CMPS 6610 Extra Credit Answers

In this extra credit assignment, we will test and review concepts you have learned since the midterm exam. Please add your written answers to `answers.md` which you can convert to a PDF using `convert.sh`. Alternatively, you may scan and upload written answers to a file named `answers.pdf`.

1. **Algorithmic Paradigms**

- My favorite algorithmic paradigm is Dynamic Programming (DP). I think intuitively it is very interesting and elegant. More specifically, it is fascinating because it allows us to solve problems that initially appear to require exponential time (like $O(2^n)$) by identifying overlapping subproblems (for example, the Fibonacci sequence). Through memoization by simply caching (building a hash table) the results of these subproblems (top-down approach) or building them iteratively (tabulation) through a bottom-up approach, we can reduce the complexity to polynomial time (often $O(n^2)$ or $O(n^3)$). DP effectively illustrates the trade-off between space and time.

2. **Divide and Conquer**

- Yes, problems that can be solved by a divide and conquer approach necessarily satisfy the optimal substructure property.
- **Proof**:
  - We will prove this by contradiction.
  - Let P represent a problem that can be solved by a Divide and Conquer Algorithm. The algorithm works by dividing P into distinct subproblems $P_1, P_2, ..., P_k$ solving them to obtain solutions $S_1, S_2, ..., S_k$ and then combining these solutions into an integrated global solution S.
  - Optimal substructure is defined as the property whereby an optimal solution to the problem contains optimal solutions to its subproblems within it.
  - We will assume for the sake of contradiction that the optimal global solution S is constructed from a sub-solution S_i that is not optimal for subproblem P_i. This implies that there exists a different solution S_i' to P_i that is "better" than S_i.
  - The combination of the step is monotonic in that improving a component improves or maintains the quality of the whole. It follows that we could replace S_i with S_i' to generate a new global solution S'. Since S_i' is better than S_i, S' would better than S. However, this contradicts the assumption that S was the optimal solution. Therefore, the optimal solution S must be composed of optimal sub-solutions S_i and problems that can be solved by a divide and conquer approach necessarily satisfy the optimal substructure property.

3. **Randomization**

- 3a.
- Let X be the random variable for the number of comparisons. We know $E[X] \approx Cn \ln n$ for some constant C. We are asking for the probability that $X \geq kn^2$ for some constant k.Using Markov's Inequality:

$$P[X \geq kn^2] \leq \frac{E[X]}{kn^2} = \frac{Cn \ln n}{kn^2} = \frac{C \ln n}{kn}$$

$$\lim_{n \to \infty} \frac{C \ln n}{kn}$$

- Apply L'Hopital's Rule: Take the derivative with respect to n of the numerator and the denominator separately. - Numerator: $\frac{d}{dn}(C \ln n) = \frac{C}{n}$ - Denominator: $\frac{d}{dn}(kn) = k$ - Evaluate the New Limit:

$$\lim_{n \to \infty} \frac{\frac{C}{n}}{k} = \lim_{n \to \infty} \frac{C}{kn}$$

Hence, as $n \to \infty$, the term $\frac{C}{kn}$ clearly goes to 0.

As $n \to \infty$, this probability approaches 0. The probability is bounded by $O(\frac{\log n}{n})$. - 3b. We will calculate the probability that Quicksort does $10^c n \ln n$ comparisons, for a given $c > 0$. 1. First, we will apply Markov's

Inequality. We are given: - Random Variable X: The number of comparisons Quicksort makes. - Expected Value $E[X]$: We know Quicksort's expected work is $E[X] \approx Kn \ln n$ (where K is a constant, typically around 2 depending on the base of the logarithm). - Threshold α: $10^c n \ln n$

- As for 3a, Markov's Inequality states:

$$P(X \geq \alpha) \leq \frac{E[X]}{\alpha}$$

- Substitute our values:

$$P(X \geq 10^c n \ln n) \leq \frac{Kn \ln n}{10^c n \ln n}$$

2. We can then simplify and the $n \ln n$ terms cancel out:

$$P(X \geq 10^c n \ln n) \leq \frac{K}{10^c}$$

(Using the standard upper bound $E[X] \leq 2n \ln n$ - from the summation of the Harmonic series, this would be $\frac{2}{10^c}$ as we discussed in class in Jupyter book 3 of the randomized algorithms module).

- This result has implications for the "concentration" of the expected work for Quicksort in terms of the tail distribution. This suggests that there is a rapid decay as the parameter c increases. The probability of the runtime reaching $10^c n \ln n$ decreases exponentially (specifically as 10^{-c} from the term $\frac{1}{10^c}$). Hence, it is extremely unlikely for Quicksort to take a very long time. For example, if $c = 2$, (checking $100nlnn$ comparisons), the probability is bounded by $\approx \frac{2}{100} = 0.02$.

4. **Greedy Algorithms**

- We will prove that scheduling jobs in order of their processing times (i.e., shortest-job-first) results in an optimal schedule where the cost is the average waiting time.
- The core logic for the proof will be based upon an "Exchange Argument"
  - Assume there is an optimal solution (O) that does not make the greedy choice.
  - Identify the first place where O differs from the greedy choice.
  - Modify (Exchange) the optimal solution by swapping the non-greedy choice with the greedy choice.
  - Prove that this new solution is valid and no worse than the original optimal solution (or strictly better, which contradicts the optimality of the original).
- **Proof**: Claim: Scheduling jobs in increasing order of processing time (Shortest-Job-First) minimizes the average waiting time.

1. Optimal Substructure: The problem exhibits optimal substructure. If we determine that the shortest job j_{min} must be scheduled first, the problem reduces to finding the optimal schedule for the remaining $n-1$ jobs. The relative ordering of the remaining jobs to minimize their own waiting times is independent of the first job's processing time (which adds a constant delay to all of them). Thus, we can solve the problem by choosing the first job and recursively solving for the rest.
2. Greedy Choice Property (Proof by Exchange Argument): We will prove that the greedy choice (shortest job first) must be part of an optimal solution.
   - Assumption: Let S be an optimal schedule that minimizes the total waiting time. Assume for the sake of contradiction that S is not sorted by processing time.
   - Identifying an Inversion: If S is not sorted, there must exist at least one pair of adjacent jobs, say job A at position i and job B at position $i + 1$, such that $p_A > p_B$. (i.e., a longer job comes immediately before a shorter job).
   - The Exchange: Consider a new schedule S' formed by swapping job A and job B.Jobs before i: Unaffected.Jobs after $i + 1$: Their waiting time depends on the sum of processing times of all previous jobs. Since $p_A + p_B = p_B + p_A$, the total time elapsed before job $i + 2$ starts remains exactly the same. Thus, jobs after the swap are unaffected.Job B (now at i): Waiting time decreases. It no longer waits for A.Job A (now at $i + 1$): Waiting time increases. It now waits for B.Cost Analysis: We calculate the change in total cost (waiting time) caused by this swap:

$$Cost(S) = \cdots + W_A + W_B + \ldots$$

$$Cost(S') = \dots + W'_B + W'_A + \dots$$

The waiting time for the job at position i is determined by the jobs before it (let's call this sum T). In S: Job A waits T. Job B waits $T + p_A$. Contribution to total: $T + (T + p_A) = 2T + p_A$.In S': Job B waits T. Job A waits $T + p_B$. Contribution to total: $T + (T + p_B) = 2T + p_B$. The change in cost is:

$$\Delta Cost = Cost(S') - Cost(S) = (2T + p_B) - (2T + p_A) = p_B - p_A$$

- Contradiction: Since we identified that $p_A > p_B$, it follows that $\Delta Cost < 0$.This implies $Cost(S') < Cost(S)$, meaning S' is strictly better than S.
- Conclusion: This contradicts our assumption that S was the optimal schedule. Therefore, an optimal schedule cannot contain any adjacent inversions. The only schedule with no adjacent inversions is the one sorted by processing time (Shortest-Job-First). Thus, the greedy strategy is optimal.

3. **Dynamic Programming**

- **Maximum Span (No Parallelism):**
- Recurrence: $T(n) = T(n-1) + O(1)$
- Example: Calculating the n-th number in a sequence where each number depends strictly on the previous one, such as a naive iterative Fibonacci calculation ($F_i = F_{i-1} + F_{i-2}$) or finding the Longest Common Subsequence using the standard diagonal dependency.
- Span: $O(n)$. The dependency graph is a simple chain (line), so you cannot compute the n-th step until the $(n-1)$-th is finished
- **Polylogarithmic Span (Ideal Parallelism):**
- Recurrence: $T(n) = T(n/2) + O(1)$ (in terms of depth)
- Example: A Divide and Conquer approach to DP, such as summing a list of numbers using a tree structure (Reduce/Tree Contraction) or parallel prefix sum.
- Span: $O(\log n)$. The dependency graph is a tree with depth $\log n$, allowing many branches to be computed simultaneously.

6. **Graphs**

- The *cycle property* states that, "Given a graph **G** that contains a cycle, then the largest weight **edge** in that graph cannot be contained in any minimum spanning tree.
- **Proof**: We will prove this by contradiction.

1. We will first assume the opposite; namely, there exists an MST T that contains edge $e = (u, v)$, where e is the max-weight edge in cycle C. If we remove e from T, the tree disconnects into two components, T_1 and T_2.
2. As e was part of a cycle C in the original graph, there must be at least one other edge $f = (x, y)$ in C that connects a node in T_1 to a node in T_2 (crossing the cut created by removing e).
3. We had denoted that e was the maximum weight edge in C and hence it follows that $w(f) < w(e)$.
4. We can construct a new tree, $T' = (T \setminus \{e\}) \cup \{f\}$. The total weight of T' is $weight(T) - w(e) + w(f)$.
5. Since $w(f) < w(e)$, the $weight(T') < weight(T)$. This **contradicts** the assumption that T was a MST. Therefore, e cannot be in the MST.