Our First Data Structure

- The story so far
  - We understand the need for data structures
  - We have seen a few analysis techniques
- This week we will
  - Attempt a definition of what is a data structure
  - see a very simple data structure, and
  - advanced applications of this data structure.

A Data Structure

- How should we view a data structure?
- From a implementation point of view
  - Should implement a set of operations
  - Should provide a way to store the data in some form.
- From a user point of view
  - should use it as a black box
  - call the operations provided
- Analogy : C programming language has a few built-in data types.
  - int, char, float, etc.

Analogy

- Consider a built-in data type such as int
- A programmer
  - can store an integer within certain limits
  - can access the value of the stored integer
  - can do other operations such add, subtract, multiply, ...
- Who is implementing the data structure?
  - A compiler writer.
  - Interacts with the system and manages to store the integer.

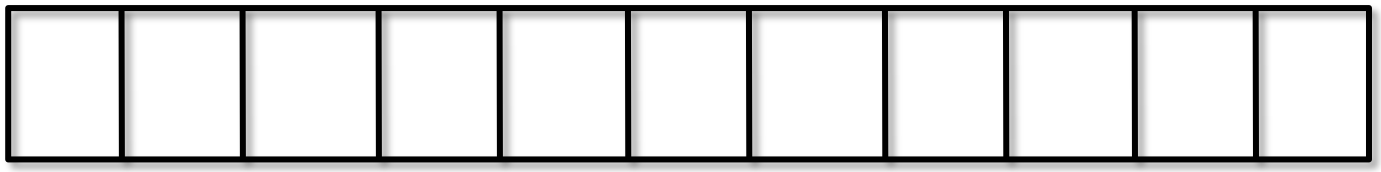
An Abstract Data Type

- A data structure can thus be looked as an abstract data type.
- An abstract data type specifies a set of operations for a user.
- The implementor of the abstract data type supports the operations in a suitable manner.
- One can thus see the built-in types also as abstract data types.

The Array as a Data Structure

- Suppose you wish to store a collection of like items.
  - say a collection of integers
- Will access any item of the collection.
- May or may not know the number of items in the collection.
- Example settings:
  - store the scores on an exam for a set of students.
  - store the temperature of a city over several days.

The Array as a Data Structure



- In such settings, one can use an array.
- An array is a collection of like objects.
  - Usually denoted by upper case letters such as A, B.
- Let us now define this more formally.

The Array ADT

- Typical operations on an array are:
  - create(name, size) : to create an array data structure,
  - ElementAt(index, name) : to access the element at a given index i
  - size(name) : to find the size of the array, and
  - print(name) : to print the contents of the array
- Note that in most languages, elementAt(index, name) is given as the operation name[index].

The Array Implementation

```
Algorithm Create(int size, string name)
begin
name = malloc(size*sizeof(int));
end
```

```
Algorithm Print(string
name)
begin
for i = 1 to n do
    printf("%d t",
        name[i]);
end-for;
end;
```

```
Algorithm ElementAt(int index,
    string name)
begin
return name[i];
end
```

```
Algorithm size(string name)
begin
return size;
end;
```

Further Operations

- The above operations are quite fundamental.
- Need further operations for most problems.
- We will now see some such operations.

Sorting

- Sorting is a fundamental concept in Computer Science.
  - several application and a lot of literature.
  - We shall see one algorithm for sorting.

QuickSort

- The quick sort algorithm designed by Hoare is a simple yet highly efficient algorithm.
- It works as follows:
  - Start with the given array A of n elements.
  - Consider a pivot, say A[1].
  - Now, partition the elements of A into two arrays A_L and A_R such that:
    - the elements in A_L are less than A[1]
    - the elements in A_R are greater than A[1].
  - Sort A_L and A_R, recursively.

How to Partition?

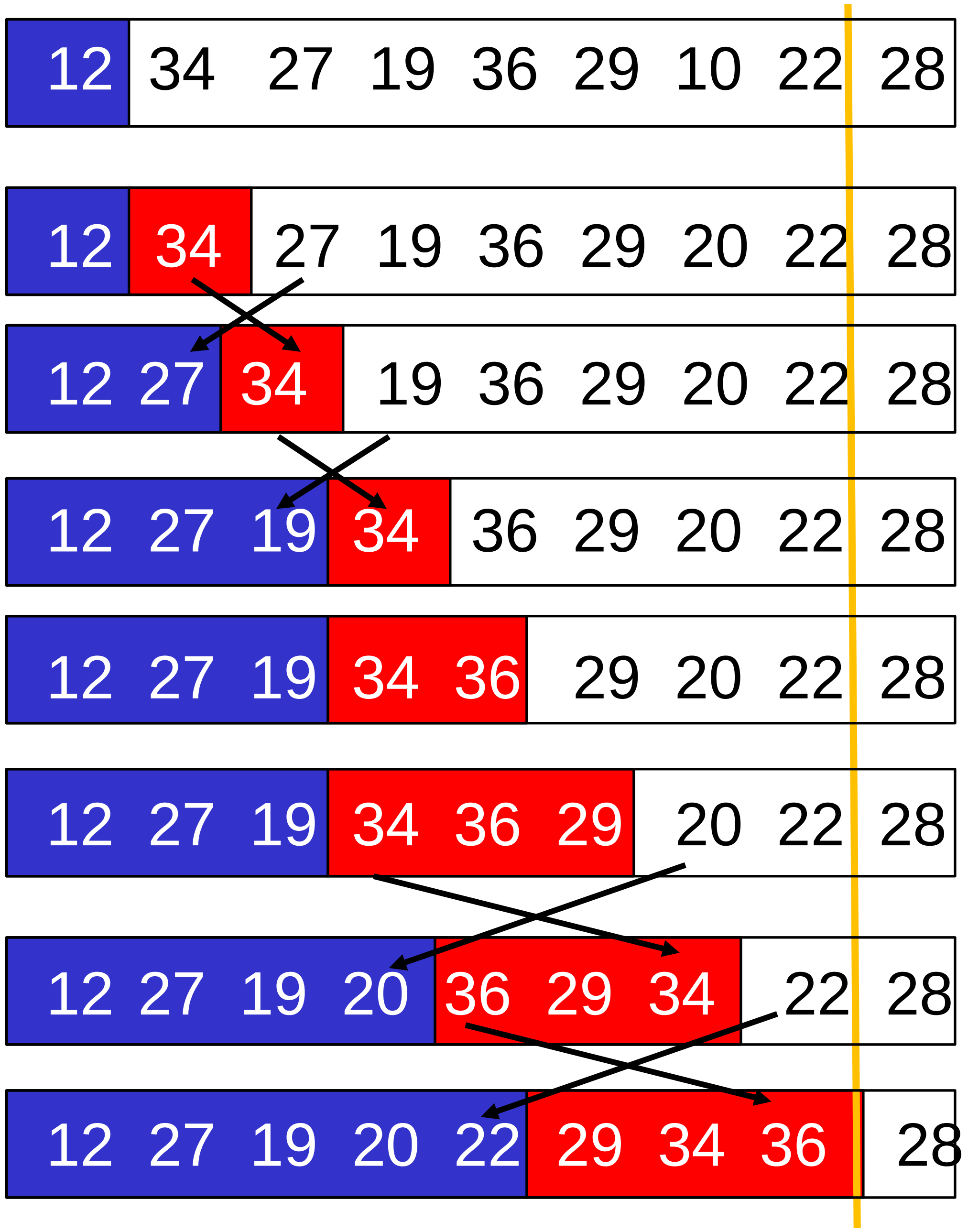
- Here is an idea.
  - Suppose we take each element, compare it with A[1] and then move it to A_L or A_R accordingly.
  - Works in O(n) time.
  - Can write the program easily.
  - But, recall that space is also an resource. The above approach requires extra space for the arrays A_L and A_R
  - A better approach exists.

Algorithm Partition

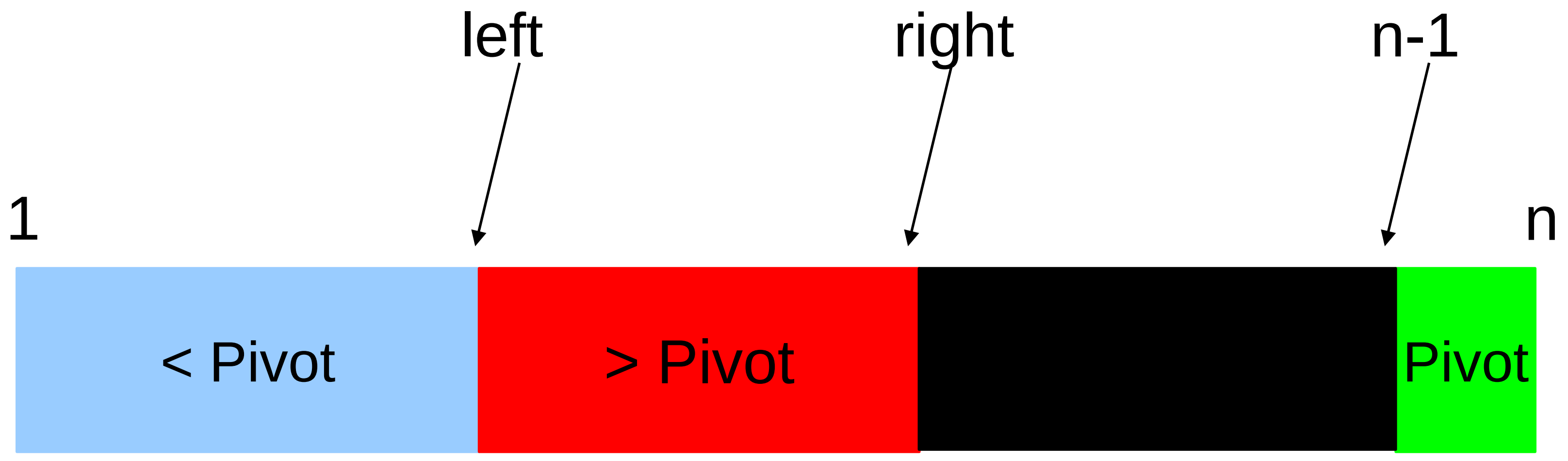
```
Procedure Partition(A,n)
begin
   pivot = A(n);
   less = 0; more = 1;
   for more = 1 to n do
      if A(more) < pivot then
          less++;
          swap(A(more), A(less));
    end
  swap A[less+1] with A[n];
end
```

- Algorithm Partition is given above.

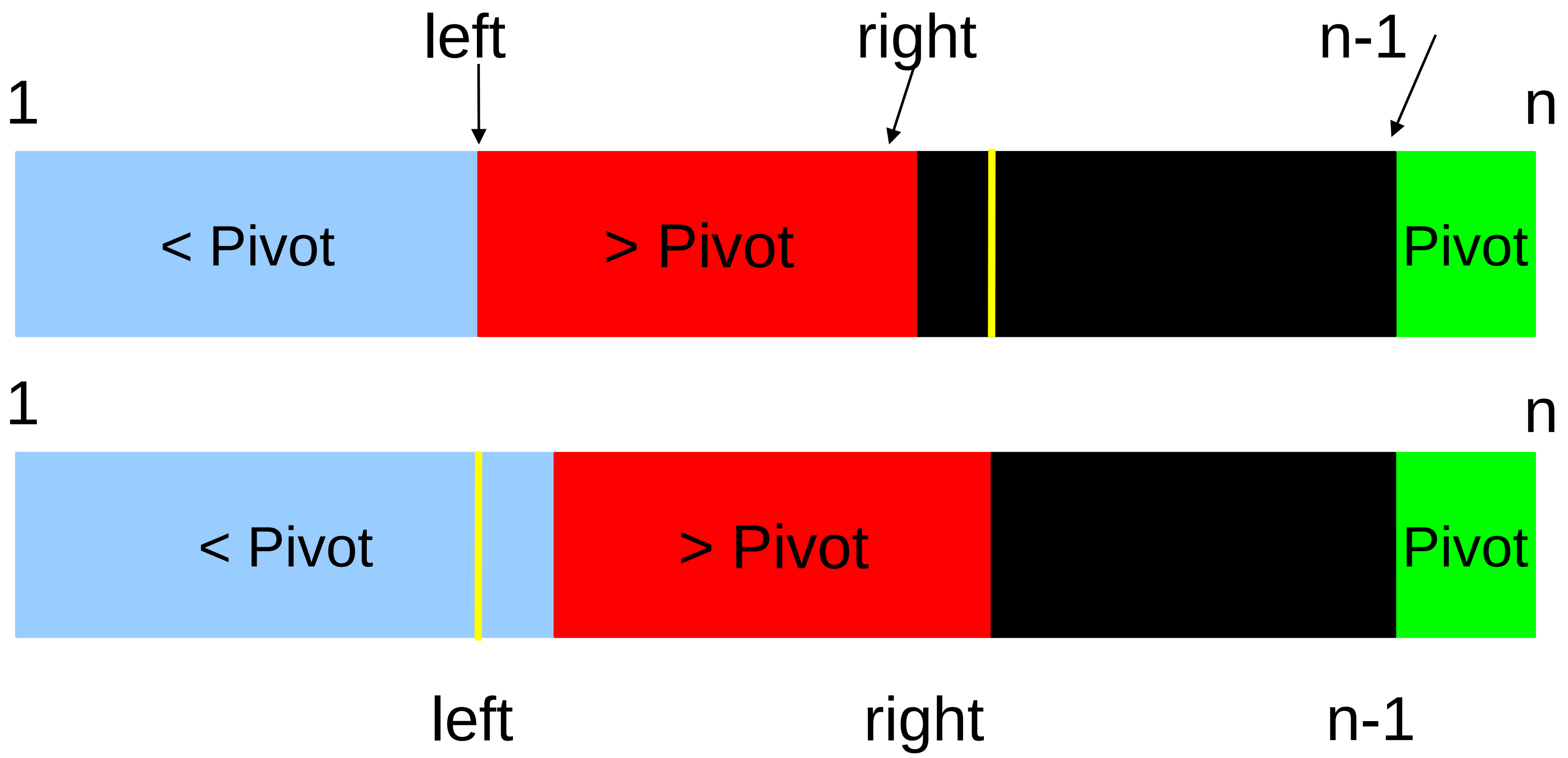
Example


12 34 27 19 36 29 10 22 28
12 34 27 19 36 29 20 22 28
12 27 34 19 36 29 20 22 28
12 27 19 34 36 29 20 22 28
12 27 19 34 36 29 20 22 28
12 27 19 34 36 29 20 22 28
12 27 19 20 36 29 34 22 28
12 27 19 20 22 29 34 36 28

The Basic Step in Partition

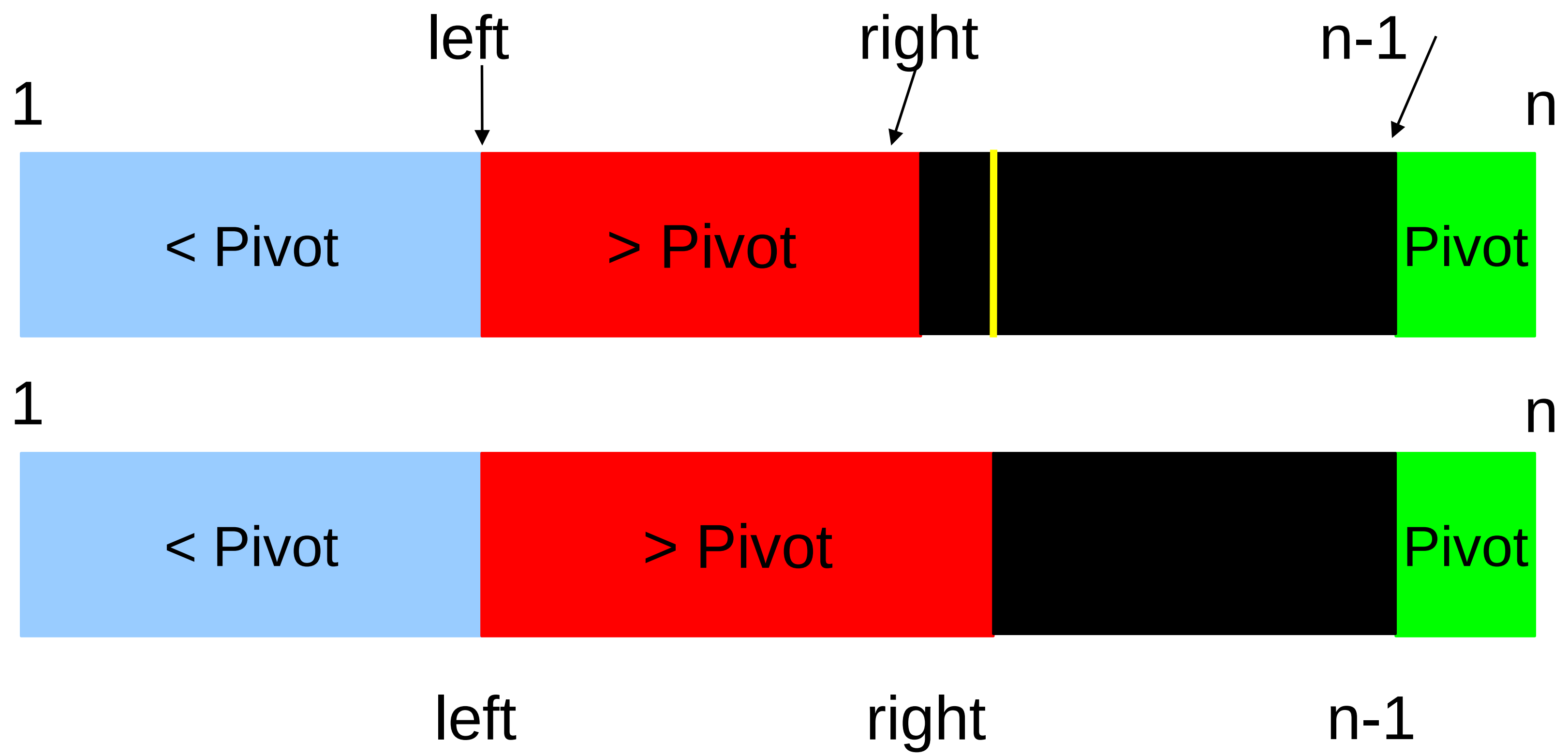


The Basic Step in Partition



- The main action in the loop is the comparison of A[k+1] with A[n].
- Consider the case when A[k+1] < A[n].

The Basic Step in Partition



- Consider the case when A[k+1]> A[n]

Time Analysis

- The time taken by partition is O(n) when A has n elements.
- What is the time taken by the quick sort procedure?
- The recurrence relation is

  $T(n) = T(|A_L|) + T(|A_R|) + O(n)$
- How to measure the size of A_L and A_R?

Time Analysis

- What is each of these subarrays has size exactly n/2?

Time Analysis

- What is each of these subarrays has size exactly n/2?
- Then, T(n) = O(n log n).

Time Analysis

- What is each of these subarrays has size exactly n/2?
- Then, T(n) = O(n log n).
- But sizes vary depending on the input.

Time Analysis

- What is each of these subarrays has size exactly n/2?
- Then, $T(n) = O(n \log n)$.
- But sizes vary depending on the input.
- Worst case calculation
  - Consider what happens when A_L has a size of n-1, always.
  - The recurrence relation becomes $T(n) = T(n-1) + O(n)$.
  - The solution is $T(n) = O(n^2)$.
- So, the runtime is between $O(n \log n)$ and $O(n^2)$.

Time Analysis

- What is each of these subarrays has size exactly n/2?
- Then, $T(n) = O(n \log n)$.
- But sizes vary depending on the input.
- Worst case calculation
  - Consider what happens when A_L has a size of n-1, always.
  - The recurrence relation becomes $T(n) = T(n-1) + O(n)$.
  - The solution is $T(n) = O(n^2)$.
- So, the runtime is between $O(n \log n)$ and $O(n^2)$.

Time Analysis

- What is each of these subarrays has size exactly n/2?
- Then, T(n) = O(n log n).
- But sizes vary depending on the input.
- Worst case calculation
  - Consider what happens when A_L has a size of n-1, always.

Time Analysis

- What is each of these subarrays has size exactly n/2?
- Then, $T(n) = O(n \log n)$.
- But sizes vary depending on the input.
- Worst case calculation
  - Consider what happens when A_L has a size of n-1, always.
  - The recurrence relation becomes $T(n) = T(n-1) + O(n)$.

Time Analysis

- What is each of these subarrays has size exactly n/2?
- Then, $T(n) = O(n \log n)$.
- But sizes vary depending on the input.
- Worst case calculation
  - Consider what happens when A_L has a size of n-1, always.
  - The recurrence relation becomes $T(n) = T(n-1) + O(n)$.
  - The solution is $T(n) = O(n^2)$.

Time Analysis

- What is each of these subarrays has size exactly n/2?
- Then, $T(n) = O(n \log n)$.
- But sizes vary depending on the input.
- Worst case calculation
  - Consider what happens when A_L has a size of n-1, always.
  - The recurrence relation becomes $T(n) = T(n-1) + O(n)$.
  - The solution is $T(n) = O(n^2)$.
- So, the runtime is between $O(n \log n)$ and $O(n^2)$.

Another Operation – Prefix Sums

- Consider any associative binary operator, such as +, and an array A of elements over which o is applicable.
- The prefix operation requires us to compute the array S so that S[i] = A[1]+A[2]+ · · · +A[i].
- The prefix operation is very easy to perform in the standard sequential setting.

Sequential Algorithm for Prefix Sum

```
Algorithm PrefixSum(A)
S[1] = A[1];
for i = 2 to n do
    S[i] = A[i] + S[i-1]
end-for
```

- Example A = (3, -1, 0, 2, 4, 5)
- S[1] = 3.
- S[2] = -1+3 = 2, S[3] = 0 + 2 = 2,...
- The time taken for this program is $O(n)$.

Our Interest in Prefix

- The world is moving towards parallel computing.
- This is necessitated by the fact that the present sequential computers cannot meet the computing needs of the current applications.
  - It would take 2,000 1-Gflops CPUs approximately 150 days to complete the computation corresponding to a 90 minute movie using digital special effects.
- Already, parallel computers are available with the name of multi-core architectures.
  - Majority of PCs today are at least dual core.

Our Interest in Prefix

- Programming and software has to wake up to this reality and have a rethink on the programming solutions.
- The parallel algorithms community has fortunately given a lot of parallel algorithm design techniques and also studied the limits of parallelizability.
- How to understand parallelism in computation?

Parallelism in Computation

- Think of the sequential computer as a machine that executes jobs or instructions.
- With more than one processor, can execute more than one job (instruction) at the same time.
  - Cannot however execute instructions that are dependent on each other.

Parallelism in Computation

- This opens up a new world where computations have to specified in parallel.
- Sometimes have to rethink on known sequential approaches.
- Prefix computation is one such example.
  - Turns out that prefix sum is a fundamental computation in the parallel setting.
  - Applications span several areas.

Parallelism in Computation

- The obvious sequential algorithm does not have enough independent operations to benefit from parallel execution.
- Computing S[i] requires computation of S[i-1] to be completed.
- Have to completely rethink for a new approach.

Parallel Prefix

- Consider the array A and produce the array B of size $n/2$ where $B[i] = A[2i - 1]+A[2i]$.
- Imagine that we recursively compute the prefix output wrt B and call the output array as C.
- Thus, $C[i] = B[1]+B[2]+ \cdots +B[i]$. Let us now build the array S using the array C.

Parallel Prefix

- For this, notice that for even indices i, C[i] = B[1]+ B[2] + · · ·+B[i] = A[1] + A[2] + · · ·+A[2i], which is what S[2i] is to be.
- Therefore, for even indices of S, we can simply use the values in array C.

Parallel Prefix

- For this, notice that for even indices i, C[i] = B[1]+ B[2] + · · ·+B[i] = A[1] + A[2] + · · ·+A[2i], which is what S[2i] is to be.
- Therefore, for even indices of S, we can simply use the values in array C.
- The above also suggests that for odd indices of S, we can apply the + operation to a value in C and a value in A.

Parallel Prefix Example

- A = (3, 0, -1, 2, 8, 4, 1, 7)
- B = (3, 1, 12, 8)
  - B[1] = A[1] + A[2] = 3 + 0 = 3
  - B[2] = A[3] + A[4] = -1 + 2 = 1
  - B[3] = A[5] + A[6] = 8 + 4 = 12
  - B[4] = A[7] + A[8] = 1 + 7 = 8
- Let C be the prefix sum array of B, computed recursively as C = (3, 4, 16, 24).
- Now we use C to build S as follows.

Parallel Prefix Example

- S[1] = A[1], always.
- C[1] = B[1] = A[1] + A[2] = S[2]
- C[2] = B[1] + B[2] = A[1] + A[2] + A[3] + A[4] = S[4]
- C[3] = B[1] + B[2] + B[3]

  = A[1] + A[2] + A[3] + A[4] +A[5] + A[6] = S[6]
- That completes the case for even indices of S.
- Now, let us see the odd indices of S.

Parallel Prefix Example

- Consider, S[3] = A[1] + A[2] + A[3]

  = (A[1] + A[2]) + A[3]

  = S[2] + A[3].
- Similarly, S[5] = S[4] + A[5] and S[7] = S[6] + A[7].
- Notice that if S[2], S[4], and S[6] are known, the computation at odd indices is independent for every odd index.

Parallel Prefix Algorithm

```
Algorithm Prefix(A)
begin
    Phase I: Set up a recursive problem
    for i = 1 to n/2 do
        B[i] = A[2i − 1] + A[2i];
    end-for
    Phase II: Solve the recursive problem
    Solve Prefix(B) into C;
    Phase III: Solve the original problem
    for i = 1 to n do
        if i = 1 then S[1] = A[1];
        else if i is even then S[i] = C[i/2];
        else if i is odd then S[i] = C[(i − 1)/2] + A[i];
    end-for
end
```

Non-recursive Program

```
Input: Array $A$ of size $n = 2k$
Output: Prefix sums in $C[0,j]$, $1 < j < n$
begin
   1. for $1 < j < n$ pardo
         $B[0,j] := A[j]$
   2. for $h = 1$ to $\log n$ do
         3. for $1 < j < n/2^h$ pardo
               $B[h,j] := B[h-1,2j-1] + B[h-1,2j]$
   4. for $h = \log n$ to $0$ do
         5. for $1 < j < n/2^h$ pardo
               if $j$ is even then $C[h,j] := C[h+1,j/2]$
               else if $i = 1$ then $C[h,1] := B[h,1]$
               else $C[h,j] := C[h+1,(j-1)/2] + B[h,j]$
end
```

Pseudocode taken from JaJa, Chapter 2.

Independent Operations

- Loop 1: All n operations are independent of each other.
- Loop 2 and 3: Operations for each h are independent of each other.
  - Number of independent operations = O(n)
  - Number of dependent steps = O(log n)
- Similar comments apply to Loops 4 and 5.

Analyzing the Parallel Algorithm

- Can use the asymptotic model developed.
- Identify which operations are independent.
- These all can be done at the same time provided resources exist.
- In our algorithm
  - Phase I : has n/2 independent additions.
  - Phase II : using our knowledge on recurrence relations, this takes time T(n/2).
  - Phase III : Here, we have another n operations, of which n/2 are independent, and another n/2 are independent.

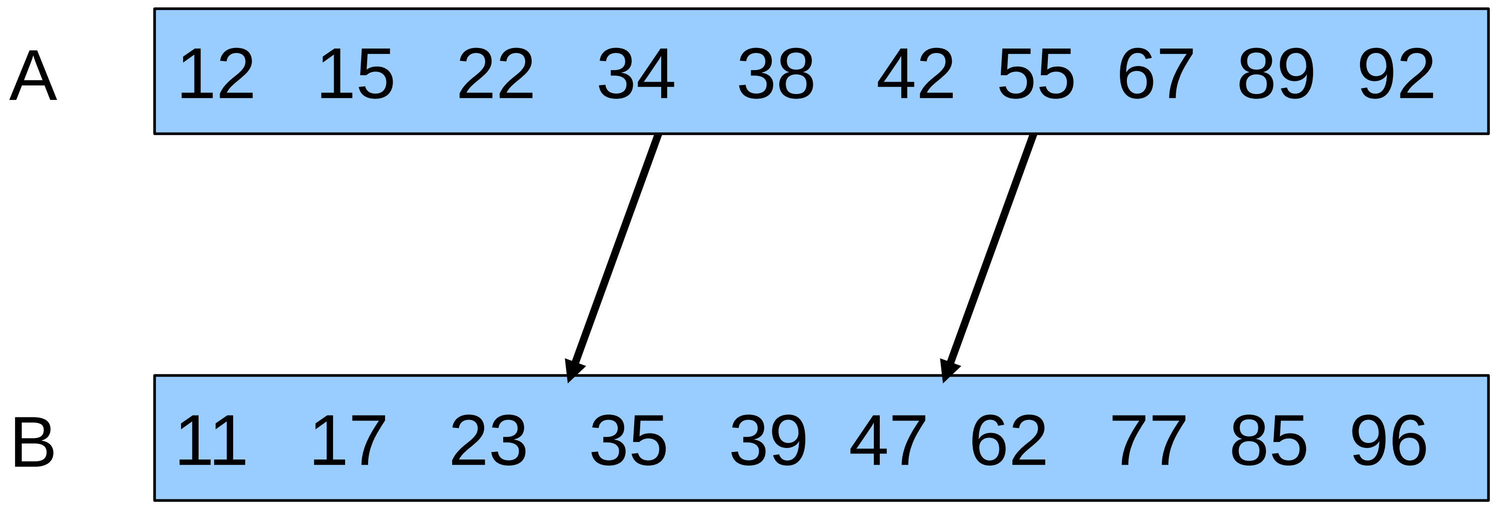
Analyzing the Parallel Algorithm

- How many independent operations can be done at a time?
  - Depends on the number of processors available.
- Assume that as many as n processors are available.
- Hence, phase I can be done in O(1) time totally.
- Phase II can be done in time T(n/2)
- Phase III can be done in O(1) time.

Analyzing the Parallel Algorithm

- Using the above, we have that
  - $T(n) = T(n/2) + O(1)$
  - Can also see that the solution to this recurrence relation is $T(n) = O(\log n)$.
- Compared to the sequential algorithm, the time taken is now only $O(\log n)$, when n processors are available.

Merging in Parallel by Partitioning



- Two sorted arrays A and B to be merged into C.
- Claim: Rank(x, C) = Rank(x, A) + Rank(x, B)

Example

A	12 15 22 34 38 42 55 67 89 92
B	11 17 23 35 39 47 62 77 85 96

Item	12	15	22	34	38	42	55	67	89	92
Rank in A	1	2	3	4	5	6	7	8	9	10
Rank in B	1	1	2	3	4	5	6	7	9	9

Example

A	12 15 22 34 38 42 55 67 89 92
B	11 17 23 35 39 47 62 77 85 96

Item	12	15	22	34	38	42	55	67	89	92
Rank in A	1	2	3	4	5	6	7	8	9	10
Rank in B	1	1	2	3	4	5	6	7	9	9

Item	11	17	23	35	39	47	62	77	85	96
Rank in A	0	2	3	4	5	6	7	8	8	10
Rank in B	1	2	3	4	5	6	7	8	9	10

Example

A	12 15 22 34 38 42 55 67 89 92
B	11 17 23 35 39 47 62 77 85 96

Item	12	15	22	34	38	42	55	67	89	92
Rank in A	1	2	3	4	5	6	7	8	9	10
Rank in B	1	1	2	3	4	5	6	7	9	9
Total	2	3	5	7	9	11	13	15	18	19

Item	11	17	23	35	39	47	62	77	85	96
Rank in A	0	2	3	4	5	6	7	8	8	10
Rank in B	1	2	3	4	5	6	7	8	9	10
Total	1	4	6	8	10	12	14	16	17	20

Finding the Ranks

- For x in A, Rank(x,A) is immediately available. To find Rank(x, B) can use binary search in parallel.
- Similar comments apply to finding ranks of elements in B.

```
Program Merge(A, B)
Begin
     for i = 1 to n  do in parallel
          RA[i]  = BinSrch(B, A[i]) + i
     for i = 1 to n do in parallel
          RB[i] = BinSrch(A, B[i]) + i
       for i = 1 to n do in parallel
          C[RA[i]] = A[i]
     for i = 1 to n do in parallel
          C[RB[i]] = B[i];
end
```

Merging in Parallel by Partitioning

- Time for each binary search is O(log n)
- Number of independent operations = O(n)
- Total time for merging = O(log n),
  - the total work is O(n log n)
  - Compare to sequential time complexity of O(n).
- Can reduce the total work to O(n).
  - Induce partitions in the arrays of equal size
  - Rank one element from each partition
  - Use these ranks to find the ranks of the other elements, sequentially.

How Realistic is Parallel Computation?

- Some discussion.

Insertion Sort

- Suppose you grade 100 exam papers.
- You pick one paper at a time and grade them.
- Now suppose you want to arrange these according to their scores.

Insertion Sort

- A natural way is to start from the first paper onwards.
- The sequence with the first paper is already sorted. Say that score is 37.
- Let us say that the second paper has a score of 25. Where does it go in the sequence?
- Check it with the first paper and then place it in order.

Insertion Sort

- Suppose that we finish k – 1 papers from the pile.
- We now have to place the kth paper. What is its place?
  - Start checking it with one paper in the sorted sequence at a time.
- This is all we need to do for every paper.
- In the end we get a sorted sequence.

Insertion Sort Example

- Consider A = [5 8 4 7 2 6]
- [5 8 4 7 2 6] – > [5 8 4 7 2 6] – > [4 5 8 7 2 6] – > [4 5 7 8 2 6] – > [2 4 5 7 8 6] – > [2 4 5 6 7 8].
- The red portion in the above shows the sorted portion during every iteration.

Insertion Sort Program

```
Program InsertionSort(A)
begin
    for k = 2 to n do
        //place A[k] in its right place
        //given that elements 1 through k-1
        //are in order.
        j = k-1;
        while A[k] < A[j]
            Swap A[k] and A[j];
            j = j -1;
        end-while
    end-for
End-Algorithm.
```

Correctness

- For algorithms described via loops a good technique to show program correctness is to argue via loop invariants.
- A loop invariant states a property of the of an algorithm in execution that holds during every iteration of the loop.
- For the insertion sort algorithm, we can observe the following.

Correctness

- At the start of a certain jth iteration of the loop the array sorted will have j-1 numbers in sorted order.
- Have to prove that the above is indeed an invariant property.

Correctness

- We show three things with respect to a loop invariant.
- Initialization: The LI is true prior to the first iteration of the loop.
- Maintenance: If the LI holds true before a particular iteration then it is true before the start of the next iteration.
- Termination: Upon termination of the loop, the LI can be used to state some property of the algorithm and establish its correctness.

Correctness Proof

- Initialization: For j=2 when the loop starts, the size of sorted is 1 and it is trivially sorted.
  - Hence the first property.
- Maintenance:
  - Let the LI hold at the beginning of the $j$th iteration.
  - To show that LI holds at the beginning of the $j+1$st iteration is not difficult.

Correctness Proof

- Termination: The loop ends when j=n+1.
- When we use the LI for the beginning of the n+1st iteration, we get that the array sorted contains n elements in sorted order.
- This means that we have the correct output.

Run Time Analysis

- Consider an element at index k to be placed in its order.
- It may have to cross k-1 elements in that process.
  - Why?
- This suggests that element at index k requires up to O(k) comparisons.
- So, the total time is $O(n^2)$.
  - Why?

Insertion Sort – Insights

- One step of the algorithm is to move an element one index closer to its final position.
- Compare this approach with that of bubble sort.
- Both have such a huge runtime of $O(n^2)$.

Insertion Sort – Insights

- The algorithm is not oblivious to input.
- Find an input which makes the insertion sort program to make only O(n) comparisons.
- Similarly, find an input which makes the insertion sort program to make only O(n) comparisons.
- So, what is the average number of comparisons?
  - Difficult to answer. Read from Knuth's book.

Insertion Sort – Insights

- Played a good role in the design of a new sorting algorithm called library sort.
  - Idea : As in shelving library books, most of the data is in order.
  - Read from the works of Michael Bender.
- Sorting is an old issue or a research topic?
  - Both at the same time!!

Merge Sort

- Another sorting technique.
- Based on the divide and conquer principle.
- We will first explain the principle and then apply it to merge sort.

Divide and Conquer

- Divide the problem P into $k \geq 2$ sub-problems P_1, P_2, …, P_k.
- Solve the sub-problems P_1, P_2, …, P_k.
- Combine the solutions of the sub-problems to arrive at a solution to P.

Basic Techniques – Divide and Conquer

- A useful paradigm with several applications.
- Examples include merge sort, convex hull, median finding, matrix multiplication, and others.
- Typically, the sub-problems are solved recursively.
  - Recurrence relation

$$T(n) = D(n) + \Sigma\ T(n_i) + C(n)$$

Divide time

Combine time

Recursive cost

Divide and Conquer

- Combination procedure : Merge

$\longrightarrow$ 8 10 12 24 $\qquad\qquad$ $\longrightarrow$ 15 17 27 32

8

8 10

8 10 12

8 10 12 15

8 10 12 15 17

8 10 12 15 17 24

8 10 12 15 17 24 27

8 10 12 15 17 24 27 32

Algorithm Merge

```
Algorithm Merge(L, R)
// L and R are two sorted arrays of size n each.
// The output is written to an array A of size 2n.
int i=1, j=1;
L[n+1] = R[n+1] = MAXINT; // so that index does not
                          // fall over
for k = 1 to 2n do
    if L[i] < R[j] then
        A[k] = L[i]; i++;
    else A[k] = R[j]; j++;
end-for
```

Algorithm Merge

- To analyze the merge algorithm for its runtime, notice that there is a for-loop of 2n iterations.
- The number of comparisons performed is O(n).
- Hence, the total time is O(n).

From Merging to Sorting

- How to use merging to finally sort?
- Using the divide and conquer principle
  - Divide the input array into two halves.
  - Sort each of them.
  - Merge the two sub-arrays. This is indeed procedure Merge.
- The algorithm can now be given as follows.

Correctness of Merge

- We can argue that the algorithm Merge is correct by using the following loop invariant.
- At the beginning of every iteration
  - L[i] and R[j] are the smallest elements of L and R respectively that are not copied to A.
  - A[1..k − 1] is in sorted order and contains the smallest i − 1 and j − 1 elements of L and R respectively.
- Need to verify these statements.

Correctness of Merge

- Initialization : At the start we have i = j = 1 and A is empty. So both the statements of the LI are valid.
- Maintenance : Let us look at any typical iteration k. Let L[i] < R[j].
- By induction, these are the smallest elements of L and R respectively and are not put into A.
- Since we add L[i] to A at position k and do not advance j the two statements of the LI stay valid after the completion of this iteration.

Correctness of Merge

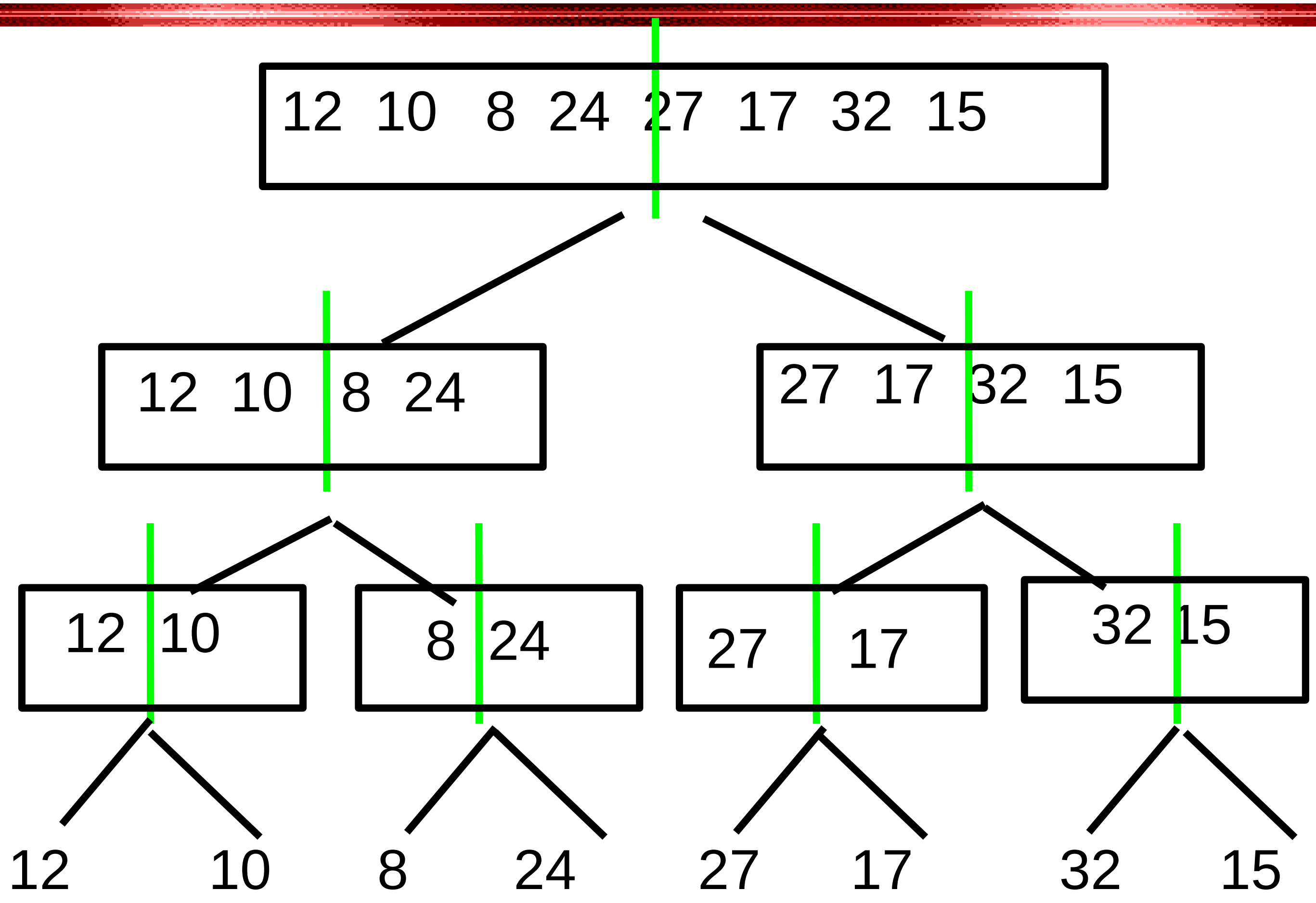
- Termination : At termination $k = l + r + 1$ and by the second statement we have that A contains $k - 1 = l + r$ elements of L and R in sorted order.
- Hence, the algorithm Merge correctly merges the sorted arrays L and R.

Algorithm MergeSort

```
Algorithm MergeSort(A)
begin
    mid = n/2; //divide step
    L = MergeSort(A[1..mid]);
    R = MergeSort(A[mid+1..n]);
    Merge(L, R); //combine step
end-Algorithm
```

- Algorithm mostly self-explanatory.

Divide and Conquer



- Example via merge sort
- Divide is split into two parts
- Recursively solve each subproblem

Runtime of Merge Sort

- Write the recurrence relation for merge sort as T(n) = 2T(n/2)+O(n).
  - This can be explained by the O(n) time for merge and
  - The two subproblems obtained during the divide step each take T(n/2) time.
  - Now use the general format for divide and conquer based algorithms.
- Solving this recurrence relation is done using say the substitution method giving us T(n) = O(n log n).
  - Look at previous examples.

Further Insights – Runtime

- How different is merge sort form insertion sort.
- In insertion sort, data moves by only one position (left or right) at a time.
- Consider an element x at index i,
  - Say x has to be moved to index j in sorted order.
  - Let |i-j| = k.
  - In each iteration of insertion sort, x can be moved only by one position.

Further Insights – Runtime

- Recall, k is the distance by which x has to move.
- So, it takes k moves for x to be in its right position.
- Notice that k can be as large as n/2, where n is the size of the input.
- Further, there can be at least elements with the property that k is at least n/2.
- So, the runtime of insertion sort can be as high as $n^2/4$.

Further Insights – Runtime

- In merge sort however, an element can be moved by a distance as large as n/2.
  - In essence, an element can jump more than one cell at a time,
- Hence the overall runtime does reduce.

Further Insights – Merge Sort is Oblivious

- Irrespective of the input, merge sort always has a runtime of O(n log n).
- Notice that in each stage of the merge sort,
  - input is divided into two halves,
  - each half is sorted, and
  - the sorted halves are merged.
- During the merge process, it is not considered whether the input is already in sorted order or not.
- So, the merge process proceeds oblivious to the input.

Further Insights – Merge Sort is Oblivious

- On the other hand, insertion sort can benefit on certain inputs.
- In fact, the number of swaps that occur in an insertion sort is equal to the number of inversions in the input data.
  - An inversion is a pair of indices i and j so that A[i] > A[j] but i <j, i.e., elements are not in the natural(sorted) order.
- For each such inversion pair, insertion sort has to perform one swap.

The Next Steps...Imagination

- Justify : For merge sort to work, one does not need to know the entire data at once.

Towards External Sorting

- Consider settings where the entire data may not fit into the memory of a computer system.
- In that case, one can bring in as much data as it is possible to work with, sort in piece and then write back the sorted piece, and bring fresh unsorted data in to the memory.
- Once all data is sorted in pieces, these pieces can be then merged in a similar fashion.
- This process is called as "external sorting".
  - we imagine that the data is available externally to the computer.

Towards External Sorting

- Merge sort is one such sorting algorithm that extends naturally to an external setting.
- The algorithm for such a setting is not very difficult to develop.

The Next Steps

- Merge sort can work with only partial data.
- Can we think of a parallel version of merge sort.

  simultaneously, or in parallel.
- Our understanding of parallelism is as follows.

Parallel Merge Sort

- An algorithm is a sequence of tasks T1, T2, ...
- These tasks may have inter-dependecies,
  - Such as task Ti should be completed before task Tj for some i,j.
- However, it is often the case that there are several algorithms where many tasks are independent of each other.
  - In some cases, the algorithm or the computation has to be expressed in that indepedent-task fashion.
  - Example is parallel prefix.

Parallel Merge Sort

- In such a setting, one can imagine that tasks that are independent of each other can be done simultaneously, or in parallel.
- Let us think of arriving at a parallel merge sort algorithm.

Parallel Merge Sort

- What are the independent tasks in merge sort?
  - Sorting the two parts of the array.
  - This further breaks down to sorting four parts of the array, etc.
  - Eventually, every element of the array is a sorted sub-array.
  - So the main work is in merge itself.

Parallel Merge

- So, we just have to figure out a way to merge in parallel.
- Recall the merge algorithm as we developed it earlier.
  - Too many dependent tasks.
  - Not feasible in a parallel model.

Parallel Merge

- Need to rethink on a parallel merge algorithm
- Start from the beginning.
  - We have two sorted arrays L and R.
  - Need to merge them into a single sorted array A.
- Consider an element x in L at index k.
- How many elements of L are smaller than x?
  - k-1.
- How many elements of R are smaller than x?
  - No easy answer, but
  - can do binary search for x in R and get the answer.
  - Say k' elements in R are smaller than x.

Parallel Merge

- How many elements in LUR are smaller than x?
  - Precisely k – 1 + k'.
- So, in the merged output, what index should x be placed in?
  - precisely at k+k'.
- Can this be done for every x in L?
  - Yes, it is an independent operation.
- Can this be done for every x in R also?
  - Yes, replace the roles of L and R.
- All these operations are independent.

Parallel Merge

- The above algorithm can be improved slightly.
- Need more techniques for that.
- So, it is a story left for another day.