



# Greedy Algorithms

---





# List of Algorithms' Categories (Brief)

- Algorithms' types we will consider include:

- Simple Recursive Algorithms
- Backtracking Algorithms
- Divide and Conquer Algorithms
- Dynamic Programming Algorithms

- ➔ ■ Greedy Algorithms

- Branch and bound Algorithms
- Brute Force Algorithms
- Randomized Algorithms



# Optimization Problems

---

- An **Optimization Problem** is one in which we want to find, not just *a* solution, but the *best* solution
- A “Greedy Algorithm” sometimes works well for optimization problems
- A **Greedy Algorithm** works in phases. At each phase:
  - We take the best we can get right now, without regard for future consequences
  - We hope that by choosing a *local* optimum at each step, we will end up at a *global* optimum



# The Greedy Algorithms

---

- **The greedy method** is a general algorithm design paradigm, built on the following elements:
  - **Configurations**: different choices/collections/values to find
  - **Objective Function**: a score assigned to configurations, which we want to either maximize or minimize
- It works best (efficiently) when applied to problems with the **greedy-choice** property:
  - Globally-optimal solution can always be found by a series of local improvements from a starting configuration.

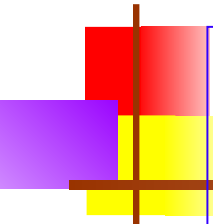


# Greedy Algorithms:

Many real-world problems are *optimization* problems for which we need to find an optimal solution among many possible *candidate* solutions. A familiar scenario is the change-making problem that we often encounter at a cash register: receiving the fewest numbers of coins to make change after paying the bill for a purchase. For example, the purchase is worth \$5.27, how many coins and what coins does a cash register return after paying a \$6 bill?

## The Make-Change Algorithm:

For a given amount (e.g. \$0.73), use as many quarters (\$0.25) as possible without exceeding the amount. Use as many dimes (\$.10) for the remainder, then use as many nickels (\$.05) as possible. Finally, use the pennies (\$.01) for the rest.



**Example:** To make change for the amount  $x = 67$  (cents). Use  $q = (x/25) = 2$  quarters. The remainder is  $x - 25q = 17$ , which we use  $d = (17/10) = 1$  dime. Then the remainder is  $17 - 10d = 7$ , therefore we can use  $n = (7/5) = 1$  nickel. So, the remainder is  $7 - 5n = 2$ , which requires  $p = \lfloor 2/1 \rfloor = 2$  pennies. Total number of coins used is  $q + d + n + p = 6$ .

**Note:** The above algorithm is optimal i.e. it uses the fewest number of coins among all possible ways to make change for a given amount. (This fact can be proven formally). However, this is dependent on the denominations of the US currency system. For example, try a system that uses denominations of 1-cent, 6-cent, and 7-cent coins, and try to make change for  $x = 18$  cents. The greedy strategy uses two 7-cents and four 1-cents, for a total of 6 coins. However, the optimal solution is to use three 6-cent coins.



## A Generic Greedy Algorithm:

- (1) Initialize  $C$  to be the set of candidate solutions
  - (2) Initialize a set  $S = \emptyset$  (the set is to be the optimal solution we are constructing). (3)
- While  $C \neq \emptyset$  and  $S$  is (still) not a solution do
- (3.1) select  $x$  from set  $C$  using a greedy strategy
  - (3.2) delete  $x$  from  $C$
  - (3.3) if  $\{x\} \cup S$  is a *feasible* solution, then  
 $S = S \cup \{x\}$
  - (4) if  $S$  is a solution then  
return  $S$
  - (5) else return *failure*

In general, the greedy algorithm is efficient because it makes a sequence of (local) decisions and never backtracks. However, the solution is not always optimal.

## Coin changing

**Goal.** Given U. S. currency denominations { 1, 5, 10, 25, 100 }, devise a method to pay amount to customer using fewest coins.

**Ex.** 34¢.

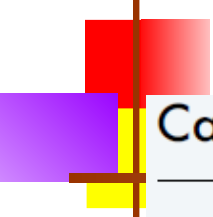


**Cashier's algorithm.** At each iteration, add coin of the largest value that does not take us past the amount to be paid.

**Ex.** \$2.89.








## Cashier's algorithm

At each iteration, add coin of the largest value that does not take us past the amount to be paid.

**CASHIERS-ALGORITHM** ( $x, c_1, c_2, \dots, c_n$ )

---

**SORT**  $n$  coin denominations so that  $0 < c_1 < c_2 < \dots < c_n$ .

$S \leftarrow \emptyset$ .  **multiset of coins selected**

**WHILE** ( $x > 0$ )

$k \leftarrow$  largest coin denomination  $c_k$  such that  $c_k \leq x$ .

**IF** (no such  $k$ )

**RETURN** “no solution.”

**ELSE**

$x \leftarrow x - c_k$ .

$S \leftarrow S \cup \{k\}$ .

**RETURN**  $S$ .

---

## Cashier's algorithm (for arbitrary coin denominations)

Q. Is cashier's algorithm optimal for any set of denominations?

A. No. Consider U.S. postage: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

- Cashier's algorithm:  $140\text{¢} = 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1$ .
- Optimal:  $140\text{¢} = 70 + 70$ .



A. No. It may not even lead to a feasible solution if  $c_1 > 1$ : 7, 8, 9.

- Cashier's algorithm:  $15\text{¢} = 9 + ?$ .
- Optimal:  $15\text{¢} = 7 + 8$ .

## Properties of any optimal solution (for U.S. coin denominations)

**Property.** Number of pennies  $\leq 4$ .

**Pf.** Replace 5 pennies with 1 nickel.

**Property.** Number of nickels  $\leq 1$ .

**Property.** Number of quarters  $\leq 3$ .

**Property.** Number of nickels + number of dimes  $\leq 2$ .

**Pf.**

- Recall:  $\leq 1$  nickel.
- Replace 3 dimes and 0 nickels with 1 quarter and 1 nickel;
- Replace 2 dimes and 1 nickel with 1 quarter.



## Optimality of cashier's algorithm (for U.S. coin denominations)

**Theorem.** Cashier's algorithm is optimal for U.S. coins  $\{ 1, 5, 10, 25, 100 \}$ .

**Pf.** [ by induction on amount to be paid  $x$  ]

- Consider optimal way to change  $c_k \leq x < c_{k+1}$  : greedy takes coin  $k$ .
- We claim that any optimal solution must take coin  $k$ .
  - if not, it needs enough coins of type  $c_1, \dots, c_{k-1}$  to add up to  $x$
  - table below indicates no optimal solution can do this
- Problem reduces to coin-changing  $x - c_k$  cents, which, by induction, is optimally solved by cashier's algorithm. ■

$k$	$c_k$	all optimal solutions must satisfy	max value of coin denominations $c_1, c_2, \dots, c_{k-1}$ in any optimal solution
1	1	$P \leq 4$	–
2	5	$N \leq 1$	4
3	10	$N + D \leq 2$	$4 + 5 = 9$
4	25	$Q \leq 3$	$20 + 4 = 24$
5	100	<i>no limit</i>	$75 + 24 = 99$



# Example: Counting Money

---

- Suppose we want to count out a certain amount of money, using the fewest possible bills and coins
- A “Greedy Algorithm” would do the following:  
**At each step, take the largest possible bill or coin that does not overshoot**
  - Example: To make \$6.39, we can choose:
    - a \$5 bill
    - a \$1 bill, to make \$6
    - a 25¢ coin, to make \$6.25
    - A 10¢ coin, to make \$6.35
    - four 1¢ coins, to make \$6.39
- For US money, the greedy algorithm always gives the optimum solution



# A Failure of the Greedy Algorithm

- In some monetary system, for example: “Rupees” come in Rs. 1, Rs. 2, Rs. 5, and Rs. 10 coins
- Using a Greedy Algorithm to count Rs. 15, we would get
  - One 10 Rs.; Five 1 Rs., for a total of 15 Rs.; requires Six coins
  - Three 5 Rs. for a total of 15 Rs.; Requires Three coins only
- A better solution would be to use one 5 Rs. piece and one 10 Rs piece for a total of Rs. 15
  - This requires Two coins only
- The Greedy Algorithm results in a solution, but not in finding an optimal solution



# Example: Text Compression

---

- Given a string of text characters  $X$ , efficiently encode  $X$  into a smaller string of characters  $Y$ 
  - Saves memory and/or bandwidth
- A good approach: **Huffman Encoding**
  - Compute frequency  $f(c)$  for each character  $c$ .
  - Encode high-frequency characters with short code words
  - No code word is a prefix for another code
  - Use an optimal encoding tree to determine code words



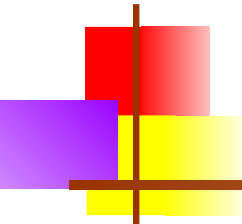


# Huffman Codes:

Suppose we wish to save a text (ASCII) file on the disk or to transmit it through a network using an encoding scheme that minimizes the number of bits required. Without *compression*, characters are typically encoded by their ASCII codes with 8 bits per character. We can do better if we have the freedom to design our own encoding.

**Example.** Given a text file that uses only 5 different letters (a, e, i, s, t), the space character, and the newline character. Since there are 7 different characters, we could use 3 bits per character because that allows 8 bit patterns ranging from 000 through 111 (so we still have one pattern to spare). The following table shows the encoding of characters, their frequencies, and the size of encoded (compressed) file.





Character	Frequency	Code	Total bits
a	10	000	30
e	15	001	45
i	12	010	
36	s	3	011
9	t	4	100
12	space	13	101
39	newline	1	110
3			
Total	58		174

Fixed-length encoding

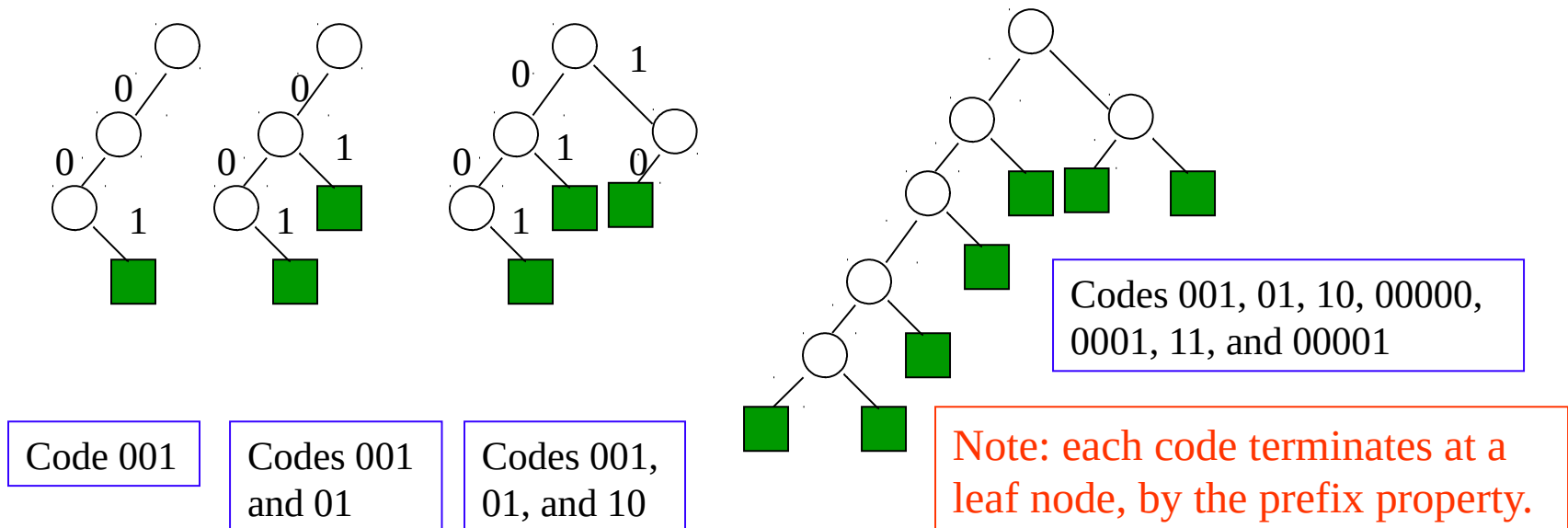
Code	Total bits
001	30
01	30
10	24
00000	15
0001	16
11	26
00001	5
	146

Variable-length encoding

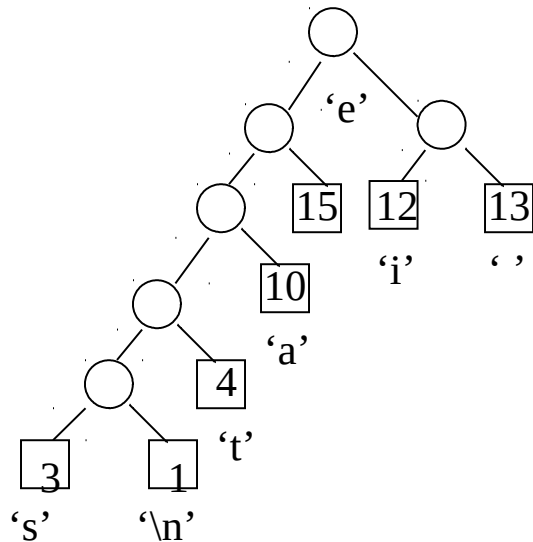
If we can use variable lengths for the codes, we can actually compress more as shown in the above. However, the codes must satisfy the property that no code is the prefix of another code; such code is called a *prefix code*.

# How to design an optimal prefix code (i.e., with minimum total length) for a given file?

We can depict the codes for the given collection of characters using a binary tree as follows: reading each code from left to right, we construct a binary tree from the root following the left branch when encountering a '0', right branch when encountering a '1'. We do this for all the codes by constructing a single combined binary tree. For example,

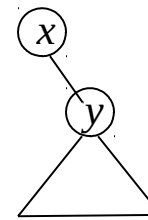


We note that the encoded file size is equal to the total weighted external path lengths if we assign the frequency to each leaf node. For example,

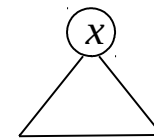


Total file size =  $3*5 + 1*5 + 4*4 + 10*3 + 15*2 + 12*2 + 13*2 = 146$ , which is exactly the total weighted external path lengths.

We also note that in an optimal prefix code, each node in the tree has either no children or has two. Thus, the optimal binary merge tree algorithm finds the optimal code (Huffman code).



Node x has only one child y

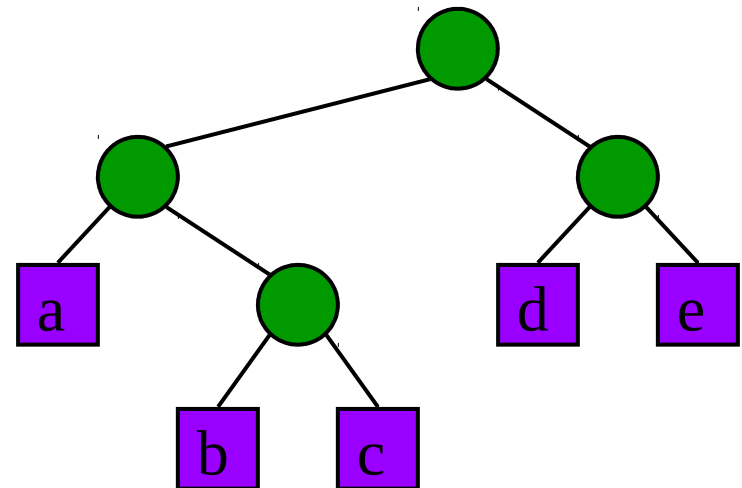


Merge x and y, reducing total size

# Encoding Tree Example

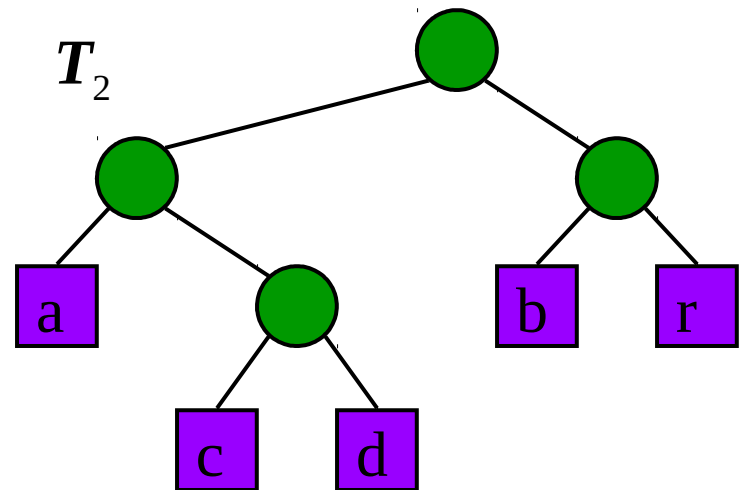
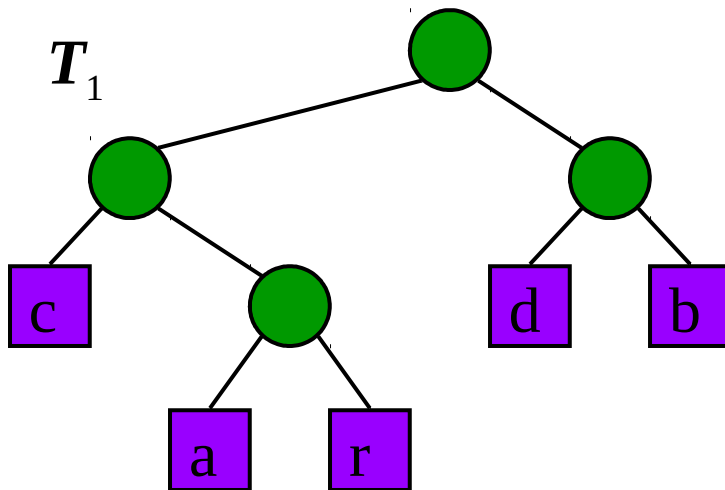
- A **code** is a mapping of each character of an alphabet to a binary code-word
- A **prefix code** is a binary code such that no code-word is the prefix of another code-word
- An **encoding tree** represents a prefix code
  - Each external node stores a character
  - The code word of a character is given by the path from the root to the external node storing the character (0 for a left child and 1 for a right child)

00	010	011	10	11
a	b	c	d	e



# Encoding Tree Optimization

- Given a text string  $X$ , we want to find a prefix code for the characters of  $X$  that yields a small encoding for  $X$ 
  - Frequent characters should have long code-words
  - Rare characters should have short code-words
- Example
  - $X = \text{abracadabra}$
  - $T_1$  encodes  $X$  into 29 bits
  - $T_2$  encodes  $X$  into 24 bits





# Huffman's Algorithm

- Given a string  $X$ , Huffman's Algorithm constructs a prefix code that minimizes the size of the encoding of  $X$
- It runs in time  $O(n + d \log d)$ , where  $n$  is the size of  $X$  and  $d$  is the number of distinct characters of  $X$
- A heap-based priority queue is used as an auxiliary structure

## Algorithm *HuffmanEncoding*( $X$ )

**Input** string  $X$  of size  $n$

**Output** optimal encoding trie for  $X$

$C \leftarrow \text{distinctCharacters}(X)$

$\text{computeFrequencies}(C, X)$

$Q \leftarrow$  new empty heap

**for all**  $c \in C$

$T \leftarrow$  new single-node tree storing  $c$

$Q.\text{insert}(\text{getFrequency}(c), T)$

**while**  $Q.\text{size}() > 1$

$f_1 \leftarrow Q.\text{minKey}()$

$T_1 \leftarrow Q.\text{removeMin}()$

$f_2 \leftarrow Q.\text{minKey}()$

$T_2 \leftarrow Q.\text{removeMin}()$

$T \leftarrow \text{join}(T_1, T_2)$

$Q.\text{insert}(f_1 + f_2, T)$

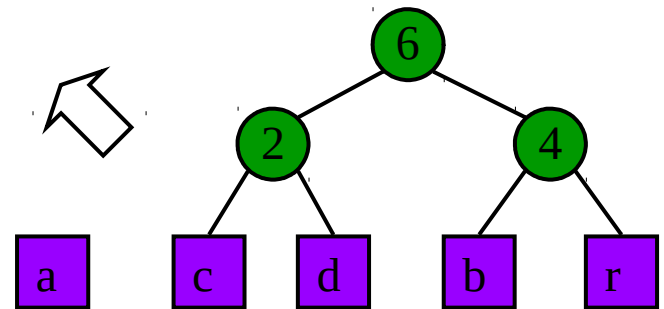
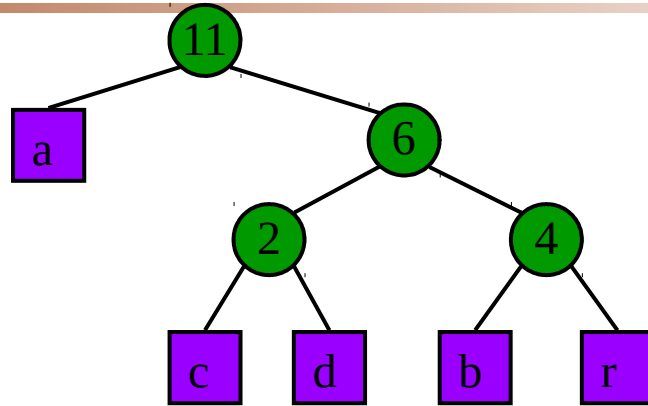
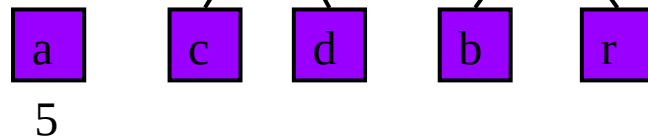
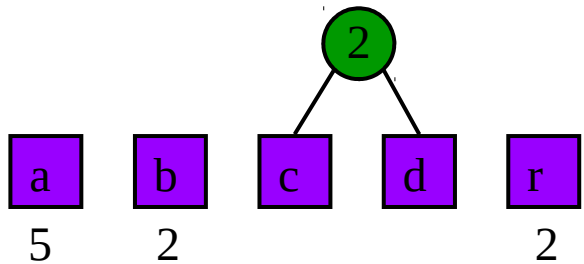
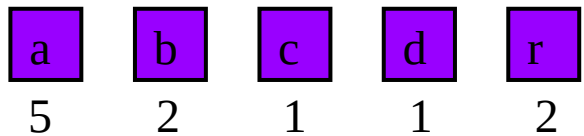
**return**  $Q.\text{removeMin}()$

# Example for Huffman Encoding

$X$  = abracadabra

Frequencies

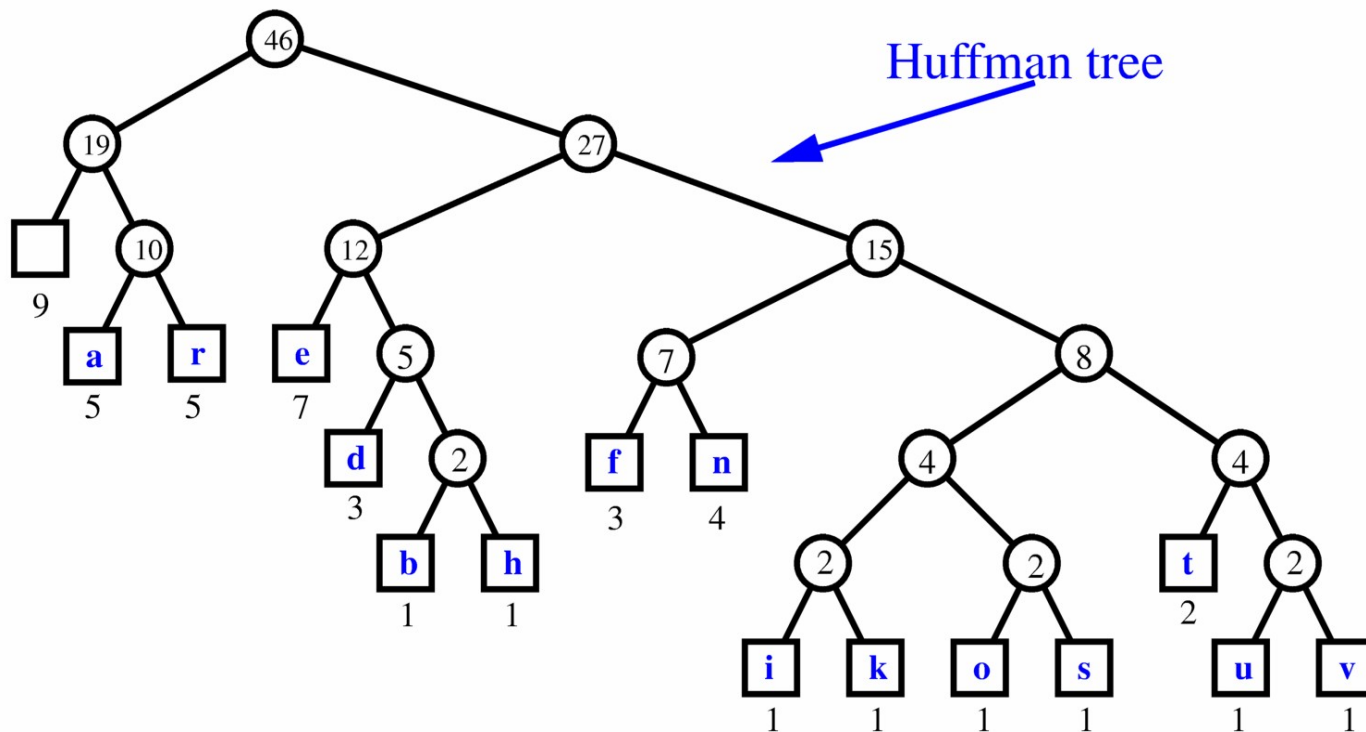
a	b	c	d	r
5	2	1	1	2



# Extended Huffman Tree Example

String: **a fast runner need never be afraid of the dark**

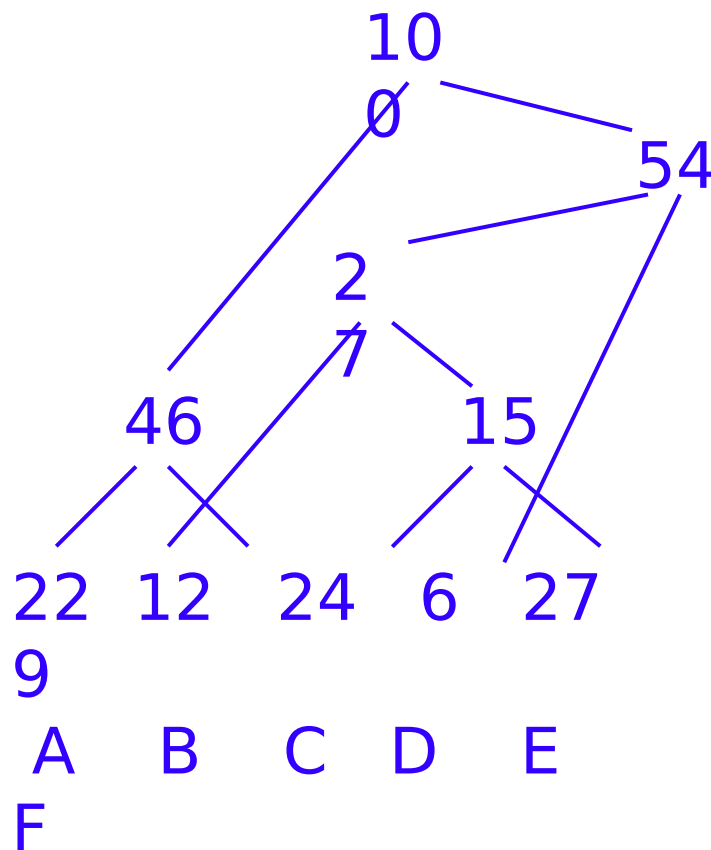
Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1





# Example for Huffman Encoding

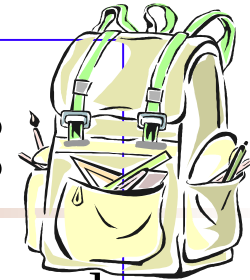
- The Huffman encoding algorithm is a greedy algorithm
- We always pick the two smallest numbers to combine



A=00  
B=100  
C=01  
D=101  
0  
E=11  
F=101  
1

- Average bits/char:  
 $0.22*2 + 0.12*3$   
+  
 $0.24*2 + 0.06*4$   
+  
 $0.27*2 + 0.09*4$   
= 2.42
- The Huffman algorithm finds an optimal solution

# The Knapsack Problem:



Given  $n$  objects each have a *Weight*  $w_i$  and a *Value*  $v_i$ , and given a knapsack of total *Capacity*  $W$ . The problem is to pack the knapsack with these objects in order to maximize the total value of those objects packed without exceeding the knapsack's capacity. More formally, let  $x_i$  denote the fraction of the object  $i$  to be included in the knapsack,  $0 \leq x_i \leq 1$ , for  $1 \leq i \leq n$ . The problem is to find the values for the  $x_i$  such that  $\sum_{i=1}^n x_i w_i \leq W$  and  $\sum_{i=1}^n x_i v_i$  is maximized.

$$\sum_{i=1}^n w_i > W$$

Note that we may assume that  $\sum_{i=1}^n w_i > W$  otherwise, we would choose  $x_i = 1$  for each  $i$  which would be considered as an obvious optimal solution.

There seem to be 3 obvious greedy strategies:

(Max value) Sort the objects from the highest value to the lowest, then pick them in that order.

(Min weight) Sort the objects from the lowest weight to the highest, then pick them in that order.

(Max value/weight ratio) Sort the objects based on the value to weight ratios, from the highest to the lowest, then select.

**Example:** Given  $n = 5$  objects and a knapsack capacity  $W = 100$  as shown in Table I. The three solutions are given in Table II.

$v$	20	30	66	40	60
$w$	10	20	30	40	50
$v/w$	2.0	1.5	2.2	1.0	1.2

Table I

select	$x_i$	value	$(\sum x_i v_i)$
Max $v_i$	0 0 1 0.5 1		146
Min $w_i$	1 1 1 1 0		156
Max $v_i/w_i$	1 1 1 0 0.8		164

Table II



# The Optimal Knapsack Algorithm:

**Input:** An integer  $n$ , positive values  $w_i$  and  $v_i$ , for  $1 \leq i \leq n$ , and another positive value  $W$ .

**Output:**  $n$  values  $x_i$  such that  $0 \leq x_i \leq 1$  and

$$\sum_{i=1}^n x_i w_i \leq W \text{ and } \sum_{i=1}^n x_i v_i \text{ is maximized.}$$

**Algorithm** (of time complexity  $O(n \log n)$ )

(1) Sort ' $n$ ' objects from large to small based on the ratios  $v_i/w_i$ . Assume the arrays  $w[1..n]$  and  $v[1..n]$ . Store the weights and values after sorting.

(2) initialize array  $x[1..n]$  to zeros.

(3) weight = 0;  $i = 1$

(4) while ( $i \leq n$  and weight  $< W$ ) do

(4.1) if weight +  $w[i] \leq W$  then  $x[i] = 1$

(4.2) else  $x[i] = (W - \text{weight}) / w[i]$

(4.3) weight = weight +  $x[i] * w[i]$

(4.4)  $i++$

# Fractional Knapsack Problem



- Given: A set  $S$  of  $n$  items, with each item  $i$  having
  - $b_i$  - a positive benefit
  - $w_i$  - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most  $W$ .
- If we are allowed to take fractional amounts (broken items), then this is known as the **Fractional Knapsack Problem**.
  - In this case, we let  $x_i$  denote the amount we take of item  $i$






- Objective: maximize 
$$\sum_{i \in S} b_i (x_i / w_i)$$

- Constraint: 
$$\sum_{i \in S} x_i \leq W$$

# Example



- Given: A set  $S$  of  $n$  items, with each item  $i$  having
  - $b_i$  - a positive benefit
  - $w_i$  - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most  $W$ .

Items:					
Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit:	\$12	\$32	\$40	\$30	\$50
Value: (\$ per ml)	3	4	20	5	50



10 ml

“Knapsack”  
Solution:

- 1 ml of 5
- 2 ml of 3
- 6 ml of 4
- 1 ml of 2

# Fractional Knapsack Algorithm



- Greedy choice: Keep taking item with highest **value** (benefit to weight ratio)
  - Since  $\sum_{i \in S} b_i (x_i / w_i) = \sum_{i \in S} (b_i / w_i) x_i$
  - Run time:  $O(n \log n)$ . Why?
- Correctness: Suppose there is a better solution
  - there is an item  $i$  with higher value than a chosen item  $j$ , but  $x_i < w_i$ ,  $x_j > 0$  and  $v_i < v_j$
  - If we substitute some  $i$  with  $j$ , we get a better solution
  - How much of  $i$ :  $\min\{w_i - x_i, x_j\}$
  - Thus, there is no better solution than the greedy one

**Algorithm** *fractionalKnapsack*( $S, W$ )

**Input:** set  $S$  of items w/ benefit  $b_i$  and weight  $w_i$ ; max. weight

$W$  **Output:** amount  $x_i$  of each item  $i$  to maximize benefit w/ weight at most  $W$

**for** *each item*  $i$  **in**  $S$

$x_i \leftarrow 0$

$v_i \leftarrow b_i / w_i$  {value}

$w \leftarrow 0$  {total weight}

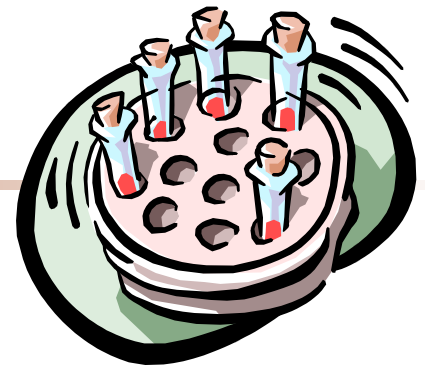
**while**  $w < W$

*remove item*  $i$  *w/ highest*  $v_i$

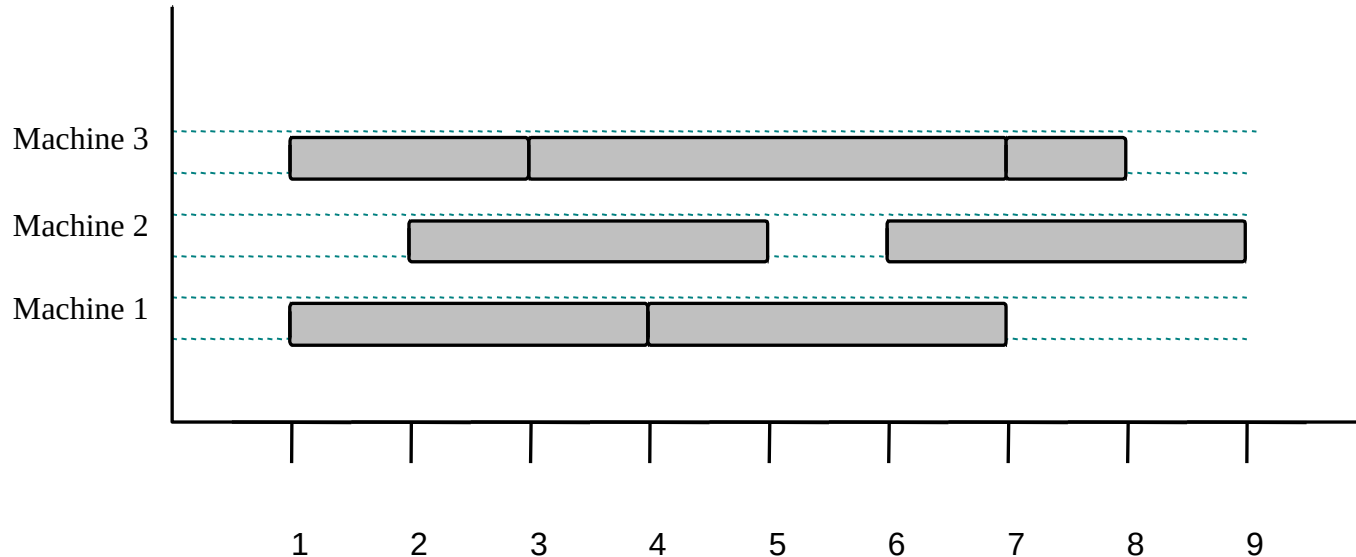
$x_i \leftarrow \min\{w_i, W - w\}$

$w \leftarrow w + \min\{w_i, W - w\}$

# Task Scheduling



- Given: a set  $T$  of  $n$  tasks, each having:
  - A start time,  $s_i$
  - A finish time,  $f_i$  (where  $s_i < f_i$ )
- Goal: Perform all the tasks using a minimum number of “machines.”





# Task Scheduling Algorithm



- Greedy choice: consider tasks by their start time and use as few machines as possible with this order.
  - Run time:  $O(n \log n)$ . Why?
- Correctness: Suppose there is a better schedule.
  - We can use  $k-1$  machines
  - The algorithm uses  $k$
  - Let  $i$  be first task scheduled on machine  $k$
  - Machine  $i$  must conflict with  $k-1$  other tasks
  - But that means there is no non-conflicting schedule using  $k-1$  machines

## Algorithm *taskSchedule*( $T$ )

**Input:** set  $T$  of tasks w/ start time  $s_i$   
and finish time  $f_i$

**Output:** non-conflicting schedule  
with minimum number of  
machines

$m \leftarrow 0$  {no. of  
machines}

**while**  $T$  is not empty

*remove task  $i$  w/ smallest  $s_i$*

**if** *there's a machine  $j$  for  $i$*  **then**  
*schedule  $i$  on machine  $j$*

**else**

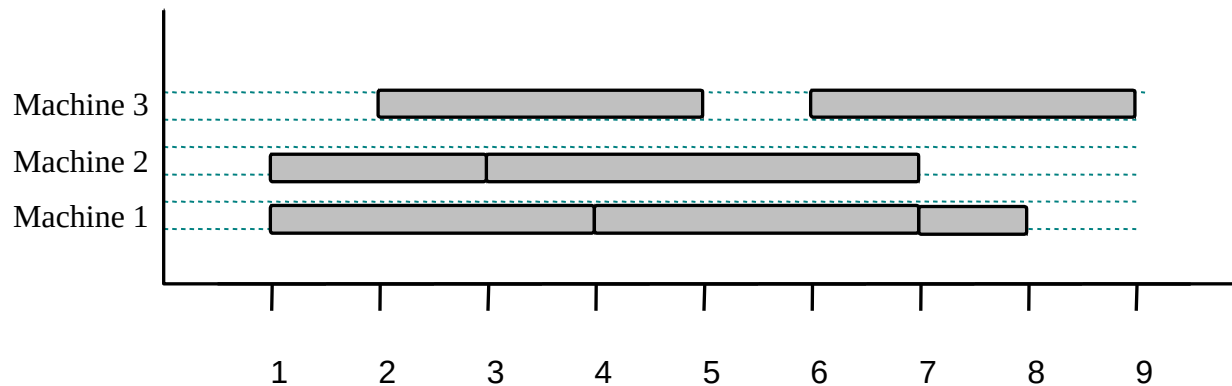
$m \leftarrow m + 1$

*schedule  $i$  on machine  $m$*

# Example for Task Scheduling

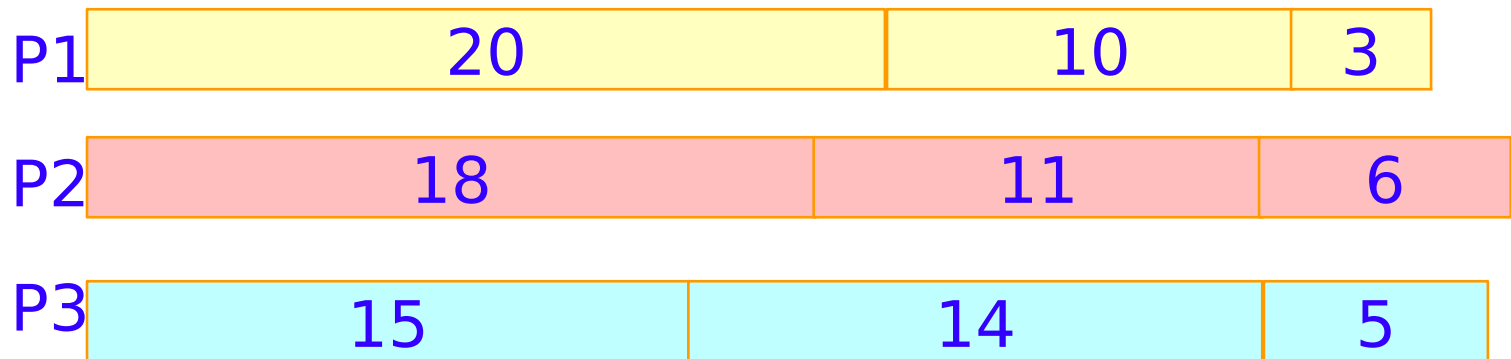


- Given: a set  $T$  of  $n$  tasks, each having:
  - A start time,  $s_i$
  - A finish time,  $f_i$  (where  $s_i < f_i$ )
  - $[1,4], [1,3], [2,5], [3,7], [4,7], [6,9], [7,8]$  (ordered by start)
- Goal: Perform all tasks on min. number of machines



# Job Scheduling Problem

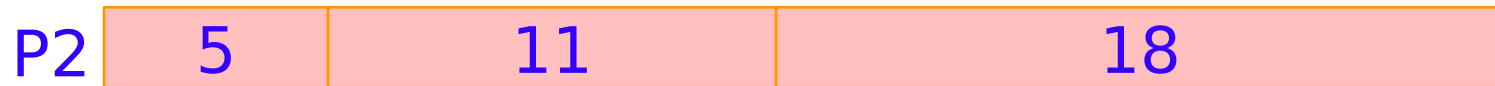
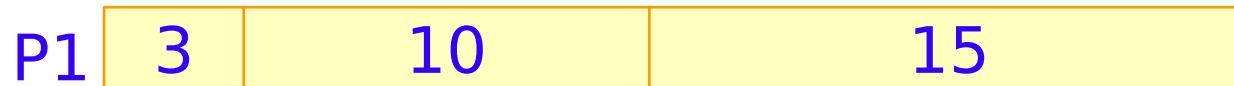
- We have to run nine jobs, with running times of 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes
- We have three processors on which we can run these jobs
- We decide to do the longest-running jobs first, on whatever processor is available



- Time for completion:  $18 + 11 + 6 = 35$  minutes
- This solution isn't that bad, but we might be able to do better

# Another Approach for Job Scheduling

- What would be the result if we ran the *shortest* job first?
- Again, the running times are 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes

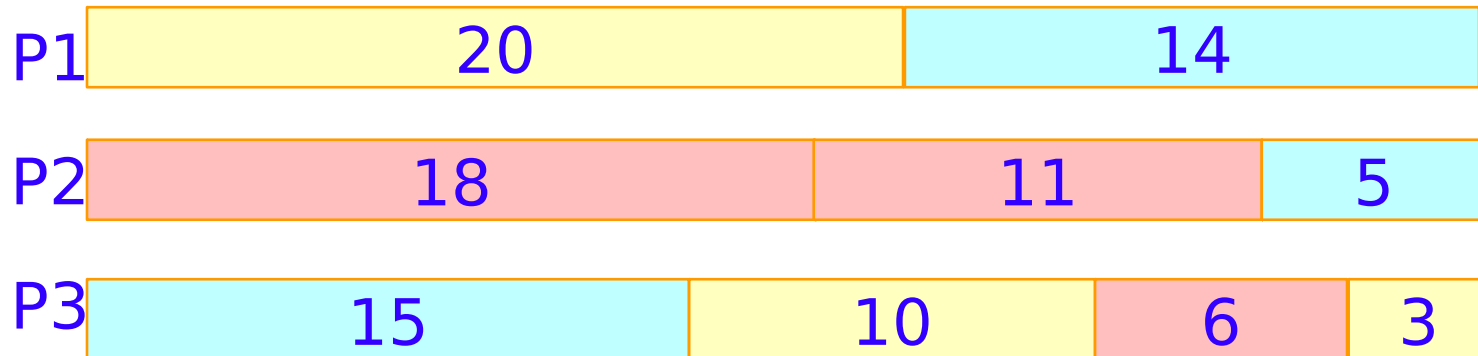


- That wasn't such a good idea; time to completion is now  $6 + 14 + 20 = 40$  minutes
- Note, however, that the greedy algorithm itself is fast
  - All we had to do at each stage was pick the minimum or maximum



# An Optimum Solution

- Better Solutions do exist:

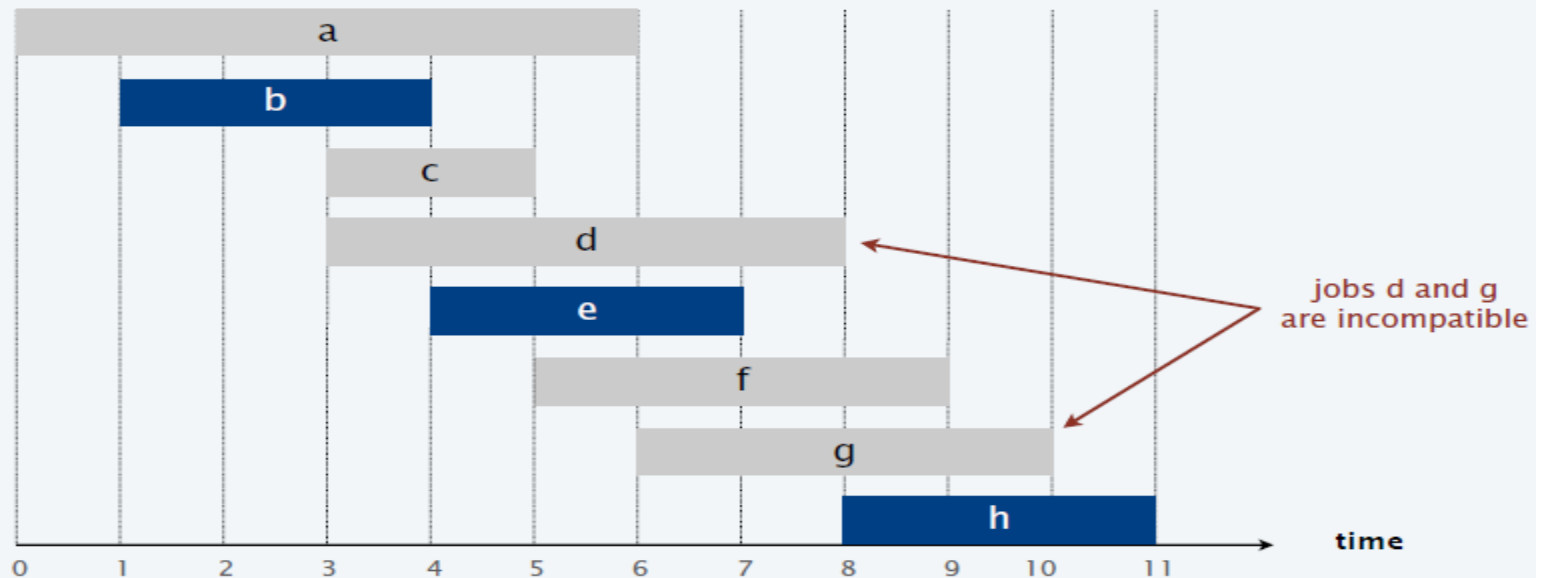


- This solution is clearly optimal (why?)
- Clearly, there are other optimal solutions (why?)
- How do we find such a solution?
  - One way: Try all possible assignments of jobs to processors
  - Unfortunately, this approach can take exponential time

# Interval Scheduling

## Interval scheduling

- Job  $j$  starts at  $s_j$  and finishes at  $f_j$ .
- Two jobs are **compatible** if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.





# Interval Scheduling

## Interval scheduling: earliest-finish-time-first algorithm

**EARLIEST-FINISH-TIME-FIRST** ( $n, s_1, s_2, \dots, s_n, f_1, f_2, \dots, f_n$ )

**SORT** jobs by finish times and renumber so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .

$S \leftarrow \emptyset$ .  $\leftarrow$  set of jobs selected

**FOR**  $j = 1$  **TO**  $n$

**IF** (job  $j$  is compatible with  $S$ )

$S \leftarrow S \cup \{ j \}$ .

**RETURN**  $S$ .

**Proposition.** Can implement earliest-finish-time first in  $O(n \log n)$  time.

- Keep track of job  $j^*$  that was added last to  $S$ .
- Job  $j$  is compatible with  $S$  iff  $s_j \geq f_{j^*}$ .
- Sorting by finish times takes  $O(n \log n)$  time.

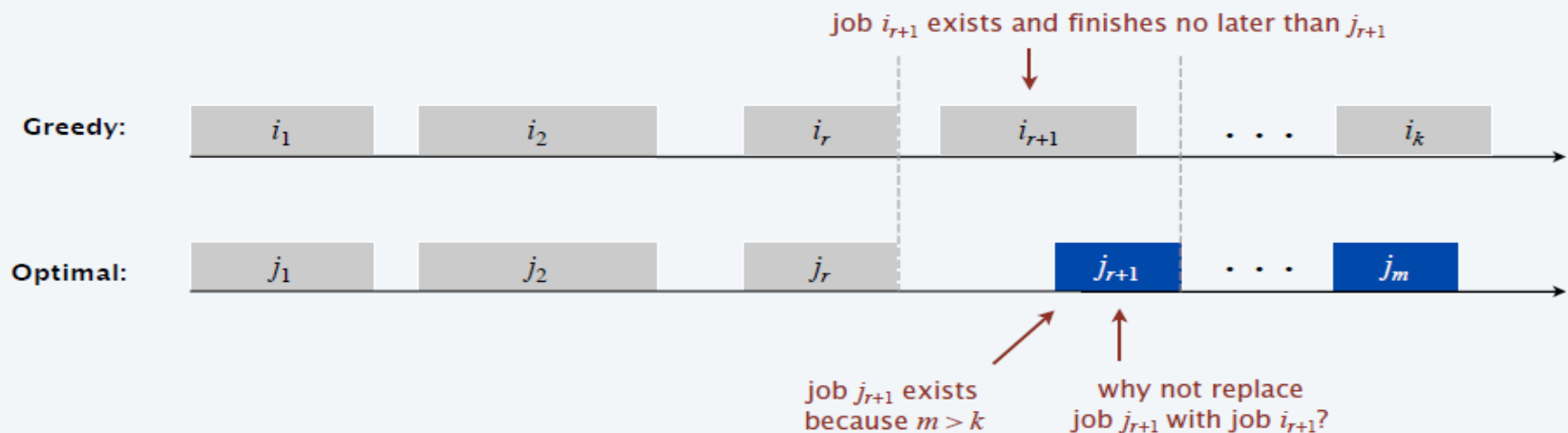
# Interval Scheduling

## Interval scheduling: analysis of earliest-finish-time-first algorithm

**Theorem.** The earliest-finish-time-first algorithm is optimal.

**Pf.** [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let  $i_1, i_2, \dots, i_k$  denote set of jobs selected by greedy.
- Let  $j_1, j_2, \dots, j_m$  denote set of jobs in an optimal solution with  $i_1 = j_1, i_2 = j_2, \dots, i_r = j_r$  for the largest possible value of  $r$ .





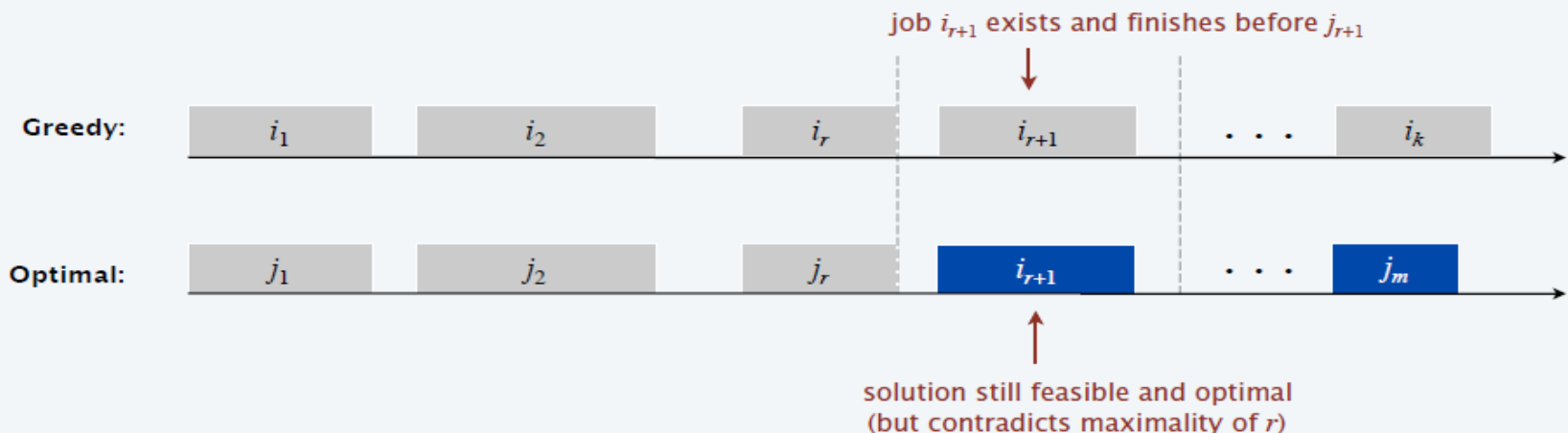
# Interval Scheduling

## Interval scheduling: analysis of earliest-finish-time-first algorithm

**Theorem.** The earliest-finish-time-first algorithm is optimal.

**Pf.** [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let  $i_1, i_2, \dots, i_k$  denote set of jobs selected by greedy.
- Let  $j_1, j_2, \dots, j_m$  denote set of jobs in an optimal solution with  $i_1 = j_1, i_2 = j_2, \dots, i_r = j_r$  for the largest possible value of  $r$ .

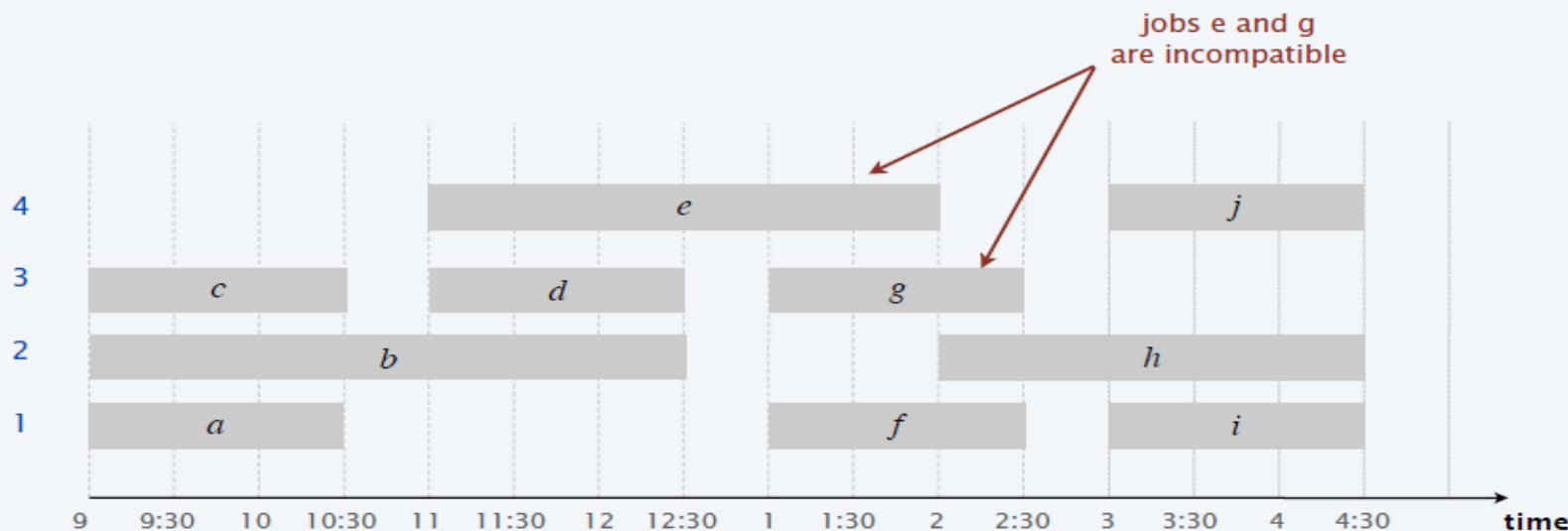


# Interval Partitioning

## Interval partitioning

- Lecture  $j$  starts at  $s_j$  and finishes at  $f_j$ .
- Goal: find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room.

Ex. This schedule uses 4 classrooms to schedule 10 lectures.

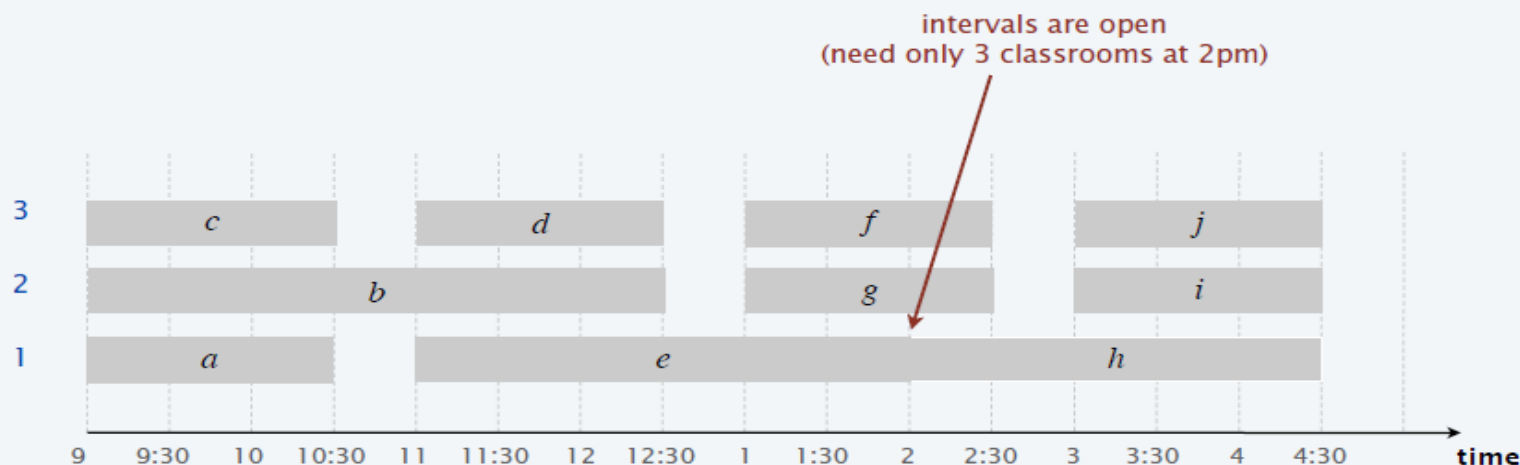


# Interval Partitioning

## Interval partitioning

- Lecture  $j$  starts at  $s_j$  and finishes at  $f_j$ .
- Goal: find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room.

Ex. This schedule uses 3 classrooms to schedule 10 lectures.





# Interval Partitioning

## Interval partitioning: earliest-start-time-first algorithm

---

EARLIEST-START-TIME-FIRST ( $n, s_1, s_2, \dots, s_n, f_1, f_2, \dots, f_n$ )

---

SORT lectures by start times and renumber so that  $s_1 \leq s_2 \leq \dots \leq s_n$ .

$d \leftarrow 0$ .  $\leftarrow$  number of allocated classrooms

FOR  $j = 1$  TO  $n$

IF (lecture  $j$  is compatible with some classroom)

Schedule lecture  $j$  in any such classroom  $k$ .

ELSE

Allocate a new classroom  $d + 1$ .

Schedule lecture  $j$  in classroom  $d + 1$ .

$d \leftarrow d + 1$ .

RETURN schedule.

---



# Interval Partitioning

## Interval partitioning: earliest-start-time-first algorithm

**Proposition.** The earliest-start-time-first algorithm can be implemented in  $O(n \log n)$  time.

**Pf.**

- Sorting by start times takes  $O(n \log n)$  time.
- Store classrooms in a **priority queue** (key = finish time of its last lecture).
  - to allocate a new classroom, INSERT classroom onto priority queue.
  - to schedule lecture  $j$  in classroom  $k$ , INCREASE-KEY of classroom  $k$  to  $f_j$ .
  - to determine whether lecture  $j$  is compatible with any classroom, compare  $s_j$  to FIND-MIN
- Total # of priority queue operations is  $O(n)$ ; each takes  $O(\log n)$  time. ▀

**Remark.** This implementation chooses a classroom  $k$  whose finish time of its last lecture is the **earliest**.

# Interval Partitioning

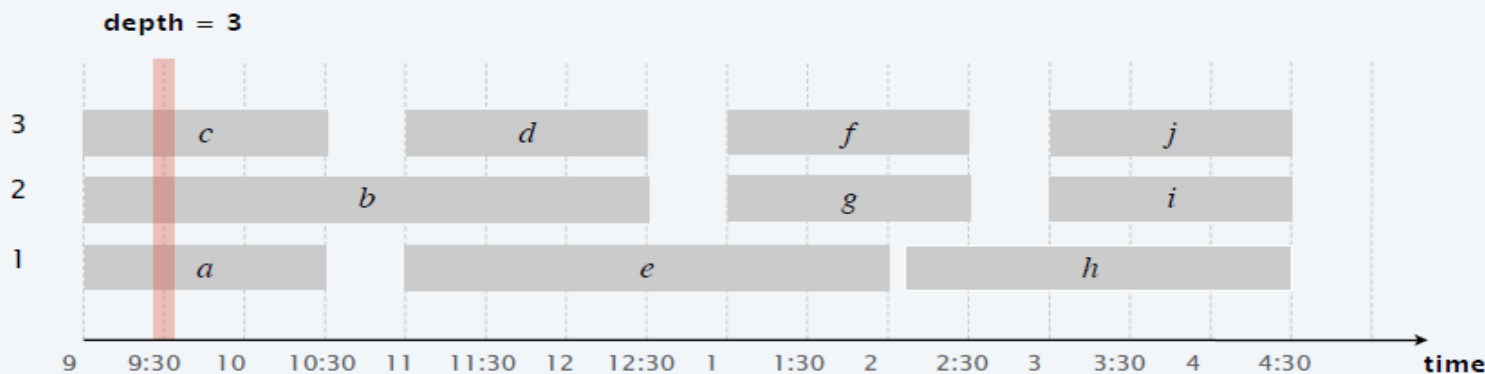
Interval partitioning: lower bound on optimal solution

**Def.** The **depth** of a set of open intervals is the maximum number of intervals that contain any given point.

**Key observation.** Number of classrooms needed  $\geq$  depth.

**Q.** Does minimum number of classrooms needed always equal depth?

**A.** Yes! Moreover, earliest-start-time-first algorithm finds a schedule whose number of classrooms equals the depth.





# Interval Partitioning

## Interval partitioning: analysis of earliest-start-time-first algorithm

---

**Observation.** The earliest-start-time first algorithm never schedules two incompatible lectures in the same classroom.

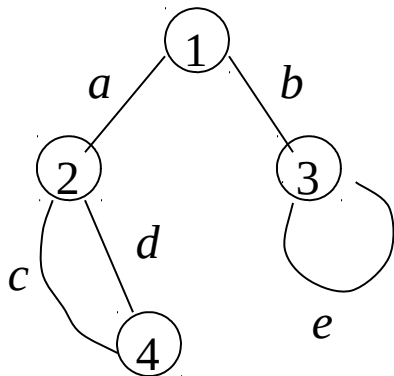
**Theorem.** Earliest-start-time-first algorithm is optimal.

**Pf.**

- Let  $d$  = number of classrooms that the algorithm allocates.
- Classroom  $d$  is opened because we needed to schedule a lecture, say  $j$ , that is incompatible with a lecture in each of  $d - 1$  other classrooms.
- Thus, these  $d$  lectures each end after  $s_j$ .
- Since we sorted by start time, each of these incompatible lectures start no later than  $s_j$ .
- Thus, we have  $d$  lectures overlapping at time  $s_j + \epsilon$ .
- Key observation  $\Rightarrow$  all schedules use  $\geq d$  classrooms. ■

# Greedy Strategies Applied to Graph Problems:

We first review some notations and terms about graphs. A graph consists of vertices (nodes) and edges (arcs, links), in which each edge “connects” two vertices (not necessarily distinct). More formally, a graph  $G = (V, E)$ , where  $V$  and  $E$  denote the sets of vertices and edges, respectively.

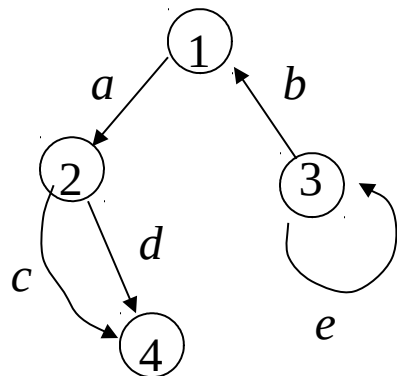


In this example,  $V = \{1, 2, 3, 4\}$ ,  $E = \{a, b, c, d, e\}$ . Edges  $c$  and  $d$  are parallel edges; edge  $e$  is a self-loop. A path is a sequence of “adjacent” edges, e.g., path *abeb*, path *acdab*.



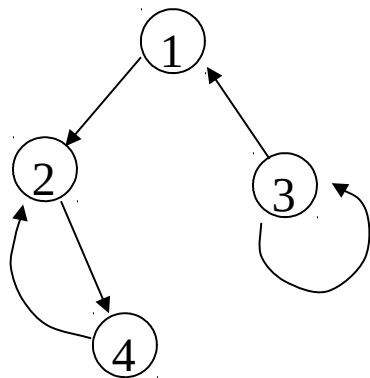
# Directed Graphs vs. Un-directed Graphs:

If every edge has an orientation, e.g., an edge starting from node  $x$  terminating at node  $y$ , the graph is called a directed graph, or digraph for short. If all edges have no orientation, the graph is called an undirected graph, or simply, a graph. When there are no parallel edges (two edges that have identical end points), we could identify an edge with its two end points, such as edge  $(1,2)$ , or edge  $(3,3)$ . In an undirected graph, edge  $(1,2)$  is the same as edge  $(2,1)$ . We will assume no parallel edges unless otherwise stated.



A directed graph. Edges  $c$  and  $d$  are parallel (directed) edges. Some directed paths are  $ad$ ,  $ebac$ .

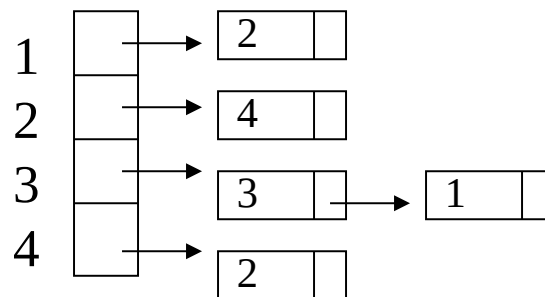
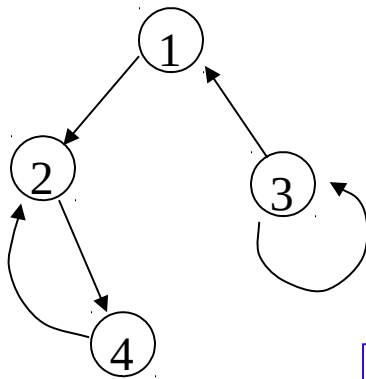
Both directed and undirected graphs appear often and naturally in many scientific (call graphs in program analysis), business (query trees, entity-relation diagrams in databases), and engineering (CAD design) applications. The simplest data structure for representing graphs and digraphs is using 2-dimensional arrays. Suppose  $G = (V, E)$ , and  $|V| = n$ . Declare an array  $T[1..n][1..n]$  so that  $T[i][j] = 1$  if there is an edge  $(i, j) \in E$ ; 0 otherwise. (Note that in an undirected graph, edges  $(i, j)$  and  $(j, i)$  refer to the same edge.)



		<i>j</i>			
		1	2	3	4
<i>i</i>	1	0	1	0	0
	2	0	0	0	1
	3	1	0	1	0
	4	0	1	0	0

A 2-dimensional array for the digraph, called the *adjacency matrix*.

Sometimes, edges of a graph or digraph are given a positive *weight* or *cost* value. In that case, the adjacency matrix can easily be modified so that  $T[i][j]$  = the weight of edge  $(i, j)$ ; 0 if there is no edge  $(i, j)$ . Since the adjacency matrix may contain many zeros (when the graph has few edges, known as *sparse*), a space-efficient representation uses linked lists representing the edges, known as the *adjacency list* representation.



The adjacency lists for the digraph, which can store edge weights by adding another field in the list nodes.



# Graph (and Digraph) Traversal

## Techniques:

Given a (directed) graph  $G = (V, E)$ , determine all nodes that are connected from a given node  $v$  via a (directed) path.

There are essentially two graph traversal algorithms, known as *Breadth-first search* (BFS) and *depth-first search* (DFS), both of which can be implemented efficiently.

**BFS:** From node  $v$ , visit each of its neighboring nodes in sequence, then visit their neighbors, etc., while avoiding repeated visits.

**DFS:** From node  $v$ , visit its first neighboring node and all its neighbors using recursion, then visit node  $v$ 's second neighbor applying the same procedure, until all  $v$ 's neighbors are visited, while avoiding repeated visits.

# Breadth-First Search (BFS):

BFS( $v$ ) // visit all nodes reachable from node  $v$

(1) Create an empty FIFO queue  $Q$ , add node  $v$  to  $Q$

(2) Create a Boolean array  $\text{visited}[1..n]$ , initialize all values to false except for  $\text{visited}[v]$  to true (3)

while  $Q$  is not empty

(3.1) delete a

node  $w$  from  $Q$

(3.2) for each node  $z$

adjacent from node  $w$

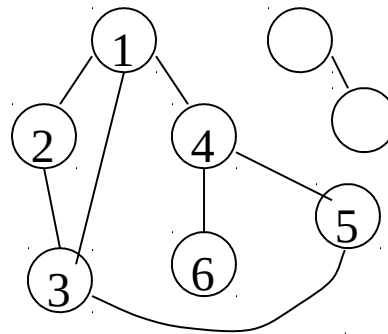
if  $\text{visited}[z]$  is false

then

add node  $z$  to  $Q$  and set  $\text{visited}[z]$

to true

The time complexity is  $O(n+e)$  with  $n$  nodes and  $e$  edges, if the adjacency lists are used. This is because in the worst case, each node is added once to the queue ( $O(n)$  part), and each of its neighbors gets considered once ( $O(e)$  part).



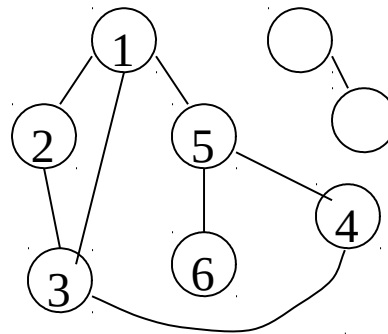
Node search order starting with node 1, including two nodes not reached

# Depth-First Search (DFS):

(1) Create a Boolean array `visited[1..n]`, initialize all values to false except for `visited[v]` to true  
(2) Call `DFS(v)` to visit all nodes reachable via a path

`DFS(v)`  
for each neighboring node `w` of `v` do  
    if `visited[w]` is false then  
        set `visited[w]` to true; call `DFS(w)` // recursive call

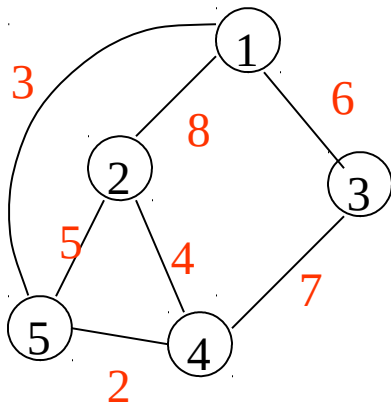
The algorithm's time complexity is also  $O(n+e)$  using the same reasoning as in the BFS algorithm.



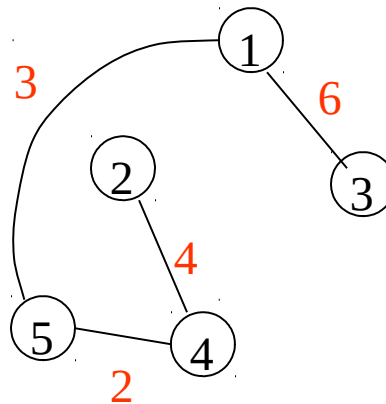
Node search order starting with node 1, including two nodes not reached

# The Minimum Spanning Tree (MST) Problem:

Given a weighted (undirected) graph  $G = (V, E)$ , where each edge  $e$  has a positive weight  $w(e)$ . A *spanning tree* of  $G$  is a *tree* (connected graph without cycles, or circuits) which has  $V$  as its vertex set, i.e., the tree connects all vertices of the graph  $G$ . If  $|V| = n$ , then the tree has  $n - 1$  edges (this is a fact which can be proved by induction). A *minimum spanning tree* of  $G$  is a spanning tree that has the minimum total edge weight.



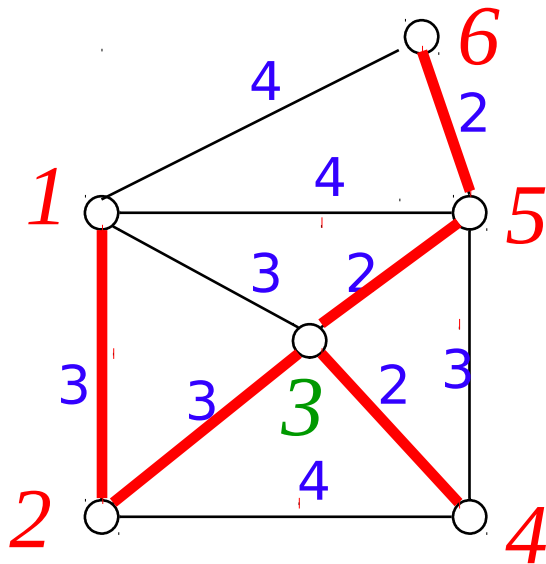
A weighted graph of no parallel edges or self-loops



A minimum spanning tree (of 4 edges), weight =  $3 + 2 + 4 + 6 = 15$ .

# Minimum Spanning Tree (MST)

- A minimum spanning tree is a least-cost subset of the edges of a graph that connects all the nodes
  - Start by picking any node and adding it to the tree
  - Repeatedly: Pick any *least-cost* edge from a node in the tree to a node not in the tree, and add the edge and new node to the tree
  - Stop when all nodes have been added to the tree

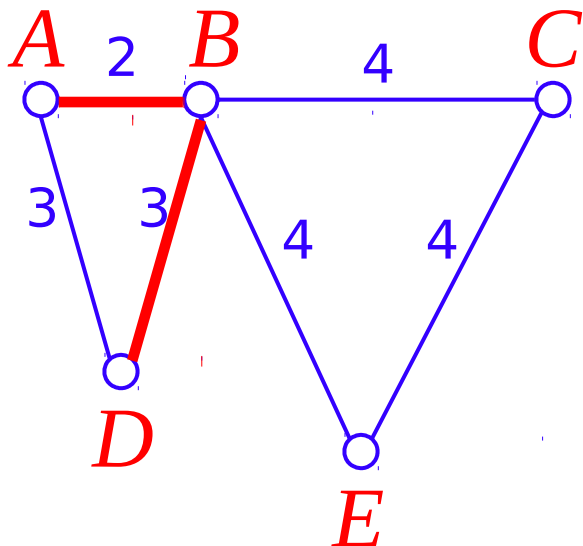


- The result is a least-cost ( $3+3+2+2+2=12$ ) spanning tree
- If we think some other edge should be in the spanning tree:
  - Try adding that edge
  - Note that the edge is part of a cycle
  - To break the cycle, we must remove the edge with the greatest cost
    - This will be the edge we just



# Traveling Salesperson Problem (TSP)

- A salesperson must visit every city (starting from city **A**), and wants to cover the least possible distance
  - He(she) can revisit a city (and reuse a road) if necessary
- He(she) does this by using a greedy algorithm: He(she) goes to the next nearest city from wherever he(she) is



- From **A** he(she) goes to **B**
- From **B** he(she) goes to **D**
- This is *not* going to result in a shortest path!
- The best result he(she) can get now will be **ABDBCE**, at a cost of **16**
- An actual least-cost path from **A** is **ADBCE**, at a cost of **14**



# Time Complexity Analysis

- A Greedy Algorithm typically makes (approximately)  $n$  choices for a problem of size  $n$ 
  - (The first or last choice may be forced)
- Hence the expected running time is:  $O(n * O(\text{choice}(n)))$ , where  $\text{choice}(n)$  is making a choice among  $n$  objects
  - Counting: Must find largest useable coin from among  $k$  sizes of coin ( $k$  is a constant), an  $O(k)=O(1)$  operation;
    - Therefore, coin counting is  $(n)$
  - Huffman: Must sort  $n$  values before making  $n$  choices
    - Therefore, Huffman is  $O(n \log n) + O(n) = O(n \log n)$
  - Minimum spanning tree: At each new node, must include new edges and keep them sorted, which is  $O(n \log n)$  overall
    - Therefore, MST is  $O(n \log n) + O(n) = O(n \log n)$

# Greedy Analysis Strategies

## Greedy analysis strategies

**Greedy algorithm stays ahead.** Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.

**Structural.** Discover a simple “structural” bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

**Exchange argument.** Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

**Other greedy algorithms.** Gale–Shapley, Kruskal, Prim, Dijkstra, Huffman, ...



GREED IS GOOD



# Other Greedy Algorithms

---

- Dijkstra's Algo for finding the shortest path in a graph
  - Always takes the *shortest* edge (path) connecting a known node to an unknown node
- Kruskal's Algo for finding a minimum-cost spanning tree
  - Always tries the *lowest-cost* remaining edge
- Prim's Algo for finding a minimum-cost spanning tree
  - Always takes the *lowest-cost* edge between nodes in the spanning tree and nodes not yet in the spanning tree



# Dijkstra's Shortest-path Algorithm

- Dijkstra's algorithm finds the shortest paths from a given node to all other nodes in a graph
  - Initially,
    - Mark the given node as *known* (path length is zero)
    - For each out-edge, set the distance in each neighboring node equal to the *cost* (length) of the out-edge, and set its *predecessor* to the initially given node
  - Repeatedly (until all nodes are known),
    - Find an unknown node containing the smallest distance
    - Mark the new node as known
    - For each node adjacent to the new node, examine its neighbors to see whether the estimated distance can be reduced (distance to known node + cost of out-edge)
      - If so, also reset the predecessor of the new node



# Analysis of Dijkstra's Algorithm I

- Assume that the *average* out-degree of a node is some constant  $k$ 
  - Initially,
    - Mark the given node as *known* (path length is zero)
      - This takes  $O(1)$  (constant) time
    - For each out-edge, set the distance in each neighboring node equal to the *cost* (length) of the out-edge, and set its *predecessor* to the initially given node
      - If each node refers to a list of  $k$  adjacent node/edge pairs, this takes  $O(k) = O(1)$  time, that is, constant time
      - Notice that this operation takes *longer* if we have to extract a list of names from a hash table



# Analysis of Dijkstra's Algorithm II

- Repeatedly (until all nodes are known), ( $n$  times)
  - Find an unknown node containing the smallest distance
    - Probably the best way to do this is to put the unknown nodes into a priority queue; this takes  $k * O(\log n)$  time *each* time a new node is marked “known” (and this happens  $n$  times)
  - Mark the new node as known --  $O(1)$  time
  - For each node adjacent to the new node, examine its neighbors to see whether their estimated distance can be reduced (distance to known node plus cost of out-edge)
    - If so, also reset the predecessor of the new node
    - There are  $k$  adjacent nodes (on average), operation requires constant time at each, therefore  $O(k)$  (constant) time
  - Combining all the parts, we get:  
 $O(1) + n*(k*O(\log n)+O(k))$ , that is,  $O(nk \log n)$  time