Search Algorithm A

- Getting "one quantity" out of this?
- Two main ways
  - Average [i.e Expected value]
  - Maximum (termed "worst case" in this field)
- To calculate average, we need?
  - Probability distribution of input
    - p = Probability of successful search
    - P(n | success) Probability of where element will be (if successful)

p can be finite if values are finite

Eg: if N = 1/2 the number of possible values, then p ≈ 1/2

Search algorithm A: Average case analysis

- Let probability of success be *p*
  - Let conditional probability of element being at index *n* be 1/N (discrete uniform distribution)
- Then, average is:
  - 3N+2.5

$$p\sum_{n=0}^{N-1} T_s(n).\frac{1}{N}+(1-p)T_u(N)$$

$$\text{¿}\; p\sum_{n=0}^{N-1}(4n+5).\frac{1}{N}+(1-p)(4N+2)$$

$$\text{¿}\; p.\frac{1}{N}[\frac{4(N-1)N}{2}+5N]+(1-p)(4N+2)$$

$$\text{¿}\; p.\frac{1}{N}[2(N-1)N+5N]+(1-p)(4N+2)$$

$$\text{¿}\; p.(2N+3)+(1-p)(4N+2)$$

$$Assume\;\; p=\frac{1}{2}$$

$$T_{avg}(N)=\frac{2N+3+4N+2}{2}=3N+2.5$$

Search Algorithm A
Worst Case Analysis

- Worst case is when?
  - Element is not found
- → Worst case time is:
  - $T_{worst}(N) = 4N+2$

$4(N-1)+6$ vs $4N+2$

Worst case analysis for Search Algorithm B

Time taken *in one function call*

A little approx-i mate

```
bsearch(vector<int> &a, int num, int begin, int end) {
  int mid;
  mid  = (begin + end)/2;
  if (begin > end) {
  }
  else {
     if (a[mid] == num)
         found = true;
    else if (num < a[mid])
       bsearch(a,num, begin, mid-1);
    else
       bsearch(a,num,mid+1, end);
  }
```

Assignment, math operations: 3

Comparison: 1

[last call: only 3+1 = 4]

Comparison:2

Comparison:2

Recursive call: 20 (just the call-number is arbitrary)

Note: 20 is cost of invoking & NOT executing

Let is assume function call is more expensive

Note: Recursion calls can involve more overheads than iterative counter parts

Owing to need for saving & retrieving parent program state (in a data structure called the "stack")

Eg: Recursion vs iterative prog for factorial

$fact(n) = n * fact(n-1)$

```
for (i=1; i<=n; i++) {
    fact = fact * i
}
```

Algorithm B Analysis (Worst Case)

- Element is not present in array [effectively scan every position]
- Time required:
  - In each call except last
    - 1 assignment & calculation + 3 comparisons + 1 call ~ 28
  - How many such calls?

Algorithm B Worst Case Analysis

- How many recursive calls?
  - Array size is N
  - First call is with array limits (0, N-1) – range N
  - Recursive calls reduce search range by half } effective
  - Calls stop when begin > end
  - N → N/2 → N/4→ N/8 → ….→ 1
- In how many calls will this happen?
  - Obviously ~ $\log_2 N$
- Time required ~ 28 $\log_2 N$ + 4 (for last call)

(Can also be solved by solving : $T(N) = T(\frac{N}{2}) + 28$)

Algorithm A vs B worst case comparison

- $4N+3$
- $\sim 28 \log_2 N + 4$
- With the given constants
  - For N=2,3,4…: Algorithm A seems faster
  - After N >= 37 : Algorithm B is faster, and remains so
- This analysis is consistent with experimental observations (for a different N)

Algorithm A vs B running time comparison

- So which is really the faster one?
- Clearly, we need to worry about running times of programs only when "input size" is large
  - For small input size, even a bad program will run fast
- → In algorithm analysis, we focus on <u>asymptotic analysis only</u>
  - "Asymptotic" = "As n goes to infinity"
- Furthermore, we focus on "order" of growth i.e., we do not want to make a big distinction between
  - 40N+400 and 2N+1 ! w.r.t aN+b, same growth rate
- But we want to make a big distinction between
  - 2N+1 and 400log N + 1000 → $O(f'(N))$ & $f(N) > f'(N)$
  - We want to say that "2N+1" is slower

$O(f(N))$

Algorithm A vs B worst case comparison, asymptotic

- The term N "grows faster" then log N
  ($f(N)$) ($f'(N)$)
- → Intuitively, irrespective of the <u>constants</u> involved in the previous analysis, we know that <u>eventually</u> (after some large value of n) Algorithm A time will exceed Algorithm B time
  - We also know that Algorithm A time is proportional to N → "LINEAR"
  - We also know that Algorithm B time is proportional to log N → LOGARITHMIC
- Constants don't matter → what matters is one was linear, other was logarithmic

$\exists \; n_0 \text{ s.t } \forall \; n \geq n_0$

$$2n + 1 > 400 \log n + 1000$$

$O(f(n)) \qquad O(f'(n))$

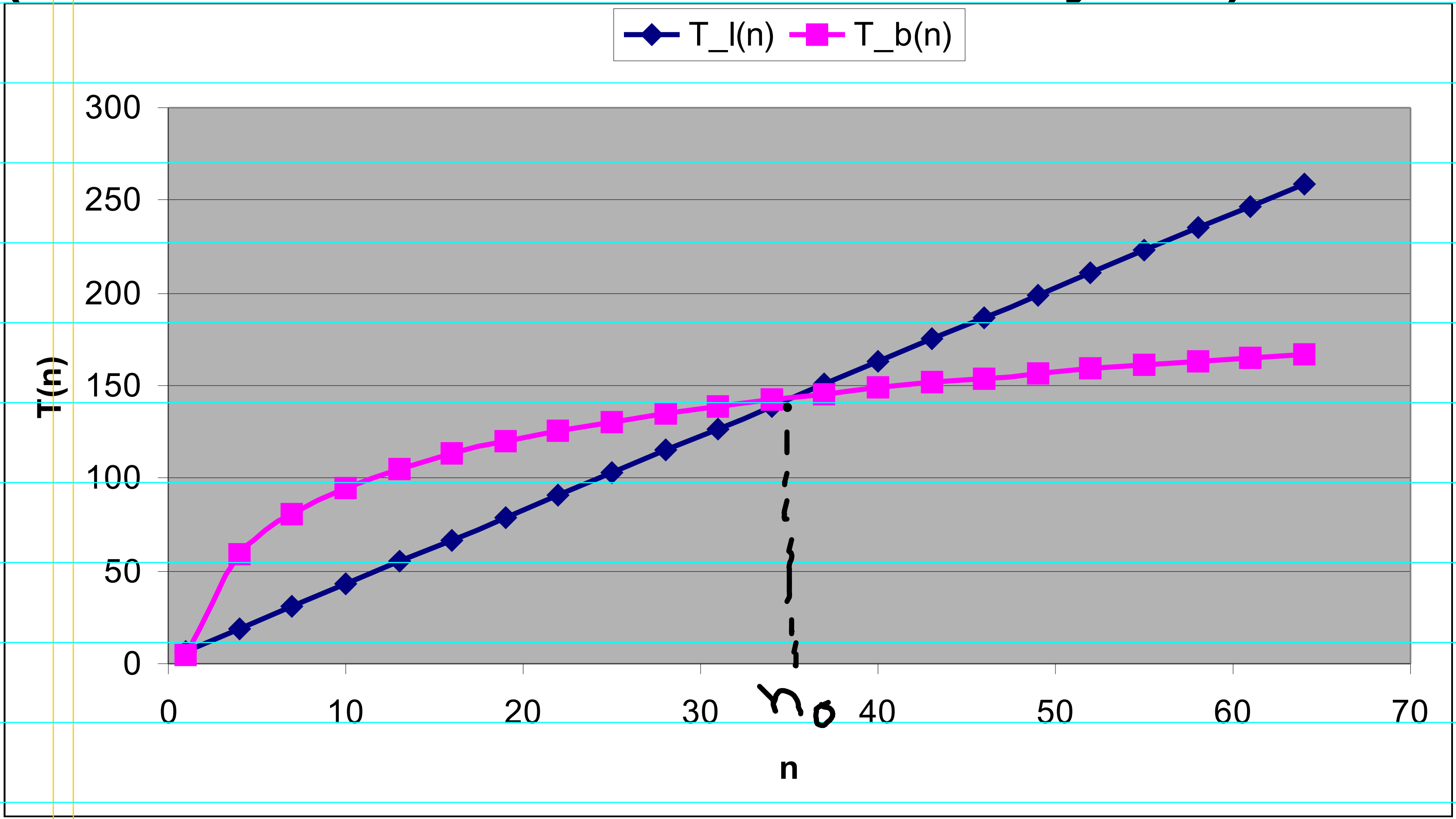
All I need to know beforehand abt $\boxed{f(n) \;\&\; f'(n)}$ to draw above conclusion is that

canonical forms

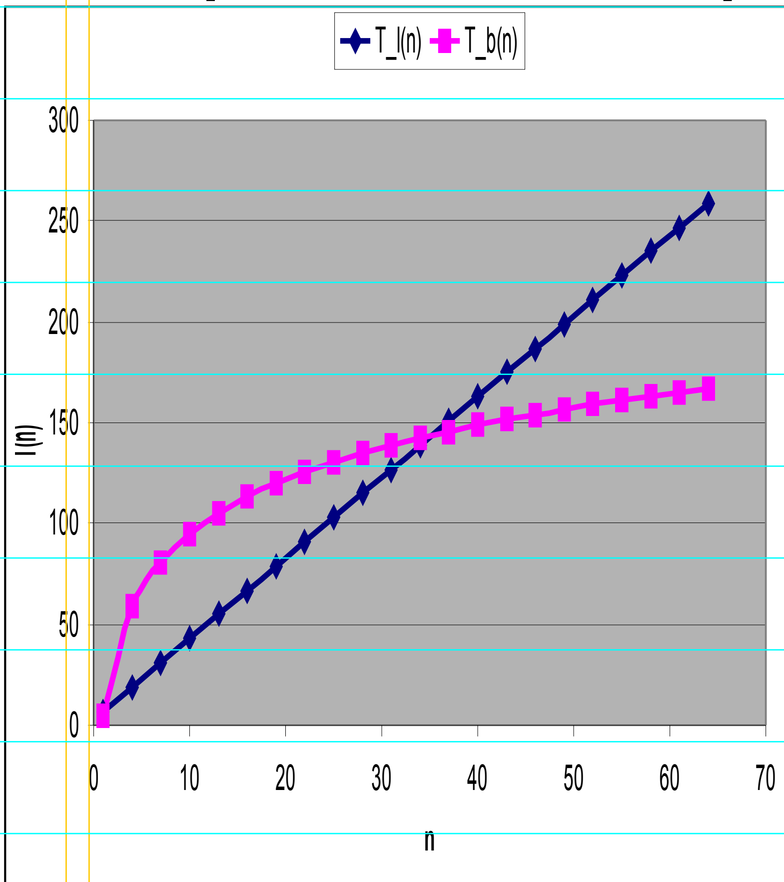
$\exists \; n_0 \text{ s.t } \forall \; n > n_0$

$$f(n) > f'(n)$$

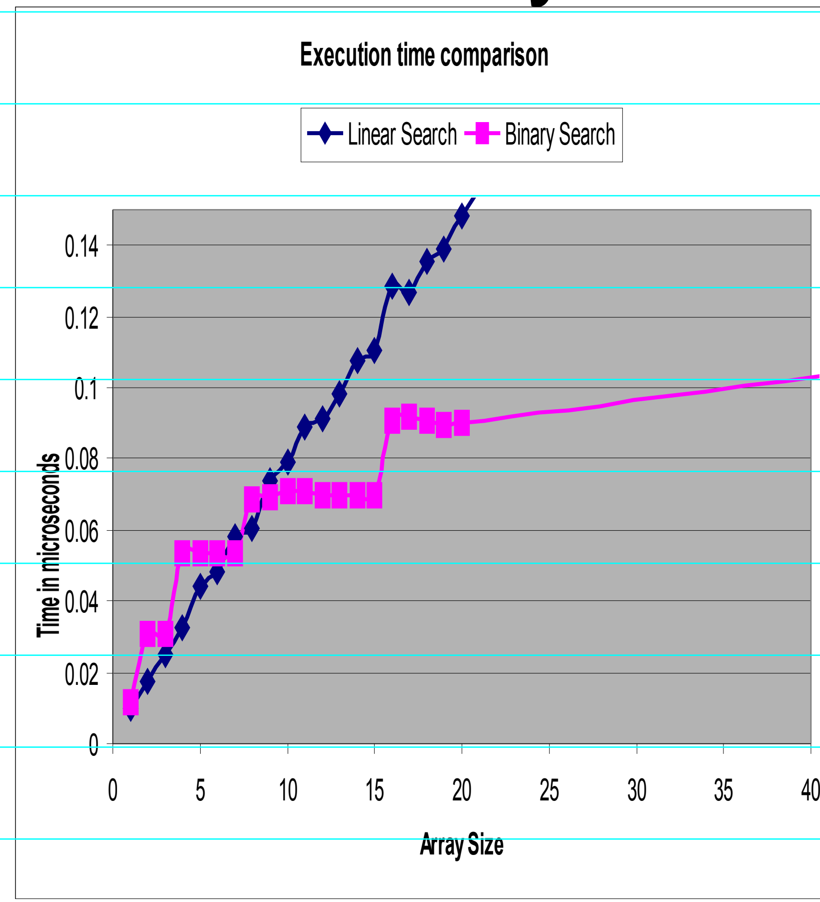
Algorithm A vs B running time (based on theoretical analysis)



Algorithm A vs B
Compare with experimental analysis



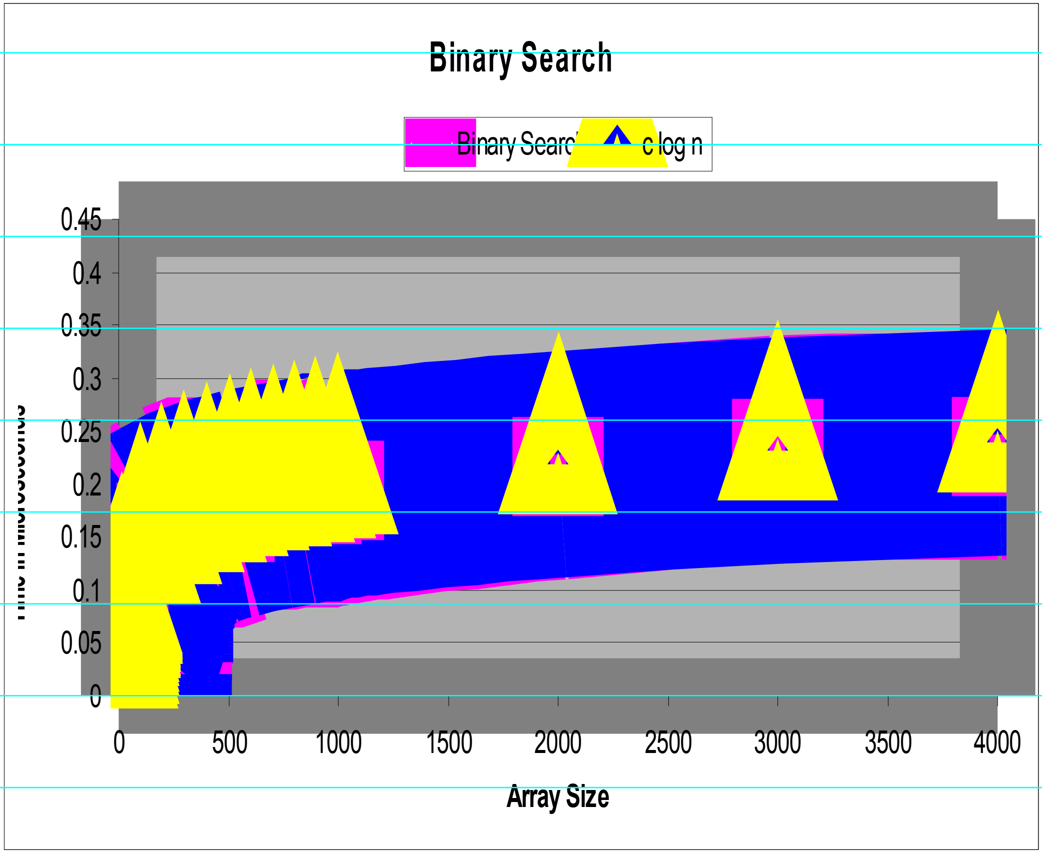
Theoretical



Experimental

Some more curiosity about experimental results

- Linear search shows linear trend
- Does binary search have log trend?
- Let’s check (unscientifically!)



Running Time formalisms

- The "fundamental" running time of an algorithm is called the "time complexity" of an algorithm : In terms of canonical fns
- Time complexity is expressed only in terms of the dominating terms, or "orders"
- "Order of complexity" of an algorithm is the most important aspect of an algorithm

For convenience of analysis canonical fn classes dont have multiplicative & additive constants

Formalizing…definitions

- $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \le c\,f(N)$ when $N \ge n_0$
- We say that T(N) is of the "order of *f(N)*" or Big-Oh *f(N)* or just O(*f(N)*)
- In words, this means *T(N)* is of the order of *f(N)* if you can find a point n_0 after which *T(N)* is smaller than a linearly scaled version of *f(N)*. Roughly speaking:
  - The point n_0 helps ignore the additive constants
  - The factor *c* helps ignore the multiplicative constants
  - Focus is only on the dominating "N" term

HOMEWORK
(Deadline 17/01/2014)

Which of the following is/are true? Prove:

$$40N + 400 = O(2N + 1)$$
$$2N + 1 = O(40N + 400)$$

$$40N + 400 = O(40\log N + 400)$$
$$40\log N + 400 = O(40N + 400)$$