COSC 6111 Advanced Design and Analysis of Algorithms
Jeff Edmonds
Assignment: Recursion

First Person:
Family Name:
Given Name:
Student #:
Email:

Second Person:
Family Name:
Given Name:
Student #:
Email:

Skip 1, 2abc, 5 and 7. Do 2de,3,4, and 6. Let me know if you have seen them before.

Problem Name	If Done Old Mark	Check if to be Marked	New Mark
1 Finite Fields			
2 Size of Recursive Instance			
3 Recursion on Matrix			
4 Parsing			
5 Harder Parsing			
6 Recursive Tiling			
7 Multiplying			

1. Finite Fields. (Done in class)

   (a) Find the inverse of 20 in Z_{67}. To show your work, make a table with columns u, v, r, s, and t and a row for each stack

   (b) Given as input $I = \langle a, b, p \rangle$ compute $a^b \ mod \ p$. The algorithm is in the notes. Do not copy the algorithm. The section reference (and necessary changes) is sufficient. What is the number of bit operations needed as a function of $n = |I| = \log_2(a) + \log_2(b) + \log_2(p)$?

   (c) Given as input $I = \langle a, c, p \rangle$ solve $a^b \equiv_{mod\ p} c$ for b. This is called *discrete log*. What is the best algorithm that you can come up with in 15min. (Do not cheat and spend more time than this.) What is the number of bit operations needed as a function of $n = |I| = \log_2(a) + \log_2(c) + \log_2(p)$?

2. (a, b, and c done in class) In *friends* level of abstracting recursion, you can give your friend any legal instance that is smaller than yours according to some measure as long as you solve on your own any instance that is sufficiently small. For which of these algorithms has this been done? If so what is your measure of the size of the instance? On input instance $\langle n, m \rangle$, either bound the depth to which the algorithm recurses as a function of n and m or prove that there is at least one path down the recursion tree that is infinite.

algorithm $R_a(n, m)$
⟨***pre−cond***⟩**:** n & m ints.
⟨***post−cond***⟩**:** Say Hi

begin
    if($n \leq 0$)
        Print("Hi")
    else
        $R_a(n-1, 2m)$
    end if
end algorithm

algorithm $R_b(n, m)$
⟨***pre−cond***⟩**:** n & m ints.
⟨***post−cond***⟩**:** Say Hi

begin
    if($n \leq 0$)
        Print("Hi")
    else
        $R_b(n-1, m)$
        $R_b(n, m-1)$
    end if
end algorithm

algorithm $R_c(n, m)$
⟨***pre−cond***⟩**:** n & m ints.
⟨***post−cond***⟩**:** Say Hi

begin
    if($n \leq 0$ or $m \leq 0$)
        Print("Hi")
    else
        $R_c(n-1, m)$
        $R_c(n, m-1)$
    end if
end algorithm

algorithm $R_d(n, m)$
⟨***pre−cond***⟩**:** n & m ints.
⟨***post−cond***⟩**:** Say Hi

begin
    if($n \leq 0$ or $m \leq 0$)
        Print("Hi")
    else
        $R_d(n-1, m+2)$
        $R_d(n+1, m-3)$
    end if
end algorithm

algorithm $R_e(n, m)$
⟨***pre−cond***⟩**:** n & m ints.
⟨***post−cond***⟩**:** Say Hi

begin
    if($n \leq 0$ or $m \leq 0$)
        Print("Hi")
    else
        $R_e(n-4, m+2)$
        $R_e(n+6, m-3)$
    end if
end algorithm

3. Recursion:

   (a) We asked for an iterative algorithm for searching within a matrix $A[1..n, 1..m]$ in which each row is sorted and each column is sorted. This requires that $T(n, m) = n+m-1$ of the matrix entries be examined. Our lower bound proves that this is tight when $n = m$. But this is clearly too big

when $m >> n$, given one can do binary search in each row in time $n \log m << n+m-1$. The goal now is to design a recursive algorithm that accesses $T(n,m) \approx n \log_2(\frac{m}{n})$ entries. As a huge hint, the recurrence relation will be $T(n,m) = \max_{i \in [1,n]} T(i, \frac{m}{2}) + T(n-i, \frac{m}{2}) + \log_2 n$.

(b) (DONT NEED TO DO) Recurrence Relations: Consider the recurrence relation $T(n,m) = \max_{n' \in [1,n]} T(n', \frac{m}{2}) + T(n-n', \frac{m}{2}) + \log_2 n$.
You must look at the recursive tree in order to get some intuition to why the time is $T(n,m) \approx n \log_2(\frac{m}{n})$. You can also plug $T(n,m) = n \log_2(\frac{m}{n}) + 2n - \log(n) - 2$ into this recurrence relation and see that it satisfies it.

(c) (DONT NEED TO DO) We have seen three algorithms for the sorted matrix problem with running times $T_1(n,m) = n + m$, $T_2(n,m) = n \log(\frac{m}{n})$ and $T_3(n,m) = n \log(m)$. We will be considering families of matrices. In a particular family, the height n can have any value and the width $m = m(n)$ is some function of n. The recursive $T_2(n,m)$ algorithm was a pain to write and implement. Hence, we only want to use it for families of matrices where the running is really little oh of the running time of other two algorithms, i.e. not Theta. Use $\mathcal{O}$, o, Θ, Ω, and ω, to bound for which functions $m(n)$ we will use this algorithm. Start by trying different functions. Give extreme examples that work and others that do not work.

4. Parsing

Look Ahead One: A grammar is said to be *look ahead one* if, given any two rules for the same non-terminal, the first place that the rules differ is a difference in a terminal. (Equivalently the rules can be views as paths down a tree.) This feature allows our parsing algorithm to look only at the next token in order to decide what to do next. Thus the algorithm runs in linear time. An example of a good set of rules would be:

A ⇒ B 'u' C 'w' E
A ⇒ B 'u' C 'x' F
A ⇒ B 'u' C
A ⇒ B 'v' G H

(Actually, even this grammar could also be problematic if when $s = bbbucccweee$, B could either be parsed as bbb or as $bbbu$. Having B *eat* the $'u'$ would be a problem.)
An example of a bad set of rules would be:

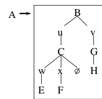
A ⇒ B C
A ⇒ D E

With such a grammar, you would not know whether to start parsing the string as a B or a D. If you made the wrong choice, you would have to back up and repeat the process.

Consider a grammar G which includes the four look ahead rules for A given above. Give the code for $GetA\,(s,i)$ that is similar to that for $GetExp\,(s,i)$. We can assume that it can be parsed, so do not bother with error detection. HINT: The code should contain NO loops.

5. Harder Parsing: If you are feeling bold, try to write a recursive program for a generic parsing algorithm. The input is $\langle G, T, s, i \rangle$, where G is a look ahead one grammar, T is a non-terminal of G, s is a string of terminals, and i is an index. The output consists of a parsing of the longest substring $s[i], s[i+1], \ldots, s[j-1]$ of s that starts at index i and is a valid "T" according to the grammar G. In other words, the parsing starts with non-terminal T and ends with the string $s[i], s[i+1], \ldots, s[j-1]$. The output also includes the index j of the token that comes immediately after the parsed expression. For example, $GetExp(s,i)$ is the same as calling this algorithm on $\langle G, exp, s, i \rangle$ where G is the grammar given above.

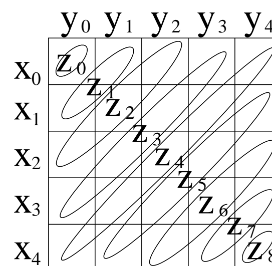
It is helpful to think of the grammar as a tree. The grammar from the last question would be:



The loop invariant is that you have parsed a prefix $s[i], s[i+1], \ldots, s[j'-1]$ of s producing a partial parsing p and the rest of the string $s[j'], s[j'+1], \ldots, s[j-1]$ will be parsed using the one of the partial rules in the set R. For example, suppose the grammar G includes the four look ahead rules for A given above, we are starting with the non-terminal $T = A$, and we are parsing the string $s = bbbucccweee$. Initially, we have parsed nothing and R contains all of each of the four rules, namely $R =$ {BuCwE, BuCxF, BuC, BvGH}. After two iterations, we have parsed $bbbu$ using a parsing p_B for bbb followed by the character u. We must parse the rest of the string $cccweee$ using one of the rules in $R =$ {CwE, CxF, C}. Note that the used up the prefix Bu from the consistent rules and the inconsistent rules were deleted. Because the grammar is *look ahead one* we know that either the first token in each rule of R is the same non-terminal B, or each rule of R begins with a terminal or is the empty rule. These are the two cases your iteration needs to deal with.

6. Recursionive Tiling: The precondition to the problem is that you are given three integers $\langle n, i, j \rangle$, where i and j are in the range 1 to 2^n. You have a 2^n by 2^n square board of squares. You have a sufficient number of tiles each with the shape ⌐̲. Your goal is to place non-overlapping tiles on the board to cover each of the $2^n \times 2^n$ tiles except for the single square at location $\langle i, j \rangle$. Give a recursive algorithm for this problem.

7. In class we got a $\Theta(n^{\frac{\log 3}{\log 2}}) = \Theta(n^{1.58})$ time algorithm for multiplying two n-bit integers by cutting the numbers into two parts and managing to using only three friends instead of four. For this question, cut the numbers into d parts. As a bit string $\langle x \rangle = \langle x_{d-1}, \ldots, x_1, x_0 \rangle$ and as an integer $x = \sum_{i=0}^{d-1} x_i 2^{ni/d}$. Similarly, $y = \sum_{j=0}^{d-1} y_j 2^{nj/d}$. Multiplying them gives $z = x \times y = \sum_{i=0}^{d-1} \sum_{j=0}^{d-1} x_i y_j 2^{n(i+j)/d} = \sum_{k=0}^{2d-2} \sum_{i=0}^{k} x_i y_{k-i} 2^{nk/d} = \sum_{k=0}^{2d-2} z_k 2^{nk/d}$, where $z_k = \sum_{i=0}^{k} x_i y_{k-i}$. A way to picture the values z_k is to the right.



You might note that getting the z_k from the x_i and the y_j is the same as the convolution of the coefficients resulting from the multiplication of two polynomials. We learned in class that this could be done using FFT in $\Theta(d \log d)$ time. We, however, want to do it with only $2d-1$ multiplications. Surely, we can't do it faster than FFT! The trick is to differentiate between multiplications that are *expensive* and additions that are *cheap*. Once we get the number of expensive multiplications down to the required $2d-1$, we could try to decrease to number of cheap additions using something like FFT. However, because they are cheap anyway, we will not worry so much about how many we have d^2 or d^3 of them would be fine. For this reason, we will keep it simple by only doing simple standard matrix operations.

(a) Suppose each x_i and y_j are big integers. Hence, getting a friend to multiply two of them is expensive, but multiplying one of them by a small integer is relatively cheap and as is adding two of them. Your goal is compute the $2d-1$ values of z_k using not d^2 expensive multiplications, but only $2d$–1 of them. For each $\ell \in [0, 2d-2]$, the ℓth such multiplication involves choose small integers $a_{\langle \ell,0 \rangle}, \ldots, a_{\langle \ell,d-1 \rangle}$ and $b_{\langle \ell,0 \rangle}, \ldots, b_{\langle \ell,d-1 \rangle}$ and computing $w_\ell = \left[\sum_{i=0}^{d-1} a_{\langle \ell,i \rangle} x_i\right] \times \left[\sum_{j=0}^{d-1} b_{\langle \ell,j \rangle} y_j\right]$. My hint is to let $a_{\langle \ell,i \rangle} = b_{\langle \ell,i \rangle} = (\alpha_\ell)^i$ for values α_ℓ to be chosen later. Rearrange these equations to define w_ℓ in terms of α_ℓ and the desired z_k.

(b) Give the matrix mapping from the z_k to the w_ℓ. Does this matrix look familiar?

(c) The inverse matrix maps from the w_ℓ to the z_k. You do not have to find this inverse. However, one can compute the z_k by first computing the w_ℓ and them multiplying the vector of them by this inverse matrix. Lets not do anything fancy, but just simple matrix multiplication.

Note the number of $x_i \times y_j$ type multiplications needed to compute the z_k is only $2d-1$, one for each of the w_ℓ.

Suppose each $a_{\langle \ell,i\rangle} \times x_i$ type of multiplication takes time $\Theta(\log(a_{\langle \ell,i\rangle}) \times \log(x_i)) = \Theta(\log(d^d) \times \frac{n}{d}) = \Theta(n \log(d))$.

Suppose each addition of numbers like $a_{\langle \ell,i\rangle} x_i$ takes time $\Theta(\log(x_i)) = \Theta(\frac{n}{d})$.

Compute time you spend, excluding the time your friends spend multiplying, i.e. the time needed to compute $\sum_{i=0}^{d-1} a_{\langle \ell,i\rangle} x_i$ and $\sum_{j=0}^{d-1} b_{\langle \ell,j\rangle} y_j$ for computing the w_ℓ and then the time to compute the z_k from the w_ℓ.

(d) Give a recurrence relation $T(n)$ giving the total running time of this algorithm as a function of n and the constant d.

(e) Suppose $T(n) = aT(\frac{n}{b}) + \alpha n^c$. Determine how the α affects the solution by setting $T(n) = \alpha T'(n)$, solving for $T'(n)$ and then multiplying by α to get $T(n)$.

(f) Oops, I thought that one got $T(n) = \Theta(n \log n)$ by setting $d = \log n$, but can't seem to get this after all. Given we wont get $T(n) = \Theta(n \log n)$ anyway, we might as well ignore factors of $\log n$ and of $\log d$ in our approximation of $T(n)$.

One thing that makes me a little nervous is using the Master Theorem when d is not a constant. Recall that when $T(n) = aT(n/b) + f(n)$ that $T(n) = \mathcal{O}(f(n))$, $T(n) = \mathcal{O}(n^{\log a / \log b})$, or $\log n$ times these. Given we are ignoring factors of $\log n$ we can ignore this worry.

Find the d that minimizes your $T(n)$. Instead of differentiating $T(n)$ wrt d and setting this to zero, I recommend doing the following. Factor the n out of $T(n)$ because it does not depend on d. Then take the log of $T(n)/n$, because $\log(T(n)/n)$ is minimized with the same d that $T(n)$ is minimized with. This gives two terms. A very useful thing to know is that the sum of two terms (within a factor of 2) is minimized by setting d to make the two terms equal. What does this give you for $T(n)$?

.