Algorithm II

2. Algorithm Analysis

WU Xiaokun 吴晓堃

xkun.wu [at] gmail

Why analyzing algorithms?

Precise **assessment** leads to better **understanding**.

- *correctness*
  - theoretical proof
  - practical implementation
- *efficiency*: iterative development
  - computable?
  - what design to choose?
  - any room for improvement? or terminate?

Why analyzing algorithms?

Precise **assessment** leads to better **understanding**.

- *correctness*
  - theoretical proof
  - practical implementation
- *efficiency*: iterative development
  - computable?
  - what design to choose?
  - any room for improvement? or terminate?

We focus on the *efficiency* of algorithms now.

Content

- Computational Tractability
- Asymptotic Order of Growth
- Implement Gale–Shapley
- Common Running Times
- Recap: Priority Queue

Computational Tractability

What is “Computational Tractability”

Loosely speaking: delimitate whether a problem can be solved *in practice*.

- usually, relative to current computing power.
  - imagine a cart driven by a motor
- also, contextual tolerance is often a key consideration.
  - e.g., patience of your customer

What is “Computational Tractability”

Loosely speaking: delimitate whether a problem can be solved *in practice*.

- usually, relative to current computing power.
  - imagine a cart driven by a motor
- also, contextual tolerance is often a key consideration.
  - e.g., patience of your customer

Intractable problem maybe solvable in theory, but in practice any solution takes too many resources to be useful.

What is “Computational Tractability”

Loosely speaking: delimitate whether a problem can be solved *in practice*.

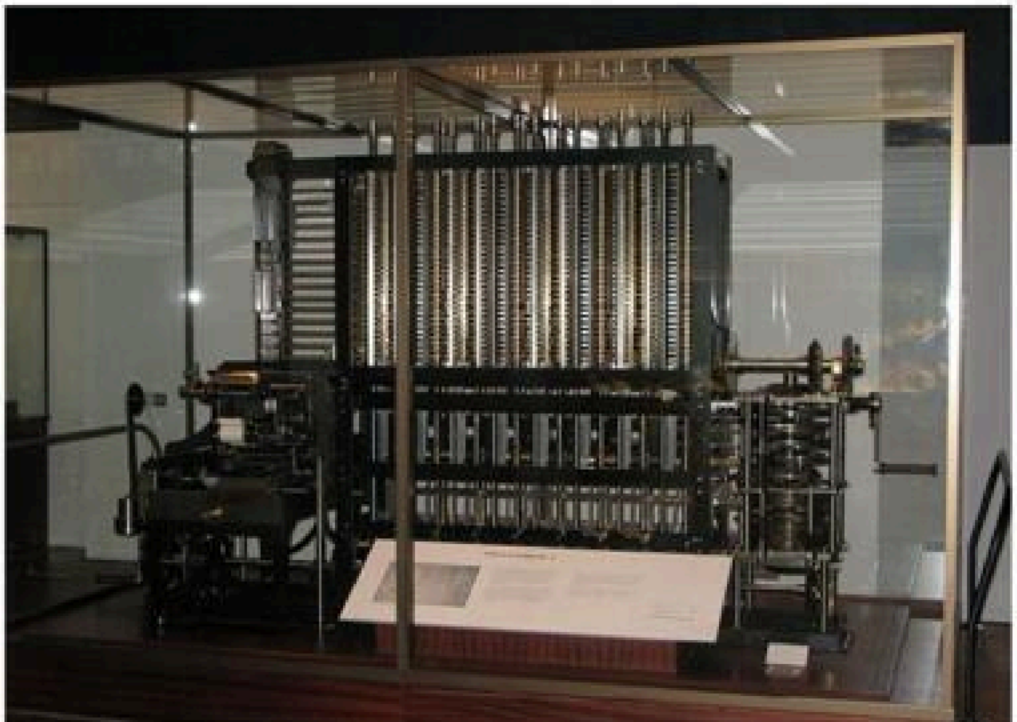
- usually, relative to current computing power.
  - imagine a cart driven by a motor
- also, contextual tolerance is often a key consideration.
  - e.g., patience of your customer

Intractable problem maybe solvable in theory, but in practice any solution takes too many resources to be useful.

So **efficiency** is about: resource requirements vs. computational power.

Analytic Engine

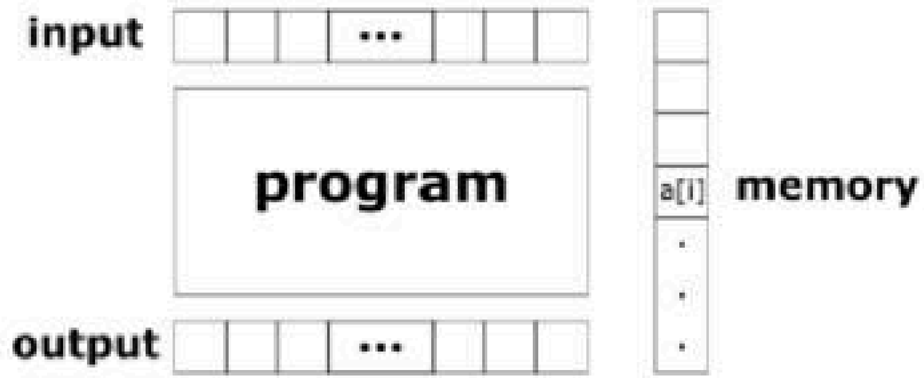
"By what course of calculation can these results be arrived at by the machine in the shortest time?" — Charles Babbage (1864)



Modern computing model

Consider a 64-bit system:

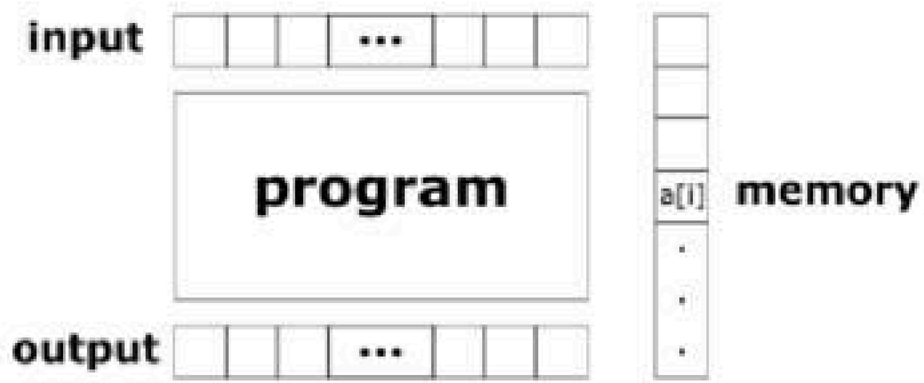
- Each **memory cell** stores a 64-bit integer.
- **Primitive operations**: arithmetic/logic operations, read/write memory, array indexing, following a pointer, conditional branch, etc.



Modern computing model

Consider a 64-bit system:

- Each **memory cell** stores a 64-bit integer.
- **Primitive operations**: arithmetic/logic operations, read/write memory, array indexing, following a pointer, conditional branch, etc.



- **Time**: Number of primitive operations, given CPU speed.
- **Space**: Number of memory cells utilized.

How to define efficiency?

Intuition. When implemented, *runs fast and uses few memory* on real inputs.

- what platform? PC, cellphone
- what is a "real" inputs? `struct`, `int`

We need a measure of algorithm *itself*, rather than external indicators.

How to define efficiency?

Intuition. When implemented, *runs fast and uses few memory* on real inputs.

- what platform? PC, cellphone
- what is a "real" inputs? `struct`, `int`

We need a measure of algorithm *itself*, rather than external indicators.

Can we measure efficiency when input number is *fixed* (same PC)?

- equal: count number of operations/cells required *per unit input*.
  - counter-example: print N number pairs vs. N numbers.

How to define efficiency?

Intuition. When implemented, *runs fast and uses few memory* on real inputs.

- what platform? PC, cellphone
- what is a “real” inputs? `struct`, `int`

We need a measure of algorithm *itself*, rather than external indicators.

Can we measure efficiency when input number is *fixed* (same PC)?

- equal: count number of operations/cells required *per unit input*.
  - counter-example: print N number pairs vs. N numbers.

Better measure: How is the algorithm *scale* with problem size.

Scalability

How resource requirements *grow with increasing input size.*

- The input has a natural "size" parameter N.
- Analyze running time mathematically as a function $T(N)$.

Scalability

How resource requirements *grow with increasing input size*.

- The input has a natural "size" parameter N.
- Analyze running time mathematically as a function $T(N)$.

So we study and compare *growth of functions*.

- **sampling**: measure efficiency at a series of fixed input numbers.

Scalability

How resource requirements *grow with increasing input size*.

- The input has a natural “size” parameter N.
- Analyze running time mathematically as a function $T(N)$.

So we study and compare *growth of functions*.

- **sampling**: measure efficiency at a series of fixed input numbers.
- compare: “standard” behavior among all possible inputs
  - sorting does nothing (thus fast), when input already sorted

Worst-Case Analysis

Worst-Case Running Times: longest possible running time.

- well-accepted standard, but *not perfect*
  - pathological inputs can lead to bad performance
  - hard to find effective alternative

Worst-Case Analysis

Worst-Case Running Times: longest possible running time.

- well-accepted standard, but *not perfect*
  - pathological inputs can lead to bad performance
  - hard to find effective alternative

Average-case analysis: averaged over "random" instances.

- more about how random inputs were generated (than algorithm itself)
  - real random generator is actually hard to implement

Worst-Case Analysis

Worst-Case Running Times: longest possible running time.

- well-accepted standard, but *not perfect*
  - pathological inputs can lead to bad performance
  - hard to find effective alternative

Average-case analysis: averaged over "random" instances.

- more about how random inputs were generated (than algorithm itself)
  - real random generator is actually hard to implement

Now consider and compare $T(N)$ on worst-cases

- need a *baseline* implementation to mark the worst possibility.

Brute-Force Search

Brute-Force Search: the most natural last-resort solution.

- enumerate all possibilities
  - no use in practice, but usually gives *exact analytical bounds*.
  - Stable matching: test all $n!$ perfect matchings for stability.

Brute-Force Search

Brute-Force Search: the most natural last-resort solution.

- enumerate all possibilities
  - no use in practice, but usually gives *exact analytical bounds*.
  - Stable matching: test all $n!$ perfect matchings for stability.

Define efficient: achieves *qualitatively better* worst-case performance, at an analytical level, than brute-force search.

- analytically shows *algorithmic heuristics* and *problem structure*
  - helps understanding, thus improve design

Brute-Force Search

Brute-Force Search: the most natural last-resort solution.

- enumerate all possibilities
  - no use in practice, but usually gives *exact analytical bounds*.
  - Stable matching: test all $n!$ perfect matchings for stability.

Define efficient: achieves *qualitatively better* worst-case performance, at an analytical level, than brute-force search.

- analytically shows *algorithmic heuristics* and *problem structure*
  - helps understanding, thus improve design

What is "qualitatively better"? Better scalability

- brute-force search usually grow *exponentially fast*
- intuitively, growth rate should be much slower

Polynomial running time

Desirable scaling property. When input size *doubles*, algorithm slow down by at most some multiplicative constant factor C.

Polynomial running time

Desirable scaling property. When input size *doubles*, algorithm slow down by at most some multiplicative constant factor C.

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

There exist constants $c > 0$ and $d > 0$ such that, for every input of size N, the running time of the algorithm is *bounded above* by cN^d primitive computational steps.

- here $C = 2^d$
- lower-degree polynomials grow slower

Polynomial efficiency

Def. An algorithm is **efficient** if it has a polynomial running time.

Polynomial efficiency

Def. An algorithm is **efficient** if it has a polynomial running time.

- exactly characterize algorithm *itself*
  - platform-, instance-independent

Polynomial efficiency

Def. An algorithm is **efficient** if it has a polynomial running time.

- exactly characterize algorithm *itself*
  - platform-, instance-independent
- works in practice
  - break-through **exponential barrier** exposes *crucial structure*
    - when exist, always found moderately growing polynomials

Polynomial efficiency

Def. An algorithm is **efficient** if it has a polynomial running time.

- exactly characterize algorithm *itself*
  - platform-, instance-independent
- works in practice
  - break-through **exponential barrier** exposes *crucial structure*
    - when exist, always found moderately growing polynomials
- becomes negatable: define **inefficiency**

Polynomial efficiency

Def. An algorithm is **efficient** if it has a polynomial running time.

- exactly characterize algorithm *itself*
  - platform-, instance-independent
- works in practice
  - break-through **exponential barrier** exposes *crucial structure*
    - when exist, always found moderately growing polynomials
- becomes negatable: define **inefficiency**

Exceptions: galactic constants and/or huge exponents

- which is better: $20n^{120}$ or $n^{1+0.02\ln n}$?

≡

Common polynomials

Assume: one million (10^6) high-level instructions per second.

	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

Notice the the huge difference between polynomial and exponential.

Common polynomials

Assume: one million (10^6) high-level instructions per second.

	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

Notice the the huge difference between polynomial and exponential.

Now we found the way to compare growth of functions.

- compare different *categories* of growth rate

Asymptotic Order of Growth

Asymptotic analysis

Mathematically, asymptotic is used for describing *limiting* behavior.

- rigorous description of scalability: growth rate
- only a coarser level of granularity is necessary
  - Ex. $1.62n^2 + 3.5n + 8$ steps

Asymptotic analysis

Mathematically, asymptotic is used for describing *limiting* behavior.

- rigorous description of scalability: growth rate
- only a coarser level of granularity is necessary
  - Ex. $1.62n^2 + 3.5n + 8$ steps

Limits are natural *bounds* for analysis.

- upper bound, lower bound, exact bound.
- especially, *upper bound* for worst case

Asymptotic analysis

Mathematically, asymptotic is used for describing *limiting* behavior.

- rigorous description of scalability: growth rate
- only a coarser level of granularity is necessary
  - Ex. $1.62n^2 + 3.5n + 8$ steps

Limits are natural *bounds* for analysis.

- upper bound, lower bound, exact bound.
- especially, *upper bound* for worst case

Caution. In CS, deal with discrete quantities.

- no such thing as "infinitesimal" in calculus.

Asymptotic Upper Bounds (Big O)

$T(n)$ is $O(f(n))$ (read as "$T(n)$ is order $f(n)$")

- for sufficiently large n, function $T(n)$ is *bounded above* by a constant multiple of $f(n)$.
- $\exists c > 0, n_0 \geq 0 : \forall n \geq n_0, T(n) \leq cf(n)$.
  - c cannot depend on n.

Asymptotic Upper Bounds (Big O)

$T(n)$ is $O(f(n))$ (read as "$T(n)$ is order $f(n)$")

- for sufficiently large n, function $T(n)$ is *bounded above* by a constant multiple of $f(n)$.
- $\exists c > 0, n_0 \geq 0 : \forall n \geq n_0, T(n) \leq cf(n)$.
  - c cannot depend on n.

Ex. $T(n) = pn^2 + qn + r$:

- $T(n) = pn^2 + qn + r \leq pn^2 + qn^2 + rn^2 = (p + q + r)n^2$
  - $T(n) \leq cn^2 \in O(n^2)$, where $c = p + q + r$.

Big O notational abuses

One-way "equality". $O(g(n))$ is a set of functions.

- $f(n) \in O(g(n))$.
- but CSer often write $f(n) = O(g(n))$.

Big O notational abuses

One-way "equality". $O(g(n))$ is a *set* of functions.

- $f(n) \in O(g(n))$.
- but CSer often write $f(n) = O(g(n))$.

Note. $O(\cdot)$ expresses only *an* upper bound.

- $T(n) = pn^2 + q^n + r = O(n^3)$, since $n^2 \leq n^3$.
  - but we cannot say $T(n) = sn^3$.
- in practice, we prefer "tightest" possible bound.

Big O notational abuses

One-way "equality". $O(g(n))$ is a *set* of functions.

- $f(n) \in O(g(n))$.
- but CSer often write $f(n) = O(g(n))$.

Note. $O(\cdot)$ expresses only *an* upper bound.

- $T(n) = pn^2 + q^n + r = O(n^3)$, since $n^2 \leq n^3$.
  - but we cannot say $T(n) = sn^3$.
- in practice, we prefer "tightest" possible bound.

Domain and Range. T and f are real-valued functions.

- domain is typically natural numbers: $\mathbb{N} \to \mathbb{R}$.
- Sometimes extend to the reals: $\mathbb{R}_{\geq 0} \to \mathbb{R}$.
- Or restrict to a subset.

Big O: properties

Reflexivity. f is $O(f)$.

Big O: properties

Reflexivity. f is $O(f)$.

Constants. If f is $O(g)$ and $c > 0$, then cf is $O(g)$.

Big O: properties

Reflexivity. f is $O(f)$.

Constants. If f is $O(g)$ and $c > 0$, then cf is $O(g)$.

Products. If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1\ f_2$ is $O(g_1 g_2)$.
Pf.

- $\exists c_1 > 0 and n_1 \geq 0$ such that $0 \leq f_1(n) \leq c_1 g_1(n)$ for all $n \geq n_1$.
- $\exists c_2 > 0 and n_2 \geq 0$ such that $0 \leq f_2(n) \leq c_2 g_2(n)$ for all $n \geq n_2$.
- Then, $0 \leq f_1(n) f_2(n) \leq c_1 c_2 g_1(n) g_2(n)$ for all $n \geq \max\{n_1, n_2\}$.

Big O: properties

Reflexivity. f is $O(f)$.

Constants. If f is $O(g)$ and $c > 0$, then cf is $O(g)$.

Products. If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 f_2$ is $O(g_1 g_2)$.
Pf.

- $\exists c_1 > 0 and n_1 \geq 0$ such that $0 \leq f_1(n) \leq c_1 g_1(n)$ for all $n \geq n_1$.
- $\exists c_2 > 0 and n_2 \geq 0$ such that $0 \leq f_2(n) \leq c_2 g_2(n)$ for all $n \geq n_2$.
- Then, $0 \leq f_1(n) f_2(n) \leq c_1 c_2 g_1(n) g_2(n)$ for all $n \geq \max\{n_1, n_2\}$.

Sums. If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 + f_2$ is $O(\max\{g_1, g_2\})$.

Big O: properties

Reflexivity. f is $O(f)$.

Constants. If f is $O(g)$ and $c > 0$, then cf is $O(g)$.

Products. If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 f_2$ is $O(g_1 g_2)$.
Pf.

- $\exists c_1 > 0 and n_1 \geq 0$ such that $0 \leq f_1(n) \leq c_1 g_1(n)$ for all $n \geq n_1$.
- $\exists c_2 > 0 and n_2 \geq 0$ such that $0 \leq f_2(n) \leq c_2 g_2(n)$ for all $n \geq n_2$.
- Then, $0 \leq f_1(n) f_2(n) \leq c_1 c_2 g_1(n) g_2(n)$ for all $n \geq \max\{n_1, n_2\}$.

Sums. If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 + f_2$ is $O(\max\{g_1, g_2\})$.

Transitivity. If f is $O(g)$ and g is $O(h)$, then f is $O(h)$.

Big O: properties

Reflexivity. f is $O(f)$.

Constants. If f is $O(g)$ and $c > 0$, then cf is $O(g)$.

Products. If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1\ f_2$ is $O(g_1 g_2)$.
Pf.

- $\exists c_1 > 0 and n_1 \geq 0$ such that $0 \leq f_1(n) \leq c_1 g_1(n)$ for all $n \geq n_1$.
- $\exists c_2 > 0 and n_2 \geq 0$ such that $0 \leq f_2(n) \leq c_2 g_2(n)$ for all $n \geq n_2$.
- Then, $0 \leq f_1(n) f_2(n) \leq c_1 c_2 g_1(n) g_2(n)$ for all $n \geq \max\{n_1, n_2\}$.

Sums. If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 + f_2$ is $O(\max\{g_1, g_2\})$.

Transitivity. If f is $O(g)$ and g is $O(h)$, then f is $O(h)$.

Ex. $f(n) = 5n^3 + 3n^2 + n + 1234$ is $O(n^3)$.

Asymptotic Lower Bounds (Big Ω)

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

- for sufficiently large n, function $T(n)$ is *at least* a constant multiple of $f(n)$.
- $\exists\epsilon > 0, n_0 \geq 0 : \forall n \geq n_0, T(n) \geq \epsilon f(n)$.
  - ϵ cannot depend on n.

Asymptotic Lower Bounds (Big Ω)

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

- for sufficiently large n, function $T(n)$ is *at least* a constant multiple of $f(n)$.
- $\exists \epsilon > 0, n_0 \geq 0 : \forall n \geq n_0, T(n) \geq \epsilon f(n)$.
  - ϵ cannot depend on n.

Ex. $T(n) = 32n^2 + 17n + 1$

- $T(n)$ is both $\Omega(n^2)$ and $\Omega(n)$.
- $T(n)$ is not $\Omega(n^3)$.

Asymptotically Tight Bounds (Big Θ)

$T(n)$ is $\Theta(f(n))$ ("$T(n) = \Theta(f(n))$")

- $T(n)$ is both $O(f(n))$ and also $\Omega(f(n))$.
- $\exists c_1 > 0, c_2 > 0, n_0 \geq 0 : \forall n \geq n_0, 0 \leq c_1 f(n) \leq T(n) \leq c_2 f(n)$.
  - c_1, c_2 cannot depend on n.

Asymptotically Tight Bounds (Big Θ)

$T(n)$ is $\Theta(f(n))$ ("$T(n) = \Theta(f(n))$")

- $T(n)$ is both $O(f(n))$ and also $\Omega(f(n))$.
- $\exists c_1 > 0, c_2 > 0, n_0 \geq 0 : \forall n \geq n_0, 0 \leq c_1 f(n) \leq T(n) \leq c_2 f(n)$.
  - c_1, c_2 cannot depend on n.

Ex. $T(n) = 32n^2 + 17n + 1$

- $T(n)$ is $\Theta(n^2)$.
- $T(n)$ is neither $\Theta(n^3)$ nor $\Theta(n)$.

Asymptotically Tight Bounds (Big Θ)

$T(n)$ is $\Theta(f(n))$ ("$T(n) = \Theta(f(n))$")

- $T(n)$ is both $O(f(n))$ and also $\Omega(f(n))$.
- $\exists c_1 > 0, c_2 > 0, n_0 \geq 0 : \forall n \geq n_0, 0 \leq c_1 f(n) \leq T(n) \leq c_2 f(n)$.
  - c_1, c_2 cannot depend on n.

Ex. $T(n) = 32n^2 + 17n + 1$

- $T(n)$ is $\Theta(n^2)$.
- $T(n)$ is neither $\Theta(n^3)$ nor $\Theta(n)$.

Compute: closing *gap* between upper bound and lower bound

- design: *a* worst-case algorithm as upper bound
  - prove: no better possibilities
- justify asymptotically *tight* bound on worst-case running time

Asymptotic bounds and limits

Proposition. If for some constant $0 < c < \infty$ $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$ then $f(n)$ is $\Theta(g(n))$

.

Pf.

- By definition of the limit, $\forall \epsilon > 0, \exists n_0 : c - \epsilon \leq \frac{f(n)}{g(n)} \leq c + \epsilon, \forall n \geq n_0$.
- Choose $\epsilon = \frac{1}{2}c > 0$.
  - $\frac{1}{2}cg(n) \leq f(n) \leq \frac{3}{2}cg(n), \forall n \geq n_0$.

Asymptotic bounds and limits

Proposition. If for some constant $0 < c < \infty$ $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ then $f(n)$ is $\Theta(g(n))$

.

Pf.

- By definition of the limit, $\forall \epsilon > 0, \exists n_0 : c - \epsilon \leq \frac{f(n)}{g(n)} \leq c + \epsilon, \forall n \geq n_0$.
- Choose $\epsilon = \frac{1}{2}c > 0$.
  - $\frac{1}{2}cg(n) \leq f(n) \leq \frac{3}{2}cg(n), \forall n \geq n_0$.

Proposition. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f(n)$ is $O(g(n))$ but not $\Omega(g(n))$.

Proposition. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, then $f(n)$ is $\Omega(g(n))$ but not $O(g(n))$.

Asymptotic bounds for common functions

Polynomials. Let $f(n) = a_0 + a_1 n + \ldots + a_d n^d$ with $a_d > 0$. Then $f(n) = \Theta(n^d)$.

Pf. $\lim_{n \to \infty} \frac{a_0 + a_1 n + \ldots + a_d n^d}{n^d} = a_d > 0$

Asymptotic bounds for common functions

Polynomials. Let $f(n) = a_0 + a_1 n + \ldots + a_d n^d$ with $a_d > 0$. Then $f(n) = \Theta(n^d)$.

Pf. $\lim_{n \to \infty} \frac{a_0 + a_1 n + \ldots + a_d n^d}{n^d} = a_d > 0$

Logarithms. $\log_a n = \Theta(\log_b n)$ for every $a > 1$ and every $b > 1$.

Pf. $\frac{\log_a n}{\log_b n} = \frac{1}{\log_b a}$.

Asymptotic bounds for common functions

Polynomials. Let $f(n) = a_0 + a_1 n + \ldots + a_d n^d$ with $a_d > 0$. Then $f(n) = \Theta(n^d)$.

Pf. $\lim_{n\to\infty} \frac{a_0 + a_1 n + \ldots + a_d n^d}{n^d} = a_d > 0$

Logarithms. $\log_a n = \Theta(\log_b n)$ for every $a > 1$ and every $b > 1$.

Pf. $\frac{\log_a n}{\log_b n} = \frac{1}{\log_b a}$.

Logarithms and polynomials. $\log_a n = O(n^d)$ for every $a > 1$ and every $d > 0$.

Pf. $\lim_{n\to\infty} \frac{\log_a n}{n^d} = 0$.

Asymptotic bounds for common functions

Polynomials. Let $f(n) = a_0 + a_1 n + \ldots + a_d n^d$ with $a_d > 0$. Then $f(n) = \Theta(n^d)$.

Pf. $\lim_{n \to \infty} \frac{a_0 + a_1 n + \ldots + a_d n^d}{n^d} = a_d > 0$

Logarithms. $\log_a n = \Theta(\log_b n)$ for every $a > 1$ and every $b > 1$.

Pf. $\frac{\log_a n}{\log_b n} = \frac{1}{\log_b a}$.

Logarithms and polynomials. $\log_a n = O(n^d)$ for every $a > 1$ and every $d > 0$.

Pf. $\lim_{n \to \infty} \frac{\log_a n}{n^d} = 0$.

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

Pf. $\lim_{n \to \infty} \frac{n^d}{r^n} = 0$.

Asymptotic bounds for common functions

Polynomials. Let $f(n) = a_0 + a_1 n + \ldots + a_d n^d$ with $a_d > 0$. Then $f(n) = \Theta(n^d)$.

Pf. $\lim_{n \to \infty} \frac{a_0 + a_1 n + \ldots + a_d n^d}{n^d} = a_d > 0$

Logarithms. $\log_a n = \Theta(\log_b n)$ for every $a > 1$ and every $b > 1$.

Pf. $\frac{\log_a n}{\log_b n} = \frac{1}{\log_b a}$.

Logarithms and polynomials. $\log_a n = O(n^d)$ for every $a > 1$ and every $d > 0$.

Pf. $\lim_{n \to \infty} \frac{\log_a n}{n^d} = 0$.

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

Pf. $\lim_{n \to \infty} \frac{n^d}{r^n} = 0$.

Factorials. $n! = 2^{\Theta(n \log n)}$.

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

Big O notation with multiple variables

Upper bounds. $f(m, n) = O(g(m, n))$ if there exist constants $c > 0, m_0 \geq 0$, and $n_0 \geq 0$ such that $0 \leq f(m, n) \leq cg(m, n)$ for all $n \geq n_0, m \geq m_0$.

Big O notation with multiple variables

Upper bounds. $f(m,n) = O(g(m,n))$ if there exist constants $c > 0, m_0 \geq 0$, and $n_0 \geq 0$ such that $0 \leq f(m,n) \leq cg(m,n)$ for all $n \geq n_0, m \geq m_0$.

Ex. $f(m,n) = 32mn^2 + 17mn + 32n^3$.

- $f(m,n)$ is both $O(mn^2 + n^3)$ and $O(mn^3)$.
- $f(m,n)$ is neither $O(n^3)$ nor $O(mn^2)$.
  - $f(m,n)$ is $O(n^3)$ if a precondition to the problem implies $m \leq n$.

Implement Gale–Shapley

Goal: $O(n^2)$ implementation

Compute: closing *gap* between upper bound and lower bound

- design: *a* worst-case algorithm as upper bound
  - prove: no better possibilities

Goal: $O(n^2)$ implementation

Compute: closing *gap* between upper bound and lower bound

- design: *a* worst-case algorithm as upper bound
  - prove: no better possibilities

Recall: Algorithm terminates in at most n^2 iterations

- worst-case bound: $O(n^2)$
  - **Goal**: find a $O(n^2)$ implementation
    - each iteration takes constant time, ie., $O(1)$
- tightest? discuss later

Recap: Gale–Shapley

INPUT: M, W, R_m, R_w

1. $P = \varnothing$; mark $m \in M$ and $w \in W$ `free`;
2. WHILE some $m \in M$ is `free`
   1. w: highest on R_m that m has not yet proposed;
   2. IF w is `free`
      1. Add (m, w) to P;
   3. ELSE IF w prefers m to current partner m'
      1. Replace (m', w) with (m, w), set m' `free`;
   4. ELSE (Nothing happens.);
3. RETURN P;

Constant time operations

Goal: the following operations take constant time:

1. identify a `free` m.
2. given m, identify highest-ranked w that m not yet proposed.
3. given w, decide if is `matched`,
   - if so, identify current partner m'.
4. identify which ranks higher for w: m or m'.

Representation 1: next `free`

List NF containing indices of M (queue or stack also works)

- initialize to n indices
  - initialization: $O(n)$ (is also $O(n^2)$)
- take next free element: $O(1)$
- if replaced, push back: $O(1)$

Representation 2: proposal

Reuse the preference list R_m, only check the head.

- head always has highest-ranked w: $O(1)$
- after taking out w, remove current head: $O(1)$

Representation 2: proposal

Reuse the preference list R_m, only check the head.

- head always has highest-ranked w: $O(1)$
- after taking out w, remove current head: $O(1)$

If R_m is `constant`, maintain a pointer to next proposal.

- moving pointer to the next: $O(1)$

Representation 3: matching

Index $M, W: \{1, 2, \ldots, n\}$.

Matching. Arrays P_m, P_w.

- if m matched to w: $P_m(m) = w, P_w(w) = m$.
  - add/remove matching pair: $O(1)$
- initialize P_m, P_w to 0: `unmatched`.
  - identify whether `matched`, and to whom: $O(1)$
  - initialization: $O(n)$ (= $O(n^2)$)

Representation 4: compare ranks

So far, operation 1-3 can be implemented in $O(1)$ time.

- now: identify which ranks higher for w: m or m'.

Representation 4: compare ranks

So far, operation 1-3 can be implemented in $O(1)$ time.

- now: identify which ranks higher for w: m or m'.

Naive implementation: walk R_w

- $O(n)$ time to find m and m' on the list
  - breaks our $O(1)$ time goal

Alternative: trade space for time.

Representation 4: compare ranks (cont.)

For each $w \in W$, maintain an array I_w contains the inverse of R_w.

pref[]	1st	2nd	3rd	4th	5th	6th	7th	8th
	8	3	7	1	4	5	6	2

rank[]	1	2	3	4	5	6	7	8
	4th	8th	2nd	5th	6th	7th	3rd	1st

- for i = 1 to n: $I_w[R_w[i]] = i$
  - only need to compare $I_w[m]$ and $I_w[m']$: $O(1)$
- $\Theta(n^2)$ time initialization: iterate for each w.

Gale–Shapley implementation: summary

Theorem. Can implement Gale–Shapley to run in $O(n^2)$ time.

Pf.

- $\Theta(n^2)$ preprocessing time to create n inverse ranking arrays.
- There are $O(n^2)$ proposals; processing each proposal takes $O(1)$ time.

Gale–Shapley implementation: summary

Theorem. Can implement Gale–Shapley to run in $O(n^2)$ time.

Pf.

- $\Theta(n^2)$ preprocessing time to create n inverse ranking arrays.
- There are $O(n^2)$ proposals; processing each proposal takes $O(1)$ time.

Theorem. In the worst case, any algorithm to find a stable matching must query R_m for $\Omega(n^2)$ times.
[proof skipped.]

Gale–Shapley implementation: summary

Theorem. Can implement Gale–Shapley to run in $O(n^2)$ time.
Pf.

- $\Theta(n^2)$ preprocessing time to create n inverse ranking arrays.
- There are $O(n^2)$ proposals; processing each proposal takes $O(1)$ time.

Theorem. In the worst case, any algorithm to find a stable matching must query R_m for $\Omega(n^2)$ times.
[proof skipped.]

Conclusion. $GS = \Theta(n^2)$

Common Running Times

Constant time

Constant time. Running time is $O(1)$.

- bounded by constant: not depend on input size n

Constant time

Constant time. Running time is $O(1)$.

- bounded by constant: not depend on input size n

Examples.

- Conditional branch.
- Arithmetic/logic operation.
- Declare/initialize a variable.
- Follow a link in a linked list.
- Access element i in an array.
- Compare/exchange two elements in an array.

Linear time

Linear time. Running time is $O(n)$.

- process input in a single pass,
  - spending constant time on each encountered item

Linear time

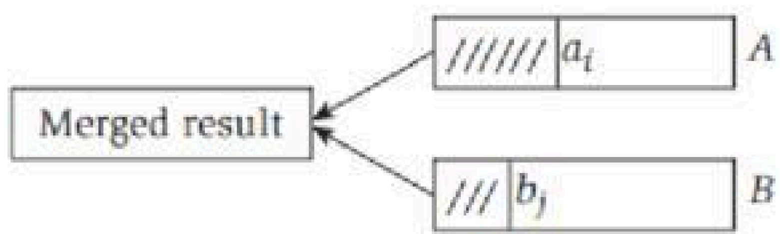
Linear time. Running time is $O(n)$.

- process input in a single pass,
  - spending constant time on each encountered item

Computing the Maximum.

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

- at most $2n$ iterations



Quiz: Target-Sum

Target-Sum. Given a *sorted* array of n distinct integers and an integer T, find two that sum to exactly T?

Hint: move two indices from opposite side towards each other.

Logarithmic time (Sublinear)

Logarithmic time. Running time is $O(\log n)$.

- splits input into two equal-sized pieces,
  - solves each piece recursively,
  - then combines two solutions in *constant time*.
- divide-and-conquer

Logarithmic time (Sublinear)

Logarithmic time. Running time is $O(\log n)$.

- splits input into two equal-sized pieces,
  - solves each piece recursively,
  - then combines two solutions in *constant time*.
- divide-and-conquer

Search in a sorted array. Given a sorted array A of n distinct integers and an integer x, find index of x in array.
Binary search.

- Invariant: If x is in the array, then x is in $A[lo..hi]$.
- After k iterations of `WHILE` loop, $(hi - lo + 1) \leq n/2^k \Rightarrow k \leq 1 + \log_2 n$.

Demo: Binary search

Linearithmic time

Linearithmic time. Running time is $O(n \log n)$.

- splits input into two equal-sized pieces,
  - solves each piece recursively,
  - then combines two solutions in *linear time*.
- divide-and-conquer

Linearithmic time

Linearithmic time. Running time is $O(n \log n)$.

- splits input into two equal-sized pieces,
  - solves each piece recursively,
  - then combines two solutions in *linear time*.
- divide-and-conquer

Sorting. Given an array of n elements, rearrange them in ascending order.

Merge sort

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Quadratic time

Quadratic time. Running time is $O(n^2)$.

- nested loops: search over all pairs of input items
  - spend constant time per pair.
- just the brute-force approach

Quadratic time

Quadratic time. Running time is $O(n^2)$.

- nested loops: search over all pairs of input items
  - spend constant time per pair.
- just the brute-force approach

Closest pair of points. Given a list of n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, find the pair that is closest to each other.
$O(n^2)$ **algorithm**. Enumerate all pairs of points (with $i < j$).

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

Cubic time

Cubic time. Running time is $O(n^3)$.

- nested loops: search over all subsets of size 3.
- almost the borderline of practical

Cubic time

Cubic time. Running time is $O(n^3)$.

- nested loops: search over all subsets of size 3.
- almost the borderline of practical

3-SUM. Given an array of n distinct integers, find three that sum to 0.
$O(n^3)$ **algorithm**. Enumerate all triples (with $i < j < k$).

Remark. $\Omega(n^3)$ seems inevitable, but $O(n^2)$ is not hard.

Polynomial time

Polynomial time. Running time is $O(n^k)$ for some constant $k > 0$.

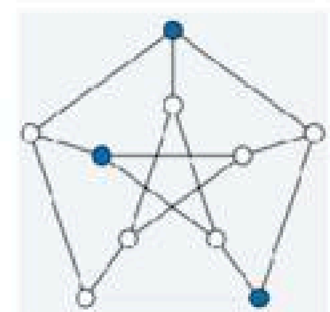
- nested loops: search over all subsets of size k.
- computationally too hard to be practical

Polynomial time

Polynomial time. Running time is $O(n^k)$ for some constant $k > 0$.

- nested loops: search over all subsets of size k.
- computationally too hard to be practical

Independent set of size k. Given a graph, find k nodes such that no two are joined by an edge.
$O(n^k)$ **algorithm**. Enumerate all subsets of k nodes.



- Check whether S is an independent set of size k takes $O(k^2)$ time.
- Number of k-element subsets = $\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots 1} \leq \frac{n^k}{k!}$
- in total: $O(k^2 n^k / k!) = O(n^k)$

Exponential time

Exponential time. Running time is $O(2^{n^k})$ for some constant $k > 0$.

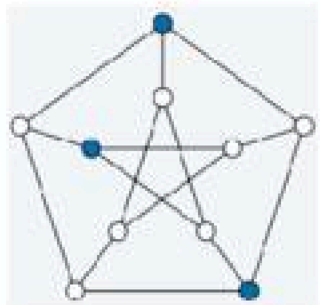
- combinatorial: enumerate all subsets

Exponential time

Exponential time. Running time is $O(2^{n^k})$ for some constant $k > 0$.

- combinatorial: enumerate all subsets

Independent set. Given a graph, find independent set of max cardinality.
$O(n^2 2^n)$ **algorithm**. Enumerate all subsets of n elements.



- total number of subsets: 2^n

Quiz: Exponential time

Which is an equivalent definition of exponential time?

- $O(2^n)$.
- $O(2^{cn})$ for some constant $c > 0$.
- Both.
- Neither.

Quiz: Exponential time

Which is an equivalent definition of exponential time?

- $O(2^n)$.
- $O(2^{cn})$ for some constant $c > 0$.
- Both.
- Neither.

Neither: take the limit of division.

Recap: Priority Queue

Priority

Primary goal. seek algorithms that improve qualitatively on brute-force search.

- use polynomial-time solvability as concrete formulation
- more complex data structures lead to better performance

Priority

Primary goal. seek algorithms that improve qualitatively on brute-force search.

- use polynomial-time solvability as concrete formulation
- more complex data structures lead to better performance

Priority Queue. Each element has a priority value.

- properties
  - always take out the highest-priority element
  - $O(\log n)$ time per operation.
- should be familiar after taking *Data Structure* & *Operating System*.

Heap

Maintain elements in sorted order of keys.

- alternatives: sorted array, sorted doubly linked list

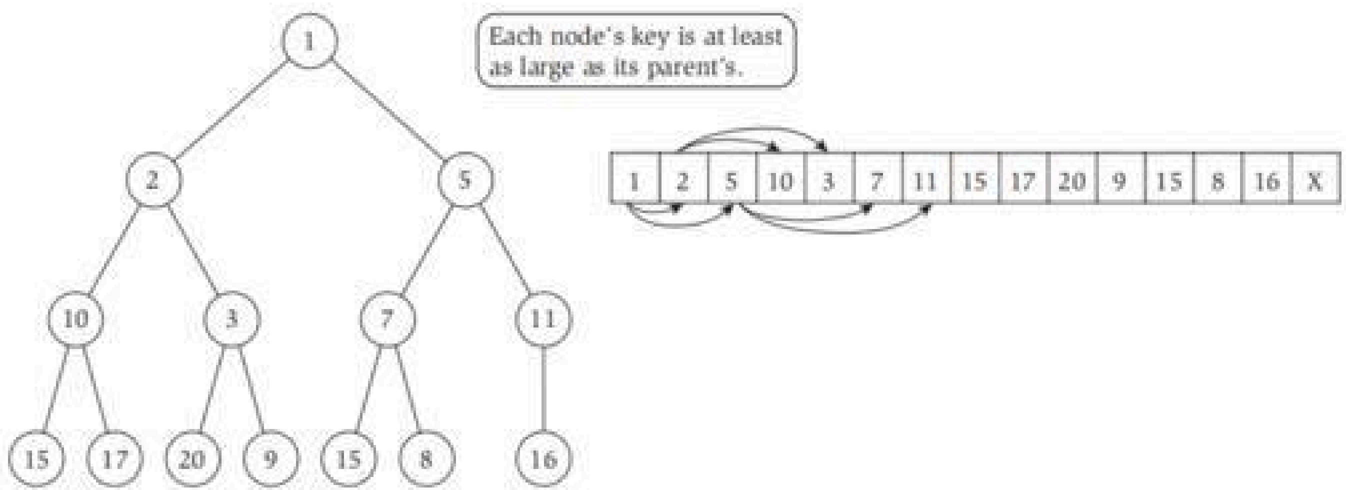
Heap

Maintain elements in sorted order of keys.

- alternatives: sorted array, sorted doubly linked list

Conceptually, think heap as *balanced binary tree*

Heap order: For every element v, at a node i, the element w at i's parent satisfies $key(w) \leq key(v)$.



Heap Operations

Heapify-up: fixing the heap by pushing the damaged part upward.

- insert a new element in a heap of n elements in $O(\log n)$ time.

Heapify-down: proceeds down the tree recursively.

- delete a new element in a heap of n elements in $O(\log n)$ time.

Implementing Priority Queues

- StartHeap(N): $O(n)$
- Insert(H, v): $O(\log n)$
- FindMin(H): $O(1)$
- Delete(H, i): $O(\log n)$
- ExtractMin(H): $O(\log n)$