# CSCI 3110

Introduction to Algorithms

*He*

**Alex Safatli**

Monday, September 9, 2013

# Contents

# Algorithm Analysis

## Example: Insertion Sort

The pseudocode for this algorithm is presented below.

```
  INSERTION-SORT(A[1..n])
1 for j <-- 2 to n
2   key <-- A[j]
3   i   <-- j-1
4   while i>0 and A[i]>key
5     A[i+1] <-- A[i]
6     i      <-- i-1
7   A[i+1] <-- key
```

## Random Access Machine (RAM)

In this abstract model of a machine, algorithms are implemented as **programs** that possess a given input and generates a value(s) as output. In this model there is a *read-only input tape*: each cell of the tape stores one value — this can be imagined as an input tape used in the past as input to computers. In the same grain, there is also a *write-only output tape*. In the computer there is also *memory*; it is assumed that the memory is **unbound**. In other words,

- there is an unbound number of memory cells, each the length of a **word**,
- each cell can hold an integer or floating point value,
- there is a sequential execution of program instructions (guided by a *location counter*), and therefore,
- the **time complexity** (running time) is equal to the number of instructions executed and the **space complexity** is the number of memory cells accessed.

Under such a model, the **instructions** that are available, its **instruction set**, are comprised of instructions commonly found in real computers. These include:

- arithmetic operations: addition, substraction, multiplication, division, remainder, floor, and ceiling,
- data movement: load, store, copy,
- control: conditional (for loop, etc.) and unconditional (jump) branch

The assumption here is that EACH INSTRUCTION REQUIRES ONE UNIT OF TIME. This removes the detail of how fast the actual computer is. Instructions in real computers not listed above are considered by making reasonable assumptions. For instance:

- $x^y$ could require more than 1 unit of time to calculate,
- $2^k$ with $k$ as an integer with $\leq$ word size is simply a **bit shift**; this can be treated as 1 unit of time.

Limitations of this model are that there is:

- no concurrency; for this we consider parallel RAM (PRAM), and
- no memory hierarchy (external/internal, etc. – see external memory model).

## Analysis: Insertion Sort

We must consider the **input size** here: when the input size grows, so does the running time. This is easy to understand: sorting merely three numbers is much faster than sorting a billion records. Therefore, the **running time** is a *function of input size*. **Input size** tends to use reasonable parameters associated with the problem. For example, with sorting the input size is the array size while with a graph it is the number of vertices and number of edges.

Analyzing the pseudocode above, we must recall that each instruction takes one unit of time. We must therefore count the number of **primitive operations** underlying the pseudocode (at each line). Pseudocode uses whatever methods that clearly and concisely describe an algorithm: cannot always describe a line as one primitive operation. This number is multiplied by the number of iterations for loops: the number of primitive operations is multiplied by the number of times a given line is executed. Once added up, we have the running time.

For the insertion sort algorithm, the input size is $n$, the array size. Note also that each line can be comprised and represent the execution of a *constant* number of instructions. $C_i$, a constant, is the number of instructions a line $i$ takes. We can also define a value $t_j$, the number of times the while loop (at line 4) is executed for that value of $j$.

| Line | Cost | Times |
|---|---|---|
| 1 | $C_1$ | $n$ |
| 2 | $C_2$ | $n-1$ |
| 3 | $C_3$ | $n-1$ |
| 4 | $C_4$ | $\sum_{j=2}^{n} t_j$ |
| 5 | $C_5$ | $\sum_{j=2}^{n} t_j - 1$ |
| 6 | $C_6$ | $\sum_{j=2}^{n} t_j - 1$ |
| 7 | $C_7$ | $n-1$ |

Table 1: Costs line-by-line for insertion sort.

For the algorithm, therefore, we can describe the total running time, $T(n)$, as:

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4(\sum_{j=2}^{n} t_j) + C_5(\sum_{j=2}^{n} t_j - 1) + C_6(\sum_{j=2}^{n} t_j - 1) + C_7(n-1)$$

We can also take note that there is a range to the value of $t_j$: $1 \leq t_j \leq j$.

The **best case** of the algorithm is the *minimum value* of $T(n)$. The best case for this algorithm is when the array, `A`, is already sorted. The value of $t_j$ would then be 1 and the value of $T(n)$ would be:

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4(n-1) + C_5(0) + C_6(0) + C_7(n-1)$$
$$T(n) = (C_1 + C_2 + C_3 + C_4 + C_7)n - (C_2 + C_3 + C_4 + C_7)$$

Note that the values of $C$ are constant and therefore this is a polynomial of order 1, a linear function. This is order notation $\Theta(n)$.

**NOTE** The reading for this part of the course is 2.2; for the next lecture has reading involving chapter 3.

The **worst case** of the algorithm is the *maximum value*. In our context, this is when the array is reverse sorted. Here, $t_j = j$. Recall that $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$.

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4(\tfrac{n(n+1)}{2} - 1) + C_5(\tfrac{n(n+1)}{2}) + C_6(\tfrac{n(n+1)}{2}) + C_7(n-1)$$
$$T(n) = (\tfrac{C_4}{2} + \tfrac{C_5}{2} + \tfrac{C_6}{2})n^2 + (C_1 + C_2 + C_3 + \tfrac{C_4}{2} - \tfrac{C_5}{2} - \tfrac{C_6}{2} + C_7)n - (C_2 + C_3 + C_4 + C_7)$$

This is a *quadratic function* of $n$ and therefore the order notation is $\Theta(n^2)$.

As an aside, the **average case** of this algorithm involves each of the $n!$ permutations of `A` which are equally likely to occur. This is a lot more difficult to analyze; for an introductory lecture this will not be proven but the result is still the same: $\Theta(n^2)$. In this course, we will mainly focus on worst case analysis:

- the worst case analysis gives us an **upper bound** on the running time, guaranteeing the algorithm cannot perform worse than this case,
- the worst case can potentially happen often, and
- the average case is often as bad as the worst case (which is true for insertion sort) but this is not always true.

Let's take a look at a specific array case for this algorithm. Given that we would like to sort the SINs for all Canadians, which means we have an $n$ of approximately 34880000 and an $n^2$ of approximately $10^15$. If we were to calculate $\log n$, we are looking at approximately 25, and $n \log n$ would be approximately $10^9$. Given that a computer can perform $10^11$ instructions every second, an insertion sort algorithm ($n^2$) would take approximately $10^4$ seconds, about 3 hours, but an $n \log n$ operation (mergesort) would take $\frac{1}{100}$ seconds.

## Order (Rate) of Growth

By making more simplifying assumptions, we can describe our analyses of algorithms. In this course, we will use **asymptotic notation** comprising symbols O, $\Omega$, $\Theta$, o, and $\omega$.

$\Theta$-notation can be formally described by the following expression. $\Theta(g(n)) = \{f(n) :$ there exist positive constants $C_1$, $C_2$ and $n_0$ such that $0 \leq C_1 g(n) \leq f(n) \leq C_2 g(n)$ for all $n \geq n_0\}$. Note that this is a **set** and that $g(n)$ is an **asymptotically tight bound** function for $f(n)$. Therefore, we can describe $f(n) = \Theta(g(n))$ which is equivalent to $f(n) \in \Theta(g(n))$.

For an example, $20n^2 + 13n + 1 = 20n^2 + \Theta(n)$ which is the same as $20n^2 + 13n + 1 = 20n^2 + f(n)$ and $f(n) \in \Theta(n)$.

Using the above formalization of $\Theta(g(n))$, we can prove that $\frac{1}{2}n^2 + \log n = \Theta(n^2)$. Proof. We must determine constants $C_1$, $C_2$, and $n_0$ such that:

$$C_1 n^2 \leq \tfrac{1}{2}n^2 + \log n \leq C_2 n^2 \forall n \geq n_0$$

Or:

$$C_1 \leq \tfrac{1}{2} + \tfrac{\log n}{n^2} \leq C_2$$

Therefore, we could choose $C_1 = \frac{1}{4}$, $C_2 = \frac{3}{4}$ and $n_0 = 2$. Other choices exist. Finding one set of values is sufficient: it will prove at least one set of values exist.

Note that **asymptotically positive functions** only are positive ($> 0$) for all sufficiently large $n$.

O-notation can provide an **asymptotic upper bound**. This is by nature of it bounding a function from above. Formally, $\mathrm{O}(g(n)) = \{f(n) :$ there exist positive constants $C$ and $n_0$ such that $0 \leq f(n) \leq Cg(n)$ for all $n \geq n_0\}$.

For example, let us consider whether the following are $O(n^2)$. $n$, $n^2$, $3n^2 + 4n + 5$ are all but $n^3$ is not.

**NOTE** The running time of insertion sort is $\mathrm{O}(n^2)$. This means that the **worst case** running time is this.

Order (Rate) of Growth continued on next page...

If we describe an algorithm that runs in **polynomial time**, this means it runs in polynomial time if there exists a constant $k$ such that its worst-case running time is $O(n^k)$. For insertion sort, $k = 2$. Algorithms are attempted to be designed to be (at most) in polynomial time.

$\Omega$-notation, on the other hand, provides an **asymptotic lower bound**. It can be formalized $\Omega(g(n)) = \{f(n) :$ there exist positive constants $C$ and $n_0$ such that $0 \leq Cg(n) \leq f(n)$ for all $n \geq n_0\}$. For example, $n^2$, $n^2 \log n$ are both but $n$ is not $\Omega(n^2)$.

THEOREM. *$f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.*

For example, $\frac{1}{2}n^2 + \log n = \Theta(n^2) \iff \frac{1}{2}n^2 + \log n = O(n^2)$ and $\frac{1}{2}n^2 + \log n = \Theta(n^2)$.

o-notation is formally defined as $o(g(n)) = \{f(n) :$ for any positive constant $C$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < Cg(n)$ for all $n \geq n_0$. $f(n)$ grows more slowly than $g(n)\}$.

**NOTE** Reading for this lecture is 3.1. For next lecture, it is 4.1.

Realize that $f(n) = o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. For example, $2n = o(n^2)$ but $2n^2 \neq o(n^2)$.

$\omega$-notation involves an $f(n) = \omega(g(n))$ *iff* $g(n) = o(f(n))$. $\omega(g(n)) = \{f(n) :$ for any positive constant $C\exists$ a constant $n_0 > 0$ such that $0 \leq Cg(n) < f(n)$ for all $n \geq n_0\}$. Realize that $f(n) = \omega(g(n))$ *iff* $\lim_{n\to\infty} \frac{f(n)}{g(n))} = \infty$.

Properties of all this asymptotic behaviour include **transitivity** where $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \implies f(n) = \Theta(h(n))$. This also applies with the other notation (0, $\Omega$, o, and $\omega$). There is also the property of **reflectivity** where $f(n) = \Theta(f(n))$; this is true for O and $\Omega$ but not for o or $\omega$. **Symmetry** is where $f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$. **Transpose symmetry** is where $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$ or $f(n) = o(g(n)) \iff g(n) = \omega(f(n))$.

Tricks that can be used here include the following.

1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$
   - $c = 0$; $f(n) = o(g(n))$,
   - $c = \infty$; $f(n) = \omega(g(n))$, and
   - $0 < c < \infty$; $f(n) = \Theta(g(n)) = \Omega(g(n)) = O(g(n))$.
   - For example, given that $d$ is an integer $\geq 0$, and $a_0, a_1, ..., a_d$ are constants with $a_d > 0$, we can express a function $p(n) = \sum_{i=0}^{d} a_i n^i$. This means that $p(n) = \Theta(n^d)$. PROOF. $\lim_{n\to\infty} \frac{p(n)}{n^d} = \lim_{n\to\infty}(\frac{a_0}{n^d} + \frac{a_1}{n^{d-1}} + ... + a_d) = a_d$.
2. A useful theorem to know for limits is **L'Hopital's Rule** where $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$.
   - For example, $\lim_{n\to\infty} \frac{\ln n}{n} = \lim_{n\to\infty} \frac{\frac{1}{n}}{1} = 0$.
   - Another example is find the relationship between $f(n) = n^n$ and $g(n) = n!$. By **Stirling's Approximation**, $n! = \sqrt{2\pi n}(\frac{n}{e})^n(1+\Theta(\frac{1}{n}))$. Therefore, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{e^n}{\sqrt{2\pi n}(1+\Theta(\frac{1}{n}))} = \frac{1}{\sqrt{2\pi}} \lim_{n\to\infty} \frac{e^n}{\sqrt{n}} = \lim_{n\to\infty} \frac{e^n}{\frac{1}{2\sqrt{n}}} = \infty$. Note that $\Theta(\frac{1}{n})$ tends to 0.
3. Another useful theorem is involved when regarding $f(n) = \omega(g(n))$, where $f(n) \leq g(n)$ for all $n > 0$ then $\lim_{n\to\infty} f(n) \leq lim_{n\to\infty} g(n)$. The **Squeeze Theorem** can be used here where $h(n) \leq f(n) \leq g(n)$ for all $n > 0$ and $lim_{n\to\infty} h(n) = \lim_{n\to\infty} g(n)$. Then $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n)$.
   - For example, $0 \leq \frac{n!}{n^n} \leq \frac{1}{n}\frac{2}{n}...\frac{n}{n} \leq \frac{1}{n}$. By the above, $\lim_{n\to\infty} \frac{n!}{n^n} = 0$.

- Therefore, $n! = o(n^n)$.

4. If $f_1(n) = O(g_1(n)$ and $f_2(n) = O(g_2(n))$ then $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$. This also applies to $\Theta, \Omega, o, \omega$.
   - For example, given $(n^3)(\log n)^3(\log\log n)$, we can state that $\log\log n = o(\log n)$ and $\log n = o(n)$. This means that $(\log n)^3(\log\log n) = o(\log^4 n) = o(n)$. Therefore, $(n^3)(\log n)^3(\log\log n) = n^3 o(n) = o(n^4) = o(n^4 \log n)$.

# Algorithm Design and Maximum Subrange Sum

## Maximum Subrange Sum

Given an **input** which is an array $x$ of $n$ possibly negative integers, the **output** will be the maximum sum found in any (possibly empty) subarray of $x$.

```
Input: An array x of n possibly negative integers.
Output: Maximum sum found in any subarray (possibly empty) of the input.
```

For an example, given the array `31, -41, [59, 26, -53, 58, 97], -93, -23, 84` where the solution is the sum of entries 59 to 97: `187`. A possible application for this would be the fluctuation of a stock price. Note that in algorithms we typically abstract a problem to something that is easy to solve or relate to another problem that is already solved.

- This problem is only interesting if there are negative numbers.
- Without negative numbers, the maximum sum would be the entire array.
- With all negative integers, the empty subarray would be the solution (a sum of 0).

## Solution 1: Brute-Force Enumeration

Enumerate all possible subarrays. Let us call this solution `maxsubrangesum1(x[1..n])`. Through this process, we want to check all `x[1..n]` and want to define a variable `max`.

```
  maxsubrangesum1(x[1..n])
1 max <-- 0
2 for l <-- 1 to n do
3   for u <-- l to n do
4     sum <-- 0
5     for i <-- l to u do
6       sum <-- sum + x[i]
7     if sum > max then
8       max <-- sum
9 return max
```

Notice that the number of times that the statements in the innermost loop are executed is $\leq n \times n \times n = n^3$. Therefore, we can define the running time, using big-O notation, is $O(n^3)$; the worst case running time can be proven to be $\Theta(n^3)$. Ultimately, the innermost loop performs a lot of recomputation. For example, if $l = 3, u = 5$, the sum of `x[3..5]` is calculated and, say, is 32. In the next iteration of the loop at the middle level, $l = 3, u = 6$, the sum of `x[3..6]` is calculated again (even though we know `x[3..5]` already).

## Solution 2: Running Sum

Uses the observation we can infer above, where the sum of `x[1..n]` = sum of `x[1..n-1]` + `x[n]`. We can modify the above, changing the innermost loop and replacing it as below.

```
  maxsubrangesum2(x[1..n])
1 max <-- 0
2 for l <-- 1 to n do
3   sum <-- 0
4   for u <-- l to n do
5     sum <-- sum + x[n]
6     if sum > max then
7       max <-- sum
8 return max
```

This is $\Theta(n^2)$. The running time was increased by a factor of $n$.

## Solution 3: Another $\Theta(n^2)$-time Solution

Some standard tricks of algorithm design are used here. The idea of a prefix sum array is used here. A subarray is **prefix** if it contains the first item in the array. A subarray is **suffix** if it ends at the last entry in the array. These can be empty. Notice that computing the prefix sum array is $\Theta(n)$ because it requires a single loop to compute the sums iteratively. The main idea of this solution is that:

- we precompute the *prefix sum array* `p[0..n]` where `p[i] = x[1] + x[2] + ... + x[i]`,
- given the above, we can calculate sum of `x[l..u]` = `p[u] - p[l-1]`.

```
  maxsubrangesum3(x[1..n])
1 p[0] <-- 0
2 for i <-- 1 to n do:
3   p[i] <-- p[i-1] + x[i]
4 max <-- 0
5 for l <-- 1 to n do:
6   for u <-- l to n do:
7     sum <-- p[u] - p[l-1]
8   if sum > max then
9     max <-- sum
10 return max
```

This is $\Theta(n^2)$.

## Solution 4: Applying an Algorithmic Paradigm

The algorithm paradigm we will apply here is **divide-and-conquer**. Recall the problem of *mergesort*: cutting an array and recursively sorting halves and then merging sorted arrays into one. Something very similar is done here.

- the array `x` is cut into two halves,
- the maximum subrange sum is found *recursively* in each half, and
- consider subranges that straddle the midpoint:

- the portion of the subrange in the first half is a *suffix* array that has the greatest sum (proven by contradiction),
- max among these subranges is formed by a maximum-sum suffix of the left half of `x` and a maximum-sum prefix of the right half of the array.

Notice that the below code is a recursive function. This function computes the maximum subrange sum of an array `x[l..u]`. To find this for an array `x[1..n]`, call `maxsubrangesum4(x,1,n)`.

```
   maxsubrangesum4(x,l,u)
1  if l > u then
2    return 0 # array is empty
3  if l = u then
4    return max(0,x[l]) # ensure not negative
5  m <-- floor((l+u)/2) # midpoint
6  # get maximum-sum suffix of left half
7  suml <-- 0
8  maxleft <-- 0
9  for i <-- m downto l do
10   # compute x[i..m]
11   suml <-- suml + x[i]
12   maxleft <-- max(maxleft,suml)
13 # compute maximum-sum prefix of right half
14 sumr <-- 0
15 maxright <-- 0
16 for i <-- m+1 to u do
17   # compute x[m+1..i]
18   sumr <-- sumr + x[i]
19   maxright <-- max(maxright,sumr)
20 maxa <-- maxsubrangesum4(x,l,m)
21 maxb <-- maxsubrangesum4(x,m+1,u)
22 return max(maxa,maxb,maxleft+maxright)
```

**NOTE** Reading is 4.1. Online solution credentials for website are username `3110` and password `NPComplete`. Assignment 1 solutions and Assignment 2 have been posted.

If we assume $n$ is a power of 2, the running time can be defined recursively.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Note that the $\Theta(n)$ portion of the $n > 1$ part of the definition comes from joining halved parts of the array. This is the same running time as mergesort; this is order $\Theta(n \lg n)$.

## Solution 5: Employing the Maximum Suffix Sum

Suppose we have found the maximum subrange sum for an array `x[1..j-1]`. To find it for `x[1..j]` (we consider one more entry), we have two possible cases. It depends on where the answer lies in the known preceding subarray. The subrange with maximum sum either:

1. lies entirely within `x[1..j-1]`, or
2. ends with `x[j]`.

Solution 5: Employing the Maximum Suffix Sum continued on next page... 

If we already know the **maximum suffix sum** of `x[1..j-1]`, we can add `x[j]` to that sum; if the result is $> 0$, it is the maximum suffix sum of `x[1..j]`. Otherwise, take 0; we have an empty suffix.

Using this, we can define the pseudocode as follows.

```
   maxsubrangesum5(x[1..n])
1  # max subrange sum in the portion of array so far
2  maxsofar <-- 0
3  # max suffix sum in portion of array so far
4  maxsuffixsum <-- 0
5  # walk through array from left to right
6  for j <-- 1 to n do
7    maxsuffixsum <-- max(maxsuffixsum + x[j],0)
8    # two cases for maximum sum
9    maxsofar     <-- max(maxsofar,maxsuffixsum)
10 return maxsofar
```

The order here is $O(n)$; the running time is a function of $n$.

**NOTE** *Programming Pearls: algorithm design techniques* by Bentley (84 CACM) describes this running time improvement.

### Algorithm Design Paradigms

The paradigms that will be discussed are as follows:

1. reduce to known problem,
2. recursion,
3. divide and conquer,
4. invent (or augment) a data structure,
5. greedy algorithms,
6. dynamic programming,
7. exploit problem structure (algebraic, geometric, etc.), and
8. use probabilistic and randomized solutions.

## Reduce to Known Problem

### Idea

The idea here is to develop an algorithm for a problem by viewing it as a *special case* of a problem that one already knows the solution to, or how to solve it efficiently. For example, the solution to sorting has been studied extensively. This has its roots in mathematics.

## Element Distinctness (Uniqueness) Problem

We would like to determine if an array of $n$ numbers contains *repeated* elements. For example, given the following array, no element appears more than once.

```
7,4,3,10,-10
```

However, in this array, two elements are not distinct.

```
4,7,3,6,7
```

Solution 1. Compare each element to every other element in the array using a *double loop*. The running time for this would be $\Theta(n^2)$. There are $\binom{n}{2}$ pairs, which means there are $\frac{n(n-1)}{2}$ comparisons that are done.

Solution 2. Sort the $n$ numbers. Compare each given element with the next (immediately to its right) for duplicates by walking through the sorted array. Using mergesort or heapsort, the first step would be $O(n \lg n)$. The second step would be $O(n)$. By reducing the problem to sorting we have an $O(n \lg n)$ solution.

**NOTE** Reading for next lecture is 33.3.

Notice that the **lower bound** of the element distinctness problem (and not the running time of a particular solution) is $\Omega(n \lg n)$, in the **comparison-based model**.

The comparison-based model is an abstract model of computation, like the Random Access Machine model, where algorithms may only do *arithmetic and comparisons*. What is most important here is what operations are forbidden: one cannot use the value of input as addresses of memory cells. Notice, too, that when we refer to the lower bound of a problem, we mean to say that *any algorithm that solves this problem (in this model) requires* $\Omega(n \lg n)$ *time in the worst case*, at least. To prove this, we require more involved concepts. However, it is important to know this result.

What happens if we *do* use the values of our input as addressing into memory cells?

Solution 3. Build a **hash table**. Insert each number into the table; if a collision occurs (two numbers hash to the same spot), we *compare* them (and not try to insert them unless they are not equal). The average case is $O(n)$ because lookups and insertions are $O(1)$. This is much faster than the sorting best solution above. The worst case here, using hash tables, is $O(n^2)$ because the worst case for a hash table is $O(n)$ in the case of collisions.

Solution 4. If numbers are relatively small integers from 1 to $M$, we can create an array $B[1..M]$, initialize all of its entries to zero, and then walk through $A$, toggling the value of $B[i]$ if $i$ appears in the input $A$. Before toggling, we check if $B[i]$ has a 1 in it already; if it does, we have found a duplicate. The running time for this is $O(n + M)$ with space complexity $O(n + M)$. This is a special case of the problem; it is good if $M$ is relatively small; if $M = O(n)$, the complexity is only $O(n)$. How does one avoid initialization of $B$? We could store the value and not 0 or 1.

## Set Intersection Problem

Given an input which is two **sets** of integers (represented as arrays; no duplicate numbers because they are sets), each of size $n$, we need to produce an output which is their intersection: the elements appearing in both arrays.

Solution 1. The naive approach involves looping through one array, checking to see if any element is located in the other; this means there is a double loop. The running time complexity of this is $O(n^2)$.

Set Intersection Problem continued on next page...

Solution 2. Sort the first array and loop through the second, checking the elements $v$. the first using a binary search method. The running time for this is $O(n \lg n)$; the first step (sorting) is $O(n \lg n)$ with binary search having complexity $O(\lg n)$.

## Colinear Points Problem

Given a set of $n$ points in a plane (array of size $n$ which has elements that are $(x, y)$ coordinate pairs), we would like to see whether there exists three that are **colinear**. Points that are colinear are those that lie on the same line.

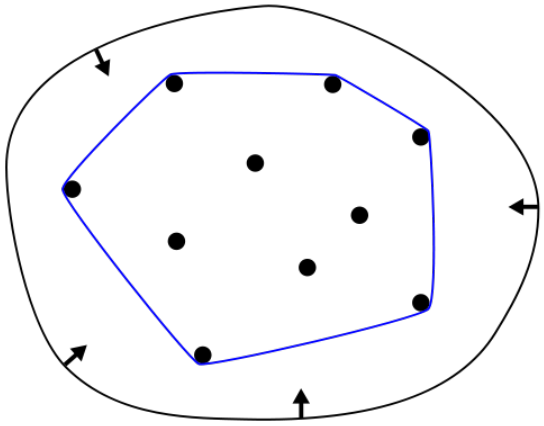
Solution 1. The naive solution involves looking, for each triplet of points, whether a triplet is colinear. For instance, if we have three points $P_1, P_2, P_3$ where $P_i = (x_i, y_i)$, we compute the slopes of the line connecting $P_1$ with $P_2$ and then $P_1$ with $P_3$. If these are equal, the points are colinear (their slopes overlap). Note that there are $\binom{n}{3}$ triplets in an array of $n$ points; therefore, the running time complexity is $\Theta(\binom{n}{3}) = \Theta(n^3)$.

Solution 2. For each point $P$, compute the slopes of all lines formed by other points joined with $P$. This means we compute $n - 1$ slopes for each point. We can use the solution for the **element distinctness problem** here for any given array of slopes for a point; if there is a duplicate element among these slopes, then there are colinear points. The running time here is $O(n^2 \lg n)$, using the $O(n \lg n)$ worst case solution for finding duplicate elements.

This problem is one of a set of problems where, if a solution can be found that runs in $O(n^2)$ time, then others in its class of problems can also be solved in such a time.

## Convex Hull Problem

A **convex polygon** is one where each internal angle is $\leq 180$ degrees. A **concave polygon** is one where this is not true. A **convex hull** of a set of points $Q$ in the plane (given by $(x, y)$ coordinates) is the smallest convex polygon $P$ that contains all of the points in $Q$ either on its boundary or inside the polygon. This implies that the vertices of $P$ must be points found within $Q$ (or else we could find a smaller polygon).



Consider an elastic band analogy here; the elastic rubber band would form such a polygon if placed around a set of pins representing the points.

From a theoretic point of view, this is a **fundamental object** in **computational geometry**: a field studying algorithms that solve problems in geometry. This is used a great deal in graphics (e.g., drawing a

Convex Hull Problem continued on next page... 

shape using a vector graphics program and then determining if the cursor is within the convex hull of that set of points).

**NOTE** Tutorial 1 questions posted last Thursday. Tutorial will be in Chemistry 226.

It is important to know here the concept of a **polar angle**. The polar angle of a point $p$ with respect to the origin point ($(0,0) = p_0$) is the angle from the *x-axis* through $p_0$ to $p$ rotating counterclockwise.

Furthermore, computing the angle $\Theta$ between two directed line segments, vectors $u$ and $v$, both starting from $p_0$ to points $(x_1, y_1)$ and $(x_2, y_2)$ respectively, has vectors that can be defined as:

$$u = (x_1, y_1), v = (x_2, y_2) \tag{1}$$

Using the **dot product**, we can find, where $\|u\| = \sqrt{u \cdot u}$:

$$u \cdot v = \|u\|\|v\| \cos\Theta = (x_1, y_1) \cdot (x_2, y_2) = x_1 x_2 + y_1 y_2 \tag{2}$$

**NOTE** Square-rooting cannot be done in constant time. However, angles can be compared by using $\cos^2\Theta$.

Let us make a simplifying assumption here: no three points are colinear in the plane. The first approach to the problem is a naive one.

Naive Approach. Consider the line joined by each pair of points $p_1$ and $p_2$. If all of the other points are on the same side of this line, the line segment between $p_1$ and $p_2$, $p_1p_2$, is an edge of the convex hull. This involves testing all other points are on the same side of $p_1p_2$ and this involves the use of geometry.

- Determine the equation of the line, say $Ax + By + C = 0$.
- Substitute the coordinates of all other points into this equation, individually. For example: $Ax_1 + By_1 + C$.
- If the results are all $> 0$ or $< 0$, we say that they are on the same side.

The complexity of this is $O(n^3)$; a double loop is needed for all pair-comparisons with a further factor of $n$ for each pair to determine the status of all other points.

Jarvis's March. Also known as the **Gift Wrapping Approach**. This uses a similar analogy to the elastic band; trying to tightly wrap all points using line segments in order to acquire the convex hull. Note that this gives us a sequence of computation because it starts with a single line.

1. Start with an **extreme point**, $p_0$, the point with minimum y-coordinate, breaking ties arbitrarily (by our assumption, the maximum number of points that can have the same y-coordinate are 2). This is an $O(n)$ operation. $p_0$ must be on $CH(Q)$ where $CH$ is the convex hull and $Q$ is the set of points.
2. To determine the polar angle of other points in relation to $p_0$, we connect $p_0$ to each other point. The next vertex $p_1$ on $CH(Q)$ has the smallest polar angle. We "march" from $p_0$ to $p_1$. This is an $O(n)$ operation. We continue this for each successive point $p_i$ until we find the highest point.
3. When we reach the highest point $p_k$, we have constructed the *right chain* of $CH(Q)$. In order to determine the *left chain*, use a **reverse negative-x axis** with respect to $p_k$, and choose $p_{k+1}$ as the point with the smallest polar angle with respect to $p_k$.

Convex Hull Problem continued on next page... 

The running time of this is $O(nh)$, which is at worst case $O(n^2)$, where $h = |CH(Q)|$, the number of points in the convex hull. Good if $h$ is small.

Graham's Scan. This involves sorting.

1. Locate $p_0$, the lowest point in $Q$. This is an $O(n)$ step.
2. Sort all of the other points by their polar angle with respect to $p_0$ giving a result $p_1, p_2, ..., p_{n-1}$. This is an $O(n \lg n)$ step. We *do not* move the origin point here; instead...
3. Process the points with the one with the smallest polar angle. Add each point in turn with the next higher angle. If doing so causes us to make a **left** turn, continue (its angle is less than 90 degrees with respect to the current position/point). Otherwise, we make a **right** turn (and in turn add concavity) and the point in the middle of the turn $\notin CH(Q)$; it is removed and the previous point is joined to the new one. *Backtrack* and discard points as you go until ending up with a left turn again.

Step 3 can be represented by the following pseudocode; the use of a stack ensures FILO movement of data.

```
S <-- empty stack
PUSH(p_0)
PUSH(p_1)
PUSH(p_2)
for i <-- 3 to n-1 do
  while the angle formed by points NEXT-TO-TOP(S)
  and TOP(S) and p_1 makes a non-left turn
    POP(S)
  PUSH(P_i,S)
return S
```

Realize that the above *for* loop is $O(n)$ and the while loop is at most $O(n)$. This leaves this step at $O(n^2)$ as a rough estimate. If we more carefully analyze this step, we notice that each point is visited at most twice: once for a push (add it to $CH(Q)$) and we may or may not have to remove it. This means that the number of pushes are $O(n)$ and the number of pops are $O(n)$. This means that this entire step is *actually* $O(n)$.

The running time for this entire algorithm is therefore $O(n \lg n)$ because of the step that involves sorting. In the worst case, this is faster than Jarvis's March. But Jarvis's March is not all that bad; for a square of uniformly distributed points, $h$ may be $O(\lg n)$ and therefore the average case running time is also $O(n \lg n)$.

## Reducing Sorting to a Convex Hull Problem

Suppose we are sorting $n$ numbers: $x_1, x_2, ..., x_n$. These are *not* points. Also suppose that we have an algorithm `HULL` that computes the $CH$ of $n$ points in $T(n)$ time. The task here is to use the `HULL` algorithm to solve sorting in $T(n) + O(n)$ time.

1. First, form the set of points $Q$ of form $(x_i, x_i^2)$ (*lifting* the values in the given array from an *x-axis*). These therefore form a **parabola** $y = x^2$. This is $O(n)$.
2. Run `HULL` to compute $CH(Q)$, a convex polygon forming the shape of the parabola with a closed top. All points are on $CH(Q)$. In this manner, it will report this convex hull and contains all points, reporting the points in a certain order. This is $T(n)$ time.
3. Let $C$ contain the output array of `HULL`. Say these are given in counter-clockwise (*ccw*) order, without loss of generality, starting from some point $p_0$.

4. A cyclic shift of $C$ gives us a *sorted list* of all $x_i$. This is $O(n)$.

We previously learned what the lower bound of a problem is. The lower bound of sorting in the comparison-based model is $\Omega(n \lg n)$. We cannot do better. This implies that the lower bound of the convex hull problem in the comparison-based model is $\Omega(n \lg n)$.

**NOTE** Suggestions on Midterm I Preparation. Study notes up to and incl. lecture for October 4, assignments 1-3, tutorial 1, set of sample questions for midterm, exercises in the textbook, and textbook readings. Cheat sheet (1pg) allowable. 47 minutes is assigned for the midterm.

# Recursion

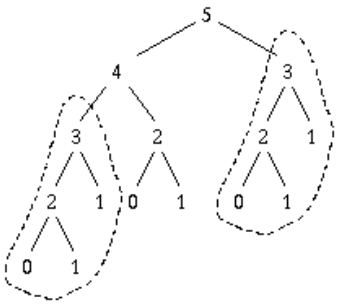
## Poor Design and the Fibonacci Series

A *poorly designed* recursive algorithm can have **exponential** running time behaviour. For example, an algorithm for computing a number in the Fibonacci number series. $F(0) = 0$ and $F(1) = 1$ where $F(n) = F(n-1) + F(n-2) \forall n \geq 2$. Using a recursive algorithm, one could acquire a running time of $\Theta((\frac{1+\sqrt{5}}{2})^n)$.

A recursive algorithm in this grain is the following.

```
FIBREC(n)
1  if n < 2 then
2    return n
3  else
4    return FIBREC(n-1) + FIBREC(n-2)
```

This is a horrible algorithm. The call trace for $n = 5$ involves making function calls to $F_4$ and $F_3$ with forks at each node in a graph. $F_1$ would be called 5 times. The base case is computed 8 times. There is a great deal of recomputation here.




The running time can be represented by the following equation.

$$T(n) = \begin{cases} 1 & : n = 0 \\ 1 & : n = 1 \\ T(n-1) + T(n-2) + 1 & : n \geq 2 \end{cases} \quad (3)$$

We could guess that this is $T(n) = 2F(n+1) - 1$, if we use the Fibonacci function. We have to prove this is true, though; a guess is not enough. By induction:

Proof. Clearly, the claim is true for $n = 0, n = 1$. This defines our base case. We assume it is true for all $n < N$ where $N \geq 2$. We prove for $n = N$ that

$$\begin{aligned} T(N) &= T(N-1) + T(N-2) + 1 \\ &= (2F(N) - 1) + (2F(N-1) - 1) + 1 \\ &= 2(F(N) + F(N-1) - 1 \\ &= 2F(N+1) - 1 \end{aligned}$$

And therefore our guess is correct. □

Using the substitution method, we can come to a very important conclusion. By our guess and induction: $T(n) = \Theta((\frac{1+\sqrt{5}}{2})^n)$. This is exponential running time.

An iterative version of this same algorithm is the following. It is $O(n)$.

```
FIBITE(n)
1  if n < 2 then
2    return n
3  else
4    a <-- 0
5    b <-- 0
6    for i <-- 2 o n do
7      (a,b) <-- (b,a+b)
8  return b
```

Memoization. However, we can still use recursion and acquire an $O(n)$ running time. This is using a method of **Memoization**; we use recursion and store function values as they are computed. When the function is invoked, we check the function argument to see if it is already computed/known. If it is, we do not compute.

The following is the pseudocode for this technique.

```
FIBMEM(n)
1  F[0] <-- 0
2  F[1] <-- 1
3  for i <-- 2 to n do
4    F[i] <-- -1 // initialize
5  return FIBRECM(n)

FIBRECM(i)
1  if F[i] != -1 then
2    return F[i] // already computed
3  else
4    F[i] <-- FIBRECM(i-1) + FIBRECM(i-2)
5    return F[i]
```

The process for `FIBRECM` has a running time $O(n)$. Values are either compared and not computed or actually computed and assigned. For each entry `F[i]`, only trivial executions are made (check, compute and store, and retrieved for `F(i+1)` and `F(i+2)`).

Some languages have constructs in place for memoization. For example, Maple allows this using one line of code.

```
option remember;
```

## Efficient Recursive Solutions

Suppose there is a problem where one wishes to print all permutations of $\{1, 2, ..., n\}$ in lexicographic order. For example, for $n = 3$, we would have $3! = 6$ permutations. The permutations that result, in lexicographic order, are:

1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

When using a brute-force solution in many problems (where we try all cases), we would have a great deal of unnecessary complexity (especially for large $n$); in this case, we would have $O(n!)$ computations. Let us make a clever observation here: the first column is either 1, 2, or 3. Recursively, we can see this as a $n$ followed by all possible permutation of all other elements other than itself. The *idea* here, then, is to take one element out and print the permutations of the rest.

What are the *parameters* here? We need:

- a prefix array of the numbers already printed out ($P$), and
- a nonempty set of remaining numbers to be printed ($S$).

The pseudocode can then be expressed as follows. $(P, i)$ signifies the concatenation of $P$ and $i$ (this has its own running time because we must create a new array: $m + 1$). The removal of an item from the set also has its own running time: $n$ for removing $i$ from S at $n$ steps.

```
   printperm(P,S)
1  if S has 1 element then // base case
2    print P followed by the single element of S
3  else
4    for each element i in S
5      printperm((P,i),S-{i})
```

To use this, we call `printperm([],{1,2,3,...,n})`. How efficient is this algorithm? The time $T(n) \geq n \times n!$ because there are $n!$ permutations. This is merely a lower bound. This is a recursive function with two parameters; let us signify $m = |P|$ and $n = |S|$. $T(m, n)$ would equate to the number of steps our algorithm takes. Our task here is to compute $T(0, n)$ (the initial call to the algorithm).

1. **Base Case**: $T(m, 1) = m + 1$.
2. **Recursive Case**: $T(m, n) = n(T(m + 1, n - 1) + (m + 1) + n)$ if $n > 1$

Let's make a guess here. $T(m, n) = (m+1+n)a(n)-n!$ where $a(n)$ is a number of "variations" (the number of permutations of nonempty subsets of $\{1, 2, 3, ..., n\}$. Proof by induction will follow on an assignment.

Online, we can look this up as sequence `A007526` in the Online Encyclopedia of Integer Sequences. From there, we can determine that $a(0) = 0$ and $a(n) = n \times a(n - 1) + n$ (using a recursive formula); after further study, $a(n)$ can be defined using a non-recursive definition. That would be $a(n) = \lfloor e \times n! - 1 \rfloor$. Investigating the numbers can also allow us to make this observation of the sequence. $T(m + 1, n) - T(m, n)$ can be recognized as depending on $n$ only; this expression can be narrowed down to $a(n)$. This expression also implies that $T(m + 1, n) = m \times a(n) + T(0, n)$. Investigating $a(n + 1) - T(0, n) = n! + n + 1$ allows us to derive the formula for our guess.

Efficient Recursive Solutions continued on next page...

If we substitute into our guess, we can compute $T(0,n) = (n+1)\lfloor e \times n! - 1 \rfloor - n!$. Taking the floor out just involves using an inequality: $T(0,n) \leq (n+1)en! - n!$. In other words, the Cost Per Symbol $\leq \frac{(n+1)en! - n!}{n \times n!} = \frac{e(n+1)-1}{n} \approx e$. This algorithm is indeed efficient because the Cost Per Symbol is approximately constant.

# Divide-and-Conquer

## Process

There are three general steps to this technique.

1. **Divide** the given problem into smaller subproblems,
2. **Conquer** the subproblems by solving them recursively (as long as they have the same structure as the original problem), and
3. **Combine** the solutions of the subproblems into the solution of the original problem.

An example of this process was done in the formulation of the **mergesort** algorithm.

## Example: Mergesort Sorting

The idea here is to sort the subarray `A[p..r]`. We can represent the pseudocode for this by the following.

```
   MERGE-SORT(A,p,r)
1  // at least two elements; otherwise, base case: no elements
2  if p < r then
3    // divide
4    q = floor((p+r)/2)
5    // conquer
6    MERGE-SORT(A,p,q)
7    MERGE-SORT(A,q+1,r)
8    // combine
9    MERGE(A,p,q,r)

    MERGE(A,p,q,r)
1   n_1 <-- q-p+1
2   n_2 <-- r-q
3   Let L[1..n_1+1] and R[1..n_2+1] be new arrays.
4   for i<-- 1 to n_1 do
5     L[i] <-- A[p+i-1]
6   for j<-- 1 to n_2 do
7     R[i] <-- A[q+j]
8   L[n_1+1] <-- infinity
9   R[n_2+1] <-- infinity
10  i <-- 1
11  j <-- 1
12  for k <-- p to r do
13    if L[i] <= R[j]
14      A[k] <-- L[i]
15      i <-- i+1
```

Example: Mergesort Sorting continued on next page... 

```
16    else
17      A[k] <-- R[j]
18      j <-- j+1
```

**NOTE** Reading for today is 2.3.1

.

We can define a formula for the running time as follows.

$$T(n) = \begin{cases} T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + n & : n > 1 \\ 1 & : n = 1 \end{cases} \tag{4}$$

Proof. For simplicity, we assume $n$ is a power of 2; therefore $T(n) = 2T(\frac{n}{2}) + n$. This is a reasonable assumption. By using a technique of **Iteration** (repeatedly plugging in the RHS into the LHS) where $n = 2^k$, $T(2^k) = 2T(2^{k-1}) + 2^k$.

$$\begin{aligned} 2T(2^{k-1}) &= 4T(2^{k-2}) + 2^k \\ 4T(2^{k-2}) &= 8T(2^{k-3}) + 2^k \\ 2^{k-1}T(2) &= 2^kT(1) + 2^k \end{aligned}$$

Adding these together gives us:

$$T(2^k) = 2^k + k2^k \tag{5}$$

This means that $T(n) = n + n \lg n = O(n \lg n)$. □

Proof. Another way to prove this is by using the **Substitution Method**.

```
n      1  2  4  8  16  32
T(n)   1  4  12 32 80  192
T(n)/n 1  2  3  4  5   6
```

This tells us that $\frac{T(n)}{n} = k + 1$ where $n = 2^k$. From this we can make a guess $T(2^k) = (k+1)2^k$. It is clearly true for k; we will prove it for $k + 1$.

$$\begin{aligned} T(2^{k+1}) &= 2T(2^k) + 2^{k+1} \\ &= 2(k+1)2^k + 2^{k+1} \\ &= (k+1)2^{k+1} + 2^{k+1} \\ &\dots \\ &= (k+2)2^{k+1} \end{aligned}$$

□

## Example: Nonnegative Binary Integer Addition

Say we wish to multiple two $n$-bit nonnegative integers together where an input is stored as an array $X[1..n]$ of digits. We can approach this using different methods.

Pencil-and-Paper Method ($O(n^2)$).

```
      1  1  0
    x 1  0  1
--------------
```

```
      1  1  0
   0  0  0
1  1  0
------------
1  1  1  1  0
```

In the middle segment while computing the multiplication there are $n$ digits and $n$ of these collections of digits, providing an $O(n^2)$ computation with the answer having $\leq 2n$ digits.

Solution 1. Say we have number $X$ and $Y$ which can be partitioned as subarrays $X = [A, B]$ and $Y = [C, D]$. $A, B, C, D$ all possess $\frac{n}{2}$ bits and $X, Y, A, B, C, D$ can be implemented as arrays of $0, 1$ bits. We see here that $XY = (A2^{n/2} + B)(C2^{n/2} + D)$. This gives us the following.

$$XY = AC2^n + (AD + BC)2^{n/2} + BD \tag{6}$$

The algorithm would therefore be comprised of the following.

- 4 multiplications: $AC, AD, BC, BD$; 4 subproblems of $n/2$ size, and
- addition between numbers of $O(n)$ bits; multiplication by $2^n$ ($2^{n/2}$), a bitshift (append 0s) — this is $O(n)$.

We can define a formula for the running time of this algorithm as follows.

$$T(n) = \begin{cases} 4T(n/2) + cn & : n > 1 \\ c & : n = 1 \end{cases} \tag{7}$$

Using the iteration approach:

$$\begin{gathered} T(2^k) = 4T(2^{k-1}) + c2^k \\ 4T(2^{k-1}) = 16T(2^{k-2}) + 4c2^{k-1} \\ \ldots \\ 4^{k-1}T(2) = 4^kT(1) + 4^{k-1}2c \end{gathered}$$

When added (notice the geometric series):

$$T(2^k) = 4^kc + c(2^k + 2^{k-1} + \ldots + 2^{2k-1}) = 4^kc + c(4^k - 2^k) = 4^k(c+1) - c2^k \tag{8}$$

If we express this using $n = 2^k$, we have $T(n) = n^2(c+1) - cn = O(n^2)$. Did not achieve any improvement. □

**NOTE** Recall that when determining the running time of mergesort, we assumed that $n$ was a power of 2 where $T(2^k) = 2^k(k+1)$. We can express the validity of this assumption by disregarding the assumption and seeing where we can go. We can transform $T(n) = T(2^{\lg n})$ where $\lg n$ is not an integer (and therefore we cannot plug this into for $k$).

$$\begin{aligned} T(n) = T(2^{\lg n}) &\leq T(2^{\lceil \lg n \rceil}) \\ &= 2^{\lceil \lg n \rceil}(\lceil \lg n \rceil + 1) \\ &< 2^{\lg n + 1}(\lg n + 1 + 1) \\ &= 2n(\lg n + 2) \\ &= O(n \lg n) \end{aligned}$$

Karatsuba's Algorithm. We still divide into halves. $X = [A, B]$ and $Y = [C, D]$. We can express the following formula: $XY = (2^n + 2^{n/2})AC + 2^{n/2}(A - B)(D - C) + (2^{n/2} + 1)BD$. By expansion, this will

eventually transform to: $XY = (AC)2^n + (AD + BC)2^{n/2} + BD$, the formula used previously. Notice that there are 3 subproblems (instead of 4).

Let's investigate the running time.

$$T(n) = \begin{cases} 3T(n/2) + cn & : n > 1 \\ c & : n = 1 \end{cases} \tag{9}$$

If we assume $n$ is a power of 2, we can make the guess $T(2^k) \leq c(3^{k+1} - 2^{k+1})$. Notice that $a^{\lg b} = b^{\lg a}$. We can therefore simplify:

$$\begin{aligned} T(n) &\leq c(3^{\lg n+1} - 2^{\lg n+1}) \\ &= c(3 \times 3^{\lg n} - 2 \times 2^{\lg n+1}) \\ = c(3n^{\lg 3} - 2n) &= O(n^{\lg 3}) \text{ where } \lg 3 \approx 1.58496 \end{aligned}$$

What if $n$ is not a power of 2? Let us define $m$, the smallest power of $2 \geq n$. In other words, $m = 2^{\lceil \lg n \rceil} < 2^{\lg n+1} = 2n$. We can create an $X' = 00...0X$ where there are $m - n$ zeroes and $n$ bits in X; this means there are in total $m$ bits in $X'$. We can also define a $Y' = 00...0Y$ in the same manner. The algorithm will now compute $X'Y' = XY$. What is the running time? $O(m^{\lg 3}) = O((2n)^{\lg 3}) = O(n^{\lg 3})$. The complexity does not change.

## Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$. This is the type of recurrence attributed to divide-and-conquer problems. By this theorem, we can attribute $n/b$ to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. This is the same as assuming that *$n$ is a power of $b$.*

Then:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and if $af(n/b) \leq cf(n)$, for some constant $c < 1$ and all sufficiently large $n$, then $T(n) = \Theta(f(n))$. This final condition is known as the **regularity condition**, a property of the function $f$.

Example: Karatsuba Algorithm. If we apply the Master Theorem on its running time $T(n) = 3T(n/2) + cn$, we see that $a = 3$, $b = 2$, and $f(n) = cn$. The expression $n^{\log_b a} = n^{\lg 3}$. This is case 1: $f(n) = O(n^{\lg 3 - \epsilon})$ where we choose any $\epsilon \in (0, \lg 3 - 1)$. Therefore, $T(n) = \Theta(n^{\lg 3})$.

Example: Mergesort Algorithm. $T(n) = 2T(n/2) + n$; therefore, $a = 2$, $b = 2$, and $f(n) = n$. We express $n^{\log_a b} = n$. This is case 2: $T(n) = \Theta(n \lg n)$.

Example. $T(n) = 3T(n/2) + n^2$; $a = 3$, $b = 2$, and $f(n) = n^2$. We express $n^{\log_a b} = n^{\lg 3}$. We cannot surmise this is case 3 yet; need to check regularity condition. $3f(n/2) = 3(n/2)^2 = \frac{3}{4}n^2 \leq cn^2$ for $c = \frac{3}{4}$. This suffices for case 3; $T(n) = \Theta(n^2)$.

Example. On the basis of the formal definition for $\Theta$, we can surmise that $T(n) = 2T(n/2) + \Theta(n)$ leads to the same result that $\Theta(n \lg n) = T(n)$. The same is true for the original definition for mergesort: $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$. One thing to note, however is that if $T(n) = 2T(n/2) + O(n)$, we can only conclude that $T(n) = O(n \lg n)$.

Master Theorem continued on next page...

EXAMPLE. $T(n) = 2T(n/2) + n \lg n$ where $n^{\log_b a} = n$. We cannot conclude this is case 3. For any $\epsilon > 0$, $lim_{n\to\infty} \frac{f(n)}{n^{1+\epsilon}} = lim_{n\to\infty} \frac{\lg n}{n^\epsilon}$. Using L'Hopital's Rule, we find that $lim_{n\to\infty} \frac{\frac{1}{n \ln 2}}{\epsilon n^{\epsilon-1}} = 0$. This only satisfies that $f(n) = o(n^{1+\epsilon})$.

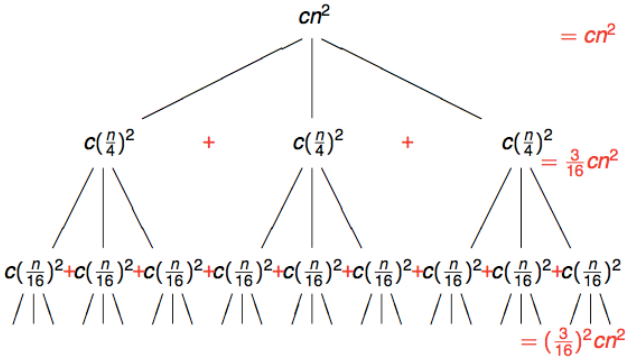
## Recursion Tree

We can represent recursive expressions for time using a **recursion tree**. Each node is the cost of a single subproblem (somewhere in the set of recursive function invocations).

EXAMPLE. Given the following function.

$$T(n) = \begin{cases} 3T(\lfloor n/4 \rfloor) + cn^2 & : n > 1 \\ c & : n = 1 \end{cases} \tag{10}$$

For simplicity, let us assume that $n$ is a power of 4. Therefore, $T(n) = 3T(n/4) + cn^2$. The root will be the first call of this function. How many levels does this tree have? Say the tree has a limited $h$ number of levels. We would go from invocation of the function for $n \to n/4 \to (1/4)^2 n \to ... \to (1/4)^{h-1} n$. And we know that $(1/4)^{h-1} n = 1$ and therefore $h = \log_4 n + 1$. The number of nodes at each level are represented in the series $1, 3, 9, ...$ for levels $1, 2, 3, ....$ The number of leaves at any given level is $3^{h-1} = 3^{\log_4 n} = n^{\log_4 3}$.




From this, let us sum up the costs of all levels: $T(n) = cn^2 + (3/16)cn^2 + (3/16)^2 cn^2 + ... + (3/16)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$ where the last two terms are the level above the leaf level and the leaf levels respectively. This can be simplified to:

$$T(n) = cn^2 \sum_{i=0}^{\log_4 n - 1} (3/16)^i + \Theta(n^{\log_4 3}) \leq cn^2 \sum_{i=0}^{\infty} (3/16)^i + \Theta(n^{\log_4 3}) = \frac{1}{1 - 3/16} cn^2 + \Theta(n^{\log_4 3}) \tag{11}$$

The final expression is therefore $T(n) = (16/13)cn^2 + \Theta(n^{\log_4 3}) = O(n^2)$. Let us verify this using the substitution method. PROOF. $T(n) \leq 2cn^2$. □

## Example: Closest Pair of Points

The idea here is to find the closest pair of points in a set $P$ of points. What we first need to understand this is the **Euclidean distance**: the distance between $(x_1, y_1)$ and $(x_2, y_2)$ is $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. The square root is *not* a constant time operation. What we can do instead is to compare $d^2$: this removes the square root operation.

Example: Closest Pair of Points continued on next page...

Solution: Brute Force. $\Theta(\binom{n}{2}) = \Theta(n^2)$.

Solution: Divide-and-Conquer. Division involves a splitting of the points in the set $P$ to create two subproblems.

*Divide.* Split into $P_L$ and $P_R$ where $|P_L| = \lceil |P|/2 \rceil$ and $|P_R| = \lfloor |P|/2 \rfloor$. We can sort the points in $P$ by $x$-coordinate and split $P$ in two with a vertical line $L$ through the median of the $x$-coordinates. We define $P_L$ as the points on or to the left of $L$ and $P_R$ as the points on or to the right of $L$.

*Conquer.* Find the closest pair of points in $P_L$ with distance $\delta_L$ and in $P_R$ with distance $\delta_R$. This is done recursively. Let $\delta = \min(\delta_L, \delta_R)$.

*Combine.* The closest pair is either the pair with distance $\delta$ or a pair of points $p_L$ with coordinates $(x_1, y_1)$ and $p_R$ with coordinates $(x_2, y_2)$, with $p_L \in P_L$ and $p_R \in P_R$ whose distance $< \delta$. If the latter exists, observe:

**NOTE** Reading for today's lecture involves 4.4 (and optionally 4.6). Next involves 33.4.

1. $(x_1 - x_2)^2 + (y_1 - y_2)^2 < \delta^2$
2. $\Rightarrow |x_1 - x_2| < \delta$ and $|y_1 - y_2| < \delta$.
3. There exists a $\delta \times 2\delta$ rectangle $E$ centred at $L$ that contains $p_L$ and $p_R$.
4. The left half of $E$ has $\leq 4$ points $\in P_L$ (which would be at the corners of the half of the rectangle). This is also true for the right half (for $P_R$). Ultimately, $E$ has $\leq 8$ points $\in P$.

To find $p_L$ and $p_R$,

1. Sort the points in the strip of width $2\delta$ centred at $L$ by $y$-coordinate.
2. Scanning from largest to lowest, compare each point with $8 - 1 = 7$ others that follow it. This is a *constant* and therefore is $O(n)$.

The running time can be defined $T(n) = 2T(n/2) + O(n \lg n)$. This cannot be solved using the Master Theorem. It can be solved that, using the Recursion Tree and Substitution Method, the running time is $O(n \lg^2 n)$.

*Speeding up the solution.* The following are ways to speed up this solution.

- To avoid sorting for each recursive call at the beginning of the algorithm, do some presorting. Sort all points by $x$-coordinate and store them into array $X$. Similarly, do so by $y$-coordinate and store them into an array $Y$.
- In the *Divide* step, divide $X[p..r]$ into $X[p..q]$ ($P_L$) and $X[q+1..r]$ ($P_R$) where $q = \lfloor (p+r)/2 \rfloor$. $X$ is global here. Locally, split $Y$ into $Y_L$ and $Y_R$ with a linear scan and check if each point is in $P_L$ or $P_R$. In $O(1)$ time: store with each entry in $Y$ the index of the corresponding point in $X$.
- In the *Combine* step, exact points can be determined.

The time here involves sorting (at the beginning), which is $O(n \lg n)$, and the time for following recursive steps ($T(n)$) where, here, $T(n) = 2T(n/2) + O(n)$. The running time is therefore $O(n \lg n)$.