ALGORITHMS AND DATA STRUCTURES

MERGE, HEAP, AND QUICK SORT (OF)

SORTING PROBLEM

- How to organize (according to a notion of ordering) an array of elements?
- How fast can it be done? How slow? Bounds
- Different types of ordering: lexicographical, numerical, …
- Output of a sorting algorithm must satisfy
  - it is in nondecreasing order
  - it is a permutation of the input data

FACTS ABOUT (INTERNAL) SORTING

- There are several easy algorithms to sort in $O(N^2)$, such as insertion sort
- There is an algorithm, `shellsort`, that is very simple to code, runs in $o(N^2)$, and is efficient in practice
- There are slightly more complicated $O(N \lg N)$ sorting algorithms
- Any general-purpose sorting algorithm requires $O(N \lg N)$ comparisons

HEAPSORT

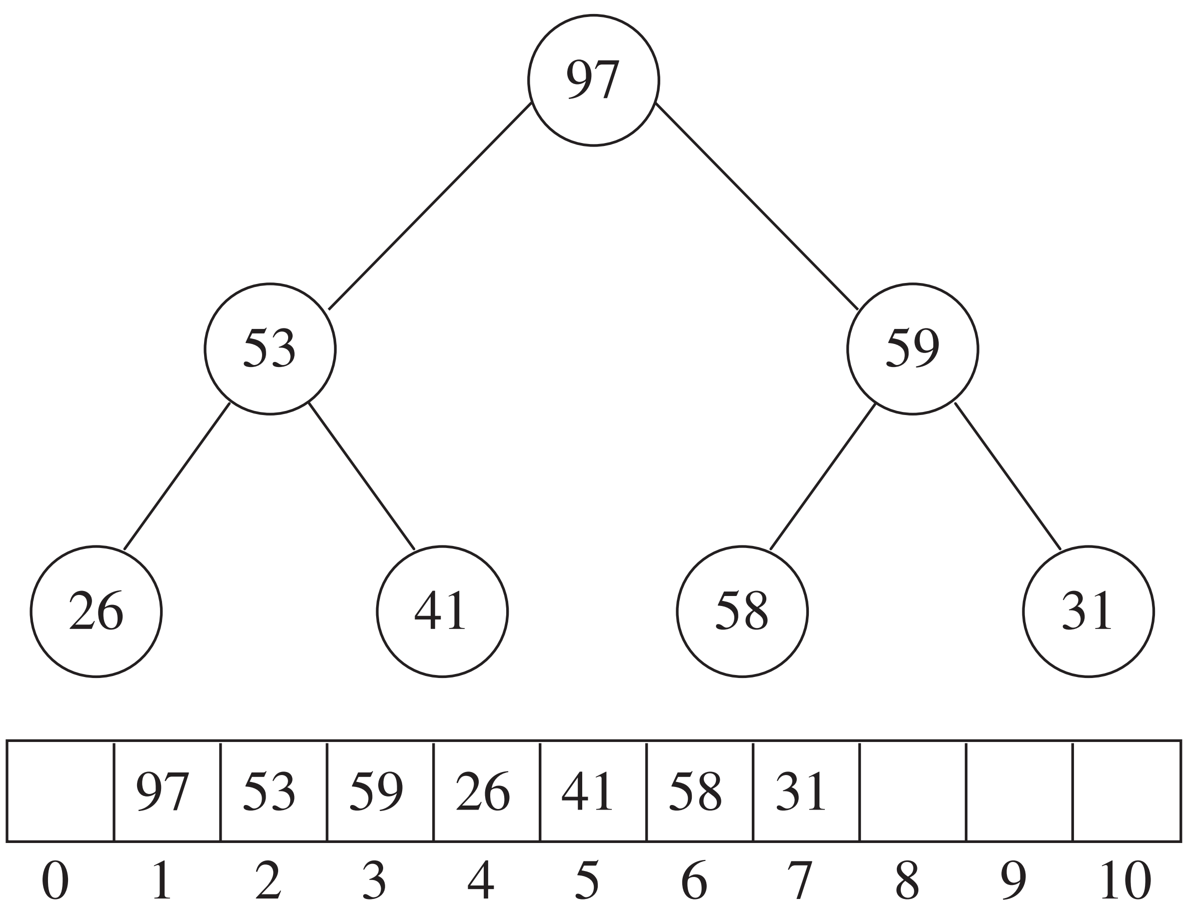
- Priority queues can be used to sort in $O(N \lg N)$ time
- Basic idea
  - build a binary heap of N elements with input data
  - perform N deletes and register data in secondary array
- Problem: we need an extra array to carry out the sorting
  - memory requirement is doubled `Ideas how to solve this?`
  - time in copying second array to the original one

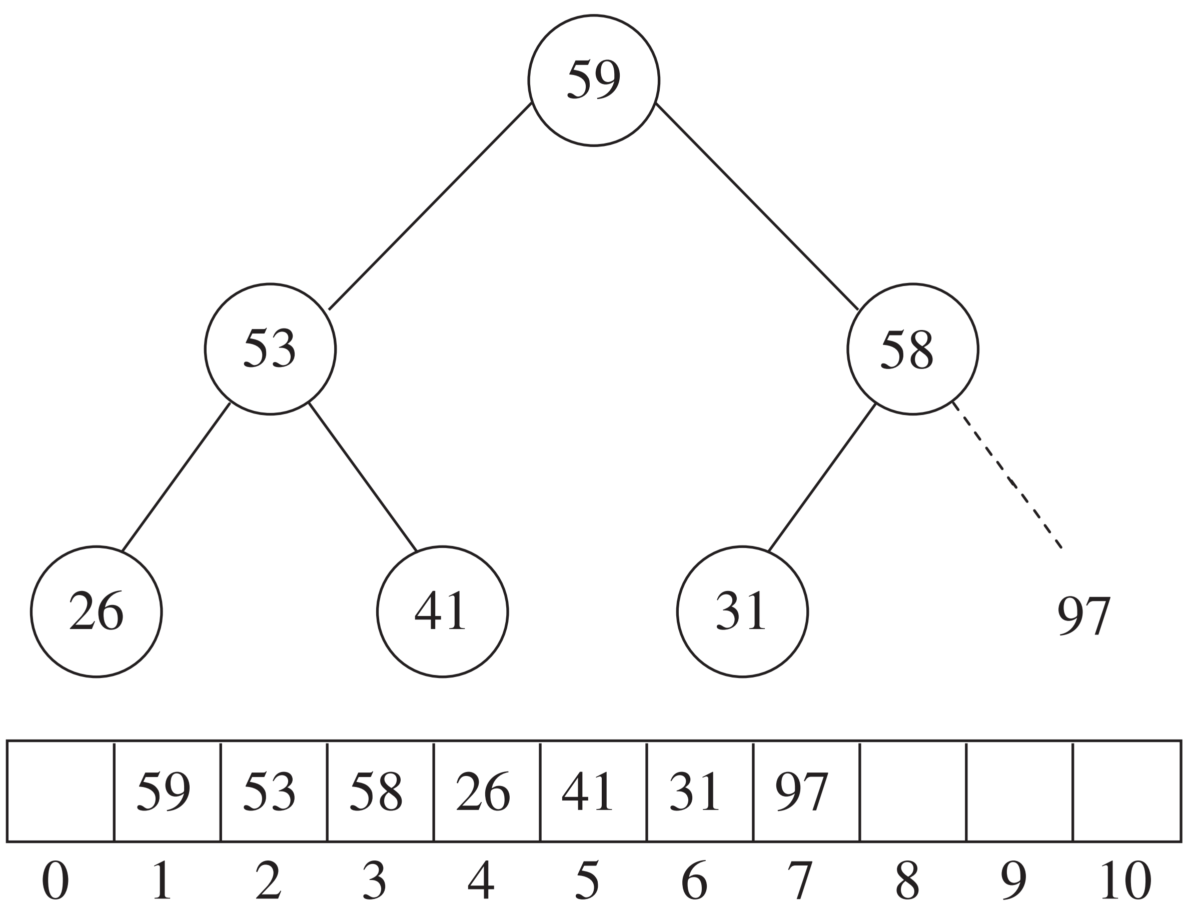
HEAPSORT

- Build min-heap with the input data
- Take advantage of extra space in heap array
  - put element removed in last position
  - end up with sorted array in decreasing order
- Problem: array in reverted order
  - Solution:… use max-heap instead of min-heap

HEAPSORT

- build (max-)heap and start removing…





```
// watch the limits
template <typename Comparable>
void heapsort(vector<Comparable> & a) {
    // build heap
    for(int i = a.size() / 2 - 1; i >= 0; --i)
        percolateDown(a, i, a.size());

    for(int j = a.size() - 1; j > 0; --j) {
        // delete max
        std::swap(a[0], a[j]);
        percolateDown(a, 0, j);
    }
}

// for 0-based arrays
int leftChild(int i){
    return 2 * i + 1;
}
```

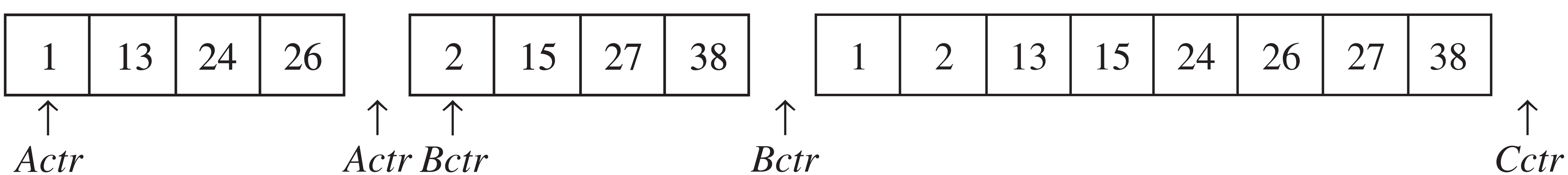
```
template <typename Comparable>
void percDown(vector<Comparable> & a, int i, int n) {
    int child;
    Comparable tmp;

    for(tmp = a[i]; leftChild(i) < n; i = child) {
        child = leftChild(i);

        if (child != n - 1 && a[child] < a[child + 1])
            ++child;

        if (tmp < a[child])
            a[i] = a[child];
        else
            break;
    }

    a[i] = tmp;
}
```

HEAPSORT

- Performance of `heapsort` is extremely consistent
  - works just below the worst-case bound
- `heapsort` always uses at least $N \lg N - O(N)$ comparisons
  - there are inputs that can achieve this bound
- Average number of comparisons is $2N \lg N - O(N \lg \lg N)$; this can be improved to $2N \lg N - O(N)$

MERGESORT

- is an algorithm that sorts an array of elements
- is an excellent instance of a recursive strategy
- runs in $O(N \lg N)$ worst-case running time
- has one <u>fundamental step</u>: merging two sorted arrays
  - done using auxiliary space $O(N)$



```
template <typename Comparable>
void mergeSort(vector<Comparable> & a) {
    vector<Comparable> tmpArray(a.size());
    mergeSort(a, tmpArray, 0, a.size() - 1);
}

template <typename Comparable>
void mergeSort(vector<Comparable> & a,
               vector<Comparable> & tmp,
               int left, int right) {
    if (left < right) {
        int center = (left + right) / 2;
        mergeSort(a, tmp, left, center);
        mergeSort(a, tmp, center + 1, right);
        merge(a, tmp, left, center + 1, right);
    }
}
```

```
template <typename Comp>
void merge(vector<Comp> & a, vector<Comp> & tmp,
           int leftPos, int rightPos, int rightEnd) {
    int leftEnd = rightPos - 1, tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;

    while (leftPos <= leftEnd && rightPos <= rightEnd)
        if (a[leftPos] <= a[rightPos])
            tmp[tmpPos++] = a[leftPos++];
        else tmp[tmpPos++] = a[rightPos++];

    while (leftPos <= leftEnd)
        tmp[tmpPos++] = a[leftPos++];

    while (rightPos <= rightEnd)
        tmp[tmpPos++] = a[rightPos++];

    for (int i = 0; i < numElements; ++i, --rightEnd)
        a[rightEnd] = tmp[rightEnd];
}
```

MERGESORT

- To analyze `mergesort` we consider the recurrence relation

$$T(1) = 1$$

$$T(N) = 2T(N/2) + N$$

- whose solution gives

$$T(N) = N \log N + N = O(N \log N)$$

- Problems with `mergesort`?
  - extra linear space complexity
  - copying to temporary array and back

MERGESORT

- Copying can be avoided by judiciously switching the roles of `a` and `tmp` at alternate levels of the recursion
- Running time of `mergesort` depends heavily on
  - relative costs of comparing elements
  - moving elements in the array `a` (and `tmp` array)
- These costs are language dependent

QUICKSORT

- is the fastest current generic sorting algorithm in practice
- has an average running time $O(N \lg N)$
- has $O(N^2)$ worst-case performance
- is simple to understand and prove correct
- uses a divide and conquer strategy for sorting
- can be combined with `heapsort` to make the latter faster

QUICKSORT

- Classic `quicksort` algorithm

1. If the number of elements in S is 0 or 1, then return
2. Pick any element v in S. This is called the pivot
3. Partition S – {v} (the remaining elements in S) into two disjoint groups:
   $S_1 = \{x \in S-\{v\} \mid x \leq v\}$, and $S_2 = \{x \in S-\{v\} \mid x \geq v\}$
4. Return {`quicksort`(S_1) followed by v followed by `quicksort`(S_2)}

QUICKSORT

- Classic `quicksort` algorithm

1. If the number of elements in S is 0 or 1, then return
2. Pick any element v in S. This is called the pivot

   How?
   Critical!

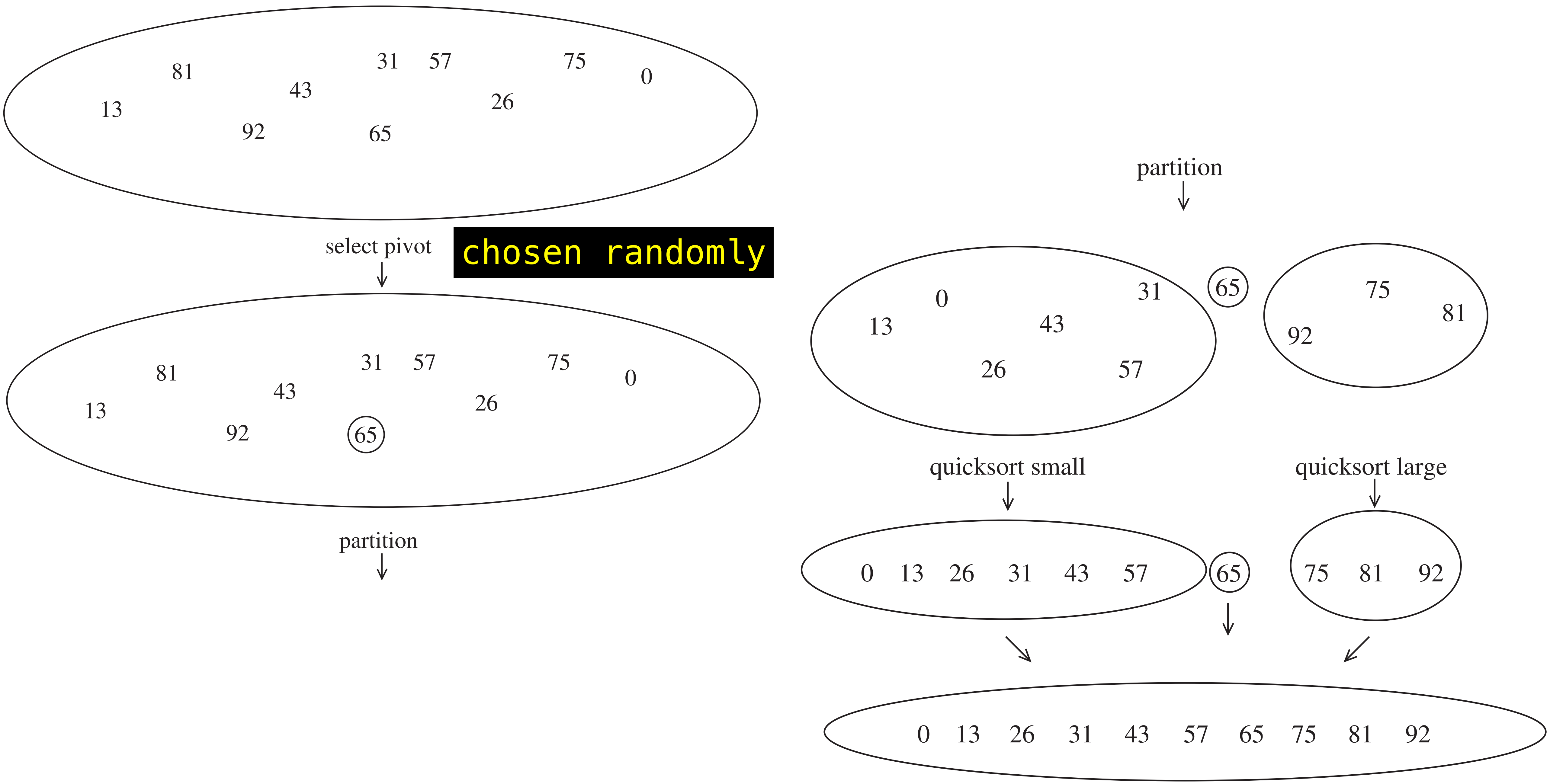
3. Partition S – {v} (the remaining elements in S) into two

   What to do with equal elements? Half and half

   $$S_1 = \{x \in S - \{v\} \mid x \leq v\}, \text{ and } S_2 = \{x \in S - \{v\} \mid x \geq v\}$$

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

QUICKSORT



81 31 57 75 0
13 43 26
92 65
select pivot
chosen randomly
81 31 57 75 0
13 43 26
92 65
partition
partition
0 31
13 43
26 57
65
75 81
92
quicksort small
quicksort large
0 13 26 31 43 57
65
75 81 92
0 13 26 31 43 57 65 75 81 92

QUICKSORT

- clear that `quicksort` works
- not clear it should be faster (than `merge` or `heap sort`)
- like `mergesort`, requires $O(N)$ additional work
- there's no guarantee that subarrays will be balanced
- however, `quicksort` is faster b/c work is done in-place
- `quicksort` is extremely sensitive to smallest deviations in the algorithm

QUICKSORT

- Picking the pivot
  - choose the first element A[0]
  - choose last element A[N−1]
  - choose it randomly between A[0], …, A[N−1]
  - choose the median of three elements

QUICKSORT

- Picking the pivot
  - choose the first element A[0]
  - choose last element A[N−1]

WRONG!

  - choose it randomly between A[0], …, A[N−1]
  - choose the median of three elements

QUICKSORT

- Picking the pivot
  - choose the first element A[0]
  - choose last element A[N−1] WRONG!
  - choose it randomly between A[0], …, A[N−1] OK!
  - choose the median of three elements

QUICKSORT

- Picking the pivot
  - choose the first element A[0]
  - choose last element A[N−1]

    WRONG!
  - choose it randomly between A[0], …, A[N−1] OK!
  - choose the median of three elements GREAT!

QUICKSORT

- Partitioning scheme

8	1	4	9	0	3	5	2	7	6
↑								↑	
i								j	

8	1	4	9	0	3	5	2	7	6
↑							↑		
i							j		

After First Swap

2	1	4	9	0	3	5	8	7	6
↑							↑		
i							j		

Before Second Swap

2	1	4	9	0	3	5	8	7	6
			↑			↑			
			i			j			

After Second Swap

2	1	4	5	0	3	9	8	7	6
			↑			↑			
			i			j			

Before Third Swap

2	1	4	5	0	3	9	8	7	6
					↑	↑			
					j	i			

After Swap with Pivot

2	1	4	5	0	3	6	8	7	9
						↑			↑
						i			pivot

```
template <typename Cmp>
const Cmp & median3(vector<Cmp> & a, int left, int right)
{
    int center = (left + right) / 2;

    if(a[center] < a[left])
        std::swap(a[left], a[center]);

    if(a[right] < a[left])
        std::swap(a[left], a[right]);

    if(a[right] < a[center])
        std::swap(a[center], a[right]);

    // Place pivot at position right - 1
    std::swap(a[center], a[right - 1]);

    return a[right - 1];
}
```

```cpp
template <typename Comp>
void quicksort(vector<Comp> & a, int left, int right)
{
    const Comparable & pivot = median3(a, left, right);

    // Begin partitioning
    int i = left, j = right - 1;
    while (true) {
        while (a[++i] < pivot) {}
        while (pivot < a[--j]) {}

        if(i < j) std::swap(a[i], a[j]);
        else break;
    }

    std::swap(a[i], a[right - 1]); // Restore pivot

    quicksort(a, left, i - 1);  // Sort small elements
    quicksort(a, i + 1, right); // Sort large elements
}
```

QUICKSORT

- In the analysis of quicksort we consider the recurrence

$$T(N) = T(i) + T(N - i - 1) + cN$$

- where $i = |S_1|$, number of elements in S_1
  - **Worst-case**: (i = 0)

$$T(N) = T(N-1) + cN, \quad N > 1$$

$$T(N) = T(1) + c\sum_{i=2}^{N} i = \Theta(N^2)$$

QUICKSORT

- In the analysis of quicksort we consider the recurrence

$$T(N) = T(i) + T(N - i - 1) + cN$$

- where $i = |S_1|$, number of elements in S_1
  - **Best-case**: ($i = N/2$)

$$T(N) = 2T(N/2) + cN$$

$$T(N) = cN \log N + N = \Theta(N \log N)$$

QUICKSORT

- In the analysis of quicksort we consider the recurrence

$$T(N) = T(i) + T(N - i - 1) + cN$$

- where i = $|S_1|$, number of elements in S_1
  - **Average-case**:

$$T(N) = \frac{2}{N}\left[\sum_{j=0}^{N-1} T(j)\right] + cN$$

$$\frac{T(N)}{N+1} = O(\log N)$$