Divide and Conquer Strategy

- Algorithm design is more an art, less so a science.
- There are a few useful strategies, but no guarantee to succeed.
- We will discuss: Divide and Conquer, Greedy, Dynamic Programming.
- For each of them, we will discuss a few examples, and try to identify common schemes.

Divide and Conquer

- Divide the problem into smaller subproblems (of the same type).
- Solve each subproblem (usually by recursive calls).
- Combine the solutions of the subproblems into the solution of the original problem.
Merge Sort

MergeSort

Input: an array $A[1..n]$
Output: Sort $A$ into increasing order.

- Use a recursive function $\text{MergeSort}(A, p, r)$.
- In main program, we call $\text{MergeSort}(A, 1, n)$.
Merge Sort

\textbf{MergeSort}(A, p, r)

1: if \((p < r)\) then
2: \hspace{1em} \(q = (p + r)/2\)
3: \hspace{1em} \text{MergeSort}(A, p, q)
4: \hspace{1em} \text{MergeSort}(A, q + 1, r)
5: \hspace{1em} \text{Merge}(A, p, q, r)
6: else
7: \hspace{1em} do nothing
8: end if

- Divide \(A[p..r]\) into two sub-arrays of equal size.
- Sort each sub-array by recursive call.
- \textbf{Merge}(A, p, q, r) is a procedure that, assuming \(A[p..q]\) and \(A[q + 1..r]\) are sorted, merge them into sorted \(A[p..r]\)
- It can be done in \(\Theta(k)\) time where \(k = r - p\) is the number of elements to be sorted.
Analysis of MergeSort

Let $T(n)$ be the runtime function of MergeSort($A[1..n]$). Then:

$$T(n) = \begin{cases} 
O(1) & \text{if } n = 1 \\
2T(n/2) + \Theta(n) & \text{if } n > 1
\end{cases}$$

- If $n = 1$, MergeSort does nothing, hence $O(1)$ time.
- Otherwise, we make 2 recursive calls. The input size of each is $n/2$. Hence the runtime $2T(n/2)$.
- $\Theta(n)$ is the time needed by $\text{Merge}(A, p, q, r)$ and all other processing.
Master Theorem

For DaC algorithms, the runtime function often satisfies:

\[
T(n) = \begin{cases} 
O(1) & \text{if } n \leq n_0 \\
 aT(n/b) + \Theta(f(n)) & \text{if } n > n_0 
\end{cases}
\]

- If \( n \leq n_0 \) (\( n_0 \) is a small constant), we solve the problem directly without recursive calls. Since the input size is fixed (bounded by \( n_0 \)), it takes \( O(1) \) time.
- We make \( a \) recursive calls. The input size of each is \( n/b \). Hence the runtime \( T(n/b) \).
- \( \Theta(f(n)) \) is the time needed by all other processing.
- \( T(n) =? \)
Master Theorem

Master Theorem (Theorem 4.1, Cormen’s book.)

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and $af(n/b) \leq cf(n)$ for some $c < 1$ for sufficiently large $n$, then $T(n) = \Theta(f(n))$.

Example: MergeSort

We have $a = 2$, $b = 2$, hence $\log_b a = \log_2 2 = 1$. So $f(n) = \Theta(n^1) = \Theta(n^{\log_b a})$.

By statement (2), $T(n) = \Theta(n^{\log_2 2 \log n}) = \Theta(n \log n)$. 
Binary Search

- Input: Sorted array $A[1..n]$ and a number $x$
- Output: Find $i$ such that $A[i] = x$, if no such $i$ exists, output “no”.

We use a rec function $\text{BinarySearch}(A, p, r, x)$ that searches $x$ in $A[p..r]$.

$\text{BinarySearch}(A, p, r, x)$

1: if $p = r$ then
2: \hspace{1em} if $A[p] = x$ return $p$
3: \hspace{1em} if $A[p] \neq x$ return “no”
4: else
5: \hspace{2em} $q = (p + r)/2$
6: \hspace{2em} if $A[q] = x$ return $q$
7: \hspace{2em} if $A[q] > x$ call $\text{BinarySearch}(A, p, q - 1, x)$
8: \hspace{2em} if $A[q] < x$ call $\text{BinarySearch}(A, q + 1, r, x)$
9: end if
Analysis of Binary Search

- If $n = p - r + 1 = 1$, it takes $O(1)$ time.
- If not, we make at most one recursive call, with size $n/2$.
- All other processing take $f(n) = \Theta(1)$ time.
- So $a = 1$, $b = 2$ and $f(n) = \Theta(n^0)$ time.
  - Since $\log_b a = \log_2 1 = 0$, $f(n) = \Theta(n^{\log_b a})$.
- Hence $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(\log n)$. 
A function makes 4 recursive calls, each with size $n/2$. Other processing takes $f(n) = \Theta(n^3)$ time.

$$T(n) = 4T(n/2) + \Theta(n^3)$$

We have $a = 4$, $b = 2$. So $\log_b a = \log_2 4 = 2$.

$$f(n) = n^3 = \Theta(n^{\log_b a + 1}) = \Omega(n^{\log_b a + 0.5}) .$$

This is the case 3 of Master Theorem. We need to check the 2nd condition:

$$a \cdot f(n/b) = 4 \left( \frac{n}{2} \right)^3 = \frac{4}{8} n^3 = \frac{1}{2} \cdot f(n)$$

If we let $c = 1/2 < 1$, we have: $a \cdot f(n/b) \leq c \cdot f(n)$. Hence by case 3, $T(n) = \Theta(f(n)) = \Theta(n^3)$. 

Master Theorem

If \( f(n) \) has the form \( f(n) = \Theta(n^k) \) for some \( k \geq 0 \), We have the following:

### A simpler version of Master Theorem

\[
T(n) = \begin{cases} 
O(1) & \text{if } n \leq n_0 \\
aT(n/b) + \Theta(n^k) & \text{if } n > n_0
\end{cases}
\]

1. If \( k < \log_b a \), then \( T(n) = \Theta(n^{\log_b a}) \).
2. If \( k = \log_b a \), then \( T(n) = \Theta(n^k \log n) \).
3. If \( k > \log_b a \), then \( T(n) = \Theta(n^k) \).

Only the case 3 is different. In this case, we need to check the 2\(^{nd}\) condition. Because \( k > \log_b a \), \( b^k > a \) and \( a/b^k < 1 \):

\[
a \cdot f(n/b) = a \cdot \left(\frac{n}{b}\right)^k = a \cdot \frac{a}{b^k} \cdot f(n) = c \cdot f(n)
\]

where \( c = \frac{a}{b^k} < 1 \), as needed.
How to understand/memorize Master Theorem?

The cost of a DaC algorithm can be divided into two parts:

1. The total cost of all recursive calls is $\Theta(n^{\log_b a})$.
2. The total cost of all other processing is $\Theta(f(n))$.

If (1) > (2), (1) dominates the total cost: $T(n) = \Theta(n^{\log_b a})$.

If (1) < (2), (2) dominates the total cost: $T(n) = \Theta(f(n))$.

If (1) = (2), the cost of two parts are about the same, somehow we have an extra factor $\log n$.

The proof of Master Theorem is given in textbook.

We’ll illustrate two examples in class.
For some simple cases, Master Theorem does not work.

Example

\[ T(n) = 2T(n/2) + \Theta(n \log n) \]

- \( a = 2, b = 2, \log_a b = \log_2 2 = 1 \)
- \( f(n) = n^1 \log n = n^{\log_b a} \log n \)
- \( f(n) = \Omega(n), \text{ but } f(n) \neq \Omega(n^{1+\epsilon}) \text{ for any } \epsilon > 0. \)
- Master Theorem does not apply.

Theorem

If \( T(n) = aT(n/b) + f(n), \) where \( f(n) = \Theta(n^{\log_b a} (\log n)^k) \), then
\[ T(n) = \Theta(n^{\log_b a} (\log n)^{k+1}). \]

In the above example, \( T(n) = \Theta(n \log^2 n) \)
Matrix Multiplication

- **Matrix multiplication** is a basic operation in **Linear Algebra**.

Let \( A = (a_{ij})_{1 \leq i,j \leq n} \) and \( B = (b_{ij})_{1 \leq i,j \leq n} \) be two \( n \times n \) matrices.

**Definition**

**Matrix Addition**

\[
C = (c_{ij})_{1 \leq i,j \leq n} = A + B
\]

is defined by \( c_{ij} = a_{ij} + b_{ij} \) for \( 1 \leq i,j \leq n \).

- We need to calculate \( n^2 \) entries in \( C \).
- Each entry takes \( O(1) \) time.
- So matrix addition takes \( \Theta(n^2) \) time.
Matrix Multiplication

Definition
Matrix Multiplication

\[ C = (c_{ij})_{1 \leq i, j \leq n} = A \times B \]

is defined by: for \( 1 \leq i, j \leq n \),

\[ c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj} \]

Example

\[
A = \begin{pmatrix}
4 & -1 \\
2 & 1
\end{pmatrix}, \quad B = \begin{pmatrix}
3 & 1 \\
0 & -2
\end{pmatrix}
\]

\[
C = \begin{pmatrix}
4 \cdot 3 + (-1) \cdot 0 & 4 \cdot 1 + (-1) \cdot (-2) \\
2 \cdot 3 + 1 \cdot 0 & 2 \cdot 1 + 1 \cdot (-2)
\end{pmatrix} = \begin{pmatrix}
12 & 6 \\
6 & 0
\end{pmatrix}
\]
Matrix Matrix Multiplication

MatrixMultiply\((A, B)\)

1: \textbf{for} \(i = 1\) \textbf{to} \(n\) \textbf{do}
2: \hspace{1em} \textbf{for} \(j = 1\) \textbf{to} \(n\) \textbf{do}
3: \hspace{2em} \(c_{ij} = 0\)
4: \hspace{1em} \textbf{for} \(k = 1\) \textbf{to} \(n\) \textbf{do}
5: \hspace{2em} \(c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}\)
6: \hspace{1em} \textbf{end for}
7: \hspace{1em} \textbf{end for}
8: \textbf{end for}

- This algorithm clearly takes \(\Theta(n^3)\) time.
- Since MM is an important operation, can we do better than this?
Strassen’s MM Algorithm

- Try DaC. Assume $n = 2^k$ is a power of 2. If not, we can pad $A$ and $B$ by extra 0’s so that this is true.
- Divide each $A, B, C$ into 4 sub-matrices, with size $n/2 \times n/2$.

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix} \quad C = \begin{pmatrix} r & s \\ t & u \end{pmatrix}$$

It can be shown:

$$r = a \times e + b \times g \quad s = a \times f + b \times h$$
$$t = c \times e + d \times g \quad u = c \times f + d \times h$$
Strassen’s MM Algorithm

- If \( n = 1 \), solve the problem directly (in \( O(1) \) time).
- If not, divide \( A \) and \( B \) into 4 sub-matrices. (This only involves manipulation of indices, no actual division is needed. It actually takes no time).
- Solve sub-problems using recursive calls. Each \( \times \) is a recursive call. There are 8 of them.
- Use the above formulas to obtain \( A \times B \). This involves 4 matrix additions of size \( n/2 \times n/2 \). It takes \( \Theta(n^2) \) time.

\[
T(n) = 8T(n/2) + \Theta(n^2)
\]

Thus: \( a = 8 \), \( b = 2 \) and \( k = 2 \). Since \( \log_b a = \log_2 8 = 3 > 2 \), we get:

\[
T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3).
\]

No better than the simple \( \Theta(n^3) \) algorithm.
Strassen’s MM Algorithm

The problem: We are making **too many recursive calls!**
To improve, we must reduce the number of recursive calls.

\[
\begin{align*}
A_1 &= a \\
A_2 &= a + b \\
A_3 &= c + d \\
A_4 &= d \\
A_5 &= a + d \\
A_6 &= b - d \\
A_7 &= a - c \\
r &= P_5 + P_4 - P_2 + P_6 \\
s &= P_1 + P_2 \\
t &= P_3 + P_4 \\
u &= P_5 + P_1 - P_3 - P_7 \\
B_1 &= f - h \\
B_2 &= h \\
B_3 &= e \\
B_4 &= g - e \\
B_5 &= e + h \\
B_6 &= g + h \\
B_7 &= e + f \\
P_1 &= A_1 \times B_1 \\
P_2 &= A_2 \times B_2 \\
P_3 &= A_3 \times B_3 \\
P_4 &= A_4 \times B_4 \\
P_5 &= A_5 \times B_5 \\
P_6 &= A_6 \times B_6 \\
P_7 &= A_7 \times B_7 \\
\end{align*}
\]

We need 7 recursive calls, and a total 18 additions/subtractions of \(n/2 \times n/2\) sub-matrices.
Strassen’s MM Algorithm

\[ T(n) = 7T(n/2) + \Theta(n^2) \]

- \( a = 7, b = 2, k = 2 \). So \( \log_a b = \log_2 7 \approx 2.81 > k \).
- Hence \( T(n) = \Theta(n^{2.81}) \).
- For small \( n \), the simple \( \Theta(n^3) \) algorithm is better.
- For larger \( n \), Strassen’s \( \Theta(n^{2.81}) \) algorithm is better.
- The break-even value is \( 20 \leq n \leq 50 \), depending on implementation.
- In some Science/Engineering applications, the matrices in MM are sparse (namely most entries are 0.) In such cases, neither the simple, nor the Strassen’s algorithm work well. Completely different algorithms have been designed.
Complexity of a Problem

The Complexity of an algorithm is the growth rate of its runtime function.

The Complexity of a problem $P$ is the complexity of the best algorithm (known or unknown) for solving it.

The complexity $C_P(n)$ of $P$ is the most important computational property of $P$.

If we have an algorithm for solving $P$ with runtime $T(n)$, then $T(n)$ is an upper bound of $C_P(n)$.

To determine $C_P(n)$, we need to find a lower bound $S_P(n)$: any algorithm (known or unknown) for solving $P$ must have runtime at least $\Omega(S_P(n))$.

If $T(n) = \Theta(S_P(n))$, then $C_P(n) = \Theta(T(n))$.

In most cases, this is extremely hard to do. (How do we determine the runtime function of infinitely many possible algorithms for solving $P$?)

Or, in a few cases, it’s trivial.
Complexity of a Problem

Example
Matrix Addition (MA)

- We have a simple algorithm for MA with runtime $\Theta(n^2)$. So $O(n^2)$ is an upper bound for $C_{MA}(n)$.
- $\Theta(n^2)$ is also a lower bound for $C_{MA}(n)$: any algorithm for solving MA must at least write down the resulting matrix $C$, doing this alone requires $\Omega(n^2)$ time.
- Since the lower and upper bounds are the same, we get $C_{MA}(n) = \Theta(n^2)$.
Complexity of a Problem

Sorting (general purpose)

Given an array $A[1..n]$ of elements, sort $A$. (The only operations allowed for $A$: comparison between array elements).

- **MergeSort** gives an upper bound: $C_{sort}(n) = O(n \log n)$.
- **The Lower Bound Theorem**: Any comparison based sorting algorithm must make at least $\Omega(n \log n)$ comparisons, and hence take at least $\Omega(n \log n)$ time.
- Since the lower and upper bounds are the same, $C_{sort}(n) = \Theta(n \log n)$.
Complexity of a Problem

Example

Matrix Multiplication (MM)

- **Strassen’s algorithm** gives an upper bound: $C_{MM}(n) = O(n^{2.81})$.
- Currently, the **best known upper bound for MM is** $C_{MM}(n) = O(n^{2.3728639})$ (Francois Le Gall, 2014).
- A **trivial lower bound**: Any MM algorithm must write down the resulting matrix $C$, this alone requires at least $\Omega(n^2)$ time. Thus $C_{MM}(n) = \Omega(n^2)$.
- If $C_{MM}(n) = \Theta(n^\alpha)$, we know $2 \leq \alpha \leq 2.3728639$.
- Determining the exact value of $\alpha$ is a long-standing open problem in CS/Math.
Selection Problem

Three basic problems for ordered sets:

- **Sorting.**
- **Searching:** Given an array \( A[1..n] \) and \( x \), find \( i \) such that \( A[i] = x \). If no such \( i \) exists, report “no”.
- **Selection:** Given an unsorted array \( A[1..n] \) and integer \( k \) \((1 \leq k \leq n)\), return the \( k^{th} \) smallest element in \( A \).

**Examples**

- Find the maximum element: \( \text{Select}(A[1..n], n) \).
- Find the minimum element: \( \text{Select}(A[1..n], 1) \).
- Find the median: \( \text{Select}(A[1..n], n/2) \).
What’s the complexity of these three basic problems?

- For **Sorting**, we already know \( C_{\text{sort}}(n) = \Theta(n \log n) \).
- For **Searching**, there are two versions:
  - \( A[1..n] \) is not sorted:
    - The simple **Linear Search** takes \( O(n) \) time.
    - A **trivial lower bound**: We must look at every element of \( A \) at least once. (If not, we might miss \( x \)). So \( C_{\text{unsorted-search}}(n) = \Omega(n) \)
    - Since the lower and upper bounds match we have: \( C_{\text{unsorted-search}}(n) = \Theta(n) \)
  - \( A[1..n] \) is sorted:
    - The simple **Binary Search** takes \( O(\log n) \) time.
    - It can be shown \( \Omega(\log n) \) is also a lower bound.
    - Since the lower and upper bounds match, we have: \( C_{\text{sorted-search}}(n) = \Theta(\log n) \)
Selection Problem

What about Selection?

**Simple-Select**$(A[1..n], k)$

1: **MergeSort**$(A[1..n])$
2: output $A[k]$

- This algorithm solves the select problem in $\Theta(n \log n)$ time.
- But this is an overkill: to solve the select problem, we don’t have to sort.
- We will present a Linear Time Select algorithm.
- $\Omega(n)$ is a trivial lower bound for Select: we must look at each array element at least once, otherwise the answer could be wrong.
- This would give: $C_{\text{Select}}(n) = \Theta(n)$. 
This algorithm uses DaC. We need a function \texttt{partition}(A,p,r).

The goal: rearrange \(A[p..r]\) so that for some \(q\) \((p \leq q \leq r)\),

\[
\begin{align*}
A[i] & \leq A[q] \quad \forall i = p, \ldots, q - 1 \\
A[q] & \leq A[j] \quad \forall j = q + 1, \ldots, r
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>(p)</th>
<th>(q - 1)</th>
<th>(q)</th>
<th>(q + 1)</th>
<th>(r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\leq A[q])</td>
<td>(A[q])</td>
<td>(\geq A[q])</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The following code partitions $A[p..r]$ around $A[r]$

**Partition**($A, p, r$)

1: $x \leftarrow A[r]$ (\(x\) is “pivot”.)
2: $i \leftarrow p - 1$
3: for $j \leftarrow p$ to $r - 1$ do
4: \hspace{1em} if $A[j] \leq x$ then
5: \hspace{2em} $i \leftarrow i + 1$
6: \hspace{2em} swap $A[i]$ and $A[j]$
7: \hspace{1em} end if
8: end for
9: swap $A[i + 1]$ and $A[r]$
10: return $i + 1$
Example: \((x = 4\) is the pivot element.\)

<table>
<thead>
<tr>
<th>i</th>
<th>p, j</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>3 1 8 5 6 2 7 4 ...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p, i</th>
<th>j</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>3 1 8 5 6 2 7 4 ...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>i</th>
<th>j</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>3 1 8 5 6 2 7 4 ...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>i</th>
<th>j</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>3 1 8 5 6 2 7 4 ...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>i</th>
<th>j</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>3 1 8 5 6 2 7 4 ...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>i</th>
<th>j</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>3 1 8 5 6 2 7 4 ...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>i</th>
<th>j</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>3 1 8 5 6 2 7 4 ...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>i</th>
<th>j</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>3 1 8 5 6 2 7 4 ...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>i</th>
<th>j</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>3 1 8 5 6 2 7 4 ...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>i</th>
<th>j</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>3 1 8 5 6 2 7 4 ...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>i</th>
<th>j</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>3 1 8 5 6 2 7 4 ...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p</th>
<th>i</th>
<th>j</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>3 1 8 5 6 2 7 4 ...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We show \textbf{Partition} \((A, p, r)\) achieves the goal. Before the loop 3-8 is entered, the following is true for any index \(k\):

1. If \(k \in [p, i]\), then \(A[k] \leq x\)
2. If \(k \in [i + 1, j - 1]\), then \(A[k] > x\)
3. If \(k = r\), then \(A[k] = x\)
4. If \(k \in [j, r - 1]\), then \(A[k]\) is unrestricted.

<table>
<thead>
<tr>
<th></th>
<th>(p)</th>
<th>(i)</th>
<th>(i + 1)</th>
<th>(j - 1)</th>
<th>(j)</th>
<th>(r - 1)</th>
<th>(r)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\leq x)</td>
<td>(&gt; x)</td>
<td>unrestricted</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Before the 1\(^{st}\) iteration, \(i = p - 1, j = p\).

- \([p, i] = [p, p - 1] = \emptyset\), condition (1) is trivially true.
- \([i + 1, j - 1] = [p, p - 1] = \emptyset\), condition (2) is trivially true.
- condition (3) and (4) are trivially true.
Partition

Case (a) $A[j] > x$: Before:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$i$</th>
<th>$i + 1$</th>
<th>....</th>
<th>$j$</th>
<th>$r - 1$</th>
<th>$r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\leq x$</td>
<td>$&gt; x$</td>
<td>$&gt; x$</td>
<td>$&gt; x$</td>
<td>unrestricted</td>
<td>$x$</td>
<td>$x$</td>
</tr>
</tbody>
</table>

After:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$i$</th>
<th>$i + 1$</th>
<th>....</th>
<th>$j$</th>
<th>$r - 1$</th>
<th>$r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\leq x$</td>
<td>$&gt; x$</td>
<td>$&gt; x$</td>
<td>$&gt; x$</td>
<td>unrestricted</td>
<td>$x$</td>
<td>$x$</td>
</tr>
</tbody>
</table>

Case (b) $A[j] \leq x$: Before:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$i$</th>
<th>$i + 1$</th>
<th>....</th>
<th>$j$</th>
<th>$r - 1$</th>
<th>$r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\leq x$</td>
<td>$a &gt; x$</td>
<td>$&gt; x$</td>
<td>$b \leq x$</td>
<td>unrestricted</td>
<td>$x$</td>
<td>$x$</td>
</tr>
</tbody>
</table>

After:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$i$</th>
<th>....</th>
<th>$j$</th>
<th>$r - 1$</th>
<th>$r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\leq x$</td>
<td>$b \leq x$</td>
<td>$&gt; x$</td>
<td>$a &gt; x$</td>
<td>unrestricted</td>
<td>$x$</td>
</tr>
</tbody>
</table>
Partition

After the loop termination \( j = r - 1 \):

\[
\begin{array}{|c|c|c|c|c|}
\hline
p & i & i + 1 & \cdots & j = r - 1 & r \\
\hline
\leq x & a > x & & > x & x \\
\hline
\end{array}
\]

After final swap:

\[
\begin{array}{|c|c|c|c|c|}
\hline
p & i & i + 1 & \cdots & j = r - 1 & r \\
\hline
\leq x & x & & > x & a > x \\
\hline
\end{array}
\]

- This is what we want.
- It is easy to see \textbf{Partition}(A, p.r) takes \( \Theta(n) \) time where \( n = r - p + 1 \) is the number of elements in \( A[p..r] \).
Select

The following algorithm returns the \( i^{th} \) smallest element in \( A[p..r] \). (It requires \( 1 \leq i \leq r - p + 1 \).)

**Select** \((A, p, r, i)\)

1. if \( p = r \), return \( A[p] \) (in this case we must have \( i = r - p + 1 = 1 \)).
2. \( x = A[r] \)
3. swap \((A[r], A[r])\)
4. \( q = \text{Partition}(A, p, r) \)
5. \( k = q - p + 1 \)
6. if \( i = k \), return \( A[q] \)
7. if \( i < k \), return \( \text{Select}(A, p, q - 1, i) \)
8. if \( i > k \), return \( \text{Select}(A, q + 1, r, i - k) \)

Note: lines (2) and (3) don’t do anything here. We will modify it later.

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q - 1 )</th>
<th>( q )</th>
<th>( q + 1 )</th>
<th>( r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \leftarrow ) q - p elements</td>
<td>→</td>
<td>( A[q] )</td>
<td>( \leftarrow ) n - k elements</td>
<td>→</td>
</tr>
</tbody>
</table>
Select

- If we pick any element \( x \in A[p..r] \), the algorithm will work correctly.
- It can be shown the expected or average runtime is \( \Theta(n) \).
- However, in the worst case, the runtime is \( \Theta(n^2) \).

Worst case example:

\( A[1..n] \) is already sorted and we try to find the smallest element.

- **Select**(\( A[1..n] \), 1) calls **Partition**(\( A[1..n] \)) which returns \( q = n \).
- **Select**(\( A[1..n-1] \), 1) calls **Partition**(\( A[1..n-1] \)) which returns \( q = n - 1 \).
- **Select**(\( A[1..n-2] \), 1) calls **Partition**(\( A[1..n-2] \)) which returns \( q = n - 2 \).
- ....

The runtime will be \( \Theta(n + (n - 1) + (n - 2) \ldots 2 + 1) = \Theta(n^2) \).
The problem: the final position $q$ of the pivot element $x = A[r]$ can be anywhere, we have no control on this.

If $q$ is close to the beginning or the end of $A[p..r]$, it will be slow.

If we can pick $x$ so that $q$ is at about the middle of $A[p..r]$, then the two sub-problems are about equal size, the runtime will be much better.

How to do this?
Replace the line (2) by the following:

**Line 2**

1: divide $A[1..n]$ into $\lceil \frac{n}{5} \rceil$ groups, each containing 5 elements (except the last group which may have < 5 elements).

2: For each group $G_i$, let $x_i$ be the median of the group.

3: Let $M = \langle x_1, x_2, \ldots, x_{\lceil n/5 \rceil} \rangle$ be the collection of these median elements.

4: recursively call $x = \text{Select}(M[1..\lceil n/5 \rceil], n/10)$. (Namely, $x$ is the median of $M$).

5: use $x$ as the pivot element.

We will show in class this modification will give a $\Theta(n)$ time selection algorithm.
The linear time selection algorithm is complex. The constant hidden in $\Theta(n)$ is large. It's not a practical algorithm. The significance:

- It settled the complexity issue of a fundamental problem: $C_{selection} = \Theta(n)$.
- It illustrates two important algorithmic ideas:
  - **Random Sampling**: randomly pick pivot $x$ to partition the array. On average, the algorithm takes $\Theta(n)$ time.
  - **Derandomization**: Make a clever choice of $x$, remove the randomness.
- These ideas are used in other algorithms.
Summary on Using DaC Strategy

- When divide into subproblems, the size of the sub-problems should be $n/b$ for some constant $b > 1$. If it is only $n - c$ for some constant $c$, and there are at least two subproblems, this usually leads to exp time.

Example:

Fib($n$)
1: if $n = 0$ return 0
2: if $n = 1$ return 1
3: else return Fib($n - 1$) + Fib($n - 2$)
Summary on DaC Strategy

We have:

\[ T(n) = \begin{cases} 
O(1) & \text{if } n \leq 1 \\
T(n-1) + T(n-2) + O(1) & \text{if } n \geq 2 
\end{cases} \]

Thus:

\[ T(n) \geq T(n-2) + T(n-2) = 2T(n-2) \geq 2^2T(n-2\cdot2) \geq \cdots \geq 2^kT(n-2\cdot k). \]

When \( k = n/2 \), we have: \( T(n) \geq 2^{n/2}T(0) = \Omega((\sqrt{2})^n) = \Omega((1.414)^n). \)

Actually, \( T(n) = \Theta(\alpha^n) \) where \( \alpha = \frac{\sqrt{5}+1}{2} \approx 1.618. \)

We make two recursive calls, with size \( n - 1 \) and \( n - 2 \). This leads to exp time.
Summary on DaC Strategy

- When divide into sub-problems, try to divide them into about equal sizes.
  In the linear time select algorithm, we took great effort to ensure the size of the sub problem is \( \leq \frac{7n}{10} \).

- After we get \( T(n) = aT(n/b) + \Theta(n^k) \). How to improve?
  - If \( \log_b a < k \), then the cost of other processing dominates the runtime. We must reduce it.
  - If \( \log_b a > k \), then the cost of recursive calls dominates the runtime. We must reduce the number of recursive calls. (Strassen’s algorithm).
  - If \( \log_b a = k \), then the cost of two parts are about same. To improve, we must reduce both. Quite often, when you reach this point, you have the best algorithm!
Computational Geometry

The branch of CS that studies geometry problems. It has applications in Computer Graphics, Robotics, Motion Planning ...

Motion Planing

Given a set of polygons in 2D plane and two points $a$ and $b$, find the shortest path from $a$ to $b$, avoiding all polygons.
Hidden Surfaces Removal

Given a set of polygons in 3D space and a view point $p$, identify the portions of the polygons that can be seen from $p$.

Closest Point Pair Problem

Input: A set $P = \{p_1, p_2 \ldots p_n\}$ of $n$ points $(p_i = (x_i, y_i))$.
Find: $i \neq j$ such that $\text{dist}(p_i, p_j) \overset{\text{def}}{=} [(x_i - x_j)^2 + (y_i - y_j)^2]^{1/2}$ is the smallest among all point pairs.

This is a basic problem in Computational Geometry.

Simple algorithm:

- For each pair $i \neq j$, compute $\text{dist}(p_i, p_j)$.
- Pick the pair with smallest distance.

- Let $f(n)$ be the time needed to evaluate $\text{dist}(\ast)$.
- Since there are $\Theta(n^2)$ point pairs, this algorithm takes $\Theta(n^2 f(n))$ time.
- By using DaC, we get a $\Theta(n \log nf(n))$ time algorithm.
Closest Point Pair Problem

**ClosestPair**($P$)

**Input:** The point set $P$ is represented by $X = [x_1, \ldots, x_n]$ and $Y = [y_1, \ldots, y_n]$.

**Preprocessing:** Sort $X$; sort $Y$. This takes $O(n \log n)$ time

1: If $n \leq 4$, find the shortest point pair directly. This takes $O(1)$ time.

2: Divide the point set $P$ into two parts as follows: Draw a vertical line $l$ that divides $P$ into $P_L$ (points to the left of $l$), and $P_R$ (points to the right of $l$), so that $|P_L| = \lceil n/2 \rceil$ and $|P_R| = \lfloor n/2 \rfloor$.

Note: Since $X$ is already sorted, we can draw $l$ between $x_{\lceil n/2 \rceil}$ and $x_{\lceil n/2 \rceil}+1$.

We scan $X$ and collect points into $P_L$ and $P_R$. This takes $O(n)$ time.

3: Recursively call **ClosestPair**($P_L$). Let

- $p_{Li}, p_{Lj}$ be the point pair with smallest distance in $P_L$.
- $\delta_L = \text{dist}(p_{Li}, p_{Lj})$. 
Closest Point Pair Problem

4: Recursively call \texttt{ClosestPair}(P_R). Let

- \( p_{Ri}, p_{Rj} \) be the point pair with smallest distance in \( P_R \).
- \( \delta_R = \text{dist}(p_{Ri}, p_{Rj}) \).

5: Let \( \delta = \min\{\delta_L, \delta_R\} \). This takes \( O(1) \) time.

6: Combine. The solution of the original problem must be one of three cases:

- The closest pair \( p_i, p_j \) are both in \( P_L \). We have solved this case in (3).
- The closest pair \( p_i, p_j \) are both in \( P_R \). We have solved this case in (4).
- One of \( \{p_i, p_j\} \) is in \( P_L \) and another one is in \( P_R \). We must find the solution in this case.

Note: Let \( S \) be the vertical strip with width \( 2\delta \) centered at the line \( l \). Then \( p_i \) and \( p_j \) must be in \( S \). (Why?)
6.1: Let $P' = \{q_1, q_2, \ldots, q_t\}$ be the points in the $2\delta$ wide strip $S$. Let $Y'$ be the $y$-coordinates of points in $P'$ in increasing order.

Note: Since $Y$ is already sorted, we scan $Y$, and only include the points that are in the strip $S$. This takes $O(n)$ time.

6.2: For each $q_i$ ($i = 1 \ldots t$) in $P'$, compute $\text{dist}(q_i, q_j)$ where $i < j \leq i + 7$. Let $\delta'$ be the smallest distance computed in this step.

Note: If $(q_i, q_j)$ is the closest pair, then both must be in the region $A$ or $B$ ($q_i$ is at the bottom edge). But any two points in $A$ have inter-distance at least $\delta$. $A$ can contain at most 4 points. Similarly $B$ can contain at most 4 points. So we only need to compare $\text{dist}$ between $q_i$ and next 7 points in $P'$!
### Closest Point Pair Problem

**6.3:** If $\delta' < \delta$, the shortest distance computed in (6.2) is the shortest distance for the original problem. If $\delta' \geq \delta$, the shortest distance computed in (3) or (4) is the shortest distance for the original problem. Output accordingly.

**Analysis:** Let $T(n)$ be the number of computation of $\text{dist}(*)$ by the alg. The algorithm makes two recursive calls, each with size $n/2$. All other processing takes $O(n)$ time. Thus:

$$T(n) = \begin{cases} 
O(1) & \text{if } n \leq 4 \\
2T(n/2) + \Theta(n) & \text{if } n > 4 
\end{cases}$$

Thus: $T(n) = \Theta(n \log n)$. 