# Algorithms Course Notes

## Motivations

**Algorithm**: Algorithms are well defined set of rules for solving computational problems. Algorithms play a key role in modern technological innovation.

* Sorting
* Shortest Path
* Scheduling
* Cryptography
* Computer Graphics
* Networks
* Computational Biology

(Delete when done) Prioritize:

* Definitions
* Proofs
* Algorithmic Tricks

For any new or existing algorithms, we need to ask ourselves and prove:

* Is it correct? (Correctness)
* Can we do better? (Efficiency)

## Part One

### Multiplication

**Integer Multiplication**

Correctness: Given two n-digit numbers, is the algorithm to compute their product correct?

Efficiency: Given two n-digit numbers, how long does it take to get a result?

* **Elementary Multiplication**
  + Correctness:
    - 
    - <https://pages.cs.wisc.edu/~cs240-1/readings/07_Program_Correctness.pdf>
  + Efficiency: O(N^2)
* **Karatsuba Multiplication**
  + Correctness:
    - 
    - <https://people.cs.uchicago.edu/~laci/HANDOUTS/karatsuba.pdf>
  + Efficiency:
    - T(n) = 3 \* T(n / 2) + O(n) 🡺 a > b^d 🡺 n^(log(a)) 🡺 n^(log(3))
    - O(n): additions and shifts have linear complexity.
    - Better for numbers with less than 1000 digits.
  + **Algorithm Notes:**
    - Karatsuba Algorithm Split
      * **x = a \* 10^(n/2) + b**
      * **y = c \* 10^(n/2) + d**
      * **x \* y = (a \* c) \* 10^n + 10^(n / 2) \* (ad + cb) + bd**
    - Clever Grouping
      * The middle term ( **10^(n / 2) \* (ad + cb)** ) is cleverly calculated by subtracting the already computer terms: **ac** and **bd.**
      * **(a + b) \* (c + d) = ac + ad + bc + bd**
      * **(ad + cb) = (a + b) \* (c + d) – ac – bd 🡺 Karatsuba((a + b), (c + d)) – ac - bd**
    - Karatsuba Recursive Calls
      * n == Max (length of x and y).
      * karatsuba(12, 56) Output: (5 \* 100) + (33 - 5 - 12) \* 10 + 12 = 672
        + a = 1, b = 2, c = 5, d = 6
      * karatsuba(1234, 5678)
        + a = 12, b = 34, c = 56, d = 78
      * karatsuba (46, 134)
        + x = (12 + 34), y = (56 + 78)
        + a = 4, b = 6, (c = 13, d = 4) ??? what to do with 3 digit numbers ??? Round up first piece, round down last piece
      * Base Case: One or both numbers have length == 1.
    - **Why This Saves Effort:**
      * We calculate three smaller multiplications (ac, bd, (a + b) \* (c + d)) instead of four.
      * The subtraction and the multiplications by powers of ten involve simple shifts and additions, which are less computationally expensive than direct multiplication.
      * Karatsuba realized we could do something clever with the distributive property to reduce the number of calculations.
    - **The essence:**
      * Karatsuba is a clever rearrangement of the standard multiplication that lets us calculate a large multiplication using smaller multiplications and some simple additions and subtractions.
* **Fast Fourier Transform**
  + Correctness:
    - Playlist: <https://www.youtube.com/watch?v=wmCIrpLBFds&list=PLHXZ9OQGMqxdhXcPyNciLdpvfmAjS82hR>
  + Efficiency:
    - O(N\*log(N))
    - Where N is the number of digits (or the degree of the polynomial).\
    - The overhead manipulations make this algorithm the most optimal when dealing with numbers such that N > 1000.
  + **Algorithm Notes:**
    - The FFT is a speed-up trick for a mathematical operation known as Discrete Fourier Transform (DFT). DFT breaks a signal S(n) down into the frequencies that make it up. But DFT takes a lot of time for larger datasets.
    - FFT works faster (especially on datasets that have sizes equal to a poser of 2).
    - Can be used to multiply number represented as polynomials.
    - Convert Multiplication of Polynomials 🡺 Multiplication in Frequency Domain 🡺 Normal Multiplication.
    - Steps:
      * Represent polynomial as a list of its coefficients.
        + Pad lists with 0 to make them the same length.
      * Apply FFT to transform coefficients from Time Domain 🡺 Frequency Domain.
      * Multiply Frequency Domain lists.
      * Apply inverse FFT to get result in Time Domain. Final Answer.

### Sorting

**Sorting**

Correctness: Given a list of n comparable objects, does the algorithm return the list in sorted order?

Efficiency: Given the list of n comparable objects, how long does it take to sort them and return a result?

* **Merge Sort**
  + Correctness:
    - ****
    - <https://www.cs.mcgill.ca/~dprecup/courses/IntroCS/Lectures/comp250-lecture16.pdf>
  + Efficiency:
    - O(N log N)
    - T(N) = 2 \* T(N / 2) + O(N)
  + **Algorithm Notes:**
    - Merge Sort is a sorting algorithm that recursively sorts a list by splitting the size in two and recursively calling itself on the smaller subproblem. When the problem is small enough (i.e. the length of the list is 2 or 1) the algorithm sorts the smaller subproblem and returns it. The algorithm combines the two recursive calls by linearly scanning through the two, now sorted, subarrays and adding their elements to a larger array in sorted order based on simpler comparisons for the beginning of each list.
* **QuickSort**
  + Correctness
    - ****
    - <https://www.cs.cmu.edu/afs/cs/academic/class/15451-s07/www/lecture_notes/lect0123.pdf>
  + Efficiency:
    - O(N\* Log(N))

**Mentionable:**

* Insertion Sort
  + Split the array into two parts, sorted and unsorted. Take the first element in the unsorted array and insert it into its correct position in the sorted array until all elements in the unsorted array have been placed.
* Selection Sort
  + Iteratively search for the smallest element in the unsorted array and place it at the end of the sorted half of the array.
* Bubble Sort
  + Repeatedly compare adjacent elements to see if they are in the correct order. If they are not, swap their positions and continue. Larger elements will bubble down, and smaller elements will bubble up.

### Asymptotic Runtime

Given f(n) and g(n)

* Big-O f(n) = O(g(n)):
  + f(n) is bounded above by a constant multiple of g(n).
  + | f(n) | <= c \* | g(n) | for some c and all n >= n0
* Big-Omega f(n) = Omega(g(n)):
  + f(n) is lower bounded by a constant multiple of g(n).
  + | f(n) | >= c \* | g(n) | for some c and all n >= n0
* Big-Theta (Both Big-O and Big-Omega):
  + f(n) is bounded above and below by two constant multiples of g(n).
  + c1 \* | g(n) | <= | f(n) | <= c2 \* | g(n) | for some c1, c2 and all n >= n0
* **The Master Method**
  + Correctness
    - ****
    - <https://www.khoury.northeastern.edu/home/jaa/CSG713.04F/Information/Handouts/master.pdf>
  + Efficiency
    - A math equations and formulas

      Description automatically generated with medium confidence
    - **T(n) = a \* T(n / b) + O(n^d)**
* **Stirling’s Approximation**
  + Correctness
    - (My Notes)
    - Notes link

**Guiding Principles for Algorithm Design**

* Think of your Algorithm’s runtime in terms of Worse-Case, Average-Case, and Best-Case terms.
* Don’t consider constant time factors / constants, unless you are looking for minor optimizations.
* Focus on running time for large input sizes.
* A Fast Algorithm is a algorithm whose worse case runtime grows slowly as the input size increases.

### Divide & Conquer

This technique splits the problem in half (Divide), solves those problems recursively (Conquer), and combines the solution in the caller function (Combine).

**Counting the Number of Inversions**

By using the logic for Merge Sort, but with some additional logic, we can **count the number of inversions on the left side** of the array, **count the number of inversions on the right side** of the array, and **count the number of split inversions** (numbers on the left that are strictly greater than numbers on the right).

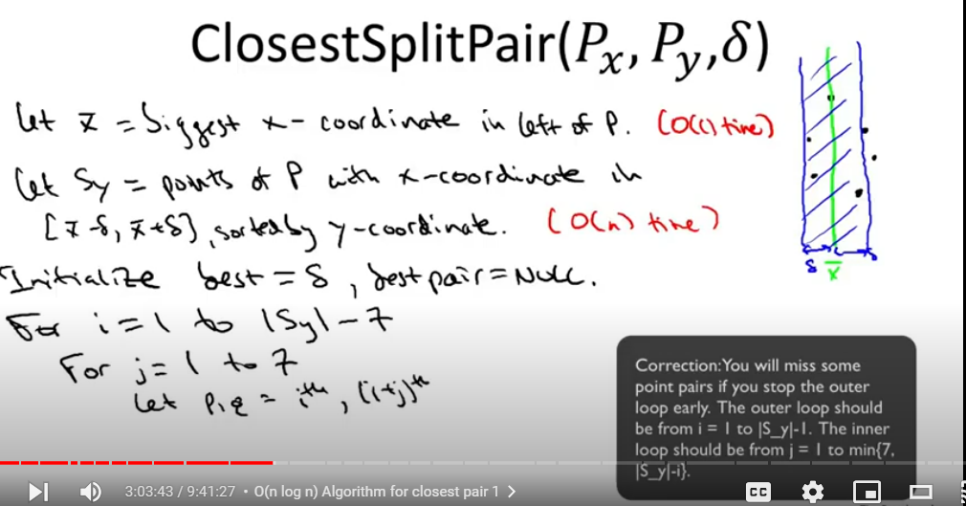
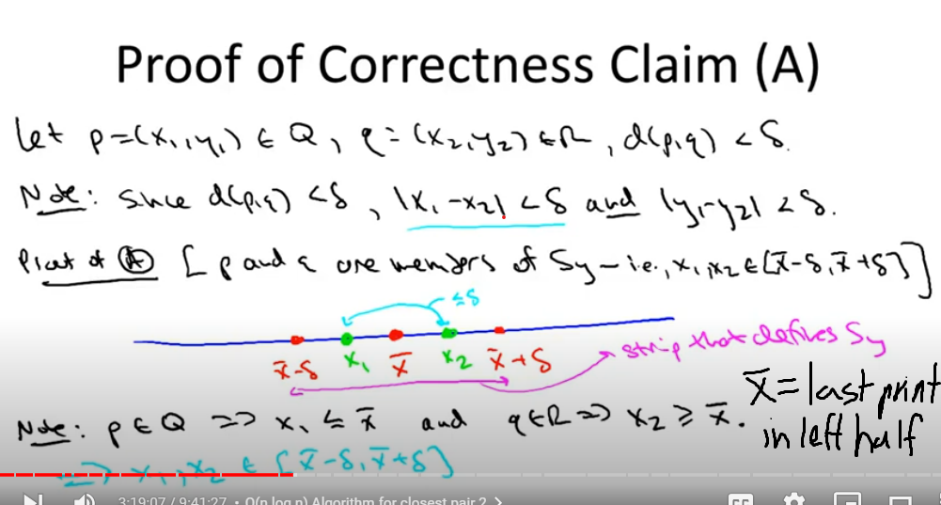
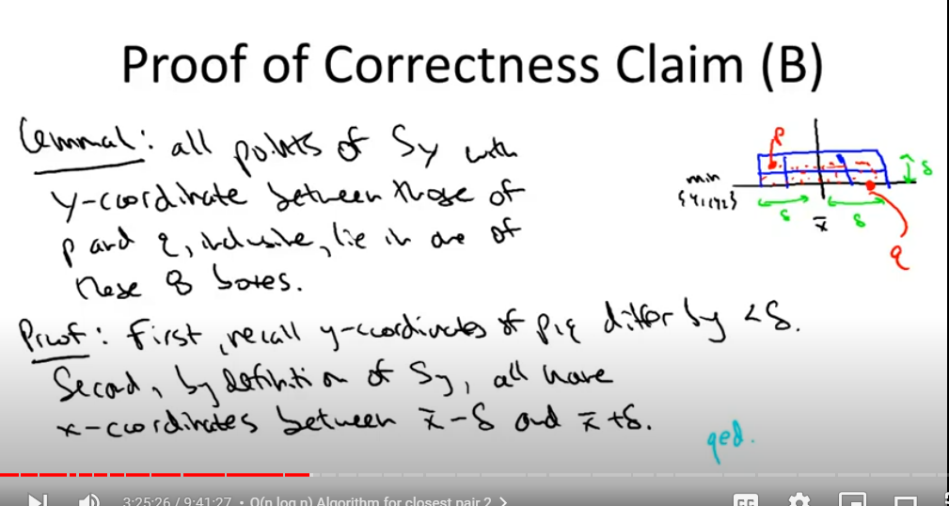
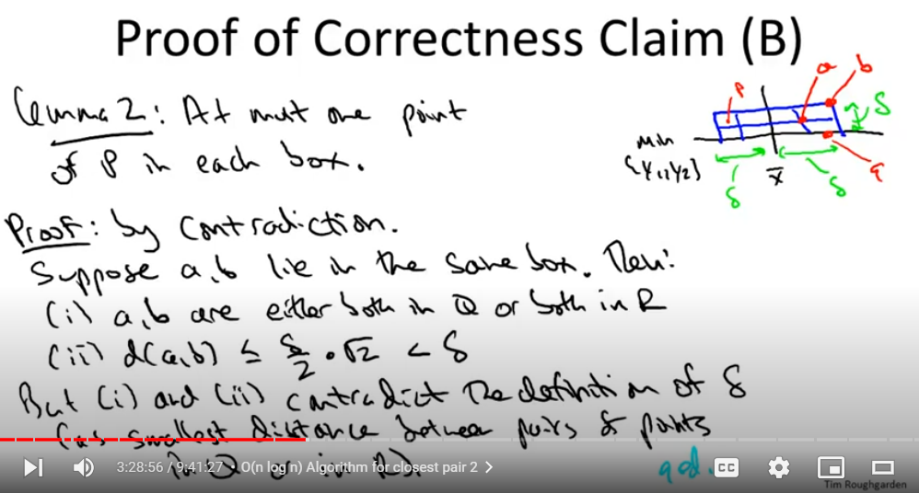
To accomplish this, when we make the recursive call on both sides, we also sort them (essentially performing Merge Sort). There are some cases to consider:

1. **Base Case**: Length of the array is 0 or 1. **Return the array.**
2. **Case 1:** When merging back, the current element on the right half of the array at index j is smaller than the rest of the elements on the left side starting at index i.
   1. **This implies that the value at index j on the right is inverted with all the values on the left that have not been processed yet.**
   2. i = 2, j = 2, l = [1, 2, 6], r = [4, 5, 7]
      1. Here, we take 6 from the left, but we know that we took 2 values from the right before we took 6, so that counts two inversions.
3. **Case 2:** When merging back, the first element of the right side of the array is not processed until all elements on the left side of the array are processed.
   1. **This implies that there are no inversions.**

* **Closest Pair Problem** (Computational Geometry)

Continuing to utilize the Divide and Conquer method, we can now tackle the problem of finding the pair of points that are the closest to each other in a given set.

To do this, we will split the problem in half again and solve for the pair of closest points in these splits. **The challenge comes** when **deciding how to split these pairs, how to sort these pairs, and what to do if the closest pair of points are on two different sides of the split**.

* + Correctness
    - 
    - <https://www21.in.tum.de/~nipkow/pubs/ijcar20-closest.pdf>
  + Efficiency
    - O(N\*Log(N))
  + 
  + 
  + 
  + 

## Part Two

### Graphs

V 🡺 Vertex, Vertices, Nodes, (n/N)

E 🡺 Edge, Edges, (SPARSE🡺MIN = V – 1, DENSE🡺MAX = V\*(V - 1) / 2), (m/M)

Edge Types 🡺 Directed, Undirected

Graph Representations 🡺 Adjacency List,(O(M + N) Space) Adjacency Matrix (O(N^2) Space)

**Cuts In Graphs**

A cut in a graph is a partitioning of the vertices in graph G such that G is split into two, non-empty sets. Not all edges must cross the cut and connected vertices can be in the same set. Crossing the cut implies that the tail (end of a directed edge) is in set B and the head (the end of the arrow) is in set A.

**The Minimum Cut Problem**

Given graph G, return the minimum set of edges that cross the cut between the two non-empty sets.

## Additional Algorithm Design Notes

1. Algorithmic Primitives (4)
   1. Sorting
      1. If the best algorithm without sorting is quadratic or worse, sorting is essentially free and gives an opportunity to bring the runtime down to N\*Log(N).
   2. ???
2. Big O Notation
   1. Understand it: Big O notation lets you analyze an algorithm's time and space complexity (how resource usage scales with input size). Focus on the dominant terms (e.g., O(n^2) is worse than O(n log n)).
   2. Aim for lower complexity: Favor linear time (O(n)) or logarithmic time (O(log n)) where possible.
   3. Common complexities: Get familiar with the time complexities of common operations and data structures (access, search, insertion, etc.).
3. Data Structures
   1. Pick the right tool: Choosing the right data structure is crucial.
   2. Arrays: Fast for random access, but resizing is expensive.
   3. Linked Lists: Efficient for insertions and deletions, less so for random access.
   4. Hash Tables: Often provide near-constant time lookup (O(1) on average).
   5. Trees: Enable logarithmic searches, useful for ordering.
   6. Heaps: Offer quick access to the minimum or maximum element.
   7. Know their strengths: Understanding the time complexity of operations within each data structure will guide your decisions.
4. Algorithmic Techniques
   1. Divide and Conquer: Break the problem into smaller, similar subproblems, solving them recursively. Examples: Merge Sort, Quick Sort.
   2. Greedy Algorithms: Make locally optimal choices at each step, hoping for a global optimum. Not always the best, but can be good for certain problems.
   3. Dynamic Programming: Store solutions to subproblems to avoid recalculations, trading space for time. Great for problems with overlapping subproblems.  
      Memoization: Specific kind of dynamic programming where results are cached.
   4. Backtracking: Explore possible solutions, pruning branches when they can't lead to a solution.
5. Optimization Techniques
   1. Profiling: Use a profiler to pinpoint bottlenecks in your code, focus your optimization efforts accordingly.
   2. Caching: Store frequently used results for quicker access.
   3. Pre-calculation: Compute expensive results beforehand if they'll be reused multiple times.
   4. Avoid nested loops: Nested loops often lead to quadratic (or worse) complexity. Find optimizations where possible.
   5. Bitwise operations: For certain tasks, bit manipulation can be faster than arithmetic operations.
6. Space-Time Tradeoffs
   1. Be aware of tradeoffs: Sometimes you can improve time efficiency at the cost of using more memory (and vice-versa).
   2. Consider problem constraints: If you have ample memory, a hash table might be better. If memory is scarce, you might optimize for space even with a slightly less time-efficient algorithm.