Decrease and Conquer Algorithms - Enumeration and Selection

The decrease and conquer technique is similar to divide and conquer, except instead of partitioning a problem into multiple subproblems of smaller size, we use some technique to reduce our problem into a single problem that is smaller than the original.

The smaller problem might be:

- Decreased by some constant
- Decreased by some constant factor
- Variable decrease

We saw that Merge Sort was an example of divide and conquer (divide a list into two separate lists to sort recursively). Binary search is an example of decrease and conquer (divide a list into half the size and search only that one list for the target).

For combinatorial problems we might need to generate all permutations or subsets of a set. For example, to solve the subset sum problem we might generate all possible subsets and see if the elements sum to the target. We can attack these types fo problems using a decrease and conquer algorithm.

Generating Permutations

If we have a set of n elements: $\{ a_1, a_2, a_3, \ldots a_n \}$ then how can we generate all n! permutations? The solution is to generate all (n-1)! permutations. Once we solve this, then we can get a solution to the larger one by inserting n into each of the positions among every permutation of (n-1) elements.

The recursive base case is that when we have a single element, there is only one permutation.

For example, to get the permutations of {1,2,3}:

Find solution to permutations of {1, 2}

Find solution to permutations of {1}

This is just {1}

Insert 2 into {1}, giving: {2 1} and {1 2}

Insert 3 into {2 1} and {1 2} giving: {3 2 1} {2 3 1} {2 1 3} {3 1 2} {1 3 2} {1 2 3}

We end up creating n! permutations, leading to a runtime of $\Theta(n!)$. This is horribly slow for all but very small values of n, but that is not the fault of the algorithm but the problem is simply to generate a huge number of items.

An example problem where this could be used would be to find all routes for the traveling salesman problem.

Generating Subsets

You might recall the knapsack problem or subset sum problem which may require us to enumerate through all subsets until we find one that solves the problem. We can generate subsets in a manner similar to generating all permutations.

To generate all subsets requires generating the power set of the items.
The subsets of A = { a_1, a_2, a_3, … a_n } can be divided into two groups: those that do not contain a_n and those that do.

The first group is simply all subsets of: S_1 = { a_1, a_2, a_3, … a_{n-1} }.
The second group is simply a_n added to each subset of S_1 unioned with S_1.

For example, to get the power set of {1, 2, 3}:

Find all subsets of {1, 2}
Find all subsets of {1}
Find all subsets of {}
This is just ∅
Insert 1 into ∅ and union with ∅ to get: {1} and ∅
Insert 2 into: { {1}, ∅ } and union with { {1}, ∅ } to get:
{ {1, 2}, {2}, {1}, ∅ }
Insert 3 into: { {1, 2}, {2}, {1}, ∅ } and union with { {1, 2}, {2}, {1}, ∅ } to get:
{ {1,2,3}, {2,3}, {1,3}, {3}, {1, 2}, {2}, {1}, ∅ }

The Selection Problem - Variable Size Decrease/Conquer

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

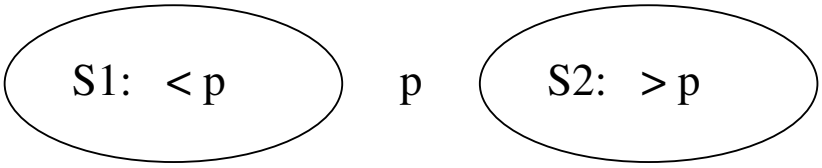
If i=1 this is finding the minimum of a set
i=n this is finding the maximum of a set
i=n/2 this is finding the median or halfway point of a set -- common problem

Selection problem defined as:

Input: A set of n numbers and a number i, with $1 \leq i \leq n$
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:



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 i^{th} smallest element from Array A:

- Select a pivot point, p, out of array A.
- Split A into S1 and S2, where all elements in S1 are <p and all elements in S2 are >p
- If I = |S1|+1 then p is the i^{th} smallest element.
- Else if $i \leq |S1|$ then the i^{th} smallest element is somewhere in S1. Repeat the process recursively on S1 looking for the ith smallest element.
- Else i is somewhere in S2. Repeat the process recursively looking for the i-|S1|-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.

- Always pick the same element, 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, S1 and S2. For each element x in A, if $x < p$ put it in S1, if $x \geq p$ put it in S2.

p=5
S1: [2 1 4 3]
S2: [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:

```
Partition(A,p,r)                ; Partitions array A[p..r]
  x ← A[p]                      ; Choose first element as partition element
  i ← p-1
  j ← r+1
  while true
        do repeat
                j ← j-1
           until
                A[j] ≤ x
           repeat
                i ← i+1
           until A[i] ≥ x
        if i<j
           then exchange A[i] ↔ A[j]
           else return j                    ; indicates index of partitions
```

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 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 array A, with lower index of p and upper index of r, the i^{th} largest number
Select(A,p,r,i)

```
    If p = r return A[p]
    q ← Partition(A,p,r)          // Q gets the index of where we made the partition
    K ← q – p + 1                 //  Size of left partition
    If i<=K return(Select(A,p,q,i)
    else return(Select(A,q+1,r,i-K)
```

Note the similarity to Quicksoft:

```
QuickSort(A,p,r)
    If p >= r return
    q← Partition(A, p, r)
    QuickSort(A, p, q);
    QuickSort(A, q+1, r);
```

Worst case running time of selection: Pick min or max as partition element, producing region of size n-1.

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

subprob time to split

Evaluate recurrence by iterative substitution method:

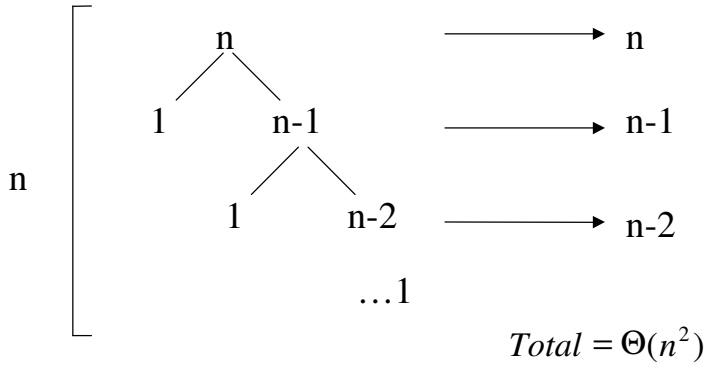
$$T(1) = \Theta(1), T(2) = \Theta(1) + \Theta(2), T(3) = \Theta(1) + \Theta(2) + \Theta(3), \ldots$$

$$\sum_{i=1}^{n} \Theta(i)$$

$$\Theta \sum_{i=1}^{n} i$$

$$= \Theta(n^2)$$

Recursion tree for worst case:



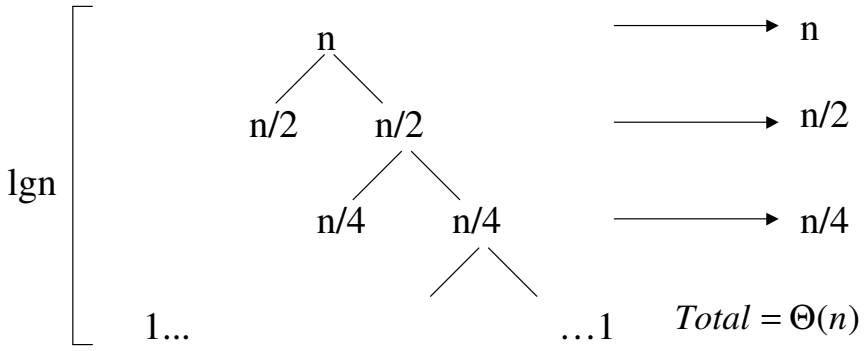
Best-case Partitioning:

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

$$T(n) = T(\frac{n}{2}) + \Theta(n)$$

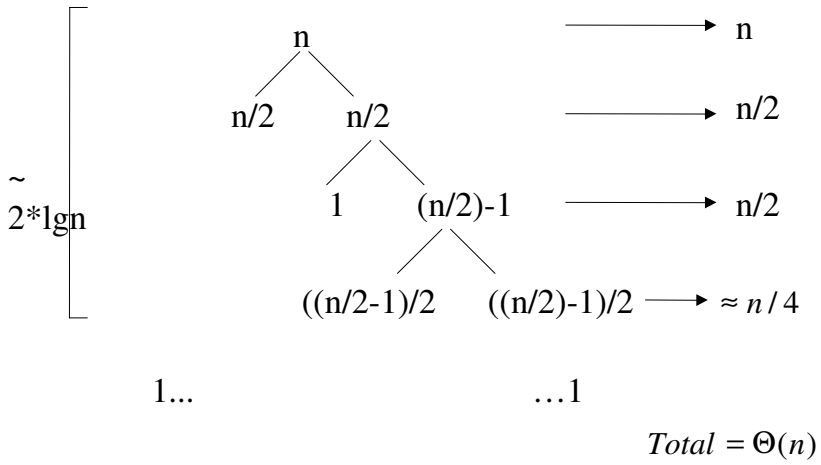
Using the master method: a=1, b=2, f(n)= $\Theta(n)$
Case 3: $1 < 2^1$ so the solution is f(n)= $\Theta(n)$

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:



$Total = \Theta(n)$

Both are $\Theta(n)$, with just a larger constant in the event of the bad/good split.
So average case still runs in time $\Theta(n)$.

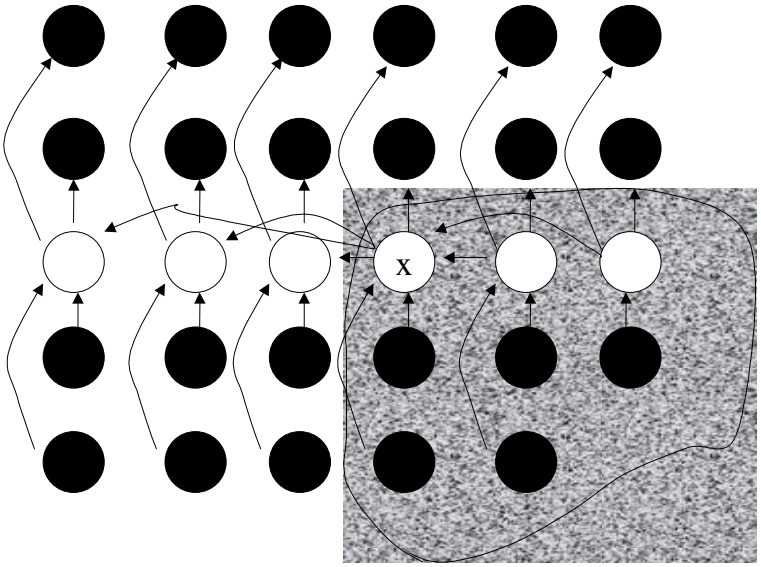
We can solve this problem in worst-case linear time, but it is trickier. In practice, the overhead of this method makes it not useful in practice, compared to the previous method. However, it has interesting theoretical implications.

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 pivot — recurse subprob — overhead of split/select

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