Introduction to Algorithms------First Assignment

NAME: Liu Chengjun Student ID: 0490303961

First Choice of Topic (Regular Assignment), based on the implementation of programming.

For the first part we have to display sorting algorithms coded on MATLAB.

Code:

Insert Sort

|  |
| --- |
| function x=insertsort(x)  for j=2:length(x)  element=x(j);  i=j-1;  while i>0 && x(i)>element  x(i+1)=x(i);  i=i-1;  end  x(i+1)=element;  end |

Merge Sort

|  |
| --- |
| function dataOut = merging(data)  lenD = size(data,2);  if(lenD<2)  dataOut = data;  else  middle = floor(lenD/2);  L = data(1:middle);  R = data(middle+1:end);  L = merging(L);  R = merging(R);  dataOut = merge(L, R);  end    function dataOut = merge(L,R)  lenL = size(L,2);  lenR = size(R,2);  i = 0;  j = 0;  merged = zeros(1,lenL+lenR);  while(i<lenL||j<lenR)  if (i<lenL && j<lenR)  if(L(i+1)<=R(j+1))  merged(i+j+1) = L(i+1);  i = i+1;  else  merged(i+j+1) = R(j+1);  j = j+1;  end  elseif(i<lenL)  merged(i+j+1) = L(i+1);  i = i+1;  elseif(j<lenR)  merged(i+j+1) = R(j+1);  j = j+1;  end  end  dataOut = merged; |

Heap Sort

|  |
| --- |
| function [ A ] = heap( A )  A=build\_max\_heap(A);  [~,n]=size(A);  for i=n:-1:2  temp=A(1);  A(1)=A(i);  A(i)=temp;  n=n-1;  A=max\_heapify(A,n,1);  end    function [ A ] = build\_max\_heap( A )  [~,n]=size(A);  for i=floor(n/2):-1:1  A=max\_heapify(A,n,i);  end    function [ A ] = max\_heapify( A,n,i )  l=left(i);  r=right(i);  if l<=n&&A(l)>A(i)  largest=l;  else  largest=i;  end  if r<=n&&A(r)>A(largest)  largest=r;  end  if largest~=i  temp=A(i);  A(i)=A(largest);  A(largest)=temp;  A=max\_heapify(A,n,largest);  end    function [ lIndex ] = left( i )  lIndex=2\*i;    function [ rIndex ] = right( i )  rIndex=2\*i+1; |

Quick Sort

|  |
| --- |
| function [ A ] = quick( A,p,r )  if p<r  [A,q]=partition(A,p,r);  A=quick(A,p,q-1);  A=quick(A,q+1,r);  end  end    function [ A,q ] = partition( A,p,r )  x=A(r);  i=p-1;  for j=p:r-1  if A(j)<=x  i=i+1;  temp=A(j);  A(j)=A(i);  A(i)=temp;  end  end  temp=A(i+1);  A(i+1)=A(r);  A(r)=temp;  q=i+1;  end |

Result:

Followed by MATLAB codes, we test on 10, 100, 1000, 10000, 100000 and 1000000 random sequences and the results are as follows in average of 100 repeated code running.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| (Unit: second) | 100 | 1000 | 10000 | 100000 | 1000000 |
| Insert Sort | 0.0002 | 0.0017 | 0.1372 | 11.5872 | \ |
| Merge Sort | 0.0005 | 0.0028 | 0.0222 | 0.2502 | 2.1911 |
| Heap Sort | 0.0003 | 0.0019 | 0.0182 | 0.2471 | 3.2486 |
| Quick Sort | 0.0002 | 0.0008 | 0.0050 | 0.0459 | 0.5641 |

Discussion:

1. For random sequences whose volume is under 1000, the Insert Sort seems to be one of the fastest in practice.

Although its running time estimation follows O(n^2), for smaller volume O(n^2) does not differ too much with other three algorithms whose running time estimations follow O(n\*log(n)).

It is revealed in the codes above that the code length of Insert Sort is about 1/3 of those of other algorithm, which means that simple code design of Insert Sort saves much more time than other algorithms in code running.

Thus it performs better in small volumes, but performs badly for volumes of over 5000. As it can be observed in the form, the running time of Insert Sort increases 100 times from 10000 to 100000, which means it generally follows O(n^2) in increase of running (10^2=100).

1. For all volume, Quick Sort outperforms other sorting algorithms.

For Merge Sort, Heap Sort and Quick Sort, they all follow O(n\*log(n)).

One reason that Quick Sort outperforms others is that the code length is about half of those of Merge Sort and Heap Sort.

|  |
| --- |
| Merge Sort: 36  Heap Sort: 36  Quick Sort: 23 |

Another reason for Merge Sort and Heap Sort is that the number of if-judgement, for-loop and while-loop structures is larger than that of Quick Sort.

|  |
| --- |
| Estimation of if, while and for language in program running:  Merge Sort: T(n)=(2n+1)\*if + n\*while + 2\*T(n/2)------T~(2n\*log(n)+n)\*if + n\*log(n)\*while  Heap Sort: T(n)=for + sum(i from 1 to log(n))(3\*if \*(n/(2^i))) + for + (n-1)\*log(n)\*(3\*if)------T~2\*for + 3\*(n+(n-1)\*log(n))\*if  Quick Sort:: T(n)=if + n\*if + for + 2\*T(n/2)------T~(n+1)\*log(n)\*if + n\*for |

Additionally, for Merge Sort algorithm, it requires new arrays and giving values to arrays in merge step (merge (L, R)) for around O(log\*(n)) times (memory usage follows O(n), differing from O(1) in Heap Sort and Quick Sort), which may cost more time in busy code running tasks and tight memory space.

For Heap Sort, the value swaps in arrays are more frequent, probably meaning more time spent in value swapping (this feature can be observed by its longer code).

Moreover, the build-max-heap step adds O(n) to code running time, probably influencing efficiency when compared with Quick Sort.

1. Total Estimation on time: (Assuming one line in codes spends equal time)

|  |
| --- |
| Merge Sort: T(n)=4+2T(n/2) +5+(3+2)\*n+1------T~5n\*log(n)+10n  Heap Sort: T(n)=1+sum(i from 1 to log(n))(n/2^(i+1))((6+4)\*log(n)\*i/log(n))+1+(n-1)\*(4+(6+4)\*log(n))------T~14n-2+10(n-1)\*log(n)  Quick Sort: T(n)=2+(4+1)n+4+2T(n/2)------T~6n+5n\*log(n) |

From the estimation above, we indeed think that Quick Sort performs best based on short code.

1. So based on the low memory usage, short code and fast speed, Quick Sort seems to be the best in practice.