Greedy

Plan

- Paradigm
- Knapsack problem
- Job scheduling problem
- Huffman codes

Optimization problems

- Maximizing or minimizing some function relative to some set, often representing a range of choices available in a certain situation.
  - Finding the *best* solution from all feasible solutions.
- Common applications: Minimal cost, maximal profit, minimal error, optimal management.

Greedy method

- Local improvement method
  - Does not look at problem globally
  - Takes best immediate step to find a solution
  - Useful in many cases where
    - Objectives or constraints are uncertain
    - An approximate answer is all that's required
  - In some cases, greedy algorithms provide optimal solutions (shortest paths, spanning trees, some job scheduling problems)
    - In most cases they are approximate algorithms
    - Sometimes used as a part of an exact algorithm (e.g., as a relaxation in an integer programming algorithm)

General greedy algorithm

```
void Greedy(S, I){
      // I: set of instances
      // A: solution
  S = Φ ;
  while (I != Φ ){
      x ← select(I);
      I ← I - {x};
      if (S ∪ {x} is acceptable)
         S ← S + {x}
  }
}
```

Plan

- Paradigm
- Knapsack problem
- Job scheduling problem
- Huffman code

Knapsack problem

- There are n objects, each with weight w_i and profit p_i. The knapsack has capacity M

$$\max \sum_{0 \le i \le n} p_i x_i$$

$$0 \le x_i \le 1$$

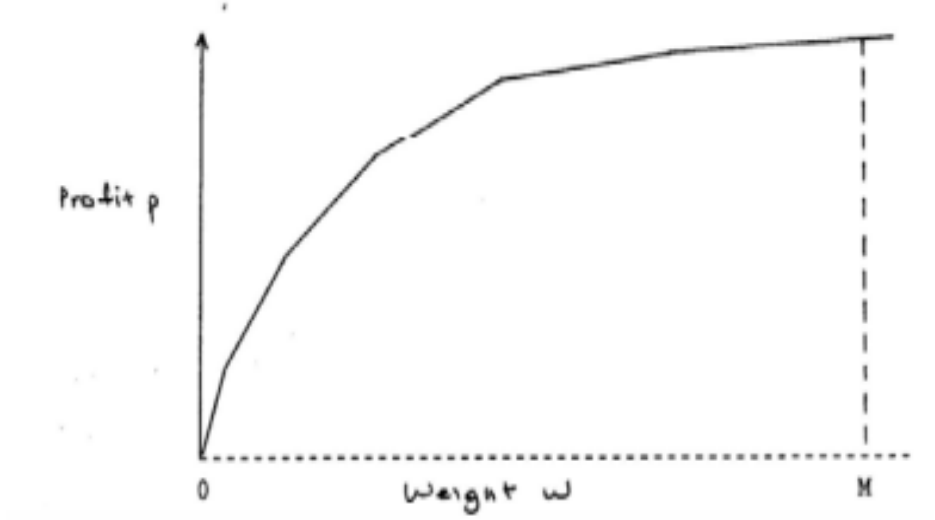
$$s.t$$

$$\sum_{0 \le i \le n} w_i x_i \le M$$

$$p_i \ge 0, w_i \ge 0, 0 \le i < n$$

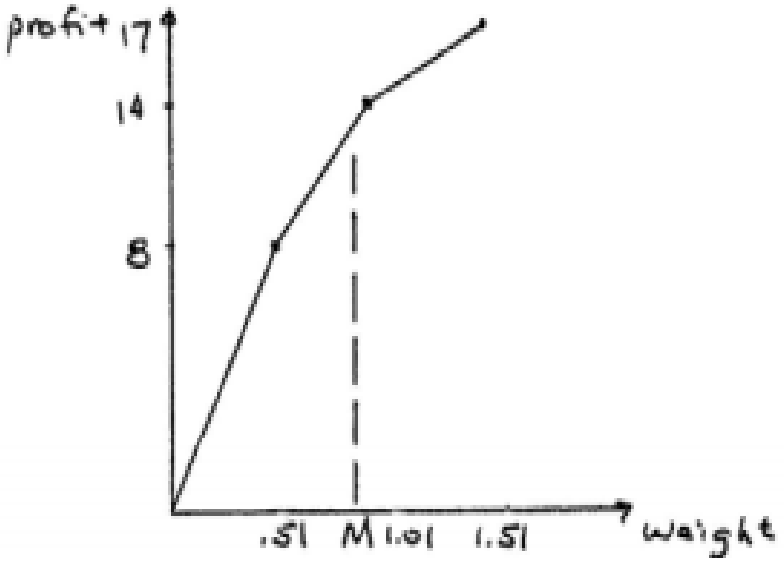
Greedy knapsack algorithm

- Algorithm chooses element with highest profit/weight ratio first, the next highest second, and so on until it reaches the capacity of the knapsack.
- This is the same as a gradient or derivative method.



Knapsack: Integer or fraction?

Item	Weight	Profit
1	0.51	8
2	0.5	6
3	0.5	3



- Let M= 1.
- Integer solution is {2, 3}, an unexpected result in some contexts.
- Fractional greedy solution is {1, 98% of 2}.
- If problem has hard constraints, need integer solution.
- If constraints are fuzzy, greedy solution may be better.

Knapsack: fractional algorithm

The knapsack has capacity M

Item	Weight	Profit	Ratio
1	w_1	p_1	p_1/w_1
	…	…	
i	w_i	p_i	p_i/w_i

- Input
  - Item[]: an array of items
  - M: capacity of the knapsack
- Output
  - x[]: answer array, 1 if item in knapsack, 0 if not ($0<=x_i<=1$)
  - totalProfit: total of profit

Knapsack: fractional algorithm

```
double solve(Item e[], int M, double x[]){
   upper = M;
   Array.sort(e); // Sort according to ratio
   for (i= 0; i < e.length; i++) {
      if (e[i].weight > upper) break;
      x[i]= 1.0;
      upper -= e[i].weight;
   }
   if (i < e.length) // If all items not in knapsack
      x[i]= (double) upper/ e[i].weight; // Fractional item

   totalProfit= 0.0;
   for (i= 0; i < e.length; i++)
      totalProfit += x[i]*e[i].weight*e[i].ratio;
   return totalProfit;
}
```

T(n) = O(nlogn)

Plan

- Paradigm
- Knapsack problem
- Job scheduling problem
- Huffman codes

Job scheduling problem

- There are n jobs to be run on a processor (CPU)
- Each job i has a deadline d_i >=1 and profit p_i >=0
- Each job takes 1 unit of time (simplification)
- We earn the profit if and only if the job is completed by its deadline

→ Find the subset of jobs that maximizes the profit

Example

- Number of jobs n=5. Time slots 1, 2, 3 (Slot 0 is sentinel)

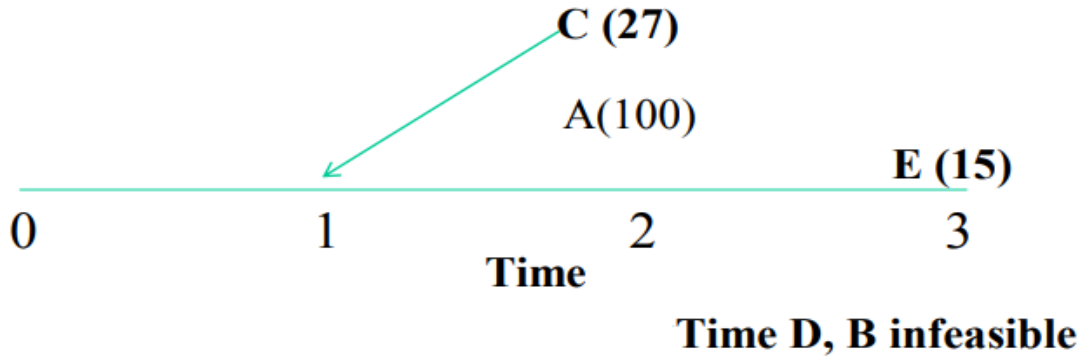
Job	Profit	Deadline	Profit/Time
A	100	2	100
B	19	1	19
C	27	2	27
D	25	1	25
E	15	3	15

Example

- A feasible solution is a subset of jobs j such that each job is completed by its deadline
- An optimal solution is a feasible solution with maximum profit value

Greedy job scheduling algorithm

- Sort jobs by profit/time ratio (slope or derivative):
  - A (deadline 2), C (2), D (1), B (1), E (3)
- Place each job at latest time that meets its deadline
  - Nothing is gained by scheduling it earlier, and scheduling it earlier could prevent another more profitable job from being done
  - Solution is {C, A, E} with profit of 142



Greedy job scheduling algorithm

- **Step 1**: Sort p_i into nonincreasing order. After sorting $p_1 >= p_2 >= \ldots >= p_n$
- **Step 2**: Add the next job i to the solution set if i can be completed by its deadline. Assign i to time slot [r-1, r], where r is the largest integer such that $1<=r<=d_i$ and [r-1, r] is free
- **Step 3**: Stop if all jobs are examined. Otherwise, go to step 2

→ $T(n) = O(n^2)$

Disjoint set for job scheduling

- Let the max deadline be m. We create m+1 individual sets.
- If a job is assigned a time slot of t where t >= 0, then the job is scheduled during [t-1, t].
  - A set with value X represents the time slot [X-1, X].
- A parent array of Disjoint Set data structure to keep track of the greatest time slot available which can be allotted to a given job having deadline.

Disjoint set for job scheduling

Operator

- **Init()**

```
parent = new int[M + 1];
for (int i = 0; i ≤ M; i++)
    parent[i] = -1;
```

- **Find()**

```
find(s) {
    // Base case
    if (parent[s]==-1) return s;
    //  Recursive  call  with  path  compression
    return parent[s] = find(parent[s]);
}
```

Disjoint set for job scheduling

Operator

- **Union()** - Merges two sets, makes u as parent of v.

```
union(u, v) {
  // update the greatest available free slot to u
  parent[v] = u;
}
```

Algorithm

```
sort(arr, nJobs);

int maxDeadline = findMaxDeadline(arr, nJobs);
DisjointSet ds(maxDeadline);

for (i = 0; i < nJobs; i++) {
  int availableSlot = ds.find(arr[i].deadLine);
  if (availableSlot > 0) {
        ds.merge(ds.find(availableSlot - 1),
                                availableSlot);
        print arr[i].id;
  }
```

→ **T(n) = O(nlogn)**

Disjoint set for job scheduling

Example

- Number of jobs n=5. Time slots 1, 2, 3 (Slot 0 is sentinel)

Job	Profit	Deadline	Profit/Time
A	100	2	100
B	19	1	19
C	27	2	27
D	25	1	25
E	15	3	15

Disjoint set for job scheduling

Example

Sorting J_A, J_C, J_D, J_B, J_E

Init

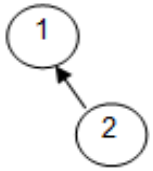
-1	-1	-1	-1
0	1	2	3

Slot

0 1 2 3

J_A: J_A.Deadline = 2

-1	-1	1	-1
0	1	2	3

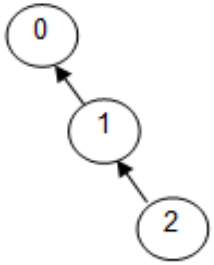


J_A

0 1 2 3

J_C: J_C.Deadline = 2

-1	0	1	-1
0	1	2	3



J_C J_A

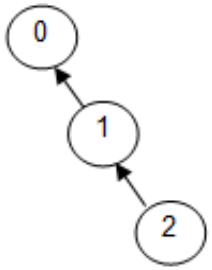
0 1 2 3

Disjoint set for job scheduling

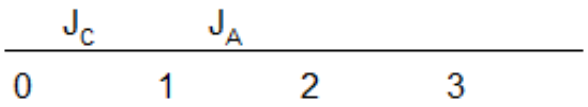
Example

J_D: J_D.Deadline = 1

-1	0	1	-1
0	1	2	3

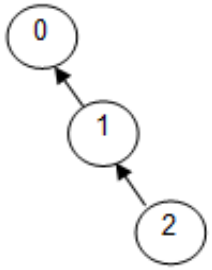


Not choose: J_D

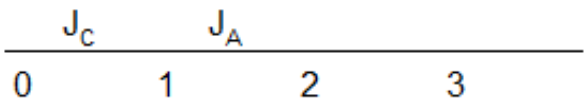


J_B: J_B.Deadline = 1

-1	0	1	-1
0	1	2	3

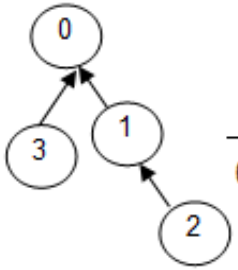


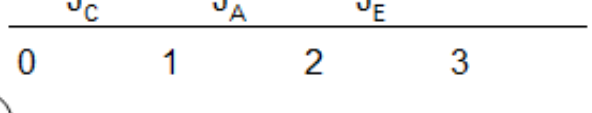
Not choose: J_B



J_E: J_E.Deadline = 3

-1	0	1	0
0	1	2	3





Plan

- Paradigm
- Knapsack problem
- Job scheduling problem
- Huffman codes

Huffman codes

- Suppose we have a 100,000-character data file that we wish to store compactly.
  - There are only 6 different characters appear, and the character a occurs 45,000 times.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- A fixed-length code requires 300,000 bits to code the entire file.
- Can we do better?

Variable length code

- Gives frequent characters short codewords and infrequent characters long codewords
- Variable length code requires

  (45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4) * 1000 = 224,000 bits

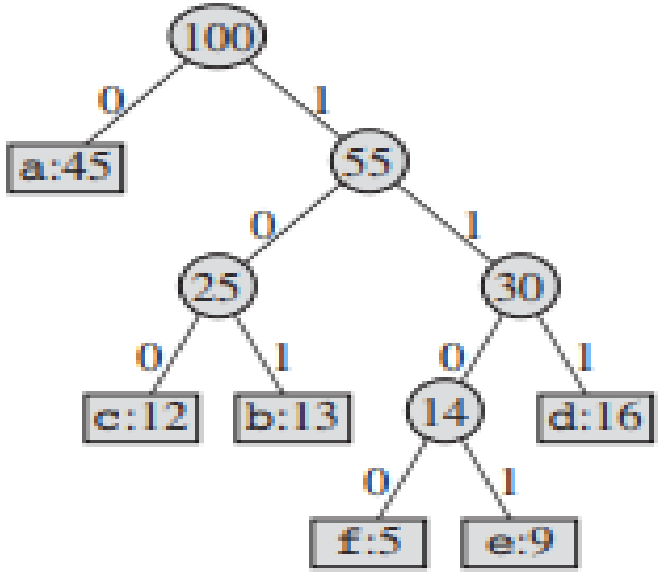
  → Savings of approximately 25%

Prefix code

- No codeword is also a prefix of some other codeword.
- A prefix code can always achieve the optimal data compression
- Encoding: concatenate the codewords representing each character of the file.
  - The 3-character file **abc** as **0**101**100**
- Decoding: simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file
  - The string 001011101 parses uniquely as 0 0 101 1101 → aabe

Huffman tree

- Represent optimal codes
- Leaves are the given characters
- The binary codeword for a character is the simple path from the root to that character, where 0 means “go to the left child” and 1 means “go to the right child.”



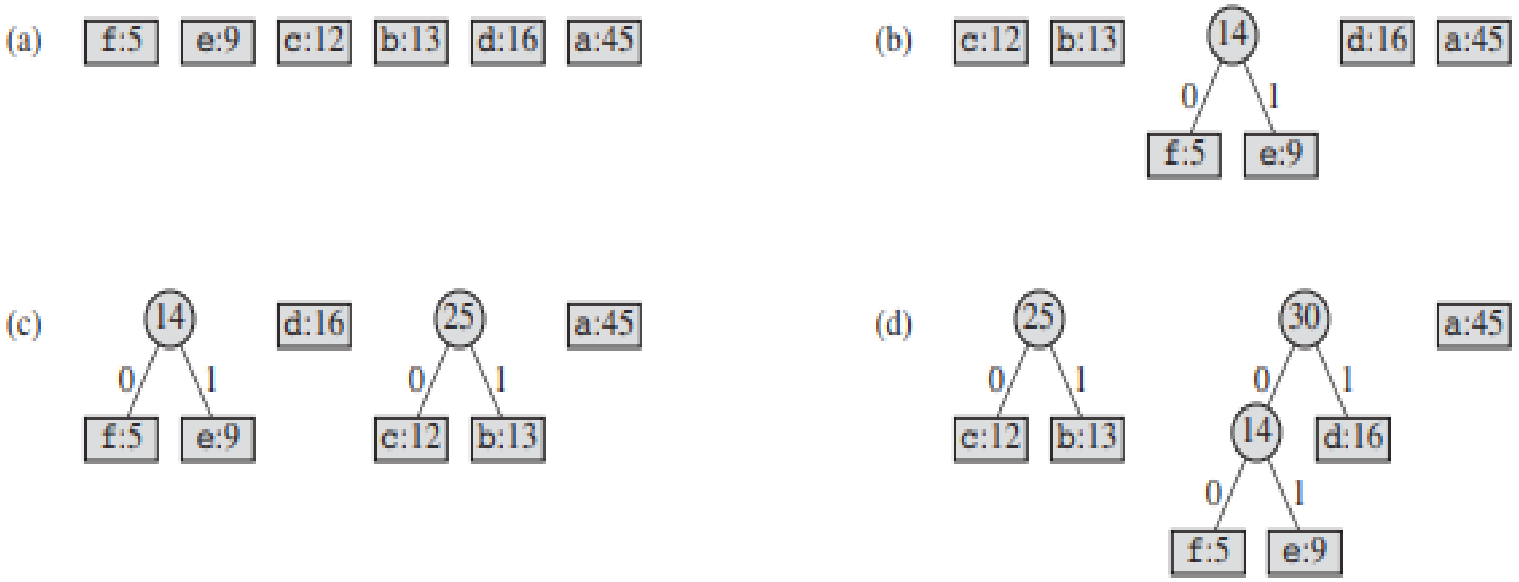
Huffman algorithm

- A greedy algorithm to construct optimal prefix codes
- Input
  - C: set of character, each has its frequency *fred*
- Output
  - Tree T consisting of optimal codes
- Idea
  - Starting with a set of |C| leaves and performs a sequence of |C| - 1 “merging” operations to create the final tree.
  - A min-priority queue Q, key on the *freq* attribute, to identify the two least-frequent objects to merge together.
    - The result is a new object whose frequency is the sum of the frequencies of the two objects that were merged

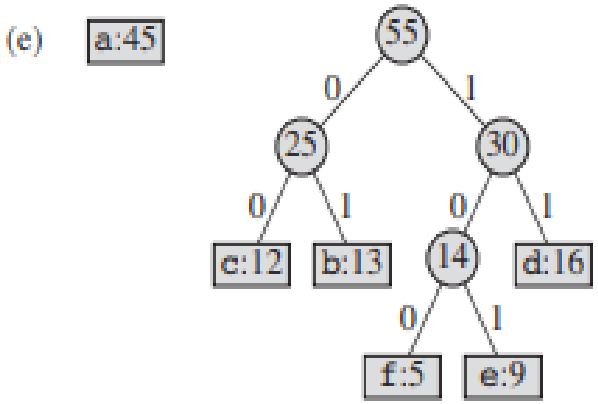
Huffman algorithm

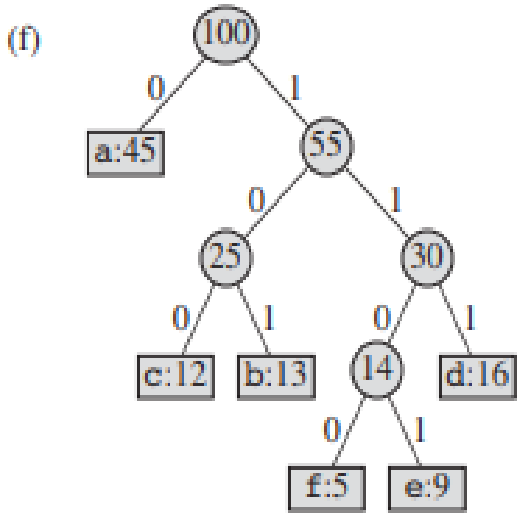
```
Huffman(C){
   n = |C|
   Q = C
   for(i=1;i<=n;i++){
       //Allocate a new node z
       z.left = x = EXTRACT_MIN(Q)
       z.right = y = EXTRACT_MIN(Q)
       z.fred = x.fred + y.fred
       INSERT(Q, z)
   }
   return EXTRACT_MIN(Q)  // The root of the tree
}
```

Example


(a)
f:5
e:9
c:12
b:13
d:16
a:45
(b)
c:12
b:13
14
0
1
f:5
e:9
d:16
a:45
(c)
14
0
1
f:5
e:9
d:16
25
0
1
c:12
b:13
a:45
(d)
25
0
1
c:12
b:13
30
0
1
14
d:16
0
1
f:5
e:9
a:45

Example


(e)
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9


(f)
100
0
1
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9

Analysis

- Q: binary min-heap
  - Initialize Q: O(n)
  - The loop O(n log n)
    - The for loop executes exactly n - 1 times
    - The heap operation requires time O(log n)
- T(n) = O(nlogn)

Thanks for your attention!