CS 105 **TA Lecture Notes**

October 1, 2014

**Running Time – Searching and Sorting**

Important questions:

- How many steps?
- What is the worst case?
- How many steps do may need to check if your data is in there.

**Linear Search:**

- We go through data points one by one
- 'n' steps if there is n pieces of data
- Running time: O(n), it does not matter if we lose one /two when n is a million

**Binary Search:**

- Requires the data to be sorted
- Split the data to be searched in, in half every time
- Worst happens in worst case? 8 pieces of data -> 3 steps
- 16 pieces of data -> 4 steps
- 32 pieces of data -> 5 steps
- Hence doubling the data just takes 1 extra step
- $\log_2(n)$ steps or lg(n) (i.e. base 2 log) steps for n pieces of data, lg(n) is smaller than n, so the running time of a binary search is smaller than that of linear search
- So if we have sorted data, binary search is definitely faster than linear search
- Running time: O(lg(n))

**Selection sort:**

- Go to the first element and check it against all elements to see if it is the smallest. If it isn't, swap it with the smallest element.
- So after the first pass, smallest is at the correct position and data size to be sorted is reduced by 1
- Now start with the second element and go through all elements that are unsorted
- Hence
    - First pass n steps
    - Second pass (n-1) steps
    - Third pass (n-2) steps and so on until the last pass takes 1 step
    - Sum of numbers up to n and is given by n(n+1)/2
    - So running time is O(n^2) basically, since for large n, n^2 >> n

Example: (first pass)

**29**, 64, 73, 34, **20**,
20, **64**, 73, 34, **29**,
20, 29, **73**, **34**, 64
20, 29, 34, **73**, **64**
20, 29, 34, 64, 73

CS 105 **TA Lecture Notes**
October 1, 2014

**Bubble sort:**

- Start at the first element
- Check and swap each pair of elements if a swap is needed
- Now the last element is in the correct place – the element at the end, largest number bubbled it's way to the correct spot (hence called bubble sort)
- For the second step, we don't have to check for the last element
- Hence
    - First time n steps
    - Second time (n-1) steps
    - Third time (n-2) steps and so on until the last pass takes 1 step
    - Sum of numbers up to n and is given by n(n+1)/2
    - So running time is O(n^2) basically, since for large n, n^2 >> n

Example: (first pass)
**7, 5**, 2, 4, 3, 9
5, **7, 2**, 4, 3, 9
5, 2, **7, 4**, 3, 9
5, 2, 4, **7, 3**, 9
5, 2, 4, 3, **7, 9**
5, 2, 4, 3, 7, 9

**Merge sort:**

- Split the data into 2 parts successively until you reach 2 elements in every sub-array
- Sort each of the small arrays separately
- Start merging the sub-arrays and while merging, do it smartly
- After merging, we need to sort it again
- This is O(nlog(n)) algorithm

Example:
Consider the following array of numbers: **27 10 12 25 34 16 15 31**

divide it into two parts: **27 10 12 25 34 16 15 31**

divide each part into two parts: **27 10 12 25 34 16 15 31**

divide each part into two parts: **27 10 12 25 34 16 15 31**

merge (cleverly-!) parts: **10 27 12 25 16 34 15 31**

merge parts: `10 12 25 27 15 16 31 34`

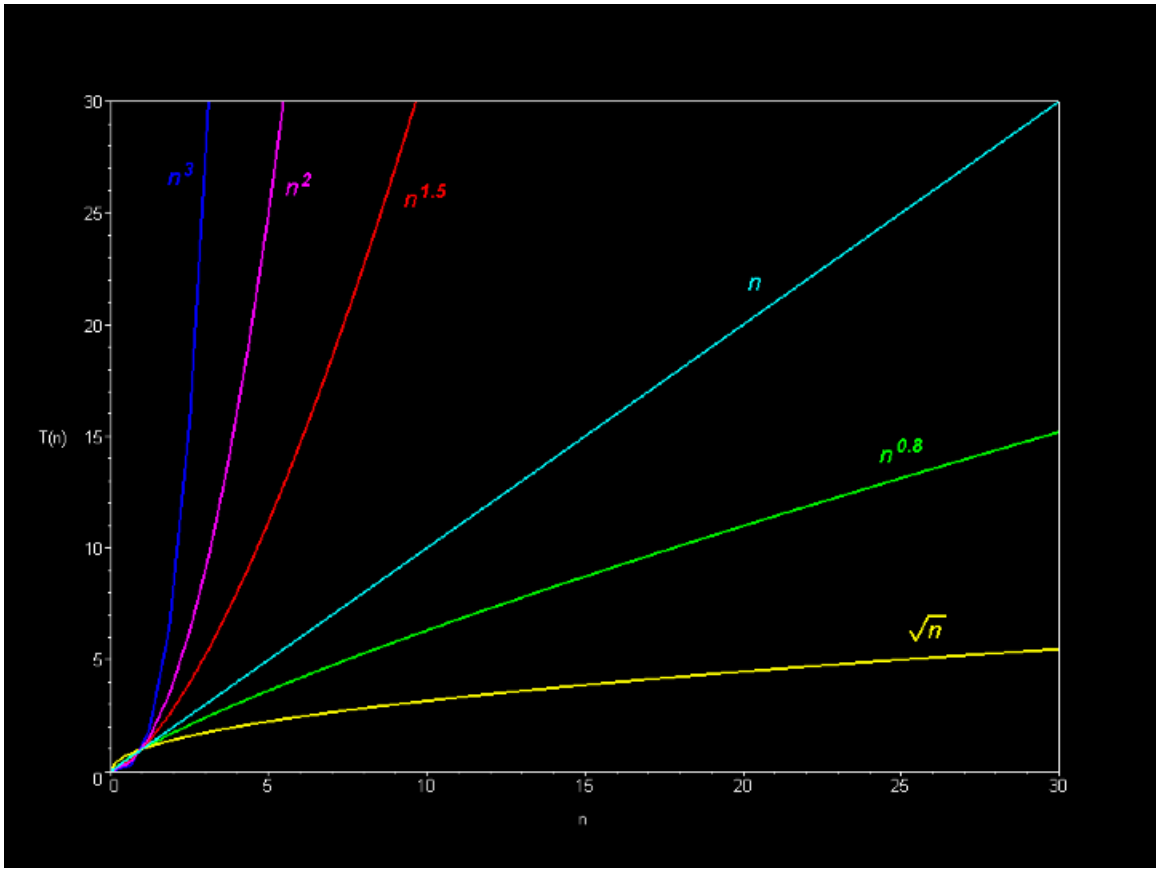
merge parts into one: `10 12 15 16 25 27 31 34`

**Running time:**

Hence, the running time of all these different sorts:

$$n + (n-1) + (n-2) + (n+3) \ldots\ldots = O(n^2)$$

**Explaining the graph:**




- Sorting takes much longer than searching on a sorted array
- The purple line in the graph represents $n^2$ which is very slow
- Binary search is way faster than linear search – Look at $lg(n)$ and $n$
- If we want to search the data only once, we can do linear search but if we have to search the data more than once, it is worth sorting the data to then use binary search

CS 105

# TA Lecture Notes

October 1, 2014

**Images**

How are images stored in the computer?

- Images are a grid of pixels
- 1 bit is allotted to every pixel for black (represented by 1) and white images (represented by 0) - Bitmap format

What about color images?

- Three primary colors – Red, Green and Blue
- Every color can be represented as a combination of these in different proportions
- Hence we can represent colors as Hexa-decimal (base 16 system 0 to 9 and a,b,c,d,e,f) digits like ff0000 – using 6 digits

Colors using hexadecimal system (two digits used for every color):

| Color – Hexa Decimal | % Red | % Green | % Blue | Color |
|---|---|---|---|---|
| ff0000 | 100 | 0 | 0 | Red |
| 00ff00 | 0 | 100 | 0 | Green |
| ffffff | 100 | 100 | 100 | White |
| 000000 | 0 | 0 | 0 | Black |
| ffff00 | 100 | 100 | 0 | Yellow |
| ff00ff | 100 | 0 | 100 | Magenta |

**Jsons in JavaScript (JS object notations) –**

'D5' is a poor way of saying 5 of diamonds. We could say that the card has rank of 5 and suit of diamonds Object makes that easier. Multiple data / variables can be placed in a single variable. Like 'card' object would have 'rank' and 'suit'.

It did not make sense to represent a card as a string completely. Hence we can have key and value pairs for a variable.

Examples:

```
var card = { rank: "5", suit: "Diamond"};

//where rank and suit are 'keys' which are just variable names
and then the value of that key.

var weather = {high: 72, low: 49};

var average = (weather.high+weather.low)/2;
```

Sample Function:

```
function distance(l1,l2)
{
     var dx = l2.x - l1.x; // change in x

     var dy = l2.y - l1.y; // change in y

     // distance formula from coordinate geometry
     var d = Math.sqrt((dx*dx)+(dy*dy));

     return d;
}

var l1 = {x:30 , y:50};
var l2 = { x:40 , y:-30  };
var d = distance(l1,l2); //returns 80.6
```