Introduction to

Algorithm Design and Analysis

[06] MergeSort

Jingwei Xu
https://ics.nju.edu.cn/~xjw/
Institute of Computer Software
Nanjing University

In the last class …

- **Heap**
    - Partial order property
        - FixHeap
        - ConstructHeap
    - Heap structure
        - Array-based implementation
- **HeapSort**
    - Complexity
    - Accelerated HeapSort

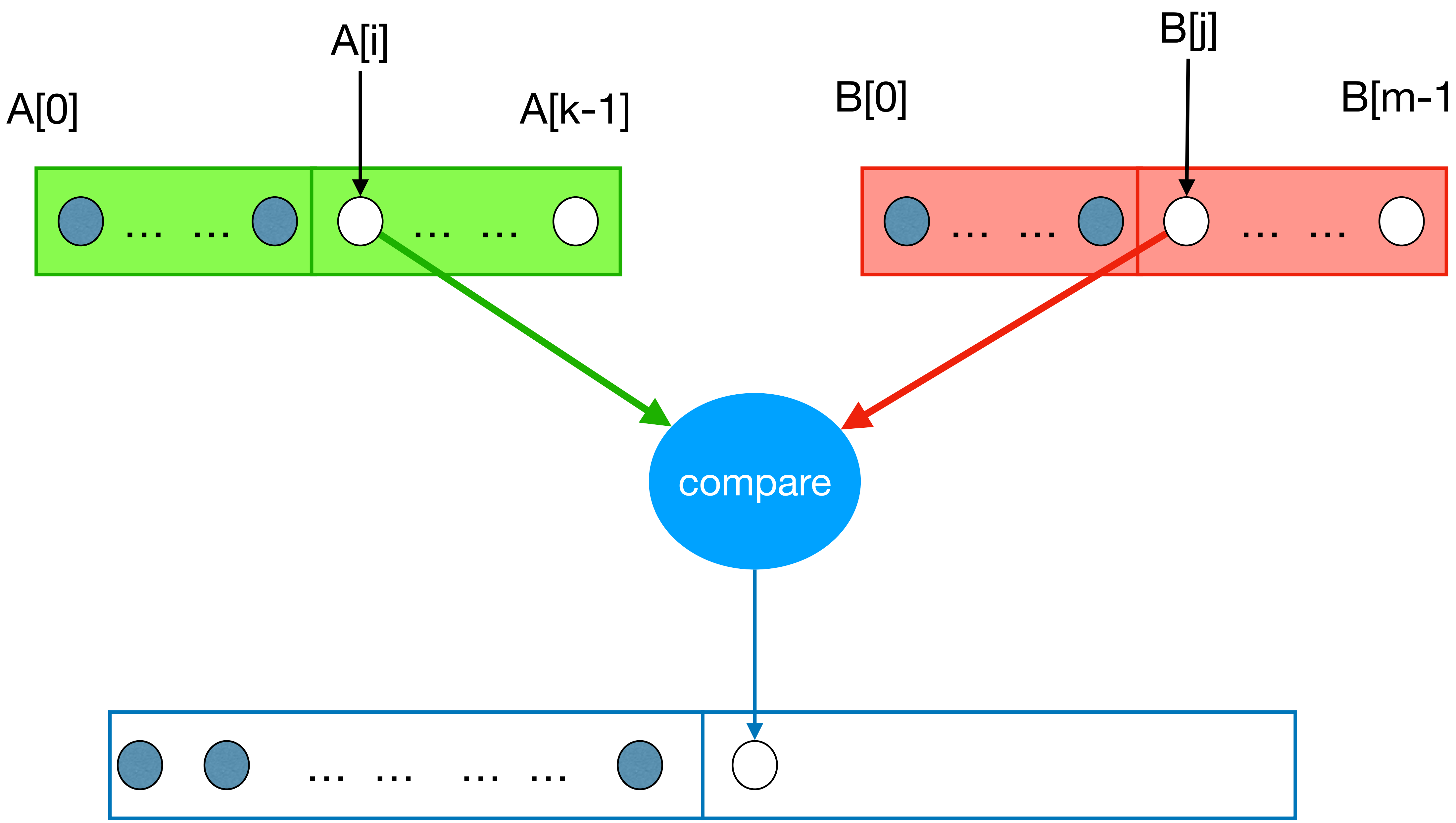
MergeSort

- **MergeSort**
  - Worst-case analysis of MergeSort
- **Lower Bounds for comparison-based sorting**
  - Worst-case
  - Average-case

MergeSort: the Strategy

- **Easy division**
  - No comparison is conducted during the division
  - Minimizing the size difference between the divided subproblems
- **Merging two sorted subranges**
  - Using Merge

Merging Sorted Arrays



Merge: the Specification

- **Input**
  - Array A with k elements and B with m elements, whose keys are in non-decreasing order
- **Output**
  - Array C containing n=k+m elements from A and B in non-decreasing order
  - C is passed in and the algorithm fills it

Merge: Recursive Version

```
merge(A,B,C)
    if (A is empty)
        rest of C = rest of B
    else if (B is empty)
        rest of C = rest of A
    else
        if (first of A ≤ first of B)
            first of C = first of A
            merge(rest of A, B, rest of C)
        else
            first of C = first of B
            merge(A, rest of B, rest of C)
    return
```

Base cases

Worst Case Complexity of Merge

- **Observations**
  - Worst case is that the last comparison is conducted between A[k-1] and B[m-1]
    - After each comparison, at least one element is inserted into Array C, at least.
    - After entering Array C, an element will never be compared again.
    - After the last comparison, at least two elements have not yet been moved to Array C. So at most n-1 comparisons are done.
- **In worst case, n-1 comparisons are done, where n=k+m**

Optimality of Merge

- **Any algorithm to merge two sorted arrays, each containing k=m=n/2 entries, by comparison of keys, does at least n-1 comparisons in the worst case.**
  - Choose keys so that:

$$b_0 < a_0 < b_1 < a_1 < \ldots < b_i < a_i < b_{i+1}, \ldots, < b_{m-1} < a_{k-1}$$

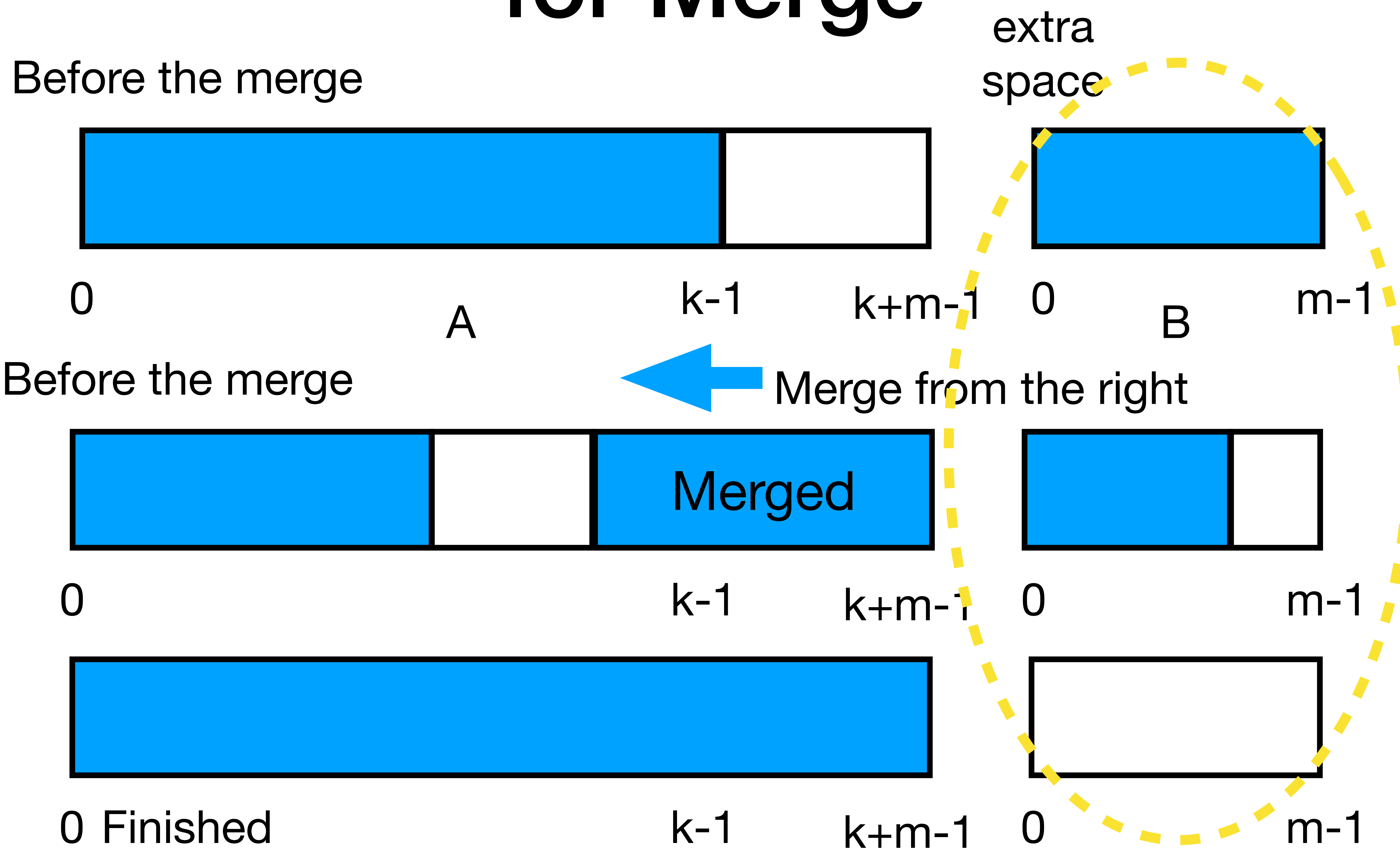
  - Then the algorithm must compare a_i with b_i for every i in [0, m-1], and must compare a_i with b_{i+1} for every i in [0, m-2], so, there are n-1 comparisons.

Valid for |k-m| ≤ 1, as well.

Space Complexity of Merge

- **An algorithm is "in space"**
  - If the extra space it has to use is in $\Theta(1)$
- **Merge is not a algorithm "in space"**
  - Since it needs $O(n)$ extra space to store the merged sequence during the merging process.

Overlapping Arrays for Merge



MergeSort

- **Input: Array E and indexes first, and last, such that the elements of E[i] are defined for first≤i≤last.**
- **Output: E[first],…,E[last] is a sorted rearrangement of the same elements.**
- Procedure

```
void mergeSort(Element[] E, int first, int last)
    if (first < last)
        int mid = (first+last) / 2;
        mergeSort(E, first, mid);
        mergeSort(E, mid + 1, last);
        merge(E, first, mid, last);
    return;
```

Analysis of MergeSort

- The recurrence equation for MergeSort

$$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1$$
$$W(1) = 0$$

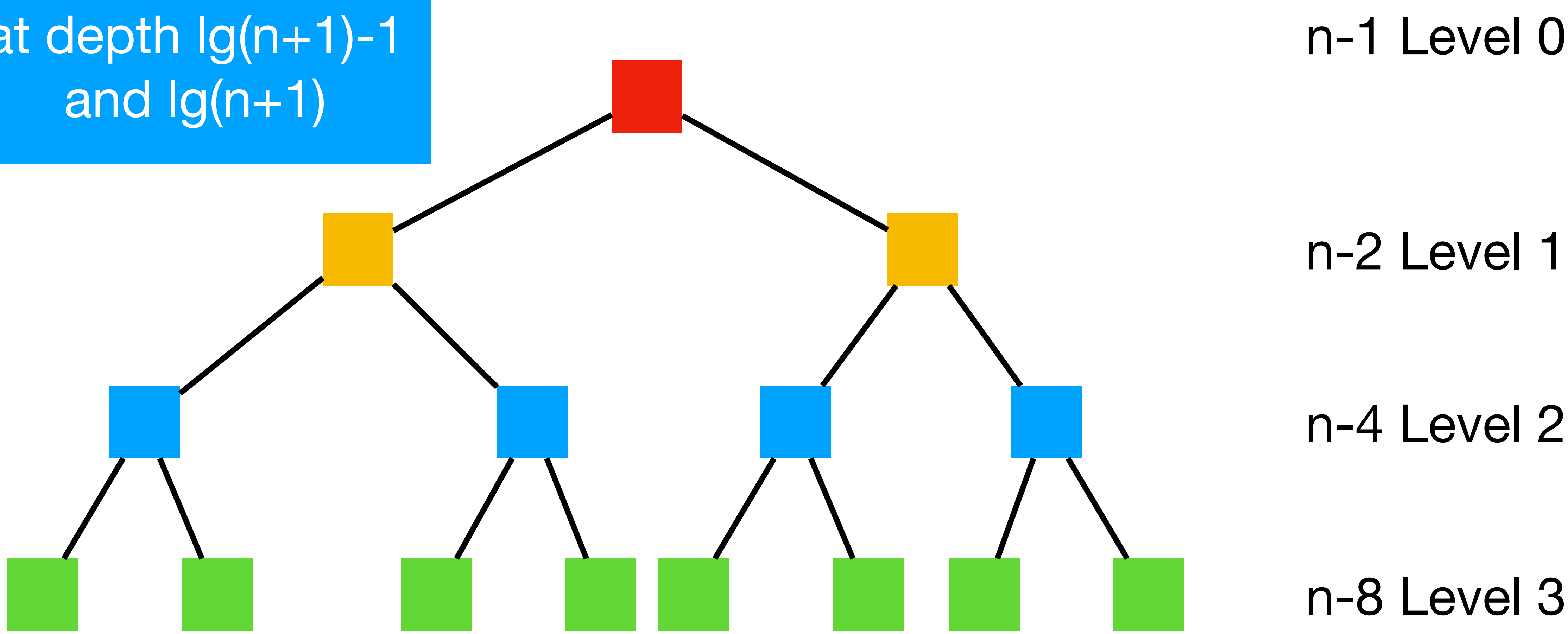
Where n = last - first + 1, the size of range to be sorted

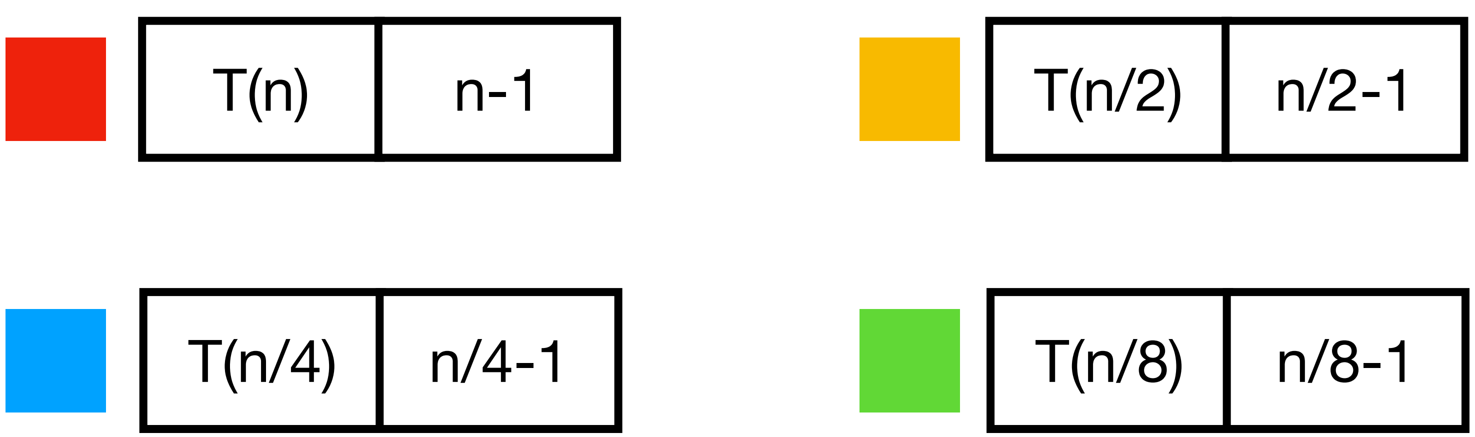
- The Master Theorem applies for the equation, so:

$$W(n) \in \Theta(n \log n)$$

Recursion Tree for MergeSort

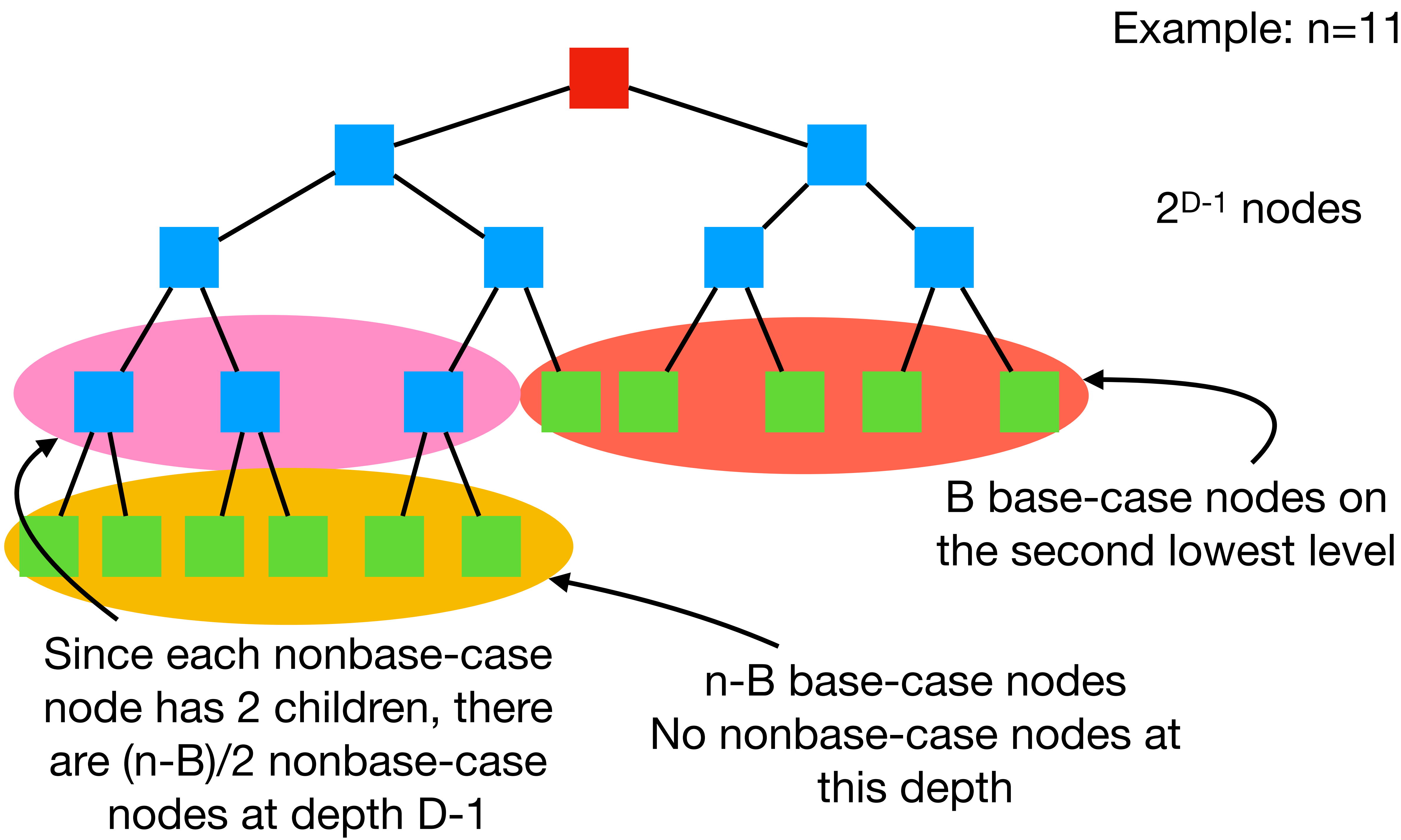
Base cases occur at depth lg(n+1)-1 and lg(n+1)





Note:
non recursive costs on level k is n-2^k for all level without base case node.

Non-complete Recursion Tree



Number of Comparison of MergeSort

- The maximum depth D of the recursive tree is ⌈lg(n+1)⌉.
- Let B base case nodes on depth D-1, and n-B on depth D, (Note: base case node has non-recursive cost 0).
- (n-B)/2 non-base case nodes at depth D-1, each has non-recursive cost 1.
- So:

$$W(n) = \sum_{d=0}^{D-2}(n - 2^d) + \frac{n-B}{2} = n(D-1) - (2^{D-1} - 1) + \frac{n-B}{2}$$

$$Since\ (2^D - 2B) + B = n,\ that\ is\ B = 2^D - n$$

$$So,\ W(n) = nD - 2^D + 1$$

$$Let\ \frac{2^D}{n} = 1 + \frac{B}{n} = \alpha,\ then\ 1 \le \alpha < 2, \quad D = \lg n + \lg \alpha$$

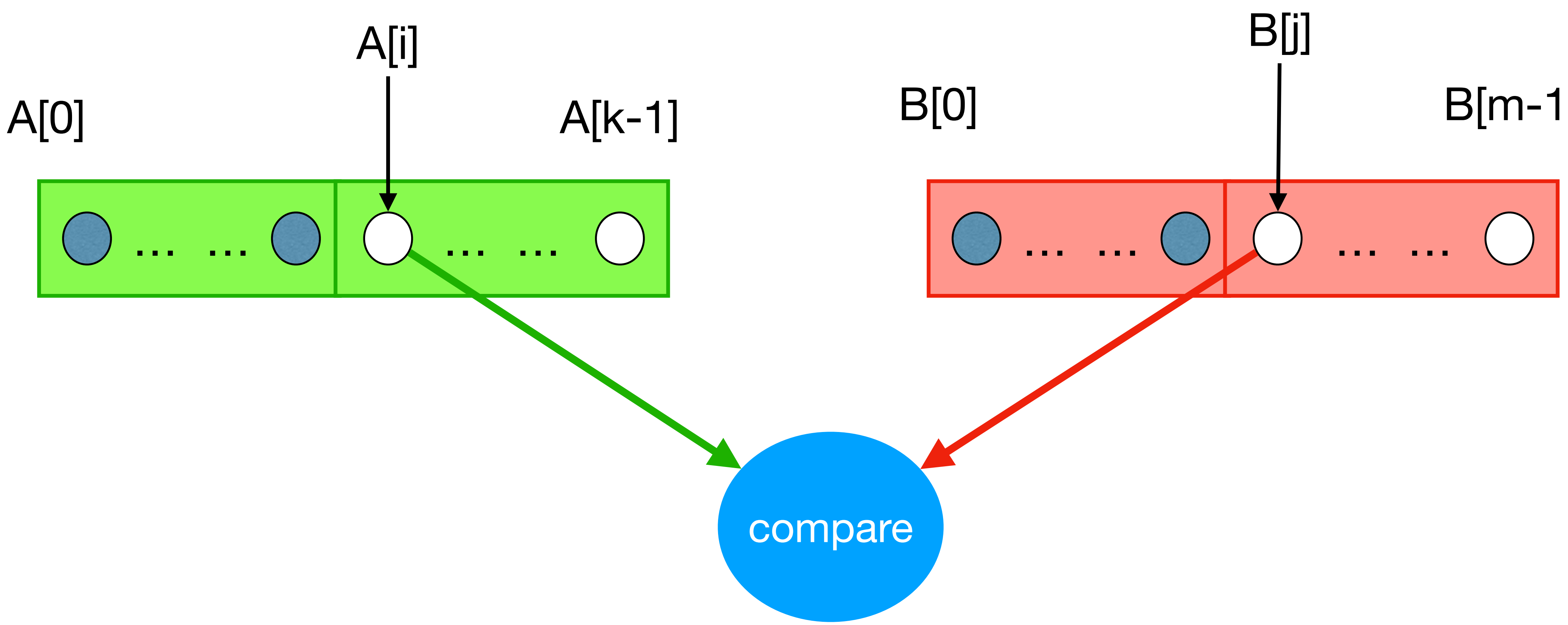
$$So,\ W(n) = n \lg n - (\alpha - \lg \alpha)n + 1$$

- ⌈nlg(n)-n+1⌉ ≤ number of comparison ≤ ⌈nlg(n)-0.914n⌉

The MergeSort D&C

- **Counting the number of inversions**
  - Brute force: $O(n^2)$
  - Can we use divide & conquer
    - In O(nlogn)=>combination in O(n)
- **MergeSort as the carrier**
  - Sorted subarrays
    - A[0..k-1] and B[0..m-1]
  - Compare the left and right elements
    - A[i] v.s. B[j]

The MergeSort D&C

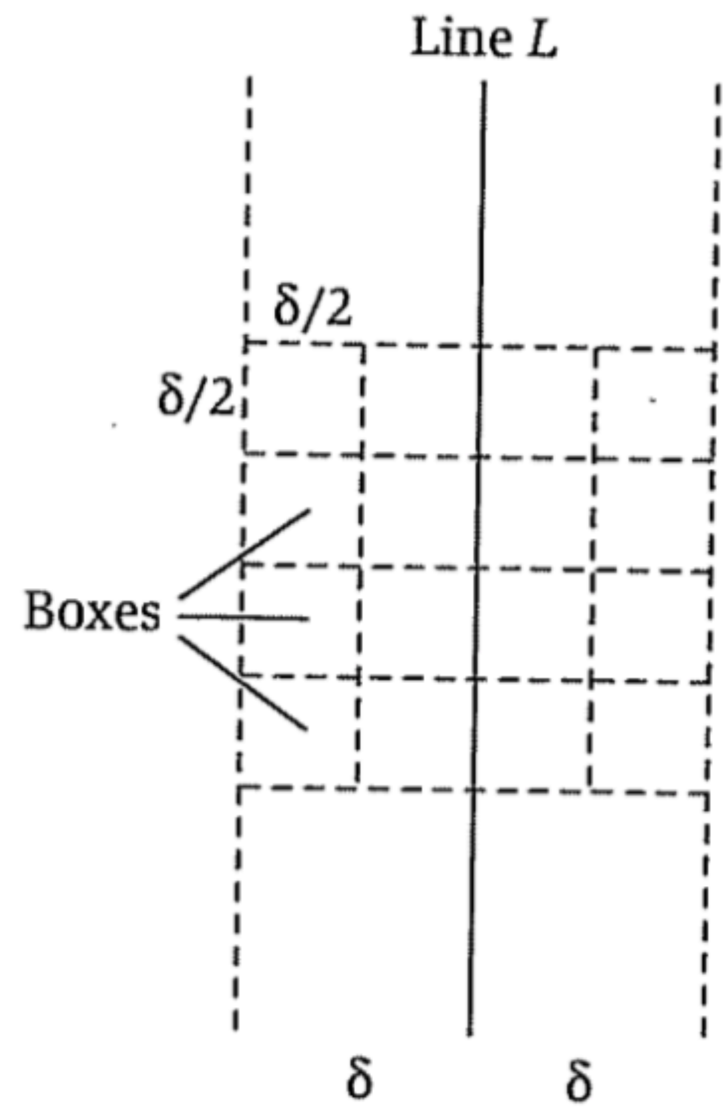


if A[i]>B[j]
(i,j) is an inversion
All (i’,j) are inversions (i’>i)
B[j] is selected

if A[i]<B[j]
No inversion found
A[i] is selected

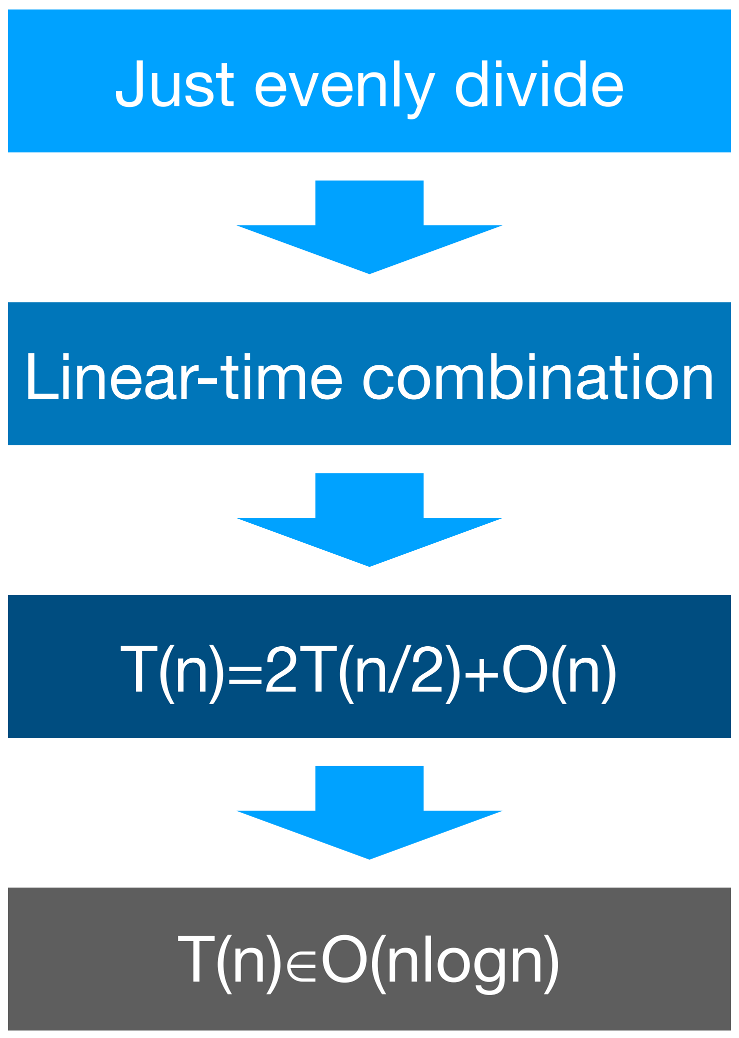
The MergeSort D&C

- **The nearest pair**
  - n nodes on a plane
  - The pair with the minimum distance
- **The MergeSort D&C**
  - T(n)=2T(n/2)+f(n)
  - f(n) must be O(n)
    - How?



The MergeSort D&C

- Max-sum subsequence
- Maxima on a plane
- Finding the frequent element
- Integer/matrix multiplication
- ...

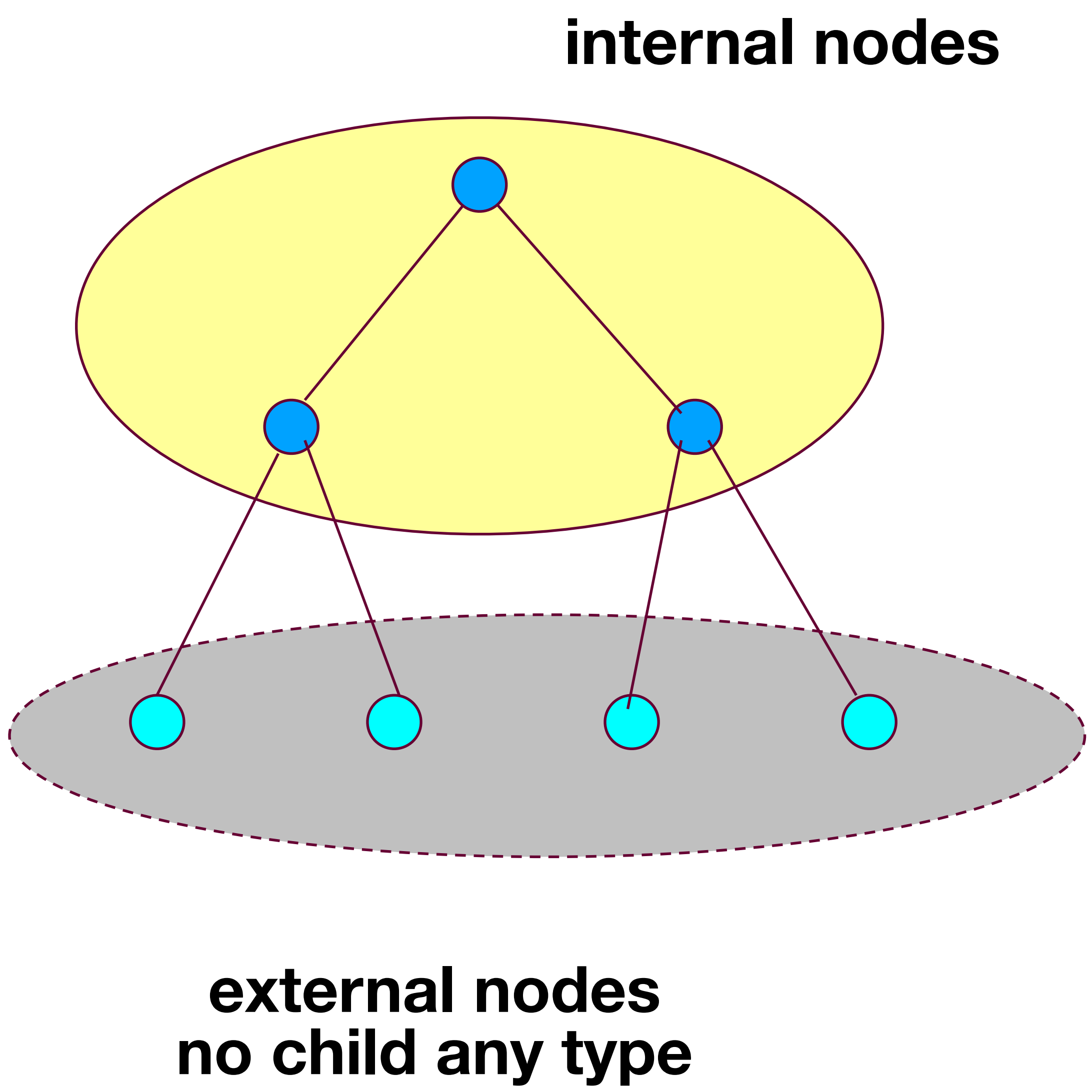


Lower Bounds for Comparison-based Sorting

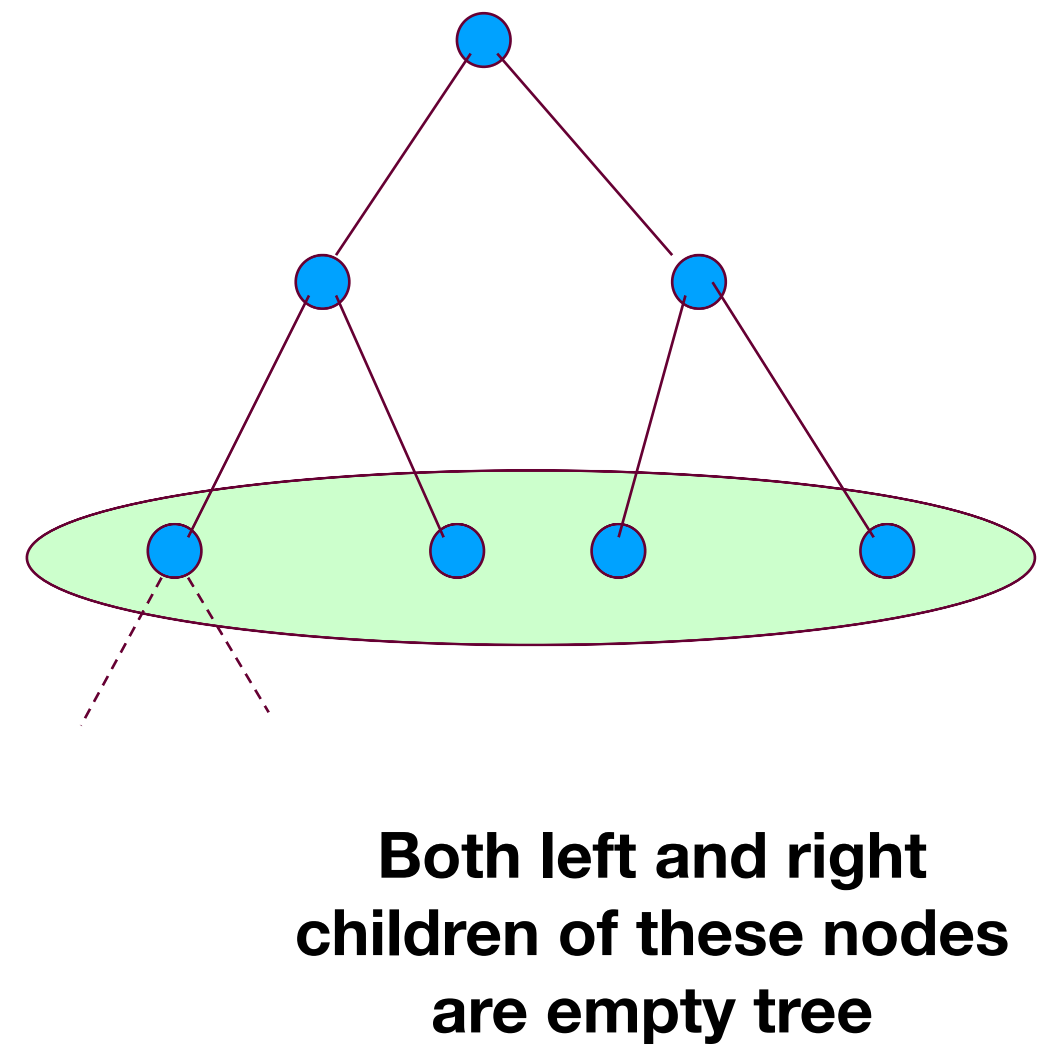
- **Upper bound, e.g., worst-case cost**
  - For any possible input, the cost of the specific algorithm A is no more than the upper bound
    - Max{cost(i) | i is an input}
- **Lower bound, e.g., comparison-based sorting**
  - For any possible (comparison-based) sorting algorithm A, the worst-case cost is no less than the lower bound
    - Min{worst-case(a) | a is an algorithm}

2-Tree

- 2-Tree

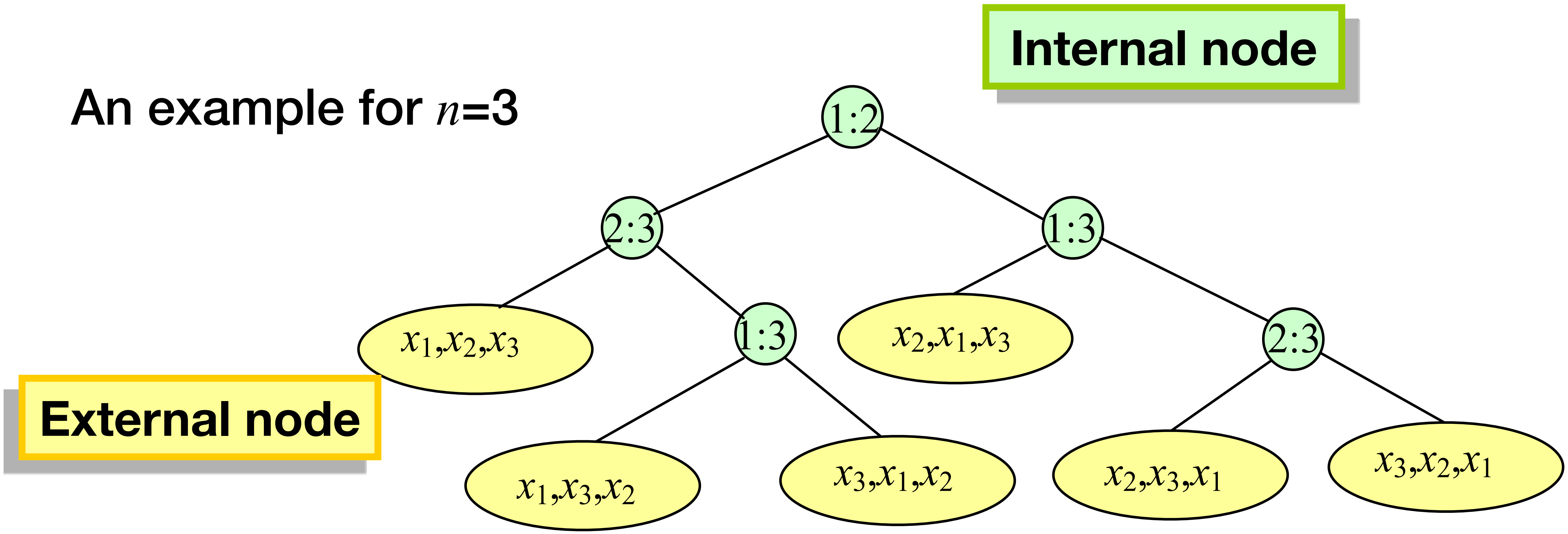


- Common Binary Tree



Decision Tree for Sorting

An example for n=3



- Decision tree is a 2-tree (Assuming no same keys)
- The action of Sort on a particular input corresponds to following on path in its decision tree from the root to a leaf associated to the specific output

Characterizing the Decision Tree

- For a sequence of n distinct elements, there are n! different permutation
  - So, the decision tree has at least n! leaves, and exactly n! leaves can be reached from the root.
  - So, for the purpose of lower bounds evaluation, we use trees with exactly n! leaves.
- The number of comparison done in the worst case is the height of the tree.
- The average number of comparison done is the average of the lengths of all paths from the root to a leaf.

Lower Bound for Worst Case

- ***Theorem***: Any algorithm to sort *n* items by comparisons of keys must do at least $\lceil \lg n! \rceil$, or approximately $\lceil n\lg n - 1.443n \rceil$, key comparisons in the worst case.
  - Note: Let $L=n!$, which is the number of leaves, then $L \le 2^h$, where h is the height of the tree, that is $h \ge \lceil \lg L \rceil = \lceil \lg n! \rceil$
  - For the asymptotic behavior:

$$\lg(n!) \ge \lg[n(n-1)\ldots\left(\left\lceil \frac{n}{2} \right\rceil\right)] \ge \lg\left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2}\lg\left(\frac{n}{2}\right) \in \Theta(n \lg n)$$

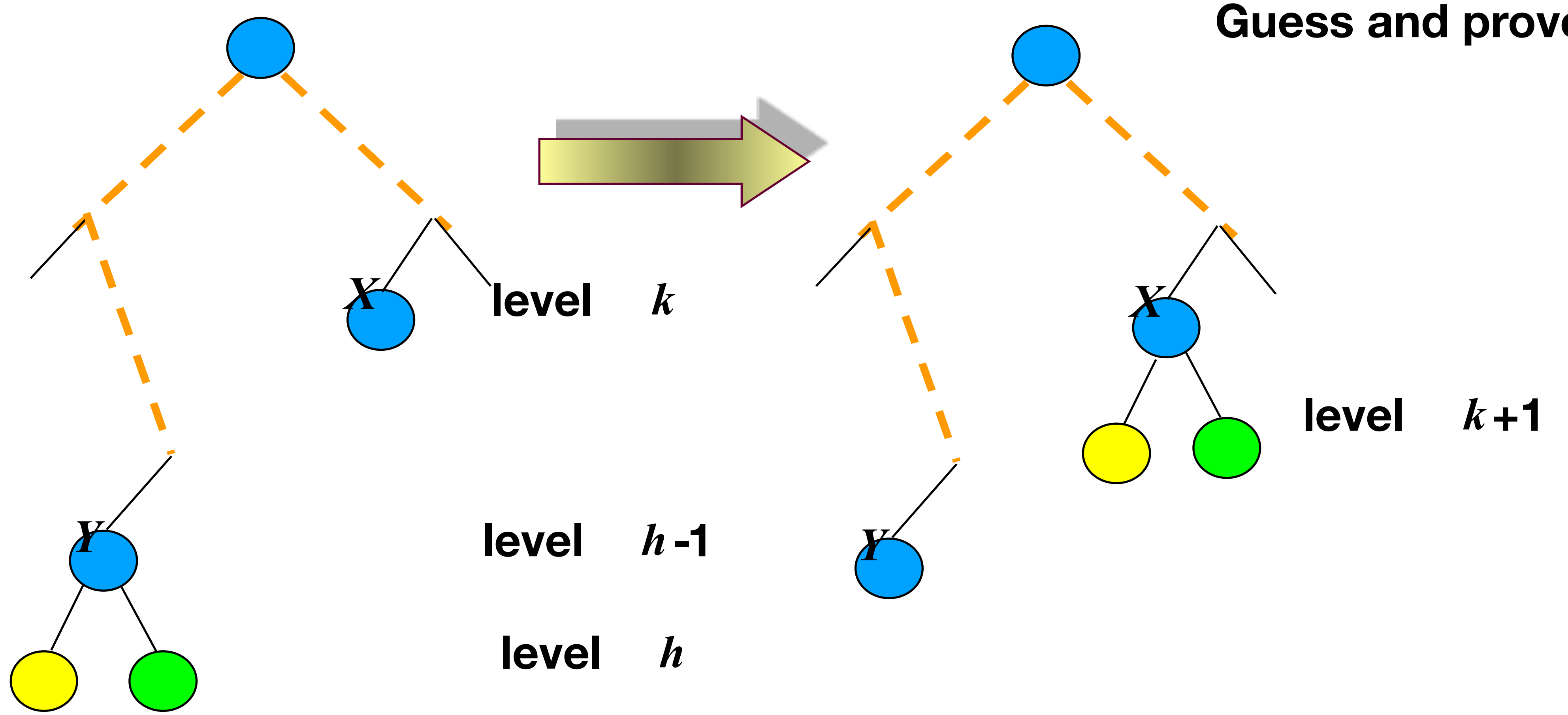
derived using: $$\lg n! = \sum_{j=1}^{n} \lg(j)$$

External Path Length (EPL)

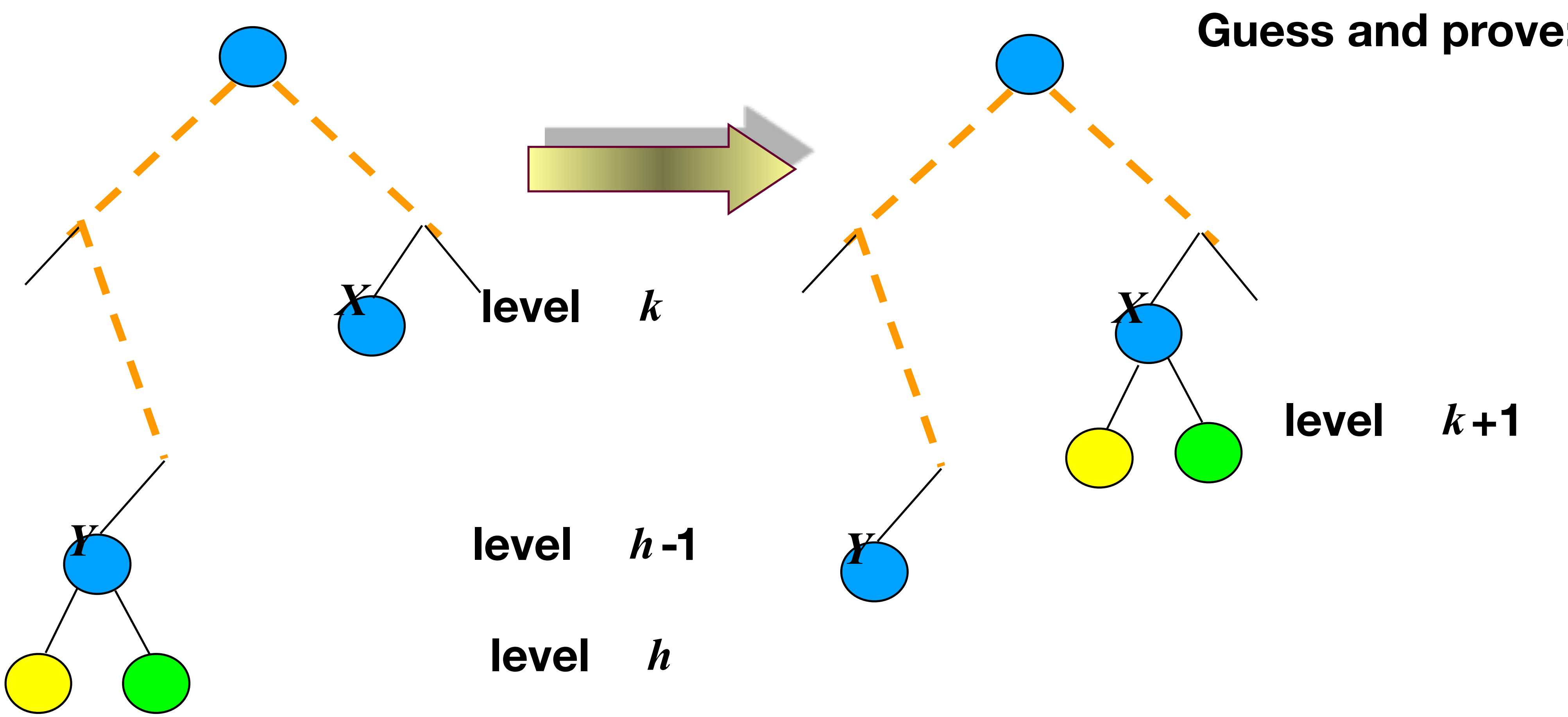
- The EPL of a 2-tree t is defined as follows:
  - [Base case] 0 for a single external node
  - [Recursion] *t* is non-leaf with sub-trees *L* and *R*, then the sum of:
    - the external path length of *L*;
    - the number of external node of *L*;
    - the external path length of *R*;
    - the number of external node of *R*;

More Balanced 2-tree, Less EPL

Guess and prove:



More Balanced 2-tree, Less EPL



Assuming that h-k>1, when calculating epl, h+h+k is replaced by (h-1)+2(k+1). The net change in epl is k-h+1<0, that is, the epl decreases.

So, more balanced 2-tree has smaller epl.

Properties of EPL

- **Let t is a 2-tree, then the epl of t is the sum of the paths from the root to each external node.**
- $epl \geq m\lg(m)$, where m is the number of external nodes in t
  - $epl = epl_L + epl_R + m \geq m_L\lg(m_L) + m_R\lg(m_R) + m$,
  - note $f(x)+f(y) \geq 2f((x+y)/2)$ for $f(x)=x\lg x$
  - so,

    $epl \geq 2((m_L+m_R)/2)\lg((m_L+m_R)/2)+m = m(\lg(m)-1)+m$

    $=m\lg m$.

Lower Bound for Average Behavior

- **Since a decision tree with L leaves is a 2-tree, the average path length from the root to a leaf is $\frac{epl}{L}$**
  - Recall that *epl* ≥ *L*lg(*L*).
- **Theorem**: The average number of comparison done by an algorithm to sort *n* items by comparison of keys is at least lg(*n*!), which is about *n*lg*n*-1.443*n*.

MergeSort Has Optimal Average Performance

- The average number of comparisons done by an algorithm to sort n items by comparison of keys is at least about $n\lg n - 1.443n$
- The **worst** complexity of MergeSort is in $\Theta(n\lg n)$
- But, the average performance can not be worse than the worst case performance.
- So, MergeSort is optimal as for its average performance.

Thank you!
Q & A