Selection Sort

- Input: a list of elements, e.g. integers
- Output: a list of the input elements in sorted order

Why do we study this problem?
- Teaching example
- Algorithm design
- Algorithm analysis
- Sorting is also useful for all sorts of applications!

Selection Sort Illustrated

Selection Sort Code

```c++
// for vectors of int
void selection_sort(vector<int> &vec) {
    int n = vec.size();
    for (int left = 0; left < n; left++) {
        int right = left;
        for (int j = left + 1; j < n; j++) {
            if (vec[j] < vec[right]) right = j;
        }
        swap(vec[left], vec[right]);
    }
}
```
### Swap Code

// exchange two int values
void swap(int &a, int &b) {
    int tmp = a;
    a = b;
    b = tmp;
}

This function is actually already available in the standard library; #include <algorithm>

### How Much Work in Selection Sort?

- Difficult to actually count CPU cycles –
  - Differs by CPU
  - Differs by compiler
  - Lots of noise factors: caching, context switching, etc.
- Simplification: just measure comparisons and swaps
  - Ignore loop counter updates, etc.
  - We’ll see later why we can get away with this
- Let’s count (only somewhat carefully):
  - Use a vector of size 10
  - Later, generalize to size \(n\)

### Analyzing Selection Sort

```cpp
void selection_sort(vector<int> &vec) {
    int n = vec.size();
    for (int left = 0; left < n; left++) {
        int right = left;
        for (int j = left + 1; j < n; j++) {
            if (vec[j] < vec[right]) right = j;
        }
        swap(vec[left], vec[right]);
    }
}
```

### Analyzing Selection Sort: 1st Loop

On first loop:

- Compare min element with each of 9 elements: cost = 9
- Do 1 swap: cost = 1

Total cost: 10

### Analyzing Selection Sort: 2nd Loop

On second loop:

- Compare min element with each of 8 elements: cost = 8
- Do 1 swap: cost = 1

Total cost: 9
Analyzing Selection Sort: Last Loop

In the end, we have a list of size 1 left and don’t have to do any work!
So work is $10 + 9 + 8 + ... + 1 + 0$
$= 55$

Now let’s generalize to vectors of size $n$
First loop: $n - 1$ comparisons, 1 swap: cost = $n$
Second loop: cost = $n - 1$
... 
Cost: $n + (n - 1) + (n - 2) + ... + 1 + 0$

This is a famous sum!

Arithmetic Series

Memorize this!

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

That is,

$0 + 1 + 2 + ... + n = \frac{n^2 + n}{2}$

How to Solve $\sum_{i=0}^{n} i$

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

$$0 + 1 + ... + n-1 + n + n + ... + n-1 + 1 + 0$$

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, geometry, ...

Visual Analysis

Counting based on code can be a pain; Sometimes, a visual approach is simpler:

Original list

| 5 | 7 | 2 | 10 | 3 | 1 | 6 |

Elements “touched” in first loop iteration

| 5 | 7 | 2 | 10 | 3 | 1 | 6 | n |

Elements “touched” in second loop iteration

| 1 | 5 | 2 | 10 | 3 | 1 | 6 | n-1 |

Elements “touched” in third loop iteration

| 1 | 2 | 5 | 2 | 10 | 3 | 1 | 6 | n-2 |

... 

Last iteration

| 1 | 2 | 3 | 5 | 2 | 10 | 3 | 1 | 6 | 7 | 1 |

Cost: $O(1 + 2 + ... + n)$

ALGORITHMIC ANALYSIS
“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 constants don’t matter
- Only interested in how functions grow with size of \( n \)

Simplifying

Typically use the simplest term in expression:
- E.g., 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 \( b \)):
\[
\begin{align*}
    n! & > 3^n \quad & 2^n & > n^3 \quad & n^2 & > n \log n \quad & n & > \log n \quad & 1
\end{align*}
\]

Practice

Simplify the following:
- \( O(n^3 + 4) \)
- \( O(12n^2 - n + 1) \)
- \( O(2^n + n^2) \)
- \( O(n + n^2 + n \log n) \)

Technicalities (for the curious)

- Defined:
  \[ f(n) = O(g(n)) \text{ means } f(n) \leq c g(n) \text{ for some } c \text{ as } n \to \infty \]
  More formally, \( f(n) = O(g(n)) \) if there exists \( c, n_0 \) such that \( f(n) \leq c g(n) \) for all \( n > n_0 \).
- The asymptotic complexity of \( f \) is upper bounded by \( g \)

Proof Sketch (for the curious)

(Just for show: not on any exams or homework!)
Prove: \( 3n^2 + n/4 + 1 = O(n^2) \)
Find \( c, n_0 \) such that \( (3n^2 + n/4 + 1) \leq c n^2 \) for all \( n \geq n_0 \)

Choose \( c = 4, n_0 = 2 \)

Inductive proof:
- Base case: \( 3(2^2) + 2/4 + 1 = 13\frac{1}{2} \leq 16 = 4(2^2) \)
- Induction:
  - Suppose \( 3n^2 + n/4 + 1 \leq c n^2 \)
  - Show for \( n + 1 \):
    \[
    \begin{align*}
    3(n + 1)^2 + (n + 1)/4 + 1 & \leq 4(n + 1)^2 \\
    3(n^2 + 2n + 1) + n/4 + 1 & \leq 4n^2 + 4 \\
    3n^2 + 6n + 3 + n/4 + 1 & \leq 4n^2 + 4 \\
    (3n^2 + n/4 + 1) + 6n + 3 & \leq 4n^2 + 4
    \end{align*}
    \]

Asymptotic Complexity Comparison
Big-O Comparisons

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

<table>
<thead>
<tr>
<th>n</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>10^3</th>
<th>10^6</th>
</tr>
</thead>
<tbody>
<tr>
<td>log(n)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>n</td>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>100000</td>
<td>100000000</td>
</tr>
<tr>
<td>n log(n)</td>
<td>10</td>
<td>200</td>
<td>3000</td>
<td>6 x 10^6</td>
<td>9 x 10^9</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>
</tr>
<tr>
<td>2^n</td>
<td>1024</td>
<td>~10^10</td>
<td>~10^20</td>
<td>Forget it</td>
<td></td>
</tr>
</tbody>
</table>

Why We Care 1

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

Datasets of size 10^6 and above are commonplace!

<table>
<thead>
<tr>
<th>n</th>
<th>10</th>
<th>50</th>
<th>100</th>
<th>10^3</th>
<th>10^6</th>
</tr>
</thead>
<tbody>
<tr>
<td>log(n)</td>
<td>&lt; 1 ns</td>
<td>&lt; 1 ns</td>
<td>&lt; 1 ns</td>
<td>1 ms</td>
<td>2 ms</td>
</tr>
<tr>
<td>n</td>
<td>&lt; 1 ms</td>
<td>&lt; 1 ms</td>
<td>&lt; 1 ms</td>
<td>50 μs</td>
<td>50 μs</td>
</tr>
<tr>
<td>n log(n)</td>
<td>&lt; 1 ns</td>
<td>&lt; 1 ns</td>
<td>1 ms</td>
<td>300 ms</td>
<td>450 ms</td>
</tr>
<tr>
<td>n^2</td>
<td>&lt;1 ms</td>
<td>125 ms</td>
<td>500 ms</td>
<td>50 s</td>
<td>1.6 years</td>
</tr>
<tr>
<td>2^n</td>
<td>50 ms</td>
<td>16 hours</td>
<td>1.5 trillion years</td>
<td>1.6 million years</td>
<td></td>
</tr>
</tbody>
</table>

Why We Care 2

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

Datasets of size 10^6 and above are commonplace!

# of unique URLs seen by Google indexer c. 2010

Up Next

- Friday, August 31
- Lab 2
- APT 1 due
- Project 1 assigned