**Merge sort**

Pseudo code:

MergeSort(data, n, sorted)

If n <= 1 return InsertionSort(data, n)

DivideInput(data, n ,smallData, n/2, 2)

MergeSort(smallData[0], n/2, smallSorted[0])

MergeSort(smallData[1], n/2, smallSorted[1])

Sorted = combineSorted(smallSorted[0], smallSorted[1])

return

Let the time required for sorting n numbers be T(n), and we have the following recursion form

We can write the recursive form into a set of formula with different input sizes:

……….

Multiply for the ith equation.

………..

We can add up all equations and simplify them, the sum becomes:

Note that each term on the right hand side evaluates to O(n) and there are terms. So the overall time complexity is .

**Master’s theorem**

a – how many subproblems the large problem is divided into

b – how much smaller the subproblem is compared to the large problem

c – what is the exponent of the polynomial time complexity required by the result merge process

recursive form of the algorithm as:

Note: only apply to divide and conquer algorithm divides the larger problem into equal-sized smaller problems

For a-nary tree: now consider the ith layer in the a-nary tree

* We will have
* Each subproblem will have a size of
* Obtaining the solution of the subproblem will need time, where r is a constant term that converts the time complexity term into the actual time

Now the time required to compute the solution for all subproblems in the ith layer becomes:

Rearranging the term, we get:

Then, the overall time for the algorithm is simply the sum of the per-layer time overall all layers. Note that we have layers, which leads to an overall time of:

By analyzing the term , we realize that r, n, and c are constants for a given algorithm and input instance. The only term that is variable between different layers is .

While computing the sum of a series, it is important to know whether it is an increasing or a decreasing series.

* We need to discuss whether the term is increasing , invariable or decreasing as I increases

Case1: an increasing series

Case2: an invariable series

Case3: an decreasing series

Note: assumption that , so if

**Median Finding**

Pseudo code:

Select(A, i)

Divide the n items into groups of 5

Find the median of each group of 5

Use Select recursively to fine the median of there n/5 medians

Partition the n items around x, and let k be the ranking of x

If I = k then return x

Else if i < k use Select recursively by calling Select(A[1…k-1], i)

Else if i > k use Select recursively by calling Select(A[k+1 … A], i-k )

endif

Proposition: The divide-and-conquer median-finding algorithm runs in O(n) time

Proof by induction:

(base case): For an input size of 1, we can complete the median finding in time c by making it sufficiently large, such as T(1) <= c \* 1 = c

(Induction): We assume running time of the algorithm T(k) <= ck for 1 <= k <= n. Now, substitute the assumption to the running time formula and we get:

Since a is a constant which associated with the running time for the partitioning process around x, we can make a large-enough constant where c > 10a and

n be negative. If n is negative, then . Which is consistent with our hypothesis at the beginning.

(conclusion): Hence the algorithm runs in O(n) time.

So we will guess a lower time complexity say O(1)

Assuming the algorithm runs in O(1) and T(n) <= c for some constant c, we have:

The formula clearly indicates an O(n) time complexity, no matter how we manipulate the constant a and c, the O(n) complexity is inconsistent with what we have guessed, hence, the algorithm dos not run in O(1).

Smallest choice that makes algorithm run O(n) time

Case1: Assume d is an odd number, then we can write the running time as:

To make a proper upper bound, we shall make negative, in other word, should be positive.

It is clear that d > 3 and since we assumed d is an odd number , the smallest value d can take is 5. When is positive, we can make c large enough such that becomes negative, and thus make the algorithm remain in O(n) time.

Case2: assume d is an even number

We must have d > 4 if d is an even number, so d = 5 works as the smallest choice that make the algorithm runs in O(n) time if combine two situations.

**Integer Multiplication**

Consider the multiplication of 2 integers x an y each with n digits

Apply master’s theorem.

* a=4, since we need to compute 4 multiplications as subproblems
* b=2, the size of the subproblem is halved
* c=1, since adding the results of the 4 multiplications clearly takes liner time
* time

Notice that we only need the sun of but not necessarily nor . And so - .

We know that we need to compute and as subproblems and we will also compute as the third subproblem.

We can approximate the computation of as a subproblem with size n/2, since the sum of two n/2-digit number can have at most n/2 + 1 digits. And all additional additions and subtractions can be done in O(n) time.

Now we revise the analysis of time complexity:

* a=3, since we need to compute 3 multiplications as subproblems
* b=2, the size of the subproblem is halved
* c=1, since adding the results of the 4 multiplications clearly takes liner time
* , which is slightly faster than the naïve algorithm.

**Counting Inversions**

**Pseudo code:**

Merge-and-count(A,B)

Maintain a current pointer into each list, initialized to pointer to the front elements

Maintain a variable count for the number of inversion, initialized to 0

While both lists are noneempty:

Let ai, aj be the elements pointed to by the Current pointer

Append the smaller of these two to the output list

If bj is the smaller element then

Increment count by the number of elements remaining in A

Endif

Advance the current pointer in the list from which the

Smaller element was selected

Endwhile

Once one list is empty, append the remainder of the other list to the output.

Return count and the merged list

**Case**

Consider the following problem: we wish to study which courses in computer science are strongly tied to EECS 660 Algorithms.

– we consider two courses are strongly tied if students do well in EECS560 also tend to do well in the other class

– for example, students do well in EECS 660 also tend to do well in EECS 560 Data structure, but the tie could be weaker in, say, Physics classes

– of course, we can quantify the similarity in statistical sense, such as using Pearson’s or Spearman’s correlation

– but here, we would like to use another measure, called rank inversions

Say we have exactly the same set of students who have taken EECS 560 and EECS 660, and we have finished the grading with the students’ rankings in the two courses.

– for example, assume the ranking of EECS 660 is: Mary (1st), Tom (2nd), and Jacob (3rd)

– and the ranking of EECS 560 is: Tom, Jacob, and Mary.

– we can associate the ranking with the students in EECS 660, and map the rankings to EECS 560

• Mary is 1, Tom is 2, and Jacob is 3; In EECS 560, Tom (2) goes first, so we have 2 as the first mapped ranking; Jacob (3) goes second, so we have 3 as the second,… eventually we get the mapped ranking as 2, 3, 1

Then, we define an inversion in a list A as the number of cases where ai > aj if I < j. Here, ai is the ith element of A

* For example, consider the mapped ranking in EECS 560 which is 2, 3, 1
* For any combination of two elements {(2,3),(2,1),(3,1)} we know that (2,1) and (3,1) are inversion per the definition
* Thus the number of inversions we have in EECS 560’s mapped ranking is 2
* If two lists are identical the new have 0 inversions, if two lists are completely reversed, then for any pair of elements it is an inversioninversions in total
* In this case, we can use the number of inversions to quantitfy the similarity of two ranking.

Counting Inversion: Given an unsorted list of n numbers 1, 2 ……n where each number occurs exactly once, count the number of inversions in the list

To solve this problem using divide an conquer, note that we can divide the entire set of numbers into two subsets, and the total number of inversions we have in the entire list is the sum of the number of inversion in the first list, the number of inversion sin the second list, and the number of inversion between the two lists

* Using the naïve O(n^2) algorithm, we know that each of these three numbers can be computed in T(n) /4 and the sum of them is 3T(n) /4, so we should be getting some efficiency gain here
* However, per the master’s theorem, time complexity for the merge phase is in fact a lower found for the entire algorithm, we have that complexity being O(n^2), hence the naïve divide an conquer algorithm has no gain in theoretical efficiency

Hence, to improve the time complexity, we shall do better in merging the results of two subproblems.

Let A1 = {2, 6, 1} and A2 = {3, 5, 4} we can sort them individually into A1’ = {1,2,6} and A2’={3,4,5}

Per definition, we know that all numbers in A2’ should be larger than those in A1’, if there is no inversion.

So, for any number in A2, the number of inversions associated with it is exactly the number of numbers in A1, that are larger that it.

Consider A1’ = {1,2,6} and A2’={3,4,5}

* 3 is smaller than 6, 1 inversion
* 4 is smaller than 6, 1 inversion
* 5 is smaller than 6, 1 inversion
* 3 between-subset inversions in toal

We can implement the idea in a way similar to merge sort

* We set up three pointer (input A1’, A2’ output A’)
* When we need to move any number in A2’ into A’, indicates that the number is smaller than all of remaining numbers in A1’. Hence it will cause that many inversion
* This process takes O(1) time

The overall running time become:

And per the master’s theorem, time complexity is .

**Closest pair of points**

For the 1-dimensional case, we can solve this problem in O(n logn) time by sorting their points according to their 1-dimensional coordinates and scan the sorted points to find the closest pair

For the 2D case, each point is associated with a coordinate (x,y)

We must divide the points

* + Each point on our 2D plane can be split into two subspaces so each subspace contains half of the points. To simplify this process, we cut based on the x coordinate. More specifically, we cut the plane with a vertical line x = x\*, where half of the points have their x coordinates less than x\*. We denote the line x = x\* as L and the two subsets of points as Q and R.
  + Let the closest pair of points within Q or R as 𝛿. This will be found by recursively solving the problem on each subset
  + Let 𝛿 be the upper bound for the between-subset distance
    - If the distance is larger, we return 𝛿 instead.

Proposition: If there exists a q in Q and r in R for which d(q, r) < 𝛿, then each of q and r lies within a distance 𝛿 of L, where L is the line that separates Q and R

Proof: Denote the x coordinate of L as x\*. If q and r exist, denote their coordinates as q = (x­q, y­­q) and r = (xr, yr). By definition of the division of subproblems, we have xq <= x\* <= xr.Thus, we have the following:

x\*-xq <= xr-xq <= d(q, r) = 𝛿

xr-x\* <= xr-xq <= d(q, r) = 𝛿

So, each of q and r has an x coordinate within 𝛿 of x\* and hence lies within distance 𝛿 of L.

* Let all points within the band be S
* Let s be a point within S. All points that could lead to a distance smaller than 𝛿 with s must satisfy the following two conditions:
  + All points must not be within the same subset as s and must be within the circle.
* Within the circle, we can only hold a limited number of points within the same subset.
  + Thus, we claim that the max number of points that can exist within the circle is upper bounded by a constant.
* Assume that we have sorted all points in S by y coordinates. Let Sy be this sorted list.

**Proposition**: If s and s’ are in S and have the property of d(s, s’) < 𝛿, then s and s’ are within 15 position of each other in the sorted list Sy

Assume we have a band and split it into small boxes into a 𝛿/2 \* 𝛿/2 grid. This way, each box is of size (𝛿^2)/4. It is not possible that two points can be within the same box, since the farthest distance between two points within the box is (sqrt(2)\* 𝛿)/2, from the diagonal of the box. This contradicts with the minimum between-subset distance being 𝛿. So, the worst case of this method places one point in each box. So, if two points differ from each other more than 15 positions, then their distances will be at least (3𝛿)/2, which is greater than 𝛿. As a result of this claim, we need only to compute the distance between s and it’s next 15 points for any point s in Sy.

Pseudo code:

Closest-Pair(P)

Construct px and py

(P0\*, P1\*) = Closest-Pair-rec(px, py)

Closest-Pair-rec(px, py)

If |P| <= 3 then find closest pair by measuring all pairwise distances

Endif

Construct Qx, Qy, Rx, Ry

(q0\*, q1\*) = Closest-Pair-rec(Qx, Qy)

(r0\*, r1\*) = Closest-Pair-rec(Rx, Ry)

= min(d((q0\*, q1\*), (r0\*, r1\*)))

X\* = maximum x-coordinate of a point in set Q

L = {(x, y) : x = x\*}

S = points in P within distance of L

Construct Sy

For each point s Sy, compute distance from s to each of next 15 points in Sy, let s, s’ be pair achieving minimum of these distances

If d(s, s’) < then

Return(s, s’)

Else if d(q0\*, q1\*) < d(r0\*, r1\*) then

Return (q0\*, q1\*)

Else

Return (r0\*, r1\*)

Endif