Big Oh notation

efficiency and sorting examples

• 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 as 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

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: its proper spot.
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 unsorted part: its proper spot.
4. Now the first two elements are sorted.
5. Repeat step 3 until done with list.
We want to sort array to be 2, 4, 6, 8.
Find the minimal element (2).
Swap the minimal with the first (2 & 8).
Fix the first position (2); repeat on the rest (e.g. find the minimum of 8, 6, 4; ...)

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){
    long *element;
    long temp;
    for(int i=0; i<sz; i++){
        element = min_element((ary+i), (ary+sz));
        swap(ary[element-ary], ary[i]);
    }
}
```

```c
void loop_selection(long ary[], size_t sz){
    size_t min_index;
    long temp;
    for(int i=0; i<sz; i++){
        min_index = i;
        for(int j=i+1; j<sz; j++)
            if (ary[j] < ary[min_index])
                min_index = j;
        // swap(ary[i], ary[min_index]);
        temp = ary[i];
        ary[i] = ary[min_index];
        ary[min_index] = temp;
    }
}
```

Just do it with indices and loops
Insertion sort

The idea:
• 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 in the unsorted side
• place it in its proper side on the left
  • unsorted down by one
  • relatively sorted up by one

Insertion Sort

sorted unsorted
ary 8 2 6 4
    0 1 2 3
ary 2 8 6 4
    0 1 2 3
ary 2 6 8 4
    0 1 2 3
ary 2 4 6 8
    0 1 2 3

insert 2 to its sorted position
insert 6 to its sorted position
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 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)

ary 9 5 4 1 7 0 3 2
    +2 +4 +6
• 7(+4) becomes 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

ary 9 5 4 1 7 0 3 2
9 5 7 0 4 1 3 2
**Loop Approach**

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

**example**

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

**Insertion Sort (STL)**

```c
example

1 3 4 5

(sorted) 1 3 4 5

2 7 6

i = 4 (working on 2)

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

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

1 3 4 5 2 7 6

note this half open range is sorted!

upr = ary+i

upr = ary+i+1

1 2 3 4 5

2 now inserted, i up by 1, 7 next

```
### 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 ints

<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>1260.67ms</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>1000</th>
<th>10000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection (index)</td>
<td>0.317</td>
<td>30.211</td>
<td>2991.53</td>
</tr>
<tr>
<td>Selection (STL)</td>
<td>0.351</td>
<td>25.719</td>
<td>2610.0</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>0.125</td>
<td>10.238</td>
<td>1260.67</td>
</tr>
<tr>
<td>Insertion (STL)</td>
<td>0.107</td>
<td>7.508</td>
<td>791.65</td>
</tr>
<tr>
<td>C++ builtin</td>
<td>0.036</td>
<td>0.425</td>
<td>15.476</td>
</tr>
</tbody>
</table>

### 10x data vs change in time

- Selection (index): ~100x time
- Selection (STL): ~100x time
- Insertion Sort: ~100x time
- Insertion (STL): ~100x time
- C++ builtin: ~20x time
BigO

- When comparing algorithms, computer scientists use BigO 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).

Algorithm complexity is written as $O(\cdot)$, where $\cdot$ 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

<table>
<thead>
<tr>
<th>Sort</th>
<th>10x</th>
<th>n</th>
<th>bigO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection (index)</td>
<td>100x</td>
<td>$n^2$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Selection (min)</td>
<td>100x</td>
<td>$n^2$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>100x</td>
<td>$n^2$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion (STL)</td>
<td>100x</td>
<td>$n^2$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>C++builtin</td>
<td>20x</td>
<td>$n$</td>
<td>$O(n \times \log(n))$</td>
</tr>
</tbody>
</table>

in general, $O(n)$ not possible for a sort but it depends on the data

A generalization

We saw that insertion sort appeared to be faster, but its growth rate is still $n^2$.

Clearly Big-Oh is not the whole story, but it does say something about, in a big picture way, which algorithm is better.
There are many sorts…

The fastest are variations on divide-and-conquer:

- QuickSort
- Merge Sort
- Shell Sort
- Radix Sort

Shell has restrictions, but is often the fastest. Otherwise Quicksort is fastest.

Merge works well when data doesn’t fit in memory.

QuickSort

1. Pick a value, the pivot.
2. Place everything smaller than pivot in one pile and everything bigger in another.
   1. pivot is in its correct location
3. Repeat on each of the two new piles 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.

1. Assume suits have a value (Diamonds=1, Clubs=2, etc.)
2. Sort by value first (ignore suit)
3. Now sort by suit
\[ O(nk) \]

(In this case \( n \) is suits and \( k \) is cards in suit)
1. Divide the items in half.
2. Sort each half.
3. Merge the two sorted halves -- like a zipper.
   You can keep repeating at step 2 by further dividing in half.
   \(O(n \log n)\)

**Sort, Search, Big O**

**Linear Search, for value in list**

Work through the list from beginning to end checking each element.

**Sort, Search, Big O**

**Linear search, \(O(n)\)**

```cpp
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;
}
```

**Sort, Search, Big O**
Better search?
How about divide-and-conquer?

Binary Search for value (divide-and-conquer)
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)
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, mid;
  while(low < high){
    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</td>
<td>0.028</td>
<td>0.247</td>
<td>2.458</td>
<td>24.6</td>
<td>~10x</td>
</tr>
<tr>
<td>Linear</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary</td>
<td>0</td>
<td>0.001</td>
<td>0.003</td>
<td>0.004</td>
<td>~1x</td>
</tr>
<tr>
<td>(already sorted)</td>
<td>0.763</td>
<td>7.259</td>
<td>141</td>
<td>2,178</td>
<td>~20x</td>
</tr>
<tr>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(not sorted)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Big O**

Linear is $O(n)$
- STL approach is a little faster
- Binary is $O(\log(n))$ but it 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.

**Today**

- Timing
- Sorting
- Searching
- Big O