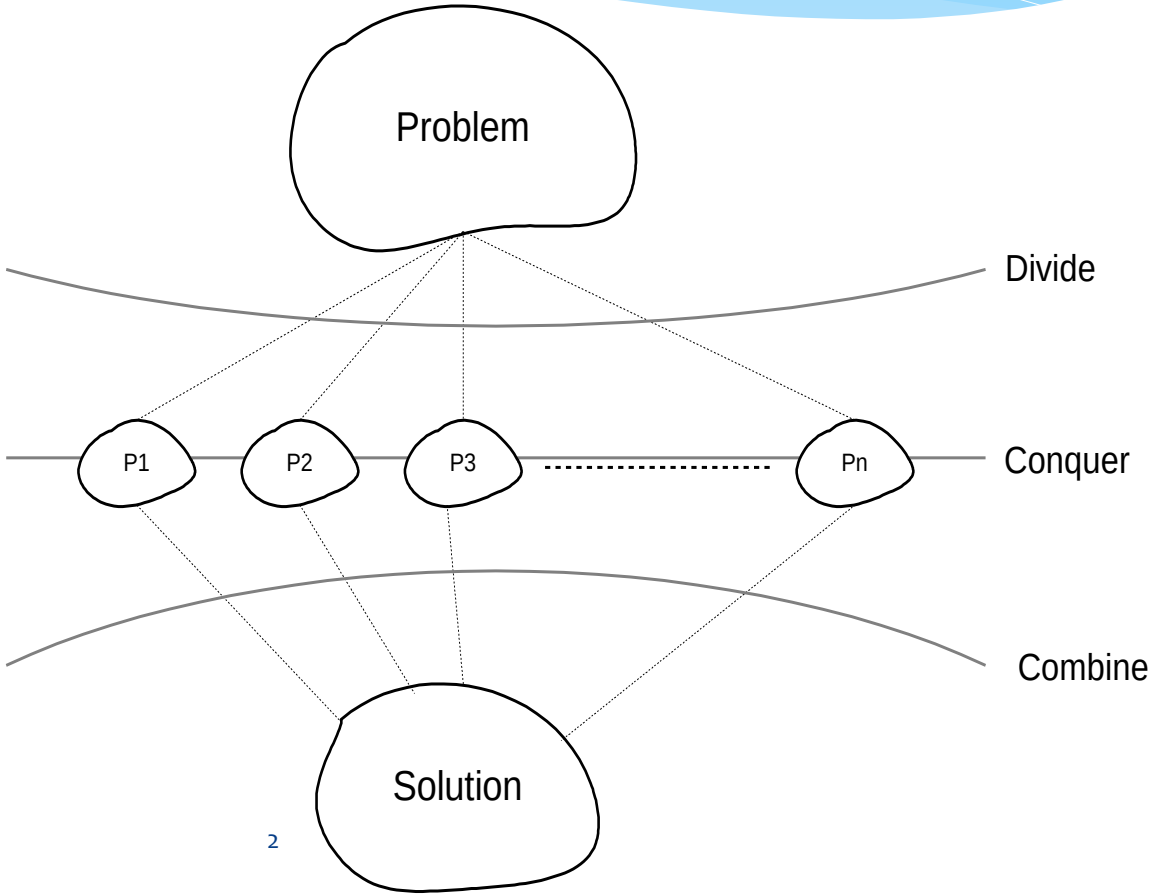
Lecture 8: Sorting Algorithms

Sorting by Exchange: Quick Sort

- Divide-and-Conquer



QUICK SORT (Basic Ideas)

- (Another divide-and-conquer algorithm)
  - Pick an element, say P (the pivot)
  - Re-arrange the elements into 3 sub-blocks,
    1. Those less than or equal to (<=) P (the left-block S1)
    2. P (the only element ¡n the middle-block)
    3. Those greater than or equal to (>=) P (the right-block S2)

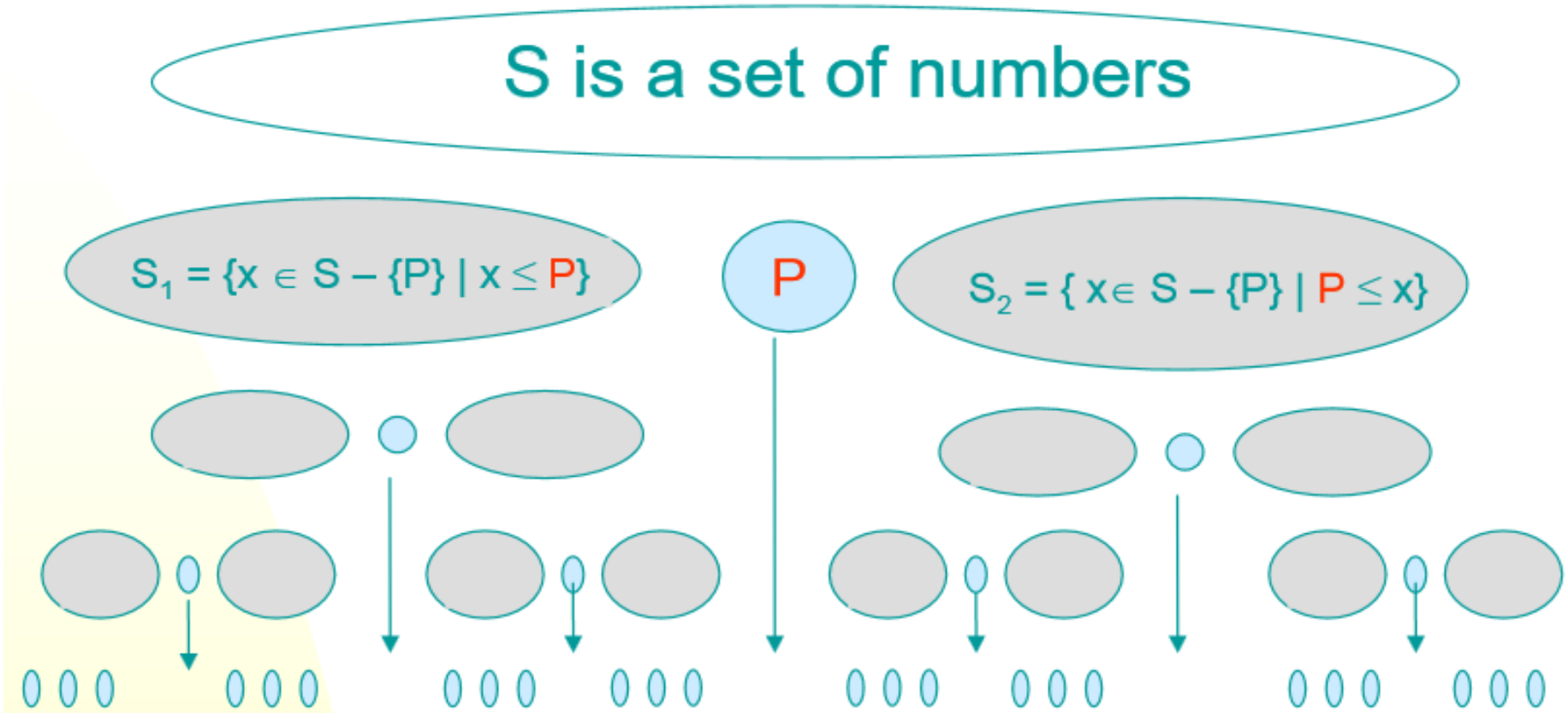
  - Repeat the process recursively for the left- and right- sub-blocks. Return (qu¡cksort(S1), P,quicksort(S2)}.

  - That is the results of quicksort(S1), followed by P, followed by the results of quicksort(S2)

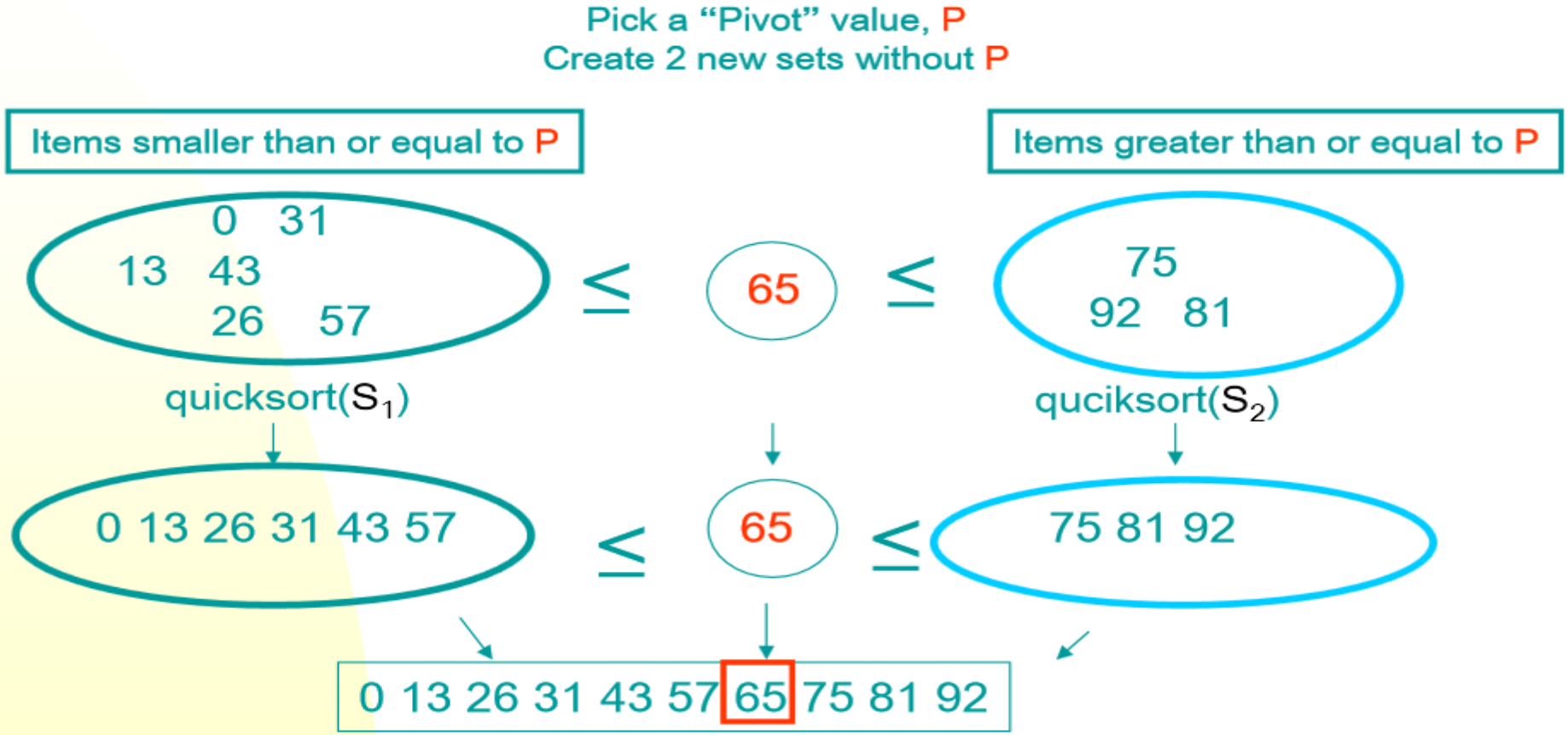
QUICK SORT (Basic Ideas)

- The main idea is to find the “right” position for the pivot element P.
- After each “pass”, the pivot element, P, should be “**in place**”.
- Eventually, the elements are sorted since each pass puts at least one element (i.e., P) into its final position.

QUICK SORT (Basic Ideas)



QUICK SORT (Basic Ideas)



Quick sort-example 1

Input:	65	70	75	80	85	60	55	50	45	
P: 65	*i*									
Pass 1:	**65**	70	75	80	85	60	55	50	45	
(i)		*i*							*j*	← swap (A[i], A[j])
	65	**45**	75	80	85	60	55	50	**70**	
(ii)			*i*					*j*		← swap (A[i], A[j])
	65	**45**	**50**	80	85	60	55	**75**	**70**	
(iii)				*i*			*j*			← swap (A[i], A[j])
	65	**45**	**50**	**55**	85	60	**80**	**75**	**70**	
(iv)					*i*	*j*				← swap (A[i], A[j])
	65	**45**	**50**	**55**	**60**	**85**	**80**	**75**	**70**	
(v)					*j*	*i*				`if (i>=j) break`
	60	**45**	**50**	**55**	**65**	**85**	**80**	**75**	**70**	swap (A[left], A[j])

Items smaller than or equal to 65

Items greater than or equal to 65

Quick sort-example 1

Result of Pass 1: 3 sub-blocks:

60 45 50 55 **65** 85 80 75 70

Pass 2a (left sub-block):

60 45 50 55 (P = 60)

i *j*

60 45 50 55

j i if (i>=j) break

55 45 50 60 swap (A[left], A[j])

Pass 2b (right sub-block):

85 80 75 70 (P = 85)

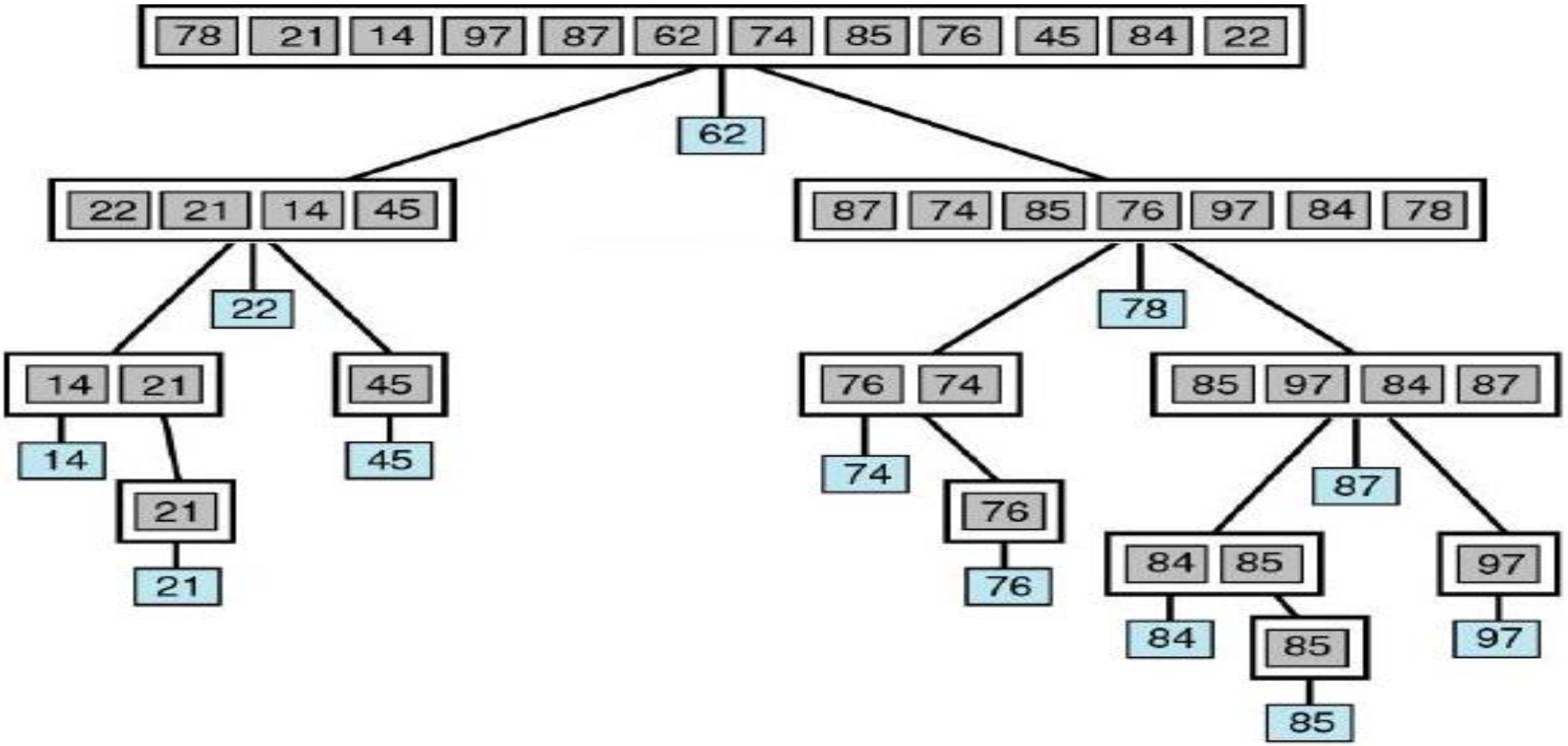
i *j*

85 80 75 70

j i if (i>=j) break

70 80 75 85 swap (A[left], A[j])

Quick sort-example 2



Quick sort-algorithm

- Let A be a linear array of *n* elements A (1), A (2), A (3).....A (*n*), low represents the lower bound pointer and up represents the upper bound pointer. Key represents the first element of the array, which is going to become the middle element of the sub-arrays.

1. Input *n* number of elements in an array A
2. Initialize low = 2, up = *n* , key = A[(low + up)/2]
3. Repeat through step 8 while (low < = up)
4. Repeat step 5 while(A [low] > key)
5. low = low + 1
6. Repeat step 7 while(A [up] < key)
7. up = up–1
8. If (low < = up)
   (*a*) Swap = A [low]
   (*b*) A [low] = A [up]
   (*c*) A [up] = swap
   (*d*) low=low+1
   (e) up=up–1
9. If (1 < up) Quick sort (A, 1, up)
10. If (low < n) Quick sort (A, low, n)
11. Exit

quicksort()

```
int partition(int arr[], int left, int right)
{
    int i = left, j = right;
    int tmp;
    int pivot = arr[(left + right) / 2];
    while (i <= j) {
        while (arr[i] < pivot)
            i++;
        while (arr[j] > pivot)
            j--;
        if (i <= j) {
            tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
            i++;
            j--;
        }
    };
    return i;
}
```

```
void quickSort (int arr[], int left, int right) {
    int index = partition(arr, left, right);
    if (left < index - 1)
        quickSort(arr, left, index - 1);
    if (index < right)
        quickSort(arr, index, right);
}
```

Picking the pivot

- A "good" pivot is one that creates two even sized partitions.
- Median will be best, but finding median could be as tough as sorting itself.
- How about choosing the first element?
  - What if array already or nearly sorted?
  - Good for a randomly populated array

Input: 65 70 75 80 85 60 55 50 45

- How about choosing a random element?
  - Good in practice if "truly random"
  - Still possible to get some bad choices
  - Requires execution of random number generator

Running time analysis

- Partitioning Step: Time Complexity is θ**(n).**

- Recall that quick sort involves partitioning, and 2 recursive calls.
  - Thus, giving the basic quick sort relation:
    - **T(n) = θ(n)+ T(i) + T(n-i-1) = cn+ T(i) + T(n-i-1)**
  - where **i** is the size of the first sub-block after partitioning.

  - We shall take **T(0) = T(1) = 1** as the initial conditions.

Running time analysis

- **Worst-Case** (Data is already sorted)

- When the pivot is the smallest (or largest) element at partitioning on a block of size **n**, the result
  - Yields one empty sub-block, one element (pivot) in the "correct" place and one sub-block of size **(n-1)**
  - Takes θ**(n)** times
- Recurrence Equation:

  T(1) = 1

  T(n) = T(n-1) + cn
- Solution: θ**(n^2)**

  Worse than Mergesort!!!

Running time analysis

- **Best case:**
- The pivot is in the middle (median) (at each partition step), i.e. after each partitioning, on a block of size **n,** the result
  - Yields two sub-blocks of approximately equal size and the pivot element in the "middle" position
  - Takes **n** data comparisons.
- Recurrence Equation becomes

  T(1) = 1

  T(n) = 2T(n/2) + cn
- Solution: θ**(n logn)**

  Comparable to Merge sort!!

So the trick is to select a good pivot

- Different ways to select a good pivot.
  - First element
  - Last element
  - Median-of-three elements
    - Pick three elements, and find the median x of these elements. Use that median as the pivot.
  - Random element
    - Randomly pick a element as a pivot.