1-

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Best | Avg | Worst |
| Bubble Sort | n |  |  |
| Merge Sort | nlogn | nlogn | nlogn |
| Quick Sort | nlogn | nlogn |  |
| Heap Sort | nlogn | nlogn | nlogn |
| Insertion Sort | n |  |  |

در توضیح بخش دوم سوال میتوان گفت هرکدام از الگوریتم های بالا میتوانند در حالات خاصی از داده ها بهترین عملکرد را داشته باشند:  
الگوریتم حبابی هنگامی که داده ها از پیش سورت شده اند-یا به سورت نزدیک اند- بسیار سریع عمل میکنند(منظور آنکه عملیات سورت و رها کردن آرایه به پایان می رسد)  
الگوریتم insertion نیز همچون الگوریتم حبابی در آرایه های نزدیک به سورت بسیار خوب عمل میکنند همچنین در حالت میانگین برای مثال اگر چه از Merge بهای بیشتری باید بپردازیم اما در صورتی که طول آرایه کوچک باشد این الگوریتم بهتر از Merge عمل میکند(به علت ضریب ثابت کوچکتری که این الگوریتم دارد)  
در مورد سه الگوریتم دیگر نیز همانطور که مشخص است بهای زمانی نزدیک است و همه چیز به حالت قرارگیری داده ها بستگی دارد مثلا برای Quick درصورتی که اعداد به گونه ای قرار گرفته باشند که عدد محوری به Median نزدیکتر باشد سرعت سورت بالاتر میرود.  
نهایتا همانطور که در ابتدا گفته شد و در بررسی نیز به آن رسیدیم، همه چیز به نوع قرار گیری ورودی ها در کنار هم بستگی دارد

2-برای ادغام این دو آرایه، الگوریتم مشخص است. از ابتدای دو آرایه هر عضو را با عضو متناظر مقایسه کرده و کوچکترین آنرا در آرایه ی ادغامی قرار دهیم و از عضو بعدی آن آرایه برای مقایسه استفاده نماییم،بیشترین مقایسه هنگامی رخ میدهد که تا آخر ادغام هیچکدام از دو آرایه تمام نشوند تا هر لحظه دلیلی برای مقایسه وجود داشته باشد! در این حالت تمام اعضای دو آرایه در مقایسه ها شرکت میکنند و از آنجا که به ازای هر مقایسه یکی از اعضا از یکی از آرایه ها رد میشود در نتیجه به ازای مجموع اعضای دو آرایه (r+p) مقایسه خواهیم داشت.

3-  
الف)همانطور که در جدول بالا مشخص است در بدترین حالت -یا نزدیک به آن- Merge با بهای بسیار کمتری کار میکند، حال بدترین حالت چیست؟ هنگامی که محور مقایسه در الگوریتم Quick کوچکترین یا بزرگترین عدد موجود انتخاب شود در این حالت این الگوریتم تقریبا مانند Insertion کار میکند و بسیار کند خواهد بود  
  
ب)بیشترین بها و پیچیدگی زمانی Binary Insertion برابر با nlogn است که از برای Quick کمتر است  
  
ج)باز هم با توجه به جدول سوال 1 میتوان گفت که بدترین پیچیدگی زمانی Heap که برابر با nlogn است از کمتر است  
  
د)بدلیل ضریب کوچکتری که برای پیچیدگی زمانی Insertion وجود دارد این الگوریتم در تعداد مقادیر داده کمتر بهتر عمل میکند  
  
ه)این الگوریتم ها تحت برخی فرضیات و با استفاده از حافظه ای خارجی عملیات سورت را انجام میدهند، مثلا counting از حافظه خارجی برای شمارش تعداد تکرار هر ورودی با فرض حضور داده ها در یک بازه عددی استفاده میکند.

4-  
الف)

|  |
| --- |
| -7 |
| -1 |
| 2 |
| 3 |
| 4 |
|  |
|  |
|  |
| 8 |
|  |
|  |
|  |
|  |

ب)c1=1 c2=1

|  |
| --- |
|  |
| -1 |
| 2 |
| 3 |
| 4 |
| -7 |
|  |
|  |
| 8 |
|  |
|  |
|  |
|  |

5-طرحی که توسط جاوا پیشنهاد شده است استفاده از یک درخت متوازن به جای یک لینک لیست است در این صورت هنگامی که یک کلید به جدول نگاشت میشود به جای جستجوی خانه ی مورد نظر جدول که یک لینک لیست است و جستجو در آن هزینه ی O(n) دارد با پیاده سازی درخت متوازن از هزینه O(h) بهره ببریم  
راه دیگر جدای استفاده از درخت،استفاده از یک hash table ثانویه در هر خانه جدول اولیه و استفاده از یک hash func برای نگاشت عناصری است که به این خانه از جدول اولیه نگاشت میشوند که با بهره گیری از یک تابع درست میتوان هزینه را به O(1) رسانده و حتی از روش درخت نیز بشدت! پیشی گرفت

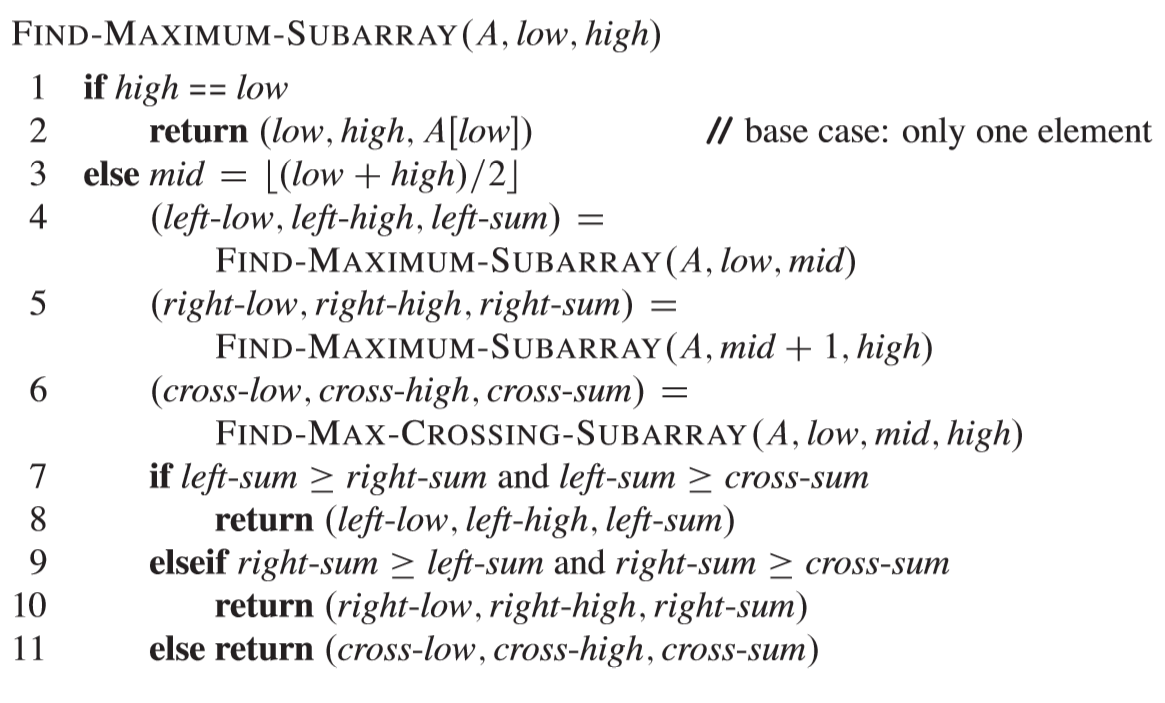
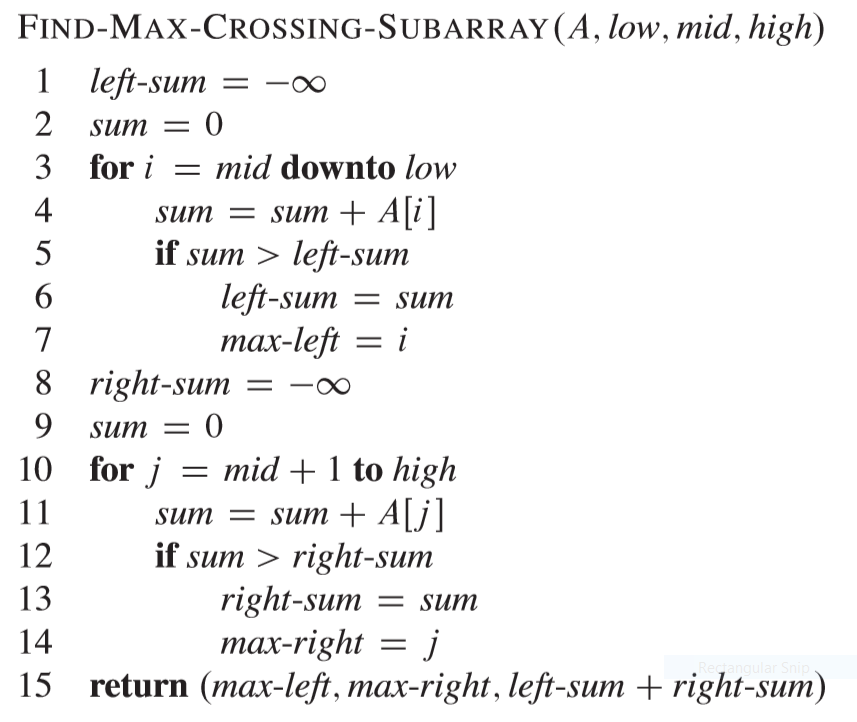
6-

Min\_heapfy(int[] A,index root){  
 If(A[root]>A[2\*root])  
 Min=2\*root  
 else  
 Min=root

If(A[Min]>A[2\*root+1])  
 Min=2\*root+1  
 else  
 Min=root

If(Min!=root)  
 exchange A[root] with A[min]

Min\_heapfy(A,2\*root)  
 Min\_heapfy(A,2\*root+1)

7-پاسخ این سوال را مستقیما از کتاب مرجع درج مینماییم:  
 

8-

Symmetric\_Substring(char[] A){  
char f=A[0]  
char l=A[A.length-1]

While(f!=l || f!=l+1){  
if(A[f]==A[l]){  
f++  
l--  
}else{  
tempf=f+1  
templ=l-1  
while((A[tempf]!=A[l] || A[templ]!=A[f]) && (tempf!=templ || tempf!=templ+1)){  
tempf++  
templ--  
}  
if(A[tempf]==A[l]){  
Remove f~tempf-1 chars and f=tempf  
}else if(A[templ]==A[f]) {  
Remove l~templ+1 chars and f=templ  
}else{  
return error  
}  
}  
}  
return A  
}

از آنجا که کافیست تمام کارکترها بررسی شوند پس پیچیدگی زمانی این الگوریتم برابر O(A.length) خواهد بود

9-

Counting\_Sort(int[] A, int n){  
int[] B=new int[2\*n] //all of them equal to zero at first  
int[] C=new int[A.length]  
for(int i=0;i<A.length;i++){  
B[A[i]]++  
}  
for(int i=1;i<B.length;i++){  
B[i]+=B[i-1]  
}  
for(int i=0;i<A.length;i++){  
C[B[A[i]]]=A[i]  
B[A[i]]--  
}  
return C}