QuickSort

- Yet another sorting algorithm!
- Usually faster than other algorithms on average, although worst-case is $O(n^2)$
- Divide-and-conquer:
  - **Divide**: Choose an element of the array for *pivot*. Divide the elements into three groups: those smaller than the pivot, those equal, and those larger.
  - **Conquer**: Recursively sort each group.
  - **Combine**: Concatenate the three sorted groups.
QuickSort running time

• Worse case:
  – Already sorted array (either increasing or decreasing)
  – $T(n) = T(n-1) + cn + d$
  – $T(n)$ is $O(n^2)$

• Average case: If the array is in random order, the pivot splits the array in roughly equal parts, so the average running time is $O(n \log n)$

• Advantage over mergeSort:
  – constant hidden in $O(n \log n)$ are smaller for quickSort. Thus it is faster by a constant factor
  – QuickSort is easy to do “in-place”
In-place algorithms

• An algorithm is *in-place* if it uses only a constant amount of memory in addition of that used to store the input.

• Importance of in-place sorting algorithms:
  – If the data set to sort barely fits into memory, we don't want an algorithm that uses twice that amount to sort the numbers.

• SelectionSort and InsertionSort are in-place: all we are doing is moving elements around the array.

• MergeSort is not in-place, because of the merge procedure, which requires a temporary array.

• QuickSort can easily be made in-place...
Algorithm partition(A, start, stop)
Input: An array A, indices start and stop.

pivot ← A[stop]
left ← start
right ← stop - 1

while left ≤ right do
    while left ≤ right and A[left] ≤ pivot) do left ← left + 1
    while (left ≤ right and A[right] ≥ pivot) do right ← right -1
    if (left < right ) then exchange A[left] ↔ A[right]

return left
Example of execution of partition

A = [ 6 3 7 3 2 5 7 5 ]  pivot = 5

A = [ 6 3 7 3 2 5 7 5 ]  swap 6, 2
A = [ 2 3 7 3 6 5 7 5 ]

A = [ 2 3 7 3 6 5 7 5 ]  swap 7, 3
A = [ 2 3 3 7 6 5 7 5 ]

A = [ 2 3 3 7 6 5 7 5 ]  swap 7, pivot
A = [ 2 3 3 5 6 5 7 7 ]
Example of execution of partition

A = [ 6 3 7 3 2 5 7 5 ]  \quad \text{pivot} = 5

A = [ 6 3 7 3 2 5 7 5 ]  \quad \text{swap 6, 2}
Example of execution of partition

\[ A = [\begin{array}{cccccccc} 6 & 3 & 7 & 3 & 2 & 5 & 7 & 5 \end{array} ] \quad \text{pivot} = 5 \]

\[ A = [\begin{array}{cccccccc} 6 & 3 & 7 & 3 & 2 & 5 & 7 & 5 \end{array} ] \quad \text{swap 6, 2} \]

\[ A = [\begin{array}{cccccccc} 2 & 3 & 7 & 3 & 6 & 5 & 7 & 5 \end{array} ] \]
Example of execution of partition

\[ A = [ 6 \ 3 \ 7 \ 3 \ 2 \ 5 \ 7 \ 5 ] \quad \text{pivot} = 5 \]

\[ A = [ 6 \ 3 \ 7 \ 3 \ 2 \ 5 \ 7 \ 5 ] \quad \text{swap 6, 2} \]

\[ A = [ 2 \ 3 \ 7 \ 3 \ 6 \ 5 \ 7 \ 5 ] \]

\[ A = [ 2 \ 3 \ 7 \ 3 \ 6 \ 5 \ 7 \ 5 ] \quad \text{swap 7,3} \]
Example of execution of partition

\[ A = [6 \ 3 \ 7 \ 3 \ 2 \ 5 \ 7 \ 5 ] \]  
pivot = 5

\[ A = [6 \ 3 \ 7 \ 3 \ 2 \ 5 \ 7 \ 5 ] \]  
swap 6, 2

\[ A = [2 \ 3 \ 7 \ 3 \ 6 \ 5 \ 7 \ 5 ] \]

\[ A = [2 \ 3 \ 7 \ 3 \ 6 \ 5 \ 7 \ 5 ] \]  
swap 7,3

\[ A = [2 \ 3 \ 3 \ 7 \ 6 \ 5 \ 7 \ 5 ] \]
Example of execution of partition

\[ A = [ \begin{array}{cccccccc} 6 & 3 & 7 & 3 & 2 & 5 & 7 & 5 \end{array} ] \quad \text{pivot} = 5 \]

\[ A = [ \begin{array}{cccccccc} 6 & 3 & 7 & 3 & 2 & 5 & 7 & 5 \end{array} ] \quad \text{swap 6, 2} \]

\[ A = [ \begin{array}{cccccccc} 2 & 3 & 7 & 3 & 6 & 5 & 7 & 5 \end{array} ] \]

\[ A = [ \begin{array}{cccccccc} 2 & 3 & 7 & 3 & 6 & 5 & 7 & 5 \end{array} ] \quad \text{swap 7,3} \]

\[ A = [ \begin{array}{cccccccc} 2 & 3 & 3 & 7 & 6 & 5 & 7 & 5 \end{array} ] \]

\[ A = [ \begin{array}{cccccccc} 2 & 3 & 3 & 7 & 6 & 5 & 7 & 5 \end{array} ] \quad \text{swap 7,pivot} \]

\[ A = [ \begin{array}{cccccccc} 2 & 3 & 3 & \underbrace{5}_{\leq 5} & 6 & 5 & 7 & \underbrace{7}_{\geq 5} \end{array} ] \]
In-place quickSort

Algorithm quickSort(A, start, stop)
Input: An array A to sort, indices start and stop
Output: A[start...stop] is sorted
if (start < stop) then
    pivot ← partition(A, start, stop)
    quickSort(A, start, pivot-1)
    quickSort(A, pivot+1, stop)
RandomizedQuicksort(A,start,stop) {
    if |A| = 0 return

    choose a pivot A[i] uniformly at random (start ≤ i ≤ stop)

    pivot ← partition(A,start,stop)

    RandomizedQuicksort(A, start, pivot-1)
    RandomizedQuicksort(A, pivot+1, stop)
}
Quicksort

Running time.
- [Best case.] Select the median element as the pivot: quicksort makes $\Theta(n \log n)$ comparisons.
- [Worst case.] Select the smallest (or largest) element as the pivot: quicksort makes $\Theta(n^2)$ comparisons.

Randomize. Protect against worst case by choosing pivot at random.

Intuition. If we always select a pivot that is bigger than 25% of the elements and smaller than 25% of the elements, then quicksort makes $\Theta(n \log n)$ comparisons.

Notation. Label elements so that $x_1 < x_2 < \ldots < x_n$. 
Theorem. Expected # of comparisons is $O(n \log n)$.

Theorem. [Knuth 1973] Stddev of number of comparisons is $\sim 0.65n$.

Ex. If $n = 1$ million, the probability that randomized quicksort takes less than $4n \ln n$ comparisons is at least $99.94\%$.

Chebyshev's inequality. $\Pr[|X - \mu| \geq k\delta] < 1 / k^2$. 

Randomized Quicksort: Expected Number of Comparisons
Strings and Pattern Matching

• Brute Force, Rabin-Karp, Knuth-Morris-Pratt

• Regular Expressions
String Searching

• The object of string searching is to find the location of a specific text pattern within a larger body of text (e.g., a sentence, a paragraph, a book, etc.).

• As with most algorithms, the main considerations for string searching are speed and efficiency.

• There are a number of string searching algorithms in existence today, but the three we shall review are Brute Force, Rabin-Karp, and Knuth-Morris-Pratt.
Brute Force

- The Brute Force algorithm compares the pattern to the text, one character at a time, until unmatching characters are found:

  - Compared characters are italicized.
  - Correct matches are in boldface type.

- The algorithm can be designed to stop on either the first occurrence of the pattern, or upon reaching the end of the text.
Brute Force

- The Brute Force algorithm compares the pattern to the text, one character at a time, until unmatching characters are found:

  | TWO ROADS DIVERGED IN A YELLOW WOOD |
  | ROADS                               |
  | TWO ROADS DIVERGED IN A YELLOW WOOD |
  | ROADS                               |
  | S DIVERGED IN A YELLOW WOOD         |
  | TWO ROADS DIVERGED IN A YELLOW WOOD |
  | ROADS                               |
  | TWO ROADS DIVERGED IN A YELLOW WOOD |

- Compared characters are italicized.
- Correct matches are in boldface type.

- The algorithm can be designed to stop on either the first occurrence of the pattern, or upon reaching the end of the text.
Brute Force

- The **Brute Force** algorithm compares the pattern to the text, one character at a time, until unmatching characters are found:

```
TWO ROADS DIVERGED IN A YELLOW WOOD
ROADS
TWO ROADS DIVERGED IN A YELLOW WOOD
ROADS
TWO ROADS DIVERGED IN A YELLOW WOOD
ROADS
TWO ROADS DIVERGED IN A YELLOW WOOD
ROADS
TWO ROADS DIVERGED IN A YELLOW WOOD
ROADS
```

- Compared characters are italicized.
- Correct matches are in boldface type.

- The algorithm can be designed to stop on either the first occurrence of the pattern, or upon reaching the end of the text.
Brute Force Pseudo-Code

- Here’s the pseudo-code
  
  ```pseudo
  do
    if (text letter == pattern letter)
      compare next letter of pattern to next letter of text
    else
      move pattern down text by one letter
  while (entire pattern found or end of text)
  ```

```plaintext
cool cat Rolo went over the fence
cat
cool cat Rolo went over the fence
cat
cool cat Rolo went over the fence
cat
cool cat Rolo went over the fence
cat
cool_cat Rolo went over the fence
cat
cool [cat] cat Rolo went over the fence
cat
```
**Brute Force-Complexity**

- Given a pattern M characters in length, and a text N characters in length...

- **Worst case**: compares pattern to each substring of text of length M. For example, M=5.

- This kind of case can occur for image data.

1) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAHH
   AAAAAH 5 comparisons made
2) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAHH
   AAAAAH 5 comparisons made
3) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAHH
   AAAAAH 5 comparisons made
4) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAHH
   AAAAAH 5 comparisons made
   ....
N) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAHH
   AAAAAH 5 comparisons made

- Total number of comparisons: M (N-M+1)
- Worst case time complexity: O(MN)
• Given a pattern $M$ characters in length, and a text $N$ characters in length...

• **Best case if pattern found:** Finds pattern in first $M$ positions of text. For example, $M=5$.

1) $\textcolor{red}{AAAAA} \textcolor{blue}{AAAAAAAAAAAAAAAAAAAAAAAAAAH}$
   
   $\textcolor{red}{AAAAA}$ 5 comparisons made

• Total number of comparisons: $M$

• Best case time complexity: $O(M)$
Brute Force-Complexity (cont.)

• Given a pattern M characters in length, and a text N characters in length...

• **Best case if pattern not found:** Always mismatch on first character. For example, M=5.

1) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   OOOOH  1 comparison made

2) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   OOOOH  1 comparison made

3) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   OOOOH  1 comparison made

4) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   OOOOH  1 comparison made

...  

N) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
   1 comparison made

• Total number of comparisons: N

• Best case time complexity: O(N)
Rabin-Karp

• The Rabin-Karp string searching algorithm calculates a **hash value** for the pattern, and for each M-character subsequence of text to be compared.

  • If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence.

  • If the hash values are equal, the algorithm will do a **Brute Force comparison** between the pattern and the M-character sequence.

  • In this way, there is only one comparison per text subsequence, and Brute Force is only needed when hash values match.

  • Perhaps an example will clarify some things...
Rabin-Karp Example

Hash value of “AAAAA” is 37
Hash value of “AAAAH” is 100

1) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAH
   37≠100     1 comparison made
2) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAH
   37≠100     1 comparison made
3) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAH
   37≠100     1 comparison made
4) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAH
   37≠100     1 comparison made
...

N) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
   AAAAH
   AAAAH
   5 comparisons made
   100=100
Rabin-Karp Algorithm

*pattern is M characters long*

hash\_p:=hash value of pattern  
hash\_t:=hash value of first M letters in body of text

\[\text{do}\]
\[\text{if} \ (\text{hash\_p} == \text{hash\_t})\]
\[\text{brute force comparison of pattern and selected section of text}\]
\[\text{hash\_t} = \text{hash value of next section of text, one character over}\]
\[\text{until} \ (\text{end of text or brute force comparison} == \text{true})\]
Rabin-Karp

• Common Rabin-Karp questions:

  “What is the hash function used to calculate values for character sequences?”

  “Isn’t it time consuming to hash every one of the M-character sequences in the text body?”

  “Is this going to be on the final?”

• To answer some of these questions, we’ll have to get mathematical.
Rabin-Karp Math

- Consider an $M$-character sequence as an $M$-digit number in base $b$, where $b$ is the number of letters in the alphabet. The text subsequence $t[i .. i+M-1]$ is mapped to the number

$$x(i) = t[i] \cdot b^{M-1} + t[i+1] \cdot b^{M-2} + ... + t[i+M-1]$$

- Furthermore, given $x(i)$ we can compute $x(i+1)$ for the next subsequence $t[i+1 .. i+M]$ in constant time, as follows:

$$x(i+1) = t[i+1] \cdot b^{M-1} + t[i+2] \cdot b^{M-2} + ... + t[i+M]$$

$$x(i+1) = x(i) \cdot b$$  \hspace{1cm} \text{Shift left one digit}  \\
- $t[i] \cdot b^M$  \hspace{1cm} \text{Subtract leftmost digit}  \\
+ $t[i+M]$  \hspace{1cm} \text{Add new rightmost digit}

- In this way, we never explicitly compute a new value. We simply adjust the existing value as we move over one character.
Rabin-Karp Math Example

• Let’s say that our alphabet consists of 10 letters.

• our alphabet = a, b, c, d, e, f, g, h, i, j

• Let’s say that “a” corresponds to 1, “b” corresponds to 2 and so on.

  The hash value for string “cah” would be ...

  \[3 \times 100 + 1 \times 10 + 8 \times 1 = 318\]
Rabin-Karp Mods

- If $M$ is large, then the resulting value ($\sim b^M$) will be enormous. For this reason, we hash the value by taking it $\text{mod}$ a prime number $q$.

- The $\text{mod}$ function ($\%$ in Java) is particularly useful in this case due to several of its inherent properties:
  - $[(x \mod q) + (y \mod q)] \mod q = (x+y) \mod q$
  - $(x \mod q) \mod q = x \mod q$
Rabin-Karp Mods

• For these reasons:

\[ h(i) = ( (t[i] \cdot b^{M-1} \mod q) + (t[i+1] \cdot b^{M-2} \mod q) + \ldots + (t[i+M-1] \mod q) ) \mod q \]

\[ h(i+1) = ( h(i) \cdot b \mod q \]
  Shift left one digit
  - \( t[i] \cdot b^M \mod q \)
  Subtract leftmost digit
  + \( t[i+M] \mod q \)
  Add new rightmost digit
  \mod q \]
Rabin-Karp Complexity

• If a sufficiently large prime number is used for the hash function, the hashed values of two different patterns will usually be distinct.

• If this is the case, searching takes O(N) time, where N is the number of characters in the larger body of text.

• It is always possible to construct a scenario with a worst case complexity of O(MN). This, however, is likely to happen only if the prime number used for hashing is small.
The Knuth-Morris-Pratt Algorithm

• The Knuth-Morris-Pratt (KMP) string searching algorithm differs from the brute-force algorithm by keeping track of information gained from previous comparisons.

• A failure function \( f \) is computed that indicates how much of the last comparison can be reused if it fails.

• Specifically, \( f \) is defined to be the longest prefix of the pattern \( P[0,..,j] \) that is also a suffix of \( P[1,..,j] \) - Note: not a suffix of \( P[0,..,j] \)
The Knuth-Morris-Pratt Algorithm

- Specifically, $f$ is defined to be the longest prefix of the pattern $P[0,\ldots,j]$ that is also a suffix of $P[1,\ldots,j]$.
  - Note: not a suffix of $P[0,\ldots,j]$

- Example:
  - value of the KMP failure function:

<table>
<thead>
<tr>
<th>$j$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[j]$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>$f(j)$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

- This shows how much of the beginning of the string matches up to the portion immediately preceding a failed comparison.
  - if the comparison fails at (4), we know the a,b in positions 2,3 is identical to positions 0,1
The KMP Algorithm (contd.)

• the KMP string matching algorithm: Pseudo-Code

Algorithm \texttt{KMPMatch}(T,P)

\textbf{Input:} Strings \(T\) (text) with \(n\) characters and \(P\) (pattern) with \(m\) characters.

\textbf{Output:} Starting index of the first substring of \(T\) matching \(P\), or an indication that \(P\) is not a substring of \(T\).

\[ f \leftarrow \text{KMPFailureFunction}(P) \quad \{ \text{build failure function} \} \]
\[ i \leftarrow 0 \]
\[ j \leftarrow 0 \]
while \(i < n\) do
  if \(P[j] = T[i]\) then
    if \(j = m - 1\) then
      return \(i - m - 1\) \{ a match \}
    \[ i \leftarrow i + 1 \]
    \[ j \leftarrow j + 1 \]
  else if \(j > 0\) then \{ no match, but we have advanced \}
    \[ j \leftarrow f(j-1) \quad \{ j indexes just after matching prefix in P \} \]
  else
    \[ i \leftarrow i + 1 \]
return “There is no substring of \(T\) matching \(P\)”
The KMP Algorithm (contd.)

- The KMP failure function: Pseudo-Code

**Algorithm** KMPFailureFunction($P$);

**Input:** String $P$ (pattern) with $m$ characters

**Output:** The failure function $f$ for $P$, which maps $j$ to the length of the longest prefix of $P$ that is a suffix of $P[1,\ldots,j]$

```
i ← 1
j ← 0
while $i \leq m-1$ do
    if $P[j] = P[i]$ then
        {we have matched $j + 1$ characters}
        $f(i) ← j + 1$
        $i ← i + 1$
        $j ← j + 1$
    else if $j > 0$ then
        {j indexes just after a prefix of $P$ that matches}
        $j ← f(j-1)$
    else  
        {there is no match}
        $f(i) ← 0$
        $i ← i + 1$
```
The KMP Algorithm (contd.)

- A graphical representation of the KMP string searching algorithm
The KMP Algorithm (contd.)

- Time Complexity Analysis
- define \( k = i - j \)

- In every iteration through the while loop, one of the three things happens.
  - 1) if \( T[i] = P[j] \), then \( i \) increases by 1, as does \( j \)
      \( k \) remains the same.
  - 2) if \( T[i] \neq P[j] \) and \( j > 0 \), then \( i \) does not change
      and \( k \) increases by at least 1, since \( k \) changes from \( i - j \) to \( i - f(j-1) \)
  - 3) if \( T[i] \neq P[j] \) and \( j = 0 \), then \( i \) increases by 1 and
      \( k \) increases by 1 since \( j \) remains the same.

- Thus, each time through the loop, either \( i \) or \( k \) increases by at least 1, so the greatest possible
  number of loops is \( 2n \)
- This of course assumes that \( f \) has already been
  computed.
- However, \( f \) is computed in much the same manner as
  \( \text{KMPMatch} \) so the time complexity argument is analogous. \( \text{KMPFailureFunction} \) is \( O(m) \)
- Total Time Complexity: \( O(n + m) \)
The KMP Algorithm (contd.)

- Thus, each time through the loop, either $i$ or $k$ increases by at least 1, so the greatest possible number of loops is $2n$

- This of course assumes that $f$ has already been computed.

- However, $f$ is computed in much the same manner as KMPMatch so the time complexity argument is analogous. KMPFailureFunction is $O(m)$

- Total Time Complexity: $O(n + m)$
Regular Expressions

- notation for describing a set of strings, possibly of infinite size
- \( \varepsilon \) denotes the empty string
- \( ab + c \) denotes the set \{ab, c\}
- \( a^* \) denotes the set \{\varepsilon, a, aa, aaa, ...\}

Examples
- \((a+b)^*\) all the strings from the alphabet \{a,b\}
- \(b^* (ab*a)^* b^*\) strings with an even number of a’s
- \((a+b)* \text{sun} (a+b)^*\) strings containing the pattern “sun”
- \((a+b)(a+b)(a+b)a\) 4-letter strings ending in a
Finite State Automaton

- “machine” for processing strings

Composition of FSA’s