PERFORMANCE

EFFICIENCY

SEARCHING

SORTING
WHAT IS PERFORMANCE?

• Since the early days in computing, computer scientists have concerned themselves with improving hardware, software, visualizations, etc.

• Performance can mean many different things.

• "The economy of human time is the next advantage of machinery in manufactures." – Charles Babbage
EXAMPLES OF PERFORMANCE

• Fewest computations
• Smaller memory usage
• Faster computations
• Improving accuracy of computations

• How we achieve these
  • Better algorithms
  • Better hardware
  • Better languages
WHY DO WE CARE?

• We want to solve real problems (large) in real time
BIG-OH COMPLEXITY

• We will focus our study of performance on time as a metric of performance.

• We can measure time experimentally like a stopwatch in our programs:
  
  ```java
  long start = System.nanoTime();
  //run algorithm
  long stop = System.nanoTime();
  double time = (stop - start)/1e9;
  ```

• We can measure time theoretically with big-oh analysis – an approximation technique for quantifying the time an algorithm takes.
BIG-OH COMPLEXITY

• A function $f(n)$ is $O(g(n))$ (pronounced "big-oh") if there exists constants $c$, and $n_0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$
  • $f(n)$ – real time taken for an algorithm. This is what we want to approximate
  • $g(n)$ – a function that "approximates" $f(n)$, more precisely it is an upper bound to $f(n)$

• We use this, as it describes how long an algorithm will take to compute as the problem size ($n$) increases

• To determine – count the operations
COMMON BIG-OH FUNCTIONS

• Logarithmic – $O(\log n)$

• Linear – $O(n)$
  • Example: searching for the minimum in an array. We must "look at" all $n$ elements of an array

• Linearithmic – $O(n \log n)$

• Quadratic – $O(n^2)$
SHAMELESS PLUG FOR CMSC 221

• This class is not about how we come up with these equations, or how we design better algorithms. For continued information, continue on in CS coursework

• In this class, I want you to have an intuitive feel of what big-oh means through a few algorithms

• In this class, understand the algorithms I present, but I do not expect you to come up with it yourself
LETS EXPLORE THESE CONCEPTS

• Case study on Searching
  • Linear Search
  • Binary Search

• Case study on Sorting
  • Bubble Sort
  • Selection Sort
  • Merge Sort
WAIT…HOW DO WE DO EXPERIMENTS?

• We vary the size of the data (usually by powers of two), so test on
  \( n = 2^1, 2^2, \ldots, 2^d \)

• Repeat each experiment numerous times to:
  • Get an accurate time for operations faster than 1 microsecond (usually one tick of the clock)
  • Average timing considering other tasks running on the computer

• Pseudocode

```
1. for \( N \leftarrow 2^1 \ldots 2^d \) do
2.   Setup before timing
3.   \( start \leftarrow \text{time()} \)
4.   for \( k \leftarrow 0 \ldots \text{repeats} \) do
5.     experiment()
6.   \( stop \leftarrow \text{time()} \)
7.   output(\( \frac{\text{start-stop}}{\text{repeats}} \))
```
CASE STUDY OF SEARCHING
**LINEAR SEARCH**

- **Pseudocode**

  **Input:** Array `arr`, Key `k`
  **Output:** `true` if `arr` contains `k`, `false` otherwise
  1. for each `a ∈ arr` do
  2.   if `a = k` then
  3.     return `true`
  4. return `false`

- **Complexity?**

  - Linear – $O(n)$
  - Reasoning – The search might have to visit each of the $n$ elements contained in the array.
  - Note – it doesn’t matter if the first element is equal to the key, that is a special case. On average we must search $\frac{n}{2}$ elements. Additionally, we don’t care about a specific size, we are interested in performance as the size tends to infinity.
CAN WE DO BETTER?

- Computer scientists always ask this kind of question, *can we do better?*
- Well in general...no, this is about the best we can do with searching.
- Computer scientists then ask a follow-up questions, *can we do better in special cases?*
- Yes! If we knew the input was sorted we could do much better.
BINARY SEARCH

• Pseudocode

Input: Sorted array arr, Key k
Output: true if arr contains k, false otherwise

1. low ← 0
2. high ← arr.length − 1
3. while lo ≤ hi do
4. mid ← \frac{high+low}{2}
5. if k < arr[mid] then
6. high ← mid − 1
7. else if k > arr[mid] then
8. low ← mid + 1
9. else
10. return true
11. return false
BINARİY AİLE

• How it works?
• Key is 7
BINARY SEARCH

• Complexity?
  • Logarithmic – $O(\log n)$
  • Reasoning – in each iteration of the loop, we eliminate half of the indices as possible cells to hold the key. The number of times you can repeatedly divide a number by 2 is the definition of a logarithm
  • Note – I am loose on the base of the logarithm. If you feel more comfortable with one, it will always be base 2. However, in big-oh complexity the base doesn't matter. See me after class if you would like a proof.
EXPERIMENT SEARCHING

• Download Search.java from the course website. It contains an experiment ready to go comparing the different searches. Let's go through the file to ensure we understand each component.

• Run the file, open up the csv file in Microsoft Excel

• Make a line scatter plot of the size vs the time of the methods
  • Convert to a log-log plot to get a better picture of the data
CONCLUSION

• A smaller complexity drastically affects runtime
• $O(\log n)$ is much faster than $O(n)$
CASE STUDY OF SORTING
BUBBLE SORT

• Pseudocode

Input: Array arr
Output: Sorted array
1. for \( i \leftarrow 1 \ldots \text{arr.length} \) do
2. \hspace{1em} for \( j \leftarrow 0 \ldots \text{arr.length} - i \) do
3. \hspace{2em} if \( \text{arr}[j] > \text{arr}[j + 1] \) then
4. \hspace{3em} swap(\text{arr}, \ j, \ j + 1)

• Complexity

• Quadratic – \( O(n^2) \)
• Reasoning – There are \( n \) passes over the array, in each pass \( n \) elements are visited and possibly swapped. \( n \times n = n^2 \)

\[ 6 \ 5 \ 3 \ 1 \ 8 \ 7 \ 2 \ 4 \]
CAN WE DO BETTER?

• Computer scientists always ask this kind of question, *can we do better?*
• Identify the weakness here, bubble sort swaps too much
• Can we fix it?
SELECTION SORT

• Pseudocode

**Input**: Array arr

**Output**: Sorted array

1. for $i \leftarrow 0 \ldots \text{arr.length} - 2$ do
2. \hspace{1em} $\text{min} \leftarrow i$;
3. for $j \leftarrow i \ldots \text{arr.length} - 1$ do
4. \hspace{2em} if $\text{arr}[j] < \text{arr}[\text{min}]$ then
5. \hspace{3em} $\text{min} \leftarrow j$
6. \hspace{1em} swap(arr, i, min)

• Complexity?

- Quadratic – $O(n^2)$
- Reasoning – In each iteration of the outer loop, we must find the minimum in the rest of the array, and we swap this minimum into place. Doing this $n$ times, takes in total $O(n^2)$ operations.
CAN WE DO BETTER?

• This was not satisfying, bubble sort and selection sort have the same complexity, even though selection sort is a much nicer idea (and performs better in practice, will see soon)

• Computer scientists always ask this kind of question, can we do better?

• Maybe we can try a radically different idea
MERGE SORT

- Split the array in half
- Sort each half recursively
- Merge the two back together
MERGE SORT

• Pseudocode Sort

**Input:** Array \( arr \)

**Output:** Sorted array

1. if \( arr \).length < 2 then return
2. \( l, r \) ← split \( (arr) \)
3. MergeSort \( (l) \)
4. MergeSort \( (r) \)
5. \( arr \) ← merge \( (l, r) \)

• Pseudocode Merge

**Input:** Sorted arrays \( l \) and \( r \)

**Output:** Sorted array \( arr \)

1. \( arr \) ← newArray \( (l \).length, \( r \).length \)
2. \( i \) ← 0; \( j \) ← 0; \( k \) ← 0
3. while \( i < l \).length \&\& \( j < r \).length do
4. if \( l[i] < r[j] \) then
5. \( arr[k] \) ← \( l[i] \); \( k \) ← \( k+1 \); \( i \) ← \( i+1 \)
6. else
7. \( arr[k] \) ← \( l[j] \); \( k \) ← \( k+1 \); \( j \) ← \( j+1 \)
8. while \( i < l \).length do
9. \( arr[k] \) ← \( l[i] \); \( k \) ← \( k+1 \); \( i \) ← \( i+1 \)
10. while \( j < r \).length do
11. \( arr[k] \) ← \( l[j] \); \( k \) ← \( k+1 \); \( j \) ← \( j+1 \)
12. return \( arr \)
MERGE SORT

• Complexity?
  • Linearithmic – $O(n \log n)$
  • Reasoning – At each iteration of the recursive function we split the array in half and merge it back together. This is $n$ work. Then we do this same amount of work at each level of the recursion tree. Since we split in half repeatedly, there are a logarithmic number of levels. Thus – $n$ work on $\log n$ levels is $O(n \log n)$

<table>
<thead>
<tr>
<th>depth</th>
<th>#seqs</th>
<th>size</th>
<th>Cost for level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>$n/2$</td>
<td>$n$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$i$</td>
<td>$2^i$</td>
<td>$n/2^i$</td>
<td>$n$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$\log n$</td>
<td>$2^{\log n}$ = $n$</td>
<td>$n/2^{\log n}$ = 1</td>
<td>$n$</td>
</tr>
</tbody>
</table>
EXPERIMENT SORTING

• Download Sort.java from the course website. It contains an experiment ready to go comparing the different searches. Let's go through the file to ensure we understand each component.

• Run the file, open up the csv file in Microsoft Excel

• Make a line scatter plot of the size vs the time of the methods
  • Convert to a log-log plot to get a better picture of the data
CONCLUSION

• Two algorithms can have the same complexity, but different actual performance
  • We need to experiment on our data

• Smaller complexity will always beat an optimized higher complexity
  • However, note that this doesn't necessarily apply to small values of $n$
  • Lesson – choosing an appropriate algorithm requires understanding the size of your data
ALGORITHM SUMMARY

• Searching
  • Linear Search – linear time or $O(n)$
  • Binary Search – logarithmic time or $O(\log n)$

• Sorting
  • Bubble Sort – quadratic time or $O(n^2)$
  • Selection Sort – quadratic time or $O(n^2)$
  • Merge Sort – linearithmic time or $O(n \log n)$