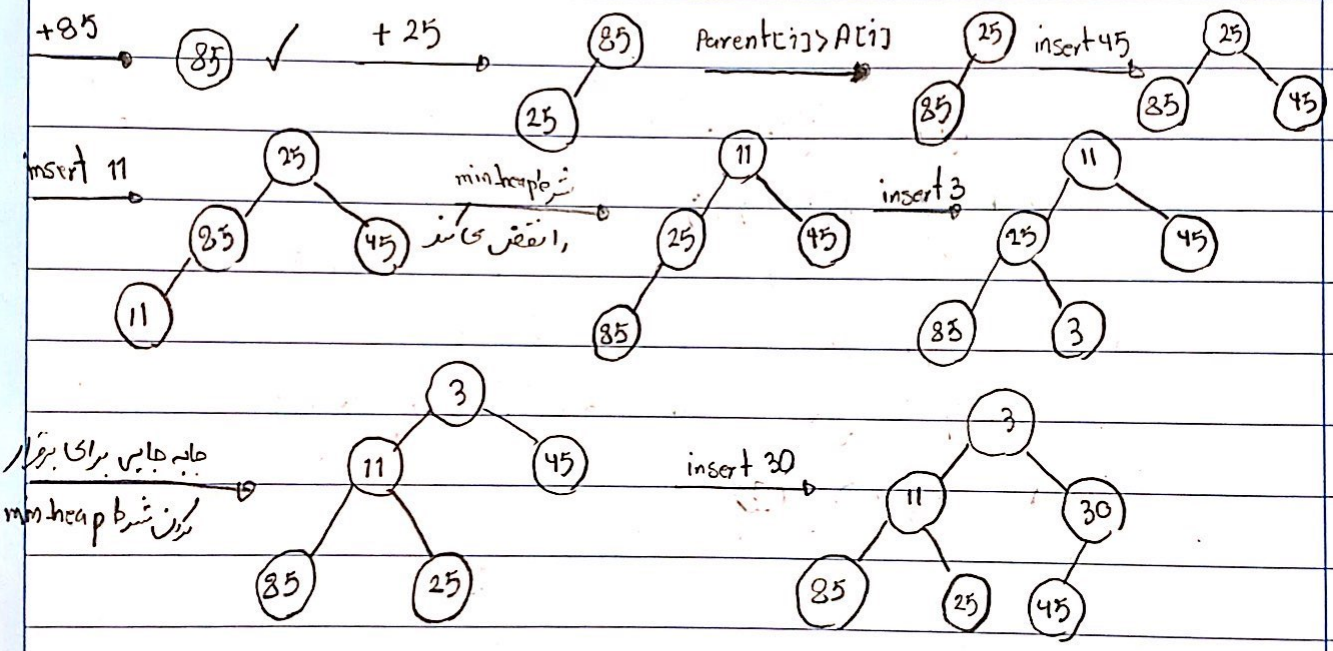
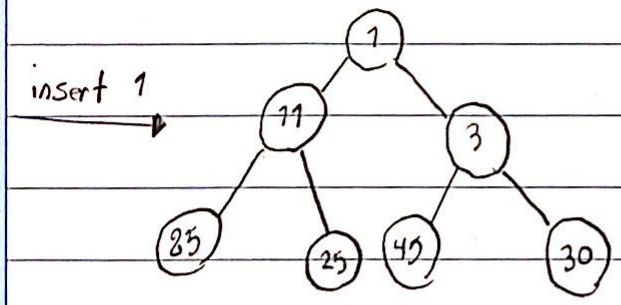
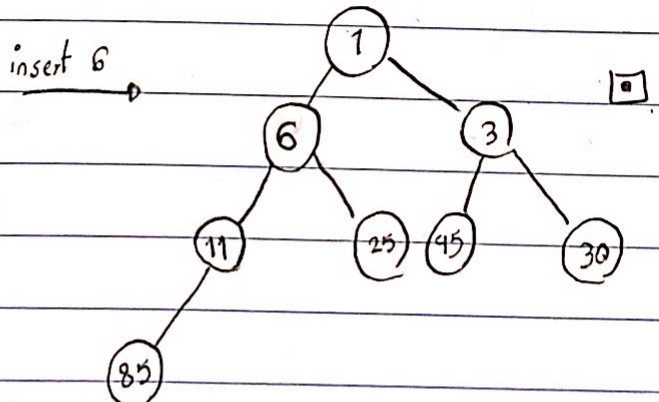
a) در minheap برای insert کردن یک عنصر به آخر heap یک نود اضافه کرده و مقدار آن را $+\infty$ می‌گذاریم و سپس به کمک همان مقداری که insert می‌خواهیم کنیم decrease key می‌کنیم و سپس آن را با Parent شش مقایسه می‌کنیم که شرط min_heap را نقض نکرده باشد. که در بهترین حالت هیچ جابه‌جایی صورت نمی‌گیرد و در بدترین حالت تا root می‌رود.



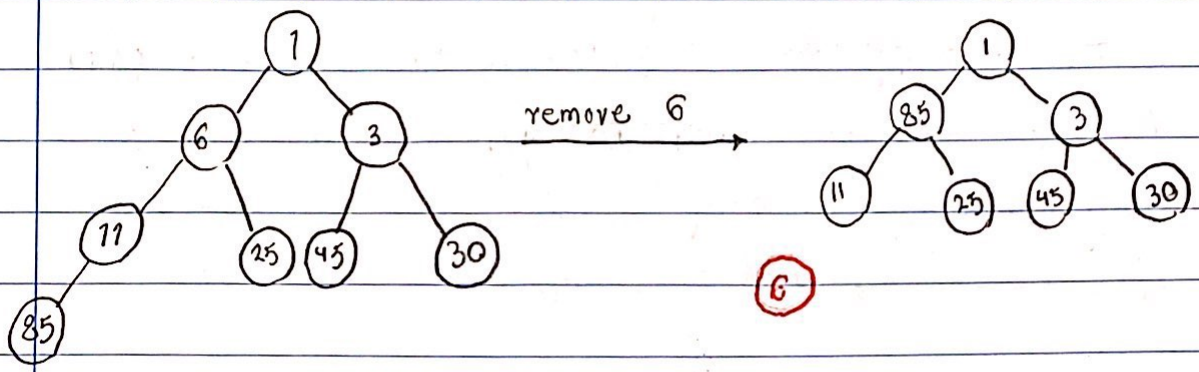




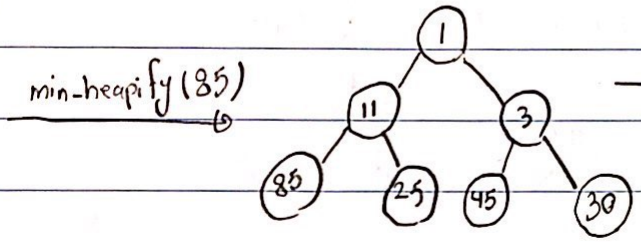


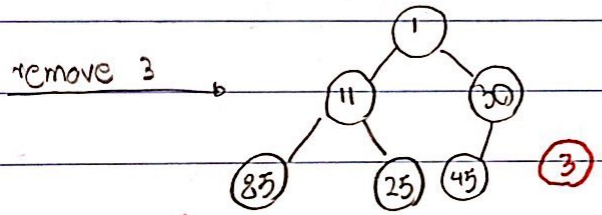

b) برای حذف کردن یک عنصر در minheap جای آن را با آخرین عنصر heap عوض کرده و روی آن min_heapify می‌زنیم تا شرطی از heap را اگر نقض کرده است جبران شود و سپس یکی از اندیس‌های heap کم می‌کنیم که عنصر آخر که حال عنصری است که ما می‌خواستیم حذف کنیم از heap خارج شود.




6

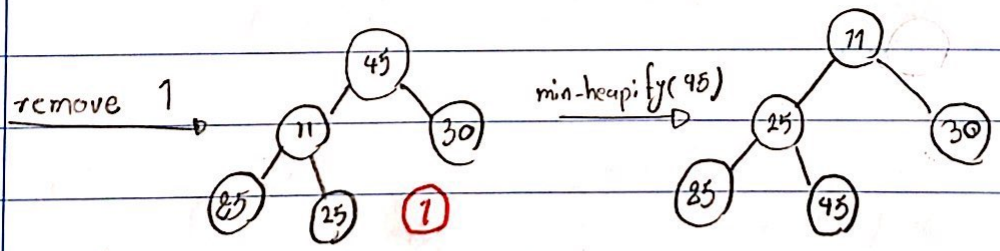



درخت نهایی پس از حذف عنصر 6




3

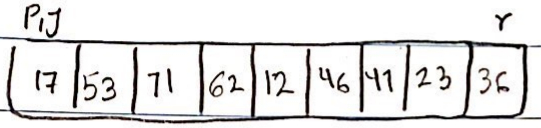
درخت نهایی پس از حذف عنصر 3

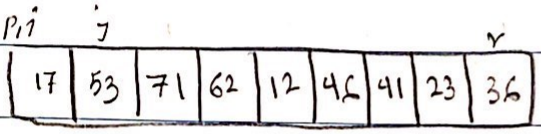


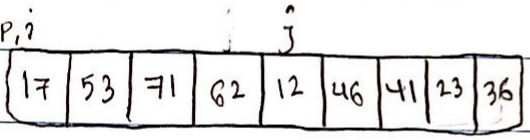

1

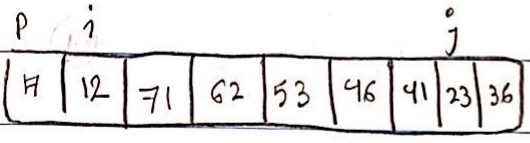
درخت نهایی پس از حذف 3 عنصر <6,3,1>

2) a) Pivot را عنصر 36 در نظر می‌گیریم پس جای 36 را با آخرین عنصر عوض می‌کنیم و پس Partition را انجام می‌دهیم.

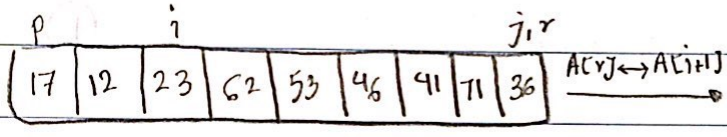


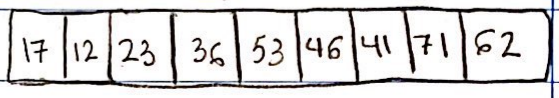




آرایه‌ی نهایی: باید پارتیشن (36)

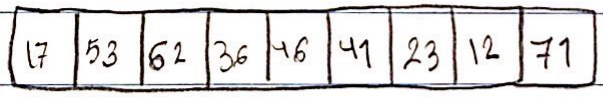



✓

b) در یک پیاپش از bubble sort عناصر از ابتدای آرایه تا انتها 2 به 2 مقایسه می‌شوند و به این ترتیب بزرگترین عنصر به انتهای آرایه می‌رسد.

آرایه نهایی: بایک تفاوت از bubble sort →




✓

3) a) ابتدا اعداد داخل آرایه را در bucket n بر اساس تعداد ارقام اعداد می‌ریزیم. چون مجموعه‌ی ارقام اعداد n است پس هر عدد حداکثر n رقم خواهد داشت. این مرحله $O(n)$ زمان می‌برد. حال کافی است اعداد داخل هر Bucket را Sort کنیم. حال فرض کنیم $K_i$ عدد با i رقم وجود دارد، می‌توانیم این اعداد را Radix Sort با رقم به رقم Sort کنیم که $O(iK_i)$ زمان می‌برد که برای همه‌ی اعداد $O(\sum_{i=0} iK_i)$ زمان خواهد برد که $\sum iK_i$ برابر با کل ارقام یعنی n است. بنابراین در نهایت:

$$O(n) + O(\Sigma iK_i) = O(n) + O(n) = O(n)$$

زمان می‌برد.

b) برای هر حرف خروجی Bucket می‌سازیم و هر کدام شامل کلماتی است که با آن حرف شروع می‌شوند. حال کلمات هر Bucket را با Radixsort بر اساس مرتبه‌ی دوم حرف اول کلمات در هر Bucket مرتب می‌کنیم و حال Bucket ها را به هم وصل می‌کنیم. حال به linked list از کلماتی داریم که به ترتیب حروف الفبا مرتب شده‌اند که مثل مثال قبل اوردر زمانی آن $O(n)$ است.

4) (a) پیدا کردن max در هر دو ساختمان داده زمان constant از مرتبه‌ی برد پس فرقی ندارد از کدام استفاده شود.

(b) delete کردن یک عضو در heap از اوردر زمان lgn است اما در sorted array، $O(n)$ است. $O(\lg n) < O(n)$ پس heap برای دیلیت کردن ساختمان داده‌ی بهتری است.

(c) ساخت heap از اوردر $O(n)$ است اما ساخت sorted array از $O(n \log n)$ است پس heap بهتر است برای ساختن.

(d) پیدا کردن min در max-heap، $O(n)$ زمان می‌برد چون باید کل درخت را پیمایش کنیم اما در sorted array در $O(1)$ انجام می‌شود.

5)

```
heap_delete(A,i)
  if i > heap-size[A] or i < 1
        error
  temp = A[i]
  A[i] = A[heap-size[A]]
  heap_size[A] = heap-size[A] - 1
  if i > 1 and A[i] > A[parent(i)]
      then heap_increas_key(A,i,A[i])
      else max_heapify(A,i)
  return temp
```

زمان اجرای heap_delete باید زمان اجرای Heap-increase-key یا max_heapify باشد که هر دوی آن‌ها در $O(\lg n)$ اجرا می‌شوند.

6) بدترین حالت در quick sort این است که یکی از زیر آرایه ها تعدادشان برابر با n-1 و دیگری برابر 0 شود.
حال یک آرایه ی مرتب شده ی نزولی داریم اگر بخواهیم به روش quick sort آن را صعودی sort کنیم
با توجه به اینکه اولین عنصر را به عنوان Pivot می گیریم و آن بزرگترین عنصر آرایه است پس تمامی اعداد بعد از آن
در زیر آرایه ی سمت چپ قرار می گیرند که این همان worst case است و پیچیدگی زمان آن $O(n^2)$ خواهد بود.
برای مرتب سازی نزولی، اولین عنصر که بزرگترین عنصر است را Pivot می گیریم و همه ی عناصر از آن کوچکتر هستند و
در سمت راست آن قرار می گیرند که این هم worst case است و پیچیدگی زمان آن $O(n^2)$ می باشد.

7) (a) $f(n)$ تعداد اعضای مرتب شده ای در آرایه ی مرتب شده ای که داریم به صورتی insert کرده که اوردر زمانی این کار $O(1) \times O(n) = O(n)$ می باشد که می شود $O(fn) \times O(n)$ . از راه دیگر می توان $f(n)$ را با هر الگوریتمی باشد insertion sort
سورت کرده و سپس طی مرحله ی Conquer در merge sort که دو آرایه سورت شده را با اوردر $O(n)$
به صورت سورت شده ادغام می کند سورت کند.

(b) ابتدا $f(n)$ را با هر الگوریتم سورتی مانند insertion sort مرتب کرده که در این صورت اوردر زمانی این سورت می شود
$O(\lg^2 n)$ که واضح است که $\lg^2 n = O(n)$ می باشد، پس مانند حالت قبل از مرحله ی Conquer در
merge sort استفاده کرده که اوردر زمانی کل می شود $O(n)+O(\lg^2 n)=O(n)$ . می توان $f(n)$ عنصر را با
الگوریتم merge sort سورت کرد که اوردر زمانی آن $(\lg n)(\lg\lg n)$ می شود که بسیار کوچک تر از $O(n)$ می باشد.

(c) با سورت کردن $f(n)$ با bubble sort یا insertion sort، $O(f(n))=O(\sqrt{n^2})=O(n)$ و سپس با ادغام کردن
آن در merge sort با اوردر $O(n)$ این دو آرایه سورت شده را به صورت سورت شده ادغام می کنیم.

(d) بهترین پیچیدگی زمانی برای یک الگوریتم سورت $O(n \lg n)$ است که یعنی سورت کردن $f(n)$ با اورجر $O(f(n)\lg(f(n))$ صورت می‌گیرد که در نهایت اوردر نهایی می‌شود $O(n + f(n))$ است. می‌خواهیم حاصل این عبارت $O(n)$ شود. حال $f(n) = O(\frac{n}{\lg n})$ پس $O(f(n)\lg(f(n)) = O(\frac{n}{\lg n} \lg(\frac{n}{\lg n}))$ که داریم $n \geq (\lg n - \lg\lg n)\frac{n}{\lg n}$ درنتیجه حاصل برابر با $O(n)$ است.

(e)

$$T_{sort}(n) = C_{sort} \cdot n \lg n \xrightarrow[\text{نامساوی}]{\text{تشکیل دهیم}} C_{select} \cdot kn \leq C_{sort} \cdot n \lg n$$

$$T_{select}(n) = C_{select} \cdot kn$$

$$\Rightarrow K \leq \frac{C_{sort}}{C_{select}} \cdot \lg n \quad *$$

اگر همین رابطه‌ای بین $n$ و $K$ برقرار باشد الگوریتم دوم بهتر است.

$n = 20,\ K = 512$ :

$$T_{sort}(1024) = C_{sort} \cdot 1024 \cdot \lg 1024 = 10240\, C_{sort} = 10.24\ ms \Rightarrow C_{sort} = 10^{-3}\ ms$$

$$T_{select}(1024) = C_{select} \cdot 1024 \cdot 1024\ ms \Rightarrow C_{select} = 10^{-3}\ ms$$

$$* \longrightarrow 512 \geq \frac{10^{-3}}{10^{-3}} \lg 20 \Rightarrow 512 \geq 20 \Rightarrow$$

در این مثال الگوریتم اول بهتر است.

۹) خطی ۶ بخشی از حلقه است که پیک می‌کند آرایه سورت شده است یا نه. حداقل ۱ بار هر بار که حلقه شروع می‌شود اجرا می‌شود. از آنجایی که الگوریتم همه‌ی جایگشت‌های ممکن آرایه را طی می‌کند خط ۶ حداقل $(n!)$ بار اجرا می‌شود (دقیقاً $n! \frac{1}{2}$ اگر جایگشت سورت شده را با نیمی از جایگشت‌ها بررسی کنیم)
از آنجایی که خط ۶ داخل حلقه‌ای است که ممکن است قبل از انجام $n$ مقایسه تمام نشود و حلقه‌ی for $n! \frac{1}{2}$ بار اجرا می‌شود پیچیدگی زمان بالا $O(nn!)$ یا $O((n-1)n!)$ است