Objectives

- Proving correctness of Stable Matching algorithm
- Analyzing algorithms
- Asymptotic running times

Wiki
Everyone log in okay?
Decide on either using a blog or wiki-style journal?

Review

- What is the stable matching problem?
  - What is given?
  - What is output?
- Provide a sketch of the algorithm
- What were our observations about how a woman's state changed over the duration of the algorithm?

Review: Observations about the Algorithm

- What can we say about any woman's partner during the execution of the algorithm?
  - Observation 1. He gets "better" → she prefers him over her last partner
- How does a woman's state change over the execution of the algorithm?
  - Observation 2. Once a woman is matched, she never becomes unmatched; she only "trades up"
- What can we say about a man's partner?
  - Observation 3. She gets "worse"

Propose-And-Reject Algorithm

[Gale-Shapley 1962]

Does algorithm terminate?

```
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
```


Review: Proof of Correctness: Termination

- Claim. Algorithm terminates after at most n^2 iterations of while loop
  - (not yet commenting on the time required for the body of the while loop)
- Pf. Each time through the while loop, a man proposes to a new woman. There are only n^2 possible proposals.

Algorithm Analysis

Prove that final matching is a *perfect* matching

- **Perfect matching**: everyone is matched monogamously
- Hint: in algorithm, we know if *m* is free at some point in the execution of the algorithm, then there is a woman to whom he has not yet proposed.

Proof of Correctness: Perfection

- Claim. All men and women get matched.
- Pf. (by contradiction)
  - Where should we start?

Suppose that some man *m* is not matched upon termination of algorithm

Proof of Correctness: Perfection

- Claim. All men and women get matched.
- Pf. (by contradiction)
  - Suppose that *m* is not matched upon termination of algorithm
  - Then some woman, say *w*, is not matched upon termination.
  - By Observation 2, *w* was never proposed to.
  - But, last man proposed to everyone, since he ends up unmatched
    - (by the while loop's condition)
  - Contradiction ▪

Proof of Correctness: Stability

- Claim. No unstable pairs.

What does it mean for a given matching S* to be unstable?

S*
Amy-Yancey
Bertha-Zeus
…

How do you think we should approach this proof?

Proof of Correctness: Stability

S*
Amy-Yancey
Bertha-Zeus
…

- Claim. No unstable pairs.
- Pf. (by contradiction)
  - Suppose m-w is an unstable pair: each prefers each other to partner in Gale-Shapley matching S*.

What are the possibilities that lead to this?

Proof of Correctness: Stability

S*
Amy-Yancey
Bertha-Zeus
…

- Claim. No unstable pairs.
- Pf. (by contradiction)
  - Suppose m-w is an unstable pair: each prefers each other to partner in Gale-Shapley matching S*.
  - **Case 1**: **m never proposed to w** ⇒ m prefers his GS partner to w. ⇒ m-w is stable.
    - men propose in decreasing order of preference
  - **Case 2: m proposed to w** ⇒ w rejected m (right away or later) ⇒ w prefers her GS partner to m. ⇒ m-w is stable.
    - women only trade up
  - In either case m-w is stable, a contradiction. ▪

Summary So Far…

- **Stable matching problem.** Given *n* men and *n* women and their preferences, find a stable matching if one exists.
- **Gale-Shapley algorithm.** Guarantees to find a stable matching for *any* input

Remaining Questions:
- If there are multiple stable matchings, which one does GS find? *(see book)*
- How to implement GS algorithm efficiently? (Monday)
  - What is our goal running time?

Review: Our Process

1. Understand/identify problem
   - Simplify as appropriate
2. Design a solution
3. Analyze
   - Correctness, efficiency
   - May need to go back to step 2 and try again
4. Implement
   - Within bounds shown in analysis

Stable Matching Summary

- **Stable matching problem.** Given preference profiles of *n* men and *n* women, find a *stable* matching.

  no man and woman prefer to be with each other than assigned partner

- **Gale-Shapley algorithm.** Finds a stable matching in $O(n^2)$ time.
  - Claim: can implement algorithm *efficiently*

TODAY'S GOAL:
DEFINE ALGORITHM EFFICIENCY

Our Process

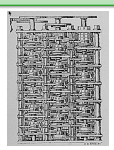
1. Understand/identify problem
   - Simplify as appropriate
2. Design a solution
3. Analyze
   - Correctness, efficiency
   - May need to go back to step 2 and try again
4. Implement (On Monday)
   - Within bounds shown in analysis

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)

http://plan28.org/

Brute Force

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

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

Example: How many possible solutions are there in the stable matching problem?
In other words, how many possible *perfect* matchings are there?
For each perfect match, we'll check if it's stable.

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
- Example: Stable matching: n! with n men and n women
  - If n increases by 1, what happens to the running time?

"Exponential"

How Do We Measure Runtime?

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

What are alternatives to worst-case analysis?

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

Towards a Definition of Efficient…

- Desirable scaling property: When input size doubles, algorithm should only slow down by some constant factor C
  - Doesn't grow multiplicatively

Polynomial-Time

Defn. 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.

✓ Desirable scaling property: When input size doubles, algorithm should only slow down by some constant factor C
  - What happens if we double N?
- Defn. An algorithm is *polynomial time* (or *polytime)* if the above scaling property holds.

Algorithm Efficiency

- **Defn.** An algorithm is ***efficient*** if its running time is *polynomial*
- Justification: It really works in practice!
  - In practice, 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 ($6.02 \times 10^{23} \times N^{20}$) and are useless in practice
  - Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare

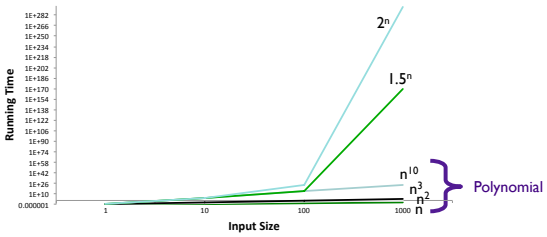
Running Times

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.

Input Size	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

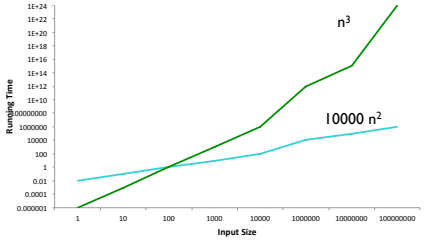
Polynomial

Visualizing Running Times



- Huge difference from polynomial to not polynomial
- Differences in runtime matter more as input size increases

Comparing 10000 n^2 and n^3



As input size increases, n^3 dominates large constant * n^2
➔Care about running time as input size approaches infinity
➔Only care about highest-order term

Asymptotic Order of Growth: Upper Bounds

- **T(n)** is the worst case running time of an algorithm
- We say that T(n) is **O(f(n))** if there exist constants

"order f(n)"

c cannot depend on n

sufficiently large n

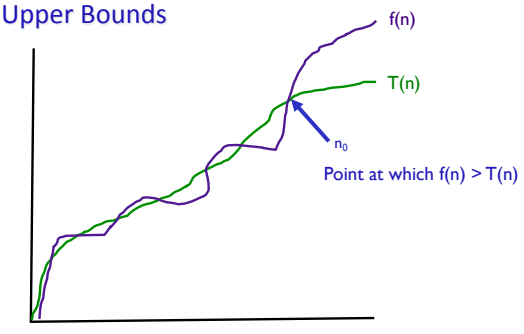
$c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have

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

T(n) is bounded above by a constant multiple of f(n)

➔T is ***asymptotically upperbounded*** by f

Asymptotic Order of Growth: Upper Bounds



Looking Ahead

- Read/summary on Wiki
  - 2 pages of preface, 1.1, 2.1-2.2
  - Due Monday/Tuesday
- Problem Set
  - Due Friday before class
  - Individual submissions
- Suggestion: read over problem set before reviewing text book
  - Have some objectives in mind/associates to make during reading