Big-Oh Notation

James Daly
CSE 232
Michigan State University
Sorting and Searching

▪ Sorting and Searching are important algorithms in computer science
▪ Using efficient sorting and searching algorithms can dramatically influence total run times of a program
▪ How does one best measure “efficient”?
“Time” is not the best measure

- Literally measuring time is ambiguous
  - CPUs get faster over time
  - One computer might be faster than another
  - How can we come up with a reference that is meaningful if we measure time?
  - Not very theoretical
Algorithm efficiency is important

- Maybe not using time, but algorithm efficiency is very important
- Easy to write an algorithm that is horribly slow and, with a little insight, can be made quite a bit faster
- We haven’t worried about it much yet but it is a very big issue
Measure effort compared to data size

- We aren’t interested in time as we are in the **relative change in time** as the data we are working with gets larger.
- How much time does your algorithm use as the amount of data it has to work with grows?
One measure, comparisons

- Today, we will examine different sorting and searching algorithms and we will compare their run times
- In particular, we will be interested in the change in time as the size of the data increases
- Algorithm efficiency is a big issue in CS and searching / sorting are convenient ways to examine that
- Covered in more detail in CSE 331
Sort Algorithm

- Write an algorithm to sort a list into increasing order
- Selection sort (2 versions)
Selection Sort

- Work through a list
  1. Find the smallest element in the list and put it at the beginning
  2. The list is now trivially sorted at the beginning (only the first element)
  3. Find the smallest element in the rest of the list (the unsorted part) and put it at the beginning of the list
  4. Repeat until done with the list
Selection Sort

- We want to sort \texttt{ary} to be [2, 4, 6, 8].
- Find the minimal element (2).
- Swap the minimal with the first (2 & 8).
- Fix the first position
- Repeat on the rest (e.g. find the minimum of [8, 6, 4])
Selection Sort

2 is smallest of [8, 2, 6, 4]; Swap 8 and 2

4 is smallest of [8, 6, 4]; Swap 8 and 4

6 is smallest of [6, 8]; Swap 6 and 6
Selection Sort, v1

- Uses `min_element` to get an iterator to the smallest element
- Uses STL swap to swap the elements
- Address of iterator – beginning = index

```c
void stl_selection(long ary[], size_t sz) {
    for (size_t i = 0; i < sz; i++) {
        long * element = min_element((ary + i), (ary + sz));
        swap(ary[element - ary], ary[i]);
    }
}
```
SelectionSort, v2

void loop_selection(long ary[], size_t sz) {
    for (size_t i = 0; i < sz; i++) {
        size_t min_index = i;
        for (size_t j = i + 1; j < sz; j++) {
            if (ary[j] < ary[min_index])
                min_index = j;
        }
        temp = ary[i];
        ary[i] = ary[min_index];
        ary[min_index] = temp;
    }
}
Insertion Sort

- Have a division of the array into two parts
  - The left side, all elements are relatively sorted with respect to each other
  - The right side, unsorted
- Pick an element on the unsorted side
- Place it in its proper side on the left
  - Unsorted down by one
  - Relatively sorted up by one
InsertionSort Sort

1. Insert 2 into its sorted position
2. Insert 6 into its sorted position
3. Insert 4. Note it swaps with each element until it reaches its final position
Helpful STL

- **upper_bound(begin, end, value)**
  - Return a ptr / iterator to the first element **just bigger** than the value
  - Assumes sorted range

- **rotate(begin, ptr, end)**
  - Half-open range(end is one past values being moved)
  - **ptr** becomes first in the range **ptr−1** the last in the range
  - Rest shifted to the left
rotate(ary+2, ary+4, ary+6)

- 7 (+4) becomes the first element of the range 4(+2) to 0 (+5)
  - Half-open range 3 (+6) one past end
- 7 and 0 move down 2 spaces, 4 behind 0, 1 behind 4
- Half open range so 3 (+6) does not move, nor anything behind it
Insertion Sort (STL)

```c
void stl_insert_sort(long ary[], size_t sz) {
    for (size_t i = 0; i < sz; i++) {
        long* upr = upper_bound(ary, ary + i, ary[i]);
        rotate(upr, ary + i, ary + (i + 1));
    }
}
```
Example

Sorted

1 3 4 5 | 2 7 6

i = 4 (working on 2)

ary

1 3 4 5

ary[i] == 2

upr = upper_bound(ary, ary+i, ary[i])

Note this half-open range is sorted!

rotate(upr, ary+i, ary+(i+1))

2 now inserted, i up by 1, 7 next
Loop Approach

```c
void insertion_sort(long ary[], size_t sz) {
    for (size_t j = 1; j < sz; j++) {
        size_t to_place = ary[j];
        for (size_t i = j-1; (i >= 0) && (ary[i] > to_place); i--)
            ary[i+1] = ary[i];
        ary[i+1] = to_place;
    }
}
```
Movement of one element

to\_place = 4, j = 3, i = 2, is \( \text{ary}[i] > \text{to\_place} \) ?

\[
\begin{array}{cccc}
\text{ary} & 2 & 6 & 8 & 4 \\
0 & 1 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{ary} & 2 & 6 & 8 & 8 \\
0 & 1 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{ary} & 2 & 6 & 6 & 8 \\
0 & 1 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{ary} & 2 & 4 & 6 & 8 \\
0 & 1 & 2 & 3 \\
\end{array}
\]

with \( i = 0 \), \( \text{ary}[i+1] = \text{to\_place} \)
## Timing: sort 1000 longs

<table>
<thead>
<tr>
<th>Sort</th>
<th>Time (ms)</th>
<th>Slower...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection (index)</td>
<td>0.371 ms</td>
<td>10.3x</td>
</tr>
<tr>
<td>Selection (STL)</td>
<td>0.351 ms</td>
<td>9.27x</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>0.125 ms</td>
<td>3.47x</td>
</tr>
<tr>
<td>Insertion (STL)</td>
<td>0.107 ms</td>
<td>2.97x</td>
</tr>
<tr>
<td>C++ builtin</td>
<td>0.036 ms</td>
<td>1</td>
</tr>
</tbody>
</table>
## Timing: sort 10000 longs

<table>
<thead>
<tr>
<th>Sort</th>
<th>Time (ms)</th>
<th>Slower...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection (index)</td>
<td>30.211 ms</td>
<td>71.08x</td>
</tr>
<tr>
<td>Selection (STL)</td>
<td>25.719 ms</td>
<td>60.52x</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>10.238 ms</td>
<td>24.08x</td>
</tr>
<tr>
<td>Insertion (STL)</td>
<td>7.508 ms</td>
<td>17.67x</td>
</tr>
<tr>
<td>C++ builtin</td>
<td>0.425 ms</td>
<td>1</td>
</tr>
</tbody>
</table>
## Timing: sort 100,000 longs

<table>
<thead>
<tr>
<th>Sort</th>
<th>Time (ms)</th>
<th>Slower...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection (index)</td>
<td>2991.53 ms</td>
<td>598x</td>
</tr>
<tr>
<td>Selection (STL)</td>
<td>2610.7 ms</td>
<td>522x</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>1260.67 ms</td>
<td>252x</td>
</tr>
<tr>
<td>Insertion (STL)</td>
<td>791.65 ms</td>
<td>158x</td>
</tr>
<tr>
<td>C++ builtin</td>
<td>5.476 ms</td>
<td>1</td>
</tr>
</tbody>
</table>
Timing: sort ints in ms

<table>
<thead>
<tr>
<th>Sort</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>10x data vs change in time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection (index)</td>
<td>0.371</td>
<td>30.211</td>
<td>2991.53</td>
<td>~100x</td>
</tr>
<tr>
<td>Selection (STL)</td>
<td>0.351</td>
<td>25.719</td>
<td>2610.7</td>
<td>~100x</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>0.125</td>
<td>10.238</td>
<td>1260.67</td>
<td>~100x</td>
</tr>
<tr>
<td>Insertion (STL)</td>
<td>0.107</td>
<td>7.508</td>
<td>791.65</td>
<td>~100x</td>
</tr>
<tr>
<td>C++ builtin</td>
<td>0.036</td>
<td>0.425</td>
<td>5.476</td>
<td>~20x</td>
</tr>
</tbody>
</table>
Big-Oh

- When comparing algorithms, computer scientists use Big-Oh notation which indicates how algorithms behave as the size of input increases.
- Only the general relationship is required. As the size increases, the details don’t matter (doesn’t indicate exact time growth).
Big-Oh

- Algorithm complexity is written as $O(?)$, where $?$ represents the change in time relative to the size of the data
  - $O(1)$: Constant time
  - $O(\log n)$: Time grows as the log of the size
  - $O(n)$: Time grows linearly with size
  - $O(n^2)$: time squares with growth
  - $O(2^n)$: Time grows as the power of size
## Big-Oh

In general, $O(n)$ is not possible for sorting, but it depends on data.

<table>
<thead>
<tr>
<th>Sort</th>
<th>10x</th>
<th>Big-Oh</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection (index)</td>
<td>100x</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Selection (STL)</td>
<td>100x</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>100x</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion (STL)</td>
<td>100x</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>C++ builtin</td>
<td>20x</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>
A generalization

- We saw the insertion sort appeared to be faster, but its growth rate is still $O(n^2)$
- Clearly Big-Oh is not the whole story, but it does say something about which algorithm is better in a big-picture way
Asymptotic Growth

- $O(1)$
- $O(n)$
- $O(n^2)$
- $O(n^3)$
There are many sorts...

- The fastest are variation on divide-and-conquer
  - QuickSort
  - Merge Sort
  - Shell Sort
  - Radix Sort
- Shell Sort has restrictions, but is often the fastest.
- Otherwise QuickSort is fastest
- Merge Sort works well when data doesn’t fit in memory
QuickSort

- Pick a value, the pivot
- Place everything smaller than pivot in one pile and everything bigger in another
  - Pivot is in its correct location (between the piles)
- Repeat on each of the two new piles (using recursion) until “small enough”
- $O(n \log n)$
Radix Sort

- Counterintuitive. Sort numbers first by least significant digit, then next, till sorted
- You use Radix sort to sort a deck of cards
  - Assume suits have a value (Clubs = 1, Diamonds = 2, Hearts = 3, Spades = 4)
  - Sort by rank first (ignore suit)
  - Now sort by suit
- $O(n*k)$
Merge Sort

- Divide the items in half
- Sort each half (recursively)
- Merge the two sorted halves (like a zipper)
- $O(n \log n)$
Search Algorithm

- Given a list and a value, find where the value is in the list (its index)
Linear Search

- Work through the list from beginning to end, checking each element
Linear Search, $O(n)$

```c
int stl_find(long ary[], size_t sz, long target) {
    auto itr = find(ary, ary+sz, target);
    if (itr == ary + sz)
        return -1;
    return itr - ary;
}
```
Better Search

- How about divide and conquer?
Binary Search

- Assume items are sorted
- Try the middle item
  - If value is less
    - Look in first half
  - else
    - Look in second half
- Repeat
STL Binary Search, $O(\log n)$

```c
int stl_binary(long ary[], size_t sz, long target) {
    // sort(ary, ary + sz);
    auto itr = lower_bound(ary, ary+sz, target);
    if (itr != ary + sz)
        return itr - ary;
    return -1;
}
```
int loop_binary(long ary[], size_t sz, long target) {
    size_t low = 0, high = sz-1;
    while (low < high) {
        size_t mid = (low + high) / 2;
        if (ary[mid] < target)
            low = mid + 1;
        else if (ary[mid] > target)
            high = mid - 1;
        else
            return mid;
    }
    return -1;
}
Timing: Search ints in ms

<table>
<thead>
<tr>
<th></th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
<th>Time vs x10 data</th>
</tr>
</thead>
<tbody>
<tr>
<td>STL Linear</td>
<td>0.027</td>
<td>0.207</td>
<td>1.992</td>
<td>20.0</td>
<td>~10x</td>
</tr>
<tr>
<td>Index Linear</td>
<td>0.028</td>
<td>0.247</td>
<td>2.458</td>
<td>24.6</td>
<td>~10x</td>
</tr>
<tr>
<td>Binary (sorted)</td>
<td>0</td>
<td>0.001</td>
<td>0.003</td>
<td>0.004</td>
<td>~1x</td>
</tr>
<tr>
<td>Binary (not sorted)</td>
<td>0.763</td>
<td>7.259</td>
<td>141</td>
<td>2,179</td>
<td>~20x</td>
</tr>
</tbody>
</table>
Big-Oh

- Linear is $O(n)$
  - STL approach is a little faster
- Binary is $O(\log(n))$ but assumes it is working on a sorted list
  - If the sort time is included, quite a bit slower
  - If the array is already sorted (for some reason), binary is a great choice