6.006 Final Review Session I

Sorting

Continuous Optimization/Computational Geometry/Complexity

Sorting

Given an array A[0...n-1] of n elements, output a sorted array.

In a comparison based model:

- Insertion sort
- Merge sort
- Heap sort

Not in a comparison based model:

- Counting sort
- Radix sort

Insertion Sort

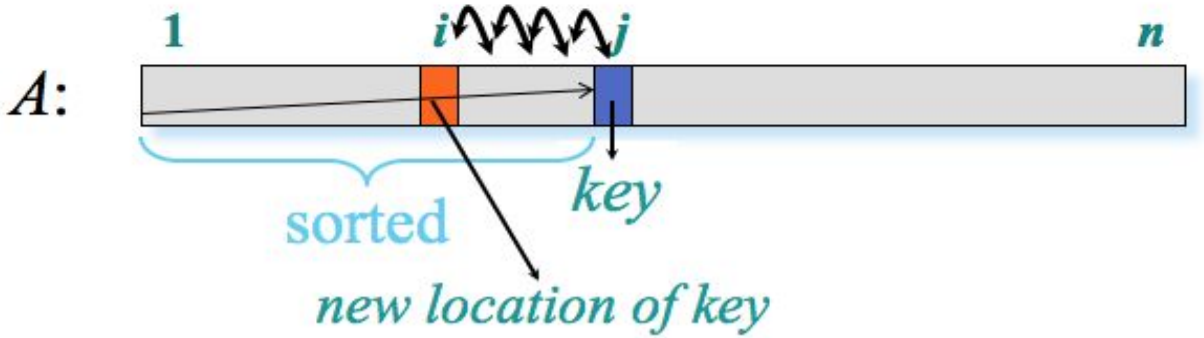
INSERTION-SORT (A, n) ▷ $A[1 \ldots n]$

for $j \leftarrow 2$ **to** n

insert key $A[j]$ **into the (already sorted) sub-array** $A[1 \ldots j\text{-}1]$**.**

by pairwise key-swaps down to its sorted position

Illustration of iteration *j*



Insertion Sort

Running time analysis => **$O(n^2)$**

- Worst case: $\Theta(n^2)$
  - Input in reversed order, so i comparisons and swaps in iteration i
  - $1+2+...+n = n(n-1) / 2 = \Theta(n^2)$
- Best case: $\Theta(n)$
  - Input in sorted order, so 1 comparison in each of the n iteration

Note: Insertion sort is an **in place sorting algorithm,** as the input array A gets modified into sorted B using only a constant amount of additional memory. It can be implemented to be **stable**, by simply not swapping keys in the case of equal values.

Problem 5. Sorting Prefixes [22 points] (3 parts)

Suppose that we are given a sequence $A[1 \ldots n]$ of n distinct elements that are *k-prefix unsorted*, for some integer $0 \leq k \leq \frac{n}{2}$. That is, the length-$s$ suffix of A, where $s = n - k$, is already sorted in non-decreasing order. (The elements of the length-k prefix can be arbitrary and in arbitrary order.)

For example, $A = [3, 1, 5, 2, 4, 6]$ is a 3-prefix unsorted sequence of length 6.

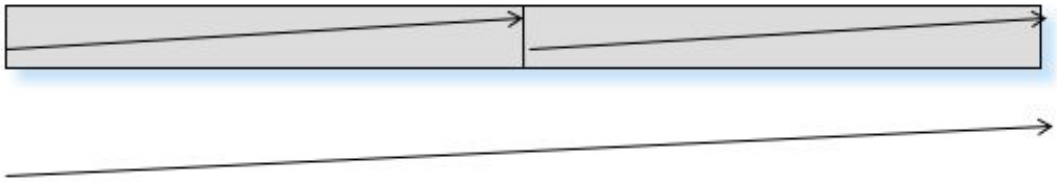
(a) [6 points] Provide an asymptotic estimate of the *worst-case* bound (in terms of n and k) on the number of *comparisons* that Insertion Sort makes on such k-prefix unsorted sequences.

Merge Sort

divide and conquer

MERGE-SORT $A[1 \ldots n]$

1. If $n = 1$, done (nothing to sort).
2. Otherwise, recursively sort $A[\,1 \ldots n/2\,]$ and $A[\,n/2+1 \ldots n\,]$.
3. "***Merge***" the two sorted sub-arrays.



Key subroutine: **MERGE**

Merge Sort

merge subroutine:

1. Takes two sorted lists
2. Create a pointer to the head of each list
3. Compare the elements at the pointers with each other, add the smaller element to our merged list, and advance the pointer associated with the smaller element
4. Repeat step 3 until both sorted lists are empty

Each iteration of step 3 takes O(1), and each iteration adds one element to our merged list. So the runtime of the merge operation is Θ(n) where n is the number of elements we are merging.

Merge Sort

Running time analysis:

MERGE-SORT $A[1 \ldots n]$	$T(n)$
1. If $n = 1$, done.	$\Theta(1)$
2. Recursively sort $A[1 \ldots \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1 \ldots n]$.	$2T(n/2)$
3. *"Merge"* the two sorted lists	$\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

$$T(n) = ?$$

Master Theorem

The Master Theorem applies to recurrences of the following form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

There are 3 cases:

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 $f(n) = \Theta(n^{\log_b a} \log^k n)$ with[1] $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$.
   Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

Merge Sort

Recurrence: $T(n) = 2\,T(n/2) + \Theta(n)$

$a = 2, b = 2, f(n) = \Theta(n)$

So we're in case two, and we have $T(n) = \Theta(n \log n)$.

Note: Merge sort (as described in this class) is **NOT in place,** as we may need $\Theta(n)$ extra space for each merge operation and up to $\Theta(n \log n)$ space for the recursive stack. It can be implemented to be **stable**, by simply incrementing the left pointer in the case of a tie in the merge subroutine.

Heap Sort

Given an unsorted array with n elements,

1. Build_heap on the array to create a max heap — $\Theta(n)$
2. Extract_max n times — $n * \Theta(\log n)$

Running time => $\Theta(n \log n)$

Note: Heap sort is **in place,** as we can heapify the input array, and each extract_max simply moves the max element to the end of the input array. No need for extra space! It is **NOT stable**, because the ordering of elements in the original sequence is lost upon building the heap.

Counting sort

Given an array A of n elements with keys in the range {1, 2, ..., k}

- Make a single pass through A and maintain an auxiliary array C with k elements, each index i corresponding to the number of times the value i appears in A
- Transform C to an array in which each index i corresponds the number of elements <= i in A, by starting from the front of C and at setting each index i to be C[i] (number of elements = i) + C[i-1] (number of elements <= i-1).
- Iterate backwards through A, and for every element A[i] = j
    - place the element in position C[j] in B, the output array
    - decrement C[j]

Counting sort

Running time analysis:

- Count elements in A to create C — Θ(n)
- Transform C to actual indices — Θ(k)
- Iterate backwards through A placing elements into B — Θ(n)

=> Θ(n + k)

Note: Counting sort is **NOT in place,** as we must maintain the auxiliary counting array C. It is **stable**, because we iterate backwards through A and decrement C[j], so the first element of value j we find from the end will be placed after all subsequent elements with value j.

Radix Sort

Given an array A of n elements in which each element can be expressed as a sequence of individually comparable characters (like digits or letters).

- Pick a stable sorting algorithm (like counting sort!)
- Starting from the last digit (LSD), sort the whole list by this last digit only, and then the next-to-last digit, and so on until you sort the whole list by the first digit (MSD)

Why does this work? If x < y, then in the first (most significant) digit where x and y differ, x has a smaller value than y, so x gets sorted to before y at that step and thereafter stays before it because the remaining sorts are stable.

Radix Sort

Running time analysis:

If there are *n* words of length *d* in an alphabet of size *b*, then each counting sort pass takes Θ(*n*+*b*) time, for a total time of **Θ((*n*+*b*) *d*)**.

Note that when we're trying to sort numbers, we can often choose the base system we represent the numbers in. So let's we know that every element lies in some range {0, 1, … 2^{h-1}}, for some h = c log n. Then if we use a base n system to represent our numbers, then each number will have O(c) digits, and each pass of counting sort will take Θ(*n*+*n*) time, for a total time of **Θ((*n*+*n*) *c*) = Θ(n)**! Linear time sorting!

Data Structures!

(b) [5 points] Which of the following are TRUE statements about sorting algorithms? In all cases, n is the number of elements in the input array.

☐ (i) Merge Sort takes time $\Theta(n \log n)$ in the worst case, and time $\Theta(n)$ if the input array happens to be already sorted.

☐ (ii) Insertion Sort takes time $\Theta(n^2)$ in the worst case, and time $\Theta(n)$ if the input array happens to be already sorted.

☐ (iii) Heap Sort based on a max-heap takes time $\Theta(n \log n)$ in the worst case, and time $\Theta(n)$ if the input array happens to be already sorted in decreasing order.

☐ (v) Radix Sort for base 10 numbers of length at most d takes time $\Theta(dn)$ in the worst case.

(a) [12 points] Design a data structure to keep track of the k middle sales prices for the current year so far. Your data structure must support two operations:

- QUERY(): Return the middle k sales prices for the year so far. (If there are fewer than k sales so far then return all of their prices.) This should run in time no worse than $O(k)$.
- ADD(x): Add information about the next sale, with sales price x, into the data structure. This should run in time no worse than $O(\log(n) + k)$, where n is the current number of recorded sales.

Describe clearly the data you would maintain and how it would be organized (e.g., "Store a list of such and such sorted in such order").

Part III:

HASHING

(c) T F [4 points] Consider an empty ($n = 0$) hash table with $m = 4$ initial slots. Assume that the table maintains a load factor ($\alpha = n/m$) of at most 2 via table doubling (i.e., doubling the table size whenever the load factor is exceeded). Then, after inserting 28 keys into the table (and not deleting any), the hash table must have $m \geq 32$.

(b) T F [4 points] Consider a hash table storing n keys in m slots, with resolutions resolved by chaining, and hash function

$$h(k) = ((k^2 + 3) \mod 8) \mod m.$$

Then DELETE has expected running time $\Theta(n/m)$.

(d) **T** **F** [4 points] Suppose we dynamically resize a hash table by the following rules:

(i) If the table becomes full (you already have m items, and you try to insert one more), the table doubles in size: $m \to 2m$.

(ii) If the table becomes less than half full ($< \frac{m}{2}$ elements), the table halves in size: $m \to \frac{m}{2}$.

Then the amortized time complexity for resizing the table when adding or removing an element is $O(1)$.

(a) **T** **F** [4 points] Consider a hash table storing n items in m slots, with collisions resolved by chaining. Assume simple uniform hashing. Then SEARCH has expected running time $O(1)$.

Sums of pairs

Alyssa P. Hacker has a large unsorted list A of n numbers, and a target number k. In $O(n)$ expected time, she wants to compute the number of pairs of numbers in A that sum to k. Assume for this problem that all arithmetical operations between two numbers take $O(1)$ time. Also, note that A may contain repeated numbers.

For example, if $A = [-1, -1, 3.5, 4.5, 9, 4, 4, 5, -2]$ and $k = 8$, then the answer would be 4 because $(-1, 9)$ can be made in two ways, $(4, 4)$ and $(3.5, 4.5)$ can be made in one way, and these are the only pairs that sum to 8.

Describe an O(n) expected time algorithm for this problem.

Close duplicates

Given an array A of n integers and an integer k, detect if there is an entry $A[i]$ that is equal to one of the k previous entries $A[i-1] \ldots A[i-k]$. Your algorithm should run in time $O(n)$. You can assume you have access to a hash function which satisfies the simple uniform hashing assumption (SUHA).

Example: Given an array $A = [1, 3, 5, 7, 6, 5, 2]$ and $k = 4$, the algorithm should output YES since $A[3] = A[6] = 5$.

Continuous Optimization

Gradient Descent I [Finding minimum of f]

- Conditions on f
  - f should be continuous
  - f should be infinitely differentiable
- By Taylor Series $f(x+\varepsilon) = f(x) + f'(x)\times\varepsilon$
- **Iterative approach** (In each step move)
  - In direction of -f'(x)
  - By amount $|\eta f'(x)|$
    - Good choice of η : 1/f''(x)

Continuous Optimization

Gradient Descent II [Finding minimum of f]

- Start with x_0
- For k= 0 … do

  x_new = x_old - $\eta \nabla$f(x_old)

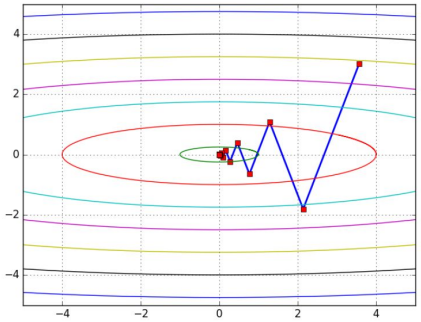
$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix} \quad \text{and} \quad \nabla^2 f = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

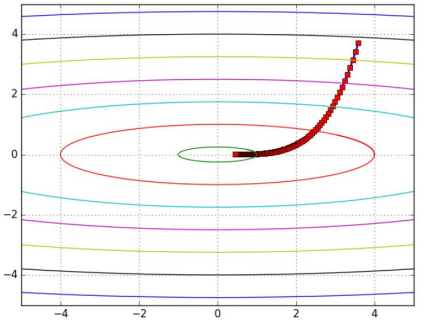
Convexity implies that local minimum is the global minimum too

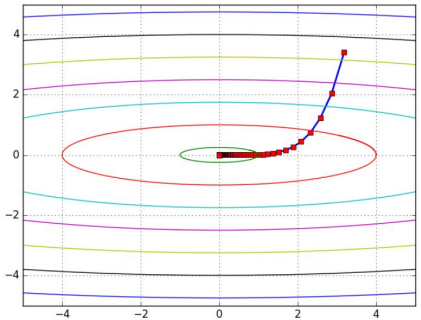
Continuous Optimization

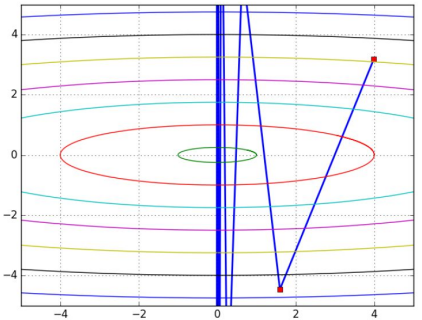
Gradient Descent II [Finding minimum of f]

Importance of the learning rate.









Continuous Optimization

Gradient Descent I [Finding minimum of f]

T F The gradient descent algorithm applied to the function

$$f(x) = (x+1)^2(x-1)(x-2) = x^4 - x^3 - 3x^2 + x + 2$$

is guaranteed to converge to a global minimum of that function.

T F Every function that has a single global minimum is convex.

Continuous Optimization

Gradient Descent II [Finding minimum of f]

Application to **Root finding**

square root of 2
Define a function whose derivative is x^2 - 2 = 0

Continuous Optimization

Newton Method [Finding roots of f]

- Finding the roots of a function

$$x\_new = x\_old - f(x\_old)/ f'(x\_old)$$

- Importance of x_0
- Convergence rate (quadratic convergence)

Exercise:

- Problem 4 in Spring 2016
- Problem 1-i/2-h Fall 2015
- Problems 1-l/2-h Spring 2015
- Problem Spring 2014 (1-f on square roots only)

Continuous Optimization

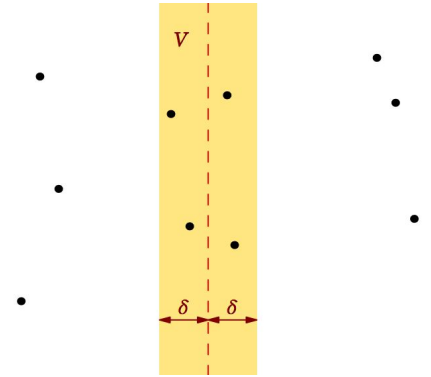
T F Newton's method is guaranteed to converge, from any starting point $x^{(0)}$.

T F In order to find the value of e^4, we can run Newton's method on the function $f(x) = \ln x - 4$ with an appropriate choice of starting point $x^{(0)}$.

More Exercises:

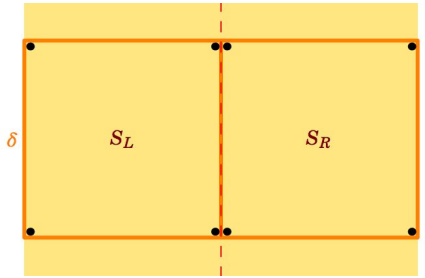
- Problem 1-i/2-h Fall 2015
- Problems 1-l/2-h Spring 2015
- Problem Spring 2014 (1-f on square roots only)

Computational Geometry



Output Sensitive Algorithms

- The running time depends on the size of output
- A bit different from worst case analysis
- e.g. Line segment intersection problem



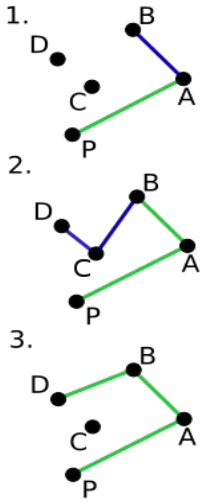
Sweep Line Technique

- Sweep a vertical line from left to right
- **Event**: Insert/Delete endpoints --- Intersection
- Maintain order of lines in an AVL tree

Closest Pair of Point Problem

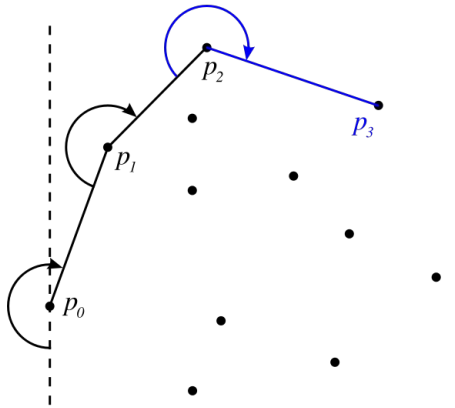
- Divide and Conquer
- Argue that in merge part, for each point, only its distance to a constant points in other parts should be computed

Computational Geometry



Convex Hull Problem

- Graham's Scan
  - Check for the lowest y-coordinate point p_0
  - Sort points by their polar angle relative to p_0
  - If adding p_i makes a **left turn** keep it
    Otherwise, remove recently added nodes till it becomes a left-turn
- Jarvis' march
  - Start with p_0
  - From each p_i, find the point with smallest polar angle relative to the previous edge



Complexity Theory

Complexity Classes

- **P** : Solvable in polynomial time [Most of the problems covered in this course, sorting/shortest path,...]
- **NP**: Given a candidate solution, can be easily verified
  - Is **P** \subseteq **NP**?
  - **NP-hard**
  - **NP-Complete**
- **EXP**: Solvable in exponential time 2^(poly n)
- **R**: Not solvable in finite time [Halting Problem]

Complexity Theory

Reduction

- Applicable both in **algorithm design** and **NP-hardness proof**
- Algorithmic Implication: If A reduces to B in **polynomial time** (A→ B) then an algorithm to B can be used to solve A.
- Hardness: If A reduces to B in **polynomial time** (A→ B) then B is at least as hard as A.
    - If **A** is NP-hard, then **B** is NP-hard

Complexity Theory

Exercise:

- Problem 1-i Spring 2014