Greedy Algorithms

Dr. Navjot Singh

Design and Analysis of Algorithms

Greedy Algorithms

What is a greedy algorithm?

Algorithm that makes a local decision with the goal of creating a globally optimal solution

Method for solving problems where optimal solutions can be defined in terms of optimal solutions to sub-problems

What does this mean? Where have we seen this before?

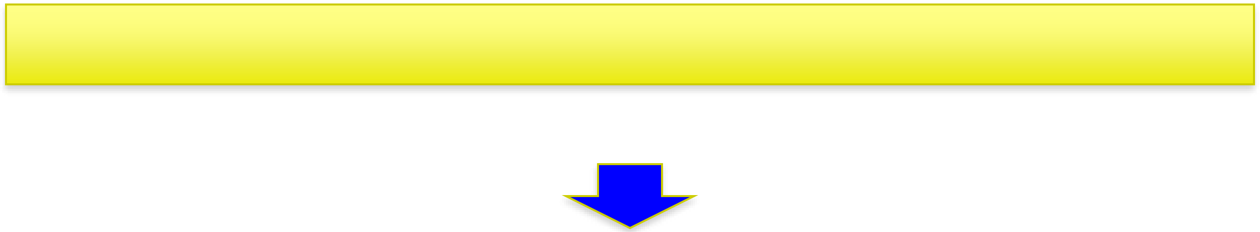
Greedy Algorithms

- Used to solve optimization problems.
- Problems exhibit optimal substructure.
- Problems also exhibit the **greedy-choice** property.
  - When we have a choice to make, make the one that looks best *right now*.
  - Make a **locally optimal choice** in hope of getting a **globally optimal solution**.
- The choice that seems best at the moment is the one we go with.
  - Prove that when there is a choice to make, one of the optimal choices is the greedy choice. Therefore, it's always safe to make the greedy choice.
  - Show that all but one of the subproblems resulting from the greedy choice are empty.

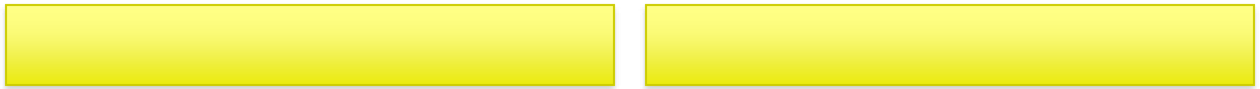
Greedy vs. divide and conquer

Divide and conquer

To solve the general problem:



Break into sum number of sub problems, solve:

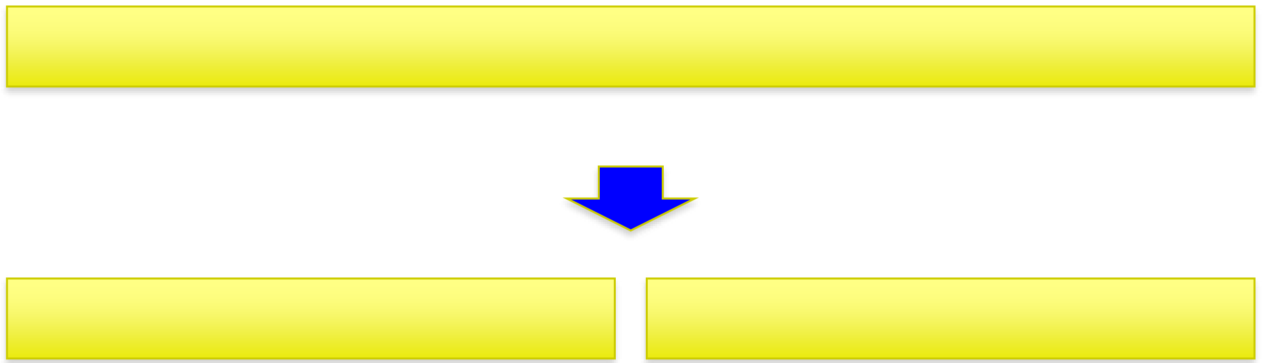


then possibly do a little work

Greedy vs. divide and conquer

Divide and conquer

To solve the general problem:

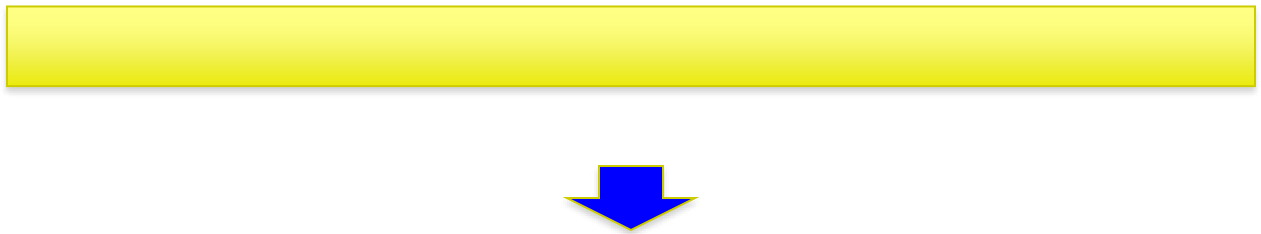


The solution to the general problem is solved with respect to solutions to sub-problems!

Greedy vs. divide and conquer

Greedy

To solve the general problem:

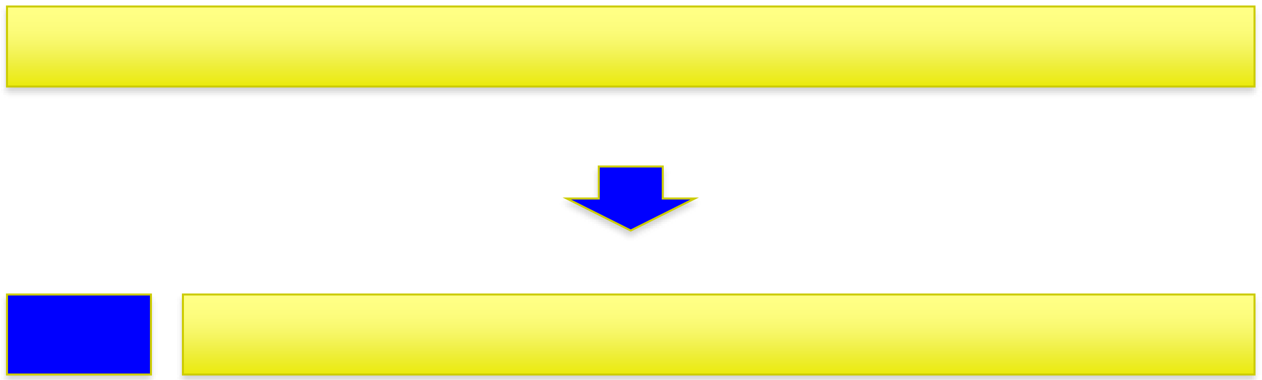


Pick a locally optimal solution and repeat

Greedy vs. divide and conquer

Greedy

To solve the general problem:



The solution to the general problem is solved with respect to solutions to sub-problems!

Slightly different than divide and conquer

Typical Steps

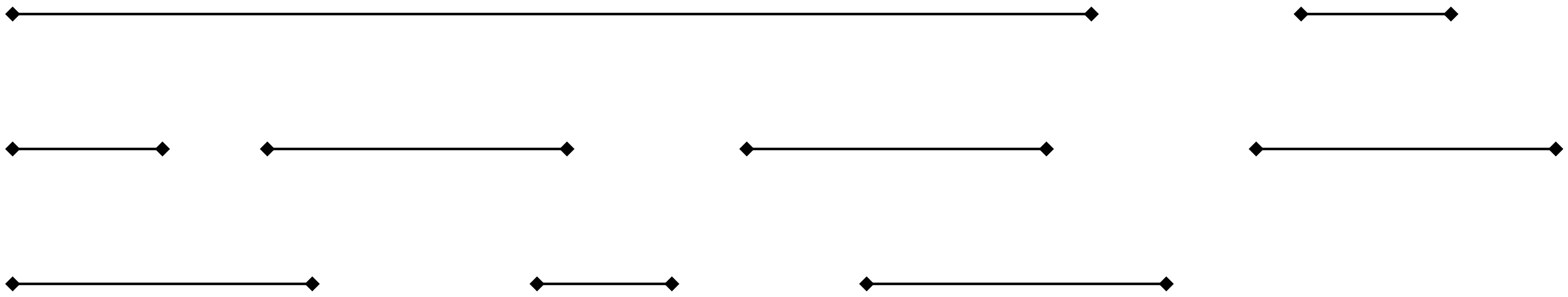
- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
- Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
- Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.
- Make the greedy choice and **solve top-down**.
- May have to preprocess input to put it into greedy order.
  - Example: Sorting activities by finish time.

Elements of Greedy Algorithms

- Greedy-choice Property.
  - A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- Optimal Substructure.

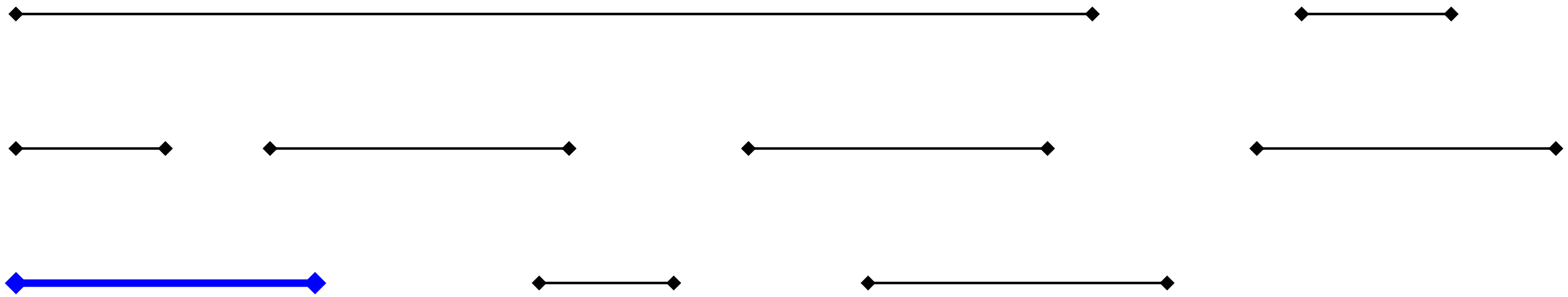
Interval scheduling

Given n activities $A = [a_1, a_2, .., a_n]$ where each activity has start time s_i and a finish time f_i. Schedule as many as possible of these activities such that they don't conflict.



Interval scheduling

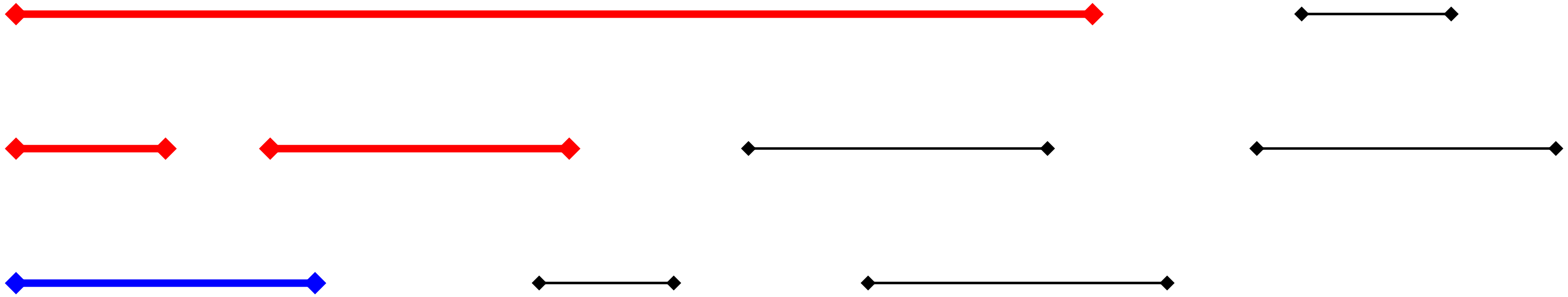
Given *n* activities A = [a_1,a_2, .., a_n] where each activity has start time s_i and a finish time f_i. Schedule as many as possible of these activities such that they don't conflict.



Which activities conflict?

Interval scheduling

Given *n* activities $A = [a_1, a_2, .., a_n]$ where each activity has start time s_i and a finish time f_i. Schedule as many as possible of these activities such that they don't conflict.



Which activities conflict?

Simple recursive solution

Enumerate all possible solutions and find which schedules the most activities

```
IntervalSchedule-Recursive(A)
if A = {}
        return 0
else
        max = −∞
        for all a ∈ A
                A′ ← A minus a and all conflicting activites with a
                s = IntervalSchedule-Recursive(A′)
                if s > max
                        max = s
        return 1 + max
```

Simple recursive solution

Is it correct?

- max{all possible solutions}

Running time?

- O(n!)

```
IntervalSchedule-Recursive(A)
if A = {}
        return 0
else
        max = −∞
        for all a ∈ A
                A′ ← A minus a and all conflicting activites with a
                s = IntervalSchedule-Recursive(A′)
                if s > max
                        max = s
        return 1 + max
```

Optimal Substructure

- Assume activities are sorted by finishing times.
  - $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Suppose an optimal solution includes activity a_k.
  - This generates two subproblems.
  - Selecting from $a_1, \ldots, a_{k-1}$, activities compatible with one another, and that finish before a_k starts (compatible with a_k).
  - Selecting from $a_{k+1}, \ldots, a_n$, activities compatible with one another, and that start after a_k finishes.
  - The solutions to the two subproblems must be optimal.
    - Prove using the cut-and-paste approach.

Recursive Solution

- Let S_{ij} = subset of activities in S that start after a_i finishes and finish before a_j starts.
- Subproblems: Selecting maximum number of mutually compatible activities from S_{ij}.
- Let $c[i, j]$ = size of maximum-size subset of mutually compatible activities in S_{ij}.

Recursive Solution:

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \phi \\ \max\limits_{i<k<j}\{c[i,k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \phi \end{cases}$$

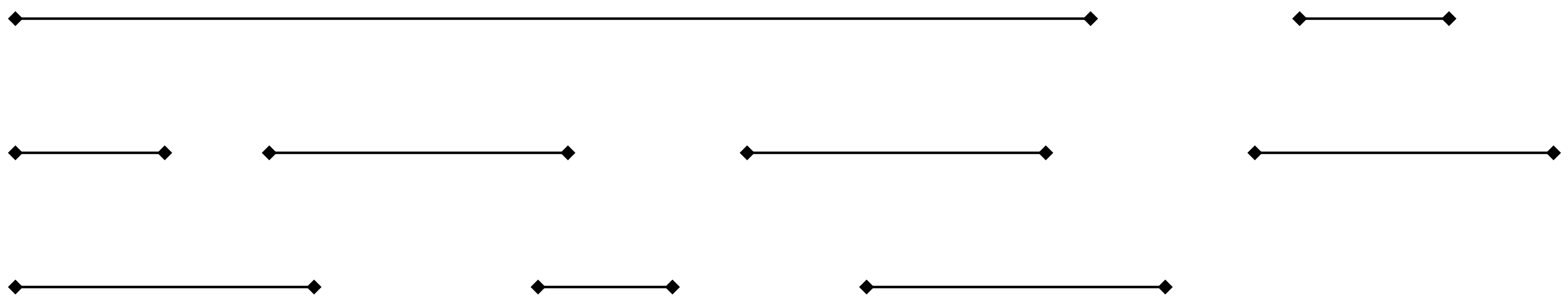
Can we do better?

Dynamic programming

- $O(n^2)$

Greedy solution – Is there a way to repeatedly make local decisions?

- Key: we'd still like to end up with the *optimal* solution



Overview of a greedy approach

Greedily pick an activity to schedule

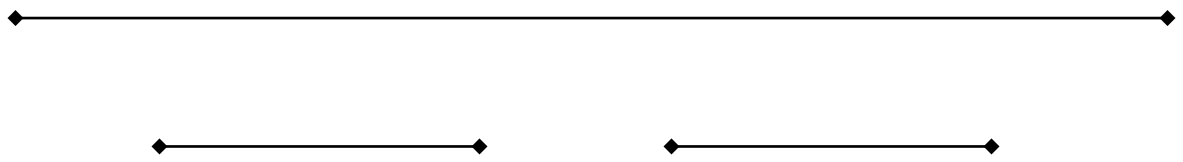
Add that activity to the answer

Remove that activity and all conflicting activities. Call this A'.

Repeat on A' until A' is empty

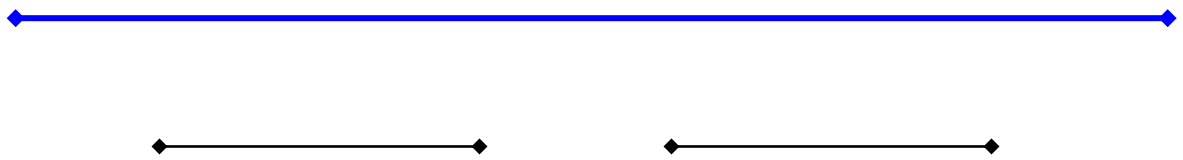
Greedy options

Select the activity that starts the earliest, i.e. argmin$\{s_1, s_2, s_3, \ldots, s_n\}$?



Greedy options

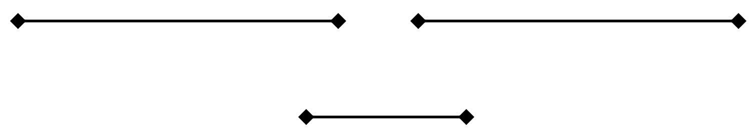
Select the activity that starts the earliest, i.e. $\text{argmin}\{s_1, s_2, s_3, \ldots, s_n\}$?



non-optimal

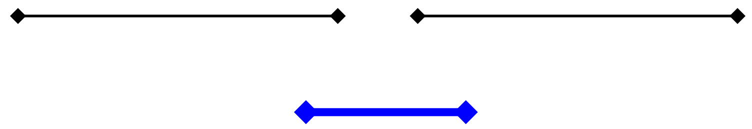
Greedy options

Select the shortest activity, i.e. argmin$\{f_1\text{-}s_1, f_2\text{-}s_2, f_3\text{-}s_3, \ldots, f_n\text{-}s_n\}$



Greedy options

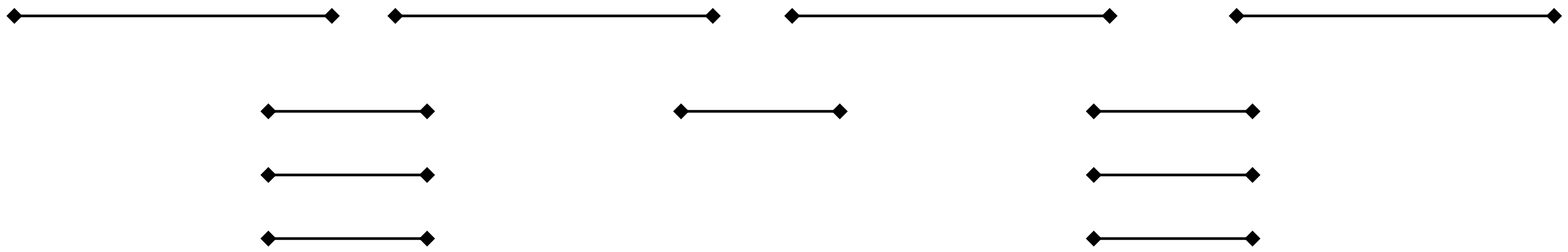
Select the shortest activity, i.e. $\mathrm{argmin}\{f_1\text{-}s_1, f_2\text{-}s_2, f_3\text{-}s_3, \ldots, f_n\text{-}s_n\}$



non-optimal

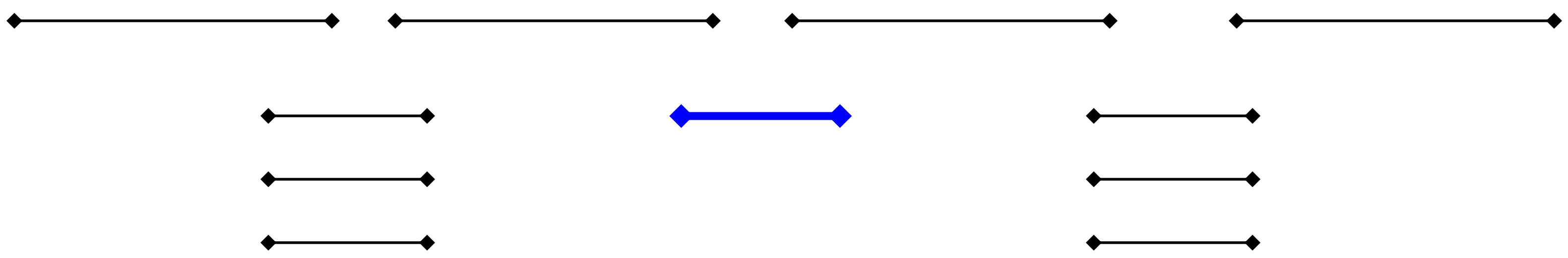
Greedy options

Select the activity with the smallest number of conflicts



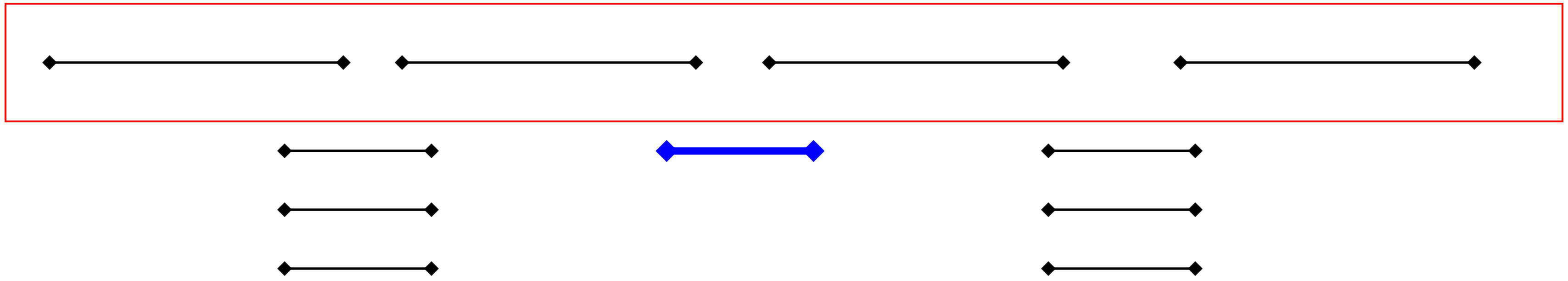
Greedy options

Select the activity with the smallest number of conflicts



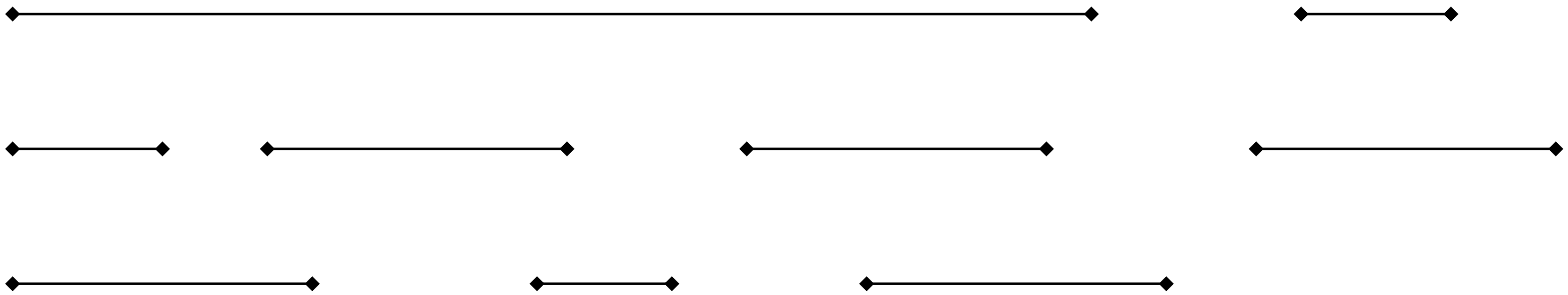
Greedy options

Select the activity with the smallest number of conflicts



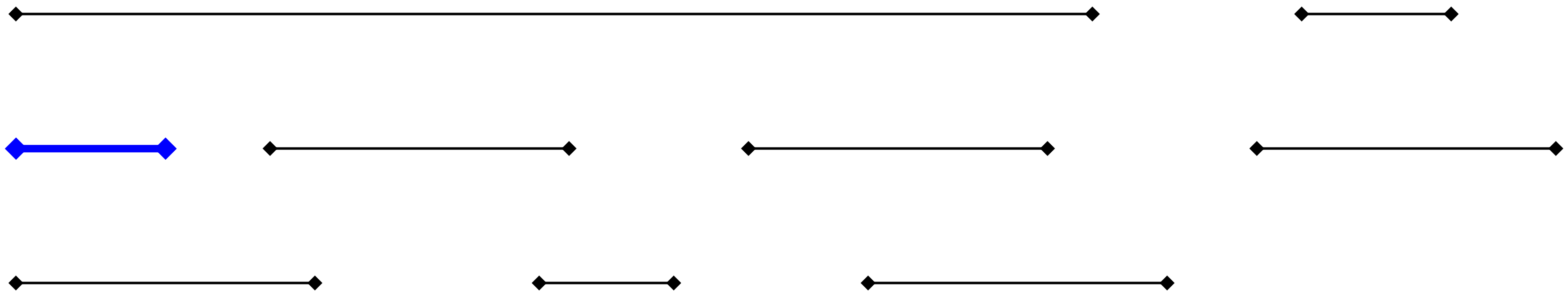
Greedy options

Select the activity that ends the earliest, i.e. argmin$\{f_1, f_2, f_3, \ldots, f_n\}$?



Greedy options

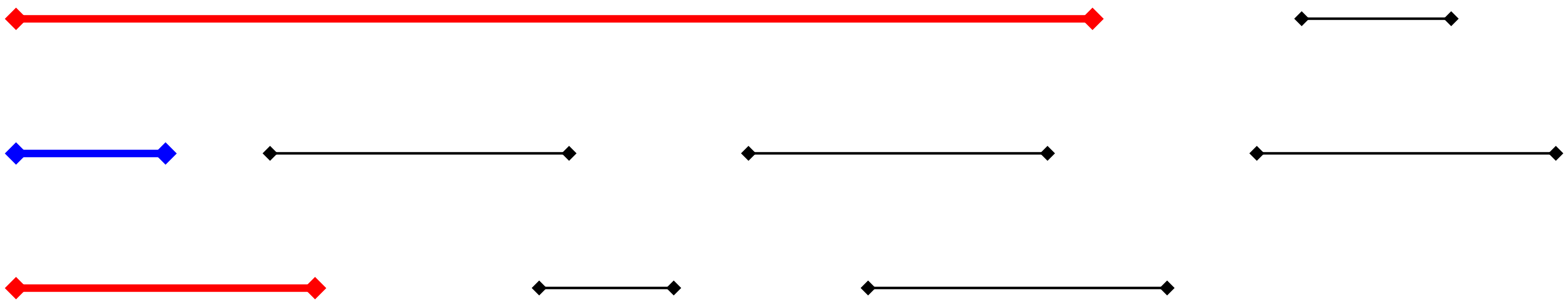
Select the activity that ends the earliest, i.e. argmin$\{f_1, f_2, f_3, \ldots, f_n\}$?



remove the conflicts

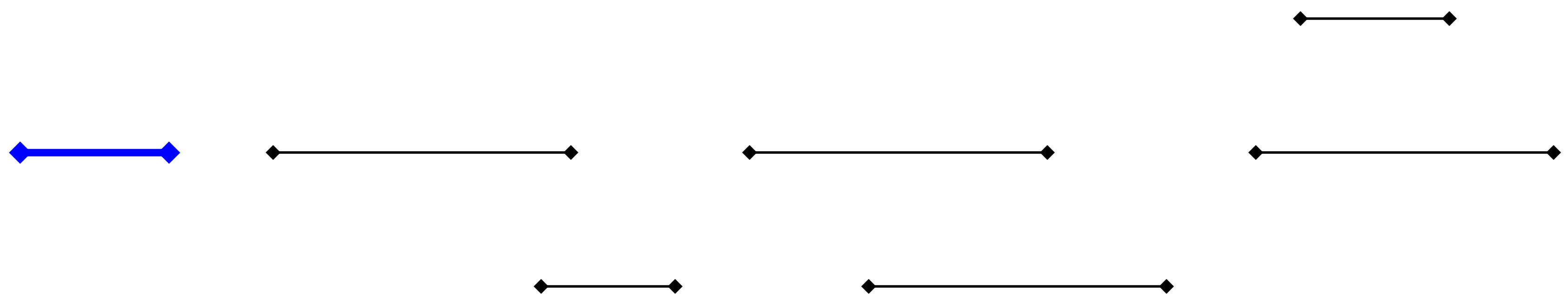
Greedy options

Select the activity that ends the earliest, i.e. argmin$\{f_1, f_2, f_3, \ldots, f_n\}$?



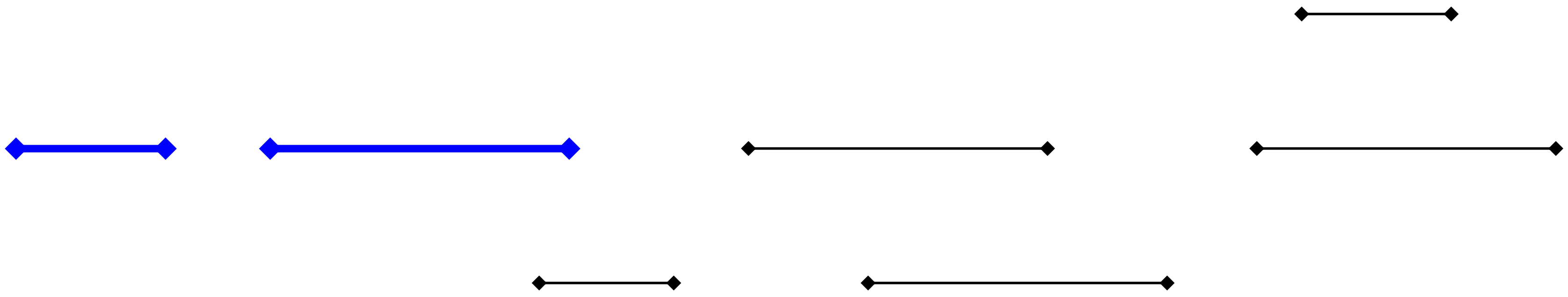
Greedy options

Select the activity that ends the earliest, i.e. argmin$\{f_1, f_2, f_3, \ldots, f_n\}$?



Greedy options

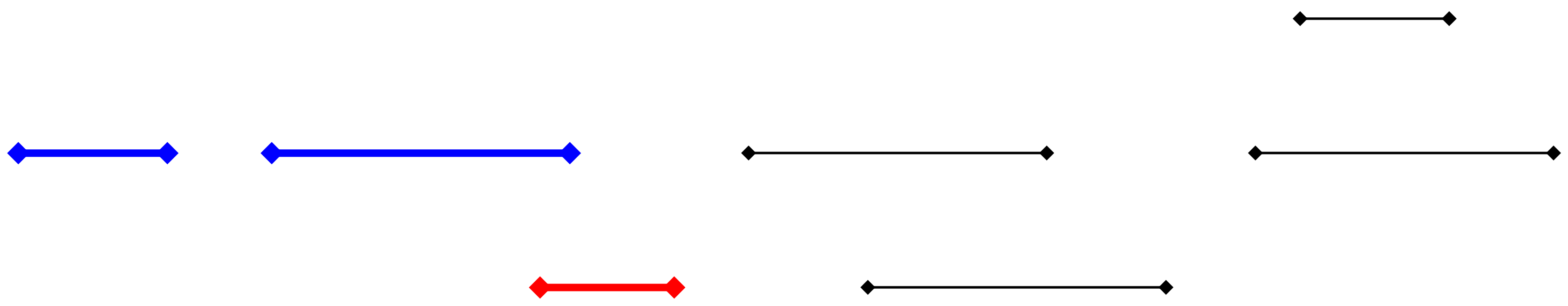
Select the activity that ends the earliest, i.e. argmin$\{f_1, f_2, f_3, \ldots, f_n\}$?



remove the conflicts

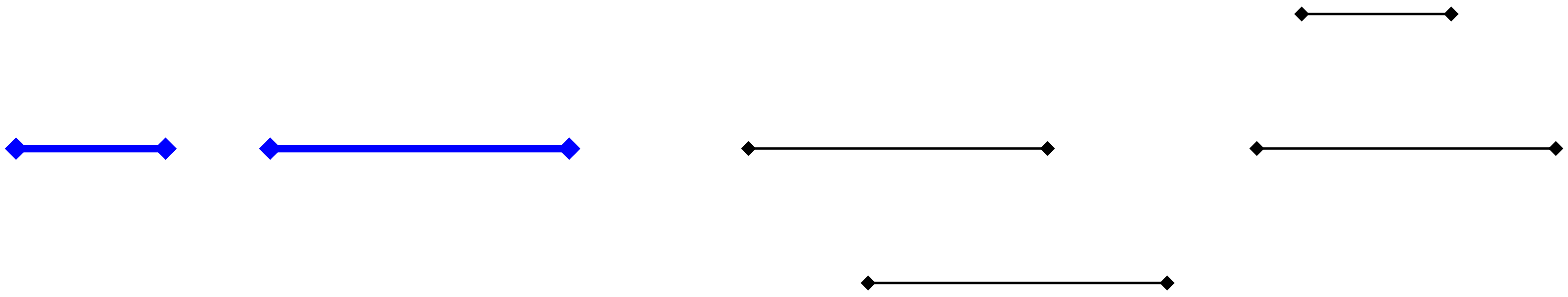
Greedy options

Select the activity that ends the earliest, i.e. argmin$\{f_1, f_2, f_3, \ldots, f_n\}$?



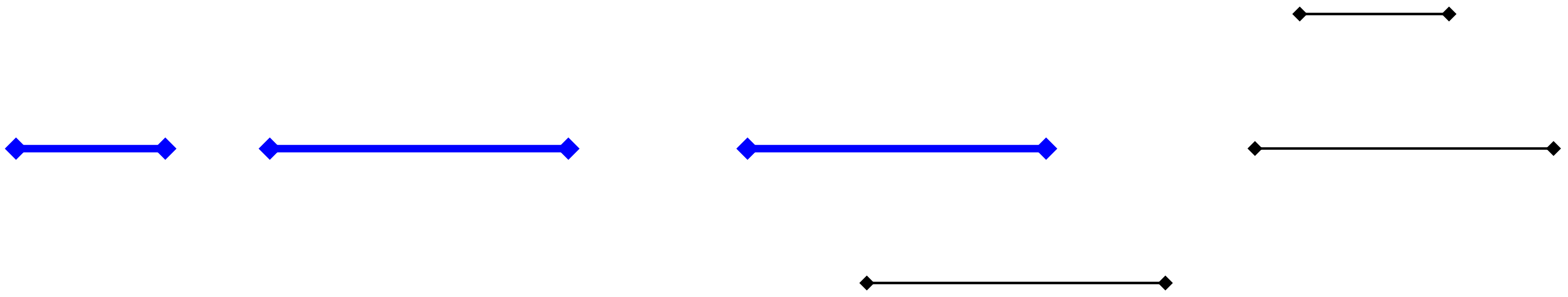
Greedy options

Select the activity that ends the earliest, i.e. argmin$\{f_1, f_2, f_3, \ldots, f_n\}$?



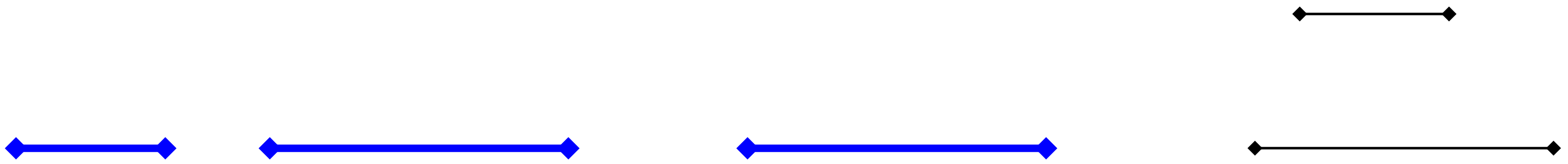
Greedy options

Select the activity that ends the earliest, i.e. $\text{argmin}\{f_1, f_2, f_3, \ldots, f_n\}$?



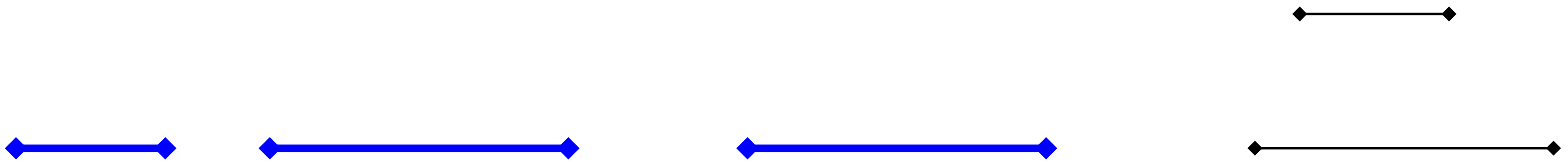
Greedy options

Select the activity that ends the earliest, i.e. argmin$\{f_1, f_2, f_3, \ldots, f_n\}$?



Greedy options

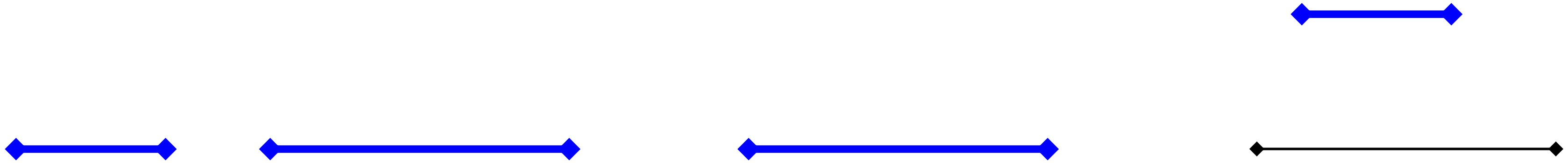
Select the activity that ends the earliest, i.e. argmin$\{f_1, f_2, f_3, \ldots, f_n\}$?



Multiple optimal solutions

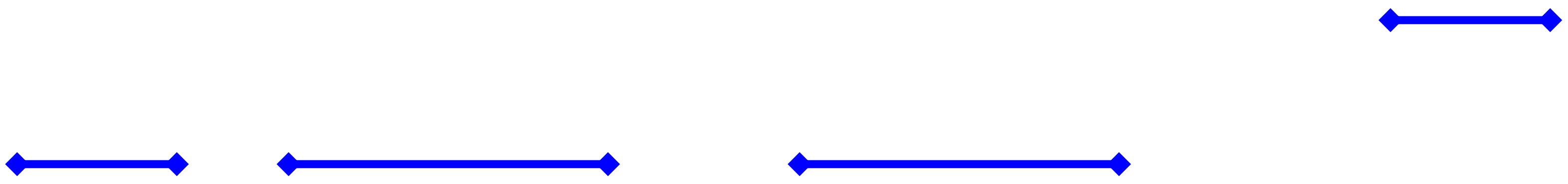
Greedy options

Select the activity that ends the earliest, i.e. $\mathrm{argmin}\{f_1, f_2, f_3, \ldots, f_n\}$?



Greedy options

Select the activity that ends the earliest, i.e. argmin$\{f_1, f_2, f_3, \ldots, f_n\}$?



Efficient greedy algorithm

Once you've identified a reasonable greedy heuristic:

- Prove that it always gives the correct answer
- Develop an efficient solution

Is our greedy approach correct?

“Stays ahead” argument:

show that no matter what other solution someone provides you, the solution provided by your algorithm always “stays ahead”, in that no other choice could do better

Is our greedy approach correct?

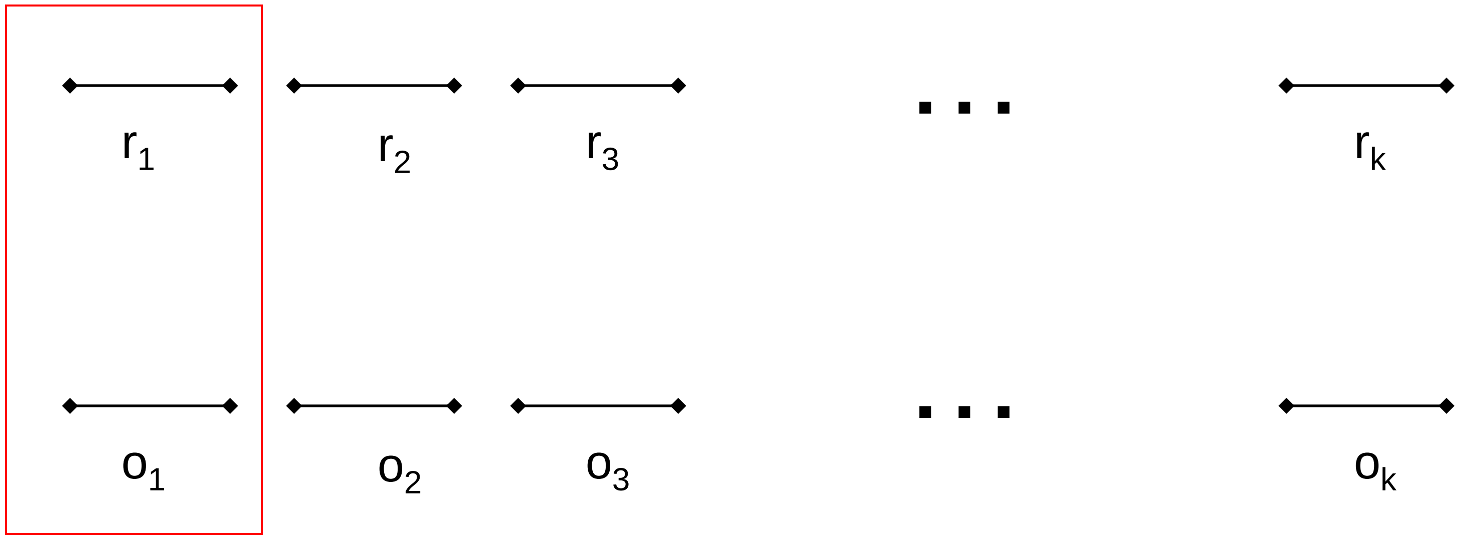
“Stays ahead” argument

Let $r_1, r_2, r_3, \ldots, r_k$ be the solution found by our approach

Let $o_1, o_2, o_3, \ldots, o_k$ of another optimal solution

Show our approach “stays ahead” of any other solution

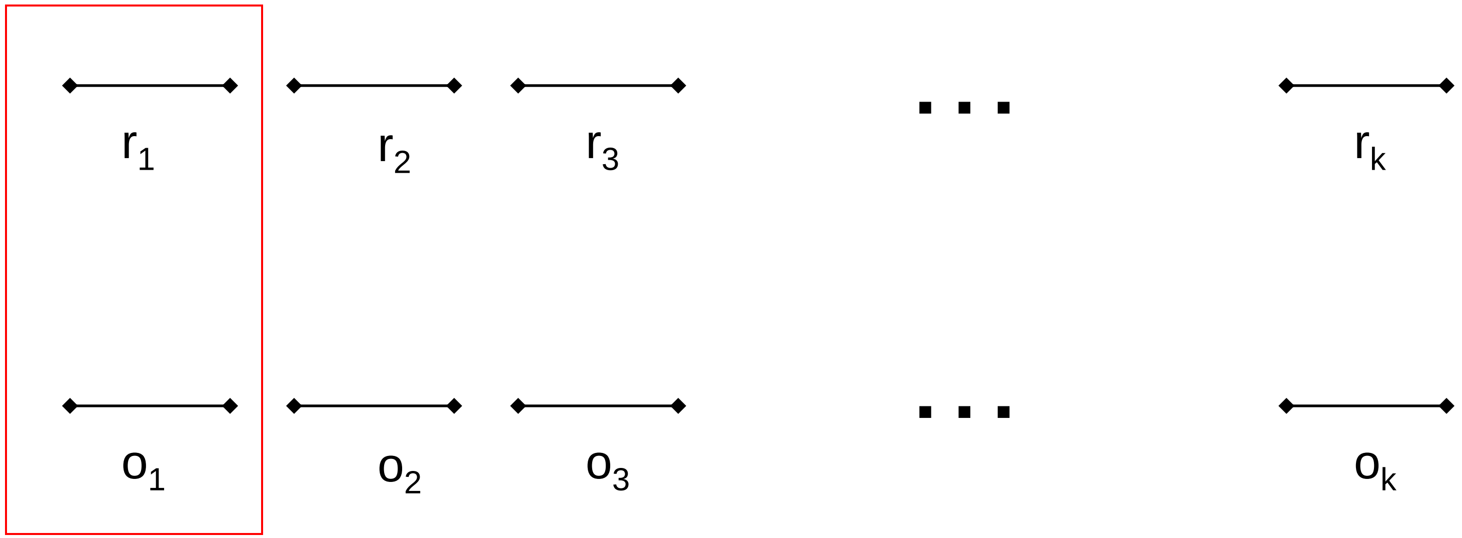
Stays ahead



Compare first activities of each solution

what do we know?

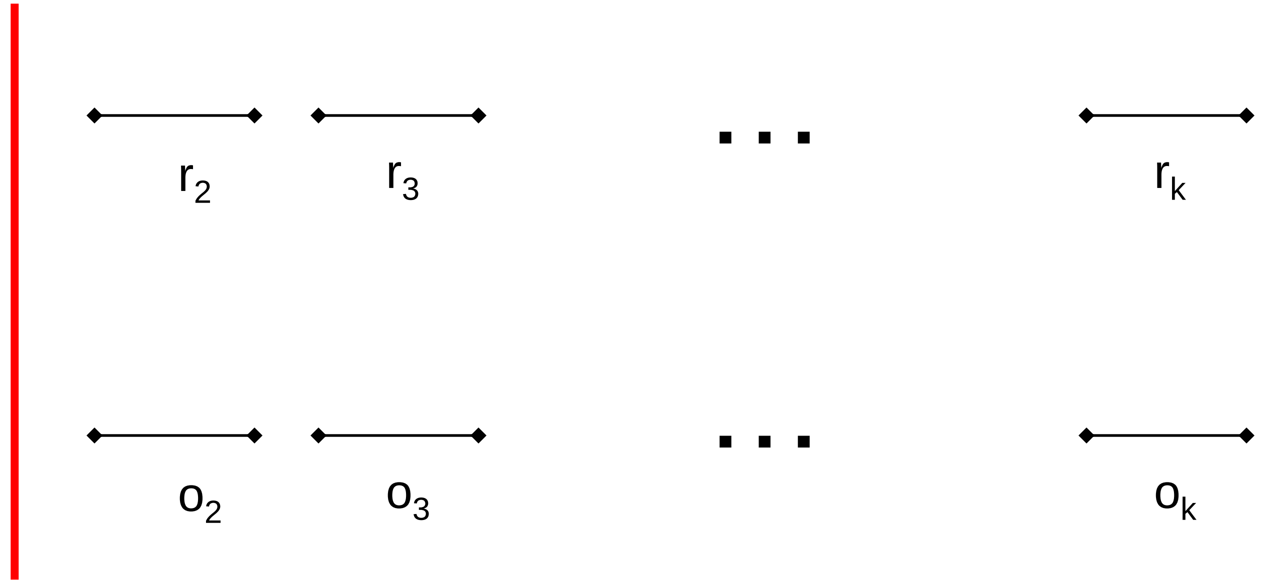
Stays ahead



finish(r_1) ≤ finish(o_1)

what does this imply?

Stays ahead



We have **at least** as much time as any other solution to schedule the remaining 2…k tasks

An efficient solution

IntervalSchedule-Greedy(A)

sort A based on finish times f_i
for $i \leftarrow 1$ to n
    add a_i to R
    $finish \leftarrow f_i$
    while $s_i < finish$
        $i \leftarrow i + 1$
return R

Running time?

INTERVALSCHEDULE-GREEDY(A)

sort A based on finish times f_i — $\Theta(n \log n)$
for $i \leftarrow 1$ to n
    add a_i to R
    $finish \leftarrow f_i$
    while $s_i < finish$
        $i \leftarrow i + 1$
return R

$\Theta(n)$

Overall: $\Theta(n \log n)$

Better than:

$O(n!)$
$O(n^2)$

Greedy-choice Property

- The problem also exhibits the greedy-choice property.
  - There is an optimal solution to the subproblem S_{ij}, that includes the activity with the smallest finish time in set S_{ij}.
  - Can be proved easily.
- Hence, there is an optimal solution to S that includes a_1.
- Therefore, make this greedy choice without solving subproblems first and evaluating them.
- Solve the subproblem that ensues as a result of making this greedy choice.
- Combine the greedy choice and the solution to the subproblem.

Recursive Algorithm

Recursive-Activity-Selector (s, f, i, j)

1. $m \leftarrow i+1$
2. **while** $m < j$ and $s_m < f_i$
3. **do** $m \leftarrow m+1$
4. **if** $m < j$
5. **then return** $\{a_m\} \cup$ Recursive-Activity-Selector(s, f, m, j)
6. else return ϕ

Initial Call: Recursive-Activity-Selector (s, f, 0, n+1)

Complexity: Θ(n) provided the activities are sorted by finishing times

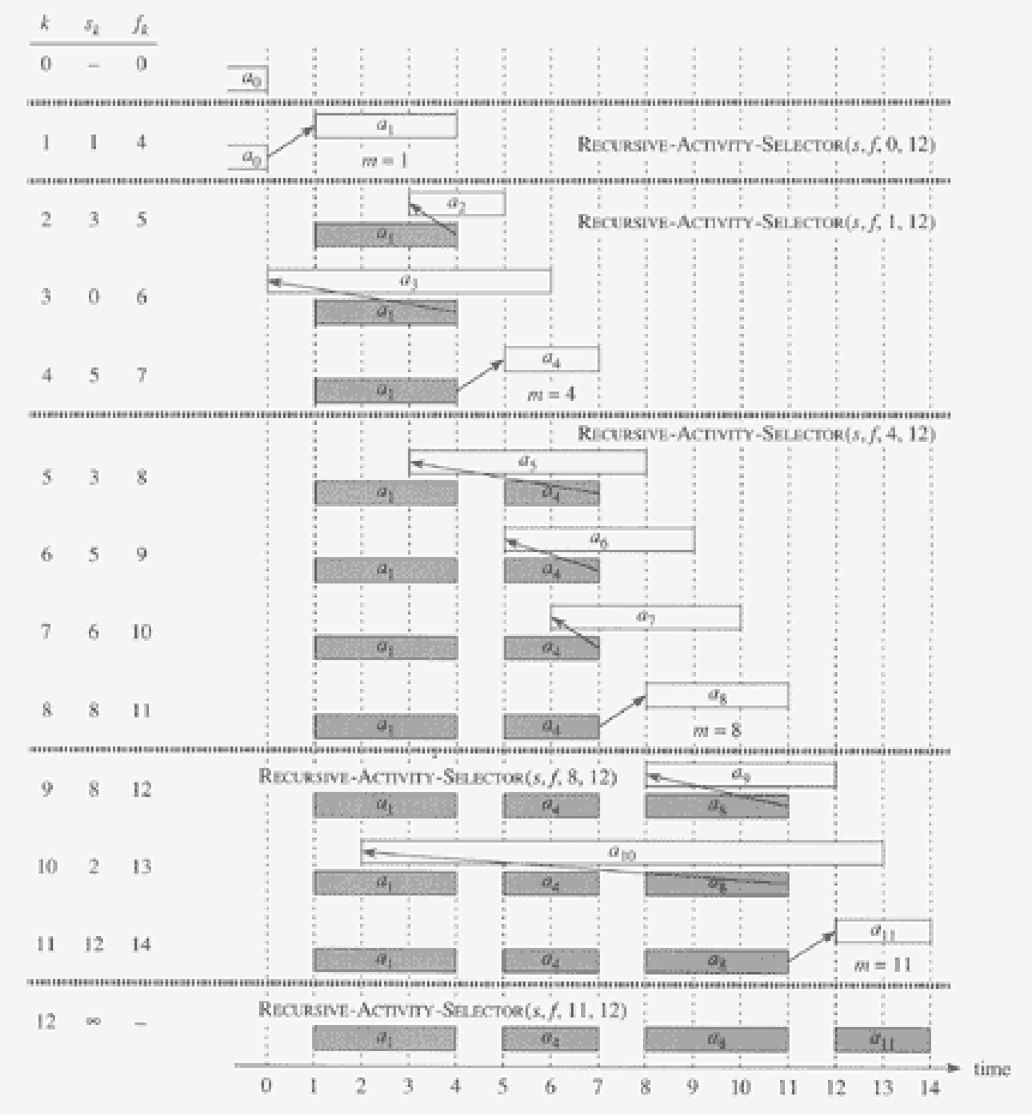
Recursive Algorithm

Given set $S = \{a_1, \ldots, a_n\}$ of activities and activity start and finish times, find the set of selected activities?

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	8	9	10	11	12	13	14

(Note: activities sorted in 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	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

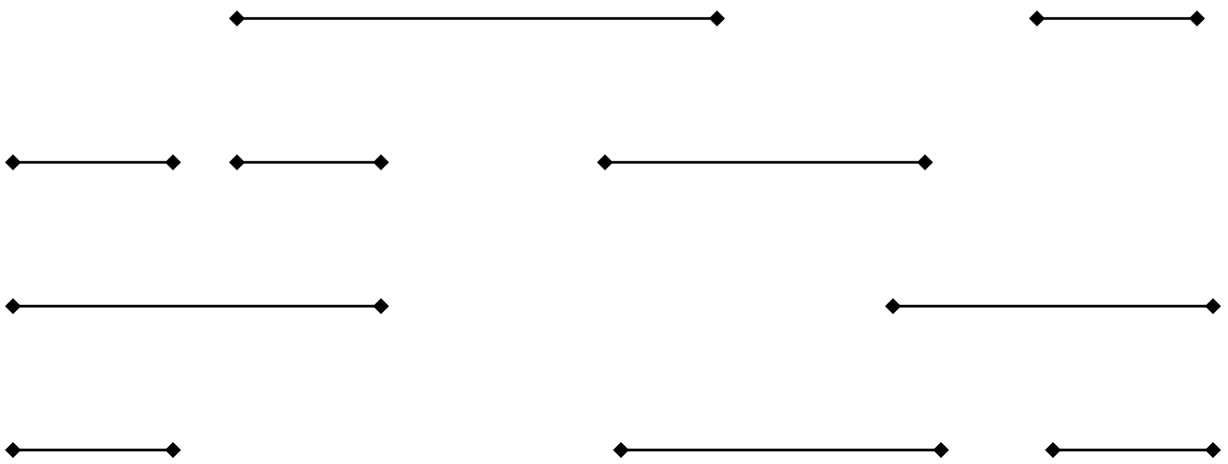


[a_1, a_4, a_8, a_{11}]

Scheduling *all* intervals

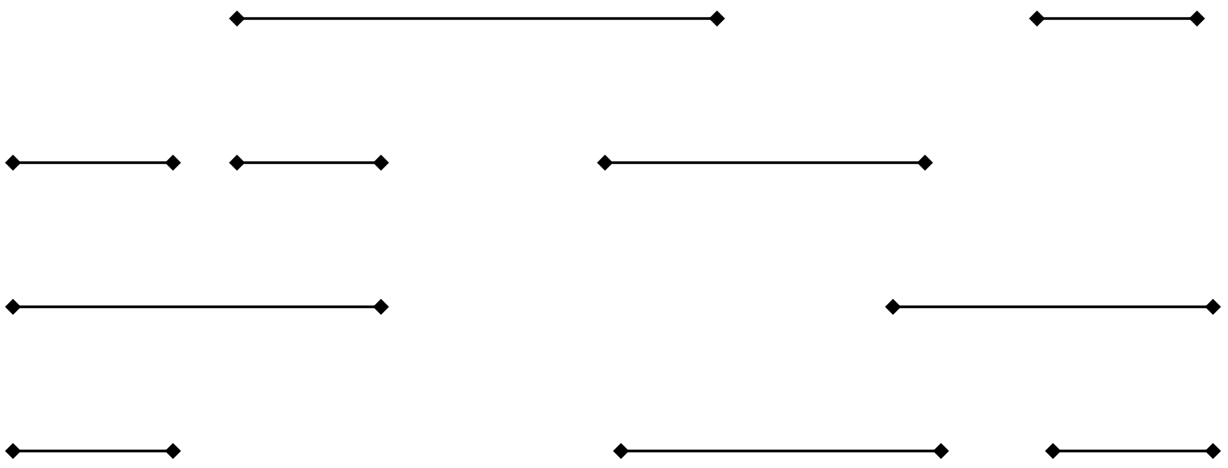
Given n activities, we need to schedule **all** activities.

Goal: minimize the number of resources required.

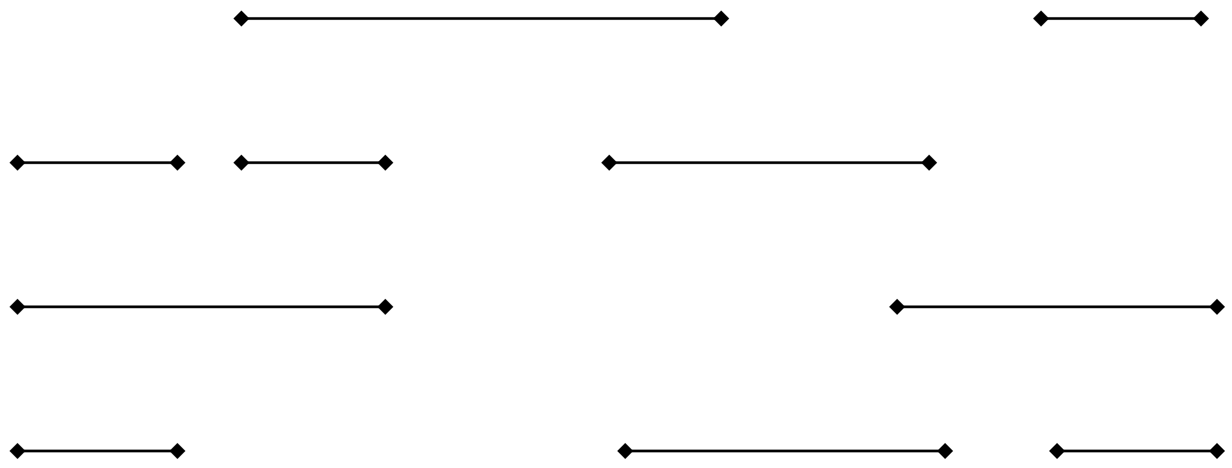


Greedy approach?

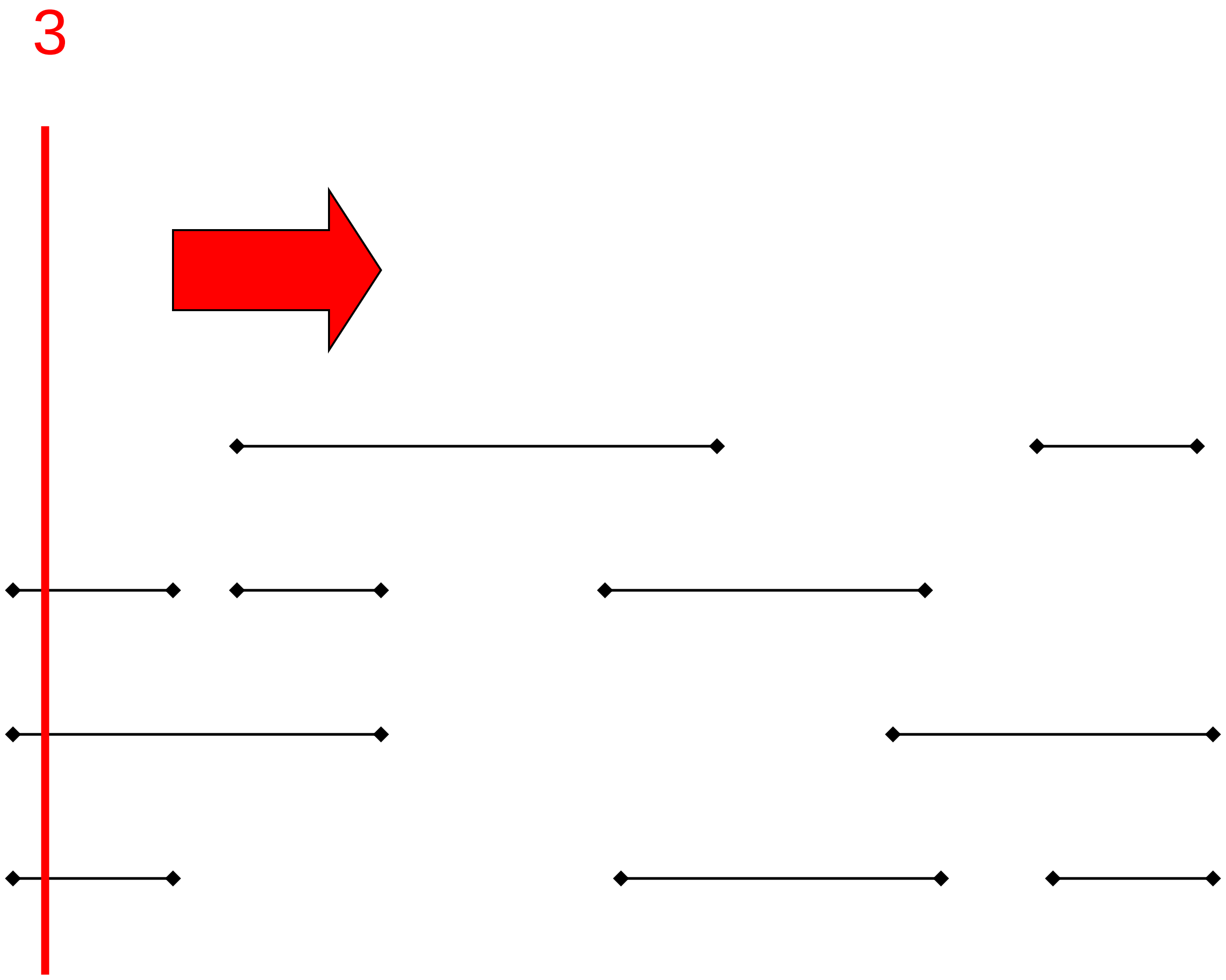
The best we could ever do is the maximum number of conflicts for any time period



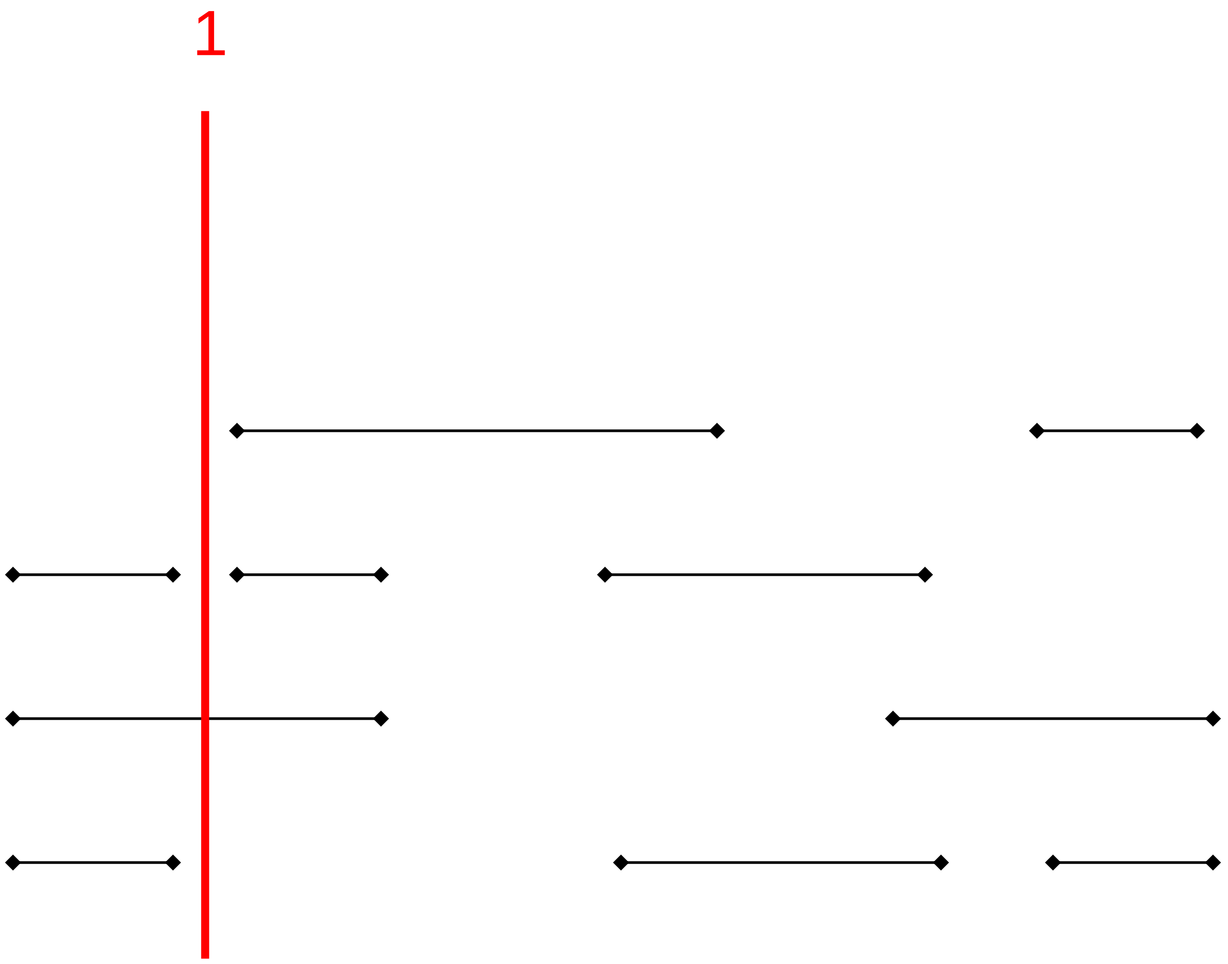
Calculating max conflicts efficiently



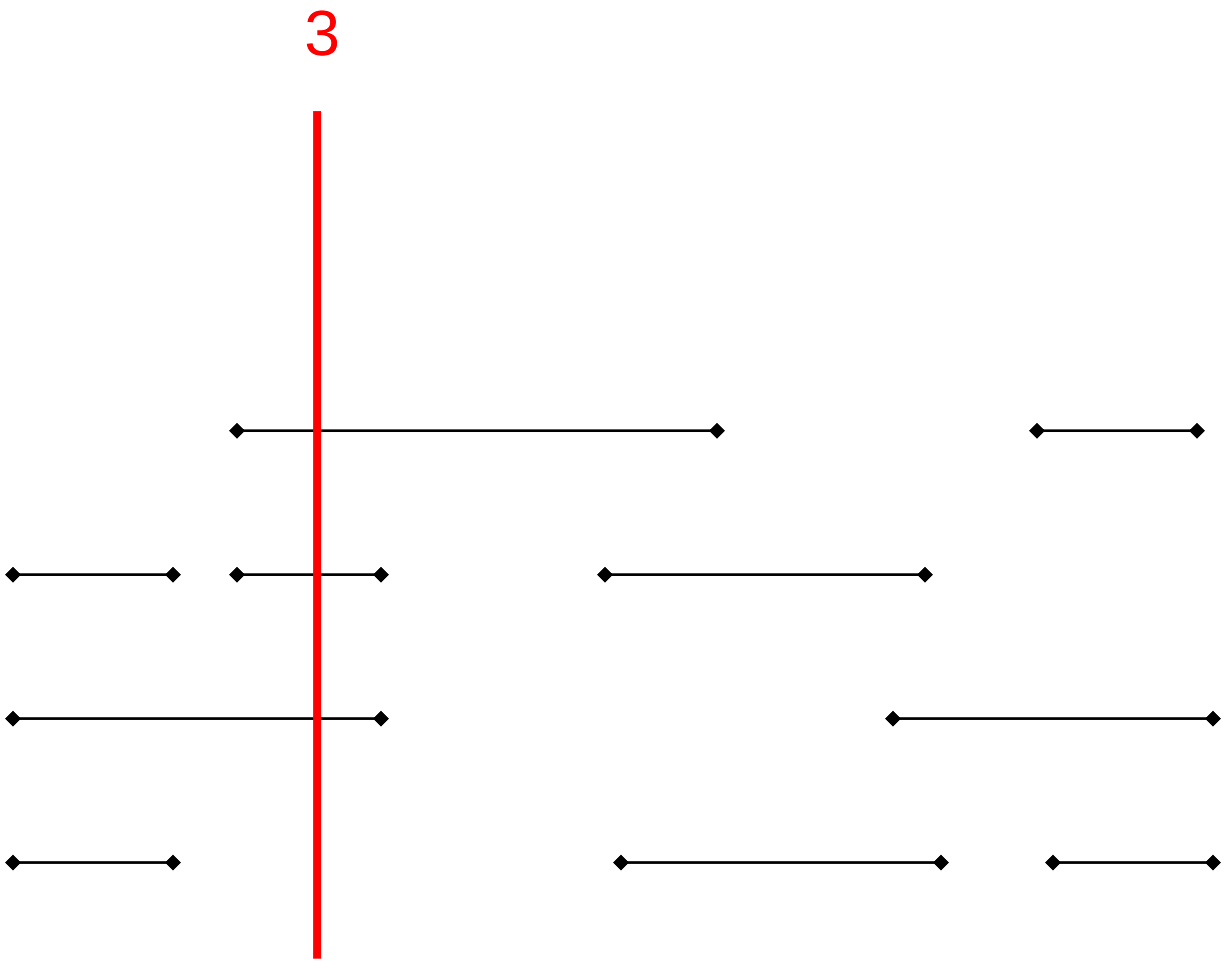
Calculating max conflicts efficiently



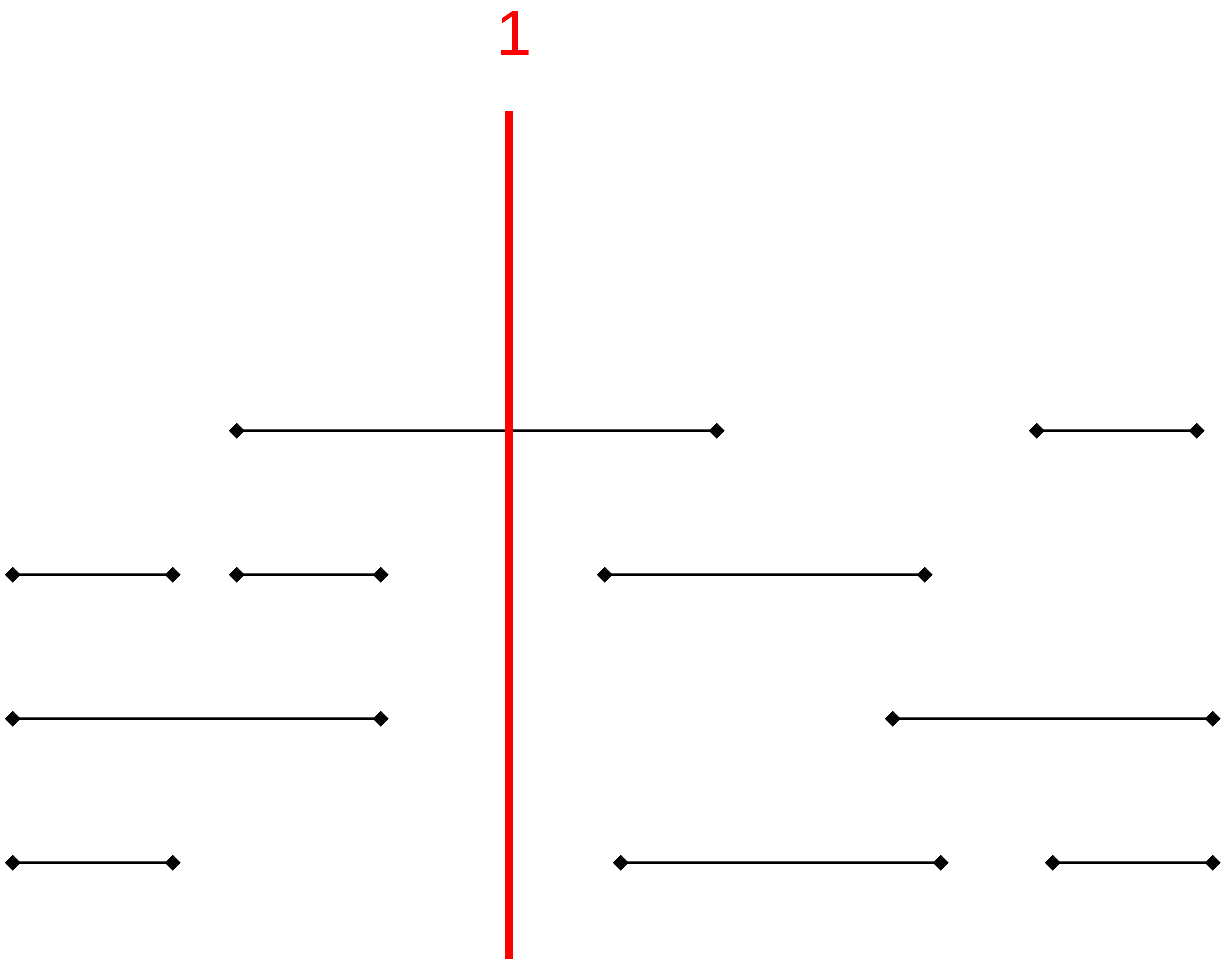
Calculating max conflicts efficiently



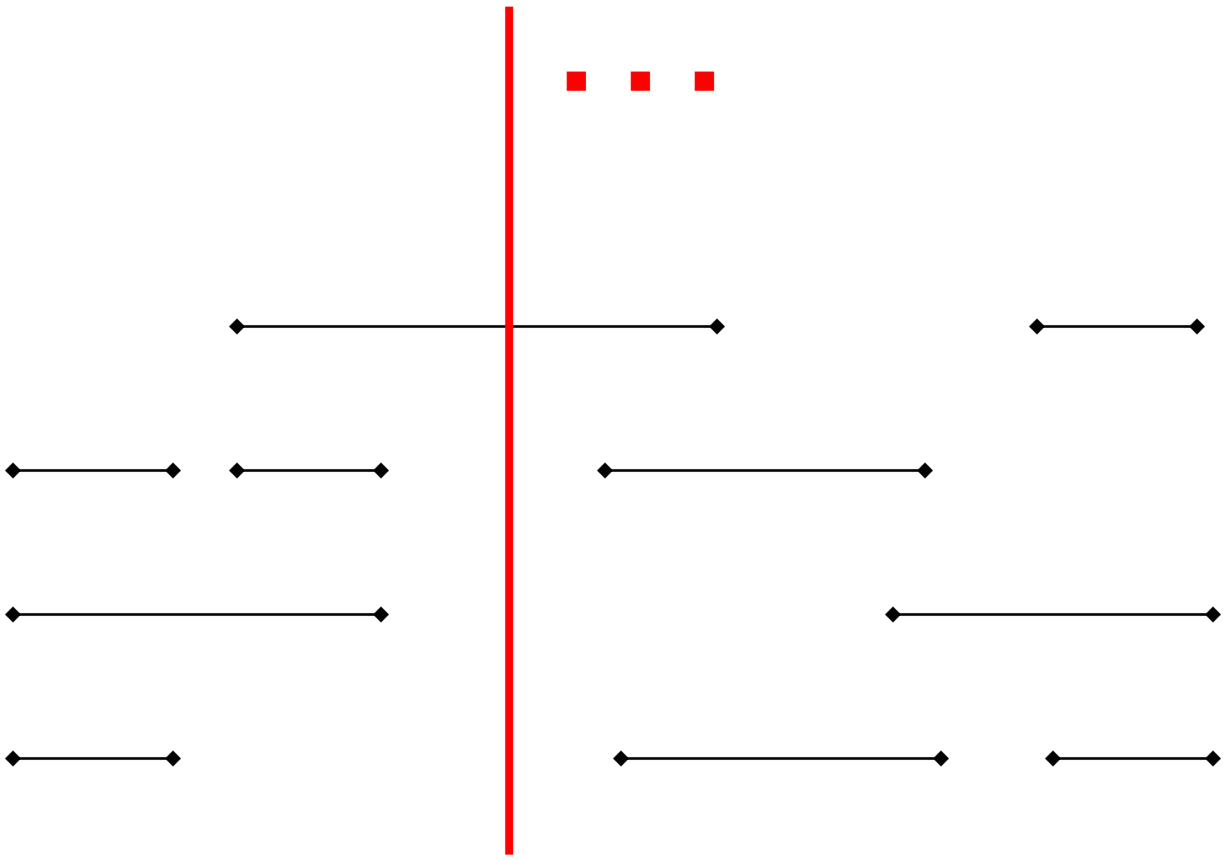
Calculating max conflicts efficiently



Calculating max conflicts efficiently



Calculating max conflicts efficiently



Calculating max conflicts

```
AllIntervalScheduleCount(A)
Sort the start and end times, call this X
current ← 0
max ← 0
for i ← 1 to length[X]
        if x_i is a start node
                current++
        else
                current--
        if current > max
                max ← current
return max
```

Correctness?

We can do no better than the max number of conflicts.
This exactly counts the max number of conflicts.

```
AllIntervalScheduleCount(A)
Sort the start and end times, call this X
current ← 0
max ← 0
for i ← 1 to length[X]
        if x_i is a start node
                current++
        else
                current--
        if current > max
                max ← current
return max
```

Runtime?

O(2n log 2n + n) = O(n log n)

AllIntervalScheduleCount(A)

Sort the start and end times, call this X
$current \leftarrow 0$
$max \leftarrow 0$
for $i \leftarrow 1$ **to** $length[X]$
    if x_i is a start node
        $current{++}$
    else
        $current{--}$
    if $current > max$
        $max \leftarrow current$
return max

Horn formulas

Horn formulas are a particular form of boolean logic formulas

They are one approach to allow a program to do logical reasoning

Boolean variables: represent some event

- x = the murder took place in the kitchen
- y = the butler is innocent
- z = the colonel was asleep at 8 pm

Implications

Left-hand side is an AND of any number of positive literals

Right-hand side is a single literal

$$z \wedge y \Rightarrow x$$

x = the murder took place in the kitchen
y = the butler is innocent
z = the colonel was asleep at 8 pm

What does this implication mean in English?

Implications

Left-hand side is an AND of any number of positive literals

Right-hand side is a single literal

$$z \wedge y \Rightarrow x$$

If the colonel was asleep at 8 pm **and** the butler is innocent **then** the murder took place in the kitchen

x = the murder took place in the kitchen
y = the butler is innocent
z = the colonel was asleep at 8 pm

Implications

Left-hand side is an AND of any number of positive literals

Right-hand side is a single literal

$$\Rightarrow x$$

x = the murder took place in the kitchen
y = the butler is innocent
z = the colonel was asleep at 8 pm

What does this implication mean in English?

Implications

Left-hand side is an AND of any number of positive literals

Right-hand side is a single literal

$$\Rightarrow x$$

the murder took place in the kitchen

x = the murder took place in the kitchen
y = the butler is innocent
z = the colonel was asleep at 8 pm

Negative clauses

An OR of any number of negative literals

$$\bar{u} \vee \bar{t} \vee \bar{y}$$

u = the constable is innocent
t = the colonel is innocent
y = the butler is innocent

What does this clause mean in English?

Negative clauses

An OR of any number of negative literals

$$\bar{u} \vee \bar{t} \vee \bar{y}$$

not every one is innocent

u = the constable is innocent
t = the colonel is innocent
y = the butler is innocent

Horn formula

A horn formula is a set of implications and negative clauses:

$$\Rightarrow x \qquad\qquad x \wedge u \Rightarrow z$$

$$\Rightarrow y \qquad\qquad \bar{x} \vee \bar{y} \vee \bar{z}$$

Goal

Given a horn formula, determine if the formula is satisfiable, i.e. an assignment of true/false to the variables that is consistent with all of the implications/causes

$$\Rightarrow x \qquad\qquad x \wedge u \Rightarrow z$$

$$\Rightarrow y \qquad\qquad \bar{x} \vee \bar{y} \vee \bar{z}$$

u	x	y	z
0	1	1	0

Goal

Given a horn formula, determine if the formula is satisfiable, i.e. an assignment of true/false to the variables that is consistent with all of the implications/causes

$$\Rightarrow x \qquad\qquad x \wedge y \Rightarrow z$$

$$\Rightarrow y \qquad\qquad \bar{x} \vee \bar{y} \vee \bar{z}$$

u x y z

not satifiable

Goal

Given a horn formula, determine if the formula is satisfiable, i.e. an assignment of true/false to the variables that is consistent with all of the implications/causes

$$\Rightarrow x \qquad x \wedge z \Rightarrow w \qquad w \wedge y \wedge z \Rightarrow x$$

$$x \Rightarrow y \qquad x \wedge y \Rightarrow w \qquad \overline{w} \vee \overline{x} \vee \overline{y}$$

?

Goal

Given a horn formula, determine if the formula is satisfiable, i.e. an assignment of true/false to the variables that is consistent with all of the implications/causes

$$x \wedge u \Rightarrow z$$

what do each of these encourage in the solution?

$$\bar{x} \vee \bar{y} \vee \bar{z}$$

Goal

Given a horn formula, determine if the formula is satisfiable, i.e. an assignment of true/false to the variables that is consistent with all of the implications/causes

$$x \wedge u \Rightarrow z$$

implications tell us to set some variables to true

$$\bar{x} \vee \bar{y} \vee \bar{z}$$

negative clauses encourage us make them false

A brute force solution

Try each setting of the boolean variables and see if any of them satisfy the formula

For n variables, how many settings are there?

- 2^n

A greedy solution?

$$\Rightarrow x \qquad x \wedge z \Rightarrow w \qquad w \wedge y \wedge z \Rightarrow x$$

$$x \Rightarrow y \qquad x \wedge y \Rightarrow w \qquad \overline{w} \vee \overline{x} \vee \overline{y}$$

w	0
x	0
y	0
z	0

A greedy solution?

$$\boxed{\Rightarrow x} \qquad x \wedge z \Rightarrow w \qquad w \wedge y \wedge z \Rightarrow x$$

$$x \Rightarrow y \qquad x \wedge y \Rightarrow w \qquad \overline{w} \vee \overline{x} \vee \overline{y}$$

w	0
x	1
y	0
z	0

A greedy solution?

$$\Rightarrow x \qquad x \wedge z \Rightarrow w \qquad w \wedge y \wedge z \Rightarrow x$$

$$\boxed{x \Rightarrow y} \qquad x \wedge y \Rightarrow w \qquad \overline{w} \vee \overline{x} \vee \overline{y}$$

w	0
x	1
y	1
z	0

A greedy solution?

$$\Rightarrow x \qquad x \wedge z \Rightarrow w \qquad w \wedge y \wedge z \Rightarrow x$$

$$x \Rightarrow y \qquad \boxed{x \wedge y \Rightarrow w} \qquad \overline{w} \vee \overline{x} \vee \overline{y}$$

w	1
x	1
y	1
z	0

A greedy solution?

$$\Rightarrow x \qquad x \wedge z \Rightarrow w \qquad w \wedge y \wedge z \Rightarrow x$$

$$x \Rightarrow y \qquad x \wedge y \Rightarrow w \qquad \boxed{\overline{w} \vee \overline{x} \vee \overline{y}}$$

w	1
x	1
y	1
z	0

not satisfiable

A greedy solution

```
HORN(H)
set all variables to false
for all implications i
        if EMPTY(LHS(i))
                RHS(i) ← true
changed ← true
while changed
        changed ← false
        for all implications i
                if LHS(i) = true and !RHS(i) = true
                        RHS(i) ← true
                        changed = true
for all negative clauses c
        if c = false
                return false
return true
```

A greedy solution

```
HORN(H)
set all variables to false
for all implications i
        if EMPTY(LHS(i))
                RHS(i) ← true
changed ← true
while changed
        changed ← false
        for all implications i
                if LHS(i) = true and !RHS(i) = true
                        RHS(i) ← true
                        changed = true
for all negative clauses c
        if c = false
                return false
return true
```

set all variables of the implications of the form "⇒x" to true

A greedy solution

```
HORN(H)
set all variables to false
for all implications i
        if EMPTY(LHS(i))
                RHS(i) ← true
changed ← true
while changed
        changed ← false
        for all implications i
                if LHS(i) = true and !RHS(i) = true
                        RHS(i) ← true
                        changed = true
for all negative clauses c
        if c = false
                return false
return true
```

if the all variables of the LHS of an implication are true, then set the RHS variable to true

A greedy solution

```
Horn(H)
set all variables to false
for all implications i
        if Empty(LHS(i))
                RHS(i) ← true
changed ← true
while changed
        changed ← false
        for all implications i
                if LHS(i) = true and !RHS(i) = true
                        RHS(i) ← true
                        changed = true
for all negative clauses c
        if c = false
                return false
return true
```

see if all of the negative clauses are satisfied

Correctness of greedy solution

Two parts:

- If our algorithm returns an assignment, is it a valid assignment?
- If our algorithm does not return an assignment, does an assignment exist?

Correctness of greedy solution

If our algorithm returns an assignment, is it a valid assignment?

```
Horn(H)
set all variables to false
for all implications i
        if Empty(LHS(i))
                RHS(i) ← true
changed ← true
while changed
        changed ← false
        for all implications i
                if LHS(i) = true and !RHS(i) = true
                        RHS(i) ← true
                        changed = true
for all negative clauses c
        if c = false
                return false
return true
```

Correctness of greedy solution

If our algorithm returns an assignment, is it a valid assignment?

```
Horn(H)
set all variables to false
for all implications i
        if Empty(LHS(i))
                RHS(i) ← true
changed ← true
while changed
        changed ← false
        for all implications i
                if LHS(i) = true and !RHS(i) = true
                        RHS(i) ← true
                        changed = true
for all negative clauses c
        if c = false
                return false
return true
```

explicitly check all negative clauses

Correctness of greedy solution

If our algorithm returns an assignment, is it a valid assignment?

```
Horn(H)
set all variables to false
for all implications i
        if Empty(LHS(i))
                RHS(i) ← true
changed ← true
while changed
        changed ← false
        for all implications i
                if LHS(i) = true and !RHS(i) = true
                        RHS(i) ← true
                        changed = true
for all negative clauses c
        if c = false
                return false
return true
```

don't stop until all implications with all LHS elements true have RHS true

Correctness of greedy solution

If our algorithm does not return an assignment, does an assignment exist?

```
Horn(H)
set all variables to false
for all implications i
        if Empty(LHS(i))
                RHS(i) ← true
changed ← true
while changed
        changed ← false
        for all implications i
                if LHS(i) = true and !RHS(i) = true
                        RHS(i) ← true
                        changed = true
for all negative clauses c
        if c = false
                return false
return true
```

Our algorithm is "stingy". It only sets those variables that **have** to be true. All others remain false.

Running time?

```
HORN(H)
set all variables to false
for all implications i
        if EMPTY(LHS(i))
                RHS(i) ← true
changed ← true
while changed
        changed ← false
        for all implications i
                if LHS(i) = true and !RHS(i) = true
                        RHS(i) ← true
                        changed = true
for all negative clauses c
        if c = false
                return false
return true
```

?

n = number of variables

m = number of formulas

Running time?

```
HORN(H)
set all variables to false
for all implications i
        if EMPTY(LHS(i))
                RHS(i) ← true
changed ← true
while changed
        changed ← false
        for all implications i
                if LHS(i) = true and !RHS(i) = true
                        RHS(i) ← true
                        changed = true
for all negative clauses c
        if c = false
                return false
return true
```

O(nm)

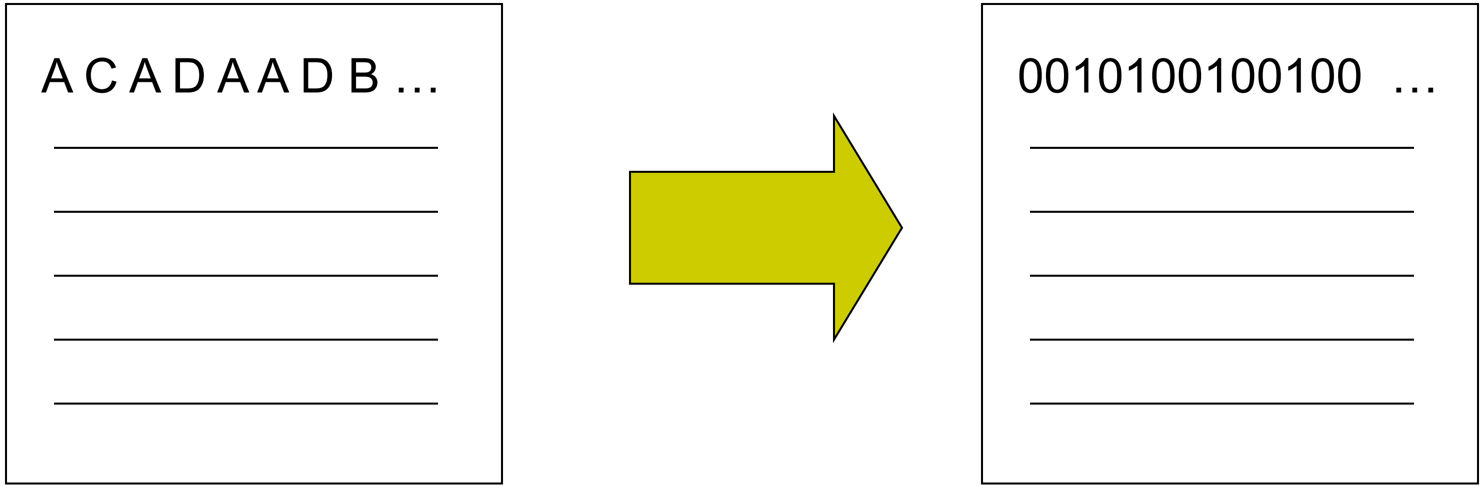
n = number of variables

m = number of formulas

Data compression

Given a file containing some data of a fixed alphabet Σ (e.g. A, B, C, D), we would like to pick a binary character code that minimizes the number of bits required to represent the data.

minimize the size of the encoded file



Compression algorithms

General purpose [edit]

- Run-length encoding (RLE) – a simple scheme that provides good compression of data containing lots of runs of the same value.
- Lempel-Ziv 1978 (LZ78), Lempel-Ziv-Welch (LZW) – used by GIF images and compress among many other applications
- DEFLATE – used by gzip, ZIP (since version 2.0), and as part of the compression process of Portable Network Graphics (PNG), Point-to-Point Protocol (PPP), HTTP, SSH
- bzip2 – using the Burrows–Wheeler transform, this provides slower but higher compression than DEFLATE
- Lempel–Ziv–Markov chain algorithm (LZMA) – used by 7zip, xz, and other programs; higher compression than bzip2 as well as much faster decompression.
- Lempel–Ziv–Oberhumer (LZO) – designed for compression/decompression speed at the expense of compression ratios
- Statistical Lempel Ziv – a combination of statistical method and dictionary-based method; better compression ratio than using single method.

Audio [edit]

- Free Lossless Audio Codec – FLAC
- Apple Lossless – ALAC (Apple Lossless Audio Codec)
- apt-X – Lossless
- Adaptive Transform Acoustic Coding – ATRAC
- Audio Lossless Coding – also known as MPEG-4 ALS
- MPEG-4 SLS – also known as HD-AAC
- Direct Stream Transfer – DST
- Dolby TrueHD
- DTS-HD Master Audio
- Meridian Lossless Packing – MLP
- Monkey's Audio – Monkey's Audio APE
- OptimFROG
- Original Sound Quality – OSQ
- RealPlayer – RealAudio Lossless
- Shorten – SHN
- TTA – True Audio Lossless
- WavPack – WavPack lossless
- WMA Lossless – Windows Media Lossless

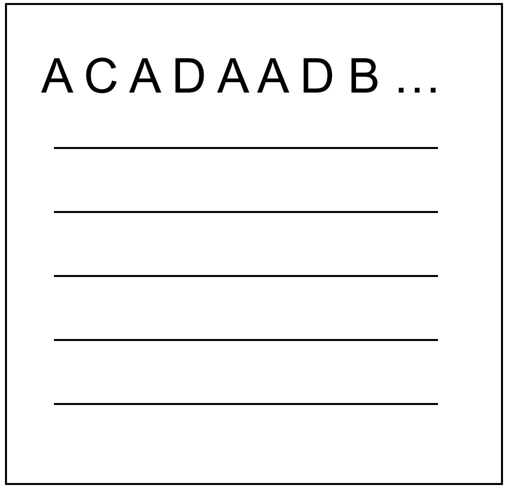
Graphics [edit]

- ILBM – (lossless RLE compression of Amiga IFF images)
- JBIG2 – (lossless or lossy compression of B&W images)
- JPEG-LS – (lossless/near-lossless compression standard)
- JPEG 2000 – (includes lossless compression method, as proven by Sunil Kumar, Prof San Diego State University)
- JPEG XR – formerly *WMPhoto* and *HD Photo*, includes a lossless compression method
- PGF – Progressive Graphics File (lossless or lossy compression)
- PNG – Portable Network Graphics
- TIFF – Tagged Image File Format
- Gifsicle (GPL) – Optimize gif files
- Jpegoptim (GPL) – Optimize jpeg files

http://en.wikipedia.org/wiki/Lossless_data_compression

Simplifying assumption: frequency only

Assume that we only have character frequency information for a file



=

Symbol	Frequency
A	40
B	3
C	20
D	37

Fixed length code

Use ceil($\log_2|\Sigma|$) bits for each character

A =

B =

C =

D =

Fixed length code

Use ceil($\log_2|\Sigma|$) bits for each character

A = 00
B = 01
C = 10
D = 11

2 x 40 +
2 x 3 +
2 x 20 +
2 x 37 =

200 bits

Symbol	Frequency
A	40
B	3
C	20
D	37

How many bits to encode the file?

Fixed length code

Use ceil($\log_2|\Sigma|$) bits for each character

Symbol	Frequency
A	40
B	3
C	20
D	37

A = 00   2 x 40 +
B = 01   2 x 3 +
C = 10   2 x 20 +
D = 11   2 x 37 =

200 bits

Can we do better?

Variable length code

What about:

A = 0
B = 01
C = 10
D = 1

1 x 40 +
2 x 3 +
2 x 20 +
1 x 37 =

123 bits

Symbol	Frequency
A	40
B	3
C	20
D	37

How many bits to encode the file?

Decoding a file

A = 0
B = 01
C = 10
D = 1

010100011010

What characters does this sequence represent?

Decoding a file

A = 0
B = 01
C = 10
D = 1

010100011010

A D or **B?**

What characters does this sequence represent?

Variable length code

What about:

A = 0
B = 100
C = 101
D = 11

Is it decodeable?

Symbol	Frequency
A	40
B	3
C	20
D	37

Variable length code

What about:

A = 0
B = 100
C = 101
D = 11

1 x 40 +
3 x 3 +
3 x 20 +
2 x 37 =

183 bits
(8.5% reduction)

Symbol	Frequency
A	40
B	3
C	20
D	37

How many bits to encode the file?

Prefix codes

A prefix code is a set of codes where no codeword is a **prefix** of any other codeword

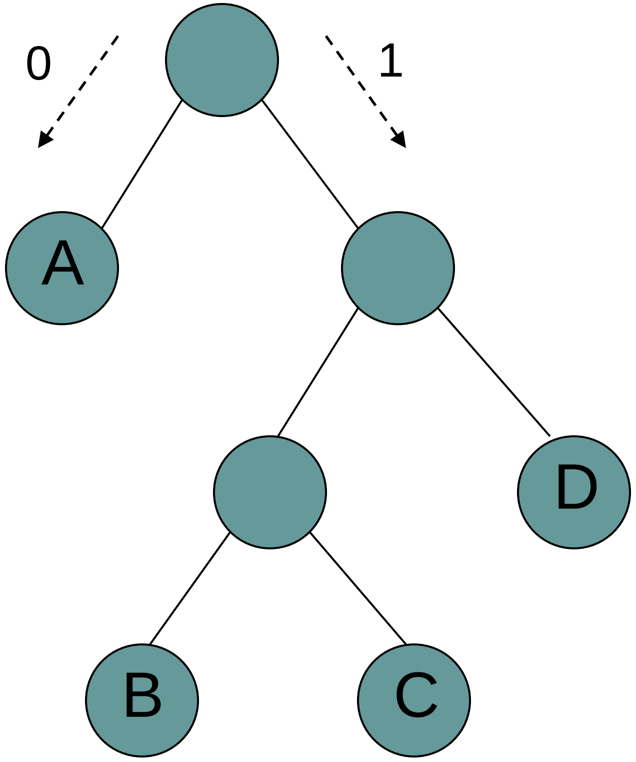
A = 0
B = 01
C = 10
D = 1

A = 0
B = 100
C = 101
D = 11

Prefix tree

We can encode a prefix code using a binary tree where each leaf represents an encoding of a symbol

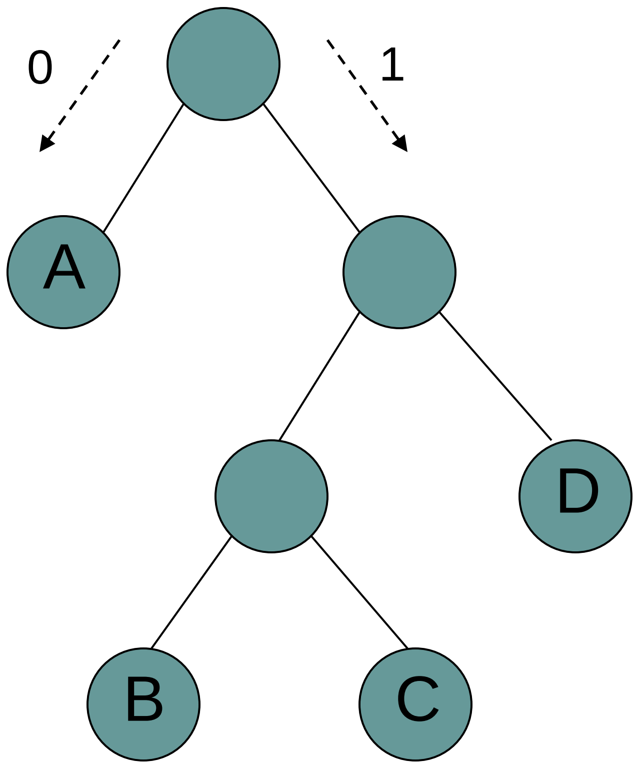
A = 0
B = 100
C = 101
D = 11



Decoding using a prefix tree

To decode, we traverse the graph until a leaf node is reached and output the symbol

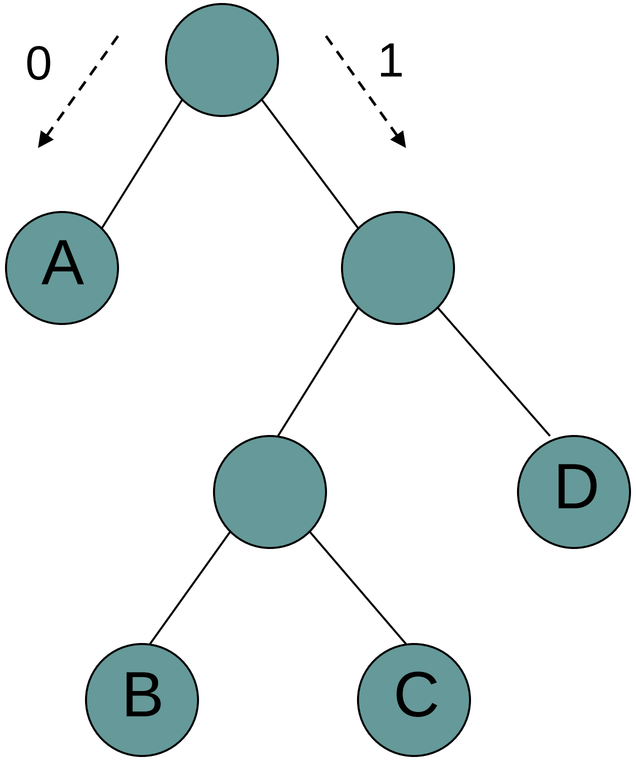
A = 0
B = 100
C = 101
D = 11



Decoding using a prefix tree

Traverse the graph until a leaf node is reached and output the symbol

1000111010100

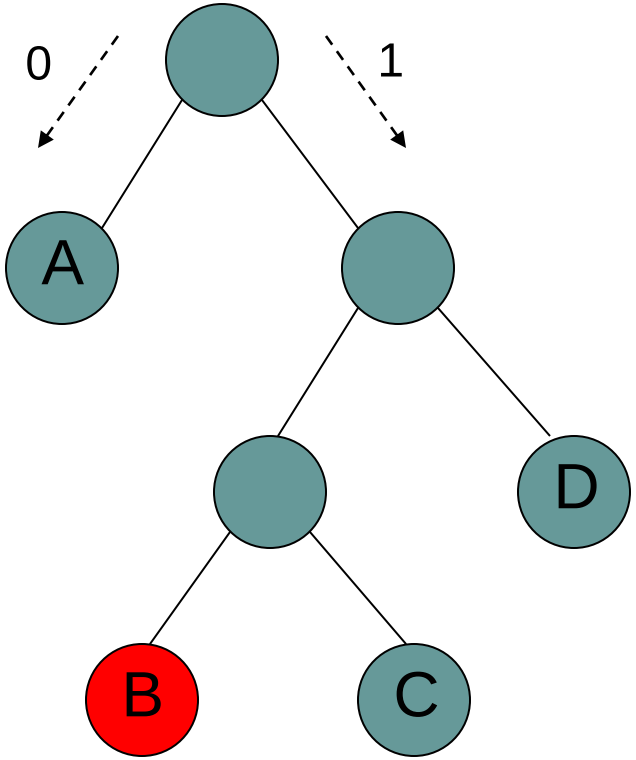


Decoding using a prefix tree

Traverse the graph until a leaf node is reached and output the symbol

1000111010100

B

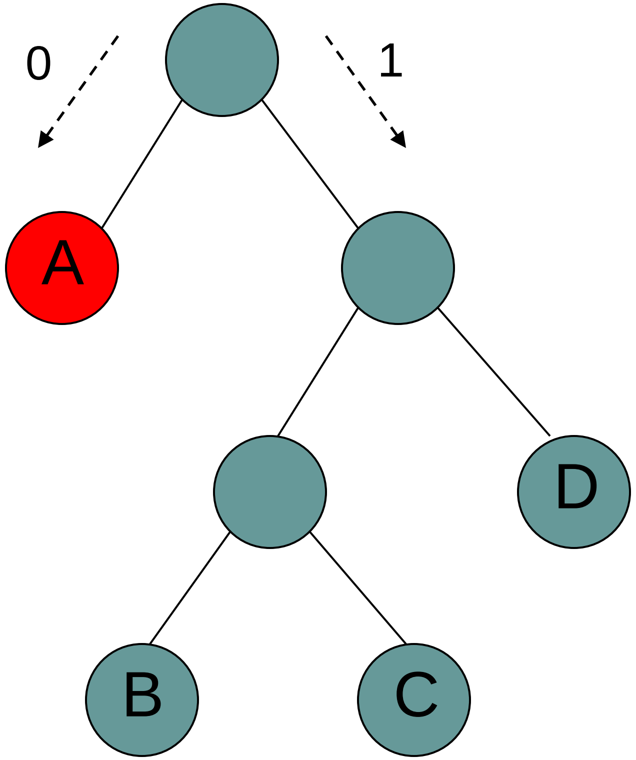


Decoding using a prefix tree

Traverse the graph until a leaf node is reached and output the symbol

1000111010100

B A

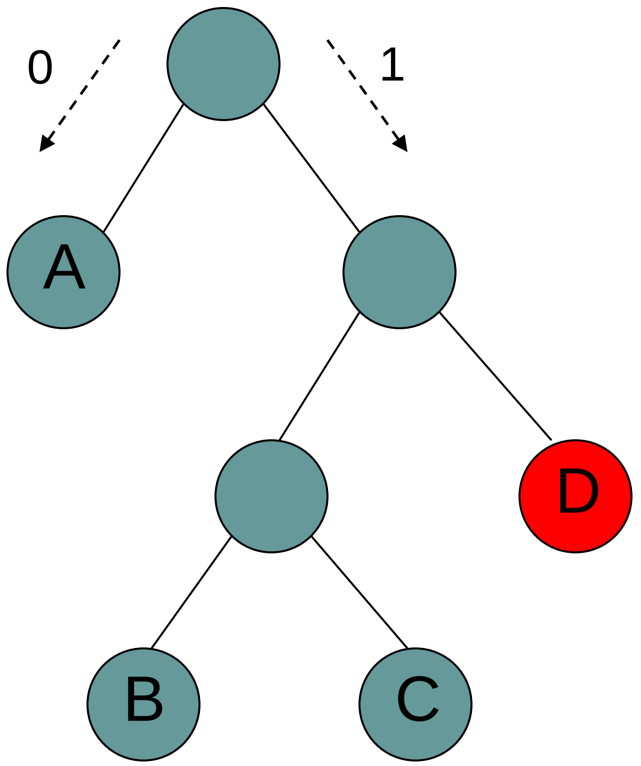


Decoding using a prefix tree

Traverse the graph until a leaf node is reached and output the symbol

1000111010100

B A D

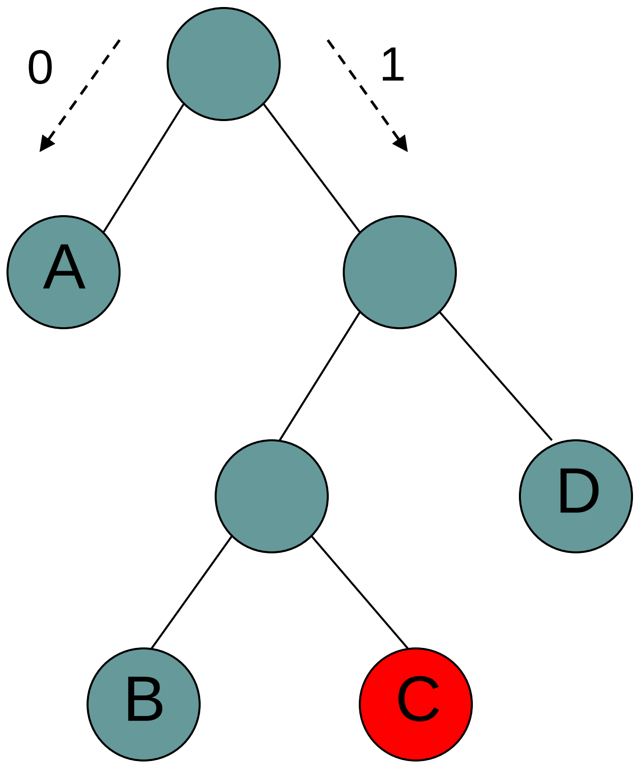


Decoding using a prefix tree

Traverse the graph until a leaf node is reached and output the symbol

1000111010100

B A D C

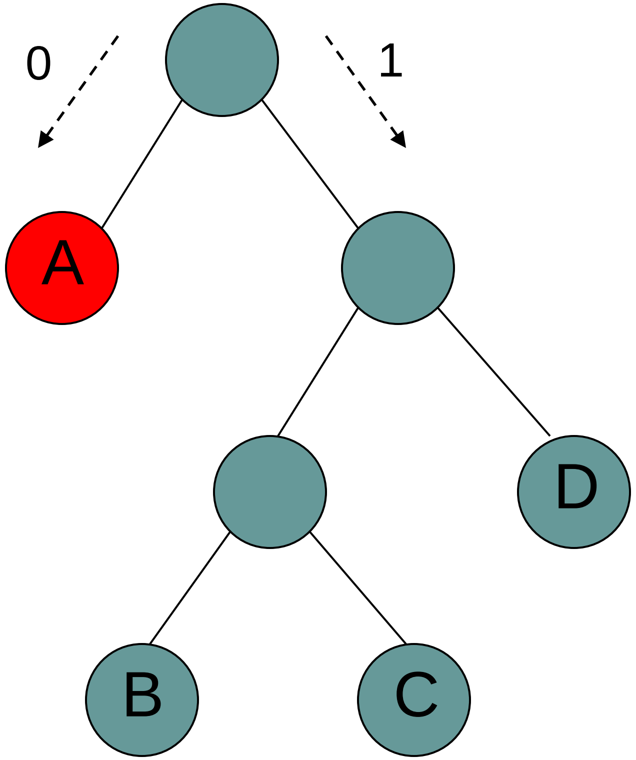


Decoding using a prefix tree

Traverse the graph until a leaf node is reached and output the symbol

1000111010100

B A D C A

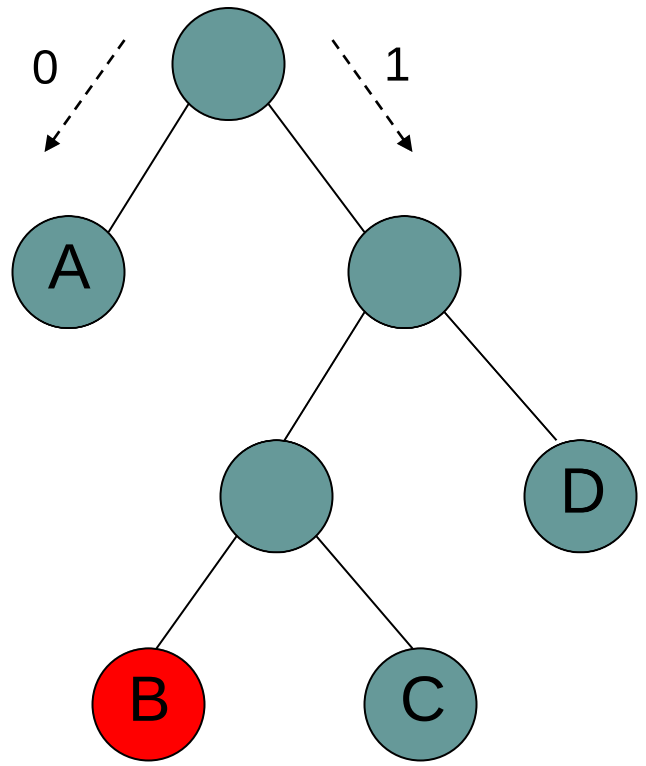


Decoding using a prefix tree

Traverse the graph until a leaf node is reached and output the symbol

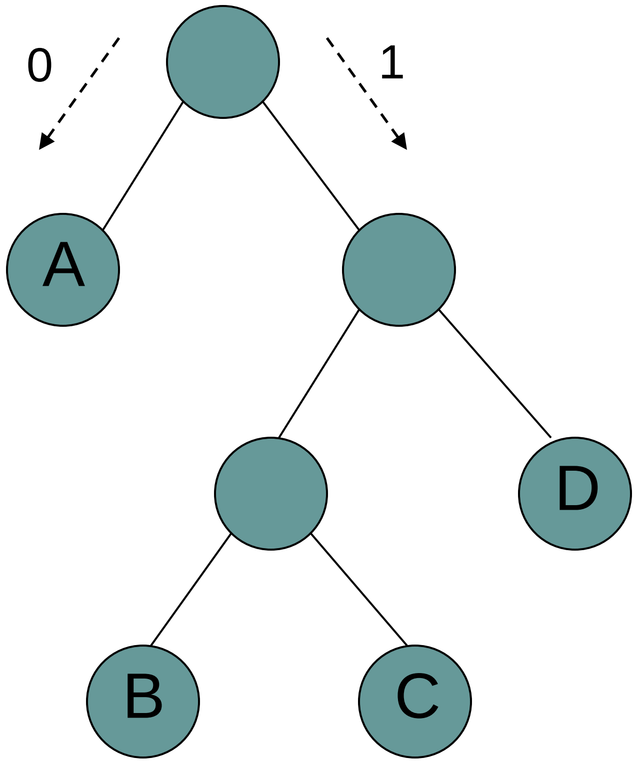
1000111010100

B A D C A B



Determining the cost of a file

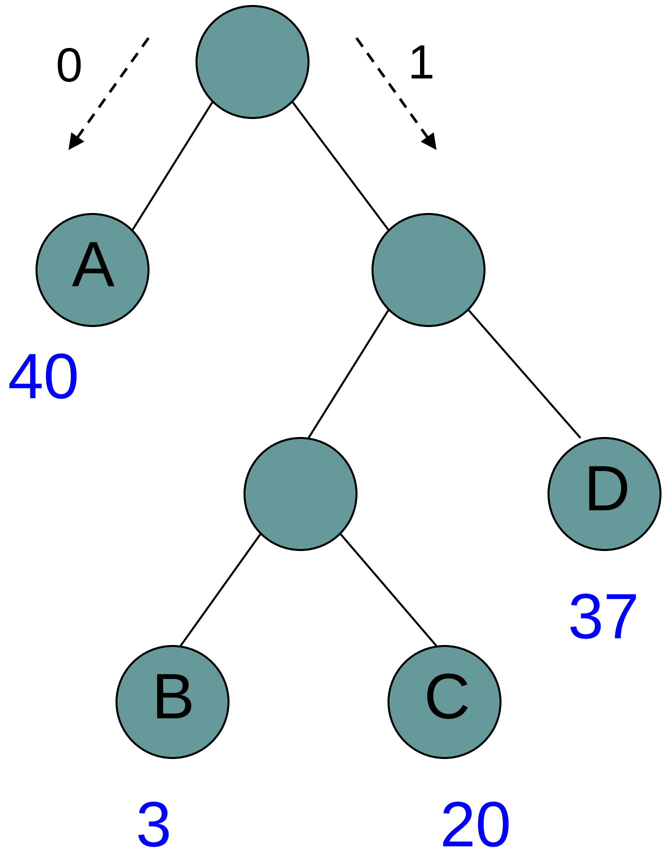
Symbol	Frequency
A	40
B	3
C	20
D	37



Determining the cost of a file

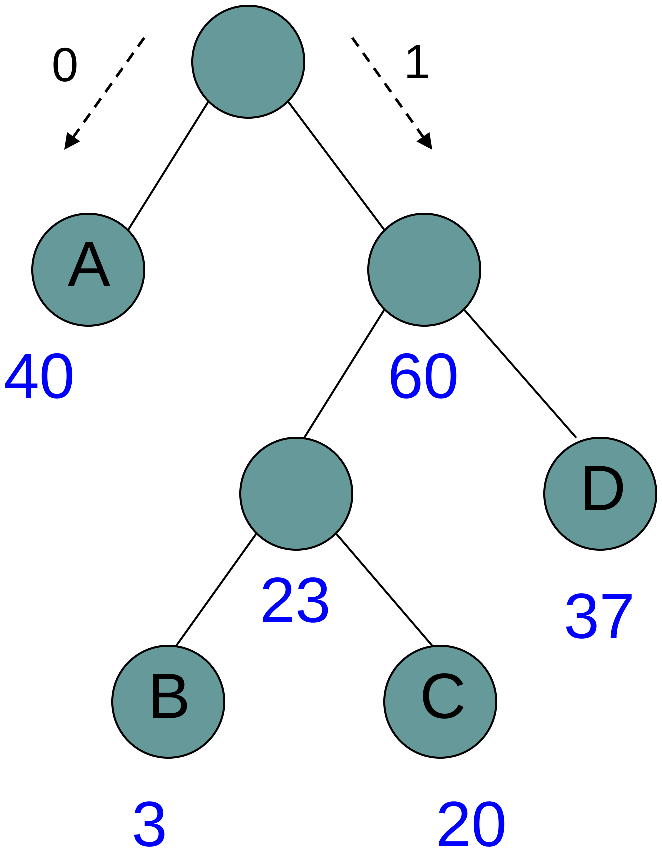
Symbol	Frequency
A	40
B	3
C	20
D	37

$$\mathrm{cost}(T) = \sum_{i=1}^{n} f_i \,\mathrm{depth}(i)$$



Determining the cost of a file

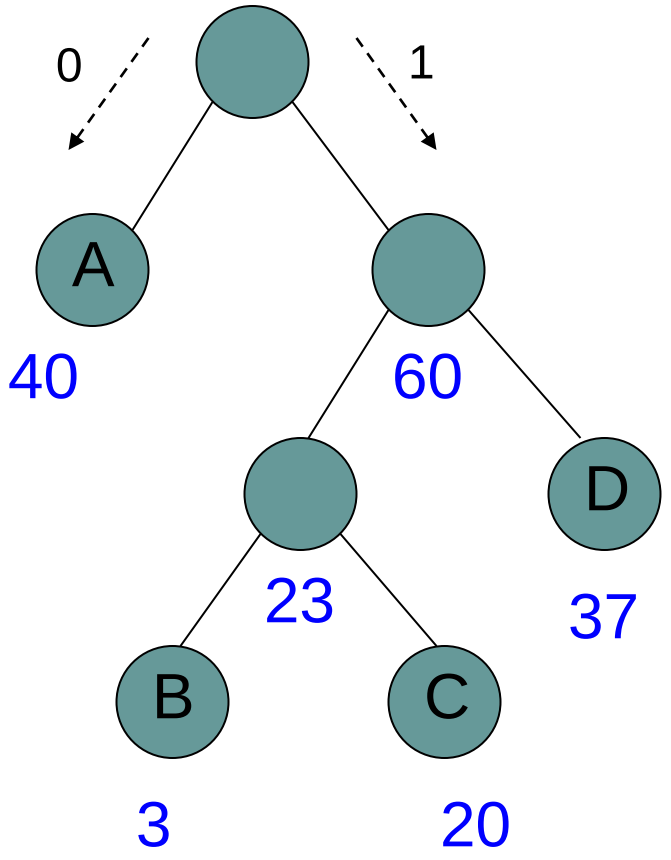
Symbol	Frequency
A	40
B	3
C	20
D	37



What if we label the internal nodes with the sum of the children?

Determining the cost of a file

Symbol	Frequency
A	40
B	3
C	20
D	37



Cost is equal to the sum of the internal nodes and the leaf nodes

Determining the cost of a file

As we move down the tree, one bit gets read for every nonroot node

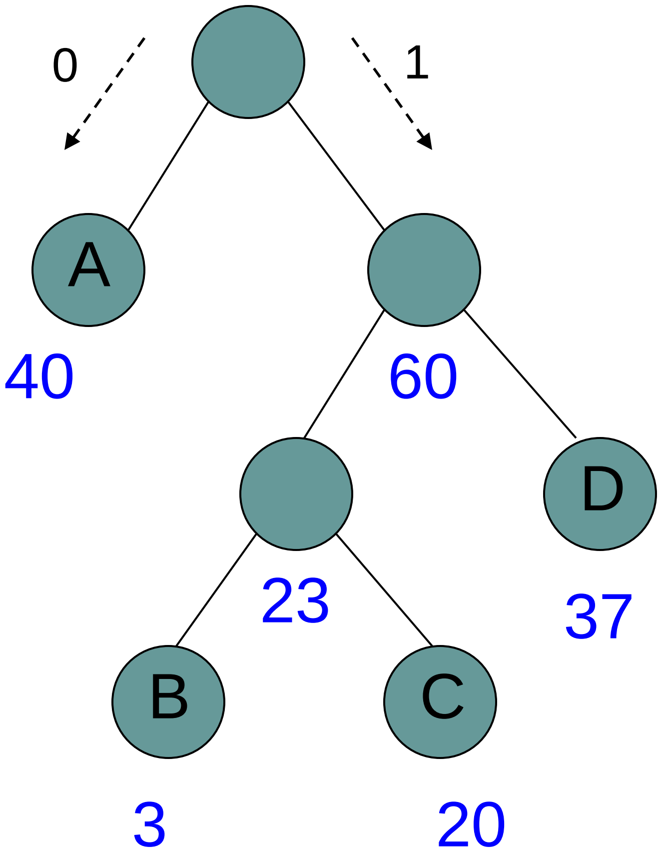
40 times we see a 0 by itself

60 times we see a prefix that starts with a 1

of those, 37 times we see an additional 1

the remaining 23 times we see an additional 0

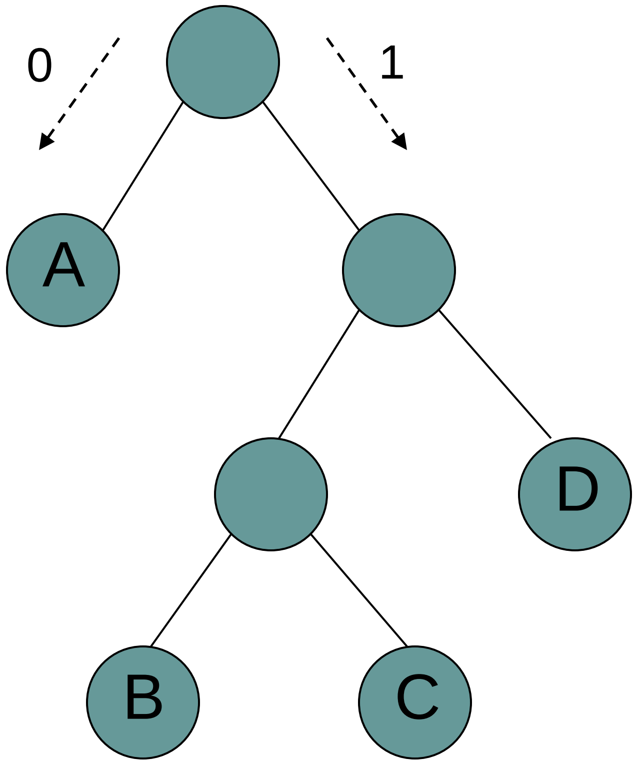
of these, 20 times we see a last 1 and 3 times a last 0



A greedy algorithm?

Given file frequencies, can we come up with a prefix-free encoding (i.e. build a prefix tree) that minimizes the number of bits?

Symbol	Frequency
A	40
B	3
C	20
D	37



A greedy algorithm?

Given file frequencies, can we come up with a prefix-free encoding (i.e. build a prefix tree) that minimizes the number of bits?

```
HUFFMAN(F)
Q ← MAKEHEAP(F)
for i ← 1 to |Q| − 1
        allocate a new node z
        left[z] ← x ← EXTRACTMIN(Q)
        right[z] ← y ← EXTRACTMIN(Q)
        f[z] ← f[x] + f[y]
        INSERT(Q, z)
return EXTRACTMIN(Q)
```

HUFFMAN(F)

$Q \leftarrow$ MAKEHEAP(F)
for $i \leftarrow 1$ **to** $|Q| - 1$
    allocate a new node z
    $left[z] \leftarrow x \leftarrow$ EXTRACTMIN(Q)
    $right[z] \leftarrow y \leftarrow$ EXTRACTMIN(Q)
    $f[z] \leftarrow f[x] + f[y]$
    INSERT(Q, z)
return EXTRACTMIN(Q)

Symbol	Frequency
A	40
B	3
C	20
D	37

Heap

```
HUFFMAN(F)
Q ← MAKEHEAP(F)
for i ← 1 to |Q| − 1
        allocate a new node z
        left[z] ← x ← EXTRACTMIN(Q)
        right[z] ← y ← EXTRACTMIN(Q)
        f[z] ← f[x] + f[y]
        INSERT(Q, z)
return EXTRACTMIN(Q)
```

Symbol	Frequency
A	40
B	3
C	20
D	37

Heap

B	3
C	20
D	37
A	40

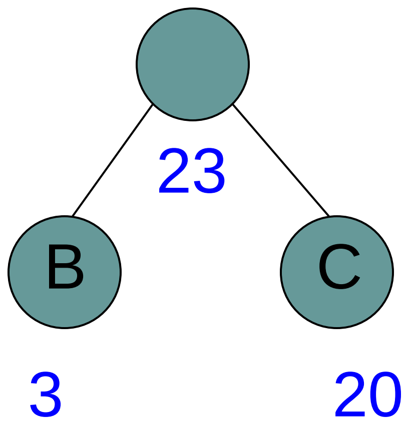
```
HUFFMAN(F)
Q ← MAKEHEAP(F)
for i ← 1 to |Q| − 1
        allocate a new node z
        left[z] ← x ← EXTRACTMIN(Q)
        right[z] ← y ← EXTRACTMIN(Q)
        f[z] ← f[x] + f[y]
        INSERT(Q, z)
return EXTRACTMIN(Q)
```

Symbol	Frequency
A	40
B	3
C	20
D	37

Heap

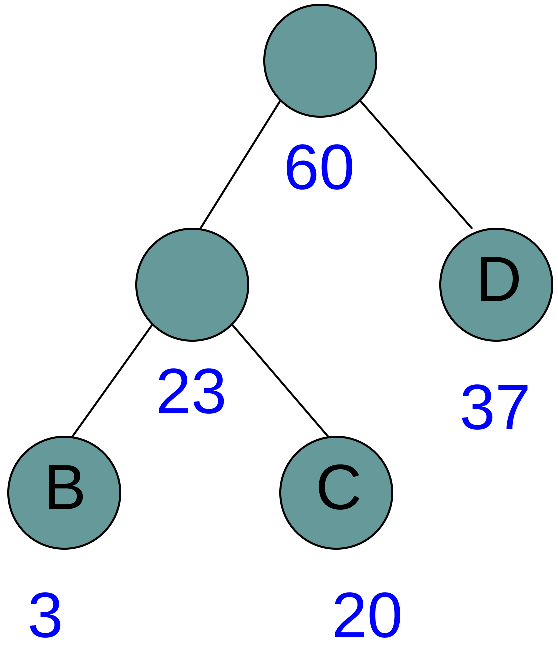
merging with this node will incur an additional cost of 23 →

BC	23
D	37
A	40



```
HUFFMAN(F)
Q ← MAKEHEAP(F)
for i ← 1 to |Q| − 1
        allocate a new node z
        left[z] ← x ← EXTRACTMIN(Q)
        right[z] ← y ← EXTRACTMIN(Q)
        f[z] ← f[x] + f[y]
        INSERT(Q, z)
return EXTRACTMIN(Q)
```

Symbol	Frequency
A	40
B	3
C	20
D	37



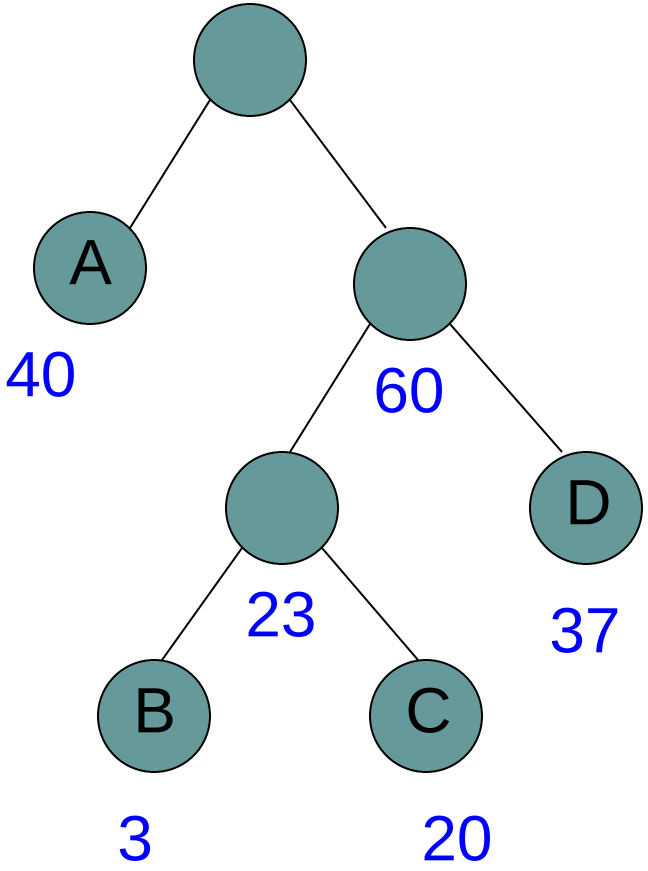
Heap

A 40

BCD 60

```
HUFFMAN(F)
Q ← MAKEHEAP(F)
for i ← 1 to |Q| − 1
        allocate a new node z
        left[z] ← x ← EXTRACTMIN(Q)
        right[z] ← y ← EXTRACTMIN(Q)
        f[z] ← f[x] + f[y]
        INSERT(Q, z)
return EXTRACTMIN(Q)
```

Symbol	Frequency
A	40
B	3
C	20
D	37



Heap

ABCD 100

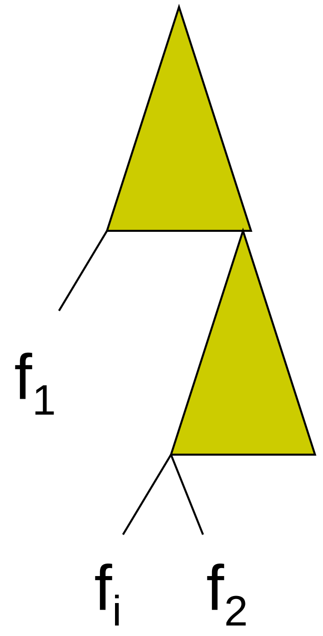
Is it correct?

The algorithm selects the symbols with the two smallest frequencies first (call them f_1 and f_2)

Is it correct?

The algorithm selects the symbols with the two smallest frequencies first (call them f_1 and f_2)

Consider a tree that did not do this (proof by contradiction):

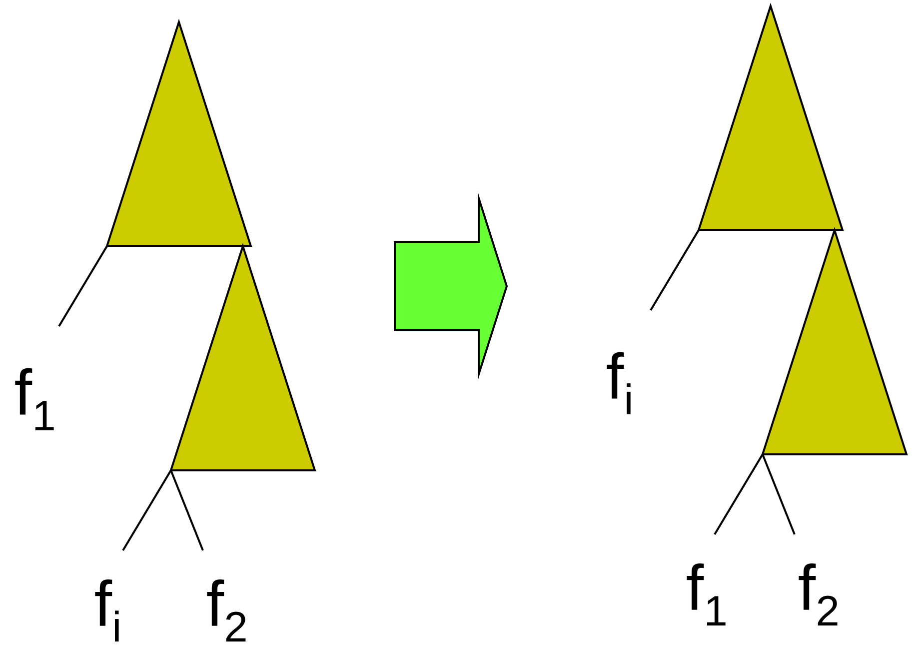


Is it optimal?

Is it correct?

The algorithm selects the symbols with the two smallest frequencies first (call them f_1 and f_2)

Consider a tree that did not do this:



$$\text{cost}(T) = \sum_{i=1}^{n} f_i \, \text{depth}(i)$$

- frequencies don't change
- cost will **decrease** since $f_1 < f_i$

contradiction

Runtime?

```
Huffman(F)
Q ← MakeHeap(F)
for i ← 1 to |Q| − 1
        allocate a new node z
        left[z] ← x ← ExtractMin(Q)
        right[z] ← y ← ExtractMin(Q)
        f[z] ← f[x] + f[y]
        Insert(Q, z)
return ExtractMin(Q)
```

1 call to MakeHeap

2(n-1) calls ExtractMin

n-1 calls Insert

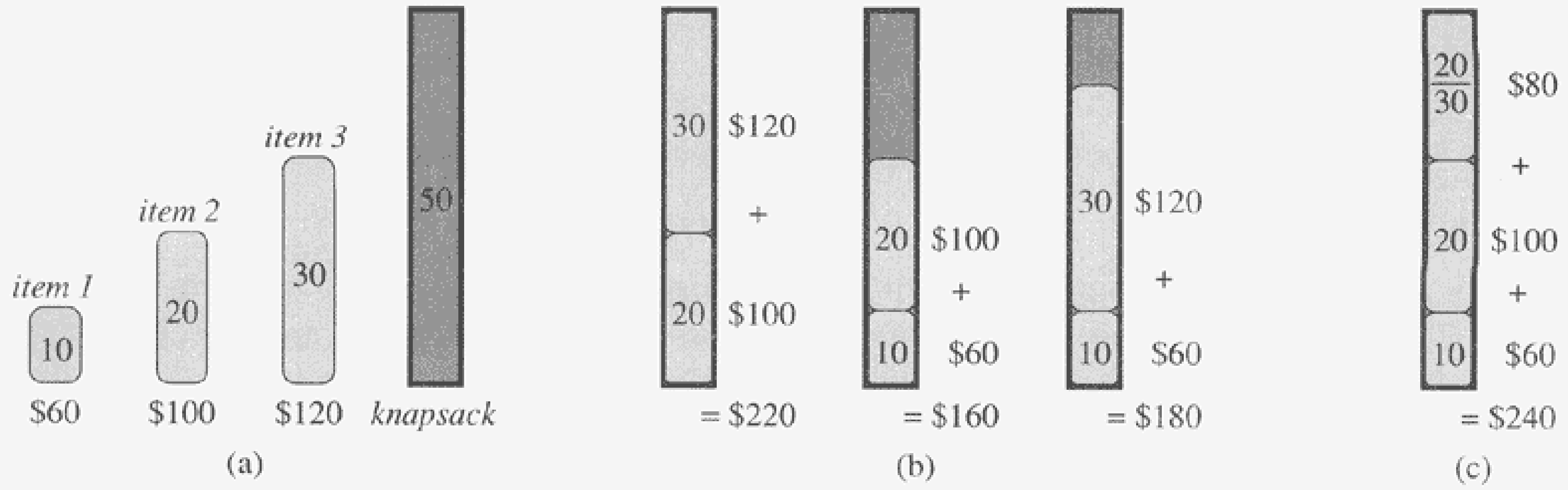
O(n log n)

Knapsack problems: Greedy or not?

0-1 Knapsack – A thief robbing a store finds n items worth $v_1, v_2, .., v_n$ dollars and weight $w_1, w_2, \ldots, w_n$ pounds, where v_i and w_i are integers. The thief can carry at most W pounds in the knapsack. Which items should the thief take if he wants to maximize value.

Fractional knapsack problem – Same as above, but the thief happens to be at the bulk section of the store and can carry fractional portions of the items. For example, the thief could take 20% of item i for a weight of $0.2w_i$ and a value of $0.2v_i$.

Knapsack problems: Greedy or not?



The greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

Non-optimal greedy algorithms

All the greedy algorithms we've looked at so far give the optimal answer

Some of the most common greedy algorithms generate good, but non-optimal solutions

- set cover
- clustering
- hill-climbing
- relaxation

Acknowledgements

- Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., Introduction to algorithms. MIT press, 2009
- Dr. David Kauchak, Pomona College
- Prof. David Plaisted, The University of North Carolina at Chapel Hill