Recap: Stable Matching Problem
- Definition of a Stable Matching
- Stable Roommate Matching Problem
  - Stable matching does not always exist!
- Gale-Shapley Algorithm (Propose-And-Reject)
  - Proof that Algorithm Terminates in $O(n^2)$ steps
  - Proof that Algorithm Outputs Stable Matching
  - Matching is male-optimal
    - If there are multiple different stable matchings each man gets his best valid partner
  - Matching is female-pessimal
    - If there are multiple different stable matchings each man gets her worst valid partner

Extensions: Matching Residents to Hospitals
Ex: Men = hospitals, Women = med school residents.

Variant 1: Some participants declare others as unacceptable.

Variant 2: Unequal number of men and women.

Variant 3: Limited polygamy.
  - Hospital X wants to hire 3 residents
  - Gale-Shapley Algorithm Still Works. Minor modifications to code to handle variations!

Def: Matching $S$ unstable if there is a hospital $h$ and resident $r$ such that:
- $h$ and $r$ are acceptable to each other;
- either $r$ is unmatched, or $r$ prefers $h$ to her assigned hospital; and
- either $h$ does not have all its places filled, or $h$ prefers $r$ to at least one of its assigned residents.

1.2 Five Representative Problems

Interval Scheduling
Input: Set of jobs with start times and finish times
Goal: Find maximum cardinality subset of mutually compatible jobs.
Interval Scheduling

Input: Set of jobs with start times and finish times.
Goal: Find maximum cardinality subset of mutually compatible jobs.

Greedy Choice: Select job with earliest finish time and eliminate incompatible jobs.

Chapter 4: We will prove that this greedy algorithm always finds the optimal solution!

Weighted Interval Scheduling

Input: Set of jobs with start times, finish times, and weights.
Goal: Find maximum weight subset of mutually compatible jobs.

Greedy Algorithm No Longer Works!
Weighted Interval Scheduling

**Input:** Set of jobs with start times, finish times, and weights.

**Goal:** Find maximum weight subset of mutually compatible jobs.

Greedy Algorithm No Longer Works!

![Graph showing job intervals with weights]

Problem can be solved using technique called Dynamic Programming

Bipartite Matching

**Input:** Bipartite graph.

**Goal:** Find maximum cardinality matching.

![Graph showing bipartite matching]

Different from Stable Matching Problem How?

Independent Set

**Input:** Graph.

**Goal:** Find maximum cardinality independent set.

A subset of nodes such that no two joined by an edge.

NP-Complete: Unlikely that efficient algorithm exists!

Brute-Force Algorithm: Check every possible subset.
Running Time: \(2^n\) steps

Positive: Can easily check that there is an independent set of size \(k\)

Competitive Facility Location

**Input:** Graph with weight on each node.

**Game:** Two competing players alternate in selecting nodes.
Not allowed to select a node if any of its neighbors have been selected.

**Goal:** Select a maximum weight subset of nodes.

Second player can guarantee 20, but not 25.
Competitive Facility Location

**Input.** Graph with weight on each node.

**Game.** Two competing players alternate in selecting nodes. Not allowed to select a node if any of its neighbors have been selected.

**Goal.** Select a maximum weight subset of nodes.

Second player can guarantee 20, but not 25.

PSPACE-Complete: Even harder than NP-Complete!

No short proof that player can guarantee value B. (Unlike previous problem)

Five Representative Problems

- Variations on a theme: independent set.
- Interval scheduling: $n \log n$ greedy algorithm.
- Weighted interval scheduling: $n \log n$ dynamic programming algorithm.
- Bipartite matching: $m^2 \log m$ max-flow based algorithm.
- Independent set: NP-complete.
- Competitive facility location: PSPACE-complete.
2.1 Computational Tractability

"For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing." - Francis Sullivan

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)

Computational Tractability

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.

Alexander Engine (schematic)

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. Unsatisfactory in practice.

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

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

Choosing C = 2^d

Worst-Case Polynomial-Time

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

Justification. It really works in practice!

- Although \(6.02 \times 10^{23} \approx 2^{70}\) 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.
2.2 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 \( O(n), \Omega(n), \Omega(n^3), \) or \( \Theta(n^3) \).

**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 \( f = O(g) \) and \( g = O(h) \) then \( f + g = O(h) \).
- If \( f = \Omega(g) \) and \( g = \Omega(h) \) then \( f + g = \Omega(h) \).
- If \( f = \Theta(g) \) and \( g = O(h) \) then \( f + g = \Theta(h) \).

**Notation**

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

- Not transitive:
  - \( f(n) = 5n^2, g(n) = 3n^2 \)
  - \( f(n) = O(g(n)) \) but \( f(n) \neq O(f(n)) \)

- Better notation: \( T(n) = \Theta(f(n)) \).

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

- Statement doesn’t “type-check.”
- Use \( \Omega \) for lower bounds.

**Asymptotic Bounds for Some Common Functions**

**Polynomials.** \( a_0 + a_1n + \ldots + a_ddn^d \) 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 \( a, b > 0 \).

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

Log grows slower than every polynomial.

Every exponential grows faster than every polynomial.
2.4 A Survey of Common Running Times

Linear Time: $O(n)$

**Merge.** Combine two sorted lists $a = a_1, a_2, ..., a_n$ with $b = b_1, b_2, ..., b_m$ into sorted whole.

$i = 1, j = 1$
while (both lists are nonempty) {
  if ($a_i \leq b_j$) 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

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

**Largest empty interval.** Given $n$ time-stamps $x_1, ..., x_n$ 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: $O(n^2)$

**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.

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

Cubic Time: $O(n^3)$

**Set disjointness.** Given $n$ sets $S_1, ..., S_n$ 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 pair of sets, determine if they are disjoint.

1/9/2018
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?

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

```plaintext
foreach subset $S$ of $k$ nodes {
    check whether $S$ is an independent set = $O(k^2)$.
    number of $k$ element subsets = $O(k^2 n^k / k!) = O(n^k)$.
}
```

```plaintext
foreach subset $S$ of $k$ nodes {
    check whether $S$ is an independent set = $O(k^2)$.
    if ($S$ is an independent set) report $S$ is an independent set.
}
```

$\binom{n}{k} \leq \frac{n^k}{k!}$

$\binom{n}{k} \leq \frac{n^k}{k!}$

$\frac{n^k}{k!}$

Poly-time for $k=17$, but not practical if $k$ is a constant.

Exponential Time

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

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

```plaintext
foreach subset $S$ of nodes {
    check whether $S$ is an independent set.
    if ($S$ is largest independent set seen so far) update $\hat{S} = S$.
}
```

Heap Data Structure

Min Heap: For each node $v$ in the tree:
- $\text{Parent}(v).Value \leq v.Value$

Max Heap: For each node $v$ in the tree:
- $\text{Parent}(v).Value \geq v.Value$

Heap Insertion

Min Heap Order: For each node $v$ in the tree:
- $\text{Parent}(v).Value \leq v.Value$

Max Heap Order: For each node $v$ in the tree:
- $\text{Parent}(v).Value \geq v.Value$

Heap Insertion

Min Heap Order: For each node $v$ in the tree:
- $\text{Parent}(v).Value \leq v.Value$

Max Heap Order: For each node $v$ in the tree:
- $\text{Parent}(v).Value \geq v.Value$
Heap Insertion

Min Heap Order: For each node v in the tree
Parent(v).Value ≤ v.Value

Theorem 2.12 [KT]: The procedure Heapify-up fixes the heap property and allows us to insert a new element into a heap of n elements in \(O(\log n)\) time.

Heap Extract Minimum

Min Heap Order: For each node v in the tree
Parent(v).Value ≤ v.Value

Theorem 2.13 [KT]: The procedure Heapify-down fixes the heap property and allows us to delete an element in a heap of n elements in \(O(\log n)\) time.
Min Heap Order: For each node $v$ in the tree, $\text{Parent}(v).\text{Value} \leq v.\text{Value}$

**Theorem 2.13 [KT]:** The procedure Heapify-down fixes the heap property and allows us to delete an element in a heap of $n$ elements in $O(\log n)$ time.

**Heap Summary**

- Insert: $O(\log n)$
- FindMin: $O(1)$
- Delete: $O(\log n)$ time
- ExtractMin: $O(\log n)$ time

Thought Question: $O(n \log n)$ time sorting algorithm using heaps?