Sorting Algorithms

There are many sorting algorithms and they provide a good illustration that there are many ways to solve the same problem, some more efficient than others in certain circumstances. In our case we'll just be sorting arrays of integers, but they could be arrays of anything else. You can visualize many of these algorithms online at http://sorting.at. Let's start with some simple sorting algorithms that are $O(n^2)$

Bubble Sort

The idea behind bubble sort is to swap consecutive elements from left to right until the largest is at the right. We repeat this except we leave the last element "bubbled" to the right alone. Each pass through the array results in the largest item "bubbled" to the right.

(Demo in class)

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):  # count len-1 down to 1
        for i in range(passnum):        # count 0 to passnum-1
            if alist[i]>alist[i+1]:
                # swap alist[i], alist[i+1]
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp

# Test Run
list = [54, 34, 99, 102, 44, 91, 22]
bubbleSort(list)
print(list)
```

Selection Sort

In selection sort we sweep through the array and find the smallest item and put it into index 0. Then we sweep through the array starting at index 1 and find the smallest item and put it into index 1. Then we sweep through the array starting at index 2 and find the smallest item and put it into index 2. Repeat!

(Demo in class)

```
def selectionSort(alist):
   for fillslot in range(len(alist)-1):   # count 0 to len-2
       positionOfMin=fillslot
       for i in range(fillslot,len(alist)):  # count fillslot to len-1
           if alist[i]<alist[positionOfMin]:
               positionOfMin = i
       # swap alist[fillslot], alist[positionOfMin]
       temp = alist[fillslot]
       alist[fillslot] = alist[positionOfMin]
       alist[positionOfMin] = temp

# Test Run
list = [54, 34, 99, 102, 44, 91, 22]
selectionSort(list)
print(list)
```

Insertion Sort

We looked at insertion sort at the beginning of class, so we'll skip the details here.

The idea is to make sure the left side of the array is sorted as we move from left to right. First, start with one element. One element by itself is sorted, so there is nothing to do. Now, expand to two elements. We take the second element (called the key) then move it to the left if it's smaller than the first element, but leave it alone if it's bigger than the first element.

In general, we move the key to the left until we find a value that is smaller than it. If we ever move all the way to the left past the first element then the key should be inserted onto the front of the list and everything else needs to move over one slot.

(Demo in class)

More Efficient Sorting Algorithms

The algorithms we've discussed so far have nested loops that results in $O(n^2)$ runtime in the worst and average case. There are faster sorting algorithms! (In fact we have already seen one, with Heap Sort. What is the runtime?)

Merge Sort

A relatively simple, higher-performance sorting algorithm is Merge Sort. This is an example of a divide and conquer algorithm:

Merge Sort:

Divide n elements into two subsequences to be sorted of size n/2

Conquer – sort subsequences recursively with merge sort

Combine – merge sorted subsequences into big sorted answer

Need termination criteria for recursion –

Quit if sequence to sort is length 1

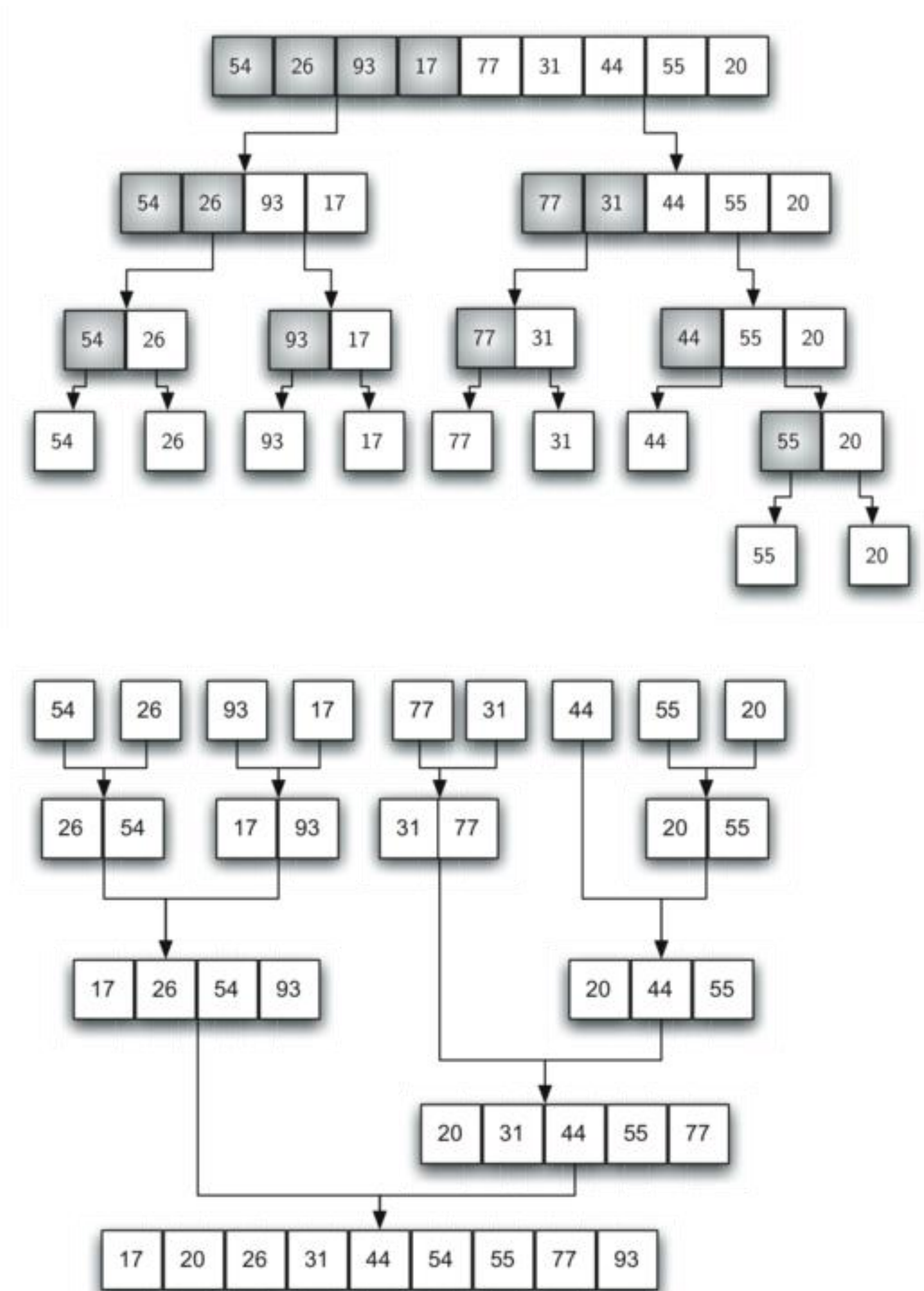
Pseudocode:

```
mergeSort(A,p,r)    #  p,r are start and end indices of the array A to sort
    If p<r then
        q ← ⌊(p+r)/2⌋
        mergeSort(A,p,q)
        mergeSort (A,q+1,r)
        Merge(A,p,q,r)  ; Merges A[p..q] with A[q+1..r] into A[p..r]
```

Call with mergeSort (A,0,length(A)-1)

What does merge do? It takes two sorted lists and merges them into one master sorted list.

Here is a visualization of the process:



Here we've visualized the sub-arrays as their own arrays, but in practice they will be copied back over the original array.

How do we merge? Here is an example of merging the array A with p=0, q=3, q+1=4, r=7:

Index	0	1	2	3	4	5	6	7
Value	3	5	22	33	7	8	12	21

Start left = p and right = q+1. Compare the entries at a[left] and a[right] and put the smallest into a new list and then increment the variable that was the smallest. In this case, we compare a[0] with a[4] and since 3 is less than 7, we put 3 into the new array and increment p:

left = 1
right = 4

Merged array:

3								

We now repeat the process, comparing a[1] with a[4] and since 5 is less than 7, we copy the 5 into the merged array and increment left:

left = 2
right = 4

Merged array:

3	5							

Next we compare a[2] with a[4] and since 7 is less than 22, we copy the 7 into the array and increment right:

left = 1
right = 5

Merged array:

3	5	7						

Repeating the process we eventually fill up the merged array. We can't do this merge in-place though (using the original array storage location only) so we need to make a copy of the merged array and then copy it back to the original array.

If n is the number of elements to merge, then it takes $\Theta(n)$ time to merge.

Here is some python code to merge an array given p, q, and r:

```python
def merge(alist, p, q, r):
    mergedList = []
    left = p
    right = q+1
    while (left <= q and right <= r):
        if alist[left] <= alist[right]:
            mergedList.append(alist[left])
            left += 1
        else:
            mergedList.append(alist[right])
            right += 1
    # Copy any remaining elements of left
    while left <= q:
        mergedList.append(alist[left])
        left += 1
    # Copy any remaining elements of right
    while right <= r:
        mergedList.append(alist[right])
        right += 1
    # Copy merged list back to alist at index p..r
    for i in range(len(mergedList)):
        alist[i+p] = mergedList[i]

# Test Run
l = [2, 4, 6, 1, 3, 5]
merge(l, 0, 2, 5)
print(l)
```

We can put this together into a complete MergeSort algorithm that is super short:

```python
def mergeSort(alist, p, r):
    if p < r:
        q = int((p+r)/2)
        mergeSort(alist, p, q)
        mergeSort(alist, q+1, r)
        merge(alist, p, q, r)

# Test Run
list = [54, 34, 99, 102, 44, 91, 22]
mergeSort(list,0,len(list)-1)
print(list)
```

What's the runtime of the algorithm? It's the cost to split + the cost to merge.

Let's define T(n) to be the runtime for a problem of size(n). This is called a recurrence relation, since it is recursively defined.

Recurrence:

$$T(n) = \begin{cases} \Theta(1), n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), n > 1 \end{cases}$$

The 2T(n/2) comes from the divide, and the Θ(n) comes from the merge.

We'll discuss a bit later how to solve these recurrence relations. You will look at it in more detail in CSCE A351.

It turns out that Merge Sort is Θ (nlgn). This can be seen from the tree: We perform lg(n) splits and merges and it takes O(n) time to perform the merge.

The Selection Problem and QuickSort

Consider the problem of finding the ith smallest element in a set of n unsorted elements. This is referred to as the selection problem or the ith "order statistic".

If i=1 this is finding the minimum of a set

i=n this is finding the maximum of a set

i=n/2 this is finding the median or halfway point of a set

All of these scenarios are common problems.

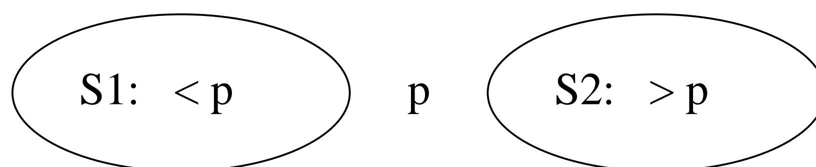
The selection problem is defined as:

Input: A set of n numbers and a number i, with 1 ≤ i ≤ n

Output: The element x in A that is larger than exactly i-1 other elements in A.

Can do in $\Theta(n \lg n)$ time easily by sorting with Merge Sort, and then pick A[i]. But we can do better!

Consider if the set of n numbers is divided as follows:



(In general we could have ≤ p or ≥ p but for simplicity let's just leave it as-is). Note that the elements in S1 are not sorted, but all of them are smaller than element p (partition). We know that p is the (|S1| +1)th smallest element of n. This is the same idea used in quicksort.

Now consider the following algorithm to find the ith smallest element from Array A:

- Select a pivot point, p, out of array A.
- Split A into S_1 and S_2, where all elements in S_1 are < p and all elements in S_2 are > p
- If $i = |S_1|+1$ then p is the ith smallest element.
- Else if $i \leq |S_1|$ then the ith smallest element is somewhere in S_1. Repeat the process recursively on S_1 looking for the ith smallest element.
- Else i is somewhere in S_2. Repeat the process recursively looking for the $i-|S_1|-1$ smallest element.

Question: How do we select p? Best if p is close to the median. If p is the largest element or the smallest, the problem size is only reduced by 1. Here are a couple strategies for picking p:

- Always pick the same element, from index n or 1
- Pick a random element
- Pick 3 random elements, and pick the median
- Other method we will see later

Once we have p it is fairly easy to partition the elements:

If A contains: [5 12 8 6 2 1 4 3]

Can create two subarrays, S_1 and S_2. For each element x in A, if x < p put it in S_1

if x ≥ p put it in S_2.

p=5

S_1: [2 1 4 3] S_2: [5 12 8 6]

This certainly works, but requires additional space to hold the subarrays. We can also do the partitioning in-place, using no additional space if we maintain pointers starting from the beginning and end of the array as illustrated below:

```
def partition(alist, p, r):
    x = alist[p]   # Choose first element as the partition/pivot
    i = p-1
    j = r+1
    while True:
        j = j-1
        while alist[j] > x:
            j = j-1
        i = i + 1
        while alist[i] < x:
            i = i + 1
        if i < j:
            # swap alist[i], alist[j]
            temp = alist[i]
            alist[i] = alist[j]
            alist[j] = temp
        else:
            return j     # Indicates index of the partition

# Test Run
l = [38, 1, 20, 2, 30, 4, 50, 5, 60]
p = partition(l, 0, len(l)-1)
# Everything from 0 to p is <= partition eleme t
# Everything from p+1 to the end is >= partition element
print(p,l)
```

Example:

A[p..r] = [5 12 8 6 2 1 4 3]

x=5

	5	12	2	6	2	1	4	3		
i									j	
	5	12	2	6	2	1	4	3		
i								j		
	5	12	2	6	2	1	4	3		
	i							j		
	3	12	2	6	2	1	4	5		swap
	i							j		
	3	12	2	6	2	1	4	5		
	i						j			
	3	12	2	6	2	1	4	5		
		i					j			
	3	4	2	6	2	1	12	5		swap
		i					j			
	3	4	2	6	2	1	12	5		
		i				j				

3	4	2	6	2	1	12	5	
		i			j			
3	4	2	6	2	1	12	5	
			i		j			
3	4	2	1	2	6	12	5	swap
			i		j			
3	4	2	1	2	6	12	5	
			i	j				
3	4	2	1	2	6	12	5	
				ij				
3	4	2	1	2	6	12	5	crossover, i>j
				j	i			

Return j. All elements in A[p..j] smaller or equal to x, all elements in A[j+1..r] bigger or equal to x. (Note this is a little different than the initial example, where we split the sets up into < p, p, and > p. In this case the sets are <=p or >=p. (Consider the case if all array elements are identical). If the pivot point selected happens to be the largest or smallest value, it will also be guaranteed to split off at least one value). This routine makes only one pass through the array A, so it takes time $\Theta(n)$. No extra space is required except to hold index variables.

To use this version of Partition in the Selection algorithm, we need to modify the selection algorithm a bit since we are not splitting into < p, p, and > p. Here is the modified algorithm:

; Select from alist, with lower index of p and upper index of r, the ith largest number

```
def selection(alist, p, r, i):
    if p==r:
        return alist[p]
    q = partition(alist, p, r)  # q = index of partition
    k = q - p + 1               # k is size of left partition
    if i <= k:
        return selection(alist, p, q, i)    # recurse on left partition
    else:
        return selection(alist, q+1, r, i-k)   # recurse on right partition


# Test Run
l = [50,40,10,20,100,30,60,80,70,90]
print(selection(l,0,len(l)-1,5))
```

Note that with just a few minor changes we get Quicksort!

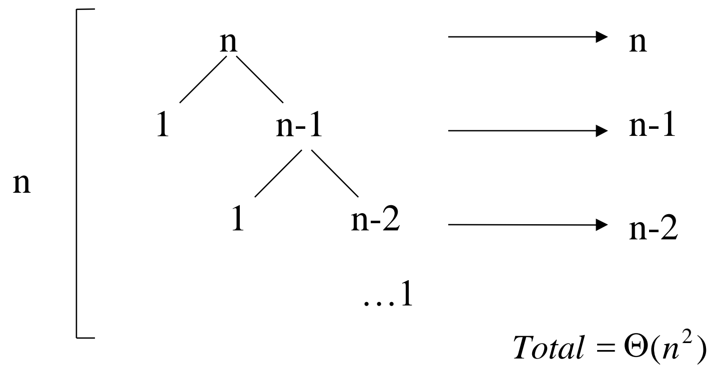
```
def quickSort(alist, p, r):
    if p>=r:
        return
    q = partition(alist, p, r)
    quickSort(alist, p, q)
    quickSort(alist, q+1, r)

# Test Run
l = [50,40,10,20,100,30,60,80,70,90]
quickSort(l,0,len(l)-1)
print(l)
```

Going back to the selection problem, the worst case running time is when we pick min or max as the partition element, producing region of size n-1.

$$T(n) = T(n-1) + \Theta(n)$$

subprob   time to split

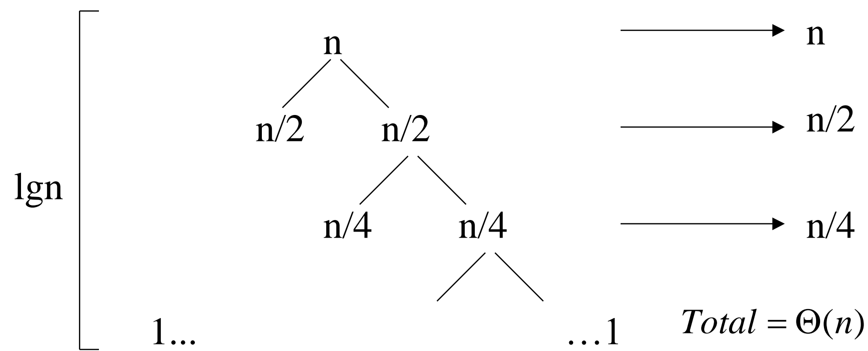
Recursion tree for worst case:



Best-case Partitioning:

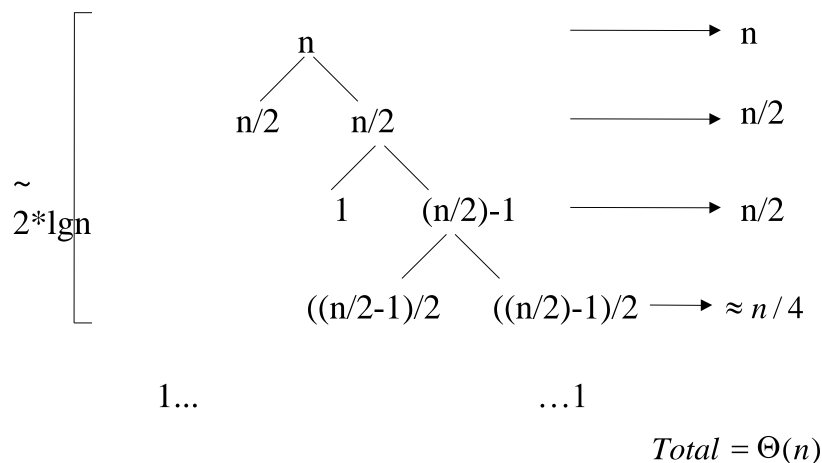
In the best case, we pick the median each time.

Recursion Tree for Best Case:



Average Case: Can think of the average case as alternating between good splits where n is split in half, and bad splits, where a min or max is selected as the split point.

Recursion tree for bad/good split, good split:



Both are $\Theta(n)$, with just a larger constant in the event of the bad/good split.

So average case still runs in time $\Theta(n)$.

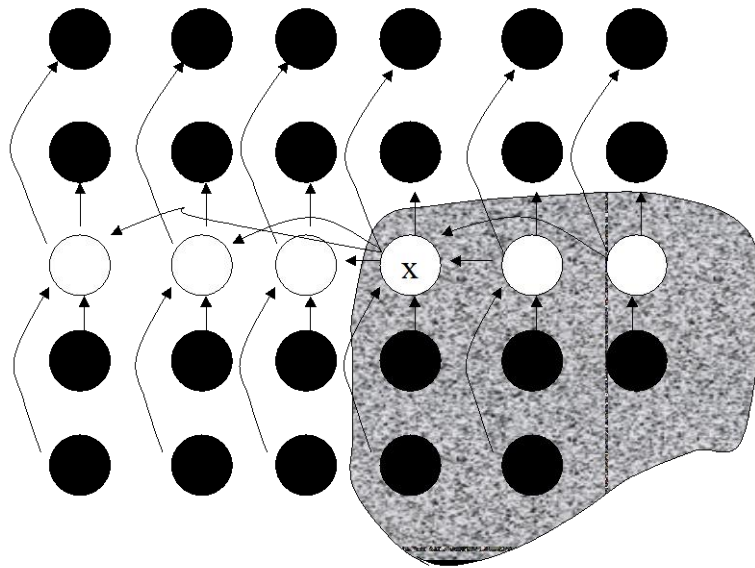
We can solve this problem in worst-case linear time, but it is trickier. In practice, the overhead of this method makes it not useful in practice, compared to the previous method. However, it has interesting theoretical implications.

Basic idea: Find a partition element guaranteed to make a good split. We must find this partition element quickly to ensure $\Theta(n)$ time. The idea is to find the median of a sample of medians, and use that as the partition element.

New partition selection algorithm:

- Arrange the n elements into n/5 groups of 5 elements each, ignoring the at most four extra elements. (Constant time to compute bucket, linear time to put into bucket)
- Find the median of each group. This gives a list M of n/5 medians. (time $\Theta(n)$ if we use the same median selection algorithm as this one or hard-code it)
- Find the median of M. Return this as the partition element. (Call partition selection recursively using M as the input set)

See picture of median of medians:



Guarantees that at least 30% of n will be larger than pivot point p, and can be eliminated each time!

Runtime: $T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$

select recurse overhead of split/select

pivot subprob

The O(n) time will dominate the computation resulting in O(n) run time.

Returning to quicksort, what is the the runtime?