Data Structures and Algorithms

Prof. Ganesh Ramakrishnan,
Prof. Ajit Diwan,
Prof. D.B. Phatak

Department of Computer Science and Engineering
IIT Bombay

Session: Running Time of a Program: Average and Worst Case Complexity, Asymptotic Analysis

Search Algorithm A : LinearScan(e, S)

- Viewing analysis as function identification.
- Two main ways
  - Average (or expected value) → across all instances of S but with |S| = N
  - Maximum (or "worst case")
- To calculate average, compute probability distribution over inputs
- Components of Probability distribution
  - Probability of successful search → pr(e is found in S)
  - Probability of position of element e → Pr(e is found in S at i)

Search Algorithm A: Average case analysis

- Success probability $= p$
  - Conditional probability of e being at index $i = \frac{1}{N}$ $(\forall i)$
- Average will be:
  $3N + 2.5$

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

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

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

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

$$p.(2N+3) + (1-p)(4N+2)$$

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

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

Search Algorithm A: Average vs. **Worst case** analysis

- Success probability $= p$
  - Conditional probability of e being at index $i = \frac{1}{N}$
- Average will be:
  $3N + 2.5$
- **Recall Worst case**
  - **When element e is not found**
- **Worst case time:**
  - $T_{worst}(N) = 4N + 2$

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

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

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

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

$$p.(2N+3) + (1-p)(4N+2)$$

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

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

Alternative (Binary) Search Algorithm B

Recursive binary search

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

$n \geq S[mid]$

mid

$n < S[mid]$

Worst case analysis for Search Algorithm B

Recursive binary search

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

Time taken in one function call

- Assignment, math operations: 3
- Comparisons: 5 (3 for the final call)
- Function call is more expensive with an arbitrary invocation cost C

Recursion vs Iterative

- Recursive calls can involve more overheads
- Need for saving, retrieving parent program state
- Uses stack to maintain states
- Recall factorial program implementations using recursion as against iterative calls

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

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

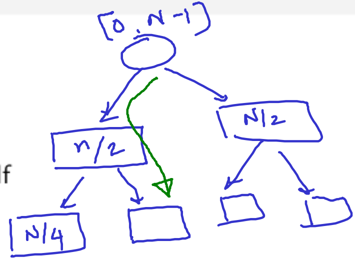
Algorithm B Analysis (Worst Case)

- Element e is not present in array
  - Amounts to scanning every position
- Time required (in each call except last)
  - $\sim$ C+8
  - How many such calls?

Algorithm B Worst Case Analysis

- How many recursive calls?
  - First call is with range $(0, N-1)$
  - Recursive calls reduce search range by factor of half
  - Termination when $begin > end$
  - $N \rightarrow \frac{N}{2} \rightarrow \frac{N}{4} \rightarrow \frac{N}{8} \rightarrow \ldots \rightarrow 1$
- Number of calls for this to happen?
  - Obviously $\sim log_2 N$
- Time required $\sim (C+8)log_2 N + 6$ (for last call)
- Recurrence is $T(N) = T(\frac{N}{2}) + (C+8)$

Solved using Masters Theorem



termination

Algorithm A vs B worst case comparison

- Algorithm A: $4N + 3$
- Algorithm B: $\sim (C + 8)\ log_2 N + 6$
- Which is faster? Assume $C \approx 10$
  - For $N = 2, 3, 4...$: Algorithm A seems faster
  - After $N \geq 21$: Algorithm B is faster, and remains so

Monotonicity

- This analysis is consistent with experimental observations (for a different N)

Asymptotic Analysis and Order of Growth

- Asymptotic Analysis: Analyse and compare running times only when "input size" is large
  - Focus on analysis for ~~n~~ N $\rightarrow \infty$ only
  - For small inputs, even a bad algorithm will perform well
- We focus on Order of growth [ignore unnecessary details]
  - Do not make a big distinction between $40N + 400$ and $2N + 1$
  - Do make a big distinction between $2N + 1$ and $400 log N + 1000$
  - We want to say that $2N + 1$ is slower

or N^2

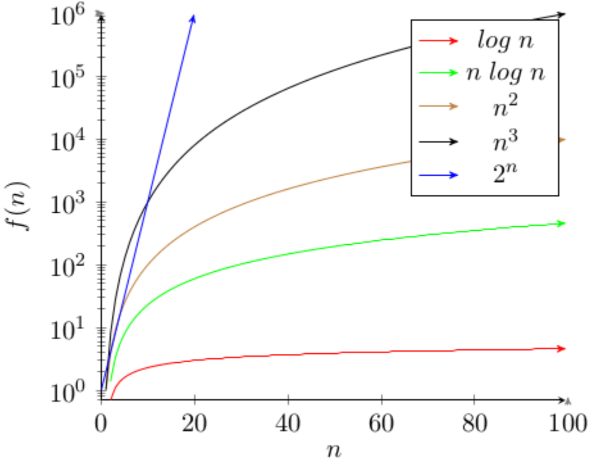
(N^2 grows much faster than 40N)

Algorithm A vs B worst case comparison, asymptotic

$\frac{\partial N}{\partial N} = 1$ $\frac{\partial \log N}{\partial N} = \frac{1}{N}$

- The function N "grows faster" than $\log N$ (Rate of Growth)
- Intuition: Irrespective of the constants involved in the analysis, we know that eventually (after some large value of n), $Algorithm\ A$ time will become slower than $Algorithm\ B$
  - Recall that for $Algorithm\ A$, $T(N)$ is proportional to $N \Rightarrow$ "LINEAR"
  - Recall that for $Algorithm\ B$, $T(N)$ is proportional to $log\ N \Rightarrow$ "LOGARITHMIC"
- Constants don't matter; what matters is one was linear, other was logarithmic

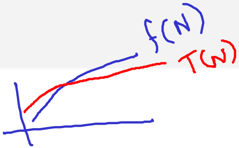
Growth rate of important functions



Running time formalisms

- The "fundamental" running time of an algorithm is called its "time complexity"
- 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

Formalizing ... definitions



- $T(N) = \mathcal{O}(f(N))$ if there are positive constants c and n_0, such that $T(N) \leq c\, f(N)$ whenever $N \geq n_0$ C= independent of N
- We say that $T(N)$ is of the "order of $f(N)$" or Big-Oh $f(N)$ or just $\mathcal{O}(f(N))$
- 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 $f(N)$
  - 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

Thank you