Advanced Analysis of Algorithms

Dr. Qaiser Abbas
Department of Computer Science & IT,
University of Sargodha, Sargodha, 40100, Pakistan
qaiser.abbas@uos.edu.pk
Material partially adopted from the following link:
http://www.cse.unl.edu/~goddard/Courses/CSCE310J

Edit Distance

- DNA Sequence Comparison: First Success Story
  - Finding sequence similarities with genes of known function is a common approach to infer a newly sequenced gene's function
  - In 1984 Russell Doolittle and colleagues found similarities between cancer-causing gene and normal growth factor (PDGF) gene.

Edit Distance

```
Score =  248 bits (129), Expect = 1e-63
Identities = 213/263 (80%), Gaps = 34/263 (12%)
Strand = Plus / Plus

Query: 161 atatcaccacgtcaaaggtgactccaactcca---ccactccattttgttcagataatgc 217
           |||||||||||||||||||||||||||||  |     | |   || ||||||||||||||
Sbjct: 481 atatcaccacgtcaaaggtgactccaact-tattgatagtgttttatgttcagataatgc 539

Query: 218 ccgatgatcatgtcatgcagctccaccgattgtgagaacgacagcgacttccgtcccagc 277
           |||||||   ||||||||||||||||||||| || |            ||||||||||||
Sbjct: 540 ccgatgactttgtcatgcagctccaccgatttg-g------------ttccgtcccagc 586

Query: 278 c-gtgcc--aggtgctgcctcagattcaggttatgccgctcaattcgctgcgtatatcgc 334
           |  || |  | ||||||||||||||||||||||||||||||||||||||| |||||||||
Sbjct: 587 caatgacgta-gtgctgcctcagattcaggttatgccgctcaattcgctgggtatatcgc 645

Query: 335 ttgctgattacgtgcagctttcccttcaggcggga------------ccagccatccgtc 382
           |||||||||||||||||||||||||||||||||||            |||||||||||||
Sbjct: 646 ttgctgattacgtgcagctttcccttcaggcgggattcatacagcggccagccatccgtc 705

Query: 383 ctccatatc-accacgtcaaagg 404
            |||||||| |||||||||||||
Sbjct: 706 atccatatcaaccacgtcaaagg 728
```

Example BLAST alignment

Edit Distance

Problem: Given two strings of size m, n and set of operations substitution (S), insert (I) and delete (D) all at equal cost. Find minimum number of edits (operations) required to convert one string into another.

Minimum Edit Distance

- The minimum edit distance between two strings is the minimum number of edit operations (insert, delete, substitution) needed to transform one string into another.
- For example the gap between “intention” and “execution” is 5 operations, which can be represented in three ways as follows:

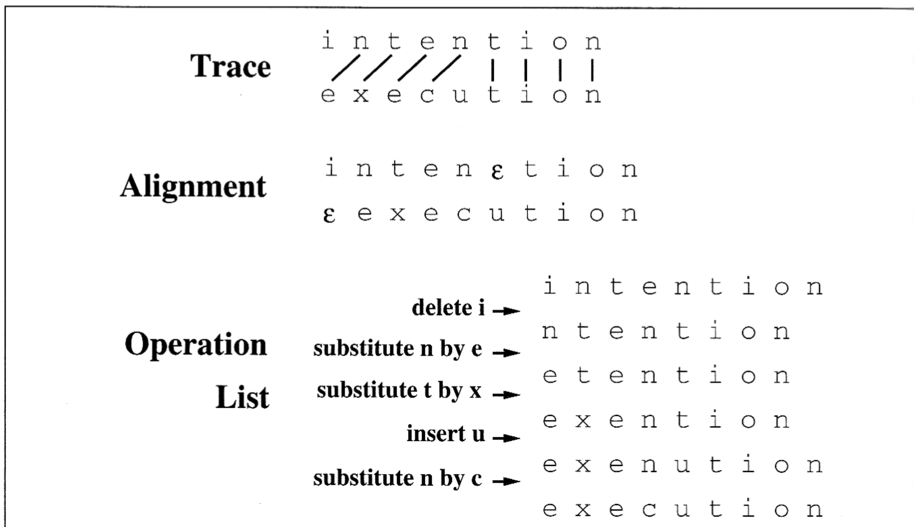


Figure 5.4 Three methods for representing differences between sequences (after Kruskal (1983))

Minimum Edit Distance

- **Applications**
  - could be used for multi-typo correction
  - used in Machine Translation Evaluation (MTEval)
- **Cost and Weight models**
  - **Levenshtein (Cost)**
    - insertion, deletion and substitution all have unit cost
  - **Levenshtein** (alternate) **(Cost)**
    - insertion, deletion have unit cost
    - substitution is twice as expensive
    - *substitution = one insert followed by one delete*
  - **Typewriter (Weight)**
    - insertion, deletion and substitution all have unit cost
    - modified by key proximity

Minimum Edit Distance

- ***Dynamic Programming***
  - divide-and-conquer
    - to solve a problem we divide it into sub-problems
  - sub-problems may be repeated
    - don’t want to re-solve a sub-problem the 2nd time around
  - idea: put solutions to sub-problems in a table
    - and just look up the solution 2nd time around, thereby saving time
    - ***memoization***

Minimum Edit Distance

- **Levenshtein (1st Version)**
- D(i,j) = score of **best** alignment from $s_1..s_i$ to $t_1..t_j$
- Min=

$$\min \left\{ \begin{array}{c} D(i-1,j-1)+d(si,tj) \ \textit{//substitute} \\ D(i-1,j)+1 \ \textit{//insert} \\ D(i,j-1)+1 \textit{//delete} \end{array} \right\}$$

j

i

		P	A	R	K
	0	1	2	3	4
S	1	1	2	3	4
P	2	1	2	3	4
A	3	2	1	2	3
K	4	3	2	2	2
E	5	4	3	3	3

Minimum Edit Distance

```
function MIN-EDIT-DISTANCE(target, source) returns min-distance

  n ← LENGTH(target)
  m ← LENGTH(source)
  Create a distance matrix distance[n+1,m+1]
  Initialize the zeroth row and column to be the distance from the empty string
    distance[0,0] = 0
    for each column i from 1 to n do
        distance[i,0] ← distance[i-1,0] + ins-cost(target[i])
    for each row j from 1 to m do
        distance[0,j] ← distance[0,j-1] + del-cost(source[j])
  for each column i from 1 to n do
    for each row j from 1 to m do
      distance[i, j] ← MIN( distance[i−1, j] + ins-cost(target_{i−1}),
                            distance[i−1, j−1] + sub-cost(source_{j−1}, ]target_{i−1}),
                            distance[i, j−1] + del-cost(source_{j−1}))
  return distance[n,m]
```


Figure 3.25 The minimum edit distance algorithm, an example of the class of dynamic programming algorithms. The various costs can either be fixed (e.g., $\forall x$, ins-cost$(x) = 1$) or can be specific to the letter (to model the fact that some letters are more likely to be inserted than others). We assume that there is no cost for substituting a letter for itself (i.e., sub-cost$(x,x) = 0$).

Minimum Edit Distance


```
function MIN-EDIT-DISTANCE(target, source) returns min-distance

n ← LENGTH(target)
m ← LENGTH(source)
Create a distance matrix distance[n+1,m+1]
Initialize the zeroth row and column to be the distance from the empty string
   distance[0,0] = 0
```

O(1) each

```
   for each column i from 1 to n do
      distance[i,0] ← distance[i-1,0] + ins-cost(target[i])
   for each row j from 1 to m do
      distance[0,j] ← distance[0,j-1] + del-cost(source[j])
```

O(n)+O(m)

```
for each column i from 1 to n do
   for each row j from 1 to m do
      distance[i, j] ← MIN( distance[i−1, j] + ins-cost(target_{i−1}),
                            distance[i−1, j−1] + sub-cost(source_{j−1}, ]target_{i−1}),
                            distance[i, j−1] + del-cost(source_{j−1}))
```

O(nm)

```
return distance[n,m]
```

Figure 3.25 The minimum edit distance algorithm, an example of the class of dynamic programming algorithms. The various costs can either be fixed (e.g., $\forall x$, ins-cost$(x) = 1$) or can be specific to the letter (to model the fact that some letters are more likely to be inserted than others). We assume that there is no cost for substituting a letter for itself (i.e., sub-cost$(x,x) = 0$).

Minimum Edit Distance

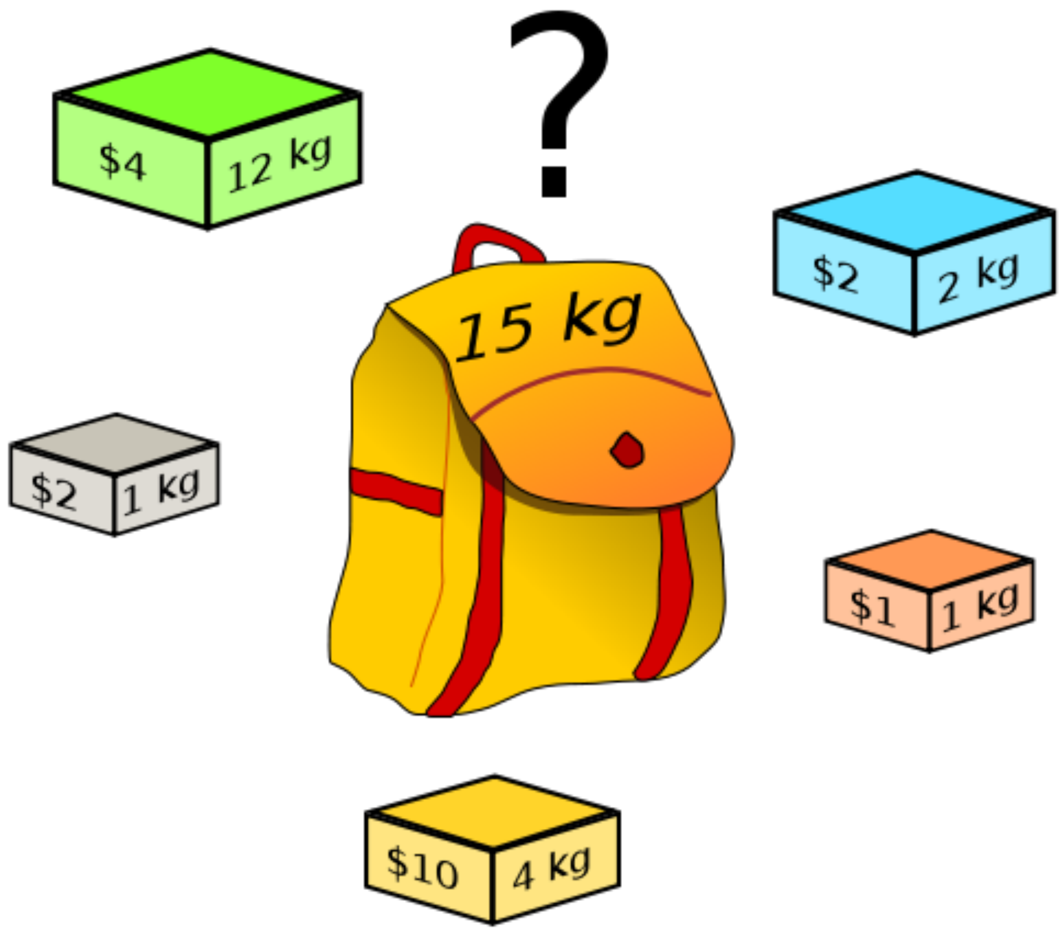
- **Levenshtein (2nd Version)**

$$distance[i,j] = \min \begin{cases} distance[i-1,j] + \text{ins-cost}(target_{i-1}) \\ distance[i-1,j-1] + \text{sub-cost}(source_{j-1}, target_{i-1}) \\ distance[i,j-1] + \text{del-cost}(source_{j-1}) \end{cases}$$

n	*9*	8	9	10	11	12	11	10	9	8
o	*8*	7	8	9	10	11	10	9	8	9
i	*7*	6	7	8	9	10	9	8	9	10
t	*6*	5	6	7	8	9	8	9	10	11
n	*5*	4	5	6	7	8	9	10	11	10
e	*4*	3	4	5	6	7	8	9	10	9
t	*3*	4	5	6	7	8	7	8	9	8
n	*2*	3	4	5	6	7	8	7	8	7
i	*1*	2	3	4	5	6	7	6	7	8
#	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
	#	**e**	**x**	**e**	**c**	**u**	**t**	**i**	**o**	**n**

Figure 3.26 Computation of minimum edit distance between *intention* and *execution* with the algorithm of Fig. 3.25, using Levenshtein distance with cost of 1 for insertions or deletions, 2 for substitutions. In italics are the initial values representing the distance from the empty string.

Knapsack Problem



Knapsack Problem

- Given some items(boxes), pack the knapsack to get the maximum total value (dollars). Each item has some weight (kg) and some value (dollars). Total weight that we can carry is no more than some fixed number W (15kg). So we must consider weights of items as well as their values.
- 3 Yellow, 3 Grey

Knapsack Problem

- Two versions of the problem:
  1. 0-1 knapsack problem
     - Items are indivisible; you either take an item or not. (Dynamic Approach)
  2. Fractional knapsack problem
     - Items are divisible: you can take any fraction of an item. (Greedy Approach)

0-1 Knapsack Problem

- Given a knapsack with maximum capacity *W*, and a set *S* consisting of *n* items
- Each item *i* has some weight w_i and benefit value b_i (all w_i and *W* are integer values)
- <u>Problem</u>: How to pack the knapsack to achieve maximum total value of packed items?

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

  – The problem is called a "*0-1*" problem, because each item must be entirely accepted or rejected.

0-1 Knapsack Problem

- Brute-force approach:
  - For *n* items, there are 2^n possible combinations.
  - Go through all combinations and find the one with maximum value and with total weight ≤ *W*
  - Running time will be $O(2^n)$
- Dynamic programming approach:
  - Can do better using dynamic programming by identifying the sub-problems.
  - Let's try this:
    If items are labeled *1..n*, then a subproblem would be to find an optimal solution for
    Sk = {items labeled 1, 2, .. k}

Defining a Subproblem

- If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_k = *{items labelled 1, 2, .. k}*
- This is a reasonable subproblem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- Unfortunately, we can't do that.

Defining a Subproblem

$w_1=2$ $b_1=3$	$w_2=4$ $b_2=5$	$w_3=5$ $b_3=8$	$w_4=3$ $b_4=4$	

?

Max weight: W = 20

For S_4:

Total weight: 14;

Maximum benefit: 20

$w_1=2$ $b_1=3$	$w_2=4$ $b_2=5$	$w_3=5$ $b_3=8$	$w_5=9$ $b_5=10$	

For S_5:

Total weight: 20

Maximum benefit: 26

Item #	Weight w_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

S_4

S_5

Solution for S_4 is not part of the solution for S_5!!!

Defining a Subproblem

- As we have seen, the solution for S_4 is not part of the solution for S_5
- So our definition of a subproblem is flawed and we need another one!
- Let's add another parameter: w, which will represent the exact weight for each subset of items
- The subproblem then will be to compute $B[k,w]$

Defining a Subproblem

Recursive formula for subproblems:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

It means, that the best subset of S_k that has total weight w is:

1) the best subset of S_{k-1} that has total weight w, **or**
2) the best subset of S_{k-1} that has total weight w-w_k plus the item k

Defining a Subproblem

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight w, either contains item k or not.
- First case: $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable.
- Second case: $w_k \leq w$. Then the item k <u>can</u> be in the solution, and we choose the case with greater value.

0-1 Knapsack Problem

```
for w = 0 to W
    B[0,w] = 0
for i = 1 to n
    B[i,0] = 0
for i = 1 to n
    for w = 0 to W
        if w_i <= w // item i can be part of the solution
            if b_i + B[i-1,w-w_i] > B[i-1,w]
                B[i,w] = b_i + B[i-1,w- w_i]
            else
                B[i,w] = B[i-1,w]
        else B[i,w] = B[i-1,w]  // w_i > w
```

Running Time is $O(nW)$, while the brute force $O(2^n)$

0-1 Knapsack Problem

- Let's run our algorithm on the following data:
- $n = 4$ (# of elements)
  $W = 5$ (max weight)
- Elements (weight, benefit):
  - (2,3), (3,4), (4,5), (5,6)

0-1 Knapsack Problem

- Let's run our algorithm on the following data:
- n = 4 (# of elements)
  W = 5 (max weight)
- Elements (weight, benefit)
  – (2,3), (3,4), (4,5), (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

```
for w = 0 to W
      B[0,w] = 0
```

0-1 Knapsack Problem

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

for i = 1 to n

    B[i,0] = 0

0-1 Knapsack Problem

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i=-1$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

0-1 Knapsack Problem

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	**3**			
2	0					
3	0					
4	0					

i=1

b_i=3

w_i=2

w=2

$w-w_i$ =0

if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

0-1 Knapsack Problem

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	**3**		
2	0					
3	0					
4	0					

$i=1$
$b_i=3$
$w_i=2$
$w=3$
$w-w_i=1$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

0-1 Knapsack Problem

Items:

1: (2,3)

2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	**3**	
2	0					
3	0					
4	0					

$i=1$
$b_i=3$
$w_i=2$
$w=4$
$w-w_i=2$

if w_i <= w // item i can be part of the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $\mathbf{B[i,w] = b_i + B[i-1,w-w_i]}$
    else
        $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

0-1 Knapsack Problem

Items:

$\boxed{1: (2,3)}$

2: (3,4)

3: (4,5)

4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	**3**
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=3$

if w_i <= w // item i can be part of the solution

  if $b_i + B[i-1,w-w_i] > B[i-1,w]$

    $\mathbf{B[i,w] = b_i + B[i-1,w-w_i]}$

  else

    $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

0-1 Knapsack Problem

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	**7**

$i=4$

$b_i=6$

$w_i=5$

$w= 5$

$w- w_i=0$

if w_i <= w // item i can be part of the solution

    if b_i + B[i-1,w-w_i] > B[i-1,w]

        B[i,w] = b_i + B[i-1,w- w_i]

    else

        B[i,w] = B[i-1,w]

else B[i,w] = B[i-1,w] // w_i > w

0-1 Knapsack Problem

- This algorithm only finds the max possible value that can be carried in the knapsack
- » I.e., the value in B[n,W]
- To know the items that make this maximum value, an addition to this algorithm is necessary.

How to find actual Knapsack Items

- All of the information we need is in the table.
- $B[n,W]$ is the maximal value of items that can be placed in the Knapsack.
- Let i=n and k=W

  if $B[i,k] \neq B[i-1,k]$ then

  mark the i_{th} item as in the knapsack

  $i = i-1,\ k = k-w_i$

  else

  $i = i-1$ // Assume the i_{th} item is not in the knapsack

  // Could it be in the optimally packed knapsack?

Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=4
k= 5
b_i=6
w_i=5
$B[i,k] = 7$
$B[i-1,k] = 7$

```
i=n, k=W
while i,k > 0
    if B[i,k] ≠ B[i−1,k] then
        mark the i^th item as in the knapsack
        i = i−1, k = k-w_i
    else
        i = i−1
```

40

Finding the Items (2)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i=4$
$k= 5$
$b_i=6$
$w_i=5$
$B[i,k] = 7$
$B[i-1,k] =7$

```
i=n, k=W
while i,k > 0
    if B[i,k] ≠ B[i−1,k] then
        mark the i^th item as in the knapsack
        i = i−1, k = k-w_i
    else
        i = i−1
```

Finding the Items (3)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=3

k= 5

b_i=6

w_i=4

$B[i,k] = 7$

$B[i-1,k] = 7$

i=n, k=W

while i,k > 0

  if $B[i,k] \neq B[i-1,k]$ then

    mark the i^{th} item as in the knapsack

    $i = i-1$, $k = k-w_i$

  else

    $\boldsymbol{i = i-1}$

Finding the Items (4)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=2

k= 5

b_i=4

w_i=3

$B[i,k] = 7$

$B[i-1,k] = 3$

$k - w_i$=2

i=n, k=W
while i,k > 0
    if $B[i,k] \neq B[i-1,k]$ then
        mark the i^{th} item as in the knapsack
        $i = i-1, k = k-w_i$
    else
        i = i–1

Finding the Items (5)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=1

k= 2

b_i=3

w_i=2

$B[i,k] = 3$

$B[i-1,k] = 0$

$k - w_i = 0$

i=n, k=W

while i,k > 0

if $B[i,k] \neq B[i-1,k]$ then

mark the i^{th} item as in the knapsack

$i = i-1, k = k-w_i$

else

$\boldsymbol{i = i-1}$

Finding the Items (6)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=0

k= 0

The optimal knapsack should contain {1, 2}

i=n, k=W
while i,k > 0
  if $B[i,k] \neq B[i-1,k]$ then
    mark the n^{th} item as in the knapsack
    $i = i-1$, $k = k-w_i$
  else
    $i = i-1$

Fractional Knapsack Problem

- We have n objects and a knapsack. The i^{th} object has positive weight w_i and positive unit value v_i. The knapsack capacity is C.
- We wish to select a set of proportions of objects to put in the knapsack so that the total values is maximum and without breaking the knapsack.

Fractional Knapsack Problem

Greedy-fractional-knapsack (*w, v, W*)

```
FOR i =1 to n
   do x[i] =0
weight = 0
while weight < W
   do i = best remaining item
      IF weight + w[i] ≤ W
         then x[i] = 1
            weight = weight + w[i]
         else
            x[i] = (W - weight) / w[i]
            weight = W
return x
```

Fractional Knapsack Problem

- **`Example:`**

Input: 5 objects, $C = 100$

w	10	20	30	40	50
v	20	30	66	40	60

- Select always the most valuable object

object	1	2	3	4	5
selected	0	0	1	0.5	1

  – Total selected weight 100 and total value 146.

- Select always the lighter object

object	1	2	3	4	5
selected	1	1	1	1	0

  – Total selected weight 100 and total value 156.

Fractional Knapsack Problem

- Select always the object with highest ratio value/weight

Input: 5 objects, $C = 100$

w	10	20	30	40	50
v	20	30	66	40	60

  – Total selected weight 100 and total value 164.

object	1	2	3	4	5
ratio	2.0	1.5	2.2	1.0	1.2
selected	1	1	1	0	0.8

Fractional Knapsack Problem

- The greedy algorithm that always selects the most valuable object does not always find an optimal solution to the Fractional Knapsack problem.
- The greedy algorithm that always selects the lighter object does not always find an optimal solution to the Fractional Knapsack problem.
- The greedy algorithm that always selects the object with better ratio value/weight always finds an optimal solution to the Fractional Knapsack problem.

Homework # 7

16.2-6 ★

Show how to solve the fractional knapsack problem in $O(n)$ time.

16.2-7

Suppose you are given two sets A and B, each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the ith element of set A, and let b_i be the ith element of set B. You then receive a payoff of $\prod_{i=1}^{n} {a_i}^{b_i}$. Give an algorithm that will maximize your payoff. Prove that your algorithm maximizes the payoff, and state its running time.

2.5 Figure out whether *drive* is closer to *brief* or to *divers* and what the edit distance is to each. You may use any version of *distance* that you like.

2.7 Augment the minimum edit distance algorithm to output an alignment; you will need to store pointers and add a stage to compute the backtrace.