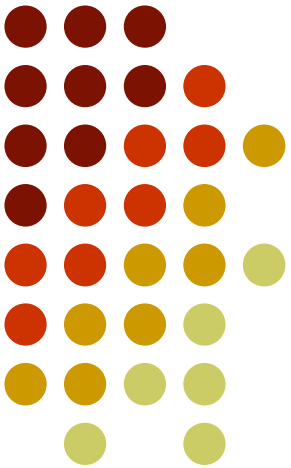
CMPT 225

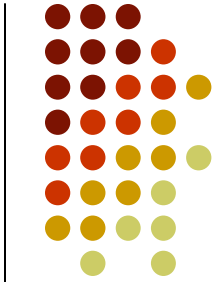
Sorting Algorithms

Algorithm Analysis:

Big O Notation

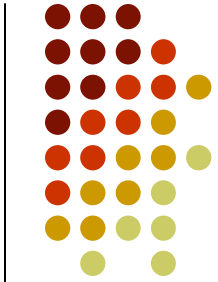


Objectives



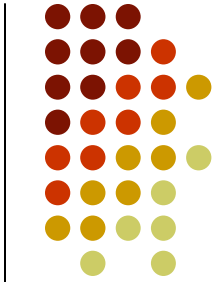
- Determine the running time of simple algorithms in the:
  - Best case
  - Average case
  - Worst case
- Sorting algorithms
- Understand the mathematical basis of O notation
- Use O notation to measure the running time of algorithms

Algorithm Analysis



- It is important to be able to describe the efficiency of algorithms
  - Time efficiency
  - Space efficiency
- Choosing an appropriate algorithm can make an enormous difference in the usability of a system e.g.
  - Government and corporate databases with many millions of records, which are accessed frequently
  - Online search engines
  - Real time systems (from air traffic control systems to computer games) where near instantaneous response is required

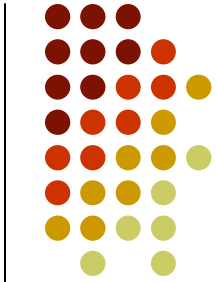
Measuring Efficiency of Algorithms



- It is possible to time algorithms
  - **`System.currentTimeMillis()`** returns the current time so can easily be used to measure the running time of an algorithm
  - More sophisticated timer classes exist

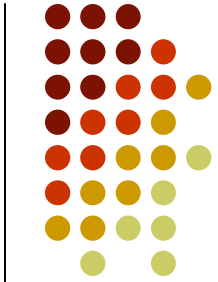
- It is possible to count the number of operations that an algorithm performs
  - Either by a careful visual walkthrough of the algorithm or by
  - Printing the number of times that each line executes (profiling)

Timing Algorithms



- It can be very useful to time how long an algorithm takes to run
  - In some cases it may be essential to know how long a particular algorithm takes on a particular system
- However, it is not a good general method for comparing algorithms
  - Running time is affected by numerous factors
    - CPU speed, memory, specialized hardware (e.g. graphics card)
    - Operating system, system configuration (e.g. virtual memory), programming language, algorithm implementation
    - Other tasks (i.e. what other programs are running), timing of system tasks (e.g. memory management)
    - Particular input used.

Cost Functions

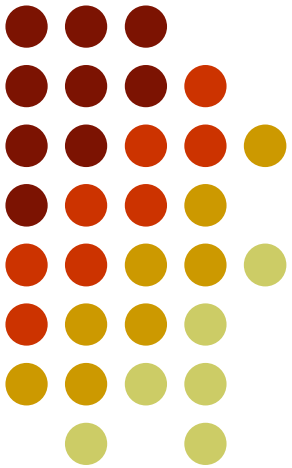


- Because of the sorts of reasons just discussed for general comparative purposes we will count, rather than time, the **number of operations** that an algorithm performs
  - Note that this does not mean that actual running time should be ignored!
- If algorithm (on some particular input) performs ***t*** operations, we will say that it runs in **time *t***.
- Usually running time ***t*** depends on the data size (the input length).
- We express the time ***t*** as a **cost function** of the data size ***n***
  - We denote the cost function of an algorithm A as $t_A()$, where $t_A(n)$ is the time required to process the data with algorithm A on input of size ***n***

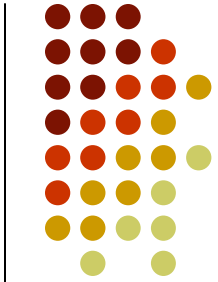
Best, Average and Worst Case

- The amount of work performed by an algorithm may vary based on its input (not only on its size)
  - This is frequently the case (but not always)
- Algorithm efficiency is often calculated for three, general, cases of input
  - Best case
  - Average (or “usual”) case
  - Worst case

Cost Functions of Sorting Algorithms



Simple Sorting

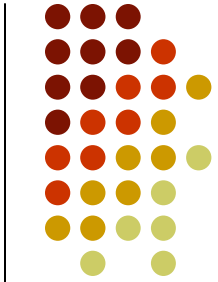


- As an example of algorithm analysis let's look at two simple sorting algorithms
  - Selection Sort and
  - Insertion Sort
- We'll calculate an approximate cost function for these sorting algorithms by analyzing exactly how many operations are performed by each algorithm
  - Note that this will include an analysis of how many times the algorithms perform loops

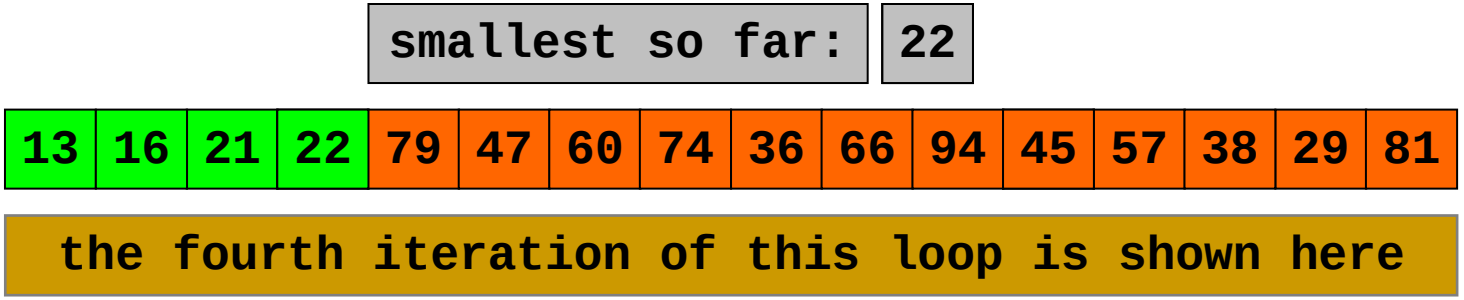
Comparing sorting algorithms

- Measures used:
  - The total number of operations (usually are not important)
  - The number of comparisons (most common; in Java comparisons are expansive operation)
  - The number of times an element is moved (in Java moving elements is cheap as references are always used, but in C++ can be expansive).

Selection Sort

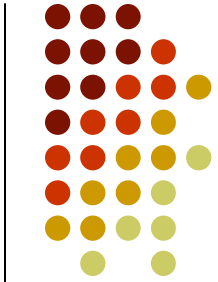


- while some elements unsorted:
  - Find the smallest element in the unsorted section; this requires all of the unsorted items to be compared to find the smallest
  - Swap the smallest element with the first (left-most) element in the unsorted section



Selection Sort Algorithm

```
public void selectionSort(Comparable[] arr) {
    for (int i = 0; i < arr.length-1; ++i) {
        int smallest = i;
        // Find the index of the smallest element
        for (int j = i + 1; j < arr.length; ++j) {
            if (arr[j].compareTo(arr[smallest])<0) {
                smallest = j;
            }
        }
        // Swap the smallest with the current item
        Comparable temp = arr[i];
        arr[i] = arr[smallest];
        arr[smallest] = temp;
    }
}
```

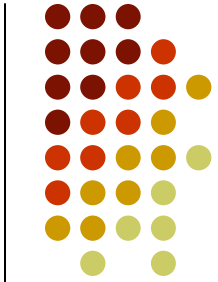


Shaded elements are selected;
boldface elements are in order.

Initial array:	29	10	14	37	13
After 1st swap:	29	10	14	13	**37**
After 2nd swap:	13	10	14	**29**	**37**
After 3rd swap:	13	10	**14**	**29**	**37**
After 4th swap:	**10**	**13**	**14**	**29**	**37**

A selection sort of an array of five integers (using the algorithm in the textbook, in which the first part of array is unsorted and the second (bold) is sorted. Instead of the smallest element we have to look for the biggest element.

Selection Sort: Number of Comparisons

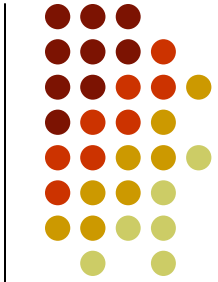


Elements in unsorted	Comparisons to find min
n	n-1
n-1	n-2
…	…
3	2
2	1
1	0
	n(n-1)/2

Selection Sort Analysis

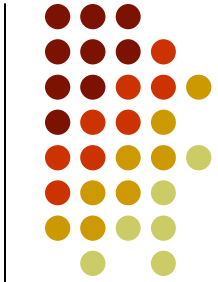
```
public void selectionSort(Comparable[] arr) {
    for (int i = 0; i < arr.length-1; ++i) {
    // outer for loop is evaluated n-1 times
        int smallest = i;  //n-1 times again
        for (int j = i + 1; j < arr.length; ++j) {
        // evaluated n(n-1)/2 times
            if (arr[j].compareTo(arr[smallest])<0) {
                // n(n-1)/2 comparisons
                smallest = j; //how many times? (*)
            }
        }
        Comparable temp = arr[i];  //n-1 times
        arr[i] = arr[smallest];    //n-1 times
        arr[smallest] = temp;      //n-1 times
    }
}
```

Selection Sort: Cost Function



- There is 1 operation needed to initializing the outer loop
- The outer loop is evaluated n-1 times
  - 7 instructions (these include the outer loop comparison and increment, and the initialization of the inner loop)
  - Cost is 7(n-1)
- The inner loop is evaluated $n(n-1)/2$ times
  - There are 4 instructions in the inner loop, but one (*) is only evaluated sometimes
  - Worst case cost upper bound: $4(n(n-1)/2)$
- Total cost: $1 + 7(n-1) + 4(n(n-1)/2)$ [worst case]
  - Assumption: that all instructions have the same cost

Selection Sort: Summary



- Number of comparisons:
  $n(n-1)/2$
- The best case time cost:
  $1 + 7(n-1) + 3(n(n-1)/2)$ *(array was sorted)*
- The worst case time cost (an upper bound):
  $\leq 1 + 7(n-1) + 4(n(n-1)/2)$
  (the real worst case time cost: $1+7(n-1)+3n(n-1)/2+n^2/4$)
- The number of swaps:
  $n-1$ [number of moves: $3(n-1)$]