ALGORITHMS & COMPLEXITY

Winter 2018

George Karakostas, Rm. ITB/218, karakos@mcmaster.ca

Analysis of algorithms

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

Analysis of algorithms

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

Other kinds of analysis: **average case analysis,**

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

Other kinds of analysis: **average case analysis, amortized analysis,**

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

Other kinds of analysis: **average case analysis, amortized analysis, best case analysis...**

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

Input size N: Typically the number of "atomic" objects handled by the algorithm.

Analysis of algorithms

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

Input size N: Typically the number of "atomic" objects handled by the algorithm. For example:

- For searching/sorting an array: $N = \#$ of keys n

Analysis of algorithms

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

Input size N: Typically the number of "atomic" objects handled by the algorithm. For example:

- For searching/sorting an array: N=# of keys n
- For DFS: N=[# of nodes n] + [# of edges m] *(adj. list)*

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

Input size N: Typically the number of "atomic" objects handled by the algorithm. For example:

- For searching/sorting an array: $N = \#$ of keys n
- For DFS: $N = [\#$ of nodes $n] + [\#$ of edges $m]$ *(adj. list)* **OR** $N = n^2$ *(adj. matrix)*

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

Input size N: Typically the number of "atomic" objects handled by the algorithm. For example:

- For searching/sorting an array: N=# of keys n
- For DFS: N=[# of nodes n] + [# of edges m] *(adj. list)* **OR** N=n^2 *(adj. matrix)*
- For integer multiplication alg: N=# of bits

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

Input size N: Typically the number of "atomic" objects handled by the algorithm. For example:

- For searching/sorting an array: N=# of keys n
- For DFS: N=[# of nodes n] + [# of edges m] *(adj. list)* **OR** $N=n^2$ *(adj. matrix)*
- For integer multiplication alg: N=# of bits
- etc...

Analysis of algorithms

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

Recall the tilde approximation $T(N) \sim g(N)$ from CS 2C03:

$$\lim_{N\to\infty} \frac{T(N)}{g(N)} = 1$$

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

Recall the tilde approximation $T(N) \sim g(N)$ from CS 2C03:

$$\lim_{N\to\infty} \frac{T(N)}{g(N)} = 1$$

What does this tell us? That $T(N)$ is actually of the form:

$$T(N) = g(N) + \textit{lower order terms...}$$

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

Recall the tilde approximation $T(N) \sim g(N)$ from CS 2C03:

$$\lim_{N\to\infty} \frac{T(N)}{g(N)} = 1$$

What does this tell us? That $T(N)$ *is actually of the form:*

$$T(N) = g(N) + \textit{lower order terms...}$$

so that we will have

$$\lim_{N\to\infty} \frac{T(N)}{g(N)} = \lim_{N\to\infty} \frac{g(N) + \textit{lower order terms...}}{g(N)}$$
$$= 1 + \lim_{N\to\infty} \frac{\textit{lower order terms...}}{g(N)} = 1 + 0 = 1$$

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

...i.e., we may not know $T(N)$ *exactly, but we need to guess* **exactly** *its highest order component* $g(N)$*.*

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

...i.e., we may not know $T(N)$ *exactly, but we need to guess* **exactly** *its highest order component* $g(N)$. *For example, if*

$$T(N) = 3N^2 + 20\sqrt{N} - 40N\log N,$$

Analysis of algorithms

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

...i.e., we may not know $T(N)$ *exactly, but we need to guess* **exactly** *its highest order component* $g(N)$. *For example, if*

$$T(N) = 3N^2 + 20\sqrt{N} - 40N \log N,$$

we need to guess

$$g(N) = 3N^2.$$

Analysis of algorithms

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

...i.e., we may not know $T(N)$ exactly, but we need to guess **exactly** *its highest order component $g(N)$. For example, if*

$$T(N) = 3N^2 + 20\sqrt{N} - 40N \log N,$$

we need to guess

$$g(N) = 3N^2.$$

A guess $g(N) = cN^2$, with some constant $c \neq 3$ **won't do!**

Worst case analysis We try to *estimate* the largest possible running time $T(N)$ of the algorithm over *all* inputs of size N.

...i.e., we may not know $T(N)$ exactly, but we need to guess **exactly** *its highest order component $g(N)$. For example, if*

$$T(N) = 3N^2 + 20\sqrt{N} - 40N \log N,$$

we need to guess

$$g(N) = 3N^2.$$

A guess $g(N) = cN^2$, with some constant $c \neq 3$ **won't do!**

BIG problem: What if we can guess N^2, but **not** the exact c?

Upper bounds. $T(n)$ is $O(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Analysis of algorithms

Upper bounds. $T(n)$ is $O(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Upper bounds. $T(n)$ is $O(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if it is **both** $O(f(n))$ and $\Omega(f(n))$.

Upper bounds. $T(n)$ is $O(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if it is **both** $O(f(n))$ and $\Omega(f(n))$.

- We write $T(n) = O(f(n))$, $T(n) = \Omega(f(n))$, $T(n) = \Theta(f(n))$ (abuse of notation!).

Analysis of algorithms

Upper bounds. $T(n)$ is $O(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if it is **both** $O(f(n))$ and $\Omega(f(n))$.

- We write $T(n) = O(f(n))$, $T(n) = \Omega(f(n))$, $T(n) = \Theta(f(n))$ (abuse of notation!).
- Our analysis is still asymptotic, since it holds for **large enough** n (at least as big as n_0).

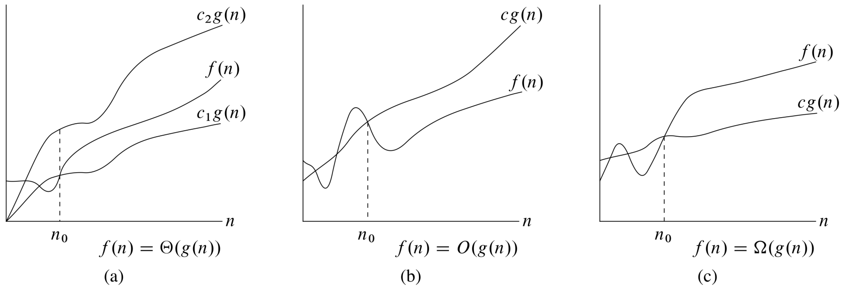
Upper bounds. $T(n)$ is $O(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if it is **both** $O(f(n))$ and $\Omega(f(n))$.

- We write $T(n) = O(f(n))$, $T(n) = \Omega(f(n))$, $T(n) = \Theta(f(n))$ (abuse of notation!).
- Our analysis is still asymptotic, since it holds for **large enough** n (at least as big as n_0).
- For input sizes $0 \leq n < n_0$ we guarantee **nothing!**

Asymptotic growth of functions



Upper bounds. $T(n)$ is $O(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if it is **both** $O(f(n))$ and $\Omega(f(n))$.

Upper bounds. $T(n)$ is $O(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if it is **both** $O(f(n))$ and $\Omega(f(n))$.

Ex: $T(n) = 32n^2 + 17n + 32$.

Asymptotic analysis of algorithms

Upper bounds. $T(n)$ is $O(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if it is **both** $O(f(n))$ and $\Omega(f(n))$.

Ex: $T(n) = 32n^2 + 17n + 32$.

- $T(n)$ is $O(n^2), O(n^3), \Omega(n^2), \Omega(n),$ and $\Theta(n^2)$.

Upper bounds. $T(n)$ is $O(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if it is **both** $O(f(n))$ and $\Omega(f(n))$.

Ex: $T(n) = 32n^2 + 17n + 32$.

- $T(n)$ is $O(n^2), O(n^3), \Omega(n^2), \Omega(n),$ and $\Theta(n^2)$.
- $T(n)$ is not $O(n), \Omega(n^3), \Theta(n),$ or $\Theta(n^3)$.

Asymptotic analysis of algorithms

Upper bounds. $T(n)$ is $O(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if it is **both** $O(f(n))$ and $\Omega(f(n))$.

Ex: $T(n) = 32n^2 + 17n + 32$.

- $T(n)$ is $O(n^2), O(n^3), \Omega(n^2), \Omega(n),$ and $\Theta(n^2)$.
- $T(n)$ is not $O(n), \Omega(n^3), \Theta(n),$ or $\Theta(n^3)$.
- $T(n) = O(1)$ means $T(n)$=constant.

Asymptotic analysis of algorithms

Upper bounds. $T(n)$ is $O(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if it is **both** $O(f(n))$ and $\Omega(f(n))$.

Ex: $T(n) = 32n^2 + 17n + 32$.

- $T(n)$ is $O(n^2), O(n^3), \Omega(n^2), \Omega(n),$ and $\Theta(n^2)$.
- $T(n)$ is not $O(n), \Omega(n^3), \Theta(n),$ or $\Theta(n^3)$.
- $T(n) = O(1)$ means $T(n)$=constant.
- Common meaningless statement:"Any comparison-based sorting algorithm **requires at least** $O(n \log n)$ comparisons!"

Asymptotic analysis of algorithms

Upper bounds. $T(n)$ is $O(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist **constants** $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if it is **both** $O(f(n))$ and $\Omega(f(n))$.

Ex: $T(n) = 32n^2 + 17n + 32$.

- $T(n)$ is $O(n^2), O(n^3), \Omega(n^2), \Omega(n),$ and $\Theta(n^2)$.
- $T(n)$ is not $O(n), \Omega(n^3), \Theta(n),$ or $\Theta(n^3)$.
- $T(n) = O(1)$ means $T(n)$=constant.
- Common meaningful statement:"Any comparison-based sorting algorithm **requires** $\Omega(n \log n)$ comparisons!"

Properties

Transitivity.

- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

Properties

Transitivity.

- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

Additivity.

- If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.
- If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.
- If $f = \Theta(h)$ and $g = O(h)$ then $f + g = \Theta(h)$.

Properties

Transitivity.

- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

Additivity.

- If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.
- If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.
- If $f = \Theta(h)$ and $g = O(h)$ then $f + g = \Theta(h)$.

Other.

- If $f = O(c \cdot h)$ for some constant c then $f = O(h)$. Same for Ω, Θ.

Comparison between O, Ω, Θ estimates and tilde estimates:

Comparison between O, Ω, Θ estimates and tilde estimates:

- Both are **asymptotic**.

Comparison between O, Ω, Θ estimates and tilde estimates:

- Both are **asymptotic**.
- O, Ω, Θ estimates are **weaker** than tilde estimates:

Asymptotic analysis of algorithms

Comparison between O, Ω, Θ estimates and tilde estimates:

- Both are **asymptotic**.
- O, Ω, Θ estimates are **weaker** than tilde estimates:

Theorem

(2.1) *Suppose that for two functions $f(n)$ and $g(n)$ we have:*

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$$

for some **constant** c. *Then* $f(n) = \Theta(cg(n)) = \Theta(g(n))$.

Comparison between O, Ω, Θ estimates and tilde estimates:

- Both are **asymptotic**.
- O, Ω, Θ estimates are **weaker** than tilde estimates:

Theorem

(2.1) *Suppose that for two functions $f(n)$ and $g(n)$ we have:*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

for some **constant** c. *Then* $f(n) = \Theta(cg(n)) = \Theta(g(n))$.

Proof: By the definition of lim. □

Asymptotic Bounds for Some Common Functions

Polynomials. $a_0 + a_1 n + \ldots + a_d n^d = \Theta(n^d)$ if $a_d > 0$.

Logarithms. $O(\log_a n) = O(\log_b n)$ for any constants $a, b > 0$.

Logarithms. For every $x > 0$, $\log n = O(n^x)$.

Exponentials. For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$.

Examples: Constant time $O(1)$

- INSERT(x,A) in an *unsorted* array A.
- FINDMIN(H) in a heap H.
- FIND(x,S) in a quick-find Union-Find structure S.

Examples: Logarithmic time $O(\log n)$

- Binary search in a *sorted* array.
- $\textsc{Search}(\textsc{x},\mathrm{T})$ in a red-black tree T.
- $\textsc{Union}$, $\textsc{Find}$ in a weighted quick-find Union-Find structure.

Examples: Linear time $O(n)$

- $\textsc{FindMax}(A)$ in an *unsorted* array A.
- Search in a hash table with chaining.
- BFS, DFS run in time $O(N) = O(n + m)$

Examples: Linearithmic time $O(n \log n)$

- Sorting. MERGESORT and HEAPSORT make $O(n \log n)$ comparisons. We have shown that no comparison-based sorting alg makes fewer than $1/2(n \log n)$, hence MERGESORT and HEAPSORT make $\Theta(n \log n)$ comparisons.
- MST takes $O(m \log n)$ time by Kruskal's or Prim's alg.
- DIJSTRA runs in $O(m + n \log n)$ time.

Examples: Quadratic time $O(n^2)$

- Multiplication of $1 \times N$ vector with $N \times N$ matrix takes $O(N^2)$ arithmetic ops.
- Quicksort makes $O(n^2)$ comparisons in the worst case. It also requires $\Omega(n \log n)$ comparisons (notice the gap).

Examples: Cubic time $O(n^3)$

- Bellman-Ford is $O(n^3)$.
- (Naive) multiplication of two $N \times N$ matrices takes $O(N^3)$ arithmetic ops.

Examples: Cubic time $O(n^3)$

- BELLMAN-FORD is $O(n^3)$.
- (Naive) multiplication of two $N \times N$ matrices takes $O(N^3)$ arithmetic ops.

  ...but can do it with $O(N^{\log_2 7})$ ops with Strassen's alg !!!

Pseudo-polynomial time

Definition

pseudo $/'s\bar{o}\bar{o}d\hat{o}/$ *adj.* not genuine; sham.

Pseudo-polynomial time

Definition

pseudo /'sōōdô/ *adj.* not genuine; sham.

Example:

INPUT: Integer n

OUTPUT: 'Yes' if n is prime

Pseudo-polynomial time

Definition

pseudo $/'s\bar{o}\bar{o}d\hat{o}/$ *adj.* not genuine; sham.

Example:
INPUT: Integer n
OUTPUT: 'Yes' if n is prime

Q: Algorithm: Divide n by $2, 3, \ldots, \sqrt{n}$; if non evenly, then 'Yes'.
Is it polynomial?

Pseudo-polynomial time

Definition

pseudo /′*sōōdô*/ *adj.* not genuine; sham.

Example:
INPUT: Integer n
OUTPUT: 'Yes' if n is prime

Q: Algorithm: Divide n by $2, 3, \ldots, \sqrt{n}$; if non evenly, then 'Yes'.
Is it polynomial?
A: **NO!** The input size is $\log_2 n$

Definition

pseudo /′*sōo͞dô*/ *adj.* not genuine; sham.

Example:

INPUT: Integer n

OUTPUT: 'Yes' if n is prime

Q: Algorithm: Divide n by $2, 3, \dots, \sqrt{n}$; if non evenly, then 'Yes'.
Is it polynomial?

A: **NO!** The input size is $\log_2 n$...and $n = 2^{\log_2 n}$ is exponential on the size of the input.

Pseudo-polynomial time

Definition

pseudo /ˈsōōdô/ *adj.* not genuine; sham.

Example:

INPUT: Integer n

OUTPUT: 'Yes' if n is prime

Q: Algorithm: Divide n by $2, 3, \ldots, \sqrt{n}$; if non evenly, then 'Yes'. Is it polynomial?

A: **NO!** The input size is $\log_2 n$...and $n = 2^{\log_2 n}$ is exponential on the size of the input.

Bottom line: Always consider the input size! (stay tuned for flow algorithms, new appreciation for Dijkstra, Kruskal, Prim...)

Examples: Exponential time

Find a clique of size k in an undirected graph.
INPUT: Graph $G = (V, E)$, an integer k
OUTPUT: 'Yes' if there is a clique with k nodes

Examples: Exponential time

Find a clique of size k in an undirected graph.
INPUT: Graph $G = (V, E)$, an integer k
OUTPUT: 'Yes' if there is a clique with k nodes

Brute force: Try all $\binom{n}{k}$ subsets of V; if clique found, output 'Yes'

Examples: Exponential time

Find a clique of size k in an undirected graph.
INPUT: Graph $G = (V, E)$, an integer k
OUTPUT: 'Yes' if there is a clique with k nodes

Brute force: Try all $\binom{n}{k}$ subsets of V; if clique found, output 'Yes'
Running time: About $O(\binom{n}{k}) = O(k^2 \cdot n^k / k!) = O(n^k)$.

Examples: Exponential time

Find a clique of size k in an undirected graph.
INPUT: Graph $G = (V, E)$, an integer k
OUTPUT: 'Yes' if there is a clique with k nodes

Brute force: Try all $\binom{n}{k}$ subsets of V; if clique found, output 'Yes'
Running time: About $O(\binom{n}{k}) = O(k^2 \cdot n^k / k!) = O(n^k)$.

In general, our algorithms search a huge (e.g., exponential on the size of the input) space for a solution; therefore, brute force searching takes exponential time (worst case).

Examples: Exponential time

Find a clique of size k in an undirected graph.
INPUT: Graph $G = (V, E)$, an integer k
OUTPUT: 'Yes' if there is a clique with k nodes

Brute force: Try all $\binom{n}{k}$ subsets of V; if clique found, output 'Yes'
Running time: About $O(\binom{n}{k}) = O(k^2 \cdot n^k / k!) = O(n^k)$.

In general, our algorithms search a huge (e.g., exponential on the size of the input) space for a solution; therefore, brute force searching takes exponential time (worst case).

Big complexity problem: For many problems, our currently best is brute force. *Can we do better?*

The complexity class P

Efficient algorithms: Algorithms that run in polynomial time $O(n^d)$ are much better than brute force, and the only practical(?) ones (especially when the degree d is small, usually smaller than 3).

The complexity class P

Efficient algorithms: Algorithms that run in polynomial time $O(n^d)$ are much better than brute force, and the only practical(?) ones (especially when the degree d is small, usually smaller than 3).

Definition

A **complexisy class** is a set of problems.

The complexity class P

Efficient algorithms: Algorithms that run in polynomial time $O(n^d)$ are much better than brute force, and the only practical(?) ones (especially when the degree d is small, usually smaller than 3).

Definition

A **complexisy class** is a set of problems.

Example: All problems that can be solved by an algorithm belong to the class of **Decidable** problems. The HALTING problem doesn't belong to this class.

The complexity class P

Efficient algorithms: Algorithms that run in polynomial time $O(n^d)$ are much better than brute force, and the only practical(?) ones (especially when the degree d is small, usually smaller than 3).

Definition

A **complexisy class** is a set of problems.

Example: All problems that can be solved by an algorithm belong to the class of **Decidable** problems. The HALTING problem doesn't belong to this class.

Definition

P is the class of all problems that can be solved by a polynomial algorithm.

The complexity class P

Efficient algorithms: Algorithms that run in polynomial time $O(n^d)$ are much better than brute force, and the only practical(?) ones (especially when the degree d is small, usually smaller than 3).

Definition

A **complexisy class** is a set of problems.

Example: All problems that can be solved by an algorithm belong to the class of **Decidable** problems. The HALTING problem doesn't belong to this class.

Definition

P is the class of all problems that can be solved by a polynomial algorithm.

Example: Essentially all the problems we studied in CS 2C03 belong in **P**.

The complexity class *P*

Efficient algorithms: Algorithms that run in polynomial time $O(n^d)$ are much better than brute force, and the only practical(?) ones (especially when the degree d is small, usually smaller than 3).

Definition

A **complexisy class** is a set of problems.

Example: All problems that can be solved by an algorithm belong to the class of **Decidable** problems. The HALTING problem doesn't belong to this class.

Definition

P is the class of all problems that can be solved by a polynomial algorithm.

Example: Essentially all the problems we studied in CS 2C03 belong in **P**. This is no coincidence: **P** is the set of problems that can be solved efficiently.

Why it matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	4 sec
$n = 30$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	18 min	10^{25} years
$n = 50$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	11 min	36 years	very long
$n = 100$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	$<$ 1 sec	$<$ 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	$<$ 1 sec	$<$ 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	$<$ 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

A typical algorithm:

How to analyze algorithms

A typical algorithm:

1. Algorithm makes some decision(s)

How to analyze algorithms

A typical algorithm:

1. Algorithm makes some decision(s)
2. Problem is broken into k subproblems $(n_1, \ldots, n_k)$

A typical algorithm:

1. Algorithm makes some decision(s)
2. Problem is broken into k subproblems $(n_1, \ldots, n_k)$
3. Solve k subproblems recursively

How to analyze algorithms

A typical algorithm:

1. Algorithm makes some decision(s)
2. Problem is broken into k subproblems $(n_1, \ldots, n_k)$
3. Solve k subproblems recursively
4. Algorithm combines decision(s) from (1) with solutions from (3), to output solution

How to analyze algorithms

A typical algorithm:

1. Algorithm makes some decision(s)
2. Problem is broken into k subproblems $(n_1, \ldots, n_k)$
3. Solve k subproblems recursively
4. Algorithm combines decision(s) from (1) with solutions from (3), to output solution

Recurrence: $T(n) = [\text{work done in (1),(2),(4)}] + \sum_{i=1}^{k} T(n_i)$

Divide-and-conquer algorithms

A typical D&C algorithm:

1. Algorithm makes some decision(s)
2. Divide: Problem is broken into k subproblems $(n_1, \ldots, n_k)$
3. Conquer: Solve k subproblems recursively
4. Algorithm combines solutions from (3), to output solution

Divide-and-conquer algorithms

A typical D&C algorithm:

1. Algorithm makes some decision(s)
2. Divide: Problem is broken into k subproblems $(n_1, \ldots, n_k)$
3. Conquer: Solve k subproblems recursively
4. Algorithm combines solutions from (3), to output solution

Recurrence: $T(n) = [\text{work done in (2),(4)}] + \sum_{i=1}^{k} T(n_i)$

Divide-and-conquer algorithms

Example of D&C: MergeSort

1. Algorithm makes some decision(s)
2. $\textsc{MergeSort}(A[1..\frac{n}{2}])$, $\textsc{MergeSort}(A[\frac{n}{2}..n])$
3. $\textsc{Merge}(A[1..n/2], A[n/2..n])$

Example of D&C: MergeSort

1. Algorithm makes some decision(s)
2. $\textsc{MergeSort}(A[1..\frac{n}{2}])$, $\textsc{MergeSort}(A[\frac{n}{2}..n])$
3. $\textsc{Merge}(A[1..n/2], A[n/2..n])$

Recurrence: $T(n) \leq cn + 2T(n/2)$, $T(1) = 0$

First method: unrolling the recurrence

Try to find the recurrence pattern by *unrolling* it:

$$
\begin{aligned}
T(n) &\leq cn + 2T(n/2) && \text{(level 1)}\\
&\leq cn + (2c(n/2) + 4T(n/4)) = 2cn + 4T(n/4) && \text{(level 2)}\\
&\leq 2cn + (4c(n/4) + 8T(n/8)) = 3cn + 8T(n/8) && \text{(level 3)}\\
&\dots\\
&\leq kcn + 2^k T(n/2^k) && \text{(level } k\text{)}\\
&\dots\\
&\leq (\log n)cn + 2^{\log n} T(n/2^{\log n}) = cn \log n && \text{(level } \log n\text{)}
\end{aligned}
$$

First method: unrolling the recurrence

Try to find the recurrence pattern by *unrolling* it:

$$
\begin{aligned}
T(n) &\leq cn + 2T(n/2) && \text{(level 1)}\\
&\leq cn + (2c(n/2) + 4T(n/4)) = 2cn + 4T(n/4) && \text{(level 2)}\\
&\leq 2cn + (4c(n/4) + 8T(n/8)) = 3cn + 8T(n/8) && \text{(level 3)}\\
&\ldots\\
&\leq kcn + 2^k T(n/2^k) && \text{(level } k\text{)}\\
&\ldots\\
&\leq (\log n)cn + 2^{\log n} T(n/2^{\log n}) = cn \log n && \text{(level } \log n\text{)}
\end{aligned}
$$

Theorem

$T(n) = O(n \log n)$

Solving recurrences

Second method: substitution

1. Try to guess the recurrence solution
2. Prove that substituting your guess for the recurrence verifies the guess

Solving recurrences

Second method: substitution

1. Try to guess the recurrence solution
2. Prove that substituting your guess for the recurrence verifies the guess

Example: $T(n) \leq cn + 2T(n/2)$, $T(1) = 0$

Second method: substitution

1. Try to guess the recurrence solution
2. Prove that substituting your guess for the recurrence verifies the guess

Example: $T(n) \leq cn + 2T(n/2)$, $T(1) = 0$

1. We guess that $T(n) = O(n \log n)$, i.e., $T(n) \leq kn \log n$ for some constant k=(?)

Second method: substitution

1. Try to guess the recurrence solution
2. Prove that substituting your guess for the recurrence verifies the guess

Example: $T(n) \leq cn + 2T(n/2)$, $T(1) = 0$

1. We guess that $T(n) = O(n \log n)$, i.e., $T(n) \leq kn \log n$ for some constant k=(?)
2. $$\begin{aligned} T(n) &\leq cn + 2T(n/2) \\ &\leq cn + 2k(n/2)\log(n/2) \\ &= cn + kn(\log n - 1) \\ &= kn\log n + cn - kn \\ &\leq kn \log n \end{aligned}$$

...provided we pick a $k > c$.

Solving recurrences

Theorem (Master Theorem)

Let $a \geq 1, b > 1$ be constants, and

$$T(n) = aT(n/b) + f(n)$$

1. $f(n) = O(n^{\log_b a - \varepsilon})$ *for constant* $\varepsilon > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$
3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ *for constant* $\varepsilon > 0$ *and* $af(n/b) \leq cf(n)$ *for some constant* $c < 1 \Rightarrow T(n) = \Theta(f(n))$

Solving recurrences

Theorem (**Master Theorem**)

Let $a \geq 1, b > 1$ *be constants, and*

$$T(n) = aT(n/b) + f(n)$$

1. $f(n) = O(n^{\log_b a - \varepsilon})$ *for constant* $\varepsilon > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$
3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ *for constant* $\varepsilon > 0$ *and* $af(n/b) \leq cf(n)$ *for some constant* $c < 1 \Rightarrow T(n) = \Theta(f(n))$

Examples

- $T(n) = 2T(n/2) + \Theta(n)$

Solving recurrences

Theorem (Master Theorem)

Let $a \geq 1, b > 1$ be constants, and

$$T(n) = aT(n/b) + f(n)$$

1. $f(n) = O(n^{\log_b a - \varepsilon})$ *for constant* $\varepsilon > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$
3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ *for constant* $\varepsilon > 0$ *and* $af(n/b) \leq cf(n)$ *for some constant* $c < 1 \Rightarrow T(n) = \Theta(f(n))$

Examples

- $T(n) = 2T(n/2) + \Theta(n)$
  $a = 2, b = 2, f(n) = \Theta(n) = \Theta(n^{log_2 2})$
  $\Rightarrow T(n) = \Theta(n^{log_2 2} \log n) = \Theta(n \log n)$ (Case 2)

Theorem (Master Theorem)

Let $a \geq 1, b > 1$ be constants, and

$$T(n) = aT(n/b) + f(n)$$

1. $f(n) = O(n^{\log_b a-\varepsilon})$ *for constant* $\varepsilon > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$
3. $f(n) = \Omega(n^{\log_b a+\varepsilon})$ *for constant* $\varepsilon > 0$ *and* $af(n/b) \leq cf(n)$ *for some constant* $c < 1 \Rightarrow T(n) = \Theta(f(n))$

Examples

- $T(n) = T(2n/3) + \Theta(1)$

Theorem (Master Theorem)

Let $a \geq 1, b > 1$ be constants, and

$$T(n) = aT(n/b) + f(n)$$

1. $f(n) = O(n^{\log_b a - \varepsilon})$ *for constant* $\varepsilon > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$
3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ *for constant* $\varepsilon > 0$ *and* $af(n/b) \leq cf(n)$ *for some constant* $c < 1 \Rightarrow T(n) = \Theta(f(n))$

Examples

- $T(n) = T(2n/3) + \Theta(1)$
  $a = 1, b = 3/2, f(n) = \Theta(1) = \Theta(n^{\log_{3/2} 1})$
  $\Rightarrow T(n) = \Theta(n^{\log_{3/2} 1} \log n) = \Theta(\log n)$ (Case 2)

Theorem (Master Theorem)

Let $a \geq 1, b > 1$ be constants, and

$$T(n) = aT(n/b) + f(n)$$

1. $f(n) = O(n^{\log_b a - \varepsilon})$ *for constant* $\varepsilon > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$
3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ *for constant* $\varepsilon > 0$ *and* $af(n/b) \leq cf(n)$ *for some constant* $c < 1 \Rightarrow T(n) = \Theta(f(n))$

Examples

- $T(n) = 3T(n/4) + \Theta(n \log n)$

Solving recurrences

Theorem (Master Theorem)

Let $a \geq 1, b > 1$ be constants, and

$$T(n) = aT(n/b) + f(n)$$

1. $f(n) = O(n^{\log_b a - \varepsilon})$ *for constant* $\varepsilon > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$
3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ *for constant* $\varepsilon > 0$ *and* $af(n/b) \leq cf(n)$ *for some constant* $c < 1 \Rightarrow T(n) = \Theta(f(n))$

Examples

- $T(n) = 3T(n/4) + \Theta(n \log n)$ $a = 3, b = 4, f(n) = n \log n = \Omega(n^{\log_4 3 + 0.2})$, $af(n/b) = 3(n/4)\log(n/4) \leq n \log n = 1 \cdot f(n)$

Theorem (Master Theorem)

Let $a \geq 1, b > 1$ be constants, and

$$T(n) = aT(n/b) + f(n)$$

1. $f(n) = O(n^{\log_b a-\varepsilon})$ *for constant* $\varepsilon > 0 \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$
3. $f(n) = \Omega(n^{\log_b a+\varepsilon})$ *for constant* $\varepsilon > 0$ *and* $af(n/b) \leq cf(n)$ *for some constant* $c < 1 \Rightarrow T(n) = \Theta(f(n))$

Examples

- $T(n) = 3T(n/4) + \Theta(n \log n)$ $a = 3, b = 4, f(n) = n \log n = \Omega(n^{log_4 3+0.2}), af(n/b) = 3(n/4)\log(n/4) \leq n \log n = 1 \cdot f(n)$ $\Rightarrow T(n) = \Theta(n \log n)$ (Case 3)

Divide-and-conquer algorithms

A typical D&C algorithm:

1. Algorithm makes some decision(s)
2. Divide: Problem is broken into k subproblems $(n_1, \ldots, n_k)$
3. Conquer: Solve k subproblems recursively
4. Algorithm combines solutions from (3), to output solution

A typical D&C algorithm:

1. Algorithm makes some decision(s)
2. Divide: Problem is broken into k subproblems $(n_1, \ldots, n_k)$
3. Conquer: Solve k subproblems recursively
4. Algorithm combines solutions from (3), to output solution

Other examples of D&C: QuickSort, counting inversions, integer multiplication, closest points, ...