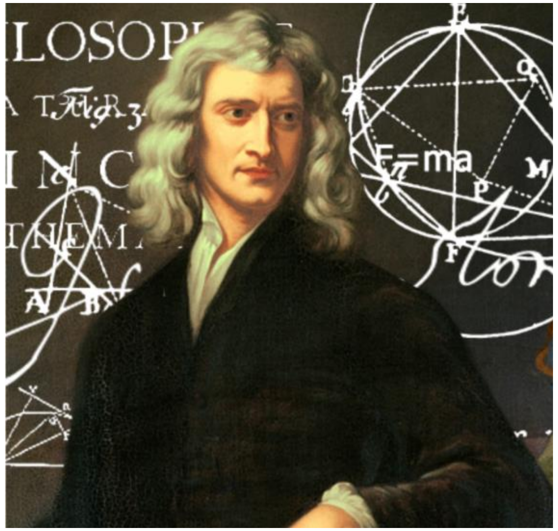
CS218: Algorithm design and analysis

# Randomized Algorithms and Average Analysis


Yan Gu

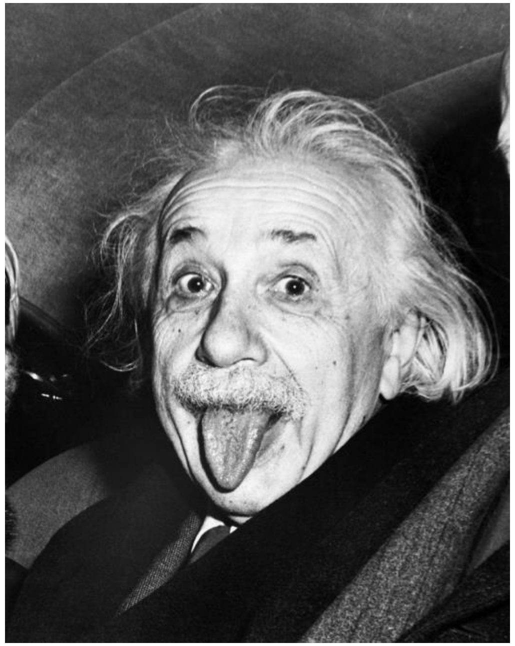
# Does the universe have <u>true randomness</u>?

- **Newtonian physics suggests that the universe evolves deterministically**




- **Quantum physics says otherwise**

# Does the universe have true randomness?



- God does not play dice with the world.

  - *Albert Einstein*

- Einstein, don't tell God what to do.

  - *Niels Bohr*

# Does the universe have true randomness?

- Even if it doesn't, we can still model our uncertainty about things using probability

- Randomness is an essential tool in modeling and analyzing nature

- It also plays a key role in computer science

# Randomness in computer science

- **Randomized algorithms**
  - Does randomness speed up computation?
- **Statistics via sampling**
  - e.g. election polls
- **Nash equilibrium in Game Theory**
  - Nash equilibrium always exists if players can have probabilistic strategies
- **Cryptography**
  - A secret is only as good as the entropy/uncertainty in it

# Randomness in computer science

- **Randomized models for deterministic objects**
  - e.g. the www graph
- **Quantum computing**
  - Randomness is inherent in quantum mechanics…
- **Machine learning theory**
  - Data follows some probability distribution
- **Coding Theory**
  - Encode data to be able to deal with random noise
- …

# Randomness and algorithms

- **How can randomness be used in computation?**
- **Where can it come into the picture?**

- **Given some algorithm that solves a problem…**
  - What if the input is chosen randomly?
  - What if the algorithm can make random choices?

# What is a randomized algorithm?

- **A randomized algorithm is an algorithm that is allowed to flip a coin**
- **(it can make decisions based on the output of the coin flip)**

- **In this course, we assume:**
  - A randomized algorithm is an algorithm that is allowed to call: `RandInt(n)`, which takes $O(1)$ time/work

# Randomness and algorithms

- **For a randomized algorithm, what should we measure:**
  - measure its correctness?
  - measure its running time?
- **If we require it to be**
  - always correct, **and**
  - always runs in time $O(T(n))$
- **then we have a deterministic alg. with time complexity $O(T(n))$**

# Randomness and algorithms

- **So for a randomized algorithm to be interesting:**
  - It is not correct all the time, **or**
  - It doesn't always run in time $O\big(T(n)\big)$

- **(It either gambles with correctness or running time.)**

# Types of randomized algorithms

- Given an array with $n$ elements ($n$ is even): A[1 … n]
- Half of the array contains 0s, the other half contains 1s
- Goal: Find an index that contains a 1

```
repeat:
    k = RandInt(n)
    if A[k] = 1, return k
```

Doesn't gamble with correctness

Gambles with run-time

```
repeat 300 times:
    k = RandInt(n)
    if A[k] = 1, return k
return "Failed"
```

Gambles with correctness

Doesn't gamble with run-time

# Monte Carlo algorithm

- **Gambles with correctness but not time**

```
repeat 300 times:
    k = RandInt(n)
    if A[k] = 1, return k
return "Failed"
```

$$\mathbf{Pr}[\text{failure}] = \frac{\mathbf{1}}{\mathbf{2^{300}}}$$

**Worst-case time complexity: $\boldsymbol{O}(\mathbf{1})$**

# Las Vegas algorithm

- **Gambles with time but not correctness**

```
repeat:
    k = RandInt(n)
    if A[k] = 1, return k
```

$$\mathbf{Pr}[\text{failure}] = \mathbf{0}$$

- **Worst-case time: cannot bound (can be big when super unlucky)**

- **Expected time: $O(1)$ (2 iterations)**

# Types of algorithms

- Given an array with $n$ elements ($n$ is even): A[1 … n]
- Half of the array contains 0s, the other half contains 1s
- Goal: Find an index that contains a 1

| | Correctness | Time complexity |
|---|---|---|
| Deterministic | Always | $\Theta(n)$ |
| Monte Carlo | With "high" probability | $O(1)$ |
| Las Vegas | Always | $O(1)$ expected |

# Analysis for
# Yan's simple hash table

# A short recall of hash table

- **Maintain a list of unordered elements, and support quick insert/delete and lookup**

# Simple uniform hashing strategy

- **For each element with key $x$, find a random position $h_1(x)$;**
  - if there's a collision, try another (i.e., $h_2(x)$)

- **Say insert key to be 5**
- **Then insert key to be 88**

- **What's the expected number of retries?**

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | - | - | 45 | - | 5 | - | - | 74 | - | 88 | 20 |

# Analyzing expected number of retries

- **Assuming at least half of the elements in the hash table are empty**
  - Recall this is the "load factor" of a hash table, and presumably should be less than ½

- **What's the probability that the first slot is occupied?**
- **What's the probability that the first two slots are occupied?**
- **What's the probability that the first $k$ slots are occupied?**

- **Total number of retries** $\leq 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = 2 = O(1)$

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | - | - | 45 | - | - | - | - | 74 | - | - | 20 |

# Conclusion for hash table analysis

- **Using this simple strategy, we can show that insert and lookup has constant ($O(1)$) cost**

- **We can show the same results for the more practical strategies**
  - Closed addressing (chaining): CLRS Section 11.2
  - Open addressing: CLRS Section 11.4

# Types of algorithms

| | Correctness | Time complexity |
|---|---|---|
| Deterministic | Always | Good |
| Monte Carlo | With good probability | Ideally even better |
| Las Vegas | Always | Better |

# The Average Analysis for Quicksort

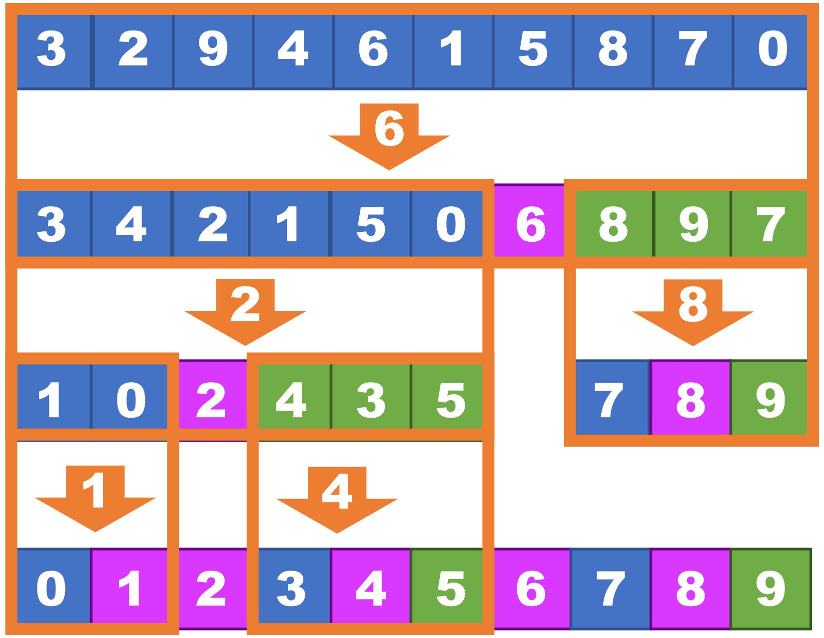
# An overview of quicksort

- **A randomized sorting algorithm based on divide-and-conquer**
- **Base case:**
  - If the subarray contains 1 element, return it directly
- **Divide:**
  - Find a random pivot $p$
  - Put all elements $< p$ on the left, call them $L$
  - Put all elements $> p$ on the right, call them $R$
- **Conquer**
  - Sort $\boldsymbol{L}$ and $\boldsymbol{R}$ recursively
- **Combine**
  - Return $\boldsymbol{L}, p, \boldsymbol{R}$

(assume no duplicates)

# An example of quicksort execution

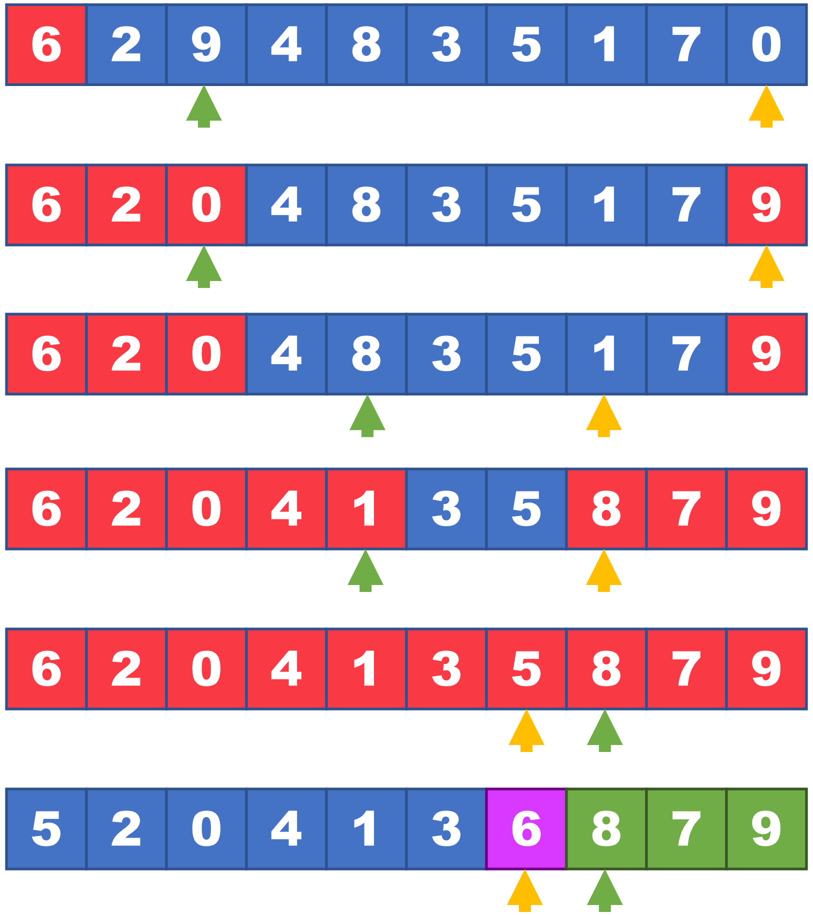
- Find a random pivot $x$ in the array
- Put all elements in $A$ that are smaller than $p$ on the left of $x$, and all elements in $A$ that are greater than $x$ on the right




The hardest part is in how to divide!

# Partition the array takes linear work

- **How to move elements around? (using 6 as a pivot)**

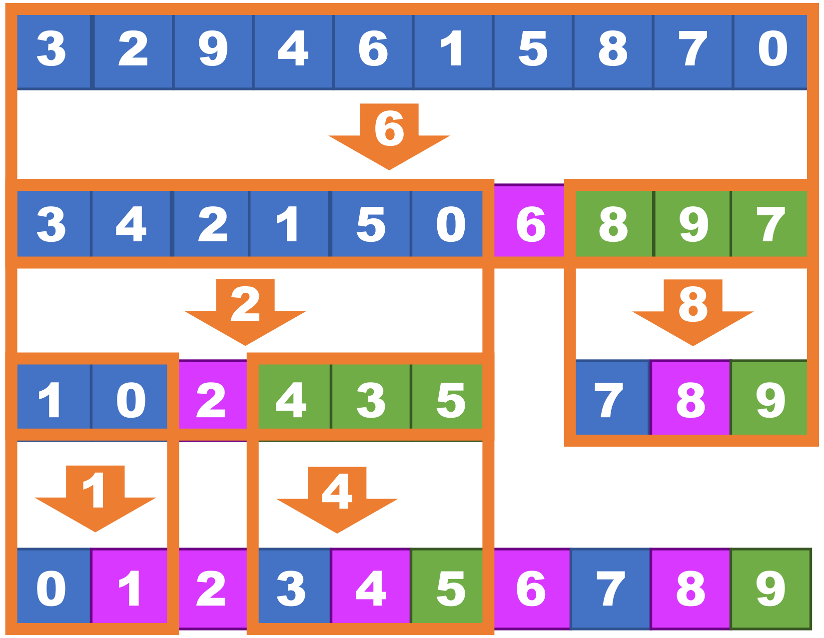



```
Partition(A, n, x) {
  i = 1; j = n-1;
  while (i < j) {
    while (A[i] < x) i++;
    while (A[j] > x) j--;
    if (i < j) swap A[i] and A[j];
    i++; j--;
  }
}
```

- **$O(n)$ cost to partition $n$ elements**
- **$O(1)$ cost per element!**

# An example of quicksort execution

- Find a random pivot $x$ in the array
- Put all elements in $A$ that are smaller than $p$ on the left of $x$, and all elements in $A$ that are greater than $x$ on the right




- $O(1)$ cost per element per level

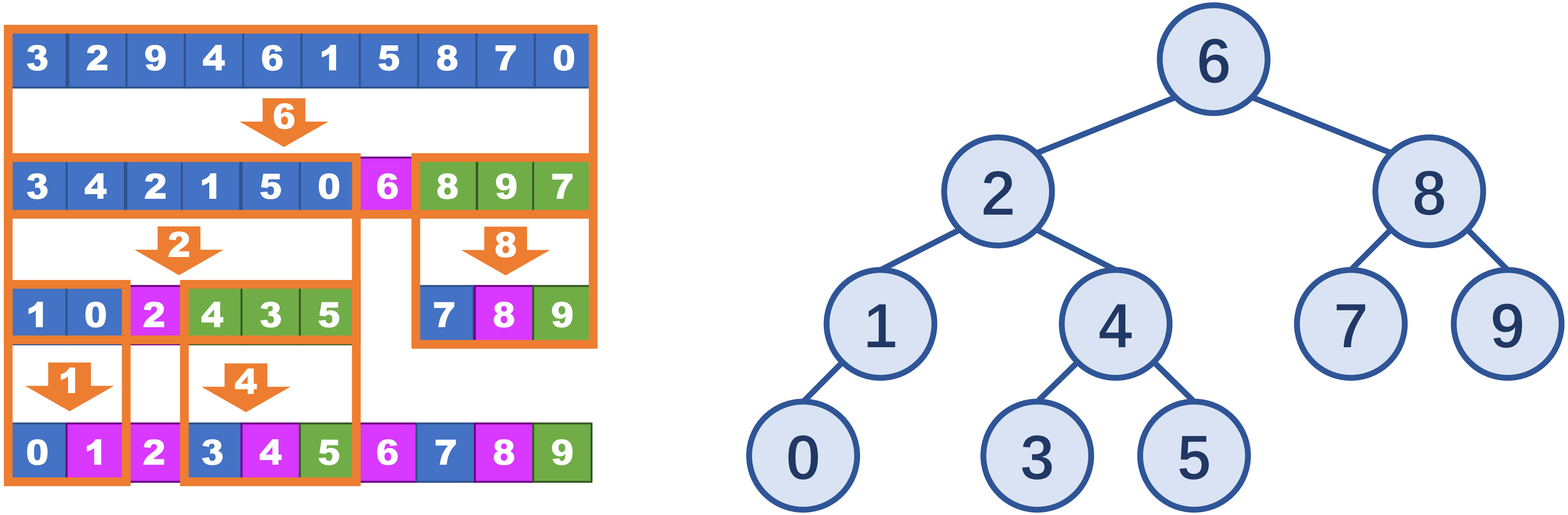
# Quicksort – cost analysis

- **We've already known that each element in one round requires $O(1)$ cost**
  - $O(n)$ for all elements in a round in total
- **How many rounds do we need?**
  - Since we are not directly dividing the array into halves, we cannot guarantee the size of problem shrink by half
- **But in the worst case, it is $O(n^2)$**
- **What is the average case?**

# Directly solving the recurrence

$$C(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(C(i) + C(n-i-1)) = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1}C(i)$$

# The equivalence of quicksort and randomized BST



# When does $i$ and $j$ compare?

- **Either $i$ or $j$ must have higher priority than all elements between $i$ and $j$**

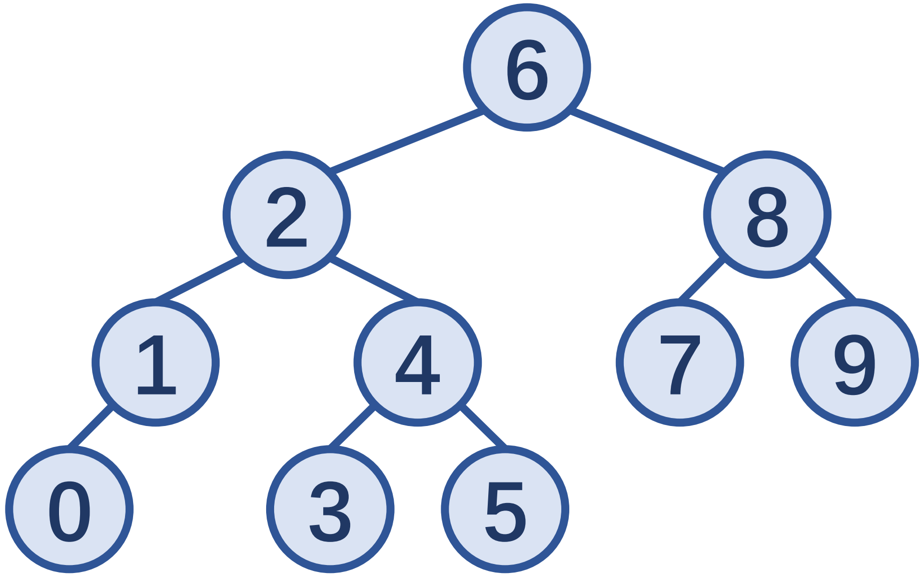
$$\Pr[t_{i,j}] \leq \frac{2}{j-i+1}$$

- **Total number of comparisons is:**

$$\sum_{i}\sum_{j>i}\Pr[t_{i,j}] = \sum_{i}\sum_{j>i}\frac{2}{j-i+1}$$

$$\leq \sum_{i}\left(2\cdot\ln(n+1)\right)$$

$$= O(n\log n)$$




| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 9 | 4 | 6 | 1 | 5 | 8 | 7 | 0 |

# Types of algorithms

| | Correctness | Time complexity |
|---|---|---|
| **Deterministic** | Always | Good |
| **Monte Carlo** | With good probability | Ideally even better |
| **Las Vegas** | Always | Better |

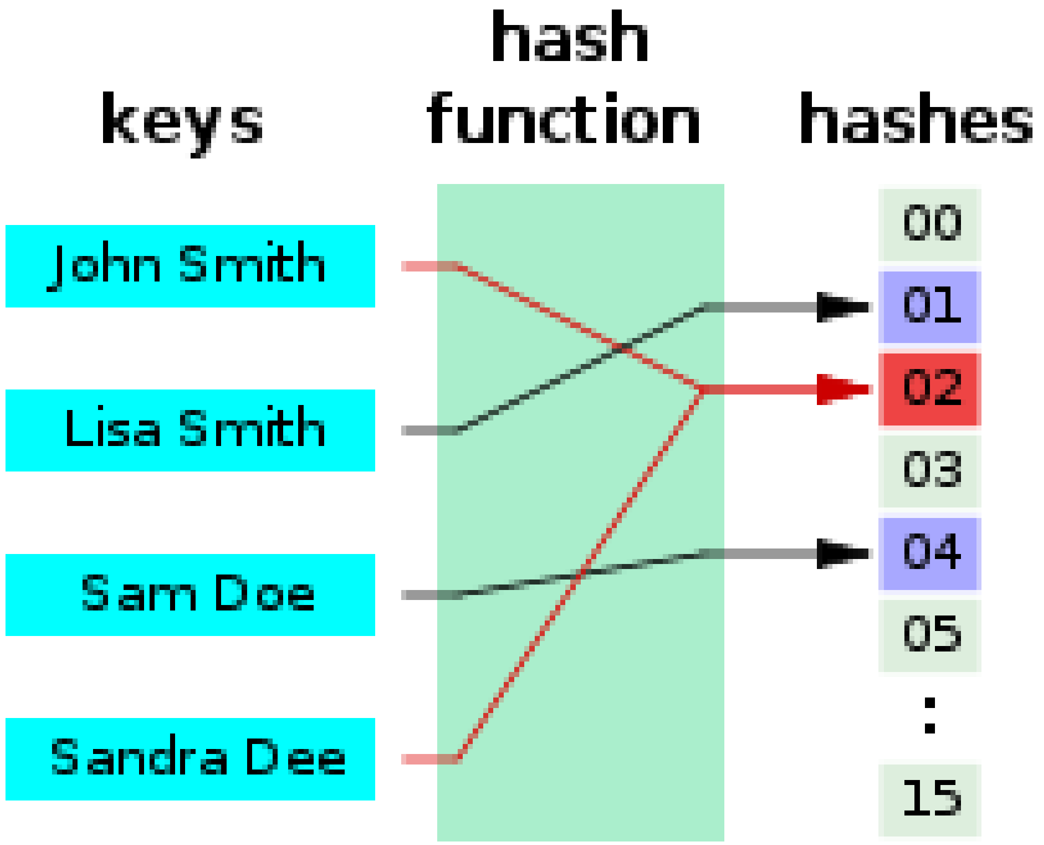
# Hashing and Rabin-Karp Algorithm

# Hash function

- **Maps arbitrary data to fixed-size values**
  - Usually integers
- **The same data are always mapped to the same value**
- **Different data are unlikely to be mapped to the same value**
  - Collision: two keys are hashed to the same hash value

# How to design a hash function

- **E.g., Strings -> integers**
- **How can we map complicated structs**
  - A pair: $(i, j)$ for $i, j$ in [1..100]
  - A triple $(i, j, k)$ for $i, j, k$ in [1..100]
  - A $2 \times 2$ matrix?
  - A string?



# Application: Rabin-Karp algorithm

- **Substring matching**
- **Given string $X[1..n]$ (text) and $Y[1..m]$ (pattern), we want to check if $Y$ is a substring in $X$**
  - $X = abcabababc, Y = caba$

- **The naïve solution cost $O(nm)$ time**

- **Knuth–Morris–Pratt algorithm (KMP) can solve this in $O(n)$ time**
  - Hard to understand ☹

# Let's try randomization!

- **To check if $Y$ appears in $X$, we just need to check all $X$'s substring and see if they are the same with $Y$**
  - Checking if $X[s..e] = Y$ takes $O(m)$ time

- **Let's use hashing!**
  - Check if the hash value of $X[s..e]$ equals to the hash value of Y

# Let's try randomization!

- **To check if $Y$ appears in $X$, we just need to check all $X$'s substring and see if they are the same with $Y$**
  - Checking if $X[s..e] = Y$ takes $O(m)$ time

- **Let's use hashing!**
  - For a string using characters a, b, c
  - $H_1$: using a=1, b=2, c=3. adding everything up

$$Y = bcab, H_1(Y) = 8$$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a | b | c | a | b | a | b | c |

| | | | |
|---|---|---|---|
| abca | 1+2+3+1=7 | abab | 1+2+1+2=6 |
| bcab | 2+3+1+2=8 | babc | 2+1+2+3=8 |
| caba | 3+1+2+1=7 | | |

# Let's try randomization!

$Y = bcab,\ H_1(Y) = 8$

- **How to compute the hash value quickly?**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a | b | c | a | b | a | b | c |

| | |
|---|---|
| abca | 1+2+3+1=7 |
| bcab | 2+3+1+2=8 |
| caba | 3+1+2+1=7 |
| abab | 1+2+1+2=6 |
| babc | 2+1+2+3=8 |

Hash value:

1+2+3+1=7

7-1+2=8

8-2+1=7

7-3+2=6

6-1+3=8

$O(n)$ time!

# Fewer collisions?

$Y = bcab,\ H_2(Y) = 2312$

- $H_2$: **treat a=1,b=2,c=3, use a decimal number**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a | b | c | a | b | a | b | c |

$O(n)$ time!

| | |
|---|---|
| abca | 1231 |
| bcab | 2312 |
| caba | 3121 |
| abab | 1212 |
| babc | 2123 |

Hash value:

1231

(1231-1000)*10+2=2312

(2312-2000)*10+1=3121

(3121-3000)*10+2=1212

(1212-2000)*10+3=2123

# A simple version

```
num (c) {return c-98;} // a=1, b=2, c=3

check_match (X, Y) {
  hy = 0;
  for (i = 1..m) hy = hy*10 + num(Y[i]); //compute hash value of Y
  hx = 0;
  for (i = 1..n) {
    if (i < m) hx = hx*10 + num(X[i]); // process the first (m-1) characters
    else {
       if (i==m) hx = hx*10 + num(X[i]);
       else hx = (hx - X[i-m+1] * pow(10, m-1))*10 + num(X[i]); // compute new hash value
       if (hx == hy)
          if (check(X[i-m+1 .. i], Y))  return true;
    }
 } }
```

# More characters?

- **What happens if we have 26 letters? Or even more?**
  - Use base-26 (or base-x with x > 26)

- **If $Y$ has 100 characters?**
- **Cannot use an integer to store?**

- **We can use $H_3(s) = H_2(s) \,\%\, p$ for some big prime $p$**
  - Still, the same strings will be mapped to the same value
  - Different strings are likely to be mapped to different values
- $(a + b) \,\%\, p = (a \,\%\, p) + (b \,\%\, p)$
- $(a \times b) \,\%\, p = (a \,\%\, p) \times (b \,\%\, p)$

# The cost of the algorithm?

- $O(n)$ **time to compute and compare all hash values**
- **But if two hash values are the same, we need to verify the strings are equal or not**
  - $O(m)$ time
- **In the worst case, all comparisons succeed, we need** $O(nm)$ **time**

- **However, the probability of two different strings is mapped to the same value is** $1/p$, **expected cost is** $O\left(\frac{mn}{p} + n\right)$
- **We can use a large** $p$ **to decrease the cost**
- **We can also use two independent hash functions to even lower the chance of collision**

# Summary for randomized algorithms

- **Randomization are powerful tools when designing algorithms**
  - Many algorithms we are using everyday are randomized (quicksort, hash table)
  - In fact, most of the algorithms that are widely used in practice are randomized

- **We talked about how to analyze the average running time of hash table and quicksort, and Rabin-Karp algorithm for pattern matching**

- **More randomized algorithms and tools to analyze randomized algorithms will be taught in CS 219**
  - Tail bounds (high probability analysis), union bounds
  - Graph min-cut, tree embeddings, low-diameter decomposition, graph partition, etc.
  - Many parallel algorithms are randomized (covered in CS 214)

# The next lecture…

- **Amortized analysis**

# Average grade so far

- Entrance Exam: 5.6/5
- HW1: 5.8/6
- HW2 Part I: 3.4/3.5
- HW2 Part II: 4.7/4.5

- Total: 19.5/19

- HW / Exams are 50% / 50%

# Score-to-grade mapping

- A+: >100%
- A: 90%
- A-: 85%
- B+/B/B-: 80%/75%/70%
- C/D: 65%/60%
- F: <60%

# About the midterm

- I graded some and it looks good

- The first three problems are mostly basic (multiple choice, basic DP, greedy proof), for 14 points

- The rest of the problems are challenging, but partial points can be given

# How the candy system works

- **You earn candies by solving bonus problems, actively participating in class discussions (up to 2 candies), online discussions (up to 2 candies), class participation, OH participation, and bonus presentations in the last week**

- **The number of candies will non-linearly map to up to 10 bonus points**
  - The first 5 candies are likely mapped to 4-5 points,
  - Then 2-4 candies per point all the way up

# About algorithms

- Design
- Analysis
- Implementation
- Exposition
- Presenting