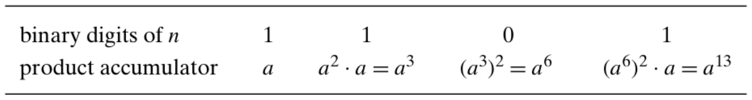
Jordan Nakamoto
415 Algorithms

Assignment on Binary Exponentiation, Dynamic Programming, and Hashing

1.

a. From left to right the rules are (Levin): (b − 1) ≤ M(n) ≤ 2(b − 1),



binary digits of n	1	1	0	1
product accumulator	a	$a^2 \cdot a = a^3$	$(a^3)^2 = a^6$	$(a^6)^2 \cdot a = a^{13}$

Where the initial digit doesn't perform any action….

product ← product ∗ product

If b_i =1 product ←product ∗a. (Levin)

Therefore,

For **a^{11}, a^{12}, a^{13}, and a^{14}**

1011, 1100, 1101, 1110 -> 4, 3, 4, 4.

From right to left the rule is:
for i ← 1 **to** l **do** term ← term ∗ term **if** b_i = 1 product ← product ∗ term

1011, 1100, 1101, 1110 -> 5, 5, 5, 6. Since we need to always square on each step

b. 3^{13} 3. 3^2 = 9 * 3 = 27. **27*27** = 729. **729* 729** = **531441 * 3** = 1594323
Bolded calculator multiplications

c. 3^{13} 3. 3^2 = 9. 9^2 = 81 * 3 = 243. **81 * 81** = **6561 * 243** = 1594323

2.

b. The function computes the value of the nth fibonacci number, so it's best to just pass only the nth number and have the helper function manage the memory required for computation.

c. Arr holds the value of each number in the fibonacci sequence, initialized at 0 and populated recursively with dynamic programming.

d. If it were valid c syntax, this assignment/declaration would only initialize the first element.

e. When each level on the recursive stack is returned back to and ready for computation, rather than recomputing a whole stack down to the base case, the result of the dependency calls have already had their results stored at their place which means at fibArr[n] we can just grab the operands n-1 and n-2 as stored values.

f. This is the return when we reach the deepest recursive call base case when fibArr[2] is assigned n-1 + n-2 which are fibArr[1] and fibArr[0] respectively. Since these terminate the recursion with the values 1 + 1, We can begin returning back up the call stack.

3.

b. Although the program exits at 17!, the largest factorial that can be computed is 12!. We recognize this by seeing that 13! does not end in right-most zero where 13 > 12, where 12 ends in a 0.

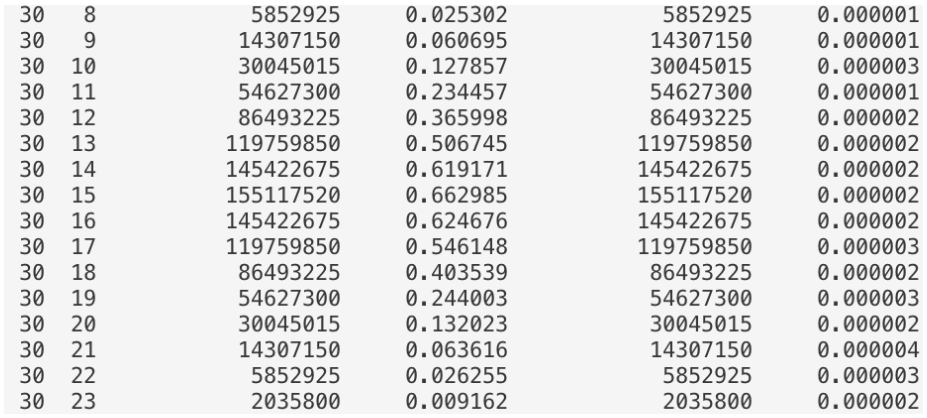
b. With the change to a long, the program can accurately calculate factorials up to 20!

4.

Once your functions work properly, add code to determine the timing of each of these approaches. Write a brief commentary to indicate the extent to which the table in the second approach increases efficiency.

From your work in Problem 3, you know values of n for which the computation of n! fails, so the expression n! / [k! (n-k)!] fails. Determine what, if any, larger values of C(n, k) can be computed with your current program.

b.



```
30   8          5852925     0.025302          5852925     0.000001
30   9         14307150     0.060695         14307150     0.000001
30  10         30045015     0.127857         30045015     0.000003
30  11         54627300     0.234457         54627300     0.000001
30  12         86493225     0.365998         86493225     0.000002
30  13        119759850     0.506745        119759850     0.000002
30  14        145422675     0.619171        145422675     0.000002
30  15        155117520     0.662985        155117520     0.000002
30  16        145422675     0.624676        145422675     0.000002
30  17        119759850     0.546148        119759850     0.000003
30  18         86493225     0.403539         86493225     0.000002
30  19         54627300     0.244003         54627300     0.000003
30  20         30045015     0.132023         30045015     0.000002
30  21         14307150     0.063616         14307150     0.000004
30  22          5852925     0.026255          5852925     0.000003
30  23          2035800     0.009162          2035800     0.000002
```

Execution times for n = 30 and k = 8-23 shown. On the right side the Dynamic Programming approach shows that we are able to reduce the computational runtime to only operate using the preceding array row of existing values, which means the addition operations should be constant and linearly increasing as the number of coefficients increases for iterating over (n,k). It should resemble Pascal’s Triangle, utilizing sums of previous values to compute n and k where k = n/2 requires the most coefficient terms.

c. Because the coefficients are summed together to represent a factorial, we don’t need to directly store n!, before dividing by k!(n!-k)!. Instead the result is computed by the summation of intermediary values, which themselves have to be stored, and spread out across more sub functions. The center of the set of coefficients if visualized through Pascal’s Triangle would be where the overflow occurs.

```
  66  44                   0     0.000001 182183167981760400     0.000015
  66  45                   0     0.000000  89067326568860640     0.000014
  67  25                   0     0.000001 1673497616379701112     0.000011
  67  26                   0     0.000001 2703342303382594104     0.000010
  67  27                   0     0.000001 4105075349580976232     0.000012
  67  28                   0     0.000000 5864393356544251760     0.000011
  67  29                   0     0.000002 7886597962249166160     0.000012
  67  30                   0     0.000000 -8457053321527274480     0.000011
  67  31                   0     0.000001 -6523564788846833744     0.000011
```

The first negative value computed representing overflow occurs at C(67,30), where we expect C(67,33) C(67,34) to represent the center of the triangle - signifying that 66 may be the highest n value supported.

0	SL
1	
2	
3	UQ
4	QR
5	GD
6	QS
7	DT
8	MR
9	
10	WZ
11	NW
12	CY
13	EM
14	NZ
15	AX

0	
1	MR
2	
3	UQ
4	QR
5	QD
6	QS
7	DT
8	
9	CY
10	WZ
11	NW
12	NZ
13	EM
14	SL
15	AX

0	⇒	
1	⇒	
2	⇒	
3	⇒	UQ
4	⇒	GR, GD, MR
5	⇒	
6	⇒	QS
7	⇒	DT
8	⇒	
9	⇒	
10	⇒	WZ, CY
11	⇒	NW,NZ
12	⇒	
13	⇒	EM
14	⇒	
15	⇒	AX, SL