Algorithms by Papdimitriou, Dasgupta, U. Vazirani - Solutions

Raymond Feng

August 2017

-1 Introduction

My solutions for Algorithms by Papadimitriou, Dasgupta, U. Vazirani The intent of this solution key was originally just to practice. But then I realized that this key was also useful for collaborating with fellow CS170 students as well. For corrections email raymondhfeng@berkeley.edu.

0 Prologue

0.1

0.1.a $f = \theta(g)$

0.1.b $f = O(g)$

0.1.c $f = \theta(g)$

This result was not clear to me immediately, but after seeing that $(log(n))^2 = O(n)$, the result quickly follows. This follows from L'Hopital's rule. $\lim_{x\to\infty} \frac{(log(n))^2}{n} = \lim_{x\to\infty} \frac{2log(n)}{n} = 0$ I use L'Hopital's rule in later problems as well, but omit the work.

0.1.d $f = \theta(g)$

0.1.e $f = \theta(g)$

0.1.f $f = \theta(g)$

0.1.g $f = \Omega(g)$

0.1.h $f = \Omega(g)$

0.1.i $f = \Omega(g)$

0.1.j $f = \Omega(g)$

We have $log(n)^{log(n)} \geq \frac{n}{log(n)} \Rightarrow log(n)^{log(n)+1} \geq n$. Then, applying the log to both sides, we get: $(log(n)+1)log(log(n)) \geq log(n)$, implying $log(n)log(log(n)) \geq log(n) \Rightarrow log(log(n)) \geq 1$, which is true.

0.1.k $f = \Omega(g)$

0.1.l $f = \theta(g)$

0.1.m $f = \Omega(g)$

0.1.n $f = \theta(g)$

0.1.o $f = \Omega(g)$

The critical realization is that $log(n!)$ grows at least as fast as $nlog(n)$. This is derived from the fact that $n! > (n/2)^{n/2}$. After that, the proof then follows quickly.

0.1.p $f = O(g)$

Justification(credit: Nicholas John Riasanovsky):

$$f(n) = (log(n))^{(log(n))}$$

$$= (2^{log(log(n))})^{log(n)}$$

$$= (2^{(log(n))})^{log(log(n))}$$

$$= n^{log(log(n))}$$

$$g(n) = 2^{log(n)log(n)}$$

$$= n^{log(n)}$$

$$f = O(g)$$

0.1.q $f = O(g)$

0.2

0.2.a

An infinite geometric series with $r < 1$ will converge to a constant. A finite geometric series with $r < 1$ must then also converge to a constant.

0.2.b

If the $r = 1$, then all the terms of the finite series must all be one.

0.2.c

If $r > 1$, then we can use the formula for finite geometric series, which is $\theta(r^n)$

0.3

0.3.a

Base case: $n = 6$. $2^{0.5*6} = 8$. $F_6 = 8$.
Inductive Hypothesis: For a number $k \geq 6$, if $F_k \geq 2^{0.5*k}$, then $F_{k+1} \geq 2^{0.5*(k+1)}$.
Inductive Step: $F_{k+1} = F_{k-1} + F_k \geq 2^{0.5*(k-1)} + 2^{0.5*k} \geq 2^{0.5*(k+1)}$.

0.3.b

To do this problem, I assume that I already found my c, and thus, can write the following.

$$2^{c(n-1)} + 2^{c(n)} \leq 2^{c(n+1)}$$

$$2^{cn-c} + 2^{cn} \leq 2^{cn+c}$$

$$2^{-c} + 1 \leq 2^{c}$$

Using algebra, we then get:

$$c \geq log_2(\sqrt{5} + 1) - 1$$

Because I assumed that c was already known, the method in which the problem was solved guarantees that the answer is correct.

0.3.c

As from the previous question, $c = log_2(\sqrt{5} + 1) - 1$

0.4

0.4.a

...that's just how matrix multiplication is defined, I guess.

0.4.b

Keep squaring the matrix, and multiply the correct powers of the matrix that add up to the proper exponent. Repeated squaring should take $O(log(n))$ time to get the the desired maximum power.

0.4.c

...not sure exactly what this question is asking? Would the intermediate results just be the matrix powers? Which if so yeah I suppose they are $O(n)$ long...whatever that means.

0.4.d

We have to do $log(n)$ matrix multiplies, each one taking $M(n)$. Thus, fib3 is $O(M(n)log(n))$.

0.4.e

Instead of assuming that all matrix multiples taking $M(n)$, we look at each matrix multiply individually. The running time is now

$$M(1) + M(2) + M(4) + ... + M(n)$$
$$= 2M(n)$$

By the formula for finite geometric sums.

1 Algorithms with Numbers

1.1

To start, the case of $b = 2$ is proved by first maximizing the value of the three single digit numbers that are going to be added together, call it a. So then $a = b-1$. Adding three of those together would then be $3a = 3b-3 = 3 = 0b11$, which is a two digit number. The cases from $3 \leq a \leq \infty$ can be generalized as follows. For the case of $b = 3$, the maximum of the sum of the three numbers is again $3b - 3$. Now, simply take the most significant digit, and set that to be 2, leaving $b - 3$ for the least significant digit. And because we care about the cases now where $b \geq 3$, $b-3$ is and always will be representable using the LST. It is now clear that for higher bases, the same holds.

1.2

For any decimal number of length n, the ratio of its length in binary vs its length in decimal is $\frac{\lceil log_2 10^n - 1 \rceil}{n}$. We want to prove that $\frac{\lceil log_2 10^n - 1 \rceil}{n} \leq 4$. However, I prefer to use proof by contradiction, and prove that the negation is false.

$$\frac{\lceil log_2(10^n - 1) \rceil}{n} > 4$$

$$4n < \lceil log_2(10^n - 1)\rceil$$
$$4n < log_2(10^n - 1)$$
$$2^{4n} < 10^n - 1$$
$$16^n + 1 < 10^n$$

Which is obviously false for $n \geq 0$. Thus the original statement must be true. To find the ratio of these two lengths for large n, we must find the limit of the ratio as $n \to \infty$.

$$\lim_{n\to\infty} \frac{\lceil log_2(10^n - 1)\rceil}{n}$$
$$\lim_{n\to\infty} \frac{log_2(10^n - 1)}{n}$$
$$\lim_{n\to\infty} \frac{log_2(10^n)}{n}$$
$$log_2 10 \approx 3.32$$

1.3

In order to minimize the depth of the d-ary tree, it should be a complete tree, that is, all depths that exist should be fully populated before beginning to populate the next level, and say that we populate from left to right for the lowest depth, should there not be enough nodes to completely populate it. Thus, this lower bound for the depth is determined by the total number of nodes in a complete tree of n nodes. So the lowest depth achievable occurs when there are no leftover nodes, where the deepest level is completely filled. Call the depth f. This would result in a d-ary tree with depth $\lfloor log_d(n)\rfloor$, where $n = d^{f+1} - 1$. Then, we can say that the depth is $\Omega(\frac{log(n)}{log(d)})$. The precise formula for the minimum depth is $\lfloor log_d(n)\rfloor$.

1.4

Observe that $n! < n^n$ for large n, because all terms in the factorial except 1, which is equal to, are less than n, which means that $log(n!) = O(log(n^n)) \Rightarrow log(n!) = O(nlog(n))$. Next, observe that $n! > (n/2)^{n/2}$ for large n, because the largest $\frac{n}{2}$ terms of $n!$ are larger than $\frac{n}{2}$, moreover, $n!$ still has its least $\frac{n}{2}$ terms. This means that $log(n!) = \Omega((\frac{n}{2})^{\frac{n}{2}}) \Rightarrow log(n!) = \Omega(nlog(n))$. These two imply that $log(n!) = \theta(nlog(n))$.

1.5

Based on the hint, we see that the harmonic series is $O(log_2(n))$ and $\Omega(log_4(n)$. Thus, we can say that the harmonic series is $\theta(log(n))$.

1.6

We know that we can use the distributive property to break up the multiplier into an addition of powers of two. We also know that the result of the multiplication will be the addition of the multiplicand multiplied by each of these powers of two that are present. Furthermore, multiplying my a power of two for a binary number is the same as shifting the number by the exponent of the power of two. With this in mind, it is clear why the algorithm on pg 24 is correct.

1.7

If y is the m bit number, then there will be m halvings of y, and thus m recursive calls. In each recursive call there will be a testing of evenness, multiplication by two, and a possible addition, taking $O(m)$, leading to a time complexity of $O(m^2)$.

1.8

There are n recursive calls because of n halvings that need to take place. Then, before each recursive call ends, there are arithmetic operations that are $O(n)$. Thus, the time complexity is $O(n^2)$. This algorithm is correct because what it is essentially doing is dividing 1 by the divisor, then doubling both the quotient and remainder, and then checking to see if the remainder has "overflowed". It continues upwards until the original value is reached, at which point the quotient and remainder will be correct.

1.9

$$x - x' + y - y' \equiv 0 \pmod N$$
$$kN + jN \equiv 0 \pmod N$$
$$0 \equiv 0 \pmod N$$

1.10

$$a \equiv b \pmod N$$
$$a \equiv b \pmod{kM}$$
$$a \equiv b \pmod M$$

1.11

Yes

1.12

1

1.13

Yes

1.14

$O(log(n) * T(log(p)))$. Because fib3 from chapter 0 takes $log(n)$ multiplication operations, and each intermediate step is $T(log(p))$ long. We also assume that a multiplication of n bit numbers take $T(n)$ time.

1.15

If you want to divide both sides of the equation by $x \pmod{c}$, then x must have an inverse modulo c. This is the case if and only if $gcd(x, c) = 1$, so x and c must be relatively prime.

1.16

In a case where it is like $71^{500} \pmod{35}$, it would be easier to first simplify 71 modulo 35, and realize that $71^{500} \equiv 1^{500} \pmod{35}$.

1.17

The iterative algorithm takes $O((log(x))^2)$ for the first iteration. Then, from then on, it takes $O(i^2(log(x))^2)$. So the time complexity would be:

$$\sum_{i=1}^{y-1} i^2(log(x))^2$$

$$O((log(x))^2 \frac{(y-1)(y)(2y-1)}{6})$$

$$O((log(x))^2 y^3)$$

The recursive algorithm has multiplication $1 * x$ for the deepest call. Then, the pattern goes x^2, x^4, x^8... Then, the time complexity for all the $log_2(y)$ recursive calls would be:

$$\sum_{i=1}^{log_2(y)} ((log_2(x^{2^i}))^2)$$

$$\sum_{i=1}^{log_2(y)} 2^{2i}(log_2(x))^2$$

$$(log_2(x))^2 \sum_{i=1}^{log_2(y)} 4^i$$

$$(log_2(x))^2(\frac{4}{3})(y^2-1)$$

$$O((log_2(x))^2y^2)$$

The recursive function has better time complexity.

1.18

$210 = 2 * 3 * 5 * 7$
$588 = 2 * 2 * 3 * 7 * 7$
$gcd(588, 210) = 42$
$gcd(588, 210) = gcd(168, 210) = gcd(168, 42) = 42$

1.19

Proof by induction.
Base case: $n = 1 \Rightarrow gcd(F_1, F_2) = gcd(1, 1) = 1$
Inductive hypothesis: For $n = k$, $gcd(F_k, F_{k-1}) = 1 \Rightarrow gcd(F_{k+1}, F_k) = 1$
Inductive step: $gcd(F_{k+1}, F_k) = 1$ is the same as $gcd(F_k + F_{k-1}, F_k) = 1$. Which is equivalent to $gcd(F_k, F_k - F_{k-1}) = 1$, which is $gcd(F_k, F_{k-2}) = 1$. Because we know from our inductive hypothesis that $gcd(F_k, F_{k-1}) = 1$ and $gcd(F_{k-1}, F_{k-2}) = 1$, then it must be the case that our original claim is true.

1.20

$20^{-1} \equiv 4 \pmod{79}$
$3^{-1} \equiv 21 \pmod{62}$
DNE
$5^{-1} \equiv 1 \pmod{23}$

1.21

In order for a number to have an inverse modulo 11^3, then this number must be relatively prime with 11^3. In the 1331 possible candidates, we must remove 11^2 of them because there are that many numbers less than or equal to 1331 that are not relatively prime to 1331. 1210.

1.22

True, because $gcd(a, b) = 1$, which means that b must have an inverse modulo a.

1.23

Suppose we have two numbers x and y that are the inverse of a modulo N.

$$ax \equiv 1 \pmod{N}$$

$$ay \equiv 1 \pmod{N}$$

$$ax \equiv ay \pmod{N}$$

Dividing both sides by a is legal because we assume that a has an inverse modulo N.

$$x \equiv y \pmod{N}$$

1.24

We have to disclude all the numbers that are a multiple of p. There are p^{n-1} of those numbers. There are p^n numbers in the set, but we have to remove 0 because zero has no inverse modulo p. Thus, there are $p^n - 1 - p^{n-1}$ inverses modulo p in that set.

1.25

Use Fermat's Little Theorem to get $2^{126} \equiv 1 \pmod{127}$. This then means that $2^{125} \equiv 1 \pmod{127}$.

1.26

Using the hint, we can do:

$$(17^{17})^4 \equiv 1 \pmod{10}$$

$$(17^{17})^{16} \equiv 1 \pmod{10}$$

Then, use the same hint to go a level deeper:

$$17^4 \equiv 1 \pmod{10}$$

$$17^{16} \equiv 1 \pmod{10}$$

And finally, go one level deeper to to final level.

$$17 \equiv 7 \pmod{10}$$

Combining all of our results gets us the following.

$$(17^{17})^{17} \equiv 7 \pmod{10}$$

1.27

$d = 235$
Encrypted message: 105

1.28

A scheme that would work is $e = 7, d = 43$.

1.29

1.29.a

In the same spirit as the example in the textbook, we can assume without loss of generality that the first two elements of some random tuple (x_1, x_2, x_3) are the same as the first two elements of another random tuple (y_1, y_2, y_3), and that $y_3 \neq x_3$. Then, let us calculate the chances that the two tuples hash to the same value. The tuples hash to the same value if and only if the following expression is true.

$$\sum_{i=1}^{2} a_i(x_i - y_1) \equiv a_3(y_3 - x_3) \pmod{m}$$

We can say that the value on the left is equal to some value c. Then we have:

$$c \equiv a_3(y_3 - x_3) \pmod{m}$$

In order for the above expression to hold true, then the following must be true.

$$a_3 \equiv c(y_3 - x_3)^{-1}$$

Because m is prime and $y_3 \neq x_3$, we know that there is a distinct value of $(y_3 - x_3)^{-1} \pmod{m}$. And of course c is distinct modulo m. Thus, the probability of choosing a_3 in such a way is $\frac{1}{m}$. This hashing function is universal, and three random bits are required to choose a function from this family.

1.29.b

This modification of the previous hashing function is no longer universal because m is not prime, and thus $(y_1 - x_1)^{-1}$ may no longer exist. Three random bits are required to choose a function from this family.

1.29.c

This hash function is not universal. This is because the chances of two inputs a and b to hash to the same thing are not $\frac{1}{m}$. The pigeonhole principle dictates that there must be at least a pair of inputs that map to the same output, due to the pre-image being larger than the image of this function. Thus, there is at least one input that, in comparison to other inputs, has a high
This hash function is universal. Because the domain is one element larger than the range, that means that there must be a pair of inputs that map to the same output. First, take the simpler family of functions

$$f : [m] \to [m]$$

This family has $m!$ functions. Now, in order to modify this family into:

$$f : [m] \to [m-1]$$

Take every function in the former family, and take the input that maps to m, and map it to something different. There are $m-1$ choices of what new to map it to. Thus, in our latter family, there are $m!(m-1)$ functions. It is clear, due to symmetry or something, that the number of functions that map two inputs to the same bucket are the number of functions divided by $m = 1$.

1.30

1.30.a

Because we have, I assume, unlimited circuits, we group the n numbers into $\frac{n}{2}$ buckets, where each group of two adjacent numbers are put into the same bucket. These two numbers in each of these buckets are added together in $O(log(m))$ time, and because the number of buckets is halved each time, we will repeat this $O(log(n))$ times. Thus, the total time taken is $O(log(n)log(m))$.

1.30.b

Incomplete

2 Divide and Conquer

2.1

I'm not going to type out this step by step. I did it on paper, it was tedious. You can do it too. Result: 111000010011110.

2.2

Imagine $[n, bn]$ as a stick that is $n(b-1)$ long. We are trying to place this stick in a way such that it fits in all the gaps in between the powers of b. The shortest this stick can be is $b-1$, and that doesn't work because the gap between 1 and b is already too small for the stick. Thus, modifying the b or n in any way does not change the answer because the gap between b and 1 will never be satisfied, even with the shortest possible stick.

2.3

2.3.a

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

$$\leq 3(3(T(\frac{n}{4}) + c(\frac{n}{2})) + cn$$

$$\leq 3^k T(\frac{n}{2^k}) + \sum_{i=0}^{log_2(n)} 3^i (\frac{cn}{2^k})$$

$$k = log_2(n)$$
$$n^{log_2(3)}T(1) + c(\frac{n^{log_3(2)}}{2} - \frac{1}{2})$$
$$T(n) = O(n^{log_2(3)})$$

A double check using the master theorem corroborates our answer.

2.3.b

$$T(n) = T(n-1) + O(1)$$
$$\leq T(n-1) + cn$$
$$\leq T(n-2) + c(n-1) + cn$$
$$= T(1) + c\sum_{i=0}^{n}(n-i)$$
$$= O(n^2)$$

2.4

I would choose algorithm C.

$$T_A(n) = O(n^{log_2(5)})$$
$$T_B(n) = O(2^n)$$
$$T_C(n) = O(n^2 logn)$$

2.5

2.5.a $O(n^{log_3(2)})$

2.5.b $O(n^{log_4(5)})$

2.5.c $O(nlogn)$

2.5.d $O(n^2 logn)$

2.5.e $O(n^3 logn)$

2.5.f $O(n^{log_2(49)})$

2.5.g $O(n)$

2.5.h $O(n^{c+1})$

I am unsure about this answer.

2.5.i $O(c^n)$

2.5.j $O(2^n)$

2.5.k $O(log(logn))$

2.6

I tried really hard, but I still don't know what this question is asking. :(

2.7

The sum of the nth roots of unity is zero.

$$1 + \omega + \omega^2 + ... + \omega^{n-1} = \frac{\omega^n - 1}{\omega - 1} = 0$$

To calculate the product of the nth roots of unity...

$$e^{(\sum_{k=0}^{n-1} \frac{2\pi ki}{n})}$$

$$= e^{\pi i(n-1)}$$

$$= \cos((n-1)\pi) + i\sin((n-1)\pi)$$

The product of the nth roots of unity are -1 and 1 when n is even and odd respectively.

2.8

I ran the FFT algorithm on pg 79 by hand, and got $(1, 1, 1, 1)$. Then, using the inversion formula on page 77, I got that $(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$ is the sequence, that when performed the FFT on, will result in $(1, 0, 0, 0)$.

2.9

2.9.a

I did the two FFT subroutines, and the one inverse FFT subroutine by hand, and got $(1, 1, 1, 1)$, which can be verified by doing a naive polynomial multiplication.

2.10

I did the calculations by hand...tedious, but straightforward. $(2, 5, 7, 6)$.

2.11

I am not sure if there is a faster way than simply using Lagrange interpolation, or inverting the Vandermonde matrix. The problem here is that the points are not simply the nth roots of unity, so we cannot use the inversion formula and FFT.

$$f(x) = -\frac{3x^4}{4} + \frac{25x^3}{3} - \frac{125x^2}{4} + \frac{137x}{3} - 20$$

2.12 $\theta(n)$

2.13

2.13.a Typesetting trees is too hard. Sorry.

2.13.b

$$B_n = 2B_{n-1} + O(1)$$

2.13.c

$$B_n \leq 2^n B_1 + cn$$
$$B_n \geq 2^n B_1 + cn$$
$$B_n = \theta(2^n)$$
$$B_n = \Omega(2^n)$$

Not really induction, but...meh.

2.14

Insert all of the elements into a binary search tree. During an insertion, if the element is already in the tree, then simply go on to inserting the next element. Then, converting back to an array is a simple traversal of the tree.

2.15

Implementing the median algorithm in place would mean first choosing a pivot randomly, as before, but instead of placing each element into new subarrays, "shift" the location of the pivot to make space for the compared element if it is less. If the element is greater than the pivot, then swap the location of the compared element with the boundary of where the algorithm has gotten to in the array. This implementation would need registers to temporarily store values, but no new memory would have to be allocated.

2.16

I am inspired by the binary search algorithm, but instead of starting at a predefined root, start at index 1(I assume this array is 1-indexed). If I find the element here, great, but if not, I double my index, and check to see if my element is equal, less than, or greater than the desired element. If it is equal, done. If it is greater than, then double my index again. If it is less than, then we went passed our element, and something must be done. At this point, halve our current index in order to return to the previous index. Shift our entire array to the left so that our index is now at 1. Add the amount we shifted to a variable that keeps track of the total amount shifted so that we can restore the original index when we finish. Then, now that we are back at index 1, restart the same algorithm. We must have at least halved our array in one iteration of this algorithm, and so our runtime is $O(nlogn)$.

2.17

Binary search on a sorted array. But WAIT, this is not a divide and conquer algorithm because we are not dividing into subproblems and the combining the results. Darn. Or maybe it is??? Who knows, but all I know is that I cannot find another way to do it.

2.18

The array is sorted, and so we have the advantage that we can eliminate any "half" of the array that is greater than or less than the element we are looking for. However, the maximum we can consistently eliminate each time is 50 percent, or half. If we wanted to gamble, and try to remove more than half the array, we could, and perhaps get lucky. But if we fail, there will be a punishment. Thus, the optimal elimination amount is half. And thus, with $log_2(n)$ halvings, we can and must find our element. Thus, any comparison based search algorithm is $\Omega(log_2(n))$.

2.19

2.19.a $O(k^2n)$

2.19.b

This is the equivalent of being at the $log_2(n)$ level of merge sort. So we just continue upwards, and the process takes $O(klogkn)$.

2.20

We could just use radix sort, which is linear time for small amounts of buckets. The $O(nlogn)$ lower bound does not apply because this is not a comparison based sort, and also because we are given information beforehand about the range of the elements in the array.

2.21

2.21.a

Suppose we have $\mu' \neq \mu_1$. Then, from the context of a sorted array, there are either more elements to the left of μ' or to the right. So, if we were to change μ' by modifying it to be the element to the right or left of it, one of the options makes the expression function described earlier, smaller. Suppose we moved μ' to the element to the left of it. Say this move shifted μ' by amount k. Then, the net difference to the function must either go up or down because μ' shifted k away from either more or less elements than it shifted k closer to. By symmetry, we can apply this argument to shifting μ' to the right. One of these shifts must increase the function, and the other shift must decrease the function. Thus, μ distinctly minimizes the function.

2.21.b

$$\sum_i (x_i - \mu)^2 = \sum_i (x_i - \mu_2)^2 + n(\mu - \mu_2)^2$$

$$\sum_i (x_i^2 - 2x_i\mu + \mu^2) = \sum_i (x_i^2 - 2x_i\mu_2 + \mu_2^2) + n\mu^2 - 2\mu\mu_2 n + n\mu_2^2$$

$$\sum_i x_i^2 - 2\mu \sum_i x_i + n\mu^2 = \sum_i x_i^2 - 2\mu_2 \sum_i x_i + n\mu_2^2 + n\mu^2 - 2\mu\mu_2 n + n\mu_2^2$$

$$-2\mu \sum_i x_i = -2\mu_2 \sum_i x_i + n\mu_2^2 - 2\mu\mu_2 n + n\mu_2^2$$

$$-2\mu \sum_i x_i + 2\mu_2 \sum_i x_i = 2n\mu_2^2 - 2\mu\mu_2 n$$

$$-2\mu\mu_2 + 2\mu_2\mu_2 = 2\mu_2^2 - 2\mu\mu_2$$

$$0 = 0$$

2.21.c