Computational Tractability

Polynomial-Time

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

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 $c N^d$ steps.

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.

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.

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

Asymptotic Order of Growth

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 $\Omega(f(n))$.

$T(n)$ is also $\Theta(f(n))$ if $T(n)$ is $O(f(n))$ and $f(n)$ is $O(T(n))$.

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 $\Theta(n^3)$.

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: $T(n) \in O(f(n))$.

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

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.

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

can avoid specifying the base

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

A Survey of Common Running Times

Constant Time: O(1)

Linear time. Running time is independent of the size of the input.

E.g. Computing whether a list is empty or not.

E.g. Returning the first element of an array.

$O(\log n)$ Time

$O(\log n)$ time. Split the input into two (often equal) pieces, and work with one piece recursively.

Binary Search. Starting from a sorted list, determining if a particular value is present in that list requires $O(\log n)$ comparisons.

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
}
```

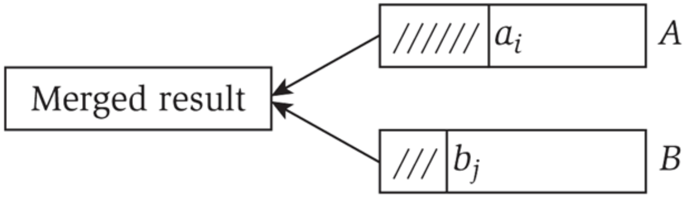
Linear Time: O(n)

Also Linear time: Completing two linear-time tasks

```
for i = 1 to n {
   do constant-time work
}
for i = 1 to n {
   do other constant-time work
}
```

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.

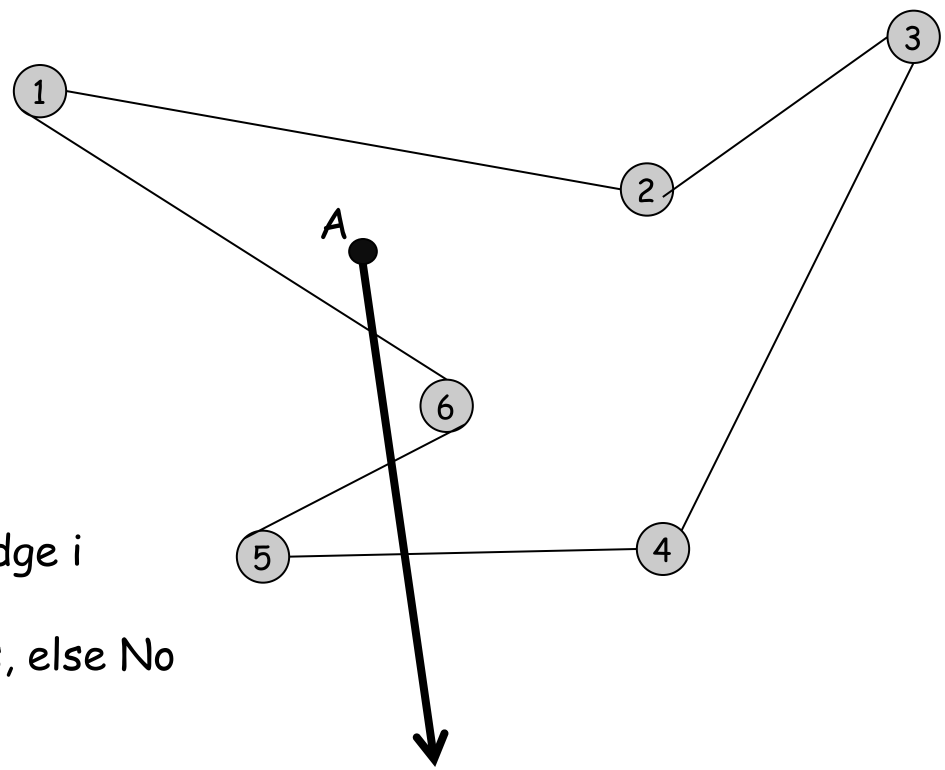



```
i = 1, j = 1
while (both lists are nonempty) {
   if (ai ≤ bj) append ai to output list and increment i
   else (ai    bj) append bj to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size n takes O(n) time.
Pf. After each comparison, the length of output list increases by 1.

Linear Time: O(n)

In or Out. Given a point $A = (a_x, a_y)$ and a set of n points (x_1, y_1), (x_2, y_2), …, (x_n, y_n) specifying a polygon, determine if the point A lies inside the polygon.

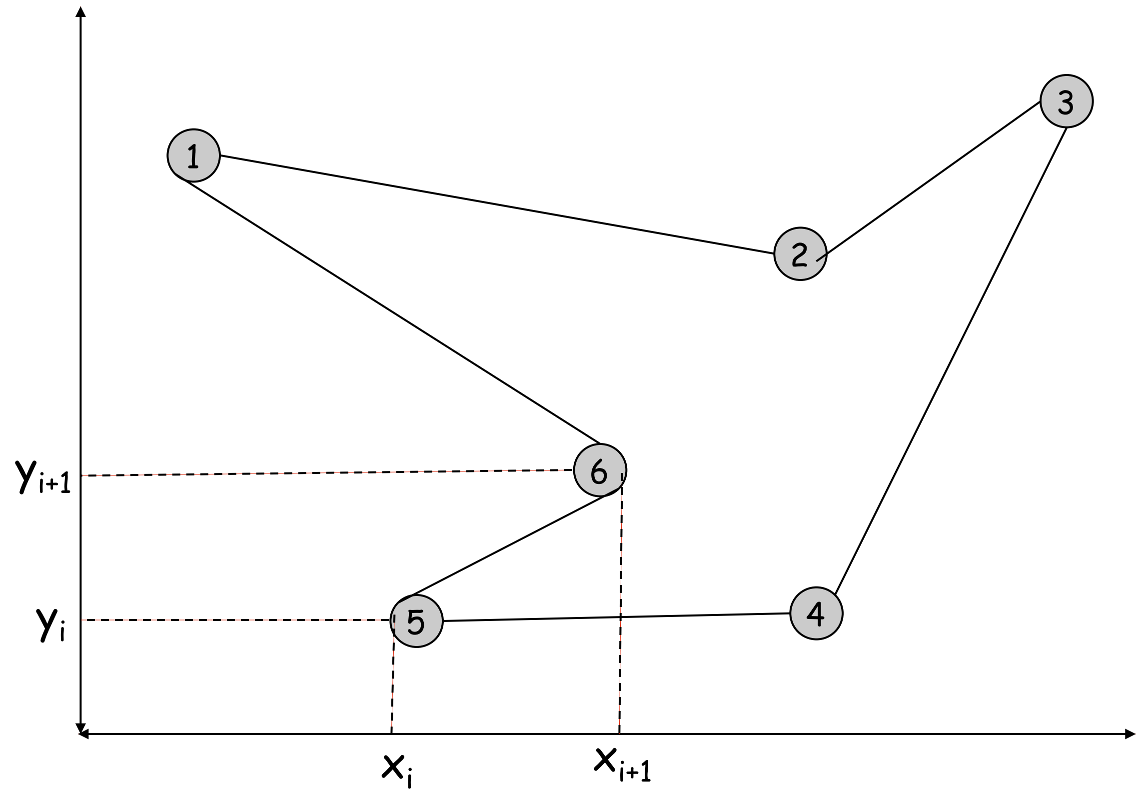


Pseudo-Code:
```
count = 0
For i = 1:n
   if A half-line crosses edge i
   count ++
If count is odd return Yes, else No
```

What if A half-line crosses a vertex?
   only count toward total if one segment above, other below

Linear Time: O(n)

Area of a Polygon. Given a set of n points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ specifying a polygon, determine the area of the polygon.



Pseudo-Code:
area = 0
For i = 1:n
    area += $(x_{i+1} - x_i)(y_i + y_{i+1}) / 2$

Return |area|

Each iteration gives the area beneath that line segment
Will be positive or negative depending on which direction the edge is going
(x's getting larger or smaller)

$O(n \log n)$ Time

$O(n \log n)$ time. Arises in divide-and-conquer algorithms – split the input into two equal pieces and call recursively.

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

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), …, (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

Efficient Implementation of Stable Matching in $O(n^2)$

Efficient implementation. We describe $O(n^2)$ time implementation. Previously we showed that the algorithm requires at most n^2 iterations, so we need to show that the work done for each iteration is constant-time.

Representing men and women.

- Assume men are named 1, …, n.
- Assume women are named 1', …, n'.

Engagements.

- Maintain a list of free men, e.g., in a stack.
- Maintain two arrays `wife[m]`, and `husband[w]`.
  - set entry to `0` if unmatched
  - if m matched to w then `wife[m]=w` and `husband[w]=m`

Men proposing.

- For each man, maintain a list of women, ordered by preference.
- Maintain an array `count[m]` that counts the number of proposals made by man `m`.

Efficient Implementation of Stable Matching in $O(n^2)$

Women rejecting/accepting.

- Does woman `w` prefer man `m` to man `m'`?
- For each woman, create inverse of preference list of men.
- Constant time access for each query after O(n) preprocessing.

Amy	1st	2nd	3rd	4th	5th	6th	7th	8th
Pref	8	3	7	1	4	5	6	2

Amy	1	2	3	4	5	6	7	8
Inverse	4th	8th	2nd	5th	6th	7th	3rd	1st

```
for i = 1 to n
   inverse[pref[i]] = i
```

Amy prefers man 3 to 6
since `inverse[3]` < `inverse[6]`

2 7

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 S_i {
   foreach other set S_j {
      foreach element p of S_i {
         determine whether p also belongs to S_j
      }
      if (no element of S_i belongs to S_j)
         report that S_i and S_j 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.

```
foreach 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?

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

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

Factorial Time

Travelling Salesman. Given a graph with n vertices including one vertex as a starting point, what is the shortest path that visits each vertex once and returns to the starting point?

O((n-1)!) solution. Consider all possible paths.

Matching problem – matching up n items with n others. Brute force solution is O(n!)

Stirling approximation: $n! \approx \sqrt{2\pi n}\,(n/e)^n$