[CSE3081(2반)] 알고리즘 설계와 분석

2020학년도 2학기

강의자료

(2020.10.06 화요일)

서강대학교 공과대학 컴퓨터공학과
임 인 성 교수

[주제 3]
Divide-and-Conquer Techniques
and
Sorting Techniques

https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms

Comparison Sorts

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	No	Partitioning	Quicksort is usually done in-place with $O(\log n)$ stack space.[5][6]
Merge sort	$n \log n$	$n \log n$	$n \log n$	n	Yes	Merging	Highly parallelizable (up to $O(\log n)$ using the Three Hungarians' Algorithm).[7]
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	$O(n+d)$, in the worst case over sequences that have d inversions.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with $O(n)$ extra space or when using linked lists.[11]
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.

Insertion Sort: Example 1

[0]:	15	10	3	1	14	19	5	11	7	16	18	4	9	2	8	0	12	17	6	13
[1]:	10	15	3	1	14	19	5	11	7	16	18	4	9	2	8	0	12	17	6	13
[2]:	3	10	15	1	14	19	5	11	7	16	18	4	9	2	8	0	12	17	6	13
[3]:	1	3	10	15	14	19	5	11	7	16	18	4	9	2	8	0	12	17	6	13
[4]:	1	3	10	14	15	19	5	11	7	16	18	4	9	2	8	0	12	17	6	13
[5]:	1	3	10	14	15	19	5	11	7	16	18	4	9	2	8	0	12	17	6	13
[6]:	1	3	5	10	14	15	19	11	7	16	18	4	9	2	8	0	12	17	6	13
[7]:	1	3	5	10	11	14	15	19	7	16	18	4	9	2	8	0	12	17	6	13
[8]:	1	3	5	7	10	11	14	15	19	16	18	4	9	2	8	0	12	17	6	13
[9]:	1	3	5	7	10	11	14	15	16	19	18	4	9	2	8	0	12	17	6	13
[10]:	1	3	5	7	10	11	14	15	16	18	19	4	9	2	8	0	12	17	6	13
[11]:	1	3	4	5	7	10	11	14	15	16	18	19	9	2	8	0	12	17	6	13
[12]:	1	3	4	5	7	9	10	11	14	15	16	18	19	2	8	0	12	17	6	13
[13]:	1	2	3	4	5	7	9	10	11	14	15	16	18	19	8	0	12	17	6	13
[14]:	1	2	3	4	5	7	8	9	10	11	14	15	16	18	19	0	12	17	6	13
[15]:	0	1	2	3	4	5	7	8	9	10	11	14	15	16	18	19	12	17	6	13
[16]:	0	1	2	3	4	5	7	8	9	10	11	12	14	15	16	18	19	17	6	13
[17]:	0	1	2	3	4	5	7	8	9	10	11	12	14	15	16	17	18	19	6	13
[18]:	0	1	2	3	4	5	6	7	8	9	10	11	12	14	15	16	17	18	19	13
[19]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Insertion Sort: Example 2

Insertion	$O(n+d)$, in the worst case over sequences that have d inversions.

When does the insertion sort run fast?

[0]:	0	1	4	3	2	7	6	5	8	11	10	9	13	12	14	17	16	15	18	19
[1]:	0	1	4	3	2	7	6	5	8	11	10	9	13	12	14	17	16	15	18	19
[2]:	0	1	4	3	2	7	6	5	8	11	10	9	13	12	14	17	16	15	18	19
[3]:	0	1	3	4	2	7	6	5	8	11	10	9	13	12	14	17	16	15	18	19
[4]:	0	1	2	3	4	7	6	5	8	11	10	9	13	12	14	17	16	15	18	19
[5]:	0	1	2	3	4	7	6	5	8	11	10	9	13	12	14	17	16	15	18	19
[6]:	0	1	2	3	4	6	7	5	8	11	10	9	13	12	14	17	16	15	18	19
[7]:	0	1	2	3	4	5	6	7	8	11	10	9	13	12	14	17	16	15	18	19
[8]:	0	1	2	3	4	5	6	7	8	11	10	9	13	12	14	17	16	15	18	19
[9]:	0	1	2	3	4	5	6	7	8	11	10	9	13	12	14	17	16	15	18	19
[10]:	0	1	2	3	4	5	6	7	8	10	11	9	13	12	14	17	16	15	18	19
[11]:	0	1	2	3	4	5	6	7	8	9	10	11	13	12	14	17	16	15	18	19
[12]:	0	1	2	3	4	5	6	7	8	9	10	11	13	12	14	17	16	15	18	19
[13]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	17	16	15	18	19
[14]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	17	16	15	18	19
[15]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	17	16	15	18	19
[16]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16	17	15	18	19
[17]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
[18]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
[19]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

이러한 insertion sort의 성질을 quick sort의 성능 향상에 활용하자.

Insertion Sort: Implementation

```
void insertion_sort(int *A, int n) {
  int i, j, tmp;

  for (i = 1; i < n; i++) {
    tmp = A[i];
    j = i;
    while ((j > 0) && (tmp < A[j - 1])) {
      A[j] = A[j - 1];
      j--;
    }
    A[j] = tmp;
  }
}
```

[0]:	**15**	10	3	1	14	19	5	11
[1]:	**10**	**15**	3	1	14	19	5	11
[2]:	**3**	**10**	**15**	1	14	19	5	11
[3]:	**1**	**3**	**10**	**15**	14	19	5	11
[4]:	**1**	**3**	**10**	**14**	**15**	19	5	11
[5]:	**1**	**3**	**10**	**14**	**15**	**19**	5	11
[6]:	**1**	**3**	**5**	**10**	**14**	**15**	**19**	11

Sort a list of elements by iteratively inserting a next element in a progressively growing sorted array.

$$T(n) = O(n^2)$$

Insertion Sort: Implementation 2 [K. Loudon]

```
#include <stdlib.h>
#include <string.h>
#include "sort.h"

int issort(void *data, int size, int esize, int (*compare)(const void *key1, const void *key2)) {
  char *a = data;
  void *key;
  int i, j;

  // Allocate storage for the key element.
  if ((key = (char *)malloc(esize)) == NULL) return -1;

  // Repeatedly insert a key element among the sorted elements.
  for (j = 1; j < size; j++) {
    memcpy(key, &a[j * esize], esize);
    i = j - 1;

    /* Determine the position at which to insert the key element. */
    while (i >= 0 && compare(&a[i * esize], key) > 0) {
      memcpy(&a[(i + 1) * esize], &a[i * esize], esize);  i--;
    }
    memcpy(&a[(i + 1) * esize], key, esize);
  }
  // Free the storage allocated for sorting.
  free(key);
  return 0;
}
```

Selection Sort: Example

[0]:	**17**	11	10	0	13	19	12	1	2	15	16	7	8	4	3	5	14	6	18	9
[1]:	0	**11**	10	17	13	19	12	1	2	15	16	7	8	4	3	5	14	6	18	9
[2]:	0	1	**10**	17	13	19	12	11	2	15	16	7	8	4	3	5	14	6	18	9
[3]:	0	1	2	**17**	13	19	12	11	10	15	16	7	8	4	3	5	14	6	18	9
[4]:	0	1	2	3	**13**	19	12	11	10	15	16	7	8	4	17	5	14	6	18	9
[5]:	0	1	2	3	4	**19**	12	11	10	15	16	7	8	13	17	5	14	6	18	9
[6]:	0	1	2	3	4	5	**12**	11	10	15	16	7	8	13	17	19	14	6	18	9
[7]:	0	1	2	3	4	5	6	**11**	10	15	16	7	8	13	17	19	14	12	18	9
[8]:	0	1	2	3	4	5	6	7	10	15	16	11	8	13	17	19	14	12	18	9
[9]:	0	1	2	3	4	5	6	7	8	15	16	11	10	13	17	19	14	12	18	9
[10]:	0	1	2	3	4	5	6	7	8	9	16	11	10	13	17	19	14	12	18	15
[11]:	0	1	2	3	4	5	6	7	8	9	10	11	16	13	17	19	14	12	18	15
[12]:	0	1	2	3	4	5	6	7	8	9	10	11	16	13	17	19	14	12	18	15
[13]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	17	19	14	16	18	15
[14]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	17	19	14	16	18	15
[15]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	19	17	16	18	15
[16]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	17	16	18	19
[17]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
[18]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
[19]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Selection sort	n^2	n^2	n^2	1	No	Selection

Selection Sort: Implementation

$$T(n) = O(n^2)$$

```
#define SWAP(a, b) { item_type tmp; tmp = a; a = b; b = tmp; }

void selection_sort(item_type *A, int n) {
  int i, j, cur;

  for (i = 0; i < n - 1; i++) {
    cur = i;
    for (j = i + 1; j < n; j++) {
      if (A[j] < A[cur])
        cur = j;
    }
    SWAP(A[i], A[cur]);  // what if i == cur?
  }
}
```

[0]:	**17**	11	10	0	13	19	12	1	2	15	16	7	8	4	3	5	14	6	18	9
[1]:	0	**11**	10	17	13	19	12	1	2	15	16	7	8	4	3	5	14	6	18	9
[2]:	0	1	**10**	17	13	19	12	11	2	15	16	7	8	4	3	5	14	6	18	9
[3]:	0	1	2	**17**	13	19	12	11	10	15	16	7	8	4	3	5	14	6	18	9
[4]:	0	1	2	3	**13**	19	12	11	10	15	16	7	8	4	17	5	14	6	18	9
[5]:	0	1	2	3	4	**19**	12	11	10	15	16	7	8	13	17	5	14	6	18	9
[6]:	0	1	2	3	4	5	**12**	11	10	15	16	7	8	13	17	19	14	6	18	9

Selection Sort: Run-Time Analysis

- **Worst case**
  - No. of comparisons:

$$\sum_{i=0}^{n-2}(n-i-1) = \frac{n(n-1)}{2} = O(\frac{n^2}{2})$$

  - No. of record assignments:

$$3(n-1) = O(3n)$$

```
    }
    SWAP(A[i], A[cur]);
}
```

- **Average case**
  - No. of comparisons:

$$\sum_{i=0}^{n-2}(n-i-1) = \frac{n(n-1)}{2} = O(\frac{n^2}{2})$$

  - No. of record assignments

$$3(n-1) = O(3n)$$

[0]:	**17**	11	10	0	13	19	12	1	2	15	16	7	8	4	3	5	14	6	18	9
[1]:	0	**11**	10	17	13	19	12	1	2	15	16	7	8	4	3	5	14	6	18	9
[2]:	0	1	**10**	17	13	19	12	11	2	15	16	7	8	4	3	5	14	6	18	9
[3]:	0	1	2	**17**	13	19	12	11	10	15	16	7	8	4	3	5	14	6	18	9
[4]:	0	1	2	3	**13**	19	12	11	10	15	16	7	8	4	17	5	14	6	18	9
[5]:	0	1	2	3	4	**19**	12	11	10	15	16	7	8	13	17	5	14	6	18	9
[6]:	0	1	2	3	4	5	**12**	11	10	15	16	7	8	13	17	19	14	6	18	9

If we code like "if (i != cur) SWAP(A[i], A[cur]);", what is the average cost?

Bubble Sort: Example

[0]:	15	19	9	4	14	3	6	10	0	12	7	5	8	17	2	16	11	1	18	13
[1]:	0	15	19	9	4	14	3	6	10	1	12	7	5	8	17	2	16	11	13	18
[2]:	0	1	15	19	9	4	14	3	6	10	2	12	7	5	8	17	11	16	13	18
[3]:	0	1	2	15	19	9	4	14	3	6	10	5	12	7	8	11	17	13	16	18
[4]:	0	1	2	3	15	19	9	4	14	5	6	10	7	12	8	11	13	17	16	18
[5]:	0	1	2	3	4	15	19	9	5	14	6	7	10	8	12	11	13	16	17	18
[6]:	0	1	2	3	4	5	15	19	9	6	14	7	8	10	11	12	13	16	17	18
[7]:	0	1	2	3	4	5	6	15	19	9	7	14	8	10	11	12	13	16	17	18
[8]:	0	1	2	3	4	5	6	7	15	19	9	8	14	10	11	12	13	16	17	18
[9]:	0	1	2	3	4	5	6	7	8	15	19	9	10	14	11	12	13	16	17	18
[10]:	0	1	2	3	4	5	6	7	8	9	15	19	10	11	14	12	13	16	17	18
[11]:	0	1	2	3	4	5	6	7	8	9	10	15	19	11	12	14	13	16	17	18
[12]:	0	1	2	3	4	5	6	7	8	9	10	11	15	19	12	13	14	16	17	18
[13]:	0	1	2	3	4	5	6	7	8	9	10	11	12	15	19	13	14	16	17	18
[14]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	15	19	14	16	17	18
[15]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	19	16	17	18
[16]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	19	17	18
[17]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	19	18
[18]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
[19]:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Bubble Sort: Implementation

$$T(n) = O(n^2)$$

```c
#define SWAP(a, b) { item_type tmp; tmp = a; a = b; b = tmp; }

void bubble_sort(item_type *A, int n) {
  int i, j;

  for (i = 0; i < n - 1; i++) {
    for (j = n - 1; j > i; j--) {
      if (A[j] < A[j - 1])
        SWAP(A[j], A[j - 1]);
    }
  }
}
```

```
[ 0]:   15  19   9   4  14   3   6  10   0  12   7   5   8  17   2  16  11   1  18  13
[ 1]:    0  15  19   9   4  14   3   6  10   1  12   7   5   8  17   2  16  11  13  18
[ 2]:    0   1  15  19   9   4  14   3   6  10   2  12   7   5   8  17  11  16  13  18
```

Bubble Sort: Run-Time Analysis

- **Worst case**
  - No. of comparisons:

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

  - No. of record assignments:

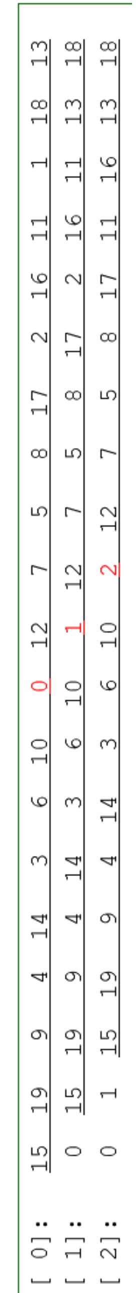
$$\sum_{i=1}^{n-1} 3i = \frac{3}{2}n(n-1) = O(\frac{3}{2}n^2)$$

- **Average case**
  - No. of comparisons:

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

  - No. of record assignments

$$\frac{1}{2}\sum_{i=1}^{n-1} 3i = \frac{3}{4}n(n-1) = O(\frac{3}{4}n^2)$$



Refer to The Art of Computer Programming (Vol. 3)

Cost Comparison

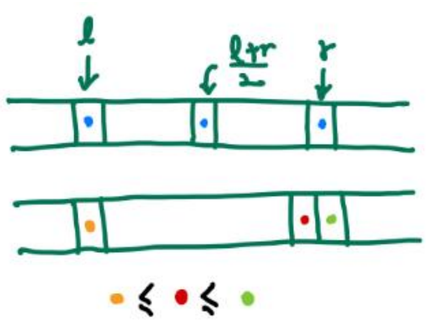
	Selection	Insertion	Bubble
# of comparisons	$n^2/2$	$n^2/4$ (average)	$n^2/2$ (average)
		$n^2/2$ (worst)	$n^2/2$ (worst)
# of record assignments	$3n$	$n^2/4$ (average)	$3n^2/4$ (average)
		$n^2/2$ (worst)	$3n^2/2$ (worst)

Comparison Sorts

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	No	Partitioning	Quicksort is usually done in-place with $O(\log n)$ stack space.[5][6]
Merge sort	$n \log n$	$n \log n$	$n \log n$	n	Yes	Merging	Highly parallelizable (up to $O(\log n)$ using the Three Hungarians' Algorithm).[7]
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	$O(n + d)$, in the worst case over sequences that have d inversions.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with $O(n)$ extra space or when using linked lists.[11]
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.

Improving the Performance of Quick Sort

- How can you **select a "good" pivot element?**
  - Choose a random element in the list.
  - Choose **the median of the first, middle, and final elements** in the list.
  - Choose **the median of the entire elements** in the list. (bad idea)
  - Etc.

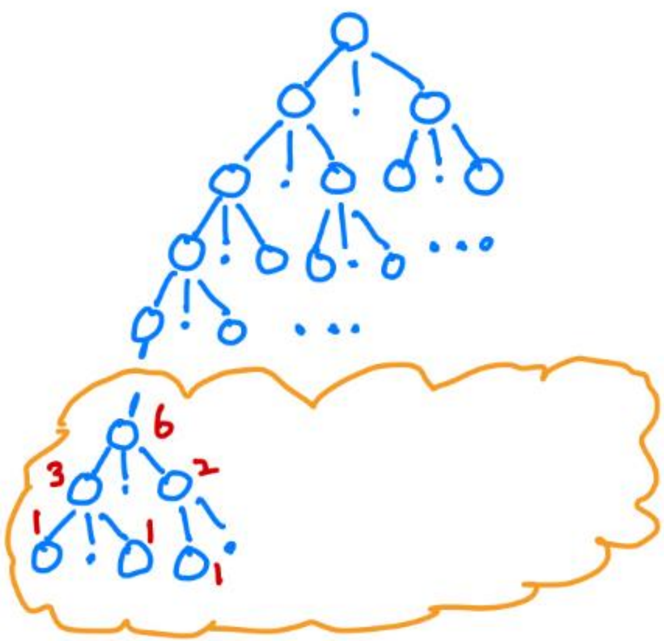


Program 7.4 Improved quicksort

Choosing the median of the first, middle, and final elements as the partitioning element and cutting off the recursion for small subfiles can significantly improve the performance of quicksort. This implementation partitions on the median of the first, middle, and final elements in the array (otherwise leaving these elements out of the partitioning process). Files of size 10 or smaller are ignored during partitioning; then, insertion from Chapter 8 is used to finish the sort.

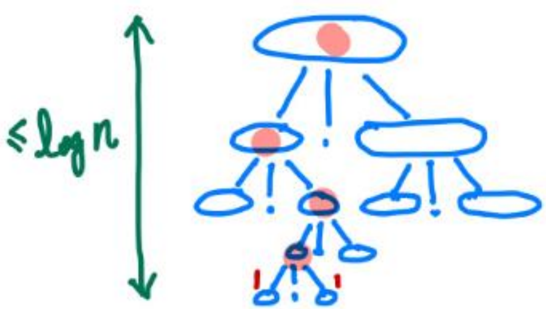
```
#define M 10
void quicksort(Item a[], int l, int r)
  { int i;
    if (r-l <= M) return;
    exch(a[(l+r)/2], a[r-1]);
    compexch(a[l], a[r-1]);
      compexch(a[l], a[r]);
        compexch(a[r-1], a[r]);
    i = partition(a, l+1, r-1);
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
  }
void sort(Item a[], int l, int r)
  {
    quicksort(a, l, r);
    insertion(a, l, r);
  }
```

- How can you **minimize the bookkeeping cost involved in the recursive calls?**
  - ✓ Much of the pushing and popping of the frame stack is unnecessary.
  - – **Lists of size smaller than M are ignored during quick sort, then do a single sorting pass at the end.**




```
#define M 10
void quicksort(Item a[], int l, int r)
  { int i;
    if (r-l <= M) return;
    exch(a[(l+r)/2], a[r-1]);
    compexch(a[l], a[r-1]);
      compexch(a[l], a[r]);
        compexch(a[r-1], a[r]);
    i = partition(a, l+1, r-1);
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
  }
void sort(Item a[], int l, int r)
  {
    quicksort(a, l, r);
    insertion(a, l, r);
  }
```

- How can you **minimize the bookkeeping cost involved in the recursive calls?**
  - ✓ Avoid making the recursive call on the larger subrange.
  - ✓ The depth of recursion <= O(log n)




```
quickSortTRO(E, first, last)
    int first1, last1, first2, last2;

    first2 = first; last2 = last;
    while (last2 − first2 > 1)
        pivotElement = E[first];
        pivot = pivotElement.key;
        int splitPoint = partition(E, pivot, first2, last2);
        E[splitPoint] = pivotElement;
        if (splitPoint < (first2 + last2) / 2)
            first1 = first2; last1 = splitPoint − 1;
            first2 = splitPoint + 1; last2 = last2;
        else
            first1 = splitPoint + 1; last1 = last2;
            first2 = first2; last2 = splitPoint − 1;
        quickSortTRO(E, first1, last1);
        // Continue loop for first2, last2.
    return;
```

first2

Example: Quick Sort

By courtesy of David R. Musser

```
Algorithm QUICKSORT(A, f, b)
                    Inputs: A, a random access data structure containing the sequence
                      of data to be sorted, in positions A[f], ..., A[b − 1];
                      f, the first position of the sequence
                      b, the first position beyond the end of the sequence
                    Output: A is permuted so that A[f] ≤ A[f+1] ≤ ... ≤ A[b − 1]
    QUICKSORT_LOOP(A, f, b)
    INSERTION_SORT(A, f, b)

Algorithm QUICKSORT_LOOP(A, f, b)
                    Inputs: A, f, b as in QUICKSORT
                    Output: A is permuted so that A[i] ≤ A[j]
                      for all i, j: f ≤ i < j < b and size_threshold < j − i
    while b − f > size_threshold
        do   p := PARTITION(A, f, b, MEDIAN_OF_3(A[f], A[f+(b−f)/2], A[b−1]))
             if (p − f ≥ b − p)
                 then QUICKSORT_LOOP(A, p, b)
                      b := p
                 else QUICKSORT_LOOP(A, f, p)
                      f := p
```

Average-case: $O(n \log n)$
Worst-case: $O(n^2)$

Performance Comparisons

By courtesy of David R. Musser

Table 1: Performance of Introsort, Quicksort, and Heapsort on Random Sequences (Sizes and Operations Counts in Multiples of 1,000)

Size	Algorithm	Comparisons	Assignments	Iterator Ops	Distance Ops	Total Ops
1	Introsort	11.9	9.4	52.9	1.2	75.4
	Quicksort	11.9	9.4	53.3	1.2	75.7
	Heapsort	10.3	15.5	136.1	159.1	320.9
4	Introsort	57.2	43.6	246.9	4.7	352.5
	Quicksort	57.2	43.6	248.6	4.6	354.0
	Heapsort	49.3	70.2	640.5	748.8	1508.8
16	Introsort	265.7	203.4	1130.6	18.5	1618.2
	Quicksort	265.7	203.4	1137.2	18.5	1624.8
	Heapsort	229.2	318.9	2945.1	3442.5	6935.7
64	Introsort	1235.1	934.6	5125.7	73.6	7369.0
	Quicksort	1235.1	934.6	5152.3	73.5	7395.5
	Heapsort	1044.7	1435.6	13316.9	15562.6	31359.7
256	Introsort	5644.4	4093.4	22965.6	293.5	32996.8
	Quicksort	5644.4	4093.4	23072.2	293.4	33103.4
	Heapsort	4691.0	6254.4	59411.1	69419.7	139776.3
1024	Introsort	24945.6	17805.8	100946.7	1177.1	144875.1
	Quicksort	24945.6	17805.8	101374.4	1176.4	145302.2
	Heapsort	20812.4	27065.6	262222.8	306349.8	616450.6

Quicksort: Implementation 2 [K. Loudon]

```
#include <stdlib.h>
#include <string.h>
#include "sort.h"

static int compare_int(const void *int1, const
    void *int2) {
// Compare two integers (used during median-of-
// three partitioning
  if (*(const int *)int1 > *(const int *)int2)
    return 1;
  else if (*(const int *)int1 <
              *(const int *)int2) return -1;
  else  return 0;
}

static int partition(void *data, int esize,
  int i, int k,
  int (*compare)(const void *key1, const void
    *key2)) {
  char *a = data;
  void *pval, *temp;
  int r[3];

  /*  Allocate storage for the partition value
      and swapping. */
  if ((pval = malloc(esize)) == NULL)
    return -1;
```

```
  if ((temp = malloc(esize)) == NULL) {
    free(pval);  return -1;
  }
  /* Use the median-of-three method to find
     the partition value.  */
  r[0] = (rand() % (k - i + 1)) + i;
  r[1] = (rand() % (k - i + 1)) + i;
  r[2] = (rand() % (k - i + 1)) + i;
  issort(r, 3, sizeof(int), compare_int);
  memcpy(pval, &a[r[1] * esize], esize);

  /* Create two partitions around the partition
     value.  */
  i--;  k++;

  while (1) {
     /* Move left until an element is found in
        the wrong partition. */           *
    do {  k--; }
      while (compare(&a[k * esize], pval) > 0);
     /* Move right until an element is found in
        the wrong partition. */
    do {  i++; }
      while (compare(&a[i * esize], pval) < 0);
```

```c
    if (i >= k) { break; }
    /* Stop partitioning when the left and
       right counters cross.  */
    else {
    /* Swap the elements now under the left and
       right counters.  */
      memcpy(temp, &a[i * esize], esize);
      memcpy(&a[i * esize], &a[k * esize],
               esize);
      memcpy(&a[k * esize], temp, esize);
    }
  }

  /* Free the storage allocated for
     partitioning. */
  free(pval);
  free(temp);
  /* Return the position dividing the two
     partitions. */
  return k;
}
```

```c
int qksort(void *data, int size, int esize,
    int i, int k,
    int (*compare)
   (const void *key1, const void *key2)) {

  int j;

  /* Stop the recursion when it is not possible
     to partition further. */
  if (i < k) {
    // Determine where to partition the elements
    if ((j = partition(data, esize, i, k,
                          compare)) < 0)
      return -1;
    // Recursively sort the left partition
    if (qksort(data, size, esize, i, j, compare)
          < 0)
      return -1;
    // Recursively sort the right partition
    if (qksort(data, size, esize, j + 1, k,
               compare) < 0)
      return -1;
  }
  return 0;
}
```

Selection of Both Maximum and Minimum Elements

- **Problem:** Find both the maximum and the minimum elements of a set containing n elements (assume $n = 2^m$ for some integer m). [Aho 2.6]

```
begin
    MAX ← any element in S;
    for all other elements x in S do
        if x > MAX then MAX ← x
end
```

$T(n) = (n\text{-}1) + (n\text{-}2) = 2n\text{-}3$ comparisons

$T(n) = 2T(n/2) + 2$ for $n > 2$,
$T(n) = 1$ for $n = 2$
$\rightarrow T(n) = (3/2)n - 2$ comparisons

This is the minimum!

```
    procedure MAXMIN(S):
1.  if ||S|| = 2 then
        begin
2.          let S = {a, b};
3.          return (MAX(a, b), MIN(a, b))
        end
    else
        begin
4.          divide S into two subsets S1 and S2, each with half the elements;
5.          (max1, min1) ← MAXMIN(S1);
6.          (max2, min2) ← MAXMIN(S2);
7.          return (MAX(max1, max2), MIN(min1, min2))
        end
```

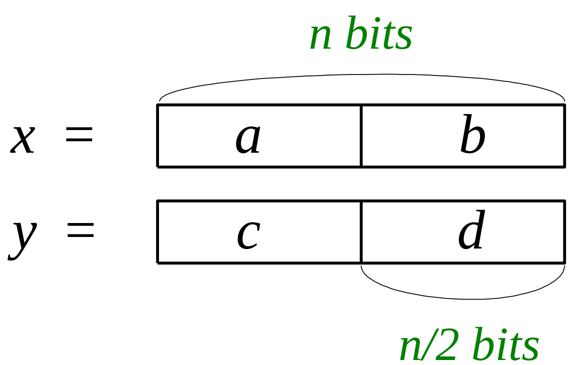
Fig. 2.12. Procedure to find MAX and MIN.

max1 max2
min1 min2

Multiplication of Two n-bit Numbers

[Aho 2.6]

- The traditional method requires $O(n^2)$ bit operations.
- A divide-and-conquer approach



$$
\begin{aligned}
xy &= (a2^{\frac{n}{2}} + b)(c2^{\frac{n}{2}} + d) \\
&= ac2^n + (ad + bc)2^{\frac{n}{2}} + bd
\end{aligned}
$$

```
u = (a+b)*(c+d);

v = a*c;

w = b*d;

z = v*pow(2, n) + (u-v-w)*pow(2, n/2) + w;
```

$T(n) = 1$ for $n = 1$
$T(n) = 3T(n/2) + cn$ for $n > 1$

$\rightarrow T(n) = O(n^{\log 3})$

$O(n^2) \rightarrow O(n^{1.59})$

$$
\begin{array}{r}
\overbrace{10110110}^{n} \\
\times\ 11001011 \\
\hline
n^2 \text{ mults}
\end{array}
$$

✓ Read [Neapolitan 2.6].

Selection of the *k*-th Smallest Element

- **Problem:** Given a sequence of S of *n* elements and an integer *k* (1 <= *k* <= *n*), find the *k*-th smallest element of *S*.

- **Solution 1:** Choose the smallest element repeatedly k times.

$$C = c(n-1) + c(n-2) + c(n-3) + \cdots + c(n-k) = c \cdot k \cdot n - c \cdot \frac{k(k+1)}{2}$$

If $k = \frac{n}{2}$, then $C = c \cdot \frac{n^2}{2} - c \cdot \frac{n^2+2n}{8} = O(n^2)$.

- **Solution 2:** Build a min-heap, and then extract the smallest element repeatedly k times.

$$C = c \cdot n + d \cdot k \cdot \log n$$

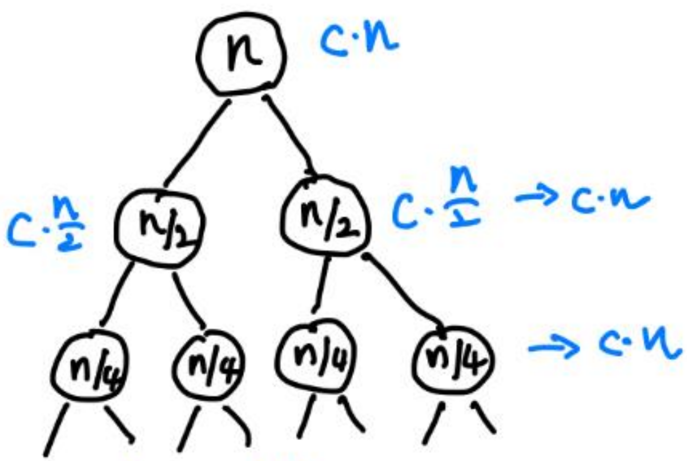
If $k = \frac{n}{2}$, then $C = c \cdot n + d \cdot \frac{n}{2} \cdot \log n = O(n \log n)$.

➢ **Can we design an O(n)-time algorithm?**

Observation

- **At least O(n) time is necessary.**

- **If we use a divide-and-conquer scheme like the merge sort,**



log · cn

$T(n) = 2T(n/2) + c \cdot n$

$\Rightarrow O(n \log n)$

What about T(n) = 3T(n/3) + cn?