October 25, 2017

Greedy

Chapters 5 of Dasgupta *et al.*

UNIVERSITY OF CALIFORNIA
UCRIVERSIDE

1

Outline

- Activity selection
- Fractional knapsack
- Huffman encoding
- Later:
  - Dijkstra (single source shortest path)
  - Prim and Kruskal (minimum spanning tree)

2

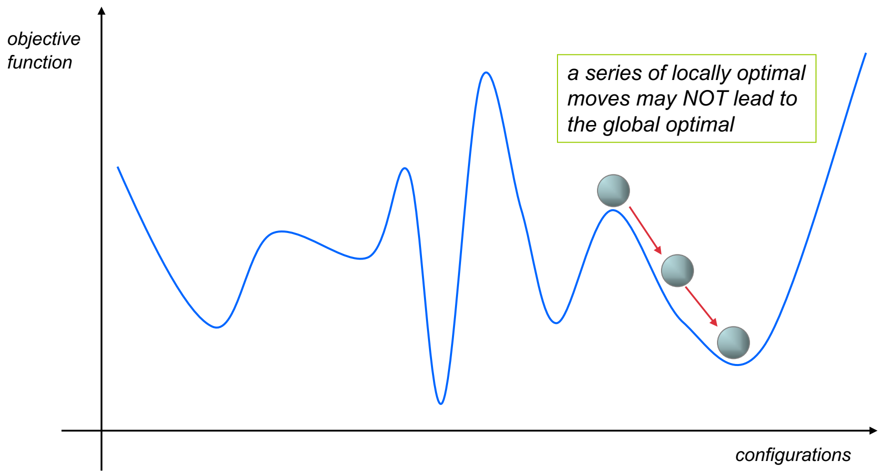
Optimization problems

- A class of problems in which we are asked to find a set (or a sequence) of "items" that satisfy some constraints and simultaneously optimize (i.e., maximize or minimize) some objective function

Greedy method

- Typically applied to *optimization problems*, that is, problems that involve searching through a set of *configurations* to find one that minimizes/maximizes an *objective function* defined on these configuration
- *Greedy strategy:* at each step of the optimization procedure, choose the configuration which seems the best between all of those possible

Searching for the global minimum



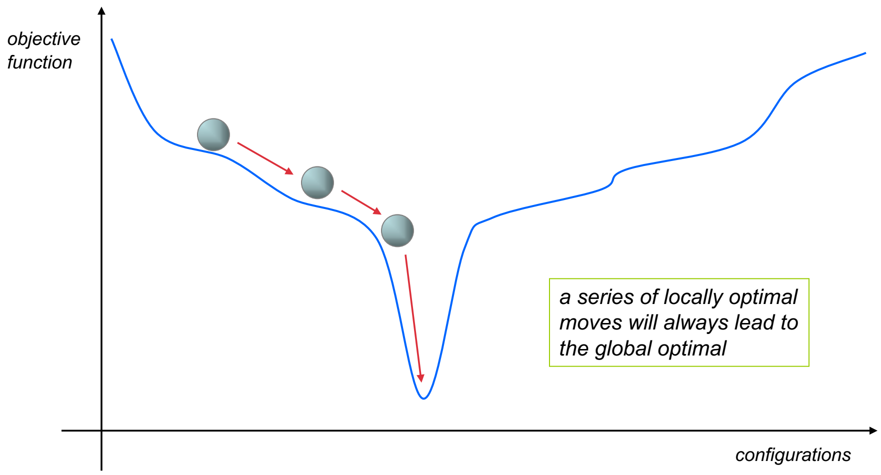
5

Greedy method

- There are problem for which the globally optimal solution can be found by making a series of locally optimal (greedy) choices
  – Make whatever choice seems best at the moment and then solve the sub-problem 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
- The greedy strategy does not always lead to the global optimal solution

6

Searching for the global minimum



7

Elements of greedy strategy

- Two ingredients that are exhibited by most problems that lend themselves to a greedy strategy
  - Greedy-choice property: a globally optimal solution can be reached by making a locally optimal choice
  - Optimal substructure: optimal solution to the problem results from optimal solutions to sub-problems

8

Activity selection

(aka, “task scheduling”)

9

Activity Selection

- Input: A set of activities $S = \{a_1, \ldots, a_n\}$
- Each activity has start time and a finish time $a_i = (s_i, f_i)$
- Two activities are *conflicting* if and only if their interval overlap
- Output: a maximum-size subset of non-conflicting activities

10

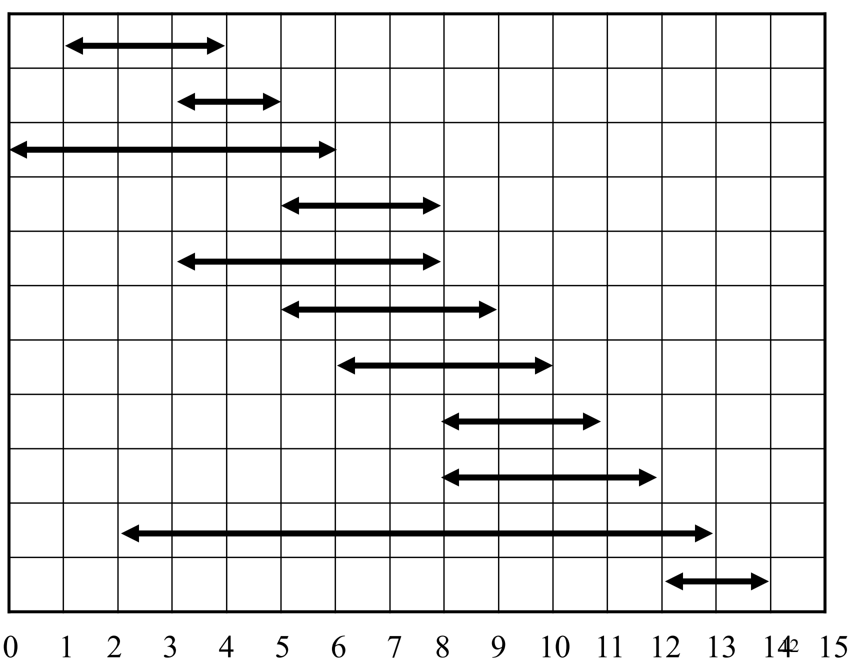
Activity Selection

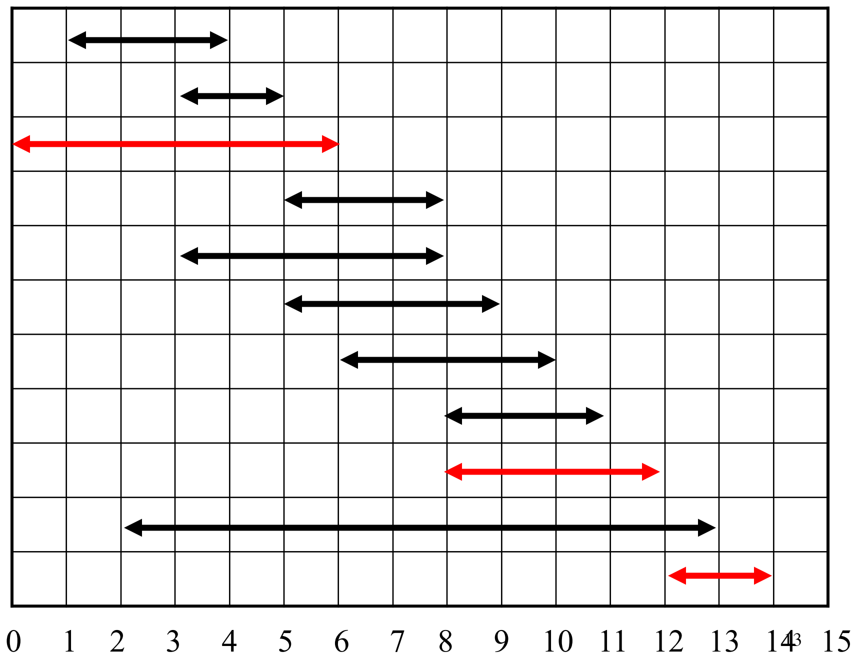
- Here are a set of start and finish times

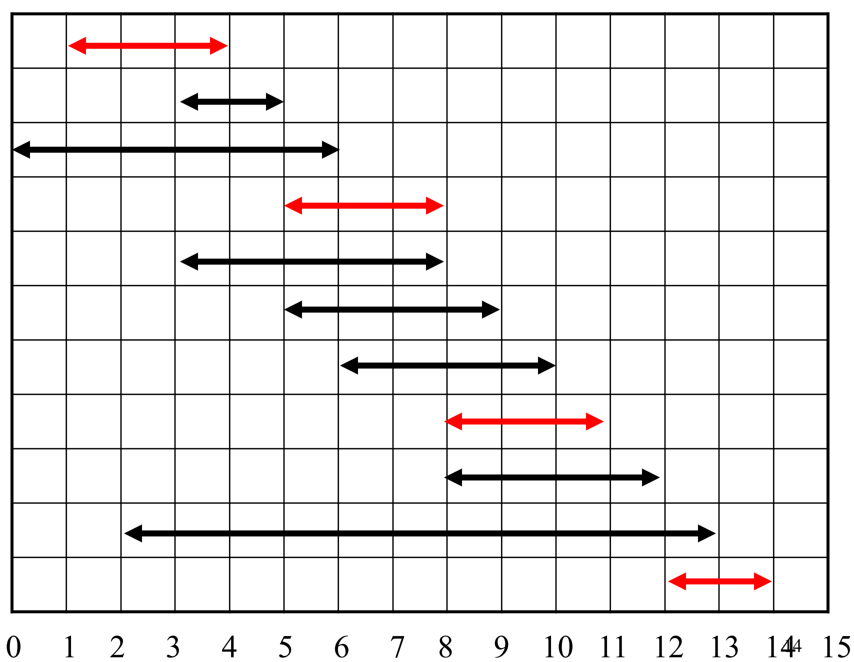
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

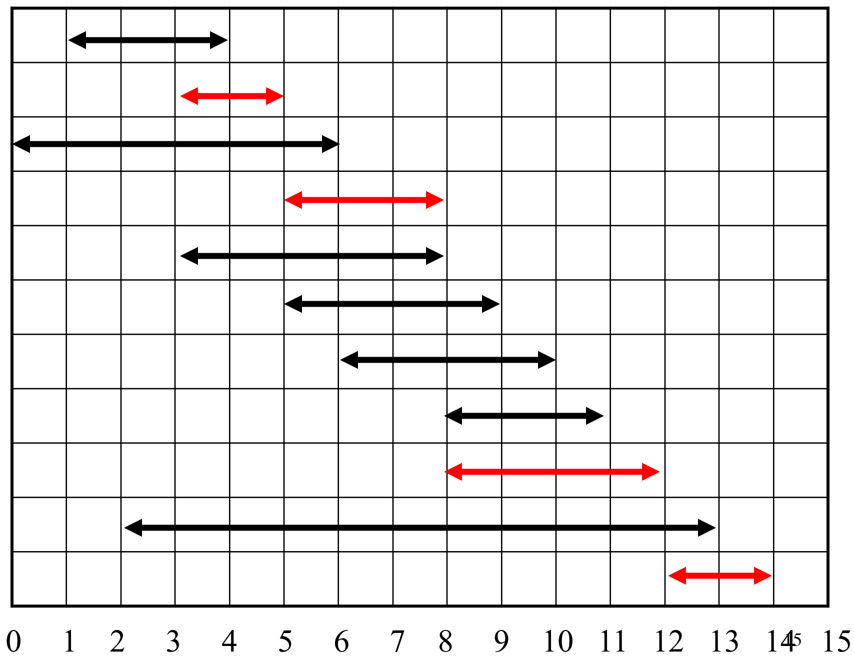
- What is the maximum number of activities that can be completed?
  - $\{a_3, a_9, a_{11}\}$ can be completed
  - But so can $\{a_1, a_4, a_8, a_{11}\}$ which is a larger set
  - But it is not unique, consider $\{a_2, a_4, a_9, a_{11}\}$

11




0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
13


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
14



15

"Greedy" Strategies

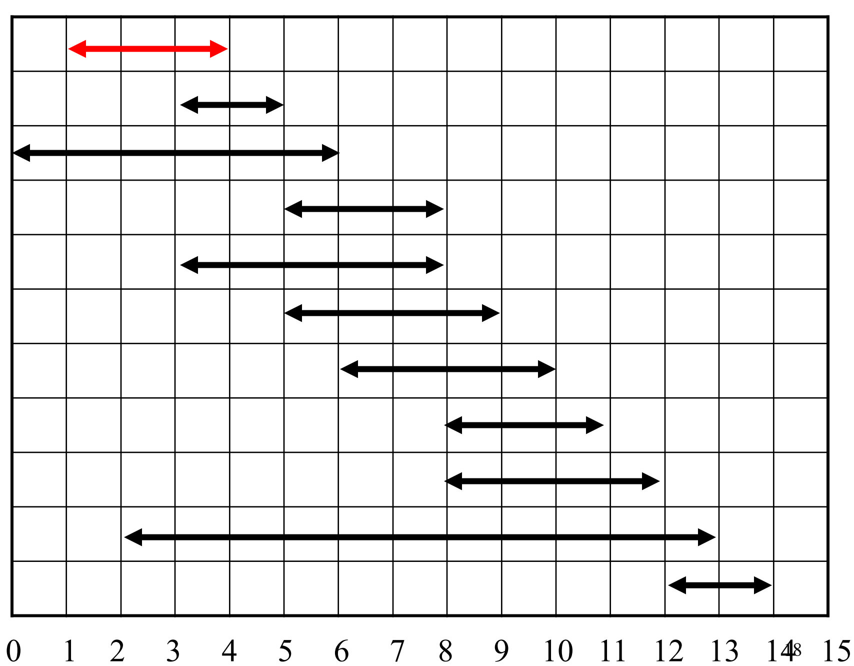
1. Longest first
2. Shortest first
3. Early start first
4. Early finish first
5. None of the above

16

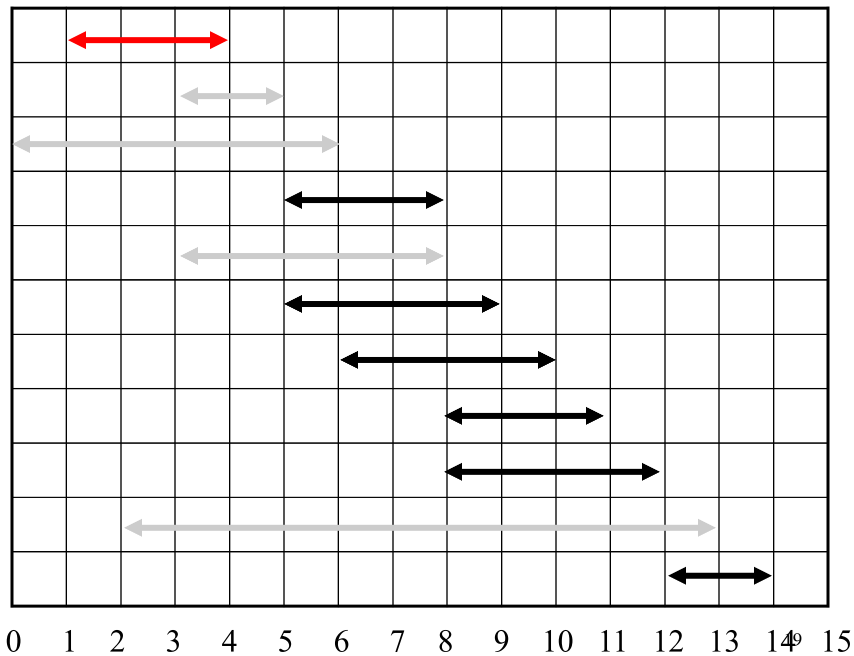
"Early Finish" Greedy strategy

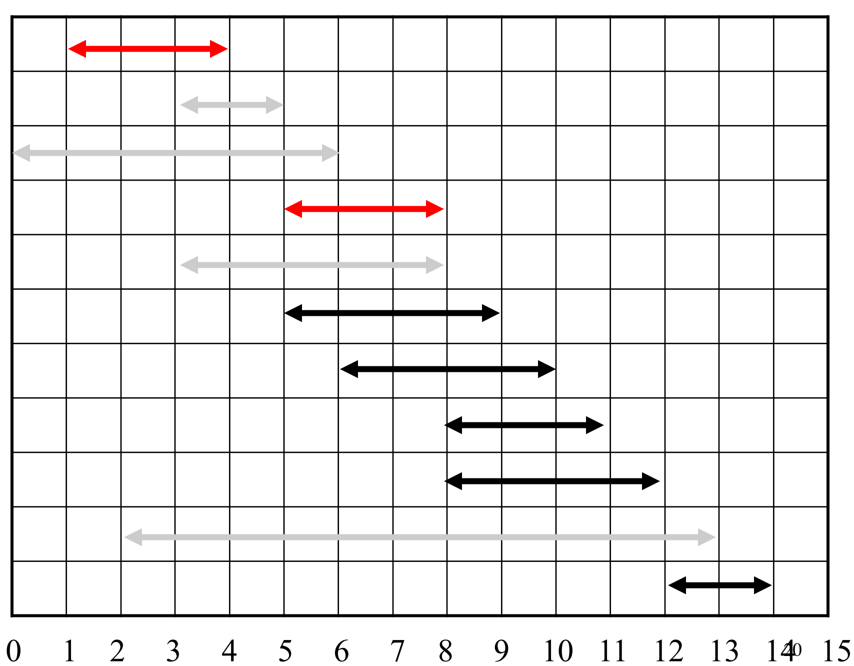
- Sort the activities by finish time
- Schedule first activity (activity 1)
- Remove all activities that conflict with 1
- Recurse on the remaining activities

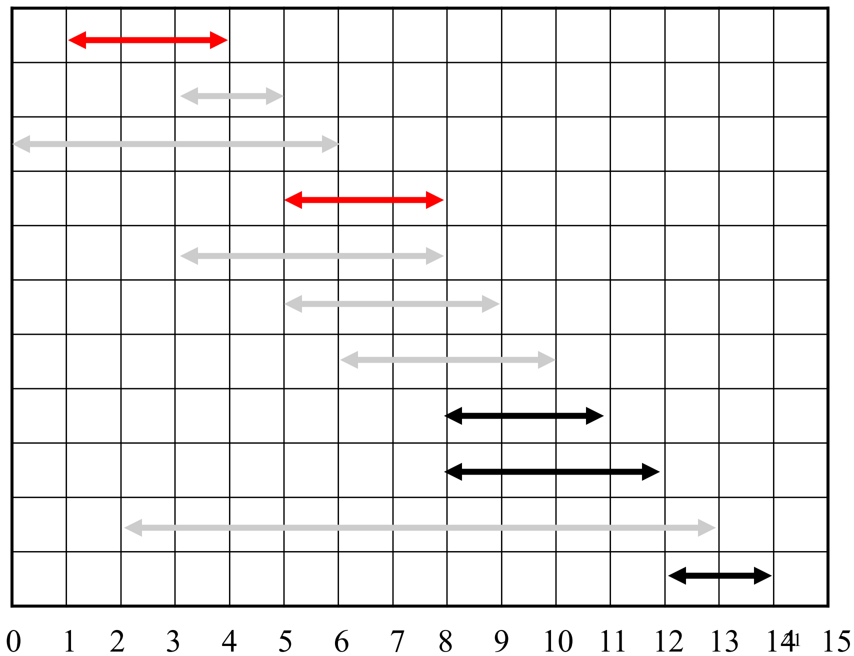
17

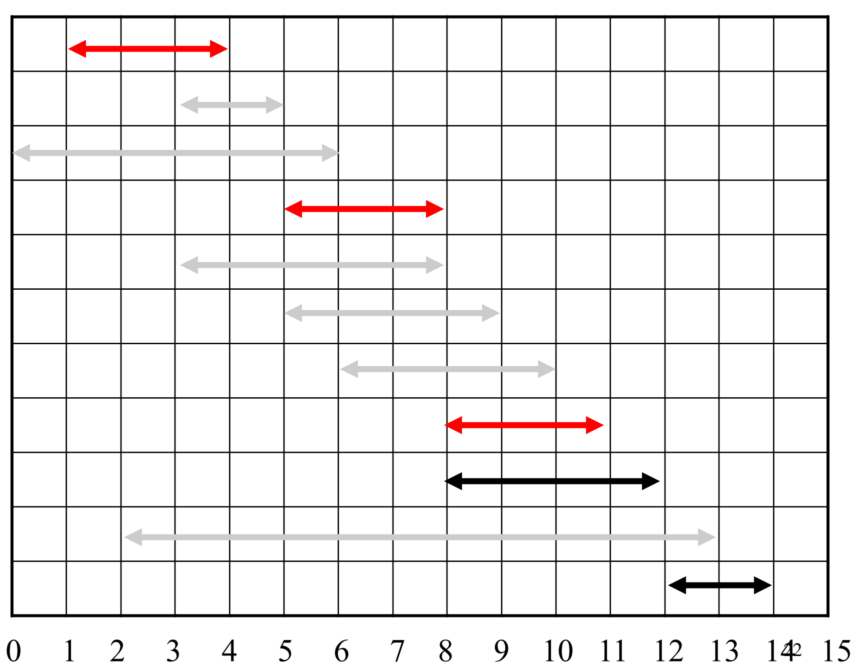


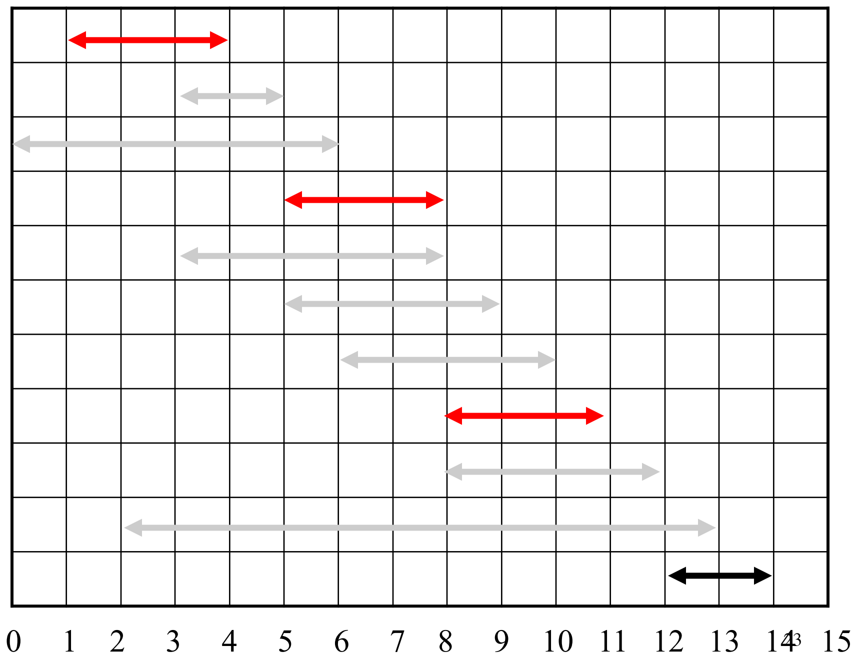
18

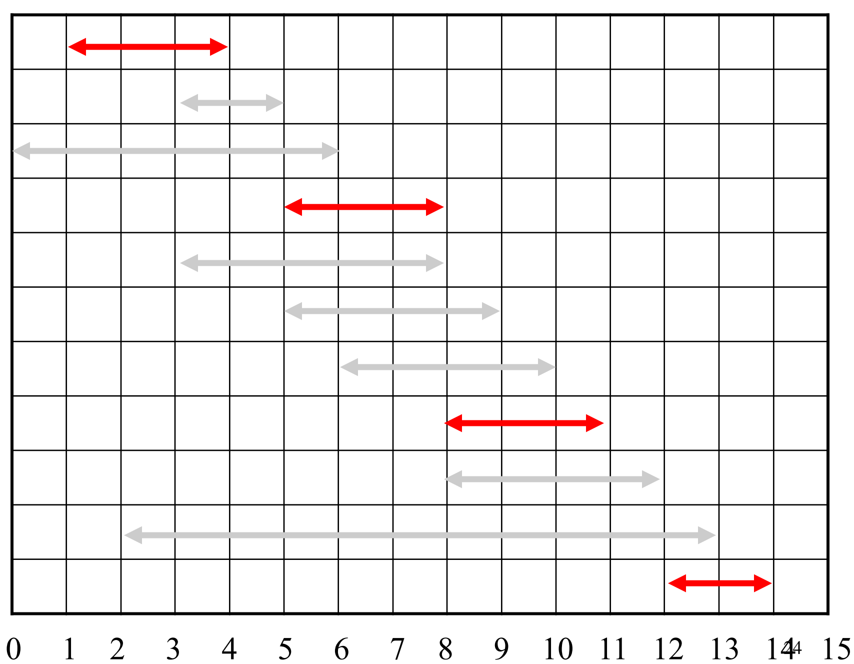

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
19


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
20


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Activity selection in Python

```
def greedy_activity_selection(A):
    A.sort(key=itemgetter(1))
    result = [A[0]]
    i = 0
    for j in range(1,len(A)):
        if A[j][0] >= A[i][1]:
            result.append(A[j])
            i = j
    return result
```

Remark: sort **A** by finish time

Remark: first activity in the solution

Remark: start[j] >= finish[i]

Time complexity? *O(n log n)* to sort, the rest is linear.

25

Greedy

- Goal: build a solution in steps, never making a "mistake" --- maintain the invariant that *the partial solution so far is always extendible to an optimal solution*.

- Choosing activity 1 for the first job maximizes the set of remaining (possible, non-conflicting) jobs.

26

Correctness (optimality)

- *Greedy choice property*:
  First choice is consistent with some opt'l soln
- *Optimal substructure property*:
  After this first choice, to solve the entire problem optimally, it is enough to solve *the remaining sub-problem* optimally.

27

Greedy-Choice Property

- We want to show there is an optimal solution that begins with a greedy choice (i.e., with the first activity, which has the earliest finish time)

28

Greedy-Choice Property

- Suppose $A \subseteq S$ is an optimal solution
  - Order the activities in A by finish time
    Let k be the first activity in A
    - If $k = 1$, the schedule A begins with a greedy choice
    - If $k \neq 1$, show that there is another optimal solution B that begins with the greedy choice (activity 1)
  - Let $B = (A - \{k\}) \cup \{1\}$
    - Activities in B are non-conflicting because activities in A are non-conflicting, k is the first activity to finish and $f_1 \leq f_k$
    - B has the same number of activities as A thus, B is optimal

29

Optimal Substructure

- After the greedy choice of the first activity, the problem reduces to finding an optimal solution for the activity-selection problem over those activities in S that are compatible with the first activity. *Formally:*

Remaining sub-problem is $S' = \{ i \text{ in } S: s_i \geq f_1 \}$.

A' is an optimal solution for S'
if and only if
$A' \cup \{1\}$ is an optimal solution for S.

30

Optimal Substructure

Remaining sub-problem is $S' = \{ i \text{ in } S\text{: } s_i \geq f_1 \}$.

Claim. *A' is an optimal solution for S'*
if and only if
A' ∪ {1} *is an optimal solution for S.*

***Proof*.** (=> *direction*)

Let *A'* be any optimal solution for *S'*.

Then *A'* ∪ {1} is a solution for *S*. (*why*?)

If it is not optimal for *S*, then (by greedy choice) there is a larger solution *B'* ∪ {1} for *S*. But then *B'* is a solution for *S'* (*why*?), and *B'* is larger than *A'*.

(*We leave the <= direction as an exercise.*) *QED*

31

Claim. *Greedy is optimal*

Proof.

greedy(*S*)
= {1} ∪ greedy(*S'*) - defn of greedy
= {1} ∪ opt(*S'*) - induction on |*S*|
= opt(*S*) - optimal substructure

base case:

greedy({}) = {} = opt({})

32

Fractional Knapsack

33

Fractional Knapsack

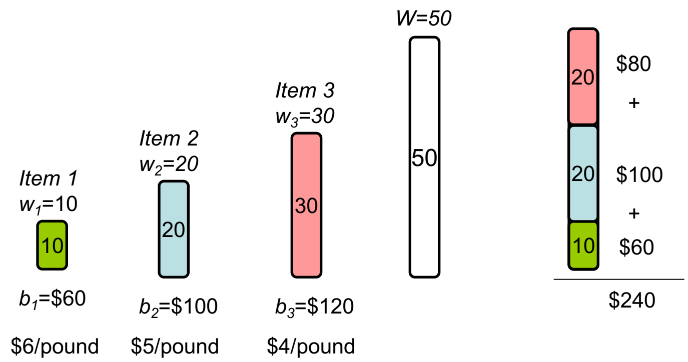
- Given a set S of n items, such that each item i has a positive benefit b_i and a positive weight w_i; the size of the knapsack W
- The problem is to find the amount x_i of each item i which maximizes the total benefit

  $$\sum_i b_i (x_i / w_i)$$

  under the condition that $0 \le x_i \le w_i$ and $\sum_i x_i \le W$

34

Fractional Knapsack - Example



35

Fractional Knapsack in Python

```
def fractional_knapsack(S, W):
    v = []
    for item in S:
        value = float(item[1]) / float(item[0])
        v.append((value,item[1],item[2]))
    v.sort(key=itemgetter(0))
    w, result = 0, []
    while w < W:
        high = v[-1]
        v.pop()
        a = min(high[1], W-w)
        w += a
        result.append((a,high[2]))
    return result
```

Remark: sort **v** by value = benefit/weight

Remark: select and remove the highest value (**high**)

Remark: **a** is how much item **high** we took

36

Fractional Knapsack

- Time complexity is $O(n \log n)$
- Fact: Greedy strategy is optimal for the fractional knapsack problem
- Proof: We will show that the problem has the optimal substructure and the algorithm satisfies the greedy-choice property

37

Greedy Choice property holds

Items (sorted by b_i/w_i)	1	2	3	… j …	n
Optimal solution:	x_1	x_2	x_3	x_j	x_n

- While $x_1 < \min(w_1, W)$:
- For some small $d > 0$, increase x_1 by d, decrease some x_j by d.
- Increase in total benefit, $d\,(b_1/ w_1 - b_j / w_j)$, is non-negative.

- Stop when $x_1 = \min(w_1, W)$.

- Yields equally good solution with $x_1 = \min(w_1, W)$.

38

Optimal substructure property

- Let $x_1 = \min(w_1, W)$ be the greedy choice for the first step.
- (S, W) is the original problem,
- (S', W') is the sub-problem, where $S' = \{2,3,\ldots,n\}$, $W' = W - x_1$

items:	1	2	3	…	n
soln for (S', W'):	-	x_2	x_3	…	x_n
soln for (S, W):	x_1	?	?	…	?

$x_2, x_3, \ldots x_n$ *is an optimal solution to* (S', W')
if and only if
$x_1, x_2, x_3, \ldots x_n$ *is an optimal solution to* (S, W)

Proof. Left as an exercise!

39

Huffman codes

40

Data Compression

- Text files are usually stored by representing each character with an 8-bit ASCII code
- The ASCII encoding is an example of fixed-length encoding, where each character is represented with the same number of bits
- In order to reduce the space required to store a text file, we can exploit the fact that some characters are more likely to occur than others

41

Data Compression

- Variable-length encoding uses binary codes of different lengths for different characters; thus, we can assign fewer bits to frequently used characters, and more bits to rarely used characters
- Huffman coding (section 5.2)

42

File Compression: Example

- An example
  - text: “java”
  - encoding: a = “0”, j = “11”, v = “10”
  - encoded text: 110100 (6 bits)
- How to decode in the case of ambiguity?
  - encoding: a = “0”, j = “11”, v = “00”
  - encoded text: 110000 (6 bits)
  - could be “java”, or “jvv”, or “jaaaa”, or …

43

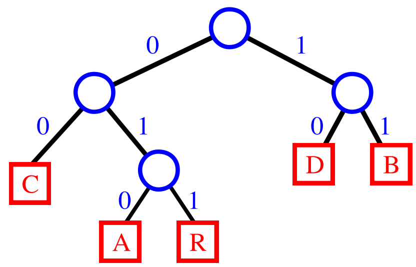
Encoding

- To prevent ambiguities in decoding, we require that the encoding satisfies the prefix rule: no code is a prefix of another
- Example
  - a = “0”, j = “11”, v = “10” satisfies the prefix rule
  - a = “0”, j = “11”, v= “00” does not satisfy the prefix rule (the code of 'a' is a prefix of the codes of 'v')

44

Trie

- We use an encoding trie to satisfy this prefix rule
  - the characters are stored at the external nodes
  - a left child (edge) means 0
  - a right child (edge) means 1



A = 010

B = 11

C = 00

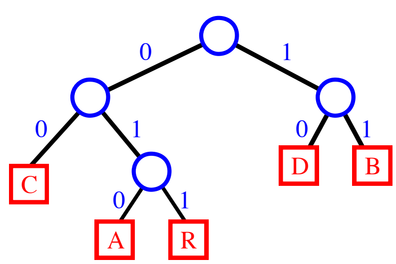
D = 10

R = 011

45

Example of Decoding

- encoded text: 01011011010000101001011011010
- text: ABRACADABRA (11 bytes=88 bits)



A = 010

B = 11

C = 00

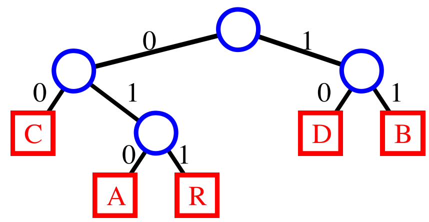
D = 10

R = 011

46

Data Compression

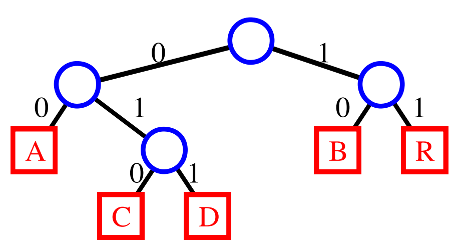
- Problem: We want the encoded text as short as possible
- Example: ABRACADABRA
  01011011010000101001011011010 **29 bits**



47

Data Compression

- Example2: ABRACADABRA
  001011000100001100101100 **24 bits**



48

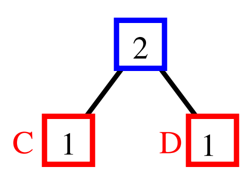
Optimization problem

- Given a character c in the alphabet Σ
  - let $f(c)$ be the frequency of c in the file
  - let $d_T(c)$ be the depth of c in the tree = the length of the codeword
- We want to minimize the number of bits required to encode the file, that is

$$\min_{\substack{\text{binary trees } T \\ \text{with } |\Sigma| \text{ leaves}}} \sum_{c\in\Sigma} f(c)d_T(c)$$

49

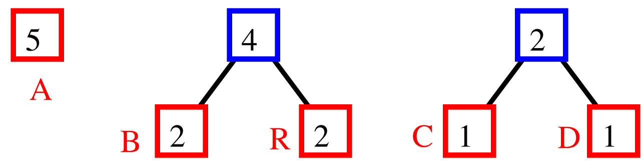
Huffman Encoding: Example

Step *0*	ABRACADABRA					
	character	A	B	R	C	D
	frequency	5	2	2	1	1
Step *1*		5 A	2 B	2 R		

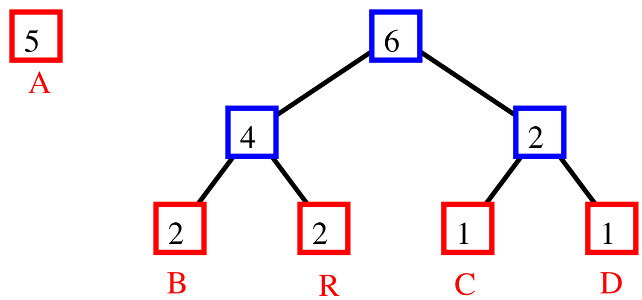
50

Huffman Encoding: Example

Step *2*



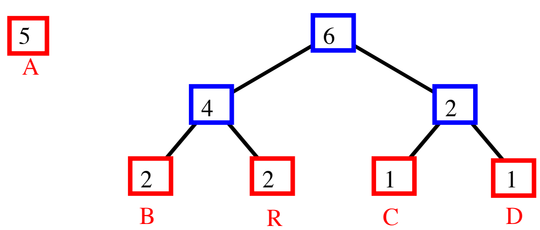
Step *3*



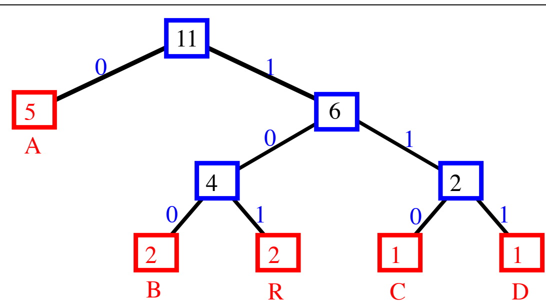
51

Huffman Encoding: Example

Step *4*



Step *5*

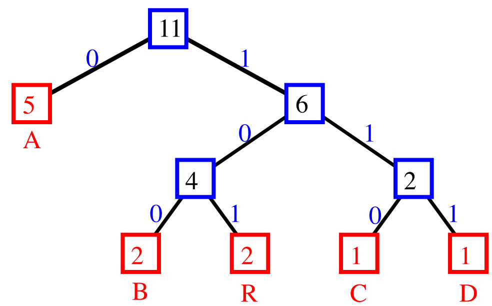


52

Final Huffman Trie

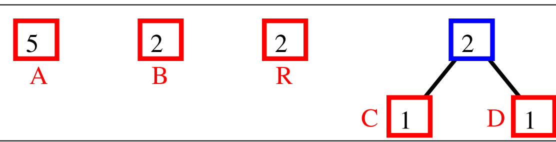
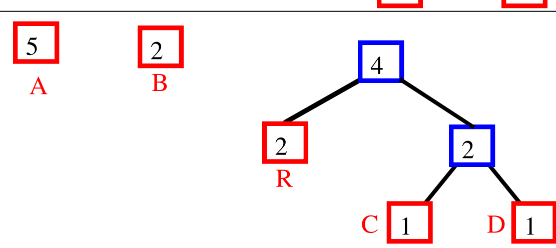
A B R A C A D A B R A

0 100 101 0 110 0 111 0 100 101 0 (23 bits)



53

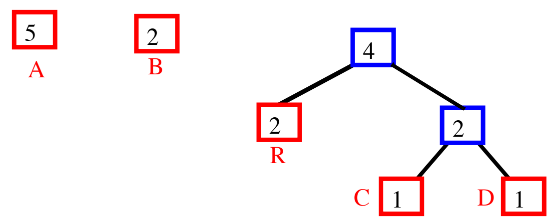
Another Example

Step *0*	ABRACADABRA character: A B R C D frequency: 5 2 2 1 1
Step *1*	
Step *2*	

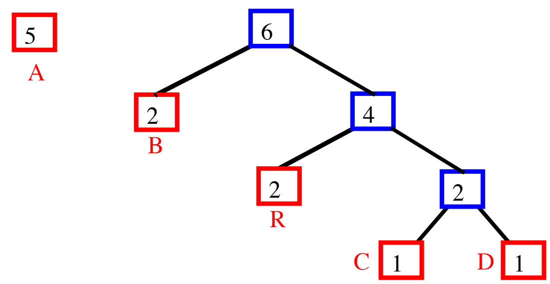
54

Another Example

Step 3



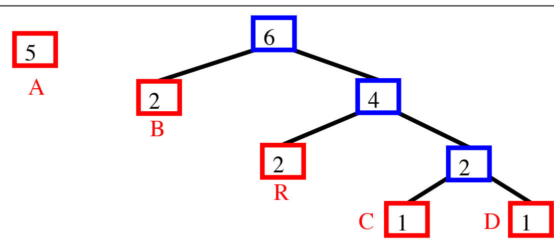
Step 4



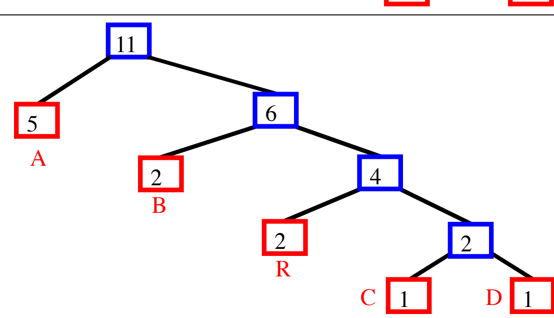
55

Another Example

Step 5



Step 6

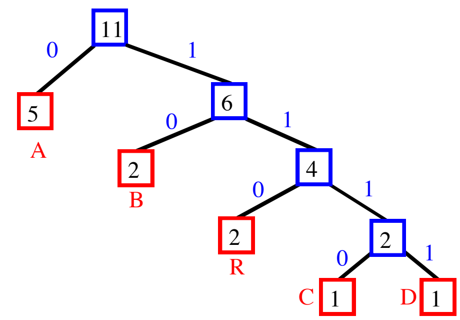


56

Final Trie

A B R A C A D A B R A

0 10 110 0 1110 0 1111 0 10 110 0 (23 bits)



57

Priority queue

- Use a priority queue for storing the nodes
- Priority queue is a queue ordered by priority (heap)
- For our application, priority = frequency
- If there are *k* elements in the queue:
  - Extracting the lowest priority is $O(\log k)$
  - Inserting takes $O(\log k)$

58

Huffman algorithm in Python

```
def makeHuffTree(t):
   heapq.heapify(t)
   while len(t) > 1:
      L, R = heapq.heappop(t), heapq.heappop(t)
      parent = (L[0] + R[0], L, R)
      heapq.heappush(t, parent)
   return t[0]

def printHuffTree(t, prefix = ''):
   if len(t) == 2:
      print t[1], prefix
   else:
      printHuffTree(t[1], prefix + '0')
      printHuffTree(t[2], prefix + '1')
```

Remark: transforms list `t` in a heap in linear time

Remark: returns the tree represented a nested tuple

59

Huffman Algorithm

- Running time for a text of length n with k distinct characters: $O(n + k \log k)$
- If we assume k to be a constant (i.e., not a function of n) then the algorithm runs in $O(n)$ time
- Fact: Using a Huffman encoding trie, the encoded text has minimal length
- Proof: omitted

60

Greedy method: summary

- Task scheduling
- Fractional knapsack
- Huffman encoding (section 5.2)

- Other greedy algorithms that will be covered later: Prim (section 5.1.5), Kruskal (section 5.1.3) and Dijkstra (section 4.4)