CSCE 629 Midterm Review Sheet

Math preliminaries, algorithm basics

✤ Arithmetic sum: $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$
✤ Geometric sum:
$\sum_{k=1}^{n} x^k = 1 + x + x^2 + ... + x^n = \frac{1-x^{n+1}}{1-x} = \frac{x^{n+1}-1}{x-1}$ when $x \neq 1$
When $x = 1$: trivial case. $\sum_{k=1}^{n} x^k = n$.
Evaluate at the infinity: $\sum_{k=1}^{\infty} x^k = \frac{1}{1-x}$ when $|x| < 1$
✤ Telescoping sum: Let a_n be a sequence of numbers. Then, $\sum_{k=1}^{n}(a_n - a_{n-1}) = a_n - a_0$.
† Partial fraction: able to break down fractions into sum of two different fractions, e.g. $\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1}$.
✞ Discrete derivative: Given $f(n)$, define $\Delta f(n) = f(n+1) - f(n)$.
Remember: $\Delta 2^n = 2^{n+1} - 2^n = 2^n$ (similar to e^n in continuous math)
Sidenote: define $n_{(m)} = n * (n-1) * ... * (n-m+1)$.
Then, $\Delta n_{(m)} = (n+1)_{(m)} - n_{(m)} = (n+1)n...(n-m+2)$ $-n(n-1)...(n-m+1) = m * n_{(m-1)}$
✞ Discrete integration: $\sum_{k=a}^{b} \Delta f(k) = \sum_{k=a}^{b}[f(k+1) - f(k)] = f(b+1) - f(a) = f(k)|_a^{b+1}$
✟ Finite version of chain rule:
$\Delta f(n)g(n) = \Delta f(n)g(n+1) + f(n)\Delta g(n)$
✠ Different asymptotic notations:

- $f(n) = o(g(n))$: $f(n)$ grows slower than $g(n)$. Growth rate $<$
- $f(n) = O(g(n))$: $f(n)$ grows at most as fast as $g(n)$. Growth rate $\leqslant$
- $f(n) = \theta(g(n))$: $f(n)$ grows the same as $g(n)$. Growth rate $=$
- $f(n) = \Omega(g(n))$: $f(n)$ grows at least as fast as $g(n)$. Growth rate $\geqslant$
- $f(n) = \omega(g(n))$: $f(n)$ grows faster than $g(n)$. Growth rate $>$

✠ Limit definition:

$$lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \text{case of small o} \\ c > 0 & \text{case of } \theta \\ \infty & \text{case of } \omega \end{cases} \tag{1}$$

c does not have to be 1; note that the case of satisfying both the small o and θ is indeed big O, and the case of satisfying both the θ and ω is indeed Ω.
✡ **Stirling's formula for n!**:
$n! = \sqrt{2\pi n}(\frac{n}{e})^n(1 + \theta(\frac{1}{n})) \approx (\frac{n}{e})^n$. Useful in limit comparison.

✡ **Master's theorem**: used to solve recurrence for divide-and-conquer cases.
Let $f(n) = af(\frac{n}{b}) + c * n^d$. Here, the divide-and-conquer algorithm subdivides into a subproblems, each of size n/b. Within each recursion, the time complexity is $O(n^d)$. $f(n)$ denotes time complexity on inputs of size n.

$$\therefore f(n) = \begin{cases} O(n^d) & a < b^d \\ O(n^d log_2 n) & a = b^d \\ O(n^{log_b a}) & a > b^d \end{cases} \tag{2}$$

✢ Sorting by comparison lower bound: Use a binary decision tree. Denote the height of the tree to be h. If the decision tree is a complete binary tree, then it would have **at most** 2^h **leaves**. When doing sorting, there would be $n!$ possibilities and they will appear as one of the leaves for **at least one time**. $\therefore n! \leqslant 2^h$
$\because n! \geqslant (\frac{n}{e})^n \to (\frac{n}{e})^n \leqslant 2^h$. That is, $h \geqslant log(\frac{n}{e})^n$
$\to h = \Omega(nlogn)$.
✢ Searching by comparison lower bound: similar approach as above - $h = \Omega(logn)$.
✢ Remark: it's usually hard to get a lower bound in theoretical computer science.

Divide-and-conquer

Multiple non-overlapping subproblems.
✣ Strassen's matrix multiplication: Assume that there are 2 matrices A and B, each of size $n * n$ (n is a power of 2).
Idea: subdivide each matrix into 4 equal parts. Use some tricks to do 7 multiplications instead of the normal 8. $\to$ 7 subproblems, each of size $\frac{n}{2}$. Within each recursion, the addition takes $O(n^2)$ time. ✓ time complexity becomes $O(n^{log_2 7})$ instead of $O(n^3)$!
✣ Polynomial multiplication using FFT: see notes for details. Straightforward approach takes $O(n^2)$ times to complete. With FFT, only need to evaluate half of the points within the original input polynomials. 2 subproblems ($A(x)$ and $B(x)$), each recursion takes $O(n)$ time to combine the multiplied terms $\to$ total runtime $=$ $O(nlogn)$ per Master's theorem.
In short: Given input polynomials $A(x)$ and $B(x)$, first performed FFT ($O(nlogn)$ time) to convert them into $A[x]$ and $B[x]$. Do the multiplication & combine terms in $O(n)$ time, then perform inverse FFT to convert the product of $A[x]$ and $B[x]$ back to $C(x)$ ($O(nlogn)$ time).

Greedy algorithm

Exploits the optimal substructure property with only 1 problem to solve.
✤ Spanning tree: a tree that connects all the vertices. It could be found using the greedy approach. (Greedy will find a spanning tree, not necessarily all of them)
⇒ Twist: Find a **minimum spanning tree** that has the minimum total edge weights. Same approach as above (see notes for details) - but! Sort the edges first. Time complexity $= O(|E||V|)$.
✤ Matroid: $M = (S, l)$ where S is a finite set, and l is a set that contains subsets of S.
Two properties - for two subsets A, B in S: 1) Hereditary property: If $B \in l$ and $A \subseteq B$, then $A \in l$; 2) Exchange property: if $A, B \in l$ and $|A| < |B|$, then there exists $x \in B - A$ s.t. $A \cup \{x\} \in l$.
Graphic matroid: Given an undirected graph $G = (V, E)$, let S be the set of edges in G. A set $A \in l$ is a set of edges that do not contain a cycle.
Maximum indep. subset of a matroid: if there is no bigger set $B \supsetneq A$ s.t. B is an indep. subset.
! Greedy could be used to find a weighted maximum independent subset of a matroid. Similar to finding the minimum spanning tree, sort the set S first. For each element in S, if adding it to the existing matroid set still belongs to l, then add it.
✤ Note that everything **minimum/maximum** is also **minimal/maximal**. -mum focuses on the absolute number, while -mal suggests that adding/removing elements from the current set would make the set automatically ineligible. (Consider the graph A — B — C. B is the minimum vertex cover. A,C is a minimal vertex cover. Remove either A or C, you are not left with a vertex cover.)

Dynamic programming

Flavor of divide-and-conquer and greedy: overlapping subproblem with the optimal substructure property.
✣ Longest common subsequence: use suffix. Notice that subsequence $\neq$ substring - could have gaps in between.
Let $l(i, j)$ be the length of the longest common subsequence of $a_1a_2...a_i$ and $b_1b_2...b_j$.
Recurrence:

$$l(i,j) = max \begin{cases} l(i-1, j-1) + 1 & \text{if } a_i = b_j \\ l(i, j-1) \\ l(i-1, j) \end{cases} \tag{3}$$

That is, the value of $l(i, j)$ is determined by its three neighbors: $l(i-1, j-1), l(i, j-1), l(i-1, j)$.
Base case: $l(i, 0) = l(0, j) = 0$. Whole approach takes $O(mn)$ time.

Best solution: $l(m,n)$ at the bottom right. To obtain the longest common subsequence: backtrack along the arrow. Whenever we go along a diagonal, add 1 more letter to the longest common subsequence.

✣ Global pairwise alignment: see notes for definition. Let $S(i,j)$ be the optimal alignment score of $a_1a_2...a_i$ of A and $b_1b_2...b_j$ of B.

Recurrence:

$$S(i,j) = max\begin{cases} S(i-1,j-1) + \Delta match \text{ if } a_i = b_j \\ S(i-1,j-1) + \Delta mismatch \text{ if } a_i \neq b_j \\ S(i,j-1) + \Delta indel \\ S(i-1,j) + \Delta indel \end{cases} \tag{4}$$

Similarly, the value of $S(i,j)$ is determined by its three neighbors: $S(i-1,j-1), S(i,j-1), S(i-1,j)$.

Base case: $S(i,0) = i * \Delta indel, S(0,j) = j * \Delta indel$.

Whole approach takes $O(|A||B|)$ time. Backtrack from the bottom right corner all the way back to $S(0,0)$ as well.

✣ Two variations:

- Local alignment: does not require all letters in string A and B to be in the alignment, but to identify a substring of A and a substring of B to have the highest alignment score.

  NOTE: same recurrence as above, but add 1 more condition - 0. That is because negative score does not help, and we would like to bound the score by the minimum of 0.

- Affine gap penalty: use 2 different types of gap penalties $\Delta Open\_gap$ and Δext_gap instead of a single $\Delta indel$.

  NOTE: reccurrence for this scenario is much more complicated, due to the need of distinguishing between 2 different kinds of scenarios (open gap or ext gap). 3 matrices would be used.

Comments: both variations could be solved using DP as well. Read notes for details.

✣ Matrix chain multiplication: Consider the multiplication of n matrices, $A_1A_2...A_n$. Notice that matrix multiplication is associative. The goal is to find an ordering that minimizes the total # of scalar multiplication between the matrices. **e.g.** $(A_1A_2)A_3$ and $A_1(A_2A_3)$ are 2 ways to multiply 3 matrices.

Note: if A_i is of dimension $P_{i-1} * P_i$ and A_{i+1} is of dimension $P_i * P_{i+1}$, number of scalar multiplication needed in A_iA_{i+1} is $P_{i-1}P_iP_{i+1}$.

Let $m(i,j)$ be the minimum number of scalar multiplication needed in $A_i...A_j$.

Recurrence:

$$m(i,j) = \begin{cases} 0 \\ min_{i \leqslant k \leqslant j}(m(i,k) + m(k+1,j) + \\ P_{i-1}P_kP_j) & \text{if } i < j \end{cases} \tag{5}$$

Time complexity: there are quadratic $O(n^2)$ numbers of $m(i,j)$'s. To get each $m(i,j)$ takes $O(n)$ time. ⇒ Overall time complexity $= O(n^3)$

Amortized analysis

✦ STACK problem: Consider a stack that has a LIFO (last in first out) structure. There are two alllowed operations: PUSH puts an element at the top of the stack, while MULTIPOP pops all objects from the stack. Analyze the time complexity for a sequence of n PUSH and MULTIPOP operations.

Straightforward approach: time complexity $= O(n^2)$ due to each MULTIPOP taking $O(n)$ time, and n operations in total. However, this is not the best bound, as PUSH could bring up to n elements in the stack and MULTIPOP can pop at most n of them.

Potential method: think of each PUSH operation as a credit and each POP operation as a way to use up that credit.

Formally, define $\phi(D_i)$ be the number of elements in the stack after the ith operation. Let C_i be the actual cost of the ith operation. Let $\hat{C}_i$ be the amortized cost of the ith operation. Then: $\hat{C}_i = C_i + [\phi(D_i) - \phi(D_{i-1})]$.

✓ Amortized time complexity is an upper bound of the actual time complexity, provided that the potential in the end is at least as large as the potential at the beginning.

⇒ Amortized time complexity of PUSH: actual cost + change in potential = 1 + 1 = 2

Amortized time complexity of MULTIPOP: actual cost + change in potential = $k + (-k) = 0$

⇒ Amortized time complexity of n PUSH and MULTIPOP operations: $n * O(2+0) \approx O(n)$.

✦ UNION-FIND problem: see notes. (Two optimization techniques: **Union by rank** (always attaching smaller depth tree under the root of the deeper tree) and **path compression** (to flatten the tree when find() is called).) With these two improvements, and a proper data structure design (**tree** instead of array/linkedlist), the amortized time complexity could be improved to $O(nG(n))$, where $G(n)$ is the smallest integer k such that $F(k) \geqslant n$, and the function F is defined as:

$$F(n) = \begin{cases} 0 & \text{if } k = 0 \\ 2^{F(k-1)} & \text{if } k > 0 \end{cases} \tag{6}$$

Proof: see notes. (Rather complicated)