CS 5112

- Huffmann Codes

- Divide & conquer

The announcements you've all been waiting for

- **HW1 to be released today**
  - Check Canvas for HW1
  - Due September 26 (midnight)
- **HW0 will be released this week as well**
  - For your benefit!
  - Must turn in for a few points of credit, but will not be graded
  - Solutions will be available to you
  - Basics in Python, review / warmup
  - Review session next week to get people up and running. Stay tuned for announcement of when/where. Will be due shortly after.

A note on doing homeworks

- Why are you taking this class?

A note on doing homeworks

- Why are you taking this class?
- We allow collaboration
  - Groups up to 3
  - Turn in your own solutions
- We allow using online resources, code assistance
  - Cite resources you used, collaborators at top of homework PDF
  - *My recommendation:* try to solve the problems on your own, then discuss with group, avoid online resources except for background
    - You will learn more
    - Be more prepared for algorithmic challenges in your future careers

Today's game plan

- Wrap up Huffman codes
- Introduce new class of algorithms:
  - Divide and conquer

Compressing data, losslessly

- Compression critical for saving space
  - History of development goes back to Shannon 1948
  - Significant evolution
  - Many tools: gzip, bzip2, zip, deflate, Google's brotli (from 2013)
- Prefix codes:
  - Shannon codes are "top down" approach
  - Huffman codes are "bottom up" approach
  - Middle-out approach? Doesn't exist, that's fiction
- Huffmann codes used widely in compression tools
  - Deflate: LZ78 algorithm followed by static or dynamic Huffmann

Prefix codes

Def. A prefix code for a set S is a function c that maps each $x \in S$ to binary string so that $x,y \in S$, $x \neq y$, $c(x)$ is not a prefix of $c(y)$

Prefix codes

Def. A prefix code for a set S is a function c that maps each x ∈ S to binary string so that x,y ∈ S, x ≠ y, c(x) is not a prefix of c(y)

Def. Let f_x for x ∈ S be a frequency associated to character x

Def. The average bits per letter of a code *c* is

$$ABL(c) = \sum_{x \in S} f_x \cdot |c(x)|$$

We want to find an ***optimal code***: lowest *ABL(c)* for given S, f_x

Binary trees and prefix codes

Letter	frequency
a	0.32
b	0.25
c	0.20
d	0.18
e	0.05

Huffmann's Bottom-up algorithm

Huffmann(S,f)

If $|S|$ = 2 then
  Let T be tree with one letter set to 0, other 1
Else
  Let v and w be the lowest-frequency letters
  Let $S' = S - \{v, w\} + \{\omega\}$ with:
    $f_\omega' = f_v + f_w$ and $f_x' = f_x$ for $x \in S' - \{\omega\}$
  T' = Huffmann(S',f')
  Let T be prefix tree with leafs v, w added below ω
Return T

Run time?

Huffmann's Bottom-up algorithm

<u>Huffmann(S,f)</u>

If $|S| = 2$ then
  Let T be tree with one letter set to 0, other 1
Else
  Let v and w be the lowest-frequency letters
  Let $S' = S - \{v, w\} + \{\omega\}$ with:
    $f_\omega' = f_v + f_w$ and $f_x' = f_x$ for $x \in S' - \{\omega\}$
  T' = Huffmann(S',f')
  Let T be prefix tree with leafs v, w added below ω
Return T

<u>Run time?</u>

At most n iterations

Linear scans per iteration

$$T(n) = T(n-1) + O(n)$$

So: $\boldsymbol{O(n^2)}$

But can use priority queue to make each iteration faster:

$$T(n) = T(n-1) + O(\log n)$$

So: $\boldsymbol{O(n \log n)}$

Huffmann on our example

Letter	frequency
a	0.32
b	0.25
c	0.20
d	0.18
e	0.05

Huffmann(S,f)

If $|S| = 2$ then
    Let T be tree with one letter set to 0, other 1
Else
    Let v and w be the lowest-frequency letters
    Let $S' = S - \{v, w\} + \{\omega\}$ with:
        $f_\omega' = f_v + f_w$ and $f_x' = f_x$ for $x \in S' - \{\omega\}$
    T' = Huffmann(S',f')
    Let T be prefix tree with leafs v, w added below ω
Return T

Optimality of Huffmann

Lemma 4.29: For any optimal tree *T**, if depth(u) < depth(v) then $f_u \geq f_v$

Lemma 4.31: Exists optimal prefix code with lowest frequency characters as siblings

Lemma 4.32: $ABL(T') = ABL(T) - f_\omega$

Theorem: The Huffmann code for a given alphabet is optimal

Optimality of Huffmann

Theorem: The Huffmann code for a given alphabet is optimal

Proof by induction on k = |S|

Base case: k = 2. Optimal tree is one that has two leaves

Inductive step: Suppose Huffman tree *T'* for *S'* of size k-1 with ω instead of *v* and *w* is **optimal**. Then we will derive contradiction:

- Suppose a better tree *Z* than what Huffmann tree *T*
- Then make *Z'* from *Z* by deleting lowest frequency items (*v*,*w*)
- But *Z'* cannot be better than *T'*, which ends up at contradiction

Optimality of Huffmann

Theorem: The Huffmann code for a given alphabet is optimal

Proof by induction on k = |S|

Base case: k = 2. Optimal tree is one that has two leaves

Inductive step: Suppose Huffman tree *T'* for *S'* of size k-1 with ω instead of *v* and *w* is **optimal**. Then we will derive contradiction:

- Suppose some other tree *Z* has *ABL(Z) < ABL(T)*
- Without loss of generality, can assume *Z* has *v*, *w* as leaves (Lemma 4.31)
- Remove *v*, *w* from *Z*, replace with parent node ω, creating *Z'*
- Because one gets *Z* from *Z'* by adding back in *v, w,* Lemma 4.32 applies
- Thus by Lemma 4.32 $ABL(Z') = ABL(Z) - f_\omega$ and $ABL(T') = ABL(T) - f_\omega$
- But by assumption *ABL(Z) < ABL(T)*, so *ABL(Z') < ABL(T').* **Contradiction!**

Greedy algorithms we saw

- **Interval scheduling** in O(n log n)
  - Trick: take earliest finish time
- **Minimum spanning tree** in O(n log n)
  - Many approaches
  - Kruskal's: Order edges by weight, iteratively select minimum weight one that doesn't cause cycle
- **Huffmann** in O(n log n)
  - Build a tree "bottom up" by taking two lowest frequency characters

All these solved problems with local update rule, avoiding brute-force

Some classes of algorithmic approaches

Class	Brute force approach?	Notes
Greedy algorithms	Inefficient (exponential)	Find “local update” rule that guarantees globally correct solution
Divide and conquer	Polynomial, e.g., $O(n^2)$	Partition input into multiple sub-problems. Solve each independently, then carefully merge
Dynamic programming	Inefficient (exponential)	Greedy approach doesn’t work; decompose into subproblems and build up solution from them

Useful to be able to identify class of algorithms that might work well for some problem
Seeing lots of examples, and thinking through them key to doing so!

Sorting a list *L*

We've assumed several times that you can sort a list in $O(n \log n)$ time
But how can this work?

Sorting a list *L*

We've assumed several times that you can sort a list in *O(n log n)* time
But how can this work?

<u>Bruteforce-Sort(*L*)</u>

While $|L| > 0$
    Find minimum value *x* in *L*
    Append *x* to *L'*
    Remove *x* from *L*

Return *L'*

Run time?

Sorting a list *L*

We've assumed several times that you can sort a list in *O(n log n)* time
But how can this work?

```
Insertion-Sort(L)

i <- 1
While i < |L|
    j <- i
    While j > 0 and L_{j-1} > L_j
        Swap L_{j-1} and L_j
        j <- j - 1
    i <- i + 1

Return L
```

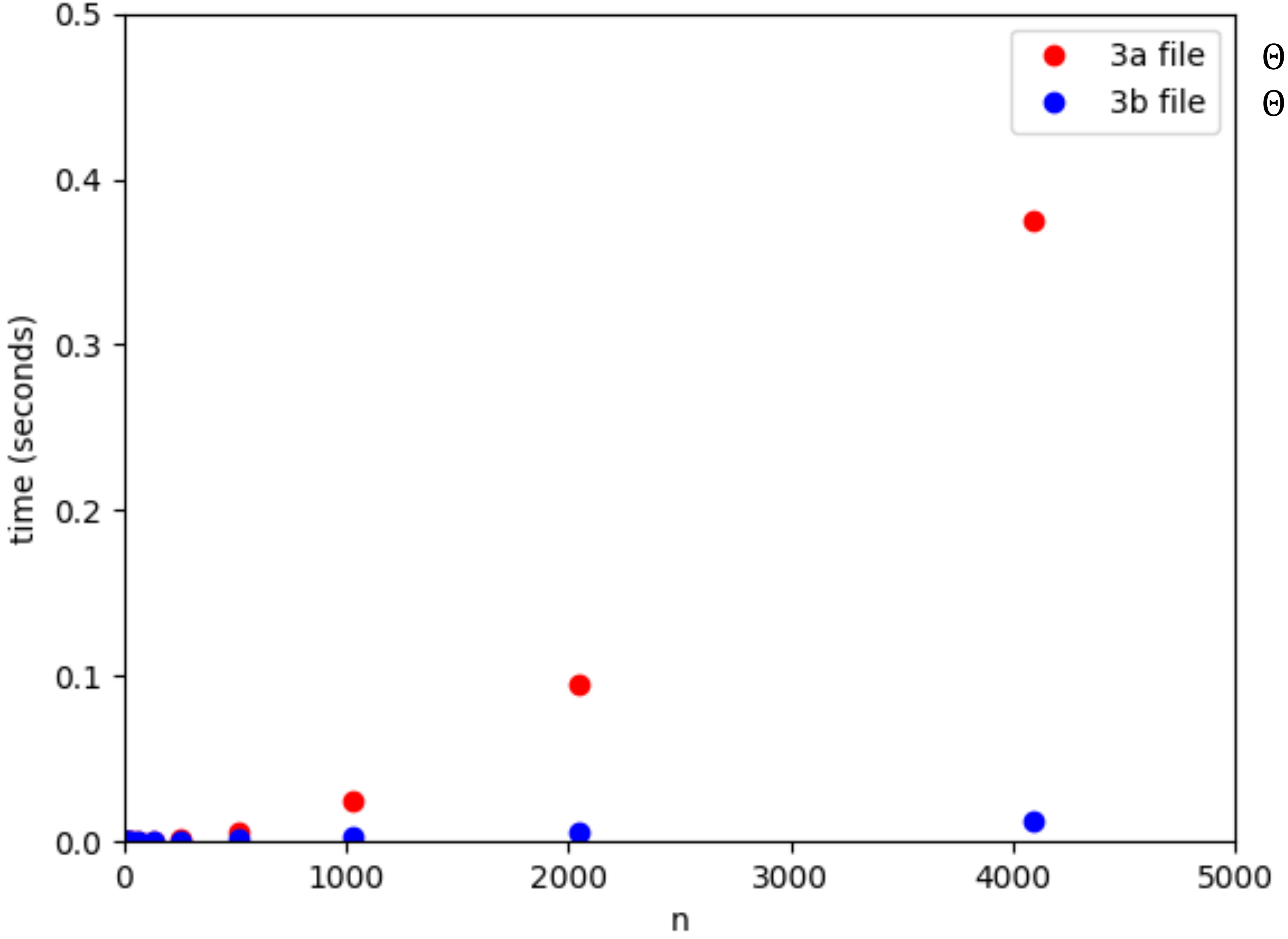
Run time?

Divide and conquer

Class of algorithmic techniques in which:

1. Divide input into sub-problems   (most often, two halves each of size n/2)
2. Solve problem on each sub-problem
3. Carefully merge solutions to solve your problem (usually, $O(n)$ time)

Often moves from time $O(n^2)$ solution using brute-force to $O(n \log n)$



$\Theta(n^2)$

$\Theta(n \log n)$

Credit: Carolina Ortega Perez

Sorting a list *L*

We've assumed several times that you can sort a list in *O(n log n)* time
But how can this work?

<u>Merge-Sort(*L*)</u>

If |*L*| = 1 then Return *L*

Split *L* into two halves *A*, *B*
A <- Merge-Sort(*A*)
B <- Merge-Sort(*B*)
L <- Merge(*A*,*B*)

Return *L*

Sorting a list *L*

We've assumed several times that you can sort a list in *O(n log n)* time
But how can this work?

<u>Merge-Sort(*L*)</u>

If |*L*| = 1 then Return *L*

Split *L* into two halves *A*, *B*
A <- Merge-Sort(*A*)
B <- Merge-Sort(*B*)
L <- Merge(*A*,*B*)

Return *L*

Run time? T(n) = T(n/2) + T(n/2) + cn

some constant

Sorting a list *L*

We've assumed several times that you can sort a list in *O(n log n)* time
But how can this work?

some constant

Run time? $T(n) = T(n/2) + T(n/2) + cn$

<u>Merge-Sort(*L*)</u>

If |*L*| = 1 then Return *L*

Split *L* into two halves *A*, *B*
A <- Merge-Sort(*A*)
B <- Merge-Sort(*B*)
L <- Merge(*A*,*B*)

Return *L*

Sorting a list *L*

We've assumed several times that you can sort a list in *O(n log n)* time
But how can this work?

Merge-Sort(*L*)

If |*L*| = 1 then Return *L*

Split *L* into two halves *A*, *B*
A <- Merge-Sort(*A*)
B <- Merge-Sort(*B*)
L <- Merge(*A*,*B*)

Return *L*

some constant

Run time? $T(n) = T(n/2) + T(n/2) + cn$

Proof by induction
Base case is $T(2) = c$
Assume $T(m) \leq cm \log_2 m$ for any $m < n$
Then:

Sorting a list *L*

We've assumed several times that you can sort a list in *O(n log n)* time
But how can this work?

<u>Merge-Sort(*L*)</u>

If |*L*| = 1 then Return *L*

Split *L* into two halves *A*, *B*
A <- Merge-Sort(*A*)
B <- Merge-Sort(*B*)
L <- Merge(*A*,*B*)

Return *L*

some constant

Run time? T(n) = T(n/2) + T(n/2) + cn

Proof by induction
Base case is T(2) = c
Assume $T(m) \leq cm \log_2 m$ for any m < n
Then:

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn \\ &\leq 2c(n/2)\log_2(n/2) + cn \\ &= cn\left((\log_2 n) - 1\right) + cn \\ &= (cn\log_2 n) - cn + cn \\ &= cn\log_2 n \end{aligned}$$

Sorting a list *L*

Can we do better than O(n log n) ?

Sorting a list *L*

Can we do better than O(n log n) ?

Comparison sorts are algorithms that only use comparisons between elements

Thm. Any comparison sort algorithm requires Ω(n log n) comparisons in the worst case

Intuition: can lower bound the number of comparisons by viewing comparison sorting as a decision tree, and lower bounding its height

Can do better using more information (counting sort, radix sort, bucket sort…)

Implementing algorithms

<u>Merge-Sort(*L*)</u>

If |*L*| = 1 then Return *L*

Split *L* into two halves *A*, *B*
A <- Merge-Sort(*A*)
B <- Merge-Sort(*B*)
L <- Merge(*A*,*B*)

Return *L*

- Directly implemented, our algorithms would be slow. Why?
  - Wasted overheads making function calls
  - Extra space overheads
- Translate abstract algorithms to implementations fast on real computers
- Other desirable properties:
  - In-place (don't allocate new buffers of size n)
  - Stable (don't move elements that are equal)

Python sort() implementation

Timsort implemented in 2002 by Tim Peters

Combines Merge-Sort and Insertion-Sort with detection of runs

See: https://en.wikipedia.org/wiki/Timsort

Another D&C example: counting inversions

Consider list of distinct numbers $L = x_1, \ldots, x_n$ (assume n is even)

An inversion is a pair of indices $i < j$ such that $x_i > x_j$

Count the number of inversions in a list L

Another D&C example: counting inversions

Consider list of distinct numbers $L = x_1, \ldots, x_n$ (assume n is even)
An inversion is a pair of indices $i < j$ such that $x_i > x_j$
Count the number of inversions in a list L

Another D&C example: counting inversions

Consider list of distinct numbers $L = x_1, \ldots, x_n$ (assume n is even)
An inversion is a pair of indices $i < j$ such that $x_i > x_j$
Count the number of inversions in a list L

<u>Sort-and-Count (L)</u>

If $|L| = 1$ then Return 0
Divide L into two halves A, B
    A has first n/2 elements
    B has last n/2 elements
(r_A , A) <- Sort-and-Count(A)
(r_B , B) <- Sort-and-Count(B)
(r , L) <- Merge-and-Count(A,B)

Return ($r + r_A + r_A$, L)

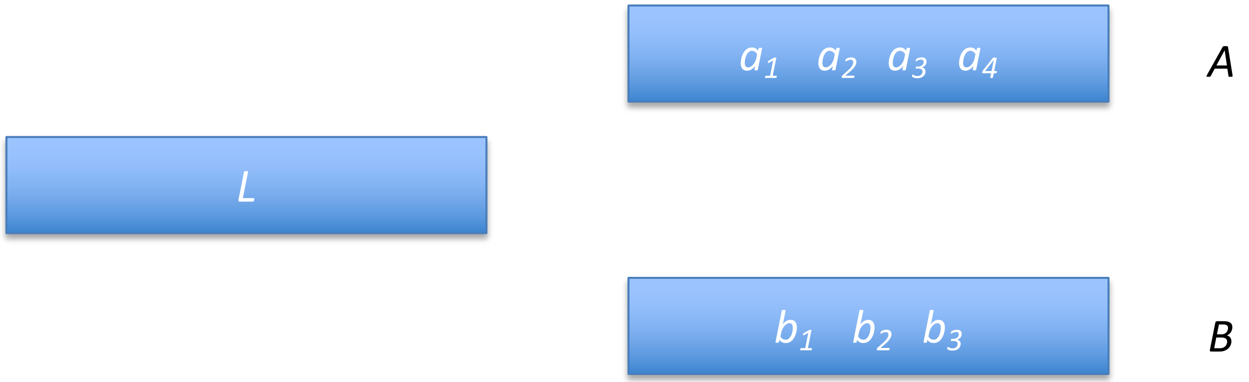
Another D&C example: counting inversions

Consider list of distinct numbers $L = x_1, \ldots, x_n$ (assume n is even)
An inversion is a pair of indices $i < j$ such that $x_i > x_j$
Count the number of inversions in a list L

<u>Sort-and-Count (L)</u>

If $|L| = 1$ then Return 0
Divide L into two halves A, B
    A has first n/2 elements
    B has last n/2 elements
(r_A, A) <- Sort-and-Count(A)
(r_B, B) <- Sort-and-Count(B)
(r, L) <- Merge-and-Count(A,B)

Return $(r + r_A + r_A, L)$



Another D&C example: counting inversions

Consider list of distinct numbers $L = x_1, \ldots, x_n$ (assume n is even)
An inversion is a pair of indices $i < j$ such that $x_i > x_j$
Count the number of inversions in a list L

<u>Sort-and-Count (L)</u>

If $|L| = 1$ then Return 0
Divide L into two halves A, B
    A has first n/2 elements
    B has last n/2 elements
(r_A , A) <- Sort-and-Count(A)
(r_B , B) <- Sort-and-Count(B)
(r , L) <- Merge-and-Count(A,B)

Return ($r + r_A + r_A$, L)

<u>Merge-and-Count(A,B)</u>
L <- empty list
Count <- 0 ; i <- 1 ; j <- 1
While i $\leq |A|$ and j $\leq |B|$
    If $a_i > b_j$ then
        Append b_j to L
        Count <- Count + ($|A|$ - i)
        j <- j + 1
    else
        Append a_i to L
        i <- i + 1
Return (Count , L)

Run time?

T(n) = 2T(n/2) + cn
for some constant c

Thus: O(n log n)

Divide-and-conquer so far

- Merge-Sort algorithm for sorting
  - Trick is that we can merge two sorted lists efficiently (linear)

- Sort-and-Count algorithm for counting inversions
  - Trick is that given two sorted lists we can count inversions efficiently (linear)

- Binary search sometimes called divide-and-conquer

Claim. ABL(T')=ABL(T)-f_ω

Pf.

$$
\begin{aligned}
\mathrm{ABL}(T) &= \sum_{x\in S} f_x \cdot \mathrm{depth}_T(x) \\
&= f_y \cdot \mathrm{depth}_T(y) + f_z \cdot \mathrm{depth}_T(z) + \sum_{x\in S, x\neq y,z} f_x \cdot \mathrm{depth}_T(x) \\
&= \left(f_y + f_z\right)\cdot\left(1+\mathrm{depth}_T(\omega)\right) + \sum_{x\in S, x\neq y,z} f_x \cdot \mathrm{depth}_T(x) \\
&= f_\omega \cdot\left(1+\mathrm{depth}_T(\omega)\right) + \sum_{x\in S, x\neq y,z} f_x \cdot \mathrm{depth}_T(x) \\
&= f_\omega + \sum_{x\in S'} f_x \cdot \mathrm{depth}_{T'}(x) \\
&= f_\omega + \mathrm{ABL}(T')
\end{aligned}
$$

[Credit: https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pearson/04Huffman.pdf]