CSC373: Algorithm Design, Analysis and Complexity
Winter/Spring 2020

Allan Borodin and Sara Rahmati

Week of January 13, 2020

Week 2

Announcements

- Assignment 1 was posted on January 12 and is due Thursday, February 6 at 4:59 PM.
- We have been able to increase the enrollment cap by a few more places, so if you are not registered try registering now.
- You should all be registered on markus now. If not, then let me know.
- My office hours: Mondays 1:30-2:30 and Wednesdays 4:30-5:30

This weeks agenda

1. Finish divide and conquer
   - randomized median/selection algorithm
   - The DFT and FFT
2. Begin greedy algorithms
   - The interval scheduling problem
   - The greedy algorithmic paradigm
   - Proving optimality of greedy algorithms
   - Interval coloring
   - Kruskal and Prim MST algorithms for MST
   - Huffman coding

Computing the median and selecting the ith smallest element

As a special case of the third part of the master theorem, the recurrence $T(n) = T(n/b) + O(n)$ implies $T(n) = O(n)$.

We would like to design an algorithm for computing the median in an unsorted list of n elements using only $O(n)$ comparisons. (We can rather more naively simply sort using time $O(n \log n)$ comparisons.) There is a deterministic algorithm based on this recurrence but it is simpler to consider a randomized algorithm whose *expected time* is $O(n)$.

It turns out that in trying to come up with an appropriate divide and conquer algorithm for the median, it is "more natural" to consider the more general problem of selecting the ith smallest (or largest) element.

The following approach is based on the partitioning idea in randomized quicksort, and then using the intended recurrence to motivate the selection algorithm (see sections 7.1 and 9.2 in CLRS).

In-place partition

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p - 1
3  for j = p to r - 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Figure: In-place parition

Randomized quicksort

RANDOMIZED-PARTITION(A, p, r)

```
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION:

RANDOMIZED-QUICKSORT(A, p, r)

```
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q - 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Figure: Quicksort

Randomized selection of ith smallest in $A[p], \ldots, A[r]$

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2      return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q - p + 1
5  if i == k            // the pivot value is the answer
6      return A[q]
7  elseif i < k
8      return RANDOMIZED-SELECT(A, p, q - 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```

Figure: Randomized selection

Randomized selection in DPV

Without worrying about the in place partitioning, here is the basic idea as stated in DPV:

$$\text{selection}(S,k) = \begin{cases} \text{selection}(S_L,k) & \text{if } k \le |S_L| \\ v & \text{if } |S_L| < k \le |S_L| + |S_v| \\ \text{selection}(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v|. \end{cases}$$

For the analysis, I also like how DPV explain the expected time. As I mentioned, choosing a random pivot in a list is likely to choose at"good pivot" which is the ith largest for $\frac{|V|}{4} \le i \le \frac{3|V|}{4}$. That is, this will occur with probability $\frac{1}{2}$. Then they claim (and this is not hard to verify) that the expected number of trials (randomly choosing possible pivot elements) to find a good pivots is 2. So that if $T(n)$ is the expected time for selection on a list of length n, the $T(n) = T(n/2) + O(n)$.

The DFT and FFT

One of the most important procedures in signal processing is the use of the Fast Fourier Transform (FFT). The FFT is an algorithm for computing the Discrete Fourier Transform (DFT).

To define the DFT, we need the concept of a primitive n^{th} root of unity in the complex plane. Namely, ω is a primitive n^{th} root of unity if

1. $\omega^n = 1$
2. $\omega^i \neq 1$ for $0 < i < n$.

The DFT_n is defined as a matrix vector multiplication $\mathbf{y} = V(\omega_n) \cdot \mathbf{a}$ where $V(\omega_n)$ is a Vandermond matrix with respect to the primitive n^{th} root of unity ω_n.

It turns out that $V^{-1} = \frac{1}{n} V(\omega_n^{-1})$, the Vandermode matrix with respect to the nth primitive root of unity ω_n^{-1}, so that recovering $\mathbf{a}$ from $\mathbf{y}$ is also the same process. You can veeify that $I = V^{-1}V$.

Defining the DFT

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} .$$

Figure: The nth order DFT_n where ω_n is a primitive n^{th} root of unity

Note: We will assume that $n = 2^k$ for some integer $k \geq 0$. For some applications of the FFT, we cannot make this assumption and then a more substantial development is needed.

The FFT; Computing the DFT_n in $O(n \log n)$ arithmetic operations

In general, computing a matrix vector product $G_{n \times n} \cdot \mathbf{z}$ requires $\Theta(n^2)$ arithmetic operations. But the Vandermonde matrix is obviously a special matrix.

Observation 1: We can think of the matrix vector product $V \cdot \mathbf{a}$ as the evaluation of the polynomial $\sum_{i=0}^{n-1} a_i x^i$ at the n points $x_i = \omega_n^i$ for $i = 0, \ldots, n-1$.

Observation 2: If ω_n is a primitve n^{th} root of unity then $\omega_{n/2} = \omega_n^2$ is a primitive $(\frac{n}{2})^{th}$ root of unity and $\omega_{n/2}^{n/2} = 1$. (Recall we are assuming $n = 2^k$.)

Observation 3: $\omega_n^{\frac{n}{2}+j} = -\omega_n^j$

To hopefully simplify notation, let $\omega_{(n);j}$ denote ω_n^j so that our goal is to evaluate $a(x)$ at the n points $\omega_{(n);j}$ $(j = 0, 1, \ldots, n-1)$.

The FFT continued: the basic obervation

We can express the polynomial evaluation as follows:
$\sum_{i=0}^{n-1} a_i x^i = \sum_{i=0}^{\frac{n}{2}-1} a_{2i} x^{2i} + x \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} x^{2i}$ (i.e. even terms + odd terms)

The FFT continued: the basic obervation

We can express the polynomial evaluation as follows:
$\sum_{i=0}^{n-1} a_i x^i = \sum_{i=0}^{\frac{n}{2}-1} a_{2i} x^{2i} + x \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} x^{2i}$ (i.e. even terms + odd terms)
Therefore, when $x = \omega_{(n);j}$ for j = 0, . . . , n-1

$$\begin{aligned}\sum_{i=0}^{n-1} a_i \omega_{(n);j}^{i} &= \sum_{i=0}^{\frac{n}{2}-1} a_{2i} \omega_{(n);j}^{2i} + \omega_n \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} \omega_{(n);j}^{2i} \\ &= \sum_{i=0}^{\frac{n}{2}-1} a_{2i} \omega_{(n/2);k}^{i} \ \{+,-\} \ \omega_n \sum_{i=0}^{\frac{n}{2}-1} a_{2i-1} \omega_{(n/2);,k}^{i}\end{aligned}$$

for $k = 0, 1, \ldots \frac{n}{2} - 1$ using the "$\{+,-\}$ trick" as presented in DPV.

The FFT continued: the basic obervation

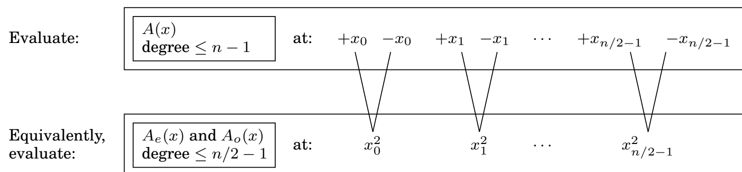
We can express the polynomial evaluation as follows:
$\sum_{i=0}^{n-1} a_i x^i = \sum_{i=0}^{\frac{n}{2}-1} a_{2i} x^{2i} + x \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} x^{2i}$ (i.e. even terms + odd terms)
Therefore, when $x = \omega_{(n);j}$ for j = 0, . . . , n-1

$$\sum_{i=0}^{n-1} a_i \omega_{(n);j}^i = \sum_{i=0}^{\frac{n}{2}-1} a_{2i} \omega_{(n);j}^{2i} + \omega_n \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} \omega_{(n);j}^{2i}$$
$$= \sum_{i=0}^{\frac{n}{2}-1} a_{2i} \omega_{(n/2);k}^i \; \{+,-\} \; \omega_n \sum_{i=0}^{\frac{n}{2}-1} a_{2i-1} \omega_{(n/2);,k}^i$$

for $k = 0, 1, \ldots \frac{n}{2} - 1$ using the "$\{+,-\}$ trick" as presented in DPV.

It follows that by this rearrangement, the computation of DFT_n reduces to two instances of the $DFT_{n/2}$ + n scalar multiplications + n scalar additions. Therefore $T(n) = 2T(n/2) + O(n)$

The plus-minus trick as in DPV text



$$
\begin{aligned}
A(x_i) &= A_e(x_i^2) + x_i A_o(x_i^2) \\
A(-x_i) &= A_e(x_i^2) - x_i A_o(x_i^2).
\end{aligned}
$$

Using the FFT to derive a fast polynomial multiplication

We can use the FFT to derive an $O(d \log d)$ polynomial time algorithm for multiplying two degree d polynomials. The basic idea is that a degree d polynomial can be uniquely represented by its value at $d+1$ distinct points. The process of recovering the coefficients of the polynomial from these $d+1$ values is called *interpolation*.

We wish to compute the coefficients of $c(x) = a(x) * b(x)$ from the coefiencients of $a(x)$ and $b(x)$, Here is how we can do it using the FFT,

- Let n be the smallest power of 2 such that $n - 1 \geq 2d$, the degree of $c(x)$. We can view $a(x), b(x)$ and $c(x)$ as degree $n-1$ polynomials by adding enough leading zero coeficients. Note: this will not impact the desired asymoptotic bound.

Fast polynomial multiplcation continued

- Evaluate $a(x)$ and $b(x)$ at the nth roots of unity; let these values be $u_0, \ldots u_{n-1}$ and $v_1, \ldots, v_{n-1}$ (for the values of $a(x)$ and respectively $b(x)$.
- Do n scalar multiplication $w_i = u_i \cdot v_i$ for $i = 0, \ldots, n-1$.
- Interpolate (using the FFT with primitive root ω_n^{-1}) to obtain the coefficients of $c(x)$.

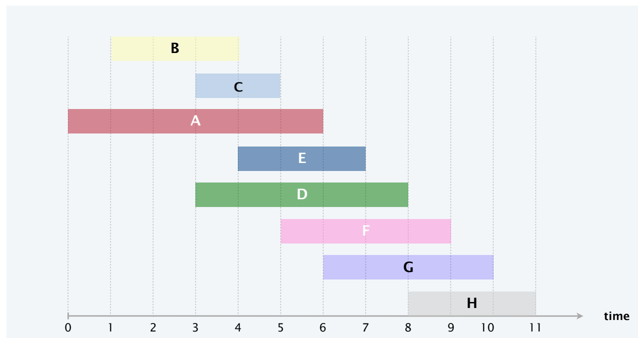
Something similar (but more involved) to this can be done for fast integer integer multiplication but now we need a discrete analogue of a primitive root of unity. This is the underlying idea of the Schonhage-Strassen fast integer multiplcation.

Begin greedy algorithms

We begin by consdering a specific problem, which we can call interval selection or interval scheduling.

Interval Scheduling Problem

- Job j starts at s_j and finishes at f_j.
- Two jobs are compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



Interval Scheduling: Greedy Algorithm

Greedy template

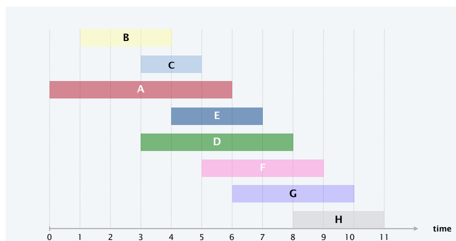
- Consider jobs in some "natural" order. What order?
- Take each job provided it's compatible with the ones already taken.

Interval Scheduling: Greedy Algorithm

Greedy template

- Consider jobs in some "natural" order. What order?
- Take each job provided it's compatible with the ones already taken.

1. Earliest start time: Consider jobs in ascending order of s_j.
2. Earliest finish time: Consider jobs in ascending order of f_j.
3. Shortest interval: Consider jobs in ascending order of $f_j - s_j$.
4. Fewest conflicts: For each job j, count the remaining number of conflicting jobs c_j. Schedule in ascending order of c_j.



Interval Scheduling: Greedy Algorithm

Greedy template

- Consider jobs in some “natural” order.
- Take each job provided it’s compatible with the ones already taken.

counterexample for earliest start time

counterexample for shortest interval

counterexample for fewest conflicts

A general greedy template

A general (greedy) myopic template

- Consider input items in some "reasonable" order.
- Consider each input item and make an irrevocable (greedy) decision regarding that input item.

Terminology and templates

We will follow the more common "greedy algorithms" terminology. However, as mentioned in the DPV text (and following older terminology), We think it would be better to use instead the suggestively broader class of myopic algorithms. You may also note that the informal template for greedy algorithms differs somewhat from that given in the DPV text. Our view is that the "greedy aspect" of these one-pass myopic algorithms relates to the nature of the irrevocable decision rather than the choice of the next input item to consider. In any case, neither the above template nor that in DPV is a precise definition. Why?

Optimality of EFT Greedy algorithm for unweighted interval scheduling

The KT text suggests the following proof for the optimality of the Earliest Finishing Time (EFT). Let S_i be the set of intervals accepted by the end of the ith iteration.

- We say S_i is *promising* if it can be extended to an optimal solution.
- Formally this means that for all i $(0 \leq i \leq n)$, there exists an optimal solution OPT_i such that $S_i \subseteq OPT_i \subseteq S_i \cup \{J(i+1), \ldots, J(n)\}$.
- By induction we can prove that S_i is promising for all i which implies that S_n must be optimal. Why?
  1. The base case is "trivial".
  2. The inductive step is proved by cases:

     Case 1: $s_{i+1} < f_i$ so that $S_{i+1} = S_i$
     Case 2A: $s_{i+1} \geq f_i$ and $i+1 \in OPT_i$ so that $OPT_{i+1} = OPT_i$
     Case 2B: $s_{i+1} \geq f_i$ and $i+1 \notin OPT_i$; this is the interesting case.
     What has to be done?

Some comments on proofs

Why do we do proofs? Even though the EFT algorithm is conceptually very simple, there were other (incorrect) possibilities so one needs a proof.

Moreover, we can gain additional insights from a proof as to what is being used in the proof. For example, in the EFT optimality proof, we do not need to assume that all finishing times $\{f_i\}$ are distinct.

Is this the only or "best" way to prove this result. Although understanding proofs can be quite subjective, some proof methods seem to lend themselves to some problems better than others. The "promising" solutions method seems well suited to greedy style algorithms.

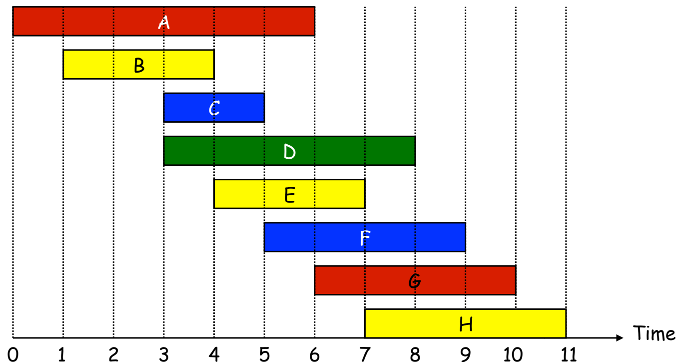
An alternative proof is by what is called a "charging argument". In this case, the charging argument informally wants to charge each interval of an optimal (or arbitrary solution) to a unique interval in the greedy solution. Charging arguments are often used in approximation algorithms.

More precisely, we want to define a 1-1 function $h : OPT \rightarrow S$. This would imply that $|OPT| \leq |S|$. Can you think of such a function h?

Interval colouring

Interval Colouring Problem

- Given a set of intervals, colour all intervals so that intervals having the same colour do not intersect
- **Goal:** minimize the number of colours used.



We use 4 colors in this example. **Question:** Is this optimal?

Interval colouring

Interval Colouring Problem

- Given a set of intervals, colour all intervals so that intervals having the same colour do not intersect
- **Goal:** minimize the number of colours used.

- We could simply apply the m-machine ISP for increasing m until we found the smallest m that is sufficient. (See the in assignment 1.)

- **Note:** This is a simple example of a polynomial time reduction which is an essential concept when we study NP-completeness. But this would not be as efficient as the greedy algorithm to follow.

Greedy interval colouring

- Consider the EST (earliest starting time) for interval colouring.
  - Sort the intervals by non decreasing starting times
  - Assign each interval the smallest numbered colour that is feasible given the intervals already coloured.
- Recall that EST is a terrible algorithm for ISP.
- **Note:** this algorithm is "equivalent" to LFT (latest finishing time first).

Theorem

EST is optimal for interval colouring

Greedy Interval Colouring

Sort intervals so that $s_1 \leq s_2 \leq \ldots \leq s_n$
FOR $i = 1$ to n
Let $k := \min\{\ell : \ell \neq \chi(j)$ for all $j < i$ such that the jth interval intersects the ith interval$\}$
$\sigma(i) := k$
% The ith interval is greedily coloured by the smallest non conflicting colour.
ENDFOR

Proof of Optimality (sketch)

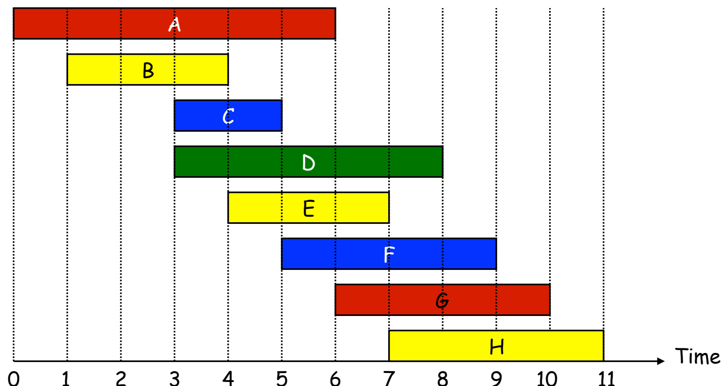
- The proof technique we will use here is also often used for proving approximations.
- The idea is to find some bound (or bounds) that any solution must satisfy and then relate that to the algorithm's solution.
- In this case, consider the maximum number of intervals in the input set that intersect at any given point.

Observation

The number of colours must be at least this large.

- It remains to show that the greedy algorithm will never use more than this number of colours.

An example



- The maximum of number of intersecting intervals is 4.
- So we can't use less than 4 colors in this example.

Why doesn't the Greedy Colouring Algorithm exceed this intrinsic bound?

- Recall that we have sorted the intervals by nondecreasing starting time (i.e. earliest start time first).
- Let $k =$ maximum number of intervals in the input set that intersect at any given point.
- Suppose for a contradiction that

the algorithm used more than k colours.

- Consider the first time (say on some interval ℓ) that the greedy algorithm would have used $k+1$ of colours.
  - Then it must be that there are k intervals intersecting ℓ.
  - Let s be the starting time of ℓ.
  - These intersecting intervals must all include s. Why?
  - Hence, there are $k+1$ intervals intersecting at s!

Greedy algorithms for the MST problem

We assume that everyone is familiar with the concept of a graph $G = (V, E)$ and an edge weighted graph where there is a real valued weight function $w : E \rightarrow \mathbb{R}$.

We will start with the assumption that the input graph G is *connected* in which case all the nodes can be connected by a *tree*. Such a tree is called a spanning tree.

Our goal is to constuct a minimum spanning tree (MST); that is, we wish to find a subset of edges $E' \subseteq E$ such that $T = (V, E')$ is a spanning tree minimizing $\sum_{e \in E'} w_e$.

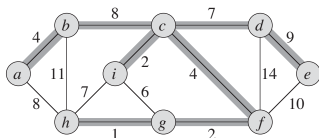


Figure: Figure 23.1 in CLRS; bold edges in a (non-unique) MST

Kruskal's and Prim's greedy MST algorithms

- We recall the graph definitions of a tree (i.e., an acyclic connected graph) and a forest (i.e., a disjoint collectionn of trees). Kruskals algorithm proceeds by iteratively, merging subtrees (in a forest) of the input graph. Prim's algorithm proceeds by iteratively extending a subtree of the graph.
- There is a basic fact that underlies Kruskal's and Prim's MST algorithms and their proofs; namely, if we add an edge to a tree, then it forms a unique cycle. Cycles can be detected by maintain sets corresponding to components of the graph.
- So in both Kruskal's and Prim's algorithm, it seems reasonable to always iteratively choose the smallest weight edge that does not form a cycle.

Implementing Kruskal's greedy MST algorithm

```
procedure kruskal(G, w)
Input:    A connected undirected graph G = (V, E) with edge
Output:   A minimum spanning tree defined by the edges X

for all u ∈ V:
   makeset(u)

X = {}
Sort the edges E by weight
for all edges {u, v} ∈ E, in increasing order of weight:
   if find(u) ≠ find(v):
      add edge {u, v} to X
      union(u, v)
```

Figure: Kruskals algorithm from DPV

Proving the optimality of Kruskals MST algorithm

We can use the same inductive "promising solution" argument that was used for the interval selection problem:

Let X_i be the partial solution (i.e. forest) after the ith iteration. By induction we can prove that X_i is promising for all i which implies that X_n must be optimal. In the algorithm let $e_i = (u, v)$ be the ith smallest weight edge.

Case 1: Adding $e_i = (u, v)$ forms a cycle so that $X_{i+1} = X_i$
Case 2A: $X_i \cup \{(u, v)\}$ is still acyclic and $e_{i+1} \in OPT_i$ so that $OPT_{i+1} = OPT)_i$
Case 2B: $X_i \cup \{(u, v)\}$ is still acyclic and $e_{i+1} \notin OPT_i$; this is again the interesting case. What has to be done?

Proving the optimality of Kruskals MST algorithm

We can use the same inductive "promising solution" argument that was used for the interval selection problem:

Let X_i be the partial solution (i.e. forest) after the ith iteration. By induction we can prove that X_i is promising for all i which implies that X_n must be optimal. In the algorithm let $e_i = (u, v)$ be the ith smallest weight edge.

Case 1: Adding $e_i = (u, v)$ forms a cycle so that $X_{i+1} = X_i$
Case 2A: $X_i \cup \{(u, v)\}$ is still acyclic and $e_{i+1} \in OPT_i$ so that $OPT_{i+1} = OPT)_i$
Case 2B: $X_i \cup \{(u, v)\}$ is still acyclic and $e_{i+1} \notin OPT_i$; this is again the interesting case. What has to be done?

As in the interval selection "promising proof", we can now create a new OPT_{i+1} by considering the cycle C created by adding e_{i+1} to OPT_i. There must be another edge $e \in C$ such that $w(e) \geq w(e_i)$ so that we can replace e by e_i.

Lossless compression: Huffman (prefix-free) encoding

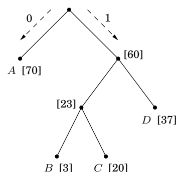
We want to encode a set of n symbols. This would take $k = \log n$ bits per symbol if we give each symbol a unique binary string of k bits. But in say a document, some symbols may only occur rarely while other occur frequently. If we want to compress the document, it would make sense (if possible) to encode frequently occuring symbols with short strings at the expense longer strings for less frequently occuring symbols.

One way to do this is by using a *prefix-free encoding* where no string (in the encoiding) is the prefix of another string. This will guarantee that the document can be recovered from the encoding symbol by symbol. Is this clear?

We can represent a prefix-free code by a binary tree where the edges are $\{0, 1\}$ labelled and the leaves represent the symbols. The depth of a path then becomes the length of the encoding for the symbol at that leaf.

Prefix-free coding continued

Symbol	Codeword
A	0
B	100
C	101
D	11

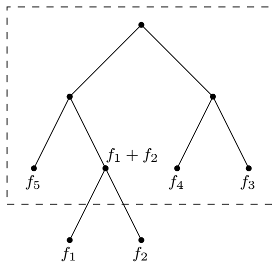


For a given document D, we would know the frequency of each symbol; otherwise we might know the probability of each symbol.
Let the symbols be $a_1, \ldots, a_n$ and let $f_i = \frac{n_i}{|D|}$ be the probability of each symbol when n_i is the number of occurrences and $|D|$ is the length of the document.

Sort the symbols so that $f_1 \leq f_2 \ldots \leq f_n$. Let ℓ_i be the length of the ith least frequent symbol in the encoding.

It follows that $\sum_i f_i \ell_i$ is the expected length of a symbol and $\sum_i n_i \ell_i$ is the total length of the encoded document. In either case, we wish the construct a prefix-tree so as to minimize these sums.

Huffman algorithm from DPV text




```
procedure Huffman(f)
Input:  An array f[1···n] of frequencies
Output:  An encoding tree with n leaves

let H be a priority queue of integers, ordered by f
for i = 1 to n:  insert(H, i)
for k = n + 1 to 2n − 1:
  i = deletemin(H),  j = deletemin(H)
  create a node numbered k with children i, j
  f[k] = f[i] + f[j]
  insert(H, k)
```

Interval graphs: The interval selection and coloring problems as graph problems

- There is a natural way to view the interval scheduling and colouring problems as **graph problems**.
- Let $\mathcal{I}$ be a set of intervals. We can construct the intersection graph $G(\mathcal{I}) = (V, E)$ where
  - $V = \mathcal{I}$
  - (u, v) is an edge in E iff the intervals corresponding to u and v intersect.
- Any graph that is the intersection graph of a set of intervals is called an interval graph.
- The interval selection (resp. interval coloring) problem can be viewed as the *maximum independent set* problem (MIS) for the class of interval graphs.
- Intersection graphs are studied for many other geometric objects and other types of objects.

Graph MIS and Colouring

- Let $G = (V, E)$ be a graph.
- The following two problems are known to be "NP- hard to approximate" for arbitrary graphs:

Graph MIS

- A subset U of V is an independent set (aka stable set) in G if for all $u, v \in U$, (u, v) is not an edge in E.
- The maximum independent set (MIS) problem is to find a maximum size independent set U.

Graph colouring

- A function c mapping vertices to $\{1, 2, \ldots, k\}$ is a valid colouring of G if $c(u)$ is not equal to $c(v)$ for all $(u, v) \in E$.
- The graph colouring problem is to find a valid colouring so as to minimize the number of colours k.

Efficient algorithms for interval graphs

- Given a set $\mathcal{I}$ of intervals, it is easy to construct its intersection graph $G(\mathcal{I})$.

Note: The following is a known interesting theorem

Given any graph G, there is a linear-time algorithm to decide if G is an interval graph and if so to construct an interval representation.

- The MIS (resp. colouring) problem for interval graphs is the MIS (resp. colouring) problem for its intersection graph and hence these problems are efficiently solved for interval graphs.
  - Question: Is there a graph theoretic explanation?
  - YES: interval graphs are *chordal graphs* having a *perfect elimination ordering.*
- The minimum colouring number (chromatic number) of a graph is always at least the size of a maximum clique.
  - The greedy interval colouring proof shows that for interval graphs

the chromatic number = max clique size.

Summarizing the greedy paradigm

- Informally, (most) greedy algorithms consider one input item at a time and make an irrevocable ("greedy") decision about that item before seeing more items.
- To make this precise for any given problem we have to say
  1. how input items are represented ; for example, what other information could we put into the representation of an interval?
  2. how an algorithm determines the order in which input items are considered.
- Key to formalizing our intuitive idea off greedy algorithm: we need to define the class of orderings of the input items that will be allowed. We cannot allow any ordering of the input set or else one can say take exponential time to compute an "optimal ordering".
- If we try to make this precise by stating that the ordering must be done in say time $O(n \log n)$ (or even $poly(n)$) then we are in the situation of trying to prove that something cannot be done in a given time bound. The Turing tarpit

One way to formalize how to order

- For a given problem, assume that input items belong to some set $\mathcal{J}$.
- For any execution of the algorithm, the input is a finite subset $\mathcal{I} \subset \mathcal{J}$.
- Let $f : \mathcal{J} \rightarrow \Re$ be any function; that is, we do not place any restriction on the complexity or even the computability of the function.
- Then for any actual input set $\mathcal{I} = \{I_1, \ldots, I_n\}$, the function f induces a total order on the input set (where we can break ties using the index of the input items as given).
- In a fixed order the function f is set initially. In an adaptive order, there can be a different function f_i in each iteration i with f_i depending on the items considered in iterations $j < i$.

Jeff Erickson's comment on greedy algorithms

5.4 Warning: Greed is Stupid

If we're very very very very lucky, we can bypass all the recurrences and tables and so forth, and solve the problem using a *greedy* algorithm. The general greedy strategy is look for the best first step, take it, and then continue. While this approach seems very natural, it almost never works; optimization problems that can be solved correctly by a greedy algorithm are *very* rare. Nevertheless, for many problems that should be solved by dynamic programming, many students' first intuition is to apply a greedy strategy.

For example, a greedy algorithm for the edit distance problem might look for the longest common substring of the two strings, match up those substrings (since those substitutions don't cost anything), and then recursively look for the edit distances between the left halves and right halves of the strings. If there is no common substring—that is, if the two strings have no characters in common—the edit distance is clearly the length of the larger string. If this sounds like a stupid hack to you, pat yourself on the back. It isn't even *close* to the correct solution.

Everyone should tattoo the following sentence on the back of their hands, right under all the rules about logarithms and big-Oh notation:

Greedy algorithms never work!
Use dynamic programming instead!

What, never?
No, never!
What, *never*?
Well... hardly ever.[6]

End of Week 2

We ended the Wednesday night class with this rather pessimistic comment by Jeff Erickson. Next week we will pick up where we left off after first possibly going over some material from this week and last week.

DP and Linear programming: two sledgehammers of the *algorithmic craft*

The following is the first paragraph of chapter 6 of DPV.

> In the preceding chapters we have seen some elegant design principles—such as divide-and-conquer, graph exploration, and greedy choice—that yield definitive algorithms for a variety of important computational tasks. The drawback of these tools is that they can only be used on very specific types of problems. We now turn to the two *sledgehammers* of the algorithms craft, *dynamic programming* and *linear programming*, techniques of very broad applicability that can be invoked when more specialized methods fail. Predictably, this generality often comes with a cost in efficiency.

My view of greedy algorithms

- First, the previous comments are in the context of emphasizing DP and LP algorithms for optimizatio problems and were a deliberate overstating of the point.
- My view of greedy algorithms is that while they may not often be optimal or as good as more sophisticated algorithms, there are many cases where they work well either in terms of provable approximations or "in practice".
- Moreover, in some cases we imediately need something that works and knowing some basic approaches to a problem becomes a starting point. If nothing esle, greedy algorithms can be a benchmark for comparison against more sophisticated algorithms.
- DP algorithms, once they are formulated, often seem quite apparant. But coming up with a correct DP formulation is often not so obvious. In contrast, coming up with a correct (albeit possibly one having poor performance) greedy algorithm is usually easy to do.
- Finally, there are applications (e.g. auctions) where conceptual simplicity is a virtue in itself.