SORTING

- Very important problem, most extensively studied
- In many applications, sorting consumes a large proportion of computing time!
- What sorting algorithms have you learned?
  - Bucket Sort and Radix Sort
  - Insertion Sort and Selection Sort
  - Merge Sort
  - Quick Sort
  - Heap Sort
- We will review the above algorithms, do more detailed analysis and prove lower bounds

Bucket Sort

- "Mailroom sort": allocate a sufficient number of *boxes* – *buckets* – and put each element in the corresponding bucket.
- Works very well only for elements from a small, simple range that is known in advance

  † e.g. sorting letters by state (by province)

  † e.g. sorting letters by zip code – we need $26^3 \cdot 10^3$ buckets!!
- Input $x_1, x_2, \ldots, x_n$, $1 \leq x_i \leq m$ and x_i are distinct integers.

  Allocate m buckets.

  For each i, we put x_i in the bucket corresponding to its value.

  Finally, we scan the buckets in order and collect all elements.
- Time and space complexity:

  time: $O(n+m)$: n for sort n elements and m for final scan

  space: $O(m)$: 1 unit for each bucket

  If $m = O(n)$ then this is linear sorting

Radix Sort

- Natural extension of bucket sort.
- We want to reduce the number of buckets (we need more passes).
- Assume that the elements are large integers represented by k digits, and each digit is in the range 0 to $d-1$.
- We use induction to show the algorithm.

Induction Hypothesis: We know how to sort elements of $< k$ digits

† Given elements with k digits, we first ignore the most significant digit (left-most digit) and sort the elements according to the rest of the digits by induction!

† Scan all the elements again and use bucket sort on the most significant digit with d buckets.

† Collect all the buckets in order.

Example: $n = 10, d = 10, k = 2$

Input: 36, 9, 0, 25, 1, 49, 64, 16, 81, 4

(first pass) Bucket	Contents	(second pass) Bucket	Contents
0	0	0	$0, 1, 4, 9$
1	$1, 81$	1	16
2		2	25
3		3	36
4	$64, 4$	4	49
5	25	5	
6	$36, 16$	6	64
7		7	
8		8	81
9	$9, 49$	9	

Append 10 queues of first pass: 0, 1, 81, 64, 4, 25, 36, 16, 9, 49 (input to second pass).
Append 10 queues of second pass: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

Why does it work?

1. Two elements that are put in different buckets in the LAST step are in the right order

- do not need induction
- most significant digits determine the order

2. Two elements having the same most significant digit

- By induction, they are in right order before the last step.
- Make sure that elements put in the same bucket REMAIN in the same order
  - using a queue for each bucket
  - appending the d queues at the end of a stage to form one global queue of all elements

(This shows how to use induction to make sure the algorithm is correct.)

Time complexity: $O(kn)$

- Initialize the queues: $O(d)$
- Put n elements into buckets: $O(n)$
- Append d queues: $O(d)$
- Therefore for one pass, total time is $O(n)$

Counting sort can be used to implement Radix Sort.

Counting Sort

Counting sort assumes that each of the n input is an integer in the range of 0 to k, for some k.

The input is in $A[1..n]$ and the output will be in $B[1..n]$.

We also use an array $C[0..k]$ for temporary space.

Counting-Sort(A, B, k)

```
for i = 0 to k
        do C[i] = 0
for j = 1 to n
        do C[A[j]] = C[A[j]] + 1
for i = 1 to k
        do C[i] = C[i] + C[i - 1]
for j = n downto 1
        do B[C[A[j]]] = A[j]
           C[A[j]] = C[A[j]] - 1
```

- An example that sorting is not based on comparison.
  - Lines 1 to 2 initialize array $C[\,]$.
  - Lines 3 to 4 calculate the number of integers in the input with value i for each i in the range of 0 and k.
  - Lines 5 to 6 calculate the correct location for the last integer with value i.
  - Lines 7 to 8 sort integers in array $A[\,]$ and put the result in array $B[\,]$.
- An important property of counting sort is that it is **stable**.
  - Integers with the same value appear in the output array exactly the same order as they do in the input array.
  - This is important in the application of counting sort.
- When $k = O(n)$, the sort runs in $\Theta(n)$ time.

Insertion Sort

- Assume we can sort $n-1$ elements, then we can find the right place for the n'th element and insert it there
- worst case:
  - data movement: $i-1$ for the i'th iteration $\Longrightarrow \Omega(n^2)$
  - comparisons: $\Omega(n \log n)$
- average case:
  - There are i positions where x (ith element) can go.
  - The probability that x belongs to any position is $1/i$.

$$\sum_{j=0}^{i-1}(1/i)j = (1/i)[i(i-1)/2] = (i-1)/2 \qquad i\text{th step}$$

$$A(n) = \sum_{i=2}^{n}[(i-1)/2] = O(n^2) \qquad \text{total}$$

Selection Sort

- find the maximum element, swap it with the last element.
- data movement: $O(n)$, 1 for each iteration
- comparisons: $O(n^2)$, i for ith largest

<u>Insertion and Selection</u>

Insertion	Data movement	$O(n^2)$
	Comparisons	$O(n\log n)$
Selection	Data movement	$O(n)$
	Comparisons	$O(n^2)$

To improve insertion sort:
Use a data structure that supports search and also insertion, for example AVL trees or red-black trees. These methods require extra space though.

To improve selection sort:
Use a data structure that supports find max and also deletions (e.g. heap sort).

Mergesort

The merge process can be considered as an improvement of insertion sort.

- *Idea:* With the time to insert one element, we can insert many elements.

Let $A = a_1, a_2, \ldots a_n$, $B = b_1, b_2, \ldots, b_m$ be sorted.

- We want to insert B into A
- We scan A from the left for the right position for b_1
- We can then continue, without going back, to scan for the right position for b_2 and so on.
- *Data movement:* copy them into a temporary array. Each element moves only once.
- $O(n + m)$ time

Mergesort: Divide-and-conquer sorting

Divide by half	$O(1)$
Solve each half recursively	$2T(n/2)$
Merge two sorted halves	$O(n)$

Time: $T(n) = 2T(n/2) + O(n) \Longrightarrow O(n\log n)$

QuickSort

```
Procedure Q-Sort(X, Left, Right)
begin
    if Left < Right then
        Middle = Partition(X, Left, Right);
        Q-Sort(X, Left, Middle-1);
        Q-Sort(X, Middle + 1, Right);
end
```

Partition

† choose a pivot, x_1

† use two pointers (indices) L and R

Initially, L points to the left end of the array and R points to the right end of the array.

The pointers move in opposite directions.

† *Induction hypothesis: At step k of the partition algorithm, pivot $\geq x_i$ for $i < L$ and pivot $< x_j$ for $j > R$.*

1. If $L \leq R$ then

– $x_L \leq$ pivot then $L \leftarrow L+1$, or
$x_R >$ pivot then $R \leftarrow R-1$

– $x_L >$ pivot and $x_R \leq$ pivot then exchange x_L and x_R, and $L \leftarrow L+1$, $R \leftarrow R-1$

2. If $L > R$ exchange x_1 and x_R
(2 = terminate condition)

By induction in step $k+1$, we can keep induction hypothesis and move either L or R

Consequently, the pointers will eventually meet termination condition

Read textbook pp. 146-148 for another partition algorithm.

How to choose a pivot?

– Choose a random element from the sequence is a good choice

– If the sequence is random, we can just choose the first element

– If we choose another element to be pivot, we can exchange it with the first element, then use our partition algorithm

```
Algorithm Partition(X, left, right);
Input: X (an array)
  left  (the left boundary of array X)
  right (the right boundary of array X)
Output: X and middle, such that
  X[i] <= X[middle] for all i <= middle,
  X[j] > X[middle] for all j > middle.
begin
  pivot := X[left]; L := left; R := right;
  while L <= R do
    while X[L] <= pivot and L <= right do
      L := L+1;
    while X[R] > pivot do
      R := R-1;
    if L < R then
      exchange X[L] and X[R]; L := L+1; R := R-1;
  middle := R;
  exchange X[left] and X[middle];
  return middle;
end;
```

Worst case

† The sequence is in the correct order.

$$W(n) = (n-1) + W(0) + W(n-1),$$

$$W(0) = W(1) = 1.$$

($(n-1)$ for partition, since the sequence is sorted we will have one empty sequence and one sequence with $(n-1)$ elements.)

Example: 3, 7, 9, 18, 20, 21

$$\begin{aligned} W(n) &= (n-1) + (n-2) + 2W(0) + W(n-2) \\ &\ldots \\ &= \textstyle\sum_{i=1}^{k} +(n-i) + kW(0) + W(n-k) \\ &= \textstyle\sum_{i=1}^{n-1} (n-i) + (n-1)W(0) + W(1) \\ &= \textstyle\sum_{i=1}^{n-1} i + n = n(n-1)/2 + n \approx n^2 \end{aligned}$$

Average Case

- Assume that all keys are distinct. All permutations are equally likely.
- Each x_i has the same probability of being selected as the pivot.
- Running time $T(n)$ of quicksort if the ith smallest element is the pivot is

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

  $n-1$ for partition

  $T(i-1)$ for sequence less than pivot

  $T(n-i)$ for sequences greater than pivot

- Since x_i has same probability to be pivot the average running time is

$$A(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}[A(i-1) + A(n-i)],$$

$$A(0) = 0, A(1) = 1.$$

$\boxed{\text{Note}}$

$$\sum_{i=1}^{n} A(n-i) = A(n-1) + A(n-2) + \cdots + A(0) =$$

$$\sum_{i=1}^{n} A(i-1).$$

which implies

$$A(n) = (n-1) + \frac{2}{n} \sum_{i=1}^{n} A(i-1) \qquad (1 * *)$$

(recurrence with full history)

(1**) involves many $A(i)$'s in $A(n)$. We use a "trick" to reduce this to first order recurrence.

$$nA(n) = n(n-1) + 2\sum_{i=1}^{n} A(i-1) \quad (2**)$$

$$(n-1)A(n-1) = (n-1)(n-2) + 2\sum_{i=1}^{n-1} A(i-1) \quad (3**)$$

Now subtract (3**) from (2**):

$$nA(n) - (n-1)A(n-1) =$$

$$n(n-1) - (n-1)(n-2) + 2A(n-1)$$

$$A(n) = \frac{n+1}{n}A(n-1) + \frac{2(n-1)}{n} \qquad (4**)$$

Let $B(n) = A(n)/(n+1)$
(Second trick use a substitution.)

$$B(n) = B(n-1) + 2(n-1)/(n+1)n, \qquad n > 1.$$

$$B(n) = 2\sum_{i=2}^{n} \frac{i-1}{(i+1)i} + B(1)$$

$$B(n) \approx 2\sum_{i=1}^{n} \frac{1}{i+1} \approx 2\sum_{i=1}^{n} \frac{1}{i}$$

$$\sum_{i=1}^{n} \frac{1}{i} \approx \ln(n) = \frac{\log n}{\log e}$$

$$B(n) \approx \frac{2}{\log e}\log n \approx 1.4\log n$$

Conclusion

$$A(n) = 1.4(n+1)\log n$$

$$A(n) = \Theta(n\log n)$$

Space Complexity

- From the appearance of the algorithm, it seems that we do not need any extra space

- However, recursions are implemented by using run-time stacks

  Each call: a pair of indices of the array has to be stacked.

- There are at most $n-1$ calls, there are at most $(n-1)$ pair of indices to be stacked.

- Space complexity: $O(n)$ (extra space)

- If we use explicit stack, we can guarantee $O(\log n)$ extra space

Improvements of the quicksort algorithm

1. Improve the selection of the pivot

   – choose a random index between L and R.

   – choose the median of x_L, x_R and $x_{(L+R)/2}$

   (We need to do extra work, but it is worth it.)

2. Use a simple algorithm for small size.

   – e.g. when size is less than 15, use insertion sort

   – avoid problem of stacking overhead ("choose the base of induction wisely")

3. Use explicit stacking : avoid overhead of system (run-time) stack

4. Minimize the size of the stack: always stack the larger part first (solve smaller part first).

5. Put pivot into register, for each comparison only one data movement from memory.

Heapsort

- Like selection sort, heapsort is in place
- Like mergesort, heapsort is $O(n \log(n))$
- heapsort combines the better features of the two sorting algorithms.
- heapsort
  - fast sorting algorithm
  - not quite as fast as quicksort
    but not much slower
  - unlike quicksort, its performance is guaranteed

$\boxed{\text{Heap Sort}}$

- *Build Heap*
- consider the largest element
  - Swap $A[1], A[n]$
  - $A[n]$ now has correct element
  - rearrange $A[1, \ldots n-1]$ to form a heap (push $A[1]$ down the tree).
- Assume $A[1, \ldots i+1]$ is a heap and $A[i+2], \ldots, A[n]$ have correct elements.
  - swap $A[i+1]$ and $A[1]$
  - $A[i+1]$ has now correct element
  - rearrange $A[1, \ldots i]$ to form a heap (push $A[1]$ down the heap).
  - time: $\sum_{i=1}^{n} \log(i) = O(n \log n)$
    (time for transforming a heap to a sorted sequence.)

Time complexity for heap sort

- Heap building
  - $2n$ comparisons
  - n data movements
- Heap sort
  - $2\sum_{i=2}^{n} \log i$ comparisons
  - $\sum_{i=2}^{n} \log i$ data movements
- $\sum_{i=2}^{n} \log i \leq n \log n - n$
  - $2n + 2\sum_{i=2}^{n} \log i \leq 2n \log n$ comparisons.
  - $n + \sum_{i=2}^{n} \log i \leq n \log n$ data movements.

Lower Bounds for Sorting Problem

- Insertion sort, Selection sort: $O(n^2)$.

  Mergesort, heapsort, (quicksort): $O(n\text{log}n)$.

  Is it possible to improve it even further?

- Lower bound for a Problem:

  A proof that NO Algorithm can solve the problem better.

  † Much harder to prove a lower bound for a problem since we have to consider ALL possible algorithms, not just one particular approach.

  † We need a model corresponding to an arbitrary (unspecified) algorithm

  And a proof that ANY algorithm that fits the model will has a running time higher than the lower bound.

  – Example:

    We cannot say we will use a special data structure for this problem. Because there may be an algorithm that do not use this data structure and runs faster.

- Decision tree model
  - † decision trees model computations that consists of comparisons.
  - † Many known lower bound proofs use decision tree model.
  - † As a computation model, decision tree model is weaker than Turing Machine or RAM model.

Decision tree model

- binary trees with two types of nodes:
  internal nodes: two children, *leaves:* no child.
  (Also called two-trees or 2-trees)
- Internal node: associated with a query, the outcome is one of two possibilities. Each one is associated to one of the branches.
- Leaf: associated with a possible output
- Input is a sequence of numbers: $x_1, x_2, \ldots, x_n$

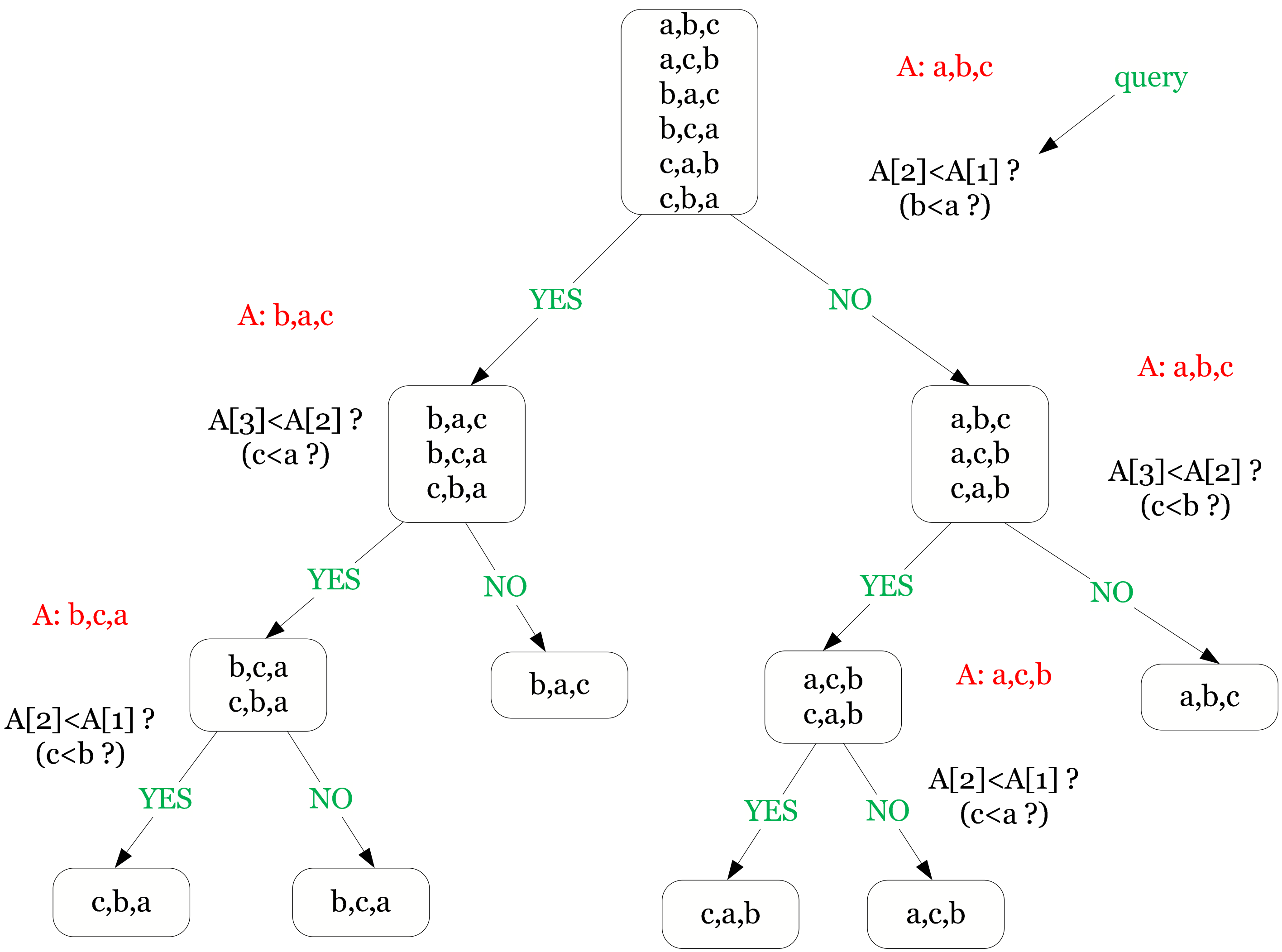
  Computation starts at the root of the tree.

  In each internal node, the query is applied.

  Either go left or go right depending on the result of the query.
- When reaching a leaf, the output associated with the leaf is the output of the computation.
- The worst-case running time of a tree T is the height of T. That is the maximum number of queries required by an input.

The Decision tree for insertion sort with n =3

Input: a, b, c. In array, $A[1, \ldots, 3]$



Lower bound for worst case

We want to find the lower bound of the height of a binary tree in terms of number of leaves.

- **Lemma:** Let l be the number of leaves in a binary tree and let h be its height. Then $l \leq 2^h$.

  Proof: Induction on h $\Box$.

- Let l and h be as in the Lemma. Then $h \geq \lceil \log l \rceil$. From the Lemma, $l \leq 2^h \Longrightarrow \log l \leq h \Longrightarrow h \geq \lceil \log l \rceil$ (since h is an integer).

- A binary tree with $n!$ leaves has a height greater than $n\log n - 1.5n$

$$\begin{aligned} h &\geq \log(n!) \geq \log(n(n-1)(n-2)\ldots(\lceil n/2 \rceil)) \\ &\geq \log([n/2]^{n/2}) \geq n/2 \ \log(n/2). \end{aligned}$$

  A closer lower bound is

$$h \geq \log(n!) \geq n\log n - 1.5n.$$

Theorem. Every decision tree algorithm for sorting has height $\Omega(n\log n)$.

Proof:

- input for sorting is $x_1, x_2, \ldots, x_n$
- output is a sorted sequence, or is a permutation of input!

  (tell us how to rearrange input such that they become sorted.)
- every permutation is a possible output (input can be in any order)
- every permutation of $(1, 2, \ldots, n)$ must be represented as an output in the decision tree for sorting.

  (Otherwise sorting algorithm is not correct!)
- two different permutations represent two different outputs. They must be associated with different leaves.
- total number of permutations is $n!$
- height of the tree is at least $\log(n!) \geq cn\log n$
- height is $\Omega(n\log n)$. $\square$

Information-theoretic lower bound

- The lower bound depends only on the amount of information contained in the output.
- It needs to distinguish between $n!$ different outputs; it can only distinguish two possibilities at a time
- Encoding $n!$ possibilities needs $\log(n!)$ bits
- We have not even defined the kind of query we allow
- This lower bound only implies:

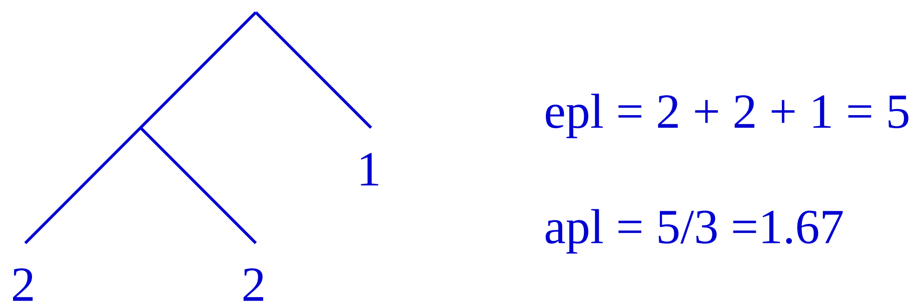
NO-COMPARISON-BASED sorting algorithms can be faster than $\Omega(n\log n)$

Lower bound for average behavior

Is it possible to find a comparison algorithm for sorting which has an average behavior better than $n\log n$?

Answer: **NO.**

- epl (external path length) = sum of the length of all paths from the root to a leaf.
- apl (average path length) = epl/ (number of leaves)
- *Example:*
  epl$= 2 + 2 + 1 = 5$ while apl$= 5/3 \approx 1.67$



Lemma. Among 2-trees with l leaves, the epl is minimal only if all the leaves are on at most two adjacent levels.

Proof. Suppose we have a 2-tree of height h that has a leaf x at level $k \leq h-2$.

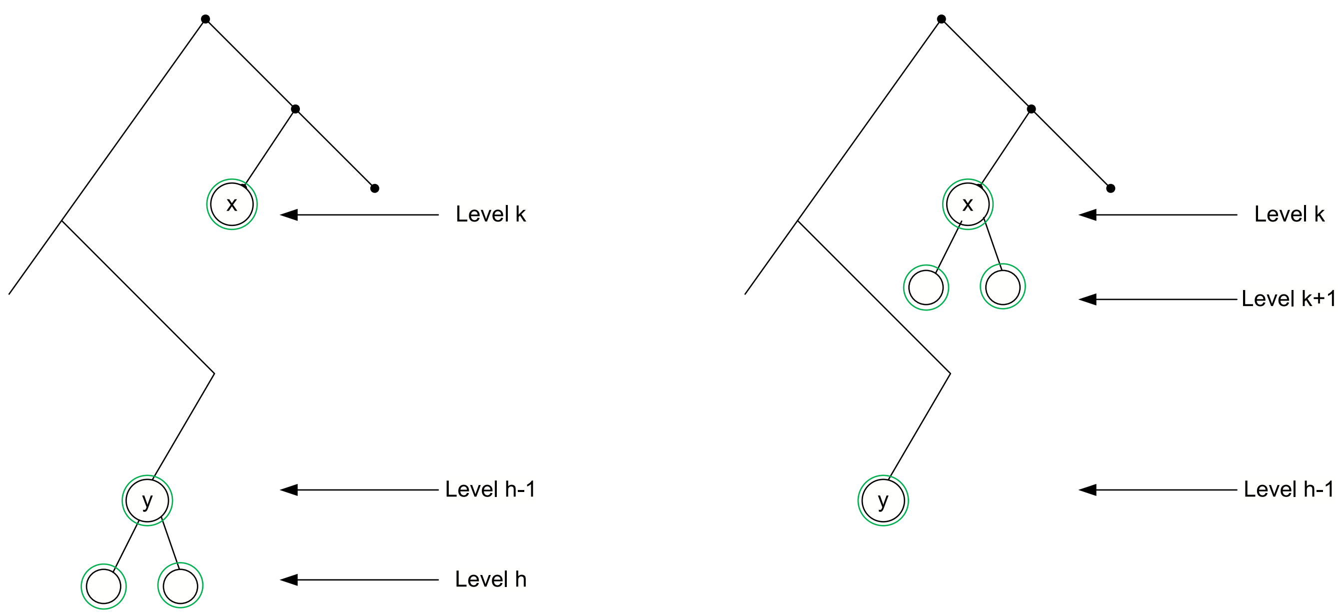
– choose a node y in level $h-1$ that is not a leaf

– remove children of y, attach them to x

– the total number of leaves is the same

– net decrease of epl is

$$2h + k - (h - 1 + 2(k+1)) = h - 1 - k > 0 \quad (\text{ since } k \leq h-2)$$

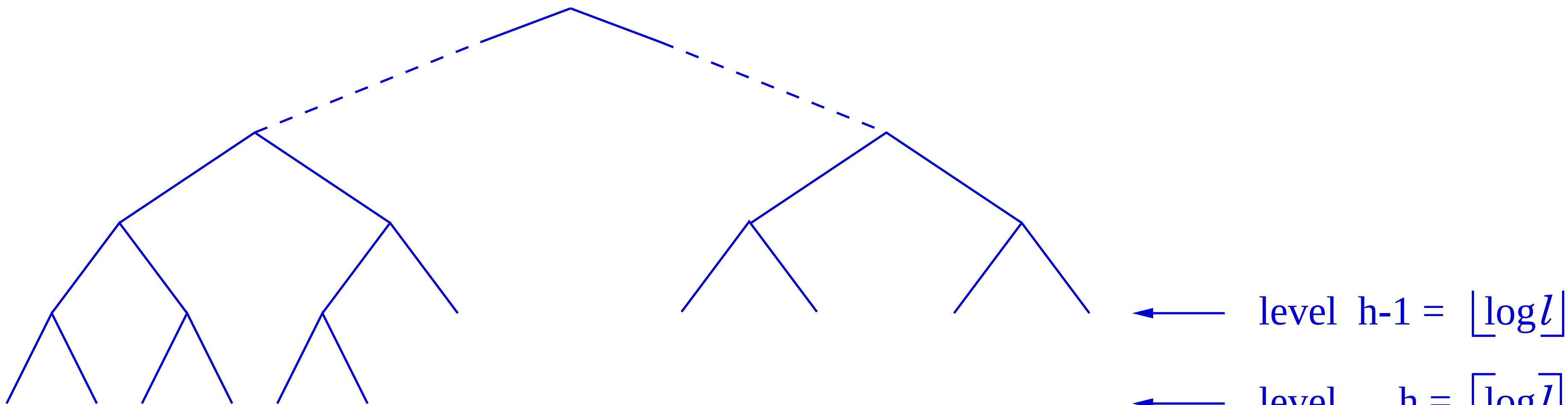


Lemma. The minimum epl for 2-tree with l leaves is

$$l\lfloor \log l \rfloor + 2(l - 2^{\lfloor \log l \rfloor})$$

Proof.

- From previous Lemma, we can consider only 2-trees of height h and leaves in levels $h-1$ and h.
- We can transform such a tree into a complete binary tree with possibly some of the right-most leaves removed WITHOUT changing the number of leaves and epl.



- If l is power of two, all leaves are at level log$l$. epl$= l \log l$
- If l is not a power of 2, the number leaves at level h is $2(l - 2^{h-1})$

  (each node in level $h-1$ that is not a leaf has two children)
- epl $= l(h-1) + 2(l - 2^{h-1}) = l\lfloor \log l \rfloor + 2(l - 2^{\lfloor \log l \rfloor})$

  $\square$

Lemma. The average path length in a 2-tree with l leaves is at least $\lfloor \log l \rfloor$.

Proof. The minimum average path length is:

$$\frac{l\lfloor \log l \rfloor + 2(l - 2^{\lfloor \log l \rfloor})}{l} =$$

$$= \lfloor \log l \rfloor + 2\frac{l - 2^{\lfloor \log l \rfloor}}{l} =$$

$$= \lfloor \log l \rfloor + \epsilon, \qquad 0 \leq \epsilon < 1. \quad \square$$

$$
\begin{aligned}
l/2 < 2^{\lfloor \log l \rfloor} \leq l \\
&\implies -l/2 > -2^{\lfloor \log l \rfloor} \geq -l \\
&\implies l - l/2 > l - 2^{\lfloor \log l \rfloor} \geq l - l \\
&\implies l/2 > l - 2^{\lfloor \log l \rfloor} \geq 0 \\
&\implies 1/2 > [l - 2^{\lfloor \log l \rfloor}]/l \geq 0 \\
&\implies 1 > 2[l - 2^{\lfloor \log l \rfloor}]/l \geq 0
\end{aligned}
$$

Theorem. The average number of comparisons done by an algorithm to sort n numbers by comparison is at least $\lfloor \log n! \rfloor \approx \lfloor n \log n - 1.5n \rfloor$. $\Box$