CSCI 262
Data Structures

7
Selection Sort
Introduction to Analysis of Algorithms

SORTING

Getting things in order

Sorting

- 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

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

A simple solution:

- Find the minimum element in the list
- Swap it with the first element in the list
- Sort the sublist following the first element

This sorting algorithm is named selection sort.

Selection Sort Illustrated

Selection Sort Code

```cpp
// 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 $10 + 9 + 8 + \ldots + 1 + 0$

$= 55$

Now let’s generalize to vectors of size $n$
First loop: $n - 1$ comparisons, 1 swap: cost = $n$
Second loop: cost = $n - 1$

$\ldots$

Cost: $n + (n - 1) + (n - 2) + \ldots + 1 + 0$

This is a famous sum!

Arithmetic Series

Memorize this!

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

That is,

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

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

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

$$0 + 1 + 2 + \ldots + n + n - 1 + \ldots + 1 + 0$$

$= n + n - 1 + \ldots + n + n - 1 + \ldots + 1 + 0 + n - 1 + \ldots + 1 + 0$

$\text{How many n’s are there in the sum? Answer: } n + 1.$

Since we took twice the summation, we have to divide by 2.
Thus we have $n(n + 1)/2$.

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

<table>
<thead>
<tr>
<th>5</th>
<th>7</th>
<th>2</th>
<th>10</th>
<th>3</th>
<th>1</th>
<th>6</th>
</tr>
</thead>
</table>

Elements “touched” in first loop iteration

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>

Elements “touched” in second loop iteration

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>

Elements “touched” in third loop iteration

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>

$\ldots$

Last iteration

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>

Cost: $O(1 + 2 + \ldots + n)$
“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):
- \( n! > 3^n > 2^n > n^3 > n^2 > n \log n > n > \log n > 1 \)

**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:
  \[ f(n) = O(g(n)) \] means \( f(n) \leq c g(n) \) for some \( c \) 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: \( 3(2^2) + 2/4 + 1 = 13.5 \leq 16 = 4(2^2) \)
- Induction:
  - Suppose \( 3n^2 + n/4 + 1 \leq n^2 \)
  - Show for \( n + 1 \):
    - \( 3(n + 1)^2 + (n + 1)/4 + 1 \leq 4(n + 1)^2 \)
    - \( 3n^2 + 6n + 3 + n/4 + 1 \leq 4n^2 + 8n + 4 \)
    - \( (3n^2 + n + 1) + 6n + 3 \leq 4n^2 + 8n + 4 \)

\( \square \)

**Asymptotic Complexity Comparison**

![Graph showing asymptotic complexity comparison](image.png)
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>$10^4$</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>1000000</td>
<td>1000000000</td>
</tr>
<tr>
<td>$n \log(n)$</td>
<td>10</td>
<td>200</td>
<td>3000</td>
<td>$6 \times 10^6$</td>
<td>$9 \times 10^6$</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>$10^{14}$</td>
<td>$10^{29}$</td>
<td>$10^{60}$</td>
<td>Forget it!</td>
<td></td>
</tr>
</tbody>
</table>

Why We Care 1

Datasets of size $10^6$ and above are commonplace!

Datasets of size $10^6$ and above are commonplace!

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>$m$</th>
<th>$10$</th>
<th>$50$</th>
<th>$100$</th>
<th>$10^4$</th>
<th>$10^6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>log($m$)</td>
<td>&lt;1 ms</td>
<td>&lt;1 ms</td>
<td>&lt;1 ms</td>
<td>1 ms</td>
<td>2 ms</td>
</tr>
<tr>
<td>$m$</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>$m \log(m)$</td>
<td>&lt;1 ms</td>
<td>&lt;1 ms</td>
<td>1 ms</td>
<td>300 ms</td>
<td>450 ms</td>
</tr>
<tr>
<td>$m^2$</td>
<td>&lt;1 ms</td>
<td>125 ns</td>
<td>500 ns</td>
<td>50 μs</td>
<td>1.6 million years</td>
</tr>
<tr>
<td>$2^m$</td>
<td>50 ns</td>
<td>16 hours</td>
<td>1.5 trillion years</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

# of unique URLs seen by Google indexer c. 2010

Up Next

- Start reading for Monday: Chapter 11
- Friday, February 1
  - Lab 4 – Analysis of Algorithms
  - APT 2 due
  - Project 2 - Mazes assigned
    - This project will be interview graded – we’ll tell you more and give you a way to sign up for an interview soon!
- Monday, February 4
  - More about functions and the function call stack; review of recursion
  - Lab 4 due

*Why haven’t I updated this slide? By misreading of Moore’s Law, CPUs could be twice as fast today as in 2011. How much should we care about this speedup?