Let P be a computational problem and $I \in D$ be an input to P with $|I| = n$.

General Format of DAC Algoriothm:

Algorithm: DAC(P,I)
    if |I| is small enough to be solved
        then solve (P,I) directly
        else ***divide*** (P,I) into $(P,I_1), (P,I_2), \ldots, (P,I_a)$);
            combine(DAC(P,I_1), …, DAC(P,I_a))
    endif;
endDAC

Characteristics of Recursive DAC Algorithms:

1. There exist one or more base cases for which solution(s) can be computed directly without any further recursion.
2. The base case(s) must be reachable to guarantee termination of the recursive algorithm.
3. The algorithm computes a solution to a general problem by combining the solutions of one or more identical, but smaller, subproblems.
4. The algorithm invokes itself (recursive call) to compute a solution to the subproblems.

Remarks:

- Performance and success of a DAC algorithm depend on the divide and combine functions.
- Complexity function of a DAC algorithm can usually be modeled using recurrence relation.

Performance Analysis of DAC Algorithm:

Let T(n) be the cost required in executing the above DAC algorithm with input I, $|I| = n$.

$T(n_0)$ = constant,
$T(n) = T(|I_1|) + T(|I_2|) + \ldots + T(|I_a|) + f(n), n > n_0$.

The function f(n) is the ***driving function*** (cost for divide and combine functions) of the DAC algorithm.

Simplification:

Assume that $|I_1| = |I_2| = \ldots = |I_a| = n/c$, c= constant, we have

$$T(n) = aT(n/c) + f(n).$$

Further Simplification:

Assume that f(n) is bounded by a polynomial function; $f(n) = O(n^k)$.

The Master Theorem of DAC:

Given $T(n_0) = d$,

$$T(n) = aT(\frac{n}{c}) + O(n^k),\ n > n_0, \text{ where } a, c, d \text{ are constants with } a > 0, c \ge 1, k \ge 0.$$

Then,
$$\begin{aligned} T(n) &= O(n^k), && \text{if } a < c^k, \\ &= O(n^k \log_c n), && \text{if } a = c^k, \\ &= O(n^{\log_c a}), && \text{if } a > c^k. \end{aligned}$$

Examples:

1. *Max-finding algorithm:*

Input: An array anArray[first..last] of integers.

Output: The maximum integer in the array.

Algorithm:

```
int maxInteger(const int anArray[], int first, int last)
{
    int mid;
    int max1, max2;

    if (first == last)                 //   base case
        return anArray[first];
    else
    {
        mid = (first + last)/2;
        max1 = maxInteger(anArray, first, mid);
        max2 = maxInteger(anArray, mid+1, last);

        if (max1 > max2)
            return max1;
        else return max2;
    }
}   //  end maxInteger
```

Complexity Analysis:

Basic operation: Comparison among elements in array.

Let $T(n)$ be the #comparisons required in finding the maximum integer in an array with n integers.

$T(1) = 0$,

$T(n) = 2T(n/2) + 1$, if $n = 2^q > 1$.

Using Master Theorem, $a = 2$, $c = 2$, $k = 0$, $a > c^k$. Hence,

$T(n) = O(n^{\log_c a}) = O(n)$.

More precisely, we can solved for T(n) using ***the method repeated substitutions***.

$$
\begin{aligned}
& \mathrm{T}(n) \\
& = 2\,\mathrm{T}(\frac{n}{2}) + 1 \\
& = 2[2\,\mathrm{T}(\frac{n}{2^2}) + 1] + 1 \\
& = 2^2\,\mathrm{T}(\frac{n}{2^2}) + 2^1 + 2^0 \\
& = \ldots \\
& = 2^q\,\mathrm{T}(\frac{n}{2^q}) + \sum_{i=0}^{q-1} 2^i \\
& = 2^q - 1 \\
& = n - 1\,.
\end{aligned}
$$

Another example on the method of repeated substitutions:

$$
\begin{aligned}
& \mathrm{T}(1) = 0, \\
& \mathrm{T}(n) = 3T(\frac{n}{3}) + \frac{5}{3}n - 2, n = 3^k > 1.
\end{aligned}
$$

$$
\begin{aligned}
& \mathrm{T}(n) = 3\mathrm{T}(\frac{n}{3}) + \frac{5}{3}n - 2 \\
& = 3[3\mathrm{T}(\frac{n}{3^2}) + \frac{5}{3}\frac{n}{3} - 2] + \frac{5}{3}n - 2 \\
& = 3^2\,\mathrm{T}(\frac{n}{3^2}) + 2(\frac{5}{3}n) - 3^1 * 2 - 3^0 * 2 \\
& = 3^2[3\mathrm{T}(\frac{n}{3^3}) + \frac{5}{3}\frac{n}{3^2} - 2] + 2(\frac{5}{3}n) - 3^1 * 2 - 3^0 * 2 \\
& = 3^3\,\mathrm{T}(\frac{n}{3^3}) + 3(\frac{5}{3}n) - 3^2 * 2 - 3^1 * 2 - 3^0 * 2 \\
& = \ldots \\
& = 3^k\,\mathrm{T}(\frac{n}{3^k}) + k(\frac{5}{3}n) - 2(3^{k-1} + 3^{k-2} + \ldots + 3^0) \\
& = \frac{5}{3}n\log_3 n - n + 1
\end{aligned}
$$

2. *MinMax-finding algorithm:*

Input: An array a[first..last] of integers.

Output: The min and the max integers in the array.

Algorithm:

```
void minMax(const int a[],int first,int last,int&min, int&max)
{
    if (first > last)              //    base case
        return;
    if (first ==last)
        min = max = a[first];
    if (last ==first+1)
    {   if a[first] > a[last]
            {       max = a[first];
                    min = a[last];  }
        else    { min = a[first];
                    max = a[last]; }
    }
    int mid = (first+last)/2;
    int min1, max1, min2, max2;
    minMax(a,first,mid,min1,max1);
    minMax(a,mid+1,last,min2,max2);
    if (min1 < min2)
            min = min1;
    else   min = min2;
    if (max1 > max2)
            max = max1;
    else   max = max2;
}   //  end minMax
```

Complexity Analysis:
Basic operation: Comparison among elements in array.

Let T(n) be the #comparisons required in finding the min and the max integers in an array with n integers.

$T(2) = 1,$
$T(n) = 2T(n/2) + 2, \text{ if } n = 2^q > 1.$

Using Master Theorem, a = 2, c = 2, k = 0, $a > c^k$. Hence,
$T(n) = O(n^{\log_c a}) = O(n).$

Using the method repeated substitutions, we have

$$
\begin{aligned}
& T(n) \\
&= 2T(\frac{n}{2}) + 2 \\
&= 2[2T(\frac{n}{2^2}) + 2] + 2 \\
&= 2^2 T(\frac{n}{2^2}) + 2^2 + 2^1 \\
&= \ldots \\
&= 2^{q-1} T(\frac{n}{2^{q-1}}) + \sum_{i=1}^{q-1} 2^i \\
&= 2^{q-1} + 2^q - 1 - 1 \\
&= \frac{3n}{2} - 2.
\end{aligned}
$$

Q: How “good” is this DAC algorithm?

3. *2Max-finding algorithm:*

Input: An array a[first..last] of integers.
Output: The two largest integers in the array.

Algorithm 1: Apply a max-finding algorithm for finding the maximum of a[]. Remove max. Apply the same max-finding algorithm for finding the maximum of a[]-{max}.

Algorithm 2: Compare a[1] with a[2] to determine the 2 maximum elements.

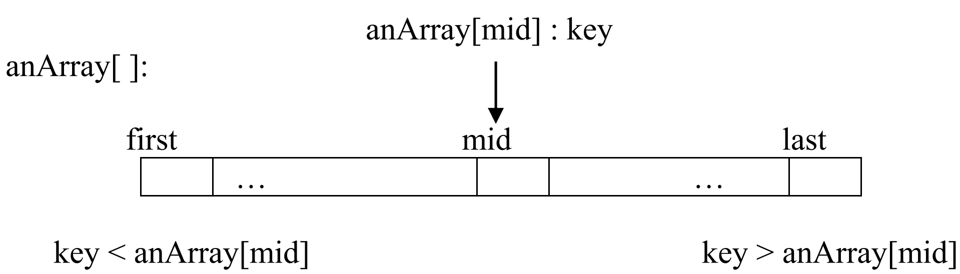
```
for i = 2 to n do
    compare a[i] with the 2 maximum elements found;
    update the 2 maximum elements if necessary;
endfor;
```

Algorithm 3: DAC.
HW. Design and analyze the above algorithms. Compute T(n) in closed-form.

4. *Searching an Ordered Array:*

Binary search:
Given a sorted array anArray[first..last] and a key.
Return array index k, first ≤ k ≤ last, such that anArray[k] = key, if exists; otherwise, return -1.

Prototype:
int bsearch(const int anArray[], int first, int last, int key);



Recursive Algorithm:

```
mid = (first + last)/2;
if key = anArray[mid]
    return mid;
else if key < anArray[mid]
    bsearch(anArray, first, mid-1, key);
else    // if x > anArray[mid]
    bsearch(anArray, mid+1, last, key);
```

Recursive Binary Search Algorithm:

```
int bsearch(const int anArray[], int first, int last, int key)
{
    int index;
    if (first > last)                          // base case; key not found
     index = -1;
   else
   {
      int mid = (first + last)/2;              // compute mid for dividing
      if (key == anArray[mid])                 //   key found
      {
         index = mid;
      }
      else if (key < anArray [mid])            //   search left sub-array
                index = bsearch(anArray, first, mid – 1, key);
            else                               //  search right sub-array
                index = bsearch(anArray, mid + 1, last, key);
   }
    return index;
}   //   end bsearch
```

Complexity Analysis:

Basic operation: Comparison between key and array element.

Let T(n) be the #comparisons required in searching for x in an array with n elements.

$T(1) = 1$,

$T(n) = T(n/2) + 1$, if $n = 2^q > 1$.

Using Master Theorem, $a = 1$, $c = 2$, $k = 0$, $a = c^k$. Hence,

$T(n) = O(n^k \log_c n) = O(\log_2 n)$.

A More Precise Recurrence for Binary Search:

$$T(1) = 1,$$
$$T(n) = T\left\lfloor \frac{n}{2} \right\rfloor + 1, n > 1.$$

Observe that
$T(n) \le T(n/2) + 1$, if $n > 1$.

Using the method repeated substitutions, we have

$$
\begin{aligned}
& T(n) \\
& \le T(\frac{n}{2}) + 1 \\
& \le [T(\frac{n}{2^2}) + 1] + 1 \\
& = T(\frac{n}{2^2}) + 2 \\
& = \ldots \\
& \le T(\frac{n}{2^q}) + q \\
& = \log_2 n + 1 \\
& = O(\log_2 n).
\end{aligned}
$$

HW. Design & analyze a 3-ary and 4-ary search algorithm.
Can you generalize this algorithm to k-ary search, $k \ge 2$?
What is your conclusion?

5. *DAC Sorting Algorithms:*

Basic Idea:

Given a linear data structure A with n records.
Divide A into substructures S1 and S2.
Sort S1 and S2 recursively.
Combine S1 and S2 to form a sorted structure.

Two cases:

1. If no restriction on keys in S1 and S2, then we must merge the two sorted lists S1 and S2 together. This is **Merge Sort**.

2. If (keys in S1 ≤ keys in S2), then concate(S1,S2) is already sorted. This is **Quick Sort**.

(i) *Merge Sort:*

```
mergeSort(A,first,last)
{
    if (first < last)
    {
        mid = (first + last)/2;
        mergeSort(A,first,mid);
        mergeSort(A,mid+1,last);
        merge(A,first,mid,last)
    }
}   //  end mergeSort
```

Q: How do we merge the two sorted lists together?
visualize the two sorted lists to be stored in two separate stacks S_1 and S_2;
compare the top elements of the two stacks and pop the smaller one to another list structure L until one of the stacks is empty;
pop, until empty, the remaining non-empty stack to L;

Given two sorted lists with sizes n_1 and n_2. Counting the #comparisons, we have

Merging:

$T(n_1,n_2) = n_1 + n_2 - 1.$

Complexity of Mergesort:

$T(1) = 0,$
$T(n) = 2T(n/2) + (n-1), n > 1.$

Using Master Theorem, $a = 2$, $c = 2$, $k = 1$, we have

$T(n) = O(n\lg n).$

HW. Design and analyze an iterative mergesort algorithm.

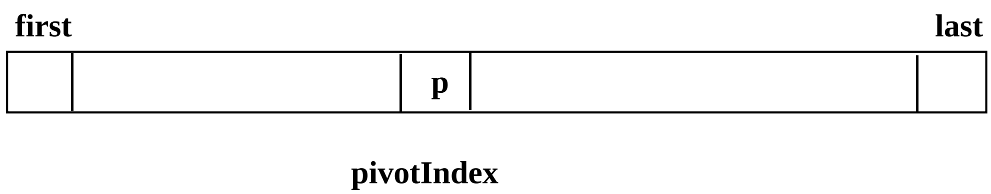
HW. Design and analyze a 3-way, and 4-way, mergesort algorithms by dividing the original list structure into 3, and 4, roughly equal-sized sub-structures. Can you generalize this algorithm to k-way mergesort, $k \geq 2$? What is your conclusion?

(ii) *Quick Sort:*

Base case: If |A| =1, A is already sorted.

General case: If |A| > 1, divide A into two sub-arrays A1 and A2 using some pivot p in A such that A1 contains only those keys that are < p and A2 contains only those keys that are ≥ p.
Sort A1 and A2 recursively.

Given A[first..last]:



A1 = A[first..pivotIndex-1], every key in A1 < p
A2 = A[pivotIndex+1,last], every key in A2 ≥ p

Assume that we have a method ***partition(A,first,last,pivotIndex)*** that will return the position of the pivot p in the **sorted** array A.

```
//  sort A[first..last] into non-decreasing order
void QuickSort(DataType A[], int first, int last)
{
    int pivotIndex;
    if (first < last)
    {   partition(A,first,last,pivotIndex);
        QuickSort(A,first,pivotIndex–1); // sort S1

        QuickSort(A,pivotIndex+1,last);  // sort S2
     }
}   //  end QuickSort
```

Two Fundamental Operations:

Q: How do we select the pivot p in A?
How do we partition A into sub-arrays S1 and S2?

Selecting a Pivot for A:
Given an array A[first..last].
Some general methods in selecting a pivot:

1. Use first element A[first]
2. Use last element A[last]
3. Use middle element A[middle] with middle = (first+last)/2
4. Use a random key among elements in A
5. If |A| > 3, use the median of A[first], A[middle], and A[last]. This is called the median-of-three method.

Q: Which method should we use?

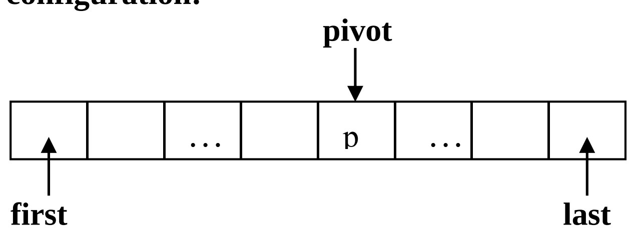
Characteristics of a "good" pivot:

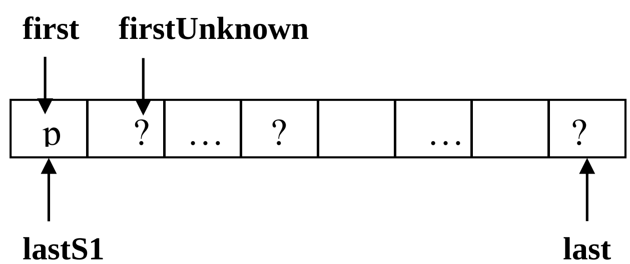
1. The pivot p can be computed in O(1) time.
2. A can be partitioned into A1 and A2 with "roughly" equal sizes.

Remark: Use median-of-three, or median-of-five, method.

Partitioning A[first..last]:

Initial configuration:



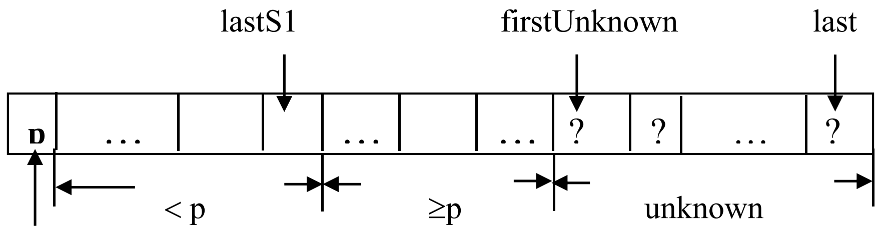


lastS1: Pointing at last element in S1.

firstUnknown: Pointing at current item to be compared with pivot.

Initially, lastS1 = first;
firstUnknown = first + 1;

General configuration:




```
if A[firstUnknown] < pivot              //elements out of order
{
    lastS1 = lastS1 + 1;                //find location to hold A[firstUnknown]
    swap(A[lastS1], A[firstUnknown]);
}
```

Let's consider using the middle key as the pivot.

Algorithm: partition(A,first,last)

```
    middle = (first+last)/2;
    pivot = A[middle];
    swap(A[middle],A[first];
    lastS1 = first;
    firstUnknown = first+1;
    for (; firstUnknown <= last; ++firstUnknown)
    {
        if (A[firstUnknown] < pivot)
        {
            ++lastS1;
            swap(A[firstUnknown],A[lastS1];
        }
    swap(A[first],A[lastS1]);
    pivotIndex = lastS1;
    }   //   endPartition
```

Complexity:

Worst-Case Complexity:

If Array a[] is in sorted order, we have

$$T(1) = 0,$$
$$T(n) = T(n-1) + (n-1), n > 1.$$

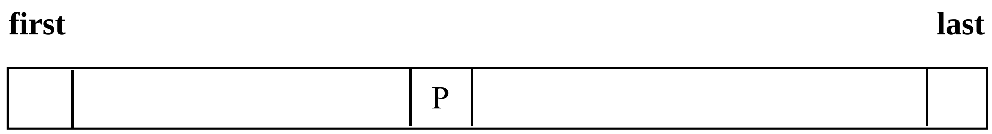
$$\begin{aligned}
\therefore \quad T(n) &= T(n-1) + (n-1) \\
&= T(n-2) + (n-2) + (n-1) \\
&= T(n-3) + (n-3) + (n-2) + (n-1) \\
&= \ldots \\
&= T(n-(n-1)) + 1 + 2 + \ldots + (n-1) \\
&= n(n-1)/2 \\
&= \Theta(n^2).
\end{aligned}$$

Average-Case Complexity:
Recall that

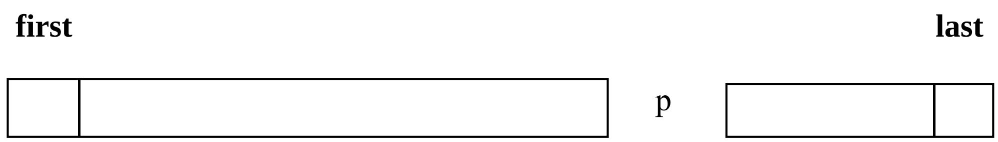
$T_a(n) = \Sigma_{I \in Dn} Pr(I) * C(I)$.

Q: What are the inputs to the problem?

Before A is sorted:



After A is sorted:



Q: Where will the pivot p go?

Assumption:
Assume that all n! permutations are equally likely. Hence, it is equally likely for the pivot p to occupy any one of the (last-first+1) positions.

$\therefore$ There are n types of inputs with p occupying the 1^{st}, 2^{nd}, …, nth position.

Pr(p occupies the ith position) = $\frac{1}{n}$.

$$
\begin{aligned}
&T(n) \\
&= \sum_{i=1}^{n} \frac{1}{n}[T(i-1) + T(n-i) + n - 1] \\
&= \frac{1}{n}\sum_{i=1}^{n} [T(i-1) + T(n-i)] + (n-1) \\
&= \frac{2}{n}\sum_{i=1}^{n} T(i-1) + (n-1).
\end{aligned}
$$

Hence,

$$nT(n) = 2\sum_{i=1}^{n} T(i-1) + n(n-1).$$

Substituting n = n-1 into the above equation, we have

$$(n-1)T(n-1) = 2\sum_{i=1}^{n-1} T(i-1) + (n-1)(n-2).$$

On subtracting, we have

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2(n-1)$$

$$nT(n) = (n+1)T(n-1) + 2(n-1).$$

Divide the above equation by n(n+1) to get

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)}.$$

$$\begin{aligned}
&\frac{T(n)}{n+1}\\
&\le \frac{T(n-1)}{n} + \frac{2}{n+1}\\
&\le \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}\\
&\le \frac{T(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}\\
&\le \ldots\\
&\le \frac{T(1)}{2} + 2\sum_{i=3}^{n+1}\frac{1}{i}\\
&= 2\sum_{i=3}^{n+1}\frac{1}{i}\\
&\le 2\int_{2}^{n+1}\frac{1}{x}dx\\
&= 2[\ln(n+1) - \ln 2].
\end{aligned}$$

$$\begin{aligned}
&T(n)\\
&= 2(n+1)[\ln(n+1) - \ln 2]\\
&= O(n \lg n).
\end{aligned}$$

Remarks:

- “Balancing” the sizes of the subproblems in DAC algorithm is very critical!
- To improve upon (local) performance, if |A| < 10, use insertion sort.