DIVIDE AND CONQUER TECHNIQUE

GENERAL ALGORITHM:

1. **Divide** problem into several smaller sub problems
   - Normally, the sub problems are similar to the original
2. **Conquer** the sub problems by solving them recursively
   - Base case: solve small enough problems by brute force
3. **Combine** the solutions to get a solution to the sub problems
   - And finally a solution to the original problem
4. Divide and Conquer algorithms are normally recursive

Advantage: Recursive algorithms are efficient,

Disadvantage: If the instance obtained as a result of division are unbalanced , then divide and algorithm are not effective.

```
Algorithm DAndC(P)
{
    if Small(P) then return S(P);
    else
    {
        divide P into smaller instances P_1, P_2, ..., P_k, k >= 1;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC(P_1),DAndC(P_2),...,DAndC(P_k));
    }
}
```

D&C Algorithm Time Complexity

- $T(n)$: running time for input size n
- $D(n)$: time of **Divide** for input size n
- $C(n)$: time of **Combine** for input size n
- a: number of subproblems
- n/b: size of each subproblem

$$T(n) = \begin{cases} O(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

1. **Binary Search**

```
Algorithm BinSearch(a, n, x)
// Given an array a[1 : n] of elements in nondecreasing
// order, n ≥ 0, determine whether x is present, and
// if so, return j such that x = a[j]; else return 0.
{
    low := 1; high := n;
    while (low ≤ high) do
    {
        mid := ⌊(low + high)/2⌋;
        if (x < a[mid]) then high := mid − 1;
        else if (x > a[mid]) then low := mid + 1;
            else return mid;
    }
    return 0;
}
```

Illustration:

Input: $-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151$,

Search element x=151

$x = 151$	*low*	*high*	*mid*
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

$x = -14$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found

$x = 9$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	4	6	5
			found

Time Complexity:

successful searches			**unsuccessful searches**
$\Theta(1)$,	$\Theta(\log n)$,	$\Theta(\log n)$	$\Theta(\log n)$
best,	average,	worst	best, average, worst

2. Min Max Algorithm:

```
Algorithm StraightMaxMin(a, n, max, min)
// Set max to the maximum and min to the minimum of a[1 : n].
{
    max := min := a[1];
    for i := 2 to n do
    {
        if (a[i] > max) then max := a[i];
        if (a[i] < min) then min := a[i];
    }
}
```

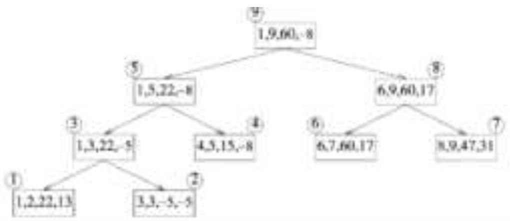
Time Complexity :
No. of Element comparisons
Best case :n-1(elements are in ascending order)
Worst Case:2(n-1)(elements are in descending order)
Average Case: 3n/2-1

MIN MAX using Divide and Conquer:

```
Algorithm MaxMin(i, j, max, min)
// a[1 : n] is a global array. Parameters i and j are integers,
// 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
// largest and smallest values in a[i : j], respectively.
{
    if (i = j) then max := min := a[i]; // Small(P)
    else if (i = j - 1) then // Another case of Small(P)
        {
            if (a[i] < a[j]) then
            {
                max := a[j]; min := a[i];
            }
            else
            {
                max := a[i]; min := a[j];
            }
        }
        else
        {   // If P is not small, divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(i + j)/2⌋;
            // Solve the subproblems.
                MaxMin(i, mid, max, min);
                MaxMin(mid + 1, j, max1, min1);
            // Combine the solutions.
                if (max < max1) then max := max1;
                if (min > min1) then min := min1;
        }
}
```

Illustration:

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	−5	−8	15	60	17	31	47



Time Complexity:

Now what is the number of element comparisons needed for MaxMin? If $T(n)$ represents this number, then the resulting recurrence relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

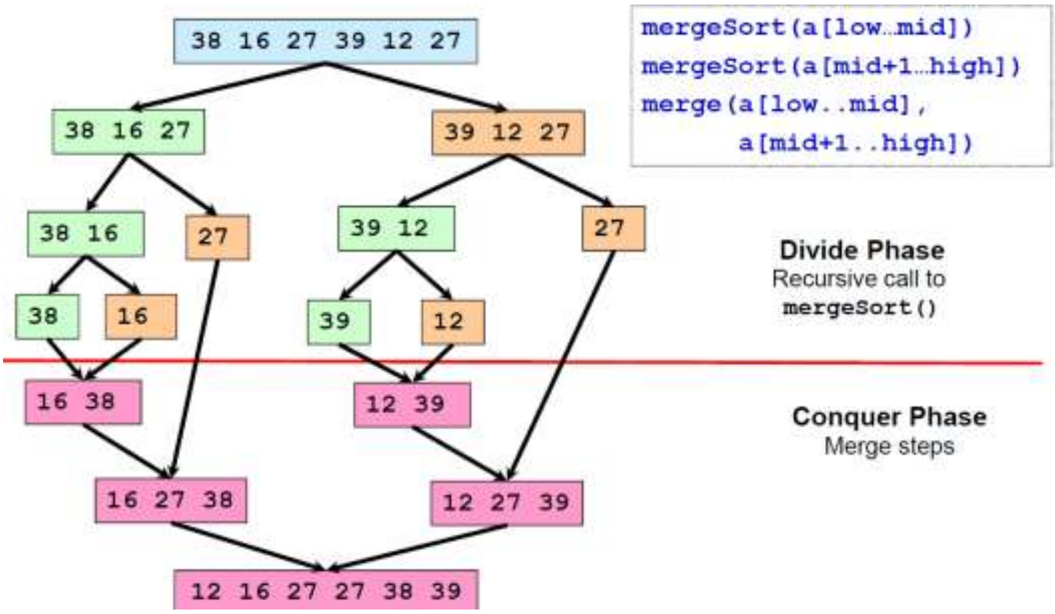
When n is a power of two, $n = 2^k$ for some positive integer k, then

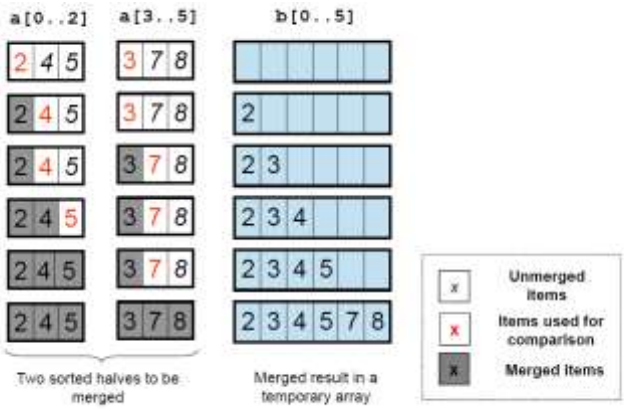
$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &\vdots \\ &= 2^{k-1}T(2) + \textstyle\sum_{1 \le i \le k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 = 3n/2 - 2 \end{aligned} \tag{3.3}$$

Note that $3n/2 - 2$ is the best-, average-, and worst-case number of comparisons when n is a power of two.

3. Merge Sort
-To sort array of elements
-Out-Place Algorithm –Needs additional memory

Illustration:






```
void mergeSort(int a[], int low, int high) {
  if (low < high) {
    int mid = (low+high) / 2;

    mergeSort(a, low   , mid );
    mergeSort(a, mid+1, high);

    merge(a, low, mid, high);
  }
}
```

Merge sort on a[low...high]

Divide a[ ] into two halves and **recursively** sort them

Conquer: merge the two sorted halves

Function to merge **a[low…mid]** and **a[mid+1…high]** into **a[low…high]**

```
void merge(int a[], int low, int mid, int high) {
  int n = high-low+1;
  int* b = new int[n];
  int left=low, right=mid+1, bIdx=0;

  while (left <= mid && right <= high) {
    if (a[left] <= a[right])
      b[bIdx++] = a[left++];
    else
      b[bIdx++] = a[right++];
  }

  while (left <= mid) b[bIdx++] = a[left++];
  while (right <= high) b[bIdx++] = a[right++];

  for (int k = 0; k < n; k++)
    a[low+k] = b[k];

  delete [] b;
}
```

b is a temporary array to store result

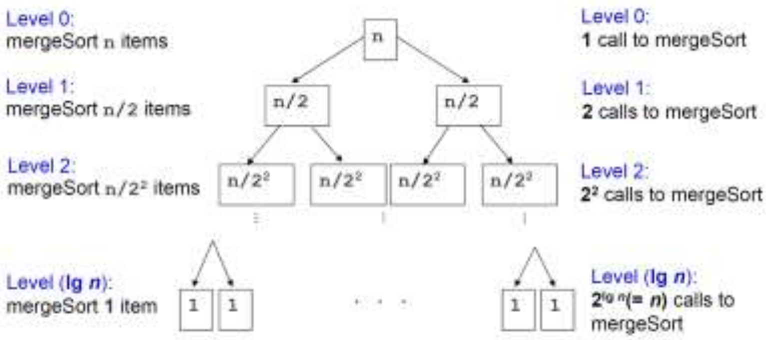
Normal Merging Where both halves have unmerged items

Remaining items are copied into `b[]`

Merged result are copied back into `a[]`

Remember to free allocated memory

Time Complexity: Best Case Analysis



$n/(2^k) = 1 \rightarrow n = 2^k \rightarrow k = \lg n$

- Level 0: **0** call to `merge()`
- Level 1: **1** calls to `merge()` with ***n*/2** items in each half,
  $O(1 \times 2 \times n/2) = O(n)$ time
- Level 2: **2** calls to `merge()` with ***n*/2²** items in each half,
  $O(2 \times 2 \times n/2^2) = O(n)$ time
- Level 3: **2²** calls to `merge()` with ***n*/2³** items in each half,
  $O(2^2 \times 2 \times n/2^3) = O(n)$ time
- …
- Level (lg n): $2^{\lg(n)-1}$ **(=** $n/2$**)** calls to `merge()` with $n/2^{\lg(n)}$ **(= 1)** item in each half, $O(n)$ time
- Total time complexity = $O(n \lg(n))$

Pros:

The performance is guaranteed, i.e. unaffected by original ordering of the input v
Suitable for extremely large number of inputs
Can operate on the input portion by portion

Cons:

Not easy to implement
Requires additional storage during merging operation
O(n) extra memory storage needed

----

5. Quick Sort

Quick Sort is a divide-and-conquer in –place algorithm

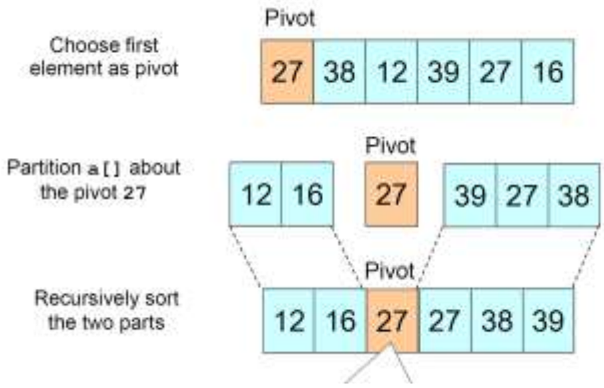
Divide step

- Choose an item *p* (known as pivot) and partition theitems of a[i...j] into two parts
- Items that are smaller than *p*
- Items that are greater than or equal to *p*
- Recursively sort the two parts

Conquer step

Do nothing!

In comparison, Merge Sort spends most of the time in conquer step but very little time in divide step



(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	p
65	70	75	80	85	60	55	50	45	$+\infty$	2	9
65	45	75	80	85	60	55	50	70	$+\infty$	3	8
65	45	50	80	85	60	55	75	70	$+\infty$	4	7
65	45	50	55	85	60	80	75	70	$+\infty$	5	6
65	45	50	55	60	85	80	75	70	$+\infty$	6	5
60	45	50	55	65	85	80	75	70	$+\infty$		

```
Algorithm QuickSort(p, q)
// Sorts the elements a[p], ..., a[q] which reside in the global
// array a[1 : n] into ascending order; a[n + 1] is considered to
// be defined and must be ≥ all the elements in a[1 : n].
{
    if (p < q) then  // If there are more than one element
    {
        // divide P into two subproblems.
            j := Partition(a, p, q + 1);
                // j is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort(p, j − 1);
            QuickSort(j + 1, q);
        // There is no need for combining solutions.
    }
}
```

```
Algorithm Partition(a, m, p)
// Within a[m], a[m + 1], ..., a[p − 1] the elements are
// rearranged in such a manner that if initially t = a[m],
// then after completion a[q] = t for some q between m
// and p − 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
// for q < k < p. q is returned. Set a[p] = ∞.
{
    v := a[m]; i := m; j := p;
    repeat
    {
        repeat
            i := i + 1;
        until (a[i] ≥ v);

        repeat
            j := j − 1;
        until (a[j] ≤ v);

        if (i < j) then Interchange(a, i, j);

    } until (i ≥ j);

    a[m] := a[j]; a[j] := v; return j;
}
```

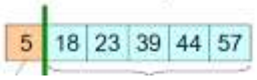
```
Algorithm Interchange(a, i, j)
// Exchange a[i] with a[j].
{
    p := a[i];
    a[i] := a[j]; a[j] := p;
}
```

Quick Sort: Best/Average Case Analysis

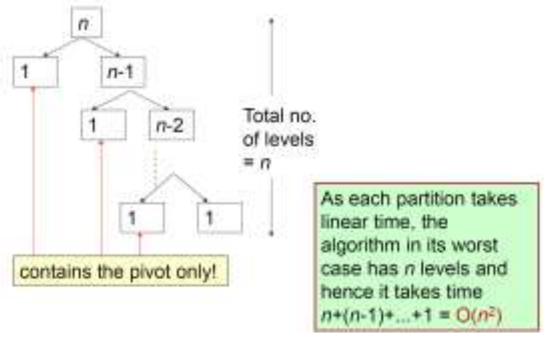
- Best case occurs when partition always splits the array into two equal halves
  Depth of recursion is log *n*. Each level takes *n* or fewer comparisons, so the time complexity is O(*n* log *n*)
- Average time is also Θ (*n* log *n*)

Worst Case Analysis:

- When the array is already in ascending order



Quick Sort: Worst Case Analysis



Average Case is also Θ(Nlogn)

6. Maximum Subarray Problem

Problem: given an array of n numbers, find the (a) contiguous subarray whose sum has the largest value.

Application: an unrealistic stock market game, in which you decide when to buy and see a stock, with full knowledge of the past and future. The restriction is that you can perform just one buy followed by a sell. The buy and sell both occur right after the close of the market.

The interpretation of the numbers: each number represents the stock value at closing on any particular day.

- Input: A sequence $A[1], A[2], \ldots, A[n]$ of integers.
- Output: Two indicex i and j with $1 \leq i \leq j \leq n$ that maximize

$$A[i] + A[i+1] + \cdots + A[j].$$

Max Subarray

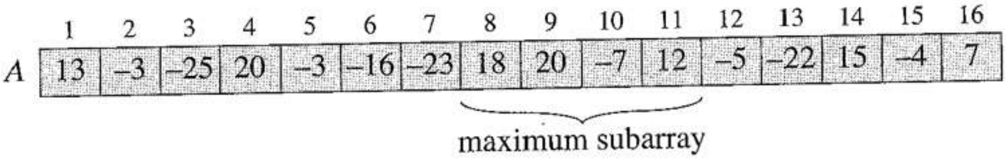


Figure 4.3 The change in stock prices as a maximum-subarray problem. Here, the subarray $A[8..11]$, with sum 43, has the greatest sum of any contiguous subarray of array A.

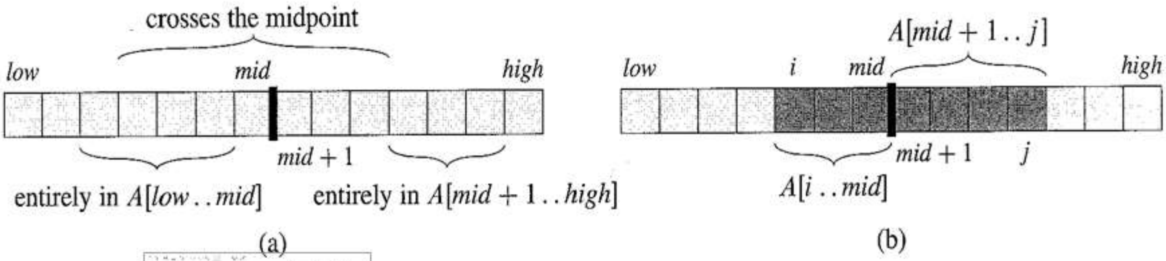


Figure 4.4 (a) Possible locations of subarrays of $A[low..high]$: entirely in $A[low..mid]$, entirely in $A[mid+1..high]$, or crossing the midpoint mid. **(b)** Any subarray of $A[low..high]$ crossing the midpoint comprises two subarrays $A[i..mid]$ and $A[mid+1..j]$, where $low \le i \le mid$ and $mid < j \le high$.

```
MaxCrossSubarray(A, i, k, j)
  left_sum = -∞
  sum=0
  for p = k downto i
    sum = sum + A[p]
    if sum > left_sum
      left_sum = sum
      max_left = p

  right_sum = -∞
  sum=0
  for q = k+1 to j
    sum = sum + A[q]
    if sum > right_sum
      right_sum = sum
      max_right = q
  return (max_left, max_right, left_sum + right_sum)
```

$$\left.\begin{array}{l} O(k-i+1) \\ O(j-k) \end{array}\right\} = O(j-i+1)$$

```
MaxSubarray(A, i, j)
  if i == j // base case
    return (i, j, A[i])
  else // recursive case
    k = floor((i + j) / 2)
    (l_low, l_high, l_sum) = MaxSubarray(A, i, k)
    (r_low, r_high, r_sum) = MaxSubarray(A, k+1, j)
    (c_low, c_high, c_sum) = MaxCrossSubarray(A, i, k, j)

  if l_sum >= r_sum and l_sum >= c_sum // case 1
    return (l_low, l_high, l_sum)
  else if r_sum >= l_sum and r_sum >= c_sum // case 2
    return (r_low, r_high, r_sum)
  else // case 3
    return (c_low, c_high, c_sum)
```

Divide

Conquer

Combine

Master Theorem

The Master Theorem applies to recurrences of the following form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

There are 3 cases:

1. If $f(n) = O(n^{\log_b a - s})$ for some constant $s > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + s})$ with $s > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$.
   Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

Method 2:

By Master theorem, we can solve the following equations :

$T(n) = aT(n/b) + \theta(n^k \log^p n)$

Where a>=1, b>=1, k>=0 and p is real numbers.

Case 1: $a>b^k$, then $T(n) = \theta(n \log b^{a})$

Case 2: $a= b^k$, then

i. If p>-1 , then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
ii. If p=-1 , then $T(n) = \theta(n^{\log_b a} \log \log n)$
iii. If p<-1 , then $T(n) = \theta(n^{\log_b a})$

Case 3: $a<b^k$, then

i. If p>=0 , then $T(n) = \theta(n^k \log^p n)$
ii. If p< 0, then $T(n) = \theta(n^k)$

Practice Problems

For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

Problem 1-1. $T(n) = 3T(n/2) + n^2$ T

$(n) = \Theta(n^2)$ (case 3).

Problem 1-2. $T(n) = 7T(n/2) + n^2$ T
$(n) = \Theta(n^{\lg 7})$ (case 1).

Problem 1-3. $T(n) = 4T(n/2) + n^2$ T
$(n) = \Theta(n^2 \lg n)$ (case 2).

Problem 1-4. $T(n) = 3T(n/4) + n \lg n$ T
$(n) = \Theta(n \lg n)$ (case 3).

Problem 1-5. $T(n) = 4T(n/2) + \lg n$ T
$(n) = \Theta(n^2)$ (case 1).

Problem 1-6. $T(n) = T(n - 1) + n$
M.T. doesn't apply. Iteration gives $T(n) = \Theta(n^2)$.

Problem 1-7. $T(n) = 4T(n/2) + n^2 \lg n$ T
$(n) = \Theta(n^2 \lg^2 n)$ (extended case 2).

Problem 1-8. $T(n) = 5T(n/2) + n^2 \lg n$ T
$(n) = \Theta(n^{\lg 5})$ (case 1).
Problem 1-9. $T(n) = 3T(n/3) + n/\lg n$
M.T. case 1 doesn't apply since $f(n) = n/\lg n$ is not polynomially smaller than $n^{\log 3\ 3-\varepsilon}$ for any ε > 0.

Problem 1-10. $T(n) = 2T(n/4) + c$ T
$(n) = \Theta(n^{1/2})$ (case 1).

Problem 1-11. $T(n) = T(n/4) + \lg n$ T
$(n) = \Theta(\lg^2 n)$ (extended case 2).

Problem 1-12. $T(n) = T(n/2) + T(n/4) + n^2$
M.T. doesn't apply. Recursion tree gives guess $T(n) = \Theta(n^2)$.

Problem 1-13. $T(n) = 2T(n/4) + \lg n$ T
$(n) = \Theta(n^{1/2})$ (case 1).

Problem 1-14. $T(n) = 3T(n/3) + n \lg n$ T
$(n) = \Theta(n \lg^2 n)$ (extended case 2).

Problem 1-15. $T(n) = 8T((n - \sqrt{n})/4) + n^2$
M.T. doesn't apply. Using Akra-Bazzi can ignore

$n/4$, which gives $\Theta(n^2)$. Could also use M.T. to get an upper bound of $O(n^2)$ by removing the $\sqrt{n}/4$ term and a lower bound of $\Omega(n^2)$ by replacing the $(n - \sqrt{n})/4$ term by $0.24n$.

Problem 1-16. $T(n) = 2T(n/4) + \sqrt{n}$ $T(n) = \Theta(n^{1/2} \lg n)$ (case 2).

Problem 1-17. $T(n) = 2T(n/4) + n^{0.51}$ $T(n) = \Theta(n^{0.51})$ (case 3).

Problem 1-18. $T(n) = 16T(n/4) + n!$ $T(n) = \Theta(n!)$ (case 3).

Problem 1-19. $T(n) = 3T(n/2) + n$ $T(n) = \Theta(n^{\lg 3})$ (case 1).

Problem 1-20. $T(n) = 4T(n/2) + cn$ $T(n) = \Theta(n^2)$ (case 1).

Problem 1-21. $T(n) = 3T(n/3) + n/2$ $T(n) = \Theta(n \lg n)$ (case 2).

Problem 1-22. $T(n) = 4T(n/2) + n/\lg n$ $T(n) = \Theta(n^2)$ (case 1).

Problem 1-23. $T(n) = 7T(n/3) + n^2$ $T(n) = \Theta(n^2)$ (case 3).

Problem 1-24. $T(n) = 8T(n/3) + 2^n$
$T(n) = \Theta(2^n)$ (case 3).

Problem 1-25. $T(n) = 16T(n/4) + n$ $T(n) = \Theta(n^2)$ (case 1).