CSE 100: Algorithm Design and Analysis
Chapter 16: Greedy Algorithms

Sungjin Im

University of California, Merced

Last Update: 3-17-2023

Greedy Algorithms

myopic.

- An algorithm is said to be greedy if it makes a choice that looks the 'best' at any given moment.
- The 'best' could be different depending on the criteria.
- (+) Greedy algorithms are usually extremely easy to implement and run fast: greedy algorithms are practitioners' best friends.
- (-) In general, greedy algorithms are not optimal: In other words, they may fail to find an optimum solution.

Greedy Algorithms

We will see why certain greedy algorithms are optimal for the following two examples:

- Interval Selection (a.k.a. Activity Selection in the textbook)
- Huffman Code

Interval Selection

Input:
$I_1 = (s_1, f_1), I_2 = (s_2, f_2), ..., I_n = (s_n, f_n)$ where they are ordered in increasing order of their finish times, i.e., $f_1 \leq f_2 \leq ... \leq f_n$.

Goal:
To find a largest subset of intervals that are mutually disjoint; we often interchangeably use 'disjoint', 'non-overlapping', and 'independent.'

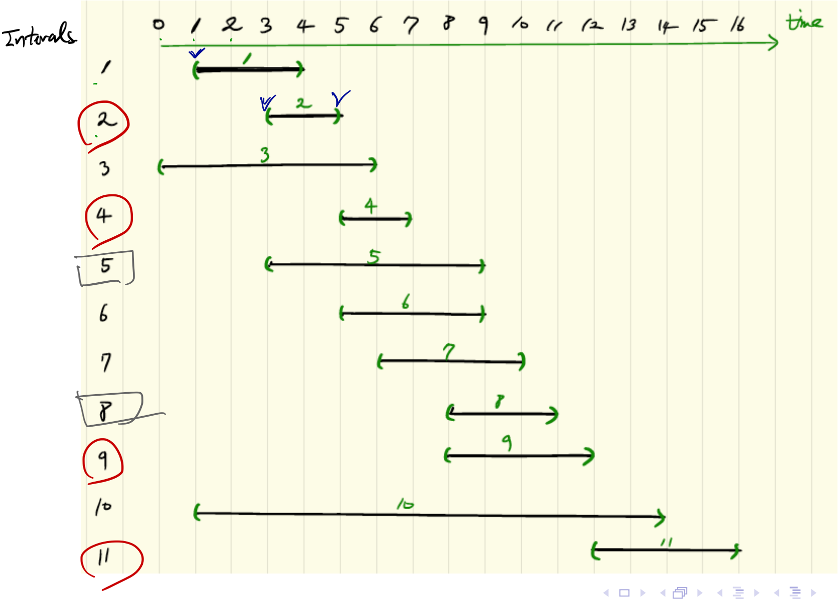
Interval Selection

Example

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

Interval Selection

Example



Interval Selection

DP solution (different from CLRS)

X

- We will first see a DP based solution; then we will consider greedy algorithms.
- In fact, using DP we can solve a more general problem: assume that each interval I_i has a certain weight $w_i > 0$ and the goal is to find a subset of mutually disjoint intervals such that the total weight of the chosen intervals is maximized.
- (If $w_i = 1$ for all intervals, then it becomes the original problem.)
- Let's consider the original problem. But you will see that the following solution will also work fort the weighted version of the problem.

Interval Selection

DP solution (different from CLRS)

For notational convenience, we introduce a 'dummy' interval I_0 which is disjoint from all other intervals. The dummy interval has no weight and every other interval has weight 1. Then, our goal is to choose a subset of mutually disjoint intervals with the maximum total weight.

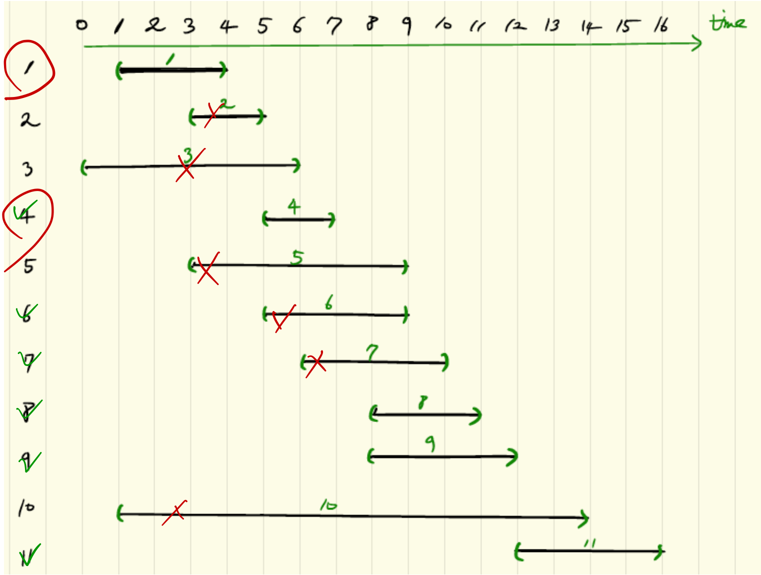
Letting $M(i)$ denote the maximum weight of any subset of mutually disjoint intervals from $I_0, I_1, I_2, ..., I_i$, we have

$$M(i) = \begin{cases} 0 & \text{if } i = 0 \\ \max\{M(j) + 1, M(i-1)\} & \text{otherwise,} \end{cases} \tag{1}$$

where I_j is the interval with the largest finish time that ends before I_i starts.

Interval Selection

Example



Interval Selection

DP solution (different from CLRS)

1. Set up table entries $M[i]$ corresponding to $M(i)$, $0 \le i \le n$.
2. Compute $M[0], M[1], ..., M[n]$ in this order using the recursion (1).
3. Return $M[n]$ as the maximum number of mutually disjoint intervals from $I_1, ..., I_n$.

Note that RT is

Interval Selection

DP solution (different from CLRS)

1. Set up table entries $M[i]$ corresponding to $M(i)$, $0 \le i \le n$.
2. Compute $M[0], M[1], \dots, M[n]$ in this order using the recursion (1).
3. Return $M[n]$ as the maximum number of mutually disjoint intervals from $I_1, \dots, I_n$.

Note that RT is $O(n \log n)$ since the number of DP entries is $\Theta(n)$ and computing each entry takes $O(\log n)$ time using binary search.

To find an actual solution achieving the optimum, we can compute $\ell(i)$ along, which is 1 if I_i can be chosen to achieve $M(i)$, otherwise 0.

i	0	1	2	3	4	5	6	7	8	9	10	11
$M[i]$	0	1	1	1	2	2	2	2	3	3	3	4
$\ell[i]$		1	1	1	1	0	1	1	1	1	0	1

Interval Selection

Greedy Algorithms

~~The DP wasn't that bad.~~ But there is another algorithm that is greedy, even simpler and more efficient. But which greedy algorithms? Here're some examples of greedy algorithms for the Interval Selection problem. counter-example for SIF being optimal.

- Shortest Interval First
- Earliest Starting Interval First

- Earliest Ending/Finishing Interval First (EF)

Interval Selection

Optimality of Earliest Finishing Interval First (EF): Key Lemma

Lemma

There is an optimal solution that includes I_1, the earliest finishing interval.

Proof.

To show this lemma consider an arbitrary optimal solution. To simplify the notation we refer to intervals by their indices. Suppose the optimal solution consists of intervals $i_1^* < i_2^* < \ldots < i_k^*$. If $i_1^* = 1$, we are already done. So assume that $i_1^* \neq 1$. We observe that intervals $1, i_2^*, \ldots, i_k^*$ are mutually disjoint, which will imply the lemma. This is because interval i_1^* ends before any of $i_2^*, \ldots i_k^*$ starts, and the interval 1 ends no later than i_1^*. Hence we obtained an optimal solution including interval 1 (I_1), proving the desired lemma. □

Interval Selection

Optimality of Earliest Finishing Interval First (EF) from the Key Lemma

We have shown the key lemma.

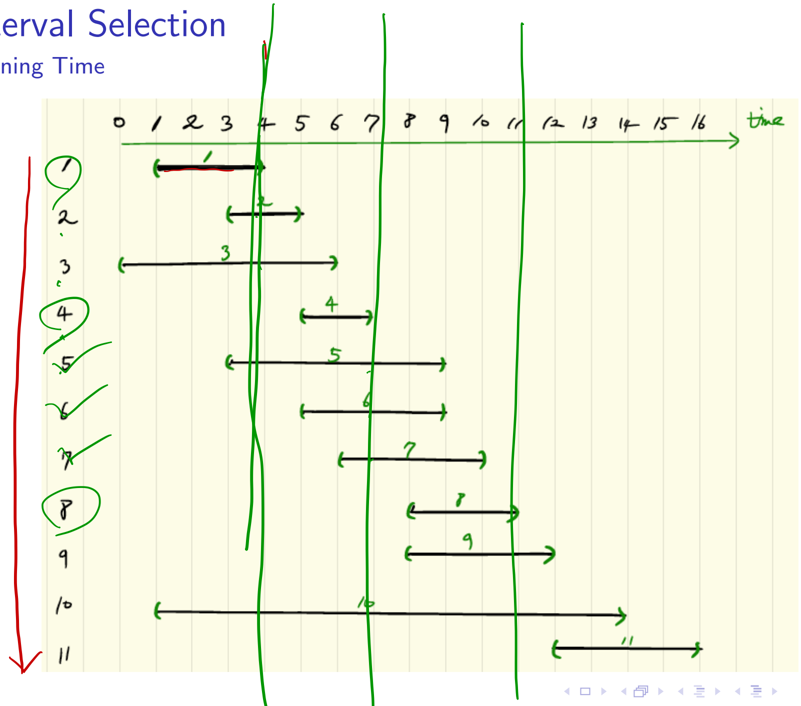
Now we want to show why the key lemma implies EF's optimality:

Proof.

Recursively apply the key lemma to show the optimality of EF. We know that we can safely choose I_1 thanks to the lemma. Then we will be forced to discard all intervals intersecting I_1. From the remaining intervals, to choose as many mutually disjoint intervals as possible, again thanks to the lemma, we can safely choose the earliest finishing interval. By repeating this argument, we see that EF is optimal. □

Interval Selection

Running Time



Interval Selection

Running Time

```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
        if s[m] ≥ f[k]
                A = A ∪ {a_m}
                k = m
return A
```

Running Time: $O(n)$ (assuming that intervals are sorted in increasing order of their finish times.)

Interval Selection

Limitations of Greedy Algorithms

As mentioned, greedy algorithms are often very simple and easy-to-implement. So if possible, use them. But you have to check if your greedy algorithm is optimal since not all greedy algorithms are optimal.

In fact, there are some problems for which greedy algorithms are unlikely to work. For example, think about the weighted interval selection problem where each interval has an arbitrary non-negative weight, and our goal is to choose a subset of mutually disjoint intervals with the maximum weight. We can find a DP-based algorithm for this problem by slightly modifying the above DP. However, we're not aware of any greedy algorithms for the weighted version.

Huffman Code

Compression

Suppose we want to compress a text file (say consisting of ASCII characters) by replacing each character with a certain binary codeword. To simplify the picture, we're given as input a set C of characters along with each character c's frequency $c.freq$. In other words, the character c appears $c.freq$ times in the text.

Example:

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5

Compression

Fixed-length code

Example:

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101

=100

Using this fixed-length code, we need 300 bits.

Compression

Fixed-length code

Example:

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101

Using this fixed-length code, we need 300 bits. Can we do better?

Compression

Variable-length code

aa → 00

b → 00

Example:

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	00	01	1	10	11

Using this variable-length code, we need 45 * 1 + 16 * 1 + (13 + 12 + 9 + 5) * 2= 139 bits.

Compression

Variable-length code

Example:

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	00	01	1	10	11

Using this variable-length code, we need $45 * 1 + 16 * 1 + (13 + 12 + 9 + 5) * 2 =$ 139 bits. But what is the problem?

Compression

Variable-length code

Example:

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	00	01	1	10	11

Using this variable-length code, we need $45 * 1 + 16 * 1 + (13 + 12 + 9 + 5) * 2 = 139$ bits. But what is the problem? Can't decode without ambiguity.

Example:

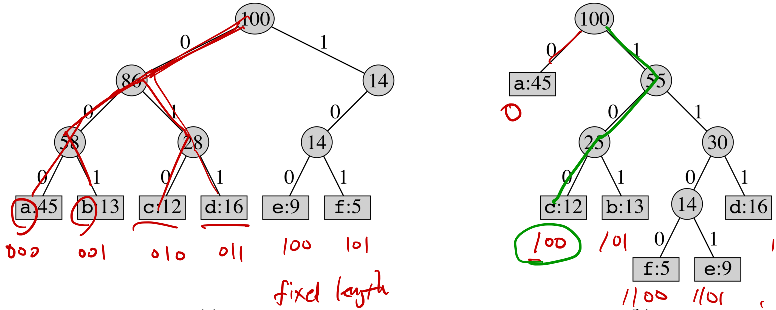
	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Prefix (prefix-free) code: no codeword is a prefix of some other codeword. Prefix code is ambiguity-free.

Using this variable-length code, we need 45 * 1 + 13 * 3 + 12 * 3 + 16 * 3 + 9 * 4 + 5 * 4 = 224 bits.

Compression

Tree representation of codes



(a) fixed length

(b) variable length prefix-free.

Cost of tree T: $B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$.

# of bits used by using code T

$d_T(c)$ = depth of c in tree T.

= c's codeword length

Compression

Tree representation of codes

Question: In a tree corresponding to a prefix code, characters appear only in leaf nodes. Correct?

Question: A fixed-length code is alway a prefix code. Correct?

Theorem

For any given input, there exists a prefix code that achieves the optimal data compression.

Compression

An Equivalent Problem Statement

Given a set C of characters along with each character c's frequency $c.freq$, find a prefix code that achieves the optimal data compression.

Compression

An Equivalent Problem Statement

Given a set C of characters along with each character c's frequency $c.freq$, find a prefix code that achieves the optimal data compression.

Surprisingly, there exists a quite simple greedy algorithm that is optimal for this problem!

Huffman Code

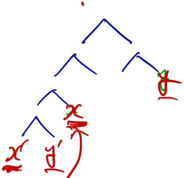
The Huffman algorithm works as follows: Create a node for each character. Each node is associated with a frequency. We recursively find two nodes of the minimum frequencies and merge them to create one; the new node has a frequency equal to the sum of the frequencies of the combined nodes. We will be done when we're left with only one node, which is the root note.

Huffman Code

Illustration

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5

Optimality of Huffman Code: Key Lemma



Lemma

There is an optimal prefix code where two characters x and y with the lowest frequencies are at the bottom of the tree and have the same parent.

Proof.

Sketch. Consider any optimal prefix code and its tree representation. If x and y appear in locations as described in the lemma statement, we are done. Otherwise, consider the bottom two nodes z and w sharing the same parent; the existence of these nodes follow from the observation that the tree has no node with exactly one child. Then, consider to swap x, y with z, w. It can only decrease the cost. $\square$

Optimality of Huffman Code from the Key Lemma

Lemma

There is an optimal prefix code where two characters x and y with the lowest frequencies are at the bottom of the tree and have the same parent.

(Sketch) We recursively apply this key lemma. Thanks to the key lemma, we know that there is an optimal prefix code/tree where x and y are at the bottom of the tree and have the same parent, which we denote as p. Then, it remains to find an optimal prefix code/tree for $C' := C \setminus \{x, y\} \cup p$ where $p.freq = x.freq + y.freq$. This is exactly the first step of the Huffman algorithm where we combine two characters with the lowest frequencies. Now we have an instance with one less characters. The optimality follows from an easy inductive argument.

Huffman Algorithm

insert all characters to Q.

HUFFMAN(C)

$n = |C|$

$Q = C$ min-priority queue.

key value : freq.

for $i = 1$ **to** $n - 1$

    allocate a new node z

O(lg n). $z.left = x =$ EXTRACT-MIN(Q)

    $z.right = y =$ EXTRACT-MIN(Q)

    $z.freq = x.freq + y.freq$

    INSERT(Q, z)

return EXTRACT-MIN(Q)   // return t

Running Time:

Huffman Algorithm

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)    // return t
```

Running Time: The code uses a min-priority queue. If you use a min-heap, then the RT becomes $O(n \log n)$. However, if you use a more advanced data structure, the RT can be improved to $O(n \log \log n)$, which we don't cover in this course.

Huffman Algorithm

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n − 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)      // return t
```

Running Time: The code uses a min-priority queue. If you use a min-heap, then the RT becomes $O(n \log n)$. However, if you use a more advanced data structure, the RT can be improved to $O(n \log \log n)$, which we don't cover in this course.

Huffman Algorithm

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)      // return t
```

Running Time: The code uses a min-priority queue. If you use a min-heap, then the RT becomes $O(n \log n)$. However, if you use a more advanced data structure, the RT can be improved to $O(n \log \log n)$, which we don't cover in this course.