CSCI 270 MIDTERM CHEATSHEET

December 8, 2022

1. Stable Matching

- A **matching** is a graph in which each node is incident on $\leqslant 1$ edge.
- A **perfect matching** is ... on exactly one edge.
- A **bipartite** graph is one where nodes are divided into two sets, and edges only exist between the two sets, not inside either.
- Given a bipartite instance with rankings, a matching is **stable** if for each pair (u, s) *not* assigned (no edge), either u prefers its assigned node over s or s their assigned node over u.
- Stable matching is not unique: men A, B, women $1, 2$, A prefers 1, B prefers 2, 1 prefers B, and 2 prefers A, then $\{(A,1),(B,2)\}$ and $\{(B,1),(A,2)\}$ are both stable.

Gale-Shapley proposal algorithm: WLOG assume there are n men and n women, each with a ranking of everyone of the opposite gender (if numbers don't equal, we can create dummies with bottom rankings). The goal is to create a stable matching between men and women.

```
Start with empty assignment
While there is still single man:
  - Pick single man m
  - Let w be m's highest ranked woman not yet
      proposed
  - If w is single OR prefers m over her
      current partner:
    - Match m with w
    - If w had partner, he becomes single
  - Else: do nothing
```

Key facts of G-S:

- Once a woman becomes matched, she never becomes single again and her matches can only improve.
- The algorithm terminates. There cannot be a single man forever: if so, since #men = #women, there will be a single woman, so she was never proposed to. The man can just proposed!

Stability proof

Proof.
- Assume not. Let (m, w') and (m', w) be pairs with m, w preferring each other.
- m have once proposed to w, and it took place earlier than when m proposed to w'.
- m either got rejected or later dumped because of some other m'' that w preferred over m.
- But then w must have chosen m'' at some point. Since she ended up with m', $m' \geqslant m''$ for w, and so $m' \geqslant m'' > m$ for w, contradiction. □

Man-optimality of G-S: we first define $P(m)$:= the set of all women w that m can end up with in some stable matching, and we define the best valid choice, $b(m)$, to be the best choice in $P(m)$ according to m's ranking.

Theorem

Gale-Shapley returns a matching in which each m is matched with $b(m)$.

Proof.
- Suppose not, that some man is not with their best valid choice $b(\cdot)$ in G-S. At some point he must have been rejected/dumped by best valid choice.
- Look at the first time that some man m got dumped/rejected by some woman in $P(m)$. Call this woman w.
- The man m' that w rejects/dumps m for must have $m' > m$ on w's ranking.
- Since $w \in P(m)$, there exists stable matching M' with (m, w) matched. In this matching, let w' be the partner of m'.
- *If m' preferred w over w', then in M', m' and w would have resulted in the matching being unstable.* Therefore, m' prefers w' over w.
- In GS, m' ended up being with w, so he must have been rejected/dumped by w', and this happened even before w rejected/dumped m.
- We assumed (m', w') could be matched in a stable matching, so we have a contradiction that m being rejected by w is *not* the first time *some* man gets rejected by *some* woman in his $P(\cdot)$.
- Therefore G-S is man-optimal. □

2. Greedy Algorithms

Greedy algorithm in a nutshell: pick whatever choice seems best at the moment and address future problems later.

Example – interval selection: given intervals $[a_i, b_i]$, pick as many without overlap as possible.

Algorithm $\mathcal{O}(n \log n)$:

```
Sort intervals by non-decreasing finish times
    f(i)
R = all intervals, A = {}
while R not empty
  - let i be the index of earliest finishing
      intervals in R
  - add i to A
  - remove interval i and all intervals
      intersecting it from R
```

Greedy stays ahead: if $i(1) < i(2) < \ldots < i(k)$ and $j(1) < j(2) < \ldots < j(k)$ are the first k intervals picked by greedy and by any optimal solution, then the end time of $i(k) \leqslant$ the end time of $j(k)$. *In each stage, greedy performs no worse than the optimal solution.*

Example – job selection: Given, n jobs with deadlines d_i, $1 \leqslant i \leqslant n$. do all jobs and minimize max lateness. NOTE: optimal solution contains no rest between jobs.

Algorithm: do jobs in the order of their deadlines, from earliest to latest.

Optimality – adjacent inversion: if the greedy order is $G(i) = i$ for $1 \leqslant i \leqslant n$ and the optimal solution $\mathcal{O}$ is not identical, then there exist jobs i, j forming an *adjacent inversion*:

- job j is scheduled immediately after job i by optimal $\mathcal{O}$
- deadline of j is earlier than deadline of i ($d_j < d_i$).

Optimality – proof:

- Let G be greedy output and $\mathcal{O}$ any optimal solution.
- Pick adjacent, inverted jobs i, j in $\mathcal{O}$.
- Show switching jobs i, j in $\mathcal{O}$ preserves optimality.
- Recover G from $\mathcal{O}$, proven.

Example: MST construction: given a graph with distinct edge weights $G = (V, E)$, construct a MST (min cost connected subgraph of G). NOTE: MST cannot contain cycles.

Proposition: Cut Property

A **cut** is a partition (S, S') of V.
If an edge e is the cheapest among all edges crossing *some* cut (S, S') then $e \in$ *every* MST.

(Proof sketch: add e to form a circle and remove the original cut-crossing edge; this results in a lower total cost subgraph.)

Kruskal's algorithm:

```
Sort edges by increasing cost
For each edge e in this order
  Add e if it does not create a cycle
```

Proof.
- Any edge added is a min cost edge across some cut, so output ⊂ MST.
- Any two disconnected components have edges connecting them, and Kruskal adds $\geqslant 1$ such edge, so output is a spanning tree. □

Kruskal's algorithm runs in $\mathcal{O}(m \log m) + \mathcal{O}(m \log^* n) = \mathcal{O}(m \log m)$ using **Union-Find** for $\mathcal{O}(\log^* n)$ amortized lookup and merge.

Prim's algorithm $\Theta(m \log n)$:

```
Start with any S = {s}
While S ≠ V
  Find the cheapest edge e = (u,v) between S
      and S'
  Add e to tree, add v to S
```

Proof. Output ⊂ MST by cut property. Output obviously is a spanning tree. □

3. Divide & Conquer

Divide & Conquer in a nutshell:

- Take a problem instance I of size n
- Divide into smaller $I(1), \cdots, I(k)$.
- Solve small instances separately and return solutions $\mathrm{Sol}(j)$.
- Post-process solutions to produce a big solution.
- If input small enough, directly solve.
- **Example**: merge sort. *Proof sketch*: prove correctness of `merge` by induction, then prove correctness of `MergeSort` by induction again.

Theorem: Master Theorem

Let $a \geqslant 1, b > 1$. Assume some recursion relation's complexity satisfies

$$T(n) = aT(n/b) + f(n) \qquad T(1) = \Theta(1).$$

(MT1) If $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $\boxed{T(n) = \Theta(n^{\log_b a})}$.

(MT2) If $f(n) = \Theta(n^{\log_b a})$, then $\boxed{T(n) = \Theta(n^{\log_b a} \log n)}$.

(MT3) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ and $\lim_{n\to\infty} \frac{af(n/b)}{f(n)} < \infty$ then $\boxed{T(n) = \Theta(f(n))}$.

Intuition: (MT1) says $f(n)$ is overwhelmed by $n^{\log_b a}$ small tasks like $T(1)$; (MT2) says $f(n)$ and small tasks have similar workload; (MT3) says most work is done by $f(n)$.

Int mult – Karatsuba algorithm: cut n-bit x into two $n/2$-bit x^+ and x^- with $x = 2^{n/2}x^+ + x^-$. Same for y. Then

$$x \cdot y = 2^n x^+ y^+ + 2^{n/2} \underbrace{(x^+ y^- + x^- y^+)}_{= x\cdot y - x^+ y^+ - x^- y^-} + x^- y^-$$

has complexity

$$T(n) = 3T(n/2) + \Theta(n)$$

which by (MT1) has $T(n) = \Theta(n^{\log_3 2})$.

Example: closest pair of points: given $\{p_i\}_{i=1}^n = \{(x_i, y_i)\}_{i=1}^n$, find the pair of closes points w.r.t. Euclidean norm.

Algorithm (high-level sketch):

```
1  // Works to be done before starting recursion:
2  // Sort all points by x-coordinates and store
       in an array
3  // Sort all points by y-coordinates and store
       in another array
4
5  Partition points into L and R based on median
       of x-coord
6  Two recursive calls on L and R subsets
7  // 2T(n/2) work
8
9  Let δ be the smaller return value from the two
       cursive calls
10 S = set of points within δ from the middle line
11 // O(n) or O(log n) passing indices
12
13 Check close pairs between L and R
14 - Index and sort points p(i) in S by y-coord
       y(i)
15     - For each point index i:
16     - start with j = i+1, stop when
           y(j) > y(i) + δ
17     - for each j, compute d(p(i), p(j))
18     - keep track of min d(p(i), p(j)), update δ
           when necessary
```

Remark. Given δ and i, the loop over j repeats at most 12 times due to the fact that points in L must be $\geqslant \delta$ apart and points in R must also be $\geqslant \delta$ apart. Then lines 14 to 18 takes $\mathcal{O}(n)$ time, giving

$$T(n) = 2T(n/2) + \mathcal{O}(n),$$

giving $\boxed{T(n) = \mathcal{O}(n \log n)}$ runtime to search for closest pair.

4. Dynamic Programming

DP in a nutshell: in order to do *this*, what do I need to do the previous step? DP is closely related to brute-force / backtracking but avoids unnecessary recomputation.

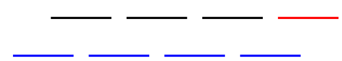
Example – Fibonacci: recursion blows up, but DP is $\mathcal{O}(n)$ using an additional array:

```
Fib[0] = Fib[1] = 1
for i in 2 : n+1
  Fib[i] = Fib[i-1] + Fib[i-2]
```

Example – interval selection w/ weights: given n intervals with start times $s(i)$, finish times $f(i)$, and additionally weights $w(i)$, find the set of non-overlapping intervals with maximal total weight.

Note: no greedy algorithm works. Consider the following example where every interval but the red one has weight 2.

- If red weight 3: optimal solution = pick first row
- If red weight 1: optimal solution = pick second row
- no way for greedy to decide which interval to pick first!



Key insight:

- If optimal solution includes interval i, then it does not include anything else intersecting i.
- If optimal solution does not include i, it is the same as the optimal solution for intervals $1, \ldots, i-1, i+1, \ldots, n$.
- Formula:

  $$\text{Opt}(j) = \max\{v_j + \text{Opt}(p(j)), \text{Opt}(j-1)\}$$

  where $p(j)$ is the largest $i < j$ such that intervals i, j are disjoint.

Memoization in one sentence: storing values for future recursive calls, like the array used in Fibonacci previously.

Algorithm using memoization ($\mathcal{O}(n)$):

```
M[0] = 0
For j = 1, 2, ..., n
  M[j] = max{v_j + M[p(j)], M[j-1]}
```

When to use DP?

- Only a polynomial number of subproblems.
- Solution to original problem can be easily computed from that to subproblems.
- Subproblems can be ordered from "smallest" to "largest" with easy recurrence.

Example – Subset Sums: given n jobs, each with w_i work time, and a cap W on total work time, maximize work time $\sum w_i$. Let $\mathcal{O}$ be optimal. Let $\text{Opt}(n, W)$ be the optimal value on first n jobs and w total time. For job i:

- If $n \notin \mathcal{O}$, $\text{Opt}(n, W) = \text{Opt}(n-1, W)$. (Makes no difference to $\mathcal{O}$ if we exclude n.)
- If $n \in \mathcal{O}$, $\text{Opt}(n, W) = w_n + \text{Opt}(n-1, W - w_n)$. (Do the job, set total time allowance to $W - w_n$, combined with the optimal solution on first $n-1$ jobs given $W - w_n$ time.)
- To sum up:

  $$\text{Opt}(i, w) = \max(\text{Opt}(i-1, w), w_i + \text{Opt}(i-1, w - w_i)). \quad (*)$$

Algorithm (tabular, $\mathcal{O}(nW)$):

```
// First initialize n × W matrix M

For i = 1, 2, ..., n
  For w = 0, 1, ..., W
    Compute M[i][w] := Opt(i, w) using (*)
```

Example – knapsacks: given n items, each with value v_i and weight w_i, and a cap W on total weight, maximize $\sum w_i$, the value of items taken. Let $\mathcal{O}$ be an optimal solution. Let $\text{Opt}(n, W)$ be the optimal value with first n items and total allowance W. For job i:

- If $n \notin \mathcal{O}$, $\text{Opt}(n, W) = \text{Opt}(n-1, W)$.
- If $n \in \mathcal{O}$, $\text{Opt}(n, W) = v_n + \text{Opt}(n-1, W - w_n)$.
- Combining:

  $$\text{Opt}(i, w) = \max(\text{Opt}(i-1, w), v_i + \text{Opt}(i-1, w - w_i)).$$

- A similar algorithm with an $n \times W$ array gives $\mathcal{O}(nW)$ runtime.