CS787: Advanced Algorithms

Scribe: David Hinkemeyer and Dalibor Zelený **Lecturer:** Shuchi Chawla
Topic: Greedy Algorithms, Divide and Conquer, and DP **Date:** September 7, 2007

Today we conclude the discussion of greedy algorithms by showing that certain greedy algorithms do not give an optimum solution. We use set cover as an example. We argue that a particular greedy approach to set cover yields a good approximate solution. Afterwards, we discuss the divide and conquer technique of algorithm design. We give two examples of divide and conquer algorithms, and discuss lower bounds on the time complexity of sorting. Finally, we introduce dynamic programming using the problem of weighted interval scheduling as an example.

2.1 Greedy Set Cover

Previously, we had seen instances where utilizing a greedy algorithm results in the optimal solution. However, in many instances this may not be the case. Here we examine an example problem in which the greedy algorithm does not result in the optimal solution and compare the size of the solution set found by the greedy algorithm relative to the optimal solution.

The Set Cover Problem provides us with an example in which a greedy algorithm may not result in an optimal solution. Recall that a greedy algorithm is one that makes the "best" choice at each stage. We saw in the previous lecture that a choice that looks best locally does not need to be the best choice globally. The following is an example of just such a case.

2.1.1 Set Cover Problem

In the set cover problem, we are given a universe U, such that $|U| = n$, and sets $S_1, \ldots, S_k \subseteq U$. A *set cover* is a collection C of some of the sets from $S_1, \ldots, S_k$ whose union is the entire universe U. Formally, C is a set cover if $\bigcup_{S_i \in C} S_i = U$. We would like to minimize $|C|$.

Suppose we adopt the following greedy approach: In each step, choose the set S_i containing the most uncovered points. Repeat until all points are covered.

Suppose $S_1 = \{1, 2, 3, 8, 9, 10\}$, $S_2 = \{1, 2, 3, 4, 5\}$, $S_3 = \{4, 5, 7\}$, $S_4 = \{5, 6, 7\}$, and $S_5 = \{6, 7, 8, 9, 10\}$. Figure 2.1.1 depicts this case. We wish to see how close a greedy algorithm comes to the optimal result. We can see that the optimal solution has size 2 because the union of sets S_2 and S_5 contains all points.

Our Greedy algorithm selects the largest set, $S_1 = \{1, 2, 3, 8, 9, 10\}$. Excluding the points from the sets already selected, we are left with the sets $\{4, 5\}$, $\{4, 5, 7\}$, $\{5, 6, 7\}$, and $\{6, 7\}$. At best, we must select two of these remaining sets for their union to encompass all possible points, resulting in a total of 3 sets. The greedy algorithm could now pick the set $\{4, 5, 7\}$, followed by the set $\{6\}$.

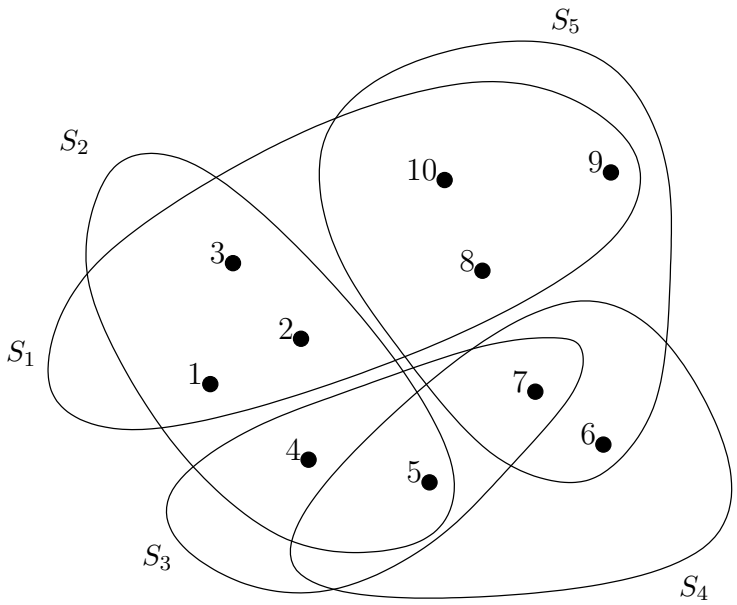

Figure 2.1.1: An instance of a set cover problem.

2.1.2 Upper bound on Greedy Set Cover Problem

In the previous example we saw a case where the greedy algorithm did not produce the optimal solution. In the following theorem we show that size of the set cover found by the greedy algorithm is bounded above by a function of the size of the optimal solution and the number of elements in the universe U.

Theorem 2.1.1 *Suppose an optimal solution contained m sets. Our greedy algorithm finds a set cover with at most $m \log_e n$ sets.*

Proof: Let the universe U contain n points, and suppose that the optimal solution has size m. The first set picked by the greedy algorithm has size at least n/m. Therefore, the number of elements of U we still have to cover after the first set is picked is

$$n_1 \leq n - n/m = n(1 - 1/m).$$

Now we are left with n_1 elements that we have to cover. At least one of the remaining sets S_i must contain at least $n_1/(m-1)$ of such points them because otherwise the optimum solution would have to contain more than m sets. After our greedy algorithm picks the set that contains the largest number of uncovered points, it is left with $n_2 \leq n_1 - n_1/(m-1)$ uncovered nodes. Notice that $n_2 \leq n_1(1 - 1/(m-1)) \leq n_1(1 - 1/m) \leq n(1 - 1/m)^2$. In general, we then have

$$n_{i+1} \leq n_i(1 - 1/m) \leq n(1 - 1/m)^{i+1}. \tag{2.1.1}$$

We would like to determine the number of stages after which our greedy algorithm will have covered all elements of U. This will correspond to the maximum number of sets the greedy algorithm has

to pick in order to cover the entire universe U. Suppose that it takes k stages to cover U. By 2.1.1, we have $n_k \leq n(1-1/m)^k$, and we need this to be less than one.

$$\begin{aligned} n(1-1/m)^k &< 1 \\ n(1-1/m)^{m\frac{k}{m}} &< 1 \\ (1-1/m)^{m\frac{k}{m}} &< 1/n \\ e^{-\frac{k}{m}} &< 1/n \ldots \text{using relation} (1-x)^{\frac{1}{x}} \approx 1/e \\ k/m &> \log_e n \\ k &< m \log_e n \end{aligned}$$

From this we can see that the size of the set cover picked by our greedy algorithm is bounded above by $m \log_e n$. ■

We have just shown that the greedy algorithm gives a $\mathcal{O}(\log_e n)$ approximation to optimal solution of the set cover problem. In fact, no polynomial time algorithm can give a better approximation unless P = NP.

2.2 Divide and Conquer

Whenever we use the divide and conquer method to design an algorithm, we seek to formulate a problem in terms of smaller versions of itself, until the smaller versions are trivial. Therefore, divide and conquer algorithms have a recursive nature. The analysis of the time complexity of these algorithms usually consists of deriving a recurrence relation for the time complexity and then solving it.

We give two examples of algorithms devised using the divide and conquer technique, and analyze their running times. Both algorithms deal with unordered lists.

Sorting We describe mergsort—an efficient recursive sorting algorithm. We also argue that the time complexity of this algorithm, $\mathcal{O}(n \log n)$, is the best possible worst case running time a sorting algorithm can have.

Finding the k-th largest element Given an unordered list, we describe how to find the k-th largest element in that list in linear time.

2.2.1 Mergesort

The goal of sorting is to turn an unordered list into an ordered list. Mergesort is a recursive sorting algorithm that sorts an unordered list. It consists of two steps, none of which are done if the length of the list, n, is equal to one because sorting a list with one element is trivial.

1. *Split* the list in two halves that differ in length by at most one, and sort each half recursively.

2. *Merge* the two sorted lists back together into one big sorted list. We maintain pointers into each of the two sorted lists. At the beginning, the pointers point to the first elements of the lists. In each step, we compare the two list elements being pointed to. We make the smaller one be the next element of the big sorted list, and advance the pointer that points to that element (i.e. we make it point to the next element of the list). We continue in this fashion until both pointers are past the ends of the lists which they point into. The process is illustrated in Figure 2.2.2.

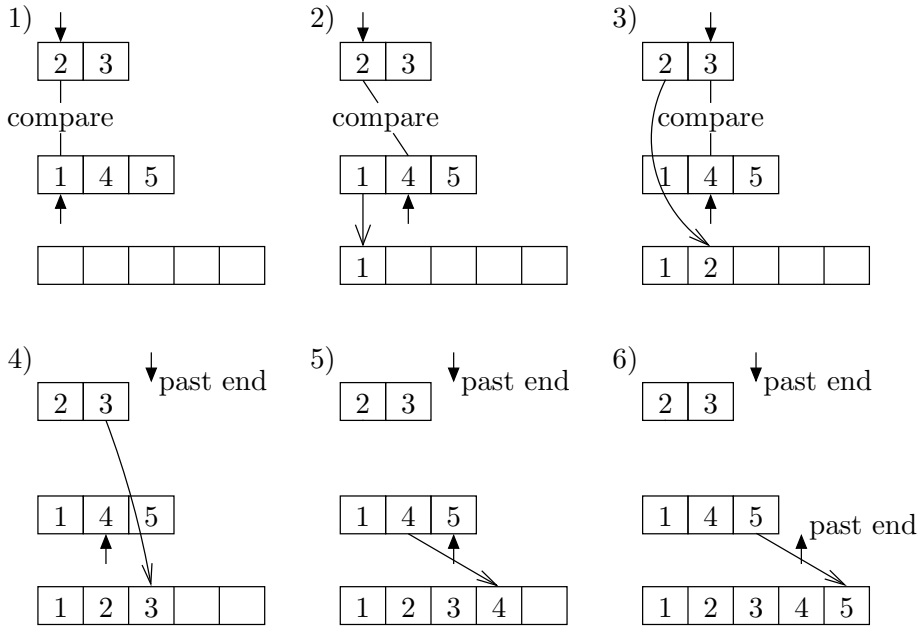

Figure 2.2.2: Merging two sorted lists into one sorted list in linear time.

Let $T(n)$ denote the time it takes mergesort to sort a list of n elements. The split step in the algorithm takes no time because we don't need to take the list and generate two smaller lists. It takes mergesort time $T(n/2)$ to sort each of the two lists, and it takes linear time to merge two smaller sorted lists into one big sorted list. Therefore, we get a recursive definition of $T(n)$ of the form

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n).$$

Solving this recurrence relation yields

$$T(n) = \mathcal{O}(n \log n).$$

We describe one way of solving the recurrence above. We can view the computation of mergesort as a tree (see Figure 2.2.3). Each node in the tree corresponds to one recursive application of mergesort. The amount of time each recursive call contributes to the total running time is shown in the node, and any recursive invocations made from some invocation of the mergesort algorithm are represented by children of the node representing that invocation. The former represents the $\mathcal{O}(n)$

part of the recurrence relation for $T(n)$, the latter represents the $2T\left(\frac{n}{2}\right)$ part of that recurrence. Notice that the tree has at most $\log n$ levels because the smallest list we ever sort has size 1, and that the values in the nodes in Figure 2.2.3 add up to $\mathcal{O}(n)$ at each level of the tree. Hence, $T(n) = \mathcal{O}(n \log n)$.

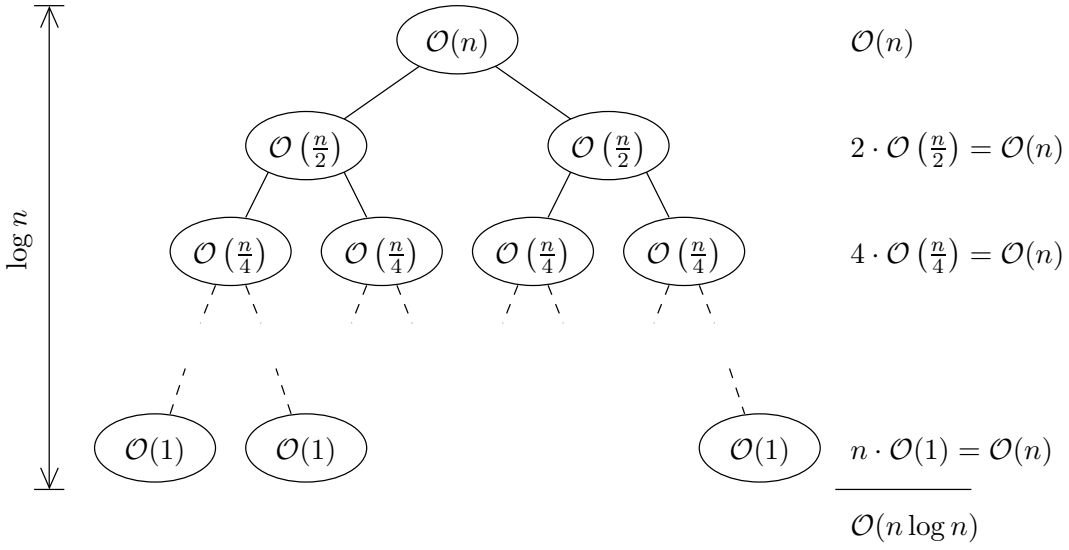

Figure 2.2.3: Computation of the time complexity of mergesort.

2.2.2 Lower Bound on the Complexity of Sorting

We define a special setting called the comparison-based model, for which we demonstrate a lower bound on the worst case complexity of any sorting algorithm.

In the *comparison-based model*, comparisons of pairs of list elements are the driving force of a sorting algorithm. In other words, the sorting algorithm branches only after comparing a pair of list elements. We can view a sorting algorithm in the comparison-based model as a decision tree where nodes correspond to comparisons of elements. Leaves of the decision tree correspond to permutations of the list elements. Each leaf represents a permutation of list elements that turns some unordered lists into ordered lists.

Without loss of generality, assume that no two elements of any list are equal. Then every comparison of list elements a and b has two possible outcomes, namely $a < b$ or $a > b$. Thus, the decision tree is a binary tree.

Theorem 2.2.1 *In the comparison-based model, any sorting algorithm takes* $\Omega(n \log n)$ *time in the worst case.*

Proof: Suppose the sorting algorithm is given an unordered list of length n. There are $n!$ possible orderings of the list elements. Assuming that no two elements of the list are equal, only one of these orderings represents a sorted list. The algorithm must permute the elements of each ordering (including the one that represents a sorted list) in a different way in order to have a sorted list when it finishes. This means that the decision tree representing the sorting algorithm must have at

least $n!$ leaves. The minimum height of the tree is $\log(n!) = \mathcal{O}(\log(n^n)) = \mathcal{O}(n \log n)$. Therefore, the sorting algorithm must make at least $n \log n$ comparisons in order to turn some ordering of list elements into a sorted one. ■

2.2.3 Finding the k-th Largest Element

Suppose that we have an unordered list, and once again assume that no two elements of the list are equal. We would like to find the k-th largest element in the list. We present some simple, but less efficient algorithms, and then give a linear time algorithm for finding the k-th largest element.

We could sort the list before we search. We can easily find the k-th largest element of a sorted list in constant time. However, sorting the list takes at least $\mathcal{O}(n \log n)$ time, which will cause this approach to finding the k-th largest element of a list run in $\mathcal{O}(n \log n)$ time. We can do better than that.

We could maintain the largest k elements in a heap and then go through the list elements one by one, updating the heap as necessary. After seeing all list elements, we use the heap to find the k-th largest element by removing $k-1$ elements from the root of the heap and then looking at the root of the heap. This approach takes $\mathcal{O}(n \log(k))$ time. We now describe an algorithm that find the k-th largest element in linear time.

Suppose that we can find the median of the list elements in linear time (we will show that this can be done afterwards). If the k-th largest element of the original list is the median, we are done. Otherwise we recursively search for the k-th largest element of the original list either in the list of elements that are greater than the median or in the list of elements that are less than the median. We need to search in only one of those, and we can forget about the other. The k-th largest element of the original list is the k'-th largest element of the new smaller list, where k' need not be the same as k.

We demonstrate how the algorithm works by using it to find the 5th largest element in some 15-element list. The process is illustrated in Figure 2.2.4. First of all, the algorithm finds the median of the original list and realizes that the median is not the 5th largest element of the list. Since the median is the 8th largest element of the list, the algorithm looks at elements of the list that are greater than the median. The 5th largest element of the original list is also the 5th largest element in the list of elements greater than the median. The algorithm now searches for the 5th largest element in the new list (second row in Figure 2.2.4). The median of the new list is its 4th largest element. Therefore, the algorithm looks for the 5th largest element of the original list in the list of elements that are smaller than the median it found. In that list, the 5th largest element from the original list is the 1st largest element. The algorithm continues in this fashion until it ends with a single-element list whose median is the 5th largest element of the original list.

In each step, the algorithm searches for the k-th largest element in a list that's half the size of the list from the previous step. The time complexity of this search is, then,

$$T(n) = \mathcal{O}(n) + T\left(\frac{n}{2}\right).$$

Solving this recurrence yields

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

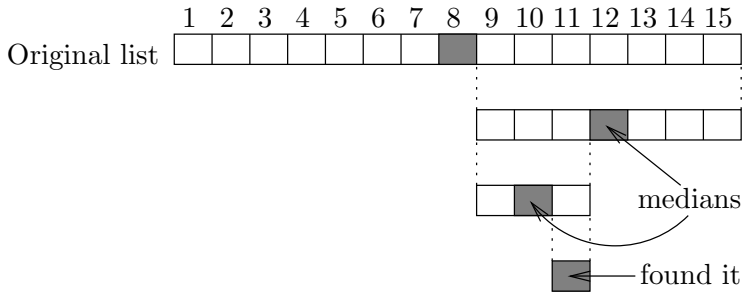

Figure 2.2.4: Finding the 5th largest element in a list of 15 elements. For simplicity of presentation, the list elements are already ordered, but this isn't necessary for the algorithm to work. The 5th largest element has index 11. Each "row" in the picture represents one recursive invocation of the algorithm.

so we have a linear time algorithm for finding the k-th largest element.

Finally, we show that it is possible to find the median of any list in linear time. We start by finding a near-median of the list. For our purposes, a near-median is an element of the list that is greater than at least 3/10 and less than at least 3/10 of the list elements. This happens in three steps

1. Split the list into blocks of five elements (and possibly one block with fewer elements).
2. Find the median in each of these blocks.
3. Recursively find the median of the medians found in the previous step.

The median found by the recursive call is greater than at least 3/10 of the list elements becaue it is greater than one half of the medians of the five-element blocks, and all those medians are greater than two elements in their respective blocks. By similar reasoning, this median is smaller than at least 3/10 of the list elements.

If the median of the original list is the near-median of the list we are currently looking at, we are done. Otherwise we recursively search for the median of the original list either in the list of elements that are greater than the near-median or in the list of elements that are less than the near-median. We need to search in only one of those, and we can forget about the other. Thus, in each step, we search for the median of the original list in a list that is at most 7/10 the size of the list from the previous step. The time complexity of this search is, then,

$$T(n) = \mathcal{O}(n) + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right).$$

The term $T\left(\frac{n}{5}\right)$ appears because we need to find a median of a list with $n/5$ elements when we want to find a median of a list of n elements. The $T\left(\frac{7n}{10}\right)$ appears because we subsequently search for the median of the list of length n in a list of length at most $7n/10$. The $\mathcal{O}(n)$ term appears because it takes linear time to find the medians of all the 5-element blocks the algorithm creates. If we solve the recurrence, we get

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

which is what we wanted.

2.3 Dynamic Programming

Dynamic programming utilizes recursion while maintaining information about subproblems that have already been solved (this is called *memoization*), allowing us to to efficiently find a solution to a subproblem. We must break up the problem into a polynomial number of subproblems if we want to use memoization.

We investigate the weighted interval scheduling problem and the difficulties associated with breaking it down into subproblems, and show how to use dynamic programming to find an optimal solution in polynomial time.

2.3.1 Weighted Interval Scheduling Problem

In the weighted interval scheduling problem, we want to find the maximum-weight subset o non-overlapping jobs, given a set J of jobs that have weights associated with them. Job $i \in J$ has a start time, an end time, and a weight which we will call w_i. We seek to find an optimum schedule, which is a subset S of non-verlapping jobs in J such that the sum of the weights of the jobs in S is greater than the sum of the weights of jobs in all other subsets of J that contaqin non-overlapping jobs. The weight of the optimum schedule is defined as

$$\min_{S \subseteq J} \sum_{i \in S} w_i.$$

In Figure 2.3.5, the optimum schedule is one that contains only job 2, and the weight of the optimum schedule is 100.

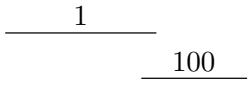

Figure 2.3.5: An instance of the weighted interval scheduling problem with two overlapping jobs. Job weights are shown above the lines representing them. Job 1 has weight $w_1 = 1$, job 2 has weight $w_2 = 100$. This instance can be used to spite the greedy approach that solves the non-weighted case of this problem.

We could try to use the greedy algorithm for job scheduling where jobs don't have weights. However, Figure 2.3.5 shows an instance where this algorithm fails to pick the optimum schedule because it would pick the first job with weight 1 and not the second job of weight 100. We can easily spite other greedy approaches (shortest job first, job overlapping with the least number of jobs first, earliest starting time first) using same counterexamples as in the non-weighted version and giving each job a weight of 1. We need to resort to a different method of algorithm design.

Consider the following algorithm: We look at the first job, consider the two possibilities listed below, and pick the better one. This approach yields the optimum solution because it tries all possible schedules.

1. Schedule the first job and repeat the algorithm on the remaining jobs that don't overlap with it.

2. Drop the first job and apply the algorithm to all jobs except for this first job.

The weights of the two schedules are as follows.

1. w_1+(max weight that can be scheduled after picking job 1)

2. max weight that can be scheduled after dropping job 1

Unfortunately, this approach yields an exponential time algorithm because in the worst case, each recursive application of this algorithm reduces the problem size by one job, which gives us

$$T(n) = \mathcal{O}(1) + 2T(n-1)$$

as a recurrence for the running time $T(n)$ of this algorithm. Solving this recurrence yields

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

which is bad. This time complexity stems from the fact that the algorithm tries all possible schedules. We get the same running time if, instead of considering whether to include the first job in the schedule, we consider whether to include a randomly picked job in the schedule.

Suppose that the jobs are sorted in the order of starting time. Now if we apply our recursive algorithm that considers whether to include the first job in the optimal schedule or whether to exclude it, we notice that it only looks at n subproblems where n is the number of jobs. Each subproblem consists of finding the optimum schedule for jobs i through n, with $1 \leq i \leq n$, and the solution to each subproblem depends only on subproblems of smaller size. Therefore, if we remember solutions to subproblems of smaller sizes that have already been solved, we can eliminate many redundant calculations the divide and conquer algorithm performs.

For $1 \leq i \leq n$, let value(i) be the maximum total weight of non-overlapping jobs with indices at least i. From our earlier discussion, we see that value(i) = $\max\{w_i + \text{value}(j), \text{value}(i+1)\}$ where j is equal to the index of the first job that starts after job i ends.

We now use the following algorithm to find the weight of the optimum schedule.

1. Sort jobs in ascending order of starting time.

2. For $1 \leq i \leq n$, find the smallest $j > i$ such that job j doesn't overlap with job i.

3. For $i = n$ down to 1, compute value(i) and store it in memory.

4. The weight of the optimum schedule is value(1).

If we want to recover the optimum schedule, we would remember the values of value(i) as well as the option that led to it.

The reader shall convince himself that once the list of jobs is sorted in ascending order of starting time, it takes time $\mathcal{O}(n \log n)$ to carry out step 2 of our algorithm. Hence, the running time of the algorithm is

$$T(n) = \mathcal{O}(n \log n) + \mathcal{O}(n \log n) + \mathcal{O}(n)\mathcal{O}(1),$$

where the two $\mathcal{O}(n \log n)$ terms come from sorting and from finding the first job that starts after job i for all jobs, and $\mathcal{O}(n)\mathcal{O}(1)$ appears because there are $\mathcal{O}(n)$ subproblems, each of which takes constant time to solve. Simplifying the expression for $T(n)$ yields

$$T(n) = \mathcal{O}(n \log n).$$