CME 112- Programming Languages II Lecture 9: Sorting & Searching

Assist. Prof.Dr. Ümit ATİLA

Sorting

- Placing a group of data in descending or ascending order.
- Sorting data is very useful for computer systems
- Makes searching and listing a group of data faster and easier.
- Most popular sorting algorithms:
  - Insertion sort
  - Selection sort
  - Bubble sort
  - Quick sort

Bubble Sort

- Time complexity is O(n2)
- If *c* number of item of *n* items is not sorted time complexity is O(c n)
- Design of algorithm is easy but algorithm is not efficient.
- Can be used for small size lists or lists having mostly sorted items.

Bubble Sort

- Array to be sorted : [7,3,5,1,2]

• Hareket 1- Çevrim-1 [3,7,5,1,2] • Hareket 1- Çevrim-2 [3,5,7,1,2] • Hareket 1- Çevrim-3 [3,5,1,7,2] • Hareket 1- Çevrim-4 [3,5,1,2,7]	• Hareket 2- Çevrim-1 [3,5,1,2,7] • Hareket 2- Çevrim-2 [3,1,5,2,7] • Hareket 2- Çevrim-3 [3,1,2,5,7] • Hareket 2- Çevrim-4 [3,1,2,5,7]	• Hareket 3- Çevrim-1 [1,3,2,5,7] • Hareket 3- Çevrim-2 [1,2,3,5,7] • Hareket 3- Çevrim-3 [1,3,2,5,7] • Hareket 3- Çevrim-4 [1,3,2,5,7]	• Hareket 4- Çevrim-1 [1,3,2,5,7] • Hareket 4- Çevrim-2 [1,2,3,5,7] • Hareket 4- Çevrim-3 [1,2,3,5,7] • Hareket 4- Çevrim-4 **[1,2,3,5,7]**

Bubble Sort

```
void bubbleSort(int dizi[],int n)
{
    int gecici;

    for (int i = 0; i < n; i++)
    {
        for (int k = 0; k < n - 1 - i; k++)
        {
            if(dizi[k]>dizi[k+1])
            {
                gecici=dizi[k];
                dizi[k] = dizi[k + 1];
                dizi[k + 1] = gecici;
            }
        }
    }
}
```

Bubble Sort

```c
#include <stdio.h>

void bubbleSort(int [],int);
int main(void)
{
    int i=0,a[5];
    printf("Siralamak istediğin 5 sayi gir\n");
    while(i<5){
        scanf("%d",&a[i]);
        i++;
    }
    i=0;
    bubbleSort(a,5);

    printf("Bubble sort isleminden sonra...\n");
    while(i<5){
        printf("%d ",a[i]);
        i++;
    }
    return 0;
}
```

Insertion Sort

- Appropriate for inserting an item into an already sorted list of data.
- Complexity of inserting an item into an already sorted list of data: O (n)
- If list or array is not sorted complexity:O (n^2)

Insertion Sort

- Array to be sorted : [7,3,5,8,2]
- Initial state : [7][3,5,8,2]

Before	After
[7,3][5,8,2]	[3,7][5,8,2]
[3,7,5][8,2]	[3,5,7][8,2]
[3,5,7,8][2]	[3,5,7,8][2]
[3,5,7,8,2][ ]	[2,3,5,7,8][ ]

Insertion Sort Example

```
void insertionSort(int dizi[], int n)
{
    int ekle,k,i;
    for (i = 1; i < n; i++)
    {
        ekle = dizi[i];
        for (k = i - 1; k >= 0 && ekle <= dizi[k]; k--)
            dizi[k + 1] = dizi[k]; //Geriye kaydırma
        dizi[k + 1] = ekle;        //Uygun yer boşaltıldı eklendi
    }
}
```

Insertion Sort Example

```c
#include <stdio.h>

void insertionSort(int [],int);
int main(void)
{
    int i=0,a[5];
    printf("Siralamak istediğin 5 sayi gir\n");
    while(i<5){
        scanf("%d",&a[i]);
        i++;
    }
    i=0;
    insertionSort(a,5);

    printf("Insertion sort isleminden sonra...\n");
    while(i<5){
        printf("%d ",a[i]);
        i++;
    }
    return 0;
}
```

Selection Sort

- If an item is in its true place it does not change its order.
- Change of items is less in half sorted group of data.
- Take the first item in the list and exchange with the minimum item of others. Repeat this until the last item.

Before	After	
[][7,3,5,1,2]	[1][3,5,7,2]	1 and 7 exchanged
[1][3,5,7,2]	[1,2][5,7,3]	2 and 3 exchanged
[1,2][5,7,3]	[1,2,3][7,5]	3 and 5 exchanged
[1,2,3][7,5]	[1,2,3,5][7]	5 and 7 exchanged
[1,2,3,5][7]	[1,2,3,5,7][]	end

Selection Sort Example

```
void selectionSort(int dizi[],int n)
{
	int i,j;
	int index, enkucuk;
	for (i = 0; i < n - 1; i++)
	{
		enkucuk = dizi[n - 1];
		index = n - 1;
		for (j = i; j < n - 1; j++)
		{
			if (dizi[j] < enkucuk)
			{
				enkucuk = dizi[j];
				index = j;
			}
		}
		dizi[index] = dizi[i];
		dizi[i] = enkucuk;
	}
}
```

Selection Sort Example

```c
#include <stdio.h>

void selectionSort(int [],int);
int main(void)
{
    int i=0,a[5];
    printf("Siralamak istediğin 5 sayi gir\n");
    while(i<5){
        scanf("%d",&a[i]);
        i++;
    }
    i=0;
    selectionSort(a,5);

    printf("Selection sort isleminden sonra...\n");
    while(i<5){
        printf("%d ",a[i]);
        i++;
    }
    return 0;
}
```

Quick Sort

- Works on divide and conquer strategy
- The list is divided into two equal parts.
- Values smaller than middle value is collected on left side and others collected on right side.
- This process is repeated for each part.
- The algorithm is implemented with a two recursive call for each divided part individually.
- Despite it is the fastest algorithm, it may not be chosen for small number of items or mostly sorted items.

Quick Sort

```
void quickSort(int dizi[],int sol,int sag)
{
    int qSol,qSag,ortadaki,gecici;
    //Dizi ikiye parçalanıyor
    qSol=sol;
    qSag=sag;
    ortadaki = dizi[(sol+sag)/2];  //Sınır değer
    do
    {
        while(dizi[qSol]< ortadaki && qSol<sag)
            qSol++;
        while(ortadaki< dizi[qSag] && qSag>sol)
            qSag--;
        if(qSol<=qSag)
        {
            gecici = dizi[qSol];
            dizi[qSol] = dizi[qSag];
            dizi[qSag] = gecici;
            qSol++; qSag--;
        }
    }
    while(qSol<=qSag); //parçalama bitti
    if(sol<qSag) quickSort(dizi,sol,qSag);
    if(qSol<sag) quickSort(dizi,qSol,sag);
}
```

Quick Sort

```c
#include <stdio.h>

void quickSort(int [],int,int);
int main(void)
{
	int i=0,a[5];
	printf("Siralamak istediğin 5 sayi gir\n");
	while(i<5){
		scanf("%d",&a[i]);
		i++;
	}
	i=0;
	quickSort(a,0,4);

	printf("Quick sort isleminden sonra...\n");
	while(i<5){
		printf("%d ",a[i]);
		i++;
	}
	return 0;
}
```

Searching

- The process of finding a particular element of an array is called searching.
- Two searching techniques will be discussed
  - Linear search
  - Binary search

Linear Search

- Compares each element of the array with the search key.
- Since the array is not in any particular order, it is just as likely that the value will be found in the first element as in the last.
- In the worst case with N number of elements, the algorithm's complexity is O(N)
- It should not be used in large size arrays.

Linear Search

```
int linearSearch(int dizi[],int aranan,int n)
{
	for (int i = 0; i < n; i++)
	{
		if (dizi[i] == aranan)
			return i;
	}
	return -1;
}
```

Linear Search

```
#include <stdio.h>

int linearSearch(int [],int,int);
int main(void)
{
	int dizi[] = { 1, 3, 5, 7, 8, 10, 11 };
	int sonuc, aranan,i;
	for (i = 0; i < 7; i++)
		printf("%d ",dizi[i]);

	printf("Arananı giriniz:");
	scanf("%d",&aranan);

	sonuc = linearSearch(dizi, aranan,7);
	if (sonuc == -1)
		printf("\nAranan dizide yok\n");
	else
		printf(sonuc + ". sırada bulundu\n");
}
```

Binary Search

- The linear search works well for small or unsorted arrays.
- However for large arrays, linear search is inefficient.
- If the array is sorted the high speed binary search can be used.
- The binary search algorithm eliminates from consideration one-half of the elements in a sorted array after each comparison.

Binary Search

- The algorithm locates the middle element of the array and compares it to the search key.
  - If equal, match found
  - If key < middle, reduce problem for looking in first half of array
  - If key > middle, reduce problem for looking in last half of array
  - Repeat until the search key is equal to the middle element of a subarray, or until the subarray consists of one element that is not equal to the search key (i.e the search key is not found).

Binary Search

```
#include <stdio.h>

int binarySearch(int [],int,int,int);
int main(void)
{
	int dizi[] = { 1, 3, 5, 7, 8, 10, 11 };
	int sonuc, aranan,i;
	for (i = 0; i < 7; i++)
		printf("%d ",dizi[i]);

	printf("Arananı giriniz:");
	scanf("%d",&aranan);

	sonuc = binarySearch(dizi, aranan,0,6);
	if (sonuc == -1)
		printf("\nAranan dizide yok\n");
	else
		printf(sonuc + ". sırada bulundu\n");
}
```

Binary Search

```
int binarySearch(int dizi[],int aranan,int sol,int sag)
{
    int orta;
    while (sol <= sag)
    {
        orta = (sol + sag) / 2; //Ortadaki elemanın indisi hesaplanıyor
        if (aranan == dizi[orta])
            return orta;
        else if (aranan > dizi[orta])
            sol = orta + 1;
        else
            sag = orta - 1;
    }
    return -1;
}
```

Binary Search

- Very fast; at most n step, where 2^n > number of elements
- 30 element array takes at most 5 step ($2^5 > 30$ so at most 5 steps)
- In a worst case-scenario, searching an array of 1023 elements takes only 10 comparisons using a binary search.
- Repeatedly dividing 1024 by 2 yields the values 512, 256, 128, 64, 32, 16, 8, 4, 2 and 1.
- The number 1024 (2^{10}) is divided by 2 only 10 times to get the value 1.
- Dividing by 2 is equivalent to one comparison in the binary search algorithm.

Binary Search

- An array of 1048576 (2^{20}) elements takes a maximum of 20 comparisons to find the search key.
- An array of one billion elements takes a maximum of 30 comparisons to find the search key.
- This is a tremendous increase in performance over the linear search that required comparing the search key to an average of half of the array elements.
- For a one-billion-element array, this is a difference between an average of 500 million comparisons and a maximum of 30 comparisons!