Unit 3.1 Divide and Conquer

Algorithms

EE/NTHU

Mar. 31, 2020

Divide and Conquer

- `Divide and Conquer` method:
  - Given an input set P, Divide and conquer approach splits the input into k distinct subsets, $1 < k < n$, yielding k subproblems.
  - These k subproblems are solved individually.
  - Then a method must be found that combines the subsolutions into a solution of the whole problem.

Algorithm 3.1.1. Divide and conquer

```
   // Divide and conquer algorithm.
   // Input: P
   // Output: Solution of P.
Algorithm DandC(P)
{
    if Small(P) then return S(P); // Small size, solve immediately and return.
    else {
        divide P into smaller instances P_1, P_2, ..., P_k, k > 1;
        // Apply DandC to each of these subproblems and combine for solution.
        return Combine( DandC(P_1), DandC(P_2), ..., DandC(P_k));
    }
}
```

Binary Search

- Given an array A with n elements sorted in nondecreasing order, the following algorithm determines if the element x is in A or not. If it is, return j such that $A[j] = x$, otherwise return 0.

Algorithm 3.1.2. Binary Search

```
  // Find if x is in nondecreasing array A[ℓ : h].
  // Input: A[ℓ : h] and x
  // Output: j, ℓ ≤ j ≤ h, such that A[j] = x, otherwise 0.
Algorithm BinSrch(A, ℓ, h, x)
{
    if (ℓ = h) then {
        if (x = A[ℓ]) then return ℓ;
        else return 0;
    } else {
        mid := ⌊(ℓ + h)/2⌋;
        if (x = A[mid]) then return mid;
        else if (x < A[mid]) then return BinSrch(A, ℓ, mid − 1, x);
        else return BinSrch(A, mid + 1, h, x);
    }
}
```

- BinSrch$(A, 1, n, x)$ is called in main function.

Iterative Binary Search

- Iterative binary search.

Algorithm 3.1.3. Iterative Binary Search

```
     // Iterative binary search for x in nondecreasing array A[1 : n].
     // Input: A, n and x
     // Output: j such that A[j] = x, otherwise 0.
1  Algorithm BinSearch(A, n, x)
2  {
3      low := 1; high := n; // initialize search range
4      while (low ≤ high) do { // more to search?
5          mid := ⌊(low + high)/2⌋; // center of search range
6          if (x = A[mid]) then return mid; // if x is found, return.
7          else if (x < A[mid]) then high := mid − 1; // reduce search range.
8          else low := mid + 1;
9      }
10     return 0; // x not found.
11 }
```

- Two element comparisons per iteration, lines 6, 7.

Binary Search Examples

- Example

$$A = \{ \text{-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151} \}.$$
$$[1]\ [2]\ [3]\ [4]\ [5]\ [6]\ [7]\ [8]\ [9]\ [10]\ [11]\ [12]\ [13]\ [14]$$

Note that $n = 14$ and A is sorted in nondecreasing order.

BinSearch$(A, 14, 151)$

iter	*low*	*high*	*mid*
1	1	14	7
2	8	14	11
3	12	14	13
4	14	14	14
return 14			

BinSearch$(A, 14, 9)$

iter	*low*	*high*	*mid*
1	1	14	7
2	1	6	3
3	4	6	5
return 5			

BinSearch$(A, 14, -14)$

iter	*low*	*high*	*mid*
1	1	14	7
2	1	6	3
3	1	2	1
4	2	2	2
5	2	1	
return 0			

Binary Search – Correctness

Theorem 3.1.4.

Algorithm BinSearch(A, n, x) works correctly.

Proof. Assuming all comparison operations are properly defined, and initially, $low = 1$, $high = n$, $A[1] \le A[2] \le \cdots \le A[n]$. If $n = 0$, then the **while** loop is not entered and 0 is returned. Otherwise, $low \le mid \le high$. If $x = A[mid]$ then the algorithm terminated successfully. Otherwise, the range is narrowed to either $[low : mid - 1]$ or $[mid + 1 : high]$. Note that if $low > mid - 1$ or $mid + 1 > high$ then the algorithm terminates and returns 0, which is also a correct result. Since n is finite, the **while** loop can be executed at most $(\lg n + 1)$ times. Therefore, the algorithm always terminates and returns the right answer. $\square$

- To fully test BinSearch algorithm:
  - To test all successful searches, $x \in A[i]$, $i = 1, \cdots, n$ – n cases,
  - To test all unsuccessful cases, $x \notin A[i]$, $i = 1, \cdots, n$ – $n + 1$ cases,
  - Totally $2n + 1$ cases.

Binary Search – Complexities

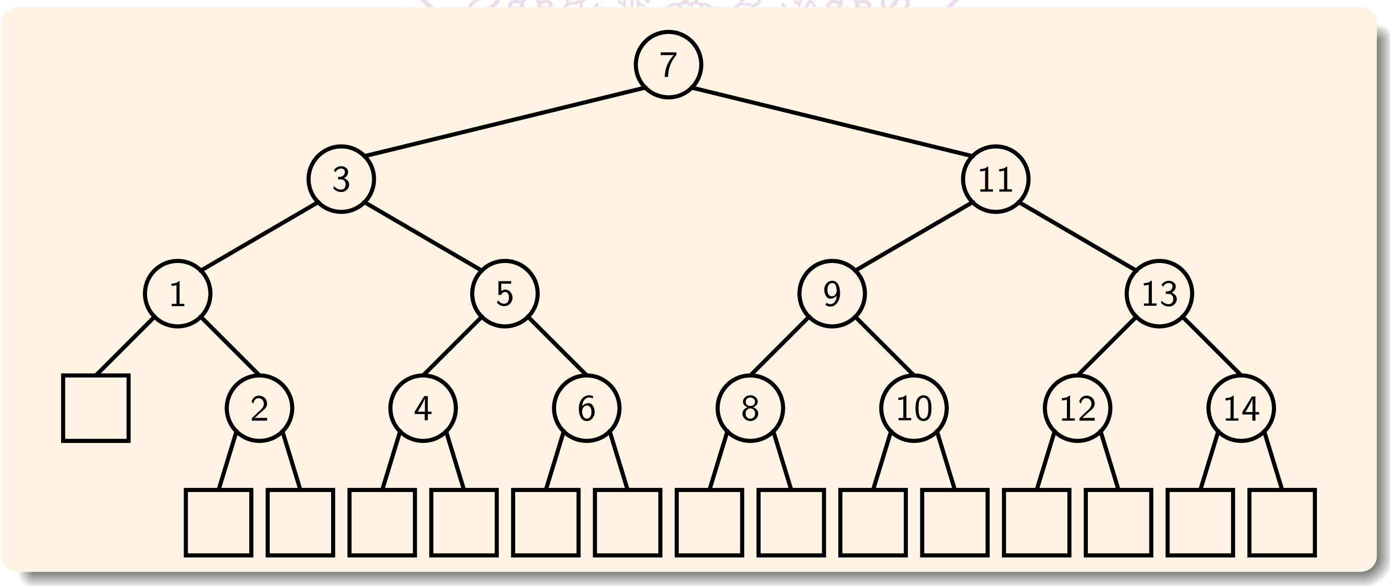
- The space complexity of `BinSearch`(A, n, x) is $(n+4)$
  - n for array A, and then low, $high$, mid and x take 4 spaces.
- The number of comparisons for each element of A

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	
$a = \{$	-15,	-6,	0,	7,	9,	23,	54,	82,	101,	112,	125,	131,	142,	151	$\}$.
Comp.,	3	4	2	4	3	4	1	4	3	4	2	4	3	4	

- Thus, for successful search
  - Best case: 1 comparison
  - Worst case: 4 comparisons
  - Average case: $\frac{45}{14} = 3.21$ comparisons

Binary Search – Unsuccessful Search

- For unsuccessful search
  - $x < A[1]$: 3 comparisons.
  - All other cases: 4 comparisons.
  - Best case: 3 comparisons.
  - Worst case: 4 comparisons.
  - Average case: $\frac{3 + 4 * 14}{15} = \frac{59}{15} = 3.93.$
- The binary decision tree for 14-element array searching



Binary Search – Number of Comparisons

Theorem 3.1.5.

If n is the in range $[2^{k-1}, 2^k)$, then BinSearch(A, n, x) makes at most k element comparisons for a successful search and either $k-1$ or k comparisons for an unsuccessful search. In other words, the time for a successful search is $\mathcal{O}(\lg n)$ and for an unsuccessful search is $\Theta(\lg n)$.

Proof. Consider the binary decision tree describing the comparisons of the BinSearch(A, n, x) algorithm. All successful searches end at a circular node whereas all unsuccessful searches end at a square node. If $2^{k-1} \le n < 2^k$, then all circular nodes are at levels 1, 2, $\cdots$, k whereas all square nodes are at levels k and $k+1$. The number of comparisons needed to terminate a circular node at level i is i whereas the number of comparisons needed to terminate at a square node at level i is $i-1$. Thus, the theorem follows. □

- The above theorem is the worst case time complexity of BinSearch algorithm.

Binary Search – Average-case Complexity

- To determine the average case complexity, focus on the binary decision tree again.
- Successful searches terminate at circular nodes – internal nodes.
  - The distance from any internal node to the root is the level -1.
  - The internal node path length, I, is the sum of the distances of all internal nodes to the root.
- Unsuccessful searches terminate at the square nodes – external nodes.
  - The external node path length, E, is the sum of the distances of all external nodes to the root.
- It can be shown that

$$E = I + n + 1 \tag{3.1.1}$$

- Let $A_s(n)$ be the average number of comparisons in a successful search then

$$A_s(n) = 1 + I/n. \tag{3.1.2}$$

- Let $A_u(n)$ be the average number of comparisons in a unsuccessful search then

$$A_u(n) = E/(n+1). \tag{3.1.3}$$

- Note that for a binary decision tree with n internal nodes, there are $n+1$ external nodes.

Binary Search – Time Complexities

- Combining these equations

$$A_s(n) = (1 + 1/n) A_u(n) - 1/n. \tag{3.1.4}$$

  - $A_s(n)$ and $A_u(n)$ have similar complexity.
- From Theorem (3.1.5) we know that E is proportional to $n \lg n$.
- Thus, both $A_u(n)$ and $A_s(n)$ are both proportional to $\lg n$.
- The following table summaries the time complexity of BinSearch(A, n, x).

	Successful search	Unsuccessful search
Best case	$\Theta(1)$	$\Theta(\lg n)$
Average case	$\Theta(\lg n)$	$\Theta(\lg n)$
Worst case	$\Theta(\lg n)$	$\Theta(\lg n)$

Binary Search – Improved

- In the algorithm BinSearch(A, n, x), two element comparisons are needed for each iteration.
- The following algorithm reduces the number of element comparisons to 1 per iteration – the complexity does not change.

Algorithm 3.1.6. Binary search with 1 comparison/iteration

```
// Improved binary search for x in nondecreasing array A[1 : n].
// Input: A, n and x
// Output: j such that A[j] = x, otherwise 0.
Algorithm BinSearch1(A, n, x)
{
    low := 1; high := n + 1; // initialize range, note high is out of range.
    while (low < high − 1) do { // iterate until one element left
        mid := ⌊(low + high)/2⌋;
        if (x < A[mid]) then high := mid; // compare to mid only
        else low := mid;
    }
    if (x = A[low]) then return low; // only one element left
    else return 0;
}
```

Finding the Maximum and Minimum

- Given a set of n elements, find the maximum and the minimum.
- The following algorithm is a straightforward implementation to solve the problem.

Algorithm 3.1.7. Find maximum and minimum

```
    // Find max and min of array A[1 : n].
    // Input: array A, int n
    // Output: max, min.
Algorithm SMaxMin(A, n, max, min)
{
    max := min := A[1]; // Initialize to a valid candidate.
    for i := 2 to n do { // Iterate for all elements.
        if (A[i] > max) then max := A[i];
        if (A[i] < min) then min := A[i];
    }
}
```

- The space complexity is $(n + 4)$.
- The time complexity, in terms of number of comparisons, is
  - Best case: $2(n - 1)$.
  - Average case: $2(n - 1)$.
  - Worst case: $2(n - 1)$.

Finding the Maximum and Minimum – Improved

- The preceding algorithm can be improved as

Algorithm 3.1.8. Find maximum and minimum

```
    // Find max and min of array A[1 : n].
    // Input: array A, int n
    // Output: max, min.
Algorithm SMaxMin1(A, n, max, min)
{
    max := min := A[1]; // Initialize to a valid candidate.
    for i := 2 to n do { // Iterate for all elements.
        if (A[i] > max) then max := A[i];
        else if (A[i] < min) then min := A[i];
    }
}
```

- The space complexity is still $(n + 4)$.
- The time complexity, in terms of number of comparisons, is
  - Best case: $n - 1$, if a is increasing order.
  - Worst case: $2(n - 1)$, if A is in decreasing order.

Finding the Maximum and Minimum – Divide and Conquer

- Using Divide and Conquer approach, we have the following algorithm

Algorithm 3.1.9. Find maximum and minimum

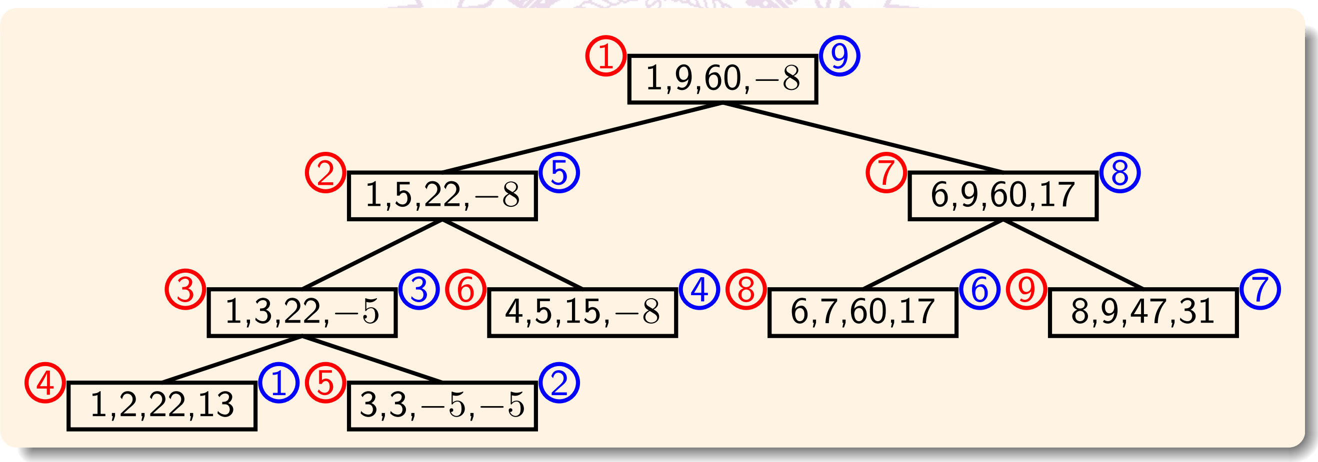
```
// Find max and min of array A[ℓ : h].
// Input: array A, int ℓ, h
// Output: max, min.
Algorithm MaxMin(A, ℓ, h, max, min)
{
    if (ℓ = h) then max := min := A[ℓ]; // Only one element.
    else if (ℓ = h − 1) then { // Two elements in the range.
        if (A[ℓ] < A[h]) then {
            max := A[h]; min := A[ℓ];
        }
        else {
            max := A[ℓ]; min := A[h];
        }
    }
    else { // Divide and conquer.
        mid := ⌊(ℓ + h)/2⌋;
        MaxMin(A, ℓ, mid, max, min);
        MaxMin(A, mid + 1, h, max1, min1);
        if (max < max1) max := max1;
        if (min > min1) min := min1;
    }
}
```


Finding the Maximum and Minimum – Example

- Example

$$A = \{\ 22,\ 13,\ -5,\ -8,\ 15,\ 60,\ 17,\ 31,\ 47\ \}$$
$$[1]\ [2]\ [3]\ [4]\ [5]\ [6]\ [7]\ [8]\ [9]$$

- The calling tree of MaxMin$(A, 1, 9, max, min)$



- Red color is the calling sequence.
- Blue color is the returning sequence.

Finding the Maximum and Minimum – Complexity

- To find the complexity of the recursive MaxMin algorithm, let $T(n)$ be the number of element comparisons.
- The recurrence relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases} \qquad (3.1.5)$$

- If $n = 2^k$, then

$$\begin{aligned} T(n) &= 2\,T(n/2) + 2 \\ &= 2(2\,T(n/4) + 2) + 2 \\ &= 4\,T(n/4) + 4 + 2 \\ &= 8\,T(n/8) + 8 + 4 + 2 \\ &= 2^{k-1}\,T(2) + \sum_{i=1}^{k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 \\ &= 3n/2 - 2 \end{aligned} \qquad (3.1.6)$$

- This is the best-case, average-case and worst-case complexity.

Finding the Maximum and Minimum – Analysis

- The worst-case time complexity of the recursive version of MaxMin algorithm (Algorithm 3.1.9) is 25% better than the straightforward implementation (Algorithm 3.1.8)
- However, Algorithm (3.1.9) has larger space complexity, $\Theta(\lfloor \lg n \rfloor \times 6)$, in addition to the space needed for the array.
  - The number of recursions is $\lfloor \lg n \rfloor$.
  - The variables for each recursive function call: i, j, max, min, $max1$, and $min1$.
- In Algigrithm (3.1.9), there are two integer comparisons
  - Lines 3 ($\ell = h$) and 4 ($\ell = h - 1$).
- Let's consider the time complexity if these comparisons are not negligible.
- These integer comparisons can be reduced in number as the following algorithm

Finding the Maximum and Minimum – Reduced Integer Comparison

Algorithm 3.1.10. Find maximum and minimum

```
// Find max and min of array A[ℓ : h].
// Input: array A, int ℓ, h
// Output: max, min.
Algorithm MaxMin1(A, ℓ, h, max, min)
{
    if (ℓ ≥ h − 1) then { // One or two elements in the range.
        if (A[ℓ] < A[h]) then {
            max := A[h]; min := A[ℓ];
        }
        else {
            max := A[ℓ]; min := A[h];
        }
    }
    else { // Otherwise, divide and conquer.
        mid := ⌊(ℓ + h)/2⌋;
        MaxMin(A, ℓ, mid, max, min);
        MaxMin(A, mid + 1, h, max1, min1);
        if (max < max1) max := max1;
        if (min > min1) min := min1;
    }
}
```


Finding the Maximum and Minimum – Complexity

- Let $C(n)$ be the number of comparisons, including integer comparisons, for the MaxMin1 algorithm, then

$$C(n) = \begin{cases} 2C(n/2) + 3 & n > 2 \\ 2 & n = 2 \end{cases} \tag{3.1.7}$$

and assume $n = 2^k$ then

$$\begin{aligned} C(n) &= 2C(n/2) + 3 \\ &= 4C(n/4) + 6 + 3 \\ &= 2^{k-1}C(2) + 3\sum_{i=0}^{k-2} 2^i \\ &= 2^k + 3 \times 2^{k-1} - 3 \\ &= 5n/2 - 3 \end{aligned} \tag{3.1.8}$$

- This is the best-case, average-case and worst-case complexity.
- Note for the straightforward implementation, Algorithm (3.1.8), the worst-case complexity, including integer comparison, is $3(n-1)$.

Finding the Maximum and Minimum – Comparisons

- Comparing the straightforward implementation, Algorithm (3.1.8), and the divide and conquer approach, Algorithm (3.1.10)
- Divide and conquer approach is effective if the key comparison, $A[i] > A[j]$, is dominating.
- But, when the key comparison is on the same order as the integer comparison then the straightforward implementation may be more effective.
  - Due to the recursion overhead.
- Design and analysis of computer algorithms needs to be carried out for specific problem instance.

- Divide-and-conquer approach often results in recursive implementation.
  - Space complexity can be larger.
- The following algorithm finds Maximum and Minimum with $3\lfloor n/2 \rfloor$ comparisons.
  - If n is even, it needs $3(n-2)/2 + 1 = 3n/2 - 2$ comparisons.
  - If n is odd, it needs $3(n-1)/2$ comparisons.

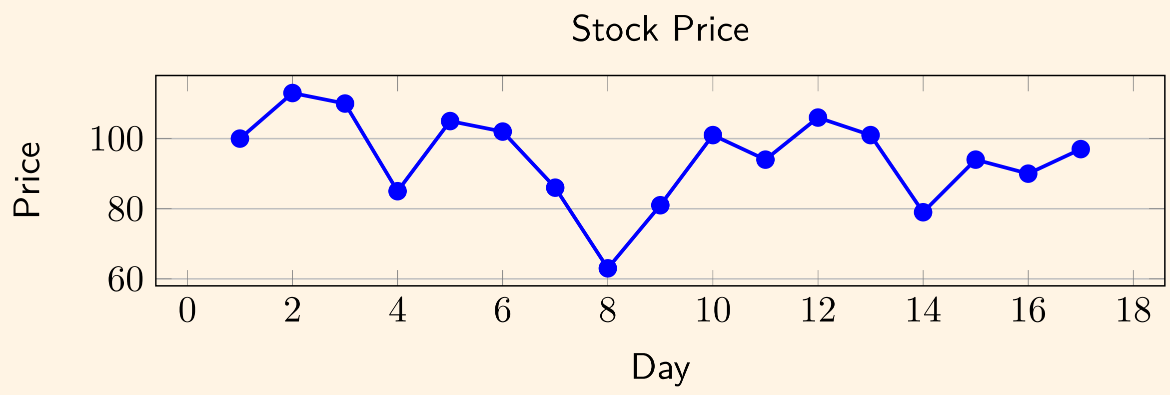
Finding the Maximum and Minimum – Iterative Algorithm

Algorithm 3.1.11. Iterative maximum and minimum

```
// Find max and min of array A[1 : n].
// Input: array A, int n
// Output: max, min.
Algorithm MaxMin_I(A, n, max, min)
{
    if (n mod 2 = 0) then { // n is even.
        if (A[1] > A[2]) then {
            max := A[1] ; min := A[2] ;
        } else {
            min := A[1] ; max := A[2] ;
        }
        i := 3 ;
    } else { // n is odd.
        min := A[1] ; max := A[1] ; i := 2 ;
    }
    while (i < n) do { // 3 comparisons for 2 elements.
        if (A[i] > A[i + 1]) { // J is the larger one.
            J := A[i] ; j := A[i + 1] ; // j is the smaller one.
        } else {
            j := A[i] ; J := A[i + 1] ;
        }
        if (j < min) min := j ; // compare j to min.
        if (J > max) max := J ; // compare J to max.
        i := i + 2 ;
    }
}
```

Maximum Subarray Problem

- Suppose the stock price of a company is known for a period of time. What is the maximum profit one can obtain for a single buy and sell transaction?



- The stock price data can be transformed into daily price change information as shown below. Then the problem is to find the range of the subarray with the maximum contiguous sum.

Day	1	2	3	4	5	6	7	8	9
Price	100	113	110	85	105	102	86	63	81
Change	0	13	-3	-25	20	-3	-16	-23	18

Day	10	11	12	13	14	15	16	17
Price	101	94	106	101	79	94	90	97
Change	20	-7	12	-5	-22	15	-4	7

Maximum Subarray Problem, II

- Maximum subarray problem:
  - Input: an array of size n, $A[n]$.
  - Output: range, low and $high$, such that

$$\sum_{i=low}^{high} A[i] = \max_{1 \le j \le k \le n} \sum_{i=j}^{k} A[i]. \tag{3.1.9}$$

  - Note that for the buying day for the stock is actually $low - 1$.
- Brute-force approach
  - To try out all possible ranges, $1 \le j \le k \le n$.
  - Total number of possbilities: $\sum_{i=1}^{n-1} = \frac{n(n-1)}{2}$.
  - Thus, the computational complexity of brute-force approach is $\Omega(n^2)$.
  - Since the summation operation needs to be carried out, the actual complexity should be $\Theta(n^3)$.

Maximum Subarray Problem – Brute-Force Approach

Algorithm 3.1.12. Maximum Subarray – Brute-Force Approach

```
// Find low and high to maximize ∑ A[i], low ≤ i ≤ high.
// Input: A[1 : n], int n
// Output: 1 ≤ low, high ≤ n and max.
Algorithm MaxSubArrayBF(A, n, low, high)
{
    max := 0 ; // Initialize
    low := 1 ;
    high := n ;
    for j := 1 to n do { // Try all possible ranges: A[j : k].
        for k := j to n do {
            sum := 0 ;
            for i := j to k do { // Summation for A[j : k]
                sum := sum + A[i] ;
            }
            if (sum > max) then { // Record the maximum value and range.
                max := sum ;
                low := j ;
                high := k ;
            }
        }
    }
    return max ;
}
```


Maximum Subarray Problem – Divide and Conquer

Algorithm 3.1.13. Maximum Subarray – Divide-and-Conquer Approach

```
// Find low and high to maximize ∑ A[i], begin ≤ low ≤ i ≤ high ≤ end.
// Input: A, int begin ≤ end
// Output: begin ≤ low, high ≤ end and max.
Algorithm MaxSubArray(A, begin, end, low, high)
{
    if (begin = end) then { // termination condition.
        low := begin; high := end;
        return A[begin] ;
    }
    mid := ⌊(begin + end)/2⌋ ;
    lsum := MaxSubArray(A, begin, mid, llow, lhigh) ; // left region
    rsum := MaxSubArray(A, mid + 1, end, rlow, rhigh) ; // right region
    xsum := MaxSubArrayXB(A, begin, mid, end, xlow, xhigh) ; // cross boundary
    if (lsum >= rsum and lsum >= xsum) then { // lsum is the largest
        low := llow; high := lhigh;
        return lsum;
    }
    else if (rsum >= lsum and rsum >= xsum) then { // rsum is the largest
        low := rlow; high := rhigh;
        return rsum;
    }
    low := xlow; high := xhigh;
    return xsum; // cross-boundary is the largest
}
```

Maximum Subarray Problem – Cross Boundary

Algorithm 3.1.14. Maximum Subarray – Cross Boundary

```
// Find low and high to maximize ∑ A[i], begin ≤ low ≤ mid ≤ high ≤ end.
// Input: A, int begin ≤ mid ≤ end
// Output: low ≤ mid ≤ high and max.
Algorithm MaxSubArrayXB(A, begin, mid, end, low, high)
{
      lsum := 0 ; // Initialize for lower half.
      low := mid ;
      sum := 0 ;
      for i := mid to begin step −1 do { // find low to maximize ∑ A[low : mid]
            sum := sum + A[i] ; // continue to add
            if (sum > lsum) then { // record if larger.
                  lsum := sum ;
                  low := i ;
            }
      }
      rsum := 0 ; // Initialize for higher half.
      high := mid + 1 ;
      sum := 0 ;
      for i := mid + 1 to end do { // find end to maximize ∑ A[mid + 1 : high]
            sum := sum + A[i] ; // Continue to add.
            if (sum > rsum) then { // Record if larger.
                  rsum := sum ;
                  high := i ;
            }
      }
      return lsum + rsum ; // Overall sum.
}
```


Maximum Subarray Problem – Complexity

- The number of comparisons for divide-and-conquer algorithm, MaxSubArray, is dominated by
$$T(n) = 2 \cdot T(n/2) + T_{XB}(n). \tag{3.1.10}$$
where T_{XB} is the number of comparisons of the algorithm MaxSubArrayXB.
- And,
$$T_{XB}(n) = n. \tag{3.1.11}$$
- Thus, assuming $n = 2^k$,
$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + n \\ &= 2(2 \cdot T(n/2^2) + n/2) + n \\ &= 2^2 \cdot T(n/2^2) + 2n \\ &= \cdots \\ &= 2^k \cdot T(n/2^k) + k \cdot n \\ &= n + n \cdot \lg n \end{aligned} \tag{3.1.12}$$
- The computational complexity of the divide-and-conquer MaxSubArray is $\Theta(n \cdot \lg n)$.

Summary

- Divide and conquer
- Binary search
  - Recursive algorithm
    - Recursion: $T(n) = T(\lceil n/2 \rceil) + c$
  - Iterative algorithm
  - Correctness
  - Complexity: $\mathcal{O}(\lg\, n)$
  - Improved algorithm
- Finding maximum and minimum
  - Straightforward implementation
  - Straightforward implementation, improved
  - Divide and conquer approach
    - Recursion: $T(n) = 2\,T(\lceil n/2 \rceil) + 2$
  - Complexity: $\mathcal{O}(n)$
  - Algorithm with reduced integer comparisons
  - Comparisons of different algorithms
- Maximum subarray problem
  - Brute-force approach
  - Divide-and-conquer approach
    - Recursion: $T(n) = 2\,T(\lceil n/2 \rceil) + n$
  - Computational complexity: $\mathcal{O}(n \cdot \lg\, n)$