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.

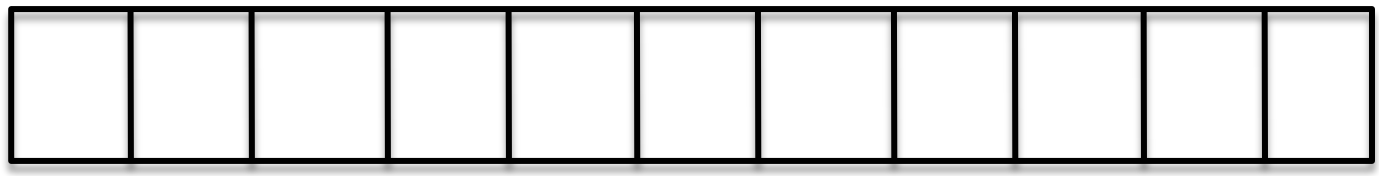
An Abstract Data Type

- 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 an 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?

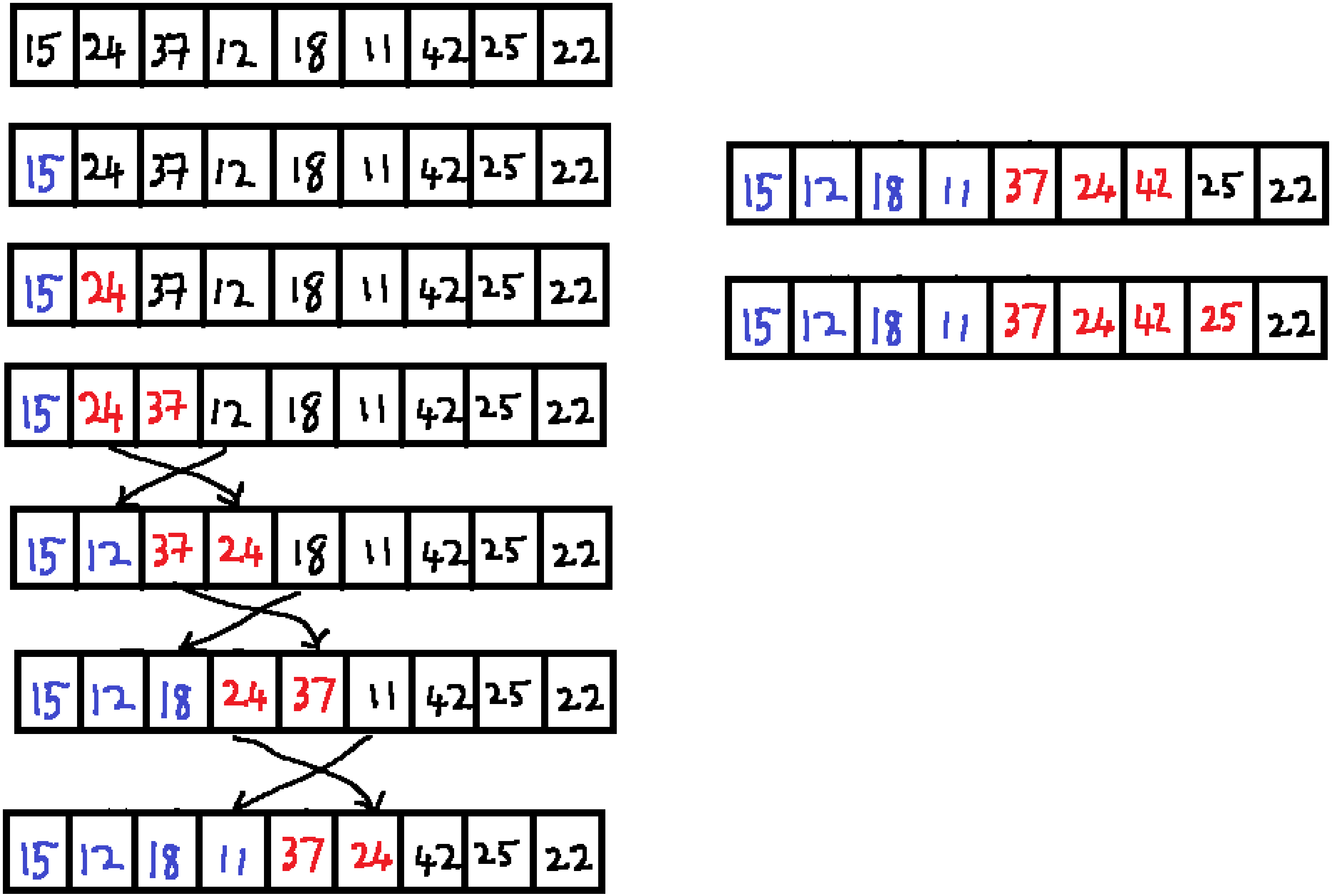
- Here is an idea.
  - Suppose we take each element, compare it with A[1] and then move it to A_L or A_R accordingly.
  - Works in O(n) time.
  - Can write the program easily.
  - But, recall that space is also an resource. The above approach requires extra space for the arrays A_L and A_R
  - A better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


15 24 37 12 18 11 42 25 22
15 24 37 12 18 11 42 25 22
15 24 37 12 18 11 42 25 22
15 24 37 12 18 11 42 25 22
15 12 37 24 18 11 42 25 22
15 12 18 24 37 11 42 25 22
15 12 18 11 37 24 42 25 22
15 12 18 11 37 24 42 25 22
15 12 18 11 37 24 42 25 22

Practice Problem

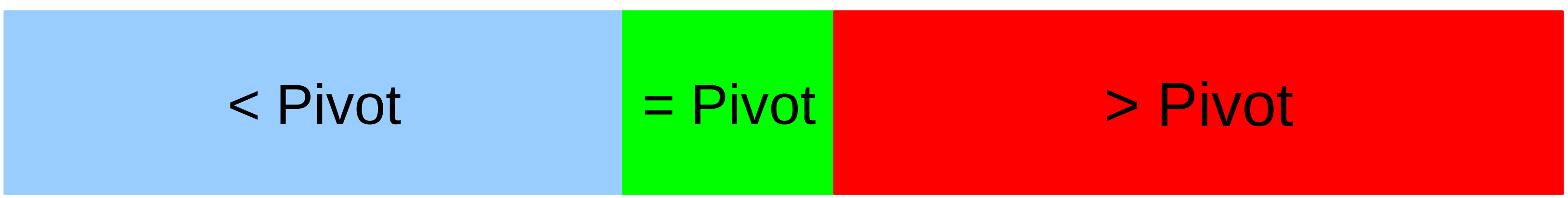
Partition the elements below around the last element as the pivot.

Show all your work.

24, 41, 9, 18, 36, 16, 19, 48, 20

Correctness by Loop Invariants

- Consider the Partition algorithm as an example.
- The Partition algorithm partitions the input data set into three parts around a pivot.



Correctness by Loop Invariants

- How to prove that the above algorithm is correct
  - We shall use a loop invariant.
- How do we come up with a loop invariant?
  - Study the loop for its purpose and construction.
  - What property of the loop can we seek during every iteration?
  - Formalize such a set of statements.

Correctness by Loop Invariants

- Statement of the loop invariant

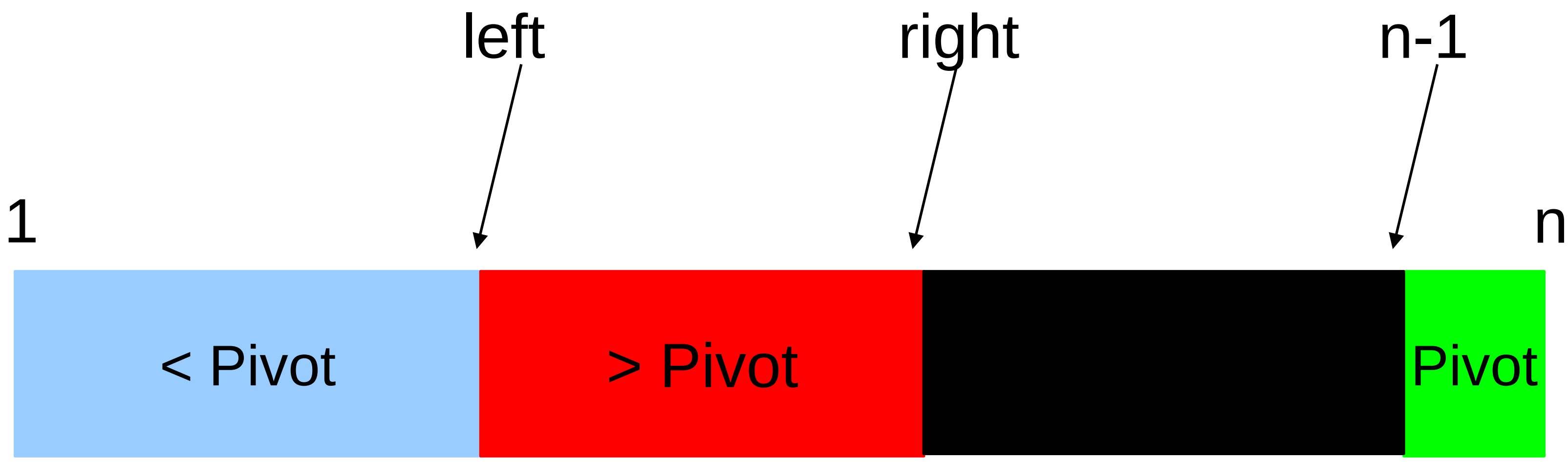
  After k iterations of the loop, the following hold:

  - Elements A(1) to A(left) are less than the pivot.
  - Elements A(left+1) to A(right) are greater than the pivot.
  - Elements A(right+1) to A(n-1) are not classified.
  - A(n) = pivot.

Using a Loop Invariant (LI)

- 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.

The Basic Step in Partition



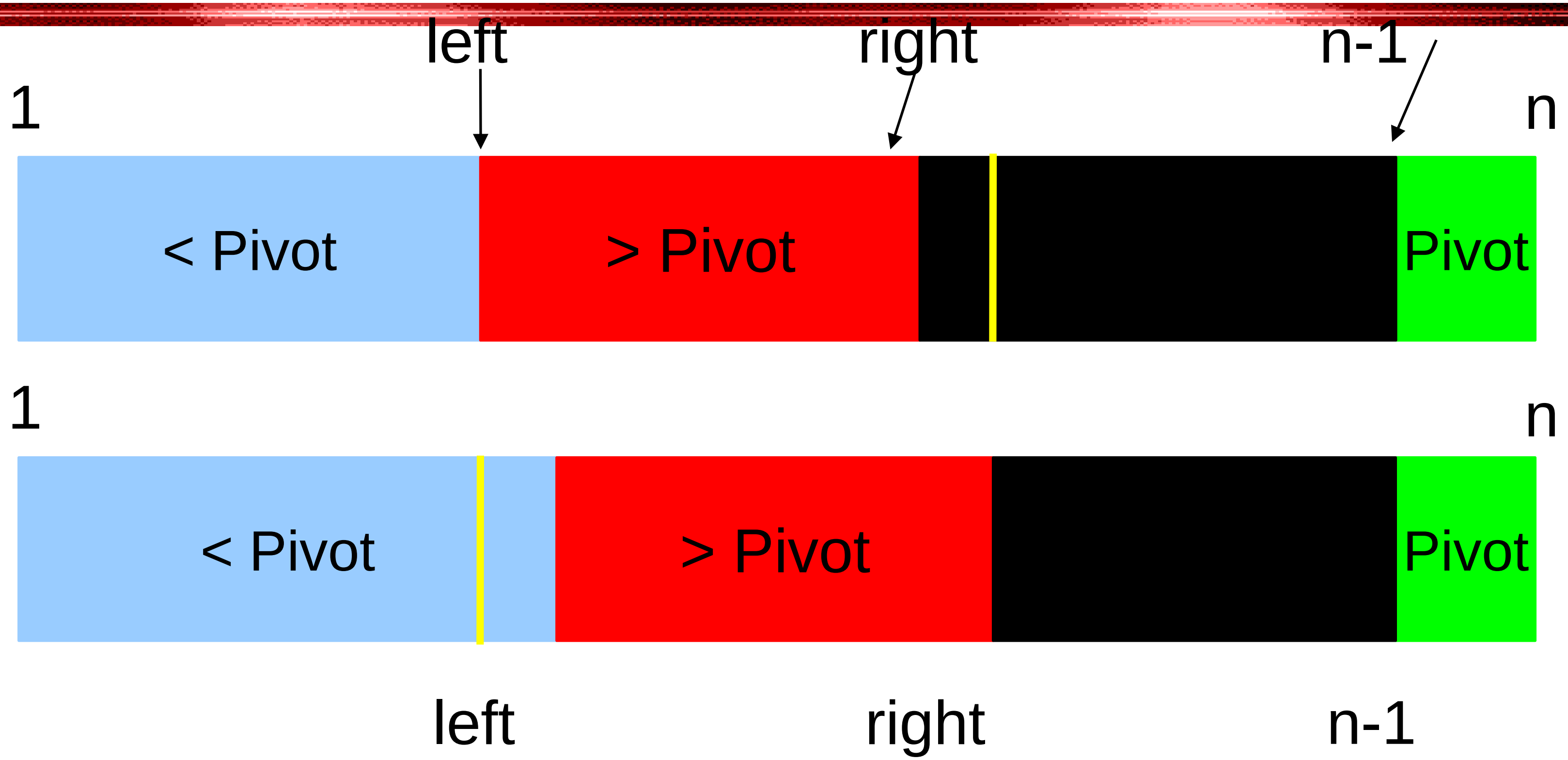
Correctness by Loop Invariants

- Initialization: Check that the loop invariant true before the start of the loop.
- In our example, left = 0, right = 0 before the start of the loop.
- So, the four conditions are met.

Correctness by Loop Invariants

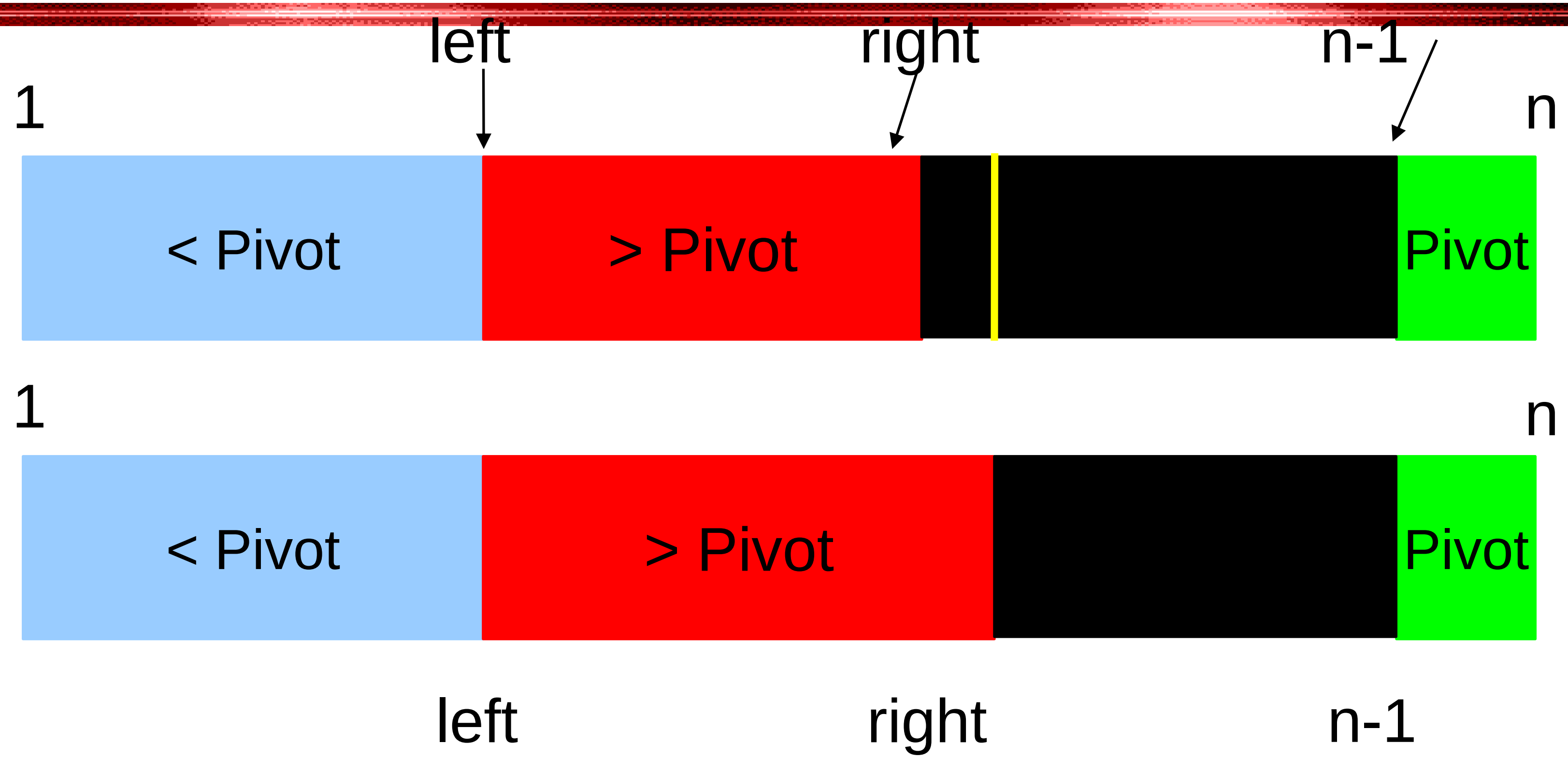
- Maintenance: Assume that the loop invariant is true for the past k iterations.
- Alike induction step, we need to show that also for the k+1 iterations, the loop invariant holds.
- Consider the actions in the loop and their effect on the loop invariant.

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]

Practice Problem:: Loop Invariants

- Consider the following algorithm. What does it do? Formulate an appropriate loop invariant, and show that the algorithm is correct.

```
Algorithm WhatIsThis(X)
Begin
    int i = 1;
    while (i <= n)
        int j = i+1;
        while (j <= n)
            if (X[i] > X[j])
                y = X[i]; X[i] = X[j]; X[j] = y;
          j++;
        i++;
End
```

Practice Problem:: Loop Invariants

- The algorithm is sorting X. The procedure is called bubble sort.
- One possible loop invariant:
  - For the first while-loop: At the end of i iterations, the elements of X are such that X[1] <= X[2] <= … X[i].
  - For the second while-loop: At the end of k iterations of the loop for k = i+1, i+2, …, n, the elements X[i+1], X[i+2], …, X[k] are all smaller than X[i].

Analyzing Quick Sort

- Suppose we run quick sort with A[n] as the pivot.
- Let A_L and A_R be the two subarrays obtained after partitioning.
- What is the time taken by quicksort?
- As a recurrence relation, $T(n) = T(|A_L|) + T(|A_R|) + O(n)$.
- To be able to solve this recurrence relation, need to know the sizes of arrays A_L and A_R.

Analyzing Quick Sort

- We know that $|A_L| + |A_R| = n-1$.
- But, if the pivot is such that all elements are smaller (or larger) than the pivot, then $|A_L|$ (or $|A_R|$) = n-1.
- The recurrence relation in that case is

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

- Suppose the same situation happens over every recursive call. So, the above recurrence relation holds during every recursive call.

Example Bad Case

$$\underbrace{12\quad 15\quad 17\quad 24\quad 29\quad 36}_{A_L} \mid 42$$

$$\underbrace{12\quad 15\quad 17\quad 24\quad 29}_{A_L} \mid 36$$

$$\underbrace{12\quad 15\quad 17\quad 24}_{A_L} \mid 29$$

$$\underbrace{12\quad 15\quad 17}_{A_L} \mid 24$$

$$\underbrace{12\quad 15}_{A_L} \mid 17$$

Analysis of Quick Sort

- Find the solution to the recurrence relation

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

Analysis of Quick Sort

- Is it always that bad?
- What if the pivot is such that each recursive iteration, the sizes of $|A_L|$ and $|A_R|$ is exactly the same?
- The recurrence relation then stands as:

$$T(n) = 2T(n/2) + O(n).$$

- Solve this recurrence relation.

Analysis of Quick Sort

- Which element as the pivot ensures that the sizes of $|A_L|$ and $|A_R|$ are exactly the same?
- Can that happen in every run?

Analysis of Quick Sort

- Which element as the pivot ensures that the sizes of $|A_L|$ and $|A_R|$ are exactly the same?
- Can that happen in every run?
- In general, if the sizes of $|A_L|$ and $|A_R|$ are such that they are a constant away from each other, then the recurrence relation is:

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

where a is a constant < 1.

- Can you solve this recurrence relation?

Analysis of Quick Sort

- In practice, it turns out that most often the partitions are not too skewed.
- So, quick sort runs in O(n log n) time almost always.

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 $S[i] = A[1]+A[2]+ \cdot\cdot\cdot +A[i]$.
- 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.
- 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 for prefix sums 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, $C[i] = B[1] + B[2] + \cdots + B[i] = A[1] + A[2] + \cdots + A[2i]$, 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, $C[i] = B[1] + B[2] + \cdots + B[i] = A[1] + A[2] + \cdots + A[2i]$, 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 C[2], C[4], and C[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]oA[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] o A[i];
    end-for
end
```

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 independent operations.

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
  - $T(n) = T(n/2) + O(1)$
  - Using Master's theorem, 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.

How Realistic is Parallel Computation?

- Our analysis suggests that the computation takes only O(log n) time, but we need n processors for this.
- Cannot ensure that the number of processors also grow with the input size.
- In practice, the number of processors on a machine does not change!

How Realistic is Parallel Computing

- The idea of the parallel algorithm is to show the extent of parallelism available in the computation.
- Plus, if there are fewer processors than what is required, can always simulate more processors.
- For instance, if there are p processors and n processors are required, then each of the p processors simulates the actions of n/p processors.

How Realistic is Parallel Computation

- Practical experience indicates that this is a viable proposition.

Merge Sort and Parallel 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

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

Divide time

Combine time

Recursive cost

Divide and Conquer

- Combination procedure : Merge

$\longrightarrow$ 8 10 12 27 $\longrightarrow$ 15 17 24 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 – Practice Problem

- Analyze the merge algorithm for its runtime.

Algorithm Merge – Practice Problem

- 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).
- Is it correct?

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.

From Merging to Sorting

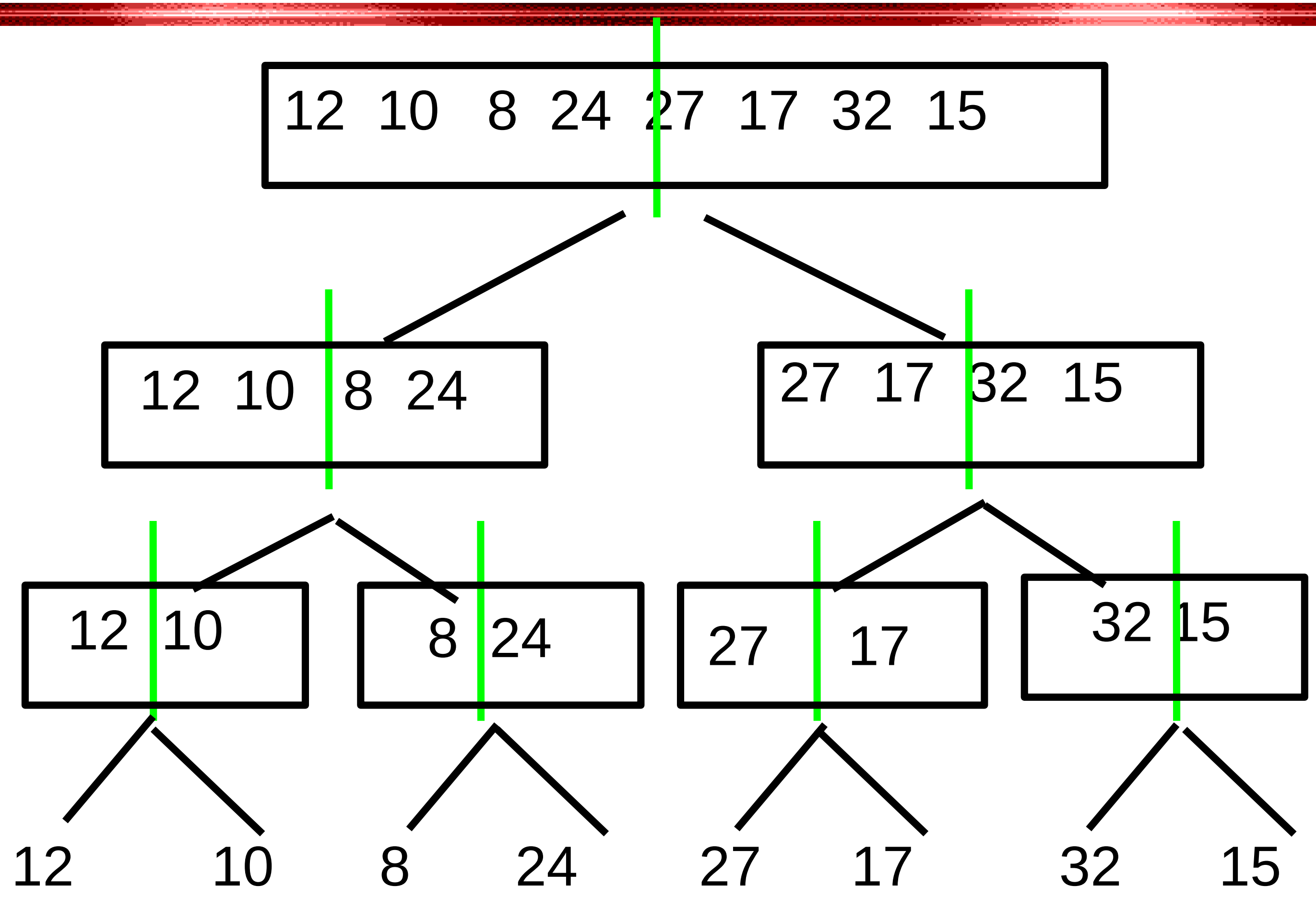
- How to use merging to finally sort?
- Using the divide and conquer principle
  - Divide the input array into two halves.
  - Sort each of them.
  - Merge the two sub-arrays. This is indeed procedure Merge.
- The algorithm can now be given as follows.

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 and solve it.

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.

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.
- Define the rank of an element x in a sorted array A as the number of elements of A that are smaller than x.
- To merge L and R, need to know the rank of every element from L and R in the merged array L U R.

Parallel Merge

- Importantly, for any x in L or R,

  Rank(x, L U R) = Rank(x, L) + Rank(x, R).

- So, merging is equivalent to finding the two ranks on the right hand side.

Parallel Merge

- Now, 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 larger than x.

Parallel Merge

- How many elements in LUR are smaller than x?
  - Precisely k + k‘ - 1.
- 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.

Example

L = [8 10 12 27]

R = [15 17 24 32]

Element	8	10	12	27	15	17	24	32
Rank in L	0	1	2	3	3	3	3	4
Rank in R	0	0	0	3	0	1	2	3
Rank in L U R	0	1	2	6	3	4	5	7

L U R = [8 10 12 15 17 24 27 32]

Parallel Merge

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

Towards Parallel Sorting

- Use the parallel merge algorithm to sort.

```
Algorithm ParallelMergeSort(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
```