CS101 Algorithms and Data Structures

Quick Sort

Textbook Ch 2.7

Outline

- Insertion sort
- Bubble sort
- Merge sort
- Quicksort

Quicksort

Merge sort splits the array into two sub-lists and sorts them

- It splits the larger problem into two sub-problems based on *location* in the array

Consider the following alternative:

- Chose an object in the array and partition the remaining objects into two groups relative to the chosen entry

Quicksort

For example, given

80	38	95	84	66	10	79	**44**	26	87	96	12	43	81	3

we can select the middle entry, 44, and sort the remaining entries into two groups, those less than 44 and those greater than 44:

38	10	26	12	43	3	**44**	80	95	84	66	79	87	96	81

Notice that 44 is now in the correct location if the list was sorted

- Proceed by recursively applying the algorithm to the first six and last eight entries

Run-time analysis

Like merge sort, we can either:

- Sort the sub-lists using quicksort
- If the size of the sub-list is sufficiently small, apply insertion sort

In the best case, the list will be split into two approximately equal sub-lists, and thus, the run time could be very similar to that of merge sort: $\Theta(n \ln(n))$

What happens if we don't get that lucky?

Worst-case scenario

Suppose we choose the middle element as our pivot and we try ordering a list:

80	38	95	84	66	10	79	**2**	26	87	96	12	43	81	3

Using 2, we partition into

2	80	38	95	84	66	10	79	26	87	96	12	43	81	3

We still have to sort a list of size $n - 1$

The run time is $\mathrm{T}(n) = \mathrm{T}(n - 1) + \Theta(n) = \Theta(n^2)$

– Thus, the run time performance drops from $n\ln(n)$ to n^2

Worst-case scenario

Our goal is to choose the median element in the list as our pivot:

80	38	95	84	**66**	10	79	2	26	87	96	12	43	81	3

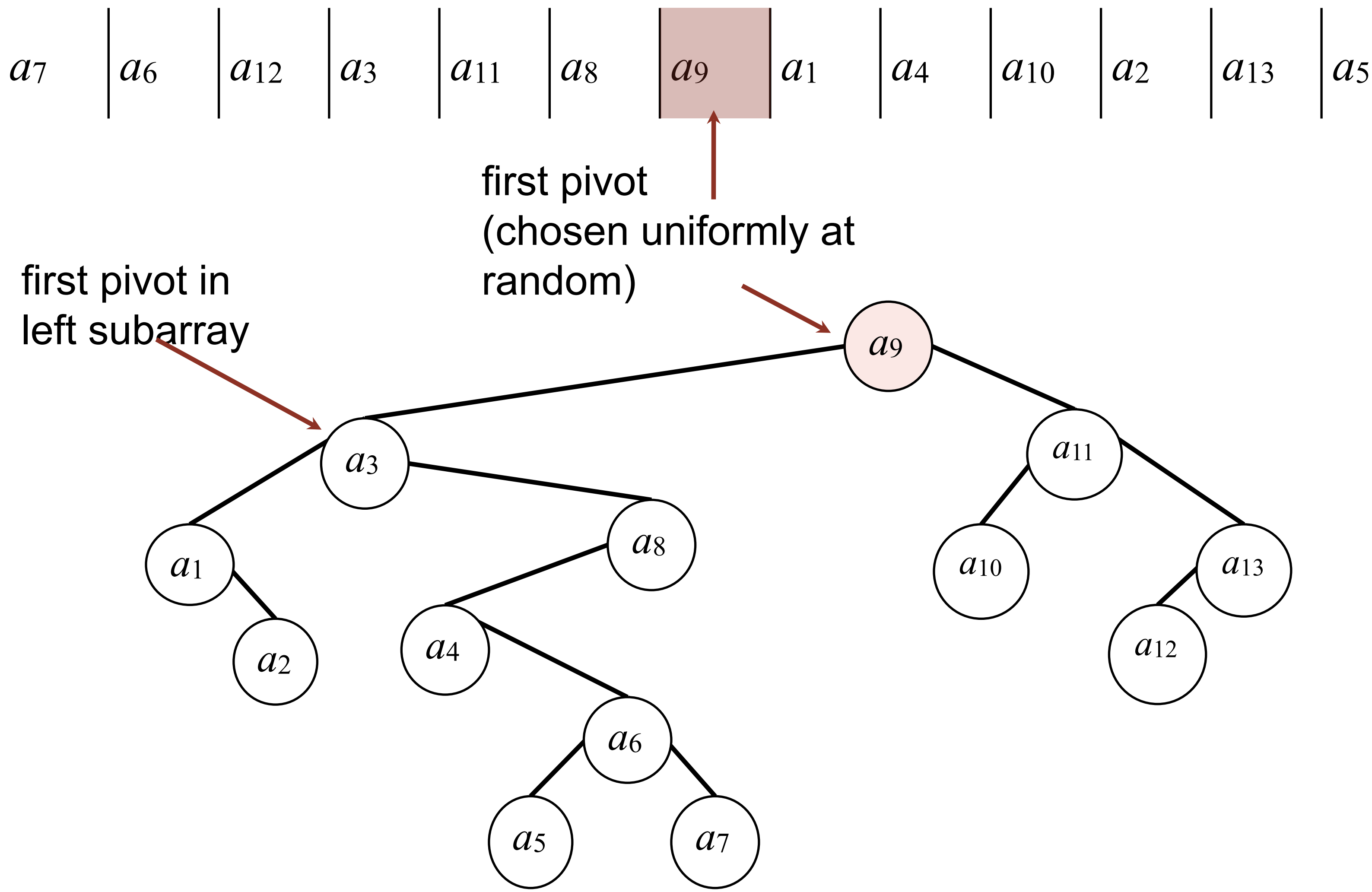
Using the median element 66, we can get two equal-size sub-lists

3	38	43	12	2	10	26	**66**	79	87	96	84	95	81	80

Unfortunately, median is difficult to find

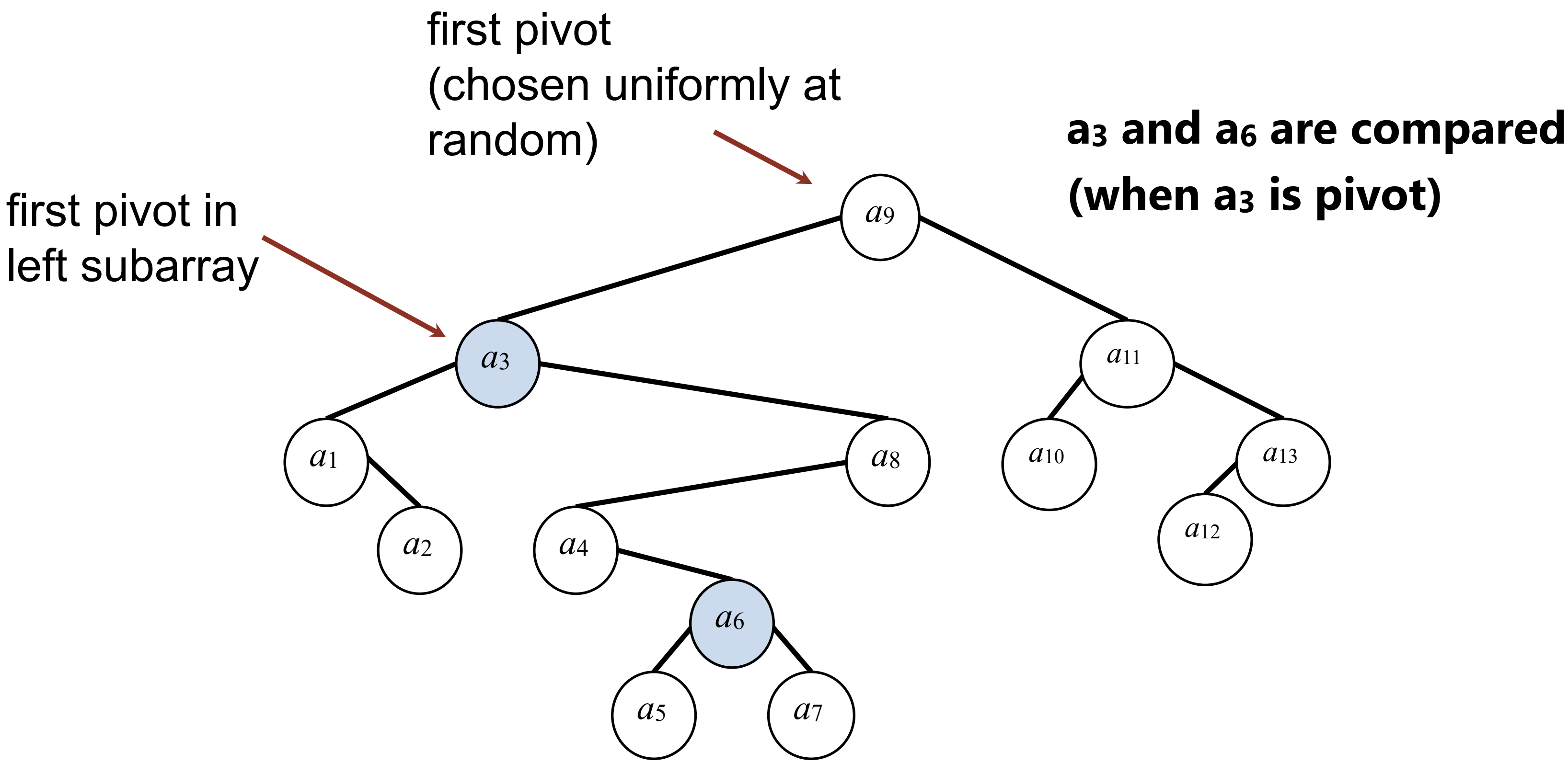
BST representation

The original array of elements A



BST representation

- Proposition. The expected number of compares to quicksort an array of n distinct elements $a_1 < a_2 < ... < a_n$ is $O(n \log n)$.
- Pf. Consider BST representation of pivot elements.
  - a_i and a_j are compared once if one is an ancestor of the other.



Question 1

- Given an array of $n \geq 8$ distinct elements $a_1 < a_2 < \dots < a_n$, what is the probability that a_7 and a_8 are compared during randomized quicksort?

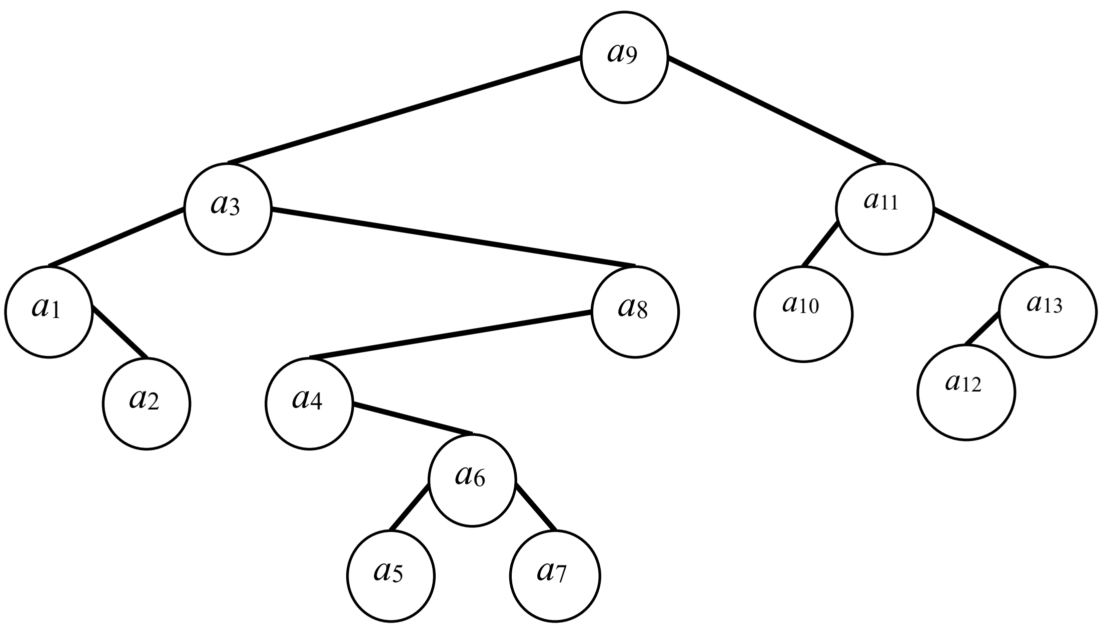
  A. 0

  B. $1 / n$

  C. $2 / n$

  D. 1

if two elements are adjacent,
one must be an ancestor of the other



Question 2

- Given an array of $n \geq 8$ distinct elements $a_1 < a_2 < \ldots < a_n$, what is the probability that a_1 and a_n are compared during randomized quicksort?

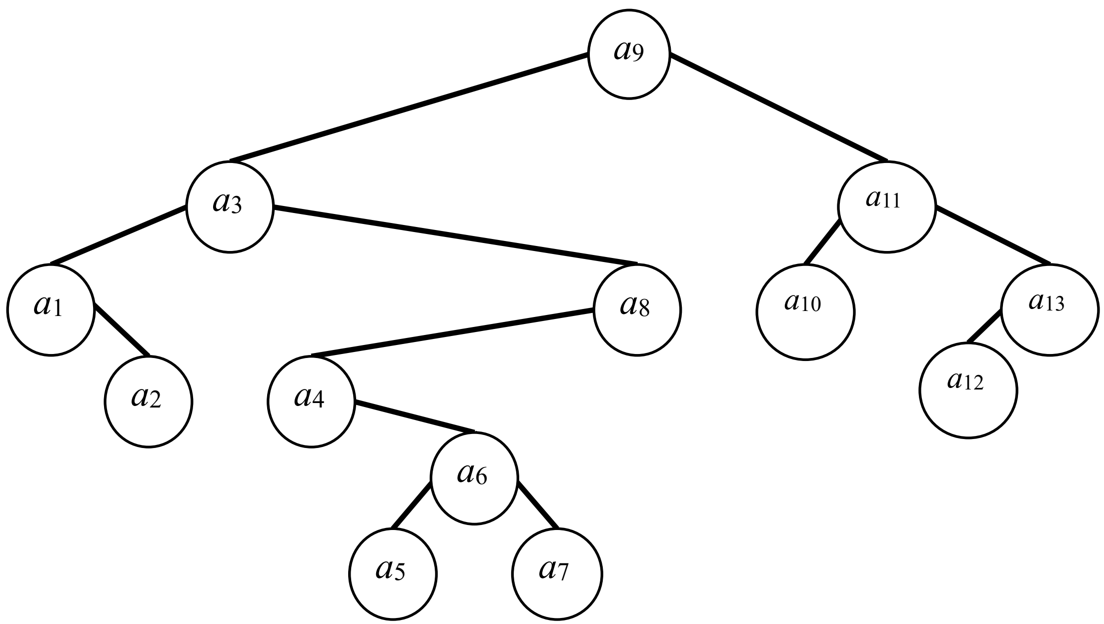
  A. 0

  B. $1 / n$

  C. $\boxed{2 / n}$

  D. 1

a_1 and a_n are compared
if first pivot is either a_1 or a_n

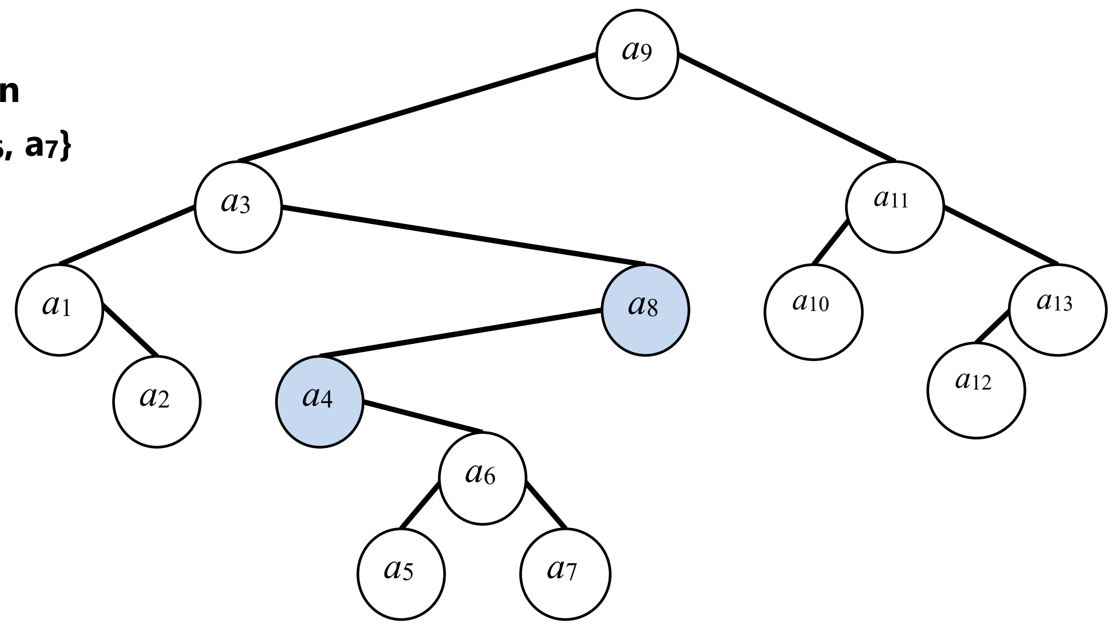


Analysis of running time

- Proposition. The expected number of compares to quicksort an array of n distinct elements $a_1 < a_2 < \ldots < a_n$ is $O(n \log n)$.

- Pf. Consider BST representation of pivot elements.
  - a_i and a_j are compared once if one is an ancestor of the other.
  - **Pr** [a_i and a_j are compared] $= 2 \; / \; (j - i + 1)$, where $i < j$.

Pr[a_2 and a_8 compared] = 2/7
compared if either a_2 or a_8 is chosen as pivot before any of { a_3, a_4, a_5, a_6, a_7}



Analysis of running time

- Proposition. The expected number of compares to quicksort an array of n distinct elements $a_1 < a_2 < \ldots < a_n$ is $O(n \log n)$.
- Pf. Consider BST representation of pivot elements.
  - a_i and a_j are compared once if one is an ancestor of the other.
  - **Pr** [a_i and a_j are compared] $= 2 \;/\; (j - i + 1)$, where $i < j$.
- Expected number of compares = $\displaystyle\sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \;=\; 2\sum_{i=1}^{n} \sum_{j=2}^{n-i+1} \frac{1}{j}$

all pairs i and j

$$\leq \; 2n \sum_{j=1}^{n} \frac{1}{j}$$

$$\leq \; 2n\,(\ln n + 1)$$

harmonic sum

Median-of-three

Consider another strategy:

– Choose the median of the first, middle, and last entries in the list

This will usually give a better approximation of the actual median

80	38	95	84	99	10	79	44	26	87	96	12	43	81	3

Median-of-three

Sorting the elements based on 44 results in two sub-lists, each of which must be sorted (again, using quicksort)

Select the 26 to partition the first sub-list:

38	10	26	12	43	3	44	80	95	84	99	79	87	96	81

Select 81 to partition the second sub-list:

38	10	26	12	43	3	44	80	95	84	99	79	87	96	81

Implementation

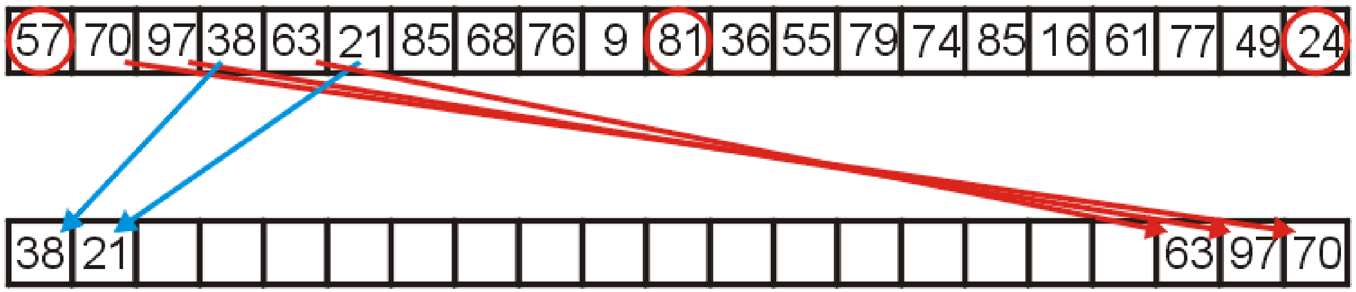
If we choose to allocate memory for an additional array, we can implement the partitioning by

- copying elements either to the front or the back of the additional array
- placing the pivot into the resulting hole

Implementation

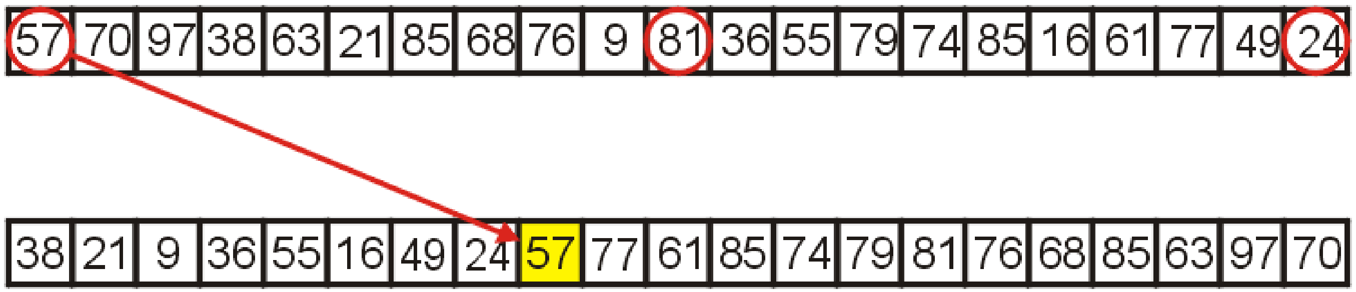
For example, consider the following:

- 57 is the median-of-three
- we go through the remaining elements, assigning them either to the front or the back of the second array



Implementation

Once we are finished, we copy the median-of-three, 57, into the resulting hole



Implementation

Can we implement quicksort in place?

Yes!

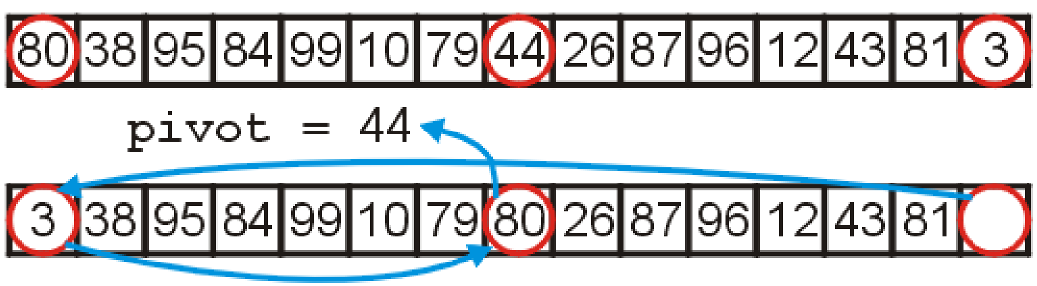
- Swap the pivot to the last slot of the list
- We repeatedly try to find two entries:
  - Staring from the front: an entry larger than the pivot
  - Starting from the back: an entry smaller than the pivot
- Such two entries are out of order, so we swap them
- Repeat until all the entries are in order
- Move the leftmost entry larger than the pivot into the last slot of the list and fill the hole with the pivot

Implementation

First, we have already examined the first, middle, and last entries and chosen the median of these to be the pivot

In addition, we can:

- move the smallest entry to the first entry
- move the largest entry to the middle entry



Implementation

The implementation is straight-forward

```
template <typename Type>
void quicksort( Type *array, int first, int last ) {
    if ( last - first <= N ) {
        insertion_sort( array, first, last );
    } else {
        Type pivot = find_pivot( array, first, last );
        int low  =     find_next( pivot, array, first + 1 );
        int high = find_previous( pivot, array,  last - 2 );

        while ( low < high ) {
            std::swap( array[low], array[high] );
            low  =     find_next( pivot, array,  low + 1 );
            high = find_previous( pivot, array, high - 1 );
        }

        array[last – 1] = array[low];
        array[low] = pivot;
        quicksort( array, first,  low );
        quicksort( array,  high, last );
    }
}
```

Quicksort example

Consider the following unsorted array of 25 entries

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

We will call insertion sort if the list being sorted of size $N = 6$ or less

Quicksort example

We call `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

`quicksort( array,  0, 25 )`

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	**57**	99	17	32	94	28	15	55	7	51	88	97	**62**

First, $25 - 0 > 6$, so find the midpoint and the pivot

```
midpoint = (0 + 25)/2; // == 12
```

`quicksort( array,  0, 25 )`

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	**62**	99	17	32	94	28	15	55	7	51	88	97	

First, $25 - 0 > 6$, so find the midpoint and the pivot

```
midpoint = (0 + 25)/2; // == 12
pivot = 57;
```

`quicksort( array, 0, 25 )`

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	62	99	17	32	94	28	15	55	7	51	88	97	

Starting from the front and back:

- Find the next element greater than the pivot
- The last element less than the pivot

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	77	49	35	61	48	73	23	95	3	89	37	62	99	17	32	94	28	15	55	7	51	88	97	

Searching forward and backward:

```
low = 1;
high = 21;
```

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	61	48	73	23	95	3	89	37	62	99	17	32	94	28	15	55	7	77	88	97	

Searching forward and backward:

```
low = 1;
high = 21;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	**61**	48	73	23	95	3	89	37	62	99	17	32	94	28	15	55	**7**	77	88	97	

Continue searching

```
low = 4;
high = 20;
```

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	73	23	95	3	89	37	62	99	17	32	94	28	15	55	61	77	88	97	

Continue searching

```
low = 4;
high = 20;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	**73**	23	95	3	89	37	62	99	17	32	94	28	15	**55**	61	77	88	97	

Continue searching

```
low = 6;
high = 19;
```

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	95	3	89	37	62	99	17	32	94	28	15	73	61	77	88	97	

Continue searching

```
low = 6;
high = 19;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	**95**	3	89	37	62	99	17	32	94	28	**15**	73	61	77	88	97	

Continue searching

```
low = 8;
high = 18;
```

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	89	37	62	99	17	32	94	28	95	73	61	77	88	97	

Continue searching

```
low = 8;
high = 18;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	89	37	62	99	17	32	94	28	95	73	61	77	88	97	

Continue searching

```
low = 10;
high = 17;
```

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	62	99	17	32	94	89	95	73	61	77	88	97	

Continue searching

```
low = 10;
high = 17;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

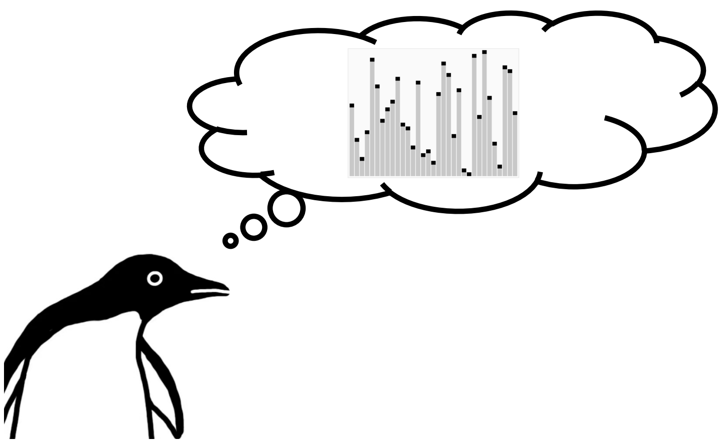
Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	62	99	17	32	94	89	95	73	61	77	88	97	

Continue searching

```
low = 12;
high = 15;
```



```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	99	17	62	94	89	95	73	61	77	88	97	

Continue searching

```
low = 12;
high = 15;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	99	17	62	94	89	95	73	61	77	88	97	

Continue searching

```
low = 13;
high = 14;
```

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	17	99	62	94	89	95	73	61	77	88	97	

Continue searching

```
low = 13;
high = 14;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	**17**	**99**	62	94	89	95	73	61	77	88	97	

Continue searching

```
low = 14;
high = 13;
```

Now, `low > high`, so we stop

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	17	**57**	62	94	89	95	73	61	77	88	97	99

Continue searching

```
low = 14;
high = 13;
```

Now, `low > high`, so we stop

`pivot = 57;`

`quicksort( array, 0, 25 )`

Quicksort example

We are calling `quicksort( array, 0, 25 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	17	**57**	62	94	89	95	73	61	77	88	97	99

We now begin calling quicksort recursively on the first half

```
quicksort( array, 0, 14 );
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32	17	57	62	94	89	95	73	61	77	88	97	99

First, $14 - 0 > 6$, so find the midpoint and the pivot

```
midpoint = (0 + 14)/2; // == 7
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	**23**	15	3	28	37	32	**17**	57	62	94	89	95	73	61	77	88	97	99

First, $14 - 0 > 6$, so find the midpoint and the pivot

```
midpoint = (0 + 14)/2; // == 7
pivot = 17
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32		57	62	94	89	95	73	61	77	88	97	99

First, $14 - 0 > 6$, so find the midpoint and the pivot

```
midpoint = (0 + 14)/2; // == 7
```

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Starting from the front and back:

- Find the next element greater than the pivot
- The last element less than the pivot

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	51	49	35	7	48	55	23	15	3	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 1;
high = 9;
```

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	49	35	7	48	55	23	15	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 1;
high = 9;
```

Swap them

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	49	35	7	48	55	23	15	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 2;
high = 8;
```

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	15	35	7	48	55	23	49	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 2;
high = 8;
```

Swap them

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	15	35	7	48	55	23	49	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 3;
high = 4;
```

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	15	7	35	48	55	23	49	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 3;
high = 4;
```

Swap them

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	15	7	35	48	55	23	49	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 4;
high = 3;
```

Now, `low > high`, so we stop

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	15	7	**17**	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

We continue calling quicksort recursively

```
quicksort( array, 0, 4 );
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 4 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	15	7	17	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

Now, $4 - 0 \le 6$, so find we call insertion sort

```
quicksort( array,  0,  4 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from `0` to `3`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	3	15	7	17	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

```
insertion_sort( array, 0, 4 )
quicksort( array,  0,  4 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from `0` to `3`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

– This function call completes and so we exit

```
insertion_sort( array, 0, 4 )
quicksort( array,  0,  4 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

This call to `quicksort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

```
quicksort( array,  0,  4 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 0, 14 )`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	**17**	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

We continue calling quicksort recursively on the second half

```
quicksort( array, 0,  4 );
quicksort( array, 5, 14 );
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

This call to `quicksort` is now also finished, so it, too, exits

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

`quicksort( array,  0, 25 )`

Quicksort example

We have now used quicksort to sort this array of 25 entries

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

Memory Requirements

The additional memory?

- Function call stack
  - Each recursive function call places its local variables, parameters, *etc*., on a stack
- Average case: the depth of the recursion is $\Theta(\ln(n))$
- Worst case: the depth of the recursion is $\Theta(n)$

Run-time Summary

To summarize the two $\Theta(n \ln(n))$ algorithms

	Average Run Time	Worst-case Run Time	Average Memory	Worst-case Memory
Merge Sort	$\Theta(n \ln(n))$		$\Theta(n)$	
Quicksort	$\Theta(n \ln(n))$	$\Theta(n^2)$	$\Theta(\ln(n))$	$\Theta(n)$

Further modifications

Our implementation is by no means optimal:

An excellent paper on quicksort was written by Jon L. Bentley and M. Douglas McIlroy:

Engineering a Sort Function

found in Software—Practice and Experience, Vol. 23(11), Nov 1993

Matching Nuts and Bolts

- Problem. A disorganized carpenter has a mixed pile of n nuts and n bolts.
  - The goal is to find the corresponding pairs of nuts and bolts.
  - Each nut fits exactly one bolt and each bolt fits exactly one nut.
  - By fitting a nut and a bolt together, the carpenter can see which one is bigger (but cannot directly compare either two nuts or two bolts).

- Brute-force solution. Compare each bolt to each nut—$\Theta(n^2)$ compares.
  - Challenge. Design an algorithm that makes $O(n \log n)$ compares.
  - Hint : use quick sort

Matching Nuts and Bolts

- Divide.
  - Pick bolt p uniformly at random; compare bolt p against all nuts; divide nuts smaller than p from those that are larger than p.
  - Let p' be the nut that matches bolt p. Compare p' against all bolts; divide bolts smaller than p' from those that are larger than p'.
- Conquer. Recursively solve two independent subproblems.

- Analysis. Almost identical to analysis of randomized quicksort.
  (but $2n$ compares to partition pile of n nuts and n bolts)

Summary

This topic covered quicksort

- On average faster than heap sort or merge sort
- Uses a pivot to partition the objects
- Using the median of three pivots is a reasonably means of finding the pivot
- Average run time of $\Theta(n \ln(n))$ and $\Theta(\ln(n))$ memory
- Worst case run time of $\Theta(n^2)$ and $\Theta(n)$ memory

Master Theorem

- Goal. Recipe for solving common divide-and-conquer recurrences:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

  with $T(0) = 0$ and $T(1) = \Theta(1)$.

- Terms.
  - $a \geq 1$ is the number of subproblems.
  - $b \geq 2$ is the factor by which the subproblem size decreases.
  - $f(n) \geq 0$ is the work to divide and combine subproblems.
- Recursion tree. [assuming n is a power of b]
  - a = branching factor.
  - a^k = number of subproblems at level k.
  - $1 + \log_b n$ levels.
  - n / b^k = size of subproblem at level k.

Run-time Analysis of Merge Sort

The time required to sort an array of size $n > 1$ is:

- the time required to sort the first half,
- the time required to sort the second half, and
- the time required to merge the two lists

That is: $$\mathrm{T}(n) = \begin{cases} \mathbf{\Theta}(1) & n = 1 \\ 2\,\mathrm{T}\left(\frac{n}{2}\right) + \mathbf{\Theta}(n) & n > 1 \end{cases}$$

Solution: $\mathrm{T}(n) = \Theta(n \ln(n))$

Recursion Tree for Merge Sort

Level		**Problem #**	**Problem size**	**Work**
0		1	n	n
1		2	(n/2)	2(n/2)
2		4	(n/4)	4(n/4)
. . .	. . .	. . .	. . .	. . .
$k=\log_2 n$	. . . .	$2^k = n$	1	$2^k(n / 2^k)$

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Total work $= \sum_1^k 2^k(n / 2^k) = \sum_1^k n = n\log n$

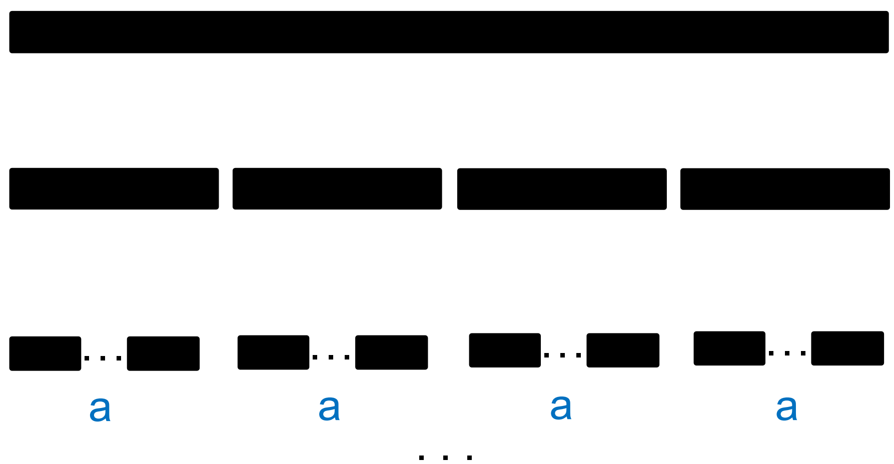
Recursion Tree for Binary Search

Level		**Problem #**	**Problem size**	**Work**
0		1	n	C
1		1	(n/2)	C
2		1	(n/4)	C
. . .	. . .	. . .	. . .	. . .
k=$\log_2 n$	. . . .	1	1	C

$$T(n) = T\left(floor(\frac{n}{2})\right) + \Theta(C)$$

Total work =$\sum_1^k C$= $\sum_1^{\log_2 n}$ C $= logn$

Master Theorem



Level	Problem #	Problem size	Work
0	1	n	n^d
1	a	(n/b)	$a(n/b)^d$
2	a^2	(n/b^2)	$a^2(n/b^2)^d$
. . .	. . .	. . .	. . .
$k=\log_b n$	$a^k=a^{\log_b n}$	1	$a^k(n/b^k)^d$

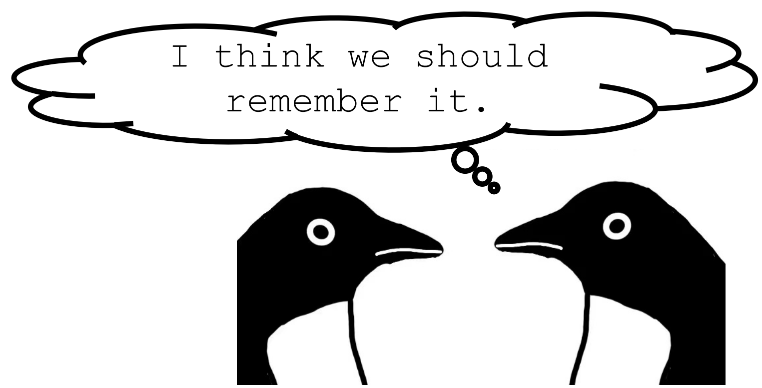
$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^d)$$

Total work $=\sum_1^k a^k(n/b^k)^d = \sum_1^k n^d(a/b^d)^k = n^d \sum_1^k (a/b^d)^k$

Master Theorem

- If $T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^d)$ for constants $a > 0, b > 1, d \geq 0$, then:

$$T(n) = \begin{cases} O(n^d) & if\ d > log_b a \\ O(n^d logn) & if\ d = log_b a \\ O(n^{log_b a}) & if\ d < log_b a \end{cases}$$



Master theorem does not apply

- Gaps in master theorem.
  - Number of subproblems is not a constant.

$$T(n) = nT(n/2) + n^2$$

  - Number of subproblems is less than 1.

$$T(n) = \frac{1}{2}T(n/2) + n^2$$

  - No polynomial separation between $f(n)$ and $n^{\log_b a}$.

$$T(n) = 2T(n/2) + \boxed{n \log n}$$

  - $f(n)$ is not positive.

$$T(n) = 2T(n/2) - \boxed{n^2}$$

  - Regularity condition does not hold.

$$T(n) = T(n/2) + \boxed{n(2 - \cos n)}$$

Question 1

- **Consider the following recurrence. Which case of the master theorem?**

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 3T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1 \end{cases}$$

  – Case 1: $T(n) = \Theta(n^{\log_2 3}) = O(n^{1.585})$.

  – Case 2: $T(n) = \Theta(n \log n)$.

  – Case 3: $T(n) = \Theta(n)$.

  – Master theorem not applicable.