11.4 Probabilistic algorithms
- take a radically different approach from deterministic algorithms explored so far
  - in some applications, probabilistic algorithms provide results that cannot be achieved deterministically

11.4.1 Numerical Probabilistic Algos
- calculate an approximate result for some mathematical problem
  - the longer the algorithm runs
    - the greater the precision
- e.g., throwing darts at a board to estimate pi
  - choose random (x,y) points in square
  - count how many land in circle (inscribed in square)
    - area of circle $\pi r^2$
    - area of square $(2r)^2 = 4r^2$
    - ratio of circle to square $\Rightarrow \pi / 4$
  - if number truly random, they will be evenly spaced across square
    - $\pi$ can estimated by $4 * \frac{c}{s}$
    - $c$ is number of darts that fall in circle
    - $s$ is number of darts through
  - $\Rightarrow$ more darts, closer to $\pi$

Monte Carlo integration
- for a continuous function $f$
  - area under curve $f$ is the integral of that function
  - some functions $\Rightarrow$ difficult or impossible to determine integral
    - but can approximate using Monte Carlo integration
  - randomly throw darts at 1 x 1 board with function drawn in
    - count how many land under curve
    - number below curve / num thrown $\Rightarrow$ approx area
    - more darts $\Rightarrow$ better precision

```
integrate(f, dartCount):
  // $f =$ function to integrate
  // $\text{dartCount} =$ number of darts to “throw”
  hits = 0
  for $i = 1 : \text{numDarts}$:
    $x = \text{random}(0,1)$
    $y = \text{random}(0,1)$
    if $y \leq f(x)$:
      hits++
  return hits / dartCount
```

Probabilistic Counting
- skipped
- same basic idea, applied to slightly different problem

11.4.2 Monte Carlo Algorithms
- will always give an answer,
  - probability that answer will be correct increases the longer the algo runs
  - can also return incorrect answer
  - such algos called $p$-correct when they return a correct answer with prob $p$ ($0.5 < p < 1$)
  - consistent: more than 1 correct answer for any given input
    - MC algorithm returns same correct answer each time.
- Two ways to improve MC algo:
  - 1) increase amount of time it runs
  - 2) call it multiple times
In this case, make several calls to algorithm
choose answer that appears most frequently
- improves algorithm..

**MC majority element**
- find the majority element in an array
  - majority element- if one element is stored in more than half of array locations
  - if solved via brute force, would take $O(N^2)$

**MC algo:**

```
Majority (list, N):
  //list of elements of size N
  choice = random(1, N)
  count = 0
  for i = 1 : N:
    if list[i] == list[choice]:
      count++
  if count > N/2:
    return true
  else:
    return false
```

- **Monte Carlo Prime Testing**
  - skipped.

**11.4.3 Las Vegas Algorithms**
- never return wrong answer
- but may return no answer
  - longer the algorithm runs, the higher the prob of success
- basic idea:
  - randomly make decision
  - check to see if successful answer
- can use in 8-queens problem:
  - place queens on board s.t. each new queen is placed randomly on next row
  - if cannot place queen on a row, just gives up and signal failure
  - (recursive version backs up and tries to fix)
  - LV version takes about 55 passes to solve
  - backtracking takes 100+ passes

**11.4.4 Sherwood Algorithms**
- always gives an answer
- answer always correct
- applied to algorithms where best, avg, and worst cases differ significantly
- **Introduce randomness** to help reduce gap b/t best and worst
  - E.g., how to choose pivot for quicksort
  - worst case: if list was already sorted AND pick smallest element
  - instead => pick random pivot element
    - (would not eliminate chance of worst case, but reduce its likelihood)
    - NOTE: will also reduce likelihood of best case (median)

- can apply to searching as well.
  - Sherwood “binary search”:
    - random pick location b/t start and end
    - (not necessarily start + end / 2)
    - sometimes better, sometimes worse
11.5 Dynamic Programming

- basic idea
  - reduce time of worst case
  - increase time of best case
  - (cannot be predetermined)

- basic idea: break problem into simpler subproblems
  - solve each subproblem only once
  - store the result (do not recalculate)

- four applications of dynamic programming:
  - 1) improve calculation efficiency of recursive algos
  - 2) deciding order of matrix must..
  - 3) shortest pairs for all nodes in graph
  - 4) approx solution to backpack problem

- E.g., #1
  - Fibonacci recursion
    - E.g., calculate 10th fib number: fibb(9) + fibb(8)
    - usually just “throw away” numbers calculated
      - traditional method:
        - calculate 8th fib # as part of determining 9th
        - but throw the numbers away
    - E.g., fibb(6) recursion tree:

      ![Recursion Tree for fibb(6)](image)

      - fibb(4) called 2x
      - fibb(3) called 3x

- for 10th fibb #,
  - fibb(3) would get called 21 times
- another method?
  - calculate from the bottom up

```python
def fibb(N):
    if N == 1 or N == 2:
        return 1
    else:
        val[1] = 1
        val[2] = 1
        for i = 3 : N:
            val[i] = val[i-1] + val[i-2]
```

- E.g., #1
  - Fibonacci recursion
    - E.g., calculate 10th fib number: fibb(9) + fibb(8)
    - usually just “throw away” numbers calculated
      - traditional method:
        - calculate 8th fib # as part of determining 9th
        - but throw the numbers away
    - E.g., fibb(6) recursion tree:

      ![Recursion Tree for fibb(6)](image)

      - fibb(4) called 2x
      - fibb(3) called 3x

- for 10th fibb #,
  - fibb(3) would get called 21 times
- another method?
  - calculate from the bottom up

```python
def fibb(N):
    if N == 1 or N == 2:
        return 1
    else:
        val[1] = 1
        val[2] = 1
        for i = 3 : N:
            val[i] = val[i-1] + val[i-2]
```
- return val[N]

- more complex example?
  - binomial coefficients
  - \( \binom{n}{k} \) (n choose k)

- 11.5.2 Dynamic Matrix Mult
  - skipped

- 11.5.3 All-pairs shortest Path algorithm
  - skipped

- 11.5.4 Dynamic programming for Backpack problem
  - skipped.