“Programming = Algorithms + Data Structures” —WIrth

* Categories of algorithms (Sorting)
  + (1) greedy
  + (2) divide and conquer
  + (3) dynamic programming
  + (4) randomized
  + (5) backtracking
* Greedy algorithms
  + Aka hill-climbing
  + Cast problems as optimizing some quantity by taking sequence of steps
  + Always take “best” (locally optimal) step and never go back
  + Doesn’t always yield best solution
  + Ex: Change-making problem
    - Make change for n cents by using the fewest coins
    - Alg: use largest possible coin to reduce amount remaining; repeat
    - US coins: 100c, 50c, 25c, 10c, 5c, 1c
      * 223c = 100 + 100 + 10 + 10 + 1 + 1 + 1
      * 7 coins (optimal)
      * For the US coinage system, this greedy algorithm works
    - Odd coins: 100c, 40c, 25c, 5c, 1c
      * 75c = 40 + 25 + 5 + 5
      * 4 coins (not optimal) — 3 quarters is optimal instead
      * For this odd coinage system, this greedy algorithm does not work
  + Ex: processor scheduling
    - p identical processors
    - n independent tasks with runtimes t1, …, tn
    - assign tasks to processors to minimize max work/processor
      * p = 3, n = 8
      * ti = {6, 4, 4, 4, 2, 5, 6, 5}
      * online, it gets task times one at a time
      * Alg (attempt): assign shortest to process with least work
        + p1: 2 + 4 + 6 p2: 4 + 5 + 6 p3: 4 + 5
        + result = 15 (p2)
      * Alg (LW): assign task to processor with least work
        + p1: 6 + 5 p2: 4 + 4 + 5 p3: 4 + 2 + 6
        + result = 13 (p2) — better than first algorithm
      * alg (LWD): sort in decreasing order, apply LW
        + 6, 6, 5, 5, 4, 4, 4, 2
        + p1: 6 + 4 + 4 p2: 6 + 4 + 2 p3: 5 + 5
        + result = 14 (p1) — better than first algorithm, worse than second
* Sorting
  + n items: x[0], …, x[n]
  + rearrange x’s so that x[0] ≤ x[1] ≤ … ≤ x[n-1]
  + operations:
    - (1) compare x[i] and x[j], 0 ≤ i ≤ j ≤ n
    - (2) swap x[i] and x[j]
  + Sequence sorted if x[i] ≤ x[j] for all 0 ≤ i ≤ j ≤ n
  + Def: inversion pair (i, j) with i < j and x[i] > x[j]
  + Sorted is defined by the # of inversions being 0
  + Initially, # inversions ≤ # pairs (i, j) with 0 ≤ i ≤ j ≤ n
    - nC2 = n(n-1)/2
    - there are n-1 pairs with i = 0
    - there are n-2 pairs with I = 1
    - .
    - .
    - .
    - There are n(n-1)/2 pairs total

1/16/19

* Divide-and-conquer algorithms
  + Aka strong induction
  + Step 1: divide
    - Break the problem up into two or some subproblems of the same kind
  + Step 2: and
    - Solve each subproblem using same algorithm (recursively) or base case
  + Step 3: conquer
    - “Glue” solutions together to get solution to original problem
  + Ex: find largest and smallest of n numbers x1, …, xn
    - Trivial solution: find largest, find smallest
      * # of compares to find largest = n – 1
      * # of compares to find smallest = n – 1
      * # of compares total = n – 1 + n – 1 = **2n – 2**
    - Alg 1: two-way split (given n is even)
      * Divide: x1, …, xn/2 and xn/2 + 1, …, xn
      * And: finds largest and smallest of {x1, …, xn/2}and of {xn/2 + 1, …, xn}
      * Conquer: largest = larger of largests; smallest = smaller of smallests
      * Base case:
        + return (x1, x1) if n = 1
        + return (larger of x1 and x2, smaller of x1 and x2) if n = 2
        + C(n) = (3/2)n – 2 if n = 2k
      * recurrence relation C(n), # of compares
        + C(1) = 0
        + C(2) = 1
        + C(n) = 2C(n/2) + 2

= 2[3/2\*n/2 – 2] + 2

= 2[(3/4)n – 2] + 2

= 3/2n – 4 + 2

= (**3/2)n – 2**

Better than trivial solution

* + - Alg 2:
      * Base cases
      * Divide: split into n/2 groups of size 2
      * And: find larger and smaller of each pair
      * Conquer: largest = largest of largers; smallest = smallest of smallers
      * (n/2 \* 1) + (n/2 – 1) + (n/2 – 1) = **(3/2)n – 2**
      * If you were to look for the second largest of n items, it would actually prove easier to find both the largest and second largest, because that way you could use divide and conquer until you have the top two numbers of each half and compare them
  + Ex: sort n numbers
    - Alg 1: MergeSort
      * {10, 5, 9, 7, 6, 8, 4, 11}
      * Base cases:
        + n = 1 —> do nothing
        + n = 2 swap if x1 > x2
      * Divide: x1, …, xn/2 and xn/2 + 1, …, xn
        + {10, 5, 9, 7}
        + {6, 8, 4, 11}
      * And: sort each group
        + {5, 7, 9, 10}
        + {4, 6, 8, 11}
      * Conquer: merge sorted groups
        + Compare the lowest number of each group, then remove it and add it to a new sorted group
        + {4, 5, 6, 7, 8, 9, 10, 11}
      * Recursive relation
        + C(1) = 0
        + C(2) = 1
        + C(n) ≤ 2C(n/2) + n – 1

The (n – 1) addend is because the last merge comparison is unnecessary and does not happen

The ≤ is there instead of the = because there may be more unnecessary comparisons than just the last one

* + - * + C(n) = nlog2n + low order terms

Low order terms is O(n)

* + - * + —> **O(nlogn)**
    - Alg 2: QuickSort
      * Split by value
      * Base case:
      * Divide: pick one item S in the group to be splitter and break the group into two groups
        + <S and >S
      * And: sort each group
      * Conquer: sorted <S group | S | sorted >S group
      * C(n) = C(k) + C(n – 1 – k) + n – 1
        + Best case: k ≈ n/2

2C(n/2) + O(n)

**nlog2n + O(n)**

* + - * + Worst case: k = 0

**n(n – 1)/2**

* Dynamic programming
  + Build solution by solving all instances of structured class of subproblems in an efficient manner
  + Ex: make change for n cents using fewest coins
    - Find # coins (k) for all 0 ≤ k ≤ n
    - nCoins[0] = 0
    - for (k = 1; k ≤ n; k++) {
      * m = 1 + nCoins[k – 1]
      * if (k ≥ 5) m = min(m, 1 + nCoins[k – 5]);
      * if (k ≥ 10) m = min(m, 1 + nCoins[k – 10]);
      * .
      * .
      * .
      * nCoins[k] = m;
    - }
  + Similar to recursion and memoization
    - Remember values from previous recursive calls to avoid repeating them
    - nCoins(n) {
      * if (n == 0) return 0;
      * else return min (1 + nCoins(n-1), 1 + nCoins(n – 5), 1 + nCoins(n – 10)…)
    - }
  + Ex: sorting
    - Alg: sort all initial sublists of list — insertionSort
      * x1, …, xk —> y1, …, yk (y1 ≤ y2 ≤ … ≤ yk)
      * use solution for k – 1 to get solution for k
      * x1, …, xk-1, xk —> y1 ≤ y2 ≤ … ≤ yk
    - linear # compares to get y’s
    - total complexity:
      * C(n) = 0 + 1 + 2 + … + n – 1 = **n(n-1)/2**
      * Tree sort: nlog2n + O(n)