \sum_{i=1}^{n} \frac{1}{2} = \frac{n}{2}$$

- Applying this to the hiring problem
  - Let X be the random variable = number of times we hire a new office assistant. Therefore,

$$E[X] = \sum_{x=1}^{n} x \cdot pr\{X = x\}$$

  - But this calculation would be cumbersome

Indicator Random Variables

- Use indicator random variable to simplify the calculation
- Let X_i be the indicator random variable where the i^{th} candidate is hired

$$X_i = I\{\text{candidate } i \text{ is hired}\}$$
$$= \begin{cases} 1 & \text{if candidate } i \text{ is hired} \\ 0 & \text{if candidate } i \text{ is not hired} \end{cases}$$

  Thus,

$$E[X_i] = pr\{\text{candidate } i \text{ is hired}\}$$

- Compute the probability that lines 5 and 6 of the `hireAssistant(n)` algorithm are executed

The Hiring Problem

- Pseudocode for Hire Assistant
  - Candidates for the job are numbered 1 through n
  - After interviewing candidate i, determine whether he/she is the best so far
  - Initialize with a dummy candidate 0, that is least qualified

```
hireAssistant(n)
1  best = 0
2  for i = 1 to n
3    interview candidate i
4    if candidate i is better than best
5      best = i
6      hire candidate i
```

Indicator Random Variables

- Candidate i is hired exactly when he/she is better than each of the previous 1 through $i - 1$ person
  - Since they arrive in random order, any one is equally likely to be the "best-qualified" so far
  - Candidate i has a probability of ${}^{1}/_{i}$ of being hired. Thus,

$$E[X_i] = pr\{\text{candidate } i \text{ is hired}\} = {}^{1}/_{i}$$

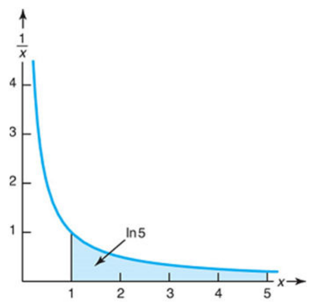
  and we can compute $E[X]$ as shown below:

$$E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i]$$
$$= \sum_{i=1}^{n} ({}^{1}/_{i})$$
$$= \ln n + O(1) \qquad \text{(why } \ln n\text{?)}$$

Mathematics – Natural Log

Natural Logarithm (ln) is log of base

$$e \approx 2.71828$$



$\ln x = \log_e x$: is the area under the curve $f(x) = \frac{1}{x}$ that lies between 1 and x

Indicator Random Variables

- So, even though n people are interviewed, only about $\ln n$ candidates get hired on average
- The algorithm `hireAssistant(n)` has an ***average-case hiring cost*** of
  $$O(c_h \ln n)$$
- This is significantly better than the worst-case hiring cost of
  $$O(c_h n)$$

Probabilistic Analysis

- Probabilistic analysis → distribution of inputs
- The algorithm is deterministic
  - For any particular input, the number of times a new assistant is hired is always the same
  - The number of times differs for different inputs, depending on the ranks of the various candidates
    - For rank list A1 = ⟨1, 2, 3, 4, 5, 6, 7, 8, 9, 10⟩ → hire 10 times
    - For rank list A2 = ⟨10, 9, 8, 7, 6, 5, 4, 3, 2, 1⟩ → hire only 1 time
    - For rank list A3 = ⟨**5**, 2, 1, **8**, 4, 7, **10**, 9, 3, 6⟩ → hire 3 times
  - Total cost depends on the number of hires
  - A1 is expensive, A2 is cheapest, and A3 is moderate

Randomized Algorithms

- Use randomized algorithm to ensure randomness (imposing the distribution)
  - Before running the algorithm, we randomly permute the candidates
  - Does not rely on the input distribution
  - The new applicants are still expect to be hired $\ln n$ times

Randomized Algorithms

- The algorithm is non-deterministic
  - Given the same input, like in A3 list, the result is different each time we run the algorithm
  - Each execution depends on the random choices made
  - Thus, no particular input elicits its worst-case behavior
  - Worst-case only happens when you get an "unlucky" permutation, which results in the A1 list

Randomized Algorithms

- Modified pseudocode for Hire Assistant
  - First randomize the list of applicants

```
randomizedHireAssistant(n)
randomly permute the list of candidates
best = 0
for i = 1 to n
  interview candidate i
  if candidate i is better than best
    best = i
    hire candidate i
```

  - The algorithm `randomizedHireAssistant(n)` has an ***expected hiring cost*** of

$$O(c_h \ln n)$$

Randomly Permuting Arrays

- How to randomly permute an array
- One common method – ***permute by sorting***
  - Assign each element $A[i]$ of the array a random priority $P[i]$
  - Sort the elements according to these priorities
    - Original array: A = ⟨1, 2, 3, 4⟩
    - Random priorities: P = ⟨8, 2, 12, 5⟩
    - Permuted array: B = ⟨2, 4, 1, 3⟩
  - The procedure is called permute by sorting

Randomly Permuting Arrays

- Pseudocode for `permuteBySorting`

```
permuteBySorting(A)
n = A.length
let P[1…n] be a new array
for i = 1 to n
  P[i] = random(1,n^3)
sort A, using P as sort keys
```

- This method produces a uniform random permutation
  - Equally likely to produce every permutation of the numbers 1 through n
  - The probability of obtaining identity permutation is $1/n!$

Randomly Permuting Arrays

- A better method – ***randomize in place***
  - Permute the given array in place (take $O(n)$ time)
  - In the i^{th} iteration, it chooses the element $A[i]$ randomly from among elements $A[i]$ through $A[n]$
  - Pseudocode for `randomizeInPlace`

```
randomizeInPlace(A)
  n = A.length
  for i = 1 to n
    swap A[i] with A[random(i,n)]
```

  - This method also computes a uniform random permutation

Randomly Permuting Arrays

- Recall that a ***k-permutation*** on a set of n elements is a non-repeating sequence containing k elements of the set

$$\frac{n!}{(n-k)!}$$

- Just prior to the i^{th} iteration of the `for` loop, for each possible $(i-1)$-permutation of the n elements, the subarray $A[1\ \dots\ i-1]$ contains this $(i-1)$-permutation with probability

$$\frac{(n-i+1)!}{n!}$$

- A randomized algorithm is often the simplest and most efficient way to solve a problem

Sample Problem

Prove the following statement on the previous slide:

Just prior to the i^{th} iteration of the `for` loop, for each possible $(i-1)$-permutation of the n elements, the subarray $A[1\ \dots\ i-1]$ contains this $(i-1)$-permutation with probability

$$\frac{(n-i+1)!}{n!}$$