ACM Challenges Lab

Even Pairs, Shelves, and Greedy Algorithms

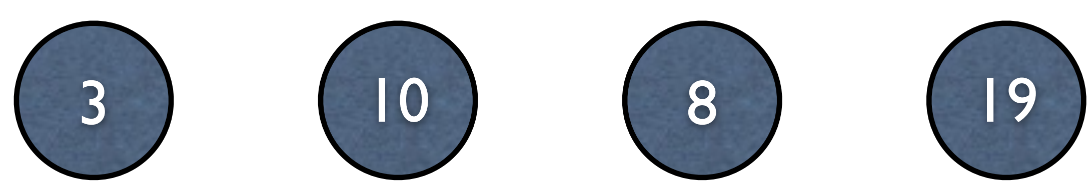
Greedy Algorithms

- Often choices that seem best at particular moment turn out not to be optimal in the long run (e.g., Chess, Life, etc..)
- However, sometimes locally optimal choices are also globally optimal! This is when we can apply Greedy Algorithms.

Example: Optimal Merging

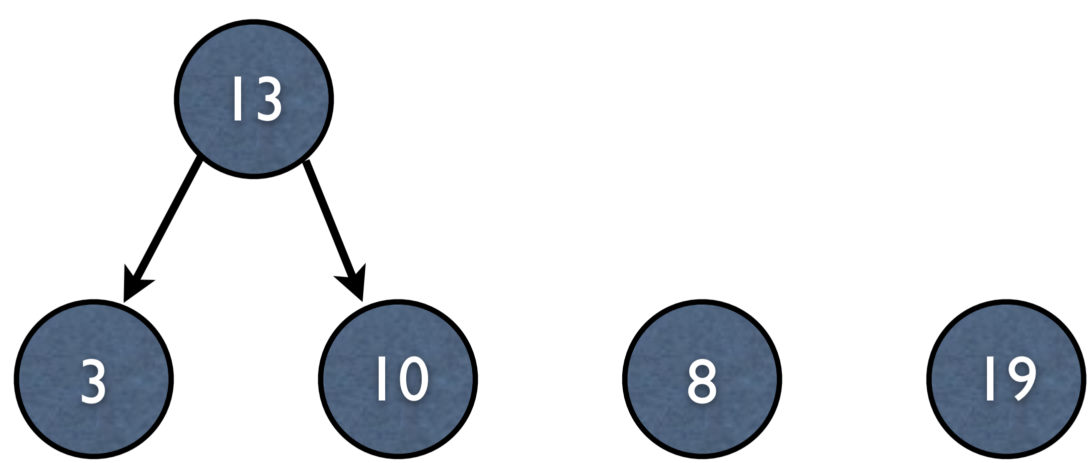
- We have n sorted arrays
- In MergeSort we take two sorted arrays and merge them into one sorted array.
- Say, this merge(array1 , array2) operation takes |array1| + |array2| time.
- What is the minimal time needed to merge all arrays into one sorted array?

Example: Optimal Merging

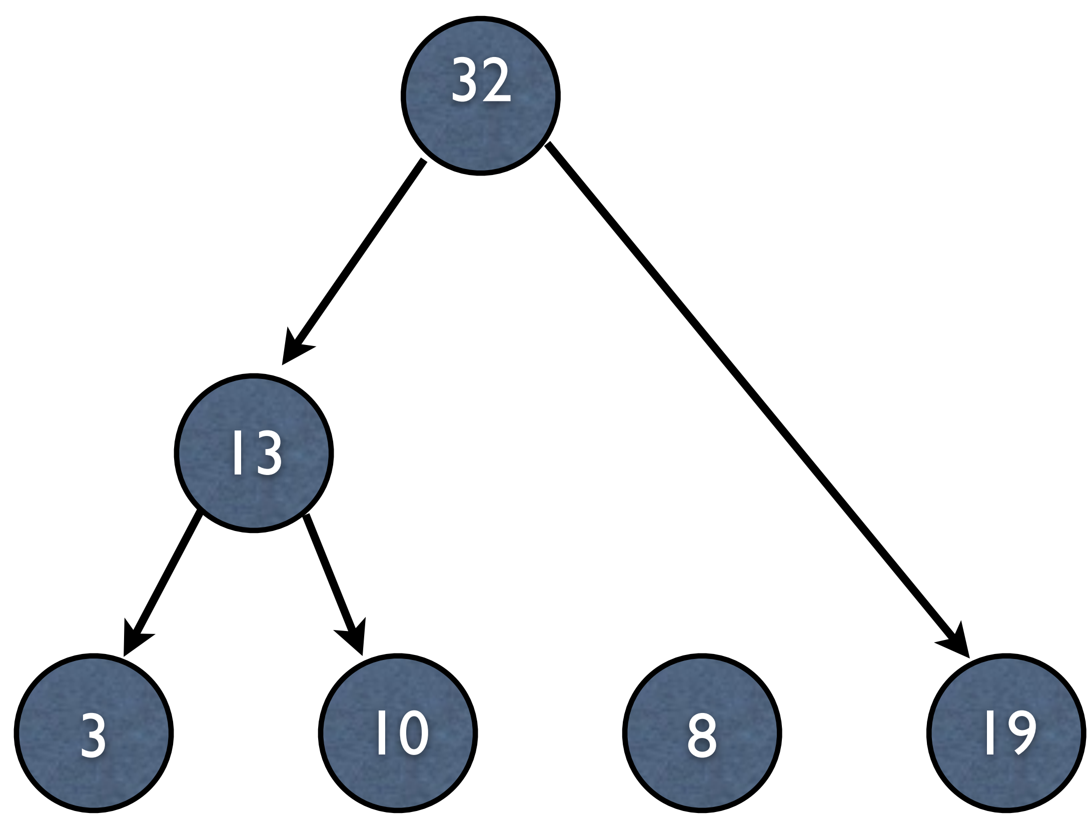


(numbers denote lengths)

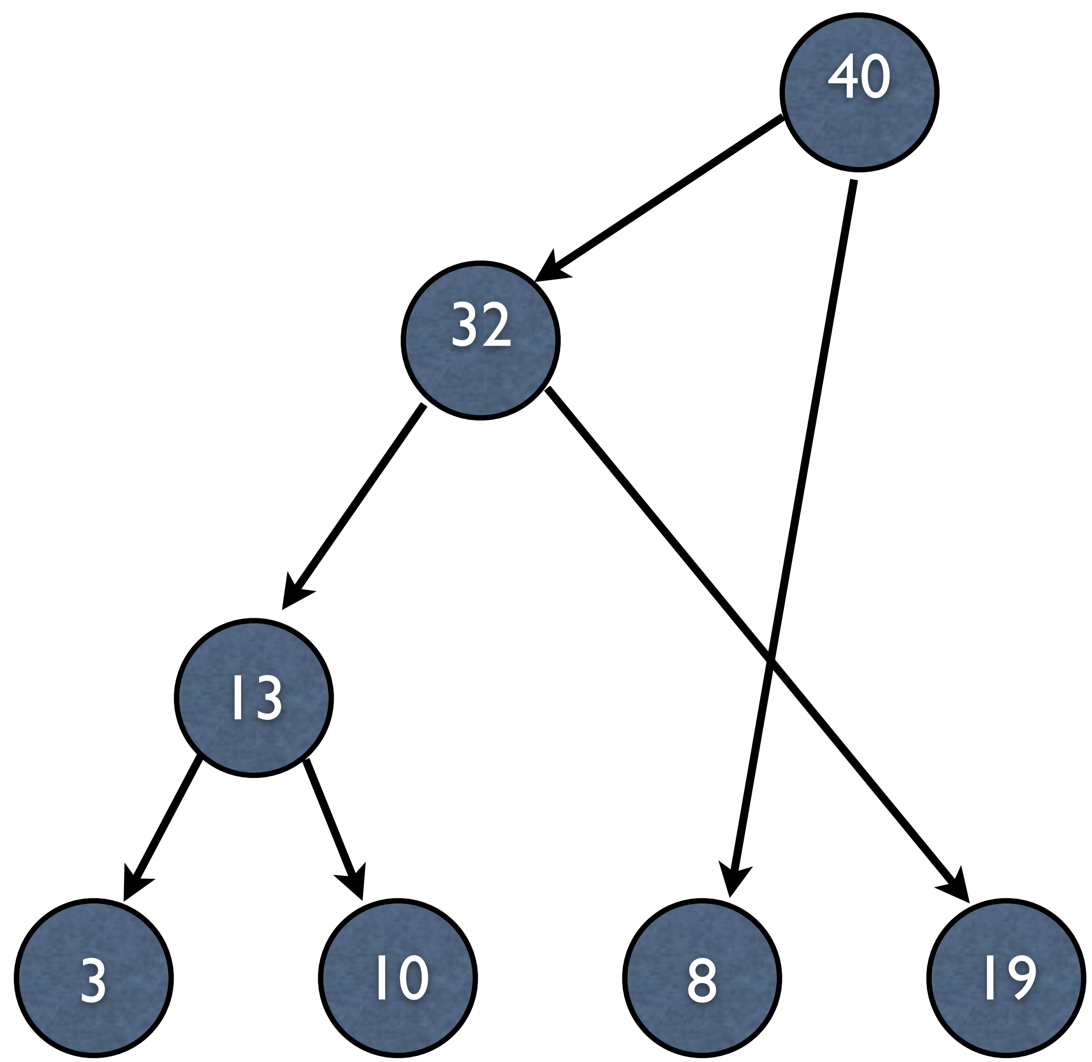
Example: Optimal Merging



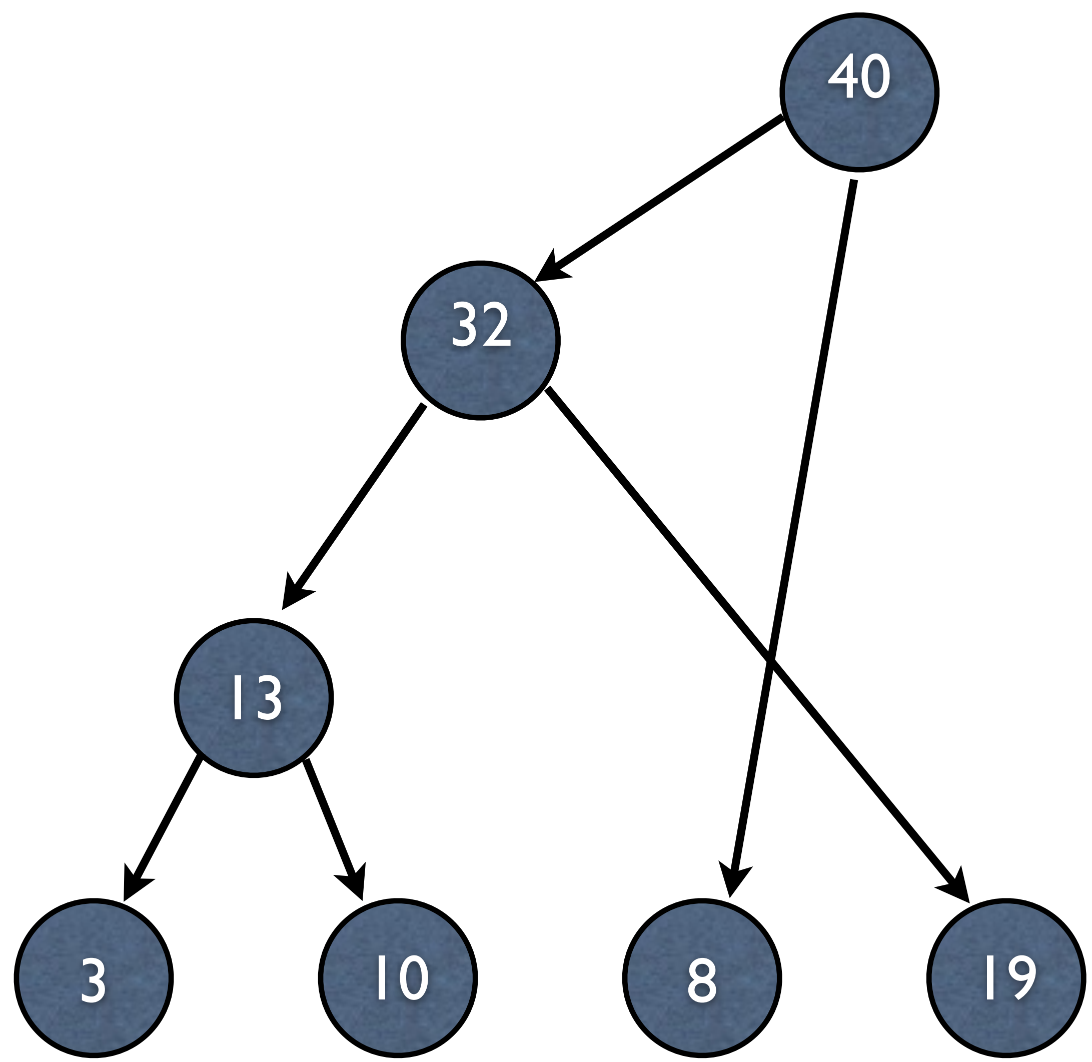
Example: Optimal Merging



Example: Optimal Merging

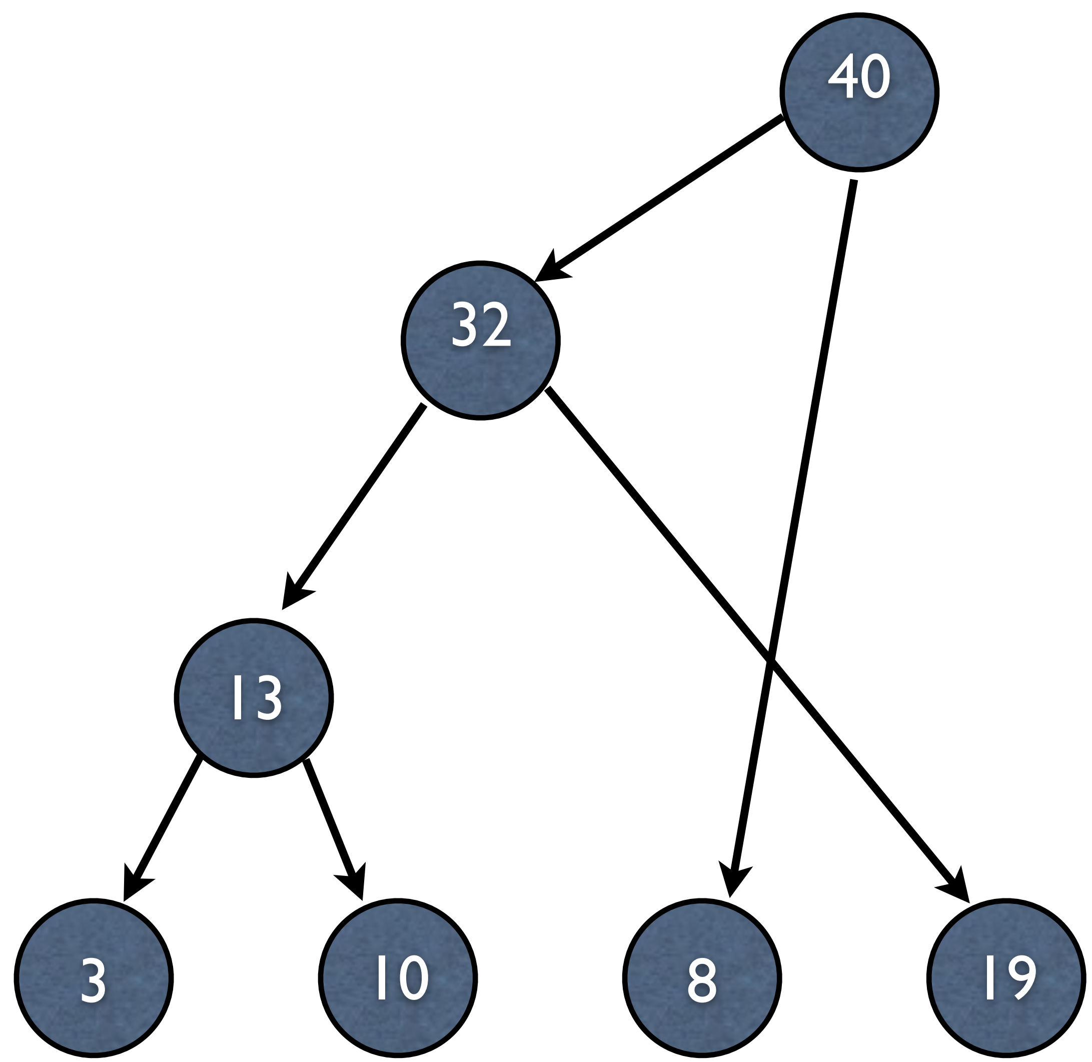


Example: Optimal Merging



Cost: 13 + 32 + 40 = 85

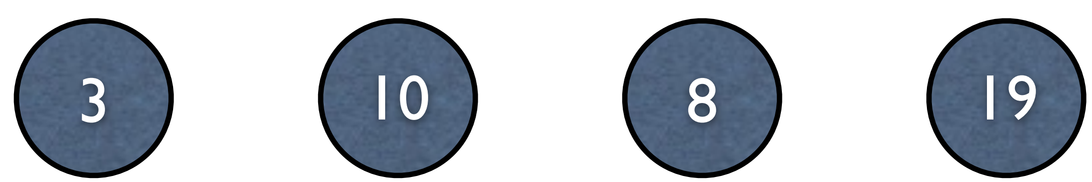
Example: Optimal Merging



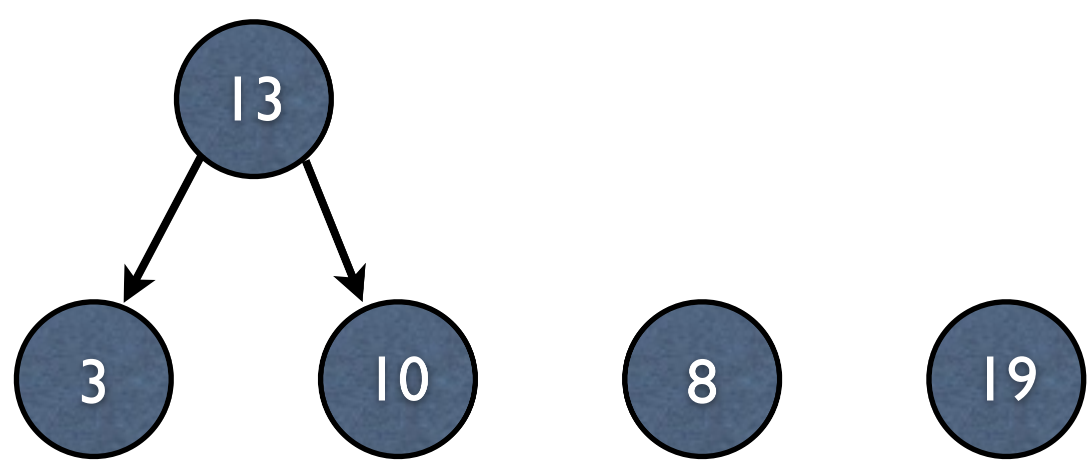
Can we do better?

Cost: 13 + 32 + 40 = 85

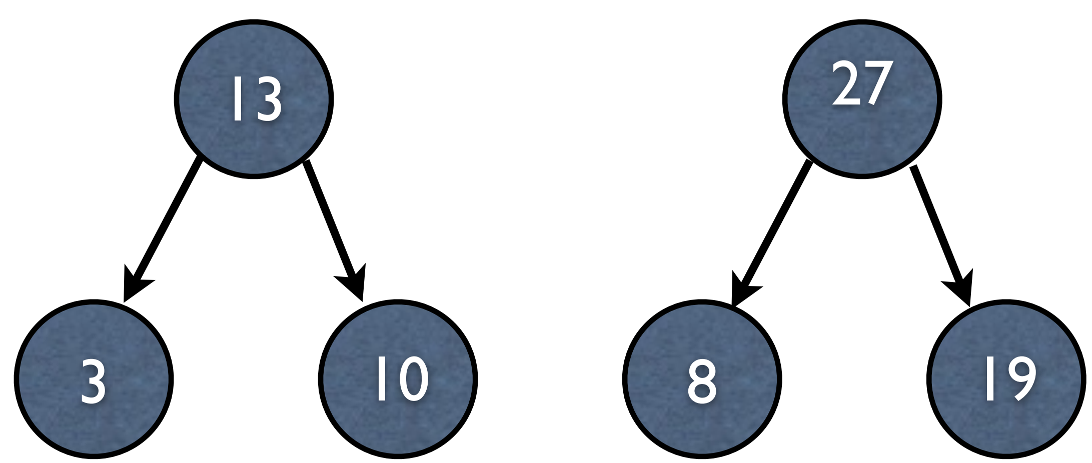
Example: Optimal Merging



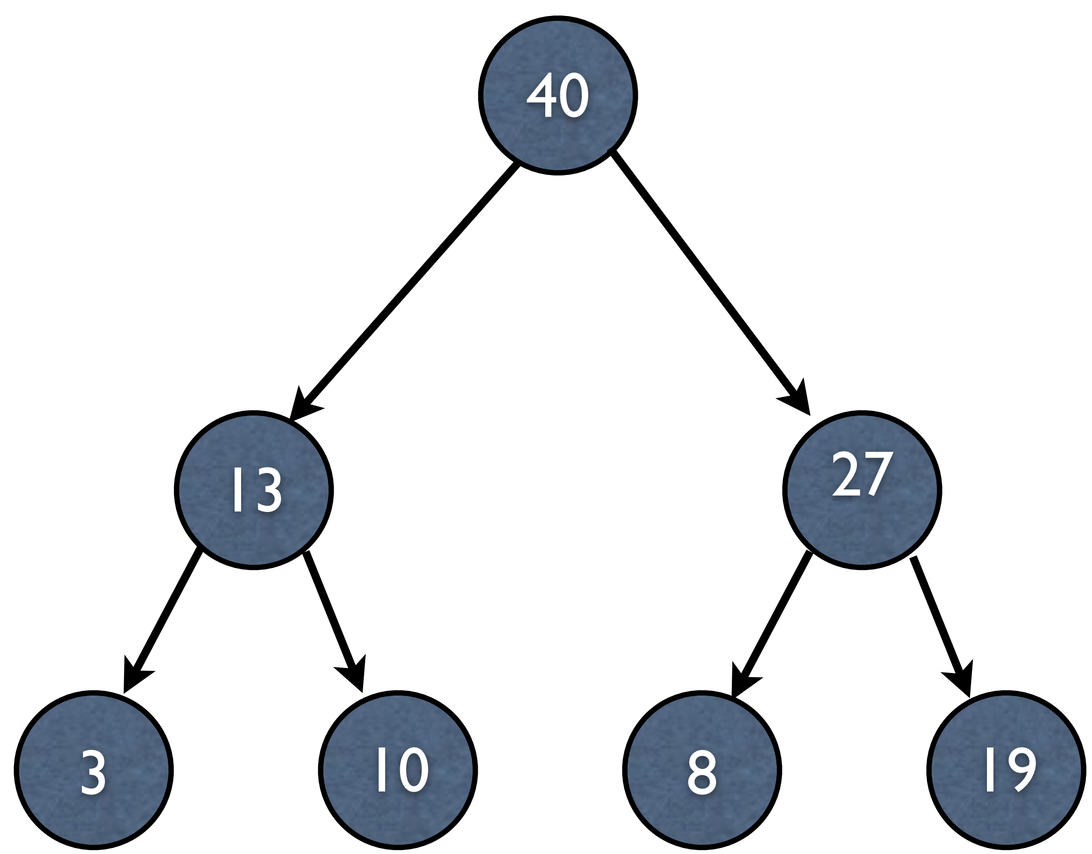
Example: Optimal Merging



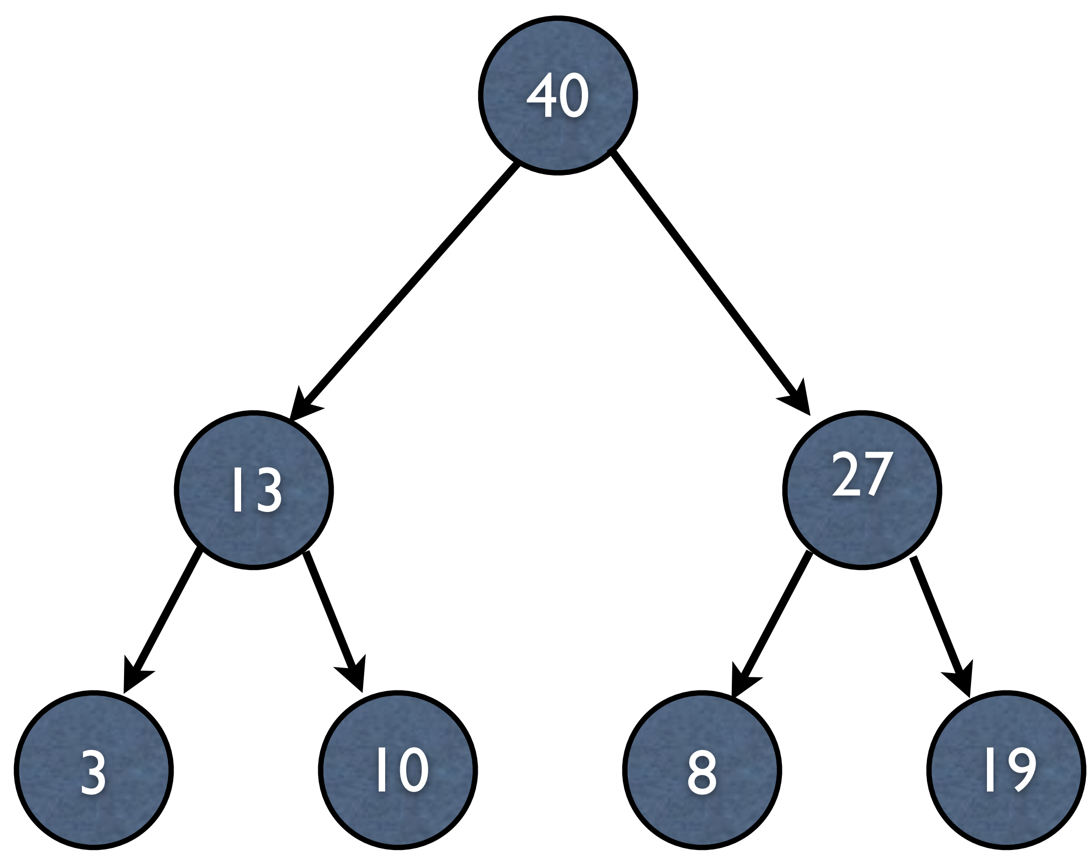
Example: Optimal Merging



Example: Optimal Merging



Example: Optimal Merging

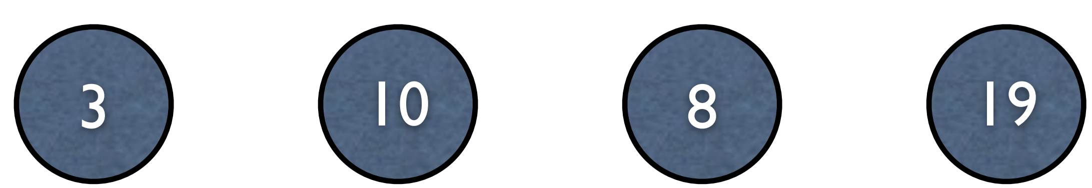


Even better?

Cost: 13 + 27 + 40 = 80

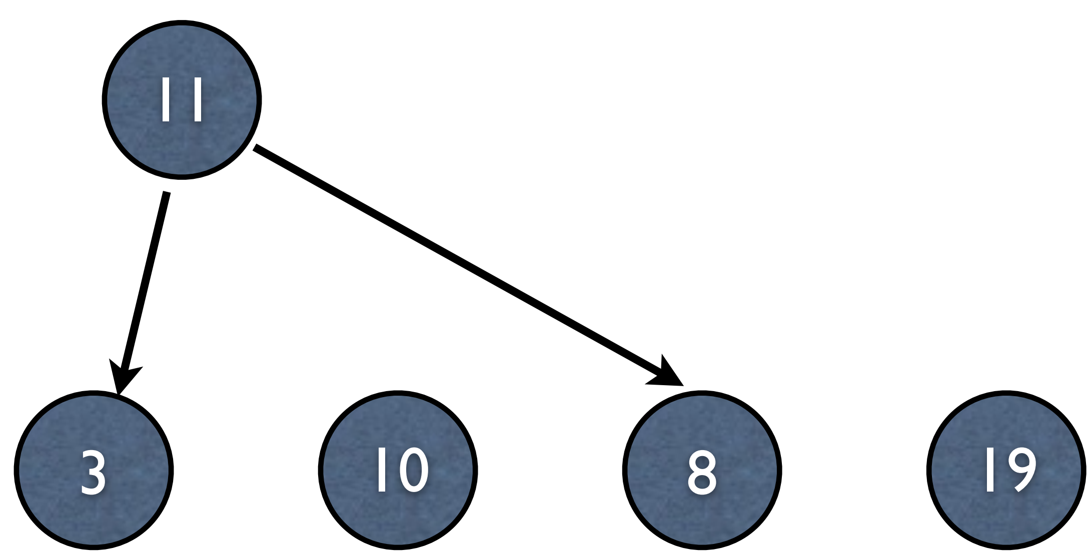
Example: Optimal Merging

Approach: Always take two shortest arrays



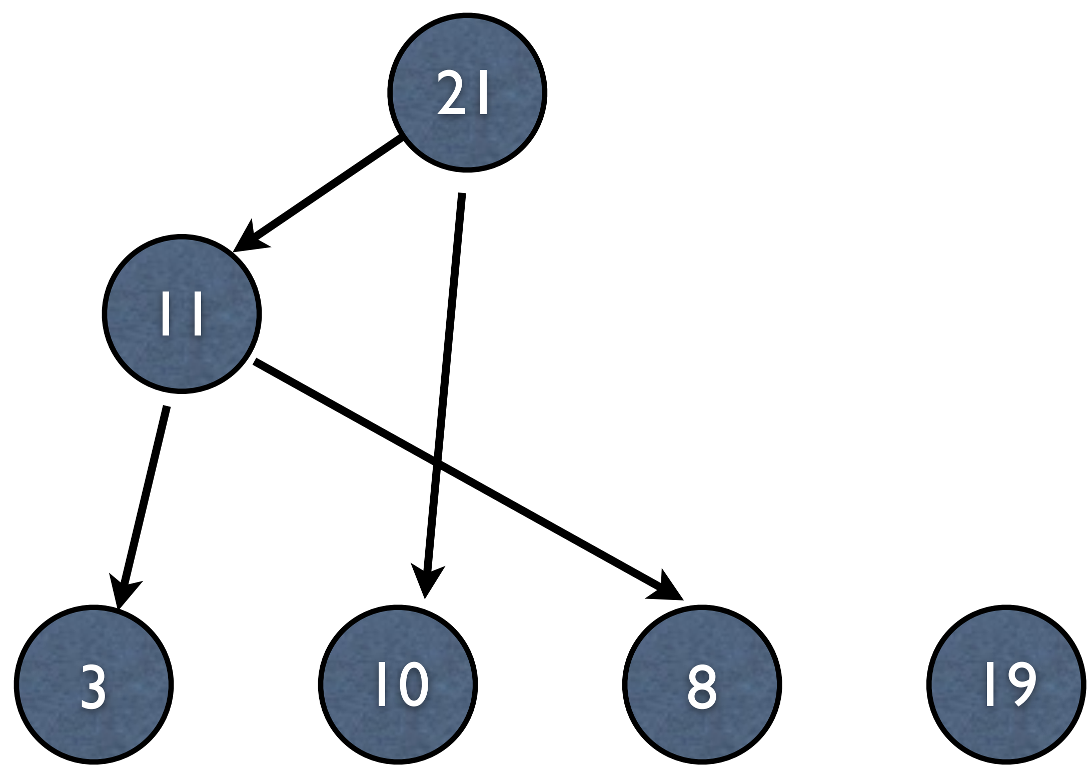
Example: Optimal Merging

Approach: Always take two shortest arrays



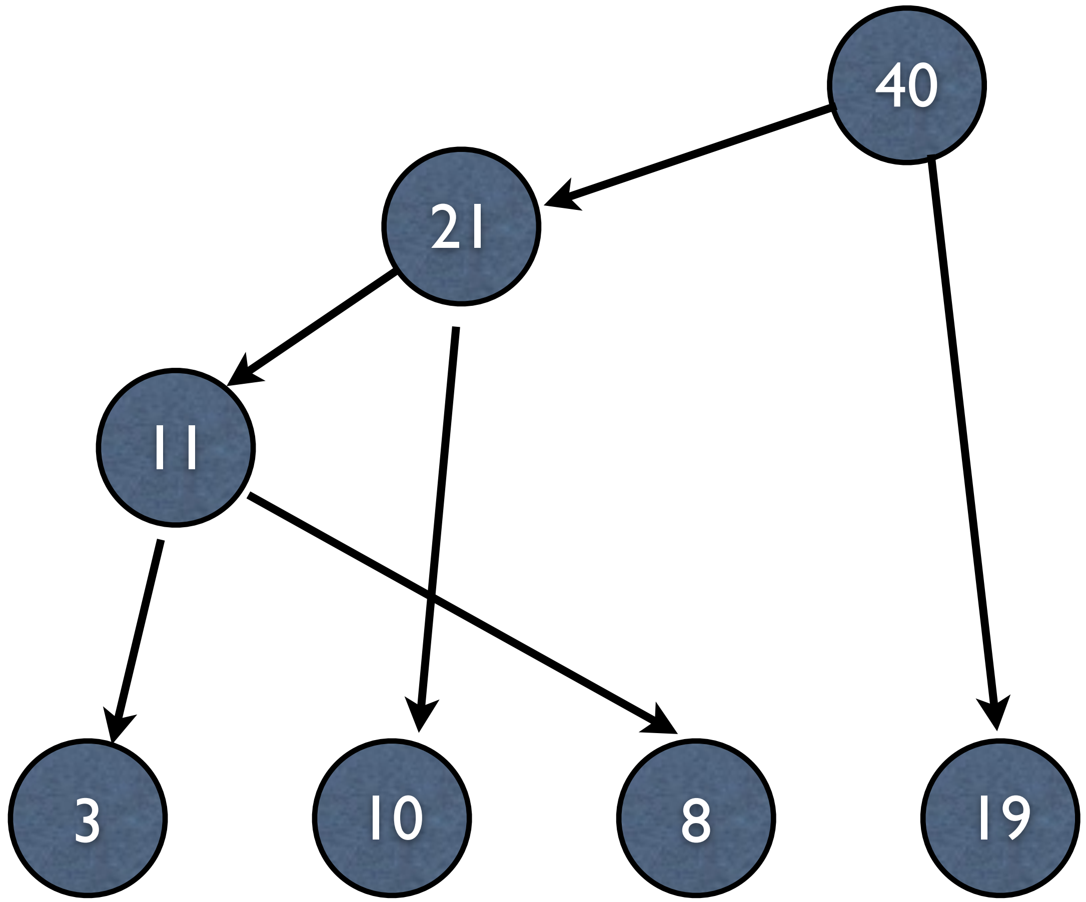
Example: Optimal Merging

Approach: Always take two shortest arrays



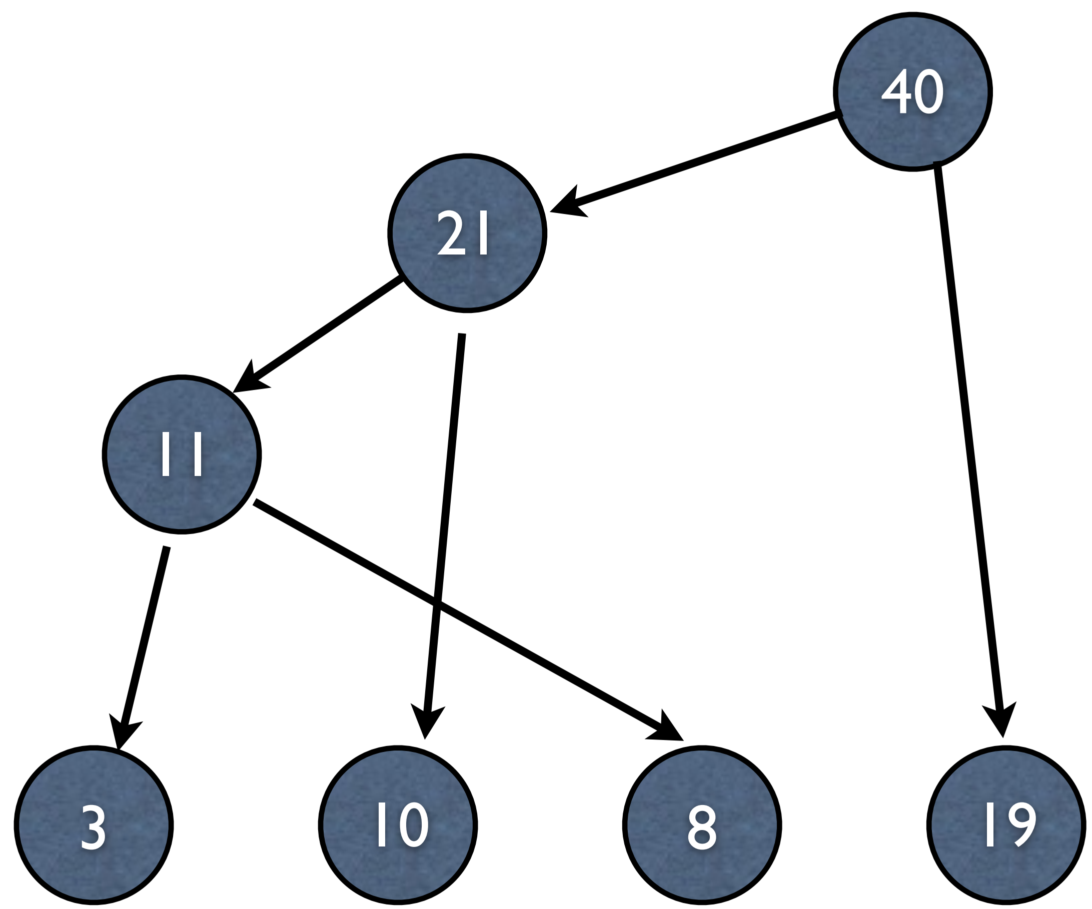
Example: Optimal Merging

Approach: Always take two shortest arrays



Example: Optimal Merging

Approach: Always take two shortest arrays



Cost: 11 + 21 + 40 = 72

Example: Optimal Merging

Approach: Always take two shortest arrays

Can we do anything better?

Example: Optimal Merging

Approach: Always take two shortest arrays

Can we do anything better?

No!

(We'll see later why)

Algorithm "Always take two shortest arrays" is a greedy algorithm.

Example: Optimal Merging

Core of every greedy algorithm

- Come up with a property by which you will make choices (merge arrays).
- This property should give you a measure of the locally optimal choice.

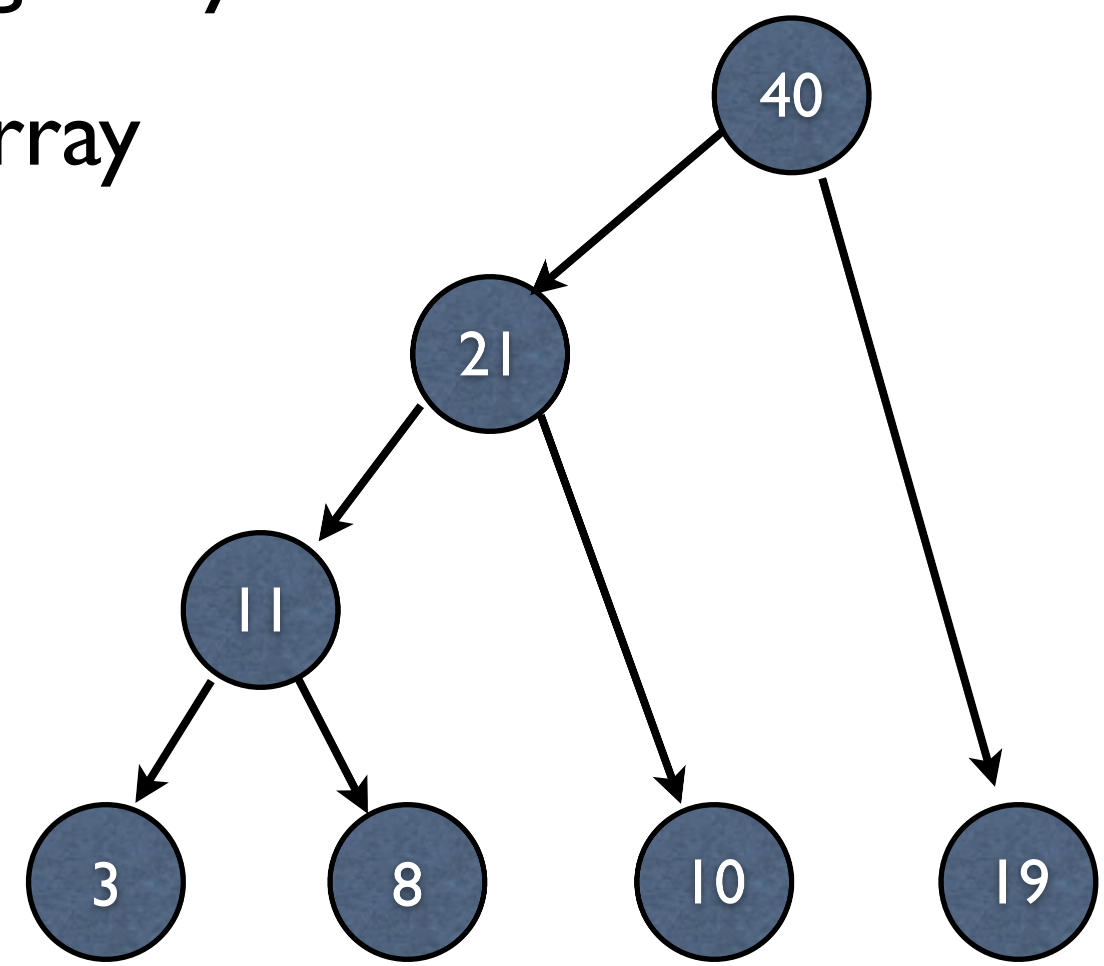
Example: Optimal Merging

How to figure out if your greedy approach works (or doesn't work)?

- Find a counter example (and prove it doesn't work)
- Exchange argument:
  - Assume you have an optimal solution.
  - Modify the optimal solution, such that it contains part of your greedy solution
  - Prove that this modification is at least as good
  - Repeat the same process on the modification
  - Stop when your modified solution is equal to greedy one

Example: Optimal Merging

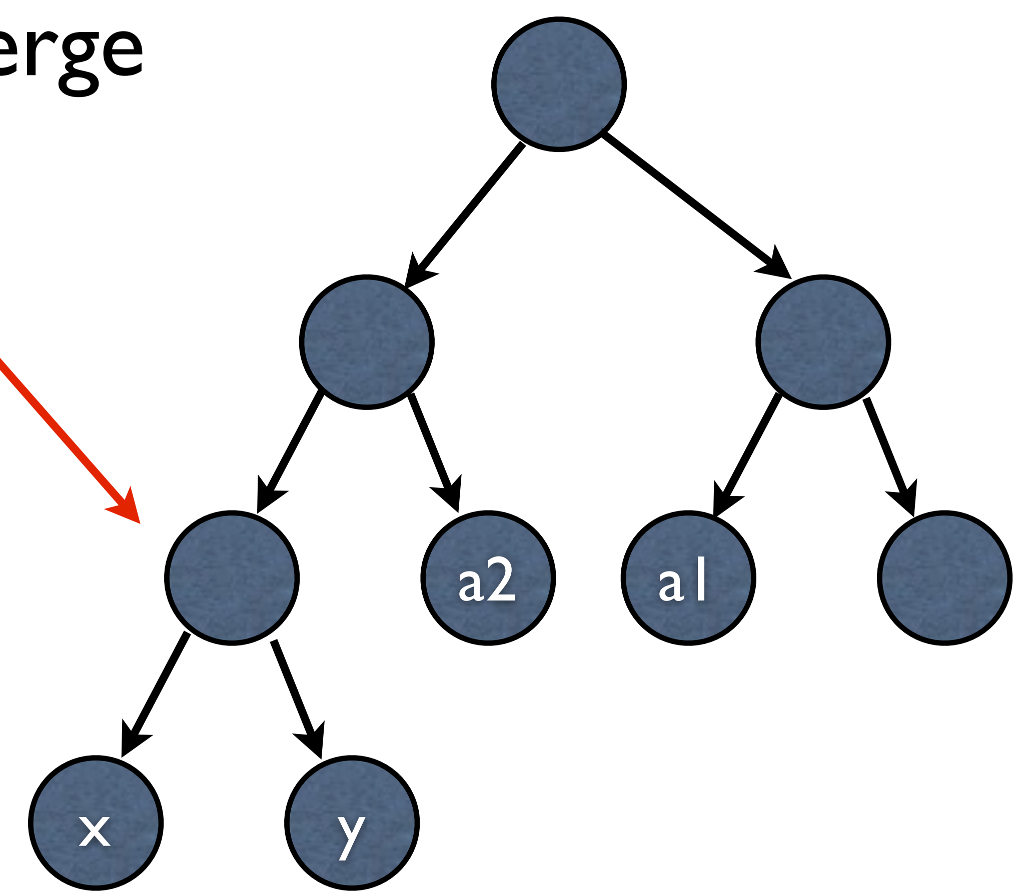
- Notice that, any sequence of merges implicitly creates a tree
- Leaves of the tree are starting arrays
- Root of the tree is the final array



Example: Optimal Merging

Sketch of the proof (Exchange argument)

- Let us look at the tree of optimal solution
- Denote by a1 and a2 two shortest leaves (arrays)
- Now, look at the “deepest” merge



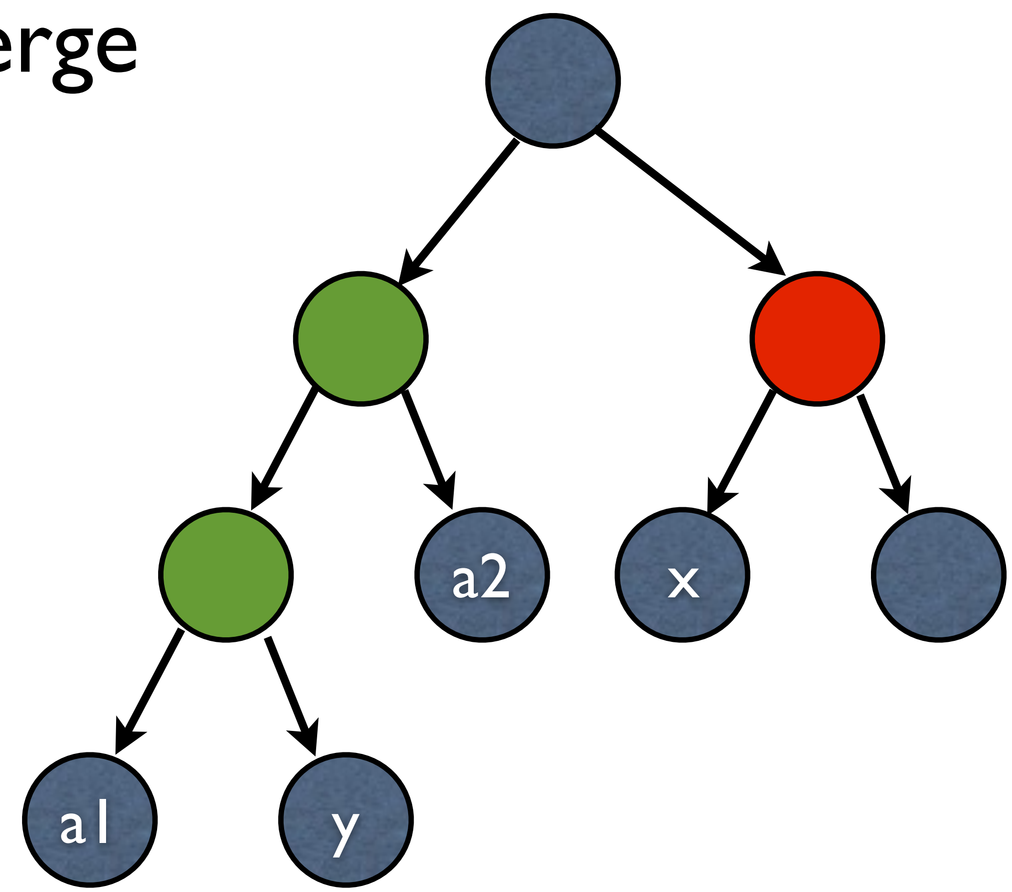
Example: Optimal Merging

Sketch of the proof (Exchange argument)

- Let us look at the tree of optimal solution
- Denote by a1 and a2 two shortest leaves (arrays)
- Now, look at the "deepest" merge

Let $d := x - a_1 \geq 0$

Because of the swap, some merges are cheaper by d, and some more expensive by d

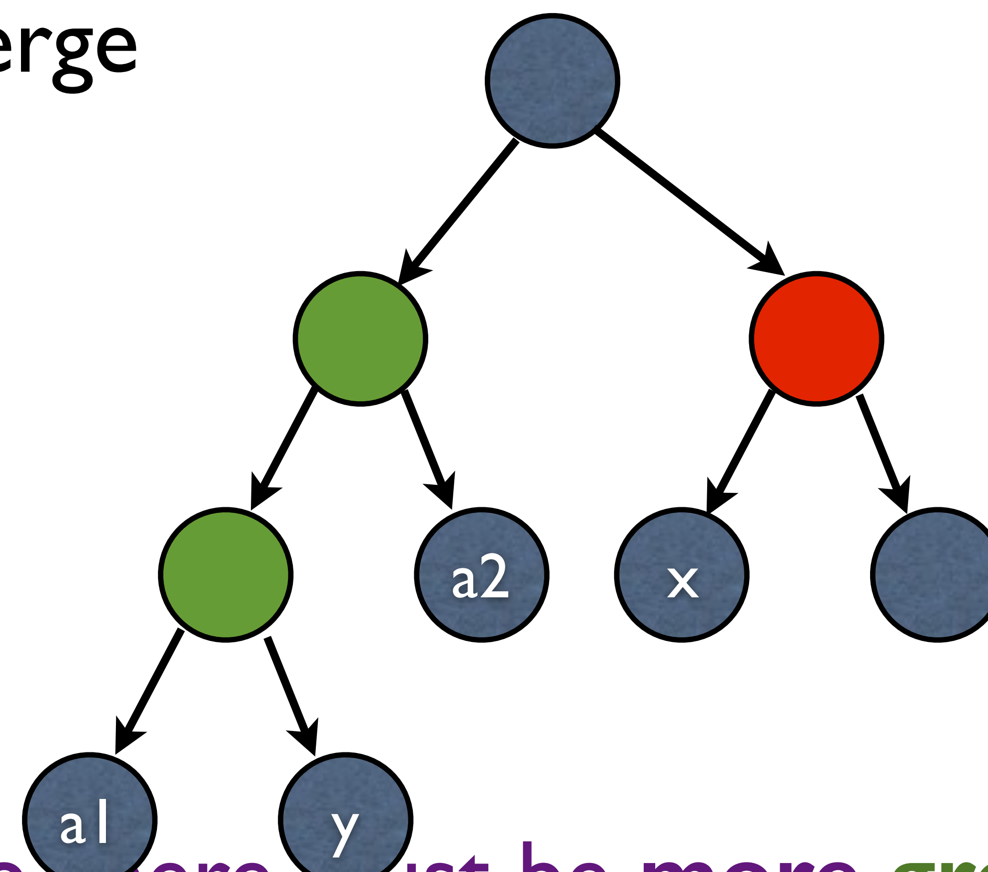


Example: Optimal Merging

Sketch of the proof (Exchange argument)

- Let us look at the tree of optimal solution
- Denote by a1 and a2 two shortest leaves (arrays)
- Now, look at the "deepest" merge

Let $d := x - a_1 \geq 0$



Because of the swap, some merges are cheaper by d, and some more expensive by d

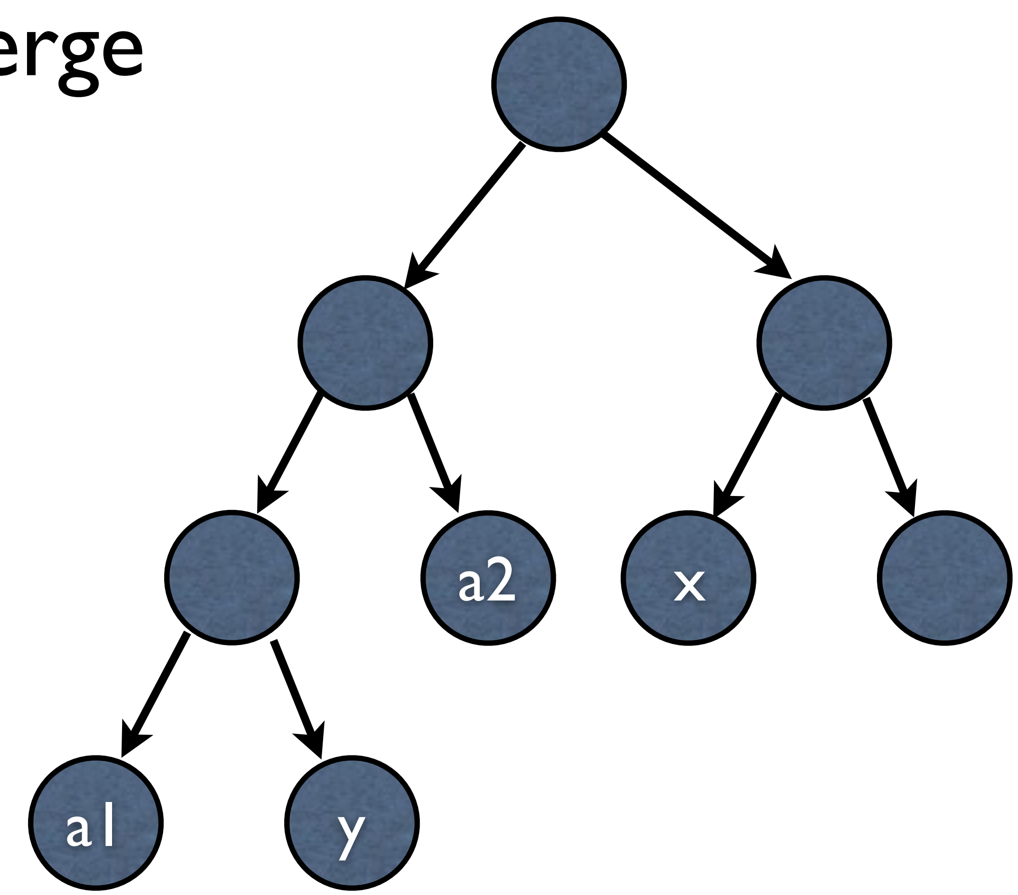
But as x was at **deepest merge**, there must be **more green** than **red** merges.

Example: Optimal Merging

Sketch of the proof (Exchange argument)

- Let us look at the tree of optimal solution
- Denote by a1 and a2 two shortest leaves (arrays)
- Now, look at the "deepest" merge

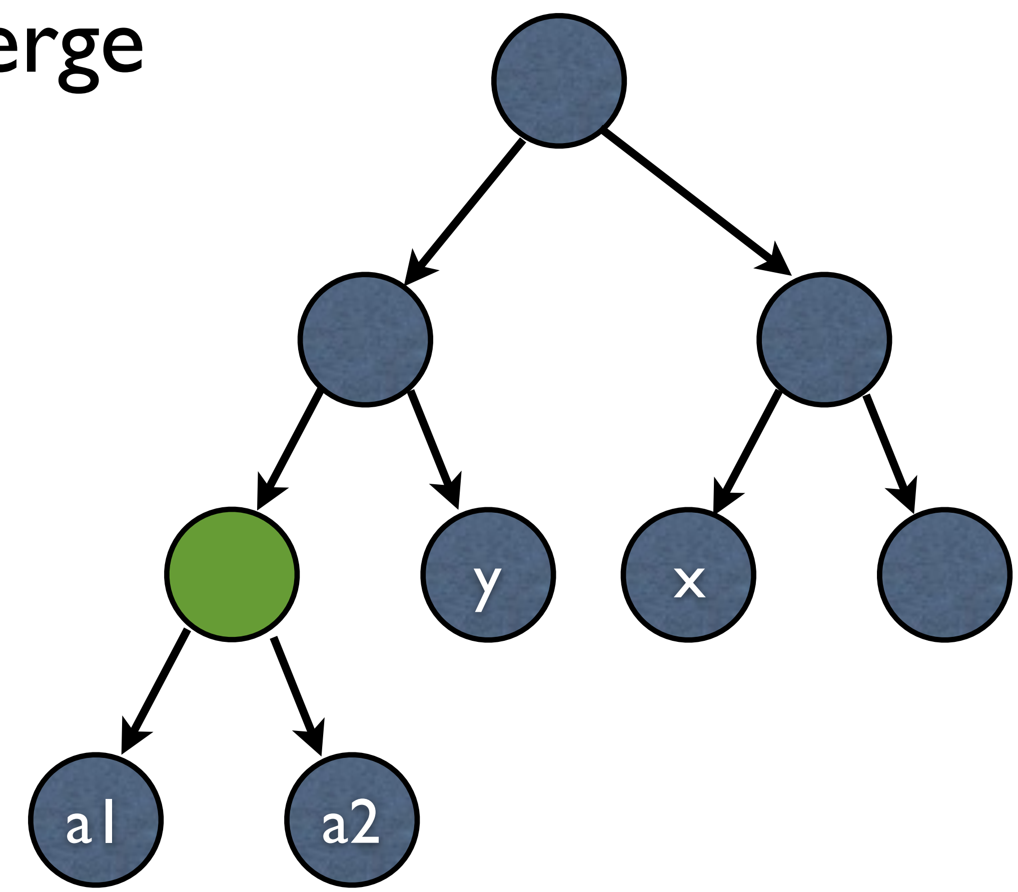
Now, we repeat same for a2



Example: Optimal Merging

Sketch of the proof (Exchange argument)

- Let us look at the tree of optimal solution
- Denote by a1 and a2 two shortest leaves (arrays)
- Now, look at the "deepest" merge



Now, we repeat same for a2

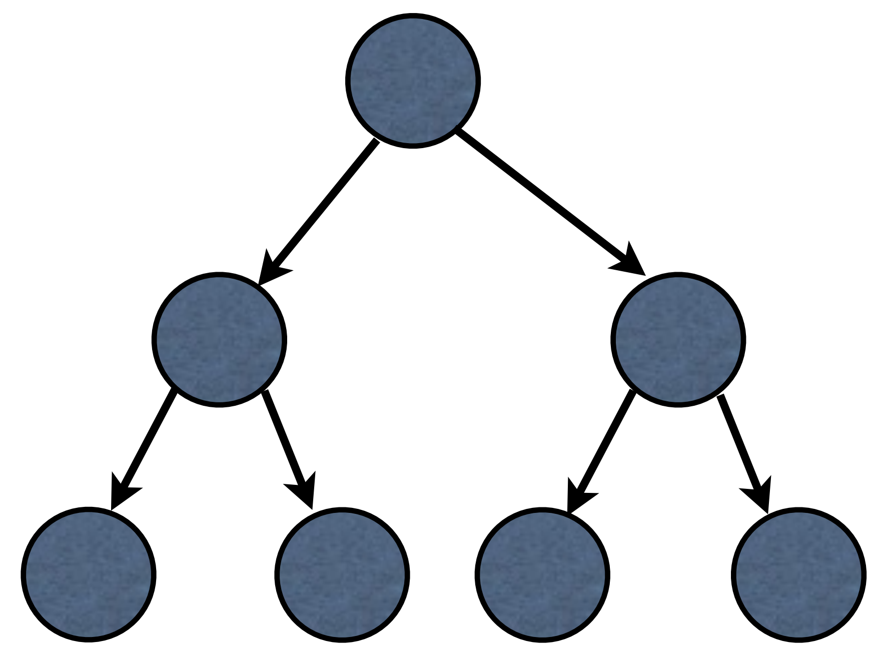
This tree has the merge(a1, a2) which was the first merge of greedy alg.

Example: Optimal Merging

Sketch of the proof (Exchange argument)

- Let us look at the tree of optimal solution
- Denote by a1 and a2 two shortest leaves (arrays)
- Now, look at the "deepest" merge

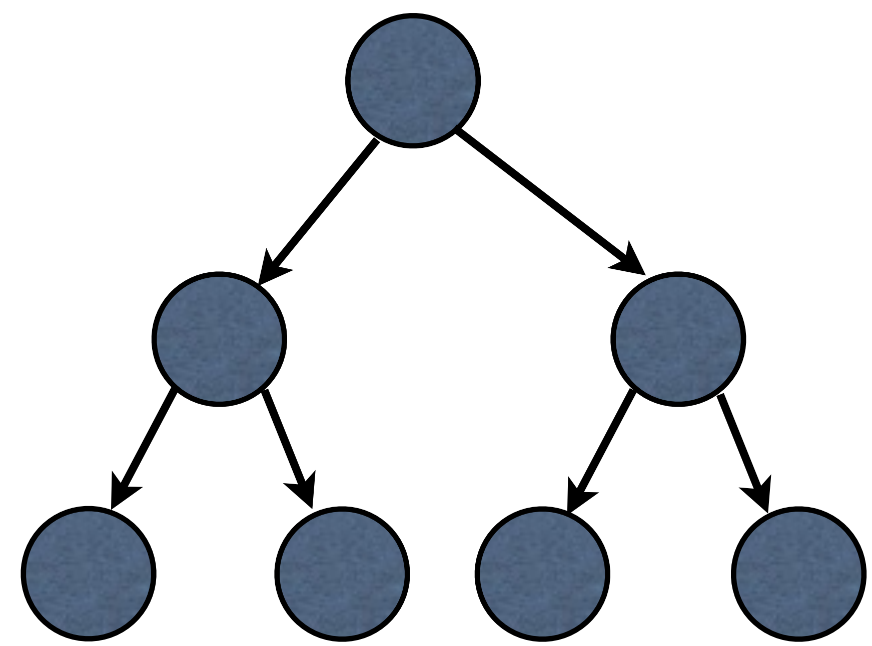
We forget about merge(a1, a2) and repeat the same logic on the tree that is left



Example: Optimal Merging

Sketch of the proof (Exchange argument)

- Let us look at the tree of optimal solution
- Denote by a1 and a2 two shortest leaves (arrays)
- Now, look at the "deepest" merge



We forget about merge(a1, a2) and repeat the same logic on the tree that is left

In the end we get the tree that has all the merges as greedy, and is at least as good as optimal!

Example: Interval Scheduling

- Your CPU needs to execute n jobs, described by time intervals $[s_1,f_1],\ldots,[s_n,f_n]$
- Job i starts at time s_i and finishes at time f_i
- Two jobs are incompatible if their intervals overlap
- What is the maximum number of mutually compatible jobs?

Example: Interval Scheduling

Approach to solving

- Come up with a property by which you will pick jobs one by one.
- This property should give you a measure of the locally optimal job.

Example: Interval Scheduling

Approach to solving

- Come up with a property by which you will pick jobs one by one.
- This property should give you a measure of the locally optimal job.

```
J = jobs sorted by the property
G = empty set
for i = 1 to |J| {
  if J[i] is in conflict with a job in G
    continue;
  else
    add J[i] to G
}
output size of G as result
```

Example: Interval Scheduling

Approach to solving

Natural candidates:

- Earliest start time - Consider jobs with ascending s_i.
- Earliest finish time - Consider jobs with ascending f_i.
- Shortest length - Consider jobs with ascending $f_i - s_i$.
- Fewest conflicts - For each job i, count the number of conflicts with other jobs c_i. Consider jobs with ascending c_i.

Example: Interval Scheduling

Approach to solving

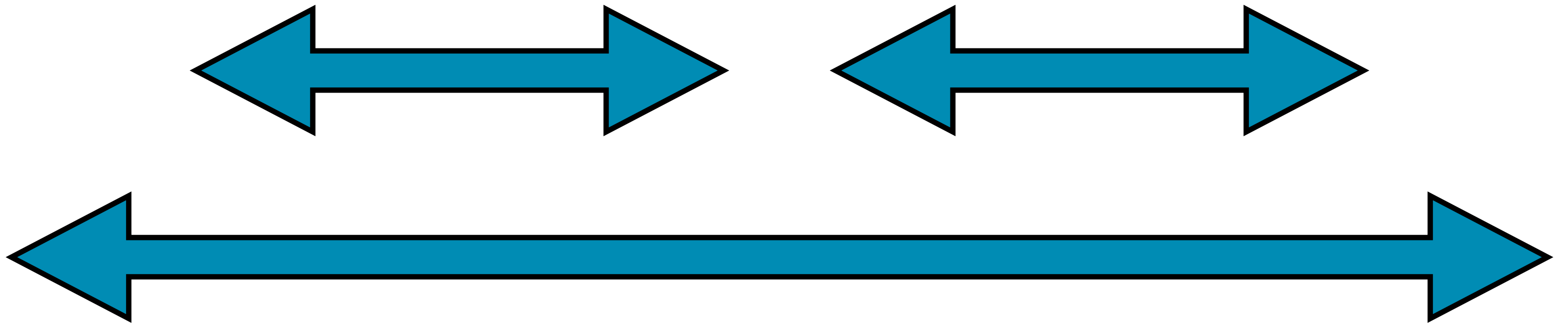
Earliest start time
Earliest finish time
Shortest length
Fewest conflicts

Which one do you think will work?

Example: Interval Scheduling

Approach to solving

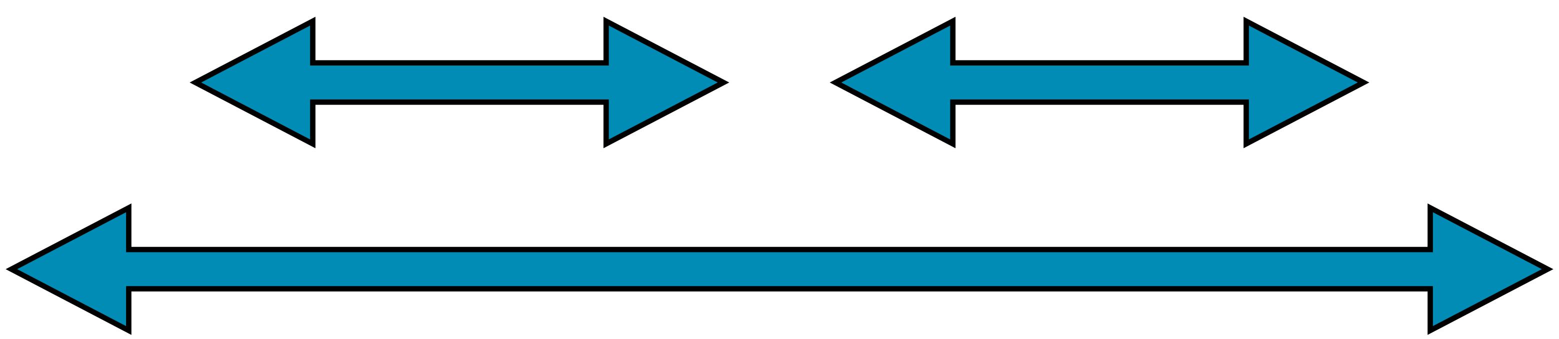
Earliest start time property.



Example: Interval Scheduling

Approach to solving

Earliest start time property.

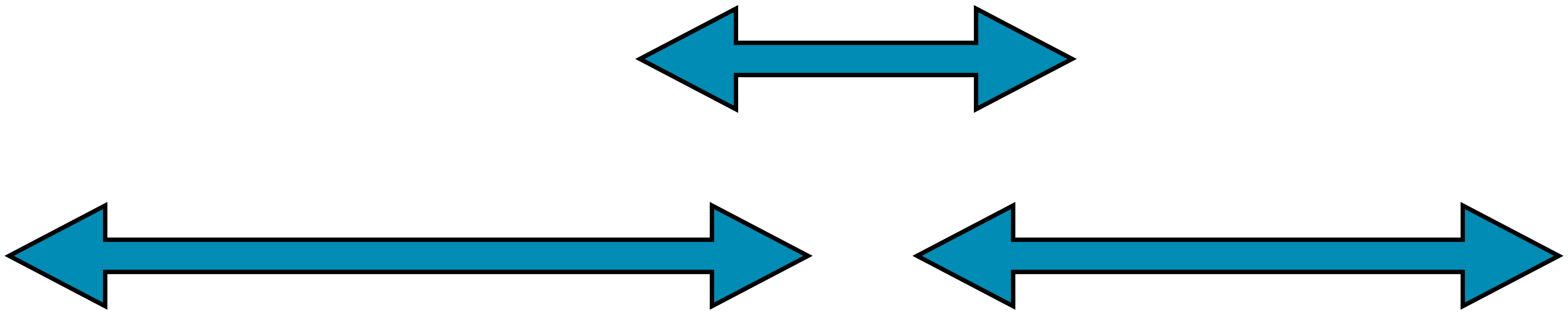


WRONG!

Example: Interval Scheduling

Approach to solving

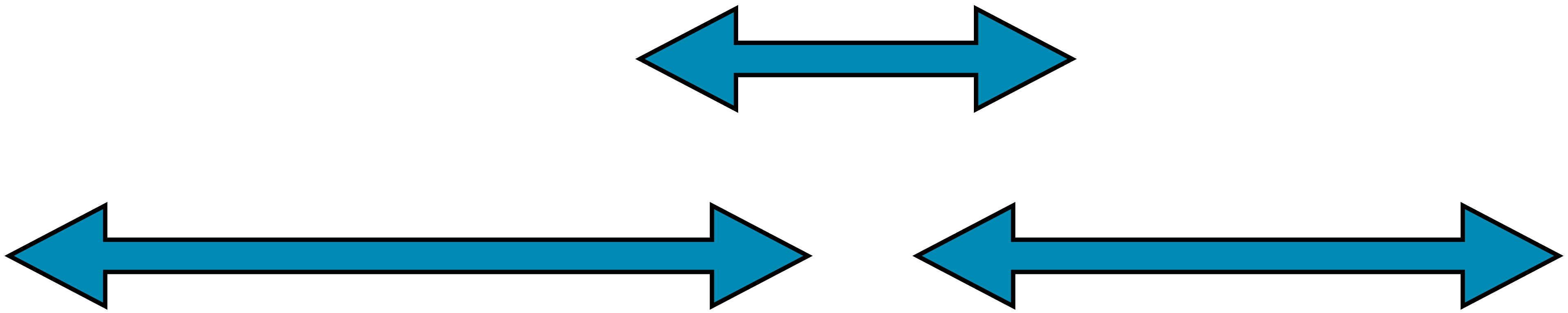
Shortest length



Example: Interval Scheduling

Approach to solving

Shortest length

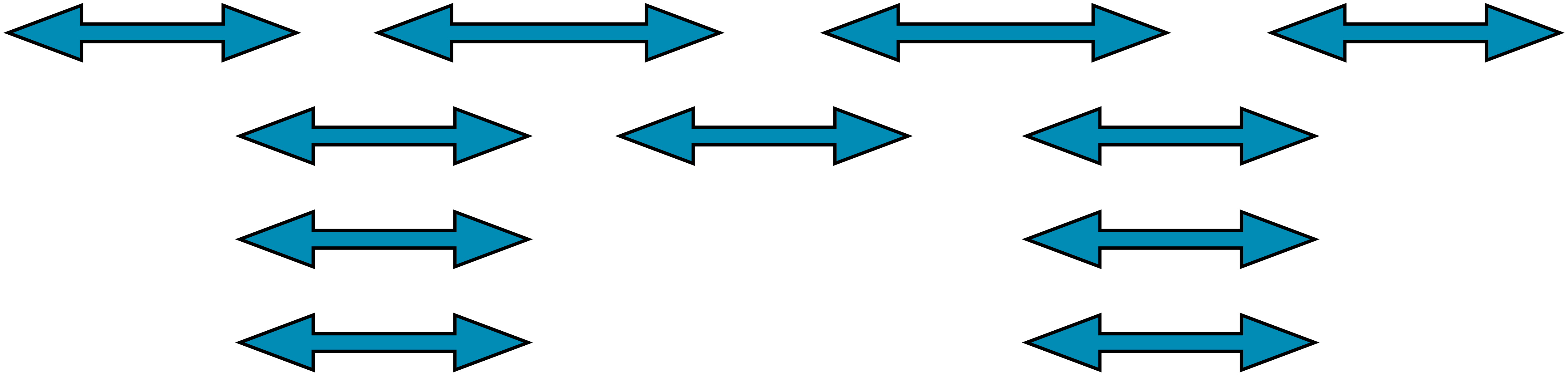


WRONG!

Example: Interval Scheduling

Approach to solving

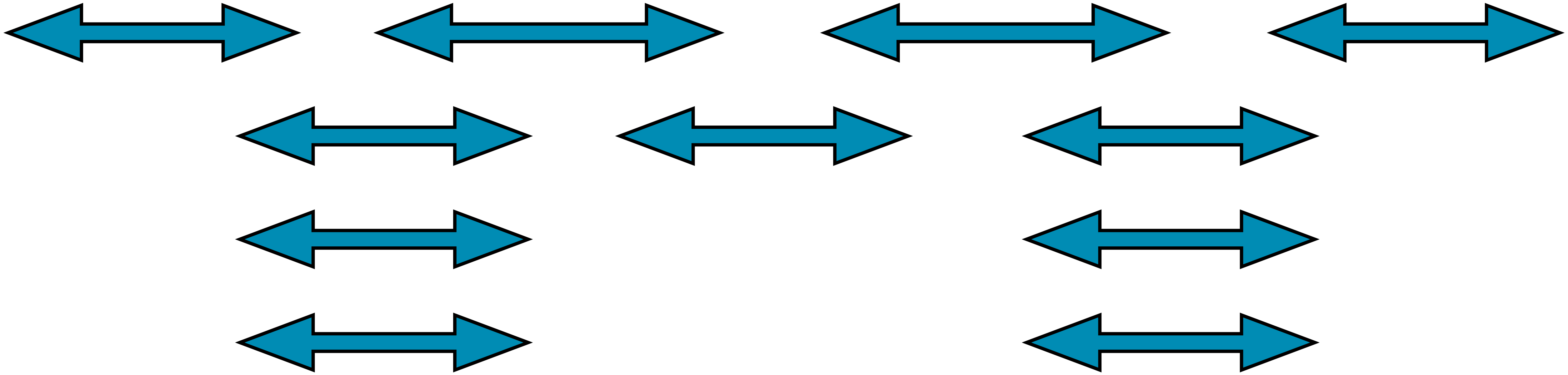
Fewest conflicts



Example: Interval Scheduling

Approach to solving

Fewest conflicts



WRONG!

Example: Interval Scheduling

Approach to solving

Earliest finish time - sketch of the proof

- Assume S is a set of intervals in the optimal solution
- Let $g_1, \ldots, g_k$ be all k jobs the greedy algorithm would select, ordered by the earliest finish time
- If g_1 in S, then we are good
- If g_1 is not in S, then there are some jobs in S in conflicting with it. However, there can be only one job in S in conflict with job g_1, denoted by c_1. Why?

Example: Interval Scheduling

Approach to solving

Earliest finish time - sketch of the proof

- Let S_1 be a set we get by removing job c_1 from S and inserting job g_1, i.e. $S_1 = S \setminus \{c_1\} \cup \{g_1\}$
- Note that $|S_1| = |S|$
- Now, consider g_2 . If g_2 is not in S_1, then there is only one job c_2 in conflict with g_2 .
- Let $S_2 = S_1 \setminus \{c_2\} \cup \{g_2\}$
- Repeat this until you inserted all the jobs from the greedy solution $g_1, \ldots, g_k \in S_k, |S_k| = |S|$

Bad Example: Knapsack

Warning: Many problems don't have a greedy solution!

- ATM has to to return you a change value of R.
- It has bills of values $v_1, v_2, \ldots, v_k$ (infinite amount of each).
- Goal: What is the minimum number of bills needed to return the change value?

Bad Example: Knapsack

- ATM has to to return you a change value of R.
- It has bills of values $v_1, v_2, \ldots, v_k$ (infinite amount of each).
- Goal: What is the minimum number of bills needed to return the change value?

Greedy algorithm:

```
M = ATM bills sorted by value (decreasing)

for i = 1 to |M| {
  take R \ M[i] bills of value M[i]
  R = R mod M[i]
  if R = 0 break
}
```

Bad Example: Knapsack

- ATM has to to return you a change value of R.
- It has bills of values $v_1, v_2, \ldots, v_k$ (infinite amount of each).
- Goal: What is the minimum number of bills needed to return the change value?

Greedy algorithm:

```
M = ATM bills sorted by value (decreasing)

for i = 1 to |M| {
  take R \ M[i] bills of value M[i]
  R = R mod M[i]
  if R = 0 break
}
```

WRONG!

Bad Example: Knapsack

- ATM has to to return you a change value of R.
- It has bills of values $v_1, v_2, \ldots, v_k$ (infinite amount of each).
- Goal: What is the minimum number of bills needed to return the change value?

Greedy algorithm:

$R = 41$

$v1 = 25, v2 = 10, v3 = 1$

Output: 25, 10, 1, 1, 1, 1, 1, 1 (8 bills)

But optimal is: 10, 10, 10, 10, 1 (5 bills)

Greedy Algorithms

What did we learn?

- Some, but not all, problems can be solved with greedy approach
- Finding a property by which we should greedily select can be non-obvious
- We can prove that our greedy idea works with exchange argument