Module 3
Comparison Based Problems

S. Lakshmivarahan
School of Computer Science
University of Oklahoma
USA-73019
Varahan@ou.edu

Comparison Based Problems

- Selection, searching, merging, and sorting problems use only one of the basic operations- namely, the comparison operation
- The goal is to store a set of records in the sorted form so that the subsequent search becomes easier and efficient
- A number of applications come to mind:
  - Libraries - Electric Utilities
  - Banks - Phone Companies
  - Universities - Cable companies

Comparison Based Problems

- Records can be easily identified by unique keys which are finite length strings over a chosen alphanumeric characters
- Examples include: social security number, university student/faculty ID, phone number, automobile VIN number, automobile license plate ID, to mention a few
- A set of records are ordered based on the chosen keys that point to these records
- We have already covered the search problem
  - Sequential Search
  - Binary Search

Merging Problem

- Let $A=\{a_1,a_2,\ldots,a_n\}$ and $B=\{b_1,b_2,\ldots b_m\}$ be two sorted files
- The goal is to create a new sorted file $c=\{c_1,c_2,\ldots c_{n+m}\}$ by merging/combining the two files A and B
- The need for merging arise from a number of directions
- First, there are sorting algorithm whose basic operation is merging. The well known two-way merge sort is a standard example- refer to the sorting algorithms

Merging Problem

- Second, a bank may have n accounts, where n is very large. On a given day, there may be m (< n) transactions. The bank has to update the database of accounts to reflect the current transaction before the start of the next business day.
- This is often done by merging the information in the current transaction file with the database
- Banks would do such merging at the end of every business day

Merging Problem

- There are two versions of this algorithm
- The first is when $n=m$, that is, files are of equal size
- The second is when $n \neq m$ and $n >> m$

Equal Size

A) Merging two sorted files of equal size

- Let A and B be two sorted files with k-distinct elements each
- Let k=3
  - $a_1 < a_2 < a_3$
  - $b_1 < b_2 < b_3$

Equal Size

- Compare a_1 with b_1. Let $a_1 > b_1$. Then $c_1 = a_1$
- Compare a_2 with b_1. Let $b_1 > a_2$. Then $c_2 = b_1$
- Compare a_2 with b_2. Let $a_2 > b_2$. Then $c_3 = a_2$
- Compare a_3 with b_2. Let $b_2 > a_3$. Then $c_4 = b_2$
- Compare a_3 with b_3. Let $a_3 > b_3$. Then $c_5 = a_3$
- Clearly, $c_6 = b_3$

Equal Size

- Thus, the sorted list is

  $a_1 < b_1 < a_2 < b_2 < a_3 < b_3$

- This case corresponds to the worst case, which requires $2k - 1 = 2 * 3 - 1 = 5$ operations

- Hence, it takes $T(k) = 2k - 1$ operations to merge two sorted files of size k

Homework

1. When do we have the best case?
2. What happens when the elements are not distinct?
3. Explore merging two sorted files of unequal sizes

Basic Sorting Algorithms

- We now describe a class of simple sorting algorithms
- Sorting by Selection
- Sorting by Insertion
  - Using sequential search
  - Using binary search
- Bubble Sort – Sorting and permutations

Basic Sorting Algorithms

- Let $x = \{x_1, x_2, \ldots, x_n\}$ be a set of n distinct elements
- The goal is to arrange the elements of x in the descending order, starting from the maximum occupying the first location and the minimum occupying the last

Algorithm 1: Sorting by Selection

- Let T(n) be the time required to sort a file of n numbers using this algorithm
- Recall from a previous lecture that we can find the maximum of n items in (n-1) comparisons
- The first step is to find the first maximum using (n-1) operations

Algorithm 1: Sorting by Selection

- Once the maximum is found, swap this max from its current location with x_1
- We are left with a file of size (n-1) from location 2 to n to be sorted
- Now find the max of the elements from location 2 to n in (n-2) operations and swap that second max element from its current location with x_2

Algorithm 1: Sorting by Selection

- Repeat this process until we are left with a file of only two items at locations (n-1) and n. Their maximum can be found in one comparison.
- The total work done is

$$T(n) = (n-1) + (n-2) + \ \dots + 3 + 2 + 1$$

$$= \sum_{i=1}^{n-1} i \rightarrow (1)$$

Algorithm 1: Sorting by Selection

- Recall the basic formula for the sum of the arithmetic series:

$$\sum_{i=1}^{k} i = \frac{k\,(k-1)}{2} \rightarrow ②$$

- Applying ② to ①, the latter becomes:

$$T(n) = \frac{(n-1)(n-2)}{2} = \frac{1}{2}(n^2 - 3n + 2)$$

- Which is a polynomial in n of degree 2

Algorithm 1: Sorting by Selection

- Hence, this algorithm is said to have a quadratic complexity
- This is an <u>in-house</u> sorting- the elements are rearranged with a given array, so it does not require extra space

Homework

4. Plot T(n) vs n for $2 \leq n \leq 50$

5. Compute $\lim_{n\to\infty} \frac{T(n)}{n^2}$

6. Let $f_1(n) = \frac{3}{4}n^2$, let $f_2(n) = \frac{1}{4}n^2$.
   - Plot $f_1(n)$, $T(n)$, $f_2(n)$ on the same panel for $2 \leq n \leq 50$
   - For what values of n do you observe the following relation:

$$f_2(n) < T(n) < f_1(n)$$

Algorithm 2a: Sorting by Insertion using Sequential Search

- Let $x = \{x_1, x_2, \dots , x_n\}$ be a set of n distinct elements
- Start with an empty array Y of size n
- Insert x_1 as the first element in the array Y. Now we have a sorted array with one element, and $Y_1 = X_1$
- Then, insert x_2 into this array Y to create an array that contains x_1 and x_2 in the sorted order

Algorithm 2a: Sorting by Insertion using Sequential Search

- To accomplish this, x_2 is compared with y_1. If $x_2 > y_1$, then x_2 is copied into the second location of Y. Else, $x_2 < y_1$. In this case, first y_1 is copied into the location 2 of Y (called data movement) and x_2 is dropped into the hole in location 1, to obtain y_1, y_2 which is sorted

Algorithm 2a: Sorting by Insertion using Sequential Search

- Consider the process of inserting x_3 in its right place in Y. We have to find the right location to insert x_3 so that after insertion we get a sorted file with 3 elements $\{y_1, y_2, y_3\}$ appearing in the sorted order occupying the first three locations in Y

Algorithm 2a: Sorting by Insertion using Sequential Search

- Now, inductively consider that we have thus inserted the first k elements $\{x_1, x_2, \ldots, x_k\}$ into the first k locations in the sorted order
- Our problem is to find the "right" place to insert x_{k+1} into this file in Y so that after insertion, it will result in a sorted file of size (k+1) containing $\{x_1, x_2, \ldots, x_{k+1}\}$ in the sorted order.

Algorithm 2a: Sorting by Insertion using Sequential Search

- There are two ways to do this:

1. Sequential Search (SS)
   - X_{k+1} is compared with Y_1 through Y_k sequentially to find the right place
   - This ignores the underlying order in Y
2. Binary Search (BS)
   - Since the first k elements of Y are sorted, use binary search to find the correct place for x_{k+1}

Algorithm 2a: Sorting by Insertion using Sequential Search

- We consider the SS based version first
- In the worst case, it would take k comparisons
- When will this event occur?

Algorithm 2a: Sorting by Insertion using Sequential Search

- Once the correct place is found, then it would also involve data movement to create a “hole” into which x_{k+1} is to be dropped so that it will result in a sorted file of size (k+1) in the first (k+1) locations in Y

Homework

7. In the insertion sort, when will the data movement be a maximum?
8. Does the maximum data movement occur when the number of comparisons needed to find the correct place is a maximum? Explain/explore.

Algorithm 2a: Sorting by Insertion using Sequential Search

- Let T(n) be the total amount of comparisons needed to sort a file using insertion sort where SS is used
- Thus, (excluding the time for data movement) the total time (in the worst case) is

$$T(n) = \sum_{k=1}^{n} k$$

$$= \frac{n(n+1)}{2} = \frac{1}{2}(n^2 + n) \rightarrow 1$$

- This is known as the $O(n^2)$ algorithm

Homework

9. Compute $\lim_{n\to\infty} \frac{T(n)}{n^2} = \frac{1}{2}(n^2 + n)$
10. Plot T(n) vs. n for $2 \le n \le 50$
11. Compare this with selection sort

Algorithm 2b: Sorting by Insertion using Binary Search

- Now we consider the binary search (BS) based version
- At stage k, using binary search it would require in the worst case $\lceil \log_2 k \rceil$ comparisons to find the correct location where x_{k+1} is to be inserted to create a new sorted list of (k+1) elements
- Hence,

$$T(n) = \sum_{k=2}^{n} \lceil \log_2 k \rceil$$

Homework

12. Why is the lower index in the summation 2 and not 1?
13. Verify $\lfloor x \rfloor \leq x \leq \lceil x \rceil$ for any real number x
14. Verify $\lceil x \rceil < x + 1$

Algorithm 2b: Sorting by Insertion using Binary Search

- Then,

$$T(n) = \sum_{k=2}^{n} \lceil \log_2 k \rceil < \sum_{2}^{n} (\log_2 k + 1)$$

$$T(n) < \sum_{k=2}^{n} 1 + \sum_{k=2}^{n} \log_2 k \rightarrow 1$$

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

$$\sum_{k=2}^{n} \log_2 k = \log_2 2 + \log_2 3 + \ldots + \log_2 n$$

$$= \log_2 (2 * 3 * \cdots * (n-1)n)$$

$$= \log_2 (n!)$$

Algorithm 2b: Sorting by Insertion using Binary Search

- Hence,

$$T(n) = \log_2(n!) + (n-1) \rightarrow ②$$

- It will be proved later that

$$\log_2(n!) = c_1 n \log_2 n + c_2 n + c_3 \log_2 n + c_4 \rightarrow ③$$

where c_1, c_2, c_3, c_4 are known constants

Algorithm 2b: Sorting by Insertion using Binary Search

- Combining:

$$T(n) = c_1 n \log_2 n + c'_2 n + c_3 \log_2 n + c'_4 \rightarrow ④$$

- This algorithm is said to be an O$(n \log n)$algorithm since the leading term in T(n) in ④ is $n \log n$

Homework

15. Prove the following:

a) $\log_2 a + \log_2 b = \log_2 ab$

b) $\log_2 a - \log_2 b = \log_2 {}^{a}/_{b}$

c) $\log_2 a^2 = n \log_2 a$

d. $\log_2 \frac{1}{a} = -\log_2 a$

e. $a^{\log_b n} = n^{\log_b a}$

f. $\log_e a = \log_2 a * \log_e 2$

g. $\log_a n = \log_b n * \log_a b$

Homework

- Stirling's approximation for n! is

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

- Error in this approximation $e(n)$ is

$$e(n) = n! - \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

- Normalized error $\bar{e}(n)$is

$$\bar{e}(n) = \frac{e(n)}{n!}$$

16. Plot the value of $\bar{e}(n)$ vs n for $2 \le n \le 20$. Explain what you find

Homework

16. Using Stirling's approximation, compute $\log_2(n!)$
17. Using this, find the values of the coefficients in ③ and ④

18. Plot T(n) vs n for $2 \le n \le 50$ for the two versions of the sorting algorithm- using insertion sort and binary search based, which are n^2 and $n \log n$ algorithms respectively.
19. Which one is better? Explain.

Algorithm 3: Bubble Sort

- Let $x = \{x_1, x_2, \ldots, x_n\}$ be an input array of n distinct elements
- Sort in the descending order using bubble sort, where the max is in location 1 and the min is in location n
- First, compare x_1 and x_2, and if $x_1 > x_2$, then leave them where they are. If not, $x_1 < x_2$ and swap x_1 and x_2 so that the minimum of $\{x_1,x_2\}$ is in location 2.

Algorithm 3: Bubble Sort

- Then, compare x_2 with x_3. If $x_3 < x_2$, leave them as they are. Else, $x_2 < x_3$ and swap x_2 and x_3. Thus, the occupant of the location 3 is the minimum of $\{x_1,x_2,x_3\}$.
- Continue this way, until you reach the n^{th} location, at which time the minimum of the file has <u>migrated</u>, or <u>bubbled</u>, to the last and is in its correct location

Algorithm 3: Bubble Sort

- This first pass through the array takes (n-1) comparisons and in turn, we have moved the minimum to the correct location.
- The second pass consists in repeating the same process starting again from location 1, but now going only up to location (n-1).
- Why?
- At the end of the second pass, after performing (n-2) comparisons, the second minimum migrates to location (n-1).

Algorithm 3: Bubble Sort

- Inductively, the kth pass would require $(n-k-1)$ comparisons, which results in the kth minimum occupying the location $(n-k+1)$.
- At the $(n-1)^{th}$ pass, the $(n-1)^{th}$ minimum (which is also the second maximum) will occupy $(n - (n-1) + 1) = 2^{nd}$ location after $(n - (n-1)) = 1$ comparisons.
- This leaves the nth minimum, which is the maximum, in location 1, giving us a sorted file in the descending order.

Algorithm 3: Bubble Sort

- The total work T(n) is given by

$$T(n) = (n-1) + (n-2) + \ \dots + (n-k) + \ \dots + 2 + 1$$

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

$$= \frac{1}{2}[n^2 - 3n + 2]$$

- Which is an $O(n^2)$ algorithm

Homework

20. It must be instructive to compare this algorithm with the selection sort algorithm.

Relation Between Sorting and Permutations

- Let $S = \{1,2,3,\ldots,n\}$. A permutation $p{:}S \rightarrow S$ is an one to one and onto map of S.
- That is, if $p(i) = p_i$ and $p(j) = p_j$, then $i \neq j \Leftrightarrow p_i \neq p_j$
- Let S_n denote the set of all n! permutations over S

Sorting and Permutations

- Permutation p is represented as a 2D array:

p =

1	2	3	…	n
p_1	p_2	p_3	…	p_n

- Identity permutation:

I =

1	2	3	…	n
1	2	3	…	N

Sorting and Permutations

- Given an input array $\{p_1, p_2, \ldots, p_i, \ldots, p_n\}$ where $p_i \in S$ and $p_i \neq p_j$ if $i \neq j$, sorting this array in increasing order is equivalent to converting/transforming the permutation p into an identity permutation
- <u>Inversion</u> in a permutation: If (i, p_i) and (j, p_j) are two pairs such that when $i<j$, then $p_i > p_j$, these two pairs are said to constitute an inversion.

Sorting and Permutations

- Consider p =

1	2	3	4
2	4	3	1

→ ①

- It can be verified that, of the six possible pairs, $\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$ and $\begin{bmatrix} 1 & 3 \\ 2 & 3 \end{bmatrix}$ are <u>not</u> inversions

- The pairs $\begin{bmatrix} 1 & 4 \\ 2 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 3 \\ 4 & 3 \end{bmatrix}, \begin{bmatrix} 2 & 4 \\ 4 & 1 \end{bmatrix}, \begin{bmatrix} 3 & 4 \\ 4 & 1 \end{bmatrix}$ are inversions
- Hence the total number of inversions in this p is 4.

Sorting and Permutations

- Since the number of inversions is an integer which is either even or odd, we define a new function called the PARITY function, defined over the set of all n! permutations of S as PARITY : $S_n \rightarrow \{+1, -1\}$

- Thus, $\text{PARITY}(p) = \begin{cases} +1 \text{ if the total number of inversions in p is even} \\ -1 \text{ if the total number of inversions in p is odd} \end{cases}$

- Thus, for p in 1, PARITY(p) = +1

Homework

21. What is the parity of the identity permutation?

- Let n=3 and list the set of all 3!=6 permutations S_3 over S={1,2,3}

22. Compute the parity of each of these permutations.

23. Verify that $\frac{1}{2}n!$ permutations are odd and the rest are even.

Homework

- Let p = $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \end{bmatrix}$ and q = $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \end{bmatrix}$
- Define the permutation r as r= $\begin{bmatrix} 1 & 2 & 3 & 4 \\ r_1 & r_2 & r_3 & r_4 \end{bmatrix}$ where r(i) = r_i = p(q(i)), which is a composition of functions p and q
- Define a permutation s as s= $\begin{bmatrix} 1 & 2 & 3 & 4 \\ s_1 & s_2 & s_3 & s_4 \end{bmatrix}$ where s(i) = s_i = q(p(i))

24. Relate s and r. Is s=r? What is the parity of s?

Homework

25. Let n=8. Generate a random permutation p. Compute the number of inversions in p. Verify that the number of comparisons needed to sort $\{p_i\}$ in the increasing order is equal to the number of inversions in p
26. Find the number of permutations $p \in S_8$ with a maximum number of inversions

Algorithm Design Techniques

- There are only a handful of basic cooking techniques that are common to all cuisines of the world

- Chopping
- Boiling
- Broiling
- Frying
- Sautéing
- Baking
- Grilling
- Soaking
- Pressure cooking
- Grinding

Algorithm Design Techniques

- Likewise, there are only a few known techniques for algorithm design

1. Divide and Conquer (DC)
2. Dynamic Programming (DP)
3. Greedy strategies (GS)

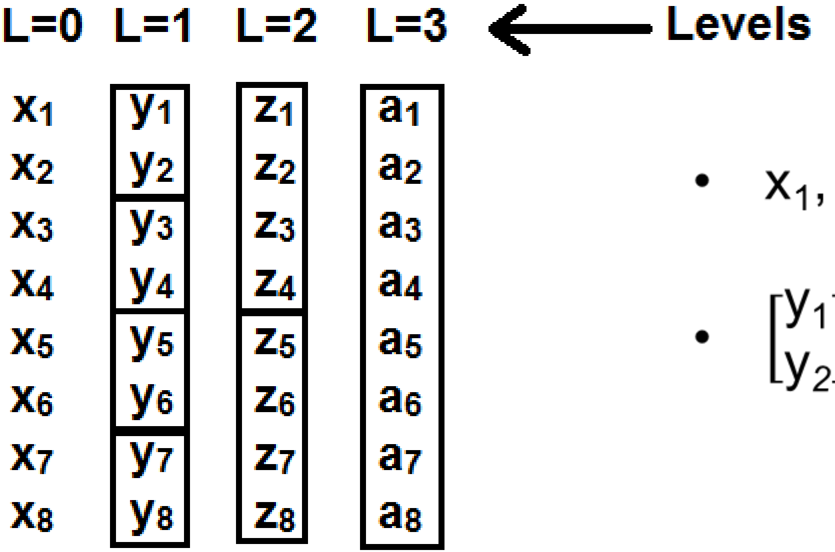
- We will illustrate the power of these techniques by solving a variety of problems drawn from different areas

Divide and Conquer

- We illustrate the power of the DC technique by using it to analyze the well known 2-way merge sort
- Consider an input file $x=\{x_1, x_2, \ldots, x_n\}$ to be sorted, where $n = 2^k$

Divide and Conquer

- We illustrate with an example with k=3



- x_1, x_2 are merged into $\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$
- $\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$ and $\begin{bmatrix} y_3 \\ y_4 \end{bmatrix}$ are merged into $\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix}$

Divide and Conquer

- A bottom up analysis first
- The input file is at level $L = 0$, and the output file is at level $L = k = \log_2 n$
- The input file at level $L = 0$ is considered to be made up of n (=8) sorted subfiles, each of size 1

Divide and Conquer

- To move up to the level 1, merge $\frac{n}{2}$ pairs of files at level 0 to create $\frac{n}{2}$ sorted files, each of size 2.
- To move to level 2, merge the $\frac{n}{2^2}$ pairs of files to create $\frac{n}{2^2}$ sorted files, each of size 2^2, at level 2

Divide and Conquer

- Inductively, recall that there are $\frac{n}{2^L}$ sorted files each of size 2^L at level L. Merge $\frac{n}{2^{L+1}}$ pairs of these files to create $\frac{n}{2^{L+1}}$ sorted files, each of size 2^{L+1} at level $(L+1)$
- Clearly, there are $\frac{n}{2^{k-1}}$ = 2 files, each of size 2^{k-1}, at level k-1. Merge the $\frac{n}{2^k}$ = 1 pair of files at level L=1 to create $\frac{n}{2^k} = 1$ file of size n=2^k at level k

Divide and Conquer

- This figure illustrates this process for n=8
- Work done to move from level L to level (L+1):
- There are 2^L sorted subfiles, each of size 2^L, at level L
- To merge a pair of these files, it takes $(2 * 2^L - 1) = (2^{l+1} - 1)$ operations.
- Since there are $\frac{n}{2^{L+1}}$, the total work done is:

$$\frac{n}{2^{L+1}}[2^{l+1} - 1] = n - \frac{n}{2^{L+1}}$$

Divide and Conquer

- Total work

$$T(n) = \sum_{L=0}^{k-1}(n - \frac{n}{2^{L+1}})$$

$$= kn - n\sum_{L=0}^{k-1}\frac{1}{2^{L+1}} \rightarrow (1)$$

Divide and Conquer

- Consider the geometric sum with |x|<1:

$$\sum_{L=0}^{k-1} x^{L+1} = \sum_{i=1}^{k} x^{i} = \sum_{i=0}^{k} x^{i} - 1$$

$$= \frac{1-x^{k+1}}{1-x} - 1 = \frac{x}{1-x}(1 - x^{k})$$

- When x = ½:

$$\sum_{L=0}^{k-1} \frac{1}{2^{L+1}} = \sum_{i=0}^{k} \frac{1}{2^{i}} = \frac{1/2}{1-1/2}\left[1 - \frac{1}{2^{k}}\right] = (1 - \frac{1}{n}) \rightarrow ②$$

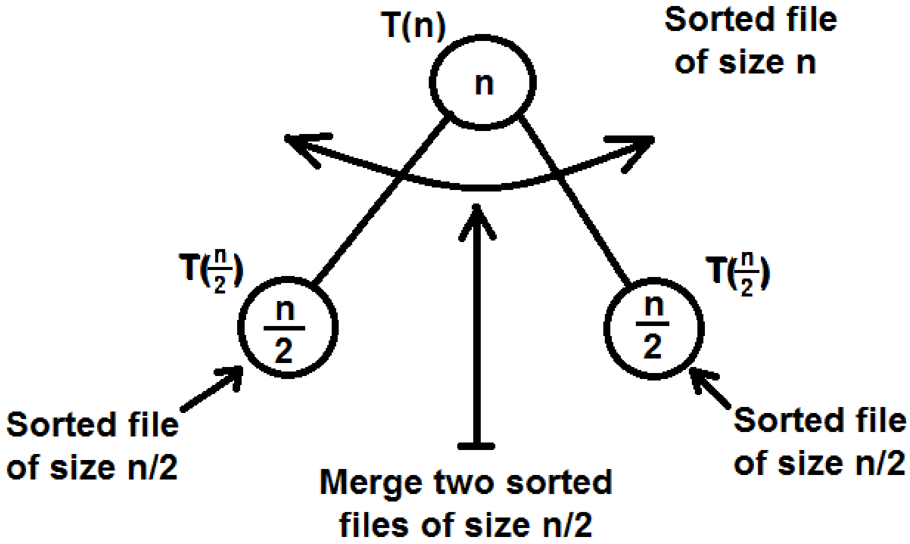
- Since k=$\log_2$n, substituting ② in ①

$$T(n) = n\log_2 n - n + 1 \rightarrow ③$$

- That is, 2-way merge sort is an O($n \log n$) algorithm

Divide and Conquer

- A top-down recursive view of the above algorithm
- The sorted file of size n at level k can be obtained by merging two sorted files, each of size $\frac{n}{2}$ at level (k-1)



Divide and Conquer

- If T(n) is the cost of sorting a file of size n, then T($\frac{n}{2}$) is the cost of sorting a file of size n/2 using the same algorithm
- Accordingly,

$$T(n) = \begin{Bmatrix} Total\ cost\ of \\ sorting\ two\ subfiles \\ of\ size\ {}^{n}/_{2} \end{Bmatrix} + \begin{Bmatrix} Cost\ of\ merging\ two \\ sorted\ files, each \\ of\ size\ {}^{n}/_{2} \end{Bmatrix}$$

$$T(n) = 2T\left(\frac{n}{2}\right) + \left(2 * \frac{n}{2} - 1\right)$$

$$= 2T\left(\frac{n}{2}\right) + (n-1) \rightarrow ④$$

Divide and Conquer

- When n=2, then

  $T(2) = 1 \rightarrow$ ⑤

  is the boundary condition that prescribes the cost of sorting a file with n=2 items using this method
- Equation ④ is called a first-order linear recurrence relation with ⑤ as its boundary condition

Divide and Conquer

- We can readily identify the DC here
- The input file of size $n = 2^k$ to be sorted is divided into two subfiles, each of size $\frac{n}{2} = 2^{k-1}$
- First, sort these two subfiles, each costing $T(\frac{n}{2})$. The total cost of solving the subproblems is $2T(\frac{n}{2})$

Divide and Conquer

- To get the final sorted file, we need to combine these two sorted files by merging them at a cost $\left(2 * \frac{n}{2} - 1\right) = n - 1$.
- Combining these two types of cost, we get the recurrence ④

Divide and Conquer

- Remark: DC is a fundamental strategy used in manufacturing processes.
- An automobile has n ($\approx$ 2000) components
- A manufacturer first produces L copies of each of these n parts and ships them to the assembly plant
- The cars are assembled using the idea of pipelining, which in turn leads to mass production

Divide and Conquer

- It is estimated that on average, a car moves out of the assembly line roughly every 30 inutes
- Thus:
  - Cost of production of car = {sum of n car parts} + {cost of assembly} → ⑥
- Of course, the MSRP is the sum of the actual cost in ⑥ + the profit
- Before solving the recurrence in ④, let us examine a few more examples of the application of DC

DC in Binary Search

- Let $n = 2^{k-1}$ and consider the process of searching for an item Y in a sorted file X of size n.
- If T(n) is the cost of searching for an item in a sorted file of size n, after the first comparison, in the worst case, it reduces to searching a file of size $\left\lceil \frac{n}{2} \right\rceil$.

DC in Binary Search

- Hence,

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \rightarrow ⑦$$

where T(1) = 1 is the boundary condition

- Again, ⑦ is a first order linear recurrence

DC in Selection Sort

- In this algorithm, we sort by successively finding the kth maximum for k=1 to n-1
- Recall that it takes (n-1) operations to find the first maximum and we are left with an unsorted file of size (n-1)
- Hence,

$$T(n) = T(n-1) + (n-1) \rightarrow ⑧$$

where T(2) = 1. (The cost of sorting a file of size 2 is 1)

- Again, ⑧ is a linear first-order recurence relation

DC in Selection Sort

- We have seen three types of recurrences:

a) $T(n) = 2T\left(\frac{n}{2}\right) + (n-1),\quad \mathrm{T}(2) = 1,\ \mathrm{n} = 2^k \geq 2$

b) $T(n) = T\left(\left\lceil\frac{n}{2}\right\rceil\right) + 1,\quad \mathrm{T}(1) = 1,\ \mathrm{n} = 2^k - 1 \geq 1$

c) $T(n) = T(n-1) + (n-1),\quad \mathrm{T}(2) = 1,$ any $n \geq 2$