Project 2

Anqi Liu G33627140

Option 0: Quick select, deterministic (median of medians method)

Median of medians

- Line up elements in groups of five (this number 5 is not important, it could be e.g. 7 without changing the algorithm much). Call each group S[i], with i ranging from 1 to n/5.
- Find the median of each group. (Call this x[i]). This takes 6 comparisons per group, so 6n/5 total (it is linear time because we are taking medians of very small subsets).
- Find the median of the x[i], using a recursive call to the algorithm. If we write a recurrence in which T(n) is the time to run the algorithm on a list of n items, this step takes time T(n/5). Let M be this median of medians.
- Use M to partition the input and call the algorithm recursively on one of the partitions, just like in quickselect.

throw away either L3 (the values greater than M) or L1 (the values less than M). Suppose we throw away L3. Among the n/5 values x[i], n/10 are larger than M (since M was defined to be the median of these values). For each i such that x[i] is larger than M, two other values in S[i] are also larger than x[i] (since x[i] is the median of S[i]). So L3 has at least 3 elements in each of at least n/10 groups S[i], for a total of at least 3n/10 elements. By a symmetric argument, L1 has at least 3n/10 elements. Therefore the final recursive call is on a list of at most 7n/10 elements and takes time at most T(7n/10).

Deterministic selection algorithm

```
select(L,k)
{
if (L has 10 or fewer elements)
{
    sort L
    return the element in the kth position
}

partition L into subsets S[i] of five elements each
    (there will be n/5 subsets total).

for (i = 1 to n/5) do
    x[i] = select(S[i],3)

M = select({x[i]}, n/10)

partition L into L1<M, L2=M, L3>M
if (k <= length(L1))
    return select(L1,k)
else if (k > length(L1)+length(L2))
```

```
        return select(L3,k-length(L1)-length(L2))
    else return M
    }
```

Theoretical Analysis

The pseudo-code above gives us a number of comparisons that can be found by solving the recurrence

```
    T(n) <= 12n/5 + T(n/5) + T(7n/10)
```

The 12n/5 term comes from two places: we can sort each of the sets S[i] with seven comparisons (homework 2.31), so the step in which we compute the x[i] values takes 7n/5 comparisons total. And then the step in which we partition L takes n-1 more comparisons. The other two terms come from the two recursive calls, in which we find M and then the overall return value. As we discussed earlier, the second recursive call is on a list of at most 7n/10 elements hence its T(7n/10) bound.
Actually with some more care we can do a little better: we don't really need to sort the sets S[i], just find their medians, which only requires 6n/5 comparisons. The resulting information, together with the computation of M, should already be enough to eliminate 3n/10 elements from L. So we could get a recurrence with 6n/5 in place of the 12n/5 above, and save a factor of two in the total comparisons. But since this result is mainly of theoretical interest, I've left it in the simpler and easier to understand form above.There are two ways to analyze this problem.

The first is the method, in which I try to form an inductive proof that something is O(n) by assuming it's cn for some specific c, expanding the right side of the recurrence, and working through the math to determine what c should be. In our case we have

```
    T(n) <= 12n/5 + T(n/5) + T(7n/10)

     = 12n/5 + cn/5 + 7cn/10

     = n (12/5 + 9c/10)
```

If this is to be at most cn, so that the induction proof goes through, we need it to be true that

```
    n (12/5 + 9c/10) <= cn

    12/5 + 9c/10 <= c

    12/5 <= c/10

    c <= 24
```

Which tells us that we can prove by induction that T(n) <= 24n (or any larger constant times n).

The second method to analyze a recurrence like this one is to draw a tree showing the sizes of the problems in each recursive call, and analyze the total size of problems on each level of the tree. The total number of comparisons can then be found by multiplying this total subproblem size by the 12/5 factor of comparisons per element in each call. The tree starts with a root problem of size n, and then each node has two subproblems, one of size 1/5 its parent, and the other of size 7/10 its parent.

```
          n
        /    \
      n/5         7n/10
     /   \        /    \
 n/25 7n/50 7n/50 49n/100
 / \   / \   / \   / \
```

Each problem on one level is replaced by two problems on the next level down, of sizes 1/5 and 7/10 the parent. So the total size on the next level is 1/5+7/10=9/10 that of the previous level (sometimes even less when a subproblem reaches a base case and doesn't make more recursive calls).
Therefore the total number of comparisons is

```
    12/5 (n + 9n/10 + 81n/100 + ...)

    = 12/5 n (1 + 9/10 + (9/10)^2 + (9/10)^3 + ...)

    = 12/5 n 1/(1-(9/10))

    = 24n
```

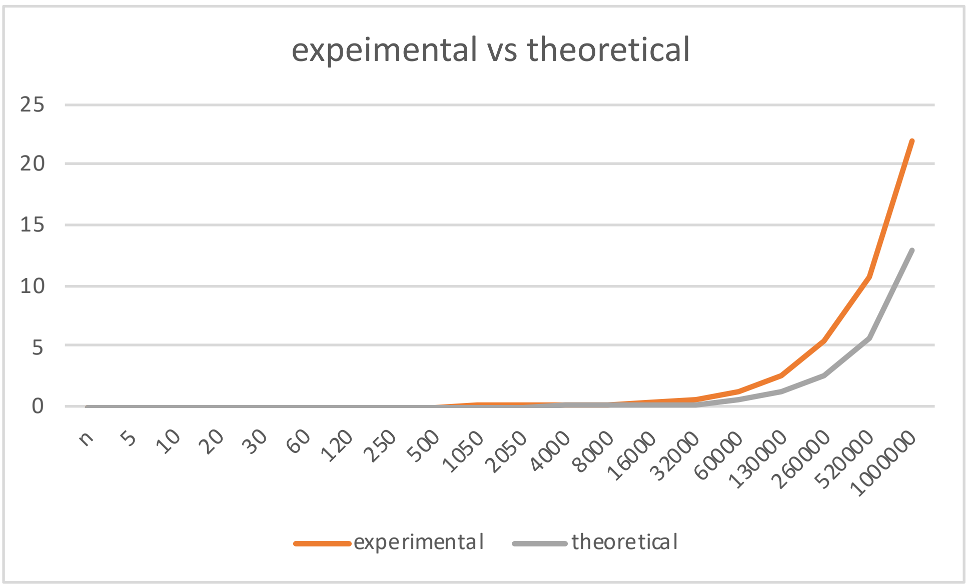
As a general rule, the geometric series sum(x^i) (for i from 0 to n–1) solves to (1–x^n)/(1–x), and whenever x is less than 1 the limit of the sum as n goes to infinity becomes 1/(1–x). The sum above is just a case of this formula in which x=9/10. The same tree-expansion method then shows that, more generally, if T(n) <= cn + T(an) + T(bn), where a+b<1, the total time is c(1/(1–a–b))n.
So our deterministic selection algorithm uses at most 24n comparisons and takes O(n) time.

Experimental Analysis

OUTPUT NUMERICAL DATA

n	experim ental	theoretic al
5	0	0
10	0	0
20	0	0
30	0	0
60	0.001	0
120	0.002	0
250	0.002	0
500	0.004	0.001
1050	0.008	0.002
2050	0.018	0.004
4000	0.037	0.009
8000	0.088	0.02
16000	0.166	0.047
32000	0.303	0.106
60000	0.649	0.236
130000	1.306	0.528
260000	2.603	1.152
520000	5.345	2.523
1000000	10.685	5.563
2000000	21.975	12.897

Graph



Conclusions

So our deterministic selection algorithm uses at most in linear n comparisons and takes O(n) time.

Option 1: Closest pair of points

Consider the "closest pair of points problem. Suppose we simply sort the points by their x-dimensions in the first step, in O (n log n) time instead of using the linear time median finding algorithm. How does this change the time complexity of the entire algorithm?
analysis:

If we sort the points by their x-dimensions, then the overall recurrence relation becomes

T(n) =o (n) +2 T(n/2)+O(n)

Using master theorem, we can solve this equation to T (n) =O (n logn).

Option 2,3: Staircase / Pareto-optimal Points

Let P be a set of n points in a 2-dimensional plane. A point p e P is Pareto optimal if no other point is both above and to the right of p. The sorted sequence of Pareto-optimal points describes a top-right staircase with the interior points of P below and to the left of the staircase. We want to compute this staircase

1. Suppose the points in P are already given in sorted order from left to right

Describe an algorithm to compute the staircase of P in O (n) time

2. Describe an algorithm to compute the staircase of P in O (n log n) time

analysis:

We observe that the right most point of the set P must be on the staircase. By the same logic, the top most point (that is, the point with the largest y-value) must also be on the staircase. Using this observation, we can compute the staircase in O (n) time if we are given the points in P in sorted order. We simply iterate the set of points from right to left, and include every point that has a y-value higher than the previously known maximum y-value.

Thus, we can compute the staircase in O (n log n) time.

Option 4,5: Convex hull

We are given a set P of n points in a two-dimensional plan, and we want to compute the convex hull of P. The convex hull is defined as the smallest convex polygon containing the points. (A way to visualize a convex hull is to imagine nails on all the points of the plane and put an elastic band around the points – the shape of the elastic band is the convex hull.) Describe an O(n log n) time divide and conquer algorithm to find the convex hull of the set P of n points.

analysis:
A simple divide and conquer algorithm simply divides the set of points by using the median in terms of their x-axis. We make two recursive calls on the left hand side and the right hand side, and then combine the two convex hulls. If we merge two convex hulls into one in linear time. Then our overall recurrence relation will look as follows:

T(n)=2T(n/2)+O(n)

Therefore, this question can be equivalently stated as, given two convex hulls, merge them into one in linear time.

To merge two convex hulls, one approach is as follows: find the"eastmost point in the left convex hull (that is, the point with the largest x-value), and the "west most point in the right convex hull (that is, the point with the smallest x-value) and connect them. Connect the next points in the individual convex hulls as well to create one large cycle. This can obviously be done in O (n) time. However, the resulting cycle may not be a convex polygon. We can now iterate through the cycle to eliminate the points where we have a"clock-wise turn, also in O (n) time.

Therefore, the entire merging algorithm runs in O (n) time, and the entire algorithm runs in O (n log n) time.

Option 6: Finding Max Number in Circular Shifted Array

We are given an array A[1..n] of sorted integers that has been circularly shifted
some positions to the right. For example, [35, 42, 5, 15, 27, 29] is a sorted array that has been circularly shifted 2 positions, while [27, 29, 35, 42, 5, 15] has been shifted 4 positions.
We can obviously find the largest element in A in O(n) time. Describe an O(log n) algorithm.

Solution:

First, notice that the smallest element in our array will tell us precisely how many times the array has been shifted, since the smallest element should go in index 0. This is equivalent to looking for an element in our array which is smaller than its neighbor to the left. Additionally, notice that given two elements on the circular sorted array, the smallest element in the array (if it isn't one of the selected elements) cannot be to the right of the smaller element and left of the larger. Therefore, we know it must exist on the other part of the array. Algorithm: Begin by choosing an arbitrary element of the list - for the purpose of this example, we'll pick the first element. Then, if we did not find the smallest element (by checking the element to the left of it), we can check the n/2 th element in the list. If that element is greater than the first element, then we know that the smallest element must be in the second-half of the array. If that element is less than the first element, then we know that the smallest element in the array must be in the first-half of the array. In any case, we know that the half-array must still be a circular sorted array, since we only remove elements, which cannot break the sortedness of an array.
Therefore, we can recurse on this half array, guaranteeing that the smallest element overall is on the half array. When the array size is 1, we certainly must have found the smallest element.

Option 7: Kruskal's Algorithm for Minimum Spanning Tree

Kruskal's algorithm

We'll start with Kruskal's algorithm, which is easiest to understand and probably the best one for solving problems by hand.

```
Kruskal's algorithm:
sort the edges of G in increasing order by length
keep a subgraph S of G, initially empty
for each edge e in sorted order
    if the endpoints of e are disconnected in S
    add e to S
return S
```

Note that, whenever you add an edge (u,v), it's always the smallest connecting the part of S reachable from u with the rest of G, so by the lemma it must be part of the MST.

This algorithm is known as a greedy algorithm, because it chooses at each step the cheapest edge to add to S. You should be very careful when trying to use greedy algorithms to solve other problems, since it usually doesn't work. E.g. if you want to find a shortest path from a to b, it might be a bad idea to keep taking the shortest edges. The greedy idea only works in Kruskal's algorithm because of the key property we proved.

Analysis: The line testing whether two endpoints are disconnected looks like it should be slow (linear time per iteration, or O(mn) total). But actually there are some complicated data structures that let us perform each test in close to constant time; this is known as the union-find problem and is discussed in Baase section 8.5 (I won't get to it in this class, though). The slowest part turns out to be the sorting step, which takes O(m log n) time.

Option 8: Merging Sorted Lists

See textbook, Section 5.3

You are given an array a[] of numbers, where a[i] is the size of the i-th list to merge. You have to produce the sequence in which to merge the lists, and the total cost of merging all the lists. *Implementation Notes: Use a heap data structure. (You don't have to implement your own heal structure, you can simply use the inbuilt one in Java/C#.)*

analysis:

O (nlog (k) +k) time O (n+) space-where where n is the total number ot array elements and k is the number of arrays

Simple analysis of heap sort: if we can build a data structure from our list in time X and finding and removing the smallest object takes time Y then the total time will be O(X + nY). In our case X will be O(n) and Y will be O(log n) so total time will be O(n + n log n) = O(n log n)

We form a binary tree with certain properties:

- The elements of L are placed on the nodes of the tree; each node holds one element and each element is placed on one node.
- The tree is balanced which as far as I'm concerned means that all paths have length O(log n); Baase uses a stronger property in which no two paths to a leaf differ in length by more than one.
- (The heap property): If one node is a parent of another, the value at the parent is always smaller than the value at the child.

We can find the smallest heap element by looking at root of the tree (e.g. the boss of whole company has the biggest salary); this is easy to see, since any node in a tree has a smaller value than all its descendants (by transitivity). The number of comparison steps in this operation is then just the length of the longest path in the tree, O(log n).

This fits into the comparison sorting framework because the only information we use to determine who should be promoted is to compare pairs of objects..

The total number of comparisons in heapsort is then O(n log n) + how much time it takes to set up the heap.

proof:
A greedy strategy works: always merge the two shortest arrays into a new one.
Below I assume that merging two arrays with a and b elements into one requires exactly $a+b$ moved elements. For this model, I can actually formally prove that my answer is optimal.
Visualize the order in which you merge the arrays as a binary tree. The leaves are the original arrays, each inner vertex represents a merge, and the root corresponds to the result. Now, look at any original array. The times we move each of its elements during the entire process is equal to the depth of the leaf that corresponds to this array.
Hence, we have n different arrays and we are trying to construct a binary tree with n leaves such that sum (array length) * (assigned leaf depth) is minimized. This is the well-known optimal prefix code problem which has a provably optimal solution: the Huffman code.

Option 9: Hoffman Coding

Given a set of symbols and their frequency of usage, find a binary code for each symbol, such that:
a. Binary code for any symbol is not the prefix of the binary code of another symbol.
b. The weighted length of codes for all the symbols (weighted by the usage frequency) is minimized.

analysis:

This can be implemented using the following greedy algorithm that utilizes a minimum heap

Create a min-heap of all the symbols in the alphabet based on the frequency. While there are more than two symbols in the alphabet, extract the two symbols with least frequencies by invoking the extract minimum operation two times. Then Create a node with the sum of two frequencies and insert it back into the heap. Also insert this node in the encoding tree with the two symbols as its left and right child nodes. When only two symbols are left, create a root node for the encoding tree and the two symbols become the child nodes of the root node

This algorithm results in a binary tree in which the symbols end up as leaves, with the lowest frequency nodes farthest away from the root and highest frequency nodes in leaves closer to the top. From the root, assign 0 and I to the edges from each parent node. The code for a symbol will be the concatenation of the edges on its path. The depth for any particular character is the length of the path, which is also be the length for the code word for that character

We observe that there may be more than one Huffman code; no code is uniquely optimal. The set of lengths of an optimal code might not be unique. However, the Expected value or e weighted average SUM (frequency[I]*code lengthlil) will be the same and will be minimized