CST207
DESIGN AND ANALYSIS OF ALGORITHMS

Lecture 4: Divide-and-Conquer and Sorting Algorithms 1

Lecturer: Dr. Yang Lu

Email: luyang@xmu.edu.my

Office: A1-432

Office hour: 2pm-4pm Mon & Thur

Outlines

- Binary Search
- Mergesort
- Quicksort
- Strassen’s Matrix Multiplication Algorithm
- Large Integer Multiplication
- Determining Threshold

Divide-and-Conquer

- The divide-and-conquer algorithm divides an instance of a problem into two or more smaller instances.
  - The smaller instance is the same problem as the original instance.
  - Assume that the smaller instance is easy to solve.
  - Combine solutions to the smaller instances to solve the original instance.
  - If the smaller instance is still difficult, divide again until it is easy.
- The divide-and-conquer is a *top-down* approach.
  - Recursion is usually adopted.

BINARY SEARCH

Review of Binary Search

Steps:

- If x equals the middle item, quit.
- Otherwise, compare x with the middle item.
  - If x is smaller, search the left subarray.
  - If x is greater, search the right subarray.

```
void binsearch(int n,
               const keytype S[ ],
               keytype x,
               index& location)
{
    index low, high, mid;

    low = 1; high = n;
    location = 0;
    while (low <= high && location == 0){
        mid = ⌊(low + high) / 2⌋;
        if (x == S[mid])
            location = mid;
        else if (x < S[mid])
            high = mid − 1;
        else
            low = mid + 1
    }
}
```

Searching subarray by moving the index bound

Non-recursive binary search

Binary Search with Divide-and-Conquer

- Steps:
  - If x equals the middle item, quit. Otherwise:
    1. *Divide* the array into two subarrays about half as large. If x is smaller than the middle item, return the result from the left subarray. Otherwise, return the result from the right subarray.
    2. *Conquer* (solve) the subarray by determining whether x is in that subarray. Unless the subarray is sufficiently small, use recursion to do this.
    3. *Obtain* the solution to the array from the solution to the subarray.
- The instance is broken down into only one smaller instance, so there is no combination of outputs.
  - The solution to the original instance is simply the solution to the smaller instance.

Design Divide-and-Conquer Algorithms

- When developing a recursive algorithm with divide-and-conquer, we need to
  - Develop a way to obtain the solution to an instance from the solution to one or more smaller instances.
  - Determine the terminal condition(s) that the smaller instance(s) is (are) approaching.
  - Determine the solution in the case of the terminal condition(s).
- Not like the non-recursive version, n, S and x are not parameters to the recursive function.
  - They ramain unchanged in each recursive call.
  - Only pass the changing variables to a recursive function.

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

    if (low > high) // can't find condition
        return 0;
    else{
        mid = ⌊(low + high) / 2⌋;
        if (x == S[mid]) // find condition
            return mid
        else if (x < S[mid])
            return binsearch_recursive(low, mid - 1);
        else
            return binsearch_recursive(mid + 1, high);
    }
}
```

Worst-Case Time Complexity of Binary Search

- The binary search doesn't have an every-case time complexity.
- The recursive equation for the worst-case is:

$$W(n) = W(n/2) + 1.$$

  - $W(n/2)$ is the number of comparisons in recursive call.
  - 1 is the comparison at top level.
- By the master method case 2, we have $f(n) = 1 \in \Theta(1) = \Theta\left(n^{\log_2 1}\right)$.
- Therefore, $W(n) \in \Theta(\lg n)$.

MERGESORT

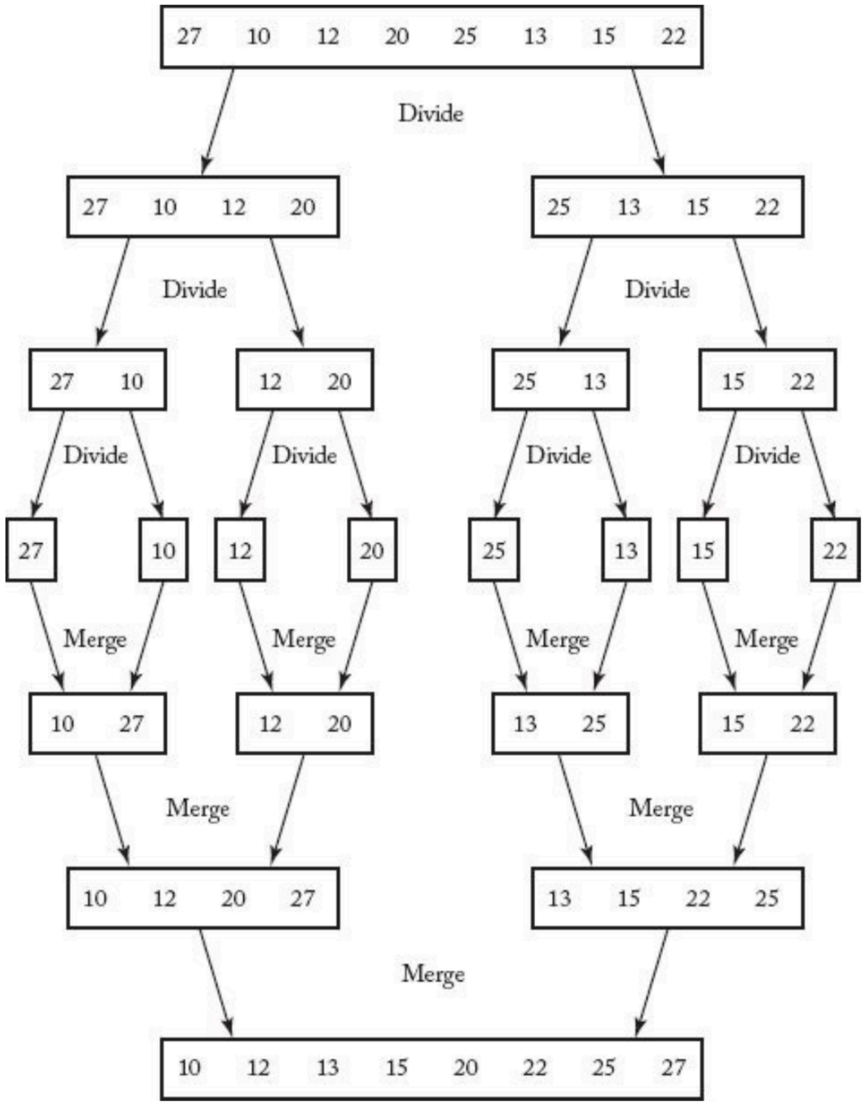
Sorting Algorithm

- A sorting algorithm is an algorithm that puts items of a list in a certain order.
- Efficient sorting is important for optimizing the efficiency of other algorithms (such as search and merge algorithms) that require input data to be in sorted lists.
- The output of any sorting algorithm must satisfy two conditions:
  1. The output is in nondecreasing order (each item is no smaller than the previous item);
  2. The output is a permutation (a reordering, yet retaining all of the original items) of the input.

Mergesort

- Combine two sorted arrays into one sorted array.
- Given an array with n items, Mergesort involves the following steps:

1. *Divide* the array into two subarrays each with $n/2$ items.
2. *Conquer* (solve) each subarray by sorting it. Unless the array is sufficiently small, use recursion to do this.
3. *Combine* the solutions to the subarrays by merging them into a single sorted array.



Example of Mergesort steps

Image source: Figure 2.2, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Mergesort Visualized Demo

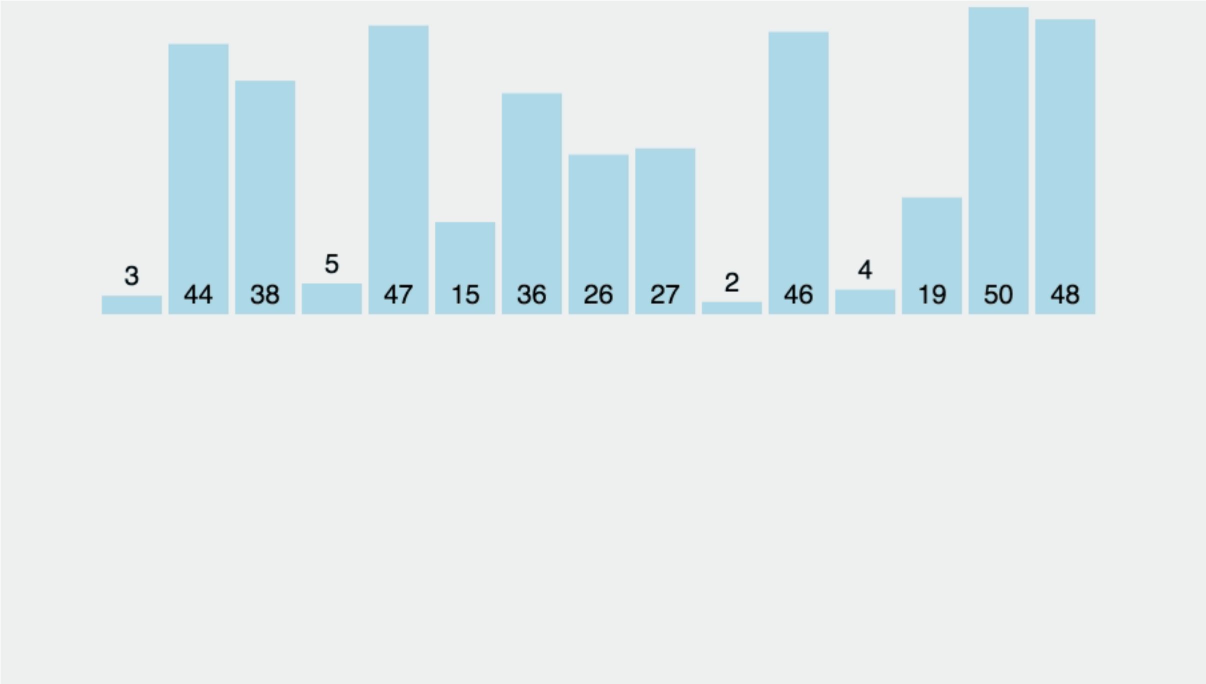


Image source: https://thumbs.gfycat.com/ZealousAdolescentBellsnake-size_restricted.gif

Pseudocode of Mergesort

```
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] through S[h] to U[1] through U[h];
        copy S[h+1] through S[n] to V[1] through V[m];
        mergesort(h, U);
        mergesort(m, V);
        merge(h, m, U, V, S);
    }
}
```

```
void merge (int h, int m, const keytype U[],
                          const keytype V[],
                                keytype S[])
{
    index i, j, k;

    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] through V[m] to S[k] through S[h+m];
    else
        copy U[i] through U[h] to S[k] through S[h+m];
}
```

Merging Process

index	U (index i, length h)	V (index j, length m)	S (index k, length $h+m$)
$k=1, i=1, j=1$	10 12 20 27 30	13 15 22 25	10
$k=2, i=2, j=1$	10 12 20 27 30	13 15 22 25	10 12
$k=3, i=3, j=1$	10 12 20 27 30	13 15 22 25	10 12 13
$k=4, i=3, j=2$	10 12 20 27 30	13 15 22 25	10 12 13 15
$k=5, i=3, j=3$	10 12 20 27 30	13 15 22 25	10 12 13 15 20
$k=6, i=4, j=3$	10 12 20 27 30	13 15 22 25	10 12 13 15 20 22
$k=7, i=4, j=4$	10 12 20 27 30	13 15 22 25	10 12 13 15 20 22 25
$k=8, i=5, j=5$	10 12 20 27 30	13 15 22 25	10 12 13 15 20 22 25 27 30

while loop terminates when $j > m$

$i <= h$ thus copy all the rest of U to the tail of S

Worst-Case Time Complexity of Merge

- For sorting algorithm, the basic operation is comparison.
  - Assignment and item exchange is not counted.
- All of the items in two arrays are compared.
- Totally $h + m - 1$ comparisons.
  - Add each item into S after comparison except the last one.

```
void merge (int h, int m, const keytype U[],
                          const keytype V[],
                                keytype S[])
{
    index i, j, k;

    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] through V[m] to S[k] through S[h+m];
    else
        copy U[i] through U[h] to S[k] through S[h+m];
}
```

Worst-Case Time Complexity of Mergesort

- The recursive equation:

$$W(n) = \underbrace{W(h)}_{\text{time to sort } U} + \underbrace{W(m)}_{\text{time to sort } V} + \underbrace{h + m - 1}_{\text{time to merge}}.$$

- By the setting of $h = \lfloor n/2 \rfloor$ and $m = n - h$, we have:

$$W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1.$$

- By the master method case 2, we have $f(n) = n \in \Theta(n) = \Theta\left(n^{\log_2 2}\right)$.
- Therefore, $W(n) \in \Theta(n \lg n)$.
- Best-case and Average-case complexity for Mergesort is also $\Theta(n \lg n)$. Why?

Space Complexity

- An *in-place sort* is a sorting algorithm that does not use any extra space beyond that needed to store the input.
- The previous version of Mergesort is not an in-place sort because it uses the arrays U and V besides the input array S.
- New arrays U and V will be created each time `mergesort` is called.
- The total number of extra array items is $n + n/2 + n/4 + \cdots = 2n$.
  - Exercise: the space usage can be improved to n. How?

The Divide-and-Conquer Approach

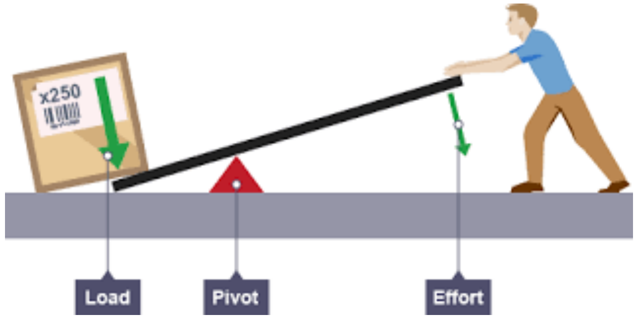
- Now, you should now better understand the following general description of this approach.
- The ***divide-and-conquer*** design strategy involves the following steps:
  1. *Divide* an instance of a problem into one or more smaller instances.
  2. *Conquer* (solve) each of the smaller instances. Unless 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.
- Why we say "if necessary" in step 3 is that in algorithms such as `binsearch_recursive`, the instance is reduced to just one smaller instance, so there is no need to combine solutions.

QUICKSORT

Quicksort

- Quicksort is developed by British computer scientist Tony Hoare in 1962.
- You can know the main property of Quicksort by its name – quick!
- When implemented well, it can be about two or three times faster than Mergesort.

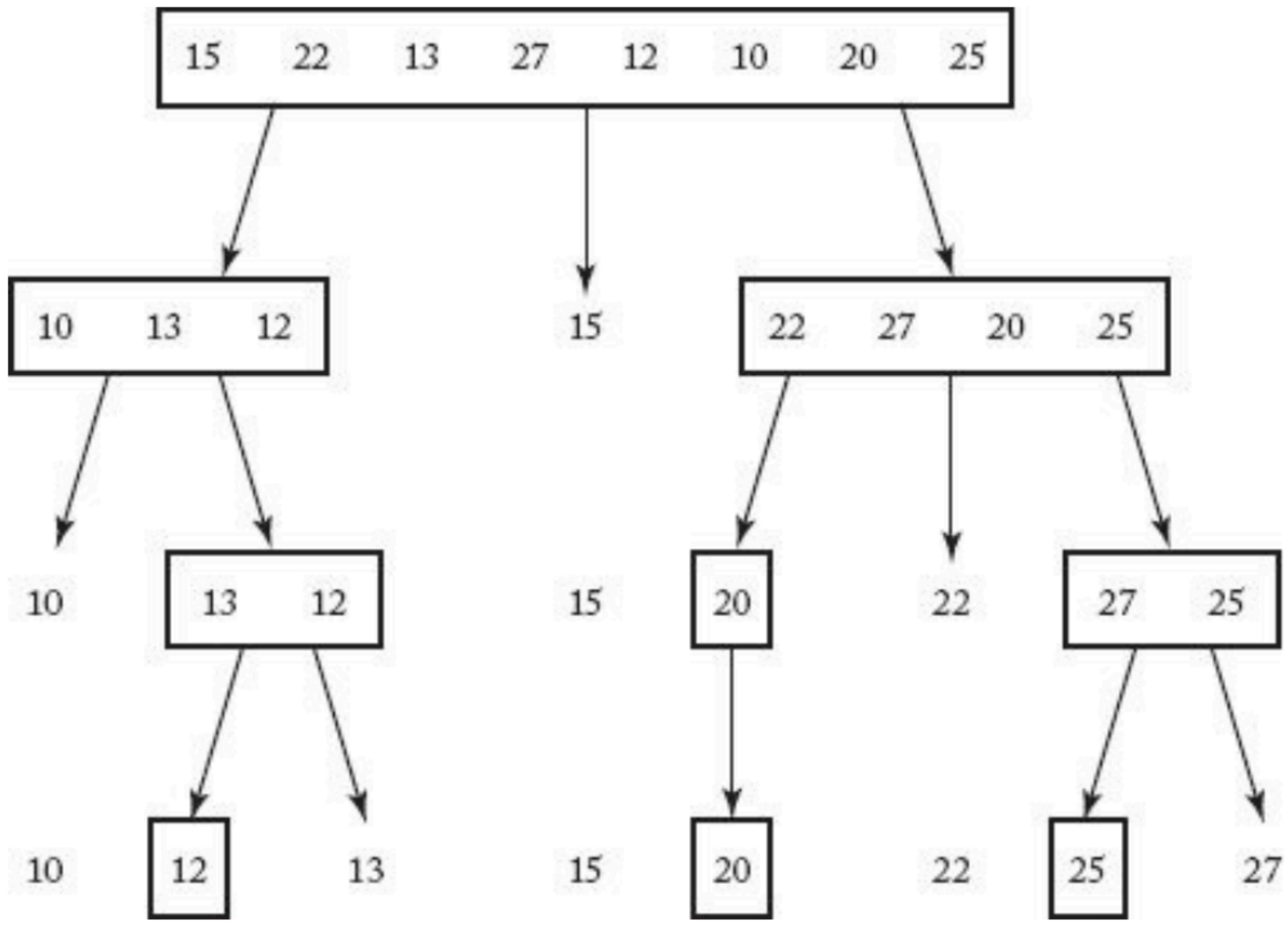
Quicksort



Steps:

- Randomly select a pivot item (conventional use the first item).
- Put all the items smaller than the pivot item on its left, and all the items greater than the pivot item on its right.
- Recursively sort the left subarray and right subarray.

Image source: http://mrtremblaycambridge.weebly.com/p15-turning-on-a-pivot.html



Example of Quicksort steps

Image source: Figure 2.3, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Quicksort Visualized Demo

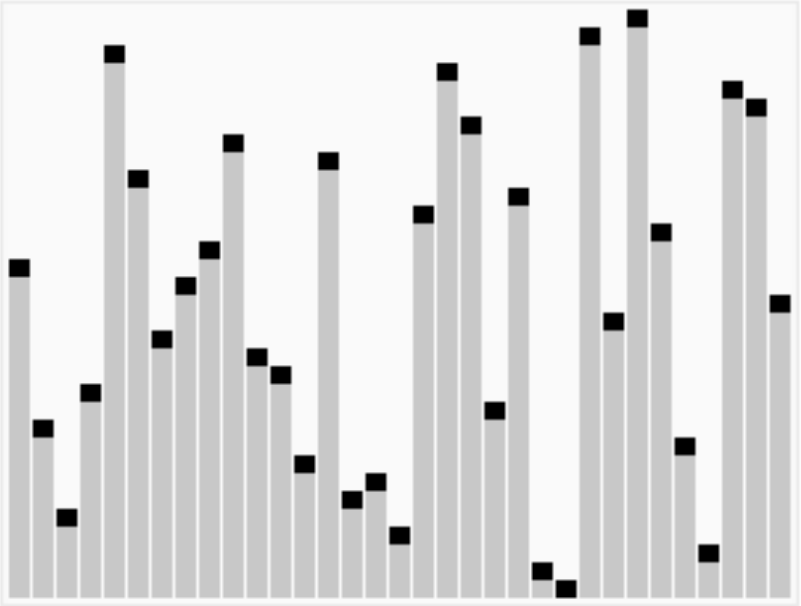


Image source: https://en.wikipedia.org/wiki/Quicksort

Pseudocode of Quicksort

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

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

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

    pivotitem = S[low]; // choose first item as the pivot
    j = low; // the index of the last item smaller than the pivot
    for (i=low+1; i<=high; i++)
        if (S[i] < pivotitem){
            j++;
            exchange S[i] and S[j];
        }
    pivotpoint = j;
    exchange S[low] and S[pivotpoint];
}
```

Partition Process

pivot

i	j	$S[1]$	$S[2]$	$S[3]$	$S[4]$	$S[5]$	$S[6]$	$S[7]$	$S[8]$	
-	-	15	22	13	27	12	10	20	25	initial
2	1	15	22	13	27	12	10	20	25	
3	2	15	22	13	27	12	10	20	25	
4	2	15	13	22	27	12	10	20	25	
5	3	15	13	22	27	12	10	20	25	
6	4	15	13	12	27	22	10	20	25	
7	4	15	13	12	10	22	27	20	25	
8	4	15	13	12	10	22	27	20	25	
-	4	10	13	12	15	22	27	20	25	finish

Every-Case Time Complexity of Partition

- Every item is compared to the pivot except itself.

$$T(n) = n - 1$$

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

    pivotitem = S[low]; // choose first item as the pivot
    j = low; // the index of the last item smaller than the pivot
    for (i=low+1; i<=high; i++)
        if (S[i] < pivotitem){
            j++;
            exchange S[i] and S[j];
        }
    pivotpoint = j;
    exchange S[low] and S[pivotpoint];
}
```

Worst-Case Time Complexity of Quicksort

- The array is already in nondecreasing order.
- In each recursion step, the pivot item is always the smallest item.
  - No item is put on the left of the pivot item.
  - Thus, n items are divided into 1 and $n-1$ items.
- Recursion equation:

$$W(n) = W(0) + W(n-1) + n - 1$$

- Using recursion tree, we can easily get $W(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$.
  - Exercise: Draw the recursion tree and use substitution method to prove it.

Worst-Case Time Complexity of Quicksort

- The closer the input array is to being sorted, the closer we are to the worst-case performance.
  - Because the pivot can't fairly separate two subarrays.
  - Recursion loses it power.
- How to wisely choose the pivot?
  - Random.
  - Median of $S[low]$, $S[mid]$, and $S[high]$. Safe to avoid the worst-case but more comparisons are needed.

Average-Case Time Complexity of Quicksort

- The worst-case of Quicksort is no faster than exchange sort (also $\Theta(n^2)$) and slower than Mergesort ($\Theta(n \log n)$).
- How dare it name itself “quick”?
  - The average-case behavior earns its name!

Average-Case Time Complexity of Quicksort

- We can’t assume that the input array is uniformly distributed from the $n!$ permutations.
- To analyze the average-case time complexity, we can add randomization.
  - Randomly permutate the input array.
  - Randomly choose the pivot item.

Average-Case Time Complexity of Quicksort

- By randomization, now the probability of pivot being any item in the array is $1/n$.

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

$$A(n) = \frac{2}{n}\sum_{p=1}^{n} A(p-1) + n - 1 \text{ (try to prove this step)}$$

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

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

Average-Case Time Complexity of Quicksort

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

- Let $a_n = \frac{A(n)}{n+1}$,

Harmonic series

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} \approx 2\sum_{i=1}^{n} \frac{1}{i} \approx 2\ln n\,.$$

- Therefore, $A(n) \approx (n+1)2\ln n = (n+1)2\ln 2\lg n \approx 1.38(n+1)\lg n \in \Theta(n\lg n)$.

Space Complexity

- Quicksort looks like an in-place sort.
  - No extra arrays are created for storing the temporary values.
- The index of the pivot item is created in each recursion call.
  - That takes storage of $\Theta(\log n)$, which equals to the stack depth of recursion.

STRASSEN'S MATRIX MULTIPLICATION ALGORITHM

Recall Matrix Multiplication

- Matrix multiplication
  - Problem: determine the product of two $n \times n$ matrices.
  - Inputs: a positive integer n, two-dimensional arrays of numbers A and B, each of which has both its rows and columns indexed from 1 to n.
  - Outputs: a two-dimensional array of numbers C, which has both its rows and columns indexed from 1 to n, containing the product of A and B.
- Recall that if we have two 2×2 matrices

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \text{ and } B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix},$$

their product $C = A \times B$ is given by

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j}.$$

```
void matrixmult(int n,
                const number A[][],
                const number B[][],
                      number C[][])
{
   index i, j, k;

   for (i=1; i<=n; i++)
      for (j=1; j<=n; j++){
         C[i][j] = 0;
         for (k=1; k<=n; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
            //C[i][j] += A[i][k] * B[k][j];
      }
}
```

Average-Case Time Complexity of Matrix Multiplication

- It can be easily shown that the time complexity is $T(n) = n^3$.
  - The number of multiplication is n^3.
  - The number of addition is $n^2(n-1) = n^3 - n^2$.
    - In the most inner loop, adding n items only needs $n-1$ times addition.
- Strassen proposed a method to make the complexity of matrix multiplication better than n^3.

Strassen's Matrix Multiplication Algorithm

- Suppose we want to product C of two 2×2 matrices, A and B, That is,

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

- Strassen determined that if we let

$$\begin{aligned}
m_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
m_2 &= (a_{21} + a_{22})b_{11} \\
m_3 &= a_{11}(b_{12} - b_{22}) \\
m_4 &= a_{22}(b_{21} - b_{11}) \\
m_5 &= (a_{11} + a_{12})b_{22} \\
m_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\
m_7 &= (a_{12} - a_{22})(b_{21} + b_{22})
\end{aligned}$$

the product C is given by

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

Strassen's Matrix Multiplication Algorithm

- To multiply two 2×2 matrices, Strassen's method requires 7 multiplications and 18 additions/subtractions.
  - The standard method requires 8 multiplications and 4 additions/subtractions.
  - Use 14 more additions/subtractions to save 1 multiplication. It that worthy?
- Obviouly, it is not worthy in terms of number multiplication and additions/ subtractions.
  - However, it is very worthy in terms of matrix multiplication and additions/subtractions.

Strassen's Matrix Multiplication Algorithm

- The divided submatrices also follow Strassen's formula:

$$\left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right] = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \times \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right]$$

$n/2$ $n/2$

- We use recursion and Strassen's formula to calculate the matrix multiplication until n is sufficiently small.
- When n is not a power of 2, one simple modification is to add sufficient numbers of columns and rows of 0s.

Pseudocode of Strassen’s Matrix Multiplication Algorithm

```
void strassen (int n,
               matrix A,
               matrix B,
               matrix& C)
{
    if (n <= threshold)
        compute C=AxB with the standard algorithm;
    else{
        partition A into four submatrices A11, A12, A21, A22;
        partition B into four submatrices B11, B12, B21, B22;
        //compute C=AxB with Strassens method;
        strassen(n/2, A11+A22, B11+B22, M1);
        strassen(n/2, A21+A22, B11, M2);
        ...
        construct C by M1...M7;
    }
}
```

Every-Case Time Complexity Analysis of Strassen's Matrix Multiplication Algorithm

- In each recursive step, we actually only do addition/subtraction. The multiplication is passed to the next recursion step.
- We need 18 times addition/subtraction for a matrix with $(n/2)^2$ items.
- Recursion equation:

$$T(n) = 7T(n/2) + 18(n/2)^2$$

- Use the master method case 1, $f(n) = \frac{18}{4}n^2 \in O(n^{\log_2 7-\epsilon}) \approx O(n^{2.81-\epsilon})$ for $\epsilon \approx 0.81$.
- Therefore, we have $T(n) \in \Theta(n^{2.81})$.

LARGE INTEGER MULTIPLICATION

Arithmetic with Large Integers

- Suppose that we need to do arithmetic operations on integers whose size exceeds the computer's hardware capability of representing integers.
  - On 32-bit and 64-bit systems, an integer in programming language C is representaed by 4 bytes
    - -2,147,483,647 ~ 2,147,483,647.
- How to do arithmetic for those large integers?

Representation of Large Integers

- A straightforward way is to use an array, in which each slot stores one digit.

  Integer 53241 fills in the array with size 5:

5	3	2	4	1

- For addition and subtraction, it's easy to write linear-time algorithms.
  - You know how addition and subtraction work at the first grade of your primary school.
- For multiplication, division and modulo with exponential based on 10, linear-time algorithm is also easy.
  - Just add zeros or take out some bits.
- For multiplication, it's also not difficult to write a quadratic algorithms.
  - Can we use divide-and-conquer to make it faster?

Large Integer Multiplication

- Let n the number of digits and $m = \lfloor n/2 \rfloor$. If we have two n-digit integers

$$u = x \times 10^m + y$$
$$v = w \times 10^m + z$$

their product is given by

$$\begin{aligned} uv &= (x \times 10^m + y)(w \times 10^m + z) \\ &= xw \times 10^{2m} + (xz + wy) \times 10^m + yz. \end{aligned}$$

- There are 4 multiplications and a few linear operations.

Pseudocode of Large Integer Multiplication

```
large_integer prod (large_integer u, large_integer v)
{
    large_integer x, y, w, z;
    int n, m;

    n = maximum(number of digits in u, number of digits in v);
    if (u == 0 || v == 0)
        return 0;
    else if (n <= threshold)
        return u * v obtained in the usual way;
    else{
        m = ⌊n / 2⌋
        x = u div 10^m; y = u mod 10^m;
        w = v div 10^m; z = v mod 10^m;
        return prod(x, w) * 10^2m + (prod(x, z) + prod(w, y)) * 10^m + prod(y, z);
    }
}
```

Worst-Case Time Complexity of Large Integer Multiplication

- No digits equal to 0.
  - Equal to 0 leads early quit from recursion, otherwise pass into the next recursion step.
- Recursive equation:

$$W(n) = 4W(n/2) + cn$$

- Use the master method case 1, $W(n) \in \Theta(n^2)$.
- It is still quadratic. Why?

Improvement of Large Integer Multiplication

- We decompose the problem of n into 4 $n/2$ subproblems.
- If we can decrease 4 to 3, by the master method we get $W(n) \in \Theta(n^{\log_2 3})$.
- Now, we need to calculate

$$xw, xz + yw, yz$$

- If instead we set

$$r = (x + y)(w + z) = xw + (xz + yw) + yz$$

we have

$$xz + yw = r - xw - yz$$

- Then, we only need to calculate

$$r, xw, yz$$

DETERMINING THRESHOLD

Determining Thresholds

- For matrix multiplication and large integer multiplication, when n is small, using standard algorithm will be even faster.
- For Mergesort, using recursive method on small array will also be slower than quadratic sorting algorithm like exchange sort.
- How to determine the threshold?

Determining Thresholds

- If we have the recursive equation of Mergesort measured by computational time:

$$W(n) = 32n \lg n \ \ \mu s$$

and exchange sort takes

$$W(n) = \frac{n(n-1)}{2} \mu s$$

- We can compare and get the threshold:

$$\frac{n(n-1)}{2} < 32n \lg n$$
$$n < 591.$$

When Not to Use Divide-and-Conquer

- An instance of size n is divided into two or more instances each almost of size n.
  - nth Fibonacci term: $T(n) = T(n-1) + T(n-2) + 1$.
  - Worst-case Quicksort is also not acceptable: $T(n) = T(n-1) + n - 1$.
- An instance of size n is divided into almost n instances of size n/c, where c is a constant.
  - E.g. $T(n) = T(n/2) + T(n/2) + \cdots + T(n/2)$.

Conclusion

After this lecture, you should know:

- What is the key idea of divide-and-conquer.
- How to divide a big problem instance into several small instances.
- How to use recursion to design a divide-and-conquer algorithm.
- How Mergesort and Quicksort work and what are their complexity.

Thank you!

- Any question?
- Don’t hesitate to send email to me for asking questions and discussion. ☺