Running time of programs

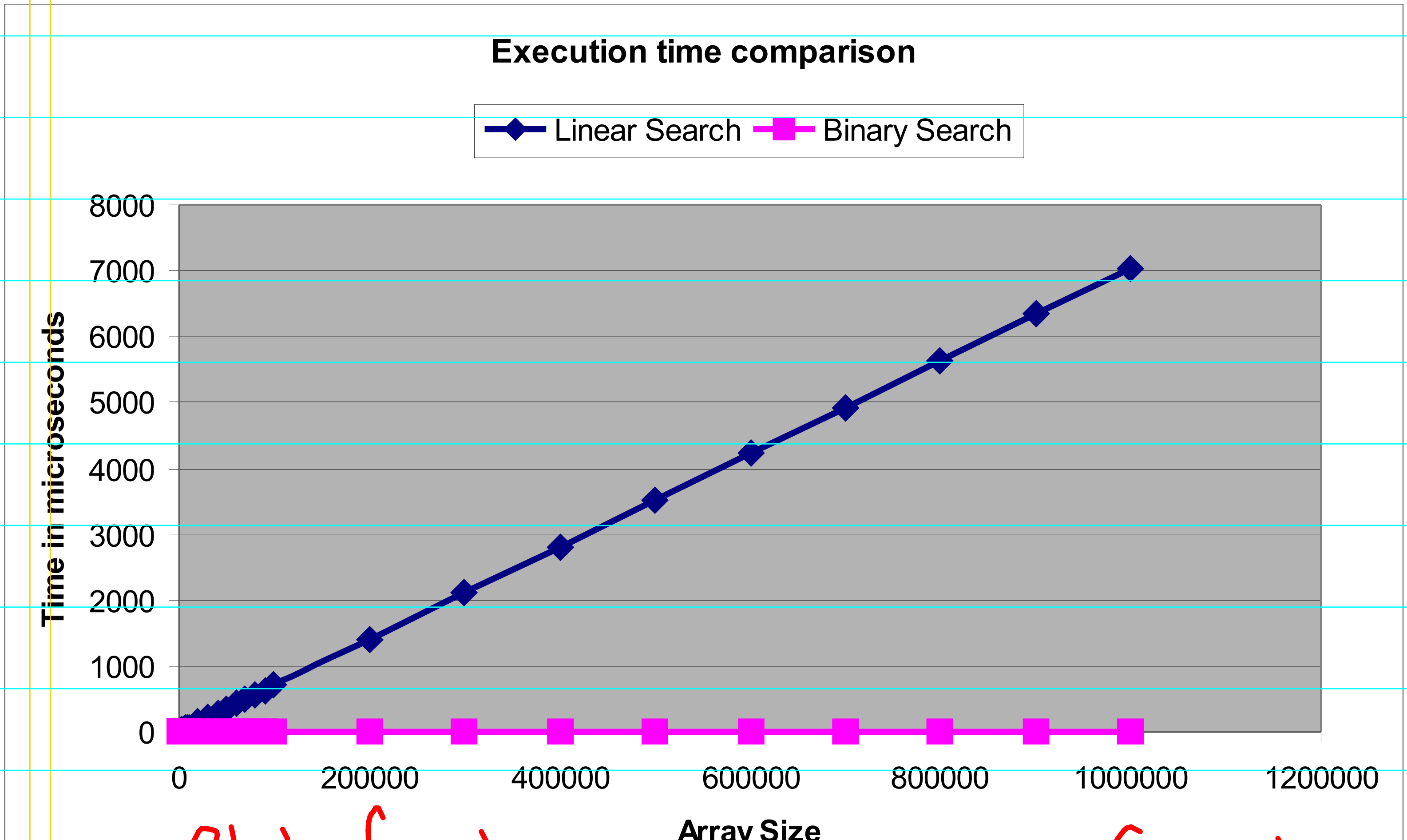
- Running time is a function of?
  - Array size
  - Number position
- Rigorous, scientific way to do find running time?
  - Run the program and time it (must properly design running time experiments)
    - Determine "input" (Array A, number to search for)

"Proper" for fixed i/ps for fair comparisons

1. No background processes
2. Comparable data types (structures)
3. Single threaded / no wasted steps

4 To normalise wrt OS/kernel related background processes (a) average across multiple runs for each program for fixed i/ps or (b) use Virtual Machines with guaranteed resources

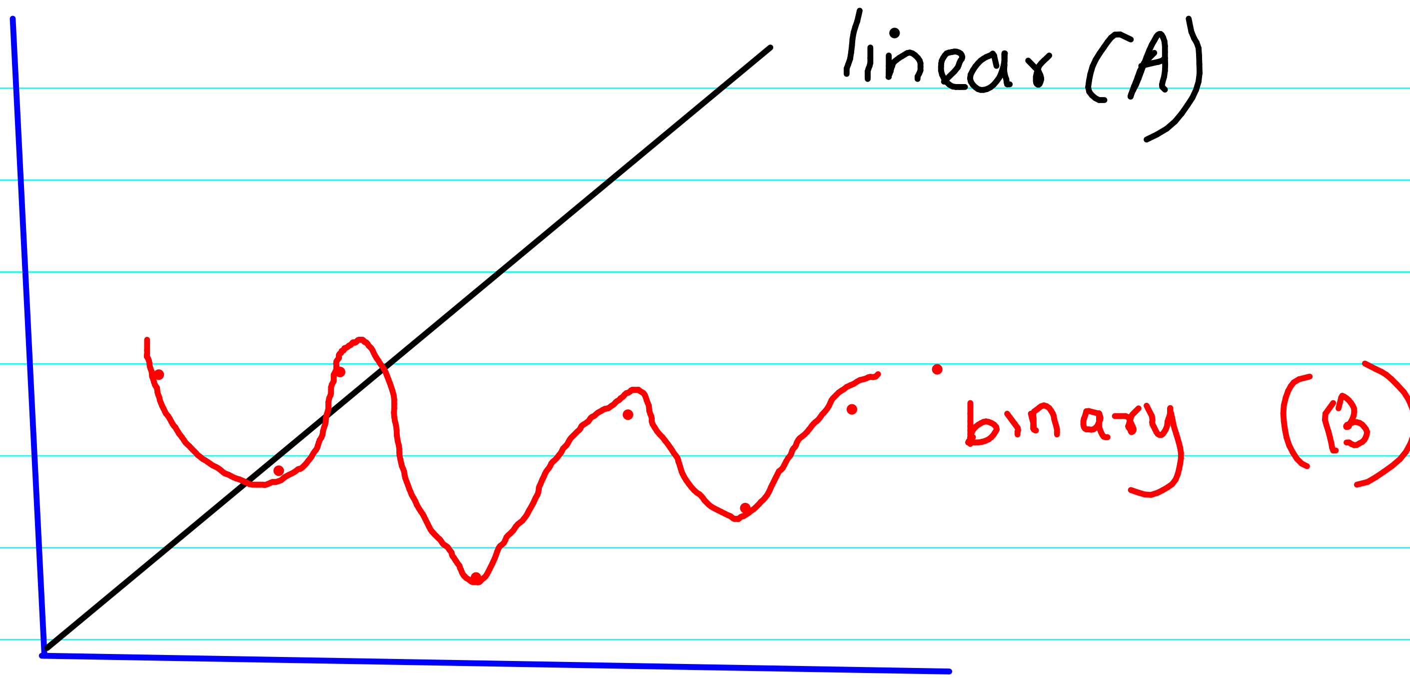
Results of experiment



Plot for time vs. array size (worst case over position of num being searched)

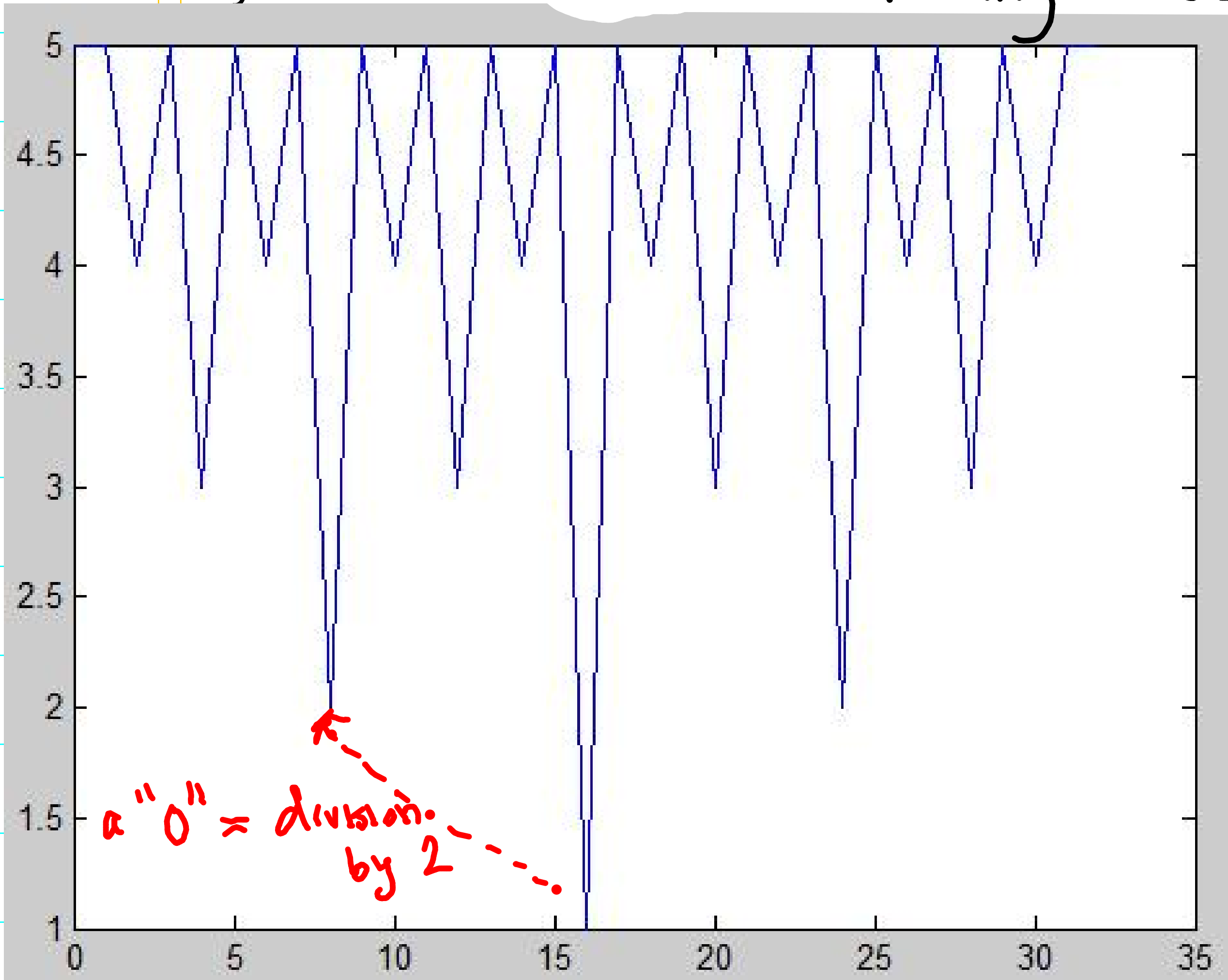
*Worst case

Q: How will plot(s) of time vs position of num being searched for look like (for fixed array size)



Look at answer posted by Sasank Chilamkurthy at https://piazza.com/class/hq4vn7sblku32q?cid=11

Assume length to be of 2^ n form. Write index of our element in binary and count. Ans = n - # trailing zeros



Q: what if $length = 2^n + \delta$

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

Disadvantages

- Can take too much time
- Affected by too many factors (the hardware, the compiler, etc)
- Too much detailed – we just need to know the "essential behaviour"
- And most importantly
  - Never do an experiment before trying to reason about the outcome!!

i.e first set your expectation right

Algorithm Analysis

- For any system, if pen-paper mathematical analysis is possible *one must always do this.* Why?
  - Usually much faster
  - Zooms in on the relevant details, allows to ignore unnecessary details
  - Gives a *fundamental* “environment free” understanding of a system
- But such analysis *always* requires a “model” of the system under study

Model for Algorithm Analysis

- "Normal" computer – sequential instructions
- All "basic" instructions take <u>one</u> unit of time – addition, multiplication, comparison, assignment
- Computer has infinite memory
- Clearly, some aspects are ignored – disk read times, "paging", context switching, etc
  - But this is intentional
  - We want to understand the *algorithm*, not implementation issues

Imagine running program in "debug" mode
\# of "return" presses = # of lines executed

Analysis of search algorithm A

Let size be *N*. Assume element is at index *n* = *0…N-1*

```
for (i=0; i < size; i++) {
    if (A[i] == num) {
        found = true;
        break;
    }
```

Assignment: 1

Array access: 1
Comparison: 1

(0.., n) X (*n+1*)

Assignment: 1 +1

for break

Comparison: 1

X (*n+1*)

Increment: 1

X *n*

int i : primitive
int[] A : non-primitive

Some of these are done multiple times – which? How many times?

Search algorithm A: time for successful search

- Total time *($T_s(n)$)*, when element is at index $n=0 \ldots N-1$ (successful search)
  - $1+ 2 \times (n+1) \;+ 1 + \; 1 \times n \;+\; 1 \times (n+1)$
  - $T_s(n) = 4n+5$ + 1 in for break

Analysis of search algorithm A

Now assume search element is not present

```
for (i=0; i < size; i++) {
    if (A[i] == num) {
        found = true;
        break;
    }
}
```

Assignment: 1

Array access: 1
Comparison: 1

X *N*

Comparison: 1

X *(N+1)*

Increment: 1

X *N*

Note: The time reqd for this case is the same as the time reqd when n = N-1

Search Algorithm A: time for unsuccessful search

- Total time for unsuccessful search:
  - 1 + 2 x N + 1 x N + 1 x (N+1)
  - $T_u(N) = 4N + 2$

Next Question: [H/W]

How would you compute the average number of instructions executed by program A, where you average across all possible values of of the position of `num' while holding the length of the list (that is, the value of end-begin) as a constant