Quick Sort

- As the name implies, it is quick, and it is the algorithm generally preferred for sorting.

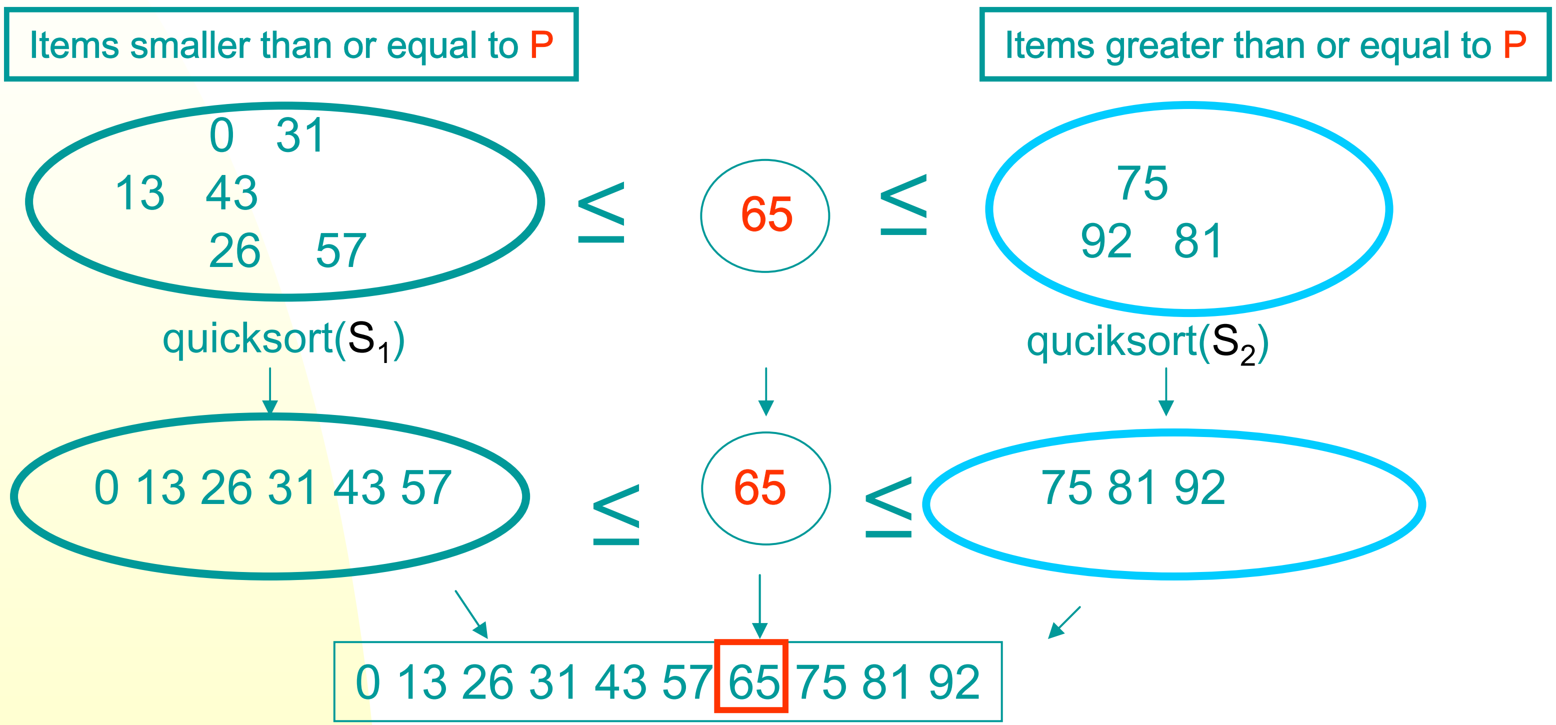
Basic Ideas

(Another divide-and-conquer algorithm)

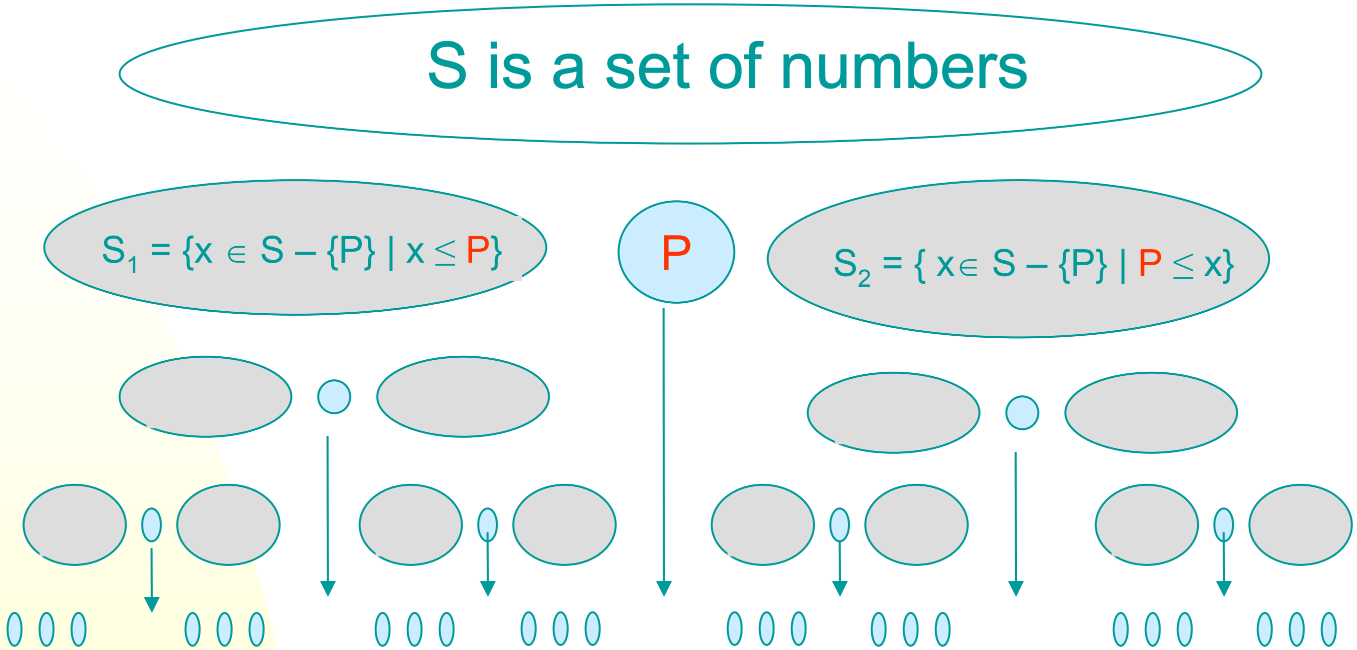
- Pick an element, say ***P*** (the pivot)
- Re-arrange the elements into 3 sub-blocks,
  1. those less than or equal to (≤) ***P*** (the **left-**block S_1)
  2. ***P*** (the only element in the **middle**-block)
  3. those greater than or equal to (≥) ***P*** (the **right-**block S_2)
- Repeat the process **recursively** for the **left-** and **right-** sub-blocks. Return {quicksort(S_1), ***P***, quicksort(S_2)}. (That is the results of quicksort(S_1), followed by ***P***, followed by the results of quicksort(S_2))

Basic Ideas

Pick a “Pivot” value, P
Create 2 new sets without P



Basic Ideas



Basic Ideas

Note:

- 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.

Issues:

- How to choose the pivot ***P*** ?
- How to partition the block into sub-blocks?

Implementation

Algorithm I:

```
int partition(int A[ ], int left, int right);

// sort A[left..right]
void quicksort(int A[ ], int left, int right)
{   int q ;
    if (right > left )
    {
     q = partition(A, left, right);
     // after 'partition'
     //→ A[left..q-1] ≤ A[q] ≤ A[q+1..right]
     quicksort(A, left, q-1);
     quicksort(A, q+1, right);
    }
}
```

Implementation

```
// select A[left] be the pivot element)
int partition(int A[], int left, int right);
{   P = A[left];
    i = left;
    j = right + 1;
    for(;;) //infinite for-loop, break to exit
    { while (A[++i] < P) if (i >= right) break;
      // Now, A[i] ≥ P
      while (A[--j] > P) if (j <= left) break;
      // Now, A[j] ≤ P
      if (i >= j ) break; // break the for-loop
      else swap(A[i], A[j]);
    }
    if (j == left) return j ;
    swap(A[left], A[j]);
    return j;
}
```

Example

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

Example

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])

Running time analysis

- The advantage of this quicksort is that we can sort "in-place", i.e., *without* the need for a temporary buffer depending on the size of the inputs. (cf. mergesort)

<u>Partitioning Step</u>: Time Complexity is θ**(n).**

Recall that quicksort involves *partitioning, and 2 recursive calls.*

Thus, giving the basic quicksort relation:

$$T(n) = \theta(n) + T(i) + T(n-i-1) = cn + T(i) + T(n-i-1)$$

where ***i*** is the size of the first sub-block after partitioning.

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

To find the solution for this relation, we'll consider three cases:

1. The Worst-case (?)
2. The Best-case (?)
3. The Average-case (?)

All depends on the value of the pivot!!

Running time analysis

Worst-Case (Data is sorted already)

- 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:

$$\begin{cases} T(1) = 1 \\ T(n) = T(n-1) + cn \end{cases}$$

Solution: $\theta(n^2)$

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

$$\begin{cases} T(1) = 1 \\ T(n) = 2T(n/2) + cn \end{cases}$$

Solution: $\theta(n \log n)$

Comparable to Mergesort!!

Running time analysis

Average case:

It turns out the average case running time also is θ***(n logn).***

We will wait until COMP 271 to discuss the analysis.

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.

Different sorting algorithms

Sorting Algorithm	Worst-case time	Average-case time	Space overhead
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$
Quick Sort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(1)$

Something extra : Selection problem.

Problem statement.

- You are given a unsorted array A[1..n] of (distinct) numbers, find a element from the array such that its rank is *i*, i.e., there are exactly *(i-1)* numbers less than or equal to that element.

Example :

A={5, 1, 2, 3, 12, 20, 30, 6, 14, -1, 0}, i=8.

Output = 12, since “6, 1, -1, 0, 2, 3, 5” (8-1=7 numbers) are all less than or equal to 12.

Selection problem : a easy answer.

A Easy algorithm.

- Sort A, and return A[i].
- Obviously it works, but it is slow !!! ***Θ (n log n) in average.***
- We want something faster.

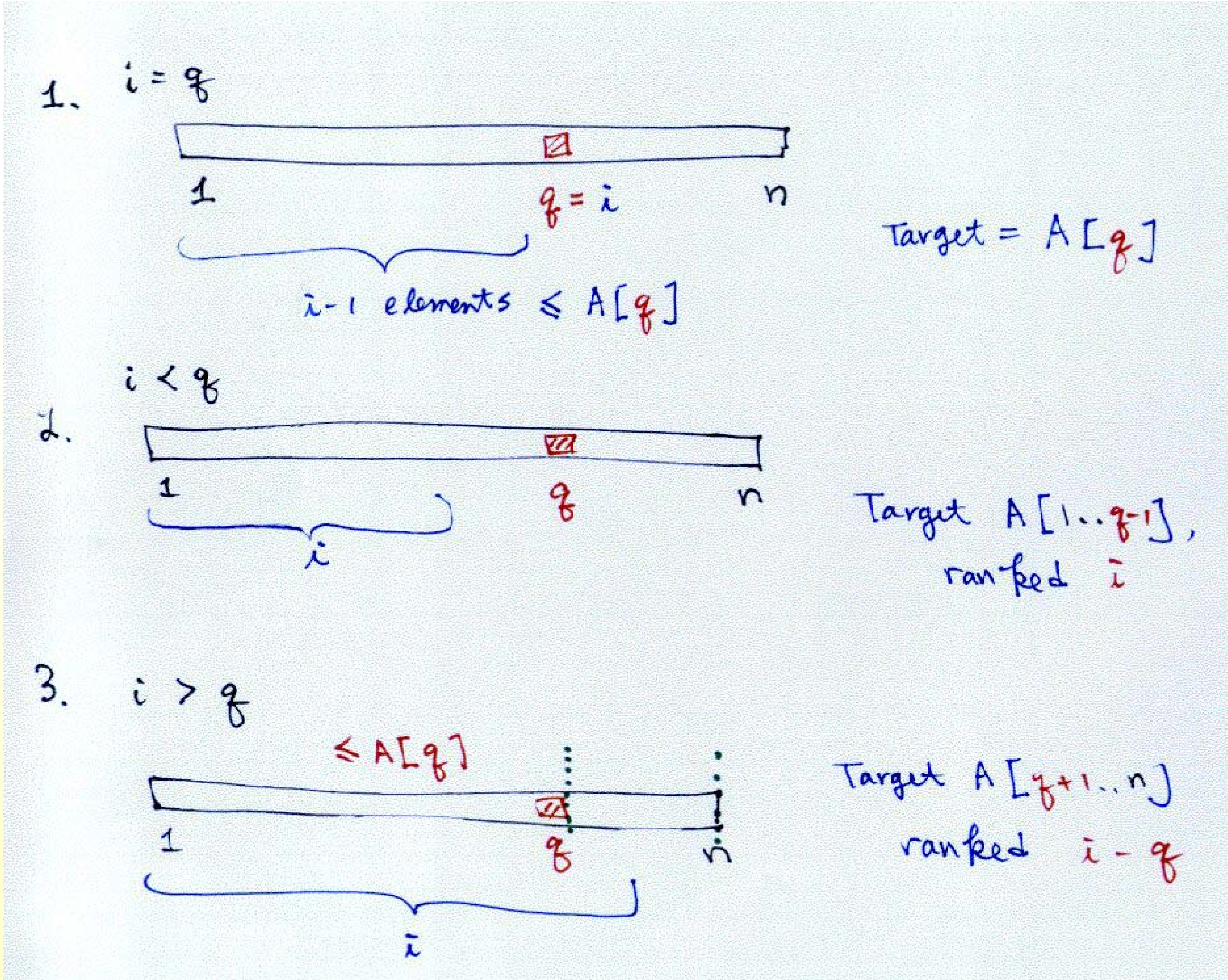
Selection problem : a 'faster' answer.

We can borrow the idea from the partition algorithm.

Suppose we want to find a element of rank i in A[1..n].

After the 1st partition call (use **a random element** as pivot):

1. If the return index 'q' = i, then A[q] is the element we want. (Since there is exactly i-1 elements smaller than or equal to A[q]).
2. If the return index 'q' > i, then the target element can NOT be in A[q .. right]. The target element is rank i in A[1.. q-1]. → Recursive call with parameters (A, 1, q-1, i).
3. If the return index 'q' < i, then the target element can NOT be in A[1 .. q]. The target element is rank i-q in A[q+1 ..n]. → Recursive call with parameters (A, q+1,n, i-q).


1. i = q
1
q = i
n
i-1 elements ≤ A[q]
Target = A[q]
i < q
2.
1
q
n
i
Target A[1..q-1],
ranked i
3. i > q
≤ A[q]
1
q
n
i
Target A[q+1..n]
ranked i - q

A ‘faster’ selection algorithm : Codes

```
RANDOMIZED-SELECT(A, p, r, i)
if p = r
    then return A[p]
q ← RANDOMIZED-PARTITION(A, p, r)
k ← q − p + 1
if i = k          ▷ the pivot value is the answer
    then return A[q]
elseif i < k
    then return RANDOMIZED-SELECT(A, p, q − 1, i)
else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

return a element of rank i in A[p..r]

A[p..q-1] $\leq$ A[q] $\leq$ A[q+1..r]

size of A[p..q]= k

Analysis the ‘faster’ answer.

- Though we claim it is a ‘fast’ algorithm, the worst-case running time is $O(n^2)$ (see if you can prove it).
- But the **average-case** running time is only $O(n)$. (Again, we will see the analysis in COMP 271).
- There is an algorithm that runs in $O(n)$ in the worst case, again, we will talk about that in COMP 271.