CPSC 3120 Final Exam Study Guide

Brendan McGuire (bmmcgui@clemson.edu) • Patrick Smathers (psmathe@clemson.edu)

Greedy Algorithms
In a greedy algorithm, we are faced with making decisions to optimize an objective function. In a greedy approach, we select the *locally optimal* choice. The challenge with greedy approaches is proving that a locally optimal choice will lead to a globally optimal solution.

Consider the examples below for situations where a greedy algorithm is the most optimal solution to the problem.

Fractional Knapsack Problem
In the fractional knapsack problem, we are given a set of items with *weight* and *benefit*. We must maximize the total benefit given a specific weight limit. We can take *fractions* of each item, getting the corresponding weight for the percentage we took. For the fractional knapsack problem, we know that *greedily* selecting items with the highest benefit per unit weight has been shown to produce an optimal solution.

Example
Your candy bag can hold up to 10 kg of candy. You go to the store and see the following candies available for sale with your rated level of tastiness. Maximize the total *tastiness value* of your bag, assuming you can take fractional amounts of each candy.

Candy	**Weight Available**	**Total Tastiness**
M&Ms	*2 kg*	10
Acorns	*10 kg*	3
Paper	*40 kg*	1
Twix	*1kg*	12

First, we can compute the *tastiness per unit weight* of each candy category and sort from highest to lowest unit tastiness.

Candy	**Weight Available**	**Total Tastiness**	**Unit Tastiness**
Twix	*1kg*	12	$12 / 1 = 12$ tastiness per kg
M&Ms	*2 kg*	10	$10 / 2 = 5$ tastiness per kg
Acorns	*10 kg*	3	$3 / 10 = 0.3$ tastiness per kg
Paper	*40 kg*	1	$1 / 40 = 0.025$ tastiness per kg

From this, we can fill our bag with the candy that has the highest tastiness per unit weight until our bag fills completely.

Step	Current Bag	Action	New Bag
0	W = 0, T = 0	Take 1 kg of Twix	W = 1, T = 0 + 12 * 1 = 12
1	W = 1, T = 12	Take 2kg of M&Ms	W = 3, T = 12 + (2 * 5) = 22
2	W = 3, T = 22	Take 7 kg of Acorns	W = 10, T = 22 + (0.3 * 7) = 24.1

Therefore, the optimal way of filling our bag is **taking 1 kg of Twix, 2 kg of M&Ms, and 7 kg of Acorns**.

Huffman Coding

Task Scheduling
In a task scheduling problem, we have N tasks, each with a start time S_i and end time F_i where a machine must complete each task. We must minimize the number of machines needed to complete all tasks in their allotted time. A greedy approach to scheduling has been shown to be optimal.

Thinking About Greedy Approaches
The difficulty of greedy algorithms to problems is not their implementation, as they are usually fairly straightforward to produce. Instead, the challenge focuses on showing that a greedy approach is an optimal solution.

Divide And Conquer

This technique divides a particular computing problem into one or more subproblems, each smaller. It recursively solves each problem before "merging" or "marrying" the subproblem solutions to the next level up's solutions. Once all the subproblems are solved and merged upwards, you end with a solution to the original problem. A good example of a divide and conquer algorithm is Merge Sort, which will be described below.

Recurrence Equation & The Master Theorem

The recurrence equation shows the complexity of a recursive algorithm based on the given input.

$$T(n) = \left\{ \begin{array}{l} c, n <= 2 \\ aT\frac{n}{b} + f(n), n > 2 \end{array} \right\}$$

a - How many subproblems are we creating?

b - What is the size of each problem?

f(n) - The cost of additional steps

The Master Theorem gives us a way to get asymptotic analysis based on three *if-then* conditions:

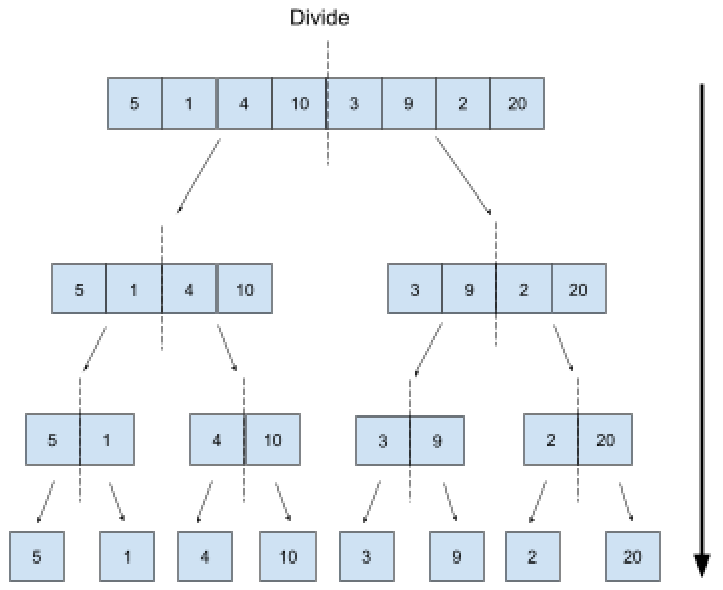
1. If there is a small constant $\epsilon > 0$, such that $f(n)$ is $O(n^{log_b a-\epsilon})$, then $T(n)$ is $\theta(n^{log_b a})$
2. If there is a constant $k \geq 0$, such that $f(n)$ is $\theta(n^{log_b a} log^k n)$, then $T(n)$ is $\theta(n^{log_b a} log^{k+1} n)$
3. If there are small constants, $\epsilon > 0$, and $\delta < 1$, such that $f(n)$ is $\Omega(n^{log_b a + \epsilon})$ and $af(\frac{n}{b}) \leq \delta f(n)$, for $n \geq d$, then $T(n)$ is $\Theta(f(n))$.

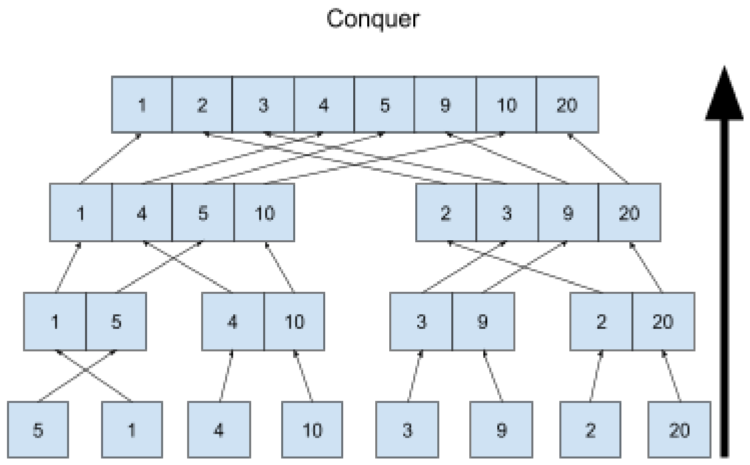
The main term to focus on is $n^{log_b a}$, which is found in all three equations. *b* and *a* are from the Recurrence Equation above. *f(n)* in the Master Theorem's rules are the cost of additional steps from the Recurrent Equation

Merge Sort

Merge sort is one of the most efficient sorting algorithms and runs in O(N log N). It divides and conquers the given array to sort by splitting it into left and right halves recursively until there is just one item in each array. Then it merges the sublists to get a fully sorted list.

Dividing will subdivide but keep the current order as it divides into smaller subproblems. When the subproblems are at their smallest, you start merging the arrays, but each element is put into the correct order during the merge





Closest Pair of Points

Dynamic Programming
We can utilize dynamic programming to solve problems easily divided into smaller dependent subproblems. Fundamentally, dynamic programming is an optimization that can be applied to recursive solutions.

This is different from Divide & Conquer because *we use the previous solutions* to aid us in solving the larger scale problem. Consider the example of computing the *n-th* Fibonacci number. A simple recursive solution to this problem has been shown below.

```
int fibonacci(int n) {
  if (n == 0) {
    return 0;
  }
  if (n == 1) {
    return 1;
  }
  return fibonacci(n - 1) + fibonacci(n - 2);
};
```

See how the process of computing the *ith* Fibonacci number can be trivially completed if you know the *i-1th* and *i-2nd* number (smaller subproblems). This is the perfect candidate for dynamic programming.

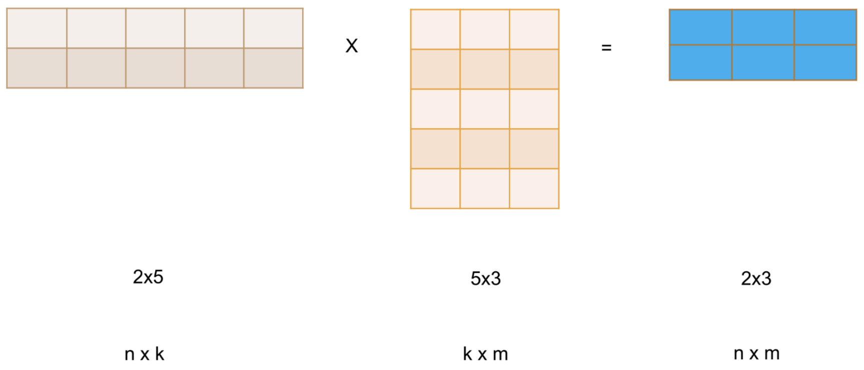
```
int fibonacci(int n) {
  std::vector<int> fib(n + 1);

  fib[0] = 0;
  fib[1] = 1;

  for (int i = 2; i <= n; i++) {
    fib[i] = fib[i - 1] + fib[i - 2];
  }

  return fib[n];
}
```

By utilizing this memoization (storing the result of smaller subproblems), we can dramatically reduce the total runtime. For example, if we were computing all of the fibonacci from 1 to n, a dynamic programming approach would allow us to compute it in O(N) time. In contrast, a more naive implementation might take O(N^2).

Matrix Chain Multiplication

Matrix Multiplication is associative, so we can multiply a series of matrices in any order and get the same result. However, the order in which you multiply can take vastly different *numbers of steps*. Compute the minimum number of steps required to multiply a series of matrices and what order you must multiply them by to achieve that number of steps.

Review of Matrix Multiplication



To multiply two matrices together, they must have the dimensions N x K and K x M. The resulting matrix is in the form of N x M. The number of steps required to multiply those matrices is n x k x m.

For example, let A be a 2x10 matrix, B be a 10x50 matrix, and C be a 50x20 matrix. To compute $A \times B \times C$, we can either do $(A \times B) \times C$ or $A \times (B \times C)$. However, these do not take the same amount of steps.

Form	**Steps Required**
$(A \times B) \times C$	$(A \times B)$ means2 * 10 * 50 = 1000 steps (produces a 2x50 matrix) $AB \times C$ means $2 * 50 * 20$ steps = 2000 steps **3000 steps total**
$A \times (B \times C)$	$(B \times C)$ means 10 * 50 * 20 steps = 10,000 steps (produces at 10 * 20 matrix) $A \times BC$ means 2 * 10 * 20 = 400 steps **10,400 steps total**

If we have to multiply $A \times B \times C \times D$ how do we know what the most efficient order of multiplication is?

➔ The naive solution is to try every possible combination, but the number of combinations grows exponentially

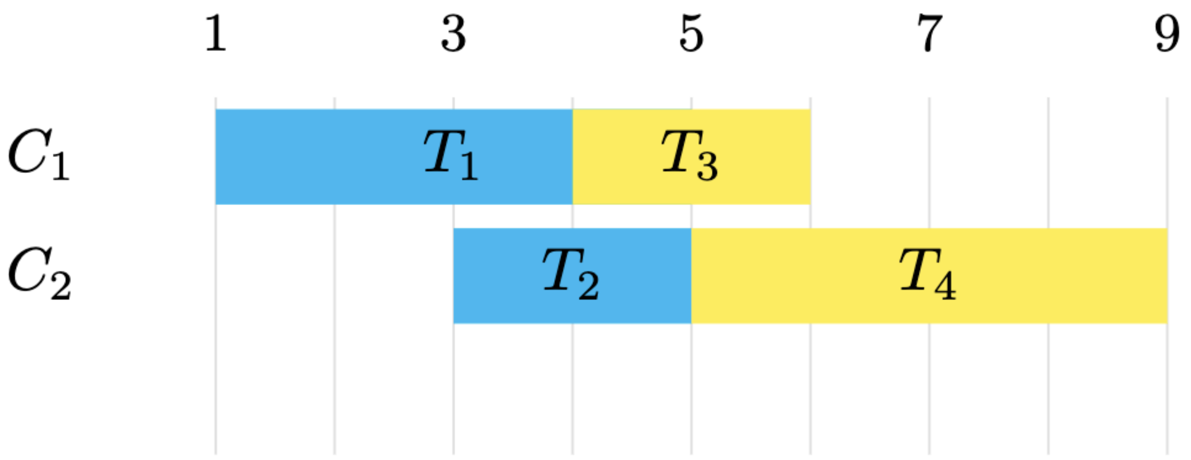
➔ However, we can apply our dynamic programming principles to simplify the computation process vastly. This is because the minimum number of steps to multiply $A \times B \times C \times D$ is just the minimum number of steps to multiply $A \times B \times C \times D$

Longest Common Subsequence

Task Scheduling

The best way to do task scheduling is taking the greedy approach of assigning the first open slot found.

Given the following tasks and times, find the optimum scheduling to use the least number of machines: (1,4), (3, 5), (4, 6), (5, 9)



Graphs & Graph Algorithms

<u>*Representations*</u>

<u>*Shortest Path Problems*</u>

<u>*Minimum Spanning Tree*</u>

Kruskal's Algorithm for MST

Classes of Complexity

Until now we have mostly considered the *solutions* to problems. But not all problems are created equal. Classifying problems into their complexity can help us understand them and create equivalences between them. Additionally, if we know a problem as difficult as another, we know we won't be able to devise an efficient solution.

P vs. NP

Typically, we define a problem as "efficiently solvable" if you can devise an algorithm that runs in $O(n^k)$ time for some constant $k > 0$. This is known as *polynomial time*.

There are two aspects of the algorithm that we care about when classifying problems:

- ➔ Can we *solve* the problem in polynomial time?
- ➔ Can we *verify* a solution in polynomial time? If given an input and an output, can we verify that the output satisfies the problem in polynomial time

NP-Complete

We use NP-Complete to refer to classes of problems that cannot be solved in polynomial time but can be verified in polynomial time. All NP-Complete problems are connected, and you can transform anyone into another, meaning if we ever find a polynomial solution to one, we have solved the entire set.

NP-Hard

These problems are so dastardly that not only can they not be *solved* in polynomial time, they can't even be *verified* (you can't check that an output is a correct solution for an input) polynomial time.

Most of the time, finding suboptimal solutions for these problems is the best approach. Techniques like iterative refinement, genetic, and greedy approaches often present solutions close enough to optimal and significantly reduced complexity.