CS2214 – Assignment Five

Dat Vo – 250983323

January 31, 2020

1 Question One

Find the formula for the summation: $\sum_1^n i(i+1)$.
If $\sum_1^n i^2 = \frac{n(n+1)(2n+1)}{6}$ and $\sum_1^n i = \frac{n(n+1)}{2}$.
Then,

$$\begin{aligned}
\sum_1^n i(i+1) &= \sum_1^n i^2 + \sum_1^n i \\
&= \frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2} \\
&= \frac{(n^2+n)(2n+1) + 3n(n+1)}{6} \\
&= \frac{2n^3 + n^2 + 2n^2 + n + 3n^2 + 3n}{6} \\
&= \frac{2n^3 + 6n^2 + 4n}{6} \\
&= \frac{n^3 + 3n^2 + 2n}{3}
\end{aligned}$$

$\therefore$ the formula for $\sum_1^n i(i+1)$ is $\frac{n^3+3n^2+2n}{3}$.

2 Question Two

Prove that the number of leaves of in a binary tree is 2^h and the size of the tree is $2^{h+1} - 1$.
Let $L(h) = 2^h$ represent a function expressing the number of leaves and $S(h) = 2^{h+1} - 1$ represent a function expressing the size of the binary tree based on the number of nodes.

2.1 Prove: $L(h) = 2^h$

If $h = 0$ then,

$$L(0) = 2^0 = 1$$

which makes $L(h)$ hold for the base case.
Suppose $L(h)$ holds for all values of h upto some $k, k \geq 1$.
So, from the root where $h = 0$, subsequently each internal node where $h \geq 0$ will have 2^h leaves. Then,

$$\begin{aligned}
L(h) &= 2^h \\
L(0) &= 2^0 = 1 \\
L(1) &= 2^1 = 2 \\
L(2) &= 2^2 = 3 \\
&\vdots \\
L(k) &= 2^k \\
L(k+1) &= 2^k + 2^k \\
&= 2(2^k) \\
&= 2^{k+1}
\end{aligned}$$

$\therefore$ by induction $L(h) = 2^h$.

2.2 Prove: $S(h) = 2^{h+1} - 1$

If $h = 0$ then,

$$S(0) = 2^{0+1} - 1 = 1$$

$\therefore$ The size of the tree is 1 which holds for the base case.
Now suppose $S(h)$ holds for all values of h upto some $k, k \geq 1$.

$$\begin{aligned}
S(h) &= 2^{h+1} - 1 \\
S(0) &= 2^{0+1} - 1 = 1 \\
S(1) &= 2^{1+1} - 1 = 3 \\
S(2) &= 2^{2+1} - 1 = 5 \\
&\vdots \\
S(k) &= 2^{k+1} - 1
\end{aligned}$$

And a binary tree of size k has 2 binary trees of height $k - 1$ plus the root;

$$\begin{aligned}
S(k) &= 2 \times S(k-1) + 1 \\
S(k) &= 2 \times (2^{(k-1)+1} - 1) + 1 \\
S(k) &= 2 \times 2^k - 2 + 1 \\
S(k) &= 2^{k+1} - 1
\end{aligned}$$

$\therefore$ by induction $S(h) = 2^{h+1} - 1$.

3 Question Three

Let $A[1..n]$ be an array of n distinct numbers.If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an **inversion** of A.

3.1 List inversions of $[2, 3, 8, 6, 1]$.

The inversions are: (1,5),(2,5),(3,5),(4,5),(3,4)

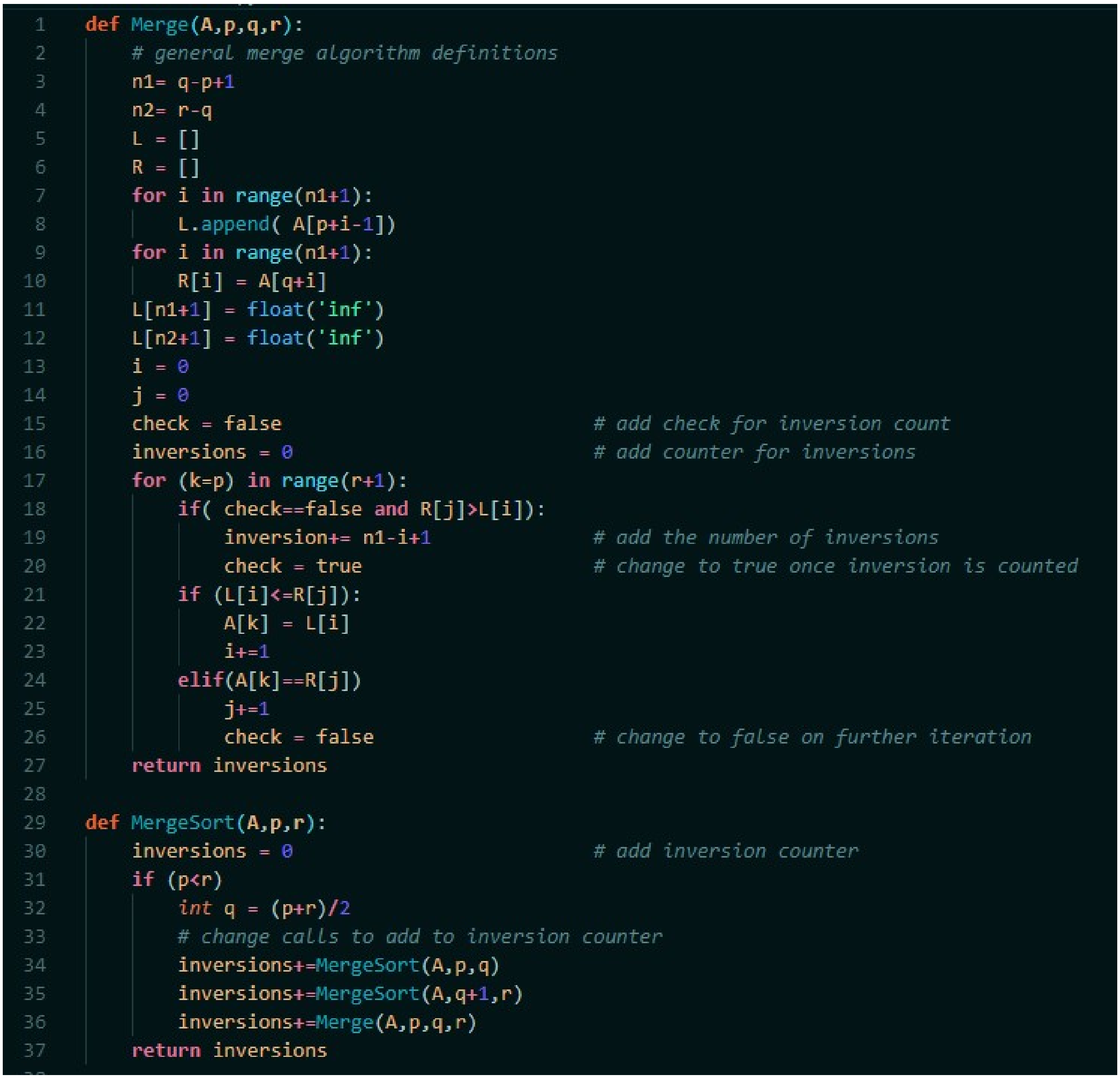
3.2 Array with most inversions from set $[1, 2, ..., n]$.

The array is $[n, n - 1, n - 2, ..., 2, 1]$.
For every element in this array there is an inversion with every other element that proceeds the element's index. The number of inversions is thus, $\frac{n(n-1)}{2}$.

3.3 Relationship between insertion sort and the number of inversions.

For each element in the unsorted array, the for loop will encounter each element that proceeds A[i] whereas the while loop will check elements that are greater than A[i] thus each successful entry into the while loop will encounter an inversion between (key,A[i]). Once the while loop has been entered it will correct every inversion found and thus there will be 1 less inversion after every iteration of the while loop.

3.4 Algorithm for number of inversions in any permutation.



```
1   def Merge(A,p,q,r):
2       # general merge algorithm definitions
3       n1= q-p+1
4       n2= r-q
5       L = []
6       R = []
7       for i in range(n1+1):
8           L.append( A[p+i-1])
9       for i in range(n1+1):
10          R[i] = A[q+i]
11      L[n1+1] = float('inf')
12      L[n2+1] = float('inf')
13      i = 0
14      j = 0
15      check = false                           # add check for inversion count
16      inversions = 0                          # add counter for inversions
17      for (k=p) in range(r+1):
18          if( check==false and R[j]>L[i]):
19              inversion+= n1-i+1              # add the number of inversions
20              check = true                    # change to true once inversion is counted
21          if (L[i]<=R[j]):
22              A[k] = L[i]
23              i+=1
24          elif(A[k]==R[j])
25              j+=1
26              check = false                   # change to false on further iteration
27      return inversions
28
29  def MergeSort(A,p,r):
30      inversions = 0                          # add inversion counter
31      if (p<r)
32          int q = (p+r)/2
33          # change calls to add to inversion counter
34          inversions+=MergeSort(A,p,q)
35          inversions+=MergeSort(A,q+1,r)
36          inversions+=Merge(A,p,q,r)
37      return inversions
```

Since the only operation that gets added is an if statement (line 18) which would take O(c) time to run prior to the original operations in the original mergesort. This algorithm should also take $\Theta(n\lg(n))$ time as well.

4 Question Four

Whether A is order of B. y = yes. n = no.

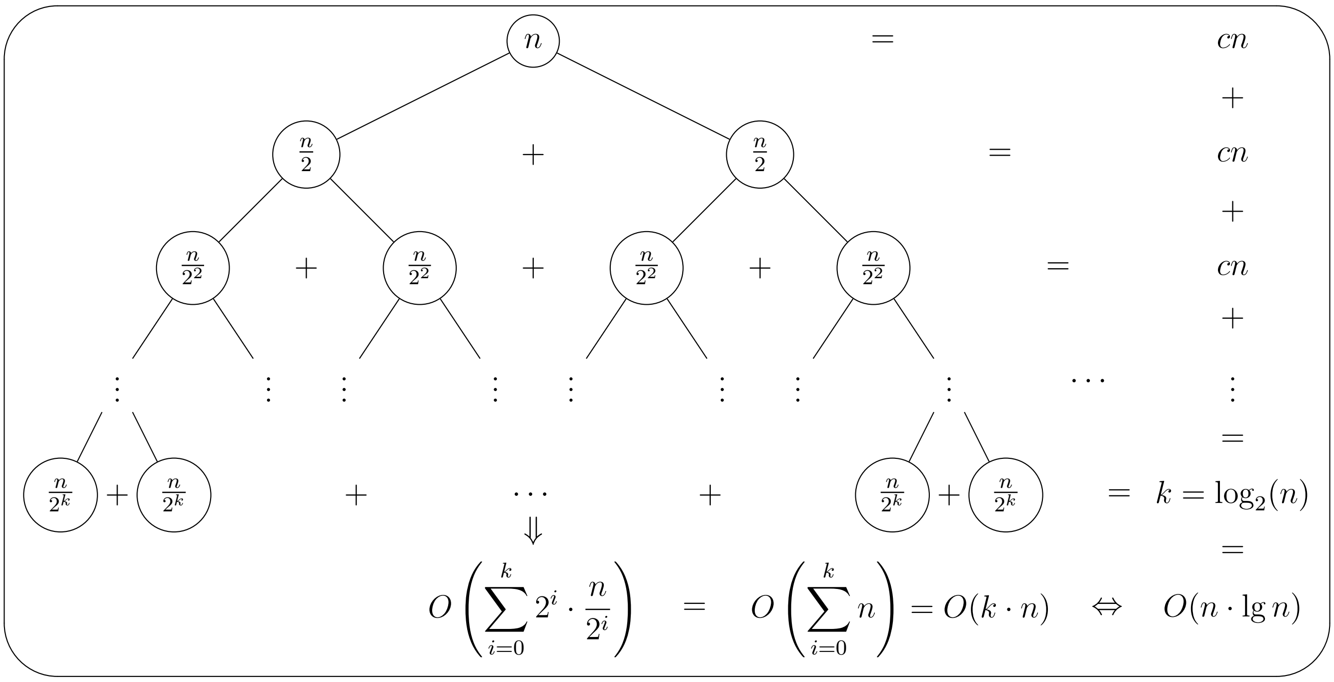
	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ϵ	y	y	n	n	n
b.	n^k	c^n	y	y	n	n	n
c.	$\sqrt{n}$	$n\sin n$	n	n	n	n	n
d.	2^n	$2^{\frac{n}{2}}$	y	n	n	y	n
e.	$n^{\lg(c)}$	$c^{\lg(n)}$	y	n	y	n	y
f.	$\lg(n!)$	$\lg(n^n)$	y	n	y	n	y

5 Question 5

Determine the time complexity of

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

Using a Recursion Tree: (*tree template taken from overleaf)


n
$= cn$
$\frac{n}{2}$ + $\frac{n}{2}$
$= cn$
$\frac{n}{2^2}$ + $\frac{n}{2^2}$ + $\frac{n}{2^2}$ + $\frac{n}{2^2}$
$= cn$
$\frac{n}{2^k}$ + $\frac{n}{2^k}$ + $\cdots$ + $\frac{n}{2^k}$ + $\frac{n}{2^k}$
$= k = \log_2(n)$
$\Downarrow$

$$O\left(\sum_{i=0}^{k} 2^i \cdot \frac{n}{2^i}\right) = O\left(\sum_{i=0}^{k} n\right) = O(k \cdot n) \Leftrightarrow O(n \cdot \lg n)$$

With every node accounting for 2 children. Each subsequent child will have n/2 time complexity of the previous node. Eventually reaching a height of k, which will yield that $n/2^k = 1$ as a stopping condition.

$$\frac{n}{2^k} = 1$$
$$n = 2^k$$
$$\lg_2(n) = k\lg_2(2)$$
$$\therefore k = \lg_2(n)$$

Additionally, with the number of Leaves as 2^k (proven earlier in 2.1) we can write the number of nodes as;

$$O(n) = \sum_{i=0}^{k} Leaves(i) \times \frac{n}{2^i}$$
$$= \sum_{i=0}^{k} n$$
$$= kn$$

then if $k = \lg_2(n)$;

$$O(n) = kn$$
$$O(n) = n\lg_2(n)$$

Thus proving that $T(n) \leq 2T(\frac{n}{2}) + n\log(n)$.

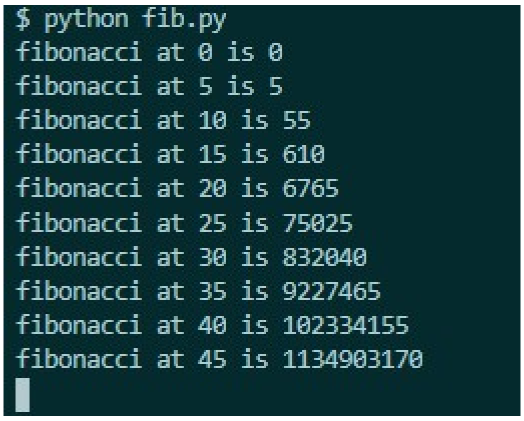
6 Question Six

6.1 Algorithm A output

Algorithm The most simple fibonacci algorithm will take the input n and create two sets of operations on n-1 and n-2 to recursively until n becomes either 1 or 0 as the base case. the call frames will then stack upon the result the last frame made to generate the final fibonacci number indexed by n in the sequence.

Correctness The algorithm is correct because each fibonacci number is a result of the sum of the last two indexes of the that number (n-1 and n-2). Therefore since this algorithm recurrently takes into account this fact it should produce the correct indexed fibonacci number.

Time Complexity The algorithm takes Fibonacci(n-1) + Fibonacci(n-2) time to execute to generate the index of each n-1 and n-2 fibonacci number. In doing so, it creates a successive amount of calls to generate n-3, n-4,... and so forth until Fibonacci(0). If each call were to take $O(2^n)$ the time complexity of this algorithm would be $T(n) = O(2^{n-1}) + O(2^{n-2}) + c$ which would make this algorithm have a time complexity of $O(2^n)$.

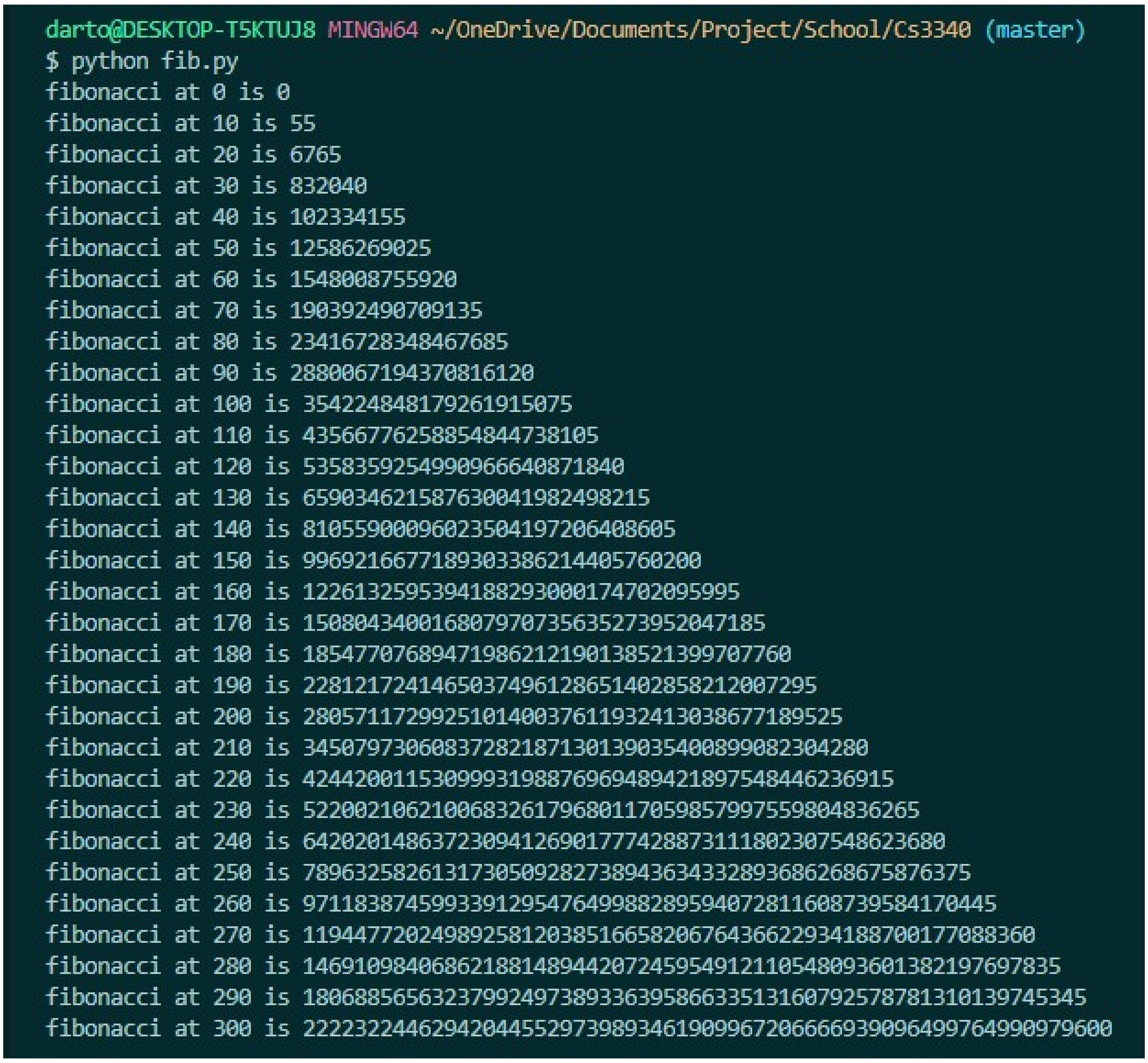

```
$ python fib.py
fibonacci at 0 is 0
fibonacci at 5 is 5
fibonacci at 10 is 55
fibonacci at 15 is 610
fibonacci at 20 is 6765
fibonacci at 25 is 75025
fibonacci at 30 is 832040
fibonacci at 35 is 9227465
fibonacci at 40 is 102334155
fibonacci at 45 is 1134903170
```

6.2 Algorithm B output

Algorithm this algorithm will use an array of length 2, (arr = [0,1]) to calculate the fibonacci number at index n. To do so, the arr[0] or index zero will store the n-2 fibonacci number and arr[1] index 1 will store the n-1 fibonacci number at the start. Then each subsequent call will push the n-1 fibonacci number to arr[0] and store the current fibonacci number at arr[1]. Which will cause arr[1] to store the final n index needed.

Correctness Similar to the reason defined in the previous algorithm, the current algorithm will instead use less call frames. This algorithm uses only the last fibonacci number or n-1 to calculate the current once with the previous already being factored into the previous call frame.

Time Complexity Since not all combinations of the n-1 or n-2 are needed, and this algorithm only uses n-1 for each call, there is not 2^n calls. Instead, this algorithm will still take $O(n)$ time to complete for each call of the fibonacci sequence to n-1, n-2, ... , 1,0 to calculate the previous number, however only the sum of n-1 and n-2 is needed for each subsequent number and will therefore also run in $O(A(n))$ where A(n) is the complexity of adding F(n-1) and F(n-2). Therefore, $O(nA(n))$.



```
darto@DESKTOP-T5KTUJ8 MINGW64 ~/OneDrive/Documents/Project/School/Cs3340 (master)
$ python fib.py
fibonacci at 0 is 0
fibonacci at 10 is 55
fibonacci at 20 is 6765
fibonacci at 30 is 832040
fibonacci at 40 is 102334155
fibonacci at 50 is 12586269025
fibonacci at 60 is 1548008755920
fibonacci at 70 is 190392490709135
fibonacci at 80 is 23416728348467685
fibonacci at 90 is 2880067194370816120
fibonacci at 100 is 354224848179261915075
fibonacci at 110 is 43566776258854844738105
fibonacci at 120 is 5358359254990966640871840
fibonacci at 130 is 659034621587630041982498215
fibonacci at 140 is 81055900096023504197206408605
fibonacci at 150 is 9969216677189303386214405760200
fibonacci at 160 is 1226132595394188293000174702095995
fibonacci at 170 is 150804340016807970735635273952047185
fibonacci at 180 is 18547707689471986212190138521399707760
fibonacci at 190 is 2281217241465037496128651402858212007295
fibonacci at 200 is 280571172992510140037611932413038677189525
fibonacci at 210 is 34507973060837282187130139035400899082304280
fibonacci at 220 is 4244200115309993198876969489421897548446236915
fibonacci at 230 is 522002106210068326179680117059857997559804836265
fibonacci at 240 is 64202014863723094126901777428873111802307548623680
fibonacci at 250 is 7896325826131730509282738943634332893686268675876375
fibonacci at 260 is 971183874599339129547649988289594072811608739584170445
fibonacci at 270 is 119447720249892581203851665820676436622934188700177088360
fibonacci at 280 is 14691098406862188148944207245954912110548093601382197697835
fibonacci at 290 is 1806885656323799249738933639586633513160792578781310139745345
fibonacci at 300 is 222232244629420445529739893461909967206666939096499764990979600
```

Algorithm B is remarkably faster than algorithm A. When algorithm A got to fibonacci at 40 it started taking minutes longer whereas alorithm B got through 300 fibonaccis in a second.
** C++ results in typescript and cpp and makefile in folder