Algorithms

A working definition: a sequence of instructions for performing a task

Some common properties of algorithms:
- provably correct
- clear and unambiguous
- automatable
- efficient?

Efficiency

What do we mean by efficiency?
Consider two functions that erase all elements of an array list:

- `void array_list::clear1() { while(_size > 0) erase(0); }`
- `void array_list::clear2() { for (int j = _size - 1; j >= 0; j--) erase(j); }`

Which does less work?

Measuring Work

- Our measure of work done will be roughly equal to the number of basic computer steps performed
- A basic computer step is any constant time operation:
  - Arithmetic operation
  - Basic variable assignment/copy
  - Comparisons/branching
  - Memory/array access, e.g., `x = a[i]` or `a[i] = x`
- Non-basic computer steps:
  - Loops
  - Memory/array copy
  - String concatenation
- In general, if time spent does not depend on the input, it is constant time

Measuring Work: erase

```cpp
class array_list {
public:
    void erase(int index);
private:
    int* _arr;
    int _size;
    int _capacity;
};
void array_list::erase(int index) {
    for (int j = index; j < _size - 1; j++)
        _arr[j] = _arr[j + 1];
    _size--;
}
```

Let `n = _size - index - 1;`
- `n` comps, incs
- `n+1` assignments
- 1 decrement

Measuring Work: clear1

How much work does this do?

```cpp
void array_list::clear1() {
    while(_size > 0) erase(0);
}
```

Suppose our array is size 10 to start with.
Keep in mind what erase(0) is doing: copying elements 1-9 over into positions 0-8.

```cpp
void array_list::clear1() {
    for (int j = _size - 1; j >= 0; j--)
        _arr[j] = _arr[j + 1];
    _size--;
}
```
Measuring Work: clear1

How much work does *erase* do?

```cpp
void array_list::clear1() {
    while (_size > 0) erase(0);
}
```

- First time through while loop: 9 comparisons, increments, and array accesses/copies + update size
- Second time through: 8 comparisons, increments, and array accesses/copies + update size
- ... Last time through: update size

So the amount of work is \( 3 \times (10 + 9 + \ldots + 1) + 10 \).

Measuring Work: clear2

How much work does this do?

```cpp
void array_list::clear2() {
    for (int j = _size - 1; j >= 0; j--)
        erase(j);
}
```

Well, how much work does *erase* do?

- Recall work is \( 3 \times (_\text{size} - j - 1) + 1 \)
- Here, \( j = _\text{size} - 1 \)
- Work \( \approx 4 \)

Comparing:

- clear1 \( \approx 3 \times (10 + 9 + \ldots + 1) + 10 \approx 175 \)
- clear2 \( \approx 5 \times (10) = 50 \)

Let’s generalize to arrays of size \( n \):

- clear1: \( 3 \times (n + (n-1) + (n-2) + \ldots + 1) + n \)
- clear2: \( 5 \times n \)

For an array of size \( n \), clear2 takes \( 5n \) steps. What about clear1?

### Arithmetic Series

Memorize this!

\[
\sum_{i=0}^{n} i = \frac{n(n+1)}{2}
\]

That is,

\[
0 + 1 + 2 + \ldots + n = \frac{n^2 + n}{2}.
\]
How to Solve $\sum_{i=0}^{n} i$

Write the sum twice, once forwards and once backwards; then sum the two:

\[
\begin{array}{cccccc}
0 & 1 & \ldots & n-1 & n \\
+ n & n & \ldots & 1 & 0 \\
\hline
n & n & \ldots & 1 & 0 \\
\end{array}
\]

How many n's are there in the sum? Answer: n+1.

Since we took twice the summation, we have to divide by 2.
Thus we have $\frac{n(n+1)}{2}$.
Can also prove easily using induction...

Measuring Work: clear1 & clear2

- clear1 $\approx 3 \cdot (n^2 + n)/2 + n$
- clear2 $\approx 5 \cdot n$

It turns out, this level of detail is excessive...

“Big O”

Big O notation:
- $O(n)$ measures asymptotic complexity of algorithm

Don’t worry about the fancy language for now – this will be explained in CSCI 406!

What is important:
- In Big O, lower order terms and constant don’t matter
- More interested in how functions grow with size of n

Simplifying

Typically use the simplest term in expression:
- Lower order polynomials can be ignored because they are completely dominated by higher order polynomials
  - $O(n)$ not $O(n + c)$
  - $O(n^2)$ not $O(n^2 + n + c)$
- Ignore constants
  - $O(n)$ not $O(3n)$
  - $O(n)$ not $O(n/2)$

Dominance relations (here $a > b$ means $a$ dominates $b$):

$n! > 3^n > 2^n > n^3 > n^2 > n \log n > n > \log n > 1$

Big O of clear Functions

- clear1: $O(3n^2/2 + 3n/2 + n) = O(n^2)$
- clear2: $O(5n) = O(n)$

Knowing how array_list works, can you think of an even better way to write a clear() function?

Big-O Comparisons
### Why We Care 1

Comparison of different orders of functions as size of input $n$:

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log(n)$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>$10^3$</td>
<td>$10^6$</td>
<td>$10^9$</td>
</tr>
<tr>
<td>$n \log(n)$</td>
<td>10</td>
<td>200</td>
<td>3000</td>
<td>$6 \times 10^3$</td>
<td>$9 \times 10^6$</td>
<td></td>
</tr>
<tr>
<td>$n^2$</td>
<td>100</td>
<td>$10^4$</td>
<td>$10^6$</td>
<td>$10^{12}$</td>
<td>$10^{24}$</td>
<td></td>
</tr>
<tr>
<td>$2^n$</td>
<td>$10^{24}$</td>
<td>$\sim 10^{31}$</td>
<td>$\sim 10^{38}$</td>
<td>Forget it!</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Why We Care 2

Assuming $2 \times 10^{10}$ operations/second (approximately the FP performance of a typical CPU c. 2011)

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log(n)$</th>
<th>10</th>
<th>50</th>
<th>100</th>
<th>$10^3$</th>
<th>$10^6$</th>
<th>$10^9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$&lt; 1$ ns</td>
<td>$&lt; 1$ ns</td>
<td>$&lt; 1$ ns</td>
<td>1 ns</td>
<td>1 ns</td>
<td>2 ns</td>
<td></td>
</tr>
<tr>
<td>$n \log(n)$</td>
<td>$&lt; 1$ ns</td>
<td>$&lt; 1$ ns</td>
<td>$&lt; 1$ ns</td>
<td>50 μs</td>
<td>50 ms</td>
<td>50 s</td>
<td></td>
</tr>
<tr>
<td>$n^2$</td>
<td>$&lt; 1$ ns</td>
<td>$&lt; 1$ ns</td>
<td>1 ns</td>
<td>300 ms</td>
<td>450 ms</td>
<td>10 min</td>
<td></td>
</tr>
<tr>
<td>$2^n$</td>
<td>$125$ ns</td>
<td>500 ns</td>
<td>50 s</td>
<td>1.6 years</td>
<td>1.6 million years</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Datasets of size $10^3$ and above are commonplace!

# of unique URLs seen by Google indexer c. 2010

### Up Next

- **Wednesday, March 21**
  - Selection Sort
- **Friday, March 23**
  - Lab 9 (continued)
  - Extra credit APTs assigned
  - **SPRING BREAK!**
- **Monday, April 2**
  - Lab 9 due