BLG 336E

Analysis of Algorithms II

Lecture 3:

Growth of Functions, Asymptomatic Analysis, Runtimes, Big-O Notation, Theta, Omega

Stable Matching Problem

Q. Is assignment X-C, Y-B, Z-A stable?

	favorite 1st	2nd	least favorite 3rd
Xavier	Amy	Bertha	Clare
Yancey	Bertha	Amy	Clare
Zeus	Amy	Bertha	Clare

Men's Preference Profile

	favorite 1st	2nd	least favorite 3rd
Amy	Yancey	Xavier	Zeus
Bertha	Xavier	Yancey	Zeus
Clare	Xavier	Yancey	Zeus

Women's Preference Profile

Stable Matching Problem

Q. Is assignment X-C, Y-B, Z-A stable?

A. No. Bertha and Xavier will hook up.

	1st (favorite)	2nd	3rd (least favorite)
Xavier	Amy	Bertha	Clare
Yancey	Bertha	Amy	Clare
Zeus	Amy	Bertha	Clare

Men's Preference Profile

	1st (favorite)	2nd	3rd (least favorite)
Amy	Yancey	Xavier	Zeus
Bertha	Xavier	Yancey	Zeus
Clare	Xavier	Yancey	Zeus

Women's Preference Profile

Stable Matching Problem

Q. Is assignment X-A, Y-B, Z-C stable?

A. Yes.

	1st (favorite)	2nd	3rd (least favorite)
Xavier	Amy	Bertha	Clare
Yancey	Bertha	Amy	Clare
Zeus	Amy	Bertha	Clare

Men's Preference Profile

	1st (favorite)	2nd	3rd (least favorite)
Amy	Yancey	Xavier	Zeus
Bertha	Xavier	Yancey	Zeus
Clare	Xavier	Yancey	Zeus

Women's Preference Profile

Algorithm Analysis

- Thinking about how the resource requirements of the algorithms will scale with increasing input size.

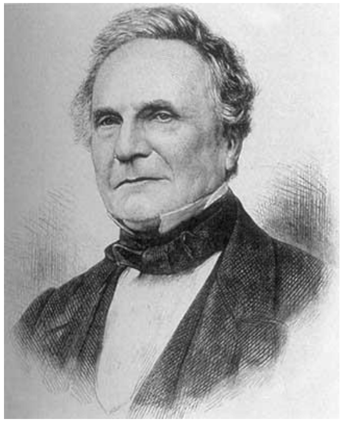
- Resources:
  - time
  - space

Computational efficiency. Efficiency in running time.

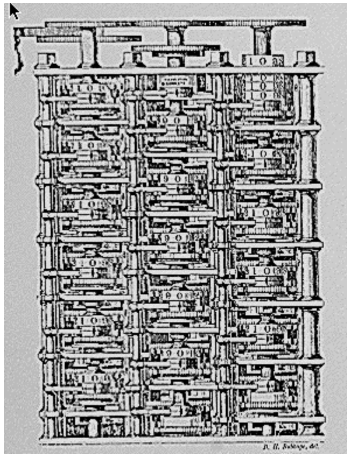
We want algorithms that run quickly!

Computational Tractability

> As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time? - *Charles Babbage*



Charles Babbage (1864)



Analytic Engine (schematic)

How to measure complexity?

- Accurate running time is not a good measure
- It depends on **input**
- It depends on the **machine** you used and who implemented the algorithm

- We would like to have an analysis that **does not depend** on those factors

FROM AoA I!!

Machine-independent

- A generic uniprocessor random-access machine (RAM) model
  - No concurrent operations
  - Each simple operation (e.g. +, -, =, *, if, for) takes 1 step.
    - Loops and subroutine calls are *not* simple operations.
  - All memory equally expensive to access
    - Constant word size
    - Unless we are explicitly manipulating bits
    - No memory hierarch (caches, virtual mem) is modeled

FROM AoA I!!

Running Time

- **Running Time:T(n):** Number of primitive operations or steps executed for an input of size n.
- Running time depends on input
  - already sorted sequence is easier to sort
- Parameterize running time by size of input
  - short sequences are easier to sort than long ones
- Generally, we seek upper bounds on running time
  - everybody likes a guarantee

FROM AoA I!!

Kinds of Analysis

- **Worst-case:** (usually)
  - T(*n*) = maximum time of algorithm on any input of size *n*
- **Average-case:** (sometimes)
  - T(*n*) = expected time of algorithm over all inputs of size *n*
  - Need assumption about statistical distribution of inputs
- **Best-case:** (difficult to determine)
  - Cheat with a slow algorithm that works fast on some input

FROM AoA I!!

Worst-Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size N.

- Generally captures efficiency in practice.
- Draconian view**, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size N.

- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.

**1. Designating a law or code of extreme severity. 2. Harsh, rigorous.

Brute-Force Search

Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.

- Typically takes 2^N time or worse for inputs of size N.
- Unacceptable in practice.

n! for stable matching with n men and n women

- Not only too slow to be useful, it is an intellectual cop-out.
- Provides us with absolutely no insight into the structure of the problem.

Proposed definition of efficiency. An algorithm is efficient if it achieves qualitatively better worst-case performance than brute-force search.

Exhaustive Search is Slow!

- Because it tries all n! permutations, it is much slower to use when there are more than 10-20 points. [For n=1000, will not be achieved in your lifetime!]

- **No efficient, correct algorithm** exists for the travelling salesman problem.

- Conclusions: The example shows the importance of the 4 questions we ask about algorithms. It also shows that they are very much related!

1. **How to devise algorithms?**
2. **How to validate/verify algorithms?**
3. **How to analyze algorithms?**
4. **How to test a program?**

Polynomial-Time

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor C.

> There exists constants $c > 0$ and $d > 0$ such that on every input of size N, its running time is bounded by cN^d steps.

A step. a single assembly-language instruction, one line of a programming language like C…

What happens if the input size increases from N to 2N?

Def. An algorithm is poly-time if the above scaling property holds.

Polynomial-Time

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor C.

> There exists constants $c > 0$ and $d > 0$ such that on every input of size N, its running time is bounded by cN^d steps.

A step. a single assembly-language instruction, one line of a programming language like C…

What happens if the input size increases from N to $2N$?
Answer: runningtime$=c(2N)^d = c2^dN^d = O(N^d)$ (because d is const)

Def. An algorithm is poly-time if the above scaling property holds.

Worst-Case Polynomial-Time

Def. An algorithm is efficient if its running time is polynomial.

Justification: It really works in practice!

- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

Exceptions.

- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

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

Note: Stirling's approximation: $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

Asymptotic Order of Growth

- We try to express that an algorithm's worst case running time is at most proportional to some function f(n).
- Function f(n) becomes a bound on the running time of the algorithm.

- Pseudo-code style.
  - counting the number of pseudo-code steps.
  - step. Assigning a value to a variable, looking up an entry in an array, following a pointer, a basic arithmetic operation…

"On any input size n, the algorithm runs for at most $1.62n^2 + 3.5n + 8$ steps."

Do we need such precise bound?

We would like to classify running times at a coarser level of granularity.
- Similarities show up more clearly.

Big-Oh Notation

Importance: Vocabulary for the design and analysis of algorithms (e.g. "big-Oh" notation).

- "Sweet spot" for high-level reasoning about algorithms.
- Coarse enough to suppress architecture/language/compiler-dependent details.
- Sharp enough to make useful comparisons between different algorithms, especially on large inputs (e.g. sorting or integer multiplication).

Asymptotic Analysis

High-level idea: Suppress constant factors and lower-order terms

too system-dependent

irrelevant for large inputs

Example: Equate $6n \log_2 n + 6$ with just $n \log n$.

Terminology: Running time is $O(n \log n)$

["big-Oh" of $n \log n$]

where n = input size (e.g. length of input array).

Example: One-Loop

Problem: Does array A contain the integer t? Given A (array of length n) and t (an integer).

Algorithm 1

```
for i = 1 to n do
    if A[i] == t then
        Return TRUE
Return FALSE
```

Question: What is the running time?

A. O(1)
B. O(n) ✓
C. O(logn)
D. O(n^2)

Example: Two-Loops

Given A, B (arrays of length n) and t (an integer). [Does A or B contain t?]

Algorithm 2

```
for i = 1 to n do
    if A[i] == t then
        Return TRUE
for i = 1 to n do
    if B[i] == t then
        Return TRUE
Return FALSE
```

Question: What is the running time?

A. O(1)
B. O(n)
C. O(logn)
D. O(n^2)

Example: Two Nested Loops

Problem: Do arrays A, B have a number in common? Given arrays A, B of length n.

Algorithm 3

```
for i = 1 to n do
    for j = 1 to n do
        if A[i] == B[j] then
            Return TRUE
Return FALSE
```

Question: What is the running time?

A. $O(1)$
B. $O(n)$
C. $O(\log n)$
D. $O(n^2)$

Example: Two Nested Loops II

Problem: Does array A have duplicate entries? Given arrays A of length n.

Algorithm 4

for $i = 1$ to n **do**
  for $j =$ i+1 to n **do**
    if $A[i] ==$ A $[j]$ **then**
      Return TRUE
Return FALSE

Question: What is the running time?

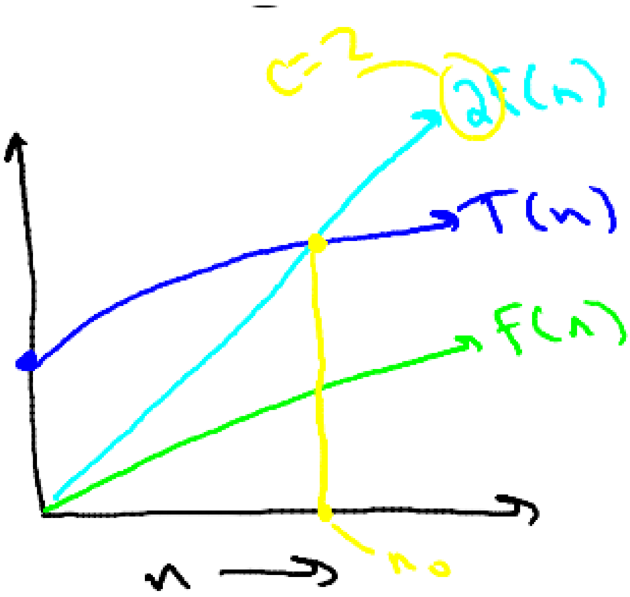
A. O(1)
B. O(n)
C. O(logn)
D. $O(n^2)$

Big-Oh Definition

Let T(n) = function on n = 1,2,3,...
[usually, the worst-case running time of an algorithm]

Q : When is T(n) = O(f(n)) ?
A : if eventually (for all sufficiently large n), T(n) is bounded above by a constant multiple of f(n)

Big-Oh formal Definition



$Picture\ \ T(n) = O(f(n))$

<u>Formal Definition</u> : T(n) = O(f(n)) if and only if there exist constants $c, n_0 > 0$ such that

$$T(n) \leq c \cdot f(n)$$

For all $n \geq n_0$

<u>Warning</u> : c, n_0 cannot depend on n

Example 1

<u>Claim</u> : if $T(n) = a_k n^k + ... + a_1 n + a_0$ then

$$T(n) = O(n^k)$$

<u>Proof</u> : Choose $n_0 = 1$ and $c = |a_k| + |a_{k-1}| + .. + |a_1| + |a_0|$

Need to show that $\forall n \geq 1, T(n) \leq c \cdot n^k$

We have, for every $n \geq 1,$

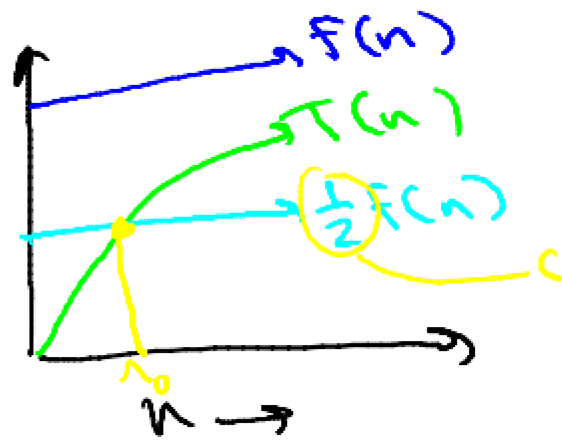
$$\begin{aligned} T(n) &\leq |a_k| n^k + ... + |a_1| n + |a_0| \\ &\leq |a_k| n^k + ... + |a_1| n^k + |a_0| n^k \\ &= c \cdot n^k \end{aligned}$$

Omega Notation

<u>Definition</u> : $T(n) = \Omega(f(n))$
If and only if there exist constants c, n_0 such that

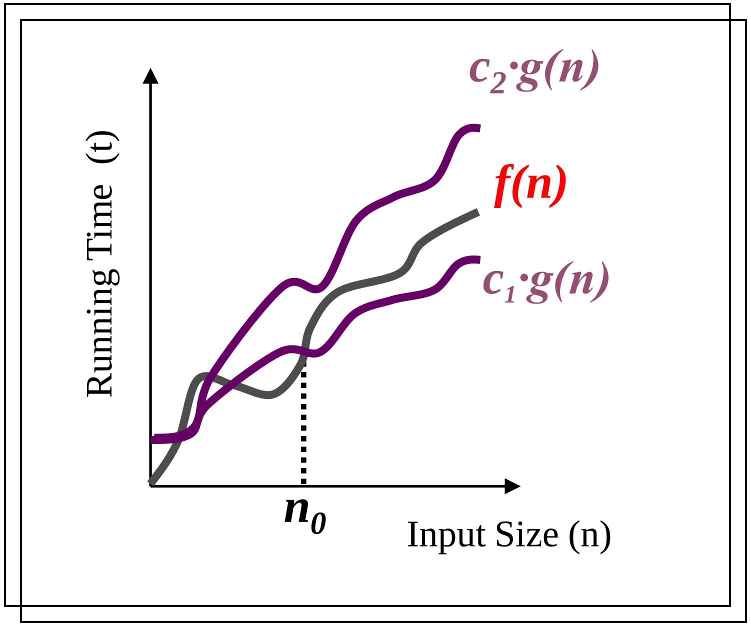
$$T(n) \geq c \cdot f(n) \quad \forall n \geq n_0 .$$

Picture



$$T(n) = \Omega(f(n))$$

Theata Notation



Definition : $T(n) = \theta(f(n))$ if and only if
$T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

Equivalent : there exist constants c_1, c_2, n_0 such that

$$c_1 f(n) \leq T(n) \leq c_2 f(n)$$

$$\forall n \geq n_0$$

Little o: Definition

- For a given function g(n), o(g(n)) is the set of functions:

$$o(g(n)) = \{f(n): \text{for any positive constant } c, \text{ there exists a constant } n_0 \text{ such that } 0 \le f(n) < c\, g(n) \text{ for all } n \ge n_0\}$$

Little o

- Note the < instead of ≤ in the definition of Little-o:

  $0 \leq f(n) < c\, g(n)$ for all $n \geq n_0$

- Contrast this to the definition used for Big-O:

  $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0$

- Little-o notation denotes an *upper bound that is not asymptotically tight*. We might call this a *loose* upper bound.

- Examples: $2n \in o(n^2)$ but $2n^2 \notin o(n^2)$

Little o: Definition

- Given that $f(n) = o(g(n))$, we know that g grows *strictly faster* than f. This means that you can multiply g by a positive constant c and beyond n_0, g will always exceed f.
- No graph to demonstrate little-o, but here is an example:

  $n^2 = o(n^3)$ but

  $n^2 \neq o(n^2)$.

  Why? Because if c = 1, then $f(n) = c\ g(n)$, and the definition insists that $f(n)$ be less than $c\ g(n)$.

Little omega: Definition

- For a given function g(n), ω(g(n)) is the set of functions:

ω(g(n))= {f(n): for any positive constant *c,*

there exists a constant n_0

such that $0 \le c\, g(n) < f(n)$

for all $n \geq n_0$ }

Little omega: Definition

- Note the < instead of ≤ in the definition:

  $0 \le c\, g(n) < f(n)$

- Contrast this to the definition used for Big-Ω:

  $0 \le c\, g(n) \le f(n)$

- Little-omega notation denotes a *lower bound that is not asymptotically tight.* We might call this a *loose* lower bound.

- Examples:

  $n \notin \omega(n^2)$ $n \in \omega(\sqrt{n})$ $n \in \omega(\lg n)$

Little omega

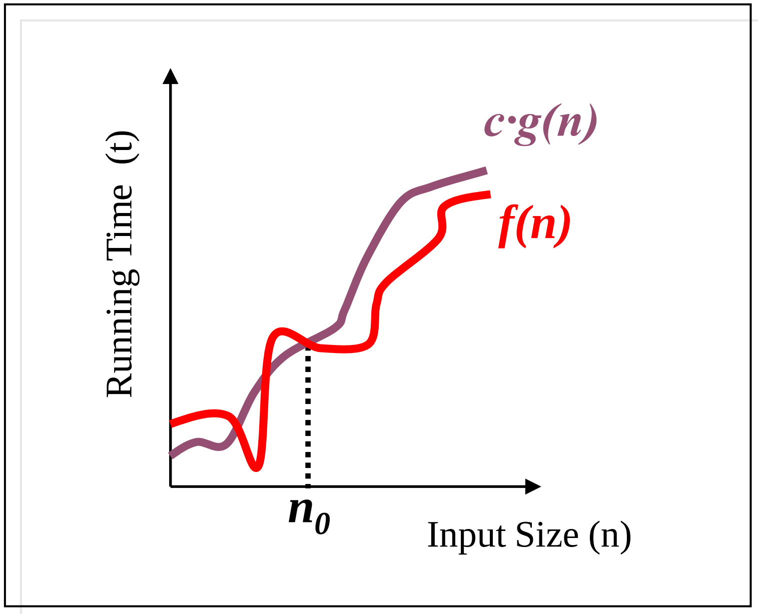
- No graph to demonstrate little-omega, but here is an example:

  n^3 is $\omega(n^2)$ but

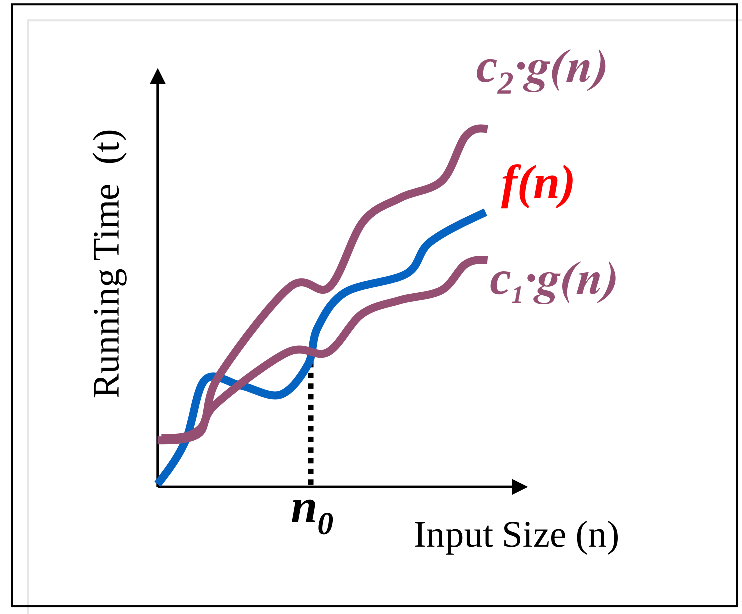
  $n^3 \neq \omega(n^3)$.

  Why? Because if c = 1, then $f(n) = c\ g(n)$, and the definition insists that $c\ g(n)$ be strictly <u>less than</u> $f(n)$.

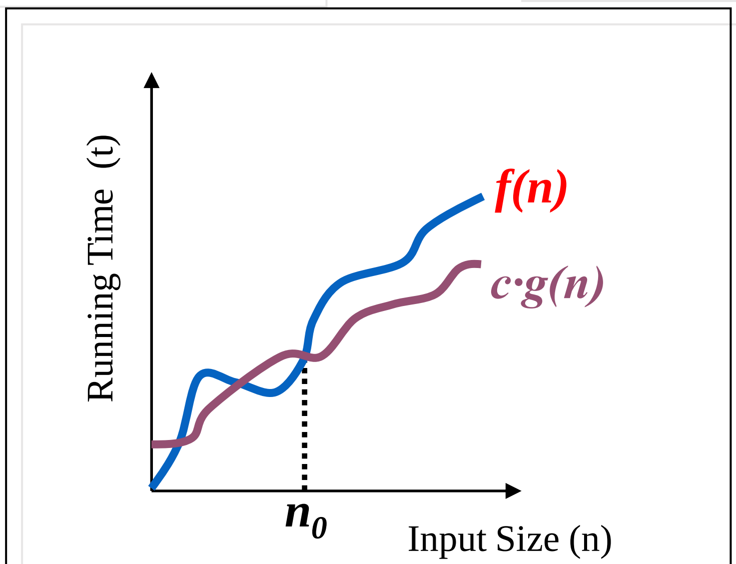
Asymptotic Notation



Big O



Big Theta



Big Omega

Example

Let $T(n) = \frac{1}{2}n^2 + 3n$. Which of the following statements are true ? (Check all that apply.)

- ☐ $T(n) = O(n)$.
- ☐ $T(n) = \Omega(n)$. $[n_0 = 1, c = 1/2]$
- ☐ $T(n) = \Theta(n^2)$. $[n_0 = 1, c_1 = 1/2, c_2 = 4]$
- ☐ $T(n) = O(n^3)$. $[n_0 = 1, c = 4]$

Where does this notation come from??

- "On the basis of the issues discussed here, I propose that members of SIGACT, and editors of compter science and mathematics journals, adopt the O, Ω, and Θ notations as defined above, unless a better alternative can be found reasonably soon".

-D. E. Knuth, "Big Omicron and Big Omega and Big Theta", SIGACT News, 1976. Reprinted in "Selected Papers on Analysis of Algorithms."

Asymptotic Order of Growth

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 $T(n)$ 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 $\Theta(n^3)$.

Asymptotically Tight Bounds

How to prove that f(n) is $\Theta(g(n))$?

Answer:

If the following limit exists and is equal to a constant c>0, then f(n) is $\Theta(g(n))$.

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

Or

show that f(n) is both $O(g(n))$ and $\Omega(g(n))$

Notation

Slight abuse of notation. $T(n) = O(f(n))$.

- Asymmetric:
  - $f(n) = 5n^3$; $g(n) = 3n^2$
  - $f(n) = O(n^3) = g(n)$
  - but $f(n) \neq g(n)$.
- Better notation: $T(n) \in O(f(n))$.

Meaningless statement. Any comparison-based sorting algorithm requires at least $O(n \log n)$ comparisons.

- Statement doesn't "type-check."
- Use Ω for lower bounds.

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 $g = \Theta(h)$ then $f + g = \Theta(h)$.

Please see the proofs in the book pp. 39+.

Asymptotic Bounds for Some Common Functions

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

Polynomial time. Running time is $O(n^d)$ for some constant d independent of the input size n. (even if d is not an integer.)
(Note that running time is also $\Theta(n^d)$. See book for the proof.)

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

can avoid specifying the base

Very slowly growing functions.

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

log grows slower than every polynomial

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

every exponential grows faster than every polynomial

2.4 A Survey of Common Running Times

Linear Time: O(n)

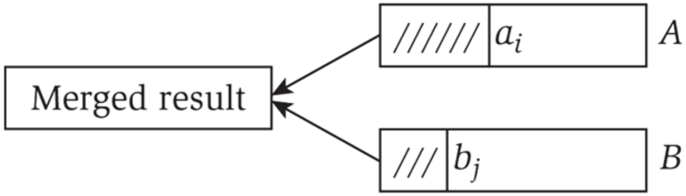
Linear time. Running time is at most a constant factor times the size of the input.

Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.

```
max ← a1
for i = 2 to n {
   if (ai > max)
      max ← ai
}
```

Linear Time: O(n)

Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.




```
i = 1, j = 1
while (both lists are nonempty) {
   if (a_i ≤ b_j) append a_i to output list and increment i
   else           append b_j to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size n takes O(n) time.

Pf. After each comparison, the length of output list increases by 1.

O(n log n) Time

O(n log n) time. Arises in divide-and-conquer algorithms.

also referred to as linearithmic time

Sorting. Mergesort and heapsort are sorting algorithms that perform O(n log n) comparisons.

Largest empty interval. Given n time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

Quadratic Time: $O(n^2)$

Quadratic time. Enumerate all pairs of elements.

Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.

$O(n^2)$ solution. Try all pairs of points.

```
min ← (x₁ - x₂)² + (y₁ - y₂)²
for i = 1 to n {
   for j = i+1 to n {
      d ← (xᵢ - xⱼ)² + (yᵢ - yⱼ)²
      if (d < min)
         min ← d
   }
}
```

← don't need to take square roots

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion. ← see chapter 5

Cubic Time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Given n sets $S_1, \ldots, S_n$ each of which is a subset of 1, 2, …, n, is there some pair of these which are disjoint?

$O(n^3)$ solution. For each pairs of sets, determine if they are disjoint.

```
foreach set Si {
   foreach other set Sj {
      foreach element p of Si {
         determine whether p also belongs to Sj
      }
      if (no element of Si belongs to Sj)
         report that Si and Sj are disjoint
   }
}
```

Polynomial Time: $O(n^k)$ Time

Independent set of size k. Given a graph, are there k nodes such that no two are joined by an edge?

k is a constant

$O(n^k)$ solution. Enumerate all subsets of k nodes.

```
for each subset S of k nodes {
   check whether S in an independent set
   if (S is an independent set)
      report S is an independent set
   }
}
```

- Check whether S is an independent set = $O(k^2)$.
- Number of k element subsets = $\binom{n}{k} = \frac{n\,(n-1)\,(n-2)\cdots(n-k+1)}{k\,(k-1)\,(k-2)\cdots(2)\,(1)} \le \frac{n^k}{k!}$
- $O(k^2\, n^k / k!) = O(n^k)$.

poly-time for k=17,
but not practical

Exponential Time

Independent set. Given a graph, what is maximum size of an independent set?

```
S* ← φ
for each subset S of nodes {
   check whether S in an independent set
   if (S is largest independent set seen so far)
      update S* ← S
   }
}
```

$O(n^2\ 2^n)$ solution. Enumerate all subsets.

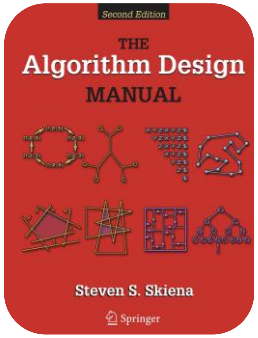
Total number of subsets: 2^n.

Sublinear Time

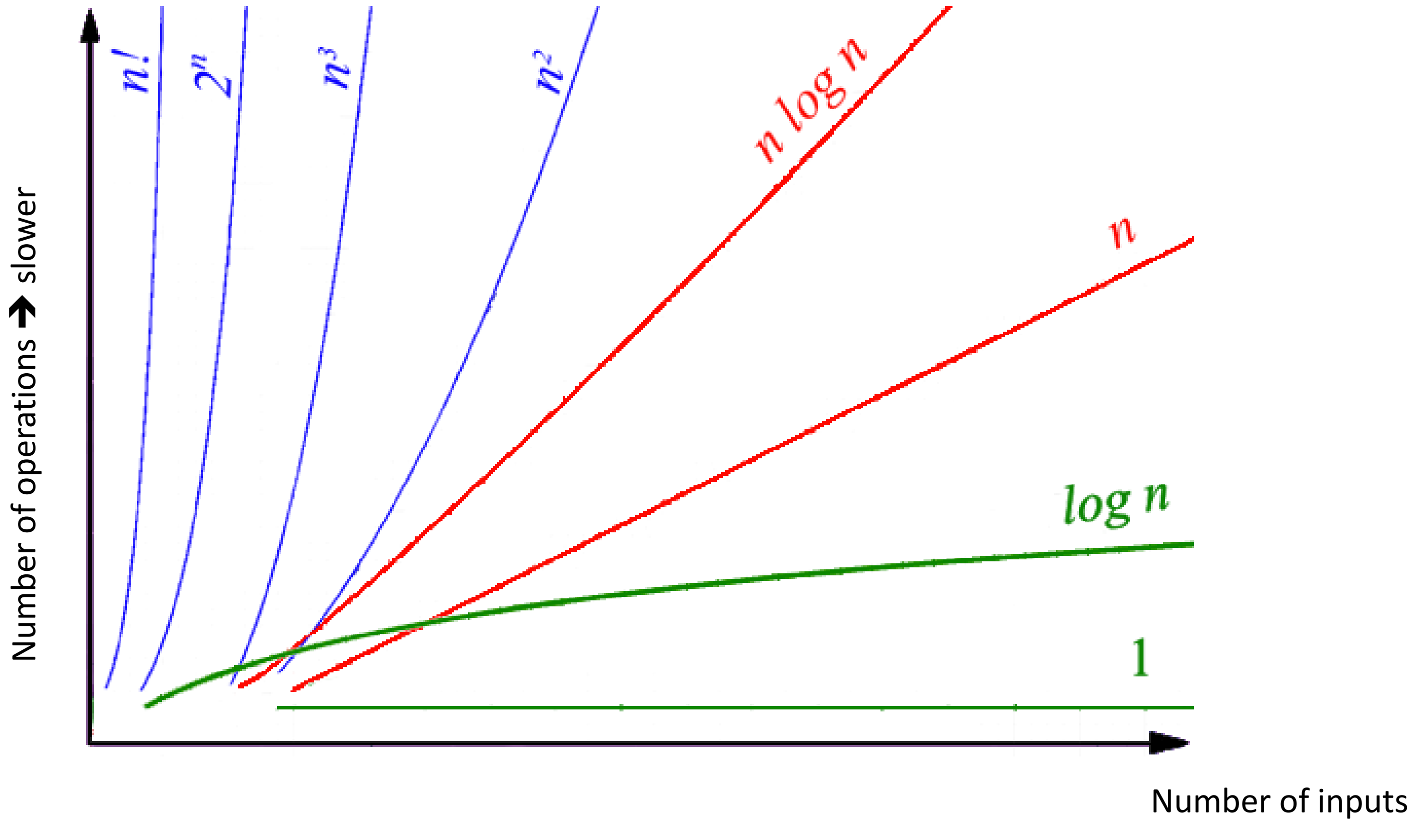
Binary search of a sorted list: $O(\log_2 n)$

Recap: factorial, exponential

- *Exponential functions*, $f(n) = c^n$ *for a given constant* $c > 1$ – Functions like 2^n arise when enumerating all subsets of n items. As we have seen, exponential algorithms become useless fast, but not as fast as...
- *Factorial functions*, $f(n) = n!$ – Functions like $n!$ arise when generating all permutations or orderings of n items.



$$n! \gg 2^n \gg n^3 \gg n^2$$



$$n! \gg 2^n \gg n^3 \gg n^2$$

Time Versus **Space** Complexities

Two main characteristics for programs

- **Time** complexity: ≈ **CPU** usage
- **Space** complexity: ≈ **RAM** usage

NB: if **time** complexity is "**high**" your algorithm <u>may run for too long</u>; if **space** complexity is **high**, your stack may be over flown, and you <u>may not be able to run the algorithm at all</u>!

Space and Time complexity

- The **space complexity** of an algorithm is the amount of memory it needs to run to completion.

- The **time complexity** of an algorithm is the amount of computer time it needs to run to completion.

Data Structure Used May Affect Complexity

Gale Shapley Algorithm (2.3) needs to maintain a dynamically changing set (list of free men).

Need fast ADD, DELETE, SELECT OPERATIONS

Use Priority Queue Data Structure.

Two implementations of priority queues:

1) Using Arrays and Lists: Slower: $O(n^2)$

2) Using Heap: Faster: O(nlogn)

Heap

A binary tree with n nodes and of height *h* is **almost complete** iff its nodes correspond to the nodes which are numbered 1 to *n* in the complete binary tree of height *h*.

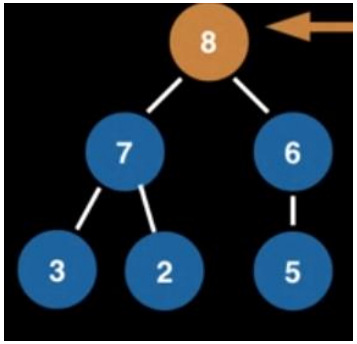
A **heap** is an *almost complete binary tree* that satisfies the **heap property**:

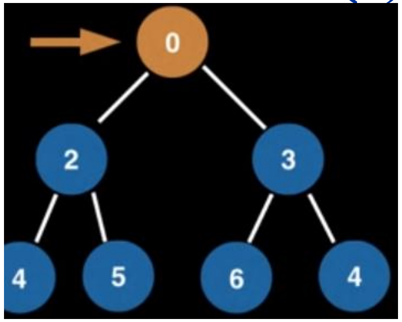
max-heap: For every node *i* other than the root:

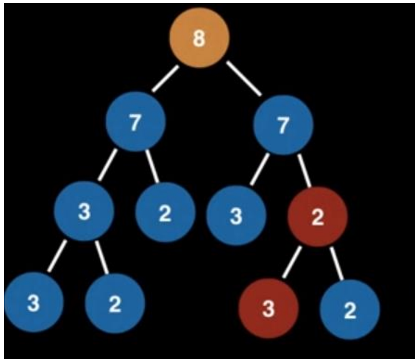
$$A[\mathrm{Parent}(i)] \geq A[i]$$

min-heap: For every node i other than the root:

$$A[\mathrm{Parent}(i)] \leq A[i]$$







Max-Heap

A **max-heap** is an *almost complete binary tree* that satisfies the **heap property**:

For every node i other than the root,

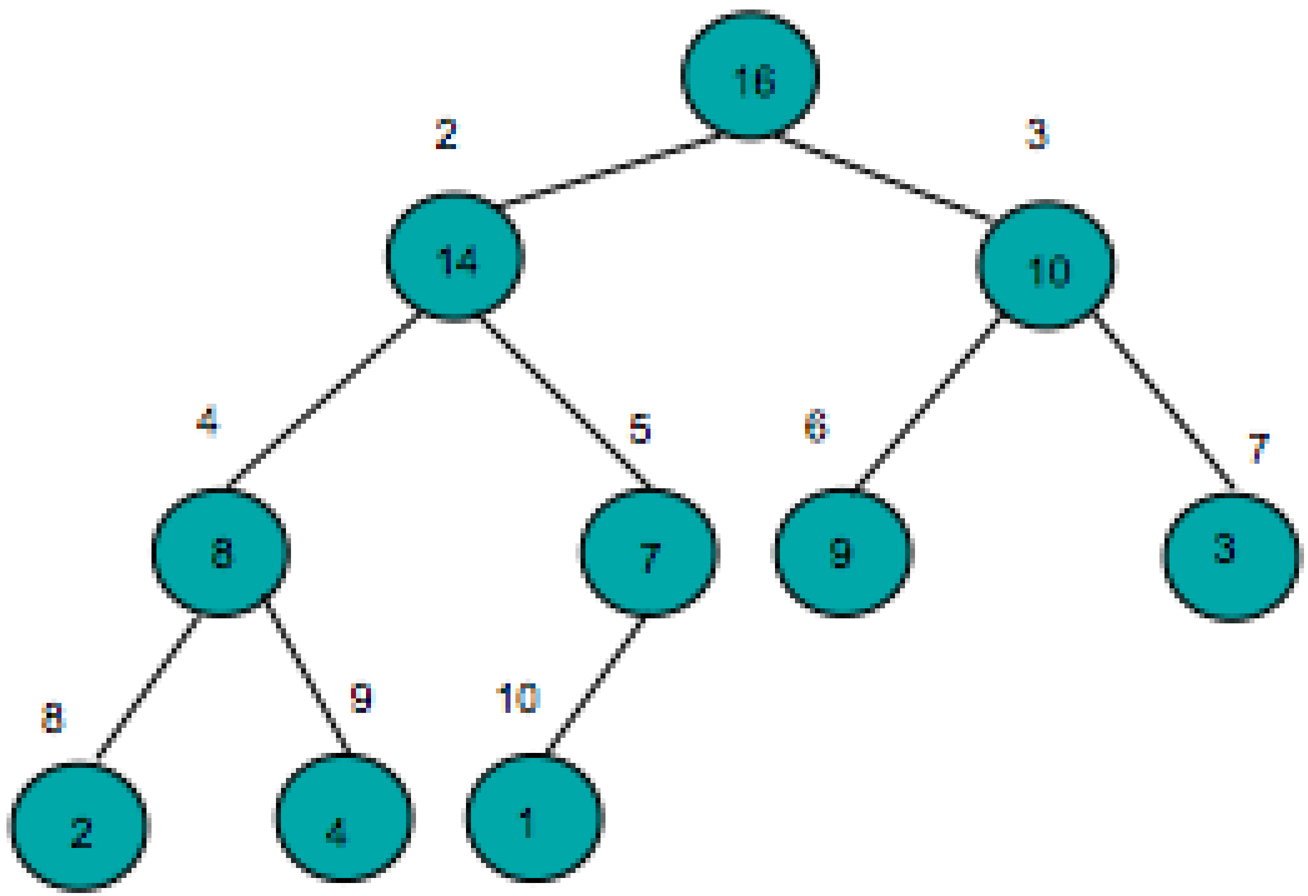
$$A[\text{PARENT}(i)] \geq A[i]$$

What does this mean?

• the value of a node is at most the value of its parent

• the largest element in the heap is stored in the root

• subtrees rooted at a node contain smaller values than the node itself



16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

Propose-And-Reject Algorithm (Gale Shapley)

Propose-and-reject algorithm. [Gale-Shapley 1962] Intuitive method that guarantees to find a stable matching.

```
Initialize each person to be free.
while (some man is free and hasn't proposed to every woman) {
    Choose such a man m
    w = 1st woman on m's list to whom m has not yet proposed
    if (w is free)
        assign m and w to be engaged
    else if (w prefers m to her fiancé m')
        assign m and w to be engaged, and m' to be free
    else
        w rejects m
}
```

Implementing the Gale Shapley Algorithm

Using Priority Queues (Heap):

There is a need to keep a dynamically changing set (e.g. The set of free men)

We need to add, delete, select elements from the list fast.

Priority queue: Elements have a priority or key, each time you select, you select the item with the highest priority.

A set of elements S

Each element v in S has an associated key value key(v)

Add, delete, search operations in O(logn) time.

A sequence of O(n) priority queue ops can be used to sort n numbers.

An implementation for a priority queue: Heap

Heap order: key(w)<=key(v) where v at node i and w at i's parent

Heap operations:

StartHeap(N), Insert(H,v), FindMin(H), Delete(H,i), ExtractMin(H), ChangeKey(H,v,α)

Take-home lesson

The heart of any algorithm is an idea.

Efficiency and correctness are to be taken into account.

There are some correct algorithms that are too impractical to compute.

There are **laws** saying what we can and what we can't compute!

As a programmer, you have to be aware of such "big" laws. (Cf. engineers have to know there are laws of physics)

In second part of the course, many problems will be fundamentally **hard**…

Good news: understanding of basic laws such as multiplication principle will take you a long way.

Next Lecture

- Basics of Graphs
- Breadth First Search
- Depth First Search
- Testing Bi-partite
- Topological Ordering

Week	Date	Topics
1	22 Feb	Introduction. Some representative problems
2	1 March	Stable Matching
3	8 March	Basics of algorithm analysis.
4	15 March	Graphs (Project 1 announced)
5	22 March	Greedy algorithms I
6	29 March	Greedy algorithms II (Project 2 announced)
7	5 April	Divide and conquer
8	12 April	Midterm
9	19 April	Dynamic Programming I
10	26 April	Dynamic Programming II (Project 3 announced)
11	3 May	BREAK
12	10 May	Network Flow-I
13	17 May	Network Flow II
14	24 May	NP and computational intractability I
15	31 May	NP and computational intractability II