Algorithm Analysis, Asymptotic notations
CISC4080
CIS, Fordham Univ.

Instructor: X. Zhang
Last class

• Introduction to algorithm analysis: fibonacci seq calculation
  • counting number of “computer steps”
  • recursive formula for running time of recursive algorithm
Outline

• Review of algorithm analysis
  • counting number of “computer steps” (or representative operations)
  • recursive formula for running time of recursive algorithm
  • math help: math. induction
• Asymptotic notations
• Algorithm running time classes: P, NP
Running time analysis, \( T(n) \)

- Given an algorithm in pseudocode or actual program

- For a given size of input, how many total number of computer steps are executed? A function of input size…

  - **Size of input**: size of an array, # of elements in a matrix, vertices and edges in a graph, or # of bits in the binary representation of input, …

  - **Computer steps**: arithmetic operations, data movement, control, decision making (if/then), comparison,…
    - each step take a **constant** amount of time
    - Ignore: overhead of function calls (call stack frame allocation, passing parameters, and return values)
running time analysis

function fib1(n)
if n = 0: return 0
if n = 1: return 1
return fib1(n - 1) + fib1(n - 2)

- Let $T(n)$ be number of computer steps to compute $\text{fib1}(n)$
  - $T(0)=1$
  - $T(1)=2$
  - $T(n)=T(n-1)+T(n-2)+3$, $n>1$
- Analyze running time of recursive algorithm
  - first, write a recursive formula for its running time
  - then, recursive formula \Rightarrow closed formula, asymptotic result
- How fast does $T(n)$ grow? Can you see that $T(n) > F_n$?
- How big is $T(n)$?
Mathematical Induction

- $F_0=0$, $F_1=1$, $F_n=F_{n-1}+F_{n-2}$
- We will show that $F_n \geq 2^{0.5n}$, for $n \geq 6$ using strong mathematical induction technique
- Intuition of basic mathematical induction
  - it’s like Domino effect
  - if one push 1st card, all cards fall because
    1) 1st card is pushed down
    2) every card is close to next card, so that when one card falls, next one falls
Mathematical Induction

• Sometimes, we needs the multiple previous cards to knock down next card...

• Intuition of strong mathematical induction
  • it’s like Domino effect: if one push first two cards, all cards fall because the weights of two cards falling down knock down the next card

• Generalization: 2 => k
Fibonacci numbers

- $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$

- show that $F_n \geq 2^{\frac{n}{2}} = \sqrt{2^n}$ for all integer $n \geq 6$ using strong mathematical induction

  - basis step: show it’s true when $n=6$, 7
  - inductive step: show if it’s true for $n=k-1$, $k$, then it’s true for $k+1$

- given

$$F_{k-1} \geq 2^{\frac{k-1}{2}}, \quad F_k \geq 2^{\frac{k}{2}}$$

$$F_{k+1} = F_{k-1} + F_k \geq 2^{\frac{k-1}{2}} + 2^{\frac{k}{2}}$$

$$\geq 2^{\frac{k-1}{2}} + 2^{\frac{k-1}{2}}$$

$$= 2 \times 2^{\frac{k-1}{2}} = 2^{1 + \frac{k-1}{2}} = 2^{\frac{k+1}{2}}$$
Fibonacci numbers

- $F_0=0$, $F_1=1$, $F_n=F_{n-1}+F_{n-2}$

\[
F_n \geq 2^{\frac{n}{2}} = 2^{0.5n}
\]

- $F_n$ is **lower bounded by** $2^{0.5n}$

- In fact, there is a tighter lower bound $2^{0.694n}$

- Recall $T(n)$: *number of computer steps to compute fib1(n)*,
  - $T(0)=1$
  - $T(1)=2$
  - $T(n)=T(n-1)+T(n-2)+3$, $n>1$

\[
T(n) > F_n \geq 2^{0.694n}
\]
Exponential running time

• Running time of Fib1: \( T(n) > 2^{0.694n} \)
• Running time of Fib1 is exponential in \( n \)
  • calculate \( F_{200} \), it takes at least \( 2^{138} \) computer steps
• On NEC Earth Simulator (fastest computer 2002-2004)
  •Executes 40 trillion (\( 10^{12} \)) steps per second, 40 teraflots
  • Assuming each step takes same amount of time as a “floating point operation”
  • Time to calculate \( F_{200} \): at least \( 2^{92} \) seconds, i.e., \( 1.57 \times 10^{20} \) years
• Can we throw more computing power to the problem?
  • Moore’s law: computer speeds double about every 18 months (or 2 years according to newer version)
Exponential algorithms

• Moore’s law (computer speeds double about every two years) can sustain for 4-5 more years…
Exponential running time

- Running time of Fib1: $T(n) > 2^{0.694n} = 1.6177^n$
- Moore’s law: computer speeds double about every 18 months (or 2 years according to newer version)
  - If it takes fastest CPU of this year calculates $F_{50}$ in 6 mins, 12 mins to calculate $F_{52}$
  - fastest CPU in two years from today can calculate $F_{52}$ in 6 minutes, $F_{54}$ in 12 mins
- Algorithms with exponential running time are not efficient, not scalable
Algorithm Efficiency vs. Speed

E.g.: sorting n numbers
Friend’s computer = $10^9$ instructions/second
Friend’s algorithm = $2n^2$ computer steps

Your computer = $10^7$ instructions/second
Your algorithm = $50n\log(n)$ computer steps

To sort n=10^6 numbers,
Your friend:
$$\frac{2 \times (10^6)^2 \text{ instructions}}{10^9 \text{instructions/second}} = 2,000 \text{ seconds}$$

You:
$$\frac{50 \times (10^6)\log(10^6) \text{ instructions}}{10^7 \text{instructions/second}} \approx 100 \text{ seconds}$$

Your algorithm finishes 20 times faster!
More importantly, the ratio becomes larger with larger n!
Compare two algorithms

- Two sorting algorithms:
  - yours: $50n \log_2 n$
  - your friend: $2n^2$
- Which one is better (for large program size)?
  - Compare ratio when $n$ is large

$$\frac{50n \log_2 n}{2n^2} = \frac{25 \log_2 n}{n} \rightarrow 0, \text{ when } n \rightarrow \infty$$

For large $n$, running time of your algorithm is much smaller than that of your friends.
Rules of thumb

• if
\[
\frac{f(n)}{g(n)} \rightarrow 0, \text{ when } n \rightarrow \infty
\]

• We say \(g(n)\) dominates \(f(n)\), \(g(n)\) grows much faster than \(f(n)\) when \(n\) is very large

• \(n^a\) dominates \(n^b\), if \(a > b\)
  • e.g., \(n^2\) dominates \(n\)

• any exponential dominates any polynomials
  • e.g., \(1.1^n\) dominates \(n^{20}\)

• any polynomial dominates any logarithm
  • \(n\) dominates \(\log n^2\)
Growth Rate of functions

(Asymptotic) Growth rate of functions of n (from low to high):

\[ \log(n) < n < n \log(n) < n^2 < n^3 < n^4 < \ldots < 1.5^n < 2^n < 3^n \]
Compare Growth Rate of functions (2)

- Two sorting algorithms:
  - yours: \( 2n^2 + 100n \)
  - your friend: \( 2n^2 \)
- Which one is better (for large arrays)?
  - Compare ratio when \( n \) is large

\[
\frac{2n^2 + 100n}{2n^2} = 1 + \frac{100n}{2n^2} = 1 + \frac{50}{n} \to 1, \text{ when } n \to \infty
\]

They are same! In general, the lower order term can be dropped.
Two sorting algorithms:
- yours: $100n^2$
- your friend: $n^2$

Your friend’s wins.

Ratio of the two functions:

$$\frac{100n^2}{n^2} = 100, \text{ as } n \to \infty$$

The ratio is a constant as n increase. =>
They scale in the same way. Your alg. always takes 100x time, no matter how big n is.
Focus on **Asymptotic Growth Rate**

- In answering “How fast \( T(n) \) grows as \( n \) approaches infinity?”, leave out
  - **lower-order terms**
  - constant coefficient: not reliable (arbitrarily counts # of computer steps), and hardware difference makes them not important
  - Note: you still want to optimize your code to bring down constant coefficients. It’s only that they don’t affect “asymptotic growth rate”
- e.g., bubble sort executes
  
  \[
  T(n) = \frac{n(n - 1)}{2} = \frac{n^2 - n}{2}
  \]
  steps to sort a list of \( n \) elements
  - bubble sort’s running time has a quadratic growth rate... 

\[ T(n) = n^2. \]
Big-O notation

- Let $f(n)$ and $g(n)$ be two functions from positive integers to positive reals.
- $f=O(g)$ means: $f$ grows no faster than $g$, $g$ is asymptotic upper bound of $f(n)$
  - $f = O(g)$ if there is a constant $c>0$ such that for all $n$, $f(n) \leq c \cdot g(n)$
  - or $\frac{f(n)}{g(n)} \leq c$
- Most books use notations $f \in O(g)$, where $O(g)$ denotes the set of all functions $T(n)$ for which there is a constant $c>0$, such that $T(n) \leq c \cdot g(n)$

In reference textbook (CLR), for all $n>n_0$, $f(n) \leq c \cdot g(n)$
Big-O notations: Example

- $f = O(g)$ if there is a constant $c > 0$ such that for all $n$, $f(n) \leq c \cdot g(n)$
- e.g., $f(n) = 100n^2$, $g(n) = n^3$

\[
\frac{f(n)}{g(n)} = \frac{100n^2}{n^3} = \frac{100}{n} \leq 100
\]

so $f(n) = O(g(n))$, or $100n^2 = O(n^3)$

Exercise: $100n^2 + 8n = O(n^2)$

$n\log(n) = O(n^2)$

$2^n = O(3^n)$
Big-$\Omega$ notations

- Let $f(n)$ and $g(n)$ be two functions from positive integers to positive reals.
- $f=\Omega(g)$ means: $f$ grows no slower than $g$, $g$ is asymptotic lower bound of $f$.
- $f = \Omega(g)$ if and only if $g=O(f)$.
- or, if and only if there is a constant $c$, such that for all $n$, $f(n) \geq c \cdot g(n)$.

Equivalent def in CLR: there is a constant $c$, such that for all $n>n_0$, $f(n) \geq c \cdot g(n)$.
Big-Ω notations

- \( f=Ω(g) \) means: \( f \) grows no slower than \( g \), \( g \) is asymptotic lower bound of \( f \)
  - \( f = Ω(g) \) if and only if \( g=O(f) \)
  - or, if and only if there is a constant \( c \), such that for all \( n \), \( f(n) ≥ c \cdot g(n) \)

- E.g., \( f(n)=100n^2 \), \( g(n)=n \)
  so \( f(n)=Ω(g(n)) \), or \( 100n^2=Ω(n) \)

Exercise: show \( 100n^2+8n=Ω(n^2) \)
  and \( 2^n=Ω(n^8) \)
Big-$\Theta$ notations

- $f = \Theta(g)$ means: $f$ grows no slower and no faster than $g$, $f$ grows at same rate as $g$ asymptotically

- $f = \Theta(g)$ if and only if $f = O(g)$ and $f = \Omega(g)$

- i.e., there are constants $c_1, c_2 > 0$, s.t.,
  $$c_1 g(n) \leq f(n) \leq c_2 g(n),$$
  for any $n$

- Function $f$ can be sandwiched between $g$ by two constant factors
Big-$\Theta$ notations

- Show that

\[
10000n^2 = \Theta(n^2)
\]

\[
\frac{0.694c}{2}(n^2 + n - 2) + n + 3 = \Theta(n^2)
\]

\[
\log_2 n = \Theta(\log_{10} n)
\]
Summary

- In analyzing running time of algorithms, what’s important is scalability (perform well for large input)
  - Constants are not important (counting if . then ... as 1 or 2 steps does not matter)
  - Focus on higher order which dominates lower order parts
    - A three-level nested loop dominates a single-level loop
- In algorithm implementation, constants matter!
Typical Running Time Functions

• 1 (constant running time):
  – Instructions are executed once or a few times

• \( \log(n) \) (logarithmic), e.g., binary search
  – A big problem is solved by cutting original problem in smaller sizes, by a constant fraction at each step

• \( n \) (linear): linear search
  – A small amount of processing is done on each input element

• \( n \log(n) \): merge sort
  – A problem is solved by dividing it into smaller problems, solving them independently and combining the solution
Typical Running Time Functions

- $n^2$ (quadratic): bubble sort
  - Typical for algorithms that process all pairs of data items (double nested loops)

- $n^3$ (cubic)
  - Processing of triples of data (triple nested loops)

- $n^K$ (polynomial)

- $2^{0.694n}$ (exponential): Fib1

- $2^n$ (exponential):
  - Few exponential algorithms are appropriate for practical use

- $3^n$ (exponential), …
NP-Completeness

- An problem that has polynomial running time algorithm is considered \textit{tractable} (feasible to be solved efficiently)
- Not all problems are tractable
  - Some problems cannot be solved by any computer no matter how much time is provided (Turing’s Halting problem) – such problems are called \textit{undecidable}
  - Some problems can be solved in exponential time, but have no known polynomial time algorithm
- Can we tell if a problem can be solved in polynomial time?
  - NP, NP-complete, NP-hard
Summary

• algorithm running time analysis
  • start with running time function, expressing number of computer steps in terms of input size
• Focus on very large problem size, i.e., asymptotic running time
  • big-O notations => focus on dominating terms in running time function
  • Constant, linear, polynomial, exponential time algorithms …
• NP, NP complete problem
Coming up?

• Algorithm analysis is only one aspect of the class.
• We will look at different algorithms design paradigm, using problems from a wide range of domain (number, encryption, sorting, searching, graph, …)
  • First, Divide and Conquer algorithms and Master Theorem
Readings

- Chapter 0, DPV
- Chapter 3, CLR