Recitation 2

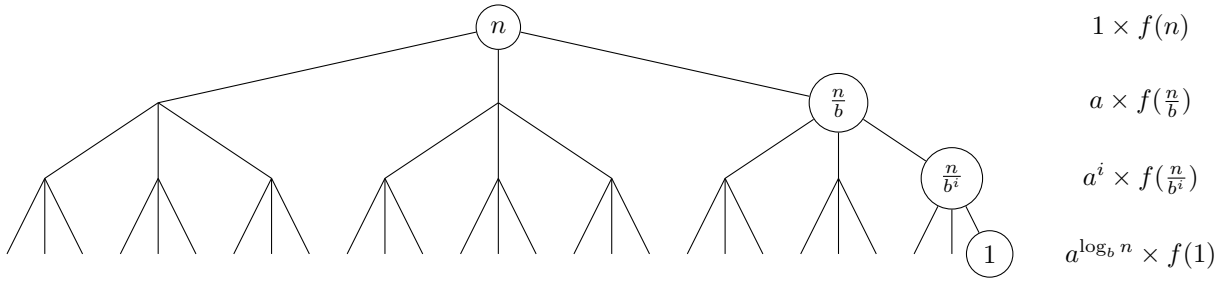
Recurrences

There are three primary methods for solving recurrences:

- **Substitution:** Guess a solution and substitute to show the recurrence holds.
- **Recursion Tree:** Draw a tree representing the recurrence and sum computation at nodes. This is a very general method, and is the one we've used in lecture so far.
- **Master Theorem:** A general formula to solve a large class of recurrences. It is useful, but can also be hard to remember.

Master Theorem

The **Master Theorem** provides a way to solve recurrence relations in which recursive calls decrease problem size by a constant factor. Given a recurrence relation of the form $T(n) = aT(n/b) + f(n)$ and $T(1) = \Theta(1)$, with branching factor $a \geq 1$, problem size reduction factor $b > 1$, and asymptotically non-negative function $f(n)$, the Master Theorem gives the solution to the recurrence by comparing $f(n)$ to $a^{\log_b n} = n^{\log_b a}$, the number of leaves at the bottom of the recursion tree. When $f(n)$ grows asymptotically faster than $n^{\log_b a}$, the work done at each level decreases geometrically so the work at the root dominates; alternatively, when $f(n)$ grows slower, the work done at each level increases geometrically and the work at the leaves dominates. When their growth rates are comparable, the work is evenly spread over the tree's $O(\log n)$ levels.



case	solution	conditions
1	$T(n) = \Theta(n^{\log_b a})$	$f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$
2	$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$	$f(n) = \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$
3	$T(n) = \Theta(f(n))$	$f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$ and $af(n/b) < cf(n)$ for some constant $0 < c < 1$

The Master Theorem takes on a simpler form when $f(n)$ is a polynomial, such that the recurrence has the from $T(n) = aT(n/b) + \Theta(n^c)$ for some constant $c \geq 0$.

case	solution	conditions	intuition
1	$T(n) = \Theta(n^{\log_b a})$	$c < \log_b a$	Work done at leaves dominates
2	$T(n) = \Theta(n^c \log n)$	$c = \log_b a$	Work balanced across the tree
3	$T(n) = \Theta(n^c)$	$c > \log_b a$	Work done at root dominates

This special case is straight-forward to prove by substitution (this can be done in recitation). To apply the Master Theorem (or this simpler special case), you should state which case applies, and show that your recurrence relation satisfies all conditions required by the relevant case. There are even stronger (more general) formulas[1] to solve recurrences, but we will not use them in this class.

Exercises

1. $T(n) = T(n-1) + O(1)$

   Solution: $T(n) = O(n)$, length n chain, $O(1)$ work per node.

2. $T(n) = T(n-1) + O(n)$

   Solution: $T(n) = O(n^2)$, length n chain, $O(k)$ work per node at height k.

3. $T(n) = 2T(n-1) + O(1)$

   Solution: $T(n) = O(2^n)$, height n binary tree, $O(1)$ work per node.

4. $T(n) = T(2n/3) + O(1)$

   Solution: $T(n) = O(\log n)$, length $\log_{3/2}(n)$ chain, $O(1)$ work per node.

5. $T(n) = 2T(n/2) + O(1)$

   Solution: $T(n) = O(n)$, height $\log_2 n$ binary tree, $O(1)$ work per node.

6. $T(n) = T(n/2) + O(n)$

   Solution: $T(n) = O(n)$, length $\log_2 n$ chain, $O(2^k)$ work per node at height k.

7. $T(n) = 2T(n/2) + O(n \log n)$

   Solution: $T(n) = O(n \log^2 n)$ (special case of Master Theorem does not apply because $n \log n$ is not polynomial), height $\log_2 n$ binary tree, $O(k \cdot 2^k)$ work per node at height k.

8. $T(n) = 4T(n/2) + O(n)$

   Solution: $T(n) = O(n^2)$, height $\log_2 n$ degree-4 tree, $O(2^k)$ work per node at height k.

[1] `http://en.wikipedia.org/wiki/Akra-Bazzi_method`

2D Peak Finding

A **peak** in a two-dimensional $n \times m$ array of integers is an element of the array that is greater than or equal to each of its eight adjacent neighbors (some of which may not exist if the element is on the boundary). As with a 1D array, every 2D array of integers contains a peak, as a maximum element of the array will be a peak. A brute force $O(nm)$ algorithm can solve this problem by checking whether each element of the array is a peak or not. Let's find a more efficient algorithm!

1. We know how to find a peak in a 1D array in $O(\log n)$ time. Describe an algorithm that uses the 1D array algorithm to find a peak in an $n \times m$ array (with $m > n$) in $O(n \log m)$ time.

   Solution: Suppose we had an array B containing a maximum element from each column. We could then use the 1D peak finding algorithm from Recitation 01 to find a 1D peak in B. Certainly a 1D peak p from B would be a peak in the original 2D array, as p is at least as large as its neighbors. Unfortunately, computing an element of B requires reading an entire column in $\Omega(n)$ time, so computing all of B would take $O(nm)$ time. However, recall the 1D peak finding algorithm only reads $O(\log m)$ elements of a 1D input array. So instead of computing all elements of B ahead of time, only computing a maximum element from columns that are touched by the 1D search algorithm yields a $O(n \log m)$ time algorithm.

2. Let's try for an even faster algorithm! Consider a two-dimensional array A and rectangular sub-array $B \subseteq A$. Prove that if sub-array B contains an element that is greater than or equal to every element of A directly outside and adjacent to B, then B contains a peak.

   Solution: There exists an integer p in B that is at least as large as all the other integers in B. Then the only way for p to **not** be a peak in A is if it were adjacent to a larger integer outside of B. Since B contains an integer (witness) that is greater than or equal to every element of A directly outside and adjacent to B, then p, being at least as large as the witness, will also be greater than or equal to the elements outside and adjacent to B. Thus p is a peak.

3. Given an $n \times m$ array with $n \geq m$, divide it by splitting its longer dimension, n, in half. Show how to identify a witness in one of the halves in $O(m)$ time, certifying a peak.

   Solution: Consider the set of $2m$ elements comprising the two rows or columns of elements adjacent to the line separating the array's longer dimension in half. A maximal integer p among those elements will be at least as large as any array element outside and adjacent to the half containing p. Thus p is a witness certifying a peak in the half containing p.

4. Our algorithm will divide the array in half by splitting its longer dimension, identify a witness in one of the halves, and then recursively search within the half containing the witness. Write and solve a recurrence relation based on this algorithm. How does the running time of this algorithm relate to input size, i.e., is the algorithm sub-linear, linear, or super-linear?

   Solution: The input size of this problem is $\Theta(nm)$. A recurrence relation for this algorithm is $T(nm) = T(nm/2) + O(\min(n, m))$. Here, we note that $\min(n, m) \leq \sqrt{nm} \leq \max(n, m)$, so by letting $N = nm$, we get the recurrence relation $T(N) = T(N/2) +$

$O(\sqrt{N})$. This recurrence relation evaluates to $T(nm) = T(N) = O(\sqrt{N}) = O(\max(n, m))$ by case 3 of the master theorem, since $f(N) = \Omega(N^{\log_2 1}) = \Omega(1)$. When $m = O(n)$, this algorithm is $O(\sqrt{nm})$, which is **sub-linear** in the input $O(nm)$.

Below is some Python code implementing this algorithm. In addition, we've made a visualizer for this algorithm in CoffeeScript, a little language that compiles into JavaScript that looks a lot like Python. You can find it here: `https://codepen.io/mit6006/pen/mGBGgO`.

```
def find_peak_2D(A, r = None, w = None):
    '''
    Find a peak in a two dimensional array.
    Input: 2D integer array A, subarray indices r, witness w
        (Note: indices of subarray r are INCLUSIVE)
    '''
    if r is None:
        r = (0, 0, len(A[0]) - 1, len(A) - 1)
    px, py, qx, qy = r
    if w is None:
        w = (0, 0)
    wx, wy = w
    if (px == qx) and (py == qy):
        return (px, py)          # base case
    if qx - px > qy - py:        # larger dimension in x
        c = (px + qx + 1) // 2   # center
        for x in [c - 1, c]:     # find new witness
            for y in range(py, qy + 1):
                if A[y][x] > A[wy][wx]:
                    wx, wy = x, y
        if wx < c:               # new witness in right half
            qx = c - 1
        else:                    # new witness in left half
            px = c
    else:                        # larger dimension in y
        c = (py + qy + 1) // 2   # center
        for y in [c - 1, c]:     # find new witness
            for x in range(px, qx + 1):
                if A[y][x] > A[wy][wx]:
                    wx, wy = x, y
        if wy < c:               # new witness in bottom half
            qy = c - 1
        else:                    # new witness in top half
            py = c
    return find_peak_2D(A, (px, py, qx, qy), (wx, wy))
```