Divide and Conquer

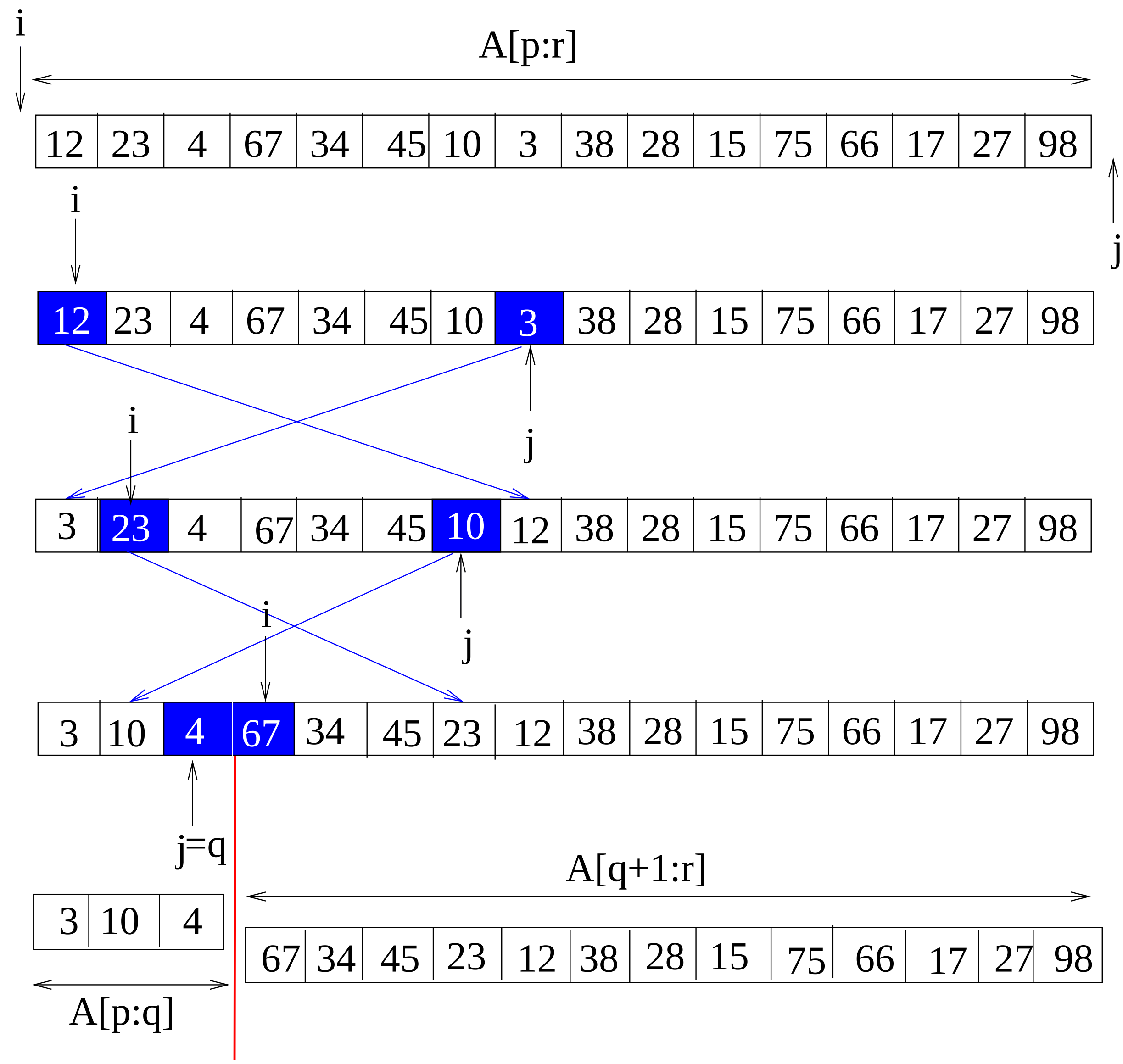
- DAC – primitive paradigm
  - Divide: partitioning the problem into subproblems of equal size.
  - Conquer: solutions of subproblems are found.
  - Combine: combine solutins of subproblems into solution of whole

Quick Sort

```
quickSort(A, p, r) {
    if (p< r) {
        q = partition(A, p, r);
        quickSort(A, p, q);
        quickSort(A, q+1, r);
    }
}

partition(A, p, r) {
    x = A[p]; i=p-1; j=r+1;
    while (1) {
        repeat j-- until A[j] ≥ x;
        repeat i++ until A[j] ≤ x;
        if (i < j) exchange(A[i], A[j]);
        else return j;
    }
}
```

Partitioning



Array Bounds

- Initially i and j point to subarray indices not within the bounds.
- But as i increments to right and j decrements to left they come within bounds of the subarray.
- If $A[p]$ happens to be the largest value then i will get continually incremented until it becomes equal to r while j will be stuck at $r-1$.
- So the value of j returned is $q = r - 1$.
- If $A[p]$ happens to be the smallest value then j will get continually decremented until it becomes equal to p while i will be stuck at $p + 1$.
- So the value of j returned is $q = p$.
- i and j never index the subarray $A[p : r]$ out of bounds.

Worst Case Performance

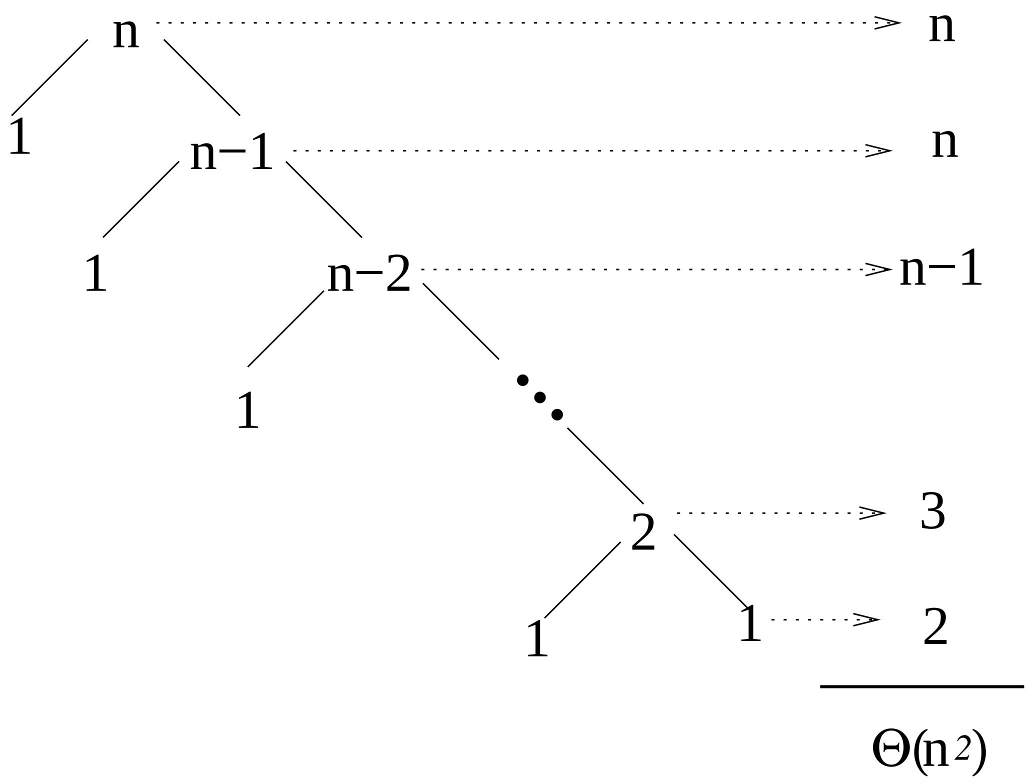
- Worst case behaviour exhibited when array is always divided with one subarray of unit size and other of size $n-1$.
- The worst case performance is reproduceable by the sorted input.
- The recurrence relation for running time

$$T(n) = T(n-1) + \Theta(n) = \sum_{k=1}^{n} \Theta(k)$$

  Applying iteration.
- So

$$\begin{aligned} T(n) &= \Theta\left(\textstyle\sum_{k=1}^{n} k\right) \\ &= \Theta(n^2) \end{aligned}$$

Worst Case Performance



Best Case Performance

- Best case performance is exhibited when partition produces two equal size subarrays.
- In this case the partitioning element is the median
- The recurrence for this case is

$$T(n) = 2T(n/2) + \Theta(n)$$

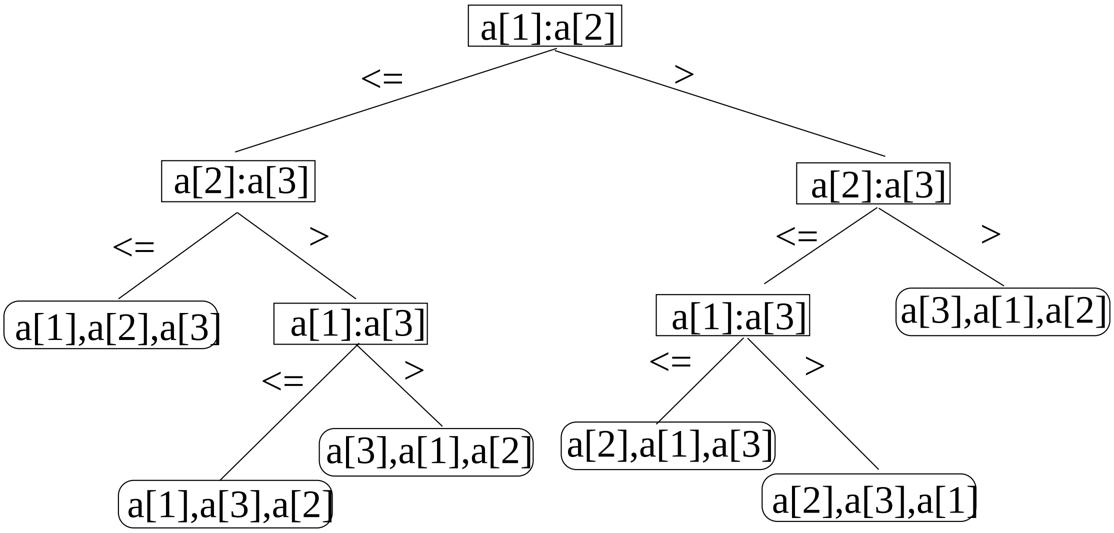
- Applying master theorem with $a, b = 2$, $n^{\log_b a} = n$, thus, $f(n) = O(n^{\log_b a})$, so the solution is

$$T(n) = \Theta(n \log n)$$

Balanced Partitioning

- Balance partitioning requires median element.
- Median element should be obtained in time not exceeding $\Theta(n)$.
- How is it possible?
- There are optimal sorting algorithms for fixed sized inputs.
- Sorting for 3 and 5-element inputs are shown.

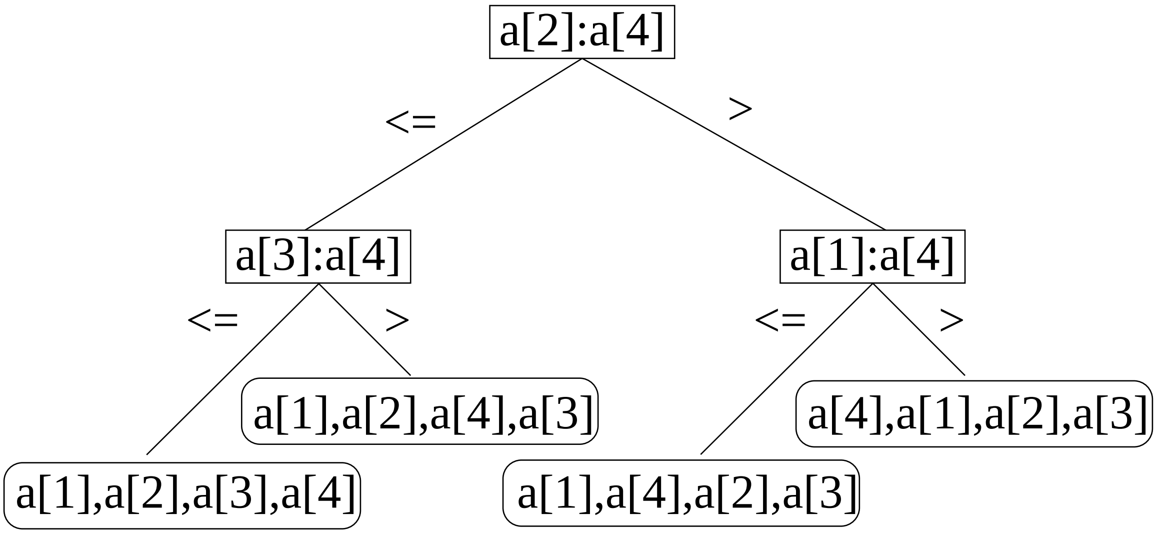
Three Element Sort



- Only three comparions are necessary to sort 3 elements

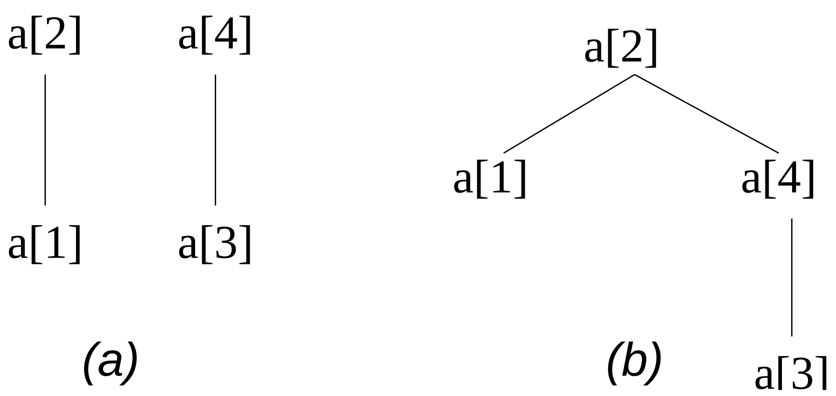
Four Elements Sort

- Sort the any three elements using 3-sorter.
- Using 2 more comparisons the remaining element can be inserted into the sorted list of three elements.
- So 4 elements can be sorted using 5 comparisons as shown.



Five Elements Sort

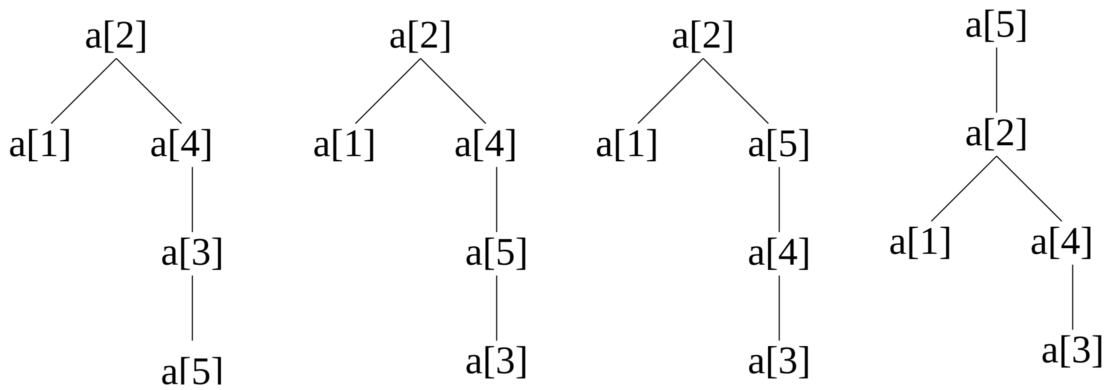
- Compare two pair of elements first – ordering is shown in figure (a) below,
- Compare the larger of the pairs – the ordering will be as in figure (b).



(a) (b)

Five Elements Sort

- Insertion of $a[5]$ within three elements $a[2]$, $a[3]$ and $a[4]$ requires 2 comparisons.
- Possible configurations are as shown in figure.



Five Elements Sort

- So we used up 3 comparison initially and 2 more for inserting $a[5]$.
- We already know that $a[2] > a[1]$.
- Thus the position of $a[1]$ needs to be determined among 3 elements $a[4], a[3], a[5]$ in one of the 4 configurations.
- This requires 2 more comparisons.
- So total number of comparisons for sorting 5 elenents is 7.

Median Finding

- Given n items, find the item which is smaller than half the items and bigger than half the items.
- If we sort the data, then $n/2$th item is the item we want.
- But sorting requires $O(n \log n)$ time.
- We want to solve it in linear time.
- The problem can be reframed more generally as follows:

  Given $k \leq n$ find kth item in a given set consisting of n items.

Simple Randomized Algorithm

- Choose a pivot element randomly
- Partition the set of given items into two sets
  - $S_{\leq}$ – set of items less than pivot
  - $S_{\geq}$ – set of items greater than pivot
- If $k = k_{\leq} + 1$ then return pivot element
- If $k \leq k_{\leq}$ then resursively find kth element in $S_{\leq}$.
- Else resursively find $k - k_{\leq}$th element in $S_{\geq}$.

Randomized Algorithm

- Running time depends on how we choose pivot p.
- Suppose we have 80% chance that p is in the middle of 80% of the sorted list, i.e. 80% of times p does not belong to 10% lower and 10% upper part of sorted list.
- With p in the middle, we can guarantee elimination of at least 10% of the list in the worst case. But if p is in the last or the first 10% almost no elmination.
- Thus

$$\begin{aligned} T(n) &\leq .2 \cdot T(n) + .8 \cdot T(.9n) + cn \\ .8 \cdot T(n) &\leq .8 \cdot T(.9n) + cn \\ T(n) &\leq T(.9n) + O(n) \end{aligned}$$

Randomized Median

- Thus we have a randomized linear time algorithm for median finding.
- The probability that every time a bad pivot being picked is $(\frac{1}{5})^n$. In this case p would belong to the first or the last 10% of the sorted list.
- Let us try to derandomize the choice of pivot to obtain a deterministic algorithm which guarantees linear time.

