Computer Algorithms

Segment 4

Greedy Method

Greedy Algorithm

- Algorithm for optimization problems typically go through a sequence of steps, with a set of choices at each step.
- For many optimization problems, greedy algorithm can be used. (not always)
- Greedy algorithm always makes the choice that looks best at the moment.
- It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
- Example:
  - Activity Selection Problem
  - Dijkstra's Shortest Path Problem
  - Minimum Spanning Tree Problem

Greedy Strategy

- 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.

- The activity-selection problem, for which a greedy algorithm efficiently computes a solution, works well for a wide range of problems, i.e, minimum-spanning-tree algorithms, Dijkstra's algorithm for shortest paths form a single source, and Chvátal's greedy set-covering heuristic.

Greedy Strategy (con.)

- The first key ingredient is the *greedy-choice property*: a globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- In a greedy algorithm, choice is determined on fly at each step, (while algorithm progresses), may seem to be best at the moment and then solves the subproblems after the choice is made. The choice made by a greedy algorithm may be depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems. Thus, unlike dynamic programming, which solves the subproblems bottom up, a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, iteratively reducing each given problem instance to a smaller one.

Greedy Strategy (con.)

- Algorithms for optimization problems typically go through a sequence of steps, using dynamic programming to determine the best choices, (bottom-up) for subproblem solutions.
- But in many cases much simpler, more efficient algorithms are possible. A *greedy algorithm* always makes the choice for a locally optimal solution which seems the best at the current moment with the hope of a globally optimal solution. In the most cases does not always yield optimal solutions, but for many problems.
- Greedy algorithms constructs a solution to an *optimization problem* piece by piece through a sequence of choices that are:
  - *Feasible*
  - *Locally optimal*
  - *Irrevocable* (binding and abiding)

Greedy Strategy (con.)

- Start with a solution to a small subproblem
- Build up to a solution to the whole problem
- Make choices that look good in the short term
- Disadvantage: Greedy algorithms don't always work (Short term solutions can be disastrous in the long term). Hard to prove correct
- Advantage: Greedy algorithm work fast when they work. Simple algorithm, easy to implement

Applications of the Greedy Strategy

- Optimal solutions:
  - change making for “normal” coin denominations
  - minimum spanning tree (MST)
  - single-source shortest paths
  - simple scheduling problems
  - Huffman codes

- Approximations:
  - traveling salesman problem (TSP)
  - knapsack problem
  - other combinatorial optimization problems

Greedy Algorithm Design

Comparison:

Dynamic Programming	Greedy Algorithms
At each step, the choice is determined based on solutions of subproblems.	At each step, we quickly make a choice that currently looks best. --A local optimal (greedy) choice.
Sub-problems are solved first.	Greedy choice can be made first before solving further sub-problems.
Bottom-up approach	Top-down approach
Can be slower, more complex	Usually faster, simpler

Activity selection problem

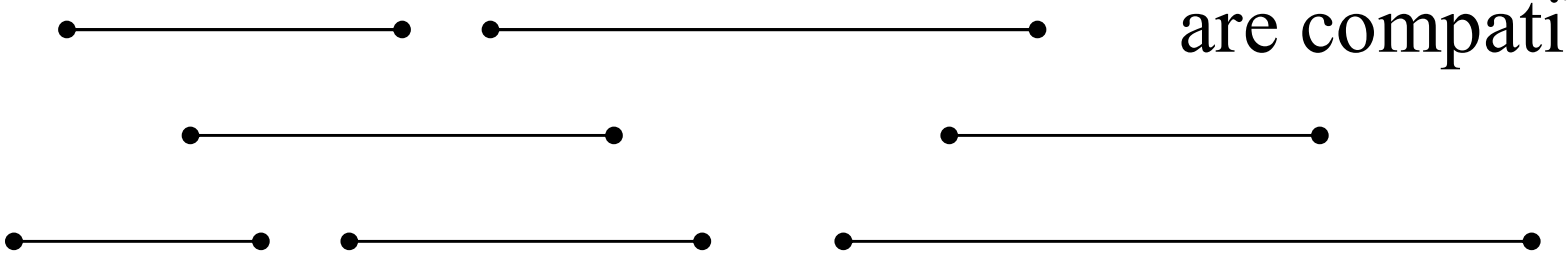
- The problem is to select a maximum-size set of mutally compatible activities.
- Example
- We have a set S = { 1,2,…,n} of n proposed activities that wish to use a resource, such as a lecture hall, which can be used by only one activities at a time.
- Input: Set S of n activities, a_1, a_2, …, a_n.

  s_i = start time of activity i.

  f_i = finish time of activity i.
- Output: Subset A of maximum number of compatible activities.
  - Two activities are compatible, if their intervals don'$_9$t overlap.

Activity selection problem(con.)

Example:



Activities in each line are compatible.

Activity selection problem(con.)

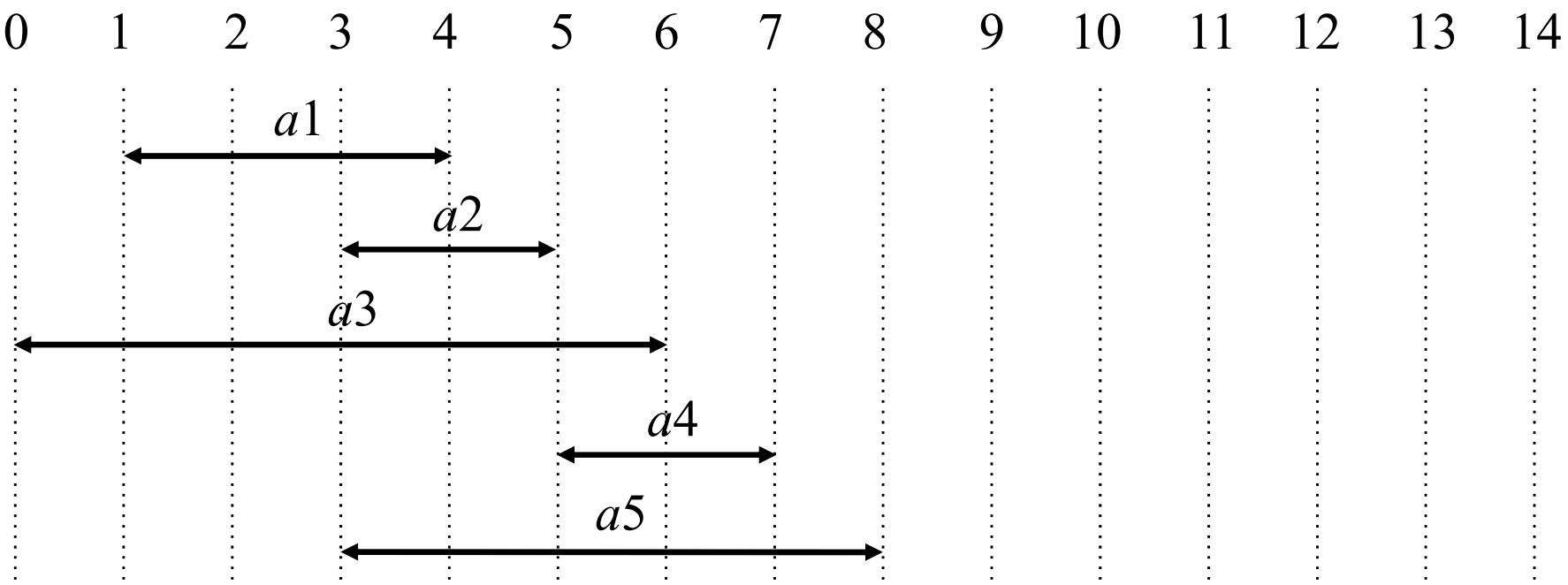
- **An activity selection problem**
  - To select a maximum-size subset of mutually compatible activities.
  - For example
    - Given n classes and 1 lecture room,
    - to select the maximum number of classes
  - A set of ***activities***: $S = \{a_1, a_2, ..., a_n\}$
  - Each activity a_i has its ***start time*** s_i and ***finish time*** f_i.
    - $0 \le s_i < f_i < \infty$
  - Activity a_i takes place during $[s_i, f_i)$
  - Activities a_i and a_j are ***compatible***

    if the intervals $[s_i, f_i)$ and $[s_j f_j)$ do not overlap (i.e., a_i , a_j are compitible if $s_i \geq f_j$ or $s_j \geq f_i$)

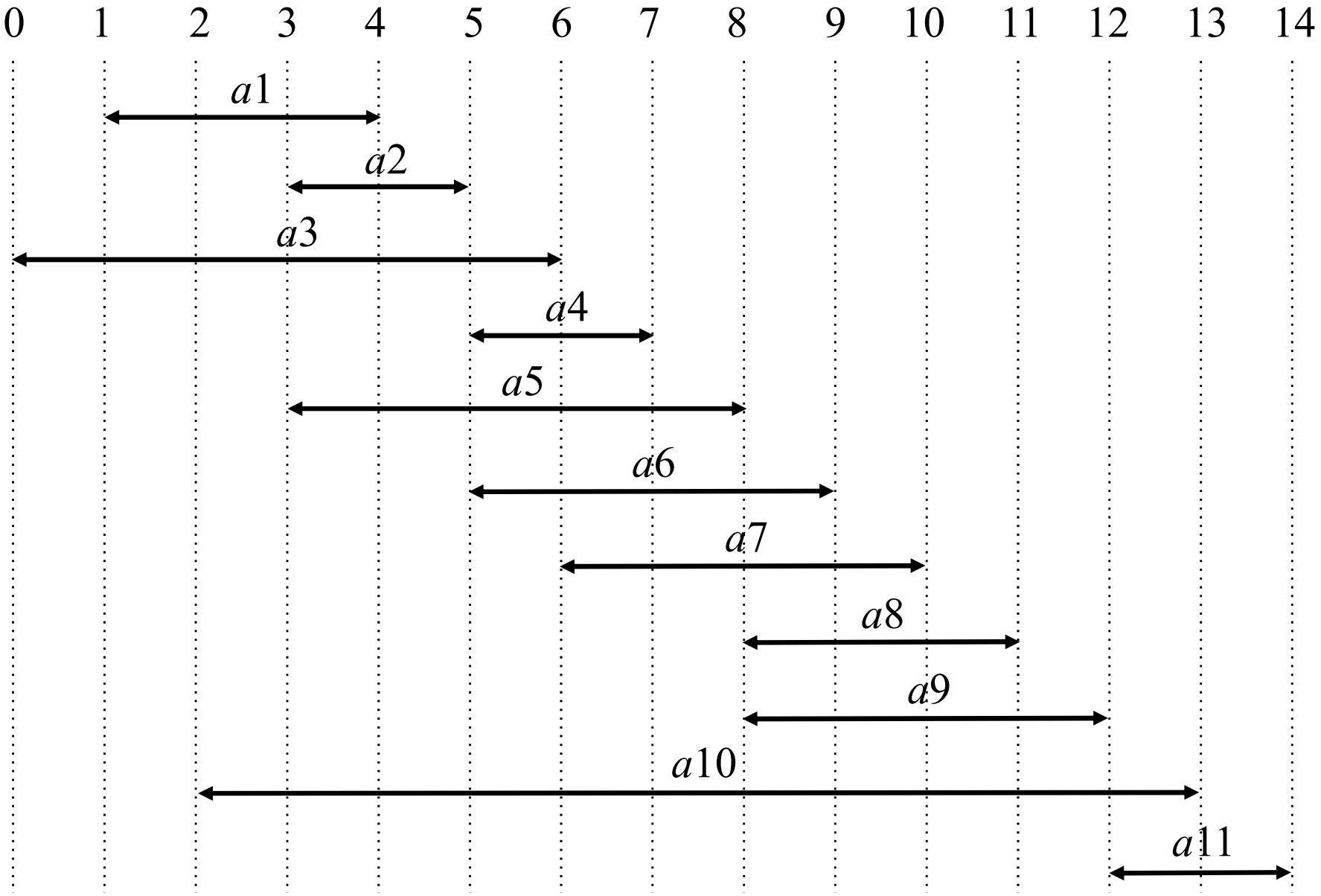
Activity selection problem(con.)

Consider the following set S of activities, which we have sorted in monotonically increasing 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

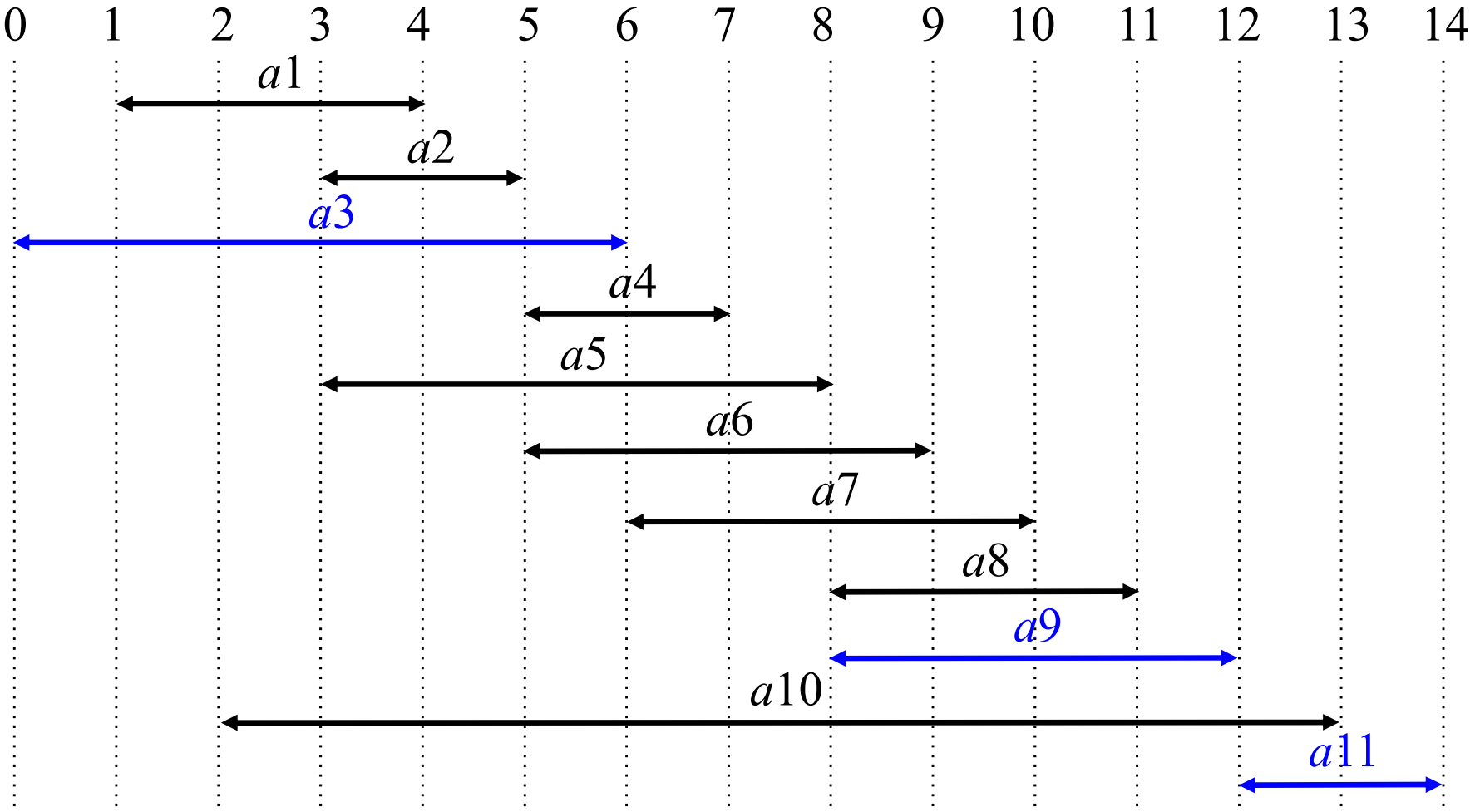


Activity selection problem(con.)



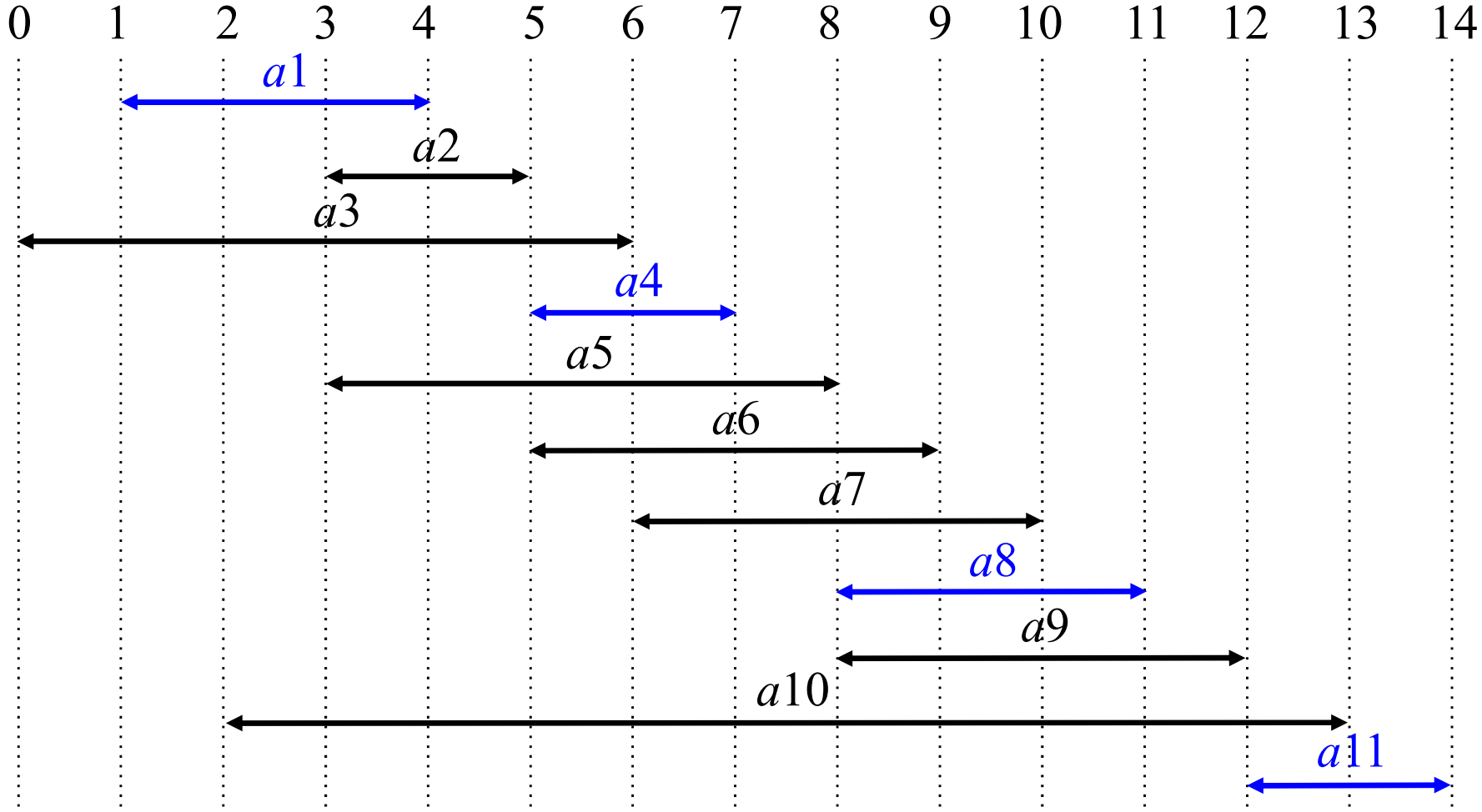
Activity selection problem(con.)

- $\{a3, a9, a11\}$: mutually compatible activities, not a largest set



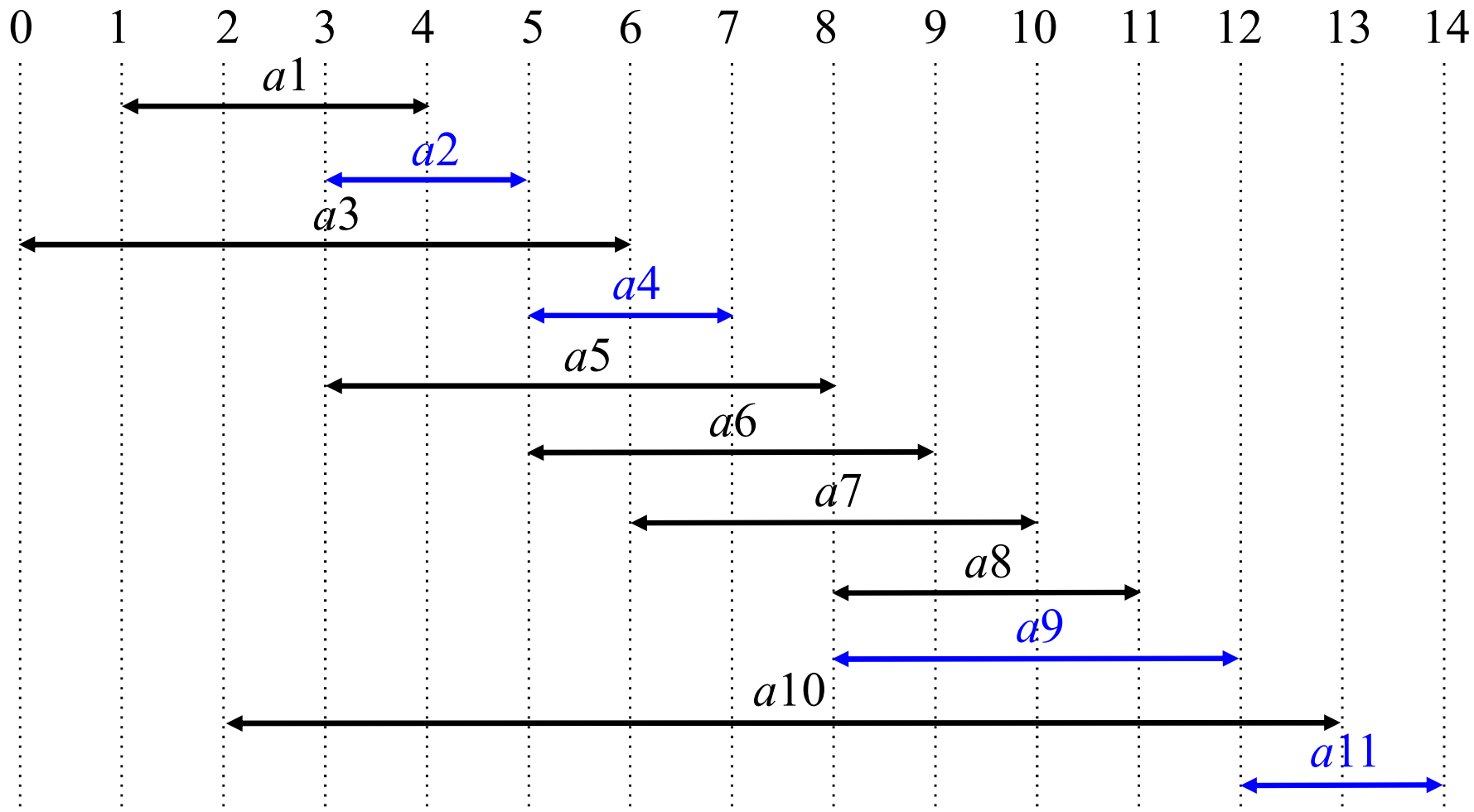
Activity selection problem(con.)

- $\{a1, a4, a8, a11\}$: A largest set of mutually compatible activities



Activity selection problem(con.)

- $\{a2, a4, a9, a11\}$: Another largest subset



Activity selection problem(con.)

- **Optimal substructure**
  - S_{ij} denote the set of activities between a_i and a_j and compatible with a_i and a_j.
    - Activities start after a_i finishes and finish before a_j starts.

$$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$$

    - For example, $S_{18} = \{a_4\}$

Activity selection problem(con.)

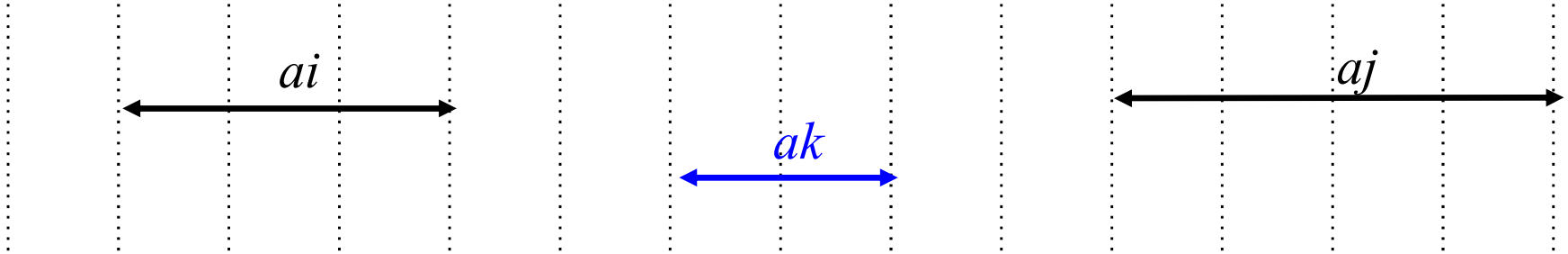
- **Optimal substructure**
  - We define a_0 and a_{n+1} such that $f_0 = 0$ and $s_{n+1} = \infty$.
  - $S = S_{0,n+1}$ for $0 \le i, j \le n + 1$.
  - Assume that activities are sorted in increasing order of finish time.

$$f_0 \le f_1 \le f_2 \le \cdots \le f_n < f_{n+1}$$

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

Activity selection problem(con.)

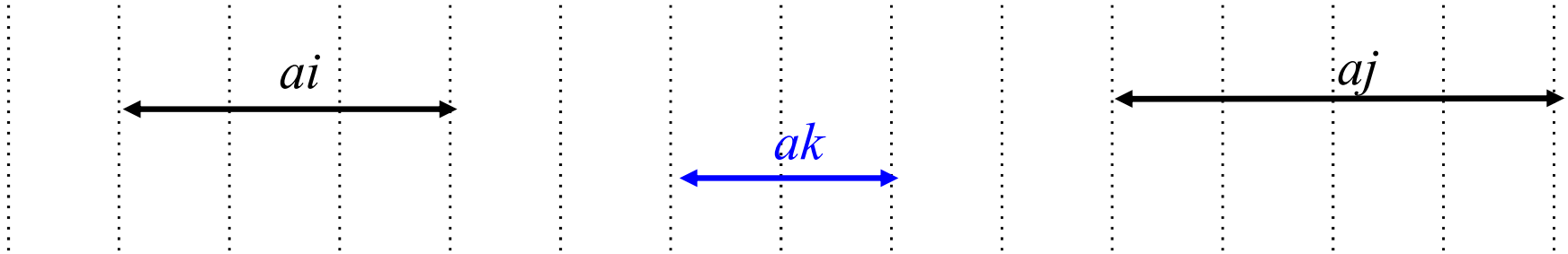
- **Optimal substructure**
  - A_{ij} denote an optimal solution to S_{ij} for $i \leq j$.
  - $S_{ij} = \emptyset$ if i>=j
  - If A_{ij} includes a_k, $A_{ij} = A_{ik}$ U $\{a_k\}$ U A_{kj}



Activity selection problem(con.)

- **A recursive solution**
  - $c[i, j]$: The number of activities in A_{ij}.

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



Activity selection problem(con.)

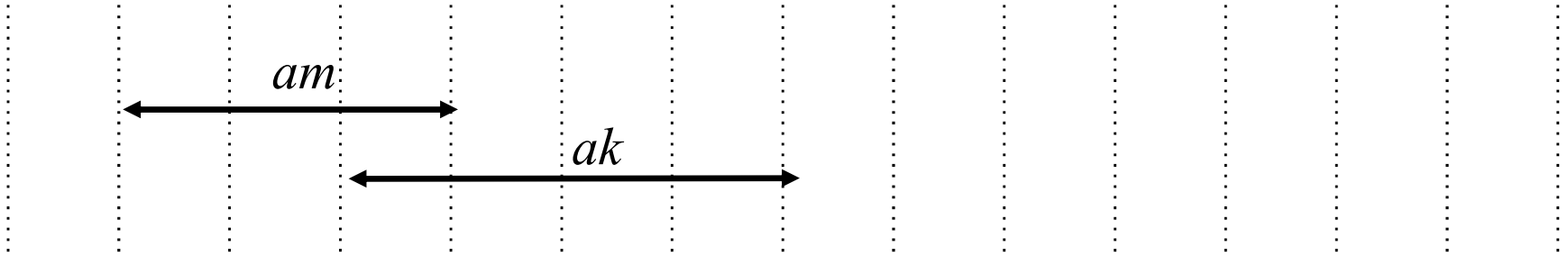
- **Greedy algorithm**
  - Consider any nonempty S_{ij}, and let a_m be the activity in S_{ij} with the earliest finish time: $f_m = \min \{f_k : a_k \in S_{ij}\}$.

  Then

  1. Activity a_m is in some A_{ij}.
  2. The subproblem S_{im} is empty, so that choosing a_m leaves the subproblem S_{mj} as the only one that may be nonempty.

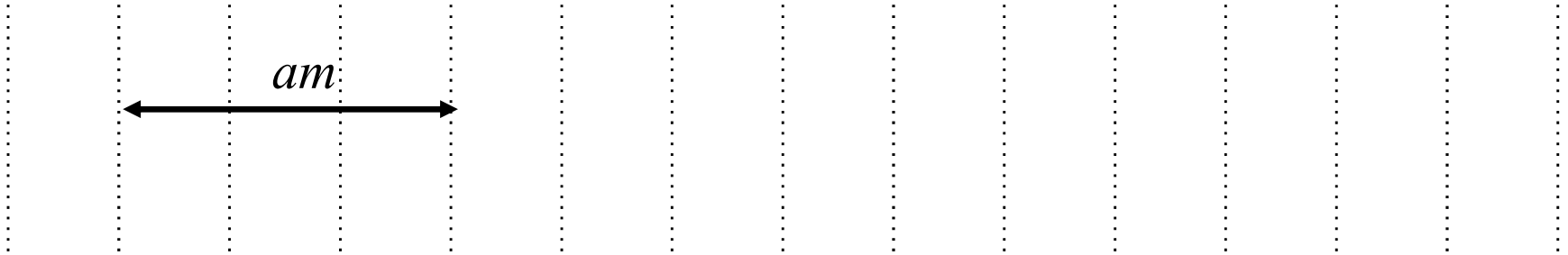
Activity selection problem(con.)

- Activity a_m is in some A_{ij}.
  - a_k: the first activity in A_{ij}
  - If $a_k = a_m$, done.
  - If $a_k \neq a_m$, remove a_k from A_{ij} add a_m. The resulting A_{ij} is another optimal solution because $f_m \leq f_k$.



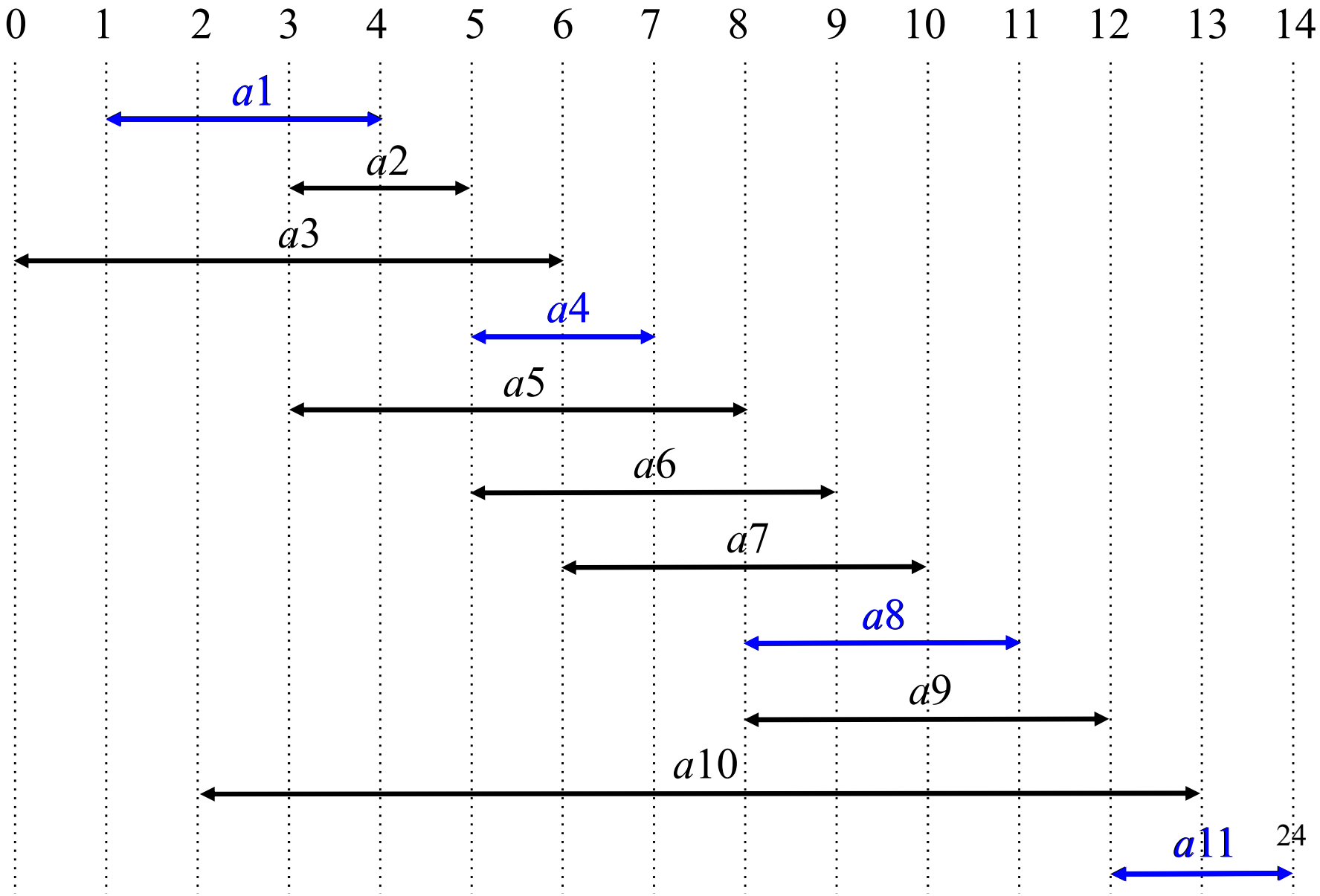
Activity selection problem(con.)

- The subproblem S_{im} is empty, so that choosing a_m leaves the subproblem S_{mj} as the only one that may be nonempty.
  - S_{im} is empty because a_m has the earlier finish time in S_{ij}.



Activity selection problem(con.)

- **Greedy algorithm**
  - Select the earliest finishing activity one by one.



Activity-Selection Problem

Greedy Strategy Solution

```
Recursive-Activity-Selector(i,j)
1    m = i+1
     // Find first activity in S_i,j
2    while m < j and start_time_m < finish_time_i
3         do m = m + 1
4    if m < j
5         then return {a_m} U Recursive-Activity-Selector(m,j)
6    else return Ø
```

m=2	m=3	m=4
Okay	Okay	break the loop

{1,4,8,11} Order of calls:

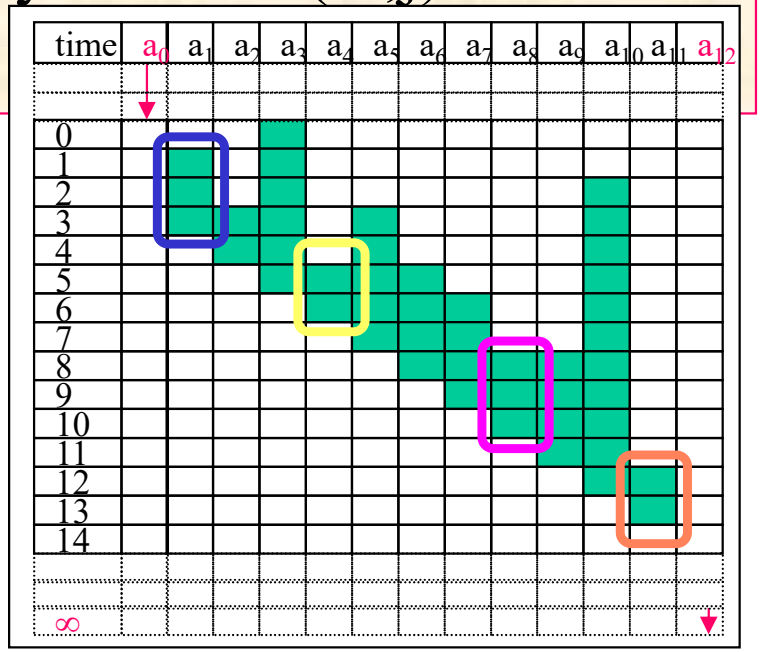
{4,8,11} Recursive-Activity-Selector(0,12)

{8,11} Recursive-Activity-Selector(1,12)

{11} Recursive-Activity-Selector(4,12)

Ø Recursive-Activity-Selector(8,12)

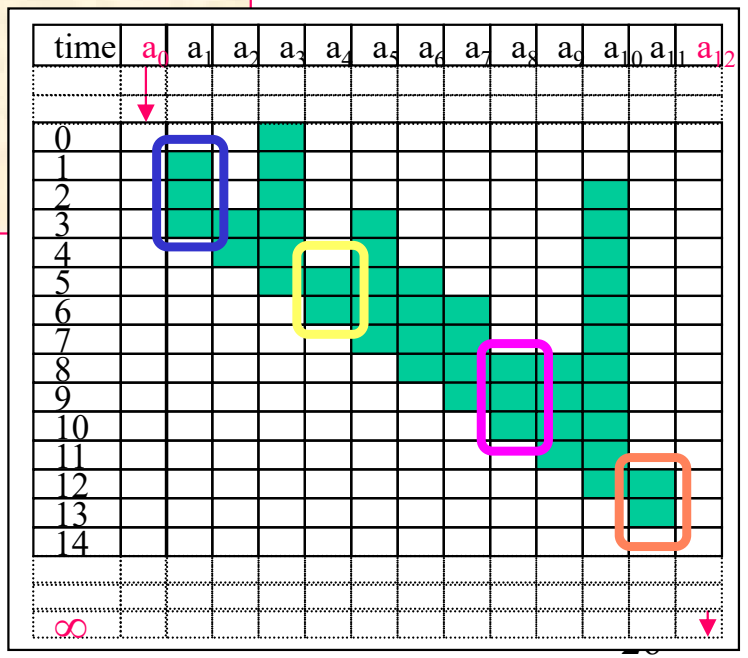
Recursive-Activity-Selector(11,12)



Activity-Selection Problem

Greedy Strategy Solution

```
Greedy-Activity-Selector(s,f)
1   n =length[s]
2   A={a1}
3   i=1
4   for m = 2 to n
5       do if sm>= fi
6       then   A = A U {am}
7                 i = m
8   return A
```



Steps of the Greedy Strategy

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution
3. Prove that at any stage of the recursion, one of the optimal choices is the greedy choice. Thus, it is always safe to make the greedy choice.
4. Show that all but one of the subproblems induced by having made the greedy choice are empty.
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

Elements of the Greedy Strategy

- How can one tell if a greedy algorithm will solve a particular optimization problem??
- There is no way in general. But there are 2 ingredients exhibited by most greedy problems:

1. Greedy Choice Property
2. Optimal Sub Structure

Greedy Choice Property

- A globally optimal solution can be arrived at by making a locally optimal (**Greedy**) choice.
- We make whatever choice seems best at the moment and then solve the sub problems arising after the choice is made.
- The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any **future choices** or on the solutions to sub problems.
- Thus, a greedy strategy usually progresses in a ***top-down*** fashion, making one greedy choice after another, iteratively reducing each given problem instance to a smaller one.

Optimal Sub Structure

- A problem exhibits ***optimal substructure*** if an optimal solution to the problem contains (within it) optimal solution to sub problems.
- In **Activity Selection Problem**, an optimal solution A begins with activity 1, then the set of activities $\bar{A} = A - \{1\}$ is an optimal solution to the activity selection problem

  $\acute{S} = \{i \in S: s_i >= f_1\}$

Knapsack Problem

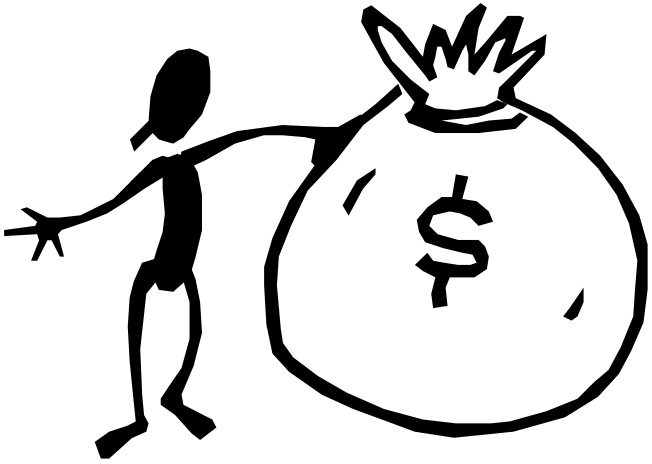
- We are given n objects and a knapsack. Object i has a weight w_i and the knapsack has a capacity M.
- If a fraction x_i, $0 \le x_i \le 1$, of object i is placed into the knapsack then a profit of $p_i x_i$ is earned.
- The objective is to obtain a filling of the knapsack that maximizes the total weight of all chosen objects to be at most M

maximize $\sum_{1 \le i \le n} p_i x_i$

subject to $\sum_{1 \le i \le n} w_i x_i \le M$

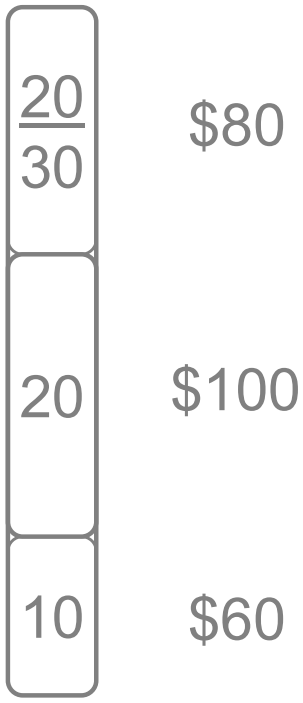
and $0 \le xi \le 1$, $1 \le i \le n$

Example



Fractional Knapsack

Taking the items in order of greatest value per pound yields an optimal solution

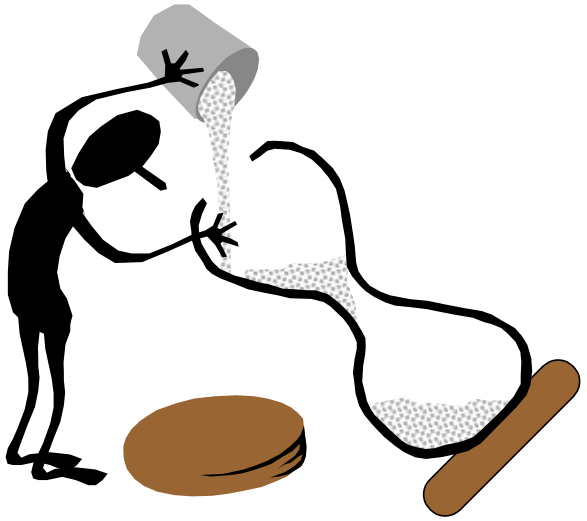


$80

$100

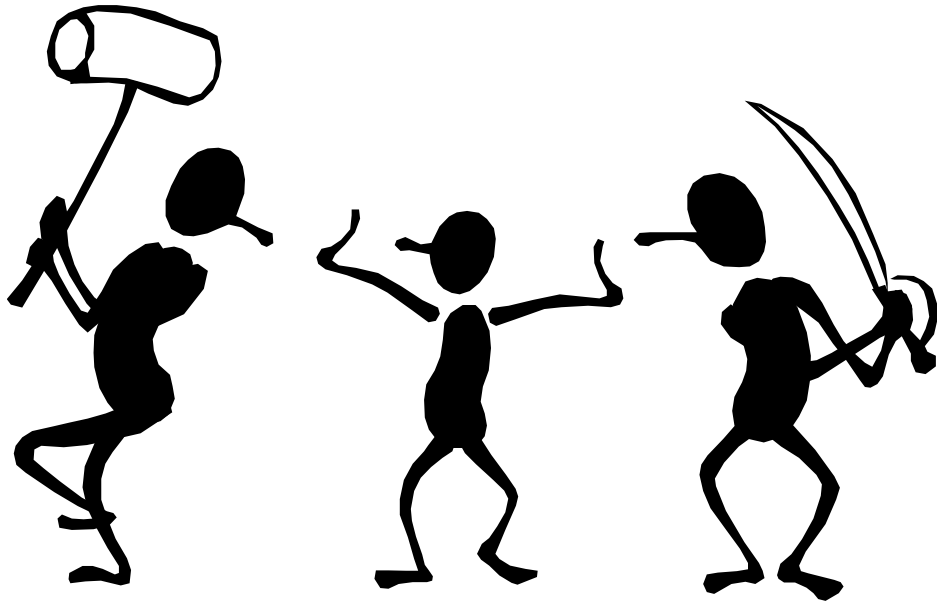
$60

Total =$240



Optimal Substructure

- Both fractional knapsack and 0/1 knapsack have an optimal substructure.



Example Fractional Knapsack

Price ($US)	20	30	65	48	50
weight (Lbs.)	10	20	30	40	50
price/weight	2	1.5	2.1	1.2	1

- Choose the most price/ weight first
  - Total weight = 30 + 10 + 20 + 40 = 100
  - Total Price = 65 + 20 + 30 + 48 = 163

More Example on fractional knapsac

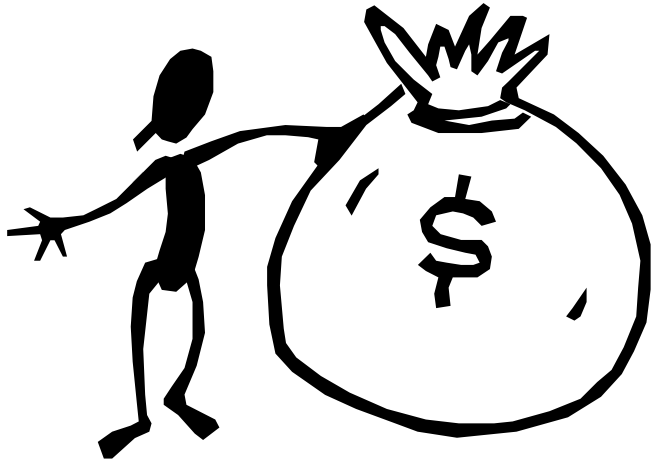
- Consider the following instance of the knapsack problem: n = 3, M = 20, $(p_1,p_2,p_3) = 25,24,15$ and $(w_1,w_2,w_3) = (18,15,10)$

	(x1,x2,x3)	$\sum w_i x_i$	$\sum p_i x_i$
1)	(1/2,1/3,1/4)	16.5	24.25
2)	(1,2/15,0)	20	28.2
3)	(0,2/3,1)	20	31
4)	(0,1,1/2)	20	31.5

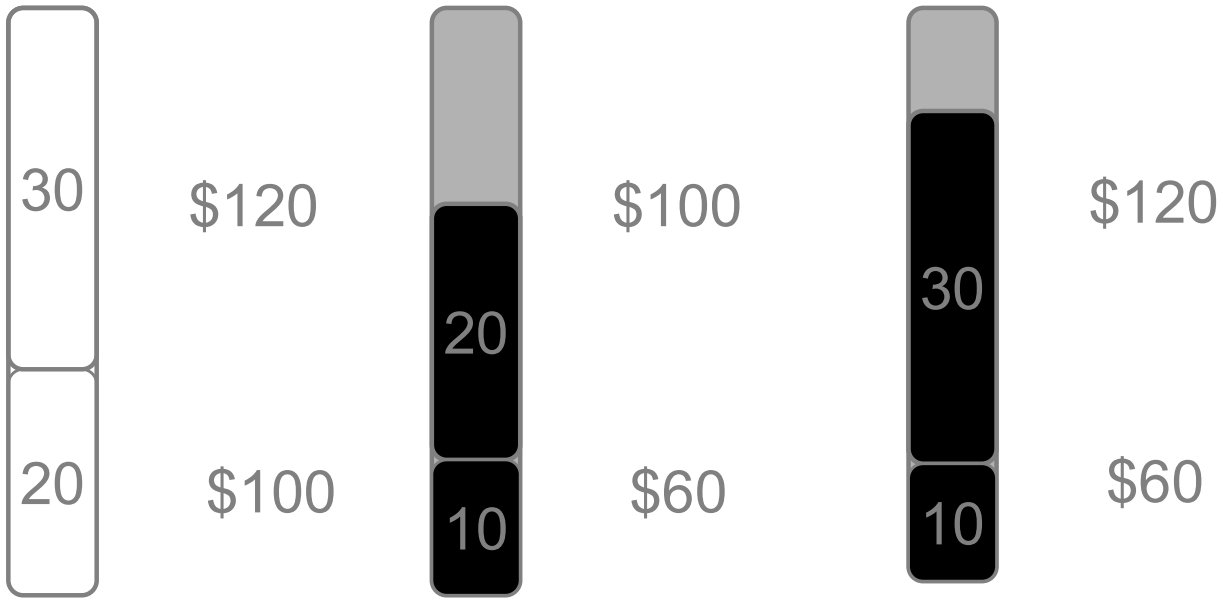
PseudoCode

```
Algorithm GreedyKnapsack(m,n)
//p(1:n) and w(1:n) contain the profits and weights respectively of the n objects
// ordered so that p(i)/w(i) ≥  p(i+1)/w(i+1). m is the knapsack size and x(1:n) is
// the solution vector
{
   for i:=1 to n do x[i]:=0.0;  // Initialized x.
      U:=m;
      for i:=1 to n do
       {
         if(w[i]>U) then break;
         x[i] :=1.0; U:=U-w[i];
       }
       if (i<=n) then x[i] :=U/w[i];
}
```

Knapsack 0/1 Can not be solved by greedy strategy



Knapsack 0/1 Can not be solved by greedy strategy



The value per pound of item 1 is 6 dollars per pound, which is greater than the value per pound of either item 2 (5 dollars per pound) or item 3 (4 dollars per pound). Therefore, the greedy strategy would take item 1 first. As can be seen from the case analysis in above figure, the optimal solution takes items 2 and 3, leaving 1 behind. The two possible solutions that involve item 1 are both suboptimal.

Huffman Codes

Huffman Codes

- For compressing data (sequence of characters)
- Widely used
- Very efficient (saving 20-90%)
- Use a table to keep frequencies of occurrence of characters.
- Output binary string.

"Today's weather is nice"

"001 0110 0 0 100 1000 1110"

Huffman Codes

Example:

A file of 100,000 characters.
Containing only ‘a’ to ‘f’

	Frequency	Fixed-length codeword	Variable-length codeword
‘a’	45000	000	0
‘b’	13000	001	101
‘c’	12000	010	100
‘d’	16000	011	111
‘e’	9000	100	1101
‘f’	5000	101	1100

eg. “abc” = “000001010”

300,000 bits

eg. “abc” = “0101100”

1*45000 + 3*13000 + 3*12000 +
3*16000 + 4*9000 + 4*5000
= 224,000 bits

Huffman Codes

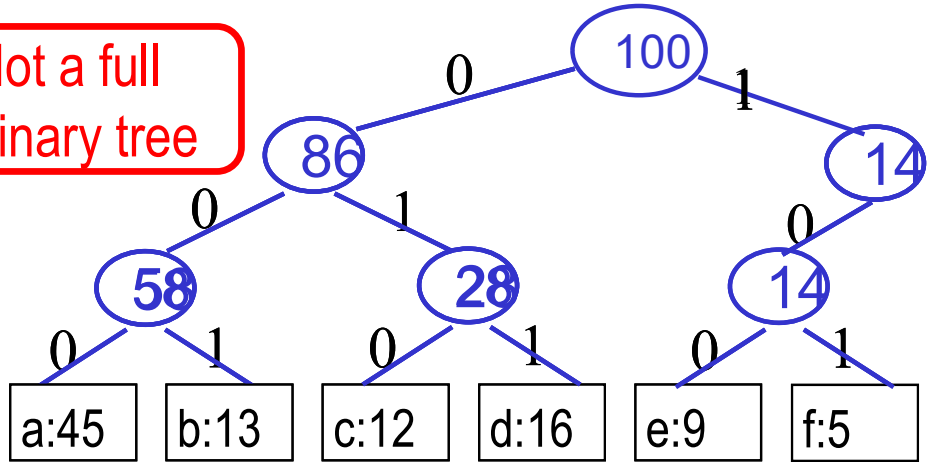
A file of 100,000 characters.

The coding schemes can be represented by trees:

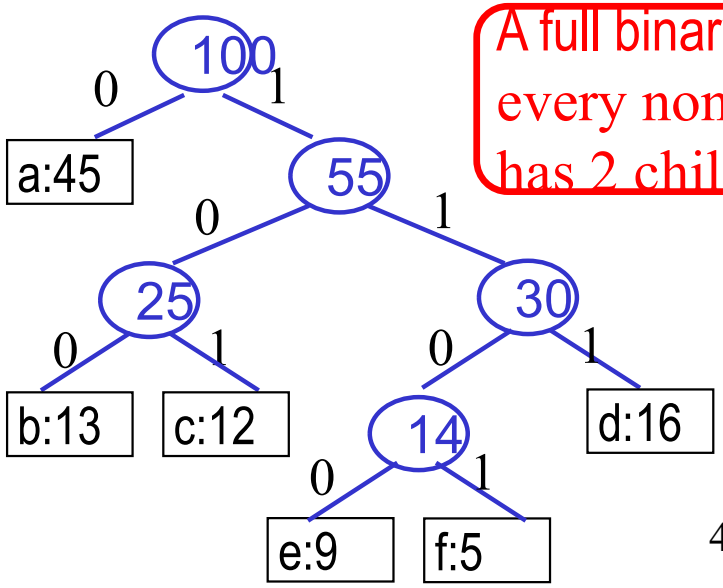
	Frequency (in thousands)	Fixed-length codeword
'a'	45	000
'b'	13	001
'c'	12	010
'd'	16	011
'e'	9	100
'f'	5	101

	Frequency (in thousands)	Variable-length codeword
'a'	45	0
'b'	13	100
'c'	12	101
'd'	16	111
'e'	9	1100
'f'	5	1101

Not a full binary tree

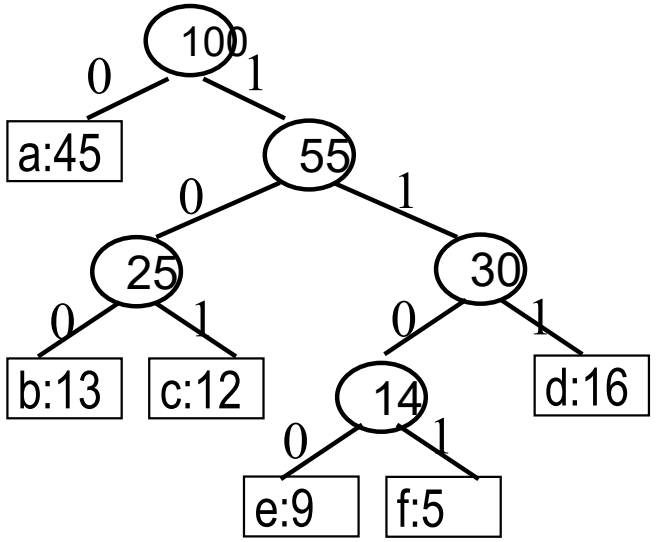


A full binary tree every nonleaf node has 2 children



Huffman Codes

	Frequency	Codeword
‘a’	45000	0
‘b’	13000	100
‘c’	12000	100
‘d’	16000	111
‘e’	9000	1100
‘f’	5000	1101



Eg. “abc” is coded as “0101100”

To find an optimal code for a file:

1. The coding must be unambiguous.

Consider codes in which no codeword is also a prefix of other codeword. => Prefix Codes

Prefix Codes are unambiguous.

Once the codewords are decided, it is easy to compress (encode) and decompress (decode).

2. File size must be smallest.

=> Can be represented by a full binary tree.

=> Usually less frequent characters are at bottom

Let C be the alphabet (eg. C={‘a’,’b’,’c’,’d’,’e’,’f’})

For each character c, no. of bits to encode all c’s occurrences = $freq_c * depth_c$

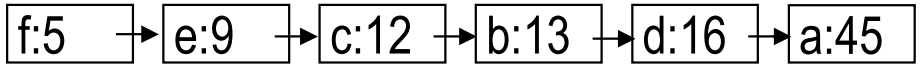
File size $B(T) = \Sigma_{c \in C} freq_c * depth_c$

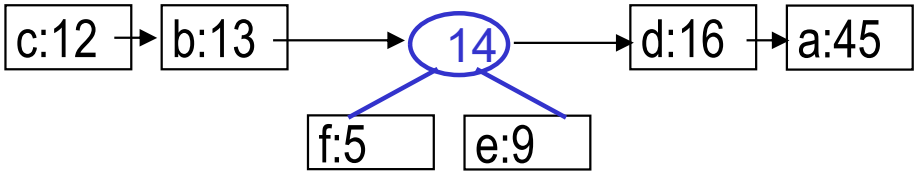
Huffman Codes

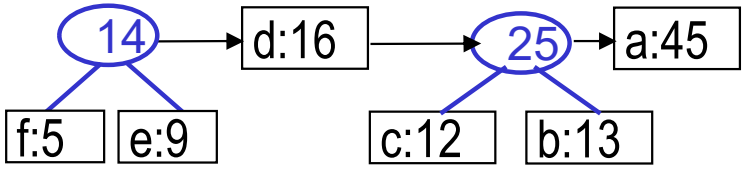
How do we find the optimal prefix code?

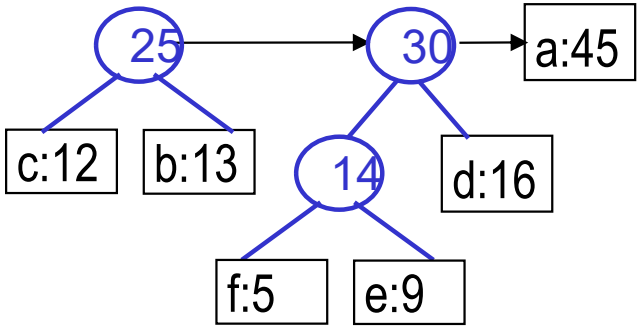
Huffman code (1952) was invented to solve it.
A Greedy Approach.

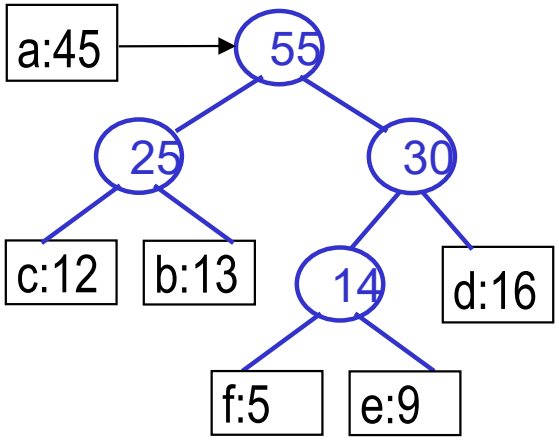
Q: A min-priority

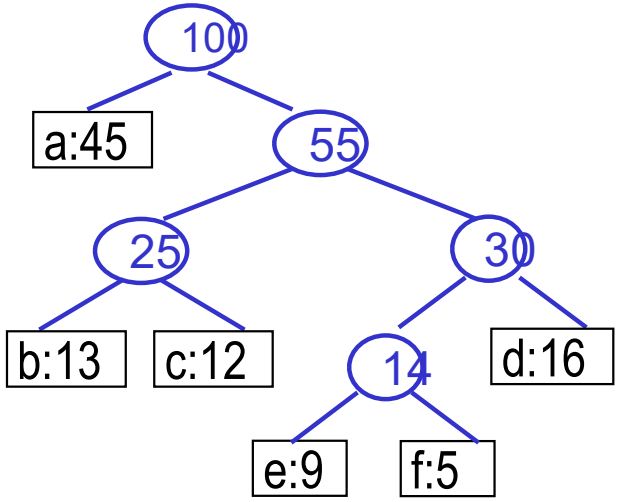






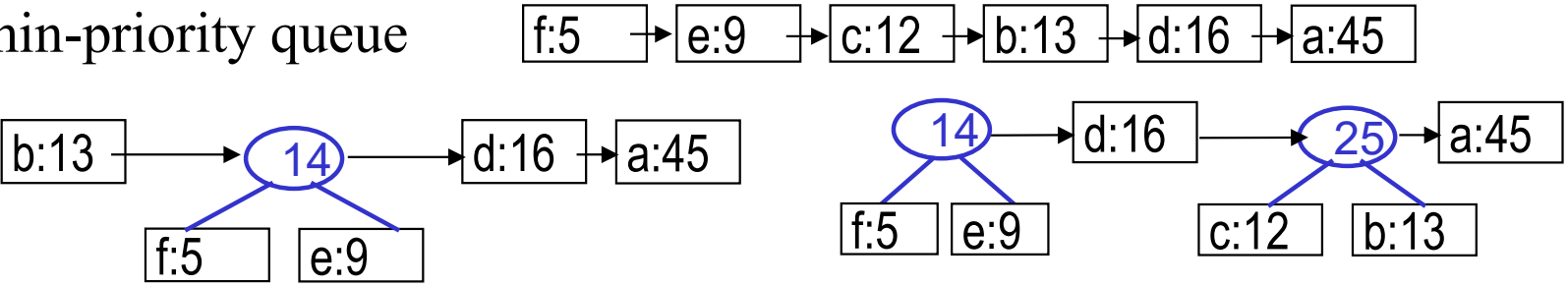






Huffman Codes

Q: A min-priority queue



. . . .

- **Coding Cost** of T: $B(T)=\sum_{c \in C} f(c)d_T(c)$
  - c : each character in the alphabet C
  - $f(c)$: frequency of c
  - $d_T(c)$: depth of c's leaf (length of the codeword of c)
- **Code design:** Given $f(c_1), f(c_2), \ldots, f(c_n)$, construct a binary tree with n leaves such that $B(T)$ is minimized.
  - Idea: more frequently used characters use shorter depth.
- Time complexity: $O(n \lg n)$.
  - Extract-Min(Q) needs $O(\lg n)$ by a **heap** operation.
  - Requires initially $O(n \lg n)$ time to build a binary heap.

```
Huffman(C)
1. n ← |C|;
2. Q ← C;
3. for I ← 1 to n-1
4.     z ← Allocate-Node();
5.     x ← left[z] ← Extract-Min(Q);
6.     y ← right[z] ← Extract-Min(Q);
7.         f[z] ← f[x]+f[y];
8.         Insert(Q, z);
9. return Extract-Min(Q)
```

The Traveling Salesman Problem (TSP)

Understanding the TSP

The traveling salesman problem consists of a salesman and a set of cities.

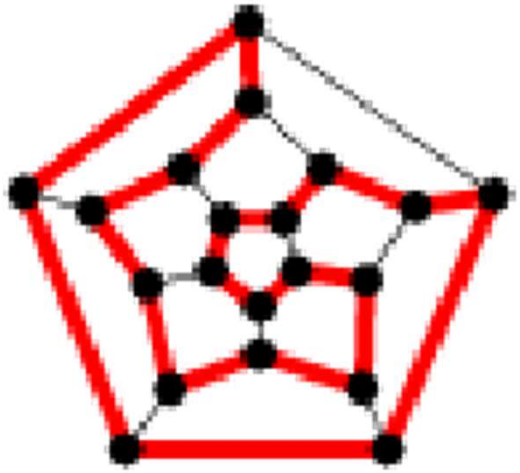
The salesman has to visit each one of the cities starting from a certain one (e.g. the hometown) and returning to the same city.

The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip.

- Complete Graph – vertices joined by a single edge
- Weighted Graph – edges carry a value

Understanding the TSP

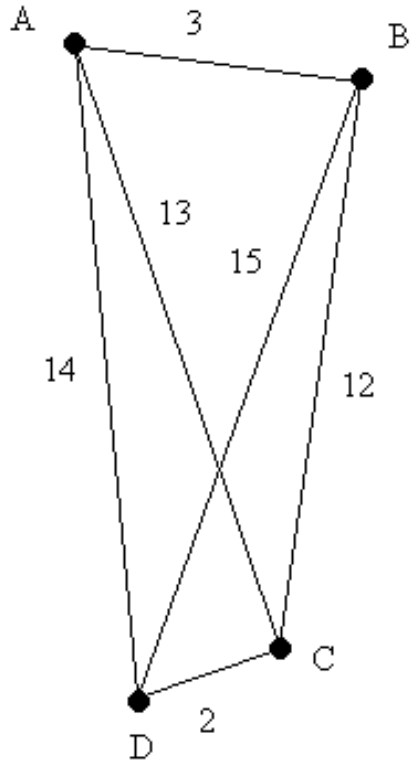
- Hamilton Circuit – connects all points on a graph, passes through each point only once, returns to origin
- Optimal Hamilton Circuit – smallest possible weight



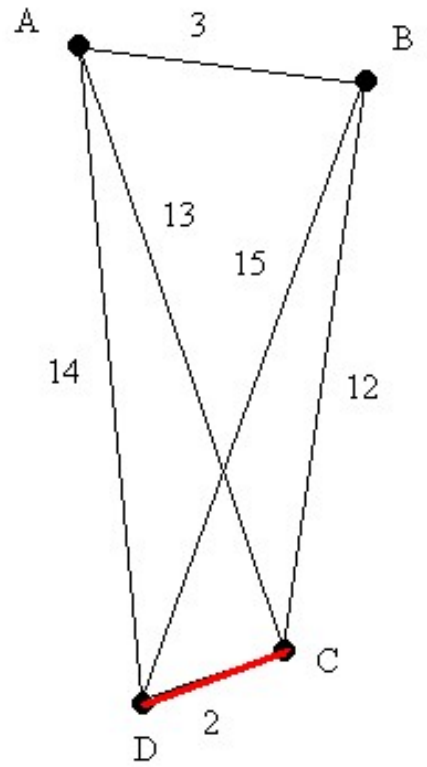
Finding an Approximate Solution

- Cheapest Link Algorigthm
  - Edge with smallest weight is drawn first
  - Edge with second smallest weight is drawn in
  - Continue unless it closes a smaller circuit or three edges come out of one vertex
  - Finished once a complete Hamilton Circuit is drawn

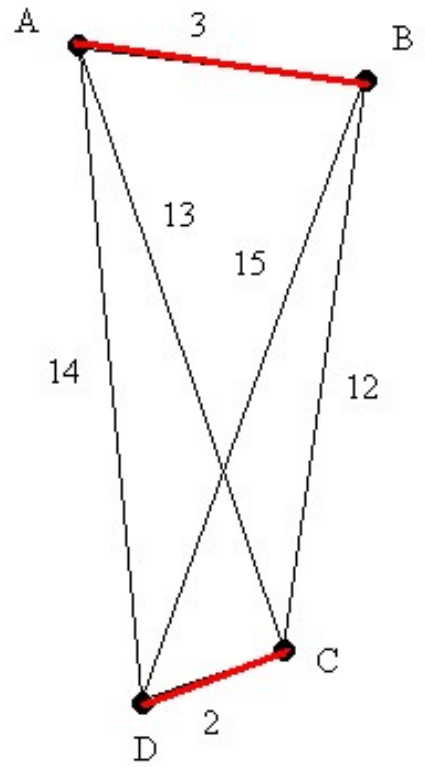
Cheapest Link Algorithm



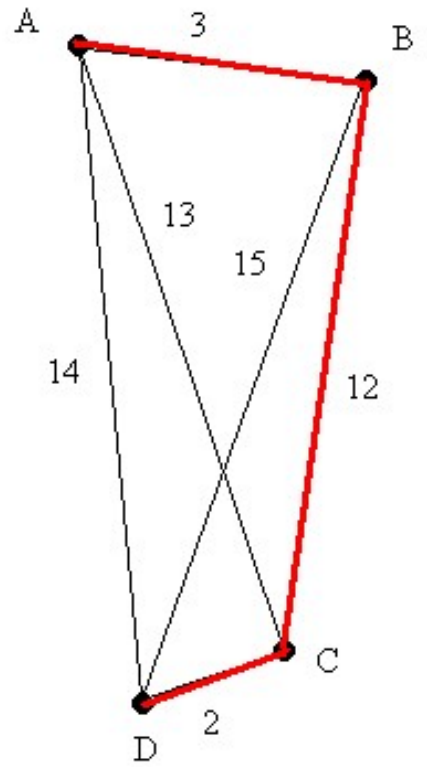
Cheapest Link Algorithm



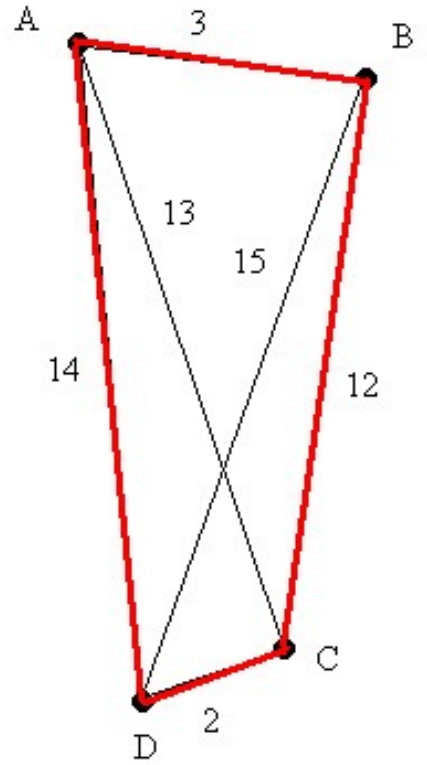
Cheapest Link Algorithm



Cheapest Link Algorithm



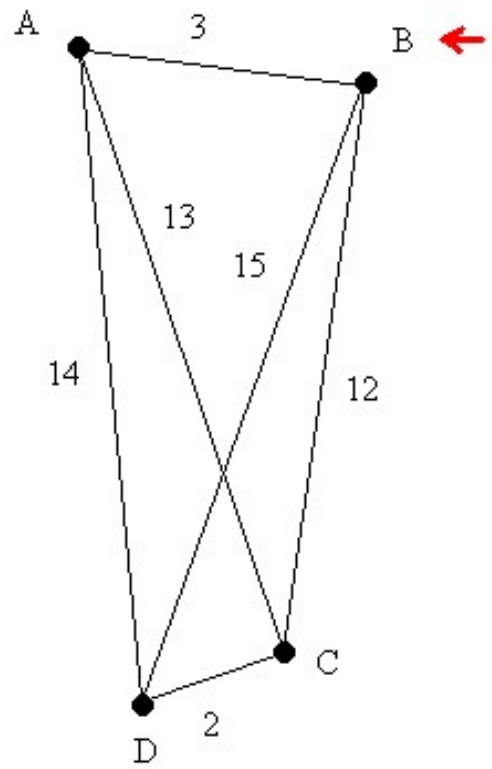
Cheapest Link Algorithm



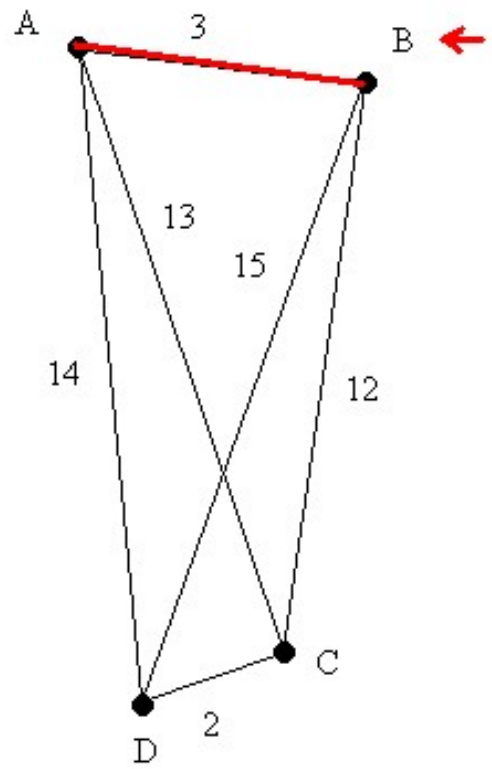
Finding an Approximate Solution

- Nearest Neighbor Algorithm
  - Start at any given vertex
  - Travel to edge that yields smallest weight and has not been traveled through yet
  - Continue until we have a complete Hamilton circuit

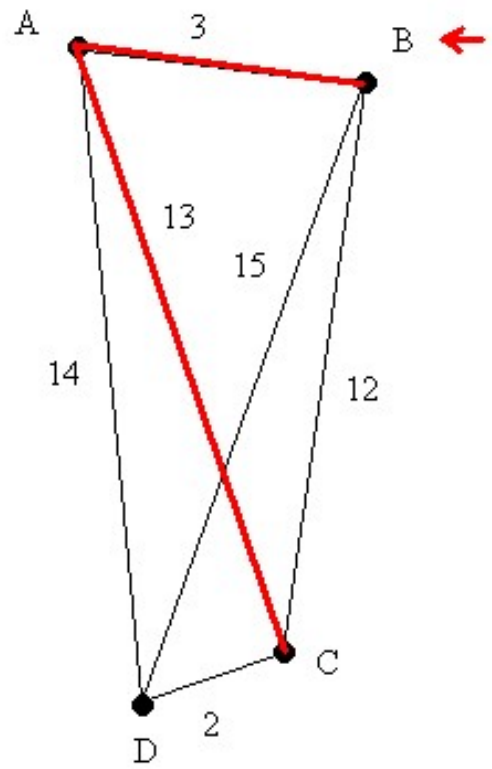
Nearest Neighbor Algorithm



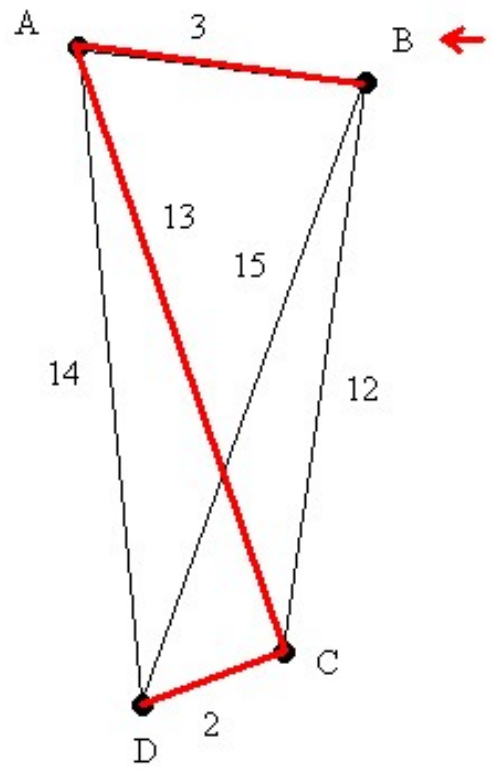
Nearest Neighbor Algorithm



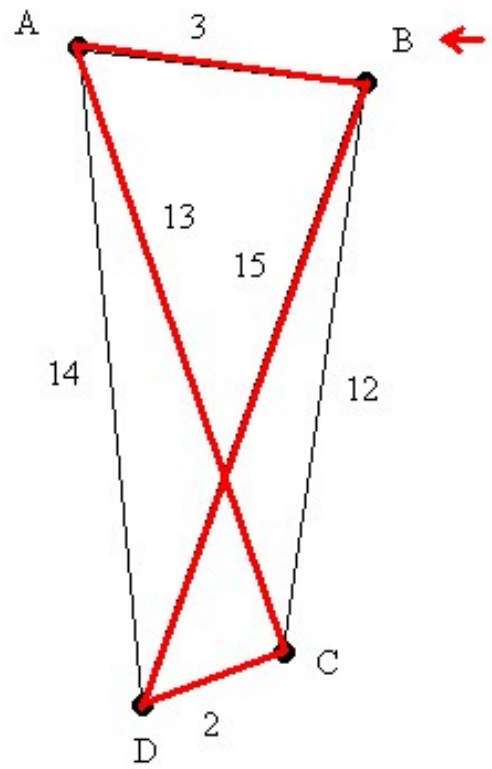
Nearest Neighbor Algorithm



Nearest Neighbor Algorithm



Nearest Neighbor Algorithm



The Nearest Neighbour Algorithm

To find a (reasonably good) Hamiltonian cycle i.e. a closed trail containing every node of a graph

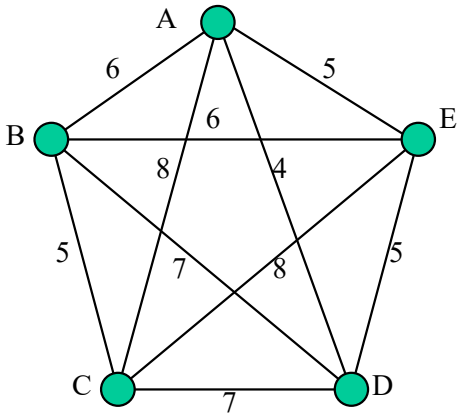
Step 1 Choose any starting node

Step 2 Consider the arcs which join the node just chosen to nodes as yet unchosen. Pick the one with minimum weight and add it to the cycle

Step 3 Repeat step 2 until all nodes have been chosen

Step 4 Then add the arc that joins the last-chosen node to the first-chosen node

To find a (reasonably good) Hamiltonian cycle i.e. a closed trail containing every node of this graph using the Nearest Neighbour Algorithm:



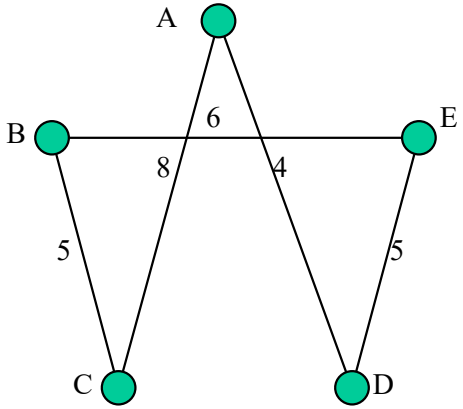
Choose any starting node – let's say A

Consider the arcs which join the node just chosen to nodes as yet unchosen. Pick the one with minimum weight and add it to the cycle

The first arc is AD as 4 is the least of 6, 8, 4 and 5

From D, DE is chosen as 5 is the least of 7, 7, and 5

Then EB, BC, and finally CA



The cycle is ADEBCA which has weight 28 (5 + 5 + 6 + 4 + 8)

The Nearest Neighbour Algorithm is 'greedy' because at each stage the immediately best route is chosen, without a look ahead, or back, to possible future problems or better solutions.