Zach Forster
CS 335 Final

1) Putting off their differences for a while, there are a few key similarities between Divide-and-Conquer algorithms and Dynamic Programming algorithms. Both are algorithms which can solve applicable problems in a reasonable amount of time (not problems of exponential or factorial complexity). More importantly, the types of problems that lend themselves to these two techniques are very similar. Both require that a problem lends itself to division into sub-problems of the same type and that a solution to the problem can be constructed from solutions to those sub-problems. In other words, the problem must have a recursive definition for these algorithms to work. However, Dynamic Programming sub-problems often have considerable overlap in their solutions, while Divide-and-Conquer problems should not have any. Dynamic Programming algorithms start with the solutions to the base sub-problems and build up to the full solution, while Divide-and-Conquer algorithms start with the full problem, working their way down to where the Dynamic Programming algorithms start through recursion before building back up. In the end, the second half of the Divide-and-Conquer algorithms exactly match the entirety of the Dynamic Programming algorithms. One can even mirror a Dynamic Programming algorithm's effect by combining a Divide-and-Conquer algorithm with a hash table of solutions found. Because of the extra memory consumed by Dynamic Programming algorithms, they should only be used when memory is abundant. In light of these facts, one should always use Dynamic Programming when the recursive definition of a problem tends to explode, generating a large number of duplicated solutions. Dynamic Programming algorithms inherently avoid repetition, while Divide-and-Conquer algorithms require additional data structures and modification to achieve this goal.

2) The Best-first search algorithm shares a special relationship with the A* algorithm, mostly due to the fact that A* is a specific type of Best-first search algorithm. Best-first search finds a good solution quickly, as opposed to the best solution slowly or a random solution almost instantaneously. Then, since it is a backtracking algorithm, it also expands all other solutions with cost estimates as good as or better than the first solution. A* is an implementation of an informed Best-first search, which means that A* takes into account the distance traveled, as well as the estimated distance-to-solution when choosing a path to expand. This is the opposite of a Greedy Best-first search, which makes decisions locally about which path to expand next, without any prior information. Often times (and I'm assuming that this is the case here by the way the question is posed), the Greedy Best-first search algorithm is simply referred to as “the Best-first search.” Both the Greedy Best-first search and A* algorithms are backtracking algorithms, and are therefore designed to solve problems with exponential or factorial complexity. Although Greedy algorithms can be used as a single stage (the Best-first search) inside backtracking algorithms, they also comprise an entire class of independent algorithms. A Greedy algorithm finds a globally optimal solution by making the locally optimal choice at each stage, often by using some kind of estimation or heuristic. For the

Greedy Best-first search, this estimation is typically made with some sort of static evaluation of the problem at a given stage. The A* algorithm does not use a Greedy Best-first search, simply because of its use of information from previous stages to make such a static evaluation. A Greedy algorithm only yields the best solution when a problem space contains no local extremes which aren't also global extremes. The reason is that Greedy algorithms can be seen as backtracking algorithms where the initial Best-first search used for pruning always results in the optimal solution. In other words, a Greedy algorithm proceeds in exactly the same way as a Backtracking algorithm, right up until the Backtracking algorithm actually backtracks. Then the Backtracking algorithm must be able to retreat to an earlier stage (backtrack), and repeat the (sometimes) Greedy process. This is also why Greedy algorithms tend to have a linear complexity, while backtracking algorithms do not. Therefore, Greedy algorithms are best used for problems in which the intermediate solutions are structured in either an ascending or descending (by some metric) order (i.e. when the next solution always has a higher weight than the current one). This eliminates the need for backtracking.

3) Dynamic Programming algorithms are usually the best choices for parallelizable problems. Greedy algorithms are extremely simple and extremely sequential in nature. In almost every case, every local decision to add a solution to the set depends on the previous solutions which were added. Recursive Divide-and-conquer algorithms actually lend themselves quite well to parallelization. Each recursive call can simply be replaced with a fork, and a join must be added at the end of the function. Dynamic Programming algorithms are also easily parallelized. They tend to result in the construction of large data tables where each value depends upon only a few others. Such arrays (the more dimensions, the better) are very easily parallelized on a massive scale with some semaphores and blocking operations. Both Divide-and-Conquer and Dynamic Programming algorithms have a sort of fan-out, fan-in pattern, during which the amount of parallelism grows and shrinks. Therefore, it would seem to me that both Dynamic Programming and Divide-and-Conquer algorithms are easily parallelized. However, the combination of threads and recursion can sometimes be difficult to manage. In addition, the parallelism can sometimes be imbalanced when the original problem space has an odd number of elements. Therefore, for the sake of choosing one, I would say that Dynamic Programming algorithms are the most straightforward to parallelize.

4) One well-known parallel algorithm is the parallel prefix sum, or scan algorithm. Like map and reduce, scan is an algorithm seen most often in functional programming languages, like Haskell. Scan takes a binary associative operator, an identity function, and an array. The operator is typically addition, but it could be any group operator, including multiplication or function composition (although that might be complicated to implement). The return value of the function is a new array, where each element is the sum of every element occurring before its corresponding index in the original array. For instance, the scan of [1, 2, 3 ,4 ,5] with addition as its binary operator is [0, 1, 3, 6, 10]. The parallel algorithm works by computing the sums of every pair in the array whose first item has an even index. If the original array was [x0, x1, x2, … xn], then these sums are z0 = x0 + x1, z1 = x2 + x3, etc. We can recursively compute the prefix

sum of the sequence [z0, z1, z2, ...]. This new sequence will be [w0, w1, w2, …]. Every element in the final prefix sum is either exactly equal to the value at half of its index in the w sequence or the sum of the previous value and something in the x sequence. Clearly, the parallelism in the algorithm occurs mainly in the recursive steps, indicating that the expanding and contracting parallelism pattern common in divide and conquer algorithms exists here as well. This algorithm can be applied to sorting algorithms such as radix sort, binary adders, and the construction of histograms, among other things. It was made for parallel systems, although because it seems to be more of a low-latency problem than a high-throughput problem, it may not be quite as well suited for distributed computing.