Lecture Notes, CS351

Chapter 4: Recurrence Relations : Iterative and The Master Method

Iteration Method: Expand the terms into a summation, and solve algebraically

Example:

T(n)= Theta(1) for n=1
T(n) = 3T(n/4) + n for n>1

$$T\left(\frac{n}{4}\right) = 3T\left(\frac{n}{4*4}\right) + \frac{n}{4}$$

We can plug this back into the original recurrence relation:

$$T(n) = 3\left(3T\left(\frac{n}{16}\right) + \frac{n}{4}\right) + n$$

We can keep on going:

$$T(n) = 3\left(3\left(3T\left(\frac{n}{64}\right) + \frac{n}{16}\right) + \frac{n}{4}\right) + n$$

If we stop at this point and do some math:

T(n) = 27T(n/64) + 9(n/16) + 3(n/4) + n

$$T(n) = n + \frac{3n}{4} + \frac{9n}{16} + 27T\left(\frac{n}{64}\right)$$

There's a pattern here! If we consider i as the index, where i=1 gives us n+(3/4)n, then we can generalize this as i increases:

$$T(n) = n + \frac{3n}{4} + \frac{9n}{16} + ... + \frac{3^j n}{4^j} + ... + 3^i T\left(\frac{n}{4^i}\right)$$

How far does i go? Does it increase to infinity? NO at some point we will have to stop. But we already know when we stop – we stop at T(1) because at this point there is no more recursion, we just return a constant number for the amount of work to do.

If we stop at T(1), this means we will stop when $1=(n/4^i)$.

$$1 = \frac{n}{4^i} \qquad n = 4^i \qquad \log_4 n = i$$

So we can now express the recurrence relation as:

$$T(n) = n + \frac{3n}{4} + \frac{9n}{16} + ... + \left(\frac{3}{4}\right)^i n + ...3^{\log_4 n}\Theta(1)$$

substituting $\Theta(1)$ for $T(n/4^i)$ since we will only do a constant amount of work on the last iteration.

We can summarize this as a single summation. First recall that

$3^{\log_4 n} = n^{\log_4 3}$; this is sublinear since $\log_4 3 < 1$

$$T(n) = \left(\sum_{i=0}^{\log_4 n - 1} n\left(\frac{3}{4}\right)^i\right) + \Theta(n^{\log_4 3})$$

$$T(n) \le \left(n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i\right) + \Theta(n^{\log_4 3})$$; up to infinity bigger, so <= applies

recall that $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$; for x<1

$$T(n) \le n\frac{1}{1-\frac{3}{4}} + \Theta(n^{\log_4 3})$$

$T(n) \le 4n + \Theta(n^{\log_4 3})$; $T(n) \le 4n + o(n)$; loose upper bound so use little-o

This means that the recurrence is O(n).

This method is accurate but can result in a lot of algebra to keep track of; can also get very challenging for more complicated recurrence relations.

Second Example: T(n)=1 if n=1
T(n)=4T(n/2)+n if n>1

T(n) =4T(n/2) + n
=4(4T(n/4)+n/2)+n
=4(4(4T(n/8)+n/4)+n/2)+n
=64T(n/8) + 4n +2n +n
=n + 2n +4n + 64T(n/8)
=n + 2n + 4n + … +2^in + … 4^iT($n/2^i$) ; hard part to figure out

What is the last term? When $(n/2^i)=1$ → i=lgn

$T(n) = n + 2n + 4n + 8n + \ldots 2^i n + \ldots 4^{\lg n}\ \Theta(1)$

$$= \left(n \sum_{i=0}^{\lg n - 1} 2^i \right) + 4^{\lg n}\Theta(1)$$

We know that $\sum_{k=0}^{m} x^k = \frac{x^{m+1}-1}{x-1}$

Let's let m=lgn-1. Then:

$$T(n) = n\left(\frac{2^{\lg n - 1 + 1} - 1}{2-1}\right) + 4^{\lg n}\Theta(1)$$

$$= n2^{\lg n} - n + 4^{\lg n}\Theta(1)$$

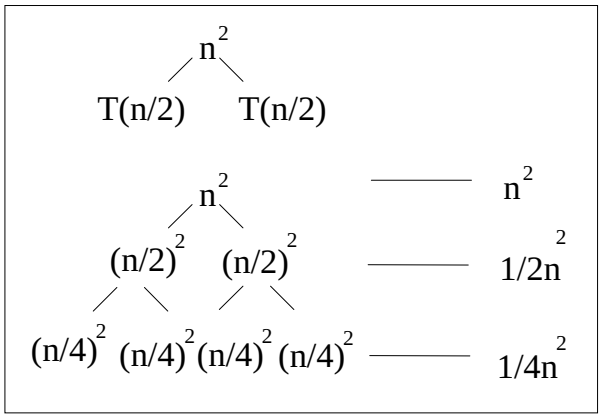
$$= n^2 - n + n^{\lg 4}\Theta(1)$$

$$= n^2(\Theta(1) + 1) - n \qquad = \qquad \Theta(n^2)$$

Sometimes a recursion tree can help:

Recursion Tree: Help to keep track of the iterations

Given $T(n) = 2T(n/2) + n^2$



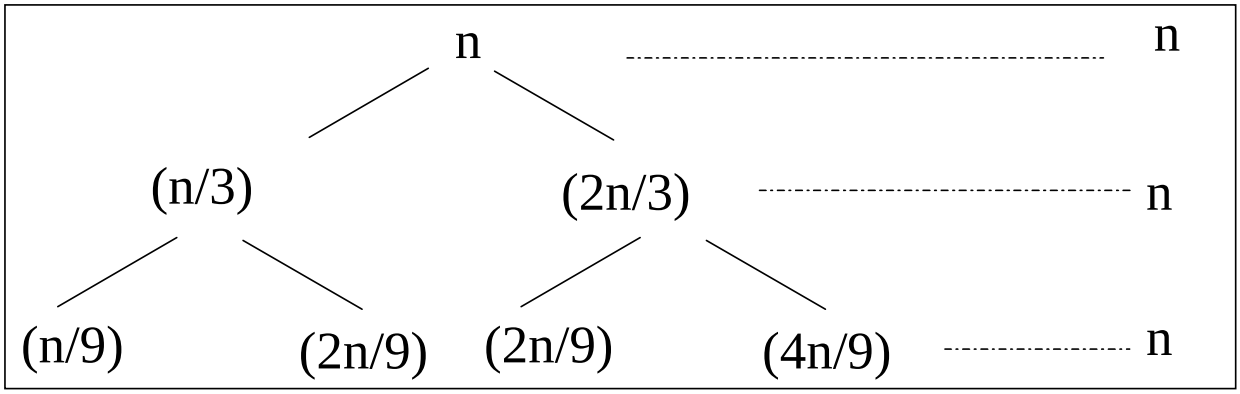
How deep does the tree go?
We stop at the leaf, and we know we're at a leaf when we have a problem of size 1.

$1 = (n/2^i)^2$
so $n^2 = 2^{2i}$; $n = 2^i$; $i = \lg n$

The amount of work done is then:

$\sum_{i=0}^{\lg n}\left(\frac{n}{2^i}\right)^2 = \Theta(n^2)$; this is geometrically decreasing in size, so it won't get any bigger than n^2.

One more example: T(n) = T(n/3) + T(2n/3) + n



Each level does work of size n; if we just know the height of the tree, i, the total work is ni.

The tree stops when the leaf is of size 1. The hard part is to figure out the formula based on the height:

$n\left(\frac{2}{3}\right)^i = 1$ (why pick the 2/3 branch and not 1/3?)

$n = \frac{1}{\left(\frac{2}{3}\right)^i} = \left(\frac{3}{2}\right)^i$

$i = \log_{3/2} n$

So the total work is $(\log_{3/2} n)n$ or O(nlog $_{3/2}$ n).

Master Method:

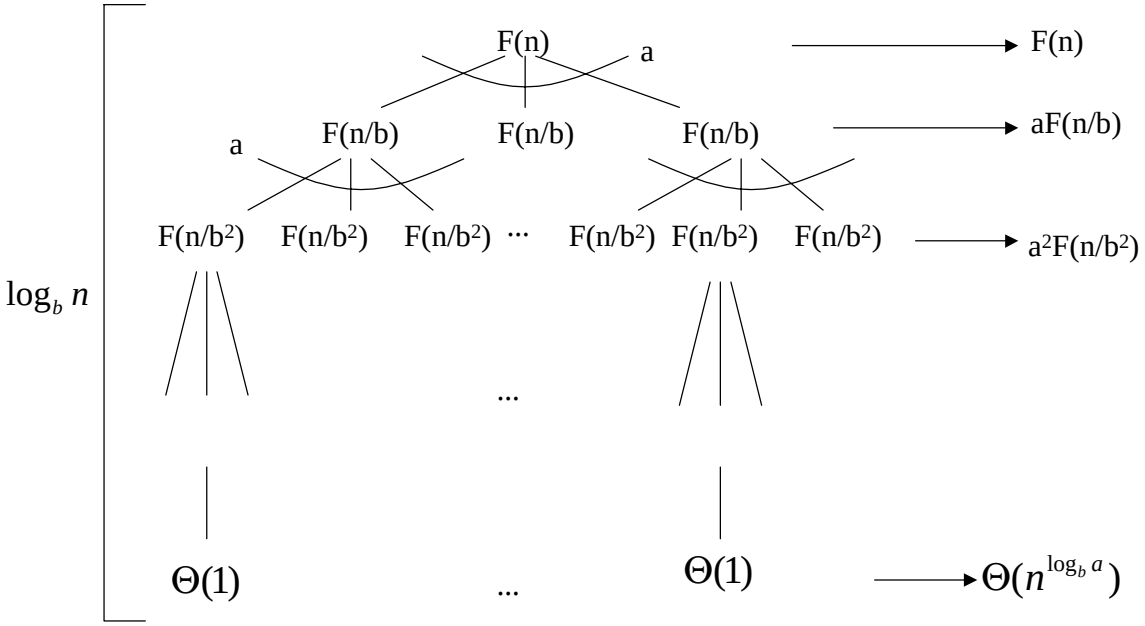
If the form of a recurrence is: $T(n) = aT\left(\frac{n}{b}\right) + f(n), a \geq 1, b > 1$

then we can use the Master Method, which is a cookbook-style method for proving the runtime of recurrence relations that fit its parameters. Note that not all recurrence of the above form can be solved through the master method. We won't prove the master method, but will give an argument as to how it works.

In the master method:

- *a* is the number of subproblems that are solved recursively; i.e. the number of recursive calls.
- *b* is the size of each subproblem relative to n; n/b is the size of the input to the recursive call.
- *f(n)* is the cost of dividing and recombining the subproblems.

Recursion tree example: T(n)=aT(n/b)+f(n)



$$Total = \Theta\left(n^{\log_b a}\right) + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right)$$

What is the height of the tree? When $f\left(\frac{n}{b^i}\right) = f(1) \rightarrow \frac{n}{b^i} = 1 \rightarrow n = b^i \rightarrow i = \log_b n$

How many leaves are there?

$a^{height} = NumberLeaves$

$a^{\log_b n} = n^{\log_b a}$

Work at the leaves is : $\Theta(1) n^{\log_b a} = \Theta\left(n^{\log_b a}\right)$

Work of dividing and combining is: $f(n) + af(\frac{n}{b}) + a^2 f(\frac{n}{b^2}) + \ldots$

$$= \sum_{i=0}^{\log_b n-1} a^i f(\frac{n}{b^i})$$

this does not include the cost of the leaves.

The total work/runtime T(n) is: $\Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b n-1} a^i f(\frac{n}{b^i})$

The time T(n) might be dominated by:
1. The cost of the leaves
2. The cost of the divide/combine or the root
3. Evenly distributed at all the levels

The master method tells us what the asymptotic running time will be depending on which cost is the highest (dominates).

If the form is:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n), a \geq 1, b > 1$$

Then based on comparing *f(n)* and $n^{\log_b a}$ we know the running time given the following three cases:

- If $f(n) = O(n^{\log_b a-\varepsilon})$ for some constant $\varepsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$; cost of leaves dominates.
- If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \lg n)$; cost is evenly distributed
- If $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$ and if $af(\frac{n}{b}) \leq cf(n)$ for some constant c<1 and all sufficiently large n, then $T(n) = \Theta(f(n))$; divide/conquer or root cost dominates

Example:

$$T(n) = 9T(\frac{n}{3}) + n$$

So a=9, b=3, f(n)=n

Case 1 works for $f(n) = O(n^{\log_b a-\varepsilon})$. We need to prove this relationship by showing that:

$$f(n) = O(n^{\log_b a-\varepsilon})$$

$$n = O(n^{\log_3 9-\varepsilon}) = O(n^{2-\varepsilon})$$

if $\varepsilon = 1$ then n=O(n) and case 1 is satisfied.

Therefore:

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_3 9}) = \Theta(n^2)$$

In this example, the cost of the leaves has dominated the runtime.

Example:

$T(n) = 2T(\frac{n}{2}) + n$; Merge Sort

So a=2, b=2, f(n)=n
Check case 1:

Is $f(n) = O(n^{\log_b a - \varepsilon})$?

$n = O(n^{\log_2 2 - \varepsilon})$

$n = O(n^{1-\varepsilon})$

For any epsilon>0, n is bigger, so case 1 does not work.

Check case 2:

Is $f(n) = \Theta(n^{\log_b a})$

$n = \Theta(n^{\log_2 2}) = \Theta(n)$ YES

therefore:

$T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n^{\log_2 2} \lg n) = \Theta(n \lg n)$

Cost is evenly distributed among leaves and upper part of tree.

Example:

$T(n) = T(\frac{2n}{3}) + 1$

So a=1, b=3/2, f(n)=1

Case 1 does not work (exercise for the reader)
Case 2:

Is $f(n) = \Theta(n^{\log_b a})$?

$1 = \Theta(n^{\log_{3/2} 1}) = \Theta(n^0) = \Theta(1)$ YES

therefore:

$T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n^{\log_{3/2} 1} \lg n) = \Theta(n^0 \lg n) = \Theta(\lg n)$

Cost is again evenly distributed.

Example:

$T(n) = 3T(\frac{n}{4}) + n \lg n$

a=3,b=4,f(n)=nlgn

Case 1 and 2 don't fit (exercise for the reader)
Case 3:

Is $f(n) = \Omega(n^{\log_b a+\varepsilon})$?

$n \lg n = \Omega(n^{\log_4 3+\varepsilon}) = \Omega(n^{0.79+\varepsilon})$

YES, if epsilon =0.21, then $n \lg n = \Omega(n)$

We also need to show the extra condition:

Is $af(\frac{n}{b}) \le cf(n)$ for c<1?

$3f(\frac{n}{4}) \le cf(n)$

$3\frac{n}{4}\lg\left(\frac{n}{4}\right) \le cn \lg n$

$3\frac{n}{4}(\lg n - \lg 4) \le cn \lg n$

$3\frac{n}{4}(\lg n - 2) \le cn \lg n$

YES, if c=¾ then $3\frac{n}{4}(\lg n - 2) \le \frac{3}{4} n \lg n$

therefore:

$T(n) = \Theta(f(n)) = \Theta(n \lg n)$

Example:

$T(n) = 4T(\frac{n}{2}) + \frac{n^2}{\lg n}$

So a=4, b=2, f(n)=$\frac{n^2}{\lg n}$

Try case 1:

Is $f(n) = O(n^{\log_b a-\varepsilon})$?

$\frac{n^2}{\lg n} = O(n^{\log_2 4-\varepsilon})$

$\frac{n^2}{\lg n} = O(n^{2-\varepsilon})$

NO, for epsilon>0, f(n) is larger.

Try case 2:

Is $f(n) = \Theta(n^{\log_b a})$?

$$\frac{n^2}{\lg n} = \Theta(n^{\log_2 4}) = \Theta(n^2)$$

NO, grows smaller than n^2.

Try case 3:

Is $f(n) = \Omega(n^{\log_b a+\varepsilon})$?

$$\frac{n^2}{\lg n} = \Omega(n^{2+\varepsilon})$$

NO, for epsilon > 0, f(n) is smaller, not bigger.

Master method does not work for this recurrence relation!
(Solution is $\Theta(n^2 \lg\lg n)$ by substitution)

Selection Problem

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 -- common problem

Selection problem 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.

How many comparisons are necessary to determine the selection?
Say we want to find the minimum:

Lower bound of at least n-1 comparisons to see every other element
Think as a tournament:

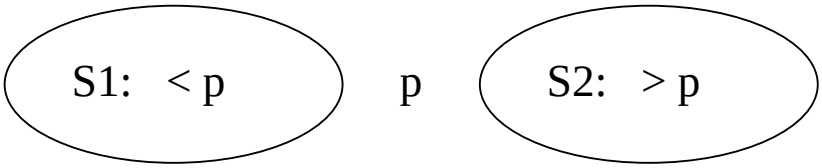
Pick contender
Contender competes with another (comparison)
Winner is the smallest element

Every element except the winner must lose one match.
This is a simple example to show that we need at least n-1 comparisons, we will use this technique later in more complex examples to show a lower bound.

Selecting the ith smallest element:

Can do in $\Theta(n \lg n)$ time easily by sorting with Merge Sort,
and then pick A[i]. But 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. We will use this idea later to also sort numbers (known as 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 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 ith smallest element.
- Else if i<=|S1| then the ith 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

How do we partition once we have p?

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>=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:

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

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 which is listed in the book:

```
Select(A,p,r,i)
    If p=r return A[p]
    Q ← Partition(A,p,r)
    K ← Q –p + 1
    If i<=K return(Select(A,p,Q,i)
    else return(Select(A,q+1,r,i-K)
```

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

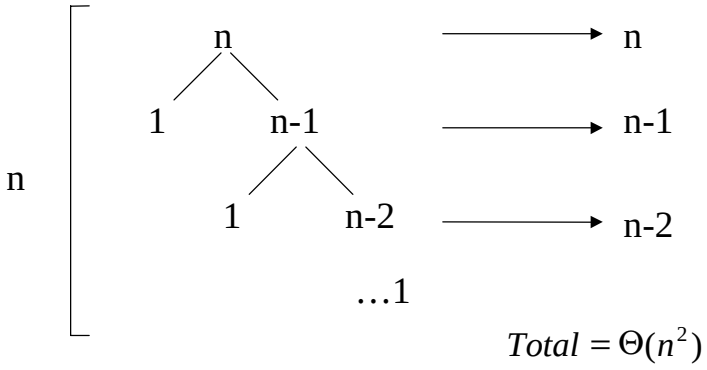
$$T(n) = T(n-1) + \Theta(n)$$

subprob time to split

Evaluate recurrence by iterative 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: Is $f(n) = \Omega(n^{\log_b a + \varepsilon})$?

$$\Theta(n) = \Omega(n^{\log_2 1 + \varepsilon})$$
$$\Theta(n) = \Omega(n^{0+\varepsilon})$$

YES if epsilon between 0 and 1, say 0.5

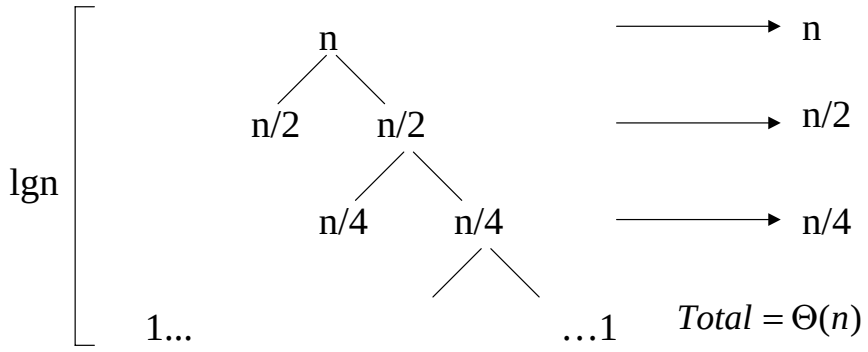
Also is $af(\frac{n}{b}) \le cf(n)$ for c<1?

$$\Theta(\frac{n}{2}) \le c\Theta(n)$$

YES if c > ½

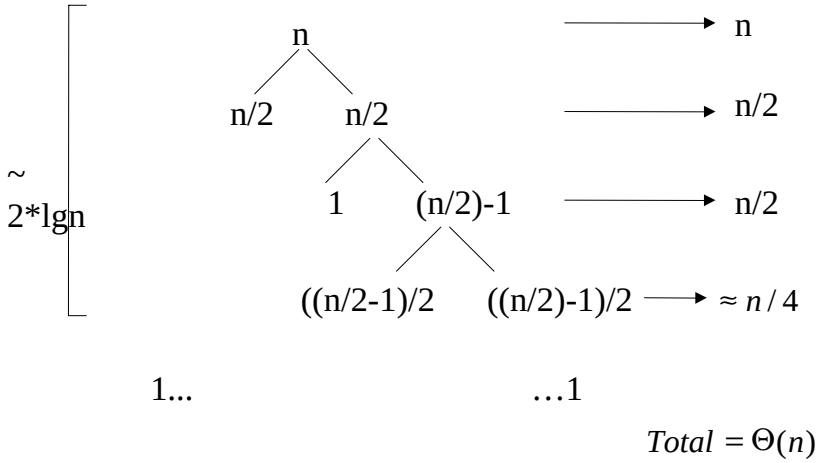
So $T(n) = \Theta(f(n)) = \Theta(n)$

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:



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

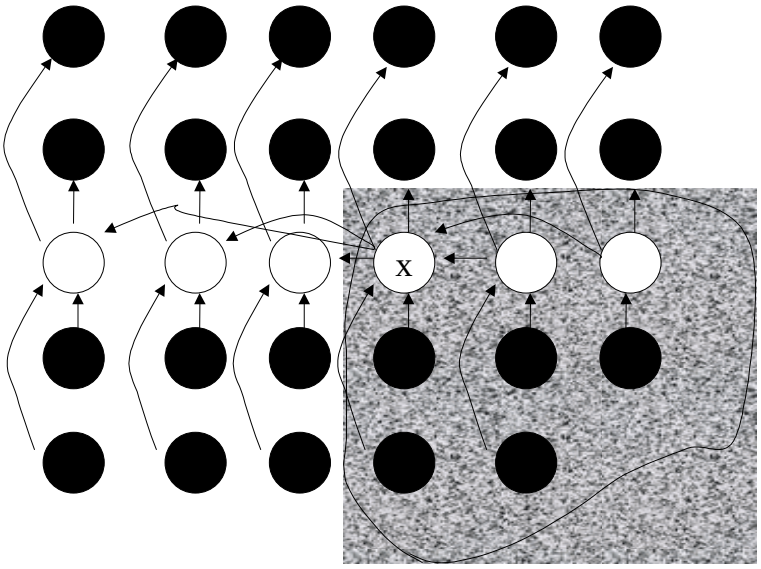
We can solve this problem in worst-case linear time, but it is trickier. In practice, the overhead of this method makes it not useful in practice, compared to the previous method. However, it has interesting theoretical implications.

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

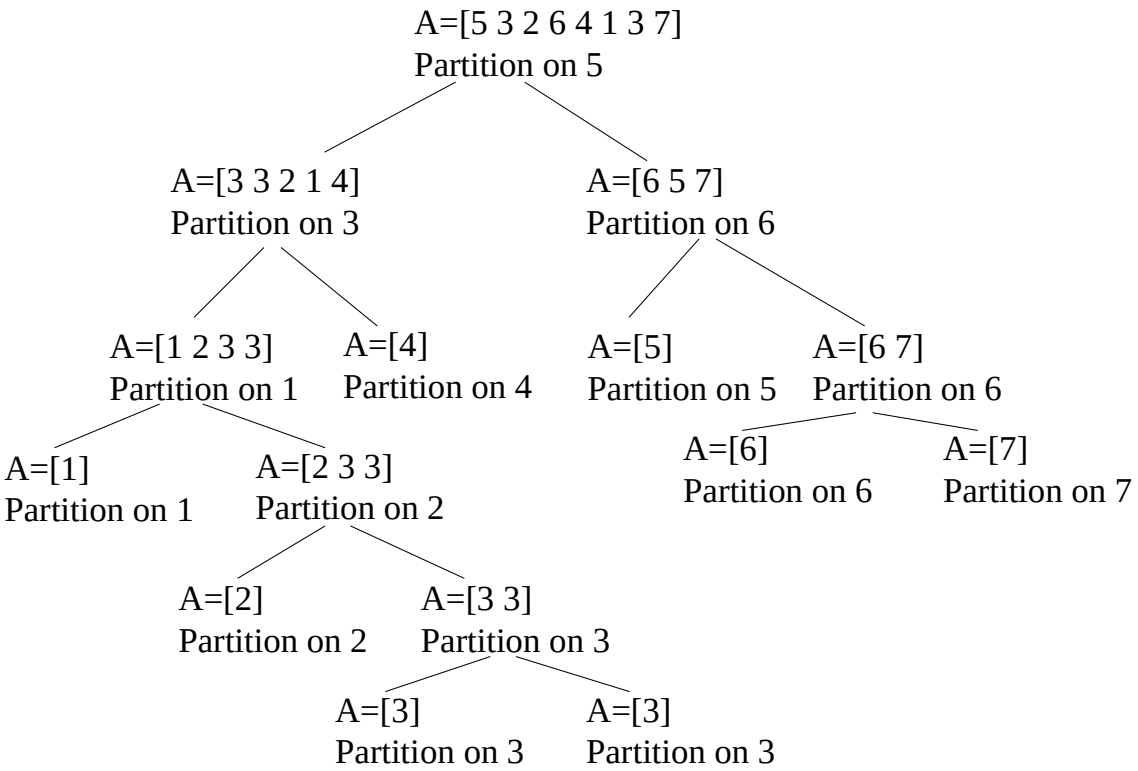
The O(n) time will dominate the computation by far resulting in O(n) run time.

Quicksort

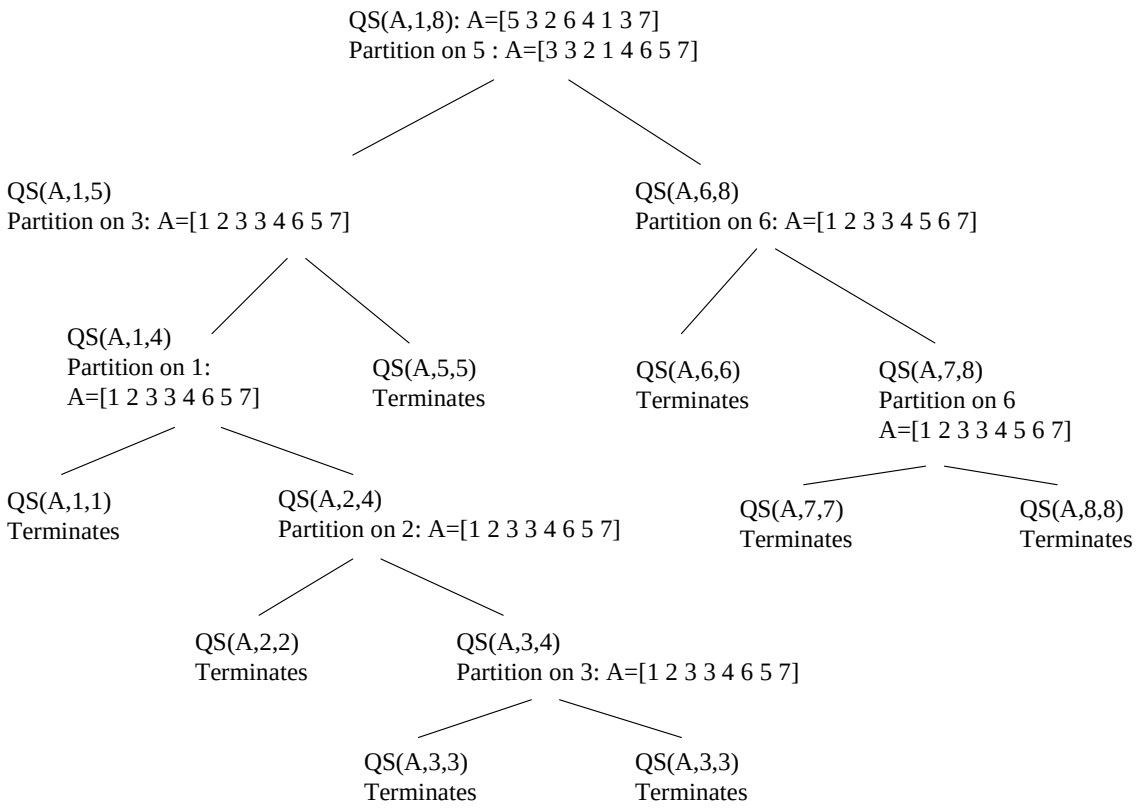
We can also use the Partition selection algorithm to do sorting, this is called Quicksort.

```
QuickSort(A,p,r)                    ; Sort A[p..r]
        if p<r
        then
                q← Partition(A,p,r)
                QuickSort(A,p,q)
                QuickSort(A,q+1,r)
```

Show tree for sorting example of A=[5 3 2 6 4 1 3 7], use first element as partition:



Now do an in-order tree-traversal and we get the list in sorted order.
What's going on if we do this in-place in the array:



We end up with the sorted array at the end of the recursive steps, following the tree from left-to-right (inorder).

All work is done in Partition.
Worst case runtime: $T(n) = T(n-1) + \Theta(n)$ which we know is $\Theta(n^2)$

Best case runtime: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ which is the same as Merge Sort we know is $\Theta(n \lg n)$

Average case: Same argument as before, alternating good and bad splits. Results in same as the best case runtime but with larger constants than the best case, $\Theta(n \lg n)$.

Even though Quick Sort has the same average case runtime than Merge Sort ($\Theta(n \lg n)$), usually Quick Sort has smaller runtime constants than Merge sort, resulting in an overall faster execution time.

What if we ran the median of median strategy to find partition point? Still would get $\Theta(n \lg n)$. Random strategy usually best, pick a small # of random elements, and use median of those elements as the partition point.