CSE1030 – Introduction to Computer Science II

Lecture #21

Searching and Sorting

Goals for Today

- Theoretical Goals:
  - Introduction to "Theory of Computing"
    - Concept of "Big-O" Notation
    - Complexity
- Practical:
  - Searching and Sorting
    - What are our options?
    - How do we decide what the best options are?

CSE1030 – Lecture #21

- Searching: Linear Search (Unordered List)
- Complexity and the "Big-O"
- Searching: Binary Search (Ordered List)
- Bubble Sort
- Selection Sort
- Insertion Sort
- Quicksort
- Mergesort
- We're Done!

Searching

- Searching is a common problem we often face when writing programs
- The question is, how best to find an item stored in a collection?
- Although the particular data (or Object) we might be looking for could be just about anything, the searching problem itself usually looks about the same

Searching Terminology

- "Data records" typically contain several fields
- The field(s) we want to search on are called the "**key**"

```
public class Person
{
   // attributes
   private String name;
   private int  age;
   private long bankAccount;
   private int  shoesize;
   etc. ...
```

If I'm searching for a Person by **Name**, then Name is the **Key**

Other fields could be the key instead (or as well), it depends upon what we're searching for

Searching Example:

What if we have an array of data, like this:

- What's the best way to find an element?
- (To keep the example simple, let's just use integers)

```
int[] array = {
   10,   8,  75,  60,  20,
    4,  86,  91,  81,  32,
   37,  84,   5,  74,  42,
   59,   2,  95,  22,  31,
   ...
   58,  27,  40,  88,  65,
   62,  68,  64,  73,  55,
   56,  18,  54,  89,  17,
   23,  63,  49,  14,  33,
    4,  36,  19,  78,  45,
};
```

Linear Search

```
// we want to find this
int key = 44;

boolean found = false;

for(int i = 0; i < array.length; i++)

   // we found it!  :-)
   if(key == array[i])
   {
      found = true;
      System.out.println("We found "
            + key + " at index " + i);
      break;
   }

// we didn't find it  :-(
if(!found)
   System.out.println("We didn't find "
            + key);
```

- Loop through the array searching for the item of interest
- On average we have to check ½ of the slots in the array to find the element of interest

Analysis of Linear Search

10	8	75	60	20	4	86	91	81	32	37	84	5	…

- How long does it take to find our number?
- We could get lucky if the key is near the front
- Otherwise we may have to search all the way to the end of the array
  - This is called the "**Worst Case**", here the worst case = **n** comparisons
- On average we would expect to have to search about half of the array
  - This is called the "**Average Case**", here the average case = ½ **n** comparisons

How Long does each Comparison Take?

```
// we want to find this
int key = 44;

boolean found = false;

for(int i = 0; i < array.length; i++)

   // we found it!  :-)
   if(key == array[i])
   {
      found = true;
      System.out.println("We found "
              + key + " at index " + i);
      break;
   }

// we didn't find it  :-(
if(!found)
   System.out.println("We didn't find "
              + key);
```

- Although it would be tough to figure out exactly, we can estimate how long it would take to perform this search…

How Long does each Comparison Take?

```
...

for(int i = 0; i < array.length; i++)

   // we found it!  :-)
   if(key == array[i])
   {
      found = true;
      System.out.println("We found "
              + key + " at index " + i);
      break;
   }

...
```

1. t(i compare)
2. t(i increment)
3. t(key compare)
4. t(loop)

- **Worst Case** time = **n × (t1 + t2 + t3 + t4)**
- **Average Case** time = ½ **n × (t1 + t2 + t3 + t4)**

"Big-O" Notation

- The problem is that those times (**t1**, **t2**, **t3**, and **t4**) all depend upon:
  - The hardware (processor clock rate & memory access speed)
  - The language (C is faster than Java)
  - On more complicated algorithms, the skill of the programmer
- To have a fair comparison of the algorithm we have to leave all of those "**t**" terms out
  - Because right now we're really interested in how good the algorithm is, not how good the hardware or language is
- This is the basis of "Big-O" notation:
  - **Worst Case** time = **n × (t1 + t2 + t3 + t4)**, but we write: Worst Case is ***O*(n)**
  - **Average Case** time = ½ **n × (t1 + t2 + t3 + t4)**, we write: Average Case is ***O*(n)**

CSE1030 – Lecture #21

- Searching: Linear Search (Unordered List)
- Complexity and the "Big-O"
- Searching: Binary Search (Ordered List)
- Bubble Sort
- Selection Sort
- Insertion Sort
- Quicksort
- Mergesort
- We're Done!

"Big-O" Notation

- The idea of "Big-O" notation is to provide an idea of the relative time-efficiency of an algorithm
  - We are also worried about memory ("space-efficiency"), but not as much as time-efficiency
- As we just saw, we remove factors that depend only upon the particular implementation (processor, language)
- Terminology:
  - An algorithm like the linear search we just saw, which is ***O*(n)**, we would say is "Order **n**"

Common Time Complexities

BETTER

- ***O*(1)** constant time
- ***O*(log n)** log time
- ***O*(n)** linear time
- ***O*(n×log n)** log linear time
- $O(n^2)$ quadratic time
- $O(n^3)$ cubic time
- $O(2^n)$ exponential time

WORSE

Why are these functions in this order?

Why is Complexity Important?

- Complexity is important because it gives us a tool to quickly check whether a problem is solvable or not
- This is important because some problems cannot be solved (in a reasonable amount of time)
- … and it is not always obvious which problems are solvable and which aren't …

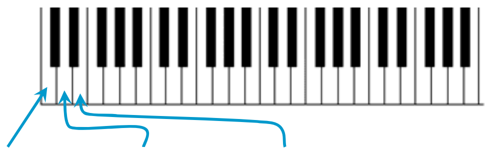
Problem: List the Piano Chords

- Can you write a program to list every possible chord that could be played on a (mechanical) piano?

Assuming no limit on the number of fingers

Piano Chord Analysis

- How many Piano Chords are there?
- There are 88 keys comprising a piano keyboard, and so there are 2^{88} possible chords



So how long would it take?

- Simply printing a list of all of the possible chords is order: ***O*(2ⁿ)**, where **n** = 88
- Let's assume you can print 1 chord every nanosecond…
- 2^{88} nanoseconds
  = 3.1×10^{26} nanoseconds
  = 3.1×10^{17} seconds
  = 5.2×10^{15} minutes
  = 8.6×10^{13} hours
  = …

So how long would it take?

- Simply printing a list of all of the possible chords is order: $O(2^n)$, where n = 88
- Let's assume you can print 1 chord every nanosecond…
- 2^{88} nanoseconds
  = 8.6×10^{13} hours
  = 3.6×10^{12} days
  = 9.8×10^{9} years ← That's 9.8 Billion Years

The Universe is only 13.7 Billion Years old, so this could take a while!

Conclusion?

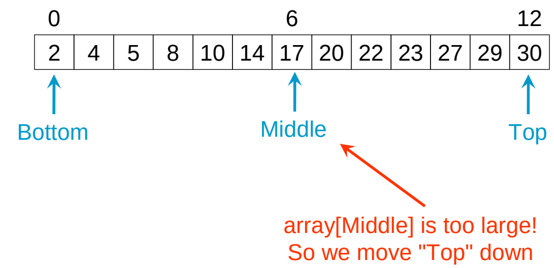
- Although the problem sounds (and is) simple, because the "complexity of our algorithm" is $O(2^n)$ we could never hope to see our program run to completion in our lifetime
- In theoretical terms our goal is to find algorithms and data structures that have a low complexity
- And in terms of applied computer science (i.e., working for "the man") our goal is to know enough about complexity to know which data structure from the API to use (*array* versus *linked-list*) and which sorting algorithm from the API to call to sort our data…

CSE1030 – Lecture #21

- Searching: Linear Search (Unordered List)
- Complexity and the "Big-O"
- Searching: Binary Search (Ordered List)
- Bubble Sort
- Selection Sort
- Insertion Sort
- Quicksort
- Mergesort
- We're Done!

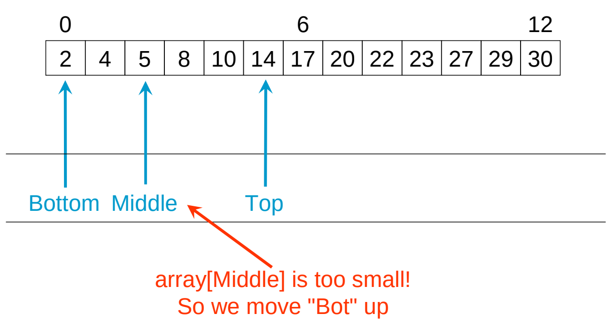
Binary Search

- What if the array of integers was sorted? We could "bisect" the array. Let's find 10….



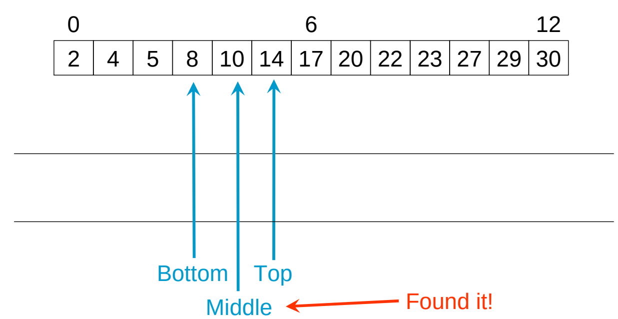
Binary Search

- What if the array of integers was sorted? We could "bisect" the array. Let's find 10….



Binary Search

- What if the array of integers was sorted? We could "bisect" the array. Let's find 10….



Binary Search Code

```
//the thing I am searching for
int key = 10;

// pointer to the top and bottom of
// the search range
int bot = 0;
int top = array.length -1;

// find the middle of the range
int mid = (top + bot)/2;

while(top > bot)
{
   // found it?  exit the loop
   if(key == array[mid])
      break;
```

```
   // mid is too high?  bring down the
   // top of the search range
   else if(key < array[mid])
      top = mid - 1;

   // mid is too low?  bring up the
   // bottom of the search range
   else
      bot = mid + 1;

   // calculate a new middle
   mid = (top + bot)/2;
}

// did we find it?
if(key == array[mid])
   System.out.println("We found " + key
         + " at index " + mid);

// we didn't find it  :-(
else
   System.out.println("We didn't find " + key);
```

Binary Search Analysis

- The binary search algorithm splits the search space in half every iteration
- This means in the worst case it will take **log(n)** steps to find the item
- So Binary Search is order: ***O*(log n)**

CSE1030 – Lecture #21

- Searching: Linear Search (Unordered List)
- Complexity and the "Big-O"
- Searching: Binary Search (Ordered List)
- Bubble Sort
- Selection Sort
- Insertion Sort
- Quicksort
- Mergesort
- We're Done!

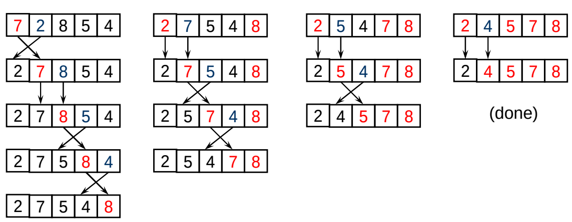
Sorting

- Having sorted data makes searching much faster
- So what options do we have for sorting?
- Let's start with the "Bubble Sort"

Bubble Sort

- Compare each element (except the last one) with its neighbor to the right
  - If they are out of order, swap them
- Then: Compare each element (except the last two) with its neighbor to the right
  - If they are out of order, swap them
- Then: Compare each element (except the last three) with its neighbor to the right
- Continue as above until you have no unsorted elements on the left

Example of Bubble Sort



Code for Bubble Sort

```
public static void bubbleSort(int[] a)
{
   int outer, inner;
   // counting down
   for (outer = a.length - 1; outer > 0; outer--)
   {
      // bubbling up
      for (inner = 0; inner < outer; inner++)
      {
         // if out of order...
         if (a[inner] > a[inner + 1]) {
            int temp = a[inner];          // ...then swap
            a[inner] = a[inner + 1];
            a[inner + 1] = temp;
         }
      }
   }
}
```

Analysis of Bubble Sort

- The outer loop is executed **n-1** times (call it **n**, that's close enough)
- Each time the outer loop is executed, the inner loop is executed
- The inner loop executes **n-1** times at first, linearly dropping to just once
- On average, inner loop executes about **n/2** times for each execution of the outer loop
- In the inner loop, the comparison is always done (constant time), the swap might be done (also constant time)
- result is $n \times n/2 \times k$, that is, $O(\frac{1}{2} n^2 \times k) \approx O(n^2)$

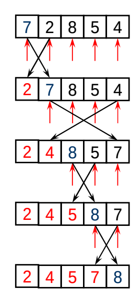
CSE1030 – Lecture #21

- Searching: Linear Search (Unordered List)
- Complexity and the "Big-O"
- Searching: Binary Search (Ordered List)
- Bubble Sort
- Selection Sort
- Insertion Sort
- Quicksort
- Mergesort
- We're Done!

Selection Sort

- Search elements 0 through n-1 and select the smallest
  - Swap it with the element in location 0
- Search elements 1 through n-1 and select the smallest
  - Swap it with the element in location 1
- Search elements 2 through n-1 and select the smallest
  - Swap it with the element in location 2
- Search elements 3 through n-1 and select the smallest
  - Swap it with the element in location 3
- Continue in this fashion until there's nothing left to search

Selection Sort



- The outer loop executes **n-1** times
- The inner loop executes about **n/2** times on average (from **n** to **2** times)
- Work done in the inner loop is constant (swap two array elements)
- Time required is roughly $(n-1)\times(n/2)$
- This is $O(n^2)$

Code for Selection Sort

```
public static void selectionSort(int[] a)
{
   // for every slot in the array
   for(int outer = 0; outer < a.length - 1; outer++)
   {
      // find the next smallest
      int min = outer;

      for(int inner = outer + 1; inner < a.length; inner++)
      {
         if(a[inner] < a[min])
            min = inner;
      }

      // and put it with where it should go
      int temp = a[outer];
      a[outer] = a[min];
      a[min] = temp;
   }
}
```

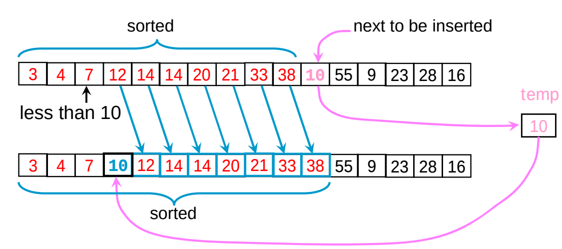
CSE1030 – Lecture #21

- Searching: Linear Search (Unordered List)
- Complexity and the "Big-O"
- Searching: Binary Search (Ordered List)
- Bubble Sort
- Selection Sort
- **Insertion Sort**
- Quicksort
- Mergesort
- We're Done!

Insertion Sort

- We have a counter that loops through the array, from bottom to top
- Each new element that the counter points to is inserted in order to the left of the counter
  - This means we have to shuffle elements up the array to make room for each newly sorted element
- Repeat for all elements

One Step of Insertion Sort



Code for Insertion Sort

```
public static void insertionSort(int[] a)
{
   // for every slot in the array
   for(int outer = 1; outer < a.length; outer++)
   {
      int newValue = a[outer];

      // find the location of the next element
      int inner;
      for(inner = 0; inner < outer; inner++)
      {
         if(newValue < a[inner])
            break;
      }

      // shuffle the elements up
      for(int shuffle = outer; shuffle > inner; shuffle--)
        a[shuffle] = a[shuffle-1];

      // put the value in its spot and move on
      array[inner] = newValue;
   }
}
```

Analysis of Insertion Sort

- Runs once through the outer loop, inserting each of **n** elements
- On average, there are **n/2** elements already sorted
- The inner loop looks at (and moves) half of these (this gives a second factor of **n/4**)
- So the time required for insertion sort to complete sorting the array of **n** elements is proportional to ¼ n^2
- Discarding constants, insertion sort is $O(n^2)$

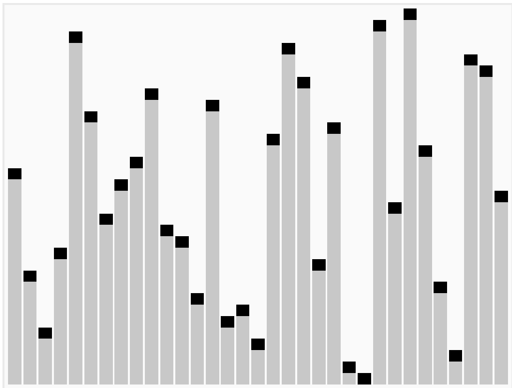
CSE1030 – Lecture #21

- Searching: Linear Search (Unordered List)
- Complexity and the "Big-O"
- Searching: Binary Search (Ordered List)
- Bubble Sort
- Selection Sort
- Insertion Sort
- Quicksort
- Mergesort
- We're Done!

QuickSort

- Quicksort is one of the fastest sorting algorithms known
- It is naturally a recursive algorithm
- The idea is:
  - Pick any element, and call it "**the pivot**"
  - Re-order the list (in 1 pass) so that all values less than the pivot come before it in the array, and all larger values come after it
  - Recursively sort the two sub-lists (of elements that are smaller than the pivot, and elements that are larger)

Quicksort Visualisation



Code for QuickSort (1/4)

```
/**
 * This is the array of integers that is to be sorted
 */
public static int[] array;


/**
 * This is a little convenience function that swaps the
 * contents of two slots in the array
 *  i.e.:  array[x] <--> array[y]
 */
public static void exchange(int x, int y)
{
   int temp = array[x];
   array[x] = array[y];
   array[y] = temp;
}
```

(2/4)

```
/**
 * This is the quicksort algorithm.
 */
public static void quicksort()
{
   quicksort(0, array.length-1);
}

public static void quicksort(int rangeStart, int rangeEnd)
{
   // these are our indexes that we will use to search for
   // pairs of data in the array that need to be exchanged
   int start = rangeStart - 1;
   int pivot = rangeEnd + 1;

   // the pivot value - once we are done with this section of
   // the array, all of the values will have been sorted into
   // those that are less than this value, and those that are
   // greater than this value
   // (note that any value in our section of the array will do
   // so we'll choose the value in the first slot)
   int pivotvalue = array[rangeStart];
```

(3/4)

```
   // this is the "pivot algorithm"
   // its purpose is to place all of the values that are
   // less than the pivotvalue to the left, and all of
   // the values that are larger to the right, with the
   // index of the pivot (the variable called "pivot"
   // denoting the place in the middle)
   while(true)
   {
      // loop down from the top looking for
      // a value that needs to be swapped
      do {
         pivot--;
      } while(array[pivot] > pivotvalue);

      // same thing, but coming up from the bottom
      do {
         start++;
      } while(array[start] < pivotvalue);
```

(4/4)

```
      // if we've found a pair to swap, then do it
      if(start < pivot)
         exchange(start, pivot);

      // otherwise, we're done this section of the array
      else
         break;
   }

   // sort the next two ranges
   // (the range below the pivot...)
   if(rangeStart < pivot)
      quicksort(rangeStart, pivot);

   // (... and the range above the pivot)
   if(pivot + 1 < rangeEnd)
      quicksort(pivot + 1, rangeEnd);
}
```

Analysis of Quicksort

- The analysis of Quicksort depends upon how lucky the algorithm gets with the pivot values
- If the pivots cause the array to be divided roughly equally every time, then Quicksort is ***O*(nlog n)**
- If the pivot values are not lucky, then the Quicksort is order ***O*(n2)**
- Although in practice things can be done to ensure that the pivots are chosen well
- And for large sets of data, Quicksort is one of the fastest sorting algorithms we have

CSE1030 – Lecture #21

- Searching: Linear Search (Unordered List)
- Complexity and the "Big-O"
- Searching: Binary Search (Ordered List)
- Bubble Sort
- Selection Sort
- Insertion Sort
- Quicksort
- Mergesort
- We're Done!

Merge Sort

1. Break the set to be sorted in half
2. Use recursion to sort each half
3. Merge the two sorted lists back together

- (For source code see Assignment #8)
- Merge sort works best with:
  - Data where sets can easily be re-ordered (like linked-lists)
  - Analysis:
    - Average Case: ***O*(n×log n)**
    - Worst Case: ***O*(n×log n)**

Sorting Summary

	Average	Worst Case
Bubble	O(n2)	$O(n^2)$
Selection	O(n2)	$O(n^2)$
Insertion	O(n2)	$O(n^2)$
Quicksort	$O(n \times \log n)$	$O(n^2)$
Mergesort	$O(n \times \log n)$	$O(n \times \log n)$

- Quicksort (or variations) are commonly used everywhere, because the worst case is avoidable
- Although it has a poor complexity, insertion sort is fast for very small data sets (small **n**)
- Mergesort is fastest for serially-accessible data

Sorting Summary

- We have covered only the most popular sorting algorithms here
- There are many many more
- But in practice you need to know only four algorithms: ***Insertion*** sort, ***Quicksort***, ***Mergesort***, and the ***HeapSort***
  - Heapsort uses a "Tree" data structure, which you won't cover until next year, and so we can't really discuss it in detail yet (although it's pretty cool, and it's about as fast as Quicksort, although its average case and worst case are both $O(n \times \log n)$).

CSE1030 – Lecture #21

- Searching: Linear Search (Unordered List)
- Complexity and the "Big-O"
- Searching: Binary Search (Ordered List)
- Bubble Sort
- Selection Sort
- Insertion Sort
- Quicksort
- Mergesort
- We're Done!