



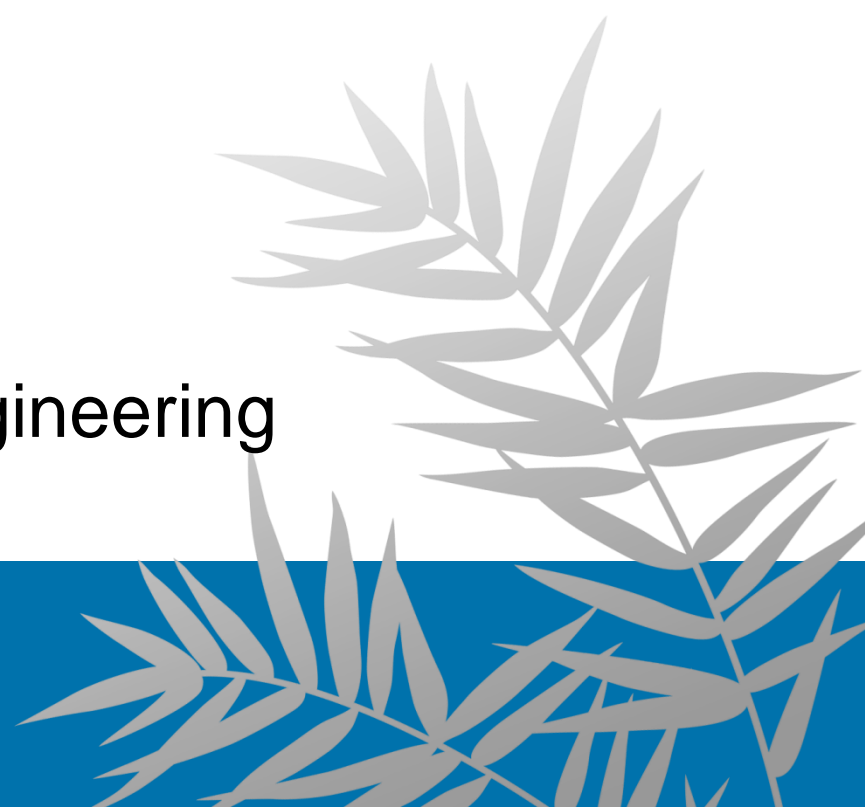
國立臺灣大學
National Taiwan University

CHAPTER 2

ALGORITHM ANALYSIS

Iris Hui-Ru Jiang
Fall 2017

Department of Electrical Engineering
National Taiwan University



Outline

- Content:
 - Computational tractability
 - Asymptotic order of growth
 - Implementing Gale-Shapley algorithm
 - Survey on common running times (Self-study)
 - Priority queues (Postponed to Section 4.5)
- Reading:
 - Chapter 2

Computational Efficiency

- Q: What is a **good** algorithm?
- A:
 - **Correct**: proofs
 - **Efficient**: run quickly and consume small memory
 - Efficiency \Leftrightarrow resource requirement \Leftrightarrow complexity
- Q: How do the resource requirements **scale** with increasing input size?
 - Time: running time
 - Space: memory usage

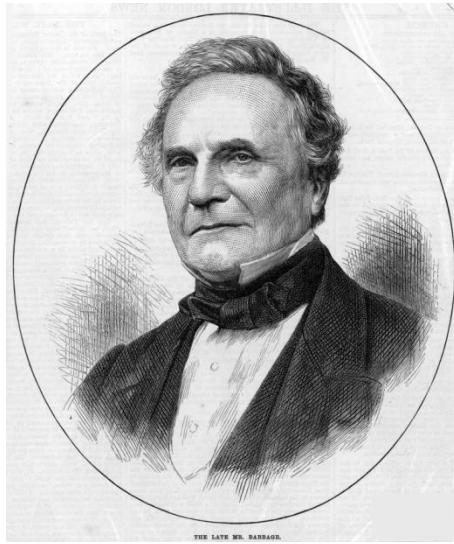


Running Time

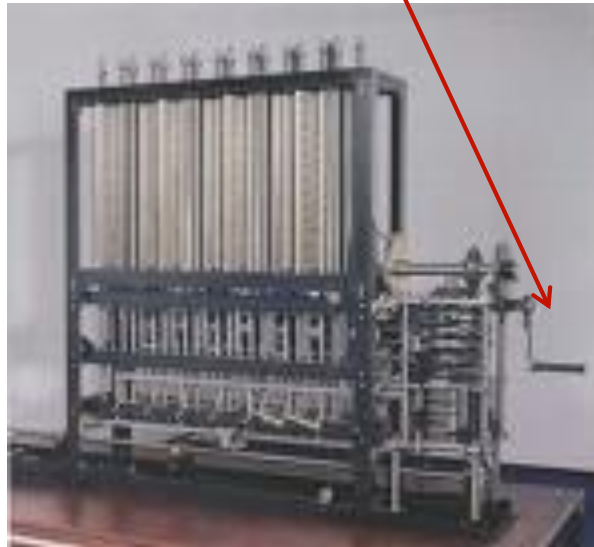
*As soon as an Analytical Engine exists,
it will necessarily guide the future course of the science.
Whenever any result is sought by its aid, the question will then arise —
by what course of calculation can these results be arrived at by the machine
in the shortest time?*

how many times
do you have to
turn the crank?

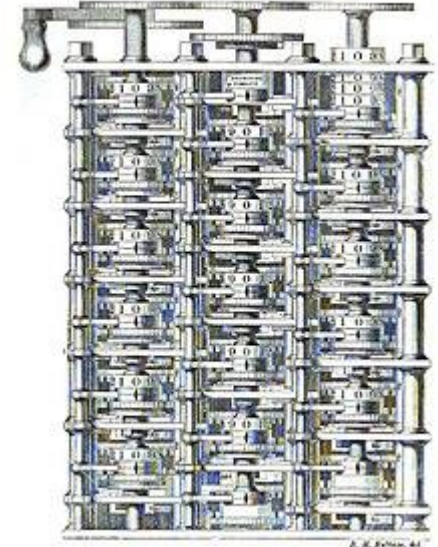
-- Charles Babbage (1864)



Charles Babbage
mathematician, philosopher,
inventor, mechanical engineer



Analytical Engine



Difference Engine

Complexity

Computational Tractability

Defining efficiency



Proposed Definition 1 (1/2)

- Q: How should we turn the fuzzy notion of an “efficient” algorithm into something more concrete?
- A: Proposed definition 1: An algorithm is efficient if, when implemented, it runs quickly on real input instances.
- Q: What’s wrong?
- A: Crucial things are missing...
 - Where and how well do we implement an algorithm?
 - Bad algorithms can run quickly with small cases on fast machines.
 - What is a real input instance?
 - We don’t know the full range of input instances in practice.
 - How well, or badly, may an algorithm scale as problem sizes grow to unexpected levels.
 - Two algorithms perform comparably on inputs of size 100; multiply the input size tenfold, and one will still run quickly, while the other consumes a huge amount of time.

Proposed Definition 1 (2/2)

- Q: What are we asking for?
- A: A **concrete** definition of efficiency
 - Platform-independent
 - Instance-independent
 - Of predictive value w.r.t. increasing input sizes
- E.g., the stable matching problem
 - Input size: N = the total size of preference lists
 - n men & n women: $2n$ lists, each of length n
 - $N = 2n^2$
 - Describe an algorithm at a high level
 - Analyze its running time mathematically as a function of N

Proposed Definition 2 (1/2)

- We will focus on the **worst-case** running time
 - A bound on the largest possible running time the algorithm could have over all inputs of a given size N .
- Q: How about **average-case** analysis?
 - The performance **averaged over random** instances
- A: But, how to generate a random input?
 - We don't know the full range of input instances
 - Good on one; poor on another
- Q: On what can we say a running-time bound is good based?
- A: Compare with **brute-force** search over the solution space
 - Try all possibilities and see if any one of them works.
 - Too slow to be useful
 - Typically takes 2^N time or worse for input size N .
 - Provide no insight into the structure of a problem

Proposed Definition 2 (2/2)

- Proposed definition 2: An algorithm is efficient if it achieves **qualitatively** better **worst-case** performance, at an **analytical** level, than **brute-force** search.
- Q: What is **qualitatively** better performance?
- A: We consider the actual running time of algorithms more carefully, and try to quantify what a reasonable running time would be.

Proposed Definition 3 (1/3)

- We expect a good algorithm has a good scaling property.
 - E.g., when the input size doubles, the algorithm should only slow down by some constant factor C .
- **Polynomial running time**: There exists constants $c > 0$ and $d > 0$ so that on every input of size N , its running time is bounded by cN^d **primitive computational steps**.
 - Proportional to N^d ; the smaller d has a better scalability.
 - The algorithm has a polynomial running time if this running time bound holds for some c and d .
- Q: What is a **primitive computational step**?
- A: Each step corresponds to...
 - A single assembly-language instruction on a standard processor
 - One line of a standard programming language
 - Simple enough and regardless to input size N

Proposed Definition 3 (2/3)

- Proposed definition 3: An algorithm is **efficient** if it has a **polynomial running time**.
- Justification: **It really works in practice!**
 - In practice, the polynomial-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 polynomial-time algorithms do have high constants and/or exponents, and are useless in practice.
 - Although $6.02 \times 10^{23} \times N^{20}$ is technically polynomial-time, it would be useless in practice.
 - Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.
 - This kind of algorithms may run quickly for average cases.

Simplex method
Unix grep



Proposed Definition 3 (3/3)

- An algorithm is **efficient** if it has a **polynomial running time**.
 - It really works in practice.
 - The gulf between the growth rates of polynomial and exponential functions is enormous.
 - It allow us to ask about the **existence** or nonexistence of efficient algorithms as a **well-defined** question.



Polynomial time

Non-polynomial 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} yrs
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 yrs	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 yrs	10^{17} yrs	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 hrs	32 yrs	very long	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 yrs	very long	very long	very long

The running times (rounded up) of different algorithms on inputs of increasing sizes, for a processor performing a million high-level instructions per second. (very long: $> 10^{25}$ yrs)

Asymptotic Order of Growth

Intrinsic computational tractability

O, Ω, Θ



Intrinsic Computational Tractability

- **Intrinsic** computational tractability: An algorithm's worst-case running time on inputs of size n grows at a rate that is at most proportional to some function $f(n)$.
 - $f(n)$: an upper bound of the running time of the algorithm
- Q: What's wrong with $1.62n^2 + 3.5n + 8$ steps?
- A: We'd like to say it grows like n^2 up to constant factors
 - Too detailed
 - Meaningless
 - Hard to classify its efficiency
 - Goal: identify broad classes of algorithms with similar behavior.
 - We'd actually like to classify running times at a coarser level of granularity so that similarities among different algorithms, and among different problems, show up more clearly.

Insensitive to constant factors
and low-order terms

O, Ω, Θ

- Let $T(n)$ be a function to describe the worst-case running time of a certain algorithm on an input of size n .
- **Asymptotic upper bound:** $T(n) = O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq cf(n)$.
- **Asymptotic lower bound:** $T(n) = \Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq cf(n)$.
- **Asymptotic tight bound:** $T(n) = \Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

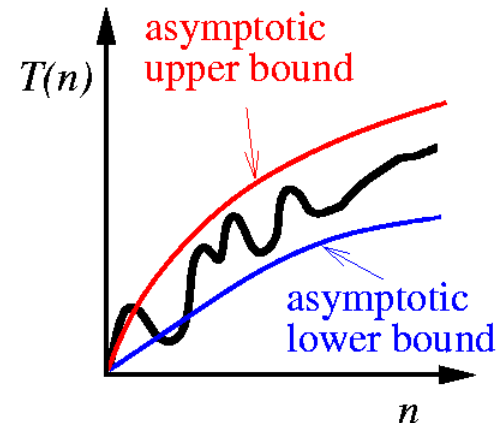
– Also, we can prove:

Let f and g be two functions that

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

exists and is equal to some number $c > 0$.

Then $f(n) = \Theta(g(n))$.

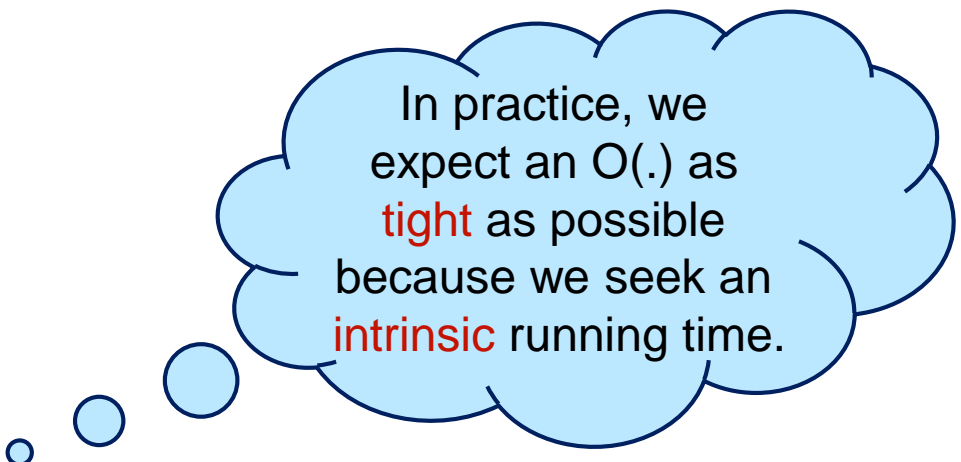


Example: O , Ω , Θ

- Q: $T(n) = 1.62n^2 + 3.5n + 8$, true or false?

1. $T(n) = O(n)$
2. $T(n) = O(n^2)$
3. $T(n) = O(n^3)$
4. $T(n) = \Omega(n)$
5. $T(n) = \Omega(n^2)$
6. $T(n) = \Omega(n^3)$
7. $T(n) = \Theta(n)$
8. $T(n) = \Theta(n^2)$
9. $T(n) = \Theta(n^3)$

- A:



In practice, we expect an $O(\cdot)$ as **tight** as possible because we seek an **intrinsic** running time.

Abuse of Notation

- Q: Why using equality in $T(n) = O(f(n))$?
 - Asymmetric:
 - $f(n) = 5n^3$; $g(n) = 3n^2$
 - $f(n) = O(n^3) = g(n)$
 - but $f(n) \neq g(n)$.
 - Better notation: $T(n) \in O(f(n))$
 - $O(f(n))$ forms a set
- Cf. “Is” in English
 - Aristotle *is* a man, but a man *isn't* necessarily Aristotle.

Properties

- **Transitivity:**
 - If $f(n) = \Pi(g(n))$ and $g(n) = \Pi(h(n))$, then $f(n) = \Pi(h(n))$, where $\Pi = O, \Omega$, or Θ .
- **Rule of sums:**
 - $f(n) + g(n) = \Pi(\max\{f(n), g(n)\})$, where $\Pi = O, \Omega$, or Θ .
- **Rule of products:**
 - If $f_1(n) = \Pi(g_1(n))$ and $f_2(n) = \Pi(g_2(n))$,
then $f_1(n)f_2(n) = \Pi(g_1(n)g_2(n))$, where $\Pi = O, \Omega$, or Θ .
- **Transpose symmetry:**
 - $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$.
- **Reflexivity:**
 - $f(n) = \Pi(f(n))$, where $\Pi = O, \Omega$, or Θ .
- **Symmetry:**
 - $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$.

Summary: Polynomial-Time Complexity

- Polynomial running time: $O(p(n))$
 - $p(n)$ is a polynomial function of input size n ($p(n) = n^{O(1)}$)
 - a.k.a. polynomial-time complexity

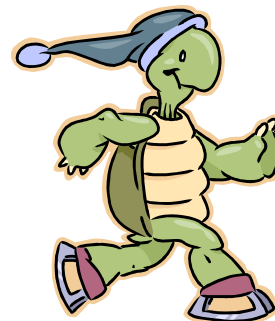
- Order

- $O(1)$: constant
- $O(\log n)$: logarithmic
- $O(n^{0.5})$: sublinear
- $O(n)$: linear
- $O(n \log n)$: loglinear
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(n^4)$: quartic
- $O(2^n)$: exponential
- $O(n!)$: factorial
- $O(n^n)$

Faster



Slower



Complexity

Implementing Gale-Shapley Algorithm

Arrays and lists



Recap: Gale-Shapley Algorithm

- Correctness:

- Termination: G-S terminates after at most n^2 iterations.
- Perfection: Everyone gets married.
- Stability: The marriages are stable.

$O(n^2)$?

- Male-optimal and female-pessimal

- All executions yield the same matching

Gale-Shapley

1. initialize each person to be free
2. **while** (some man m is free and hasn't proposed to every woman) **do**
3. w = **highest** ranked woman in m 's list to whom m **has not yet proposed**
4. **if** (w is free) **then**
5. (m, w) become engaged
6. **else if** (w prefers m to her fiancé m') **then**
7. (m, w) become engaged
8. m' become free
9. **return** the set S of engaged pairs

What to Do?

- Goal: Each iteration takes $O(1)$ time and then $O(n^2)$ in total.
 - Line 2: Identify a free man.
 - Line 3: For a man m , identify the highest ranked woman whom he hasn't yet proposed.
 - Line 4: For a woman w , decide if w is currently engaged, and if so, identify her current partner.
 - Line 6: For a woman w and two men m and m' , decide which of m and m' is preferred by w .

Gale-Shapley

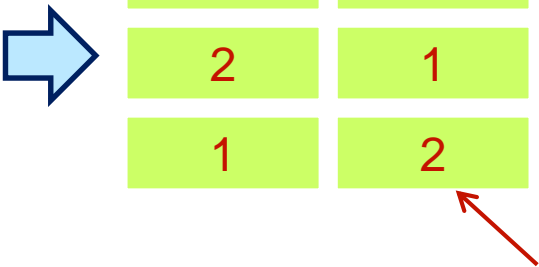
1. initialize each person to be free
2. **while** (some man m is free and hasn't proposed to every woman) **do**
3. w = highest ranked woman in m 's list to whom m has not yet proposed
4. **if** (w is free) **then**
5. (m, w) become engaged
6. **else if** (w prefers m to her fiancé m') **then**
7. (m, w) become engaged
8. m' become free
9. **return** the set S of engaged pairs

Representing Men and Women

- Q: How to represent **men** and **women**?
- A: Assume the set of men and women are both $\{1, \dots, n\}$.
 - Record in two arrays
- Q: How to store men's preference lists?
- A: Record in an $n \times n$ array or (n arrays, each of length n)

ID	Men	Men's Preference Profile			ManPref[,]		
1	Xavier	Amy	Bertha	Clare	1	2	3
2	Yancey	Bertha	Amy	Clare	2	1	3
3	Zeus	Amy	Bertha	Clare	1	2	3

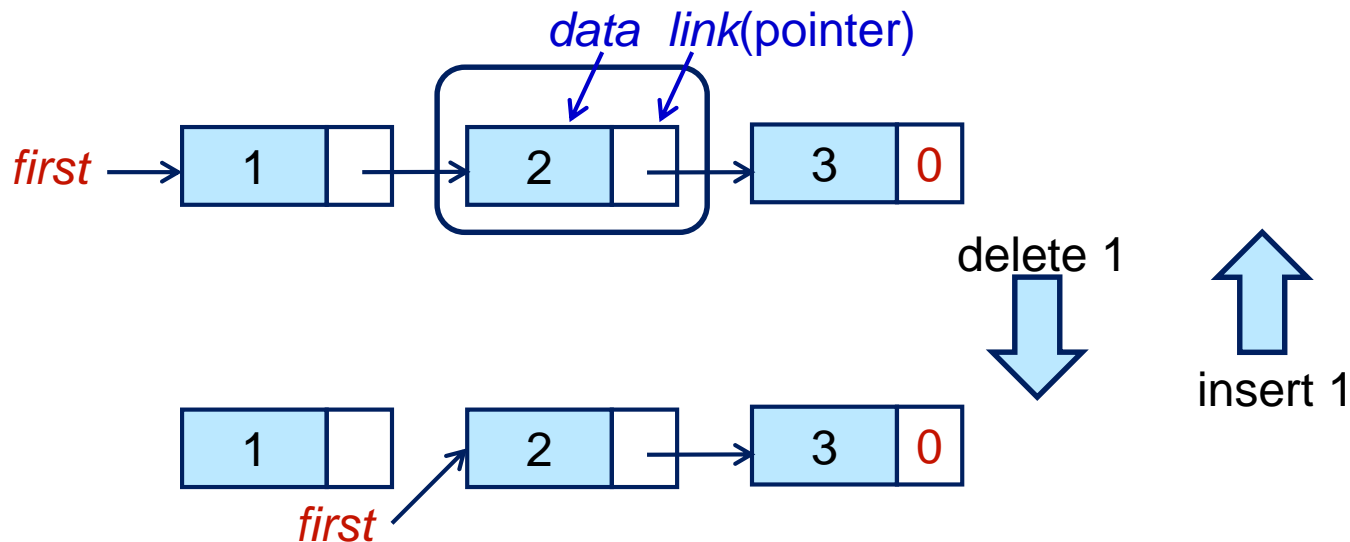
ID	Women	Women's Preference Profile		
1	Amy	Yancey	Xavier	Zeus
2	Bertha	Xavier	Yancey	Zeus
3	Clare	Xavier	Yancey	Zeus



ManPref[m , i]:
 i^{th} woman in m 's list

Identifying a Free Man

- Q: Line 2: How to identify a free man in $O(1)$ time?
- A:
 - Since the set of free men is **dynamic**, a static array is not good for insertion/deletion.
 - How about a **linked list**? It can be accessed in $O(1)$ time and allows various sizes.
 - Read/insert/delete from **first**



Whom to Propose?

- Q: Line 3: For a man m , how to identify the highest ranked woman whom he hasn't yet proposed in $O(1)$ time?
- A:
 - An extra array `Next[]` indicates for each man m the position of the next woman he will propose to on his list.
 - m will propose to $w = \text{ManPref}[m, \text{Next}[m]]$
 - $\text{Next}[m]++$ after proposal, regardless of rejection/acceptance

ID	Men	Initial value	In progress	ManPref[,]		
		Next	Next			
1	Xavier	1	2	1	2	3
2	Yancey	1	2	2	1	3
3	Zeus	1	3	1	2	3

Identifying Her Fiancé

- Q: Line 4: For a woman w , how to decide if w is currently engaged? If so, how to identify her current fiancé in $O(1)$ time?
- A:
 - An array `Current[]` of length n .
 - w 's current fiancé = `Current[w]`
 - Initially, `Current[w] = 0` (unmatched)

				Initial value	In progress			
ID	Men	ID	Women	Current	Current	WomanPref[,]		
1	Xavier	1	Amy	0	1	2	1	3
2	Yancey	2	Bertha	0	3	1	2	3
3	Zeus	3	Clare	0	0	1	2	3

Women's Preferences (1/2)


- Q: Line 6: For a woman w and two men m and m' , how to decide w prefers m or m' in $O(1)$?
- A:
 - An array $\text{WomanPref}[w, i]$ is analogous to $\text{ManPref}[m, i]$
- Q: Does it work? Why?
- A:
 - No. Scan $\text{WomanPref}[,]$ to extract the rank in $O(n)$ time
 - When man 1 proposes to woman 1 ...

ID	Women	Current	WomanPref[,]					
1	Amy	6	3	1	4	5	6	2
			1 st	2 nd	3 rd	4 th	5 th	6 th

Women's Preferences (2/2)

- Q: Can we do better?
- A: Yes.
 - Create an $n \times n$ array `Ranking[,]` at the beginning.
 - `Ranking[w, m]` = the rank of m is w 's preference list (**inverse list**)
 - When man 1 proposes to woman 1, compare them in $O(1)$ time.
 - Construct `Ranking[,]` while parsing inputs in $O(n^2)$ time.

```
for (i=1 to n) do
    Ranking[WomanPref[w, i]] = i
```

ID	Women	Current	WomanPref[,]					
1	Amy	6	3	1	4	5	6	2
			1 st	2 nd	3 rd	4 th	5 th	6 th
								
			Ranking[,]					
			2	6	1	3	4	5
	Men		1	2	3	4	5	6

What Did We Learn?

- The meaning of efficiency
 - Efficiency \Leftrightarrow polynomial-time complexity
- The analysis of an algorithm
 - We try to figure out the intrinsic running time before implementing.
 - Data structures play an important role; if necessary, preprocess the data into a proper form.

Running Times Comparison

- Q: Arrange the following list of functions in ascending order of growth rate.
 - $f_1(n) = n^{1/3}$
 - $f_2(n) = \lg n$ ($\lg = \log_2$)
 - $f_3(n) = 2^{\sqrt{\lg n}}$
- A: Convert them into a common form:
 - Take logarithm!
 - Let $z = \lg n$
 - $\lg f_1(n) = \lg n^{1/3} = (1/3)\log n = (1/3)z$
 - $\lg f_2(n) = \lg (\lg n) = \lg z$
 - $\lg f_3(n) = \lg (2^{\sqrt{\lg n}}) = \sqrt{\lg n} = \sqrt{z}$
 - DIY the rest! (see Solved Exercise 1)