University of Houston

Assignment 1: Programming & Theory

COSC 3320
Algorithms and Data Structures

Dr. Ernst L. Leiss

Carlos Flores

February 3, 2020

Computer Science Department

Problem 1

Hanoi Algorithm Design, Complexity, & Implementation Analysis

Implement an algorithm and program it to solve a Tower of Hanoi puzzle for the following graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with the Vertices, $\mathcal{V}$ = {Start, Aux 1, Aux 2, Aux 3, Aux 4, and Dest}, and Directed Edges $\mathcal{E}$ = {(Start, Aux 1), (Aux 1, Aux 2), (Aux 2, Aux 3), (Aux 3, Aux 4), (Aux 4, Aux 1), and (Aux 1, Dest)}.

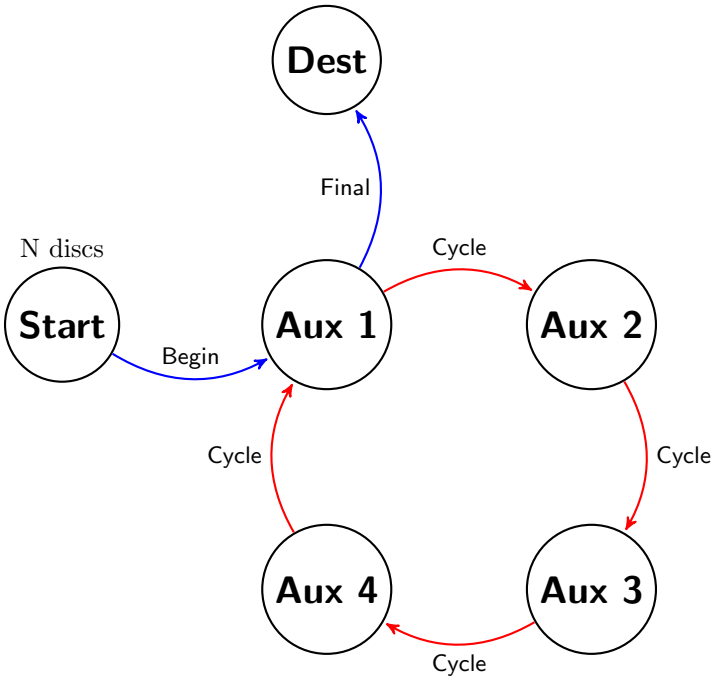


Figure 1.a Tower of Hanoi puzzle for $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.

Design Rationale

Our goal is to move a stack of N number of discs from Start to Dest. In order to move N number of discs from Start to Dest we must move (N-1) discs to access the last disc. This can be achieved by the following recursive pattern:

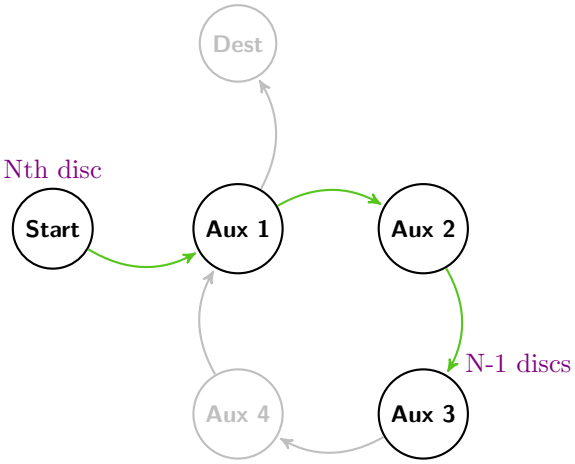


Figure 1.b Move N-1 discs from Start to Aux 3.

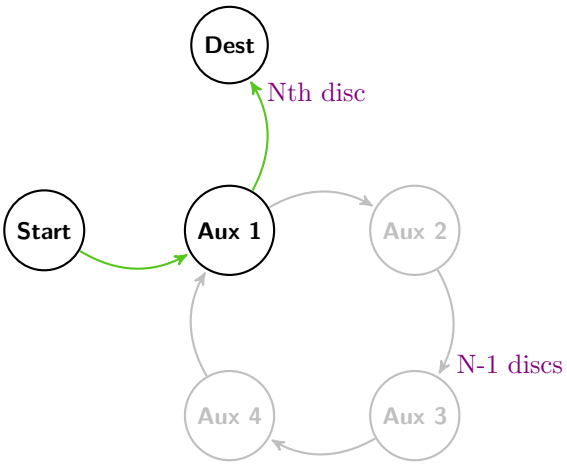


Figure 1.c Move the largest disc from Start to Dest.

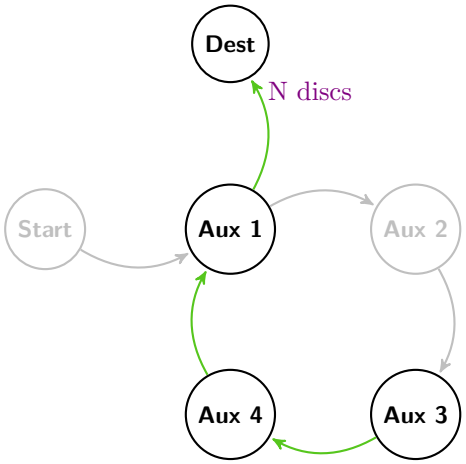


Figure 1.d Move N-1 discs from Aux 3 to Dest.

In order to move a stack of N discs from Start to Dest we must use the Hanoi algorithm as described below:

Hanoi(N, A, B, C, D, E, F){
  if(N = 0) {
    Move 1 disc from A to F.
    A $\xrightarrow{1}$B, B $\xrightarrow{1}$F 2 Atomic Actions = $\theta(2)$
  if(N > 0) {
    if N is greater than 0 then move then take the path:
    Move N-1 discs 3 towers from A to D.
    H3(N-1, A, B, C, D, E, F) 1^{st} Recursive Invocation $\theta(N-1)$
    Move 1 disc from A to F.
    A $\xrightarrow{1}$B, B $\xrightarrow{1}$F 2 Atomic Actions = $\theta(2)$
    Move N-1 discs 3 towers from D to F.
    H3(N-1, D, E, B, F, A, C) 2^{nd} Recursive Invocation $\theta(N-1)$
}

The Hanoi algorithm has the following recurrence relation:

$$T_H = \left\{ \begin{array}{ll} \theta(2), & \text{for } N = 1 \\ 2T_3(N-1)+2, & \text{for } N > 1 \end{array} \right\}$$

We must evaluate the H3 function to calculate the complexity of the Hanoi algorithm. With every invocation of Hanoi we will recursively invoke two H3 functions. Those two functions must be evaluated to see how the recursion trees grow with respect to the size of the stack (N discs).

H3(K, A, B, C, D)
  if(K = 1)
    A $\xrightarrow{1}$B, B $\xrightarrow{1}$C, C $\xrightarrow{1}$D 3 Atomic Actions = $\theta(3)$
  if(K > 1)
    H3(K-1 ,A, B, C, D) 1^{st} Recursive Invocation $\theta(K-1)$
    A $\xrightarrow{1}$ B , B $\xrightarrow{1}$ C 2 Atomic Actions = $\theta(2)$
    H2(K-1, D, A, B, C) 2^{nd} Recursive Invocation $\theta(K-1)$
    C$\xrightarrow{1}$D 1 Atomic Action = $\theta(1)$
    H2(K-1, B, C, D, A) 3^{rd} Recursive Invocation $\theta(K-1)$

$$T_3 = \left\{ \begin{array}{ll} \theta(3), & \text{for } K = 1 \\ T_3(K-1)+2T_2(K-1)+3, & \text{for } K > 1 \end{array} \right\}$$

The H3 algorithm will invoke one H3 and two H2 recursive calls and the arguments will all have diminished by one. The H2 algorithm is defined as a series of functions to move a stack of L discs 2 spaces:

H2(L, A, B, C, D)
if(L = 1)
A $\xrightarrow{1}$B, B $\xrightarrow{1}$C — 2 Atomic Actions = $\theta(2)$
if(L > 1)
H3(L-1, A, B, C, D) — 1^{st} Recursive Invocation $\theta(L-1)$
A$\xrightarrow{1}$B, B$\xrightarrow{1}$C — 1 Atomic Action = $\theta(1)$
H3(L-1, D, A, B,C) — 2^{nd} Recursive Invocation $\theta(L-1)$

$$T_2 = \left\{ \begin{array}{ll} \theta(2), & \text{for } L = 1 \\ 2T_3(L-1)+2, & \text{for } L > 1 \end{array} \right\}$$

The H2 algorithm will invoke two recursive H3 algorithms. Now that we have defined the required recursive algorithm definitions we can analyze the recurrence relationship further:

$$T_H = \left\{ \begin{array}{ll} \theta(2), & \text{for } N = 1 \\ 2T_3(N-1)+2, & \text{for } N > 1 \end{array} \right\}$$

$$T_3 = \left\{ \begin{array}{ll} \theta(3), & \text{for } K = 1 \\ T_3(K-1)+2T_2(K-1)+3, & \text{for } K > 1 \end{array} \right\}$$

$$T_2 = \left\{ \begin{array}{ll} \theta(2), & \text{for } L = 1 \\ 2T_3(L-1)+2, & \text{for } L > 1 \end{array} \right\}$$

The Recurrence Relation for H3 will invoke three functions while Hanoi and H2 will invoke two H3 functions each. Therefore it can be seen that the growth rate exists strictly between $O(2^n)$ and $O(3^n)$. The worst case complexity assumes that every recursive call will triple and give us the simplified Recurrence Relation:

$$\{T(N) = 3T(N-1) + 2(1) | \, R = 3, D = 1\}$$

Therefore, according to the Master Theorem of Reduction by Subtraction:

T(N)= $\theta(3^{(N/1)})$ which simplifies to:

$$\boxed{T_{time}(N) = \theta(3^N)} \tag{1.1}$$

Additionally, the space required to hold the data within memory will remain constant since all recursive invocations free up memory that was used when they terminate. Therefore the space complexity is:

$$\boxed{T_{space}(N) = \theta(N)} \tag{1.2}$$

Source Code Demonstration:

```
14  // Solve the problem with the helper functions
15  void Hanoi(const int n, const string a, const string b, const string c, const string d, const string e, const string f, graph &g){
16      if (n){
17          // Move N-1 discs from Start to Aux3
18          H3( n: n-1, a, b, c, d, e, f, &: g);
19          // Move one disc from Start to Dest
20          g.move(a, b, f);
21          // Move N-1 discs from Aux3 to Dest
22          H3( n: n-1, a, b, c, d, e, f, &: g);
23      }
24  }
```

```
97  // Move a stack of N discs 3 spaces within the cycle
98  void H3mid(const int n, const string a, const string b, const string c, const string d, graph &g){
99      if (n){
100         // Move N-1 discs 2 spaces within the cycle.
101         H2( n: n-1, a, b, c, d, &: g);
102         // Move one disc one space.
103         g.move(a, b);
104         // Move N-1 discs two spaces within the cycle.
105         H2( n: n-1, c, d, a, b, &: g);
106         // Move one disc two spaces.
107         g.move(b, c, d);
108         // Move N-1 discs 3 spaces within the cycle.
109         H3mid( n: n-1, a, b, c, d, &: g);
110     }
111 }
112 // Move a stack of N discs 2 spaces within the cycle
113 void H2(const int n, const string a, const string b, const string c, const string d, graph &g){
114     if (n){
115         g.invoke();
116         // Move N-1 discs 3 spaces within the cycle.
117         H3mid( n: n-1, a, b, c, d, &: g);
118         // Move 1 disc two spaces.
119         g.move(a, b, c);
120         // Move N-1 discs 3 spaces within the cycle.
121         H3mid( n: n-1, d, a, b, c, &: g);
122     }
123 }
124
```

```
(base) Carloss-MacBook-Pro:hanoi carlos$ ./demo
Enter the number of disks:
9
Enter the speed in the animation:
    (1 is slow and 10 is fast)
10
1. Move disc 1 from start to aux1.
.   .   .   .   .   .
2   .   .   .   .   .
3   .   .   .   .   .
4   .   .   .   .   .
5   .   .   .   .   .
6   .   .   .   .   .
7   .   .   .   .   .
8   .   .   .   .   .
9   1   .   .   .   .
----------------------
S   A1  A2  A3  A4  D
```

```
.   .   .   .   .   .
9   .   1   8   .   .
----------------------
S   A1  A2  A3  A4  D

2636. Move disc 1 from aux2 to aux3.
.   .   .   .   .   .
.   .   .   1   .   .
.   .   .   2   .   .
.   .   .   3   .   .
.   .   .   4   .   .
.   .   .   5   .   .
.   .   .   6   .   .
.   .   .   7   .   .
9   .   .   8   .   .
----------------------
S   A1  A2  A3  A4  D
```

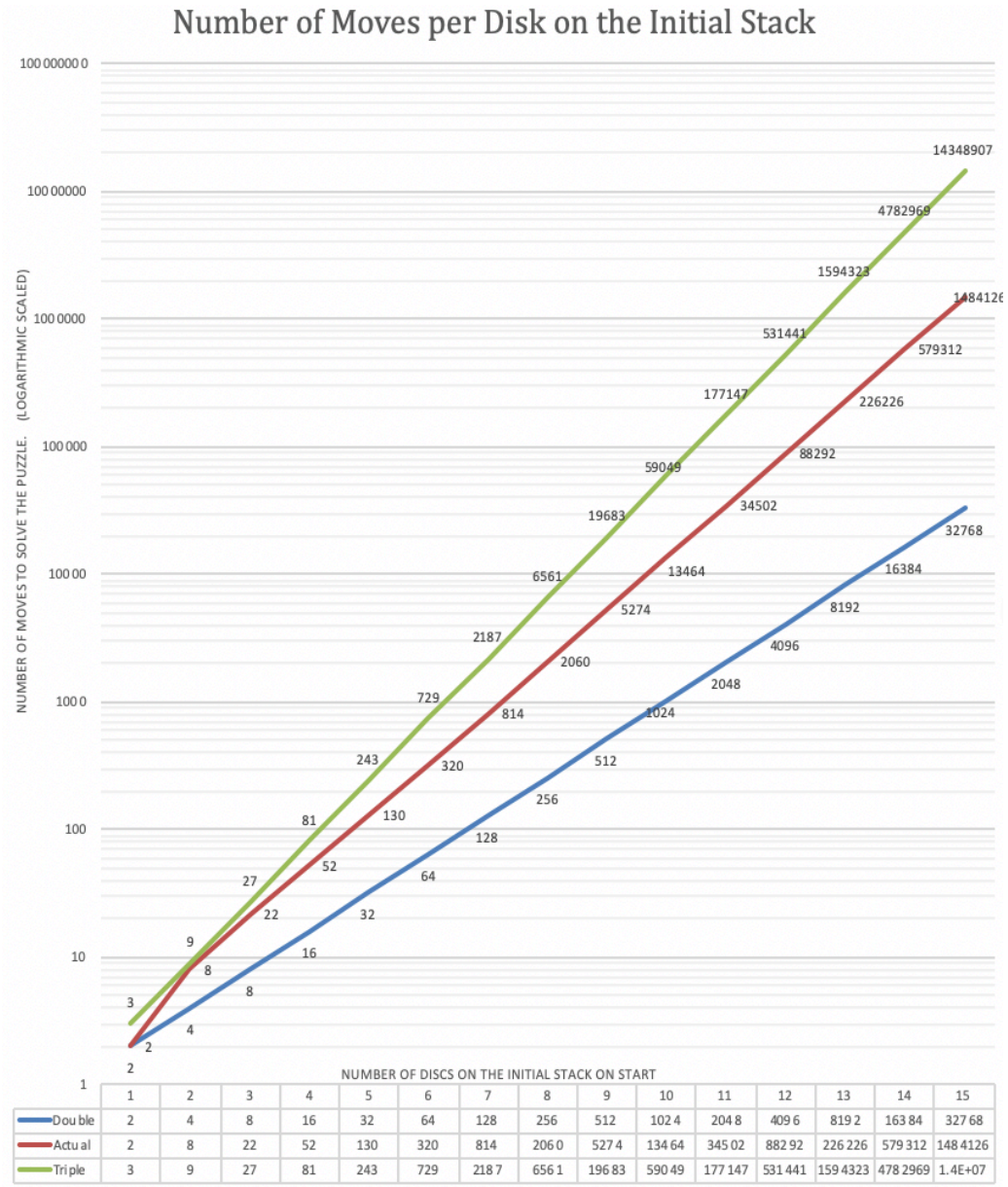
```
5274. Move disc 1 from aux1 to dest.
.   .   .   .   .   1
.   .   .   .   .   2
.   .   .   .   .   3
.   .   .   .   .   4
.   .   .   .   .   5
.   .   .   .   .   6
.   .   .   .   .   7
.   .   .   .   .   8
.   .   .   .   .   9
----------------------
S   A1  A2  A3  A4  D

The number of recursive calls: 882
O(2^n) is: 512
The number of actual moves: 5274
O(3^n) is: 19683
```

The entire source code and demonstration can be found in the link:

github.com/cflores713/HanoiVer2.

The theoretical algorithms for Hanoi, H4, and H3 are implemented verbatim in C++. It is a testament to how recursion can be used to respect the edges in a graph and provide for an easier method than an Adjacency Matrix or List to represent a directed graph.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Double	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768
Actual	2	8	22	52	130	320	814	2060	5274	13464	34502	88292	226226	579312	1484126
Triple	3	9	27	81	243	729	2187	6561	19683	59049	177147	531441	1594323	4782969	1.4E+07

By looking at the plotted growth rates on a logarithmic scale of the actual algorithm in comparison to algorithms that double (2^n) or triple (3^n). It is evident that the complexity is bounded from above by $O(3^n)$.

Problem 2

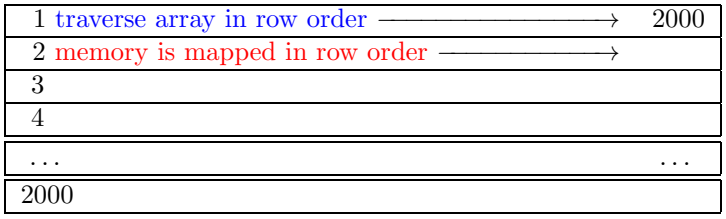
Calculate the Amount of Page Transfers Between Disks

Determine for the following code how many pages are transferred between disk and main memory, assuming each page has 2000 words, the active memory set is 1000 (no more than 1000 pages may be in main memory), and the replacement strategy is LRU (the Least Recently Used page is always replaced); also assume that all two-dimensional arrays are of size (1:2000, 1:2000), with each array element occupying one word, N=2000.

```
for I := 1 to 2000 do
    for J := 1 to 2000 do {
        A[I,J] := A[I,J] * B[I,J]
        B[I,J] := C[N-I+1,J] * B[I,J]
    }
```

Since the J for loop is nested inside of the I for loop we will traverse the array by row. Provided the arrays are mapped into main memory space:
(a.) in Row-Major order. **(b.)** in Column-Major order.

2a.) Memory Mapped by Row-Major



Since we are traversing the array in row order, where J is nested inside I it would provide a lower amount of page transfers since each read/write would only need to scan one page into main memory. Each page contains 2000 words, the array contains 2000 elements per row, and each element contains one word. Therefore in order to read 2000 words we have to transfer one page. By pulling one page we can access all 2000 words required for one read/write.

$$\underbrace{A[I,J]}_{1\text{ write}} := \underbrace{A[I,J]}_{1\text{ read}} * \underbrace{B[I,J]}_{1\text{ read}}$$

The first action requires one write to A by reading A and multiplying it by reading the value in B. This results in three transfers per calculation and we must do this for each column in the array. Which results in 3 x 2000 = 6000

$$\underbrace{B[I,J]}_{1\text{ write}} := \underbrace{C[N-I+1,J]}_{1\text{ read}} * \underbrace{B[I,J]}_{1\text{ read}}$$

The next action will also result in three transfers. The C array will be read in reverse but it would still read the information linear to the mapped row-order. This would also require 6000 page transfers. The total amount of page transfers to perform the given code on an array that is mapped in Row-Order would result in:

$$\boxed{12,000\ transfers} \tag{2.1}$$

2b.) Memory Mapped by Column-Major

traverse array in row order $\longrightarrow$
$\downarrow$ memory is mapped in column order

1	2	3	4	5	⋮	2000

Since we are traversing the array in row order, where J is nested inside I it would provide a high amount of page transfers since each page transfer would only give you one word from main memory. Each page provides 1 word, the array contains 2000 elements per row, and each element contains one word. Therefore in order to read 2000 words we have to transfer 2000 pages.

The first action requires 2000 page transfers for each read/write. The action performs 3 reads/writes since A is written with the value in A and multiplying it by reading the value in B.

This results in 2000 transfers per read/write calculation and we must do this for each column in the array. Which results in 3 x 2000 x 2000 = 12,000,000

The next action will also result in 12,000,000 transfers. The C array will be read in reverse but it would still read the information orthogonal to the mapped column-order. This would also require 12,000,000 page transfers. The total amount of page transfers to perform the given code on an array that is mapped in Column-Order would result in:

$$\boxed{24,000,000\ transfers} \tag{2.2}$$

Problem 3

Asymptotics of Quick Sort With a Fixed Pivot

Construct an infinite sequence of numbers for N and construct an assignment of the numbers 1...N to the array elements that causes Quick Sort, with the pivot as the **First** element in the array, to:

(a.) Execute optimally - A[lo:mid] and A[mid:hi] are always equal size.
(b.) Execute poorly - That is in the slowest possible way.

3a.)Optimal Performance

In order to ensure that the recursive invocations of Quick Sort will return subarrays of equal length we must restrict the domain of possible sizes to a particular set of integers. This is due to the inherent quality that not all sizes of arrays will split evenly. The Assignment and Construction of such a sequence starts with $P(0)$, which is an array of a single element with a value of 1:

Basis Step: $\quad P(0)=\boxed{1}$

We can grow this array by an inductive process. Take $P(0)$, place it in the middle of an array with one element $\mathcal{M}$, and a copy of $P(0)$. The element in $\mathcal{M}$ will always be 2^i for step $P(i)$. Also, add the single element in $\mathcal{M}$ to each element in $P(0)$.

$$\begin{array}{rc} & \overset{2^1}{\quad} \quad \overset{\quad}{\quad} \quad \overset{P(0)+2^1}{\quad} \\ P(1)= & \boxed{2 \mid 1 \mid 1+2} \\ & \mathcal{M} \quad P(0) \quad P(0)+\mathcal{M} \end{array} \implies P(1)=\boxed{2 \mid 1 \mid 3}$$

Then, repeat the process. From now on the first number in the middle array will will be placed behind the last element in middle array, and that the right array will add 2^2 element-wise.

$$\begin{array}{rc} & \overset{2^2}{\quad} \quad \overset{2\curvearrowright}{\quad} \quad \overset{P(1)+2^2}{\quad} \\ P(2)= & \boxed{4 \mid 2,1,3 \mid 2{+}4,\ 1{+}4,\ 3{+}4} \\ & \mathcal{M} \quad P(1) \quad P(1)+\mathcal{M} \end{array} \implies \boxed{4 \mid 1,3,2 \mid 6,5,7}$$

Repeat Again: Given P(2) = 4,1,3,2,6,5,7

	2^3	move 4 to the end ↷	add $\mathcal{M}$ to $P(2)$
P(3)=	8	1,3,2,6,5,7,4	12,9,11,10,14,13,15
	$\mathcal{M}$	$P(2)$	$P(2)+\mathcal{M}$

$\therefore$ **Inductive Step:** P(N)= $\overset{\curvearrowright}{2^N}$ | $P(N-1)$ | $P(N-1)+\mathcal{M}$

For any arbitrary step $P(j)$ the size of the array is: $\{2^{j+1}-1 \mid j \in \mathbb{Z}^+\}$

Assume that P(3) = {8,1,3,2,6,5,7,4,12,9,11,10,14,13,15} The first QuickSort invocation will take the first element 8 as the pivot.

P(3)=	8	1,3,2,6,5,7,4	12,9,11,10,14,13,15
	Pivot	⟷ Swap with 4	

A[low:mid]		A[mid:high]
4,1,3,2,6,5,7	8	12,9,11,10,14,13,15
4 ↔ 2		12 ↔ 10

P(2) =	2,1,3	4	6,5,7	8	10,9,11	12	14,13,15	

P(1) =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

P(0) =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The array of 15 elements is sorted in three recursive invocations and each subarray is the same size. The upper bound on the number of recursive calls is $O(N) = \lceil log_2(N) \rceil$. Since each element must be compared in the sub array it would lead to O(N) number of comparisons. The Inductive Assignment and Construction of the P(N) Sequence leads to the best performance:

$$\boxed{Sequence\ \&\ Assignment = P(N),\ O(N) = Nlog_2(N)} \tag{3.1}$$

3b.) Worst Performance

An array that is already sorted in ascending order will provide the slowest execution possible. That is because the pivot will always be the least value if it is in ascending order. A simple sequence from 1 to infinity would suffice. The recursive calls would create an empty A[lo:mid] array and an array with N-1 elements in A[mid:high]. This would generate N number of recursive calls and N number of comparisons leading to:

$$Seq = \{a_i\}_{i=1}^{\infty},\ Assign = \boxed{1 \mid 2 \mid 3 \mid \ldots \mid \infty},\ O(N) = N^2 \tag{3.2}$$