Algoritmiek, bijeenkomst 3

Mathijs de Weerdt

Today

- Introduction
- Greedy
- Divide and Conquer (very briefly)
- Dynamic programming

Slides with thanks to Kevin Wayne and Pearson Education
(made available together with the book "Algorithm Design" by Kleinberg & Tardos)

Planning komende “bijeenkomsten”

Bijeenkomst:

4. 31-3. Greedy. Bespreken opgaven met *Mathijs via Skype.*
5. 21-4. Divide & Conquer, Network Flow. *Tomas.* (& Bespreken DFS)
6. 5-5? Divide & Conquer. Bespreken opgaven met *Paul/Tomas*.
7. 19-5. Dynamic programming. Bespreken opgaven met Mathijs. *Skype.*
8. 2-6. Network Flow. Bespreken opgaven met Tomas *via Skype.*
9. 16-6. Eigen les. Vragenuur. Challenges (in Utrecht)
10. 30-6. Exam

DomJudge: wachtwoord

Tussendoor vragen in het forum.

Skype id:

`mathijs.de.weerdt`

TU Delft

Beoordeling

Beoordeling Algoritmiek cursus:

- 5 programmeeropdrachten tijdens cursus
- Eigen les voorbereiden, geven in eigen klas, presenteren aan elkaar
- Toets met open vragen

TU Delft

Als je baas een app is

Deeleconomie

Chauffeurs bij Uber en koeriers bij Foodora en Deliveroo krijgen hun opdrachten via een algoritme. „Soms is het wel een gemis dat je niet gewoon met mensen kunt overleggen."

Wouter van Noort 28 oktober 2016

Jan doet het nu ongeveer drie maanden. De zelfstandig financieel adviseur, vijftiger, wilde vooral kijken of chauffeur zijn voor Uber iets voor hem was. „Het is superflexibel", zegt hij„Je zet de app aan en je kunt de weg op en geld verdienen." Jan, die niet met zijn achternaam in de krant wil, wil dit werk waarschijnlijk blijven doen zodat hij minder tijd hoeft te besteden aan zijn financiële adviespraktijk.

Als hij zijn app aanzet, bepaalt het algoritme van Uber welke ritten hij krijgt. Er is wel een optie om te weigeren, maar als hij dat te vaak doet, vervalt de omzetgarantie die Uber chauffeurs normaal gesproken biedt. „En zonder die omzetgarantie kun je er moeilijk van rondkomen." In Rotterdam, waar hij werkt, is die omzetgarantie op de meeste momenten 20 euro per uur. In steden waar de gemiddelde verdiensten hoger zijn, geeft Uber geen omzetgarantie.

Zijn 'acceptatieratio' moet minimaal 70 procent zijn. De afstand van de rit krijgt hij pas na het

Illustratie Tomas Schats

NRC, 28 oktober 2016

TU Delft

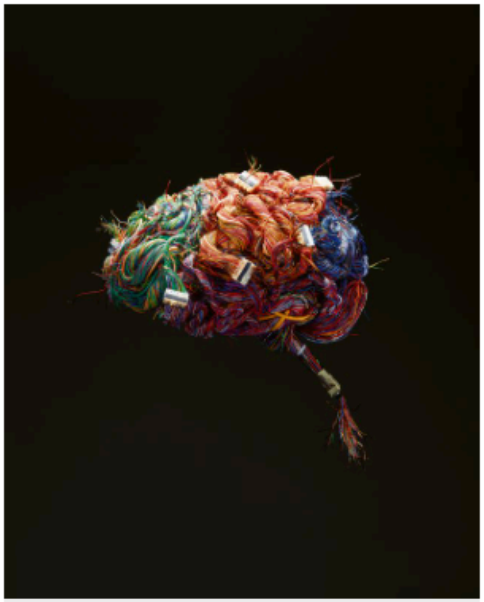
Kunstmatige intelligentie gaat regeren

Technologie

Wie alle data beheert, kan de samenleving runnen, met kunstmatige intelligentie. Google, Apple en Facebook liggen voor op Washington.

Wouter van Noort 4 november 2016

Niet toevallig is Barack Obama deze verkiezingsmaand gasthoofdredacteur van het toonaangevende technologieblad *Wired Magazine*. Technologie, en in het bijzonder kunstmatige intelligentie, gaat de komende jaren ongelooflijk veel veranderen, voorspelt de Amerikaanse president. En daar kan zijn opvolger maar beter goed op letten.



Een model van de menselijke hersenen, gemaakt van draden en stekkers.
Foto Getty Images

Obama waarschuwt onder meer voor de gevolgen van zelflerende computers die op de aandelenbeurzen zonder menselijk toezicht hun gang gaan en volstrekt onvoorspelbare koersbewegingen zullen veroorzaken. Het risico van ongekende volatiliteit op de financiële markten en zelfs manipulatie ligt volgens hem op de loer. Zelfrijdende auto's op basis van kunstmatige intelligentie gaan de komende jaren zorgen voor een revolutie in het vervoer, denkt hij. En als computers die uit zichzelf slimmer worden bepaalde banen overbodig gaan maken, moeten overheden serieus gaan nadenken over het verschaffen van een basisinkomen aan alle burgers, volgens Obama.

NRC, 4 november 2016

TU Delft

Learning Objectives: Algorithmic Paradigms

After this course you are able to design, analyze and implement algorithms in the following paradigms:

- Greedy.
- Divide-and-conquer.
- Dynamic programming.

By giving examples of problems and algorithms that solve them optimally.

We focus on algorithms and techniques that are useful in practice.
We also guarantee their correctness and optimality.

You will further develop your skills regarding

- critical thinking,
- implementing,
- proving correctness,
- proving time complexity, and
- problem-solving.

TU Delft

4 Greedy algorithms

Greedy

The efficient wine expert

- He would like C bottles of a specific champagne for new year's eve.
- There are n shops in the neighborhood who sell this.
- For each shop i, price p_i and the amount available c_i are given.

Determine where he should buy the C bottles.

Q. What would be a good greedy strategy to minimize costs?

1. Visit the shops in order of increasing price p_i.
2. Visit the shops in order of decreasing price p_i.
3. Visit the shops in order of increasing capacity c_i.
4. Visit the shops in order of decreasing capacity c_i.
5. I don't know.

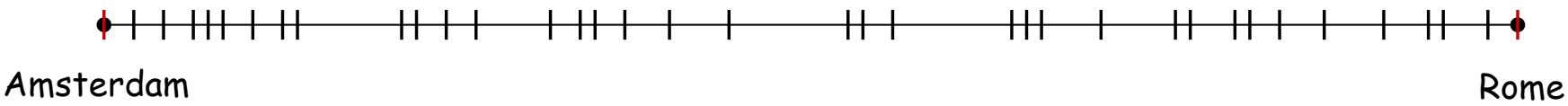
Greedy Buying Algorithm

```
Sort shops by price so that p1 ≤ p2 ≤ ... ≤ pn.

A ← ϕ
For k = 1 to n {
   A ← A ∪ {k}
   if (ck ≥ C)
      buyk ← C
      exit
   else
      buyk ← ck
      C ← C - ck
}
return A
```

Q. What is the tightest worst-case upper bound on the running time?

1. O(1)
2. O(log n)
3. O(n)
4. O(n log n)
5. $O(n^2)$
6. I don't know.

Selecting Breakpoints



Selecting breakpoints.

- Road trip from Amsterdam to Rome along fixed route of length L.
- Refueling stations at certain points b_0, b_1, …, b_n along the way.
- Let distances $\delta_i = (b_{i+1} - b_i)$
- Fuel capacity (distance) = C.
- Goal: makes as few refueling stops as possible.

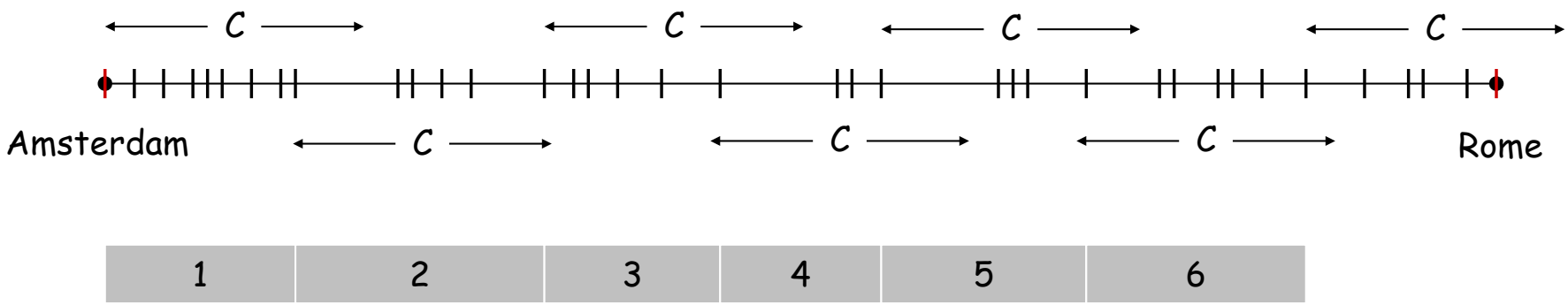
Q. What is a good strategy?

Selecting Breakpoints

Selecting breakpoints.

- Road trip from Amsterdam to Rome along fixed route.
- Refueling stations at certain points b_0, b_1, …, b_n along the way.
- Let distances $\delta_i = (b_i - {}_{bi-1})$
- Fuel capacity = C.
- Goal: makes as few refueling stops as possible.

Greedy algorithm. Go as far as you can before refueling.



Selecting Breakpoints: Greedy Algorithm

```
Sort breakpoints so that: 0 = b_0 < b_1 < b_2 < ... < b_n = L

S ← {0}   ←—  breakpoints selected
x ← 0     ←—  current location

while (x ≠ b_n)
   let p be largest integer such that b_p ≤ x + C
   if (b_p = x)
      return "no solution"
   x ← b_p
   S ← S ∪ {p}
return S
```

Q. What is the tightest worst-case upper bound on the run time?

1. O(1)
2. O(log n)
3. O(n)
4. O(n log n)
5. $O(n^2)$
6. I don't know.

Selecting Breakpoints: Greedy Algorithm

```
Sort breakpoints so that: 0 = b_0 < b_1 < b_2 < ... < b_n = L

S ← {0}   ←—  breakpoints selected
x ← 0     ←—  current location

while (x ≠ b_n)
   let p be largest integer such that b_p ≤ x + C
   if (b_p = x)
      return "no solution"
   x ← b_p
   S ← S ∪ {p}
return S
```

Implementation. O(n log n)

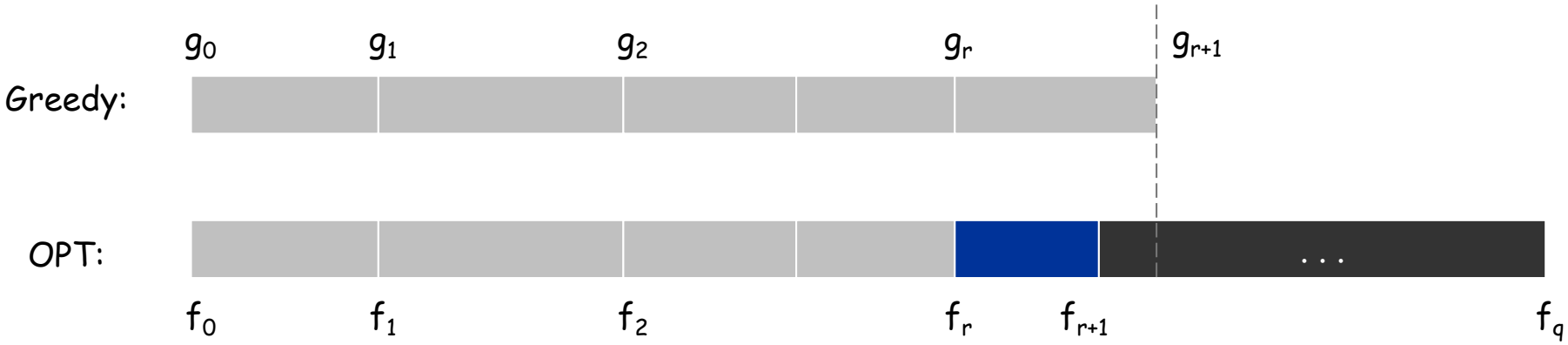
Selecting Breakpoints: Correctness

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal.
- *What we will do:*
  - Reason about some specific optimal solution that is as similar to the solution produced by the Greedy algorithm as possible.
  - Show that an optimal solution exists that is **even more similar**.

- Contradiction! So greedy is optimal.

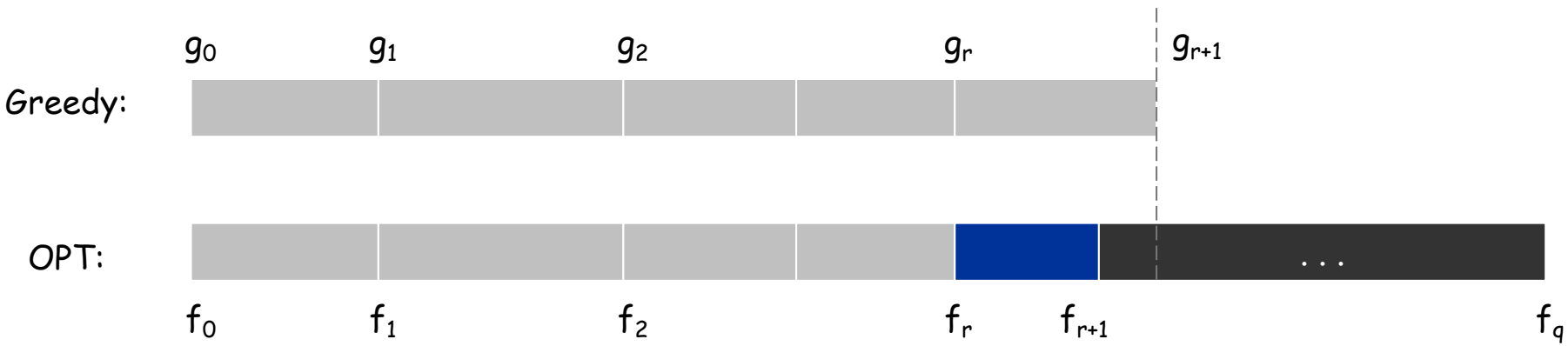


Selecting Breakpoints: Correctness

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal.
- Let $0 = g_0 < g_1 < \ldots < g_p = L$ denote set of breakpoints chosen by greedy.
- Let $0 = f_0 < f_1 < \ldots < f_q = L$ denote set of breakpoints in the optimal solution with $f_0 = g_0$, $f_1 = g_1$, . . . , $f_r = g_r$ for *largest possible* value of r.
- Note: $g_{r+1} \geq f_{r+1}$ by greedy choice of algorithm, so $g_{r+1} > f_{r+1}$
- …
- Contradiction! So greedy is optimal.



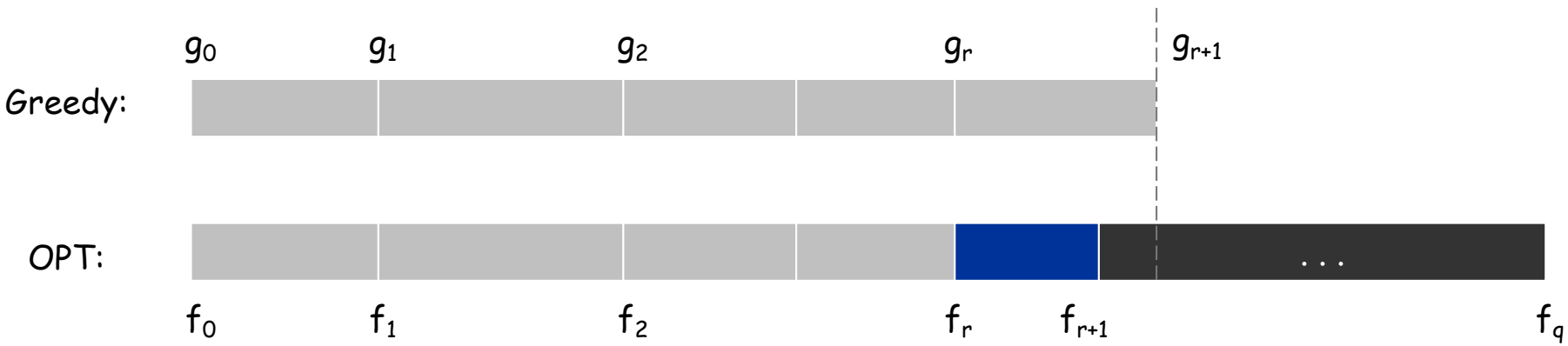
Selecting Breakpoints: Correctness

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal.
- Let $0 = g_0 < g_1 < \ldots < g_p = L$ denote set of breakpoints chosen by greedy.
- Let $0 = f_0 < f_1 < \ldots < f_q = L$ denote set of breakpoints in the optimal solution with $f_0 = g_0$, $f_1 = g_1$, . . . , $f_r = g_r$ for *largest possible* value of r.
- Note: $g_{r+1} \geq f_{r+1}$ by greedy choice of algorithm, so $g_{r+1} > f_{r+1}$

Q. Where is the contradiction?

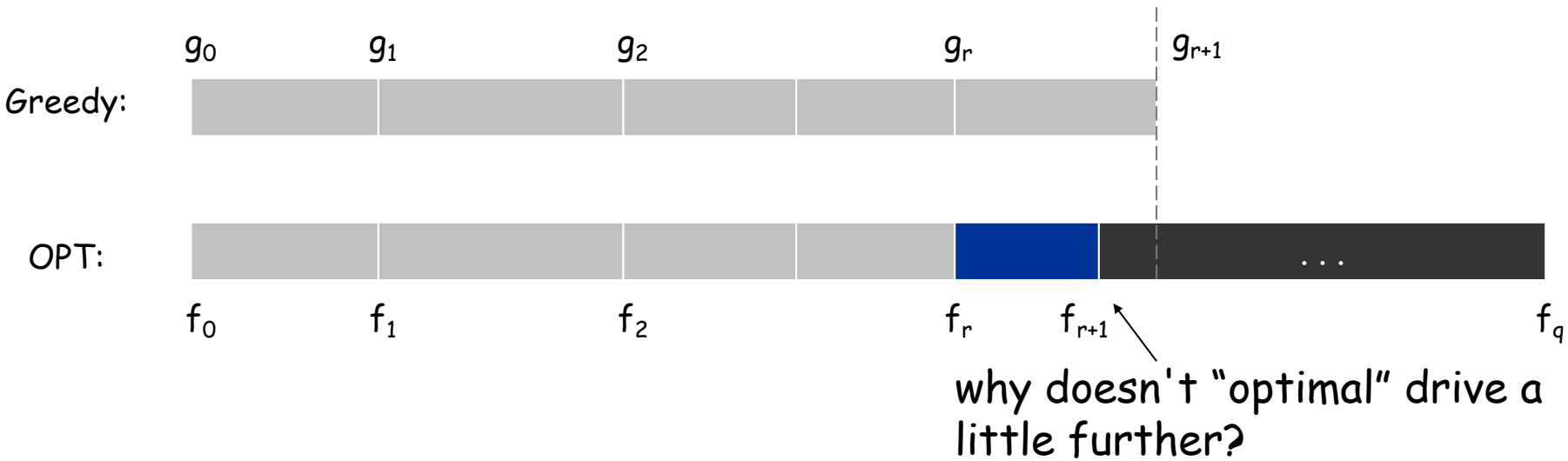


Selecting Breakpoints: Correctness

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal.
- Let $0 = g_0 < g_1 < \ldots < g_p = L$ denote set of breakpoints chosen by greedy.
- Let $0 = f_0 < f_1 < \ldots < f_q = L$ denote set of breakpoints in the optimal solution with $f_0 = g_0$, $f_1 = g_1$, . . . , $f_r = g_r$ for *largest possible* value of r.
- Note: $g_{r+1} \geq f_{r+1}$ by greedy choice of algorithm, so $g_{r+1} > f_{r+1}$

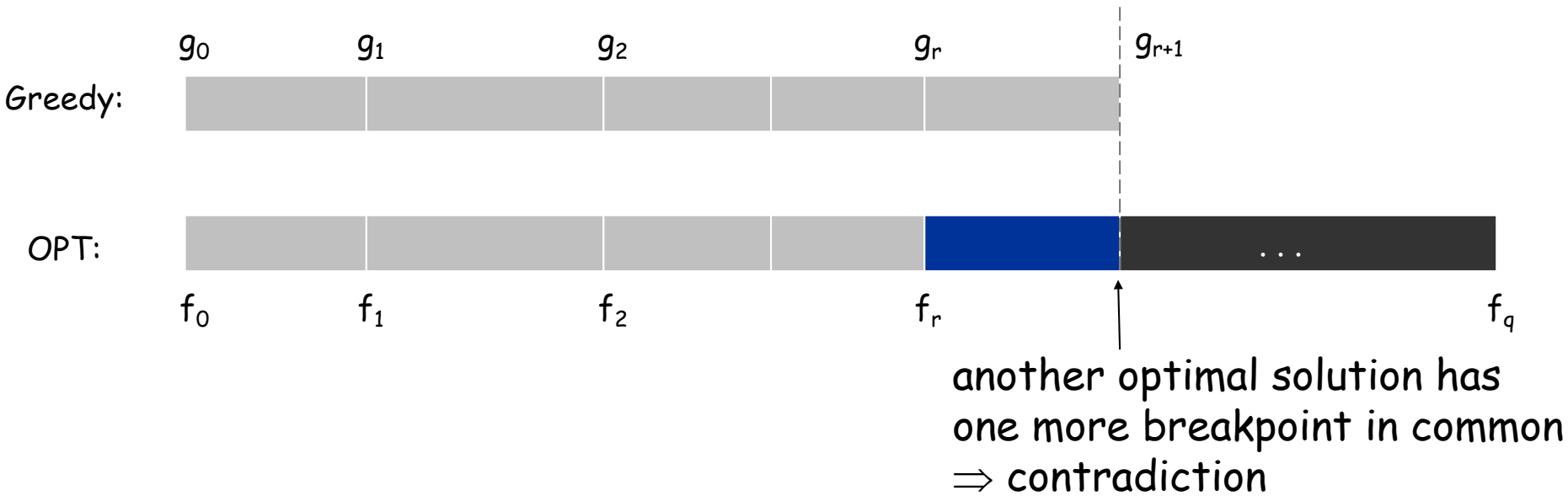


Selecting Breakpoints: Correctness

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal.
- Let $0 = g_0 < g_1 < \ldots < g_p = L$ denote set of breakpoints chosen by greedy.
- Let $0 = f_0 < f_1 < \ldots < f_q = L$ denote set of breakpoints in the optimal solution with $f_0 = g_0$, $f_1 = g_1$, . . . , $f_r = g_r$ for *largest possible* value of r.
- Note: $g_{r+1} \geq f_{r+1}$ by greedy choice of algorithm, so $g_{r+1} > f_{r+1}$



More problems with a greedy solution

Known greedy algorithms:

- Dijkstra (see if you understand the proof) (Ch 4.4)

- Interval Scheduling (Ch 4.1)
- Minimizing maximum lateness (Ch 4.2)
- Minimal Spanning Trees (Ch. 4.5)

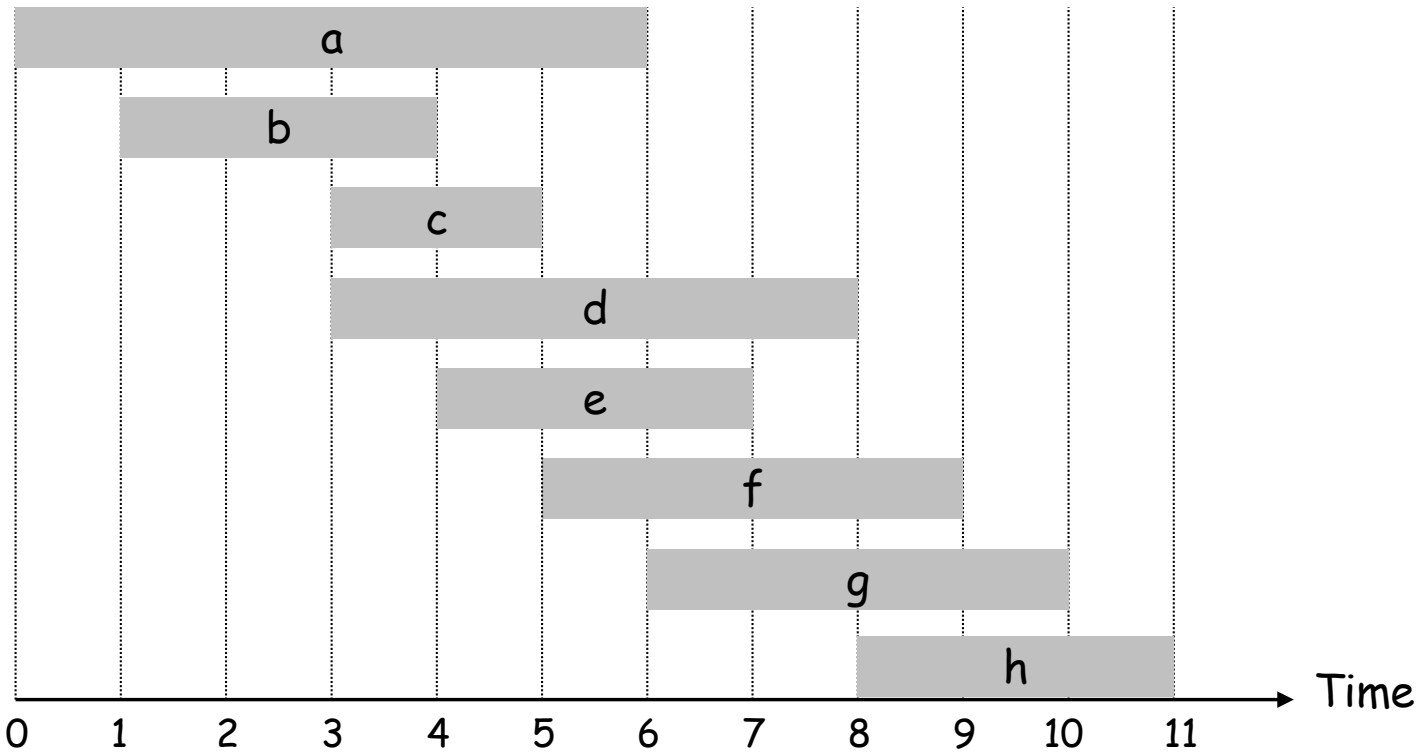
4.1 Interval Scheduling

Interval Scheduling

Interval scheduling (activity selection)

- Job j starts at s_k and finishes at f_k.
- Two jobs compatible if they don't overlap.

Select as many compatible intervals as possible.

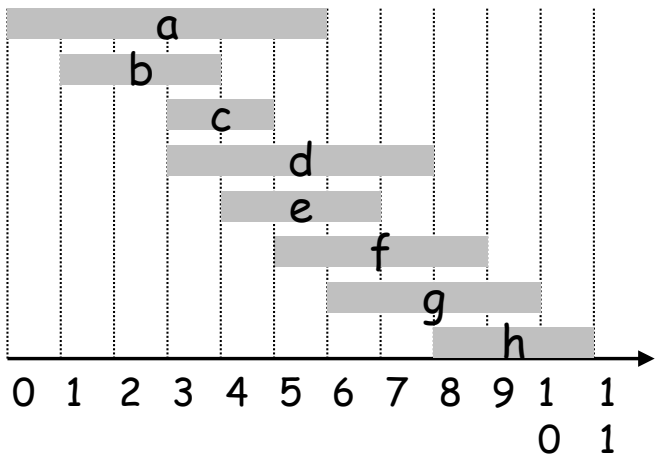


Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

Q. In which order should we consider the jobs?

A. [Earliest start time] Consider jobs in ascending order of start time s_k.
B. [Earliest finish time] Consider jobs in ascending order of finish time f_k.
C. [Shortest interval] Consider jobs in ascending order of interval length $f_k - s_k$.
D. [Fewest conflicts] For each job, count the number of conflicting jobs c_k. Schedule in ascending order of conflicts c_k.
E. I don't know.



Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

earliest start time?

shortest interval?

fewest conflicts?

Interval Scheduling: Greedy Algorithm

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

  jobs selected
A ← φ
For k = 1 to n {
   if (job k compatible with A)
      A ← A ∪ {k}
}
return A
```

4.2 Scheduling to Minimize Maximum Lateness

Scheduling to Minimizing Maximum Lateness

Minimizing lateness problem.

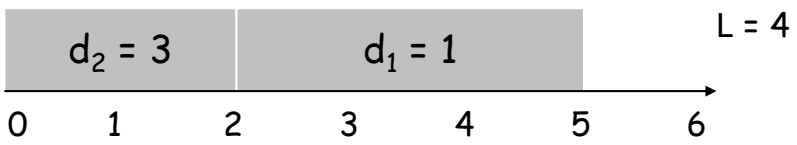
- Single resource processes one job at a time.
- Job j requires t_j units of processing time and is due at time d_j.

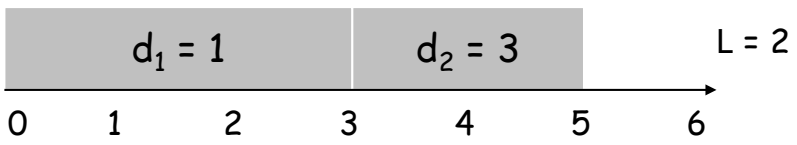
- If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.
- Lateness: $\ell_j = \max \{ 0, \ f_j - d_j \}$.
- Goal: schedule all jobs to minimize maximum lateness $L = \max \ell_j$.

Ex:

	1	2	
t_j	3	2	time required
d_j	1	3	deadline

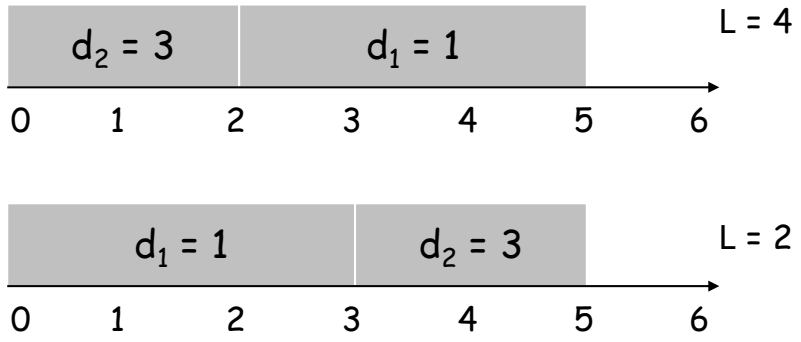
job number





Minimizing Maximum Lateness: Greedy Algorithms

	1	2	job number
t_j	3	2	time required
d_j	1	3	deadline



Q. In which order should we consider the jobs?

A. [Shortest processing time first] Consider jobs in ascending order of processing time t_j (least work first).

B. [Earliest deadline first] Consider jobs in ascending order of deadline d_j (nearest deadline).

C. [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$ (least time to start to make deadline).

D. I don't know.

Minimizing Maximum Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j (least work first).

	1	2
t_j	1	10
d_j	100	10

counterexample

- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$ (least time to start to make deadline).

	1	2
t_j	1	10
d_j	2	10

counterexample

Minimizing Maximum Lateness: Greedy Algorithm

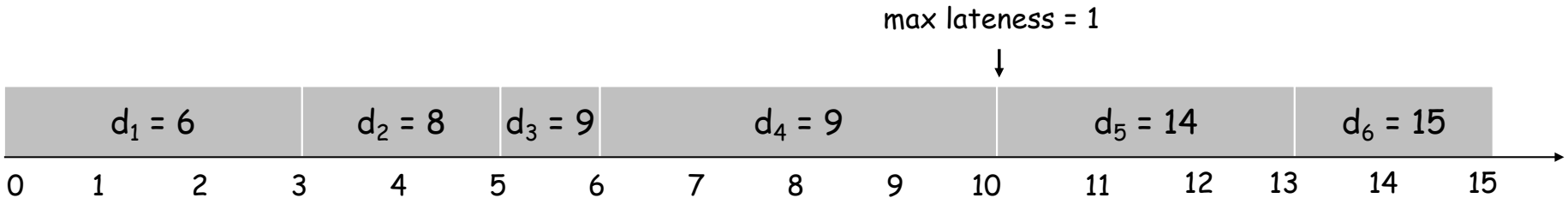
Greedy algorithm. Earliest deadline first.

```
Sort n jobs by deadline so that d1 ≤ d2 ≤ … ≤ dn

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + tj]:
       sj ← t
       fj ← t + tj
   t ← t + tj
output intervals [sj, fj]
```

Minimizing Maximum Lateness: Greedy Algorithm

Greedy algorithm. Earliest deadline first.

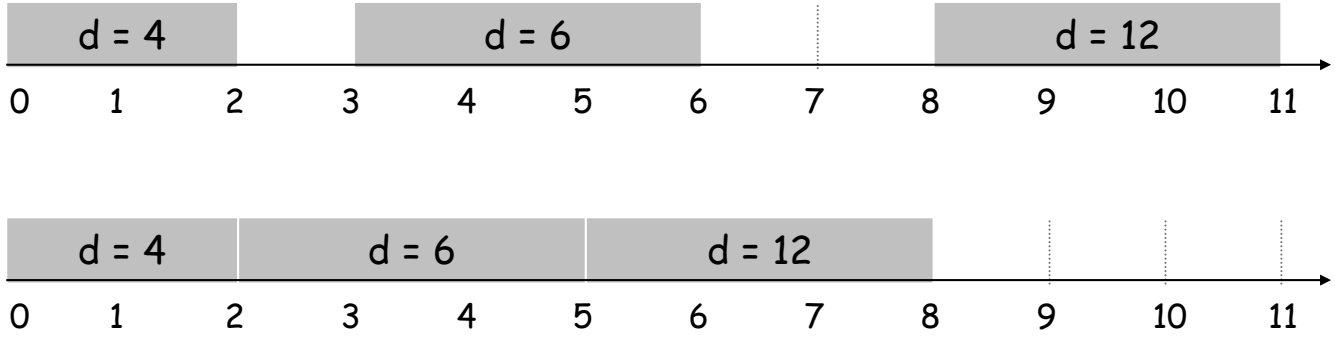
	1	2	3	4	5	6	← job number
t_j	3	2	1	4	3	2	← time required
d_j	6	8	9	9	14	15	← deadline



Observation. The greedy schedule has no idle time.

Minimizing Maximum Lateness: No Idle Time

Observation. There exists an optimal schedule with no idle time.

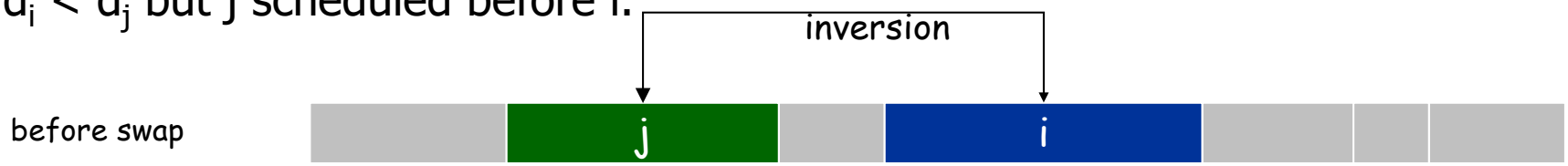


Prove that earliest-deadline-first greedy algorithm is optimal by exchange argument:

- Take an optimal schedule that is as much as Greedy as possible.
- Change more into greedy schedule without losing optimality…
- A contradiction!

Minimizing Maximum Lateness: Inversions

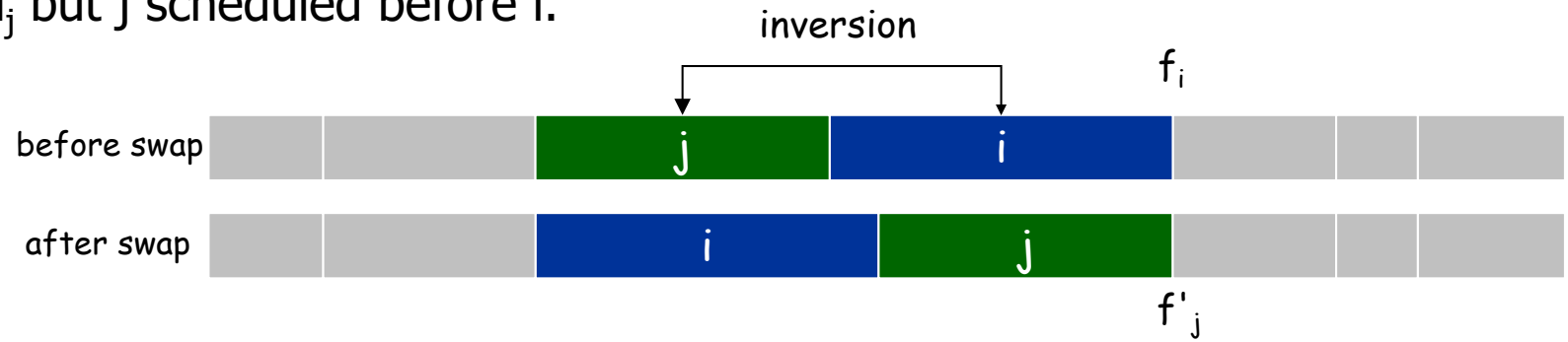
Def. An inversion in schedule S is a pair of jobs i and j such that:
$d_i < d_j$ but j scheduled before i.



Observation. Greedy schedule has no inversions.

Minimizing Maximum Lateness: Inversions

Def. An inversion in schedule S is a pair of jobs i and j such that: $d_i < d_j$ but j scheduled before i.

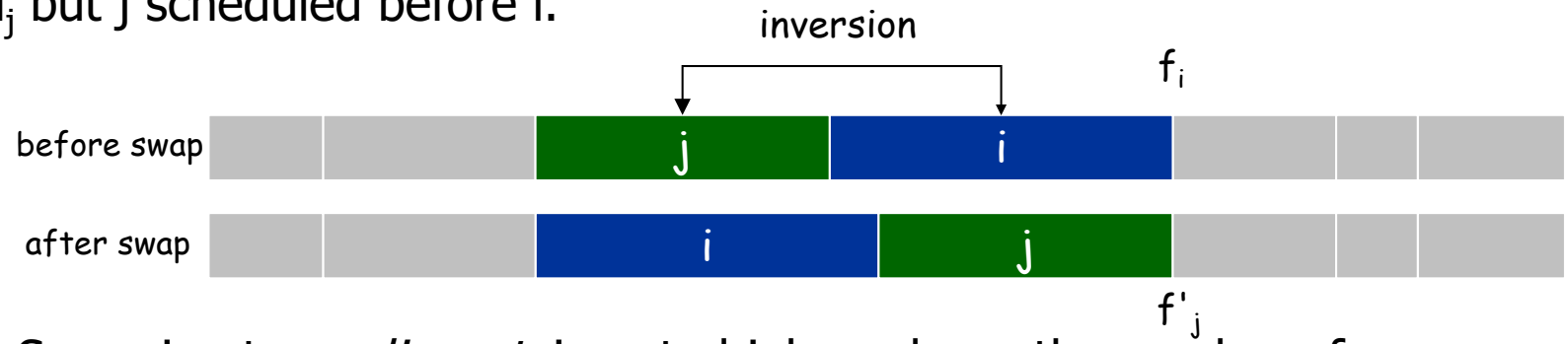


Q. Suppose two *adjacent*, inverted jobs j and i with $d_i < d_j$ are swapped, what happens to the maximum lateness?

A. The maximum lateness cannot become smaller.
B. The maximum lateness cannot become larger.
C. The maximum lateness stays the same.
D. I don't know.

Minimizing Maximum Lateness: Inversions

Def. An inversion in schedule S is a pair of jobs i and j such that: $d_i < d_j$ but j scheduled before i.



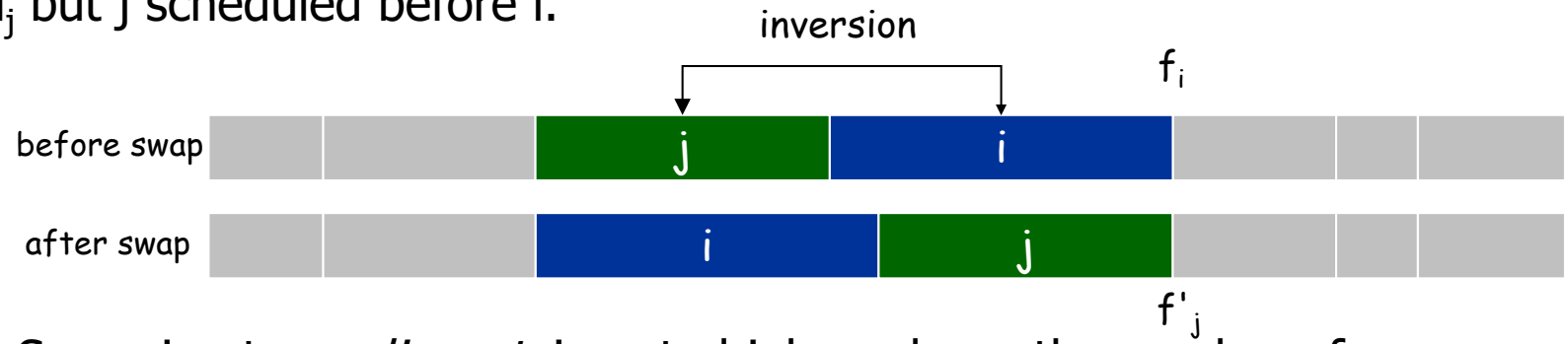
Claim. Swapping two *adjacent*, inverted jobs reduces the number of inversions by one and does not increase the maximum lateness.

Pf.

- … for all $k \neq i, j$
- … for i
- … for j

Minimizing Maximum Lateness: Inversions

Def. An inversion in schedule S is a pair of jobs i and j such that: $d_i < d_j$ but j scheduled before i.



Claim. Swapping two *adjacent*, inverted jobs reduces the number of inversions by one and does not increase the maximum lateness.

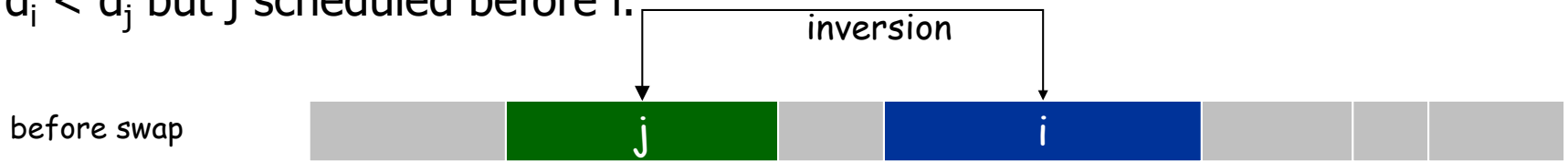
Pf. Let ℓ be the lateness before the swap, and let ℓ' be it afterwards.

- $\ell'_k = \ell_k$ for all $k \neq i, j$
  (lateness other jobs the same)
- $\ell'_i \leq \ell_i$
  (new lateness for i smaller)
- If job j is late:

$$
\begin{aligned}
\ell'_j &= f'_j - d_j && \text{(definition)} \\
&= f_i - d_j && (j \text{ finishes at time } f_i) \\
&\leq f_i - d_i && (d_i < d_j) \\
&\leq \ell_i && \text{(definition)}
\end{aligned}
$$

Minimizing Maximum Lateness: Inversions

Def. An inversion in schedule S is a pair of jobs i and j such that: $d_i < d_j$ but j scheduled before i.



Observation. If a schedule (with no idle time) has an inversion, then it has one with a pair of inverted jobs scheduled consecutively (*adjacent* jobs).

Pf.

- Suppose there is an inversion.
- There is a pair of jobs i and j such that: $d_i < d_j$ but j scheduled before i.
- Walk through the schedule from j to i.
- Increasing deadlines (= no inversions), at some point deadline decreases.

Minimizing Lateness: Analysis of Greedy Algorithm

Theorem. Greedy schedule S is optimal.

Pf. (by contradiction)

Idea of proof:

- Suppose S is not optimal.
- Take a optimal schedule S* that is as much like greedy.
- Change to look like greedy schedule (less inversions) without losing optimality.

Minimizing Lateness: Analysis of Greedy Algorithm

Theorem. Greedy schedule S is optimal.

Pf. (by contradiction)

Suppose S is not optimal.

Define S* to be an optimal schedule that has the fewest number of inversions (of all optimal schedules) and has no idle time.

Clearly S≠S*.

Q. How can we arrive at a contradiction?

Minimizing Lateness: Analysis of Greedy Algorithm

Greedy has no inversions.

All schedules without inversions have same lateness

(only diff is jobs with equal deadlines).

Theorem. Greedy schedule S is optimal.

Pf. (by contradiction)

Suppose S is not optimal.

Define S* to be an optimal schedule that has the fewest number of inversions (of all optimal schedules) and has no idle time.

Clearly S≠S*. Case analysis:

- If S* has no inversions, then the same lateness. Contradiction.
- If S* has an inversion, let i-j be an adjacent inversion.
  - swapping i and j does not increase the maximum lateness and strictly decreases the number of inversions
  - this contradicts definition of S*

So S is an optimal schedule. ▪

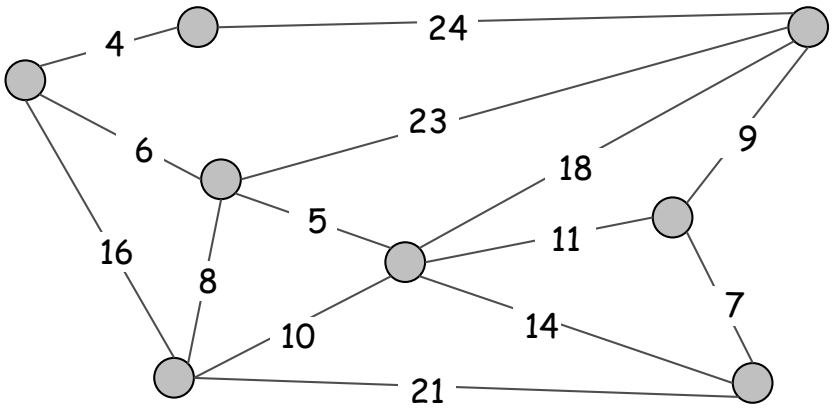
This proof can be found on pages 128-131.

4.5 Minimum Spanning Tree

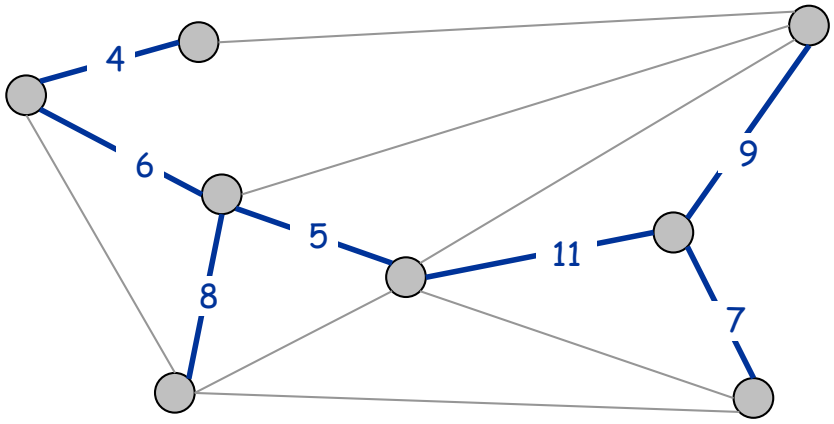
Minimum Spanning Tree

Minimum spanning tree. Given a connected graph G = (V, E) with edge weights c_e, an MST is a subset of the edges $T \subseteq E$ such that

- T is a tree
- T connects all vertices, and
- the sum of edge weights is minimized



$G = (V, E)$



T, $\Sigma_{e \in T} c_e = 50$

Q. How to find such a minimum spanning tree efficiently?

Greedy Algorithms

Q. How to find a minimum spanning tree efficiently?

A. For each vertex add cheapest edge, then join subtrees by adding cheapest edge.
B. Add the cheapest edge to T that has exactly one endpoint in T.
C. Add edges to T in ascending order of cost unless doing so would create a cycle.
D. Start with all edges from G in T. Delete edges in descending order of cost unless doing so would disconnect T.
E. All of the above.
F. None of the above.
G. I don't know.

Greedy Algorithms

Kruskal's algorithm. Start with $T = \varnothing$. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.

Reverse-Delete algorithm. Start with T = E. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T.

Prim's algorithm. Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T.

(Boruvka, 1926). Was first. (For each vertex add cheapest edge, then join subtrees by adding cheapest edge.)

Remark. *All these algorithms produce an MST*. We will prove this for the first three above using two general properties: the cut property and the cycle property.

Greedy Algorithms

Simplifying assumption. All edge costs c_e are distinct. (Lengths in example are related to their visual representation.)

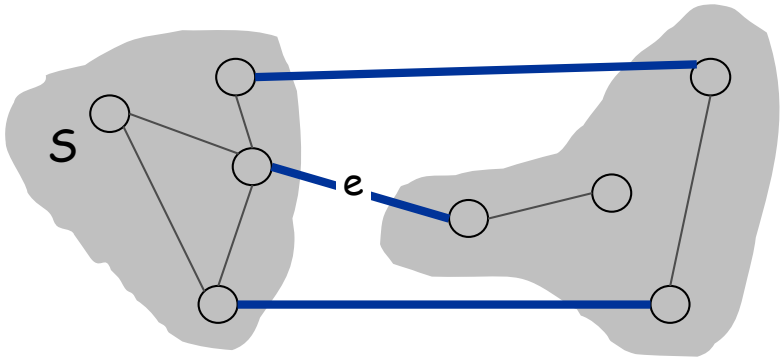
Q. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Should e be in an MST?

A. Yes

B. No

C. It depends.

D. I don't know.

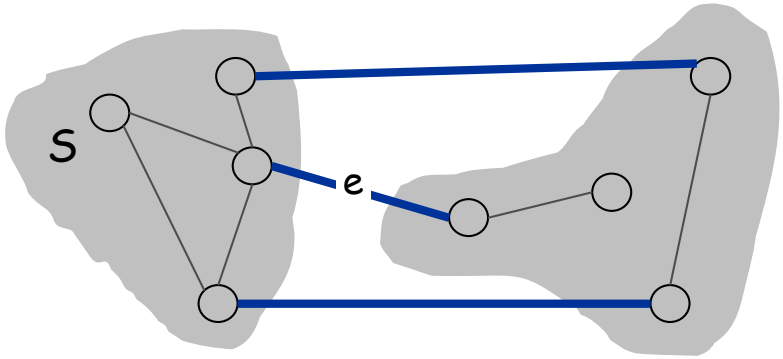


Greedy Algorithms

Simplifying assumption. All edge costs c_e are distinct. (Lengths in example are related to their visual representation.)

Q. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Should e be in an MST?

A. Yes (in very one) ➔ cut property



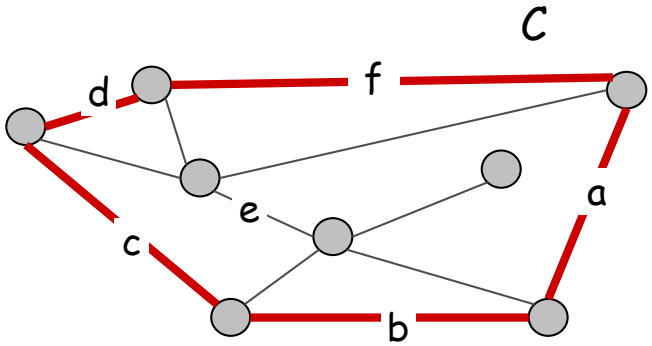
e is in every MST

Greedy Algorithms

Simplifying assumption. All edge costs c_e are distinct.

Q. Let C be any cycle, does a MST exist that has all of C's edges?
A. No.
Q. Which one should be not in the MST (Lengths in example are related to their visual representation.)?

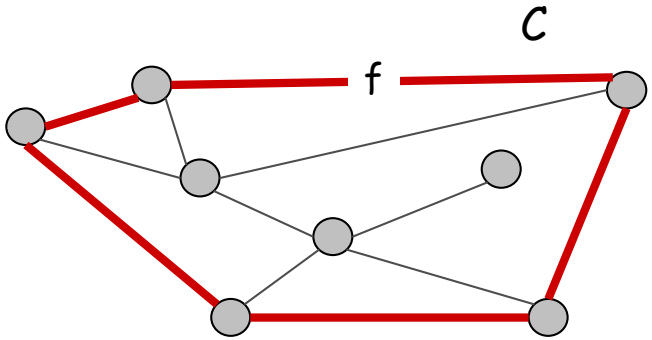
A. a
B. b
C. c
D. d
E. e
F. f



Greedy Algorithms

Simplifying assumption. All edge costs c_e are distinct.

Q. Let C be any cycle, does a MST exist that has all of C's edges?
A. No.
Q. Which one should be not in the MST (Lengths in example are related to their visual representation.)?
A. The max cost ➔ cycle property



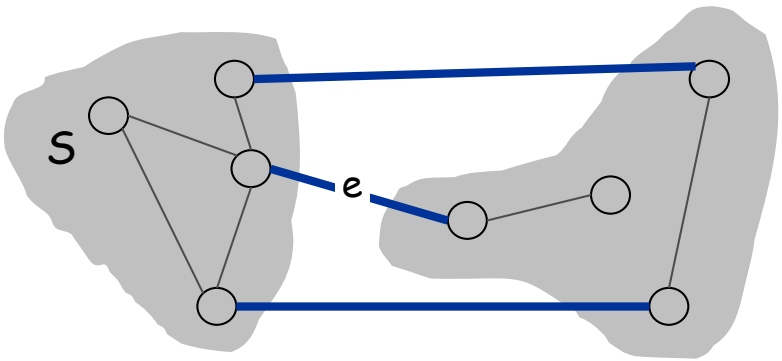
f is not in the MST

Greedy Algorithms

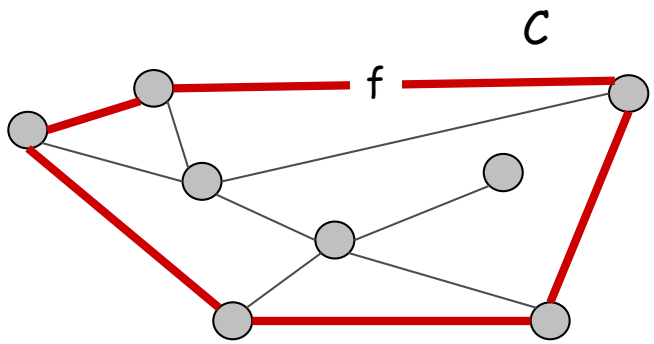
Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any cut, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e.

Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f.



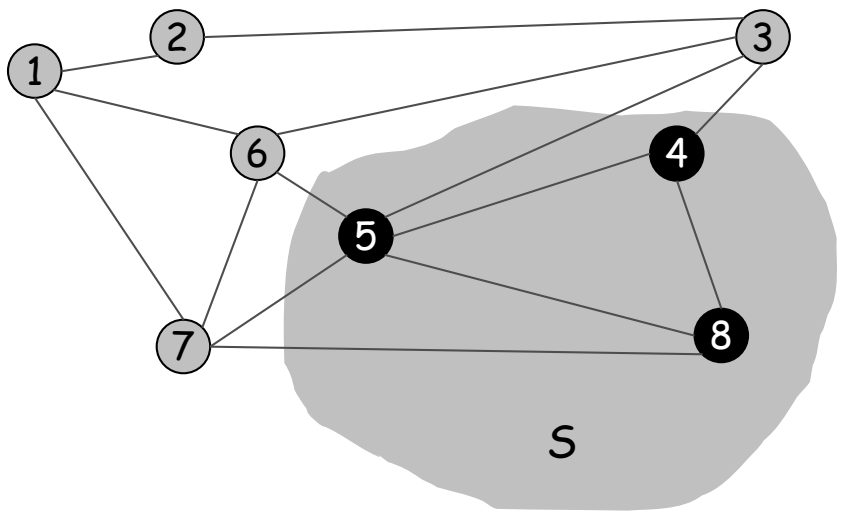
e is in the MST



f is not in the MST

Cut and cutset

Def. A cut is a subset of nodes S. (Note: compare to s-t cut.)

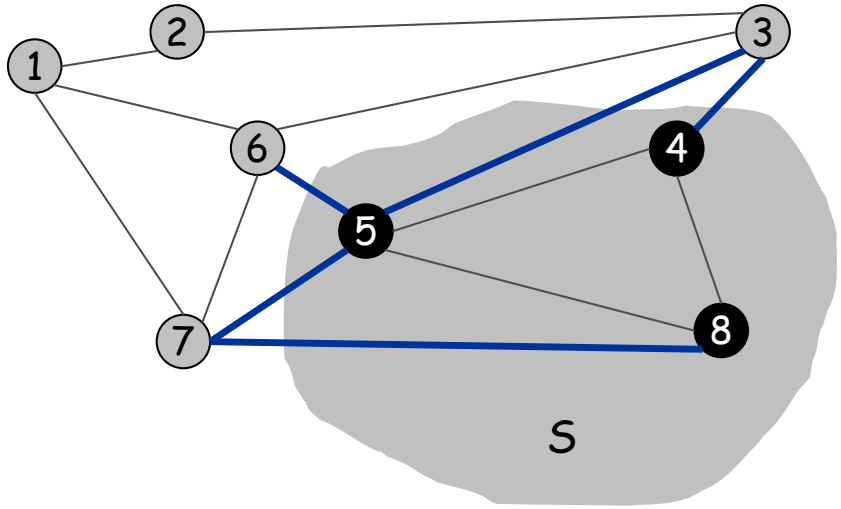


Cut S = { 4, 5, 8 }

Cut and cutset

Def. A cut is a subset of nodes S. (Note: compare to s-t cut.)

Def. A cutset D of a cut S is the subset of (cut)edges with exactly one endpoint in S.



Cut S = { 4, 5, 8 }
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8

Cycle-Cut Intersection

Q. Consider the intersection of a cycle and a cutset. How many edges are there in such an intersection?

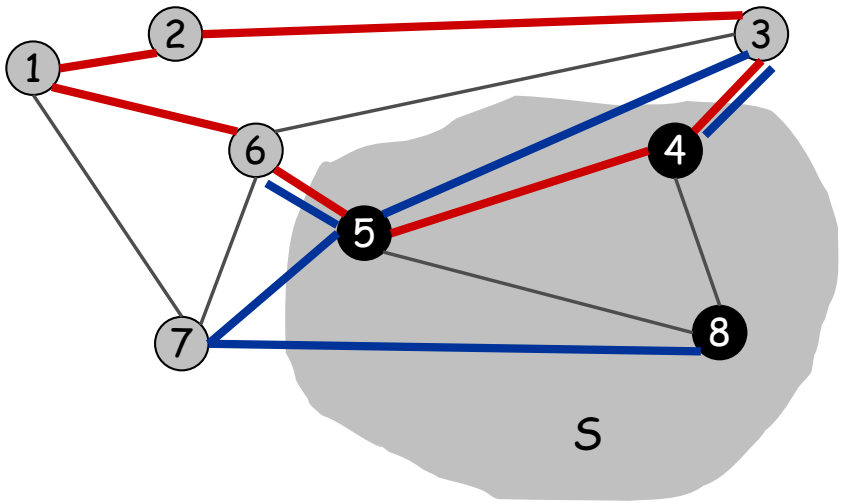
A. 1

B. 2

C. odd

D. even

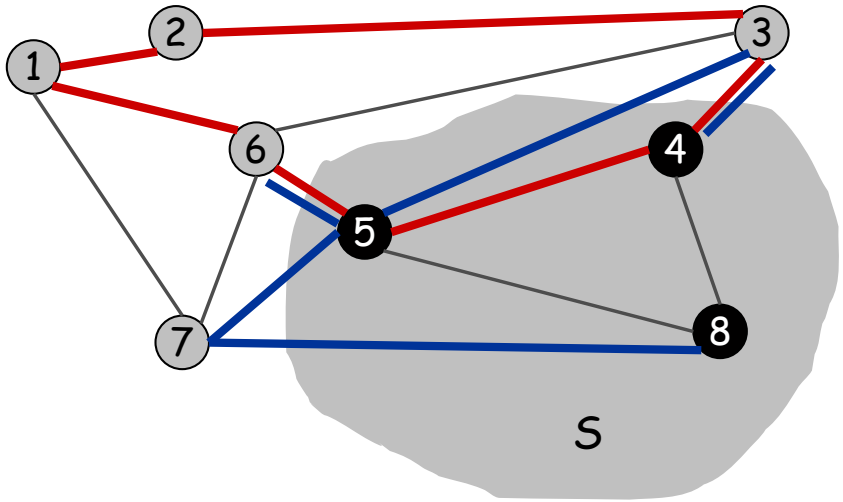
E. I don't know.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersection = 3-4, 5-6

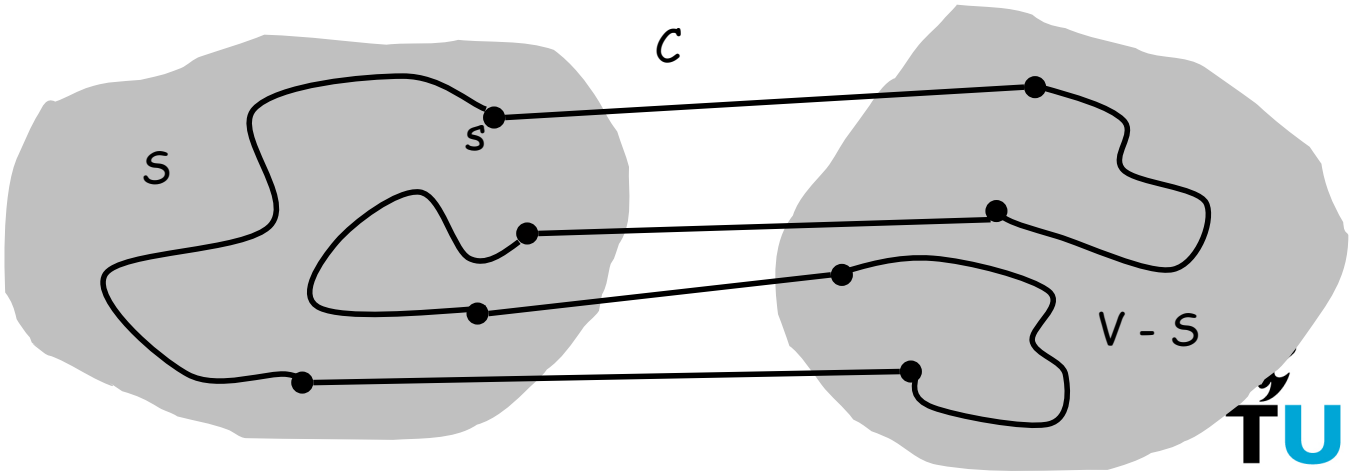
Cycle-Cut Intersection

Claim. A cycle and a cutset intersect in an *even* number of edges.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersection = 3-4, 5-6

Pf. Walk along cycle from a node s∈S: for every edge leaving S, there should (first) be an edge to a node in S before returning to s.



Cut property

Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST T* contains e.

Pf.

Q. What proof technique to use?

Cut property

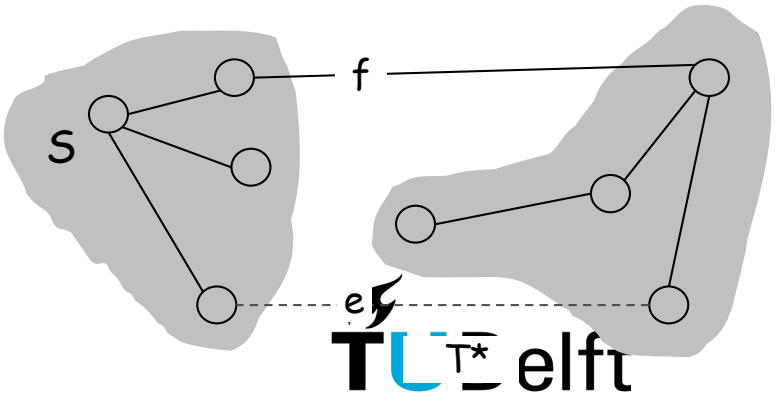
Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST T* contains e.

Pf. (by contradiction)

- Suppose e does not belong to T*, and let's see what happens.

- This is a contradiction.
- Thus e belongs to T* ▪



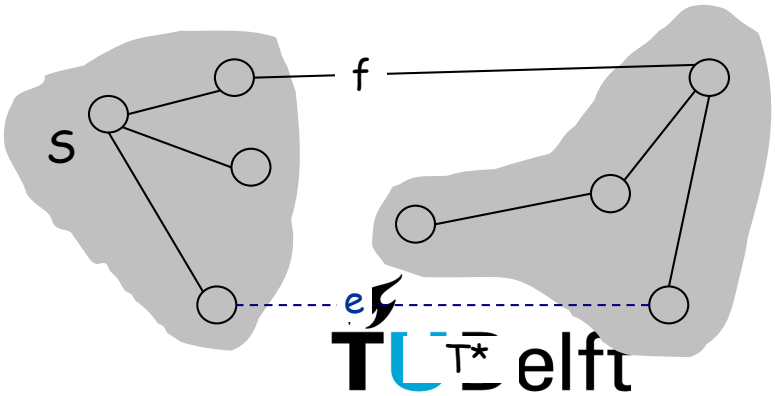
Cut property

Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST T* contains e.

Pf. (by contradiction)

- Suppose e does not belong to MST T*, and let's see what happens.
- Adding e to T* creates a cycle C in T*.
- Edge e is both in the cycle C and in the cutset D corresponding to S $\Rightarrow$ there exists another edge, say f, that is in both C and D. (use claim)
- $T' = T^* \cup \{ e \} - \{ f \}$ is also a spanning tree.
- Since $c_e < c_f$, cost(T') < cost(T*).
- This is a contradiction with T* being MST.
- Thus e belongs to T* ▪



This proof can be found on page 145.

Cycle property

Simplifying assumption. All edge costs c_e are distinct.

Cycle property. Let C be any cycle in G, and let f be the max cost edge belonging to C. Then the MST T* does not contain f.

Pf. (3 min)

Q. What proof technique to use?

Cycle property

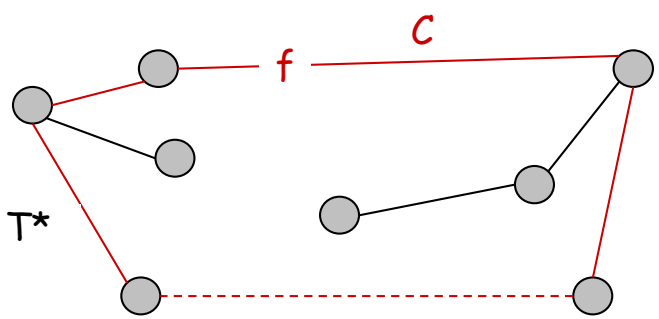
Simplifying assumption. All edge costs c_e are distinct.

Cycle property. Let C be any cycle in G, and let f be the max cost edge belonging to C. Then the MST T* does not contain f.

Pf. (by contradiction) (1 min)

- Suppose f belongs to T*, and let's see what happens.

- This is a contradiction.
- Thus f does not belong to T* ▪



Cycle property

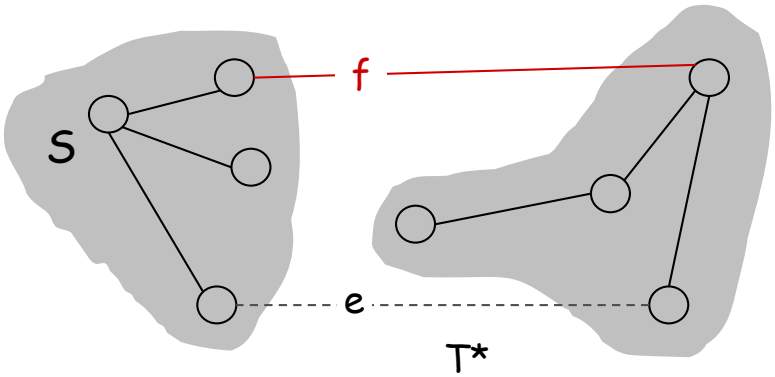
Simplifying assumption. All edge costs c_e are distinct.

Cycle property. Let C be any cycle in G, and let f be the max cost edge belonging to C. Then the MST T* does not contain f.

Pf. (by contradiction)

- Suppose f belongs to T*, and let's see what happens.
- Deleting f from T* creates a cut S in T*.

- This is a contradiction.
- Thus f does not belong to T* ▪



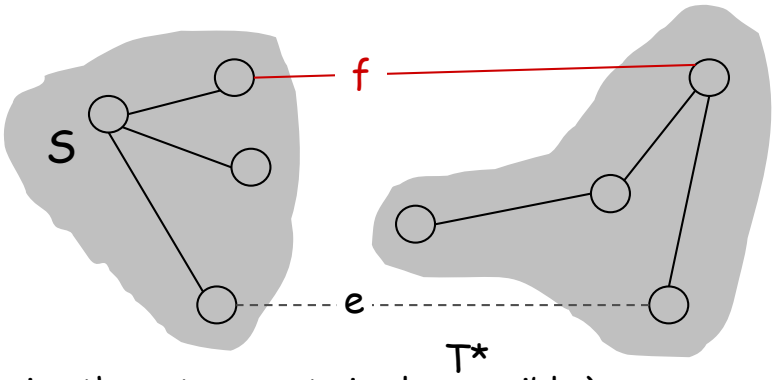
Cycle property

Simplifying assumption. All edge costs c_e are distinct.

Cycle property. Let C be any cycle in G, and let f be the max cost edge belonging to C. Then the MST T* does not contain f.

Pf. (by contradiction)

- Suppose f belongs to T*, and let's see what happens.
- Deleting f from T* creates a cut S in T*.
- Let S be all vertices connected in T* to one endpoint of f.
- Edge f is both in the cycle C and in the cutset D corresponding to S $\Rightarrow$ there exists another edge, say e, that is in both C and D. (use claim)
- $T' = T^* \cup \{ e \} - \{ f \}$ is also a spanning tree.
- Since $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$.
- This is a contradiction.
- Thus f does not belong to T* ▪



This proof can be found on page 147-148. (NB. An alternative proof using the cut property is also possible.)

Generic MST Algorithm (blue rule, red rule)

Blue rule: Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST T* contains e. Color e blue.

Red rule: Cycle property. Let C be any cycle in G, and let f be the max cost edge belonging to C. Then the MST T* does not contain f. Color f red.

Generic greedy algorithm.
Apply these rules until all edges are colored.

Prim's Algorithm: Proof of Correctness

Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]

- Initialize S = {any node}. Apply cut property to S.
- Add min cost edge in cutset corresponding to S to MST, and respective explored node u to S.

Q. Implementation is similar to which algorithm you have already seen?

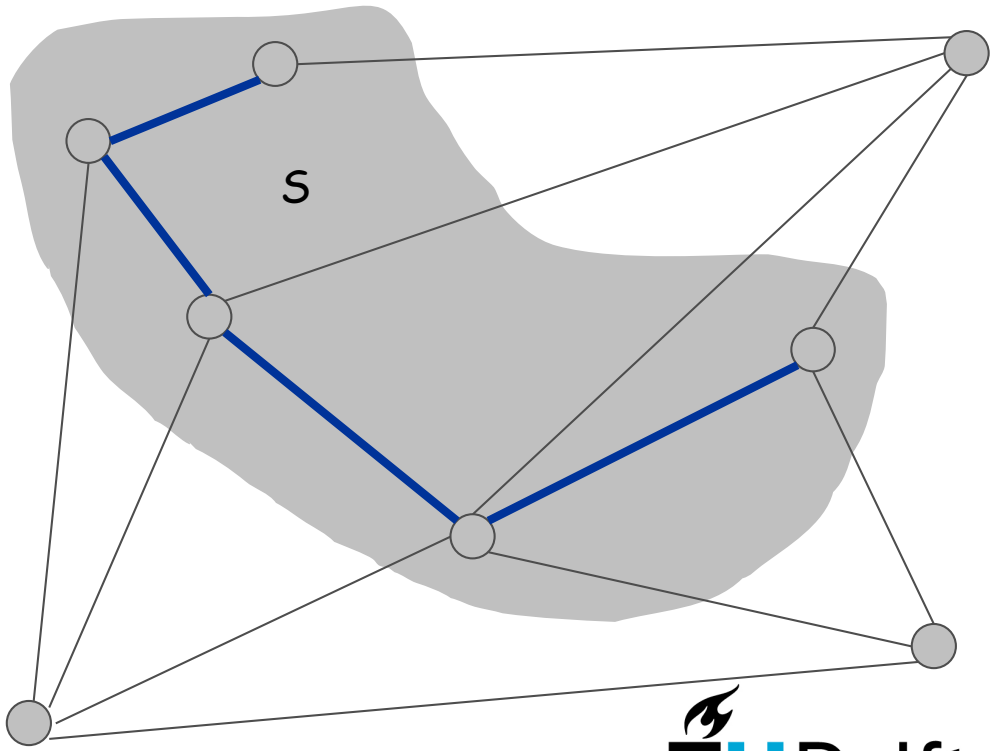
A. BFS

B. DFS

C. Dijkstra

D. Topological Sorting

E. I don't know.



Implementation: Prim's Algorithm

Implementation. Use a priority queue a la Dijkstra.

- Maintain set of explored nodes S.
- For each unexplored node v, maintain attachment cost a[v] = cost of cheapest edge e[v] to a node in S.
- $O(n^2)$ with an array; O(m log n) with a binary heap.

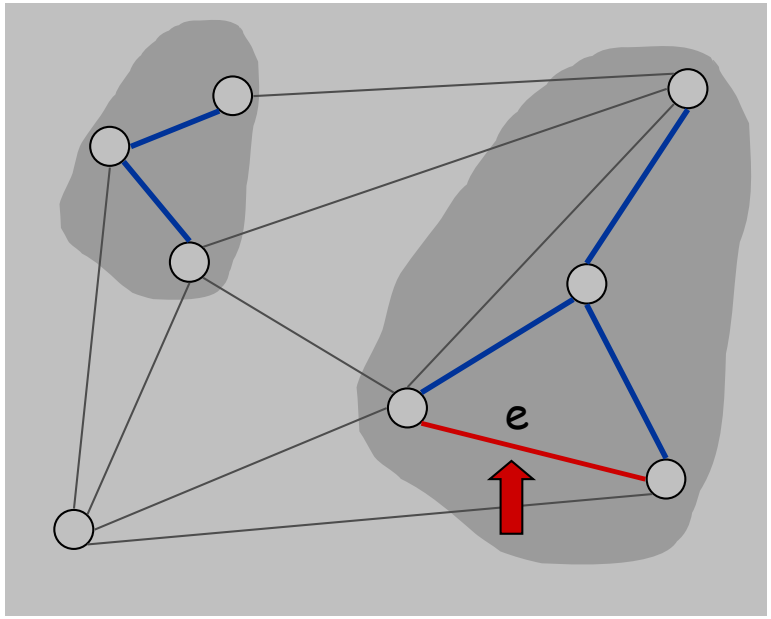
```
Prim(G, c) {
   foreach (v ∈ V) a[v] ← ∞; e[v] ← ϕ
   foreach (v ∈ V) insert v into Q
   Initialize set of explored nodes S ← ϕ, T ← ϕ

   while (Q is not empty) {
      u ← delete min element from Q
      S ← S ∪ { u }
      T ← T ∪ { e[u] } (unless e[u] = ϕ)
      foreach (edge e = (u, v) incident to u)
          if ((v ∉ S) and (c_e < a[v]))
             decrease priority a[v] to c_e
             e[u] ← e
}
```

ft

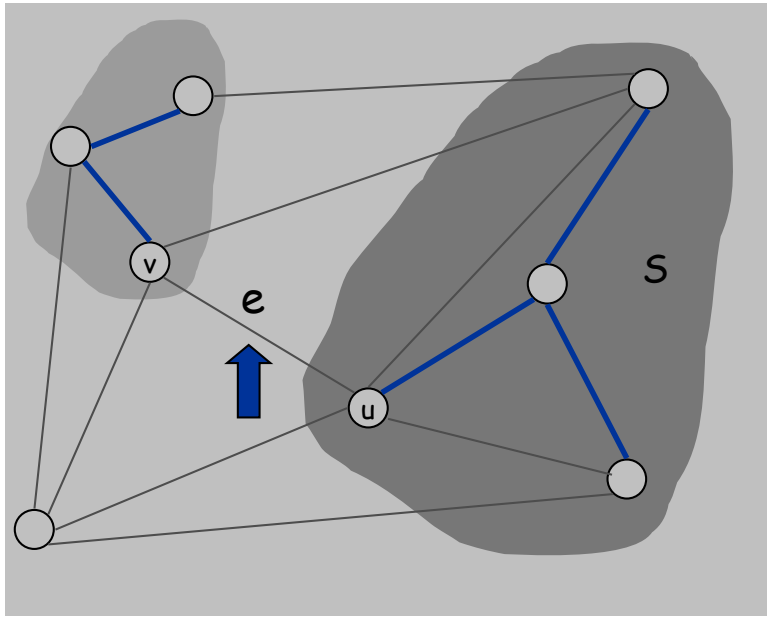
Kruskal's Algorithm: Proof of Correctness

Kruskal's algorithm. [Kruskal, 1956]

- Consider edges in ascending order of weight.
- Case 1: If adding e to T creates a cycle, discard e according to cycle property.
- Case 2: Otherwise, insert e = (u, v) into T according to cut property where S = set of nodes in u's connected component in T.



Case 1



Case 2

Implementation: Kruskal's Algorithm

Implementation. Use the union-find data structure.

- Build set T of edges in the MST.
- Maintain set for each connected component.

```
Kruskal(G, c) {
   Sort edges weights so that c₁ ≤ c₂ ≤ ... ≤ cₘ.
   T ← φ

   foreach (u ∈ V) make a set containing singleton u

   for i ← 1 to m
      (u,v) ← eᵢ
      if (u and v are in different sets) {
         T ← T ∪ {eᵢ}
         merge the sets containing u and v
      }
   return T
}
```

are u and v in different connected components, ie, find(u)=find(v)?

merge two components, ie, union(u,v)

Union-Find

Union-Find.

Efficient data structure to do two operations on

- Union: merge two components
- Find: give the representative of the component

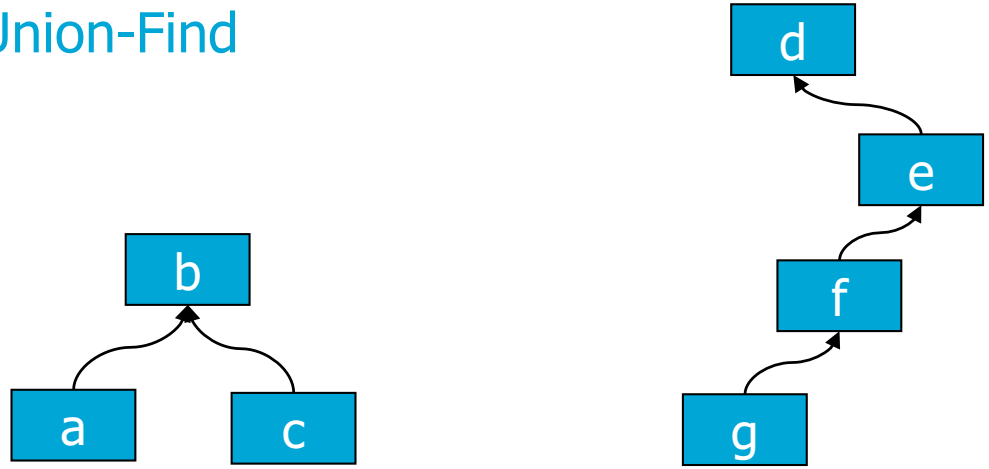
Q. How to implement efficiently?

TU Delft

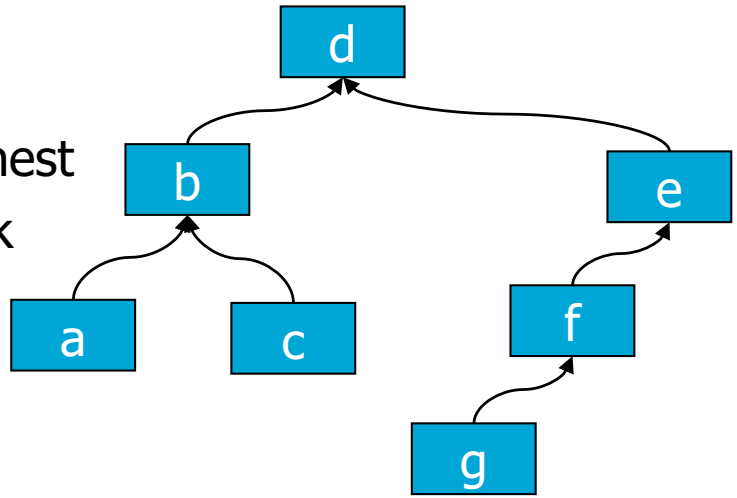
Union-Find

Union-Find.

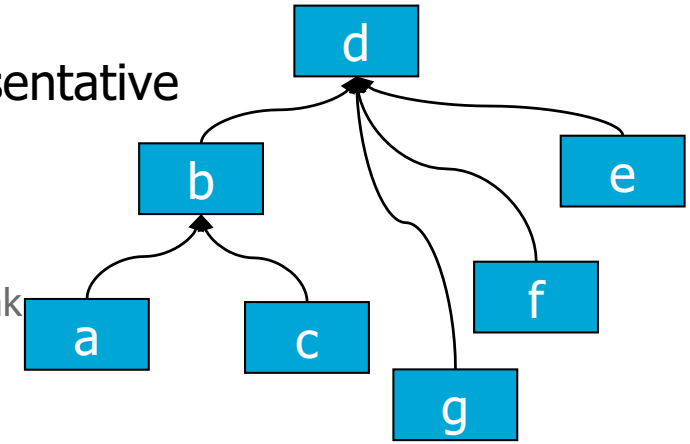
- Represent component by tree



- Union (by rank): merge two components
  - assign each node a rank
  - place root with lowest rank under highest
  - increase rank of new root if equal rank



- Find: give the representative
  - path compression
    (eg find(g))
  - btw, do not update rank



TU Delft

Implementation: Kruskal's Algorithm

Implementation. Using the union-find data structure.

- O(m log n) for sorting and O(m $\underbrace{\alpha(m, n)}_{\text{essentially a constant}}$) for union-find.

$m \le n^2 \Rightarrow \log m$ is $O(\log n)$

```
Kruskal(G, c) {
   Sort edges weights so that c₁ ≤ c₂ ≤ ... ≤ cₘ.
   T ← φ

   foreach (u ∈ V) make a set containing singleton u

   for i ← 1 to m
      (u,v) ← eᵢ
      u_root ← find(u)
      v_root ← find(v)
      if (u_root != v_root) {
         T ← T ∪ {eᵢ}
         union( u_root, v_root )
      }
   return T
}
```

$O(m)$

$O(\alpha(m, n))$

$O(1)$

5. Divide & conquer

Divide & conquer

History

"*divide ut regnes*" / "*divide et impera*"
(divide and rule)

Often used in politics and as military strategy
(Julius Caesar, Machiavelli, Napoleon)

General idea

1. Break a problem into subproblems,
2. solve each subproblem
   a. usually recursively,
   b. if small enough, solve as base case,
3. then combine results.

Correctness proof by induction.

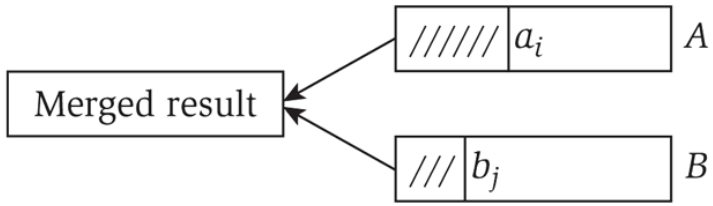
Examples

- recursive algorithms on trees
- merge sort, quicksort

TU Delft

Linear Time: O(n)

Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.

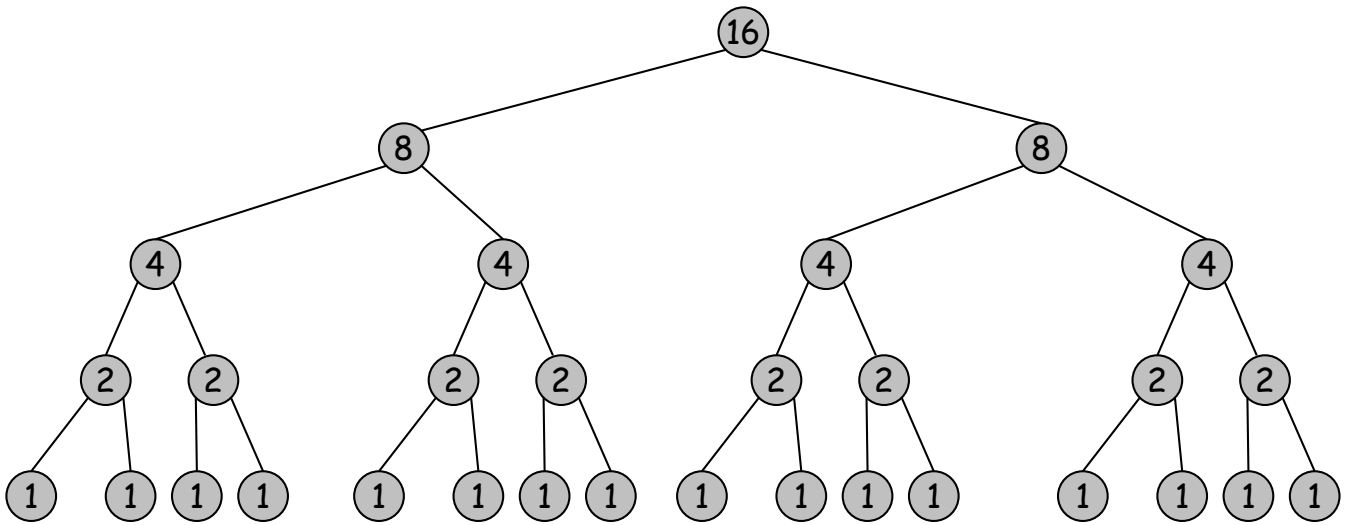



```
i = 1, j = 1
while (both lists are nonempty) {
   if (a_i ≤ b_j) append a_i to output list and increment i
   else           append b_j to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size n takes O(n) time.

Pf. After each comparison, the length of output list increases by 1.

Call graph of Mergesort



Call graph of Mergesort of a string of length 16
(the nodes contain size of substring; nodes typically represent different substrings)

6. Dynamic programming

Today

Dynamic Programming (Ch.6.1-6.4)

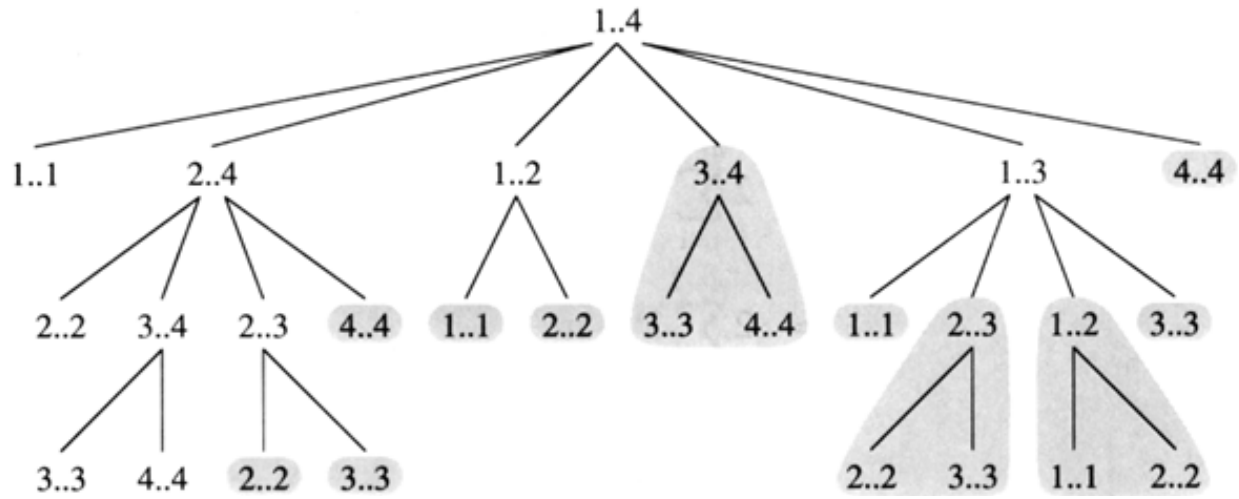
- Binary choice: weighted interval scheduling
- Multi-way choice: word segmentation
- Extra variable: knapsack

Algorithmic Paradigms

Greedy. Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer. Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.



recursive matrix chain
optimal multiplication order
(Cormen et al., p.345)

Dynamic Programming Applications

Areas.

- Bioinformatics.
- Control theory. (E.g. heating)
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, systems, ….

Some famous dynamic programming algorithms.

- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- Smith-Waterman for sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

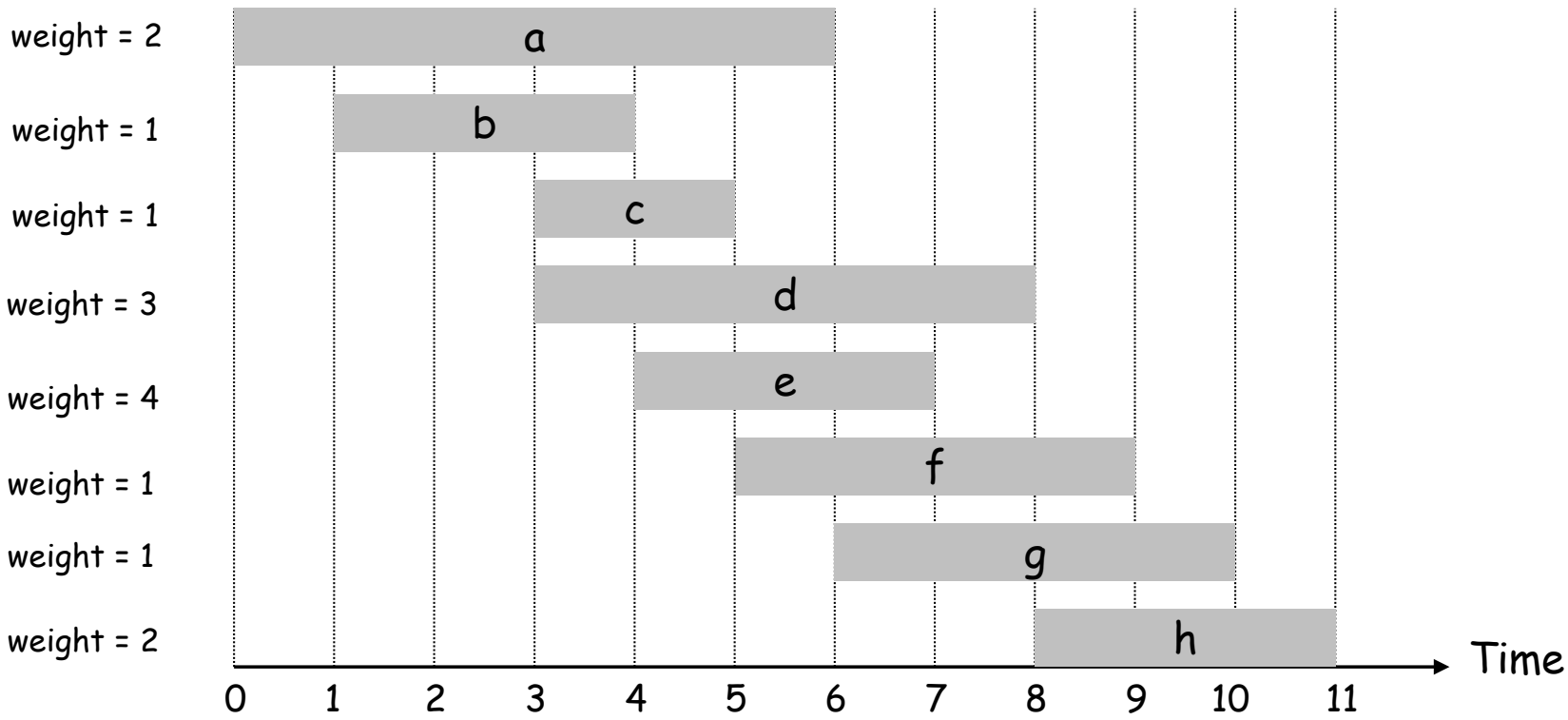
6.1 Dynamic Programming: Binary Choice

Weighted Interval Scheduling

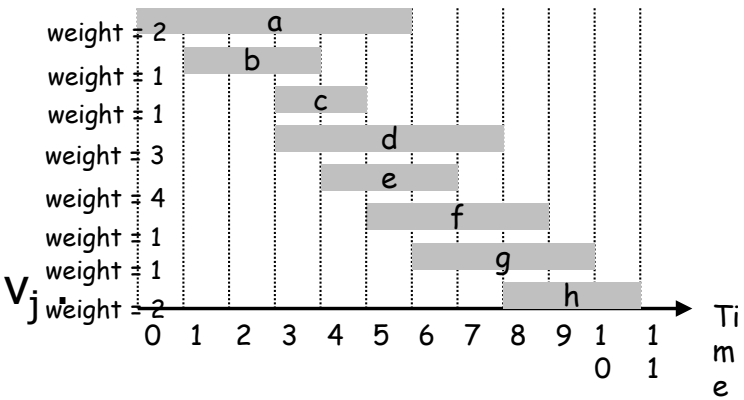
Weighted interval scheduling problem.

- Job j starts at s_j, finishes at f_j, and has weight v_j .
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.

Q. How to efficiently solve this problem?



Weighted Interval Scheduling



Weighted interval scheduling problem.

- Job j starts at s_j, finishes at f_j, and has weight v_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.

Q. How to efficiently solve this problem with weights?

A. Consider all possible subsets of jobs that are all compatible, and take the maximum set.

B. Consider jobs in ascending order of finish time and add if compatible with already selected jobs.

C. Consider jobs in descending order of finish time and add if compatible with already selected jobs.

D. Consider jobs in ascending order of start time and add if compatible with already selected jobs.

E. Consider jobs in descending order of start time and add if compatible with already selected jobs.

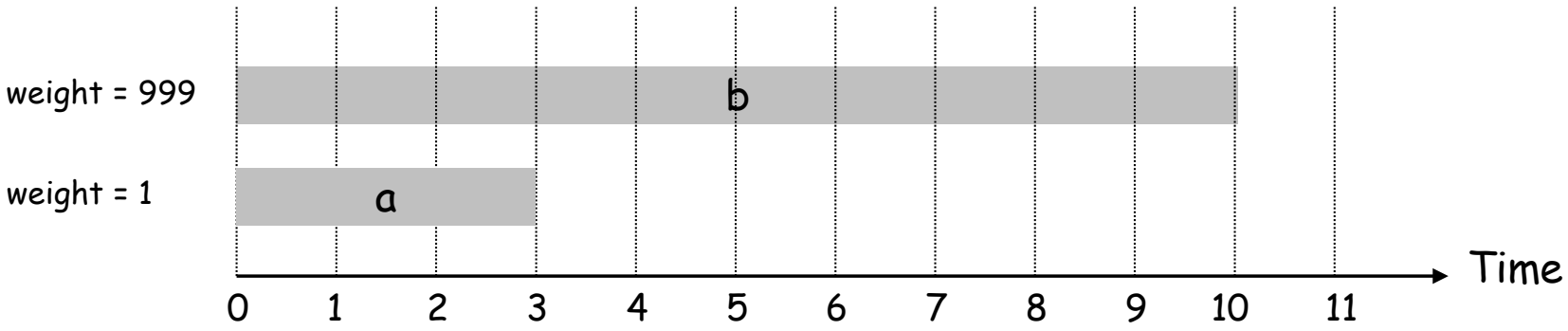
F. I don't know.

Weighted Interval Scheduling: Greedy

Recall. Greedy algorithm works if all weights are 1.

- Consider jobs in ascending order of finish time.
- Add job to subset if it is *compatible* with previously chosen jobs.

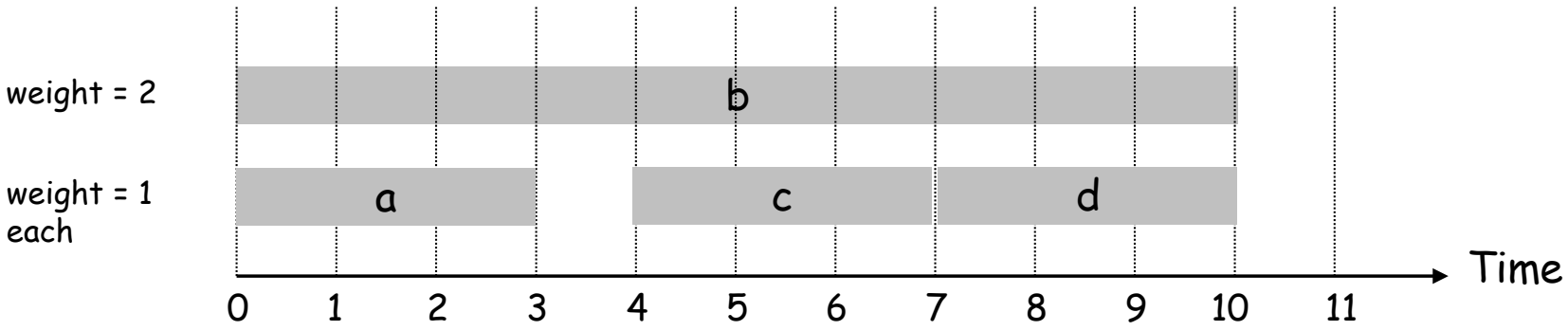
C. Ascending order of finish time used for weighted interval scheduling



Weighted Interval Scheduling: Greedy

B. Descending order of weight

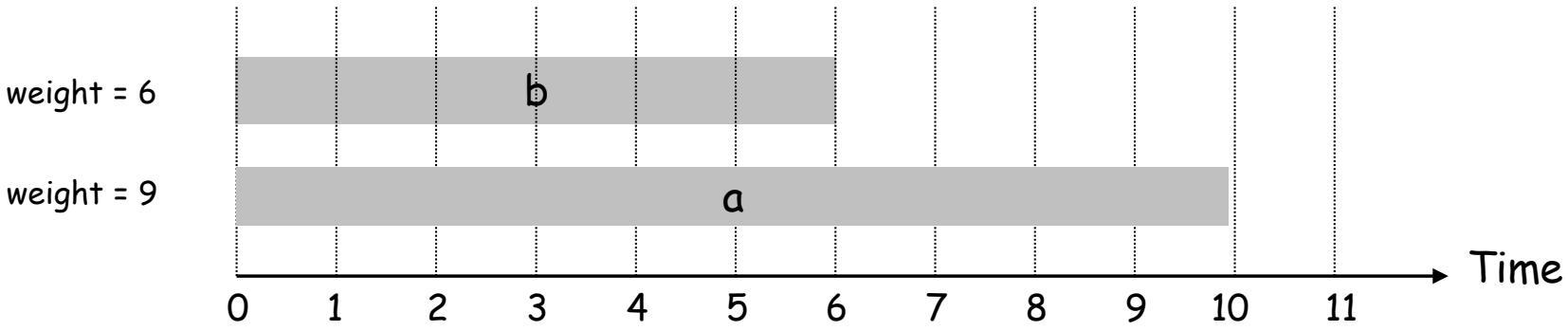
Fails



Weighted Interval Scheduling: Greedy

D. Descending order of relative weight (weight per time unit)

Fails (by a factor 2 at most).



Weighted Interval Scheduling: Brute Force

A. All possible subsets of jobs

Q. How many possible selections of jobs are there at most?

A. O(n log n)

B. $O(n^2)$

C. $O(n^3)$

D. $O(2^n)$

E. $O(n^!)$

F. I don't know.

Weighted Interval Scheduling: Brute Force

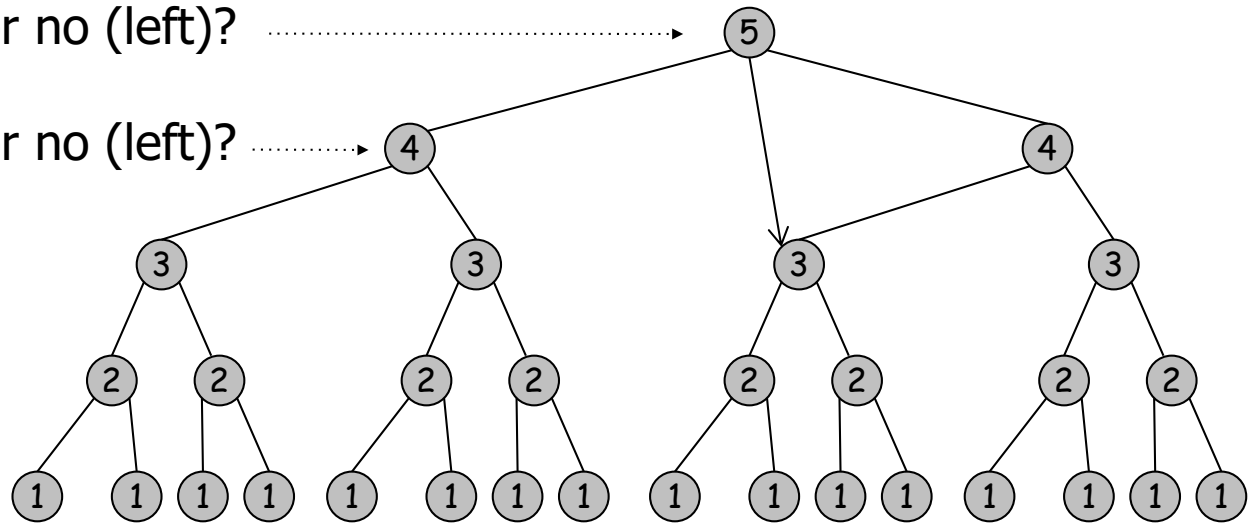
include 5? yes (right) or no (left)?

include 4? yes (right) or no (left)?

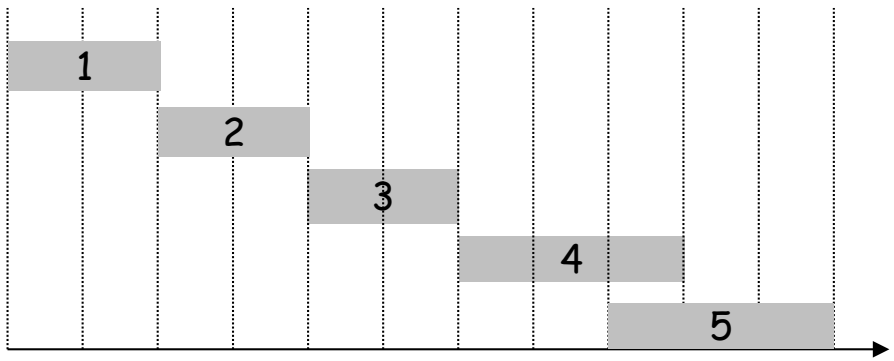
include 3?

include 2?

include 1?



Note: recursion! (Is common with back-tracking).
1.Some combinations can be infeasible…
2.Some subproblems are identical

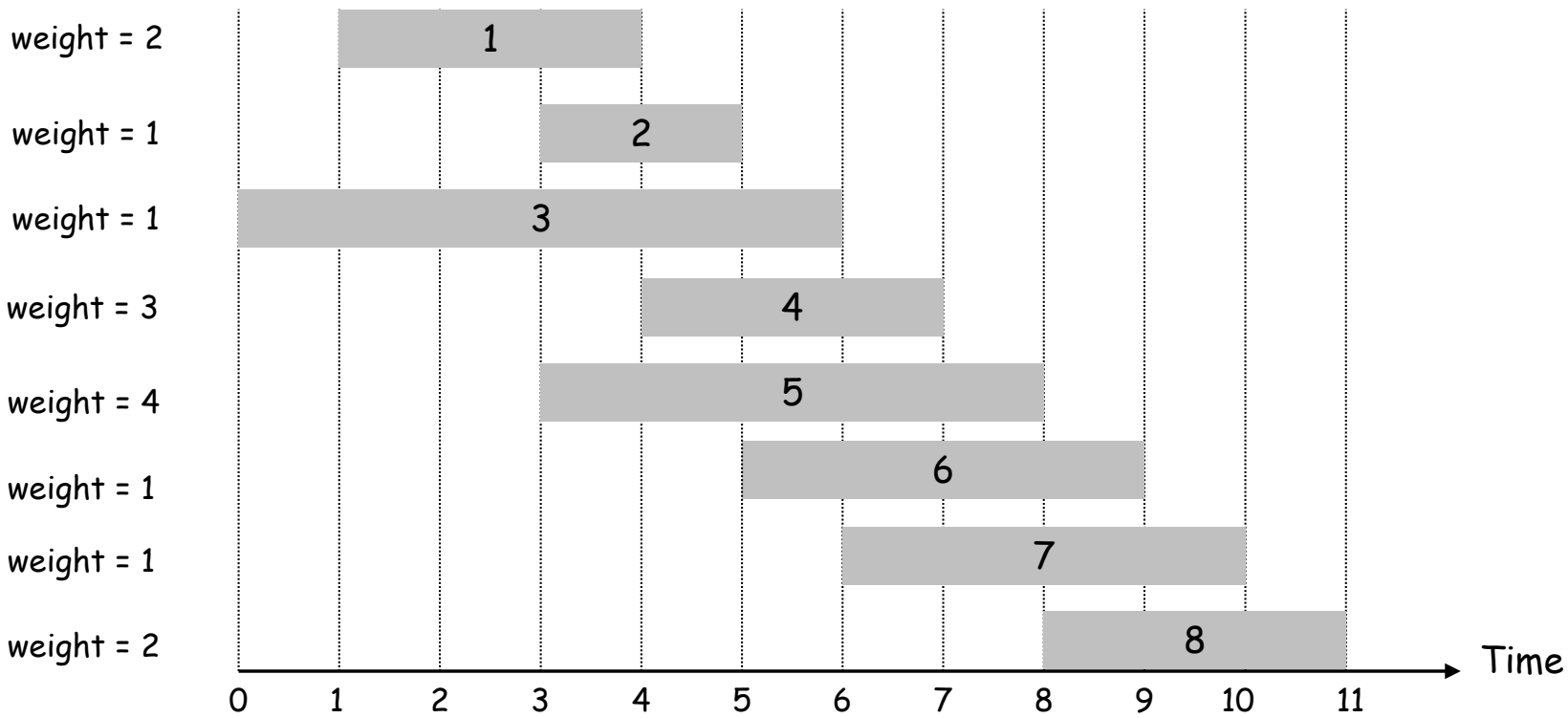


Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. p(j) = largest index i < j such that job i is compatible with j. (predecessor)

Q. p(8) = ?, p(7) = ?, p(2) = ?.



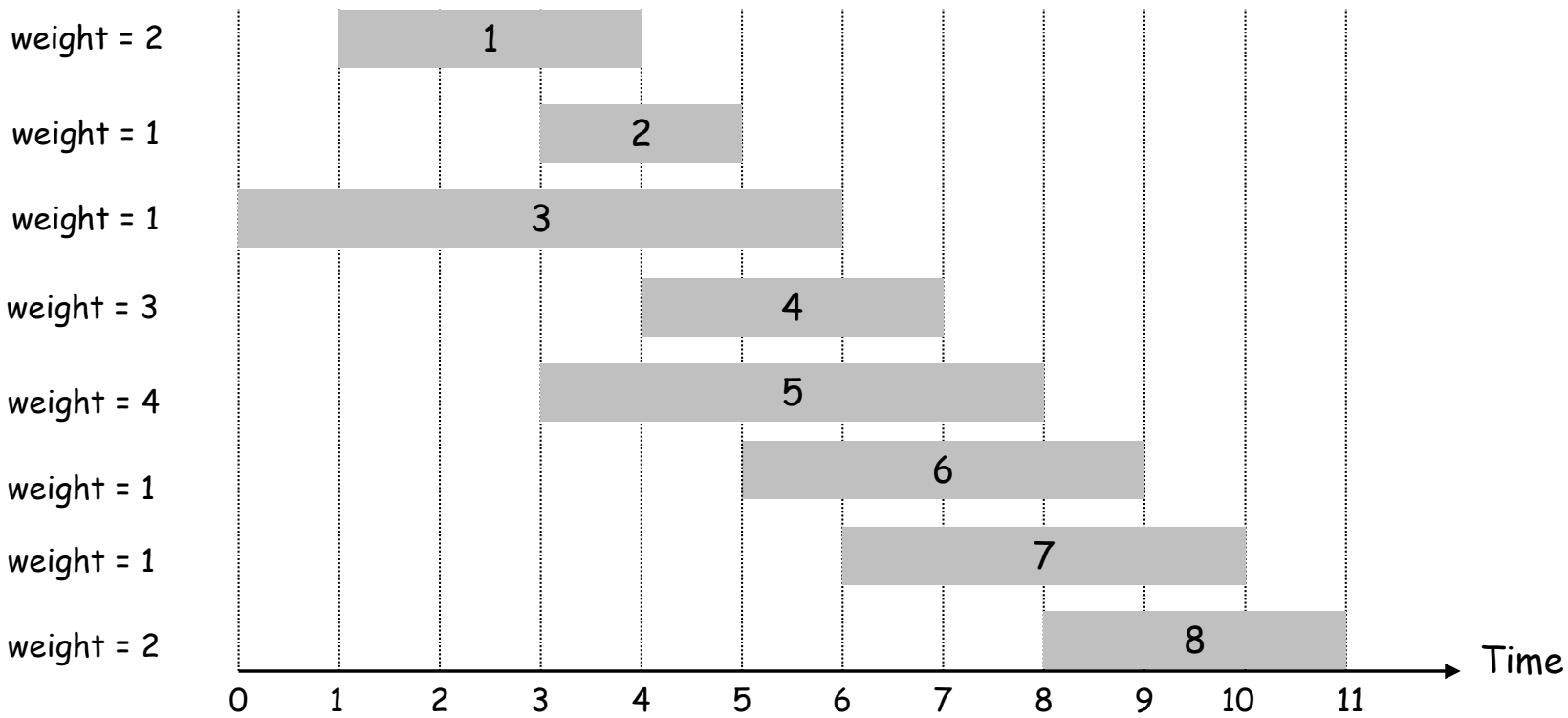
Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. p(j) = largest index i < j such that job i is compatible with j. (predecessor)

Q. p(8) = ?, p(7) = ?, p(2) = ?.

A. p(8) = 5, p(7) = 3, p(2) = 0.



Weighted Interval Scheduling: Recursion

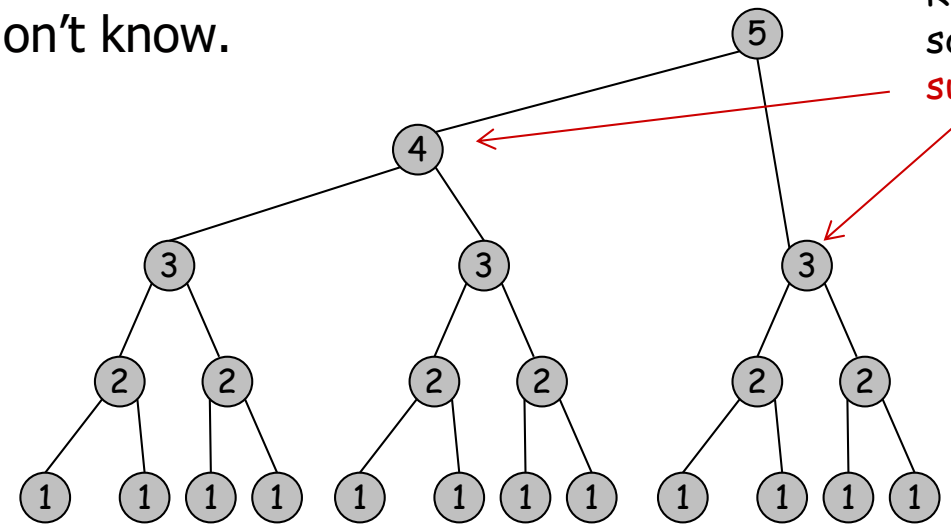
Q. Suppose optimal weight of selection up to three (OPT(3)) and four (OPT(4)) are known, what to do with job 5 with p(5)=3?

A. Select if OPT(3) ≥ OPT(4)

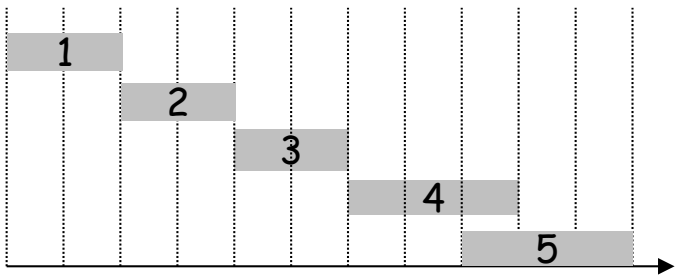
B. Select if OPT(3) + v_5 ≥ OPT(4)

C. Select if OPT(4) + v_5 ≥ OPT(3)

D. I don't know.



Recursion:
so assume optimal value of subproblems is known.



Example: Weighted Interval Scheduling: Brute Force'

Notation. OPT(j) = *value* of optimal solution to the problem consisting of job requests 1, 2, ..., j (ordered by finishing time).

- Case 1: OPT selects job 5.
  - can't use incompatible job 4
  - must include optimal solution to problem consisting of remaining compatible jobs, so OPT(3)

optimal substructure

- Case 2: OPT does not select job 5.
  - must include optimal solution to problem consisting of remaining compatible jobs, so OPT(4)

$$OPT(5) = \max \left\{ v_5 + OPT(3),\ OPT(4) \right\}$$

General: Weighted Interval Scheduling: Brute Force'

Notation. OPT(j) = *value* of optimal solution to the problem consisting of job requests 1, 2, ..., j (ordered by finishing time).

- Case 1: OPT selects job j.
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

optimal substructure

- Case 2: OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

General: Weighted Interval Scheduling: Brute Force'

Notation. OPT(j) = *value* of optimal solution to the problem consisting of job requests 1, 2, ..., j (ordered by finishing time).

- Case 1: OPT selects job j.
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

optimal substructure

- Case 2: OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Case 1 Case 2

Weighted Interval Scheduling: Brute Force'

Brute force algorithm (with smart skipping of predecessors).

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(vj + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling: Brute Force'

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(vj + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

Q. What is the worst-case tight upper bound of this algorithm?

A. O(n log n)

B. $O(n^2)$

C. $O(n^3)$

D. $O(2^n)$

E. O(n!)

F. I don't know.

Weighted Interval Scheduling: Brute Force'

Q. What is the worst-case tight upper bound of this algorithm?

A. O(n log n)

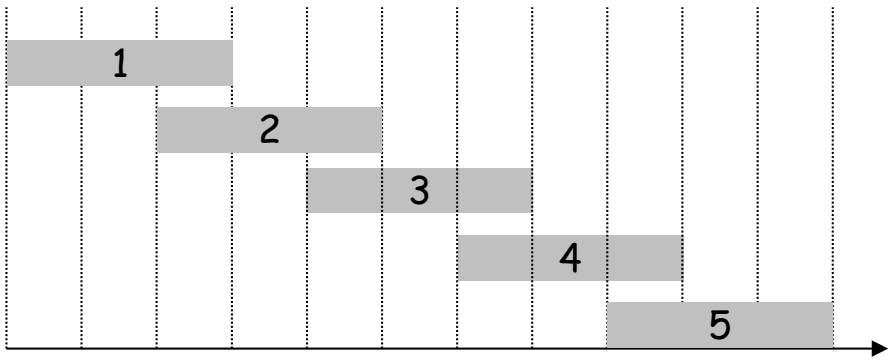
B. $O(n^2)$

C. $O(n^3)$

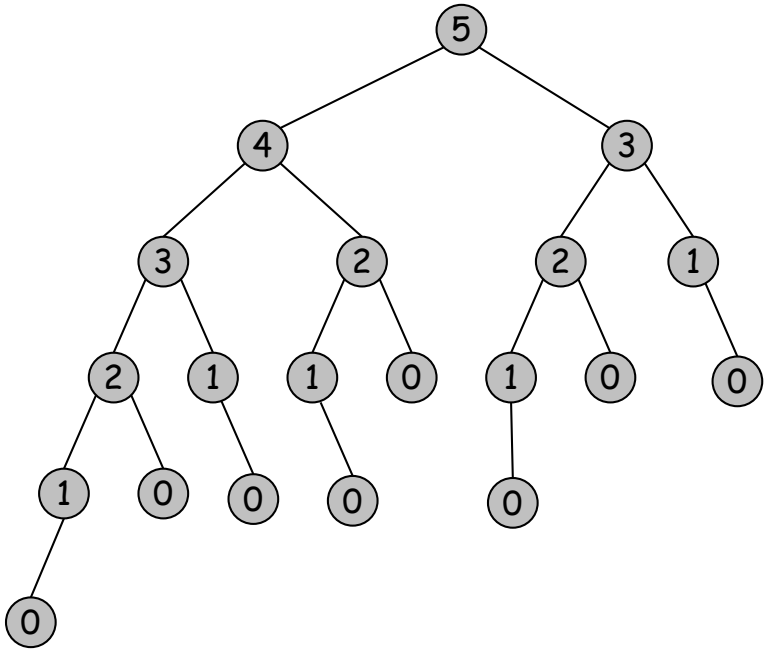
D. $O(2^n)$

E. O(n!)

F. I don't know.



p(1) = 0, p(j) = j-2




```
return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
```

Weighted Interval Scheduling: Brute Force'

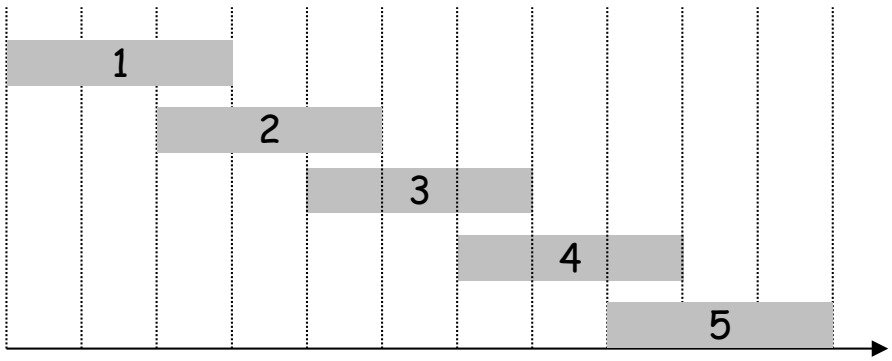
Q. What is the worst-case tight upper bound of this algorithm?

A. $T(0)=O(1)$ and $T(n) = T(n-1) + T(n-2) + O(1)$

NB: worst-case is $T(n-2)$, because if $p(j)=j-1$ there is only one subproblem

Observation. Number of recursive calls grow like Fibonacci sequence $\Rightarrow$ exponential.

Observation. Recursive algorithm has many (redundant) sub-problems.



$p(1) = 0$, $p(j) = j-2$

```
return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
```

Weighted Interval Scheduling: Brute Force'

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(vj + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

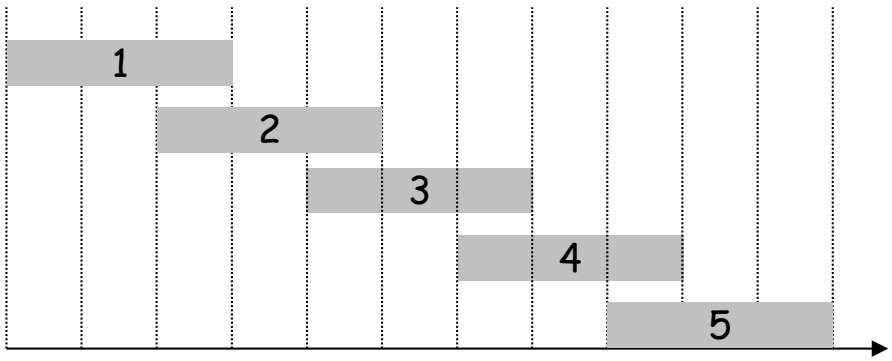
Q. What can we do to obtain polynomial run time?

Weighted Interval Scheduling: Memoization

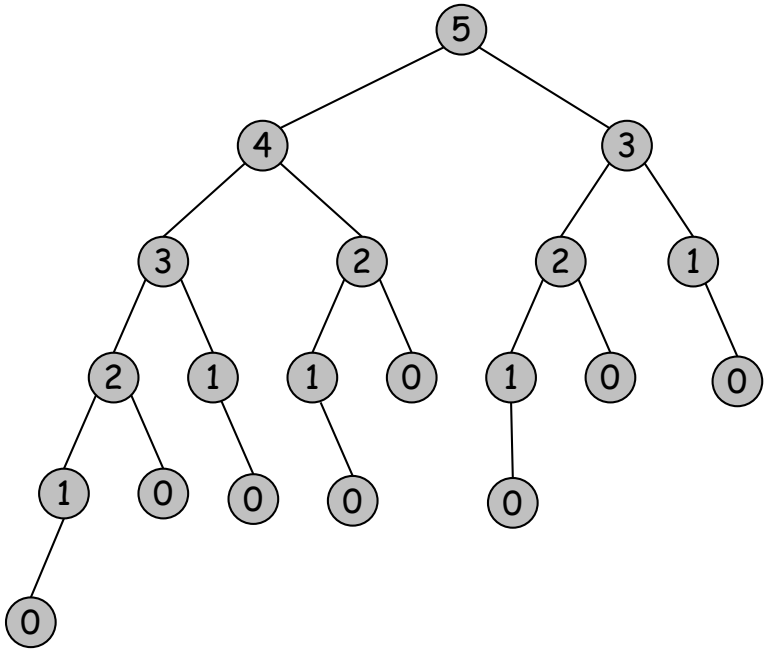
Memoization. Store results of each sub-problem in a cache; lookup as needed.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty   ← global array
M[0] = 0

M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

Weighted Interval Scheduling: Memoization



p(1) = 0, p(j) = j-2



Weighted Interval Scheduling: Memoization

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty
M[0] = 0

M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

Q. What is the worst-case tight upper bound of this algorithm?

A. O(n log n)

B. $O(n^2)$

C. $O(n^3)$

D. $O(2^n)$

E. O(n!)

F. I don't know.

Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes O(n log n) time.

Proof.

Q. How many iterations in initialization?

Q. How many iterations in one invocation?

Q. How many invocations?

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty
M[0] = 0

M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(vj + M-Compute-Opt(p(j)), M-
Compute-Opt(j-1))
   return M[j]
}
```

Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes O(n log n) time.

Proof.

Q. How many iterations in initialization?

- Sort by finish time: O(n log n).
- Computing p(·) : O(n log n) (e.g. O(n) if by decreasing start time)

Q. How many iterations in one invocation?

Q. How many invocations?

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty
M[0] = 0

M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(vj + M-Compute-Opt(p(j)), M-
Compute-Opt(j-1))
   return M[j]
}
```

Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes O(n log n) time.

Proof.

Q. How many iterations in initialization?

- Sort by finish time: O(n log n).
- Computing p(·) : O(n log n) (e.g. O(n) if by decreasing start time)

Q. How many iterations in one invocation?

- `M-Compute-Opt(j)`: each invocation takes O(1) time and either
  - (i) returns an existing value `M[j]`
  - (ii) fills in one new entry `M[j]` and makes two recursive calls

Q. How many invocations?

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty
M[0] = 0

M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(vj + M-Compute-Opt(p(j)), M-
Compute-Opt(j-1))
   return M[j]
}
```

Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes O(n log n) time.

Proof.

Q. How many iterations in initialization?

- Sort by finish time: O(n log n).
- Computing p(·) : O(n log n) (e.g. O(n) if by decreasing start time)

Q. How many iterations in one invocation?

- `M-Compute-Opt(j)`: each invocation takes O(1) time and either
  - (i) returns an existing value `M[j]`
  - (ii) fills in one new entry `M[j]` and makes two recursive calls

Q. How many invocations?

- Progress measure Φ = # nonempty entries of `M[]`.
  - initially $\Phi = 0$, throughout $\Phi \leq n$.
  - (ii) increases Φ by 1 and only then at most 2 recursive calls.
- Overall running time (without init) of `M-Compute-Opt(n)` is O(n). ▪

Remark. O(n) if jobs are pre-sorted by start and finish times.

Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(vj + M[p(j)], M[j-1])
}
```

Note: reason top-down, implement bottom-up.

Weighted Interval Scheduling: Bottom-Up

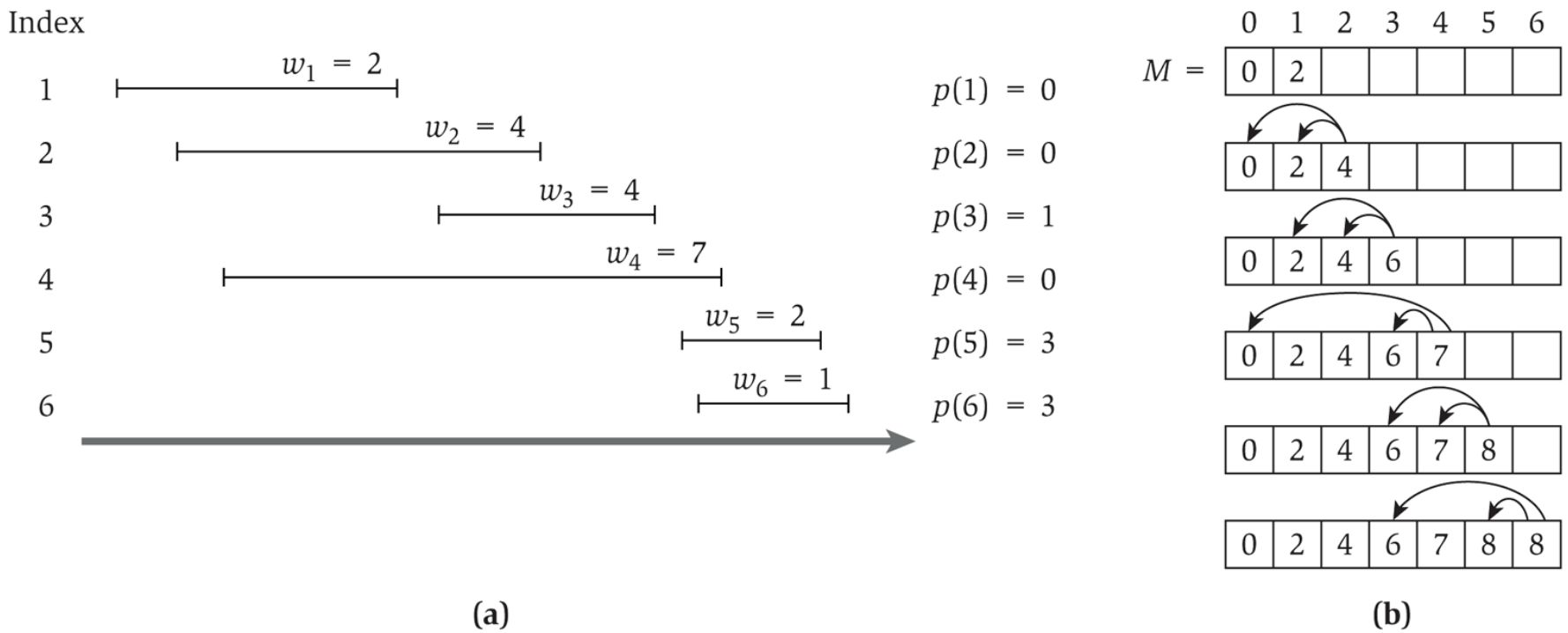


Figure 6.5 Part (b) shows the iterations of `Iterative-Compute-Opt` on the sample instance of Weighted Interval Scheduling depicted in part (a).

Weighted Interval Scheduling: Finding a Solution

Dynamic programming algorithms computes optimal value.
The solution can be found by post-processing the cache.

```
Run M-Compute-Opt(n)
Run Find-Solution()

Find-Solution(n) {
   j = n
   while (j>0 and v_j + M[p(j)] >= M[j-1])
      print j
      j = p(j)
```

Q. In what order are the jobs printed?

Weighted Interval Scheduling: Finding a Solution

Dynamic programming algorithms computes optimal value.
The solution can be found by post-processing the cache recursively.

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
   if (j = 0)
      output nothing
   else if (v_j + M[p(j)] > M[j-1])
      Find-Solution(p(j))
      print j
   else
      Find-Solution(j-1)
}
```

- # of recursive calls $\leq$ n $\Rightarrow$ O(n).

Dynamic Programming Summary

Recipe.

1. Characterize structure of problem.
2. Recursively define value of optimal solution: OPT(j) = …
3. Compute value of optimal solution iteratively.
4. Construct optimal solution from computed information.

Dynamic programming techniques.

- Binary choice: weighted interval scheduling.
- Multi-way choice: segmented least squares, word segmentation.

6.2 Dynamic Programming: Multi-Way Choice

Word segmentation

Problem

- Given a string x of letters $x_1x_2...x_n$,
- Given a quality function $q(i,j)$ that gives the value of substring $x_ix_{i+1}...x_j$.
- Give an efficient algorithm to split x into words (substrings) such that sum of quality of these words is maximized.

Example. "mogenzeslapen":

q(mo) + q(gen) + q(ze) + q(sla) + q(pen) = ?
q(mogen) + q(ze) + q(slapen) = ?

word	quality
mogen	4
enz	1
gen	2
sla	2
pen	2
slapen	5
ze	1
en	1

TU Delft

Word segmentation

Problem

- Given a string x of letters $x_1x_2...x_n$,
- Given a quality function $q(i,j)$ that gives the value of substring $x_ix_{i+1}...x_j$.
- Give an efficient algorithm to split x into words (substrings) such that sum of quality of these words is maximized.

Example. "mogenzeslapen":

q(mo) + q(gen) + q(ze) + q(sla) + q(pen) = 7
q(mogen) + q(ze) + q(slapen) = 10

word	quality
mogen	4
enz	1
gen	2
sla	2
pen	2
slapen	5
ze	1
en	1

TU Delft

Dynamic Programming: Multiway Choice

Notation.

- OPT(j) = maximum quality of string $x_1, x_2, \ldots, x_j$.
- q(i, j) = quality of substring $x_i, x_{i+1}, \ldots, x_j$.

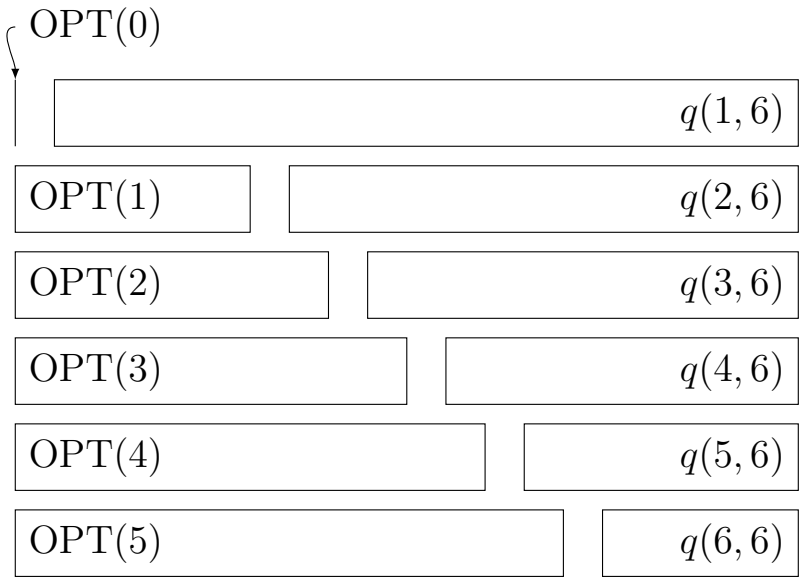
Reason backward, computing OPT(j) using subproblems

Q. How can value of OPT(j) be expressed based on subproblems?

Dynamic Programming: Multiway Choice

Example. Compute OPT(6).

Choose optimal value of the following segmentations:



Dynamic Programming: Multiway Choice

Notation.

- OPT(j) = maximum quality of string $x_1, x_2, \ldots, x_j$.
- q(i, j) = quality of substring $x_i, x_{i+1}, \ldots, x_j$.

Reason backward, computing OPT(j) using subproblems

Q. How can value of OPT(j) be expressed based on subproblems?

Q. What are the options here?

A. The start i of the last word.

- Last word uses characters $x_i, x_{i+1}, \ldots, x_j$ for some i.
- Value = q(i, j) + OPT(i-1).

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\limits_{1 \le i \le j} \left\{ \underbrace{q(i,j) + OPT(i-1)}_{\text{Value of this choice}} \right\} & \text{otherwise} \end{cases}$$

Choose i ∈ [1, j]

Word Segmentation: DP Algorithm

```
INPUT: n, q_ij

Word-Segmentation() {
   M[0] = 0

   for j = 1 to n
      M[j] = max_{1 ≤ i ≤ j} (q_ij + M[i-1])

   return M[n]
}
```

Q. What is the worst-case tight upper bound of this algorithm?

A. O(n)

B. O(n log n)

C. $O(n^2)$

D. $O(n^3)$

E. $O(2^n)$

F. O(n!)

G. I don't know.

Word Segmentation: Finding a Solution

```
Run Find-Solution(n)
Find-Solution(j) {
   if (j = 0)
      output nothing
   else
      i = j
      while(i >= 1 && q_ij+M[i-1]!=M[j])
        i = i-1
      Find-Solution(i-1)
      output i
}
```

Dynamic Programming Summary

Recipe.

1. Characterize structure of problem.
2. Recursively define value of optimal solution: OPT(j) = …
3. Compute value of optimal solution iteratively.
4. Construct optimal solution from computed information.

Dynamic programming techniques.

- Binary choice: weighted interval scheduling.
- Multi-way choice: segmented least squares, word segmentation.
- Extra variable: knapsack

6.4 Knapsack Problem

Knapsack Problem

Knapsack problem.

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has limit of W kilograms.
- Goal: fill knapsack so as to maximize total value.

Q. What is the maximum value here?

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Problem

Knapsack problem.

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has limit of W kilograms.
- Goal: fill knapsack so as to maximize total value.

Q. What is the maximum value here?

A. { 3, 4 } attains 40

W = 11

Item	Value	Weight	Ratio
1	1	1	1
2	6	2	3
3	18	5	3.6
4	22	6	3.66
5	28	7	4

A reasonable greedy algorithm seems to repeatedly add item with maximum ratio v_i / w_i.

Q. Is this greedy algorithm optimal?

Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

Dynamic Programming: False Start

Recursively define value of optimal solution:

Def. OPT(i) = max profit subset of items 1, …, i.

- Case 1: OPT does not select item i.
  - OPT selects best set out of { 1, 2, …, i-1 }

- Case 2: OPT selects item i.
  - accepting item i does not immediately imply that we will have to reject other items; this depends on the remaining weight!
  - (does not only depend on best set out of { 1, 2, …, i-1 })

Conclusion. Need more sub-problems!

Q. What is the missing parameter to identify a sub-problem?

Q. And how to express the optimal value of a set of items and a capacity in terms of these sub-problems? (1 min)

TU Delft

Dynamic Programming: Adding a New Variable

Recursively define value of optimal solution:

Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

- Case 1: OPT does not select item i.
  - OPT selects best set out of { 1, 2, …, i-1 } using weight limit w
- Case 2: OPT selects item i.
  - new weight limit = $w - w_i$
  - OPT selects best set out of { 1, 2, …, i–1 } using this new weight limit

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ \ v_i + OPT(i-1,w-w_i)\,\} & \text{otherwise} \end{cases}$$

Knapsack Algorithm: Recursive

W + 1

OPT(i,w):

i: \ w:		0	1	2	3	4	5	6	7	8	9	10	11
0	ϕ												
1	{ 1 }												
2	{ 1, 2 }												
3	{ 1, 2, 3 }												
4	{ 1, 2, 3, 4 }												
5	{ 1, 2, 3, 4, 5 }												

n + 1

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\ \ v_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

Knapsack Algorithm: Recursive

W + 1 →

OPT(i,w):

n + 1 ↓

i: \ w:		0	1	2	3	4	5	6	7	8	9	10	11
0	ϕ	0	0	0	0	0	0	0		0	0	0	0
1	{ 1 }	0	1	1	1	1	1	1			1		1
2	{ 1, 2 }	0				7	7	7					7
3	{ 1, 2, 3 }					7	18						25
4	{ 1, 2, 3, 4 }					7							40
5	{ 1, 2, 3, 4, 5 }												40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\ \ v_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

Knapsack Algorithm: Bottom-Up

W + 1 →

OPT(i,w):

i: \ w:		0	1	2	3	4	5	6	7	8	9	10	11
0	ϕ												
1	{ 1 }												
2	{ 1, 2 }												
3	{ 1, 2, 3 }												
4	{ 1, 2, 3, 4 }												
5	{ 1, 2, 3, 4, 5 }												

n + 1

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\ \ v_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

Knapsack Algorithm: Bottom-Up

W + 1 →

OPT(i,w):

n + 1 ↓

i:	w:	0	1	2	3	4	5	6	7	8	9	10	11
0	ϕ	0	0	0	0	0	0	0	0	0	0	0	0
1	{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
2	{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
3	{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
4	{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
5	{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\ \ v_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up

Compute value of optimal solution iteratively.

Knapsack. Fill up an n-by-W array.

```
Input: n, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 0 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

Q. What is the running time? (1 min)

A.

Knapsack Problem: Bottom-Up

Compute value of optimal solution iteratively.

Knapsack. Fill up an n-by-W array.

```
Input: n, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 0 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

Q. What is the running time? (1 min)

A. $\Theta(n\ W)$.

Knapsack Problem: Running Time

Running time. $\Theta(n\ W)$.

- Not polynomial in input size!
- "Pseudo-polynomial."
- (Decision version of) Knapsack is NP-complete.
  (Complexity theory, 3rd year)

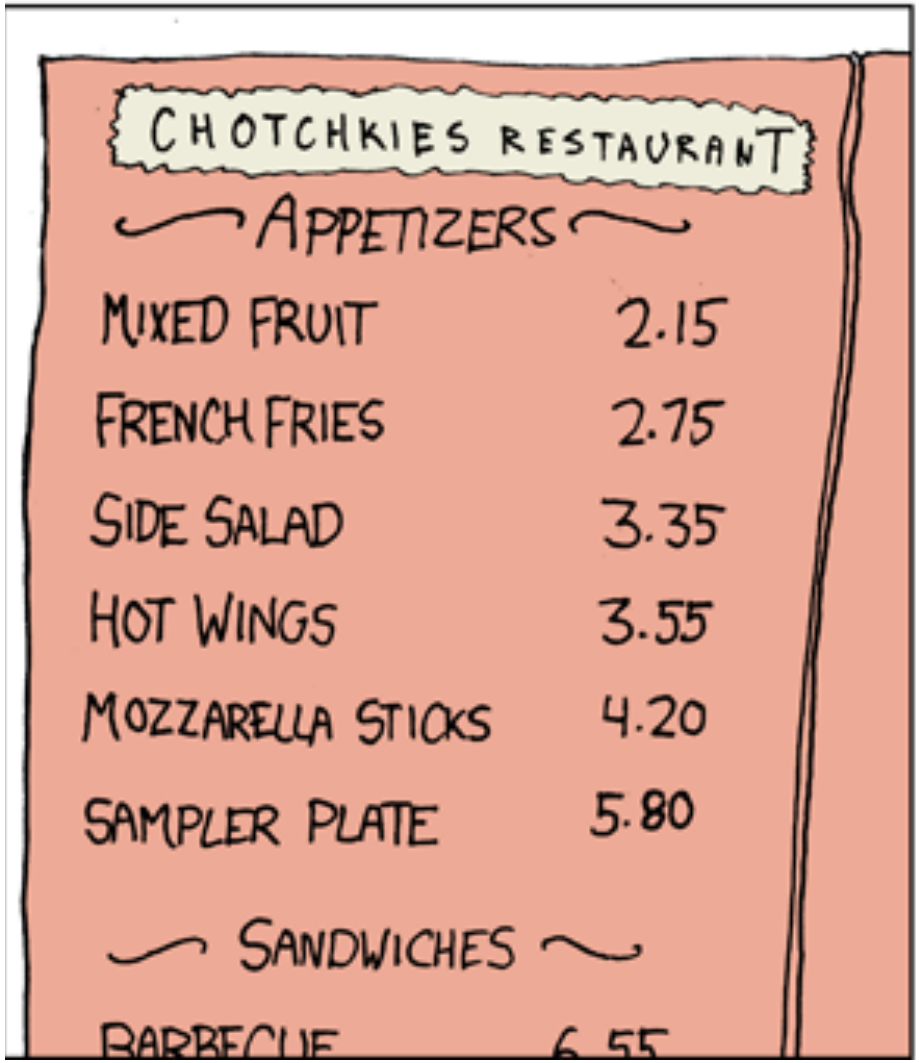
Knapsack approximation algorithm. There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum.
[Section 11.8, Master course on Advanced Algorithms]

Knapsack Problem: Finding a Solution

Construct optimal solution from computed information.

```
Run Knapsack()
Run Find-Solution(n,W)

Find-Solution(i,w) {
   if (i = 0 or w = 0)
      output nothing
   else if ( M[i,w] = M[i-1, w] )
         Find-Solution(i-1,w)
   else
         Find-Solution(i-1,w-w_i)
         print i
}
```

MY HOBBY:
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS



http://xkcd.com/287/