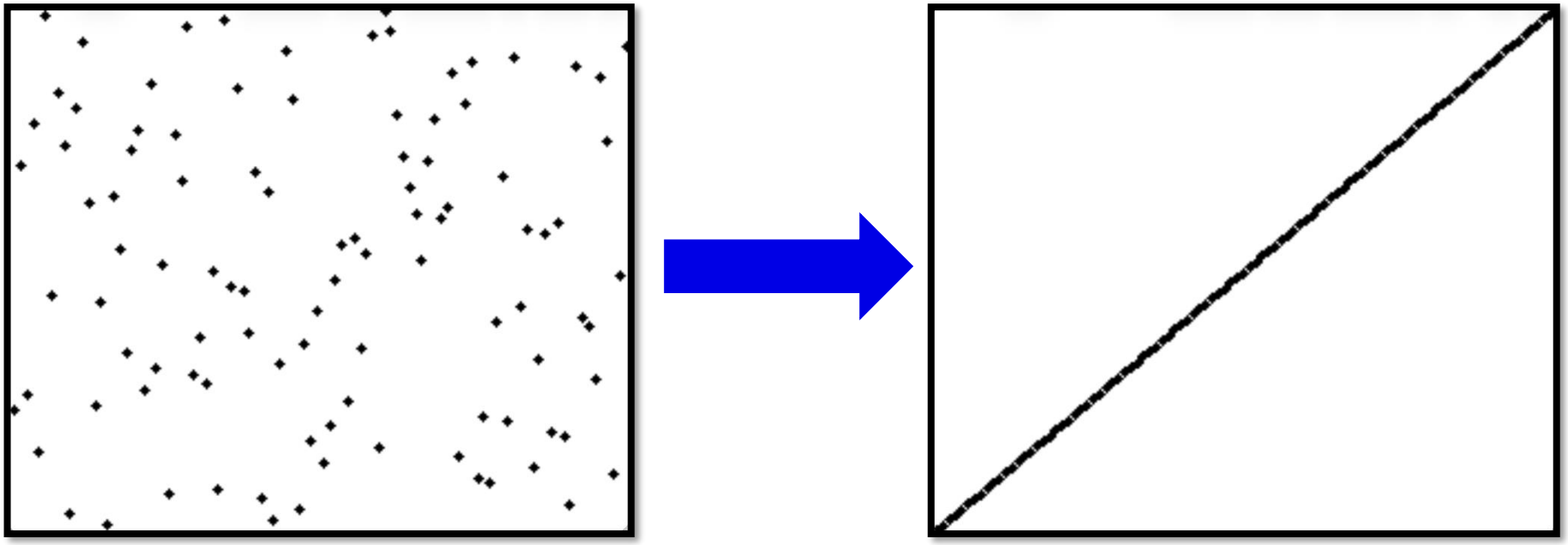
Entropy-Sensitive Sorting

Michael T. Goodrich
University of California, Irvine

Some slides adapted from material from Ian Munro, Sabastian Wild, and Wikipedia.

Recall the Sorting Problem

- Given an array, A, of n elements, reorder the elements to be in nondecreasing order.



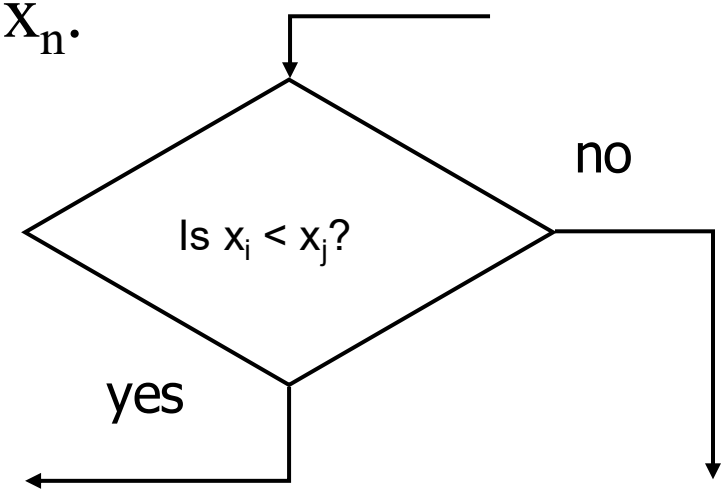
Worst-Case Running Time for Sorting

- There are several comparison-based algorithms that sort in $O(n \log n)$ time in the worst case:
  - Heap-sort
  - Merge-sort
  - Quick-sort (runs in $O(n \log n)$ time with high probability)

- **Note**: if we can assume that all keys are integers in the range from 0 to N (or even N^c for some constant, c), we can sort in $O(n + N)$ time using radix-sort, but this is not a comparison-based algorithm.

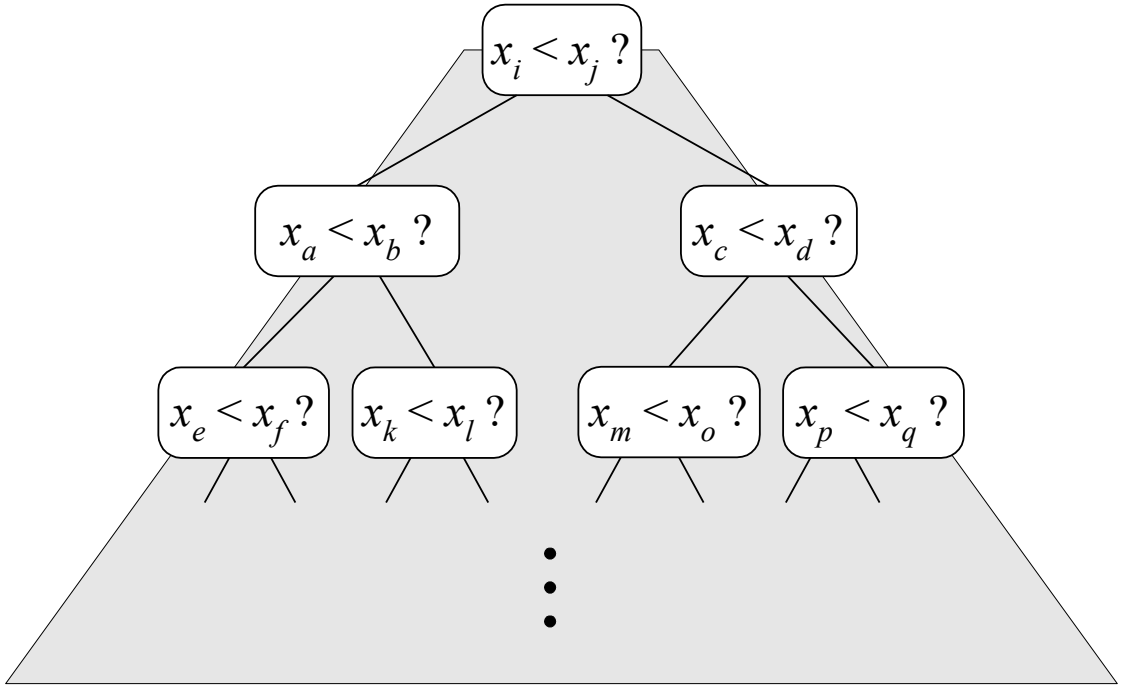
Comparison-Based Sorting

- Many sorting algorithms are comparison based.
  - They sort by making comparisons between pairs of objects
  - Examples: bubble-sort, selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort, ...
- Let us therefore derive a lower bound on the running time of any algorithm that uses comparisons to sort n elements, $x_1, x_2, \ldots, x_n$.



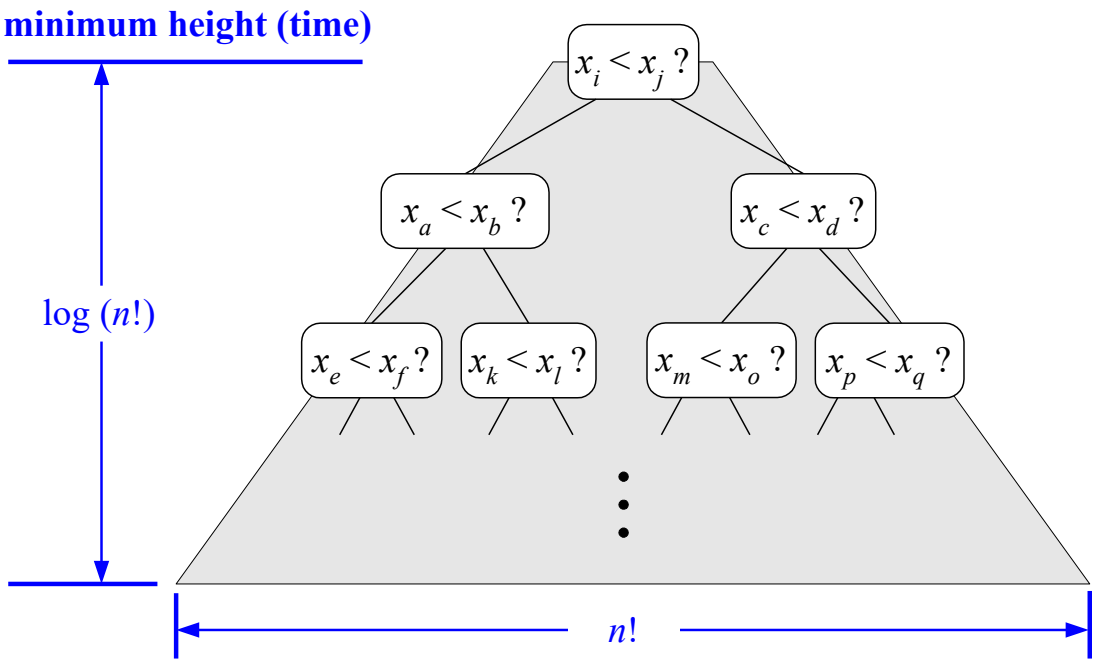
Counting Comparisons

- Let us just count comparisons then.
- Each possible run of the algorithm corresponds to a root-to-leaf path in a decision tree



Decision Tree Height

- The height of the decision tree is a lower bound on the running time
- Every input permutation must lead to a separate leaf output
- If not, some input …4…5… would have same output ordering as …5…4…, which would be wrong
- Since there are n!=1·2 · … ·n leaves, the height is at least log (n!)



Sorting Lower Bound

- Any comparison-based sorting algorithms takes at least log $n!$ time (assuming all permutations are possible).
- **Stirling's Approximation**:

$$\ln n! = n \ln n - n + O(\ln n)$$

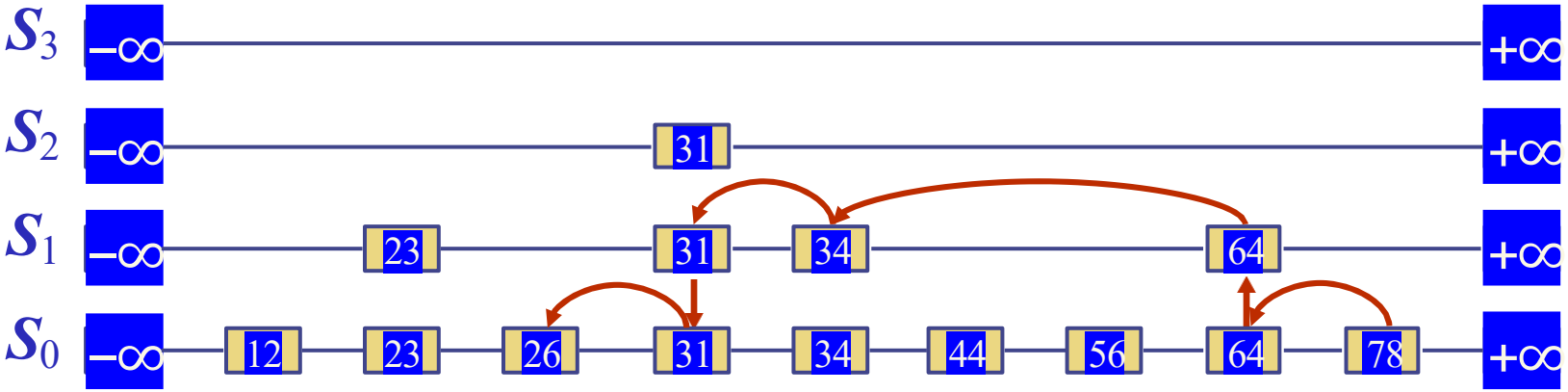
or

$$\log_2 n! = n \log_2 n - n \log_2 e + O(\log_2 n).$$

- Therefore, any comparison-based sorting algorithm must run in $\Omega(n \log n)$ time.

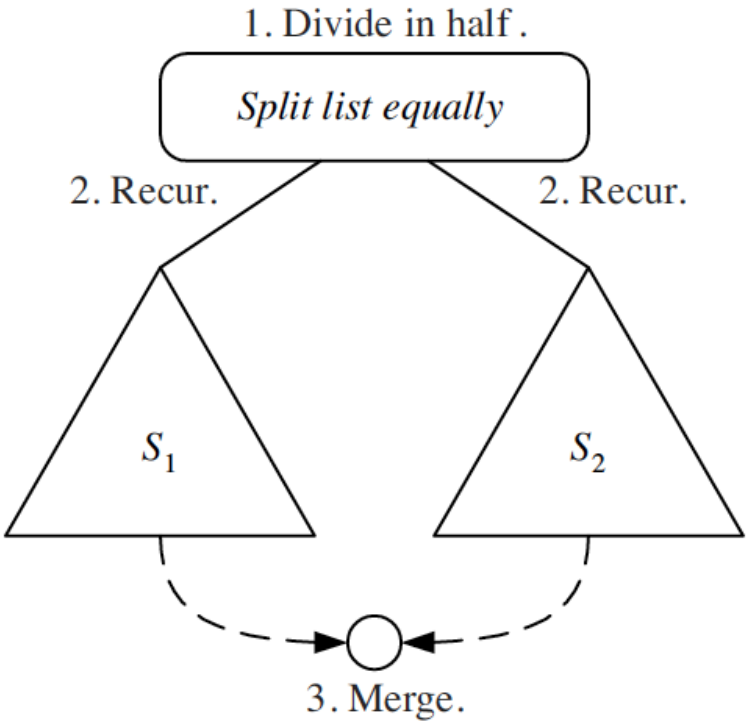
Skip-List Insertion Sort

- Expected running time is O(n(1+ log (1+Inv(A)/n))).
- Can beat the worst-case sorting lower bound in inputs with a smaller than worst-case number of inversions.



Review: Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two disjoint subsets S_1 and S_2
  - Recur: solve the subproblems associated with S_1 and S_2
  - Conquer: combine the solutions for S_1 and S_2 into a solution for S
- The base case for the recursion are subproblems of size 0 or 1



The Merge-Sort Algorithm

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

Algorithm *mergeSort*(S)
  Input sequence S with n elements
  Output sequence S sorted according to C
  if $S.size() > 1$
    $(S_1, S_2) \leftarrow partition(S, n/2)$
    mergeSort(S_1)
    mergeSort(S_2)
    $S \leftarrow merge(S_1, S_2)$

Merging Two Sorted Sequences

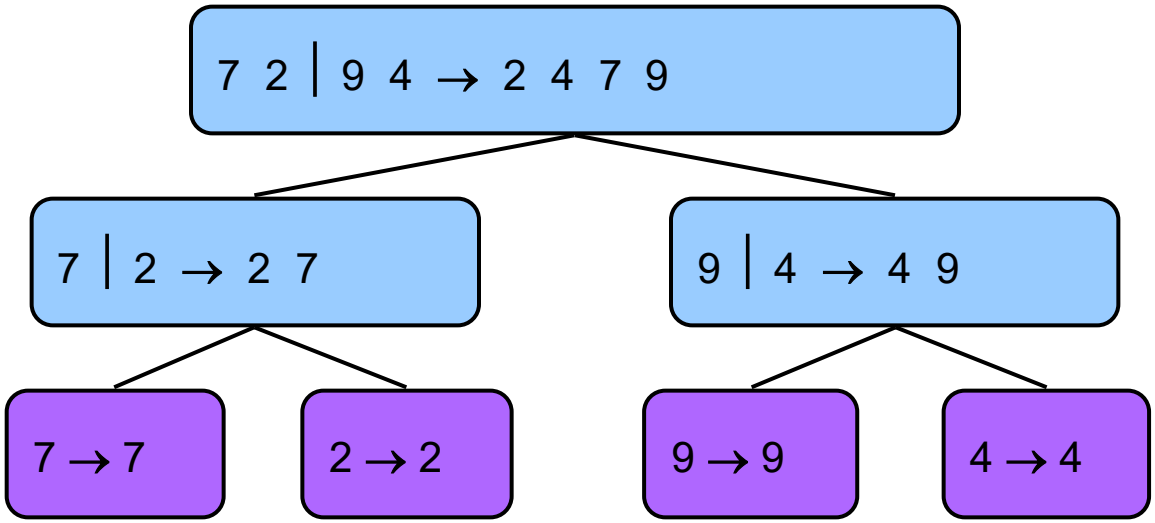
- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

Algorithm merge(S_1, S_2, S):
Input: Two arrays, S_1 and S_2, of size n_1 and n_2, respectively, sorted in non-decreasing order, and an empty array, S, of size at least $n_1 + n_2$
Output: S, containing the elements from S_1 and S_2 in sorted order

$i \leftarrow 1$
$j \leftarrow 1$
while $i \leq n$ **and** $j \leq n$ **do**
  if $S_1[i] \leq S_2[j]$ **then**
    $S[i + j - 1] \leftarrow S_1[i]$
    $i \leftarrow i + 1$
  else
    $S[i + j - 1] \leftarrow S_2[j]$
    $j \leftarrow j + 1$
while $i \leq n$ **do**
  $S[i + j - 1] \leftarrow S_1[i]$
  $i \leftarrow i + 1$
while $j \leq n$ **do**
  $S[i + j - 1] \leftarrow S_2[j]$
  $j \leftarrow j + 1$

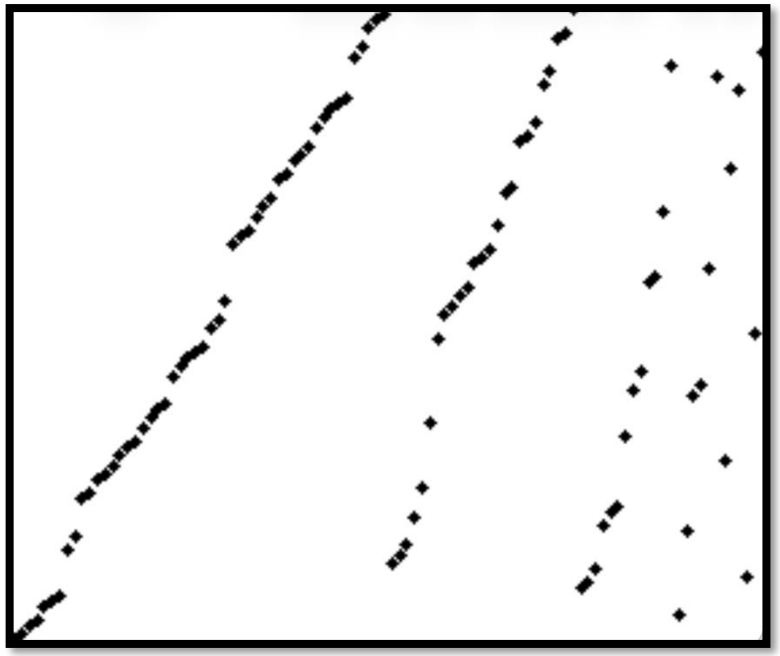
Merge-sort Tree

- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
  - the leaves are calls on subsequences of size 0 or 1
- The sum of the cost across all calls on a single level is $O(n)$.
- There are O(log n) levels; hence, the total time is $O(n \log n)$.

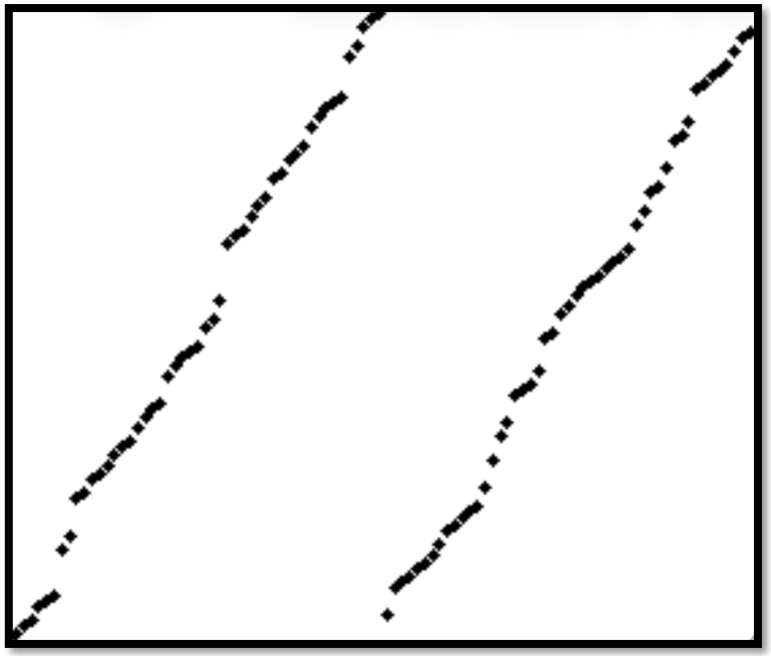


Pre-sortedness Based On Runs

- A **run** is a sequence of non-decreasing contiguous cells in the input array.



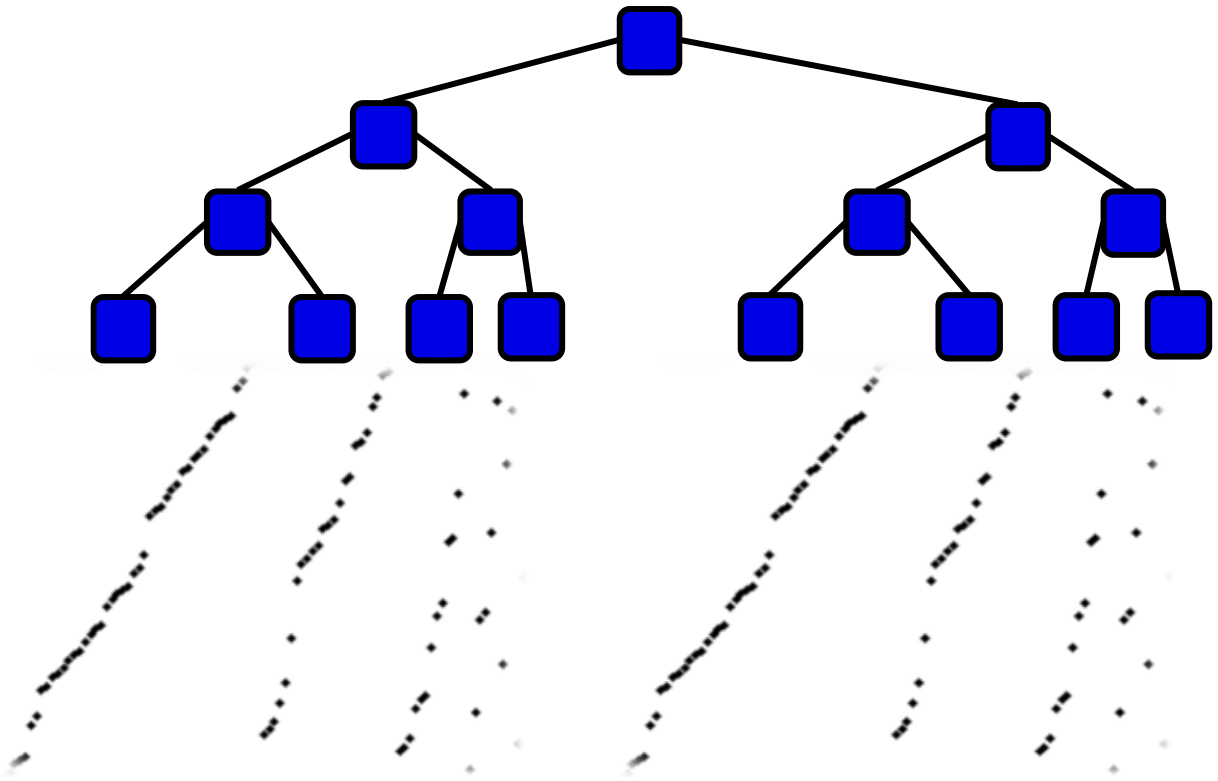
6 runs: some long, some short



2 runs: both long

Natural Merge-Sort

- The term "**natural merge-sort**" was coined by Don Knuth to refer to a merge-sort algorithm that starts with the maximal runs in an input array as the leaf-level inputs.
- All the maximal runs can be found in $O(n)$ time.



This natural merge-sort algorithm takes $O(n \log R)$ time, where R is the number of runs.

But it is not sensitive to the fact that some runs are long and some are short.

Run-Based Entropy

If an array is decomposed into R runs with lengths $L_1, L_2, \ldots, L_R$, the run-based entropy $H(R)$ is calculated based on the relative lengths of these runs:

$$H(R) = \sum_{i=1}^{R} \frac{L_i}{n} \log\left(\frac{n}{L_i}\right)$$

Where:

- n is the total number of elements.
- L_i is the length of the i-th run.

Entropy Sorting Lower Bound

- **Theorem**: The running time for sorting a sequence of size n is $\Omega(n\,H(R))$.

- Proof uses the multinomial coefficient for the number of permutations in the set, W, of all possible inputs:

Suppose an array of length n is composed of R runs with lengths $L_1, L_2, \ldots, L_R$.

If the elements within each run are already sorted, the only "disorder" comes from how these runs are interleaved. The number of ways to form a sequence of length n using R pre-sorted blocks of lengths $L_1, L_2, \ldots, L_R$ is given by the **multinomial coefficient**:

$$\text{Permutations}(W) = \frac{n!}{L_1!L_2!\ldots L_R!}$$

Proof of Entropy-Based Sorting Lower Bound

$$\text{Lower Bound} \geq \log_2 \left(\frac{n!}{\prod_{i=1}^{R} L_i!} \right)$$

Using **Stirling's Approximation** ($\ln n! \approx n \ln n - n$), we can simplify the logarithm of the multinomial coefficient:

$$\log_2(W) \approx (n \log_2 n - n) - \sum_{i=1}^{R} (L_i \log_2 L_i - L_i)$$

Since $\sum L_i = n$, the "$-n$" terms cancel out:

$$\log_2(W) \approx n \log_2 n - \sum_{i=1}^{R} L_i \log_2 L_i$$

Completing the Proof

To get to the entropy formula, we manipulate the sum. Notice that $n = \sum L_i$. We can rewrite $n \log_2 n$ as $\sum(L_i \log_2 n)$:

$$\log_2(W) \approx \sum_{i=1}^{R} L_i \log_2 n - \sum_{i=1}^{R} L_i \log_2 L_i$$

$$\log_2(W) \approx \sum_{i=1}^{R} L_i (\log_2 n - \log_2 L_i)$$

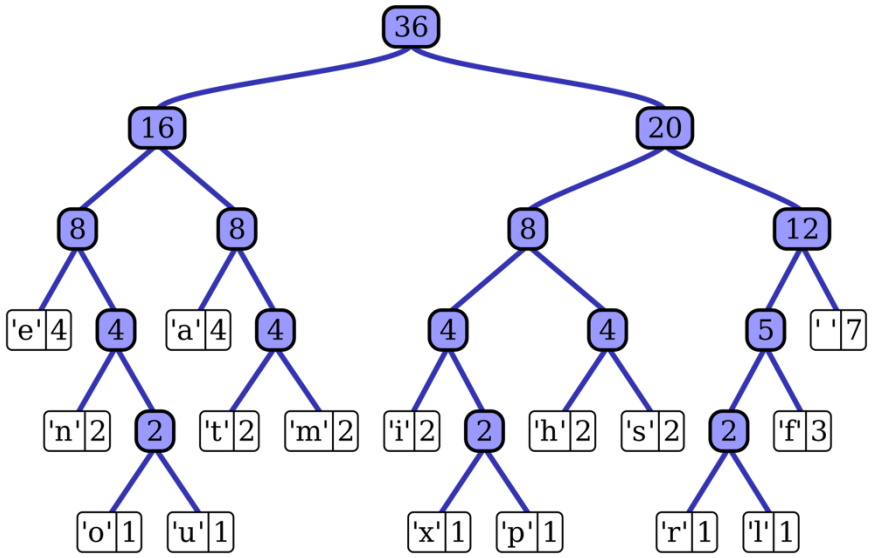
$$\log_2(W) \approx \sum_{i=1}^{R} L_i \log_2 \left(\frac{n}{L_i} \right)$$

If we multiply and divide the inside by n, we can pull n out of the summation:

$$\text{Lower Bound} \approx n \cdot \sum_{i=1}^{R} \frac{L_i}{n} \log_2 \left(\frac{n}{L_i} \right) \quad = \quad n \cdot H(R)$$

Building a Better Merge-sort Tree

- Let's try to design a better merge-sort algorithm by building a better merge-sort tree, which is sensitive to the run-based entropy.
- Idea: Use a Huffman tree, where the "frequencies" or "probabilities" assigned to leaves are the run sizes.



Public domain image of a Huffman tree from Wikipedia.

Huffman Tree Idea: Good News

- Huffman tree method: Given a set of symbols and their probabilities, put each into a leaf subtree and repeatedly combine the two subtrees with smallest weights.
- The sum of the depths of all the leaves matches the entropy bound.

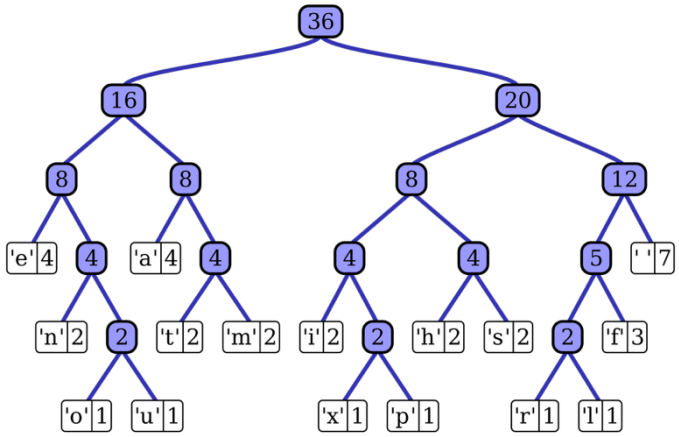
As defined by Shannon (1948), the information content h (in bits) of each symbol a_i with non-null probability is

$$h(a_i) = \log_2 \frac{1}{w_i}.$$

The entropy H (in bits) is the weighted sum, across all symbols a_i with non-zero probability w_i, of the information content of each symbol:

$$H(A) = \sum_{w_i > 0} w_i h(a_i) = \sum_{w_i > 0} w_i \log_2 \frac{1}{w_i} = -\sum_{w_i > 0} w_i \log_2 w_i.$$

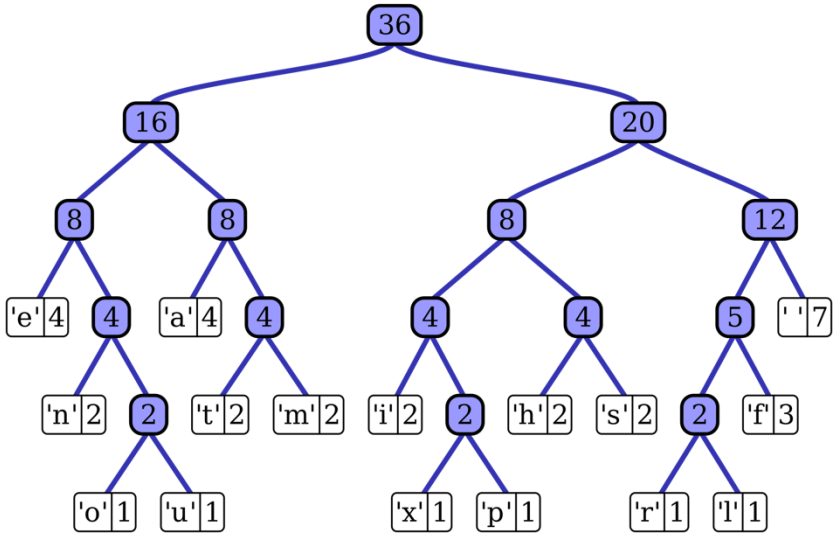
For us, the "probability" for leaf i is L_i/n.



Public domain image of a Huffman tree from Wikipedia.

Huffman Tree Idea: Bad News

- The textbook algorithm for building a Huffman tree takes O(n log n) time, where n is the number of leaves.
- In our case, we have R leaves, so the time needed to build the Huffman tree is $O(R \log R)$, which is too large, e.g., if R is $\Omega(n)$.



Public domain image of a Huffman tree from Wikipedia.

Huffman Tree Idea: The Fix

- Since run lengths are integers between 1 and n, we can sort the runs by their lengths in $O(n)$ time by radix-sort. Then we can construct the Huffman tree in $O(R)$ time.
- If the symbols (i.e., runs) are sorted by probability (i.e., run lengths), there is a linear-time method to create a Huffman tree using two queues, the first one (Queue A) containing the initial weights (along with pointers to the associated leaves), and combined weights (along with pointers to the trees) being put in the back of the second queue (Queue B).
- This assures that the lowest weight is always kept at the front of one of the two queues, and the second lowest weight is right after the lowest weight in the same queue or it is at the front of the other queue.

Huffman Tree Idea: The Fixed Algorithm

1. Start with as many leaves as there are symbols.
2. Enqueue all leaf nodes into Queue A (by probability in increasing order so that the least likely item is in the head of the queue).
3. While there is more than one node in the queues:
    1. Dequeue the two nodes with the lowest weight by examining the fronts of both queues.
    2. Create a new internal node, with the two just-removed nodes as children (either node can be either child) and the sum of their weights as the new weight.
    3. Enqueue the new node into the rear of Queue B.
4. The remaining node is the root node; the tree has now been generated.

Huffman Tree Idea: Proof

- The reason this algorithm works:
  - Queue A is sorted by definition (the input).
  - Queue B will naturally be sorted. Because we always pick the two smallest available values to create a new sum, each new sum added to Queue B will be equal to or greater than the previous sum added to Queue B.
- Since we only ever look at the front of the queues and add to the back, every operation is $O(1)$. We perform this R - 1 times, resulting in a total complexity of $O(R)$.
- **Drawbacks**: We need to explicitly build the merge tree, and the order of merges is likely different than the ordering of the runs (which is bad for cache misses and can cause the sorting method to not be stable).

Weight-Balanced Binary Search Trees

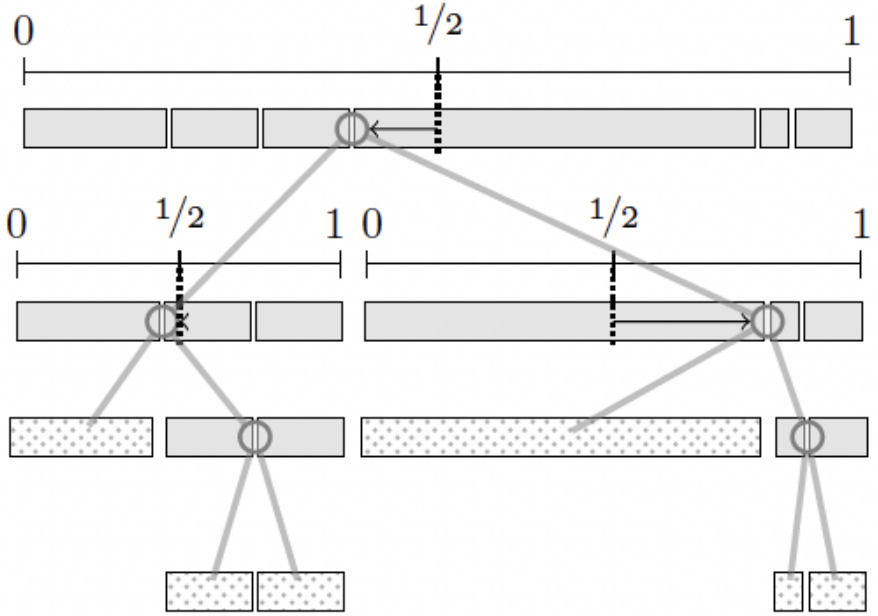
- Suppose we want to build a binary search trees on weighted elements, such that element i has weight w_i and the depth of each element is $O(\log W/w_i)$, where W is the sum of the weights.
- Then taking $w_i = L_i$, the sum of the depths of all the leaf elements in each run is proportional to

$$\sum_{i=1}^{R} L_i \log_2\left(\frac{n}{L_i}\right) \quad = \quad n \cdot \sum_{i=1}^{R} \frac{L_i}{n} \log_2\left(\frac{n}{L_i}\right) \quad = \quad n \cdot H(R)$$

- Thus, using such a tree as a merge tree gives an entropy-sensitive sorting algorithm.

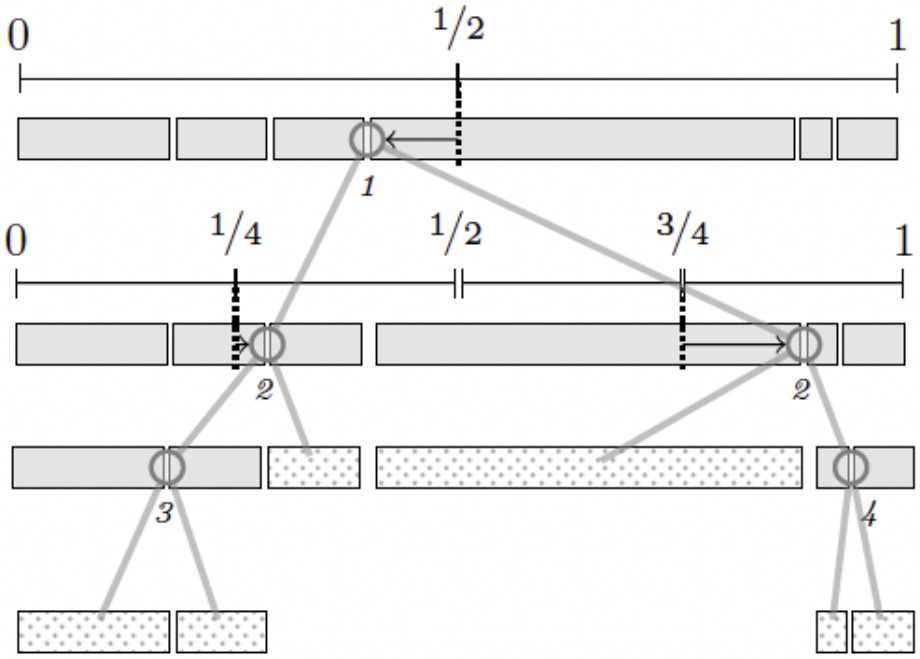
Method 1 for a Weighted BST

- Normalize run lengths to sum to 1.
- Choose the split closest to the midpoint of the subtree's actual weights (1/2 after (re)normalizing)
- Recursively split the left and right children (renormalizing based on the sum of the sizes of their respective sets of runs).



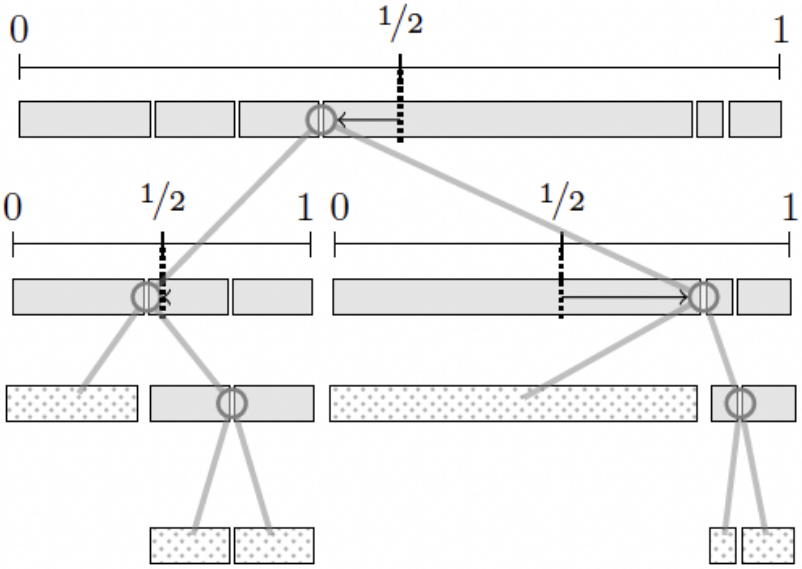
Method 2 for a Weighted BST

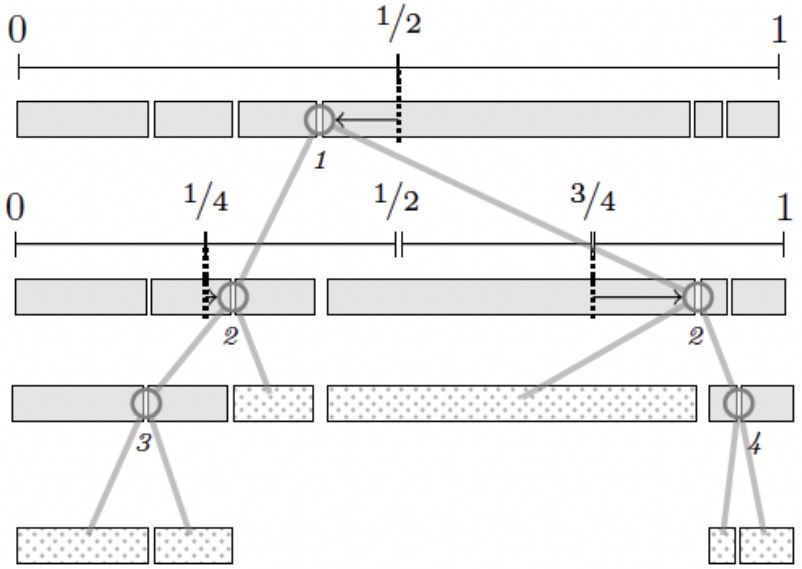
- Normalize run lengths to sum to 1.
- Choose the split closest to the midpoint of the subtree's assigned subinterval.
- Recursively split the left and right children (but don't renormalize weights).



Method 1 and 2 Both Work

- Both Method 1 and 2 build a binary search tree on weighted elements, such that element i has weight w_i and the depth of each element is $O(\log W/w_i)$, where W is the sum of the weights.





Method 1 Leads to Peeksort

```
PEEKSORT(A[ℓ..r], e, s)
 1  if e == r or s == ℓ then return
 2  m := ℓ + ⌊(r−ℓ)/2⌋
 3  if m ≤ e                    //  ℓ | m | e | s | r
 4      PEEKSORT(A[e+1..r], e+1, s)
 5      MERGE(A[ℓ..e], A[e+1..r])
 6  else if m ≥ s               //  ℓ | e | m | s | r
 7      PEEKSORT(A[ℓ..s−1], e, s−1)
 8      MERGE(A[ℓ..s−1], A[s..r])
 9  else
        // Find existing run A[i..j] containing position m
10      i := EXTENDRUNLEFT(A[m], ℓ);   j := EXTENDRUNRIGHT(A[m], r)
11      if i == ℓ and j == r return
12      if m − i < j − m    //  ℓ | e | m | i | j | s | r
13          PEEKSORT(A[ℓ..i−1], e, i−1)
14          PEEKSORT(A[i..r], j, s)
15          MERGE(A[ℓ..i−1], A[i..r])
16      else                //  ℓ | e | m | i | j | s | r
17          PEEKSORT(A[ℓ..j], e, i)
18          PEEKSORT(A[j+1..r], j+1, s)
19          MERGE(A[ℓ..j], A[j+1..r])
```

Algorithm 1 Peeksort: A simple top-down version of nearly-optimal natural mergesort. The initial call is PEEKSORT($A[1..n], 1, n$). Procedures EXTENDRUNLEFT (-RIGHT) scan left (right) starting at $A[m]$ as long as the run continues (and we did not cross the second parameter).

Method 2 Leads to Powersort

PowerSort($A[1..n]$)

```
1  X := stack of runs                (capacity ⌊lg(n)⌋ + 1)
2  P := stack of powers              (capacity ⌊lg(n)⌋ + 1)
3  s1 := 1;  e1 = ExtendRunRight(A[1], n)    // A[s1..e1] is leftmost run
4  while e1 < n
5        s2 := e1 + 1;  e2 := ExtendRunRight(A[s2], n)    // A[s2..e2] next run
6        p := NodePower(s1, e1, s2, e2, n)    // Pj for node (j) between A[s1..e1] and A[s2..e2]
7        while P.top() > p      // previous merge deeper in tree than current
8              P.pop()          // ⇝ merge and replace run A[s1..e1] by result
9              (s1, e1) := Merge(X.pop(), A[s1..e1])
10       X.push(A[s1, e1]);  P.push(p)
11       s1 := s2;  e1 := e2
12 end while // Now A[s1..e1] is the rightmost run
13 while ¬X.empty()
14       (s1, e1) := Merge(X.pop(), A[s1..e1])
```

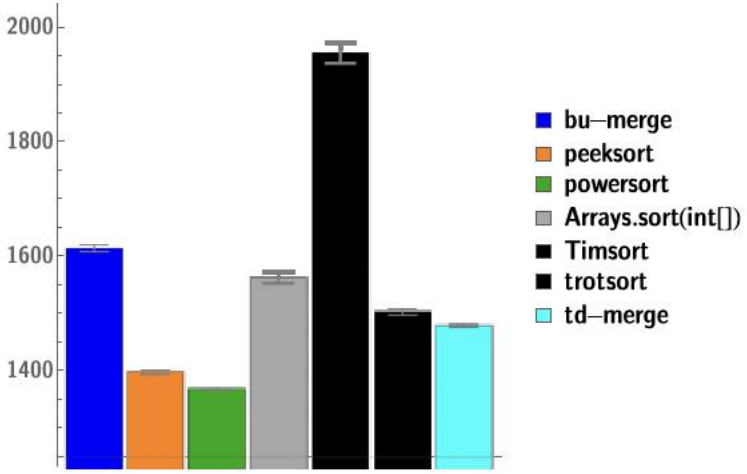
NodePower(s_1, e_1, s_2, e_2, n)

```
1  n1 := e1 − s1 + 1;  n2 := e2 − s2 + 1;  ℓ := 0
2  a := (s1 + n1/2 − 1)/n;  b := (s2 + n2/2 − 1)/n
3  while ⌊a · 2^ℓ⌋ == ⌊b · 2^ℓ⌋ do ℓ := ℓ + 1 end while
4  return (ℓ)
```

Algorithm 2 Powersort: A one-pass stack-based nearly-optimal natural mergesort. Procedure ExtendRunRight scans right as long as the run continues.

Both Are Good, Powersort is a Little Better

- Both Peeksort and Powersort run in time $O(n(1+H(R))$, and both are fast in practice.



- Powersort is now the reference sorting algorithm in the PyPy JIT Python complier, CPython, NumPy, and AssemblyScript.