stable matching

- input: given 2 distinct groups of people each with their own rankings of people from the opposite group
- output: stable matching of people: there does not exist 2 pairs (x1, y1), (x2, y2) such that x1 likes y2 more than y1 and y2 likes x1 more than x2 Gale-shapley algorithm:

```
while # exists free man, m, that has not proposed to every woman:
  w = # first woman on m's list not he has not proposed to yet
  if w.status = free:
    engage(w, m)
  else if # w prefer m over current partner m_old:
    engage(w, m)
    free(m_old)
  else:
    continue # w rejects m
```

- men optimal: gale shapely always returns arrangement where men get their best possible stable matching → reason because men always propose from highest down, if highest is stable already will not change
- women pessimality: gale shapely always returns worst stable matching for women
- **Word ram model**: RAM = array of words; runtime counts instructions (e.g. mul = 1), ignore compiler optimizations/ OS multithreading etc.
- Notations: for function `f(n)` w/ asymptotic `(n^2)`
  - Big O: `O(n^2)` → `0 <= f(n) <= c*(n^2)` for some constant c and where `n > k`, for some constant k.
  - Big Omega: `Ω(n^2)` → `0 <= c*(n^2) <= f(n)` for some constant c and where `n > k`, for some constant k.
  - Big Theta: `Θ(n^2)` → `0 <= a*(n^2) <= f(n) <= b*(n^2)` for some constant a, band where `n > k`, for some constant k.
  - small o `o(n^2)`, omega `ω(n^2)`: like big notation, but **not equal** → `o(n^2)`, `f(n) < c*(n^2)` for all values c, when `n > k` for some constant k
- Limit method: as n → infinity, `f(n)/g(n)` → x
  - `x == 0`: `f(n) = o(g(n))`, since `f(n)/g(n) = 0`, `f(n)/g(n) < k`, `f(n) < k*g(n)`.
  - `x < infinity`: `f(n) = O(g(n))`.
  - `x < infinity, x > 0`: `f(n) = Θ(g(n))`.
  - `x > 0`: `f(n) = Ω(g(n))`.
  - `x == infinity`: `f(n) = ω(g(n))`.

Asymptotic useful shit

- **Stirling approximation**: `log(n!) = Θ(nlogn)`; also, `log(a) * log(b) = log(ab)`.
- e^x >= `1 + x`
- k^n > n^a, for any k > 1, and a.
- `lg^2(n) == (lgn)^2 != lglgn`, `lg(ab) == lga + lgb`, `lg(a^n) == nlga`, `logb(a) == 1/loga(b)`, `a^logb(c) == c^logb(a)`.
- 1 + 1/2 + 1/3 + ... = `ln(n) + O(1)` (Harmonic), 1 + 2 + ... n = `n(n + 1)/2` (AP), 1 + x + x^2 + ... x^n = `(x^n + 1 - 1)/(x - 1)` (GP).
- L'hopitals: lim x → infinity, `f(x)/g(x) == f'(x)/g'(x)`, e.g. (`logn/n == 1/n/1`)

Correctness

Using a loop invariant (e.g. `array [1 .. j] sorted after j iterations`)

1. initialization: show loop invariant true before iteration
2. mantenance: show if invariant true before iteration, remain true after 1 iteration
3. termination: when algo terminates, can be used as property to show correctness (e.g. `array [1..n-1] is sorted and no element gt [n] exists`, imply whole array sorted)

solving recurrence

1. Recursion tree: draw whole recursion tree, e.g. merge sort; lgn levels each level n ops, total nlgn
2. master method: `T(n) = aT(n/b) + f(n)` where a >= 1, b > 1, `f` asymptotic positive. compare `f(n)` w/ `n^(log(a)baseb)`.
   1. `f(n) = O(n^(log a base b - k))` for some constant `k > 0`: `T(n) = Θ(n ^ (log a base b))`, fn grows polynomially slower than n ^ log a base b (by n ^ k factor) $f(n) = O(n^{\log_b a - \varepsilon})$ $T(n) = \Theta(n^{\log_b a})$
   2. `f(n) = Θ(n^(log a base b) * lg ^ k(n))`, `k >= 0`, (e.g. k = 2 → lglgn), then `T(n) = Θ(n ^ (log a base b) * lg ^ (k + 1)(n))`, both grow at similar rates $f(n) = \Theta(n^{\log_b a} \lg^k n)$ $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$
   3. `f(n) = Ω(n^(log a base b + k))`, `k > 0`, `T(n) = Θ(f(n))`, `f(n)` grows faster by `n ^ k`, **satisfies regularity condition** `af(n/b) <= cf(n)` for some c $f(n) = \Omega(n^{\log_b a + \varepsilon})$ $T(n) = \Theta(f(n))$
3. substitution method:
   1. guess a time complexity, e.g. `T(n) = n^2`, so T(n) <= cn^2 for some c
   2. substitute recurring T's with eqn: `T(n) = T(n/2)` → `T(n) = c(n^2)/4`.
   3. show result is less than guess: `c(n^2)/4 <= cn^2`.

Randomized algorithms

- Basis works on E(X) = sum(All possible cases * probability of each case)
- Ex: quicksort, let Ai be a permutation where the ith largest element is the first index (i.e. picked as pivot). E(X) = sum(Expected Ai runtime) / n
- Randomized algorithms usually are simple, can approximate good solutions much faster than deterministic

Hashing

- `U`: universe, all possible values to be hashed, `M`: the size of array to be mapped to, `N`: number of stored items
- Randomization: randomize the hash function used, so we do not keep colliding on specific indexes
- Universal hashing: suppose randomized hash function, set of all hash functions = `H`. Universal hashing → `|h(x) == h(y)|/|H| <= 1/M`: for all x, y where `x != y`, number of hash functions `h` that collide / number of hash functions total <= `1/M`.
- For any universal set of hash functions mapping U → M, for any N elements the expected number of collisions < N/M
- For any universal set of hash functions,if M >= N, expected cost for N insertion/deletion/queries = O(N) → for each in N expected cost O(N/M) = O(1) each

- Perfect hashing: expected worst case constant time, if `M=N^2`, expected num collisions < 1. (**Theorem: there exists hash function U → M where |M| = N^2 with no collisions**)
  - 2 level scheme: each index is another hash map. at each index choose second level hash function with no collisions (`M=N^2`)
  - if H is universal, Expected size of all secondary arrays in 2 level scheme < 2n

Amortized analysis

- `O(n)` = worst case of a single operation; Amortized worst case = average each case in worst case running all k operations.
- **NOT the same as average case analysis**.
- Aggregate method
  - sum worst possible k operations, divide by k. (if many diff operations pick worst possible permuation of operations)
  - tedious, provide upper bound on true cost. cannot provide amortized cost of each operation, only of whole set of operations
- Accounting/ Bankers method
  - Assign common, low cost ops a higher cost, which pays off higher cost, less common operations e.g. increase `INSERT` cost by 1 so `DELETE ALL` cost can be 0
  - Actual cost must be <= amortized cost
- Potential method
  - define potential function ϕ(i) wuch that ϕ(i) >= 0 for all i, ϕ(i) = potential at the end of ith operation. Try to select ϕ such that costly operation, Δφ is negative, negates actual expensive cost. (find decreasing qty during operation)
  - Amortized cost of ith operation = `actual_cost(i) + φ(i) - φ(i - 1)`.(potential difference, `Δφ(i)` = `φ(i) - φ(i - 1)`)
  - Amortized cost n operations = actual cost n ops + `φ(n) - φ(0)`.
  - To show actual cos n ops = `O(f(n) + φ(0))`, just show amortized `O(f(n))`.
  - e.g. binary addition, expensive operation, flip k bits to 0, longest suffix of 1's decrease. `φ(i) = 1 + length longest suffix`.
  - draw table, columns `actual cost`, Δφ, `amortized cost`. each row = 1 case, e.g. expensive case + normal case. show amortized cost still same, total cost = n * amortized cost.