CSIEB0120
Lecture 02
Divide-and-Conquer

Shiow-yang Wu & Wei-Che Chien
(吳秀陽 & 簡暐哲)
Department of Computer Science and Information Engineering
National Dong Hwa University

Objectives

- Describe the Divide-and-Conquer(D&C) strategy for solving problems(**what**)
- Apply the divide-and-conquer approach(**how**)
- Determine **when** to apply the divide-and-conquer strategy
- Complexity analysis of divide-and-conquer algorithms
- Contrast worst-case and average-case complexity analysis

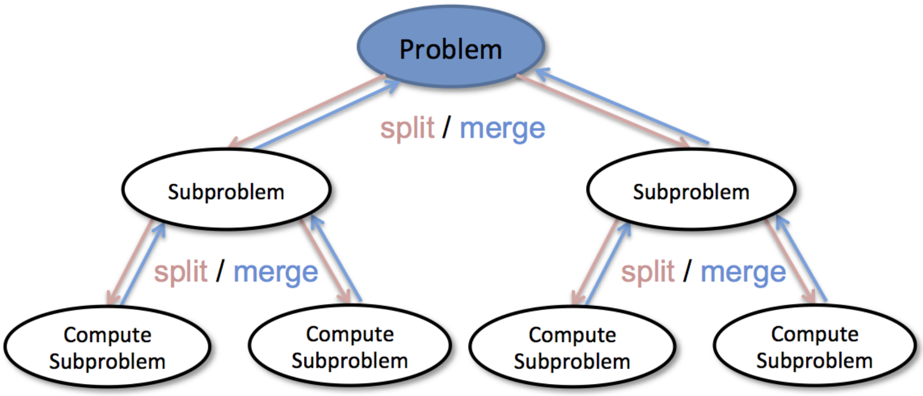
The Military Tactic of "Divide and Conquer"

- The famous Battle of Austerlitz on December 2, 1805 between Napoleon and Austro-Russian coalition army.
- Napoleon’s army was outnumbered by 15,000.
- Napoleon split the Austro-Russian army in two and conquer the smaller armies individually.
- Divide an instance of a hard problem into 2 or more smaller(easier) instances.
- Repeat the strategy until solvable instances.
- Top-down approach

Battle of Austerlitz

La bataille d'Austerlitz. 2 decembre 1805 (François Gérard)

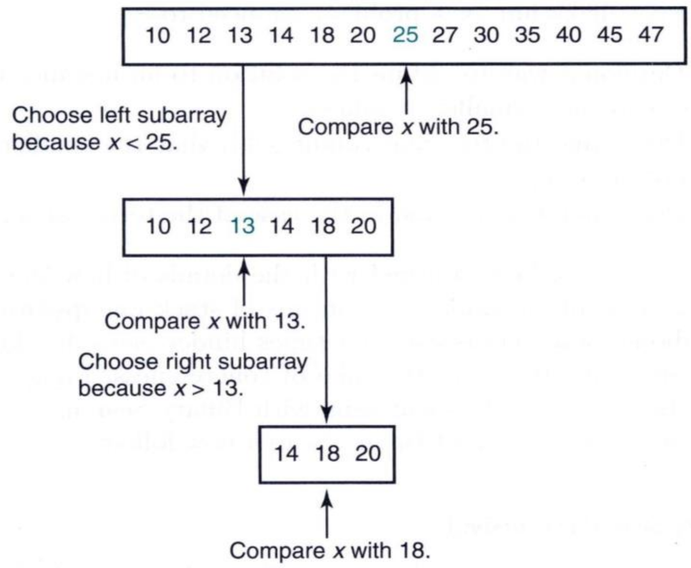
Principle of Divide-and-Conquer



Binary Search

- **Problem**: Locate key *x* in a sorted(non-decreasing) array of size *n*.
- If *x* equals the middle item *m* – found – quit. Else
  - *Divide* the array into two sub-arrays approximately in half
    - If *x* is smaller than *m*, select left sub-array
    - If *x* is larger than *m*, select right sub-array
  - *Conquer* (solve) the sub-array: Is *x* in the sub-array using recursion until the sub-array is sufficiently small (can be solved directly).
  - *Obtain* the solution to the array from the solution to the subarray.

Binary Search Example (x = 18)



Binary Search (Recursive)

```
index location(index low, index high) {
  index mid;

  if (low > high)
    return 0;  // Not found.
  else {
    mid = (low + high) / 2  // Integer div. Split in half.
    if (x == S[mid])
      return mid;  //  Found.
    else if (x < S[mid])
      return location(low, mid-1);   // Choose the left half.
    else
      return location(mid+1, high);  // Choose the right half.
  }
}
// Call as follows:  locationout = location(1, n);
```

Observations

- Reason for using a local variable *locationout*
  - Parameters n, S, x, are not changed during execution.
  - Dragging them over recursive calls are unnecessary.
- Tail-recursion
  - No operations are done after the recursive call.
  - Straightforward to produce an iterative version.
  - Recursion clearly illustrates the D&C process.
  - However, recursions is overburdensome due to excessive uses of activation records.
  - Memory can be saved by eliminating the stack for activation records. (reason for preferring to iteration)
  - Iterative version is better only as constant factor. Order is same.

Worst-Case Analysis (Binary Search)

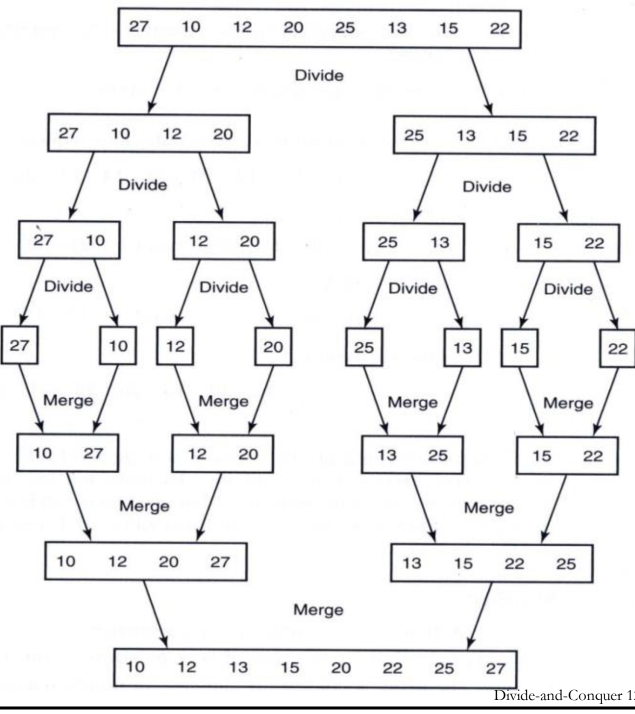
- **Basic operation**: compare x with S[n]
- **Input size**: n, the number of items in S
- **Analysis**:
  - Let *n* be a power of 2. Worst case occurs when *x* > S[n].
  - W(n) = W(n/2) + 1 for n>1 and n power of 2
    - W(n/2) = the no. of comparisons in the recursive call
    - 1 comparison at the top level
  - W(1) = 1
  - Example B1 in Appendix B: W(n) = lg n + 1
  - If n not a power of 2
    - $W(n) = \lfloor \lg n \rfloor + 1 \in \Theta(\lg n)$

Mergesort (Recursive)

- **Problem**: Sort an array S of size n (for simplicity, let n be a power of 2)
- *Divide* S into 2 sub-arrays of size n/2
- *Conquer* (solve) recursively sort each sub-array until array is sufficiently small (size 1)
- *Combine* (merge) the solutions to the sub-arrays into a single sorted array.

CSIEB0120 Algorithm Design & Analysis Divide-and-Conquer 11

Mergesort Example



CSIEB0120 Algorithm Design & Analysis Divide-and-Conquer 12

Mergesort: the merge Function

```
void merge(int h, int m, const keytype U[],
           const keytype V[], keytype S[]) {
  index i = 1 , j = 1, k = 1;
  while (i <= h && j <= m) {
    if (U[i] < V[j]) { S[k] = U[i]; i++; }
    else { S[k] = V[j]; j++; }
    k++;
  }
  if (i > h)
    copy V[j] ~ V[m] to S[k] ~ S[h+m];
  else
    copy U[i] ~ U[h] to S[k] ~ S[h+m];
}
```


Mergesort Algorithm

```
void mergesort(int n, keytype S[])
{
  if (n>1) {
    const int h=⌊n/2⌋, m = n - h;
    keytype U[1..h], V[1..m];
    copy S[1] ~ S[h] to U[1] ~ U[h];
    copy S[h+1] ~ S[n] to V[1] ~ V[m];
    mergesort(h, U);
    mergesort(m, V);
    merge(h, m, U, V, S);
  }
}
```

Mergesort Analysis

- Merges the two arrays U and V created by the recursive calls to mergesort
- **Input size**
  - h the number of items in U
  - m the number of items in V
- **Basic operation**: Comparison of U[I] to V[j]
- Worst case:
  - Loop exited with one index at exit point and the other at the exit point - 1

Worst-Case Analysis (Mergesort) 1

- W(n) = time_sort_U + time_sort_V + time_merge
- $W(n) = W(h) + W(m) + h+m-1$
- First analysis assumes n is a power of 2
  - $h = \lfloor n/2 \rfloor = n/2$
  - $m = n - h = n - n/2 = n/2$
  - $h + m = n/2 + n/2 = n$
- $W(n) = W(n/2) + W(n/2) + n - 1 = 2W(n/2) + n-1$
- $W(1) = 0$
- From B19 in Appendix B
  - $W(n)=n \lg n - (n-1) \in \Theta(n \lg n)$

Worst-Case Analysis (Mergesort) 2

- If n is not a power of 2
- $W(n)=W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n-1$
- From Theorem B4: $W(n) \in \Theta(n \lg n)$

Space Analysis (Mergesort)

- New arrays U and V will be created when *mergesort* is called.
- The total number of extra array items created is

$$n + \frac{n}{2} + \frac{n}{4} + \cdots = 2n$$

- In other words, the space complexity is $2n \in \Theta(n)$
- We may reduce the extra space to *n.* (Read textbook on Mergesort 2, Algorithm 2.4)
- But it is not possible to make mergesort algorithm to be an in-place sort.

Divide-and-Conquer Strategy

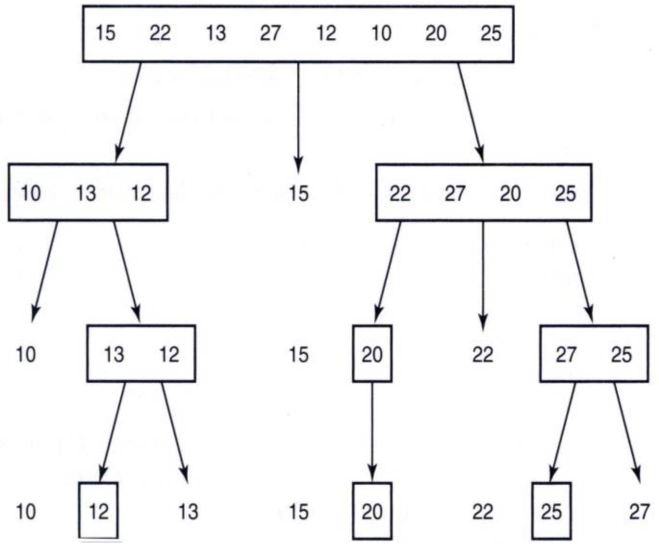
1. ***Divide*** an instance of a problem into one or more smaller instances.
2. ***Conquer*** (solve) each of the smaller instances. Unless a smaller instance is sufficiently small, use recursion to do this.
3. If necessary, ***combine*** the solutions to the smaller instances to obtain the solution to the original instance.

- One of the most widely used design stragety.

Quicksort

- Array recursively divided into two partitions and recursively sorted.
- Division based on a pivot.
- The pivot divides the two sub-arrays.
- All items < pivot placed in sub-array before pivot.
- All items >= pivot placed in sub-array after pivot.

Quicksort Example



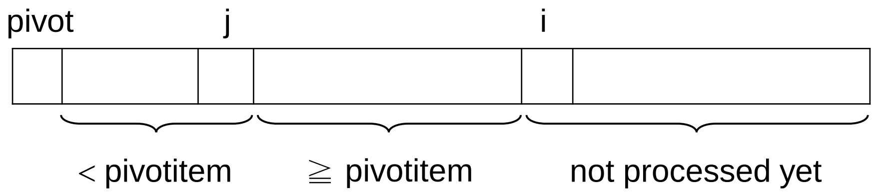
Quicksort Algorithm

```
void quicksort(index low, index high) {
  index pivotpoint;

  if (high > low){
    partition(low, high, pivotpoint);
    quicksort(low, pivotpoint - 1);
    quicksort(pivotpoint + 1, high);
  }
}
```

The `partition` Function

```
void partition(index low, index high,
               index& pivotpoint) {
  index i, j;
  keytype pivotitem;

  pivotitem = S[low];  // Choose 1st item as pivot
  j = low;
  for (i = low + 1; i <= high; i++)
    if (S[i] < pivotitem) {
      j++;
      swap S[i] and S[j];
    }
  pivotpoint = j;
  swap S[low] and S[pivotpoint];  // Place pivotitem
}
```


How does it work?



Every-Case Analysis (Partition)

- **Baisc operation**: Comparison of S[i] with pivotitem
- **Input size**: n = high – low + 1, no. items in subarray
- **Analysis**:
  - Every item except the first is compared.
  - T(n) = n - 1

Worst-Case Analysis (Quicksort) 1

- Occurs when the array is already sorted in non-decreasing order.
- The pivot(1^{st} item) is always the smallest.
- Array is repeatedly sorted into an empty sub-array which is less than the pivot and a sub-array of n-1 containing items greater than pivot.
- If there are k keys in the current sub-array, k-1 key comparisons are executed.

Worst-Case Analysis (Quicksort) 2

- $T(n)$ is used because analysis is for the every-case complexity for the class of instances already sorted in non-decreasing order
- $T(n)$ = time to sort left sub-array + time to sort right sub-array + time to partition
- $T(n) = T(0) + T(n-1) + n - 1$
- $T(n) = T(n - 1) + n - 1$ for $n > 0$
- $T(0) = 0$
- From B16: $T(n) = n(n-1)/2$

Worst-Case Analysis (Quicksort) 3

- From $T(n)$ above, we know that worst-case is at least $n(n-1)/2$.
- By induction, we can show it is the worst case
  - $W(n) = n(n-1)/2 \in \Theta(n^2)$

Average-Case Analysis (Quicksort)1

- Value of pivotpoint is equally likely to be any of the numbers from 1 to n.
- The probability for the pivot position to be the p-th is 1/n.
- The average time to sort if the pivot position is the p-th is [*A*(*p*-1) + *A*(*n*-*p*)] and the time to partition is *n*-1.
- Therefore, the average time complexity is

$$A(n) = \sum_{p=1}^{n} \frac{1}{n}[A(p-1) + A(n-p)] + n - 1$$

$$= \frac{2}{n}\sum_{p=1}^{n} A(p-1) + n - 1$$

Average-Case Analysis (Quicksort)2

$$nA(n) = 2\sum_{p=1}^{n} A(p-1) + n(n-1) \quad (1)$$

$$(n-1)A(n-1) = 2\sum_{p=1}^{n-1} A(p-1) + (n-1)(n-2) \quad (2)$$

$$nA(n) - (n-1)A(n-1) = 2A(n-1) + 2(n-1)$$

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

$$a_n = \frac{A(n)}{n+1} \qquad a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} \qquad n > 0$$

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} \quad a_{n-1} = a_{n-2} + \frac{2(n-2)}{(n-1)n} \quad a_2 = a_1 + \tfrac{1}{3} \quad a_1 = a_0 + 0$$

Average-Case Analysis (Quicksort)3

$$a_n = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)}$$
$$= 2\left(\sum_{i=1}^{n} \frac{1}{i+1} - \sum_{i=1}^{n} \frac{1}{i(i+1)}\right) \approx 2\ln n$$

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} = \ln n$$

$$A(n) \approx (n+1)2\ln n$$
$$= (n+1)2(\lg n)/(\lg e)$$
$$\approx 1.38(n+1)\lg n$$
$$\in \Theta(n \lg n)$$

Maximum Subarray Sum Problem

- **Problem**: Given an array A[1…n] of integers (positive and negative), compute a subarray A[i*… j*] with maximum sum.
- If $s(i, j)$ denotes the sum of the elements of a subarray A[i…j], that is $s(i, j) = \sum_{k=i}^{j} A[k]$
- We want to compute indices i* ≤ j* such that
  $$s(i^*, j^*) = \max\{s(i, j) \mid 1 \le i \le j \le n\}$$
- Example: 3 -4 $\underbrace{5\ \ -2\ \ -2\ \ 6\ \ -3\ \ 5}_{\text{max sum = 9}}$ -3 2

Brute Force Solution

- Compute the sum of every subarray and pick the maximum.
- Try every pair of indices i, j with $1 \le i \le j \le n$, and for each one compute s(i, j).
- Time complexity: $\Theta(n^3)$. (why?)
- With a little more care, can improve to $\Theta(n^2)$: can compute the sums of all the subarrays in time $\Theta(n^2)$.

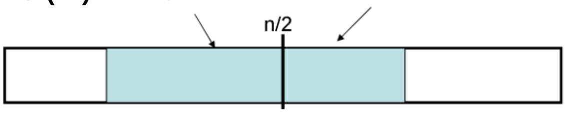
Brute Force Solution (improved)

- How to improve to $\Theta(n^2)$?
- Can compute the sums for all subarrays with same left end in O(n) time $\Rightarrow$ compute the sums of all the subarrays (there are n(n-1)/2 + n subarrays) in time $O(n^2)$.

```
for i = 1 to n {
  s(i,i)=A[i];
  for j = i+1 to n
    s(i,j) = s(i,j-1)+A[j];
}
```

Divide-and-Conquer Solution

- A subarray A[i*…j*] with maximum sum is
  - Either contained entirely in the left half , i.e. $j^* \le n/2$
  - Or contained entirely in the right half , i.e. $i^* \ge n/2$
  - Or cross the mid element: $i^* \le n/2 \le j^*$
- We can compute the best subarray of the first two types with recursive calls.
- The best subarray of the third type consists of the best subarray that ends at n/2 and the best subarray that starts at n/2. We can compute these in O(n) time.



Maximum Subarray Sum Algorithm

- It is a good exercise to write the pseudo code for the D&C solution.
- Also quite easy to convert the pseudo code into any programming language.
- Do it by yourself first w/o searching the Internet.
- Then search for a solution on the Internet.
- Compare your solution with the found one.

Worst-Case Analysis

- W(n) = 2W(n/2) + O(n)
- W(n) $\in$ O(n log n)
- It is possible to do even better: can compute the maximum subarray sum in O(n) time. (exercise)
- *Note: Not divide and conquer.*

Advantages of Divide-and-Conquer

- Solving difficult problems
- Algorithm efficiency: often help in the discovery of efficient algorithms.
- Parallelism: D&C algorithms can be easily executed on parallel machines.
- Memory access: D&C algorithms tend to make efficient use of memory caches.
- Widely applicable: D&C turns out to be a good strategy for many different problems. (Try to identify the D&C algorithms in Lecture 1)

When Not to Use Divide-and-Conquer

- Avoid using D&C in the following two cases:
  - An instance of size n is divided into two or more instances each almost of size n.
  - An instance of size n is divided into almost n instances of size n/c, where c is a constant.
- The first type leads to an exponential-time algorithm.
- The second type leads to an $n^{\Theta(\lg n)}$ algorithm.
- If Napoleon did this, he would have met his Waterloo much sooner.

Assignment 2: Divide-and-Conquer

1. Design and implement the improved and D&C version of the maximum subarray sum algorithm.
2. The Closest Pair of Points problem is to find the closest pair of points in a set of points in x-y plane. Design and implement a D&C algorithm to solve the problem.
3. Textbook exercises: 2.6, 2.7, 2.13

Due date: two weeks after previous due date of previous assignment.