CIS 5511. Programming Techniques

Greedy Algorithms

1. The idea

Greedy algorithm is another technique often used for optimization. Such an algorithm always makes the *locally* optimal choice, i.e., the next step that looks the best among the alternatives at the moment. Some greedy algorithms are optimal, but many others are not.

Take the planning matrix-chain multiplication problem as example: inspired by the first example, a greedy algorithm may "reduce" the largest dimension in each step, which in the second example leads to the solution $((A_1*A_2)*A_3)*(A_4*(A_5*A_6))$, which is not optimal.

Unlike dynamic programming, a greedy algorithm may fail to find the optimal solution, though it often finds a reasonably good one, and runs much faster than an algorithm using dynamic programming.

2. Example: Activity Selection

Problem: for a given set of activities A_1 ... A_n, each with its start and finish time for using a resource exclusively, find a schedule that maximizes the number of compatible activities. Please note that this is not the same as maximizing the time for the resource to be used.

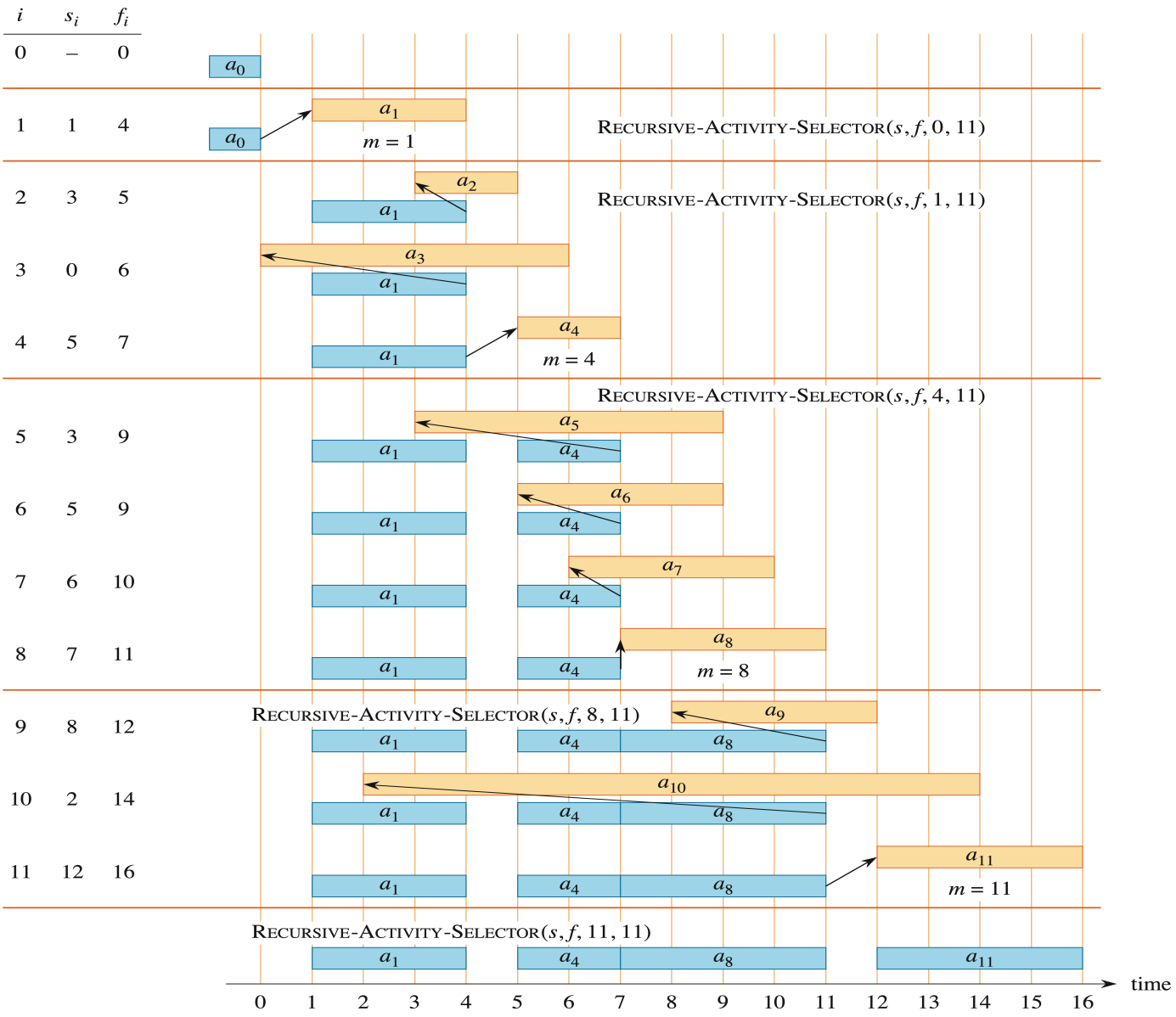
A greedy solution: recursively find the activity with the earliest finish time, and add it into the schedule.

In the following algorithm, array *s* and *f* represent the start and finish time of the activities, and they are sorted in monotonically increasingly order of finish time:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Beside the two arrays, the algorithm also takes indexes *k* and *n* as input, to indicate the range of activities to be considered. Initially, it is called as *Recursively-Activity-Selector(s, f, 0, n)* by algorithm *Activity-Selector(s, f)*.

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```



The recursive algorithm can be converted into a more efficient iterative algorithm:

GREEDY-ACTIVITY-SELECTOR(s, f, n)

```
1  A = {a_1}
2  k = 1
3  for m = 2 to n
4      if s[m] ≥ f[k]          // is a_m in S_k?
5          A = A ∪ {a_m}       // yes, so choose it
6          k = m               // and continue from there
7  return A
```

Both algorithms take Θ(n) time. The optimality of this algorithm can be proved by showing that any optimal solution can be changed into one that contains the activity with the earliest finish time.

3. Example: Knapsack problem

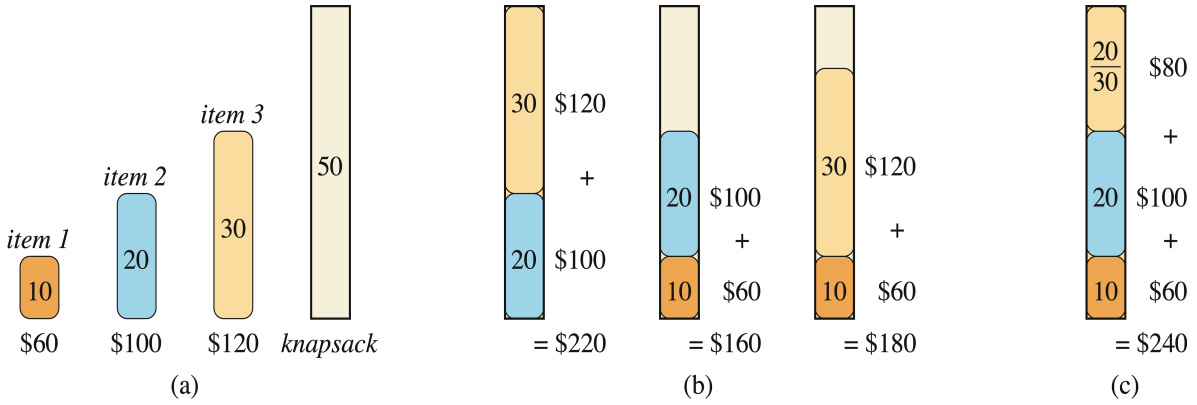
Given a set of items, each with a weight w_i and a value v_i, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit

W and the total value is as large as possible.

If it is allowed to take a fraction of an item, the problem is called "fractional knapsack problem" and a greedy algorithm can find an optimal solution by repeatedly picking the item with the highest unit value until that item is exhausted or *W* has been reached.

If every item must be completely taken or left, the problem is called "0-1 knapsack problem" and no greedy algorithm (such as the previous one, or repeatedly picking the item with the highest value) can always find an optimal solution. Instead, dynamic programming should be used for that.

Example:



4. Example: Huffman coding

Huffman coding is an efficient way to compress data for communication and storage. The idea is to use variable-length code for data units, where higher-frequent ones have shorter codes. There is no ambiguity as long as no codeword is also a prefix of some other codeword.

In the following example, the variable-length codeword has an average of 2.24 bits:

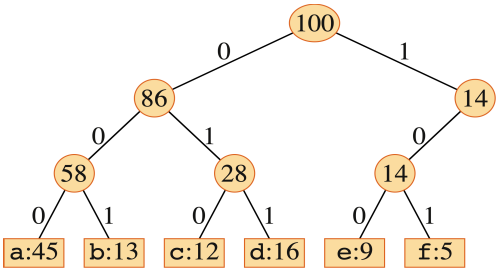
	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Assuming binary prefix coding, a solution can be represented by a binary tree, with each leaf corresponds to an unit, and its path from root represents the code for the unit (left as 0, right as 1). An optimal code is a binary tree with the minimum expected path length from the root to the leaves.

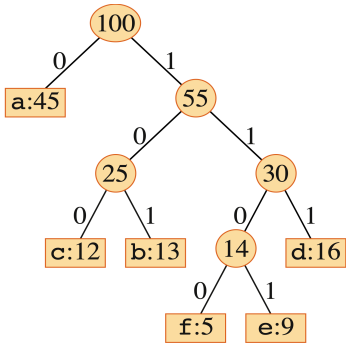
In the following algorithm, *C* contains a set of units, and *Q* is a priority queue containing binary trees prioritized by the frequency of their roots.

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       x = EXTRACT-MIN(Q)
6       y = EXTRACT-MIN(Q)
7       z.left = x
8       z.right = y
9       z.freq = x.freq + y.freq
10      INSERT(Q, z)
11 return EXTRACT-MIN(Q)      // the root of the tree is the only node left
```

Procedure HUFFMAN produces an optimal prefix code. Example:



(a)



(b)

5. Related techniques

There are many situations where an in-time solution is more valuable than an optimal solution. The following techniques deal with various restrictions on running time.

Hill Climbing: When an optimization problem can be represented as searching for the maximum value of a function in a multidimensional space, "hill climbing" is a greedy algorithm which attempts to find a better solution by making an incremental change to the current solution.

Gradient Descent: The idea is to take repeated steps in the opposite direction of the gradient (or approximate gradient) of the function at the current point, because this is the direction of steepest descent.

Best-First Search: If the next step is selected at the neighborhood of all explored points, "hill climbing" becomes "best-first search". As the selection is based on a heuristic function, it is also called *heuristic search*.

Real-time Computing: The response time is included in the specification of each problem instance. Time restrictions can be "hard" (as deadline) or "soft" (as degrading utility). The solution often involves hardware factors.

Anytime Algorithm: Such an algorithm improves the quality of its solution over time, and can stop at *anytime* with a best-so-far solution (e.g., approximating pi). An anytime

algorithm no longer has a determined solution and running time for a given problem instance.

Case-by-Case Problem Solving: To solve each *occurrence*, or *case*, of a problem using currently available knowledge and resources on the case. It is different from the traditional *Algorithmic Problem Solving*, which applies the same algorithm to all occurrences of all problem instances. Case-by-case Problem Solving is suitable for situations where the system has no applicable algorithm for a problem as a whole, while still may solve some special cases of it. This approach gives the system flexibility, originality, and scalability, at the cost of predictability, repeatability, and terminability.