CHAPTER 16

Greedy Algorithms

• In dynamic programming, the optimal solution is described in a recursive manner, and then is computed ``bottom-up".

• Dynamic programming is a powerful technique, but it often leads to algorithms with higher than desired running times.

• Today we will consider an alternative design technique, called greedy algorithms.

• This method typically leads to simpler and faster algorithms, but it is not as powerful or as widely applicable as dynamic programming.

• The greedy concept: - make the choice that looks best at the moment
- hope that "local optimal" choices lead to "global optimal" solution

• Greedy strategy: At each stage, a greedy choice is made and the problem is reduced to a subproblem.

• We will give some examples of problems that can be solved by greedy algorithms. (This technique can be applied to a number of graph problems as well.)

• Even when greedy algorithms do not produce the optimal solution, they often provide fast heuristics (non-optimal solution strategies), and are often used in finding good approximations.

Huffman Codes: Data Encoding

- ***Huffman codes*** provide a method of encoding data efficiently.

- Normally when characters are coded using standard codes like ASCII, each character is represented by a fixed-length codeword of bits (e.g. 8 bits per character). Fixed-length codes are popular, because it is very easy to break a string up into its individual characters, and to access individual characters and substrings by direct indexing.

- However, fixed-length codes may not be the most efficient from the perspective of minimizing the total quantity of data.

- Consider the following example. Suppose that we want to encode strings over the (rather limited) 4-character alphabet C = {a, b, c, d}. We could use the following fixed-length code:

Character	a	b	c	d
Fixed-Length Codeword	00	01	10	11

- A string such as ``abacdaacac'' would be encoded by replacing each of its characters by the corresponding binary codeword.

a	b	a	c	d	a	a	c	a	c
00	01	00	10	11	00	00	10	00	10

- The final 20-character binary string would be ``00010010110000100010''.

- Now, suppose that you ``knew'' the relative probabilities of characters in advance. (This might happen by analyzing many strings over a long period of time. In applications like data compression, where you want to encode one file, you can just scan the file and determine the exact frequencies of all the characters.)

Variable-Length Encoding

• You can use this knowledge to encode strings differently. Frequently occurring characters are encoded using fewer bits and less frequent characters are encoded using more bits.

• For example, suppose that characters are expected to occur with the following probabilities. We could design a ***variable-length code*** which would do a better job.

Character	a	b	c	d
Probability	0.60	0.05	0.30	0.05
Variable-Length Codeword	0	110	10	111

• Notice that there is no requirement that the alphabetical order of character correspond to any sort of ordering applied to the codewords. Now, the same string would be encoded as follows.

a	b	a	c	d	a	a	c	a	c
0	110	0	10	111	0	0	10	0	10

• Thus, the resulting 17-character string would be ``0110010111001010''. We have achieved a savings of 3 characters, by using this alternative code. More generally, what would be the expected savings for a string of length *n*?

• For the 2-bit fixed-length code, the length of the encoded string is just *2n* bits. For the variable-length code, the expected length of a single encoded character is equal to the sum of code lengths times the respective probabilities of their occurrences. The expected encoded string length is just *n* times the expected encoded character length.

$$n(0.60 \cdot 1 + 0.05 \cdot 3 + 0.30 \cdot 2 + 0.05 \cdot 3) = n(0.60 + 0.15 + 0.60 + 0.15) = 1.5n.$$

• Thus, this would represent a 25% savings in expected encoding length.

Prefix Codes

• The question that we will consider today is how to form the best code, assuming that the probabilities of character occurrences are known.

• One issue that we didn't consider in the example above is whether we will be able to decode the string, once encoded. In fact, this code was chosen quite carefully.

• Suppose that instead of coding the character *`a'* as 0, we had encoded it as 1. Now, the encoded string ``111" is ambiguous. It might be *`d'* and it might be *`aaa'*. How can we avoid this sort of ambiguity?

• You might suggest that we add separation markers between the encoded characters, but this will tend to lengthen the encoding, which is undesirable. Instead, we would like the code to have the property that it can be *uniquely decoded*.

• Note that in the variable-length codes given in the example above no codeword is a prefix of another. This turns out to be the key property.

• Observe that if two codewords did share a common prefix, e.g. *a* → 001 and *b* → 00101, then when we see 00101… how do we know whether the first character of the encoded message is *a* or *b*.

• Conversely, if no codeword is a prefix of any other, then as soon as we see a codeword appearing as a prefix in the encoded text, then we know that we may decode this without fear of it matching some longer codeword.

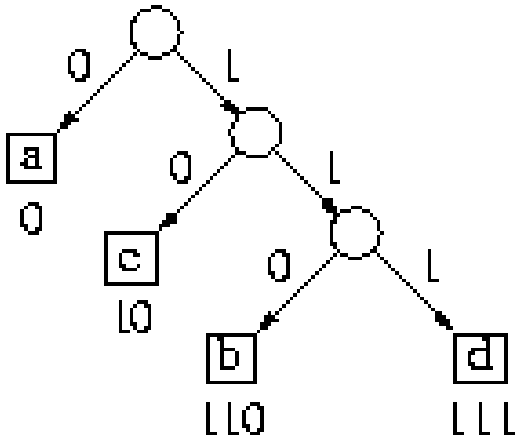
• Thus we have the following definition.

• ***Prefix Code:*** An assignment of codewords to characters so that no codeword is a prefix of any other.

Prefix Codes (cont.)

• Observe that any binary prefix coding can be described by a binary tree in which the codewords are the leaves of the tree, and where a left branch means ``0" and a right branch means ``1".

• The code given earlier is shown in the following figure. The length of a codeword is just its depth in the tree. The code given earlier is a prefix code, and its corresponding tree is shown in the following figure.



• Decoding a prefix code is simple. We just traverse the tree from root to leaf, letting the input character tell us which branch to take. On reaching a leaf, we output the corresponding character, and return to the root to continue the process.

Optimal Code Generation Problem

• Once we know the probabilities of the various characters, we can determine the total length of the encoded text.

• Let *p(x)* denote the probability of seeing character *x*, and let $d_T(x)$ denote the length of the codeword (depth in the tree) relative to some prefix tree *T*.

• The expected number of bits needed to encode a text with *n* characters is given in the following formula: $B(T) = n \sum_{x \in C} p(x) d_T(x).$

• This suggests the following problem:

Optimal Code Generation: Given an alphabet *C* and the probabilities *p(x)* of occurrence for each character *x* from *C*, compute a prefix code *T* that minimizes the expected length of the encoded bit-string, *B(T)*.

• Note that the optimal code is not unique. For example, we could have complemented all of the bits in our earlier code without altering the expected encoded string length.

• There is a very simple algorithm for finding such a code. It was invented in the mid 1950's by David Huffman, and is called a ***Huffman code***.

• By the way, this code is used by the Unix utility *pack* for file compression. (There are better compression methods however. For example, *compress*, *gzip* and many others are based on a more sophisticated method called the *Lempel-Ziv coding*.)

Huffman’s Algorithm

• Here is the intuition behind the algorithm. Recall that we are given the occurrence probabilities for the characters. We are going build the tree up from the leaf level.

• We will take two characters *x* and *y*, and ``merge" them into a single super-character called *z*, which then replaces *x* and *y* in the alphabet. The character *z* will have a probability equal to the sum of *x* and *y*'s probabilities.

• Then we continue recursively building the code on the new alphabet, which has one fewer character. When the process is completed, we know the code for *z*, say 010. Then, we append a 0 and 1 to this codeword, given 0100 for *x* and 0101 for *y*.

• Another way to think of this, is that we merge *x* and *y* as the left and right children of a root node called *z*. Then the subtree for *z* replaces *x* and *y* in the list of characters.

• We repeat this process until only one super-character remains. The resulting tree is the final prefix tree.

• Since *x* and *y* will appear at the bottom of the tree, it seems most logical to select the two characters with the smallest probabilities to perform the operation on. The result is Huffman's algorithm.

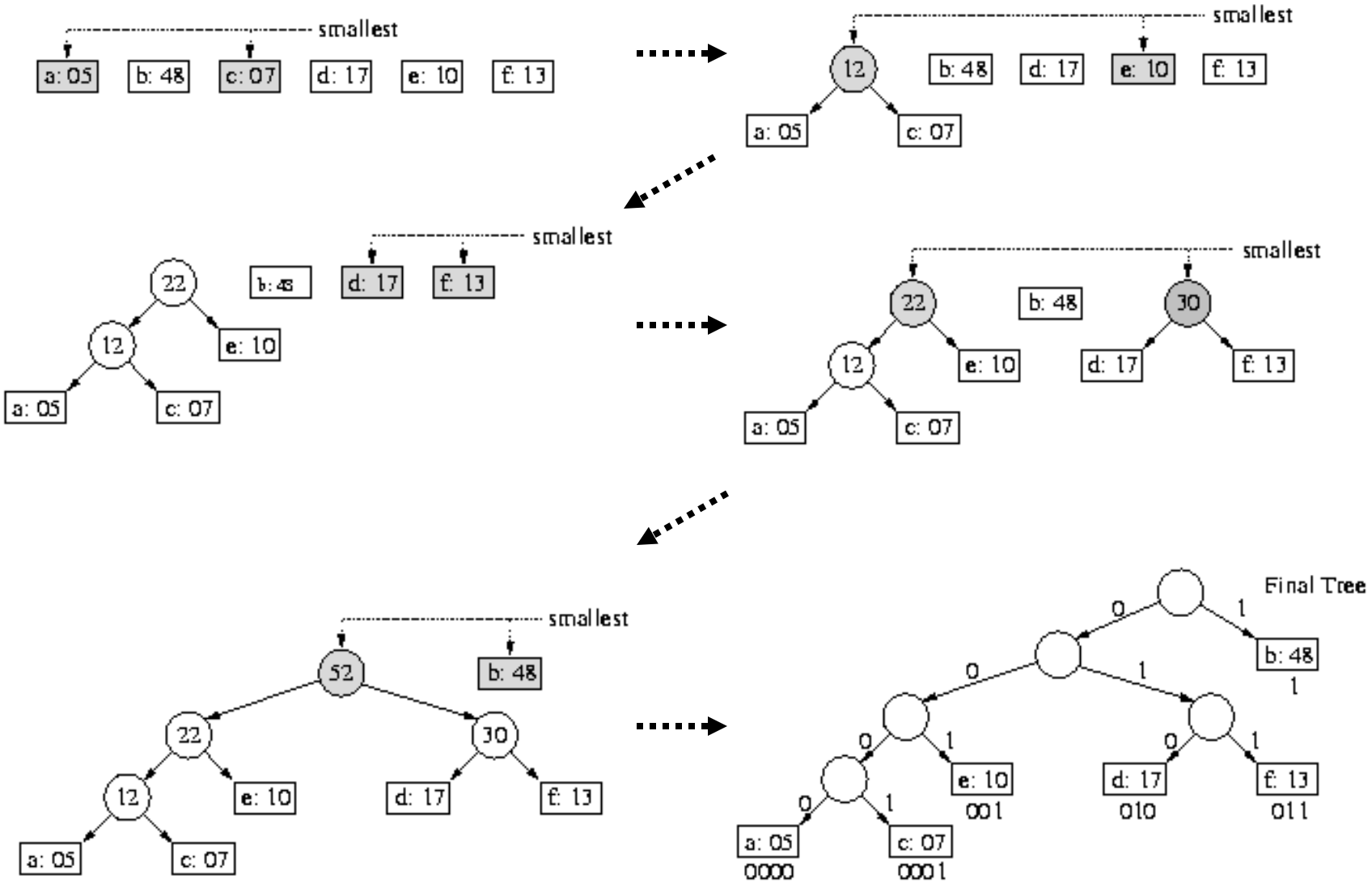
• The pseudocode for Huffman's algorithm is given on the next slide.

Huffman's Algorithm (cont.)

```
Huffman(int n, character C[1..n]) {
    Q = C;                                  // priority queue sorted by probability
    for i = 1 to n-1 {
        z = new internal tree node;
        z.left = x = Q.extractMin();        // extract the two smallest probabilities
        z.right = y = Q.extractMin();
        z.prob = x.prob + y.prob;           // z's probability is sum of x and y
        Q.insert(z);                        // insert z into priority queue
    }
    return the last element left in Q as the root;
}
```

- Let *C* denote the set of characters.
- Each character *x* from *C* is associated with a occurrence probability *x.prob*.
- Initially, the characters are all stored in a priority queue *Q*. Recall that this data structure can be built initially in *O(n)* time, and we can extract the element with the smallest key in *O(log n)* time and insert a new element in *O(log n)* time. The objects in *Q* are sorted by probability.
- Note that with each execution of the for-loop, the number of items in the queue decreases by one. So, after *n-1* iterations, there is exactly one element left in the queue, and this is the root of the final prefix code tree. Hence, *O(n logn)* totally.

Huffman's Algorithm: Example



Huffman's Algorithm: Correctness

• The big question that remains is why is this algorithm correct?

• Recall that the cost of any encoding tree T is $B(T) = \sum_{x \in C} p(x) d_T(x)$.

• Our approach will be to show that any tree that differs from the one constructed by Huffman's algorithm can be converted into one that is equal to Huffman's tree without increasing its cost.

• First, observe that the Huffman tree is a full binary tree, meaning that every internal node has exactly two children. It would never pay to have an internal node with only one child (since such a node could be deleted), so we may limit consideration to full binary trees.

• Our correctness prove is based on the following two claims.

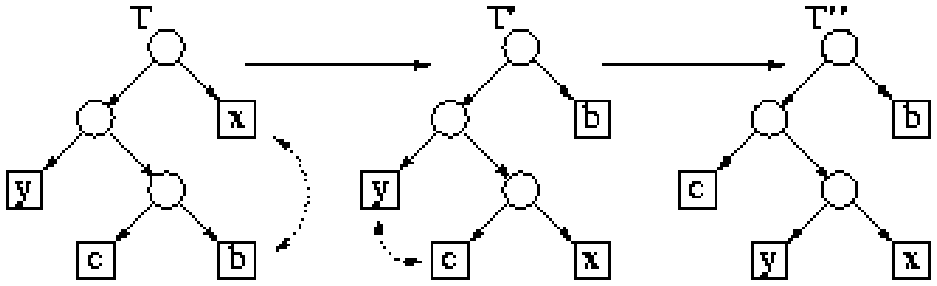
Claim 1: Consider the two characters, x and y with the smallest probabilities. Then there is an optimal code tree in which these two characters are siblings at the maximum depth in the tree.

• The above claim asserts that the first step of Huffman's algorithm is essentially the proper one to perform. The complete proof of correctness for Huffman's algorithm follows by induction on n (since with each step, we eliminate exactly one character).

Claim 2: Huffman's algorithm produces the optimal prefix code tree.

Proof of Claim 1

Let T be any optimal prefix code tree, and let b and c be two siblings at the maximum depth of the tree. Assume without loss of generality that $p(b) \leq p(c)$ and $p(x) \leq p(y)$ (if this is not true, then rename these characters). Now, since x and y have the two smallest probabilities it follows that $p(x) \leq p(b)$ (they may be equal) and $p(y) \leq p(c)$ (may be equal). Because b and c are at the deepest level of the tree we know that $d(b) \geq d(x)$ and $d(c) \geq d(y)$. Thus, we have $p(b)\text{-}p(x) \geq 0$ and $d(b)\text{-}d(x) \geq 0$, and hence their product is nonnegative. Now switch the positions of x and b in the tree, resulting in a new tree T'. This is illustrated in the following figure.



The cost of T' is

$$\begin{aligned} B(T') &= B(T) - p(x)d(x) + p(x)d(b) - p(b)d(b) + p(b)d(x) \\ &= B(T) + p(x)(d(b) - d(x)) - p(b)(d(b) - d(x)) \\ &= B(T) - (p(b) - p(x))(d(b) - d(x)) \\ &\leq B(T) \qquad \text{because } (p(b) - p(x))(d(b) - d(x)) \geq 0. \end{aligned}$$

Thus the cost does not increase, implying that T' is an optimal tree. By switching y with c we get a new tree T'' which by a similar argument is also optimal. The final tree T'' satisfies the statement of the claim.

Proof of Claim 2

We want to prove that Huffman's algorithm produces the optimal prefix code tree. The proof is by induction on *n*, the number of characters. For the basis case, $n = 1$, the tree consists of a single leaf node, which is obviously optimal. Assume inductively that when strictly fewer than *n* characters, Huffman's algorithm is guaranteed to produce the optimal tree. We want to show it is true with exactly *n* characters. Suppose we have exactly *n* characters. The previous claim states that we may assume that in the optimal tree, the two characters of lowest probability *x* and *y* will be siblings at the lowest level of the tree. Remove *x* and *y*, replacing them with a new character *z* whose probability is $p(z) = p(x) + p(y)$. Thus *n-1* characters remain. Consider any prefix code tree *T* made with the new set of *n-1* characters. We can convert it into a prefix tree *T'* for the original set of characters by undoing the previous operation and replacing *z* with *x* and *y* (adding a ``0" bit for *x* and a ``1" bit for *y*). The cost of the new tree is

$$\begin{aligned} B(T') &= B(T) - p(z)d(z) + p(x)(d(z)+1) + p(y)(d(z)+1) \\ &= B(T) - (p(x)+p(y))d(z) + (p(x)+p(y))(d(z)+1) \\ &= B(T) + (p(x)+p(y))(d(z)+1-d(z)) \\ &= B(T) + p(x) + p(y). \end{aligned}$$

Since the change in cost depends in no way on the structure of the tree *T*, to minimize the cost of the final tree *T'*, we need to build the tree *T* on *n-1* characters optimally. By induction, this exactly what Huffman's algorithm does. Thus the final tree is optimal.

Elements of the Greedy Strategy

- ***The Greedy Concept***
  - Makes the choice that looks best at the moment.
  - Hopes that "local optimal" choices lead to "global optimal‘ solution.
- ***Two basic properties of optimal greedy algorithms***
  - ***Optimal Substructure Property:*** A problem has optimal substructure if an optimal solution to the problem contains within it optimal solutions to its subproblems.
  - ***Greedy Choice Property:*** If a local greedy choice is made, then an optimal solution including this choice is possible.
- ***Dynamic Programming vs Greedy Approach***
  - Optimal substructure is basic to both
  - Greedy algorithms are generally much simpler
  - Proof of correctness for a greedy algorithm is often much harder
  - If a greedy algorithm works, then using a dynamic is an overkill.

Elements of the Greedy Strategy (cont.)

- ***Greedy Strategy Features***
  - Proceeds in a top-down order.
  - At each stage, a greedy choice is made and the problem is reduced to a subproblem.
  - *Must Show:* Making a greedy choice at each step produces a globally optimal solution.
  - *Necessary Requirements:* For a greedy strategy to work, the problem must have the optimal substructure property.
  - But optimal substructure is not enough to guarantee success.
- A proof that a greedy algorithm produces an optimal result can normally be obtained by combining
  - the proof that the algorithm satisfies the greedy choice property, and
  - the proof the algorithm satisfies the optimal substructure property.
- In Section 16.4, this topic is studied and sufficient conditions to ensure that a greedy algorithm is optimal are obtained (***Matroid Theory***).

Activity Selection Problem

• Last time we showed one greedy algorithm, Huffman's algorithm. Today we consider a couple more examples. The first is called activity scheduling and it is a very simple scheduling problem.

• We are given a set $S = \{1, 2, \ldots, n\}$ of n activities that are to be scheduled to use some resource, where each activity must be started at a given start time s_i and ends at a given finish time f_i.

• For example, these might be lectures that are to be given in a lecture hall, where the lecture times have been set up in advance, or requests for boats to use a repair facility while they are in port.

• Because there is only one resource, and some start and finish times may overlap (and two lectures cannot be given in the same room at the same time), not all the requests can be honored. We say that two activities i and j are non-interfering if their start-finish intervals do not overlap, that is $[s_i, f_i) \cap [s_j, f_j) = \emptyset$.

• The ***activity scheduling problem*** is to select a maximum-size set of mutually non-interfering activities for use of the resource. (Notice that there are many other criteria that we might have considered instead. For example, we might want to maximize the total utilization time for the facility, instead.)

• So how do we schedule the largest number of activities on the resource?

• Intuitively, we do not like long activities, because they occupy the resource and keep us from honoring other requests. This suggests the following greedy strategy: repeated select the job with the smallest duration $(f_i - s_i)$ and schedule it, provided that it does not interfere with any previously scheduled activities.

(This turns out to be non-optimal).

Greedy Algorithm

- Here is a simple greedy algorithm that does work.
- The intuition is the same. Since we do not like jobs that take a long time, let us select the job that finishes first and schedule it. Then, among all jobs that do not interfere with this first job, we schedule the one that finishes first, and so on.
- We begin by assuming that the activities have all be sorted by finish times, so that

$$f_1 \leq f_2 \leq \ldots \leq f_n,$$

(and of course the s_i's are sorted in parallel).

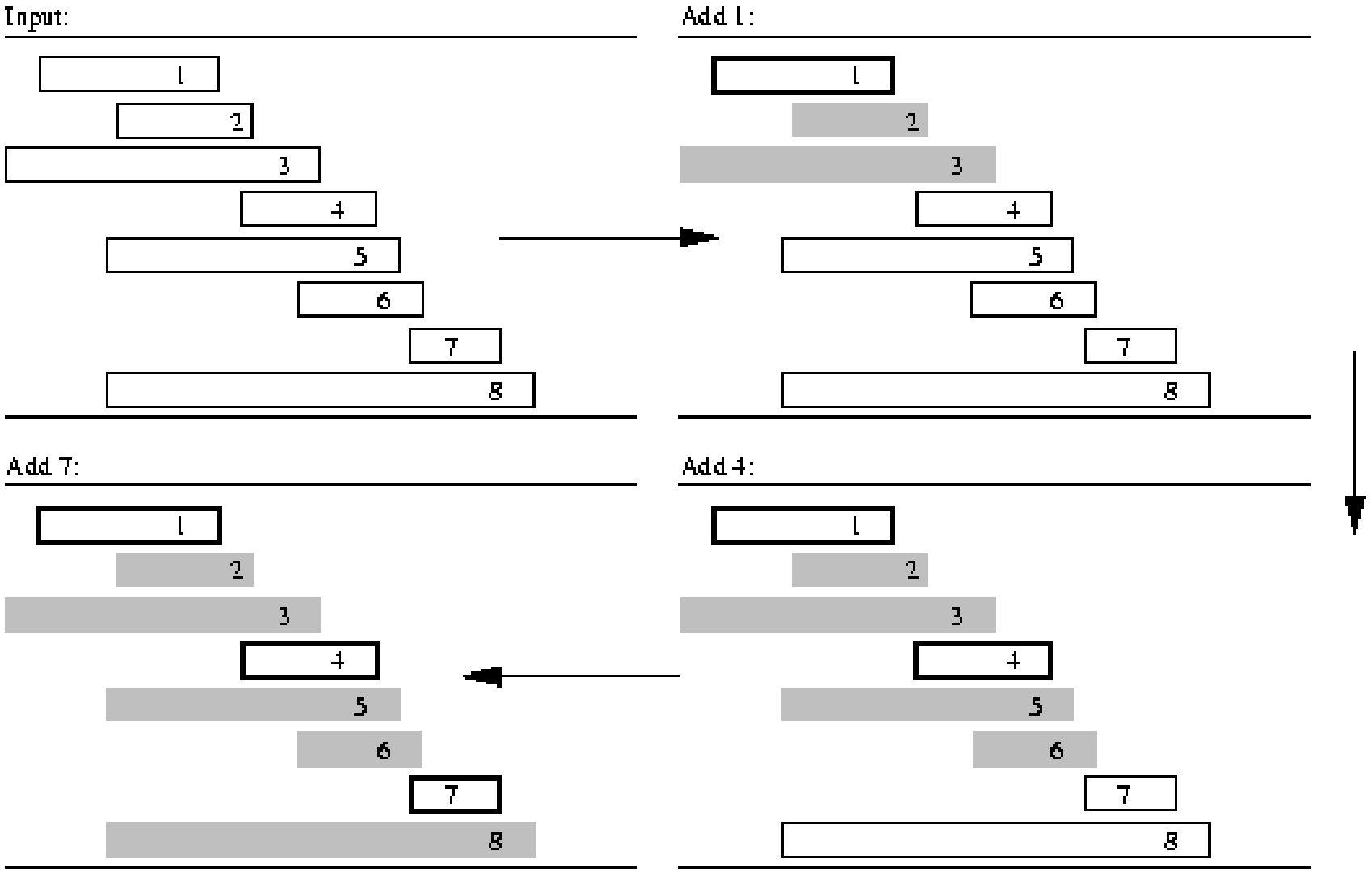
- The pseudocode is presented below, and assumes that this sorting has already been done. The output is the list *A* of scheduled activities.
- The variable *prev* holds the index of the most recently scheduled activity at any time, in order to determine interferences.

```
schedule(int n, int s[1..n], int f[1..n]) {
    // we assume f[1..n] is already sorted
    A = <1>;  prev = 1;                     // schedule activity 1 first
    for i = 2 to n {
        if (s[i] >= f[prev]) {              // no interference?
            append i to A;  prev = i;       // schedule i next
        }
    }
    return A;
}
```

- It is clear that the algorithm is quite simple and efficient. The most costly work is that of sorting the activities by finish time, so the total running time is $\Theta(n \log n)$.

Greedy Algorithm: Example

- The final output will be {1,4,7}.



Greedy Algorithm: Correctness

• Our proof of correctness is based on showing that the first choice made by the algorithm is the best possible, and then using induction to show that the algorithm is globally optimal.

• The proof's structure is noteworthy, because many greedy correctness proofs are based on the same idea: show that any other solution can be converted into the greedy solution without increasing its cost.

Claim 1: Let $S = \{1, 2, \ldots, n\}$ be a set of activities to be scheduled, sorted by increasing finish times. Then there is an optimal schedule in which activity *1* is scheduled first.

Proof: Let *A* be an optimal schedule. Let *x* be the activity in *A* with the smallest finish time.

• If $x = 1$ then we are done. Otherwise, we form a new schedule *A'* by replacing *x* with activity *1*.

• We claim that this is a *feasible* schedule (i.e., it has no interfering activities).

• This is because $A-\{x\}$ cannot have any other activities that start before *x* finished, since otherwise these activities would interfere with *x*. Since *1* is by definition the first activity to finish, it has an earlier finish time than *x*, and so it cannot interfere with any of the activities in $A-\{x\}$. Thus, *A'* is a feasible schedule.

• Clearly *A* and *A'* contain the same number of activities, implying that *A'* is also optimal.

Greedy Algorithm: Correctness (cont.)

Claim 2: The greedy algorithm gives an optimal solution to the activity scheduling problem.

Proof: The proof is by induction on the number of activities.

• For the basis case, if there are no activities, then the greedy algorithm is trivially optimal.

• For the induction step, let us assume that the greedy algorithm is optimal on any set of activities of size strictly smaller than $|S|$, and we prove the result for S.

• Let S' be the set of activities that do not interfere with activity *1*. That is

$$S' = \{i \in S : s_i \geq f_1\}.$$

• Observe that any solution for S' can be made into a solution for S by simply adding activity *1*, and vice versa.

• Since *1* is in the optimal schedule (by the above claim), it follows that to produce an optimal solution for the overall problem, we should first schedule *1* and then append the optimal schedule for S'.

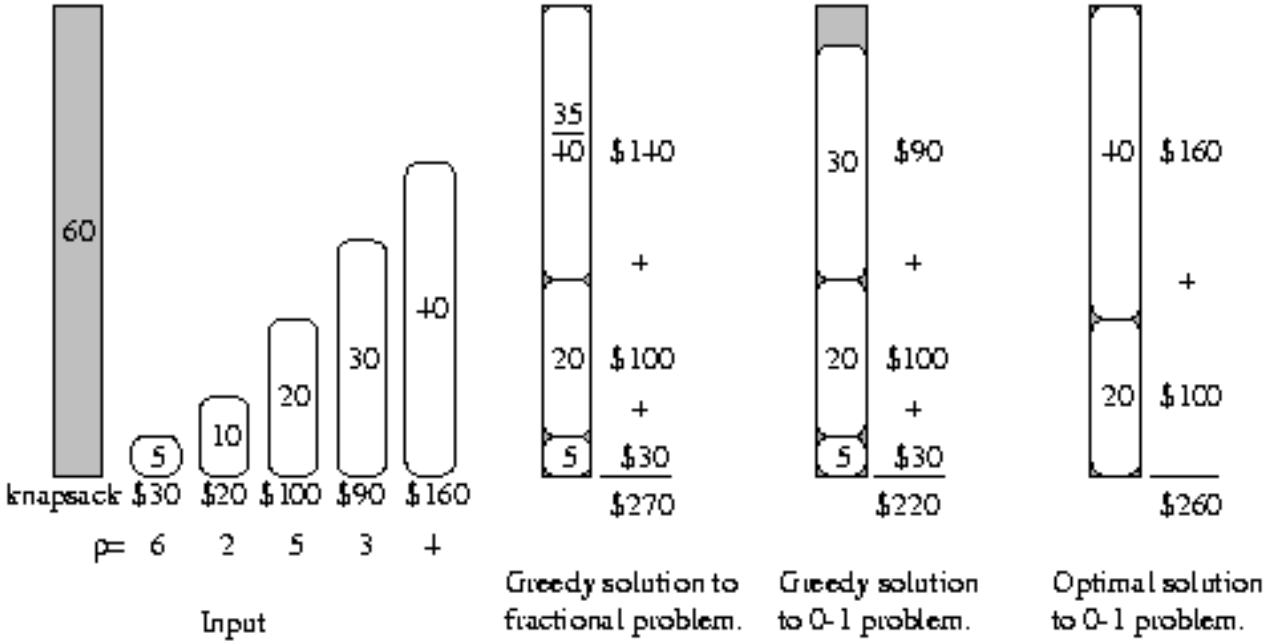
• But by induction (since $|S'|<|S|$), this is exactly what the greedy algorithm does.

Fractional Knapsack Problem

- The classical ***(0-1) knapsack problem*** is a famous optimization problem.
- A robber/thief is robbing a store, and finds *n* items which can be taken. The *i*th item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. He wants to take as valuable a load as possible, but has a knapsack that can only carry *W* total pounds. Which items should he take? (The reason that this is called 0-1 knapsack is that each item must be left (0) or taken (1) in it entirety. It is not possible to take part of an item or multiple copies of an item.)
- This optimization problem arises in industrial packing applications. For example, you may want to ship some subset of items on a truck of limited capacity.
- In contrast, in the ***fractional knapsack problem*** the setup is exactly the same, but the robber is allowed to take any *fraction* of an item for a fraction of the weight and a fraction of the value. So, you might think of each object as being a sack of gold/silver/bronze, which you can partially empty out before taking.
- The 0-1 knapsack problem is hard to solve, and in fact it is an NP-complete problem (meaning that there probably doesn't exist an efficient solution).
- However, there is a very simple and efficient greedy algorithm for the fractional knapsack problem.
- Let $p_i = v_i / w_i$ denote the *value-per-pound ratio*. We sort the items in decreasing order of p_i , and add them in this order. If the item fits, we take it all. At some point there is an item that does not fit in the remaining space. We take as much of this item as possible, thus filling the knapsack entirely. This is illustrated in the figure on the next slide.

Fractional Knapsack: Example

• It is easy to see that the greedy algorithm is optimal for the fractional problem. Given a room with sacks of gold, silver, and bronze, you would obviously take as much gold as possible, then take as much silver as possible, and then as much bronze as possible.



• But it would never benefit you to take a little less gold so that you could replace it with an equal volume of bronze.

• You can also see why the greedy algorithm may not be optimal in the 0-1 case. Consider the example shown in the figure. If you were to sort the items by p_i, then you would first take the items of weight 5, then 20, and then (since the item of weight 40 does not fit) you would settle for the item of weight 30, for a total value of $30 + $100 + $90 = $220.

• On the other hand, if you had been less greedy, and ignored the item of weight 5, then you could take the items of weights 20 and 40 for a total value of $100+$160 = $260. This feature of ``delaying gratification'' in order to come up with a better overall solution is your indication that the greedy solution is not optimal.

Homework 2: Will be posted on the web