CSE 3101: Introduction to the Design and Analysis of Algorithms

Instructor: Suprakash Datta (datta[at]cse.yorku.ca) ext 77875

Lectures: Tues, BC 215, 7–10 PM

Office hours: Wed 4-6 pm (CSEB 3043), or by appointment.

Textbook: Cormen, Leiserson, Rivest, Stein.
Introduction to Algorithms (3rd Edition)

Greedy algorithms for optimization problems

Basic ideas:

1. In order to get an optimal solution, just keep grabbing what looks best.
2. No backtracking (reversing earlier choices) allowed.
3. Local algorithm; often produces globally optimal solutions.
4. Typically the algorithm is simple. The proof that a greedy algorithm produces an optimal solution may be harder.

"Every two year old knows the greedy algorithm"

Greedy algorithm examples

Game show problem: Choose the best m prizes.

Iterative Greedy Algorithm:

Loop: grabbing the best, then second best, ...
    if it conflicts with committed objects
    or fulfills no new requirements.
        Reject this next best object
    else
        Commit to it.

Making change problem: Find the minimum # of quarters, dimes, nickels, and pennies that total to a given amount.

Keep grabbing the largest coin that keeps the solution cost less than or equal to the given amount.

Making change

Example: 71 cents

Goal: Find an optimal non-conflicting solution.

Solution: A subset of the coins that total the amount.
(25, 25, 10, 10, 1)

Cost of Solution: The number of objects in solution or the sum of the costs of objects (5)

The greedy algorithm does not always work

Problem: Find the minimum number of 4, 3, and 1 cent coins to make up 6 cents.

Problem: Coins: 4,3 and 1 cents. Make change for 6 cents.

Greedy solution: (4, 1, 1) cost 3.
Optimal Solution: (3,3) cost 2.

Lessons:

1. Not all problems admit greedy algorithms.
2. For those that do, all greedy algorithms do not work.
3. The proof that a greedy algorithm works is subtle but essential.

Proving optimality of greedy algorithms

Loop Invariant: There is at least one optimal solution consistent with the choices made so far.

Initially no choices have been made and hence all optimal solutions are consistent with these choices.

It is often easier to carry out the proof by contradiction.

For denominations 1,5,10,25 cents, prove optimality for amount C cents.

Proving optimality of coin changing

Consider solutions from greedy algorithm Sol(G) and that from optimal Sol(O). Sort both in decreasing order.

Look at first place (k) where they differ. Sol(G) MUST contain a coin of higher denomination.

Case 1: Sol(G) has a 5 c coin, Sol(O) does not. Sol(O) must make 5 c with 1 c; cannot be optimal.

Case 2: Sol(G) has a 10 c coin, Sol(O) does not. Must make 10 c with 5 c and 1 c. Sol(O) cannot be optimal.

Case 3: Sol(G) has a 25 c coin, Sol(O) does not. If Sol(G) has 2 or more 25 c coins, Sol(O) must make 50 cents with 10c, 5c, 1c; cannot be optimal. Else Sol(O) must use 1 or 2 or 3 or more 10c; in each case, Sol(O) must be suboptimal.

Proving optimality of coin changing

Q: How is this consistent with

"LI: There is at least one optimal solution consistent with the choices made so far."

A: Take a different view of what we have done:

We proved that the next coin of Sol (G) agrees with that of some Sol(O).

Specifically, we proved that if no solution in Sol(O) agrees with Sol(G), then Sol(O) cannot be optimal.

Another example: Knapsack problem

Problem: Given n commodities, with values v_i dollars and weight w_i pounds, and a knapsack that can carry maximum weight K, to put in the knapsack a set of items that maximize total value. You can take arbitrary fractions of any item.

Greedy algorithm:

Sort in decreasing order of v_i/w_i.

Fill knapsack greedily.

Correctness: Compare Sol(G) with Sol(O), with both solutions sorted in decreasing order of v_i/w_i. If they differ, then prove that by replacing the object in Sol(O) with the object in Sol(G), we violate the optimality of Sol(O).

Next: more examples of greedy algorithms

Huffman codes, greedy scheduling (Ch 16).

A real problem: data compression

Let's think for a moment: when can you compress data?

Key question: Do you allow information to be lost?

Answer: depends on the application:

Music/movies: small loss ok.

Text/data file transmission/storage: no loss permitted.

Lossy compression: uses signal processing techniques.

- Used in computer vision, image and speech processing.

- Utilizes the fact that some part of the data (signal) can be discarded without perceptible quality loss.

Lossy data compression

Let's think for a moment: if you cannot throw away any data, how can you reduce its size?

Answer: by removing redundancy in the data.

E.g.: My wife sends me an sms "where are you?"

I could answer "I am at York", "at York", "York".

This is really lossy compression!

Aside: What about the obvious redundancy in language?

(utilized by sms-language, e.g. I lv u, wt 4 me …)

Why/when is redundancy useful?

Lossless data compression

Assume: message is given, and cannot be altered.

How can you reduce the size?

Ans: one way is to use variable length encodings.

If there are k characters in the alphabet, each character could be encoded using log k bits (fixed length encoding), or some characters could use 1 bit, some 2 bits, etc.

Tradeoff: ease of parsing.

Given: 011100100 011100 010 011001100 001

0111001000 0111000100110011000001

Lossless data compression

Idea: the more frequent the letter, the shorter its encoding.

We want unique parse trees (PREFIX codes).

	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

Figure 16.3 A character-coding problem. A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated. If each character is assigned a 3-bit codeword, the file can be encoded in 300,000 bits. Using the variable-length code shown, the file can be encoded in 224,000 bits.

Fixed and variable length codes

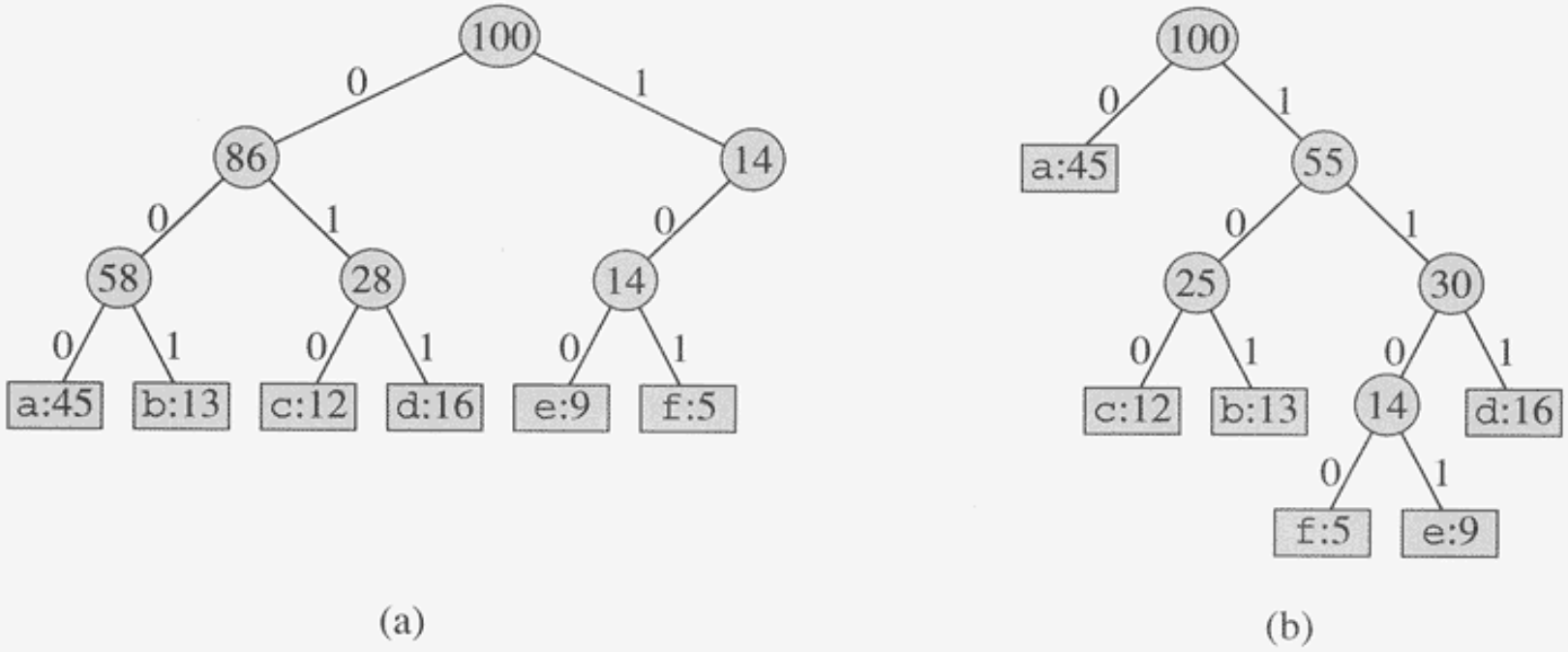


Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. **(a)** The tree corresponding to the fixed-length code a = 000, . . . , f = 101. **(b)** The tree corresponding to the optimal prefix code a = 0, b = 101, . . . , f = 1100.

Huffman codes - algorithm

Greedy strategy: select the two least weight nodes and make them children of the tree.

Replace the nodes with a new node with the sum of the weights

```
HUFFMAN(C)
1  n ← |C|
2  Q ← C
3  for i ← 1 to n − 1
4       do allocate a new node z
5          left[z] ← x ← EXTRACT-MIN(Q)
6          right[z] ← y ← EXTRACT-MIN(Q)
7          f[z] ← f[x] + f[y]
8          INSERT(Q, z)
9  return EXTRACT-MIN(Q)          ▷ Return the root of the tree.
```

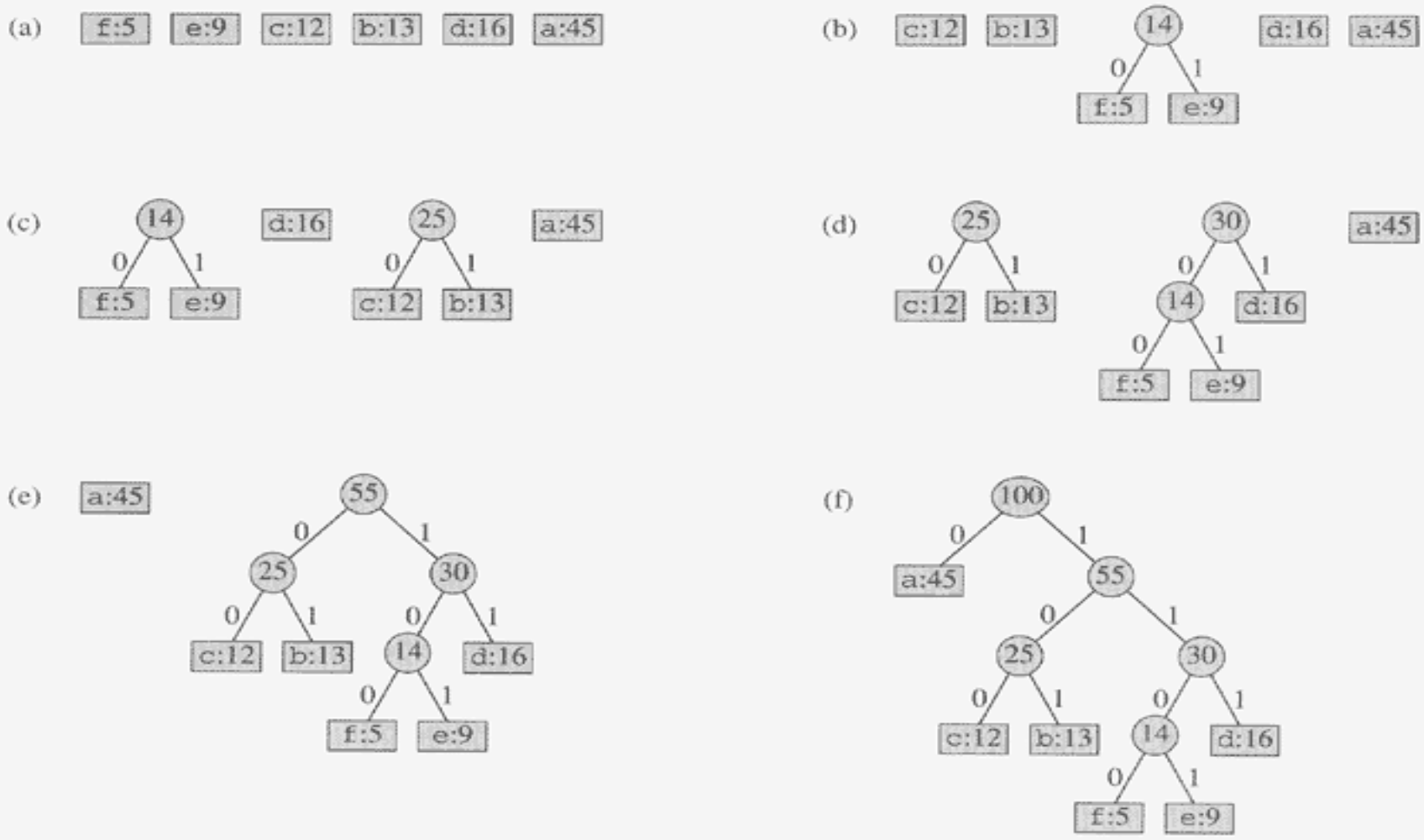


Figure 16.5 The steps of Huffman's algorithm for the frequencies given in Figure 16.3. Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of its children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child. The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter. **(a)** The initial set of $n = 6$ nodes, one for each letter. **(b)–(e)** Intermediate stages. **(f)** The final tree.

Optimality proof

Lemma: If x,y have the lowest frequencies, then there is an optimal prefix code in which they are sibling leaves.

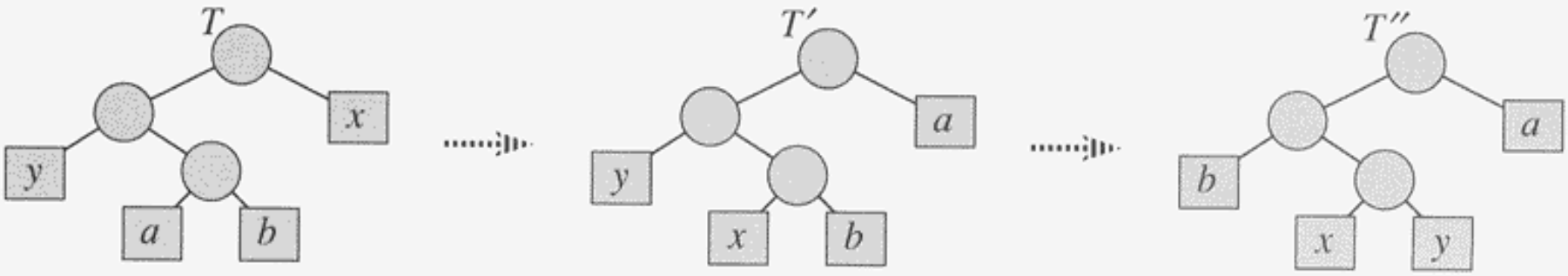


Figure 16.6 An illustration of the key step in the proof of Lemma 16.2. In the optimal tree T, leaves a and b are two of the deepest leaves and are siblings. Leaves x and y are the two leaves that Huffman's algorithm merges together first; they appear in arbitrary positions in T. Leaves a and x are swapped to obtain tree T'. Then, leaves b and y are swapped to obtain tree T''. Since each swap does not increase the cost, the resulting tree T'' is also an optimal tree.

Optimality proof

Note: If a,b have the lowest frequencies, then the greedy algorithm replaces them by another “character” c whose frequency is the sum of that of a,b.

Inductive argument!

Suppose that the greedy algorithm is optimal for k-1 letter alphabets.

For a k letter alphabet, it produces a tree S with x,y as children. Inductively, the tree S” obtained by fusing nodes x,y must be optimal.

Suppose there exists a *lower cost* tree T. There must exist a tree T’ with the same lower cost with x,y as children (*previous lemma*). Fuse nodes x,y in T’ to get T”. T” has strictly lower cost than S”; CONTRADICTION!.

Running time

```
HUFFMAN(C)
1  n ← |C|
2  Q ← C
3  for i ← 1 to n − 1
4       do allocate a new node z
5          left[z] ← x ← EXTRACT-MIN(Q)
6          right[z] ← y ← EXTRACT-MIN(Q)
7          f[z] ← f[x] + f[y]
8          INSERT(Q, z)
9  return EXTRACT-MIN(Q)          ▷ Return the root of the tree.
```

Naively, this requires $O(n^2)$ time.
With priority queues implemented with heaps, Extract-Min takes logarithmic time.
This gives total running time O(n log n).

More examples of greedy algorithms

The Job/Event Scheduling Problem

Inputs:

Jobs: Events with starting and finishing times
$<<s_1,f_1>,<s_2,f_2>,\ldots,<s_n,f_n>>$.

Solution: Schedules: A set of events that do not overlap.
Cost of Solution: The number of events scheduled.

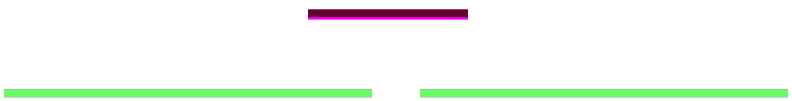
Goal: Given a set of events, schedule as many as possible.

Possible Greedy Criteria

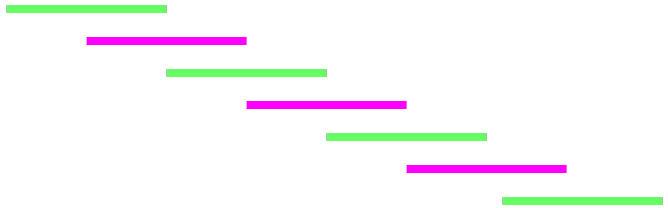
Greedy Criteria: The Shortest Event

Motivation: Does not book the room for a long period of time.

Counter Example



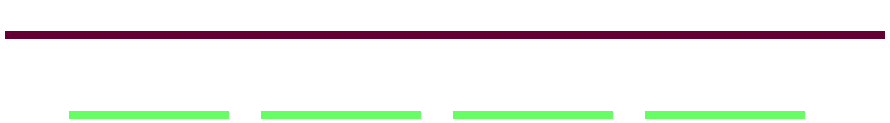
Possible Greedy Criteria



Optimal

Greedy Criteria: The Earliest Starting Time

Motivation: Common scheduling algorithm.



Schedule first

Optimal

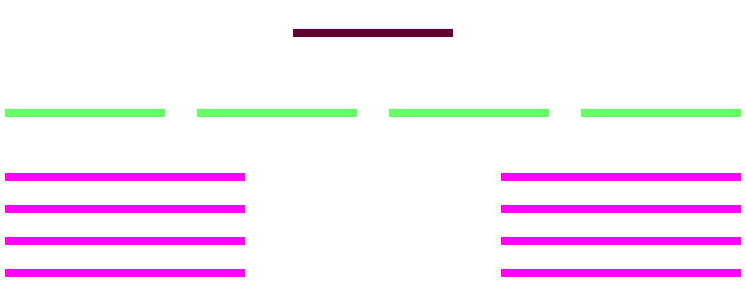
Counter Example

Possible Greedy Criteria

Optimal

Greedy Criteria: Conflicting with the Fewest Other Events

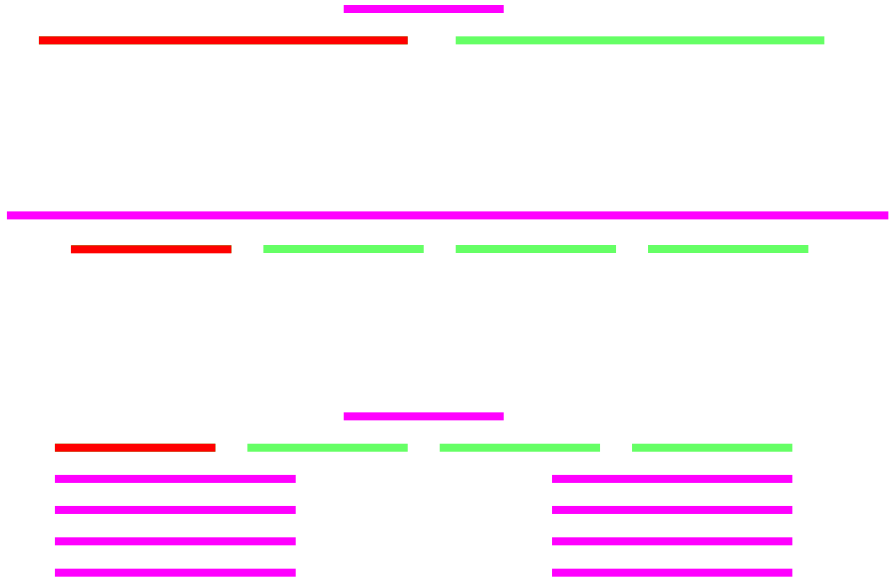
Motivation: So many can still be scheduled.



Schedule first
Optimal

Counter Example

Possible Greedy Criteria

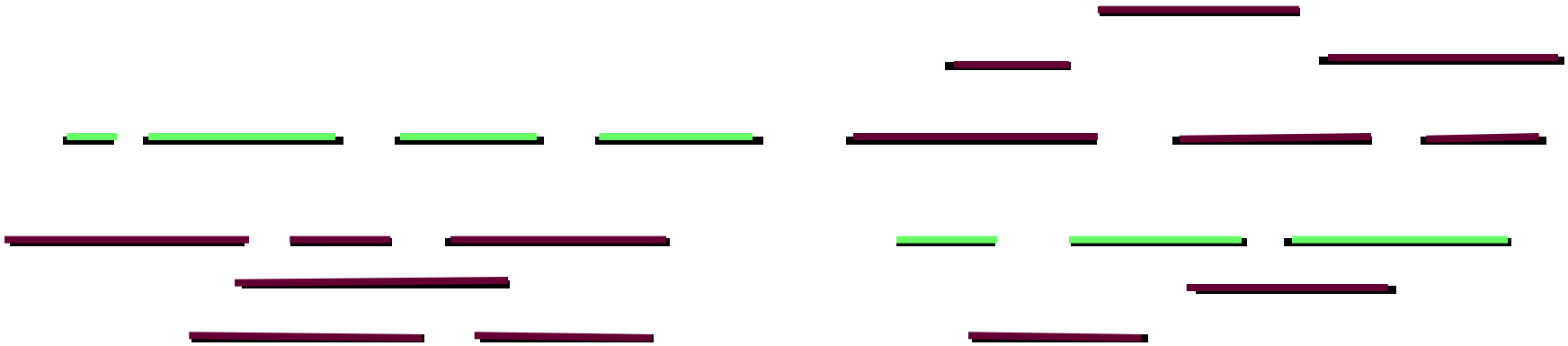


Greedy Criteria: Earliest Finishing Time

Motivation: Schedule the event who will free up your room for someone else as soon as possible.

Works!

Earliest Finishing Time



The Greedy Algorithm

algorithm *Scheduling* $(\langle\langle s_1, f_1\rangle, \langle s_2, f_2\rangle, \ldots, \langle s_n, f_n\rangle\rangle)$

$\langle pre-cond\rangle$: The input consists of a set of events.

$\langle post-cond\rangle$: The output consists of a schedule that maximizes the number of events scheduled.

```
begin
    Sort the events based on their finishing times f_i
    Commit = ∅      % The set of events committed to be in the schedule
    loop i = 1...n  % Consider the events in sorted order.
        if( event i does not conflict with an event in Commit ) then
            Commit = Commit ∪ {i}
    end loop
    return(Commit)
end algorithm
```

Optimality proof

Key step: Suppose a_m is the job with the earliest finish time. Then there is an optimal schedule which contains a_m.

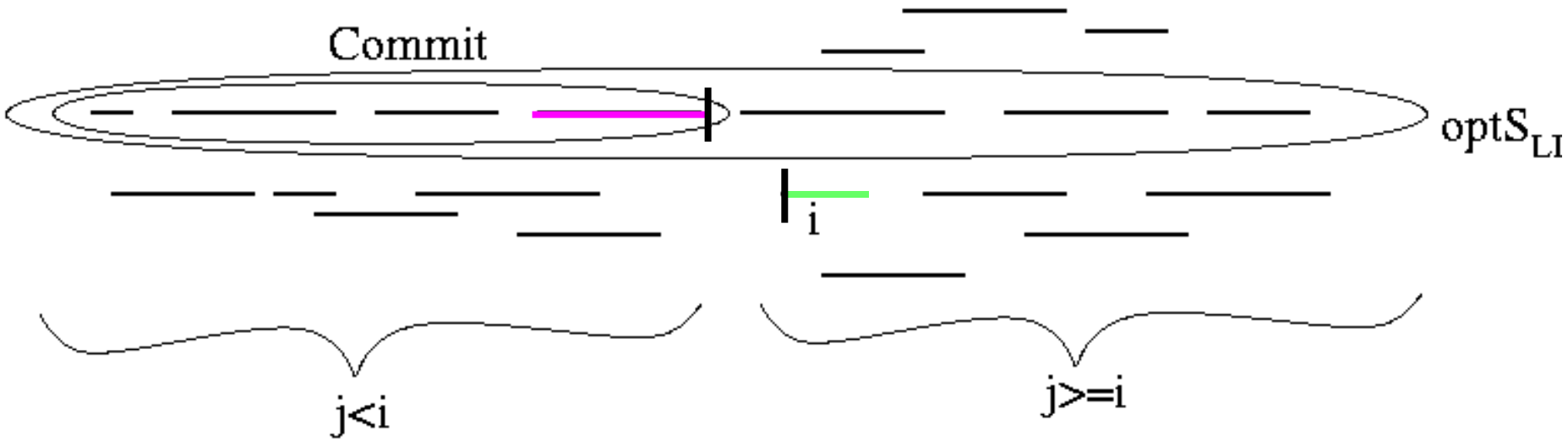
Consider any optimal solution S. Sort the jobs by increasing finish times. If the first job is a_m, we are done.
Else, the first job is a_k different from a_m . Consider the schedule S" = S – $\{a_k\}$ +$\{a_m\}$.

S" must have the same number of jobs as in S, since a_m does not conflict with any job that a_k did not conflict with. Therefore S" is optimal.

Running Time

Checking whether next event i conflicts with previously committed events requires

only comparing it with the last such event.



Running time is O(n), assuming that the algorithm is given a list of jobs already sorted by finish times.