DIVIDE AND CONQUER

DIVIDE AND CONQUER

Divide and conquer is used for data parallelism

- The same operation is done on all parts of the data (add, sort, min/max)
- Some parts of the data can be eliminated (binary search)

Complexity of divide and conquer

- $T(n) = \text{Constant}$ (time for smallest subproblem)
  - $T(n) = aT(n/b) + D(n) + C(n)$

Each subproblem divided into 1/b the portion

- $D(n)$: Time to divide
- $C(n)$: Time to combine
- Can be solved by Master's Theorem in most cases

SOME EXAMPLES

Given a set of integers

- Find the sum
- Find the minimum
- Find whether a number exists

Find n!

Matrix Multiplication

Find the most frequently occurring number

Divide and Conquer can be applied to any associative operation (add, multiply, etc.) on a series of numbers

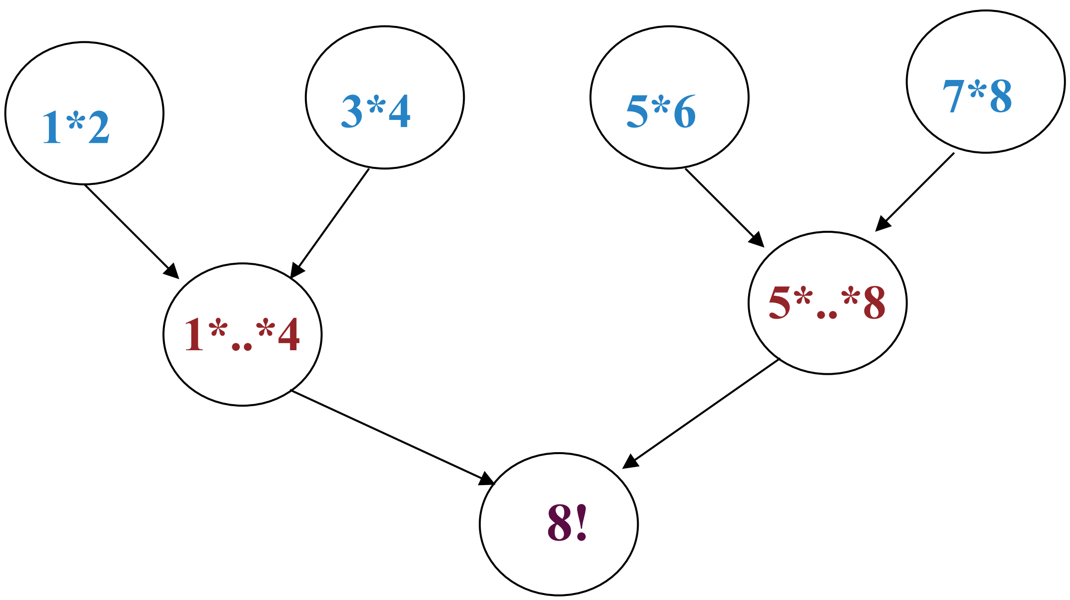
Often the complexity is not changed

However, we can apply parallelism to improve the speed

COMPUTING THE FACTORIAL

Find n!

- Divide the numbers into k processors
- Compute sequential products on elements in each processor
- Multiply the results from each processor

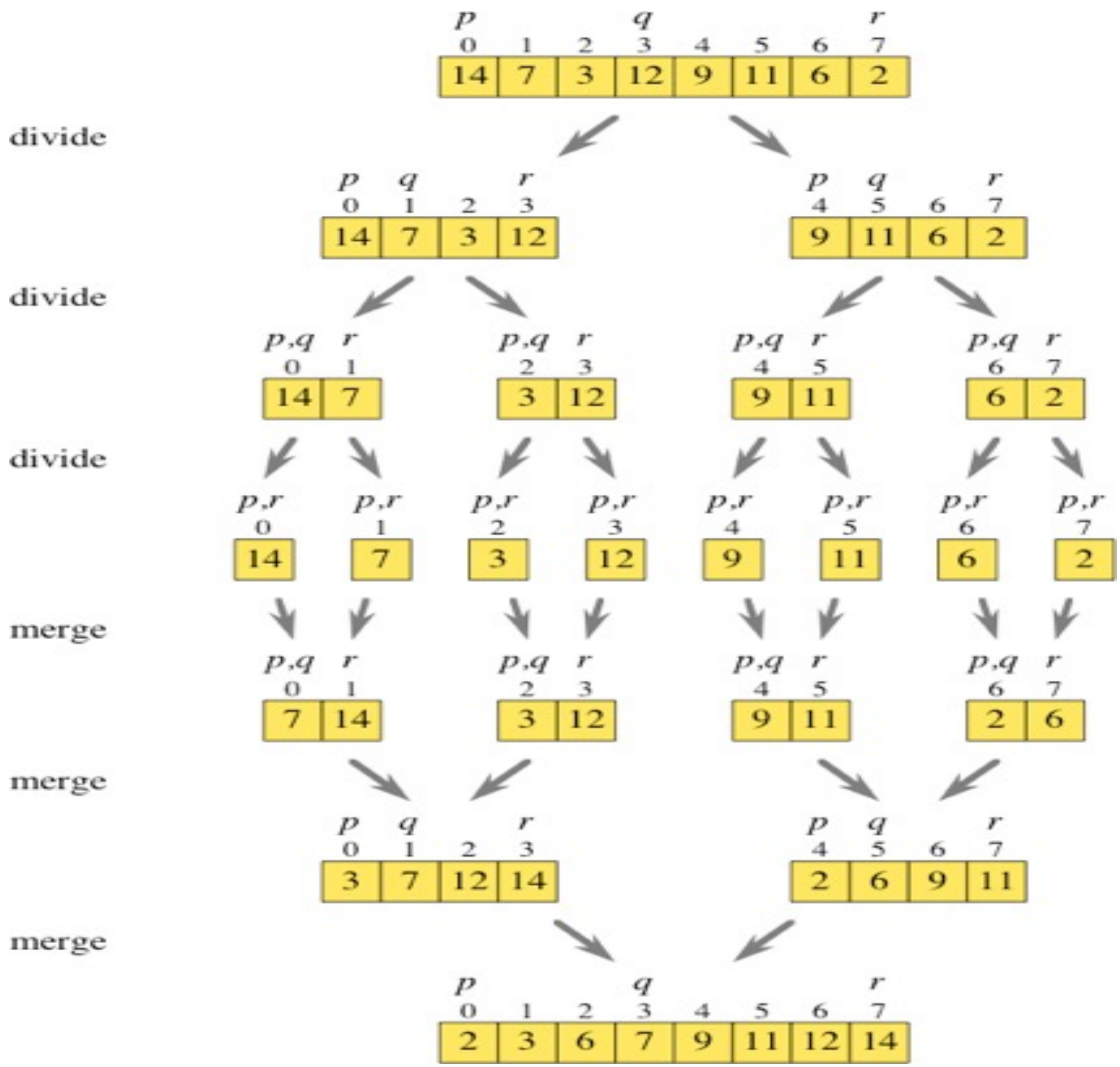


With p processors and n entries

Multiplication: n/p
Combining log(p)

Total n/p+log(p)

With sequential= n


divide
divide
divide
merge
merge
merge
p q r
0 1 2 3 4 5 6 7
14 7 3 12 9 11 6 2
p q r
0 1 2 3
14 7 3 12
p q r
4 5 6 7
9 11 6 2
p,q r
0 1
14 7
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
6 2
p,r 0 14
p,r 1 7
p,r 2 3
p,r 3 12
p,r 4 9
p,r 5 11
p,r 6 6
p,r 7 2
p,q r
0 1
7 14
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
2 6
p q r
0 1 2 3
3 7 12 14
p q r
4 5 6 7
2 6 9 11
p q r
0 1 2 3 4 5 6 7
2 3 6 7 9 11 12 14

COMPLEXITY OF MERGE SORT

Dividing the array:

- Constant time C1

We divide it in two equal size arrays that have to be processe

- 2T(n/2)

Combining two arrays each of size n

- C2*2(n/2)=C2*n

Total Complexity: C1+2T(n/2)+C2*n

We can ignore C1 as it is less than the combining complexity of n

T(1)=1

T(n)= 2T(n/2)+C2*n

QUICKSORT

Quicksort is one of the fastest known sorting algorithm in practice.

Quicksort is a divide-and-conquer recursive algorithm.

Average running time O(N logN).

$O(N^2)$ worst-case performance, but this can be made exponentially unlikely with a little effort.

QUICKSORT ALGORITHM

quicksort(S):

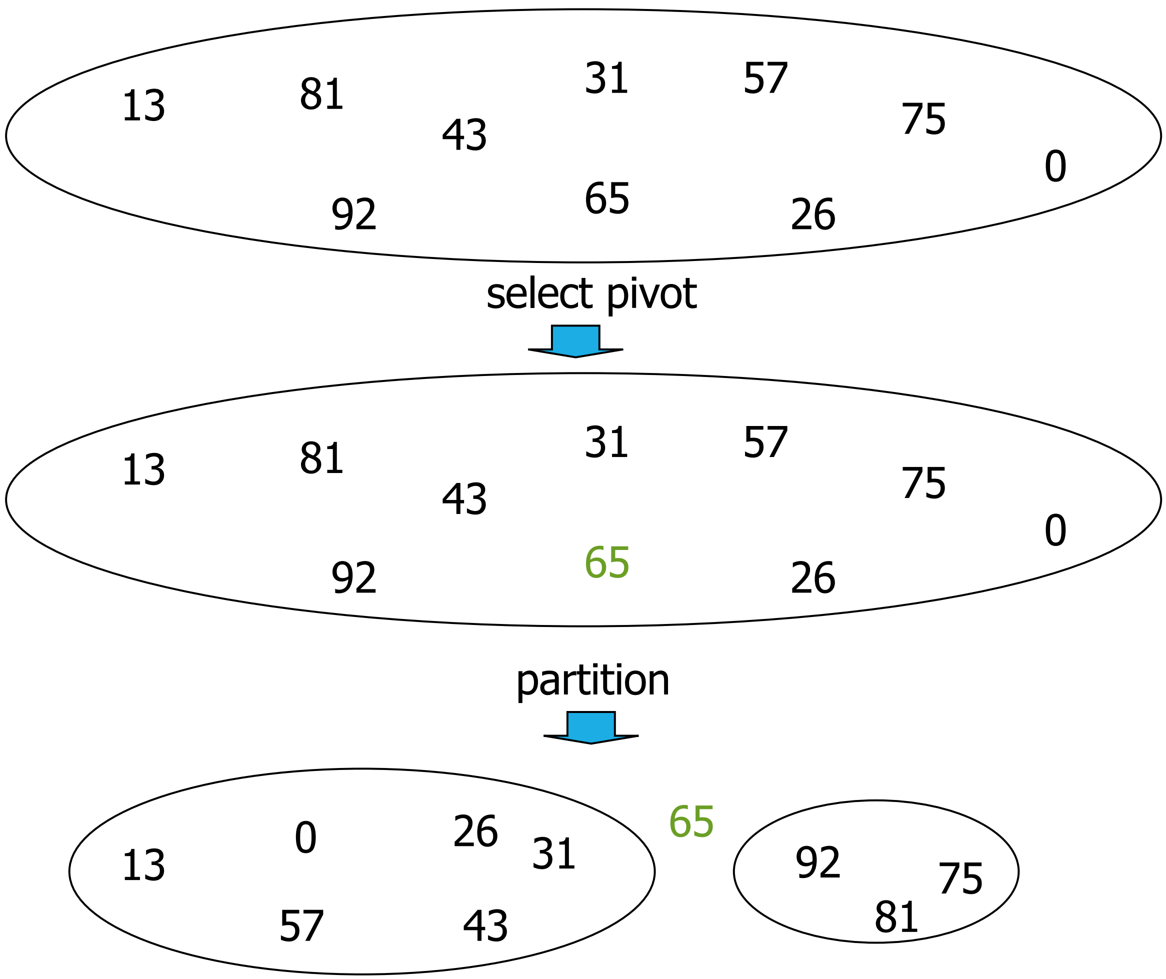
If the number of elements in S is 0 or 1, then return.

Pick any element v in S. This is called the pivot.

Partition $S-\{v\}$ (the remaining elements in S) into two disjoint groups: $S_1=\{x \in S-\{v\} \mid x \leq v\}$, and $S_2=\{x \in S-\{v\} \mid x \geq v\}$.

Return {quicksort(S_1) followed by v followed by quicksort(S_2)}.

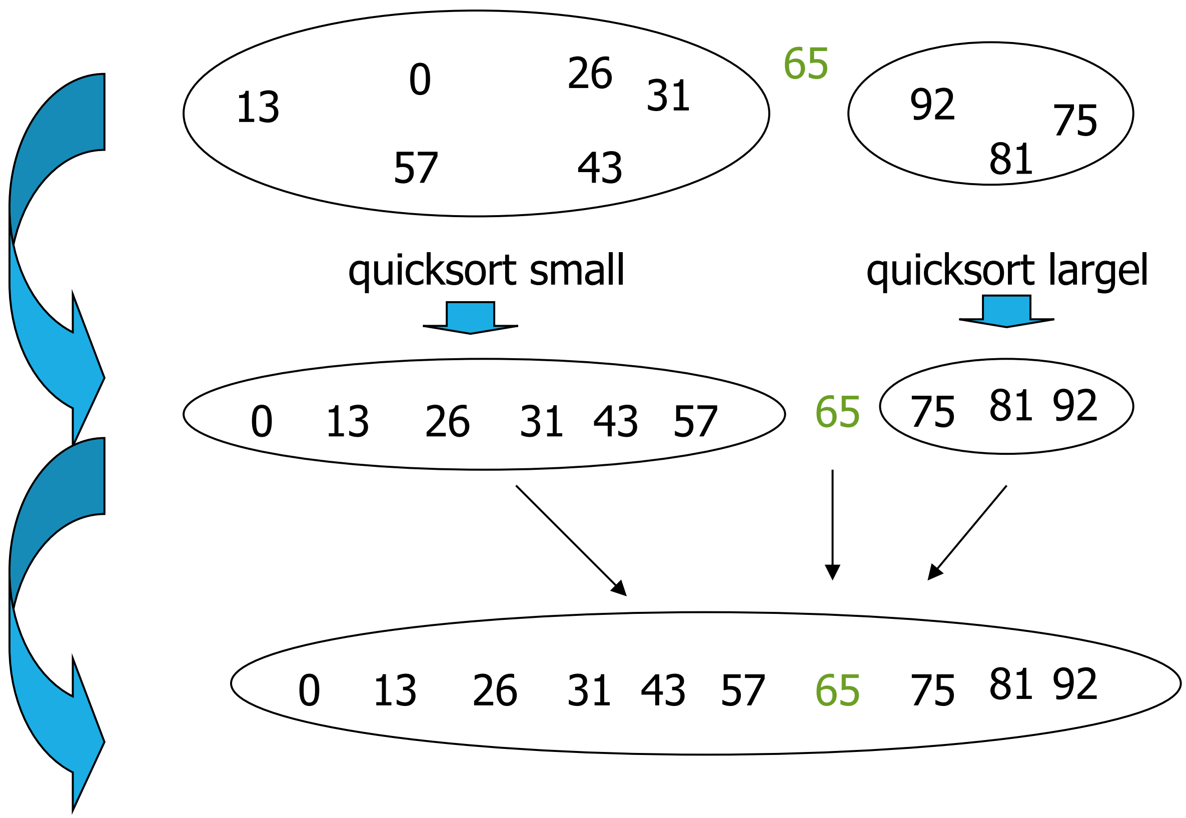
QUICKSORT


13
81
43
31
57
75
0
92
65
26
select pivot
13
81
43
31
57
75
0
92
65
26
partition
13
0
26
31
57
43
65
92
75
81

QUICKSORT

Recursively sorting the set of smaller numbers

Merge



WHY IS IT FASTER THAN MERGESORT?

It is clear that this algorithm works but it is not clear why it is faster than mergesort.

- Like mergesort, it recursively solves two subproblems and requires linear additional work.
- Unlike mergesort, the subproblems are not guaranteed to be of equal size. (potentially bad)
- Picking a good pivot makes the partitions well balanced
- The main reason: the partitioning step can be performed in place and very efficiently. This efficiency more than makes up for the lack of equal-sized recursive calls.
- However for small arrays insertion sort works better

PICKING PIVOT: MEDIAN-OF-THREE

8	1	4	9	5	3	7	2	6	0
↑				↑					↑
Left				Center					Right

Pivot is 5

0	1	2	3	4	5	6	7	8	9
↑				↑					↑
Left				Center					Right

Using median-of-three partitioning eliminates the bad case for sorted input.

Picking Pivot: median-of-three

8 1 4 9 5 3 7 2 6 0

pivot

0 1 4 9 5 3 7 2 6 8

Swap 5 and 6

0 1 4 9 6 3 7 2 5 8

i j pivot

We are now *ready for partitioning*

PARTITIONING STRATEGY

Move all the small element to the left part of the array and all the large elements to the right part.

- Place pivot at position right-1;
- Set i = left and j =right-2;
- Repeat until i < j
  - If a[i] < pivot, i++
    - Move i right, skipping over elements that are smaller than the pivot.
  - If a[j] > pivot, j--
    - Move j left, skipping over elements that are large than the pivot.
  - Swap a[i] with a[j].
- Finally, Swap the pivot element with the element pointer to by i.

PARTITIONING

0 1 4 9 6 3 7 2 5 8

i j pivot

Swap 9 and 2

0 1 4 2 6 3 7 9 5 8

i j pivot

Swap 6 and 3

0 1 4 2 3 6 7 9 5 8

i j pivot

At this state, i and j have crossed, so no swap is performed.

PARTITIONING STRATEGY

0 1 4 2 3 6 7 9 5 8

j i pivot

Finally, swap a[i] with pivot

0 1 4 2 3 5 7 9 6 8

ANALYSIS OF QUICKSORT

- The running time of quick sort is equal to the running time of the two recursive calls plus the linear time spent in the partition (the pivot selection takes only constant time)

$T(0) = T(1) = 1$

$T(N) = T(i) + T(N - i - 1) + N$ (time for partitioning)

where $i = |S_1|$ is the number of elements in S_1.

Best Case Analysis

- Best Case: the pivot is always in the middle; i = n/2

- T(1)=1, T(0)=1
  T(N)=T(N/2)+T(N/2)+O(N)

Pivot operation

Right recursion

Left recursion

Looks familiar?

WORST-CASE ANALYSIS

Worst-case: the pivot is the smallest element all the time; i = 0;

$T(0) = T(1) = 1$

$T(N) = T(0) + T(N - 1) + O(N)$

Pivot operation

N-1 recursion

0 recursion

MAXIMUM SUBARRAY SUM

Given an array of numbers that contains both positive and negative values, find the sum of the continuous subarray which has the largest sum.

{-2, -5, **6, -2, -3, 1, 5**, -6} = maximum subset sum is 7

Complexity=2T(n/2)+n+1

1. Divide A[low...high] into two subarrays of as equal size as possible by finding the midpoint mid

2. Conquer:
(a) finding maximum subarrays of A[low...mid] and A[mid + 1...high]
(b) finding a max-subarray that crosses the midpoint

3. Combine: returning the max of the three

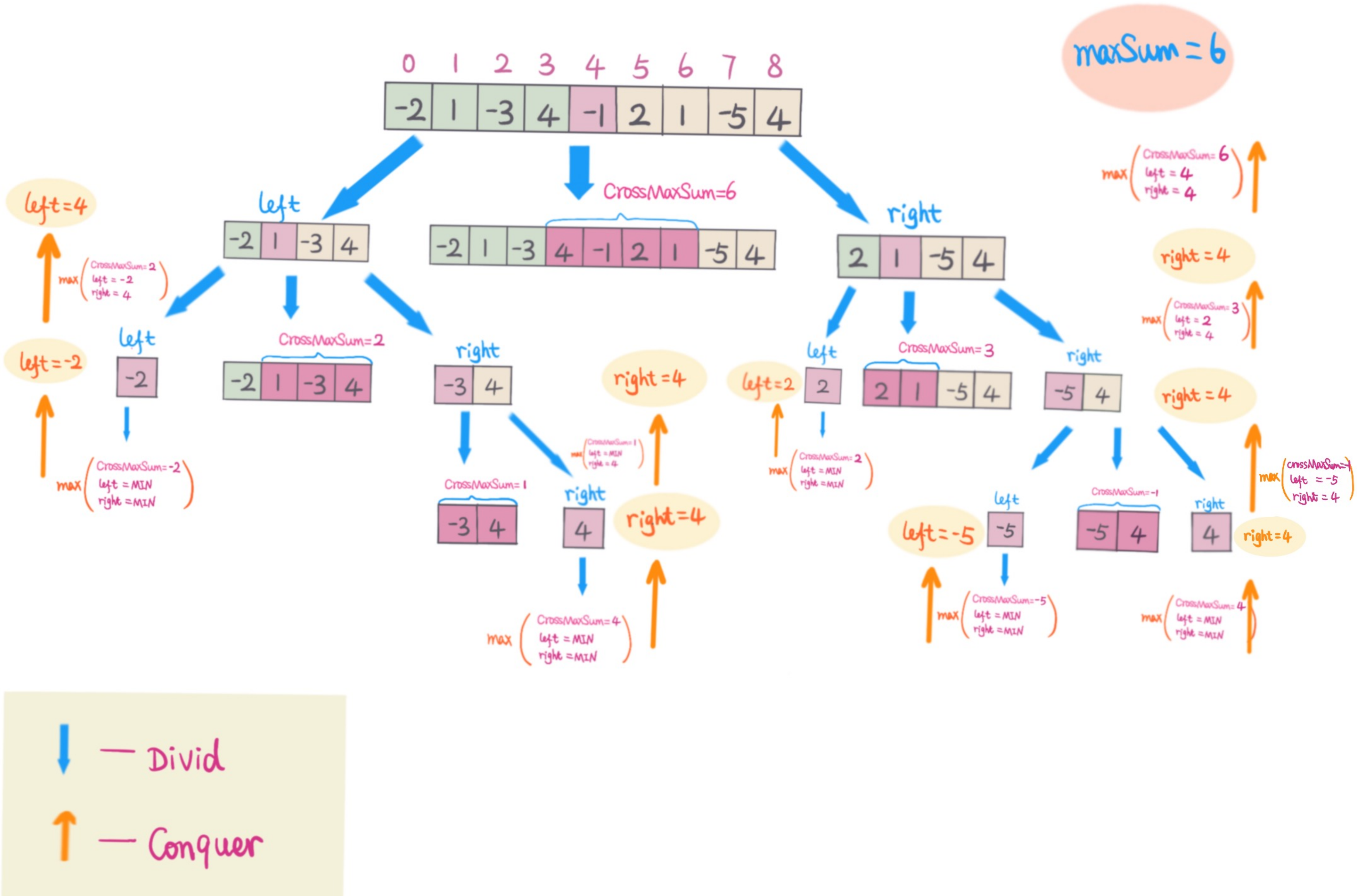


Image from: https://snowan.gitbook.io/study-notes/leetcode-33/english-solution/53.maximum-sum-subarray-en

BINARY SEARCH

Find whether a target number exists in a sorted array

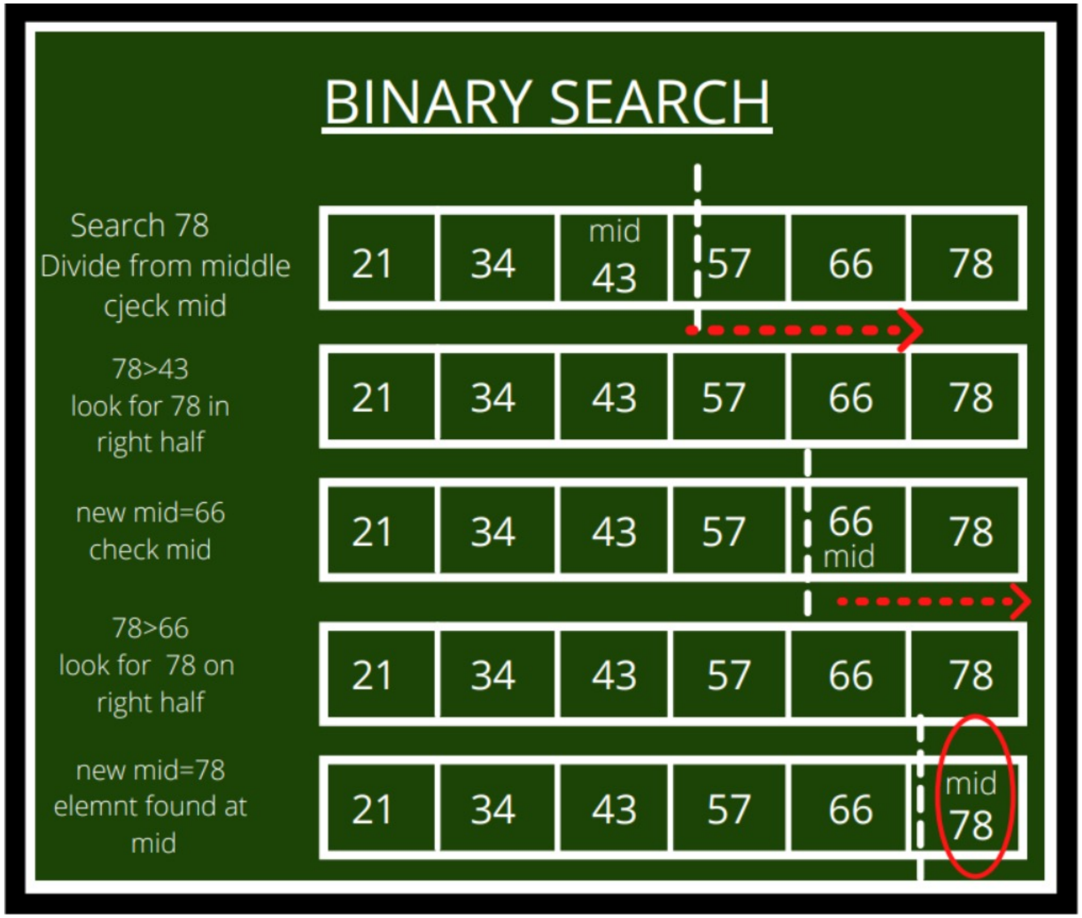
Divide array into 2; get the middle number

If target > middle number recursively check the right half

Otherwise check the left half

Stop when number if found

Complexity: $2T(n/2)+1$



FINDING THE MINIMUM IN A CIRCULARLY SORTED ARRAY

Sorted array

{2,4,7,8,10,11,15}

Circularly sorted means that we have rotated the array n times

- First rotation
- {15,2,4,7,8,10,11}
- Second rotation
- {11, 15,2,4,7,8,10}
- Third rotation
- {10, 11, 15,2,4,7,8}

Note the the minimum number is the only one whose predecessor and successor are higher than it.

Position of the minimum number also gives us the number of rotations done.

FINDING THE MINIMUM IN A CIRCULARLY SORTED ARRAY

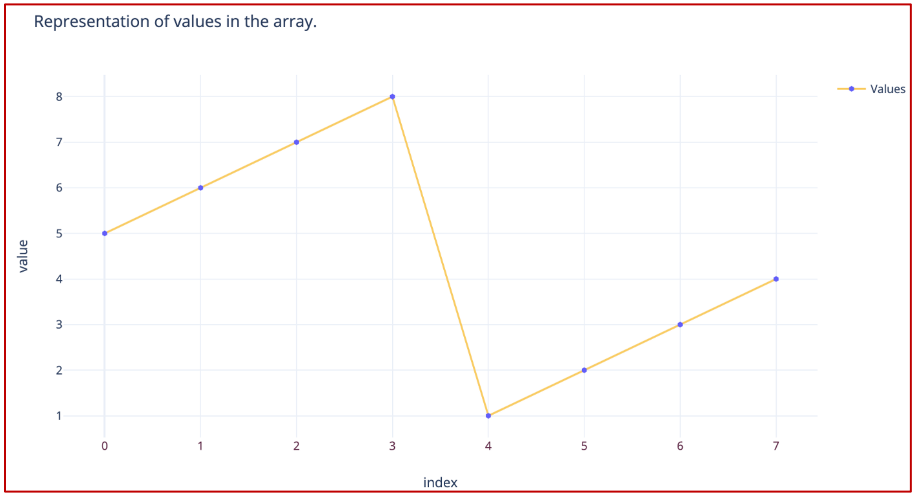
Check that the array is rotated, i.e. first element is not the minimum.

Find middle element in array A =>A[mid]

If A[mid-1] > A[mid] & A[mid+1]> A[mid]; then A[mid] is the minimum

If A[mid]> A[0]; recursively sort for A[mid to end]

Else A[mid]< A[0]; recursively sort for A[0 to mid]



KARATSUBA'S ALGORITHM

Multiplying two numbers

$X=x1B^m + x0$; $Y=y1B^m + y0$;

Traditionally

$X*Y= (x1B^m + x0)(y1B^m + y0)$

$=x1y1B^{2m}+(x1y0+y1x0) B^m+x0y0$ => 4 multiplications

Complexity (for binary numbers)=$4T(n/2)+n$ (length of number ≈ number of multiplications)

Karatsuba's method

$z2=x1y1$; $z1=x1y0+y0x1$; $z0=x0y0$

$Z1=(x1+x0)(y1+y0)-z2-z0$ => only 3 multiplications needed

Complexity $=3T(n/2)+n$