Unit 8. Lower Bound Theory

Algorithms

EE3980

May 16, 2018

Lower Bounds

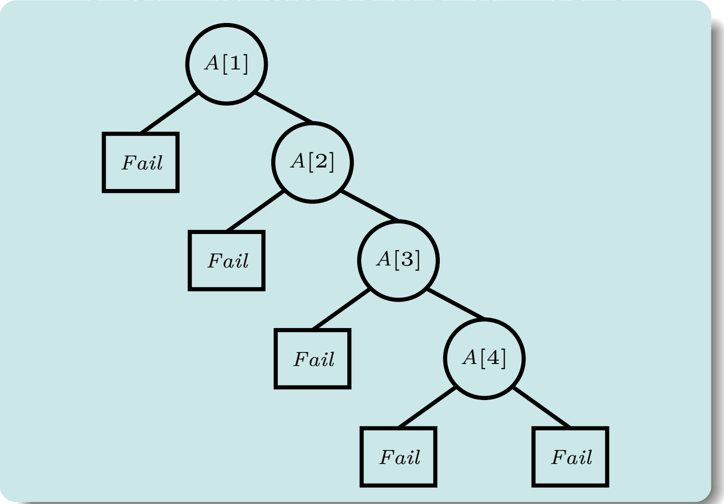
- Given a problem, one can device algorithms to solve the problem.
  - Once an algorithm is developed, we know how to analyze the time and space complexity.
  - Over all the algorithms, the one with the minimum complexity is usually preferred.
  - If we know the lower bound of a given problem, then we can strive to solve it with the lowest complexity possible.
- Some problems have been studied extensively and the results are listed in this unit.
- Lower bounds for searching and sorting algorithms are studied first.

Ordered Searching

- Comparison based complexity analysis is assumed.
- To find x in an ordered array $A[i]$, $A[i] < A[j]$ if $i < j$.
- A series of comparisons are to be performed.
- Each comparison can have one of three results:

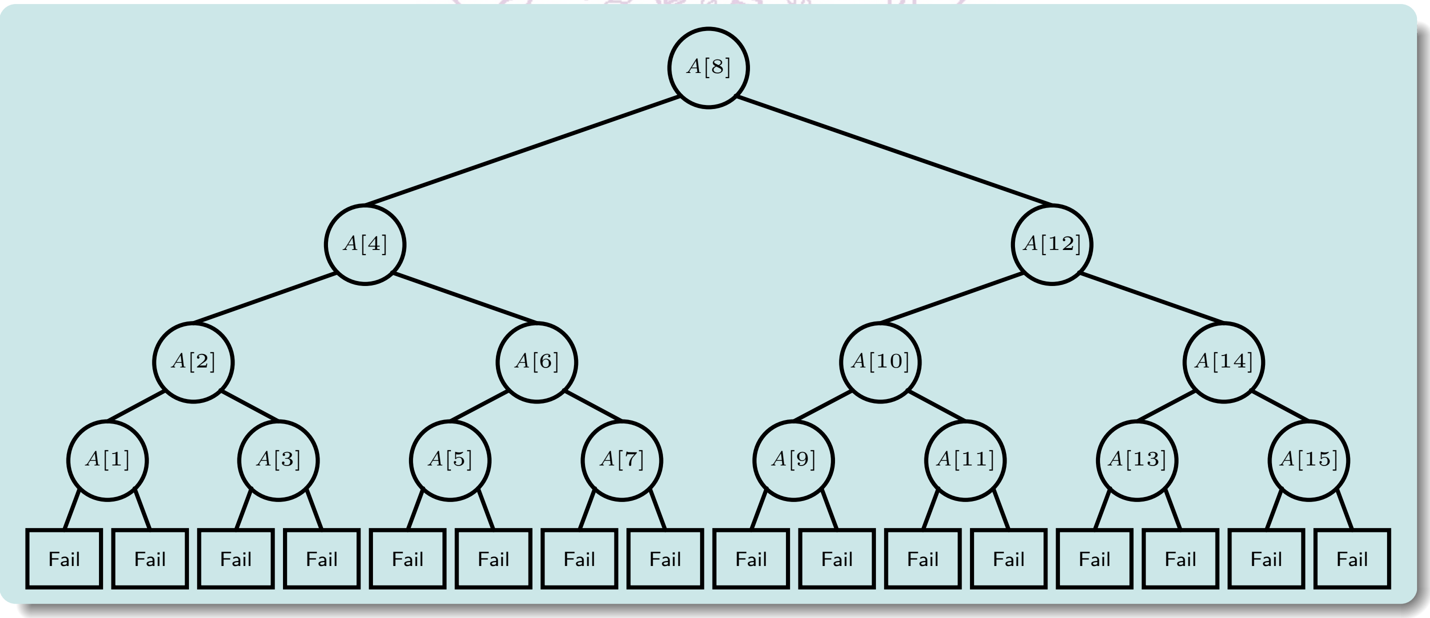
$$x < A[i],\ x = A[i], \text{ or } x > A[i].$$

- Array A can be stored as a tree.
- A linear search is shown below.
  - The worst-case complexity is $\mathcal{O}(n)$.



Ordered Searching, II

- A binary search tree is shown below.
- For any array of n elements, there are $n+1$ possible fails.
- If there are k levels in the tree, then there are at most $2^k - 1$ internal nodes.
- Therefore, for an array with n elements for the tree with k levels, $n \le 2^k - 1$, or $k \ge \lg(n+1)$.



Ordered Searching, III

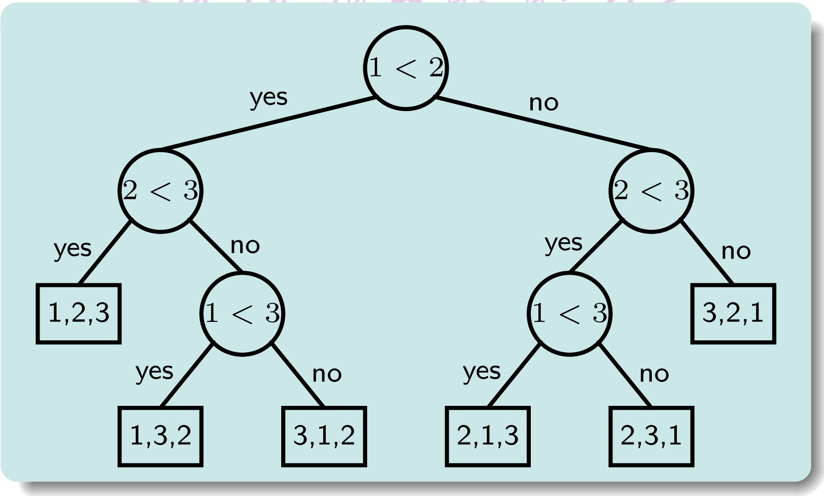
Theorem 8.1.1.

Let $A[1:n]$, $n \geq 1$, contains n distinct elements, ordered so that $A[1] < A[2] < \cdots < A[n]$. Let $\texttt{FIND}(n)$ be the minimum number of comparisons needed, in the worst case, by any comparison-based algorithm to recognize whether $x \in A[1:n]$. Then $\texttt{FIND}(n) \geq \lceil \lg(n+1) \rceil$.

- As a consequence of this algorithm, the binary search algorithm is an optimal worst-case algorithm for the ordered searching problem.

Sorting

- Given an array $A[1:n]$ with all elements distinct. The sorting problem is to rearrange the array A such that $A[i] < A[j]$, if $1 \leq i < j \leq n$.
- An example of sorting 3-integer array, $\{1, 2, 3\}$, is shown below.
  - Each internal node performs a comparison, $A[i] < A[j]$.
    - The comparison can have only two results: `true` or `false`.
  - Each external node represents one of the possible sorting results.
    - With 3 elements, there are $6 = 3!$ external nodes.



Sorting — Lower Bound

- Given $A[1:n]$, the comparison based algorithm should have a state space with $n!$ external nodes, and these external nodes are the leaves of the binary tree.
- Assuming that the binary tree has k levels, it takes k comparisons to perform the sorting algorithm.
- Let $T(n)$ be the minimum number of comparisons to sort $A[1:n]$, then

  $$2^{T(n)} \geq n!$$

  And

  $$T(n) \geq \lceil \lg n! \rceil$$

  By Stirling's approximation

  $$\lg n! = n \lg n - n/(\lg 2) + (\lg n)/2 + \mathcal{O}(1)$$

- Thus, any comparison-based sorting algorithm needs at least $\Omega(n \lg n)$ time.

Sorting Complexity Example — Merge Sort

- Merge sort starts by comparing two elements to form $n/2$ groups of 2 elements.
- Then two two-element groups are sorted.
  - 3 comparisons are needed to form $n/4$ groups.
- The next step compares 4-element groups to form $n/8$ groups.
  - 7 comparisons are needed to sort two 4-element groups.
- Thus, the total number of comparisons is

  $$T(n) = \sum_{i=1}^{k} \frac{n}{2^i}(2^i - 1) = \sum_{i=1}^{k} n - n \sum_{i=1}^{k} \frac{1}{2^i}$$

  where $k = \lg n$.
- Thus, $T(n) = n \lg n - \mathcal{O}(n)$.
- Merge sort achieves the lowest time complexity, but the coefficients can still be improved.
  - See textbook [Horowitz], pp. 481-483.

Merging

- Given two ordered arrays $A[1:m]$ and $B[1:n]$, a third ordered array $C[1:m+n]$ is formed by merging these two arrays together.
- Given the numbers m and n, there are $\binom{m+n}{n}$ combinations of possibilities combining $A[1:m]$ and $B[1:n]$.
- Using comparison based algorithms, a tree can be formed and there should be at least $\binom{m+n}{n}$ external nodes.
- Let $\texttt{MERGE}(m,n)$ be the minimum number of comparisons to merge $A[1:m]$ and $B[1:n]$, then

$$\texttt{MERGE}(m,n) \geq \left\lceil \lg \binom{m+n}{n} \right\rceil.$$

- It has been shown in Unit 3 that the upper bound of $\texttt{MERGE}(m,n)$, thus

$$\left\lceil \lg \binom{m+n}{n} \right\rceil \leq \texttt{MERGE}(m,n) \leq m+n-1.$$

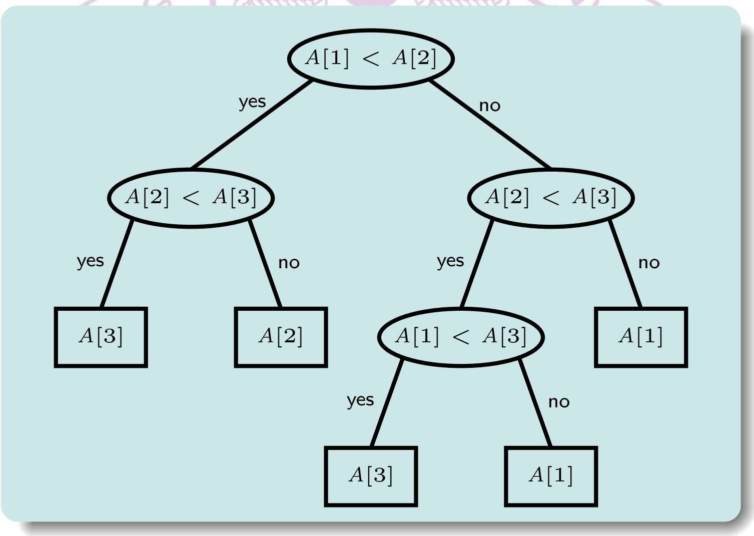
- A special case when $m=n$

Theorem 8.1.2.

$\texttt{MERGE}(m,m) = 2m-1$, for $m \geq 1$.

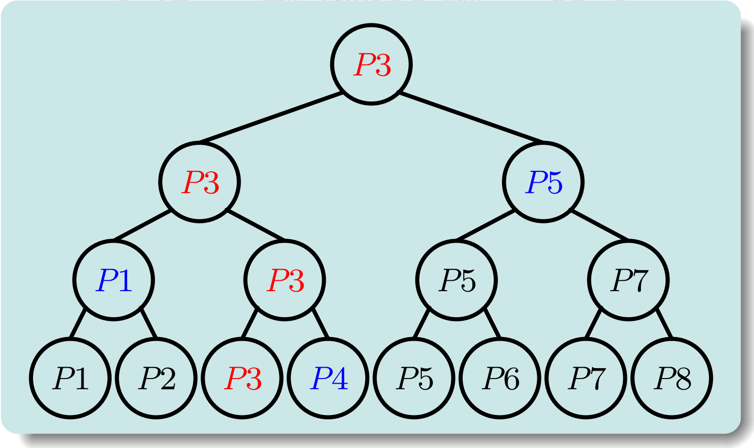
Finding the Largest Element

- To find the largest element of an n-element array A, there must be at least $n-1$ nodes in the tree.
  - After k comparisons, only one element remains that is greater than any other element. The smallest k is $n-1$.
- Thus, the minimum number of comparisons for finding the largest elements of an n-element array is $L_1(n) = n-1$.
- Example of the comparison tree of finding the largest element of a 3-element array, $A[1:3]$.



Largest and 2nd Largest

- Given an unordered set $A[1:n]$, finding the largest element needs $n-1$ comparison.
- The comparison tree can be arranged as the following.



Largest and 2nd Largest, II

- To find the 2nd largest element, one needs to compare only those elements that compared to the largest element and were found to be smaller.
  - There are only $\lg n$ such elements.
  - To find the largest among them needs $\lg n - 1$ comparison.
- Thus to find the largest and second largest elements needs $n + \lg n - 2$ comparisons.

Theorem 8.1.3.

Any comparison-based algorithm that computes the largest and the second largest element of a set of n unordered elements requires $n - 2 + \lceil \lg n \rceil$ comparisons.

The Largest to the k-th Largest Elements

- The comparison tree of finding the largest to the k-th largest elements of $A[1:n]$ needs to have $n \cdot (n-1) \cdots (n-k+1)$ external nodes.
- Thus, let $L_k(n)$ be the minimum number of comparisons of finding the largest to the k largest elements

$$L_k(n) \geq \left\lceil \lg\left(n \cdot (n-1) \cdots (n-k+1)\right) \right\rceil.$$

- More detailed analysis shows that

Theorem 8.1.4.

$L_k(n) \geq n - k + \left\lceil \lg\left(n \cdot (n-1) \cdots (n-k+2)\right) \right\rceil$ for all integers k and n, where $1 \leq k \leq n$.

- Note that this is an estimate of the lower bound.

Find the Largest k elements

Theorem 8.1.5.

Given an unordered set with n elements, the $(k-1)$th largest element itself needs at least $(k-1)\left\lceil \lg \dfrac{n}{2(k-1)} \right\rceil$ comparisons to be identified.

- Proof please see textbook [Horowitz], p. 491.

Theorem 8.1.6.

Given an unordered set with n elements, all $k-1$ largest elements can be found with at least $n - k + (k-1)\left\lceil \lg \dfrac{n}{2(k-1)} \right\rceil$ comparisons.

- Proof please see textbook [Horowitz], pp. 491-492.

Finding the Maximum and Minimum

- Given n distinct elements, find the maximum and the minimum.
- Using comparison-based algorithms, define 4-tuple (a, b, c, d) as
  - a is the number of elements that have not been compared,
  - b is the number of elements that have won and never lost,
  - c is the number of elements that have lost and never won,
  - d is the number of elements that have both won and lost.
- Then given a state (a, b, c, d), an additional comparison can result in one of the following states:

$(a-2, b+1, c+1, d)$	if $a \geq 2$	// Compare two items from a.
$(a-1, b+1, c, d)$		// Compare one item from a
$(a-1, b, c+1, d)$	if $a \geq 1$	// with one item from b
$(a-1, b, c, d+1)$		// or from c.
$(a, b-1, c, d+1)$	if $b \geq 2$	// Compare two items from b.
$(a, b, c-1, d+1)$	if $c \geq 2$	// Compare two items from c.

Finding the Maximum and Minimum, II

- The initial state is $(n, 0, 0, 0)$ since all elements have not been compared.
- Then it takes $n/2$ comparisons, comparing elements in a, to move to the state $(0, n/2, n/2, 0)$.
- The final state is $(0, 1, 1, n-2)$ since we want to find the maximum, only one element left in a, and the minimum, only one element left in b, the rest elements must be in d.
  - The minimum number is $n-2$ since d can only be increased by 1 with each comparison.

Theorem 8.1.7.

Any algorithm that computes the largest and the smallest elements of a set of n unordered elements requires $\lceil 3n/2 \rceil - 2$ comparisons.

Problem Reduction

Definition 8.1.8. Problem reduction.

Let P_1 and P_2 be any two problems. We say P_1 reduces to P_2, denoted by $P_1 \propto P_2$, in time $\tau(n)$ if an instance of P_1 can be converted into an instance of P_2 and solution for P_1 can be obtained from a solution of P_2 in time $\leq \tau(n)$.

- Example
  - P_1 is the problem of selection (Finding the kth smallest element.)
  - P_2 is the problem of sorting.
  - If the input have n numbers and the number are sorted in an array $A[1:n]$,
  - The kth smallest element of the input can be obtained as $A[k]$.
  - Thus, P_1 reduces to P_2 in $\mathcal{O}(1)$ time.
- Note there are three steps in this formulation
  - Convert the inputs of problem P_1 to P_2
    - In this example, no special action is required.
  - Solve problem P_2.
    - $\mathcal{O}(n \lg n)$ if comparison based algorithm is adopted.
  - Convert the solution of P_2 to that of P_1.
    - $\mathcal{O}(1)$ since $A[k]$ is the solution of P_1.

Problem Reduction, II

- Example 2
  - Given two sets S_1 and S_2 with m elements each.
  - P_1 is the problem to check if S_1 and S_2 are disjoint, i.e., $S_1 \cap S_2 = \emptyset$.
  - P_2 is the sorting problem.
  - Then $P_1 \propto P_2$ in $\mathcal{O}(m)$ time.

  - Let $S_1 = \{k_1, k_2, \cdots, k_m\}$ and $S_2 = \{h_1, h_2, \cdots, h_m\}$, then we can create a set $X = \{(k_1, 1), (k_2, 1), \cdots, (k_m, 1), (h_1, 2), (h_2, 2), \cdots, (h_m, 2)\}$.
  - This X can be created in $2m$ time ($\mathcal{O}(m)$).
  - Then X can be sorted by the first element of each tuple.
    - $\mathcal{O}(n \lg n)$, $n = 2m$, if comparison-based method is used.
  - After sorting, we can check whether there are two successive elements $(x, 1)$ and $(y, 2)$ such that $x = y$.
    - $2m - 1$ comparisons are needed ($\mathcal{O}(m)$).
  - If there are no such elements, then S_1 and S_2 are disjoint; otherwise they are not.

Lower Bounds Through Reductions

- Given two problems P_1 and P_2 such that P_1 reduces to P_2 in $\tau(n)$,
  - The input of P_1 is converted to the input of P_2 and the solution is obtained from P_2 in $\tau(n)$.
  - Suppose problem P_1 can be solved in time $T_1(n)$ and
  - Problem P_2 can be solved in time $T_2(n)$, then

$$T_1(n) \le \tau(n) + T_2(n). \tag{8.1.1}$$

  Or,

$$T_2(n) \ge T_1(n) - \tau(n). \tag{8.1.2}$$

- Thus, the lower bound for solving problem P_2 is $T_1(n) - \tau(n)$.

Finding Convex Hull

- Let P_1 be a sorting problem on n numbers.
  - $T_1(n) = \mathcal{O}(n \lg n)$.
- These numbers can be transformed into n points on a 2-D plane as $\{(k_1, k_1^2), (k_2, k_2^2), \cdots, (k_n, k_n^2)\}$.
  - This transformation takes $\mathcal{O}(n)$ time.
- Let P_2 be the problem of finding the convex hull of the n points.
  - $T_2(n)$ is solution time for $P_2(n)$.
- Note that the n points arranged in sorted order (sorted by x coordinate) form a convex hull with the first point appended to the end.
- In this case

$$T_2(n) \le T_1(n) - \mathcal{O}(n) = \mathcal{O}(n \lg n) - \mathcal{O}(n). \tag{8.1.3}$$

- Thus, we have

Lemma 8.1.9. Find Convex Hull

Computing the convex hull of n given points in the plane needs $\Omega(n \lg n)$ time.

Multiplying Triangular Matrices

- Given an $n \times n$ matrix A whose elements are $\{a_{i,j} | 1 \leq i, j \leq n\}$
- A is said to be upper triangular if $a_{ij} = 0$ whenever $i > j$.
- A is said to be lower triangular if $a_{ij} = 0$ for $i < j$.
- A is said to be triangular if it is either upper triangular or lower triangular.
- We are interested in the question if multiplying two lower (or upper) triangular matrices is faster than multiplying two full matrices.
- Let $M(n)$ be the time complexity of multiplying two full matrices, and $M_t(n)$ be the time complexity of multiplying two lower triangular matrices.
  - Note that $M_t(n) \leq M(n)$.
- And $M(n) = \Omega(n^2)$ since there are $2n^2$ elements in the input and n^2 elements in the output.

Multiplying Triangular Matrices, II

- Let P_1 be the problem of multiplying two full matrices A and B, each of size $n \times n$.
- Let P_2 be the problem of multiplying two lower triangular matrices.
- The problem of P_1 can be transformed into an instance of P_2 problem as

$$A' = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & A & \mathbf{0} \end{bmatrix} \qquad B' = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ B & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

  where $\mathbf{0}$ denotes a zero matrix, that is, an $n \times n$ matrix with all elements 0.
- Note that both A' nd B' are lower triangular matrices.
- And

$$A'B' = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ AB & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

Multiplying Triangular Matrices, III

- Thus, the product of full matrices can be obtained from product of lower triangular matrices.
- Transforming full matrices to triangular matrices takes $\mathcal{O}(n^2)$ time.
- Getting the product AB from $A'B'$ also takes $\mathcal{O}(n^2)$.
- And we have

$$M_t(3n) \geq M(n) - \mathcal{O}(n^2) = \Omega(n^2) - \mathcal{O}(n^2) = \Omega(n^2) \tag{8.1.4}$$

- Or

$$M_t(n) \geq \Omega((\frac{n}{3})^2) = \Omega(n^2) = \Omega(M(n)). \tag{8.1.5}$$

- Thus we have

Lemma 8.1.10. Multiplying triangular matrices

$M_t(n) = \Omega(M(n))$.

- Since $M(n) \geq M_t(n)$ we conclude that $M_t(n) = \Theta(M(n))$.

Inverting a Lower Triangular Matrix

- An $n \times n$ matrix I is an identity matrix if

$$I_{j,k} = \begin{cases} 1, & \text{if } j = k, \\ 0, & \text{otherwise.} \end{cases} \tag{8.1.6}$$

- Given an $n \times n$ matrix A, if there exists a matrix B such that $AB = I$, then B is called the inverse of A and A is said to be invertible. Also, the inverse of A is denoted as A^{-1}.
- Note that not every matrix is invertible.
- Given an $n \times n$ lower triangular matrix A, if all the diagonal elements $a_{i,j} \neq 0$, $1 \leq i, j \leq n$, then A is invertible.
- In the following we are interested in the time complexity of inverting a lower triangular matrix, especially, compared to the full matrix multiplication.

Inverting a Lower Triangular Matrix, II

- Let P_1 be the problem of multiplying two full matrices, and P_2 be the problem of inverting a lower triangular matrix.
- Let $I_t(n)$ be the time complexity of inverting a lower triangular matrix of dimension $n \times n$, and $M(n)$ is the complexity of multiplying two full matrices.
- Given two full $n \times n$ matrices A and B, the following $3n \times 3n$ lower triangular matrix can be constructed

$$C = \begin{bmatrix} I & \mathbf{0} & \mathbf{0} \\ B & I & \mathbf{0} \\ \mathbf{0} & A & I \end{bmatrix} \tag{8.1.7}$$

where I is the identity matrix of dimension $n \times n$ and $\mathbf{0}$ is the zero matrix of the same dimension.

Inverting a Lower Triangular Matrix, III

- And it can be shown that the inverse matrix is

$$C^{-1} = \begin{bmatrix} I & \mathbf{0} & \mathbf{0} \\ -B & I & \mathbf{0} \\ AB & -A & I \end{bmatrix} \tag{8.1.8}$$

- Thus, matrix product can be obtained from inverting a matrix.
- Furthermore, we have $I_t(3n) \le M(n) - \mathcal{O}(n^2)$.
- Since $M(n) = \Omega(n^2)$ we have the following Lemma.

Lemma 8.1.11.

$I_t(n) = \Omega(M(n))$.

Inverting a Lower Triangular Matrix, IV

- Given an $n \times n$ lower triangular matrix A, we can partition it into 4 submatrices of dimension $\frac{n}{2} \times \frac{n}{2}$ each as

$$A = \begin{bmatrix} A_{11} & \mathbf{0} \\ A_{21} & A_{22} \end{bmatrix} \tag{8.1.9}$$

where both A_{11} and A_{22} are lower triangular matrices, but A_{21} can be full.

- It can be shown that

$$A^{-1} = \begin{bmatrix} A_{11}^{-1} & \mathbf{0} \\ -A_{22}^{-1} A_{21} A_{11}^{-1} & A_{22}^{-1} \end{bmatrix} \tag{8.1.10}$$

Thus, the inverse of A can be constructed using divide-and-conquer approach.

- The inverse of submatrices A_{11} and A_{22} are first found, $2I_t(\frac{n}{2})$, and then two matrix multiplications are performed, $2M(\frac{n}{2})$, followed by negating all elements of the products, $\mathcal{O}(\frac{n}{4})$).

Inverting a Lower Triangular Matrix, V

- And the currence equation is

$$\begin{aligned} I_t(n) &= 2I_t(\frac{n}{2}) + 2M(\frac{n}{2}) + \frac{n^2}{4} \\ &= 4I_t(\frac{n}{4}) + 4M(\frac{n}{4}) + 2\frac{n^2}{16} + 2M(\frac{n}{2}) + \frac{n^2}{4} \\ &= 2M(\frac{n}{2}) + 4M(\frac{n}{4}) + \cdots + \frac{n^2}{4} + \frac{n^2}{8} + \cdots \\ &= \mathcal{O}(M(n) + n^2) \end{aligned}$$

The last equality comes from $M(n) = \Omega(n^2)$. The following Lemma is obtained.

Lemma 8.1.12.

$I_t(n) = \mathcal{O}(M(n))$.

- Combining the last two lemmas, we conclude that $I_t(n) = \Theta(M(n))$. That is inverting a lower triangular matrix has the same time complexity as multiplying two full matrices.

Summary

- Theoretical lower bounds
- Ordered searching
- Sorting
  - Merge sort
- Merging ordered arrays
- Finding the largest element
- The largest and 2nd largest elements
- The largest to the k-th largest elements
- Finding the maximum and the minimum
- Problem reduction
- Lower bound through problem reduction
- Finding convex hull.
- Lower triangular matrix multiplication
- Lower triangular matrix inversion