algorithms

view on github

asymptotics

- big-o
  - big-oh: o(g): functions that grow no faster than g - upper bound, runs in time less than g
    - $f(n) \leq c \cdot g(n)$ f(n)≤c·g(n)
    - set of functions s.t. there exists c,k>0, 0 ≤ f(n) ≤ c*g(n), for all n > k
  - big-theta: θ(g): functions that grow at the same rate as g
    - big-oh(g) and big-theta(g) - asymptotic tight bound
  - big-omega: ω(g): functions that grow at least as fast as g
    - f(n)≥c*g(n) for some c, large n
  - example: f = 57n+3
    - o(n^2) - or anything bigger
    - θ(n)
    - ω(n^.5) - or anything smaller
    - input must be positive
    - we always analyze the worst case run-time
  - little-omega: omega(g) - functions that grow faster than g
  - little-o: o(g) - functions that grow slower than g
    - we write f(n) ∈ o(g(n)), not f(n) = o(g(n))
  - they are all reflexive and transitive, but only θ is symmetric.
    - θ defines an equivalence relation.
- add 2 functions, growth rate will be $O(max(g_1(n) + g_2(n))$ O(max(g1(n)+g2(n))
- recurrence thm:$f(n) = O(n^c) => T(n) = 0(n^c)$ f(n)=O(nc)=>T(n)=0(nc)
  - $T(n) = a * T(n/b) + f(n)$ T(n)=a∗T(n/b)+f(n)
  - $c = log_b(a)$c=logb(a)
- over bounded number of elements, almost everything is constant time

recursion

- moving down/right on an nxn grid - each path has length (n-1)+(n-1)
  - we must move right n-1 times
  - ans = (n-1+n-1 choose n-1)
  - for recursion, if a list is declared outside static recursive method, it shouldn't be static
- *generate permutations* - recursive, add char at each spot
- think hard about the base case before starting
  - look for lengths that you know
  - look for symmetry
- n-queens - one array of length n, go row by row

dynamic programming

```
//returns max value for knapsack of capacity W, weights wt, vals val
int knapSack(int W, int wt[], int val[])
int n = wt.length;
int K[n+1][W+1];
//build table K[][] in bottom up manner
for (int i = 0; i <= n; i++)
   for (int w = 0; w <= W; w++)
           if $(i==0 \vert \vert  w==0)$ // base case
                   K[i][w] = 0;
           else if (wt[i-1] <= w) //max of including weight, not including
                   K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);
           else //weight too large
                   K[i][w] = K[i-1][w];
return K[n][W];
```

min-cut / max-cut

hungarian

- assign n things to n targets, each with an associated cost

max-flow

- a list of pipes is given, with different flow-capacities. these pipes are connected at their endpoints. what is the maximum amount of water that you can route from a given starting point to a given ending point?

sorting

- you can assume w.l.o.g. all input numbers are unique
- sorting requires ω(nlog n) (proof w/ tree)
    - considerations: worst case, average, in practice, input distribution, stability (order coming in is preserved for things with same keys), in-situ (in-place), stack depth, having to read/write to disk (disk is much slower), parallelizable, online (more data coming in)
- adaptive - changes its behavior based on input (ex. bubble sort will stop)

comparised-based

bubble sort

- keep swapping adjacent pairs

```
for i=1:n-1
        if a[i+1]<a[i]
                swap(a,i,i+1)
```

- have a flag that tells if you did no swaps - done
- number of passes ~ how far elements are from their final positions
- o(n^2)

odd-even sort

- swap even pairs
- then swap odd pairs
- parallelizable

selection sort

- move largest to current position for i=1:n-1 for j=1:n x = max(x,a[j]) jmax = j swap(a,i,j)
- 0(n^2)

insertion sort

- insert each item into lists for i=2:n insert a[i] into a[1..(i-1)] shift
- o(n^2), o(nk) where k is max dist from final position
- best when alsmost sorted

heap sort

- insert everything into heap
- kepp remove max
- can do in place by storing everything in array
- can use any height-balanced tree instead of heap
  - traverse tree to get order
  - ex. b-tree: multi-rotations occur infrequently, average o(log n) height
- 0(n log n)

smooth sort

- adaptive heapsort
- collection of heaps (each one is a factor larger than the one before)
- can add and remove in essentially constant time if data is in order

merge sort

- split into smaller arrays, sort, merge
- t(n) = 2t(n/2) + n = 0(n log n)
- stable, parallelizable (if parallel, not in place)

quicksort

- split on pivot, put smaller elements on left, larger on right
- o(n log n) average, o(n^2) worst
- o(log n) space

shell sort

- generalize insertion sort
- insertion-sort all items i apart where i starts big and then becomes small
  - sorted after last pass (i=1)
- o(n^2), o(n^(3/2), … unsure what complexity is
  - no matter what must be more than n log n
- not used much in practice

not comparison-based

counting sort

- use values as array indices in new sort
- keep count of number of times at each index
- for specialized data only, need small values
- 0(n) time, 0(k) space

bucket sort

- spread data into buckets based on value
- sort the buckets
- o(n+k) time
- buckets could be trees

radix sort

- sort each digit in turn
- stable sort on each digit
  - like bucket sort d times
- 0(d*n time), 0(k+n) space

meta sort

- like quicksort, but 0(nlogn) worst case
- run quicksort, mergesort in parallel
  - stop when one stops
- there is an overhead but doesn’t affect big-oh analysis
- ave, worst-cast = o(n log n)

sorting overview

- in exceptional cases insertion-sort or radix-sort are much better than the generic quicksort / mergesort / heapsort answers.
- merge a and b sorted - start from the back

searching

- binary sort can’t do better than linear if there are duplicates
- if data is too large, we need to do external sort (sort parts of it and write them back to file)
- write binary search recursively
  - use low<= val and high >=val so you get correct bounds
  - binary search with empty strings - make sure that there is an element at the end of it
- “a”.compareto(“b”) is -1
- we always round up for these
- finding minimum is ω(n)
  - pf: assume an element was ignored, that element could have been minimum
  - simple algorithm - keep track of best so far
  - thm: n/2 comparisons are necessary because each comparison involves 2 elements
  - thm: n-1 comparisons are necessary - need to keep track of knowledge gained
    - every non-min element must win atleast once (move from unkown to known)
- find min and max

  - naive solution has 2n-2 comparison
  - pairwise compare all elements, array of maxes, array of mins = n/2 comparisons
    - check min array, max array = 2* (n/2-1)
  - 3n/2-2 comparisons are sufficient (and necessary)
    - pf: 4 categories (not tested, only won, only lost, both)
    - not tested-> w or l =n/2 comparisons
    - w or l -> both = n/2-1
    - therefore 3n/2-2 comparisons necessary
- find max and next-to-max
  - thm: n-2 + log(n) comparisons are sufficient
  - consider elimination tournament, pairwise compare elements repeatedly
    - 2nd best must have played best at some point - look for it in log(n)
- selection - find ith largest integer
  - repeatedly finding median finds ith largest
  - finding median linear yields ith largest linear
    - t(n) = t(n/2) + m(n) where m(n) is time to find median
  - quickselect - partition around pivot and recur
    - average time linear, worst case o(n^2)
- median in linear time - quickly eliminate a constant fraction and repeat
  - partition into n/5 groups of 5
    - sort each group high to low
    - find median of each group
    - compute median of medians recursively
    - move groups with larger medians to right
      - move groups with smaller medians to left
    - now we know 3/10 of elements larger than median of medians
      - 3/10 of elements smaller than median of medians
    - partition all elements around median of medians
      - recur like quickselect
      - guarantees each partition contains at most 7n/10 elements
    - t(n) = t(n/5) + t(7n/10) + o(n) -> f(x+y)≥f(x)+f(y)
    - t(n) ≤ t(9n/10) + o(n) -> this had to be less than t(n)

computational geometry

- range queries
  - input = n points (vectors) with preprocessing
  - output - number of points within any query rectangle
  - 1d
    - range query is a pair of binary searches
    - o(log n) time per query
    - o(n) space, o(n log n) preprocessing time
  - 2d
    - subtract out rectangles you don’t want
    - add back things you double subtracted
    - we want rectangles anchored at origin
  - nd
    - make regions by making a grid that includes all points
    - precompute southwest counts for all regions - different ways to do this - tradeoffs between space and time
    - o(log n) time per query (after precomputing) - binary search x,y
- polygon-point intersection
  - polygon - a closed sequence of segments
  - simple polygon - has no intersections
  - thm (jordan) - a simple polygon partitions the plane into 3 regions: interior, exterior, boundary
  - convex polygon - intersection of half-planes
  - polytope - higher-dimensional polygon
  - raycasting
    - intersections = odd - interior, even - exterior
    - check for tangent lines, intersecting corners
    - o(n) time per query, o(1) space and time
  - convex case
    - preprocessing

            - find an interior point p (pick a vertext or average the vertices)
            - partition into wedges (slicing through vertices) w.r.t. p
            - sort wedges by polar angle
        - query
            - find containing wedge (look up by angle)
            - test interior / exterior
                - check triangle - cast ray from p to point, see if it crosses edge
        - o(log n) time per query (we binary search the wedges)
        - o(n) space and o(n log n) preprocessing time
    - non-convex case
        - preprocessing
            - sort vertices by x
            - find vertical slices
            - partition into trapezoids (triangle is trapezoid)
            - sort slice trapezoids by y
        - query
            - find containing slice
            - find trapezoid in slice
            - report interior/ exterior
        - o(log n) time per query (two binary searches)
        - o(n^2) space and o(n^2) preprocessing time
- convex hull
    - input: set of n points
    - output: smallest containing convex polygon
    - simple solution 1 - jarvis's march
    - simple solution 2 - graham's scan
    - mergehull
        - partition into two sets - computer mergehull of each set
        - merge the two resulting chs
            - pick point p with least x
            - form angle-monotone chains w.r.t p
            - merge chains into angle-sorted list
            - run graham's scan to form ch
        - t(n) = 2t(n/2) + n = 0(n log n)
        - generalizes to higher dimensions
        - parallelizes
    - quickhull (like quicksort)
        - find right and left-most points
            - partition points along this line
            - find points farthest from line - make quadrilateral
                - eliminate all internal points
            - recurse on 4 remaining regions
            - concatenate resulting chs
        - o(n log n) expected time
        - o(n^2) worst-case time - ex. circle
        - generalizes to higher dim, parallelizes
    - lower bound - ch requires ω(n log n) comparisons
        - pf - reduce sorting to convex hull
        - consider arbitrary set of x_i to be sorted
        - raise the x_i to the parabola (x_i,x_i^2) - could be any concave function
        - compute convex hull of the parabola - all connected and line on top
        - from convex hull we can get sorted x_i => convex hull did sorting so at least n log n comparisons
        - corollary - graham's scan is optimal
    - chan's convex hull algorithm
        - assume we know ch size m=h
        - partitoin points into n/m sets of m each
- convex polygon diameter
- voronoi diagrams - input n points - takes o(nlogn) time to compute
    - problems that are solved
        - voronoi cell - the set of points closer to any given point than all others form a convex polygon
        - generalizes to other metrics (not just euclidean distance)
        - a voronoi cell is unbounded if and only if it's point is on the convex hull
            - corollary - convex hull can be computed in linear time
        - voronoi diagram has at most 2n-5 vertices and 3n-6 edges

- every nearest neighbor of a point defines an edge of the voronoi diagram
  - corollary - all nearest neighbors can be computed from the voronoi diagram in linear time
  - corollary - nearest neighbor search in o(log n) time using planar subdivision search (binary search in 2d)
- connection points of neighboring voronoi diagram cells form a triangulation (delanuay triangulation)
- a delanuay triangulation maximizes the minimum angle over all triangulations - no long slivery triangles
  - euclidean minimum spanning tree is a subset of the delanuay triangulation (can be computed easily)

- calculating voronoi diagram
  - discrete case / bitmap - expand breadth-first waves from all points
    - time is o(bitmap size)
    - time is independent of #points
  - intersecting half planes
    - voronoi cell of a point is intersection of all half-planes induced by the perpendicular bisectors w.r.t all other points
    - use intersection of convex polygons to intersect half-planes (nlogn time per cell)
  - can be computed in nlogn total time
    1. idea divide and conquer
       - merging is complex
    2. sweep line using parabolas