CSC3 73 Week 2 Notes

Greedy Algorithm:

1. Introduction:

- With greedy algorithms, you want to get the piece with the most immediate benefit at each step.

- Note: You can't go back.
  I.e. After you make a choice, it's final.

- E.g. Suppose we have coins of 1, 7, and 10 denomination and we want to make \$18 with as little coins as possible.

  Using a greedy algorithm, we would first choose the \$10 coin, then \$7 and then \$1.

  However, if we want to make \$15, then we run into a problem. We first choose a \$10 coin, so we have \$4 left over. That means we have to use 4 \$1 coins.

  \$10, \$1, \$1, \$1, \$1 ⇒ \$15

  However, we can make \$15 from 2 \$7 coins and 1 \$1 coin, using 3 coins instead of 5.

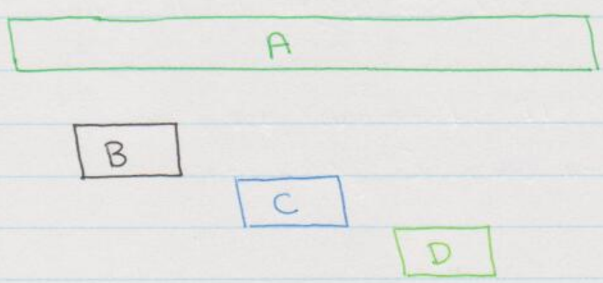
2. Interval Scheduling:

- Problem: We have a list of jobs and each job has a start time and a finish time.
  E.g. For job J, S_J denotes its start time while F_J denotes its end time.
  2 jobs, I and J, are compatible if $[S_i, F_i)$ and $[S_j$ and $F_j)$ don't overlap. (We allow a job to start right away when another finishes.) We want to find the maximum number of mutually compatible jobs.

- Here are a few ways we can order the jobs:
  1. Earliest start time: Ascending order of S_j.
  2. Earliest Finish time: Ascending order of F_j.
  3. Shortest Interval: Ascending order of $f_j - s_j$.
  4. Fewest conflicts: Ascending order of c_j, where c_j is the number of remaining jobs that conflict with j.

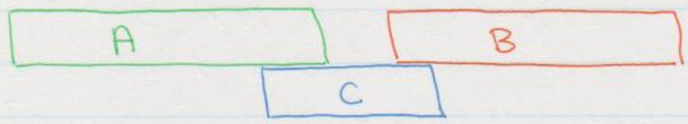
However, out of the 4 ways above, only "Earliest Finish Time" works. Here are some counterexamples.

1. For "Earliest Start Time", consider this:



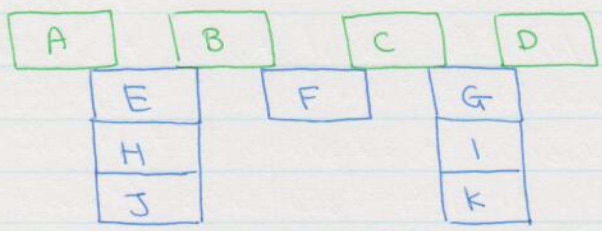
Notice how even though job A starts the earliest, it blocks 3 other jobs.

2. For "Shortest Interval"



Notice how even though C has the shortest interval, it's blocking 2 other jobs.

3. For "Fewest Conflicts"



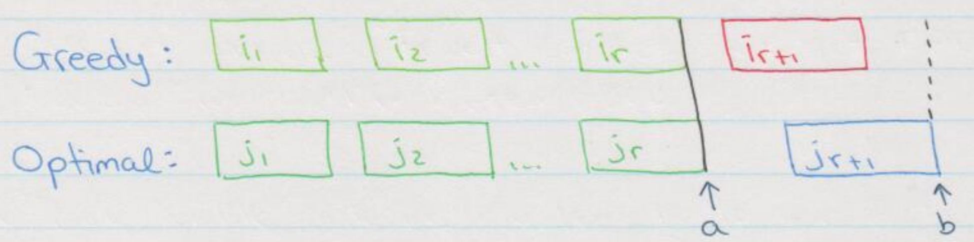
Here, if we use "Fewest conflicts", we get FAD. However, we could've gotten ABCD.

- The only viable ordering system is to use earliest finish time.

  Sorting will take $O(n \lg n)$.
  For each job j, we only need to check if it's compatible with the end time of the last added job. We can perform each check in $O(1)$.

  $\therefore$ The overall running time is $O(n \lg n)$.

- Proof of Optimality by Contradiction:
  - Suppose for contradiction that greedy soln is not optimal.
  - Say the greedy algo selects jobs $i_1, i_2, \ldots, i_k$ sorted by finish time.
  - Consider an optimal soln $J_1, J_2, \ldots, J_m$ also sorted by finish time and matches the greedy soln for as many indices as possible.
    I.e. We want $i_1 = j_1, \ldots, i_r = j_r$ for the greatest possible value of r.
  - We know both i_{r+1} and j_{r+1} must be compatible with the prev selection.



We know that both i_{r+1} and j_{r+1} must be between points a and b and we also know that i_{r+1} must end before J_{r+1}. This is bc we used the greedy algo to get i_{r+1}.

- Suppose we switch jobs i_x and J_x for $1 \leq x \leq r+1$.
  I.e. We get this new soln: $i_1, i_2, \ldots, i_r, i_{r+1}, J_{r+2}, \ldots, J_m$
  This is still feasible because $f_{i_{r+1}} \leq f_{j_{r+1}} \leq S_{j_t}$ for $t \geq 2$.

  This is still optimal cause m jobs are selected, but it matches the greedy soln in $r+1$ indices.

- Proof of Optimality by Induction:
  - We will define S_j to be the subset of jobs picked by the greedy algo.

    Note: $S_0 = \emptyset$

  - We call this partial soln **promising** if there is a way to extend it to an optimal soln by picking some subset of jobs $j+1, \dots, n$.
    I.e. $\exists\, t \subseteq \{j+1, \dots, n\}$ s.t. $O_j = S_j \cup t$ is optimal.

  - WTP: $\forall t \in \{0, \dots, n\}$, S_t is promising.

  - Proof:

    Base Case ($t=0$):
    Let $t=0$.
    $S_t = \emptyset$ and is promising bc any optimal soln extends it.

    Induction Hypothesis:
    Suppose the claim holds for $t=j-1$ and optimal soln O_{j-1} extends S_{j-1}.

    Induction Step:
    At $t=j$, we have 2 possibilities:

    1. Greedy did not select job j so $S_j = S_{j-1}$.
       Job j must conflict with some job in S_{j-1}.
       Since $S_{j-1} \subseteq O_{j-1}$, it also cannot include job j.
       $O_j = O_{j-1}$ extends $S_j = S_{j-1}$.

    2. Greedy selects job j.
       $S_j = S_{j-1} \cup \{j\}$
       Consider the earliest job r between S_{j-1} and O_{j-1}.
       Consider O_j obtained by replacing r with j in O_{j-1}.
       O_j is still feasible and extends S_j as desired.

3. Interval Partitioning Problem:

- Problem: Job j starts at s_j and finishes at f_j. 2 jobs are compatible if they don't overlap. The goal is to group the jobs into the fewest partitions s.t. jobs in the same partition don't overlap (I.e. They're compatible)

- We'll be ordering the jobs based on their earliest start time.

- Pseudo-Code:

```
def partition (s1, s2, ..., sn , f1, ..., fn):
    Sort the jobs by start time s.t. s1 ≤ s2 ≤ ... ≤ sn.
    p=0 ← Number of partitions

    for j=1 to n:
        if job j is compatible with some partition:
            put job j in that partition
        else:
            create a new partition, p+1, and put job j
            in there
            p=p+1

    return p
```

- Running Time
  - Sorting will take $O(n\lg n)$.
  - We can use a priority queue to store the end times of each partition. We will do n compares and each compare is $\lg n$, so in total, we have $O(n\lg n)$.
  - $\therefore$ The total running time complexity is $O(n\lg n)$.

- Proof of Optimality:
  - Let d be the # of partitions used by the greedy algo.
  - Let depth be the max num of jobs running at any time.

1. Lower Bound:
   $d \geq$ depth (Have at least 1 partition per job)

2. Upper Bound:
   Partition d was opened bc there's a job j that's incompatible with some job in the other $d-1$ partitions. This means that these jobs end after S_j. However, bc we're sorting by start time, we know that they start before or at S_j.

   Hence, at time S_j, there are d overlapping jobs. This means that depth $\geq d$.

   Since we have $d \geq$ depth and depth $\geq d$, depth $= d$.

   $\therefore$ The greedy algo uses exactly as many partitions as the depth.

4. Minimizing Lateness:

- Problem: We have a single machine. Each job j requires t_j units of time to complete and is due by d_j. If it's scheduled to start at s_j, it will finish at $f_j = s_j + t_j$. The lateness of a job is $\ell_j = \max\{0, f_j - d_j\}$. The goal is to minimize the max lateness.

- We'll sort the jobs in ascending order of due time.

- Pseudo-Code:

```
def EarliestDueFirst(n, t1, ..., tn, d1, ..., dn):
    Sort the jobs in ascending order of due time.
    I.e. d1 ≤ d2 ≤ ... ≤ dn

    t = 0

    for j = 1 to n:
        Assign job j to interval [t, t + tj]
        sj = t
        fj = t + tj
        t = t + tj

    return [s1, f1], [s2, f2], ..., [sn, fn]
```

- Some observations:
  1. There's an optimal schedule with no idle time
  2. The job with the earliest deadline has no idle time.

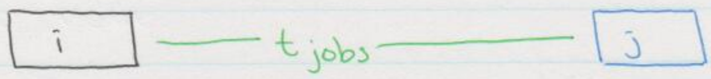
Let an inversion be (i, j) s.t. $d_i < d_j$ but j is scheduled before i.
I.e. Job i is due before job j but job j is scheduled before job i.

3. The earliest deadline algo has no inversions.

4. If a schedule with no idle time has at least 1 inversion, then it has a pair of inverted jobs scheduled consecutively.

Proof:

Let jobs i and j be inverted and suppose they are the only 2 inverted jobs and that there are t jobs in between.



Job j is due before i but is scheduled after i.

Let's consider job i+1.

There are 2 possibilities:

1. Job i+1 is due before Job j. 2
   In this case, we now have inversions (i, i+1) and (i, j) which contradicts our assumption.

2. Job i+1 is due after Job j.
   In this case, we also get more than 1 inversion, which contradicts our assumption.

∴ i and j must be together.

5. Swapping adj scheduled inverted jobs doesn't increase lateness but it does reduce the num of inversions by 1.

Proof:

1. Reducing the num of inversions by 1 is easy to see.
   Suppose j and i are inverted, meaning that j is due before i but scheduled after. By switching them, j is now scheduled before i.

2. Let l_k and l'_k denote the lateness of job k before and after swap.

Let $L = \max_k l_k$ and $L' = \max_k l'_k$.

We know that:

1. $l_k = l'_k \quad \forall k \neq i, j$
2. $l'_i \leq l_i$ (Since i is moved early)
3. $$\begin{aligned} l'_j &= f'_j - d_j \\ &= f_i - d_j \\ &\leq f_i - d_i \\ &= l_i \end{aligned}$$

$$\begin{aligned} \therefore L' &= \max \{ l'_i, l'_j, \max_{k \neq i,j} l'_k \} \\ &\leq \max \{ l_i, l_i, \max_{k \neq i,j} l'_k \} \\ &\leq L \end{aligned}$$

- Proof of Optimality (By contradiction):
  - Suppose for contradiction that the greedy soln is not optimal.
  - Let S^* be an optimal soln with the fewest inversions. Assume, without loss of generality (wlog) that there is no idle time.
  - Because the greedy soln isn't optimal, there's at least 1 inversion in S^*.
  - By observation 4, there's an adjacent inversion.
  - By observation 5, we can swap the inversions to decrease the num of inversions by 1.
    This is a contradiction.

5. Lossless Compression

- Problem: We have a document that is written in n distinct labels, and we want to compress this without losing any info.

- Naive Soln: Represent each label using $\log_2(n)$ bits. If the doc has length m, this use $m\log_2 n$ bits.

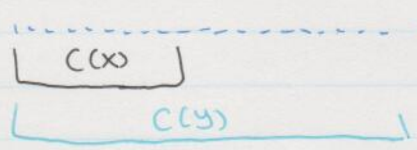
- E.g. Consider a doc that only contains the 26 English letters.

  There are 26 letters, so we need $\lceil \log_2 26 \rceil$ or 5 bits per letter.

  $$\left.\begin{aligned} a &= 00000 \\ b &= 00001 \\ c &= 00010 \\ d &= 00011 \end{aligned}\right\} \text{Not optimal}$$

- What if some letters, such as a, e, r, s, are more frequent than others, like x, q, z? We can use shorter codes for these frequent letters. However, we need to use prefix-free encoding to avoid conflicts.

  Prefix-free encoding will map each label x to a bit-string c(x) s.t. $\forall$ distinct labels x and y, c(x) is not a prefix of c(y). In this case we can never get a scenario like the one shown below:



← bits

Here, we don't know if the starting bits c(x) or the beginning of c(y).

- Given this new info, we can rewrite our original problem more formally.

  Formal Problem: Given n symbols and their frequencies $(w_1, \ldots, w_n)$ where the higher the num the more frequent they appear, find a prefix-free encoding with lengths $(\ell_1, \ldots, \ell_n)$ which minimizes $\sum_{i=1}^{n} w_i \cdot \ell_i$ where $\sum_{i=1}^{n} w_i \cdot \ell_i$ is the length of the compressed doc.

- We can use Huffman Coding Algorithm.

  Huffman Coding:

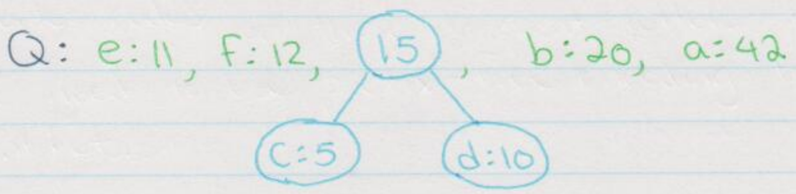
  1. Build a priority queue by adding (x, w_x) for each symbol x.

  2. While $|queue| \geq 2$
     a) Take the 2 symbols with the lowest weight (x, w_x) and (y, w_y).
     b) Merge them into 1 symbol with weight $w_x + w_y$.

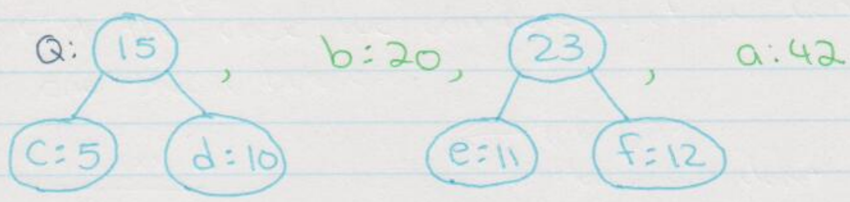
E.g. Suppose we have the following letters with their frequency.

Q: c: 5, d: 10, e: 11, f: 12, b: 20, a: 42

1. We'll merge C and D together.

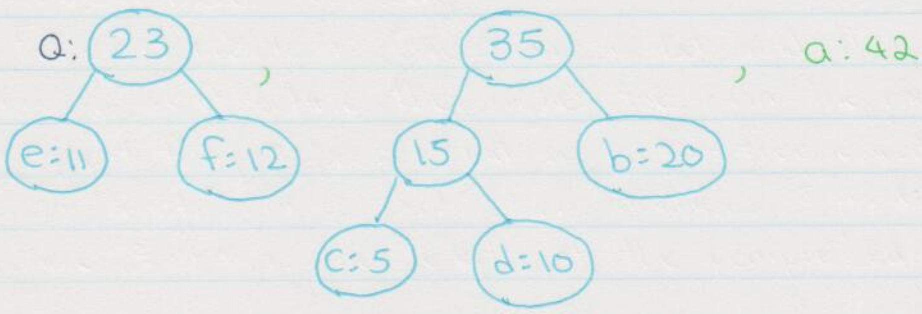


2. We'll merge e and f together.



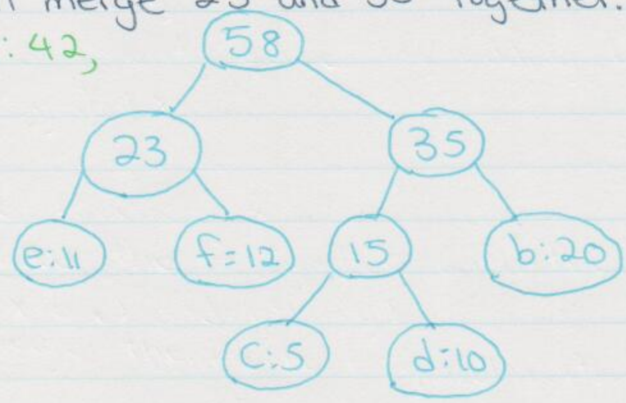
3. We'll merge 15 (c+d) and b together.

Q:



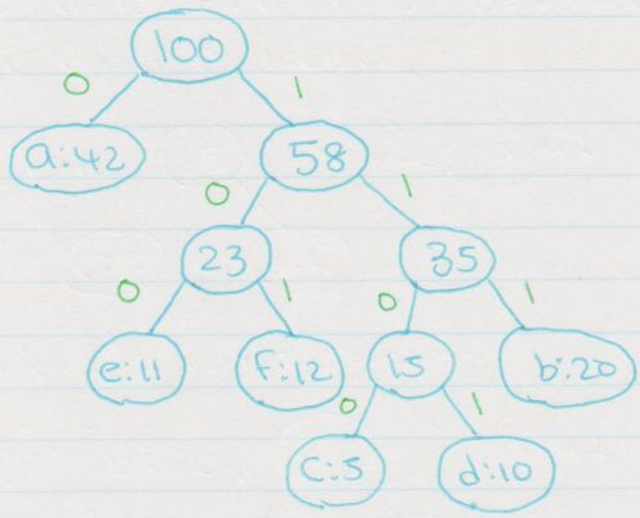
4. We'll merge 23 and 35 together.

Q: a:42,



5. We'll merge a and 58 together.

Q:



a=0, e=100, f=101, b=111, c=1100, d=1101

- Running Time:
  - If the labels are not sorted, then it takes $O(n\log n)$.
  - If the labels are sorted by their frequencies, then it takes $O(n)$. We can use 2 queues to achieve this.

- Proof of Optimality:
  We will use induction to prove this.

  Base Case:
  Let $n = 2$.
  In this case, we can just assign 1 bit to each symbol, which is optimal.

  Induction Hypothesis:
  Assume for some k s.t. $1 \le k < N$, that the algo returns an optimal encoding.

  Before we go to the induction step, here are a few lemmas that will help us.

  Lemma1: If $w_x < w_y$, then $l_x \ge l_y$ in any optimal tree.
  Proof:
  - Suppose for contradiction that $w_x < w_y$ and $l_x < l_y$.
  - Swapping x and y strictly reduces the overall length.

  $w_x \cdot l_y + w_y \cdot l_x < w_x \cdot l_x + w_y \cdot l_y$

  Comes from $(l_y - l_x) > 0$

  $w_x(l_y - l_x) < w_y(l_y - l_x)$ ← Multiply $l_y > l_x$ with $(l_y - l_x)$ on both sides

  $w_x \cdot l_y - w_x \cdot l_x < w_y l_y - w_y l_x$

  $w_x l_y + w_y l_x < w_x l_x + w_y l_y$

  - Since we assume that $w_x l_x + w_y l_y$ is optimal, this is a contradiction.

Lemma 2: Consider the 2 symbols x and y with the lowest frequency which Huffman's algo combines in the first step. Those 2 are siblings.

Proof:

1. Take any opt tree.
2. Let x be the label with the lowest freq.
3. If x doesn't have the longest encoding, swap it with one that has. The overall length of the tree won't be affected bc of Lemma 1.
4. Due to optimality, x must have a sibling. If it doesn't, then it can't combine with another symbol.
5. If it's not y, swap with y. Again, bc of Lemma 1, the tree won't be affected.

Induction Step:

Let x and y be the 2 least freq symbols that Huffman combines in the first step into xy.
Let H be the tree Huffman produces.
Let T be an optimal tree in which x and y are siblings.
Let H' and T' be obtained from H and T by treating xy as 1 symbol with freq $w_x + w_y$.
By I.H., $\text{len}(H') < \text{len}(T')$
$\text{len}(H) = \text{len}(H') + (w_x + w_y)$
$\text{len}(T) = \text{len}(T') + (w_x + w_y)$
$\therefore \text{len}(H) \leq \text{len}(T)$