zachary_hightower_433_hwk_2

Exercises 2.1:
2

a. Consider the definition-based algorithm for adding two n × n matrices.
What is its basic operation?

Ans: The basic operation is addition. This is because we obtain the final matrix by adding the sums of corresponding elements from the rest of the matrix.
E.g.

$$3[i][j] = 1[i][j] + 2[i][j]$$

How many times is it performed as a function of the matrix order n?

Ans: The numbers of times we perform the basic operation is n x n. This is because we have n portions of the matrix and we perform addition on it n numbers of times. Thus we can say that the number of times the function is performed on the matrix order n, is as follows:

$$Operations = n * n = n^2$$

As a function of the total number of elements in the input matrices?

Ans: The number of times we perform the operation as a function of the number of elements in our input matrices is n x n, which we can write as big N. We perform 1 addition operation to fill each cell, so the amount of basic operations we perform is.
So

$$Operations = N$$

Additionally we can say that both of these conclusions are equal.

All of this operates under the assumption that we are not working with ragged arrays / uneven matrices. In order to implement that, we would need more complex algorithms and formulas.

b. Answer the same questions for the definition-based algorithm for matrix
multiplication.

What is its basic operation?

Ans: The basic operation is multiplication, followed by addition. This is because we obtain the final matrix by multiplying corresponding cells and then adding the sums to form our final matrix.

So, we actually perform two basic operations here. Multiplication, followed by addition.

$$3[i][j] = \sum_{k=0}^{n}(1[i][k] * 2[k][j])$$

How many times is it performed as a function of the matrix order n?

Ans: We perform n amount of multiplications. We perform n amount of additions. We do this for a matrix of order n. We can define this by n x n x n. The formula looks something like as follows:

$$Operations = n * n * n = n^3$$

As a function of the total number of elements in the input matrices?

Ans: The number of elements in our input matrices is n x n, which we can write as big N. We perform 1 multiplication operation to fill each cell. We perform 1 multiplication operation to fill each cell. So the amount of basic operations we perform is.

$$Operations = N^2$$

Because we have an n x n amount of elements and we must perform n x n amount of operations to fill each cell of our final matrix.

These two formulas are noticeably unequal.

4

a. Glove selection There are 22 gloves in a drawer: 5 pairs of red gloves, 4 pairs of yellow, and 2 pairs of green. You select the gloves in the dark and can check them only after a selection has been made. What is the smallest number of gloves you need to select to have at least one matching pair in the best case?

Ans:

In the best case we can get a lucky run. We select two gloves of the same color.
Step 1: 1 total selections, RR, YY, or GG, 2 total gloves

In the worst case?

Ans:

Step 1: 1 selection of two gloves, red and yellow or any two unmatched colors, 2 total gloves.

Step 2: We leave these gloves out in the lighted room.

Step 3: 1 selection of two gloves, no matter what we select, we will have a pair now.

2 total selections, 4 total gloves

Because if we select RR, YY, RY, GR, GY, or GG, all of these options will result in at least 1 pair with the gloves we've already selected in the first step

b. Missing socks Imagine that after washing 5 distinct pairs of socks, you discover that two socks are missing. Of course, you would like to have the largest number of complete pairs remaining. Thus, you are left with 4 complete pairs in the best-case scenario and with 3 complete pairs in the worst case. Assuming that the probability of disappearance for each of the 10 socks is the same,

find the probability of the best-case scenario;

Ans:
We can consider the number of socks we have as n = 10. The number of socks we will lose as k = 2.
The total number of possible lost sock combinations is given by:

$$\frac{n!}{k!(n-k)!} = \frac{10!}{2!(10-2)!} = 45$$

So there are 45 possible variations of our scenario. We know that each pair is distinct. We only have five pairs. So, we can assume that there are only five possible variations for the best case scenario.
So we can find the probability for one of our five best case combinations to occur from:

$$\frac{5}{45} = \frac{1}{9}$$

The probability of the worst-case scenario;

Ans:

In the worst case scenarios, we've simply encountered one of the other forty combinations, so we can find that from the formula:

$$\frac{40}{45} = \frac{8}{9}$$

the number of pairs you should expect in the average case.

Ans:

On average we should expect 3 pairs. Most of the time we're going to be operating in the 40 worst case scenarios out of our 45 possible scenarios.

7

Gaussian elimination, the classic algorithm for solving systems of n linear equations in n unknowns, requires about $\frac{1}{3}n^3$ multiplications, which is the algorithm's basic operation.

a. How much longer should you expect Gaussian elimination to work on a system of 1000 equations versus a system of 500 equations?

Ans:

We can solve this by finding the amount of multiplications 500 and 1000 systems use, and the amount of speed increase will be the ratio of equations system A over equation system B

$$(\frac{1}{3} * 500^3) = SystemA = 41666667$$

$$(\frac{1}{3} * 1000^3) = SystemB = 333333333$$

$$\frac{SystemA}{SystemB} = \frac{41666667}{333333333} = 0.125$$

So the smaller of the two is about 12.5% faster.

b. You are considering buying a computer that is 1000 times faster than the one you currently have. By what factor will the faster computer increase the sizes of systems solvable in the same amount of time as on the old computer?

Ans:

Let's represent how long the old system takes in the following way, where x represents the time factor of the old system.

$$T_{old} = x * (\frac{1}{3} * 500^3)$$

This formula represents how much faster our new system is.

$$T_{new} = T_{old} * \frac{1}{1000}$$

So we just need to solve the ratio to find the speed increase:

$$Speed = \frac{T_{old}}{T_{new}} = \frac{T_{old}}{T_{old} * \frac{1}{1000}} = \frac{1}{\frac{1}{1000}} = 1000$$

So, we will be able to solve, in the same time that we solved on the old system, a system 1000 times that size, on our new system.

9 For each of the following pairs of functions, indicate whether the first function of each of the following pairs has a lower, same, or higher order of growth (to within a constant multiple) than the second function.

a. $n(n+1)$ and $2000n^2$ **b.** $100n^2$ and $0.01n^3$
c. $\log_2 n$ and $\ln n$ **d.** $\log_2^2 n$ and $\log_2 n^2$
e. 2^{n-1} and 2^n **f.** $(n-1)!$ and $n!$

Ans-a: Growth rate is the same, other contributing factors are constant, growth rate recorded below:

$$n^2$$

Ans-b: Growth rate is lower, growth rates recorded below:

$$n^2 < n^3$$

Ans-c: Growth rate is the same, other contributing factors are constant, growth rate recorded below:

$$log\, n$$

Ans-d: Growth rate is slower for first term, Explanation below:
The first option grows slower than the second option, because the exponent is applied to the log function itself, making it slower than a normal log. Whereas the other function is applied to the term after the logarithm has been performed, making it a faster version of log n time.

Ans-e: Growth rate is lower, Explanation below:
The two terms will end up being somewhat similar at high values of n, but the second term will always be significantly faster, even with the continually diminishing slow from subtracting 1 from the exponent in the first term.

Ans-f: Growth rate is lower, Explanation below, though it is the same as above essentially:
The two terms will end up being somewhat similar at high values of n, but the second term will always be significantly faster, even with the continually diminishing slow from the factorial being one number shorter, in the first term.

10
Invention of chess

a. According to a well-known legend, the game of chess was invented many
centuries ago in northwestern India by a certain sage. When he took his

invention to his king, the king liked the game so much that he offered the
inventor any reward he wanted. The inventor asked for some grain to be
obtained as follows: just a single grain of wheat was to be placed on the
first square of the chessboard, two on the second, four on the third, eight
on the fourth, and so on, until all 64 squares had been filled. If it took just
1 second to count each grain, how long would it take to count all the grain
due to him?
64 squares
Function for increase:

$$2^0 + 2^1 + 2^2 + 2^3 + ... + 2^{64} = \sum_{i=0}^{64} 2^i = 2^{65} - 1$$

So, it will take 2 to the 65th power, minus one, seconds to count all the grain due to the sage. Meaning the kingdom will crumble, fall, and new ones rise before someone would be finished counting.

b. How long would it take if instead of doubling the number of grains for each
square of the chessboard, the inventor asked for adding two grains?

If it's 2 grains for each 64 squares
2 x 64 = 128 seconds

Exercises 2.2:
2

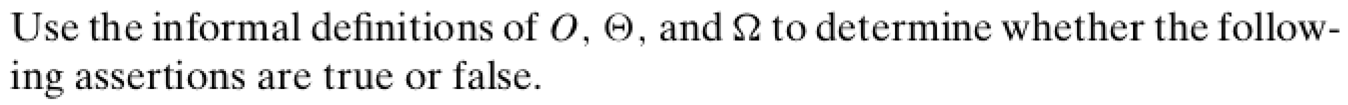

Use the informal definitions of O, Θ, and Ω to determine whether the following assertions are true or false.

TABLE 2.2 Basic asymptotic efficiency classes

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	*linearithmic*	Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

a. $n(n+1)/2 \in O(n^3)$ **b.** $n(n+1)/2 \in O(n^2)$

c. $n(n+1)/2 \in \Theta(n^3)$ **d.** $n(n+1)/2 \in \Omega(n)$

Ans-a:

False, because the term goes at a time of n to the 2nd power, meaning it is not an element of of Omega n to the 3rd power.

Ans-b:

True, because the term goes at a time of n to the 2nd power. Even with the constants causing it to run somewhat faster than n to the 2nd power usually does, it will still be part of the set of Omega n to the 2nd power.

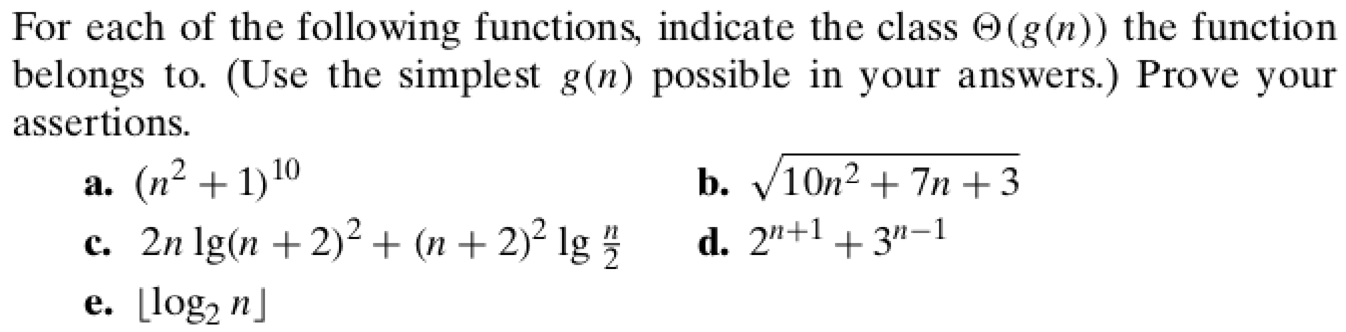
Ans-c:

False, n to the 2nd power is not equal or approaching equal to n to the 3rd power. That means that it cannot be an element of the Theta set of n to the 3rd power.

Ans-d:

False, n to the 2nd power is a worse case than allowed for by the set of all time complexities faster than n. This means that n to the 2nd power cannot be an element of the Sigma set of n

3



For each of the following functions, indicate the class $\Theta(g(n))$ the function belongs to. (Use the simplest $g(n)$ possible in your answers.) Prove your assertions.

a. $(n^2+1)^{10}$ **b.** $\sqrt{10n^2+7n+3}$

c. $2n\lg(n+2)^2+(n+2)^2\lg\frac{n}{2}$ **d.** $2^{n+1}+3^{n-1}$

e. $\lfloor \log_2 n \rfloor$

Ans-a:

The time complexity is as follows:

$$\theta(g(n^{20}))$$

Because once we multiply exponents to resolve the first term, 2 x 10, the dominant term in the function is n to the 20th. We drop all the other terms as per convention, and arrive at the above answer.

Ans-b:

The time complexity is as follows:

$$\sqrt{10n^2+7n+3}=\sqrt{10n^2}=\sqrt{n^2}=n=\theta(g(n))$$

Ans-c:

We can observe that the second term will grow slower than the first term, so we can focus our attention there to determine what the bound should be.

The time complexity is as follows:

$$2n\log{(n+2)}^2=4n\log n+2=\theta(g(n\log n))$$

Ans-d:

The time complexity is as follows:

$$\theta(g(3^{n-1}))$$

Because the second term in the function has the higher base from which the exponent grows. This means that it will grow much larger over time than our other term, and we can drop it as per our convention.

Ans-e:

The time complexity is as follows:

$$\lfloor \log_2 n \rfloor = \log_2 n = \theta(g(\log n))$$

Because the floor function will restrict things to less than or equal to the value of the interior term, we can just consider it as we would normally.

4.b

$$\log n, \quad n, \quad n \log_2 n, \quad n^2, \quad n^3, \quad 2^n, \quad n!$$

$$\log n, n, n \log_2 n, n^2, n^3, 2^n, n!$$

Prove that the functions are indeed listed in increasing order of their order of growth.

log n

$$\lim_{n->\infty} \frac{\log n}{n^k} = 0 \; For\ any\ constant\ k > 0$$

This proves that log n growth is lower than any of the growth rates listed to the right of it.

n

We can consider the above proof as valid for n being higher than log n. The next proof will tell us that it is not higher than *n log (sub 2) n* so its spot in the order is justified.

n log (sub 2) n

The first limit shows that *n log (sub 2) n* belongs in the position allocated to it. Linear (n) time is the highest growth rate to the left, and it is shown that *n log (sub 2) n* surpasses it.

$$\lim_{n->\infty} \frac{n \log_2 n}{n^k} = \infty \; For\ any\ constant\ k = 1$$

The second proof demonstrates that all exponential times, from quadratic growth onward, are higher than *n log (sub 2) n.* So its position in the list is justified.

$$\lim_{n->\infty} \frac{n \log_2 n}{n^k} = 0 \; For\ any\ constant\ k > 1$$

n^2

We know from the proof directly preceding this that quadratic growth rate is greater than *n log (sub 2) n.*

The proof below shows us that the growth rate of cubic functions and those that come after it outstrips that of quadratic functions.

$$\lim_{n->\infty} \frac{n^2}{n^k} = 0 \; For\ any\ constant\ k > 2$$

n^3

We know that cubic functions are higher than the function to its left in the listing as per the previous proof.

The proof below demonstrates that any polynomial growth is dwarfed by exponential growth.

So, we can say that the position of *n^3* is justified.

$$\lim_{n->\infty} \frac{n^k}{2^n} = 0 \; For\ any\ constant\ k > 0$$

2^n

We know that *2^n* is higher than the function to its left in the order, as shown by the previous proof.

The limit below shows us that as we approach infinity, the factorial growth rate outstrips that of exponential.

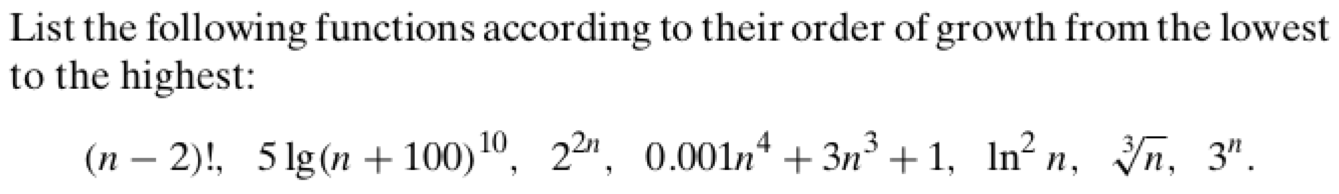
So *2^n* position in the list is justified.

$$\lim_{n->\infty} \frac{2^n}{n!} = 0$$

n!

We've shown that *n!* has a higher growth rate than all the previous functions in the list, so we can say that its position is justified.

5



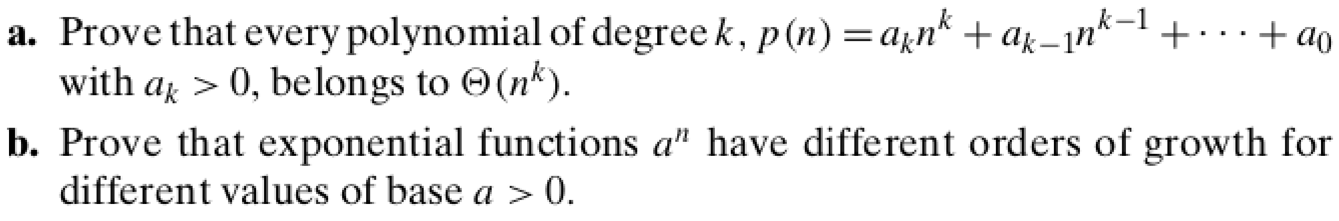
List the following functions according to their order of growth from the lowest to the highest:

$$(n-2)!, \quad 5\lg(n+100)^{10}, \quad 2^{2n}, \quad 0.001n^4 + 3n^3 + 1, \quad \ln^2 n, \quad \sqrt[3]{n}, \quad 3^n.$$

FROM LOWEST(left) TO HIGHEST(right)

$$\sqrt[3]{n},\ 5\log(n+100)^{10},\ \ln^2 n,\ (n-2)!,\ 0.001n^4 + 3n^3 + 1,\ 2^{2n},\ 3^n$$

6



a. Prove that every polynomial of degree k, $p(n) = a_k n^k + a_{k-1}n^{k-1} + \cdots + a_0$ with $a_k > 0$, belongs to $\Theta(n^k)$.

b. Prove that exponential functions a^n have different orders of growth for different values of base $a > 0$.

Ans-a:
Prove that all the polynomials of

$$a_k n^k + a_{k-1} n^{k-1} + a_{k-2} n^{k-2} + \ldots + a_0$$

Denoted throughout the rest of the problem as A for ease of writing
Belong to

$$A \in \theta(n^k)$$

To do this we can demonstrate that there are positive constants such that

$$A \leq c_1 n^k$$

AND

$$A \geq c_2 n^k$$

Which, if true, means our polynomial is both

$$A \in O(n^k) \ AND \ A \in \Omega(n^k)$$

Let us also denote the following formula

$$\mid a_k \mid + \mid a_{k-1} \mid + \ldots + \mid a_0 \mid$$

As

$$X$$

For ease of writing.

$$Proof : A \in O(n^k)$$

$$A \leq \mid a_k \mid n^k + \mid a_{k-1} \mid n^{k-1} + \ldots + \mid a_0 \mid$$

$$A \leq (X) n^k$$

$$(X) n^k = c_1 n^k$$

Proving that we have membership in the worst case set

$$Proof : A \in \Omega(n^k)$$

$$A \geq \mid a_k \mid n^k - \mid a_{k-1} \mid n^{k-1} - \ldots - \mid a_0 \mid$$

$$A \geq \mid a_k \mid n^k - (X) n^{k-1}$$

$$(X) n^k = c_1 n^k$$

Which is true because as *n* becomes very large, the value of

$$c_1 n^{k-1}$$

Becomes very small compared to

$$|a_k|n^k$$

So, we have proof that the polynomial belongs to both the best case and worst case, which means that it must also belong to the equal case.
So the equation below is true.

$$A \in \theta(n^k)$$

Ans-b:
Let's observe the performance of the following limit, where *b* is defined as:

$$b > a$$

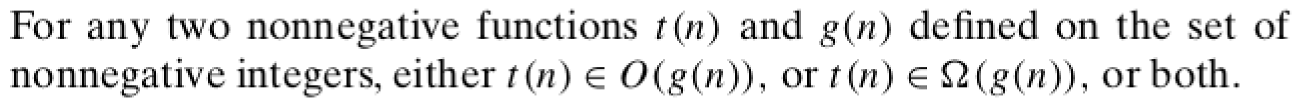
$$\lim_{n->\infty} \frac{b^n}{a^n}$$

If b is truly more than a, we can reach the conclusion that as n approaches infinity, the limit will eventually become:

$$\lim_{n->\infty} \frac{b^n}{a^n} = \infty$$

This shows that exponential functions of the form *a^n* have different orders of growth for different values of *a.*

7.d
Prove the following assertions by using the definitions of the notations involved, or disprove them by giving a specific counterexample.


For any two nonnegative functions $t(n)$ and $g(n)$ defined on the set of nonnegative integers, either $t(n) \in O(g(n))$, or $t(n) \in \Omega(g(n))$, or both.

We've already observed in the earlier problem in *2.2 - 6 - a*, that the above statement is true.
We can observe this from the basic definitions of the notations.
The statement.

$$t(n) \in O(g(n))$$

Means that the function t(n) is an element of the worst cases of g(n). That means that it can be anything higher than, or equal to g(n), in terms of time complexity.
The statement.

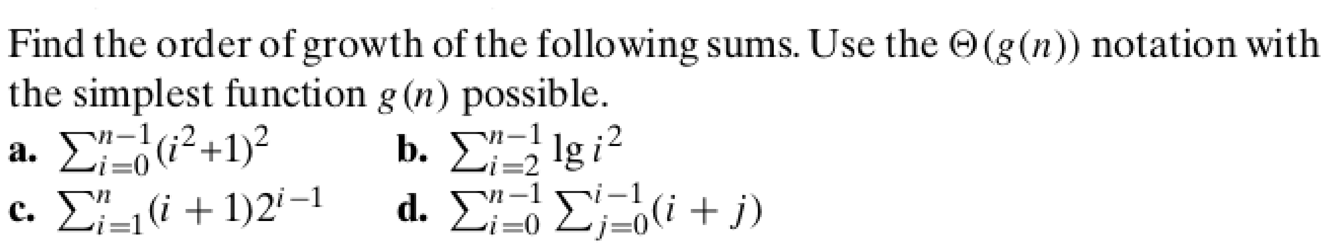
$$t(n) \in \Omega(g(n))$$

Means that the function t(n) is an element of the best cases of g(n). That means that it can be anything lower than, or equal to g(n), in terms of time complexity.

So, t(n) can be higher than, lower than, or equal in terms of time complexity, to g(n).
This shows that the statement at the beginning of the problem, is true.

Exercises 2.3:

2


Find the order of growth of the following sums. Use the $\Theta(g(n))$ notation with the simplest function $g(n)$ possible.

a. $\sum_{i=0}^{n-1}(i^2+1)^2$ **b.** $\sum_{i=2}^{n-1}\lg i^2$

c. $\sum_{i=1}^{n}(i+1)2^{i-1}$ **d.** $\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}(i+j)$

Ans-a:

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

Drop the lower order terms so we can find the tightest possible bound.
The next equation should be roughly equal to:

$$\sum_{i=0}^{n-1} i^4 = \frac{n(n-1)(2n-1)(3n^2+3n+1)}{30}$$

So we can find that the most significant term creates the following order of growth:

$$\Theta(g(n^5))$$

Ans-b:

$$\sum_{i=0}^{n-1}\log_{10} i^2 = \sum_{i=0}^{n-1} 2\log_{10} i = \Theta(g(\log n))$$

Ans-c:
We can drop the first term, as it's essentially a constant. We then simplify and arrive at our final answer.

$$\sum_{i=0}^{n-1}(i+1)2^{i-1} = \sum_{i=0}^{n-1}2^{i-1} = 2^{-1}+2^0...2^{n-2} = 2^{n-1}-1 = \Theta(g(2n^{n-1}))$$

Ans-d:

$$\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}(i+j)$$

Evaluate the inner sum first to simplify things

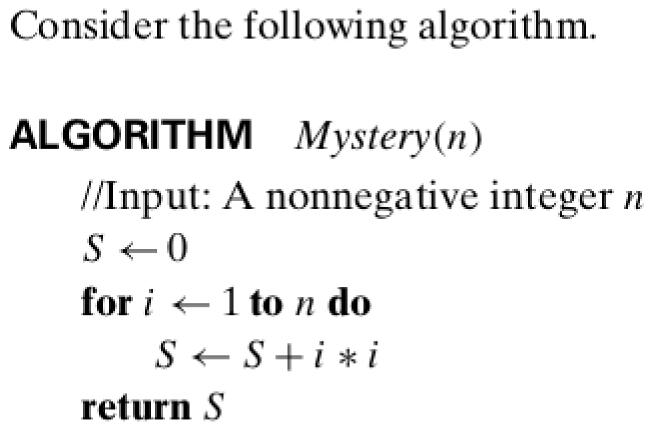
$$\sum_{j=0}^{i-1}(i+j) = \frac{i(i-1)}{2} + \frac{i(i+1)}{2} = i^2$$

So we can rewrite our starting equation to be:

$$\sum_{i=0}^{n-1} i^2 = \frac{n(n-1)(2n-1)}{6} = \Theta(g(n^3))$$

And once we've simplified everything and dropped our constants we've got the final solution.

4



Consider the following algorithm.

```
ALGORITHM  Mystery(n)
    //Input: A nonnegative integer n
    $S \leftarrow 0$
    for $i \leftarrow 1$ to $n$ do
        $S \leftarrow S + i * i$
    return $S$
```

a. What does this algorithm compute?

Ans-a:
The algorithm computes the sum of the squares from one to n.

b. What is its basic operation?

Ans-b:
The basic operation of the algorithm is 1 multiplication and 1 addition operation.

c. How many times is the basic operation executed?

Ans-c:
The basic operation (1 mult. and 1 add.), occurs a total of n times. With n being the number of iterations of the loop.

d. What is the efficiency class of this algorithm?

Ans-d:
Linear

e. Suggest an improvement, or a better algorithm altogether, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.

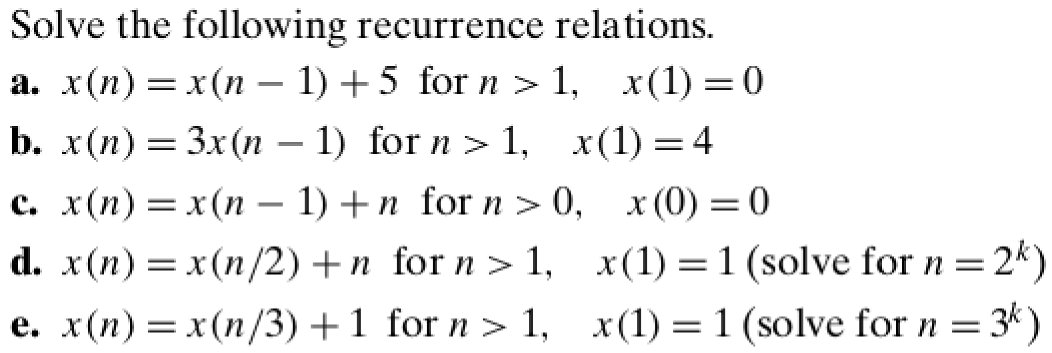
Ans-e:

The sum of the squares, as used in other problems above, has a distinct formula. If we use that formula, instead of running through the loop, we can squish the time down to constant.

```
ALGORITHM better(n)
//Input: A non-negative integer n
return n * (n+1) * (2*n+1)/6
```

Exercises 2.4:

1



Solve the following recurrence relations.

a. $x(n) = x(n-1) + 5$ for $n > 1$, $x(1) = 0$

b. $x(n) = 3x(n-1)$ for $n > 1$, $x(1) = 4$

c. $x(n) = x(n-1) + n$ for $n > 0$, $x(0) = 0$

d. $x(n) = x(n/2) + n$ for $n > 1$, $x(1) = 1$ (solve for $n = 2^k$)

e. $x(n) = x(n/3) + 1$ for $n > 1$, $x(1) = 1$ (solve for $n = 3^k$)

Ans-a:

$$x(n) = x(n-1) + 5 for\ n > 1, x(1) = 0$$

$$x(n) = a(n-1) + b + 5 = an - a + b + 5 =$$

Solved for a, yields:

$$a = 1$$

Set up another equation using this fact and that x(1)=0 to solve for b:

$$0 = 1 + b\ ,\ b = -1$$

So we can conclude that

$$x(n) = n - 1$$

Ans-b:

$$x(n) = 3x(n-1) for\ n > 1, x(1) = 4$$

Since we can see a geometric progression in the term above, we can set up our equation in the form of:

$$x(n) = x(1) * 3^{n-1}$$

Which we sub x(1) into for our answer:

$$x(n) = 4 * 3^{n-1}$$

Ans-c:

$$x(n) = x(n-1) for\ n > 0, x(1) = 0$$

We can see that x(n) is always going to be = to its previous value, so we can simply state the recurrence relations as:

$$x(n) = 0$$

Ans-d:

$$x(n) = x(n/2) + n for\ n > 1, x(1) = 1$$

We want to solve for *n=2^k* so we need to fit that into our equation.

$$x(2^k) = x(\frac{2^k}{2}) + 2^k = x(2k^{k-1}) + 2^k$$

We want to repeat this relation until the end, which goes along the lines of:

$$x(2^k) = x(2k^{k-1}) + 2^k,\ x(2^{k-1}) = x(2k^{k-2}) + 2^{k-1},\ x(2^{k-2}) = x(2k^{k-3}) + 2^{k-2}$$

Which eventually leads us to:

$$x(2) = x(1) + 2 = 1 + 2$$

Now that we've fit in x(1) and we have a complete equation on the right side, we can find the pattern for *2^k* according to this model:

$$x(2^k) = 1 + 2 + 2^2 + ... + 2^k$$

We can interpret this as a geometric series with the formula:

$$S_n = \frac{a(r^n - 1)}{r - 1}$$

Plugging in the equivalents from our function to the formula yields the following equation:

$$x(2^k) = \frac{1(2^{k+1} - 1)}{2 - 1} = 2^{k+1} - 1$$

Which gives us our final answer for the recurrence relation.

Ans-e:

$$x(n) = x(n/3) + 1 for\ n > 1, x(1) = 1$$

We want to solve for *n=3^k* so we need to fit that into our equation.

$$x(3^k) = x(\frac{3^k}{3}) + 1 = x(3k^{k-1}) + 1$$

We want to repeat this relation until the end, which goes along the lines of:

$$x(3^k)=x(3k^{k-1})+1,\ x(3^{k-1})=x(3k^{k-2})+1,\ x(3^{k-2})=x(3k^{k-3})+1$$

Which eventually leads us to:

$$x(3)=x(1)+1=1+1$$

Now that we've fit in x(1) and we have a complete equation on the right side, we can find the pattern for *2^k* according to this model:

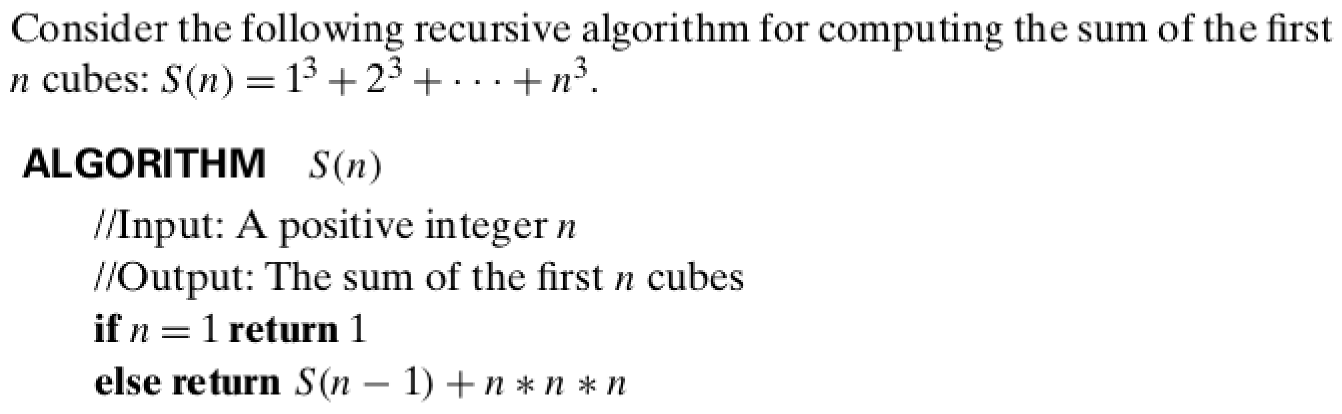
$$x(3^k)=1+1+1...+1$$

There are *k+1* terms in the sum, so we can write it as:

$$x(3^k)=k+1$$

Which is the solution to the recurrence relation.

3



Consider the following recursive algorithm for computing the sum of the first n cubes: $S(n)=1^3+2^3+\cdots+n^3$.

```
ALGORITHM   $S(n)$
    //Input: A positive integer $n$
    //Output: The sum of the first $n$ cubes
    **if** $n=1$ **return** 1
    **else return** $S(n-1)+n*n*n$
```

a. Set up and solve a recurrence relation for the number of times the algorithm's basic operation is executed.

Ans-a:

`Note, I only realized that it should have technically been called S(n) while reviewing, I don't want to go through and type all the changes for that though.`

First we establish the basic operation

In this case, we can say that the basic operation is 1 addition and 2 multiplication.

We can define the number of times that our basic operation occurs as *T(n)*

The recurrence relation can then be defined as:

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

For our base case, we have T(1)=1, because we perform one basic operations when n=1.

To solve the recurrence relation, we can use the same method that we've used earlier, and take it out to the end.

$$T(n) = T(n-1) + 1 = T(n-2) + 2 = T(n-3) + 3 = \ldots = T(1) + (n-1) = n$$

So we can see that the solution of the recurrence relation is linear. That means we've got a linear time complexity on our hands.

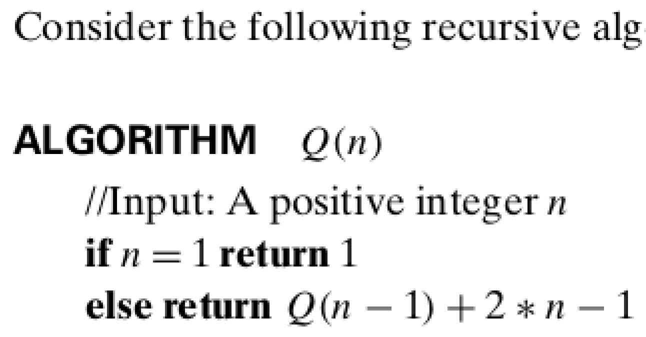
b. How does this algorithm compare with the straightforward non-recursive algorithm for computing this sum?

Ans-b:
We can compare the two algorithms pretty directly, as we've already calculated that a non-recursive solution to this problem using a for loop will also operate on linear time.

However, we should take into account the additional overhead that is given to recursive algorithms. If the system carves out more memory space for the recursive version due to the continual recursive calls, it may end up being more costly.

We could also end up implementing the solution in constant time if we apply a mathematical function that essentially skips the time intensive solution. Like how euclid's algorithm saves an enormous amount of time when finding the GCF.

4



Consider the following recursive algorithm.

```
ALGORITHM  $Q(n)$
    //Input: A positive integer $n$
    if $n = 1$ return 1
    else return $Q(n-1) + 2 * n - 1$
```

a. Set up a recurrence relation for this function’s values and solve it to determine what this algorithm computes.

Ans-a:
We can define our relation as

$$T(n) = 1,\ When\ n = 1$$

And

$$T(n) = T(n-1) + 2n - 1,\ n > 1$$

We start with the second equation:

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

Isolate *2n-1*

$$T(n) - T(n-1) = 2n - 1$$

This allows us to set up a sum for both sides, we can simplify the left easily, but the right will take more time:

$$T(n) - T(1) = \sum_{k=2}^{n} 2k - 1$$

We're going up from k=2, because we are considering only instances up from 1

$$T(n) = T(1) + \sum_{k=2}^{n} 2k - 1$$

Bring our other term back over, then we can work on the sum knowing that *T(1)=1*

$$T(n) = 1 + \sum_{k=2}^{n} 2n - 1 = 1 + 2\sum_{k=2}^{n} k - \sum_{k=2}^{n} 1$$

We can simplify the nested sums further into a more workable equation.

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

Which after performing all our operations and the final simplification, yields us:

$$T(n) = n^2$$

So, we have the solution to our recurrence relation.

b. Set up a recurrence relation for the number of multiplications made by this algorithm and solve it.

Ans-b:
Let the number of multiplication operations made by this algorithm be *M(n)*
We know that every recursive call only causes 1 multiplication to occur.
No multiplication occurs at our base case.
So we can say that the number of multiplication operations is:

$$M(n) = n - 1$$

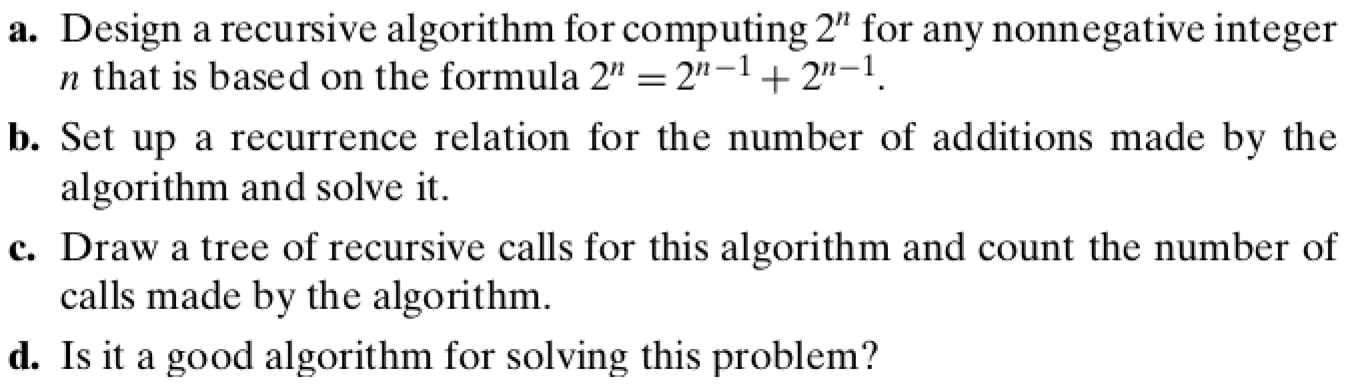
c. Set up a recurrence relation for the number of additions/subtractions made by this algorithm and solve it.

Ans-c:
Let the number of additions and subtractions be *AS(n)*
We know that every recursive call causes 1 addition and 1 subtraction.

No addition or subtractions occurs at base case.
So, we can say that the number of additions and subtractions is:

$$AS(n) = 2n - 2$$

8



a. Design a recursive algorithm for computing 2^n for any nonnegative integer n that is based on the formula $2^n = 2^{n-1} + 2^{n-1}$.

b. Set up a recurrence relation for the number of additions made by the algorithm and solve it.

c. Draw a tree of recursive calls for this algorithm and count the number of calls made by the algorithm.

d. Is it a good algorithm for solving this problem?

Ans-a:

```
ALGORITHM recurse(n)
if n = 0 return 1
else return 2 * recurse(n-1)
```

Ans-b:

Let *T(n)* define the number of addition operations made for computing the *2^n*

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

I've written it like this because the full cycle takes *T(n-1)* and then it does one more addition to combine everything at the end.

Ans-c:

Note: I did it this way because I don't really want to try drawing all of that out and then appending the image to this. The file's already going to be big enough.

```
recurse(4)
├── recurse(3)
│ ├── recurse(2)
│ │ ├── recurse(1)
│ │ │ ├── recurse(0)
│ │ │ │ └── returns 1
│ │ │ └── returns 2
│ │ └── recurse(1)
│ │ ├── recurse(0)
│ │ │ └── returns 1
│ │ └── returns 2
│ └── recurse(2)
│ ├── recurse(1)
│ │ ├── recurse(0)
│ │ │ └── returns 1
│ │ └── returns 2
│ └── recurse(1)
│ ├── recurse(0)
│ │ └── returns 1
│ └── returns 2
└── recurse(3)
├── recurse(2)
│ ├── recurse(1)
│ │ ├── recurse(0)
│ │ │ └── returns 1
│ │ └── returns 2
│ └── recurse(2)
│ ├── recurse(1)
│ │ ├── recurse(0)
│ │ │ └── returns 1
│ │ └── returns 2
│ └── recurse(1)
│ ├── recurse(0)
│ │ └── returns 1
│ └── returns 2
└── recurse(2)
├── recurse(1)
│ ├── recurse(0)
│ │ └── returns 1
│ └── returns 2
└── recurse(1)
├── recurse(0)
│ └── returns 1
└── returns 2
```

The total number of recursive calls made by the algorithm should be:

$$2^{n+1} - 1$$

Ans-d:
The algorithm is correct and it's simple to write, but it isn't really a great way of solving the problem while effectively using our time and resources. Implementing this the iterative way would be better, in my opinion.

Exercises 2.5:

3

Climbing stairs

Find the number of different ways to climb an n-stair staircase if each step is either one or two stairs. For example, a 3-stair staircase can be climbed three ways: 1-1-1, 1-2, and 2-1.

Ans:

We can define this problem as a recurrence relation using *T(n)* as the total number of different ways to climb the steps given *n* number of steps.

$$T(n) = T(n-1) + T(n-2) \ldots T(1)$$

We can observe that this, is the fibonacci sequence represented as a recurrence relation. To solve for any particular n, there is, thankfully, a single formula we can implement.

Binet's formula is as follows:

$$T(n) = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n(\frac{1-\sqrt{5}}{2})^n)$$