Algorithm Lab With C
in Linux

Quick Sort, Randomized Quick Sort, Heap Sort

Anupam Pattanayak[1]

M.C.A., M. Tech.

Assistant Professor,

Department of Computer Science,

Raja N. L. Khan Women's College (Autonomous),

Midnapore, West Bengal

April 8, 2020

[1]anupam.pk@gmail.com

Contents

1 Quick Sort **1**
1.1 Randomized Quick Sort 3

2 Heap Sort **7**

1

Quick Sort

In quicksort, we chose a pivot element, and then place the pivot elements it's appropriate position had the list been sorted. So, all the elements that are left to the pivot element after the partition are $\leq$ elements in the left, and al the elements that are right to the pivot element after partition are $\geq$ pivot element.

Quicksort is named so because it sorts *quickly* in average. Worst-case time complexity of quicksort is $\mathcal{O}(n^2)$ where n is the input size. Average-case time complexity of quicksort is $\Theta(n \times \log_2 n)$.

Quicksort follows *divide-and-conquer* algorithm design paradigm. So, it can be solved recursively. Top-level structure of quicksort for integer array A[ ] is as follows:

```
 quicksort(int A[ ], int left, int right) {
   q = partition(A[ ], left, right);
   quicksort(A[ ], left, q-1);
   quicksort(A[ ], q+1, right);
}
```

The *partition* algorithm places the *pivot* element in it's appropriate place, say q.

Following is the program of quick sort.

```
/* C prog for Quick Sort, quick_sort.c */
#include <stdio.h>
#include <stdlib.h>

void quicksort(int [], int, int);
int partition(int [], int, int);
int main() {
 int A[20], n, i;

 printf("\n Enter number of elements: ");
 scanf("%d",&n);

 printf("\n Enter %d Numbers: ",n);
 for(i=0;i<n;i++)
   scanf("%d",&A[i]);

 quicksort(A,0,n-1);

 printf("\n Sorted Elements: ");
 for(i=0;i<n;i++)
  printf("%d ",A[i]);
 return 0;
}

void quicksort(int a[], int p, int r) {
 int q,i;
 if(p<r) {
  q=partition(a, p, r);
  /*printf("\n q=%d p=%d r=%d a[r]=%d ",q,p,r,a[r]); */
  /*for(i=p;i<=r;i++)
   printf("%d ",a[i]); */

  quicksort(a, p, q-1);
  quicksort(a, q+1, r);
 }
}

int partition(int a[], int p, int r) {
 int x,i,j,tmp;
 x=a[r];
 i=p-1;
 for(j=p;j<r;j++) {
   if(a[j]<=x) {
     i++;
     tmp=a[i];
     a[i]=a[j];
     a[j]=tmp;
   }
 }
 tmp=a[i+1];
 a[i+1]=a[r];
 a[r]=tmp;

 return i+1;
}
```

There are some *comment lines* which you may uncomment to see the step-wise output of quick sort process.

Compile the program and then execute it. A sample output is shown below:

```
$ gcc quick_sort.c
$ ./a.out

 Enter number of elements: 8

 Enter 8 Numbers: 20 8 0 63 41 -9 120 5

 Sorted Elements: -9 0 5 8 20 41 63 120
}
```

Hopefully, you have typed the program correctly and executed it successfully.

1.1 Randomized Quick Sort

Now quicksort is very important algorithm because off it's interesting properties. For detail you may study the book of Cormen[1]. The *pivot* element we are choosing in the *partition* function is always the last element in the sub-arrray. However, it does not need be the same always. We can choose any element as pivot element. If we randomly choose pivot element and then *swap* it with the last element of the sub-array. This gives better possibility of *balanced partitioning*.

So, to implement *randomized quicksort*, we will use *rand( )* function that we have used earlier. Following is the program of randomized quicksort.

```
/* structure prog: read student info and display those */

/* C prog for Quick Sort */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

[1]Introduction To Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Third Edition, PHI

```
void quicksort(int [], int, int);
int partition_r(int [], int, int);

int main() {
 int A[20], n, i;

 printf("\n Enter number of elements: ");
 scanf("%d",&n);

 printf("\n Enter %d Numbers: ",n);
 for(i=0;i<n;i++)
   scanf("%d",&A[i]);

 quicksort(A,0,n-1);

 printf("\n Sorted Elements: ");
 for(i=0;i<n;i++)
  printf("%d ",A[i]);
 return 0;
}

void quicksort(int a[], int p, int r) {
 int q,i;
 if(p<r) {
  q=partition_r(a, p, r);
  /*printf("\n q=%d p=%d r=%d a[r]=%d ",q,p,r,a[r]); */
  /*for(i=p;i<=r;i++)
   printf("%d ",a[i]); */

  quicksort(a, p, q-1);
  quicksort(a, q+1, r);
 }
}

int partition_r(int a[], int p, int r) {  /* randomized
    partition */
 int x,i,j,q,tmp;

 srand(time(NULL));
 q=p+(rand()%(r-p));
 tmp=a[q];
 a[q]=a[r];
 a[r]=tmp;

 x=a[r];
 i=p-1;
 for(j=p;j<r;j++) {
    if(a[j]<=x) {
      i++;
      tmp=a[i];
      a[i]=a[j];
      a[j]=tmp;
    }
  }
  tmp=a[i+1];
  a[i+1]=a[r];
  a[r]=tmp;

  return i+1;
}
```

Now, type this program, compile it, and execute the program and see the output.

2

Heap Sort

Heap sort is a sorting technique that is also used heavily in practice. Heap sort uses *heap* data structure. There are two type of heap: *min heap* and *max heap*. Heap is a binary tree which is nearly *complete*. A binary tree is said to be a min heap, if all of it's nodes satisfy the property: $Node_{val} \leq LeftChild_{val}$ and $Node_{val} \leq RightChild_{val}$. So, in a min heap, the smallest element of the tree is at the root node. This min heap is used heavily in Computer Science to implement *priority queue*.

Similarly, the all the nodes of a max heap satisfies the property: $Node_{val} \geq LeftChild_{val}$ and $Node_{val} \geq RightChild_{val}$. So, in a max heap, the largest element of the tree is at the root node.

Binary tree data structure can be implemented either using *array* or by using *linked list*. For heap sort, array implementation of heap is generally used. When we implement a complete binary tree as an array, then if a node is at array index i, then it's left child is located in array index $2 \times i$, and right child is located in the array index $2 \times i + 1$.

Worst-case time complexity of heapsort is $\mathcal{O}(n \times \log_2 n)$ where n is the input size. For further discussion on heap sort, refer the book of Cormen[1].

Following is the program of heap sort.

```c
/* C prog for Heap Sort */
/* File Name: heap_sort.c */

#include <stdio.h>
#include <stdlib.h>
```

[1]Introduction To Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Third Edition, PHI

```
void max_heapify(int [], int, int);
void build_max_heap(int [], int);
void heap_sort(int [], int);

int main() {
 int A[20], n, i;

 printf("\n Enter number of elements: ");
 scanf("%d",&n);

 printf("\n Enter %d Numbers: ",n);
 for(i=0;i<n;i++)
   scanf("%d",&A[i]);

 heap_sort(A,n);
 printf("\n Sorted Elements: ");
 for(i=0;i<n;i++)
  printf("%d ",A[i]);
 return 0;
}

void max_heapify(int A[], int n, int i) {
 int left, right, largest, tmp,j;

 if(i==0) {  /* root node */
   left=1;
   right=2;
 }
 else {   /* non-root node */
   left=2*i;
   right=left+1;
 }

 if((left<n)&&(A[left]>A[i]))
   largest=left;
 else
   largest=i;

 if((right<n)&&(A[right]>A[largest]))
   largest=right;

 if(largest != i) {   /* swap */
   tmp=A[i];
   A[i]=A[largest];
   A[largest]=tmp;

  max_heapify(A, n, largest);
 }
}

void build_max_heap(int A[], int n) {
 int i,j;
 for(i=n/2; i>=0; i--)
   max_heapify(A,n,i);

}

void heap_sort(int A[], int n) {
 int i,tmp,j,m;
 m=n;
 build_max_heap(A, n);

 for(i=n-1; i>=0;i--) {  /* swap root, maximum, and last node*/
   tmp=A[0];
   A[0]=A[i];
   A[i]=tmp;

   n--;     /* reduce tree size */
   max_heapify(A,n,0);
 }

}
```

Compile the program, *heap_sort.c* and then execute it. A sample output is shown below:

```
$ gcc heap_sort.c
$ ./a.out

 Enter number of elements: 10

 Enter 10 Numbers: 4 11 3 2 16 9 10 14 8 7

 Sorted Elements: 2 3 4 7 8 9 10 11 14 16
}
```