13. Sorting and Searching
13. Searching and Sorting

- A typical client
- Binary search
- Insertion sort
- Mergesort
- Longest repeated substring
A typical client: Whitelist filter

A **blacklist** is a list of entities to be *rejected* for service.

A **whitelist** is a list of entities to be *accepted* for service.

**Whitelist filter**
- Read a list of strings from a *whitelist* file.
- Read strings from StdIn and write to StdOut only those in the whitelist.

**Example. Email spam filter (message contents omitted)**

<table>
<thead>
<tr>
<th>whitelist</th>
<th>StdIn</th>
<th>StdOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice@home</td>
<td>bob@office</td>
<td>bob@office</td>
</tr>
<tr>
<td>bob@office</td>
<td>carl@beach</td>
<td>mallory@spam</td>
</tr>
<tr>
<td>carl@beach</td>
<td>bob@office</td>
<td>dave@boat</td>
</tr>
<tr>
<td>dave@boat</td>
<td>eve@airport</td>
<td>alice@home</td>
</tr>
<tr>
<td></td>
<td>alice@home</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bob@office</td>
<td></td>
</tr>
<tr>
<td></td>
<td>marvin@spam</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bob@office</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dave@boat</td>
<td></td>
</tr>
<tr>
<td></td>
<td>alice@home</td>
<td></td>
</tr>
</tbody>
</table>

Examples: Overdrawn account Spammers

Examples: Account in good standing Friends and relatives
public class WhiteFilter
{
    public static int search(String key, String[] a)
    // Search method (stay tuned).
    public static void main(String[] args)
    {
        In in = new In(args[0]);
        String[] words = in.readAllStrings();
        while (!StdIn.isEmpty())
        {
            String key = StdIn.readString();
            if (search(key, words) != -1)
                StdOut.println(key);
        }
    }
}
Hey, Alice. I think I'm going to start an internet company.

Me too. I'm thinking about having 1 thousand customers next month and 1 million next year.

We're hoping to grow even faster than that.

Good luck! BTW, you're going to need a whitelist filter.

Yes, I know. I'm going to a hackathon to knock it out.

I'm going to take a few CS courses first.
Strawman implementation: Sequential search (first try)

Sequential search

- Check each array entry 0, 1, 2, 3, ...
  - for match with search string.
- If match found, return index of matching string.
- If not, return -1.

```java
public static int search(String key, String[] a)
{
    for (int i = 0; i < a.length; i++)
        if (a[i] == key) return i;
    return -1;
}
```

Compares references, not strings!
Strawman implementation: Sequential search

Sequential search
• Check each array entry 0, 1, 2, 3, ...
  for match with search string.
• If match found, return index of matching string.
• If not, return -1.

```java
public static int search(String key, String[] a)
{
    for (int i = 0; i < a.length; i++)
        if (a[i].compareTo(key) == 0) return i;
    return -1;
}
```

<table>
<thead>
<tr>
<th>i</th>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>alice</td>
</tr>
<tr>
<td>1</td>
<td>bob</td>
</tr>
<tr>
<td>2</td>
<td>carlos</td>
</tr>
<tr>
<td>3</td>
<td>carol</td>
</tr>
<tr>
<td>4</td>
<td>craig</td>
</tr>
<tr>
<td>5</td>
<td>dave</td>
</tr>
<tr>
<td>6</td>
<td>erin</td>
</tr>
<tr>
<td>7</td>
<td>eve</td>
</tr>
<tr>
<td>8</td>
<td>frank</td>
</tr>
<tr>
<td>9</td>
<td>mallory</td>
</tr>
<tr>
<td>10</td>
<td>oscar</td>
</tr>
<tr>
<td>11</td>
<td>peggy</td>
</tr>
<tr>
<td>12</td>
<td>trent</td>
</tr>
<tr>
<td>13</td>
<td>walter</td>
</tr>
<tr>
<td>14</td>
<td>wendy</td>
</tr>
</tbody>
</table>

Still, this was even easier than I thought!
Mathematical analysis of whitelist filter using sequential search

Model
- $N$ strings on the whitelist.
- $cN$ transactions for constant $c$.
- String length not long.

Analysis
- A random search *hit* checks *about half* of the $N$ strings on the whitelist, on average.
- A random search *miss* checks *all* of the $N$ strings on the whitelist, on average.
- Expected order of growth of running time: $N^2$. 

<table>
<thead>
<tr>
<th>whitelist</th>
<th>transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dobqi</td>
<td>xwnzb</td>
</tr>
<tr>
<td>xwnzb</td>
<td>lnuqv</td>
</tr>
<tr>
<td>dqwak</td>
<td>czpwx</td>
</tr>
<tr>
<td>lnuqv</td>
<td>czpwx</td>
</tr>
<tr>
<td>czpwx</td>
<td></td>
</tr>
<tr>
<td>bshla</td>
<td>dqwak</td>
</tr>
<tr>
<td>idhld</td>
<td>dobqi</td>
</tr>
<tr>
<td>utfyw</td>
<td>dobqi</td>
</tr>
<tr>
<td>hafah</td>
<td>dobqi</td>
</tr>
<tr>
<td>tsirv</td>
<td>tsirv</td>
</tr>
<tr>
<td></td>
<td>dqwak</td>
</tr>
<tr>
<td></td>
<td>dobqi</td>
</tr>
<tr>
<td></td>
<td>idhld</td>
</tr>
<tr>
<td></td>
<td>dqwak</td>
</tr>
<tr>
<td></td>
<td>dobqi</td>
</tr>
<tr>
<td></td>
<td>lnuqv</td>
</tr>
<tr>
<td></td>
<td>xwnzb</td>
</tr>
<tr>
<td></td>
<td>idhld</td>
</tr>
<tr>
<td></td>
<td>bshla</td>
</tr>
<tr>
<td></td>
<td>xwnzb</td>
</tr>
</tbody>
</table>
Random representative inputs for searching and sorting

Generate N random strings of length L from a given alphabet

```java
public class Generator {
    public static String randomString(int L, String alpha) {
        char[] a = new char[L];
        for (int i = 0; i < L; i++) {
            int t = StdRandom.uniform(alpha.length());
            a[i] = alpha.charAt(t);
        }
        return new String(a);
    }
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        int L = Integer.parseInt(args[1]);
        String alpha = args[2];
        for (int i = 0; i < N; i++)
            StdOut.println(randomString(L, alpha));
    }
}
```

Examples:
- `% java Generator 10 3 abc bab bab bbb cac aba abb bab ccb cbc bab`
- `% java Generator 15 8 0123456789 62855405 83179069 79061047 27258805 54441080 76592141 95956542 19442316 75032539 10528640 42496398 34226197 10320073 80072566 87979201`
Empirical tests of sequential search

<table>
<thead>
<tr>
<th>$N$</th>
<th>$T_N$ (seconds)</th>
<th>$T_N/T_{N/2}$</th>
<th>transactions per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>3</td>
<td></td>
<td>3,333</td>
</tr>
<tr>
<td>20,000</td>
<td>9</td>
<td></td>
<td>2,222</td>
</tr>
<tr>
<td>40,000</td>
<td>35</td>
<td>3.9</td>
<td>1,143</td>
</tr>
<tr>
<td>80,000</td>
<td>149</td>
<td>4.3</td>
<td>536</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.28 million</td>
<td>38,500</td>
<td>4</td>
<td>34</td>
</tr>
</tbody>
</table>

Whitelist filter scenario
- Whitelist of size $N$.
- $10N$ transactions.

Hypothesis. Order of growth is $N^2$. Does NOT scale.

... = 10 a-z | java TestSS
a-z = abcdefghijklmnopqrstuvwxyz

1.28 million transactions at a rate of 34 per second and dropping

more than 10.5 hours

% java Generator 10000 ...
3 seconds
% java Generator 20000 ...
9 seconds
% java Generator 40000 ...
35 seconds
% java Generator 80000 ...
149 seconds

Hmmmm. That doesn't seem too good.
13. Sorting and Searching

- A typical client
- Binary search
- Insertion sort
- Mergesort
- Longest repeated substring
Binary search

- Keep the array in **sorted order** (stay tuned).
- Examine the middle key.
- If it matches, return its index.
- If it is larger, search the half with lower indices.
- If it is smaller, search the half with upper indices.

```
<table>
<thead>
<tr>
<th>i</th>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>alice</td>
</tr>
<tr>
<td>1</td>
<td>bob</td>
</tr>
<tr>
<td>2</td>
<td>carlos</td>
</tr>
<tr>
<td>3</td>
<td>carol</td>
</tr>
<tr>
<td>4</td>
<td>craig</td>
</tr>
<tr>
<td>5</td>
<td>dave</td>
</tr>
<tr>
<td>6</td>
<td>erin</td>
</tr>
<tr>
<td>7</td>
<td>eve</td>
</tr>
<tr>
<td>8</td>
<td>frank</td>
</tr>
<tr>
<td>9</td>
<td>mallory</td>
</tr>
<tr>
<td>10</td>
<td>oscar</td>
</tr>
<tr>
<td>11</td>
<td>peggy</td>
</tr>
<tr>
<td>12</td>
<td>trent</td>
</tr>
<tr>
<td>13</td>
<td>walter</td>
</tr>
<tr>
<td>14</td>
<td>wendy</td>
</tr>
</tbody>
</table>
```

Match found. Return 10
**Binary search arithmetic**

**Notation.** $a[lo,hi)$ means $a[lo], a[lo+1] ... a[hi-1]$ (does not include $a[hi]$).

Search in $a[lo,hi)$

- $lo \rightarrow$
- $hi \rightarrow$

$mid = lo + (hi-lo)/2$

- $lo \rightarrow$
- $mid \rightarrow$
- $hi \rightarrow$

Lower half: $a[lo,mid)$

- $lo \rightarrow$
- $mid \rightarrow$
- $hi \rightarrow$

Upper half: $a[mid+1,hi)$

- $lo \rightarrow$
- $mid \rightarrow$
- $mid+1 \rightarrow$
- $hi \rightarrow$

Tricky! Needs study...
public static int search(String key, String[] a)
{
    return search(key, a, 0, a.length);
}

public static int search(String key, String[] a, int lo, int hi)
{
    if (hi <= lo) return -1;
    int mid = lo + (hi - lo) / 2;
    int cmp = a[mid].compareTo(key);
    if (cmp > 0) return search(key, a, lo, mid);
    else if (cmp < 0) return search(key, a, mid+1, hi);
    else           return mid;
}
Recursion trace for binary search

```java
public static int search(String key, String[] a)
{
    return search(key, a, 0, a.length);
}

public static int search(String key, String[] a, int lo, int hi)
{
    if (hi <= lo) return -1;
    int mid = lo + (hi - lo) / 2;
    int cmp = a[mid].compareTo(key);
    if (cmp > 0) return search(key, a, lo, mid);
    else if (cmp < 0) return search(key, a, mid+1, hi);
    else return mid;
}
```

```
search("oscar")
    return 10
    
search("oscar", a, 0, 15)
    mid = 7;
    > "eve"
    return 10

search("oscar", a, 8, 15)
    mid = 11;
    < "peggy"
    return 10

search("oscar", a, 8, 11)
    mid = 9;
    > "mallory"
    return 10

search("oscar", a, 10, 11)
    mid = 10;
    == "oscar"
    return 10;
```
Mathematical analysis of binary search

**Exact analysis for search miss for** $N = 2^n - 1$

- Note that $n = \log(N+1) \sim \log N$.
- Subarray size for 1st call is $2^n - 1$.
- Subarray size for 2nd call is $2^{n-1} - 1$.
- Subarray size for 3rd call is $2^{n-2} - 1$.
- ...
- Subarray size for $n$th call is 1.
- Total # compares (one per call): $n \sim \log N$.

**Proposition.** Binary search uses $\sim \log N$ compares for a search miss.

**Proof.** An (easy) exercise in discrete math.

**Proposition.** Binary search uses $\sim \log N$ compares for a random search hit.

**Proof.** A slightly more difficult exercise in discrete math.
Empirical tests of binary search

Whitelist filter scenario
- Whitelist of size $N$.
- $10N$ transactions.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$T_N$ (seconds)</th>
<th>$T_N/T_{N/2}$</th>
<th>transactions per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200,000</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>400,000</td>
<td>6</td>
<td>2</td>
<td>67,000</td>
</tr>
<tr>
<td>800,000</td>
<td>14</td>
<td>2.35</td>
<td>57,000</td>
</tr>
<tr>
<td>1,600,000</td>
<td>33</td>
<td>2.33</td>
<td>48,000</td>
</tr>
<tr>
<td>10.28 million</td>
<td>264</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

% java Generator 100000 ... 1 seconds
% java Generator 200000 ... 3 seconds
% java Generator 400000 ... 6 seconds
% java Generator 800000 ... 14 seconds
% java Generator 1600000 ... 33 seconds

... = 10 a-z | java TestBS
a-z = abcdefghijklmnopqrstuvwxyz

Conforms hypothesis that order of growth is $N \log N$.

Will scale.
13. Sorting and Searching

- A typical client
- Binary search
- Insertion sort
- Mergesort
- Longest repeated substring
Sorting: Rearrange N items to put them in ascending order

Applications
• Binary search
• Statistics
• Databases
• Data compression
• Bioinformatics
• Computer graphics
• Scientific computing
• ...
• [Too numerous to list]
TEQ 0 on sorting

Q. What's the most efficient way to sort 1 million 32-bit integers?
### Insertion sort algorithm

**Insertion sort**

- Move down through the array.
- Each item *bubbles up* above the larger ones above it.
- Everything above the current item is in order.
- Everything below the current item is untouched.

Like bubble sort, but not bubble sort. We don't teach bubble sort any more because this is simpler and faster.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>wendy</td>
</tr>
<tr>
<td>1</td>
<td>alice</td>
</tr>
<tr>
<td>2</td>
<td>dave</td>
</tr>
<tr>
<td>3</td>
<td>walter</td>
</tr>
<tr>
<td>4</td>
<td>carlos</td>
</tr>
<tr>
<td>5</td>
<td>carol</td>
</tr>
<tr>
<td>6</td>
<td>erin</td>
</tr>
<tr>
<td>7</td>
<td>oscar</td>
</tr>
<tr>
<td>8</td>
<td>peggy</td>
</tr>
<tr>
<td>9</td>
<td>trudy</td>
</tr>
<tr>
<td>10</td>
<td>eve</td>
</tr>
<tr>
<td>11</td>
<td>trent</td>
</tr>
<tr>
<td>12</td>
<td>bob</td>
</tr>
<tr>
<td>13</td>
<td>craig</td>
</tr>
<tr>
<td>14</td>
<td>frank</td>
</tr>
<tr>
<td>15</td>
<td>victor</td>
</tr>
</tbody>
</table>
## Insertion sort trace

<table>
<thead>
<tr>
<th></th>
<th>wendy</th>
<th>alice</th>
<th>alice</th>
<th>alice</th>
<th>alice</th>
<th>alice</th>
<th>alice</th>
<th>alice</th>
<th>alice</th>
<th>alice</th>
<th>alice</th>
<th>alice</th>
<th>alice</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>alice</td>
<td>wendy</td>
<td>dave</td>
<td>dave</td>
<td>carlos</td>
<td>carlos</td>
<td>carlos</td>
<td>carlos</td>
<td>carlos</td>
<td>carlos</td>
<td>carlos</td>
<td>carlos</td>
<td>bob</td>
</tr>
<tr>
<td>2</td>
<td>dave</td>
<td>dave</td>
<td>wendy</td>
<td>walter</td>
<td>dave</td>
<td>carol</td>
<td>carol</td>
<td>carol</td>
<td>carol</td>
<td>carol</td>
<td>carol</td>
<td>carol</td>
<td>carlos</td>
</tr>
<tr>
<td>3</td>
<td>walter</td>
<td>walter</td>
<td>wendy</td>
<td>walter</td>
<td>dave</td>
<td>dave</td>
<td>dave</td>
<td>dave</td>
<td>dave</td>
<td>dave</td>
<td>dave</td>
<td>carol</td>
<td>carol</td>
</tr>
<tr>
<td>4</td>
<td>carlos</td>
<td>carlos</td>
<td>carlos</td>
<td>carlos</td>
<td>wendy</td>
<td>carol</td>
<td>walter</td>
<td>carol</td>
<td>carol</td>
<td>carol</td>
<td>carol</td>
<td>carol</td>
<td>dave</td>
</tr>
<tr>
<td>5</td>
<td>carol</td>
<td>carol</td>
<td>carol</td>
<td>carol</td>
<td>wendy</td>
<td>carol</td>
<td>walter</td>
<td>carol</td>
<td>carol</td>
<td>carol</td>
<td>carol</td>
<td>carol</td>
<td>craig</td>
</tr>
<tr>
<td>6</td>
<td>erin</td>
<td>erin</td>
<td>erin</td>
<td>erin</td>
<td>erin</td>
<td>erin</td>
<td>erin</td>
<td>erin</td>
<td>erin</td>
<td>erin</td>
<td>dave</td>
<td>dave</td>
<td>dave</td>
</tr>
<tr>
<td>7</td>
<td>oscar</td>
<td>oscar</td>
<td>oscar</td>
<td>oscar</td>
<td>oscar</td>
<td>oscar</td>
<td>oscar</td>
<td>oscar</td>
<td>oscar</td>
<td>oscar</td>
<td>oscar</td>
<td>oscar</td>
<td>eve</td>
</tr>
<tr>
<td>8</td>
<td>peggy</td>
<td>peggy</td>
<td>peggy</td>
<td>peggy</td>
<td>peggy</td>
<td>peggy</td>
<td>peggy</td>
<td>peggy</td>
<td>peggy</td>
<td>wendy</td>
<td>walter</td>
<td>trudy</td>
<td>peggy</td>
</tr>
<tr>
<td>9</td>
<td>trudy</td>
<td>trudy</td>
<td>trudy</td>
<td>trudy</td>
<td>trudy</td>
<td>trudy</td>
<td>trudy</td>
<td>trudy</td>
<td>wendy</td>
<td>walter</td>
<td>trudy</td>
<td>trudy</td>
<td>trent</td>
</tr>
<tr>
<td>10</td>
<td>eve</td>
<td>eve</td>
<td>eve</td>
<td>eve</td>
<td>eve</td>
<td>eve</td>
<td>eve</td>
<td>eve</td>
<td>eve</td>
<td>eve</td>
<td>wendy</td>
<td>walter</td>
<td>trudy</td>
</tr>
<tr>
<td>11</td>
<td>trent</td>
<td>trent</td>
<td>trent</td>
<td>trent</td>
<td>trent</td>
<td>trent</td>
<td>trent</td>
<td>trent</td>
<td>wendy</td>
<td>walter</td>
<td>treudy</td>
<td>trent</td>
<td>trent</td>
</tr>
<tr>
<td>12</td>
<td>bob</td>
<td>bob</td>
<td>bob</td>
<td>bob</td>
<td>bob</td>
<td>bob</td>
<td>bob</td>
<td>bob</td>
<td>bob</td>
<td>bob</td>
<td>bob</td>
<td>wendy</td>
<td>walter</td>
</tr>
<tr>
<td>13</td>
<td>craig</td>
<td>craig</td>
<td>craig</td>
<td>craig</td>
<td>craig</td>
<td>craig</td>
<td>craig</td>
<td>craig</td>
<td>craig</td>
<td>craig</td>
<td>craig</td>
<td>craig</td>
<td>wendy</td>
</tr>
<tr>
<td>14</td>
<td>frank</td>
<td>frank</td>
<td>frank</td>
<td>frank</td>
<td>frank</td>
<td>frank</td>
<td>frank</td>
<td>frank</td>
<td>frank</td>
<td>frank</td>
<td>frank</td>
<td>frank</td>
<td>walter</td>
</tr>
<tr>
<td>15</td>
<td>victor</td>
<td>victor</td>
<td>victor</td>
<td>victor</td>
<td>victor</td>
<td>victor</td>
<td>victor</td>
<td>victor</td>
<td>victor</td>
<td>victor</td>
<td>victor</td>
<td>victor</td>
<td>wendy</td>
</tr>
</tbody>
</table>
Insertion sort: Java implementation

```java
public class Insertion {
    public static void sort(String[] a) {
        int N = a.length;
        for (int i = 1; i < N; i++)
            for (int j = i; j > 0; j--)
                if (a[j-1].compareTo(a[j]) > 0)
                    exch(a, j-1, j);
            else break;
    }

    private static void exch(String[] a, int i, int j) {
        String t = a[i]; a[i] = a[j]; a[j] = t;
    }

    public static void main(String[] args) {
        String[] a = StdIn.readStringStrings();
        sort(a);
        for (int i = 0; i < a.length; i++)
            StdOut.print(a[i]);
    }
}
```
Empirical tests of insertion sort

Sort random strings
- Array of length $N$.
- 10-character strings.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$T_N$ (seconds)</th>
<th>$T_N/T_{N/2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>20,000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>40,000</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>80,000</td>
<td>35</td>
<td>9</td>
</tr>
<tr>
<td>160,000</td>
<td>225</td>
<td>6.4</td>
</tr>
<tr>
<td>320,000</td>
<td>1019</td>
<td>4.5</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.28 million</td>
<td>14400</td>
<td>4</td>
</tr>
</tbody>
</table>

Confirms hypothesis that order of growth is $N^2$.

% java Generator 20000 ... 1 seconds
% java Generator 40000 ... 4 seconds
% java Generator 80000 ... 35 seconds
% java Generator 160000 ... 225 seconds
% java Generator 320000 ... 1019 seconds

... = 10 a-z | java Insertion a-z = abcdefghijklmnopqrstuvwxyz

And $4 \times 64/24 = 10+$ days to sort 10 million? Sounds bad.

Do you have anything better?

will NOT scale
A rule of thumb

**Moore's law.** The number of transistors in an integrated circuit doubles about every 2 years.

**Implications**
- Memory size doubles every two years.
- Processor speed doubles every two years.

**Sedgewick's rule of thumb.**
It takes *seconds* to access every word in a computer.

<table>
<thead>
<tr>
<th>computer</th>
<th>instructions per second</th>
<th>words of memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDP-9</td>
<td>tens of thousands</td>
<td>tens of thousands</td>
</tr>
<tr>
<td>VAX 11-780</td>
<td>millions</td>
<td>millions</td>
</tr>
<tr>
<td>CRAY 1</td>
<td>tens of millions</td>
<td>tens of millions</td>
</tr>
<tr>
<td>MacBook Air</td>
<td>billions</td>
<td>billions</td>
</tr>
</tbody>
</table>
**Scalability**

An algorithm *scales* if its running time doubles when the problem size doubles.

**2x faster computer with 2x memory using an alg that scales?**
- Can solve problems we're solving now in half the time.
- Can solve a 2x-sized problem in the *same* time it took to solve an x-sized problem.
- Progress.

<table>
<thead>
<tr>
<th>order of growth</th>
<th>scales?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>✓</td>
</tr>
<tr>
<td>$N \log N$</td>
<td>✓</td>
</tr>
<tr>
<td>$N^2$</td>
<td>✗</td>
</tr>
<tr>
<td>$N^3$</td>
<td>✗</td>
</tr>
</tbody>
</table>

**2x faster computer with 2x memory using quadratic alg?**
- Can solve problems we're solving now in half the time.
- Takes *twice* as long solve a 2x-sized problem as it took to solve an x-sized problem.
- Frustration.

Bottom line. Need **algorithms that scale** to keep pace with Moore's law.
13. Sorting and Searching

• A typical client
• Binary search
• Insertion sort
• Mergesort
• Longest repeated substring
### Mergesort algorithm

#### Merge sort
- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.

<table>
<thead>
<tr>
<th>Divide</th>
<th>Sort halves</th>
<th>Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>wendy</td>
<td>alice</td>
<td></td>
</tr>
<tr>
<td>alice</td>
<td>carlos</td>
<td></td>
</tr>
<tr>
<td>dave</td>
<td>carol</td>
<td></td>
</tr>
<tr>
<td>walter</td>
<td>dave</td>
<td></td>
</tr>
<tr>
<td>carlos</td>
<td>erin</td>
<td></td>
</tr>
<tr>
<td>carol</td>
<td>oscar</td>
<td></td>
</tr>
<tr>
<td>erin</td>
<td>walter</td>
<td></td>
</tr>
<tr>
<td>oscar</td>
<td>wendy</td>
<td></td>
</tr>
<tr>
<td>peggy</td>
<td>bob</td>
<td></td>
</tr>
<tr>
<td>trudy</td>
<td>craig</td>
<td></td>
</tr>
<tr>
<td>eve</td>
<td>eve</td>
<td></td>
</tr>
<tr>
<td>trent</td>
<td>frank</td>
<td></td>
</tr>
<tr>
<td>bob</td>
<td>peggy</td>
<td></td>
</tr>
<tr>
<td>craig</td>
<td>trent</td>
<td></td>
</tr>
<tr>
<td>frank</td>
<td>trudy</td>
<td></td>
</tr>
<tr>
<td>victor</td>
<td>victor</td>
<td></td>
</tr>
</tbody>
</table>
**Merge: Java implementation**

**Abstract inplace merge**
- Merge $a[lo, mid]$ with $a[mid, hi]$.
- Use auxiliary array for result.
- Copy back when sort complete.

```java
private static String[] aux;
public static void merge(String[] a, int lo, int mid, int hi) {
    // Merge a[lo, mid] with a[mid, hi] into aux[0, hi-lo).
    int i = lo, j = mid, N = hi - lo;
    for (int k = 0; k < N; k++)
    {
        if (i == mid) aux[k] = a[j++];
        else if (j == hi) aux[k] = a[i++];
        else if (a[j].compareTo(a[i]) < 0) aux[k] = a[j++];
        else aux[k] = a[i++];
    }
    // Copy back into a[lo, hi)
    for (int k = 0; k < N; k++)
        a[lo + k] = aux[k];
}
```
Merge sort: Java implementation

Merge sort

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.

```
public class Merge {
    private static String[] aux;
    public static void merge(String[] a, int lo, int mid, int hi) {
        // See previous slide. }
    public static void sort(String[] a) {
        sort(a, 0, a.length); }
    public static void sort(String[] a, int lo, int hi) {
        // Sort a[lo, hi).
        int N = hi - lo;
        if (N <= 1) return;
        int mid = lo + N/2;
        sort(a, lo, mid);
        sort(a, mid, hi);
        merge(a, lo, mid, hi);
    }
}
```
Mergesort trace

Merge sort
- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.
## Mergesort analysis

**Cost model. Count data moves.**

### Exact analysis for $N = 2^n$.
- Note that $n = \lg N$.
- 1 subarray of size $2^n$.
- 2 subarrays of size $2^{n-1}$.
- 4 subarrays of size $2^{n-2}$.
- ...
- $2^n$ subarrays of size 1.
- Total # data moves: $2N \lg N$.

![Subarrays](image)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>alice</td>
<td>alice</td>
<td>alice</td>
<td>wendy</td>
</tr>
<tr>
<td>bob</td>
<td>carlos</td>
<td>dave</td>
<td>wendy</td>
<td>bob</td>
</tr>
<tr>
<td>carlos</td>
<td>carol</td>
<td>walter</td>
<td>dave</td>
<td>carlos</td>
</tr>
<tr>
<td>carol</td>
<td>dave</td>
<td>wendy</td>
<td>walter</td>
<td>carlos</td>
</tr>
<tr>
<td>craig</td>
<td>erin</td>
<td>carol</td>
<td>carol</td>
<td>carol</td>
</tr>
<tr>
<td>dave</td>
<td>oscar</td>
<td>carol</td>
<td>carol</td>
<td>carol</td>
</tr>
<tr>
<td>erin</td>
<td>walter</td>
<td>erin</td>
<td>oscar</td>
<td>oscar</td>
</tr>
<tr>
<td>eve</td>
<td>wendy</td>
<td>eve</td>
<td>trudy</td>
<td>trudy</td>
</tr>
<tr>
<td>frank</td>
<td>bob</td>
<td>peggy</td>
<td>eve</td>
<td>eve</td>
</tr>
<tr>
<td>oscar</td>
<td>craig</td>
<td>peggy</td>
<td>trudy</td>
<td>trent</td>
</tr>
<tr>
<td>peggy</td>
<td>eve</td>
<td>trent</td>
<td>eve</td>
<td>trent</td>
</tr>
<tr>
<td>trent</td>
<td>frank</td>
<td>trent</td>
<td>eve</td>
<td>trent</td>
</tr>
<tr>
<td>trudy</td>
<td>peggy</td>
<td>bob</td>
<td>frank</td>
<td>frank</td>
</tr>
<tr>
<td>victor</td>
<td>trent</td>
<td>craig</td>
<td>frank</td>
<td>frank</td>
</tr>
<tr>
<td>walter</td>
<td>trudy</td>
<td>bob</td>
<td>victor</td>
<td>victor</td>
</tr>
<tr>
<td>wendy</td>
<td>victor</td>
<td>victor</td>
<td>victor</td>
<td>victor</td>
</tr>
</tbody>
</table>

- 1 subarray of size $N$.
- 2 subarrays of size $N/2$.
- 4 subarrays of size $N/4$.
- 8 subarrays of size $N/8$.
- 16 subarrays of size $N/16$.

$2N$ data moves  $2N$ data moves  $2N$ data moves  $2N$ data moves

Interested in details? Take a course in algorithms.
Empirical tests of mergesort

Sort random strings
- Array of length $N$.
- 10-character strings.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$T_N$ (seconds)</th>
<th>$T_N/T_{N/2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 million</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2 million</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4 million</td>
<td>5</td>
<td>2.5</td>
</tr>
<tr>
<td>8 million</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>16 million</td>
<td>20</td>
<td>2.5</td>
</tr>
<tr>
<td>...</td>
<td>1280</td>
<td>2</td>
</tr>
<tr>
<td>1.02 billion</td>
<td>1280</td>
<td>2</td>
</tr>
</tbody>
</table>

Confirms hypothesis that order of growth is $N \log N$.
13. Sorting and Searching

- A typical client
- Binary search
- Insertion sort
- Mergesort
- Longest repeated substring
Detecting repeats in a string

Longest repeated substring
• Given: A string s.
• Task: Find the longest substring in s that appears at least twice.

Example 1.  a a c a a g t t t a c a a g c

Example 2.  a a c a a g t t t a c a a g t t t a c a a g c t a g c

Example 3 (first 100 digits of \(\pi\)).

| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 | 8 | 9 | 7 | 9 | 3 | 2 | 3 | 8 | 4 |
| 6 | 2 | 6 | 4 | 3 | 3 | 8 | 3 | 2 | 7 | 9 | 5 | 0 | 2 | 8 | 8 | 4 | 1 | 9 | 7 |
| 1 | 6 | 9 | 3 | 9 | 9 | 3 | 7 | 5 | 1 | 0 | 5 | 8 | 2 | 0 | 9 | 7 | 4 | 9 | 4 |
| 4 | 5 | 9 | 2 | 3 | 0 | 7 | 8 | 1 | 6 | 4 | 0 | 6 | 2 | 8 | 6 | 2 | 0 | 8 | 9 |
| 9 | 8 | 6 | 2 | 8 | 0 | 3 | 4 | 8 | 2 | 5 | 3 | 4 | 2 | 1 | 1 | 7 | 0 | 6 | 9 |
LRS example: repetitive structure in music

Mary had a little lamb

Fur Elise

source: http://www.bewitched.com/match/
Analysts seek repeated sequences in real-world data because they are causal.

Example 1: Digits of π
- Q. Are they “random”? 
- A. No, but we can’t tell the difference.
- Ex. Length of LRS in first 10 million digits is 14.

Example 2: Cryptography
- Find LRS.
- Check for “known” message header information.
- Break code.

Example 3: DNA
- Find LRS
- Look somewhere else for causal mechanisms
- Ex. Chromosome 11 has 7.1 million nucleotides
Warmup: Longest common prefix

Longest common prefix
• Given: Two strings string s and t.
• Task: Find the longest substring that appears at the beginning of both strings.

Example.  

```
aaacaaagtttaacaaagc
aaacaaagttttacaaagtttaacaaagc
gttttacaaagc
```

Implementation (easy)

```java
private static String lcp(String s, String t) {
    int N = Math.min(s.length(), t.length());
    for (int i = 0; i < N; i++)
        if (s.charAt(i) != t.charAt(i))
            return s.substring(0, i);
    return s.substring(0, N);
}
```
LRS: Brute-force implementation

```java
class LRS {
    public static String lcp(String s) {
        // See previous slide.
    }

    public static String lrs(String s) {
        int N = s.length();
        String lrs = "";
        for (int i = 0; i < N; i++)
            for (int j = i+1; j < N; j++)
                String x = lcp(s.substring(i, N), s.substring(j, N));
                if (x.length() > lrs.length()) lrs = x;
        return lrs;
    }

    public static void main(String[] args) {
        String s = StdIn.readString();
        StdOut.println(lrs(s));
    }
}
```

Analysis
- ~$N^2/2$ calls on lcp().
- Obviously does not scale.
LRS: An efficient solution that uses sorting

1. Form suffix strings

```
0  a a c a a g t t t a c a a g c
1  a c a a g t t t a c a a g c
2  c a a g t t t a c a a g c
3  a a g t t t a c a a g c
4  a g t t t a c a a g c
5  g t t t a c a a g c
6  t t t a c a a g c
7  t t c a a a g c
8  t a c a a g c
9  a c a a g c
10 c a a g c
11 a a g c
12 a g c
13 g c
14 c
```

2. Sort suffix strings

```
0  a a c a a g t t t a c a a g c
11 a a g c
3  a a g t t t a c a a g c
9  a c a a g c
1  a c a a g t t t a c a a g c
12 a g c
4  a g t t t a c a a g c
14 c
10 c a a g c
2  c a a g t t t a c a a g c
13 g c
5  g t t t a c a a g c
8  t a c a a g c
7  t t c a a a g c
6  t t t a c a a g c
```

3. Find longest LCP among adjacent entries.
LRS implementation

```java
public class LRS {
    public static String lcp(String s) {
        // See previous slide. }

        int N = s.length();
        String[] suffixes = new String[N];
        for (int i = 0; i < N; i++)
            suffixes[i] = s.substring(i, N);

        String lrs = "";
        for (int i = 0; i < N-1; i++)
            String x = lcp(suffixes[i], suffixes[i+1]);
            if (x.length() > lrs.length()) lrs = x;

        return lrs;
    }
}
```

**Important note**

- Efficiency depends on constant-time substring operation.
- Forming suffix string array takes quadratic time and space if substring operation copies the substring to make a new string.
- [see next slide]
Two alternatives for implementing substrings

1. Refer to original string.
   - No need to copy characters.
   - *Constant* time.

   ```java
   String genome = "aacaagtttacaagc";
   String s = genome.substring(1, 5);
   String t = genome.substring(9, 13);
   ```

2. Copy the characters to make a new string.
   - Allows potential to free up memory when the original string is no longer needed.
   - *Linear* time (in the length of the substring).
LRS: Empirical analysis

Model
- Alphabet: actg.
- $N$-character random strings.

<table>
<thead>
<tr>
<th>Doubling</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>$T_N$</td>
<td>$T_N/T_{N/2}$</td>
</tr>
<tr>
<td>2,000,000</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4,000,000</td>
<td>7</td>
<td>2.3</td>
</tr>
<tr>
<td>8,000,000</td>
<td>16</td>
<td>2.3</td>
</tr>
<tr>
<td>16,000,000</td>
<td>39</td>
<td>2.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x10</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>$T_N$</td>
<td>$T_N/T_{N/10}$</td>
</tr>
<tr>
<td>1,000,000</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>10,000,000</td>
<td>21</td>
<td>10</td>
</tr>
</tbody>
</table>

Confirms hypothesis that the order of growth is $N \log N$ (for the sort).

Bottom line. Scales with the size of the input and enables new research and development.
Important notes on LRS implementation

Long repeats

- More precise analysis reveals that running time is \textit{quadratic} in the length of the longest repeat.
- Model has no long repeats.
- Real data may have long repeats.
- \textit{Linear} time algorithm (guarantee) is known.

String representation

- Efficiency depends on constant-time substring operation.
- 1995–2012: Java substring is constant-time.
- 2013: Java 7 changes to \textit{linear}-time substring operation! (breaks this and many classic algorithms).
- Need to implement our own constant-time-substring.

Example: Chromosome 11 has a repeat of length 12,567.
Summary

**Binary search.** Efficient algorithm to search a sorted array.

**Merge sort.** Efficient algorithm to sort an array.

**Applications.** Many, many, many things are enabled by fast sort and search.

Hey, Bob. Our IPO is next week!

I think I'll take a few CS courses.
13. Sorting and Searching