CH08-320201

Algorithms and Data Structures

ADS

Lecture 18

Dr. Kinga Lipskoch

Spring 2019

Analysis of Open Addressing (1)

Theorem:

- Assume uniform hashing, i.e., each key is likely to have any one of the $m!$ permutations as its probe sequence.
- Given an open-addressed hash table with load factor $\alpha = n/m < 1$.
- The expected number of probes in an unsuccessful search is, at most $\frac{1}{1-\alpha}$.

Analysis of Open Addressing (2)

Proof:

- At least, one probe is always necessary.
- With probability n/m, the first probe hits an occupied slot, i.e., a second probe is necessary.
- With probability $(n-1)/(m-1)$, the second probe hits an occupied slot, i.e., a third probe is necessary.
- With probability $(n-2)/(m-2)$, the third probe hits an occupied slot, i.e., a fourth probe is necessary.
- ...

Analysis of Open Addressing (3)

Given that $\frac{n-i}{m-i} < \frac{n}{m} = \alpha$ for $i = 1, 2, \ldots, n$.

$$1 + \frac{n}{m}\left(1 + \frac{n-1}{m-1}\left(1 + \frac{n-2}{m-2}\left(\cdots\left(1 + \frac{1}{m-n+1}\right)\cdots\right)\right)\right)$$

$$\leq 1 + \alpha(1 + \alpha(1 + \alpha(\cdots(1+\alpha)\cdots)))$$

$$\leq 1 + \alpha + \alpha^2 + \alpha^3 + \cdots$$

$$= \sum_{i=0}^{\infty} \alpha^i$$

$$= \frac{1}{1-\alpha}.$$

Analysis of Open Addressing (4)

- The successful search takes less number of probes $\left[\text{expected number is at most } \frac{1}{\alpha} \ln \frac{1}{1-\alpha}\right]$.
- We conclude that if α is constant, then accessing an open-addressed hash table takes constant time.
- For example, if the table is half full, the expected number of probes is $1/(1-0.5) = 2$.
- Or, if the table is 90% full, the expected number of probes is $1/(1-0.9) = 10$.

Summary

- Dynamic sets with queries and modifying operations.
- Array: Random access, search in $O(\lg n)$, but modifying operations $O(n)$.
- Stack: LIFO only. Operations in $O(1)$.
- Queue: FIFO only. Operations in $O(1)$.
- Linked list: Modifying operations in $O(1)$, but search $O(n)$.
- BST: All operations in $O(h)$.
- Red-black trees: All operations in $O(\lg n)$.
- Heap: All operations in $O(\lg n)$.
- Hash tables: Operations in $O(1)$, but additional storage space.

Design Concepts

- We have been looking into different algorithms and, in particular, emphasized one design concept, namely, the Divide & Conquer strategy, which was based on recursions and whose analysis was given by recurrences.
- Now, we are going to look into further design concepts.

Activity-Selection Problem (1)

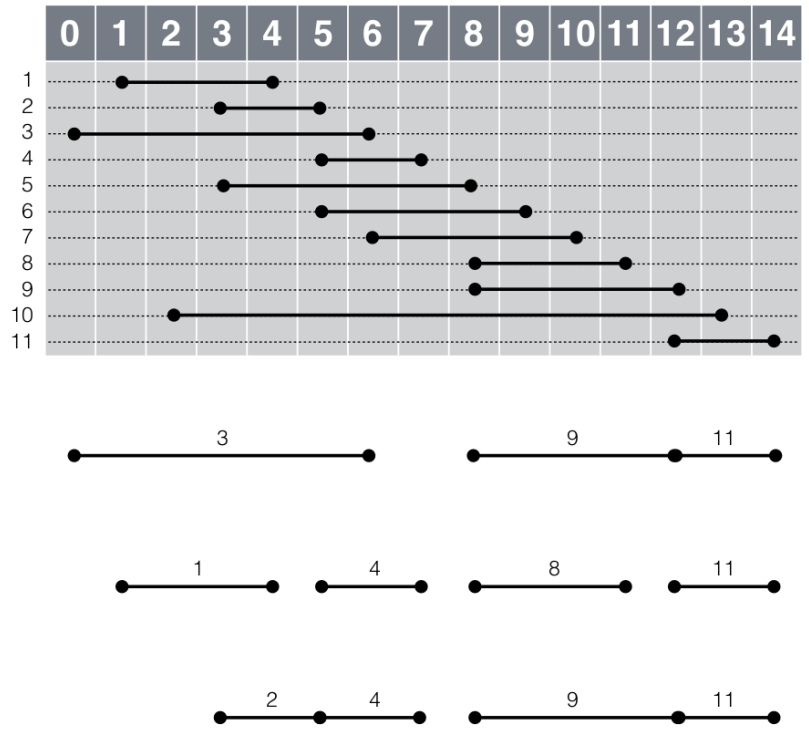
- Suppose we have a set $S = \{a_1, a_2, ..., a_n\}$ of n activities.
- The activities wish to use a resource, which can only be used by one activity at a time.
- Each activity a_i has a start time s_i and a finish time f_i, where $0 \leq s_i < f_i < \infty$.
- Two activities a_i and a_j are compatible, if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint.
- The activity-selection problem is to select a maximum-size subset of mutually compatible activities.

Activity-Selection Problem (2)

- Example:

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

- $\{a_3, a_9, a_{11}\}$ is a subset of mutually compatible activities.
- $\{a_1, a_4, a_8, a_{11}\}$ is a largest subset of mutually compatible activities.
- $\{a_2, a_4, a_9, a_{11}\}$ is another largest subset of mutually compatible activities.



Sorting

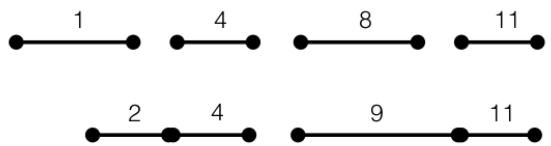
- We can apply a sorting algorithm to the finish times, which operates in $O(n \lg n)$ time.
- Then, we can assume that the activities are sorted, i.e., $f_1 \leq f_2 \leq \ldots \leq f_n$.

Greedy Algorithm

- A greedy algorithm always makes the choice that looks best at the moment.
- I.e., it makes a locally optimal choice in the hope that it will lead to a globally optimal solution.

Greedy Approach (1)

- After sorting, a_1 has the earliest finish time f_1.
- A greedy approach starts with taking a_1 as a locally optimal choice.
- Lemma:
  The greedy choice of picking a_1 as first choice is optimal.
- Proof:
  - Suppose A is a globally optimal solution for set S.
  - Let $a_k \in A$ be the activity with earliest finish time f_k in A.
  - If $k = 1$, then $a_1 \in A$ and we are done.
  - If $k > 1$, then we can replace A by $(A \setminus \{a_k\}) \cup \{a_1\}$.
  - Since $f_1 \leq f_k$, this is still an optimal solution.
  - Hence, we can always start with a_1.



Greedy Approach (2)

- After the first step, we consider the subproblem $S' = \{a_i \in S : s_i \geq f_1\}$.
- We apply the same greedy strategy.
- Lemma:
  $A \setminus \{a_1\}$ is the optimal solution for S'.
- Proof:
  - Let B be a solution for S' that is larger than $A \setminus \{a_1\}$.
  - Then, $B \cup \{a_1\}$ would be solution for S that is larger than A.
  - Contradiction.

Greedy Approach (3)

Using the two lemmata we can prove by induction that the greedy approach delivers the globally optimal solution.

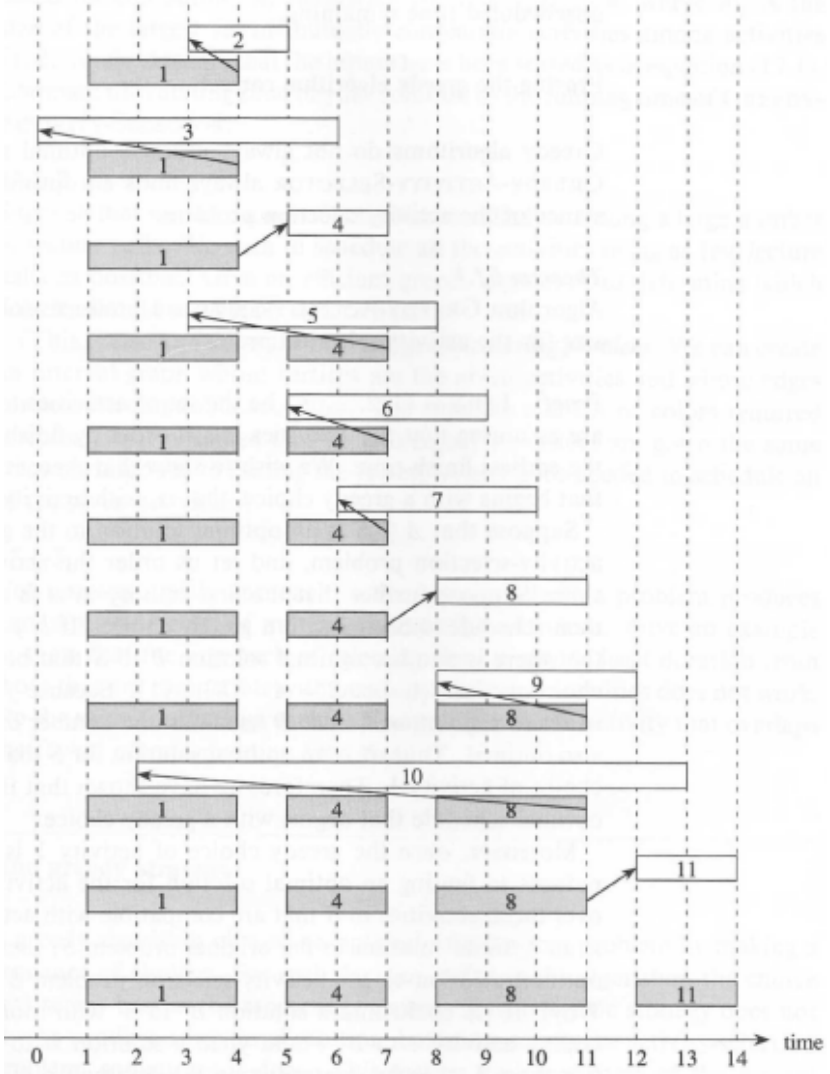
Greedy Algorithm

```
1  Greedy-Selector(S)
2    // Assume S = {a[1], ..., a[n]}
3    // with activities sorted by f[i].
4    A := {a[1]}
5    j := 1
6    for i := 2 to n do
7      if s[i] >= f[j]
8        then A := A union {a[i]}
9             j := i
10   return A
```

Example

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

Alternative:
Recursive function



Greedy Algorithm (Recursive)

RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)

```
1  m = k + 1
2  while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.

Time Complexity of Greedy Approach

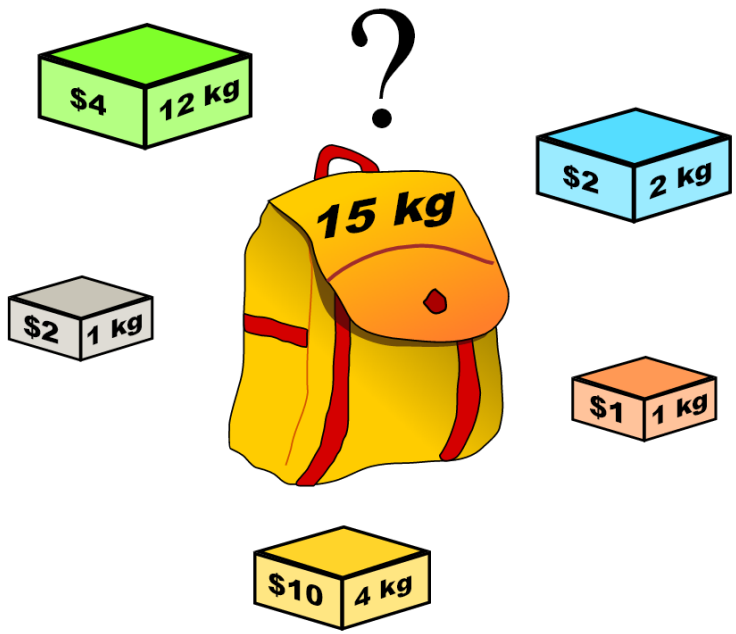
- $O(n)$ – if already sorted
- $O(n \lg n)$ – if not sorted
- Comparison to brute-force approach:
  - Brute-force approach would try all combinations, reject all the combinations that have incompatibilities, and pick among the remaining ones one with maximum number of activities.
  - Time complexity: $O(2^n)$

Greedy Algorithm in General

- Greedy algorithms build upon solving subproblems.
- Greedy approaches make a locally optimal choice.
- There is no guarantee that this will lead to a globally optimal solution.
- Having found a global optimum requires a proof.

The Knapsack Problem (1)

A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?



The Knapsack Problem (2)

Problem:

- Given some items, pack the knapsack to get the maximum total value.
- Each item has some weight and some value.
- Total weight that we can carry is no more than some fixed number W.
- We must consider weights of items as well as their values.

Item #	Weight	Value
1	1	8
2	3	6
3	5	5

The Knapsack Problem (3)

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and value v_i (assuming that all w_i and W are integer values)
- How to pack the knapsack to achieve maximum total value of packed items?
- Mathematically:

$$\max \sum_{i \in T} v_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

where T is a set of indices of the items from another set $S' \subseteq S$

Brute-Force Approach for the Knapsack Problem

Algorithm:

- Generate all 2^n subsets
- Discard all subsets whose sum of the weights exceed W (not feasible)
- Select the maximum total benefit of the remaining (feasible) subsets

Time complexity: $O(2^n)$

Example: Brute-Force Approach for the Knapsack Problem

$S = \{(item_1, 5, \$70), (item_2, 10, \$90), (item_3, 25, \$140)\}, W = 25$

Subsets:

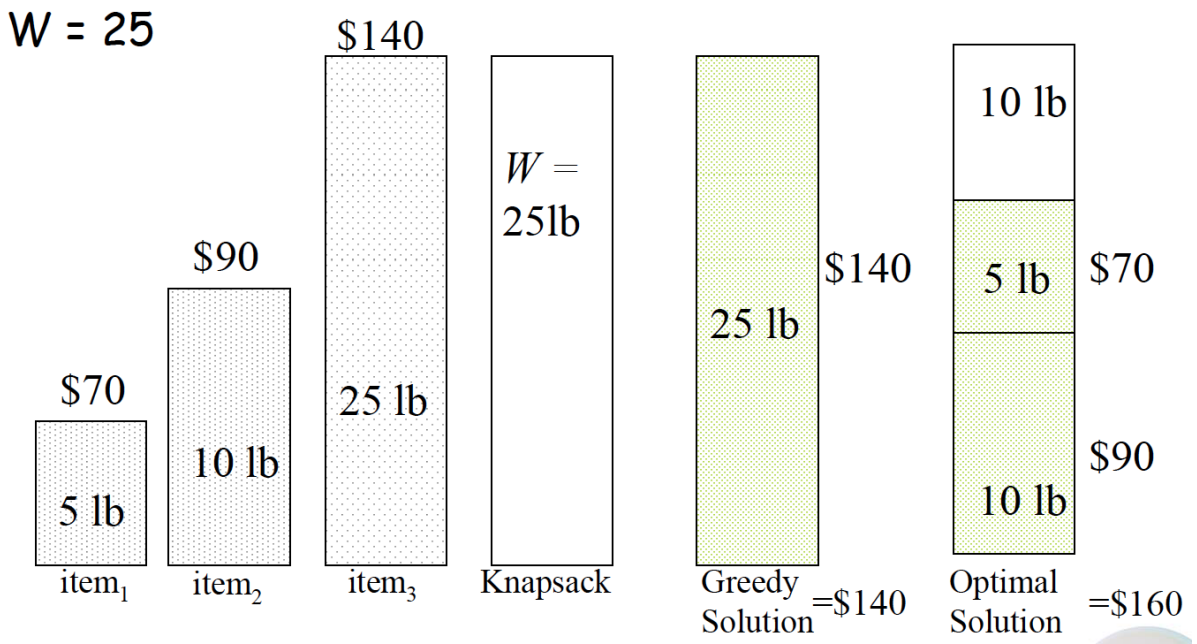
1. $\{\}$ $Profit = \$0$
2. $\{(item_1, 5, \$70)\}$ $Profit = \$70$
3. $\{(item_2, 10, \$90)\}$ $Profit = \$90$
4. $\{(item_3, 25, \$140)\}$ $Profit = \$140$
5. $\{(item_1, 5, \$70), (item_2, 10, \$90)\}$ $Profit = \$160$
6. $\{(item_2, 10, \$90), (item_3, 25, \$140)\}$ exceeds W
7. $\{(item_1, 5, \$70), (item_3, 25, \$140)\}$ exceeds W
8. $\{(item_1, 5, \$70), (item_2, 10, \$90), (item_3, 25, \$140)\}$ exceeds W

Greedy Approach for the Knapsack Problem

- Use a greedy approach to be more efficient
- What would be the greedy choice?
  - Maximum beneficial item
  - Minimum weight item
  - Maximum weight item
  - Maximum value per unit item
- Asymptotic time complexity: $O(n \lg n)$

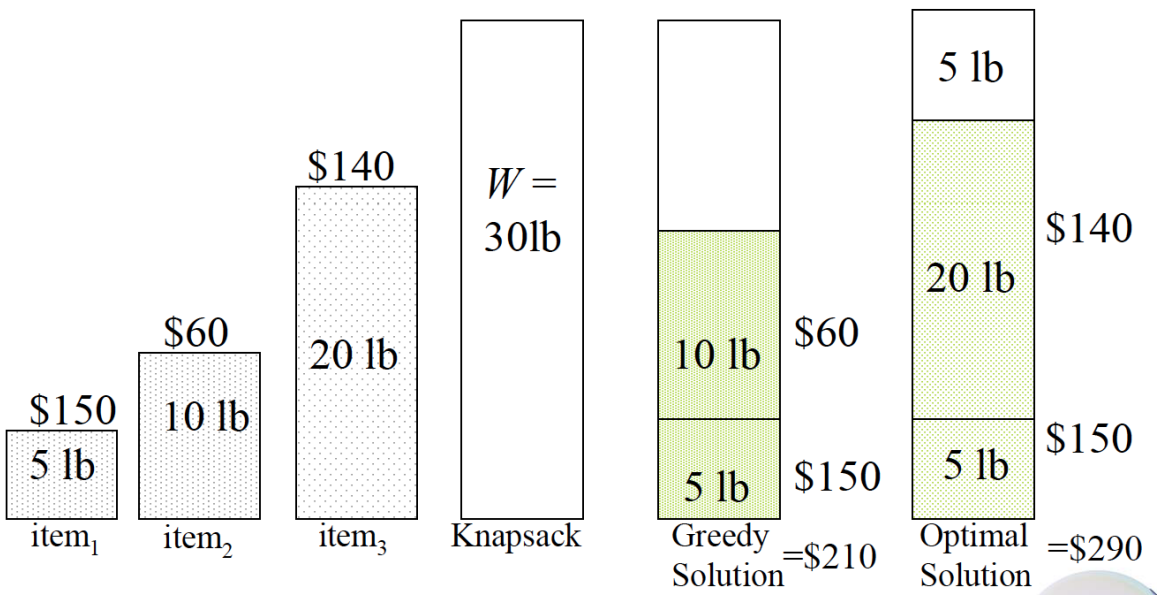
Greedy 1: Maximum Beneficial Item

$$S = \{(item_1, 5, \$70), (item_2, 10, \$90), (item_3, 25, \$140)\}, W = 25$$



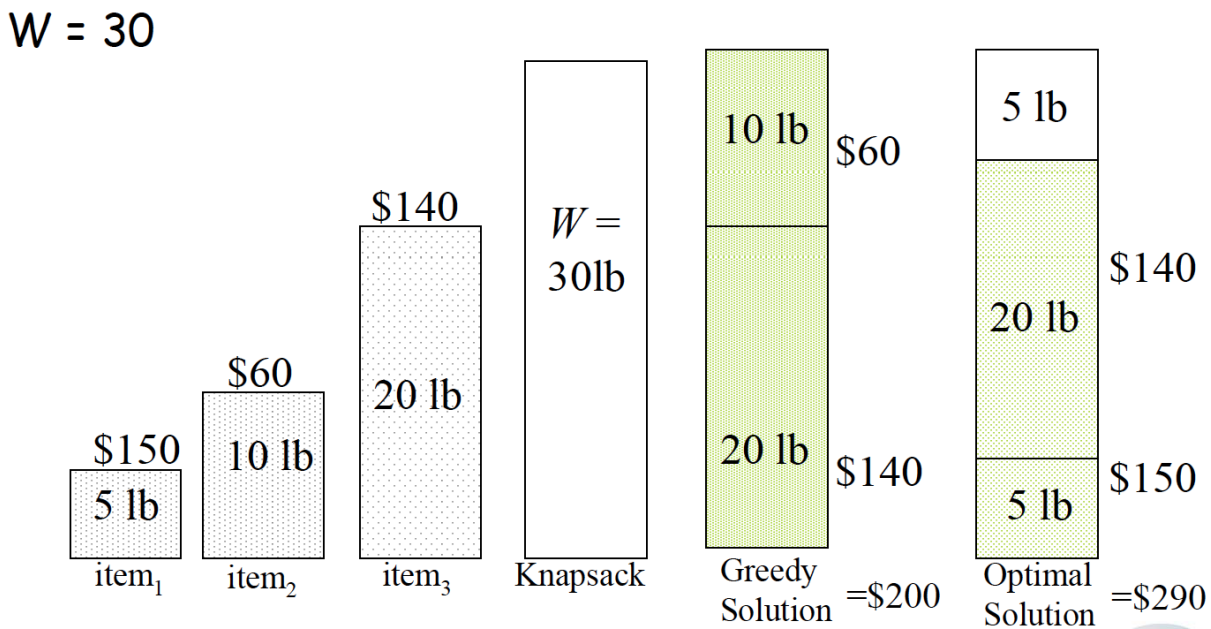
Greedy 2: Minimum Weight Item

$$S = \{(item_1, 5, \$150), (item_2, 10, \$60), (item_3, 20, \$140)\}, W = 30$$



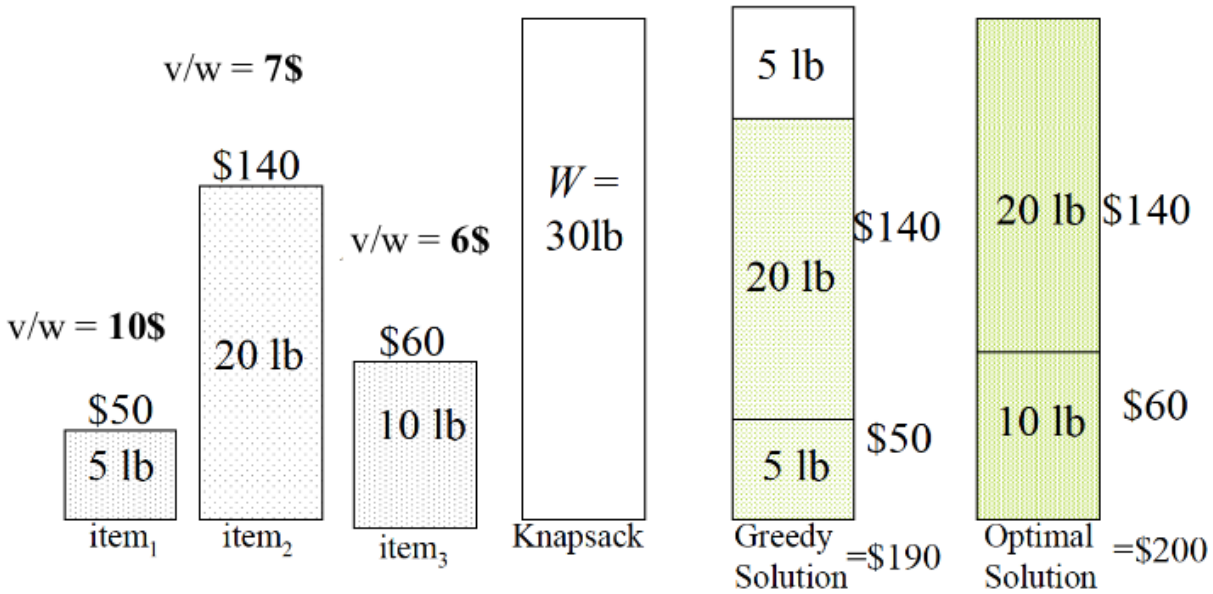
Greedy 3: Maximum Weight Item

$$S = \{(item_1, 5, \$150), (item_2, 10, \$60), (item_3, 20, \$140)\}, W = 30$$



Greedy 4: Maximum Value per Weight

$$S = \{(item_1, 5, \$50), (item_2, 20, \$140), (item_3, 10, \$60)\}, W = 30$$

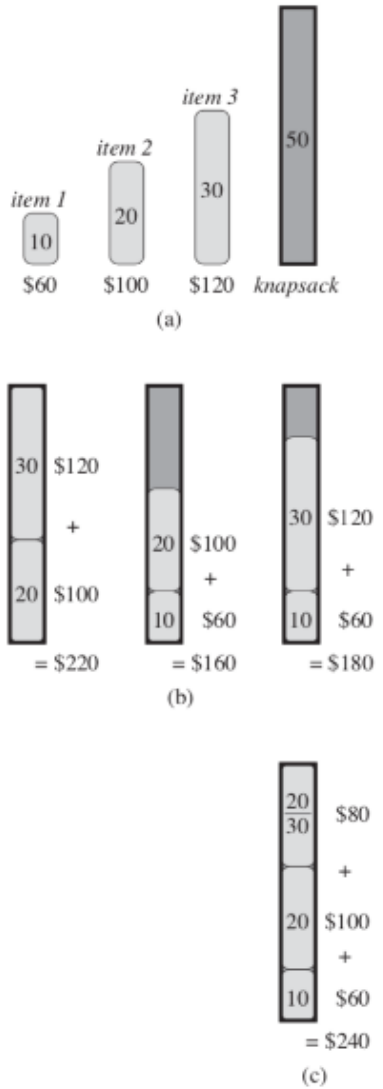


Conclusions: Greedy Approach for the Knapsack Problem

- As already mentioned, the locally optimal choice of a greedy approach does not necessary lead to a globally optimal one.
- For the knapsack problem, the greedy approach actually fails to produce a globally optimal solution.
- However, it produces an approximation, which sometimes is good enough.

0-1 vs. Fractional Knapsack Problem

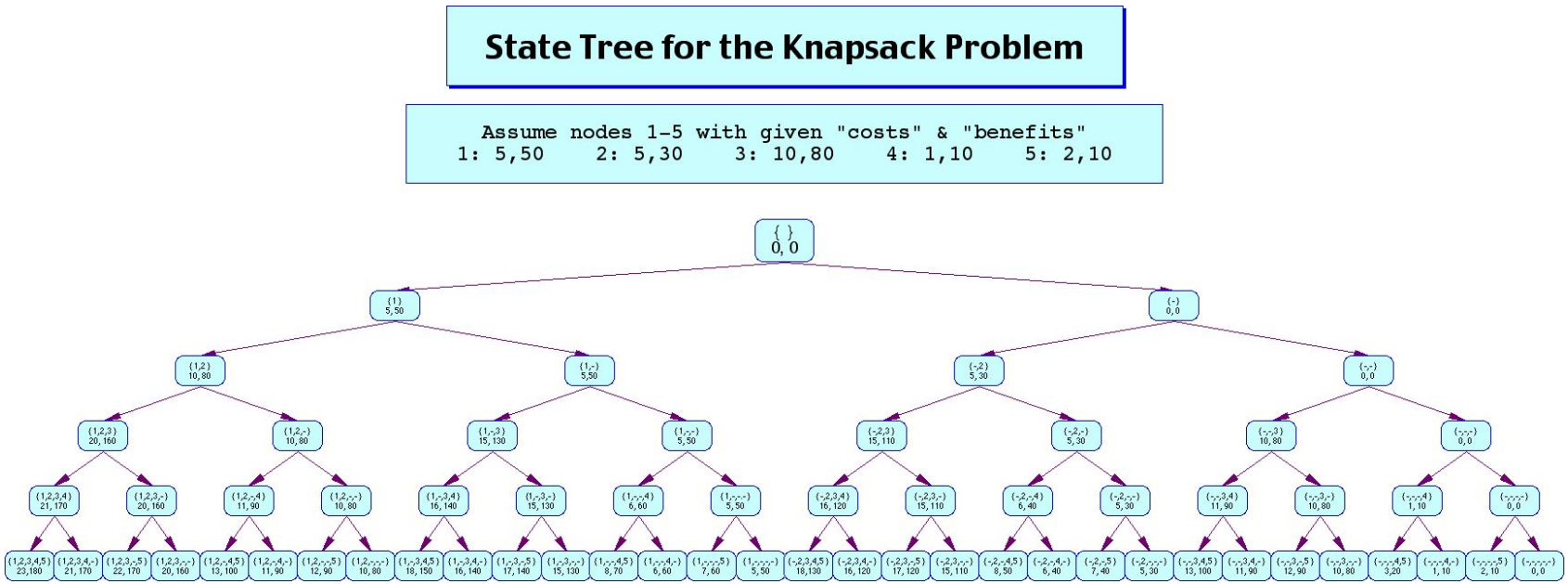
- 0-1 knapsack problem
  - Either take (1) or leave an object (0)
  - Greedy fails to produce global optimum
- fractional knapsack problem
  - You can take fractions of an object
  - Greedy strategy: value per weight v/w $\rightarrow$ begin taking as much as possible of item with greatest v/w , then with next greater v/w, ...
  - Leads to global optimum (proof by contradiction)
- What is the difference?



Alternatives for 0-1 Knapsack (1)

Brute-Force:

- Benefit: it finds the optimum
- Drawback: it takes very long - $O(2^n)$
- Because recomputing the results of the same subproblems over and over again



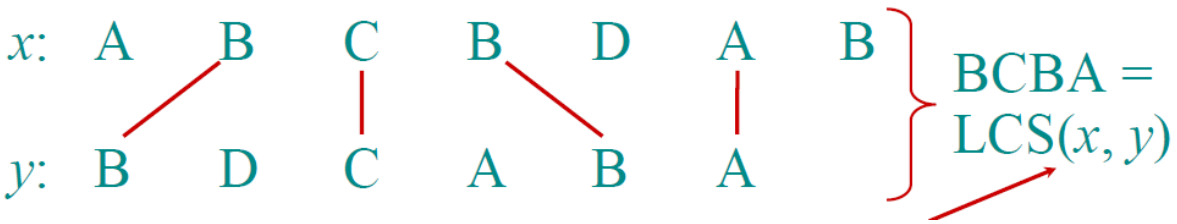
Alternatives for 0-1 Knapsack (2)

Dynamic programming:

- Optimal substructure:
  - optimal solution to problem consists of optimal solutions to subproblems
- Overlapping subproblems:
  - few subproblems in total, many recurring instances of each
- Main idea:
  - use a table to store solved subproblems

Dynamic Programming: Problem

- Given two sequences $x[1..m]$ and $y[1..n]$, find a longest subsequence common to both of them.
- Example:



Brute-Force Solution

Check every subsequence of $x[1..m]$ to see if it is also a subsequence of $y[1..n]$.

Analysis:

- Checking per subsequence is done in $O(n)$.
- As each bit-vector of m determines a distinct subsequence of x, x has 2^m subsequences.
- Hence, the worst-case running time is $O(n \cdot 2^m)$, i.e., it is exponential.

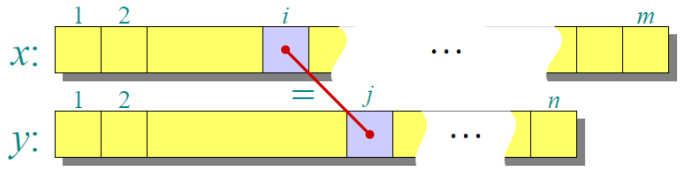
Strategy

- Look at length of longest-common subsequence.
- Let $|s|$ denote the length of a sequence s.
- To find $LCS(x, y)$, consider prefixes of x and y (i.e. we go from right to left)
- Definition: $c[i, j] = |LCS(x[1..i], y[1..j])|$.
  In particular, $c[m, n] = |LCS(x, y)|$.
- Theorem (recursive formulation):

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise.} \end{cases}$$

Proof (1)

Case $x[i] = y[j]$:



Let $z[1..k] = LCS(x[1..i], y[1..j])$ with $c[i,j] = k$.
Then, $z[k] = x[i] = y[j]$ (else z could be extended).
Thus, $z[1..k-1]$ is CS of $x[1..i-1]$ and $y[1..j-1]$.
Claim: $z[1..k-1] = LCS(x[1..i-1], y[1..j-1])$.

- Assume w is a longer CS of $x[1..i-1]$ and $y[1..j-1]$, i.e., $|w| > k-1$.
- Then the concatenation $w + +z[k]$ is a CS of $x[1..i]$ and $y[1..j]$ with length $> k$.
- This contradicts $|LCS(x[1..i], y[1..j])| = k$.
- Hence, the assumption was wrong and the claim is proven.

Hence, $c[i-1,j-1] = k-1$, i.e., $c[i,j] = c[i-1,j-1] + 1$.

Proof (2)

Case $x[i] \neq y[j]$:

Then, $z[k] \neq x[i]$ or $z[k] \neq y[j]$.

- $z[k] \neq x[i]$:
  Then, $z[1..k] = LCS(x[1..i-1], y[1..j])$.
  Thus, $c[i-1, j] = k = c[i, j]$.
- $z[k] \neq y[j]$:
  Then, $z[1..k] = LCS(x[1..i], y[1..j-1])$.
  Thus, $c[i, j-1] = k = c[i, j]$.

In summary, $c[i, j] = \max\{c[i-1, j], c[i, j-1]\}$.

Dynamic Programming Concept (1)

Step 1: Optimal substructure.
An optimal solution to a problem contains optimal solutions to subproblems.

Example:
If $z = LCS(x, y)$, then any prefix of z is an LCS of a prefix of x and a prefix of y.

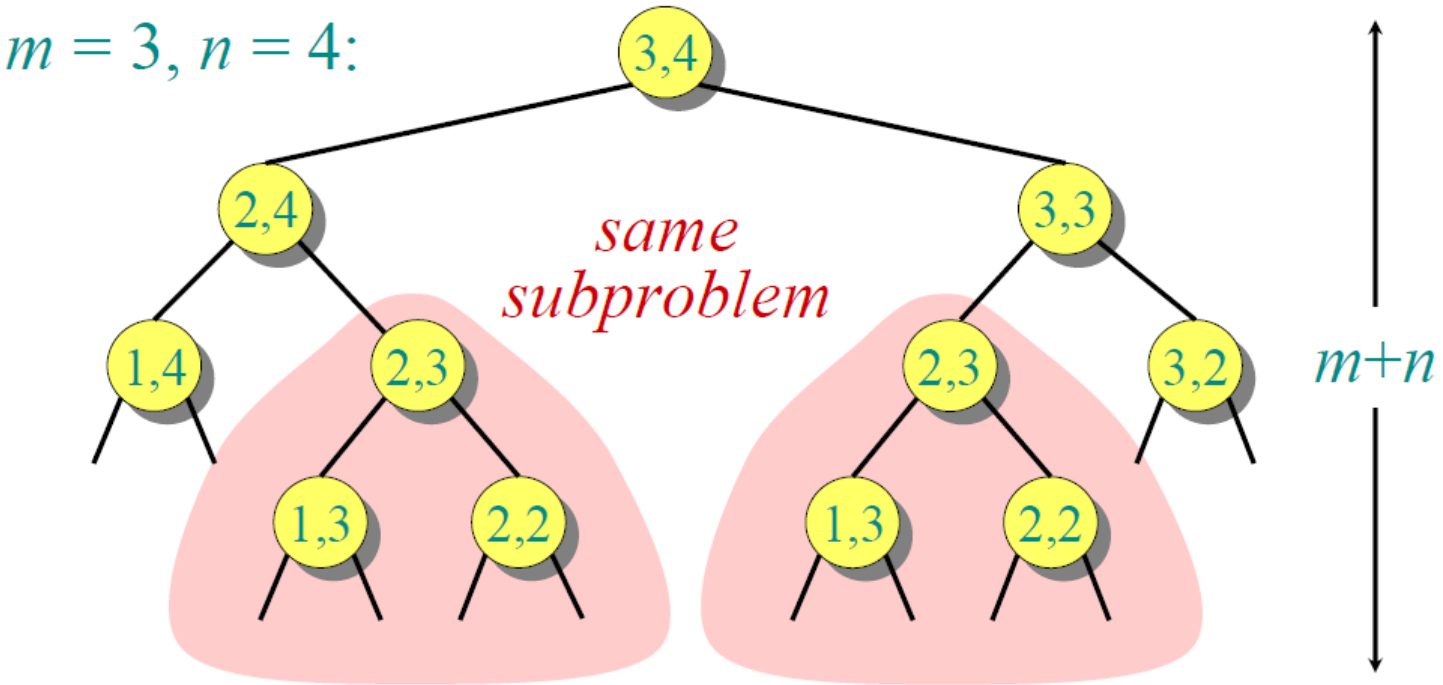
Recursive Algorithm

- Computation of the length of *LCS*:

```
1 LCSlength(x,y,i,j):
2   if i=0 or j=0
3     return 0
4   else if x[i] = y[j]
5     return LCSlength(x,y,i-1,j-1)+1
6   else return max {LCSlength(x,y,i-1,j),
7                    LCSlength(x,y,i,j-1)}
```

- Remark: if $x[i] \neq y[j]$, the algorithm evaluates two subproblems that are very similar.

Recursive Tree



Height $= m + n \Rightarrow$ work potentially exponential, but we're solving subproblems already solved!

Dynamic Programming Concept (2)

Step 2: Overlapping subproblems.
A recursive solution contains a "small" number of distinct subproblems repeated many times.

Example:
The number of distinct LCS subproblems for two prefixes of lengths m and n is only $m \cdot n$.

Memoization Algorithm

Memoization:

- After computing a solution to a subproblem, store it in a table.
- Subsequent calls check the table to avoid repeating the same computation.

Recursive Algorithm with Memoization

Computation of the length of *LCS*:

```
1 LCSlength (x,y,i,j):
2   if c[i,j] = NIL
3     then if i=0 or j=0
4       c[i,j] = 0
5     else if x[i] = y[j]
6       c[i,j] = LCSlength (x,y,i-1,j-1)+1
7     else c[i,j] = max {LCSlength (x,y,i-1,j),
8                        LCSlength (x,y,i,j-1)}
9   return c[i,j]
```

Dynamic Programming

Compute the table bottom-up:

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4