بسمه‌تعالی

درس طراحی الگوریتم ها

(با شبه کد های c ++)

مرجع جدیدترین مقالات و جزوات

تجارت الکترونیک

فصل اول:

کارایی ، تحلیل و مرتبه الگوریتم ها

- تکنیک ، روش مورد استفاده در حل مسائل است.
- مسئله ، پرسشی است که به دنبال پاسخ آن هستیم.
- به کار بردن تکنیک منجر به روشی گام به گام (الگوریتم) در حل یک مسئله می شود.
- منظور از سریع بودن یک الگوریتم، یعنی تحلیل آن از لحاظ زمان و حافظه.
- نوشتن الگوریتم به زبان فارسی دو ایراد دارد:

۱- نوشتن الگوریتم های پیچیده به این شیوه دشوار است.

۲- مشخص نیست از توصیف فارسی الگوریتم چگونه می توان یک برنامه کامپیوتری ایجاد کرد.

الگوریتم ۱-۱: **جست و جوی ترتیبی**

```
Void  seqsearch ( int n
                  const keytype  S[ ]
                  keytype  x,
                  index& location)
{
    location = 1;
    while (location <= n && S[location]  ! = x)
          location++;
    if  (location  >  n )
         location  = 0 ;
```

الگوریتم ۱-۲: **محاسبه مجموع عناصر آرایه**

```
number sum (int n , const number s[ ])
{
    index i;
    number  result;
    result = 0;
    for (i = 1; i <= n; i++)
        result = result + s[i];
    return  result ;
}
```

الگوریتم ۱-۳: **مرتب سازی تعویضی**

مسئله: n کلید را به ترتیب غیر نزولی مرتب سازی کنید.

```
void  exchangesort (int n , keytype S[ ])
 {
     index i,j;
     for (i = 1 ; i<= n -1; i++)
         for (j = i +1; j <= n ; j++)
             if ( S[j] < S[i])
                 exchange S[i]  and S[j];
}
```

الگوریتم ۱-۴: **ضرب ماتریس ها**

```
void matrixmult (int n
                 const number  A [ ] [ ],
                 const number  B [ ] [ ],
                       number  C [ ] [ ],
{
    index i , j, k;
    for  ( i = 1; i <= n ; i++)
        for ( j = 1; j <= n ; j++)}
            C [ i ] [ j ] = 0;
         for (k = 1 ; k <= n ; k++)
                C [ i ][ j ] = C[ i ] [ j ] + A [ i ][ k ] * B [ k ][ j ]

     }
}
```

۱-۲ اهمیت ساخت الگوریتم های کارآمد:

- جست و جوی دودویی معمولا بسیار سریع تر ازجست و جوی ترتیبی است.
- تعداد مقایسه های انجام شده توسط جست و جوی دودویی برابر با $\lg n + 1$ است .

نکته: در تحلیل الگوریتم ها به جای log از نماد خلاصه lg استفاده می کنیم.

اندازه آرایه	تعداد مقایسه های انجام شده توسط جستجوی ترتیبی	تعداد مقایسه های انجام شده توسط جستجوی دودویی
۱۲۸	۱۲۸	۸
۱۰۲۴	۱۰۲۴	۱۱
۱۰۴۸۵۷۶	۱۰۴۸۵۷۶	۲۱
۴۲۹۴۹۶۷۲۹۴	۴۲۹۴۹۶۷۲۹۴	۳۳

الگوریتم ۱-۵: **جست و جوی ترتیبی**

```
Void  seqsearch ( int n
                  const keytype  S[ ]
                  keytype  x,
                  index& location)
{
    location = 1;
    while (location <= n && S[location]  ! = x)
          location++;
    if  (location  >  n )
         location  = 0 ;
```

الگوریتم ۱-۶: **جست و جوی دودویی**

```
Void  binsearch (int n,
                   const keytype  S[ ],
                   keytype  x,
                   index& location)
{
    index low, high, mid;
    low = 1 ; high = n;
    location = 0;
    while (low <= high && location = = 0) {
         mid = ⌊(low + high)/2⌋;
  if ( x = = S [mid])
         location = mid;
```

```
    else if (x < S [mid])
          high = mid – 1;
    else
          low = mid + 1;
    }
  }
```

الگوریتم ۱-۷: **جمله n ام فیبوناچی (بازگشتی)**

مسئله : جمله n ام از دنباله فیبوناچی را تعیین کنید.

```
int fib  (int n)
{
    if ( n <= 1)
        return n;
    else
        return fib (n – 1)  + fib (n – 2);
}
```

الگوریتم ۱ -۸: **جمله n ام فیبوناچی (تکراری)**

```
int fib2 (int n)
 {
    index i;
    int f [0..n];
    f[0] = 0;
    if (n > 0)  {
         f[1] = 1;
```

```
        for  (i = 2 ; i <= n; i++)
            f[i] = f [i -1] + f [i -2];
    }
    return  f[n];
}
```

۱-۳ تحلیل الگوریتم ها

- برای تعیین میزان کارایی یک الگوریتم را باید تحلیل کرد.

۱-۳-۱ تحلیل پیچیدگی زمانی

- تحلیل پیچیدگی زمانی یک الگوریتم ، تعیین تعداد دفعاتی است که عمل اصلی به ازای هر مقدار از ورودی انجام می شود.
- $T(n)$ را پیچیدگی زمانی الگوریتم در حالت معمول می گویند.
- $W(n)$ را تحلیل پیچیدگی زمانی در بدترین حالت می نامند.
- $A(n)$ را پیچیدگی زمانی در حالت میانگین می گویند.

تحلیل پیچیدگی زمانی برای حالت معمول برای الگوریتم(جمع کردن عناصرآرایه)

عمل اصلی: افزودن یک عنصر از آرایه به sum.

اندازه ورودی: n، تعداد عناصر آرایه.

$$T(n) = n$$

تحلیل پیچیدگی زمانی برای حالت معمول برای الگوریتم(مرتب سازی تعویضی)

عمل اصلی: مقایسه S [j] با S[i] .

اندازه ورودی: تعداد عناصری که باید مرتب شوند.

$$T(n) = n(n -1) /2$$

تحلیل پیچیدگی زمانی دربدترین حالت برای الگوریتم(جست و جوی ترتیبی)

عمل اصلی: مقایسه یک عنصر آرایه با x.

اندازه ورودی: n , تعداد عناصر موجود در آرایه.

$$W (n) = n$$

تحلیل پیچیدگی زمانی در بهترین حالت برای الگوریتم(جست وجوی ترتیبی)

عمل اصلی: مقایسه یک عنصر آرایه با x.

اندازه ورودی: n , تعداد عناصر آرایه.

$$B (n) = 1$$

۱-۴مرتبه الگوریتم

- الگوریتم هایی با پیچیدگی زمانی ازقبیل n و100n را الگوریتم های زمانی خطی می گویند.
- مجموعه کامل توابع پیچیدگی را که با توابع درجه دوم محض قابل دسته بندی باشند، $\theta (n^2)$ می گویند.
- مجموعه ای ازتوابع پیچیدگی که با توابع درجه سوم محض قابل دسته بندی باشند، $\theta (n^3)$ نامیده می شوند.

- برخی از گروه های پیچیدگی متداول در زیر داده شده است:

$$\theta(\lg n) < \theta(n) < \theta(n \lg n) < \theta(n^2) < \theta(n^3) < \theta(2^n)$$

۱-۴-۱آشنایی بیشتر با مرتبه الگوریتم ها

- برای یک تابع پیچیدگی مفروض $f(n)$، $O(f(n))$ "O بزرگ" مجموعه ای از توابع پیچیدگی $g(n)$ است که برای آن ها یک ثابت حقیقی مثبت c و یک عدد صحیح غیر منفی N وجود دارد به قسمی که به ازای همه ی $n >= N$ داریم:

$$g(n) >= c \times f(n)$$

- برای یک تابع پیچیدگی مفروض $f(n)$ $\Omega(f(n))$ مجموعه ای از توابع پیچیدگی $g(n)$ است که برای آن ها یک عدد ثابت حقیقی مثبت c و یک عدد صحیح غیر منفی N وجود دارد به قسمی که به ازای همه ی $n >= N$ داریم:

$$g(n) <= c \times f(n)$$

- برای یک تابع پیچیدگی مفروض $f(n)$، داریم: $\theta(f(n)) = O(f(n)) \cap \Omega(f(n))$ یعنی $\theta(f(n))$ مجموعه ای از توابع پیچیدگی $g(n)$ است که برای آن ها ثابت های حقیقی مثبت c وd و عدد صحیح غیر منفی N وجود دارد به قسمی که :

$$c \times f(n) <= d \times f(n)$$

- برای یک تابع پیچیدگی $f(n)$ مفروض،$o(f(n))$ "o کوچک" عبارت ازمجموعه کلیه توابع پیچیدگی$g(n)$ است که این شرط را برآورده می سازند : به ازای هرثابت حقیقی مثبت c ،یک عدد صحیح غیر منفی N وجود دارد به قسمی که به ازای همه ی $n >= N$ داریم:

$$g(n) =< c \times f(n)$$

ویژگی های مرتبه:

۱- $g(n) \in O(f(n))$ اگر و فقط اگر $f(n) \in \Omega(g(n))$.

۲- $g(n) \in \theta(f(n))$ اگر و فقط اگر $f(n) \in \theta(g(n))$.

۳- اگر $b > 1$ و $a > 1$، در آن صورت:

$$\log_n a \in \theta(\log_n b)$$

۴- اگر $b > a > 0$، در آن صورت:

$$a^n \in o(b^n)$$

۵- به ازای همه ی مقادیر $a > 0$ داریم : $a^n \in o(n!)$

۶- اگر$c >= 0$، $d > 0$ ، $g(n) \in o(f(n))$ و $h(n) \in \theta(f(n))$ باشد، درآن صورت:

$$c \times g(n) + d \times h(n) \in \theta(f(n))$$

۷- ترتیب دسته های پیچیدگی زیر را در نظربگیرید:

$$\theta(\lg n)\ \theta(n)\ \theta(n \lg n)\ \theta(n^2)\ \theta(n^j)\ \theta(n^k)\ \theta(a^n)\ \theta(b^n)\ \theta(n!)$$

که در آن $k > j > 2$ و $b > a > 1$ است. اگر تابع پیچیدگی $g(n)$ در دسته ای واقع در طرف چپ دسته ی حاوی $f(n)$ باشد، در آن صورت:

$$g(n) \in o(f(n))$$

فصل دوم :

روش تقسیم و حل

- روش تقسیم و حل یک روش بالا به پایین است.
- حل یک نمونه سطح بالای مسئله با رفتن به جزء و به دست آوردن حل نمونه های کوچک تر حاصل می شود.
- هنگام پی ریزی یک الگوریتم بازگشتی ، باید:

۱- راهی برای به دست آوردن حل یک نمونه از روی حل یک نمونه ازروی حل یک یا چند نمونه کوچک تر طراحی کنیم.

۲- شرط (شرایط) نهایی نزدیک شدن به نمونه(های) کوچک تر را تعیین کنیم.

۳- حل را در حالت شرط (شرایط) نهایی تعیین کنیم.

الگوریتم۲-۱: **جست و جوی دودویی (بازگشتی)**

```
index location ( index low, index high )
{
    index mid;
    if  (low > high )
        return  0;
    else  {
        mid = ⌊ (low + high) /2⌋ ;
        if (x = = S [mid])
            return  mid;
        else if ( x < S [mid])
                return location (low , mid – 1);
```

```
        else
                return   location (mid + 1, high);
        }
    }
```

تحلیل پیچیدگی زمانی دربدترین حالت برای الگوریتم جست و جوی دودویی بازگشتی

عمل اصلی: مقایسه x با S [mid].

اندازه ورودی: n ، تعداد عناصر آرایه.

$$W(n) = W(n / 2) + 1$$

برای $n > 1$ ، n توانی از ۲ است

$$W(n) = W(n / 2) + 1$$

$$W(1) = 1$$

$$W(n) = \lfloor \lg n \rfloor + 1 \in \theta(\lg n)$$

2-2مرتب سازی ادغامی

- ادغام یک فرآیند مرتبط با مرتب سازی است.
- ادغام دوطرفه به معنای ترکیب دو آرایه مرتب شده در یک آرایه ی مرتب است.
- مرتب سازی ادغامی شامل مراحل زیر می شود:

۱- تقسیم آرایه به دو زیر آرایه، هر یک با n/2 عنصر.

۲- حل هر زیر آرایه با مرتب سازی آن.

۳- **ترکیب** حل های زیر آرایه ها از طریق ادغام آن ها در یک آرایه مرتب.

الگوریتم۲-۲: **مرتب سازی ادغامی**

```
void  mergsort (int n , keytype S [ ])
   {
        const int  h = ⌊ n/2 ⌋ , m = n – h;
        keytype U [1...h],V [1..m];
        if (n >1)  {
           copy S[1] through  S[h] to U[h];
           copy S [h + 1] through S[h] to V[1] through V[m];
           mergesort(h, U);
           mergesort(m,V);
           merge (h , m , U,V,S);
          }
    }
```

الگوریتم۲-۳: **ادغام**

```
void  merg ( int h , int m, const keytype U[ ],
                                const keytype V[ ],
                                      keytype  S[ ] )
  {
      index i , j , k;
      i = 1; j = 1 ; k = 1;
      while (i <= h && j <= m) {
         if (U [i] < V [j]) {
            S [k] = U [i]
              i+ + ;
    }
```

```
    else {
        S [k] = V [j];
         j+ +;
      }
      k+ +;
}
if ( i > h)
     copy V [j] through V [m] to S [k]  through  S [ h + m ]
else
     copy U [i] through U [h] to S [k]  through  S [ h + m ]
}
```

تحلیل پیچیدگی زمانی دربدترین حالت برای الگوریتم ۲-۳(ادغام)

عمل اصلی: مقایسه U [i] با V[j] .

اندازه ورودی: h و m ،تعداد عناصر موجود در هر یک از دو آرایه ورودی.

$$W (h , m) = h + m - 1$$

تحلیل پیچیدگی زمانی دربدترین حالت برای الگوریتم ۲-۲(مرتب سازی ادغامی)

عمل اصلی: مقایسه ای که درادغام صورت می پذیرد.

اندازه ورودی: n ، تعداد عناصر آرایه S.

$$W(n) = W(h) + W(m) + h + m - 1$$

زمان لازم برای ادغام

زمان لازم برای مرتب سازی V

زمان لازم برای مرتب سازی U

برای $n > 1$ که n توانی از 2 است

$$W(n) = 2\,W(n/2) + n - 1$$

$$W(1) = 0$$

$$W(n) \in \theta(n \lg n)$$

الگوریتم۲-۴: **مرتب سازی ادغامی ۲(mergesort 2)**

```
void mergesort2  (index low, index high)
{
    index mid;
    if  (low  < high)  {
            mid = ⌊ ( low + high) / 2 ⌋;
          mergesort 2 (low, mid);
          mergesort 2 (mid +1, high);
          merge2(low,mid,high)
       }
    }
```

الگوریتم۲ -۵: **ادغام۲**

مسئله: ادغام دو آرایه ی مرتب S که در mergesort ایجاد شده اند.

```
void  mrge2 (index low, index mid, index high)
 {
    index i, j , k;
    keytype U [ low..high]
    i = low; j = mid +1 ; k = low;
    while ( i <= mid && j <=  high)  {
          if ( S [i]  < S [j] )  {
              U [k] = S [i];
              i + + ;
           }
   else  {
         U [k] = S [j]
         j ++;
      }
      k ++;
    }
    if ( i >  mid )
        move S [j]  through S [high] to U [k]  through U [high]
    else
       move S [i]  through S [mid] to U [k]  through U [high]
    move U [low] through U [high] to S [low] through S [high]
}
```

۲-۳روش تقسیم و حل

راهبرد طراحی تقسیم و حل شامل مراحل زیر است:

۱- تقسیم نمونه ای ازیک مسئله به یک یا چند نمونه کوچک تر.

۲- حل هر نمونه کوچکتر. اگر نمونه های کوچک تر به قدر کوچک تر به قدر کافی کوچک نبودند، برای این منظور از بازگشت استفاده کنید.

۳- در صورت نیاز، حل نمونه های کوچک تر را ترکیب کنید تا حل نمونه اولیه به دست آید.

۲-۴ مرتب سازی سریع (quicksort)

- در مرتب سازی سریع، ترتیب آن ها از چگونگی افراز آرایه ها ناشی می شود.
- همه عناصر کوچک تر از عنصر محوری در طرف چپ آن و همه عناصربزرگ تر، درطرف راست آن واقع هستند.
- مرتب سازی سریع، به طور بازگشتی فراخوانی می شود تا هر یک از دوآرایه را مرتب کند، آن ها نیز افراز می شوند و این روال ادامه می یابد تا به آرایه ای با یک عنصربرسیم.

  چنین آرایه ای ذاتاً مرتب است.

الگوریتم۲-۶ : **مرتب سازی سریع**

مسئله : مرتب سازی n کلید با ترتیب غیر نزولی.

```
void quicksort  (index low , index high)
 {
      index  pivotpoint;
      if   ( high >  low) {
            partition (low , high , pivotpoint)
            quicksort (low , pivotpoint – 1)
            quicksort (pivotpoint + 1 , high);
```

```
        }
    }
```

الگوریتم۲-۷: **افراز آرایه**

مسئله: افراز آرایه S برای مرتب سازی سریع.

```
void partition (index low, index high)
                    index & pivotpoint)
    {
        index i , j;
        keytype pivotitem;
        pivotitem = S [low];
        j = low
        for ( i = low +1 ; i <= high; i ++)
if ( S [i] < pivotitem )  {
            j++;
            exchange S [i]  and S [j];
        }
    pivotpoint  = j;
    exchange S [low] and S [ pivotpoint];
}
```

تحلیل پیچیدگی زمانی در حالت معمول برای الگوریتم ۲-۷(افراز)

عمل اصلی: مقایسه S [i] با pivotitem .

اندازه ورودی: n = high – low +1 ، تعداد عناصرموجود در زیر آرایه.

$$T(n) = n - 1$$

تحلیل پیچیدگی زمانی در بدترین حالت برای الگوریتم ۲-۶(مرتب سازی سریع)

عمل اصلی: مقایسه S [i] با pivotitem در روال partition .

اندازه ورودی: n ، تعداد عناصر موجود درآرایه S.

$$T(n) = T(0) + T(n-1) + n - 1$$

زمان لازم برای افراز — زمان لازم برای مرتب سازی زیرآرایه طرف راست — زمان لازم برای مرتب سازی زیرآرایه طرف چپ

به ازای $n > 0$

$$T(n) = T(n-1) + n - 1$$

$$T(0) = 0$$

$$W(n) = n(n-1)/2 \in \theta(n^2)$$

تحلیل پیچیدگی زمانی در حالت میانگین برای الگوریتم ۲-۶(مرتب سازی سریع)

عمل اصلی: مقایسه S [i] با pivotitem در partition .

اندازه ورودی: n ، تعداد عناصر موجود در S.

$$A(n) \in \theta(n \lg n)$$

۲-۵ الگوریتم **ضرب ماتریس استراسن**

- پیچیدگی این الگوریتم از لحاظ ضرب، جمع و تفریق بهتر از پیچیدگی درجه سوم است.
- روش استراسن در مورد ضرب ماتریس های ۲×۲ ارزش چندانی ندارد.

الگوریتم ۲-۸: **استراسن**

مسئله : تعیین حاصل ضرب دو ماتریس n ×n که در آن n توانی از ۲ است.

```
void starssen ( int n
                n × n _ matrix A,
                n × n _ matrix B,
                n × n _ matrix & C)
 {
   if ( n <= threshold)
      compute C = A × B using the standard algorithm;
           else {
        partition  A into  four  submatrics A11, A12 , A21,A22;
        partition  B into  four  submatrics B11, B12 , B21,B22;
        compute C = A × B using Starssen's Method;
      }
   }
```

تحلیل پیچیدگی زمانی تعداد ضرب ها در الگوریتم ۲-۸(استراسن)در حالت معمول

عمل اصلی: یک ضرب ساده.

اندازه ورودی: n ، تعداد سطرها و ستون ها در ماتریس.

به ازای $n > 1$ که n توانی از ۲ است $T(n) = 7\,T(n/2)$

$T(1) = 1$

$$T(n) \in \theta(n^{2.81})$$

تحلیل پیچیدگی زمانی تعدادجمع ها و تفریق های الگوریتم (استراسن)درحالت معمول

عمل اصلی: یک جمع یا تفریق ساده.

اندازه ورودی: n ، تعداد سطرها و ستون ها در ماتریس.

به ازای $n > 1$ که n توانی از ۲ است $T(n) = 7T(n/2) + 18\,(n/2)^2$

$T(1) = 1$

$$T(n) \in \theta(n^{2.81})$$

الگوریتم۲-۹: **ضرب اعداد صحیح بزرگ**

مسئله : ضرب دو عدد صحیح بزرگ u و v

```
large _ integer prod ( large_integer u, large_integer v)
  {
    large_integer x , y , w , z ;
    int n , m ;
   n = maximum(number of digits in u,number of digits in v)
   if  (u = = 0 || v = = 0)
```

```
        return 0 ;
    else if    (n < = threshold)
        return   u ×  v obtained in the  usual way;
    else {
         m = ⌊ n / 2 ⌋ ;
          x = u divide  10 ^ m  ;  y = rem 10 ^ m;
         w = v divide  10 ^ m  ;  z = rem 10 ^ m;
        return prod (x ,w) × 10 ^2m +  ( prod ( x, z) + prod
       (w, y )) × 10 ^ m + prod ( y, z);
       }
}
```

تحلیل پیچیدگی زمانی در بدترین حالت برای الگوریتم۲-۹(ضرب اعداد صحیح)

عمل اصلی: دستکاری یک رقم دهدهی در یک عدد صحیح بزرگ در هنگام جمع کردن ، تفریق کردن، یا انجام اعمال divide 10 ^ m، rem 10 ^m یا 10 ^ m×. هر یک از این اعمال را m بار انجام می دهد.

اندازه ورودی: n ، تعداد ارقام هر یک از دو عدد صحیح.

به ازای n > s که n توانی از ۲ است

$$W(n) = 4\,W(n/2) + cn$$

$$W(s) = 0$$

$$\mathbf{W(n) \in \theta(n^2)}$$

الگوریتم ۲-۱۰: **ضرب اعداد صحیح بزرگ ۲**

```
large_integer prod2 (large_integer u , large_ integer v)
 {
  large_integer x , y , w , z , r , p , q;
  int n , m;
  n = maximum (number of digits in u,number of digits in v);
  if  (u = = 0 || v = = 0)
        return 0 ;
  else if    (n < = threshold)
       return   u ×  v obtained in the  usual way;
else {
        m = ⌊ n / 2 ⌋;
          x = u divide  10 ^ m  ;  y = rem 10 ^ m;
          w = v divide  10 ^ m  ;  z = rem 10 ^ m;
           r =  prod2 (x + y, w + z );
           p = prod2 ( x , w )
           q = prod2 ( y , z );
           return  p ×10 ^ 2m + ( r – p – q ) × 10 ^ m +q ;
            }
    }
```

تحلیل پیچیدگی زمانی در بدترین حالت برای الگوریتم۲-۱۰(ضرب اعداد صحیح۲)

عمل اصلی: دستکاری یک رقم دهدهی در یک عدد صحیح بزرگ درهنگام جمع کردن ، تفریق کردن، یا انجام اعمال divide 10 ^ m، rem 10 ^m یا ×10 ^ m. هر یک از این اعمال را m بار انجام می دهد.

اندازه ورودی: n ، تعداد ارقام هر یک از دو عدد صحیح.

به ازای n > s که n توانی از ۲ است

$$3W(n/2)+ c\ n <= W(n) <= 3W(n / 2 +1) + c\ n$$

$$W(s) = 0$$

$$\mathbf{W(n) = \theta(n \wedge 1.58)}$$

فصل سوم:

برنامه نویسی پویا

- برنامه نویسی پویا، از این لحاظ که نمونه به نمونه های کوچک تر تقسیم می شود ، مشابه روش تقسیم و حل است ولی در این روش ، نخست نمونه های کوچک تر را حل می کنیم ، نتایج را ذخیره می کنیم و بعدا هر گاه به یکی از آن ها نیاز پیدا شد، به جای محاسبه دوباره کافی است آن را بازیابی کنیم.
- مراحل بسط یک الگوریتم برنامه نویسی پویا به شرح زیر است:

۱- ارائه یک ویژگی بازگشتی برای حل نمونه ای از مسئله .

۲- حل نمونه ای از مسئله به شیوه جزء به کل با حل نمونه های کوچک تر.

الگوریتم ۳-۱: **ضریب دو جمله ای با استفاده از تقسیم و حل**

```
int bin ( int n , int k)
 {
      if ( k == 0 || n ==k )
          return 1 ;
      else
          return bin (n – 1 , k -1 ) + bin ( n-1 , k);
  }
```

الگوریتم ۳-۲: **ضریب دو جمله ای با استفاده از برنامه نویسی پویا**

```
int bin2  ( int n , int k )
{
   index i , j ;
   int B [0..n][0..k]
   for  ( i = 0; i <= n ; i ++)
```

```
            if ( j == 0 || j == i )
                B [i] [j] = 1;
            else
                B [i] [j] = B [ i – 1 ] [ j -1 ] + B [ i -1 ] [j];
        return  B[n] [k]:
}
```

الگوریتم ۳-۳: **الگوریتم فلوید برای یافتن کوتاه ترین مسیر**

```
void  floyd ( int  n
                const  number W [][],
                       number D  [][],
    {
        index  i , j , k ;
        D  = W ;
        for ( k = 1 ; k <= n ; k ++)
            for ( i = 1; i <= n ; i++)
                for ( j = 1 ; j <= n ; j ++)
                    D [i] [j] = minimum ( D [i][j], D[i][k] + D[k][j]);
     }
```

تحلیل پیچیدگی زمانی در بدترین حالت برای الگوریتم۳-۳ (الگوریتم فلوید برای یافتن کوتاه ترین مسیر)

عمل اصلی: دستورهای موجود در حلقه for .

اندازه ورودی: n ، تعداد رئوس گراف.

$$T(n) = n \times n \times n = n^3 \in \theta(n^3)$$

الگوریتم ۳-۴: **الگوریتم فلوید برای یافتن کوتاه ترین مسیر ۲**

```
void  floyd2  ( int n,
                const number W [][],
                      number D [][],
                      index    P [][])
    {
        index  i , j , k ;
        for  ( i = 1 ;  i <= n ; i ++)
             for  ( j = 1 ; j <= n ; j++)
        P [i] [j] = 0;
         D = W;
    for ( k = 1 ;  k <= n ; k ++)
       for  ( i = 1 ;  i <= n ; i ++)
           for ( j = 1 ;  j <= n ; j ++)
                if ( D [i] [k] + D [k] [j] < D [i] [j] ) {
                      P [i] [j] = k;
                      D [i] [j] = D [i] [k] + D [k] [j];
                  }
            }
```

الگوریتم ۳-۵: **چاپ کوتاه ترین مسیر**

مسئله: چاپ رئوس واسطه روی کوتاه ترین مسیر از راسی به راس دیگر در یک گراف موزون.

```
void path ( index q , r)
 {
    if  (P [q] [r] != 0 )  {
          path (q , P [q] [r] );
          cout << "v" << P [q] [r];
          path (P [q] [r] , r );
        }
 }
```

۳-۳ برنامه نویسی پویا و مسائل بهینه سازی

- حل بهینه ، سومین مرحله از بسط یک الگوریتم برنامه نویسی پویا برای مسائل بهینه سازی است. مراحل بسط چنین الگوریتمی به شرح زیر است:

۱- ارائه یک ویژگی بازگشتی که حل بهینه ی نمونه ای از مسئله را به دست می دهد.

۲- محاسبه مقدار حل بهینه به شیوه ی جزء به کل.

۳- بنا کردن یک حل نمونه به شیوه ی جزء به کل.

- هر مسئله بهینه سازی را نمی توان با استفاده از برنامه نویسی پویا حل کرد چرا که باید اصل بهینگی در مسئله صدق کند.
- اصل بهینگی در یک مسئله صدق می کند اگریک حل بهینه برای نمونه ای از مسئله ، همواره حاوی حل بهینه برای همه ی زیر نمونه ها باشد.

۳-۴ ضرب زنجیره ای ماتریس ها

- هدف بسط الگوریتمی است که ترتیب بهینه را برای n ماتریس معین کند.
- ترتیب بهینه فقط به ابعاد ماتریس ها بستگی دارد.
- علاوه بر n ، این ابعاد تنها ورودی های الگوریتم هستند.
- این الگوریتم حداقل به صورت نمایی است.

الگوریتم ۳-۶: **حداقل ضرب ها**

```
int minimult ( int n,
               const  int d [],
               index   P [][] )
  {
      index  i , j , k , diagonal;
      int M [1..n] [1..n];
      for ( i = 1 ; i <= n ; i ++)
          M [i] [i] = 0:
  for (diagonal = 1; diagonal <= n -1 ; diagonal ++)
     for ( i = 1 ; i <= n – diagonal ; i ++) {
          j = i + diagonal ;
          M[i][j] = minimum (M[i][k] + M[k  +1 ][j] +
          d [ i – 1] * d [k] * d [j]);
          P [i][j] = a value of k that gave the minimum;
       }
    return M[1][n];
  }
```

تحلیل پیچیدگی زمانی حالت معمول برای الگوریتم۳-۶(حداقل ضرب ها)

عمل اصلی: می توان دستورات اجرا شده برای هر مقدار k را عمل اصلی در نظر بگیریم. مقایسه ای را که برای آزمون حداقل انجام می شود، به عنوان عمل اصلی در نظر می گیریم.

اندازه ورودی: n ، تعداد ماتریس ها که باید ضرب شوند.

$$N (n -1) (n + 1) / 6 \in \theta (n^3)$$

الگوریتم ۳-۷: **چاپ ترتیب بهینه**

مسئله :چاپ ترتیب بهینه برای ضرب n ماتریس.

```
void order ( index i, index j)
{
   if ( i == j)
         cout << "A" << i ;
   else  {
        k =  P [i] [j];
        cout << "(";
        order ( i , k);
        order ( k +1 , j );
        cout <<  ")";
     }
  }
```

۳-۵ درخت های جست و جوی دودویی بهینه

- درخت جست و جوی دودویی یک دودویی از عناصر(که معمولا کلید نامیده می شوند) است که از یک مجموعه مرتب حاصل می شود، به قسمی که:

۱- هر گره حاوی یک کلید است.

۲- کلید های موجود در زیردرخت چپ یک گره مفروض، کوچک تر یا مساوی کلید آن گره هستند.

۳- کلیدهای موجود درزیردرخت راست یک گره مفروض، بزرگ تر یا مساوی کلید آن گره هستند.

الگوریتم ۳-۸: **درخت جست و جوی دودویی**

```
void search ( node _  pointer tree ,
                keytype  keyin,
                node _ poiner & p)
  {
      bool found ;
      p = tree;
      found = false;
      while (! found)
    if ( p - > key == keyin )
          found = true ;
    else if ( keyin < p - > key )
           p = p  -> left ;
    else
         p = p - > right ;
  }
```

الگوریتم ۳-۹ : **درخت جست و جوی بهینه**

مسئله: تعیین یک درخت جست وجوی بهینه برای مجموعه ای از کلید ها، هر یک با احتمالی مشخص.

```
void optsearchtree  ( int n ,
                      const  float  p[];
                      float & minavg,
                      index   R[][])
    {
         index  i , j , k , diagonal ;
float A [1..n + 1] [0..n];
   for ( i = 1 ; i <= n ; i ++) {
        A [i] [i -1] = 0
        A [i] [i]  = p [i];
         R [i] [i] = i ;
         R [i] [ i – 1 ] = 0;
   }
      A[ n + 1 ] [n] = 0 ;
      R[ n + 1 ] [n] = 0 ;
  for (diagonal = 1;diagonal <= n – 1; diagonal++)
        for  ( i = 1; i <= n – diagonal ; i ++) {
          j = i + diagonal ;
         A[i][j] = minimum(A[i][k-1]+A[k+1][j])+∑pm
         R[i][j] = a value of k that gave the minimum;
        }
     minavg = A [1][n];
    }
```

تحلیل پیچیدگی زمانی حالت معمول برای الگوریتم درخت جستجوی دودویی بهینه

عمل اصلی: دستورات اجرا شده به ازای هر مقدار از k .این دستورات شامل یک مقایسه برای آزمون حداقل است.

اندازه ورودی: n ، تعداد کلید.

$$T(n) = n (n -1) (n + 4) / 6 \in \theta (n^3)$$

الگوریتم ۳ -۱۰: **ساخت درخت جست و جوی دودویی بهینه**

```
node _ pointer tree ( index i , j )
 {
     index k ;
     node _ pointer p ;
     k = R [i] [j];
     if ( k == 0 )
          return NULL;
     else  {
p =  new nodetype ;
    p - > key = key [k] ;
    p - > left = tree (i , k – 1);
    p - > right = tree ( k+1 , j);
    return  P;
      }
    }
```

الگوریتم۳-۱۱: **الگوریتم برنامه نویسی پویا برای مسئله فروشنده دوره گرد**

```
void tarvel ( int , n
                const number W[ ][ ] ,
                index  P[ ] [ ] ,
                number & minlength)
    {
        index  i , j , k ;
        number D[1..n][subset of  V -  { v1 } ] ;
        for ( i = 2 ; i <= n ; i ++)
D[i] [Ø] = W [i] [1] ;
   for (k = 1 ; k <= n - 2 ; k ++)
      for (all subsets A Ç V – { v1 } containing k vertices)
        for (i such that  i  !=  1 and  vi  is not inA){
        D[i] [A] = minimum ( W [i] [j] + D [j] [A – {vj}]);
        P [i] [A] = value of j that gave the minimum;
D [1] [ V – { v1 } ] = minimum ( W [1] [j] +
    D [j]  [ V – { v1 – v j } ] );
       P [1] [ V – { v1 } ] = value of  j  that gave the minimum;
      minlength = D [1] [ V – { v1 } ];
    }
```

تحلیل پیچیدگی فضا و زمان در حالت معمول برای الگوریتم ۳-۱۱ (الگوریتم برنامه نویسی پویا برای مسئله فروشنده دوره گرد)

عمل اصلی: زمان در هر دو حلقه ی اول و آخر ، در مقایسه با زمان در حلقه میانی چشم گیر نیست، زیرا حلقه میانی حاوی سطوح گوناگون تودرتویی است . دستورات اجرا شده برای هر مقدار v j را می توان عمل اصلی در نظر گرفت.

اندازه ورودی : n ، تعداد رئوس موجود در گراف.

$$T(n) = n\,2^n \in \theta(n\,2^n)$$

فصل چهارم:

روش حریصانه در طراحی الگوریتم

- الگوریتم حریصانه ، به ترتیب عناصر را گرفته ، هر بار آن عنصری را که طبق ملاکی معین "بهترین" به نظر می رسد، بدون توجه به انتخاب هایی که قبلا انجام داده یا در آینده انجام خواهد داد، بر می دارد.
- الگوریتم حریصانه ، همانند برنامه نویسی پویا غالبا برای حل مسائل بهینه سازی به کار می روند، ولی روش حریصانه صراحت بیشتری دارد.
- در روش حریصانه ، تقسیم به نمونه های کوچک تر صورت نمی پذیرد.
- الگوریتم حریصانه با انجام یک سری انتخاب، که هر یک در لحظه ای خاص ، بهترین به نظر می رسد عمل می کند، یعنی انتخاب در جای خود بهینه است.امید این است که یک حل بهینه سرتاسری یافت شود، ولی همواره چنین نیست.
- برای یک الگوریتم مفروض باید تعیین کرد که آیا حل همواره بهینه است یا خیر.
- الگوریتم حریصانه ، کار را با یک مجموعه تهی آغاز کرده به ترتیب عناصری به مجموعه اضافه می کند تا این مجموعه حلی برای نمونه ای از یک مسئله را نشان دهد.

هر دور تکرار ، شامل مولفه های زیر است:

۱- **روال انتخاب**، عنصربعدی را که باید به مجموعه اضافه شود، انتخاب می کند.انتخاب طبق یک ملاک حریصانه است.

۲- **بررسی امکان سنجی**، تعیین می کند که آیا مجموعه جدید برای رسیدن به حل، عملی است یا خیر.

۳- **بررسی راه حل**، تعیین می کند که آیا مجموعه جدید، حل نمونه را ارائه می کند یا خیر.

۴-۱ درخت های پوشای کمینه

- فرض کنید طراح شهری می خواهد چند شهر معین را با جاده به هم وصل کند، به قسمی که مردم بتوانند از هر شهر به شهر دیگر بروند. اگر محدودیت بودجه ای در کار باشد ، ممکن است طراح بخواهد این کار را با حداقل مقدار جاده کشی انجام دهد.
- برای این مسئله دو الگوریتم حریصانه متفاوت : پریم و کروسکال بررسی می شود.
- هر یک از این الگوریتم ها از یک ویژگی بهینه محلی استفاده می کند.
- تضمینی وجود ندارد که یک الگوریتم حریصانه همواره حل بهینه بدهد، ثابت می شود که الگوریتم های کروسکال و پریم همواره درخت های پوشای کمینه را ایجاد می کنند.

۴-۱-۱ الگوریتم پریم

- الگوریتم پریم با زیر مجموعه ای تهی از یال های F و زیرمجموعه ای از رئوس Y آغاز می شود، زیرمجموعه حاوی یک راس دلخواه است. به عنوان مقداراولیه، $\{v1\}$ را به Y می دهیم. نزدیک ترین راس به Y، راسی در $V - Y$ است که توسط یالی با وزن کمینه به راسی در Y متصل است.

الگوریتم ۴-۱: **الگوریتم پریم**

```
void prim ( int n,
            const number W[ ] [ ],
            set_ of_edges & F )
  {
      index i , vnear;
      number  min;
      edge  e;
      index   nearest [2..n];
```

```
number distance [2..n];
F = Ø ;
for ( i = 2 ; i <= n ; i ++) {
        narest [i] = 1 ;
        distance [i] = W [1] [i] ;
 }
 repeat ( n-1 times )  {
    min = ∞ ;
     for ( i = 2 ; i <= n ; i ++)
         if ( 0 <= distance [i] < min )  {
    min = distance [i] ;
    vnear = i ;
  }
e = edge connecting vertices indexed by
        near and  nearest [ vnear ] ;
add e to F ;
distance [ vnear ]  = -1 ;
for ( i = 2 ; I <= n ; i ++)
if ( W[i] [ vnear ] < distance [i])  {
         distance [i] = W [i] [ vnear ] ;
         nearest [i] = vnear ;
```

```
        }
    }
}
```

تحلیل پیچیدگی زمانی در حالت معمول برای الگوریتم ۴-۱(الگوریتم پریم)

عمل اصلی: در حلقه repeat دو حلقه وجود دارد که هر یک (n – 1) بار تکرار می شود . اجرای دستورات داخل هر یک از آن ها را می توان به عنوان یک بار اجرای عمل اصل در نظر گرفت.

اندازه ورودی: n ، تعداد رئوس.

$$T(n) = 2(n-1)(n-1) \in \theta(n^2)$$

قضیه ۴-۱

■ **الگوریتم پریم همواره یک درخت پوشای کمینه تولید می کند.**

اثبات : برای آن که نشان دهیم مجموعه F پس از هر بار تکرارحلقه **repeat**، امید بخش است ازاستقرا استفاده می کنیم.

مبنای استقرا : واضح است که مجموعه Ø امید بخش است.

الگوریتم ۴-۲: **الگوریتم کروسکال**

```
void kruskal ( int n , int m,
                    set _ of _ edges E,
                    set _ of _edges & F )
  {
      index i , j ;
      set _pointer p , q;
      edge e ;
      sort  the m edges in E by weight in
nondecreasing order;
      F = Ø ;
      intitial (n) ;
      while( number of edges in F is less than n-1){
            e = edge with least weight not  yet
                considered ;
            i , j = indices of vertices connected by e;
            p = find (i) ;
            q = find (i) ;
if (!  equal  ( p, q ))  {
           merge ( p , q ) ;
           add e to F ;
```

```
        }
    }
}
```

تحلیل پیچیدگی زمانی در بدترین حالت برای الگوریتم ۴-۲(الگوریتم کروسکال)

عمل اصلی: یک دستور مقایسه.

اندازه ورودی : n ، تعداد رئوس و m تعداد یال ها.

درباره این الگوریتم سه نکته را باید در نظر داشت:

۱- زمان لازم برای مرتب سازی یال ها .

$$W(m) \in \theta(m \lg m)$$

۲- زمان در حلقه while .

$$W(m) \in \theta(m \lg m)$$

۳- زمان لازم برا ی مقدار دهی اولیه به n مجموعه متمایز .

$$W(m, n) \in \theta(m \lg m)$$

در نتیجه ، بدترین حالت:

$$W(m, n) \in \theta(n^2 \lg n^2) = \theta(n^2 \lg n)$$

قضیه ۴-۲

- **الگوریتم کروسکال همواره یک درخت پوشای کمینه تولید می کند.**

اثبات : اثبات از طریق استقرا با شروع از مجموعه ای تهی از یال ها آغاز می شود.

۴-۲ الگوریتم دیکسترا برای کوتاه ترین مسیر تک مبدا

- برای کوتاه ترین مسیر از هر راس به همه رئوس دیگر در یک گراف موزون و بدون جهت یک الگوریتم$\theta(n^2)$ از روش حریصانه داریم، که آن دیکسترا نام دارد.
- الگوریتم را با این فرض ارائه می دهیم که از راس مورد نظر به هر یک از رئوس دیگر، مسیری وجود دارد.
- این الگوریتم مشابه الگوریتم پریم برای مسئله درخت پوشای کمینه است.

الگوریتم۴-۳: **الگوریتم دیکسترا**

```
void dijkstra ( int n,
                const number W[ ] [ ],
                set _ of _ edges & F)
    {
        index i , vnear,
        edge e ;
        index touch [2..n];
        number length[2..n];
      F = Ø ;
      for ( i = 2; i <=  n ; i ++ )
```

```
{
        touch [i] = 1 ;
        length [i]  = W [1][i];
    }
    repeat ( n – 1 times ) {
        min = ∞ ;
        for ( i = 2 ; i <= n ; i ++)
            if ( 0 <= length [i] < min )  {
                min = length [i] ;
      vnear = i ;
    }
  e = edge from vertix indexed by touch [vnear]
      to vertex indexed by vnear;
  add e to F;
  for ( i = 2 ; i <= n; i ++)
     if ( length [vnear] + W [vnear] [i] < length[i]) {
        length [i] = length [vnear] + W [vnear] [i];
      touch [i] = vnear ;
    }
   length [vnear]  = -1;
  }
```

```
}
```

قضیه ۴-۳

تنها زمان بندی که زمان کل درسیستم را کمینه سازی می کند، زمان بندی است که در آن کارها بر حسب افزایش زمان ارائه خدمات مرتب می شوند.

الگوریتم ۴-۴ : **زمان بندی با مهلت معین**

مسئله : تعیین زمان بندی با سود کل بیشینه ، با این فرض که هر کاری دارای سود است و فقط وقتی قابل حصول است که آن کار در مهلت مقررش انجام شود.

```
void schedule ( int n ,
                const int deadline [ ] ,
                sequence _ of _ integer& j )
 {
     index  i ;
  sequence_ of_integer K ;
  j = [1];
  for ( i = 2 ; i <= n ; i ++) {
  K = J with i added according to nondecreasing
       values of deadline [i];
       if ( K  is  feasible)
           J = K;
  }
}
```

تحلیل پیچیدگی زمانی در بدترین حالت برای الگوریتم زمان بندی با مهلت معین

عمل اصلی: باید برای مرتب سازی کارها، مقایسه هایی انجام پذیرد و هنگامی که K با J (پس از افزوده شدن کار i ام) مساوی قرار داده می شود، به مقایسه های بیشتری نیاز داریم و هنگامی که امکان پذیر بودن K چک می شود، به مقایسه های بیشتری نیاز است. عمل اصلی مقایسه است.

اندازه ورودی : n ، تعداد کارها.

$$W(n) \in \theta(n^2)$$

قضیه ۴-۴

الگوریتم ۴-۴ (زمان بندی با مهلت معین) همواره یک مجموعه بهینه تولید می کند.

اثبات: ازطریق استقرا روی تعداد کارها،n، صورت می پذیرد.

مبنای استقرا: اگر یک کار داشته باشیم ، قضیه بر قراراست.

قضیه ۴-۵

الگوریتم هافمن یک کد دودویی بهینه تولید می کند.

اثبات : ازطریق استقرا صورت می گیرد. با این فرض که درخت های به دست آمده درمرحله i، انشعاب هایی در درخت دودویی متناظر با کد بهینه اند، نشان می دهیم که درخت های به دست آمده در مرحله (i + 1) نیز انشعاب هایی در درخت دودویی متناظر با یک کد بهینه اند.

فصل پنجم:

راهبرد عقبگرد

- از تکنیک عقبگرد برای حل مسائلی استفاده می شود که در آن ها دنباله ای از اشیاء از یک مجموعه مشخص انتخاب می شود، به طوری که این دنباله ، ملاکی را در بر می گیرد.
- یک مثال کلاسیک از عقبگرد، مسئله n وزیر است.
- هدف از مسئله n وزیر ، چیدن n مهره وزیر در یک صفحه شطرنج است ، به طوری که هیچ دو وزیری یکدیگر را گارد ندهند. یعنی هیچ دو مهره ای نباید در یک سطر، ستون یا قطر یکسان باشند.
- عقبگرد حالت اصلاح شده ی جست و جوی عمقی یک درخت است.
- الگوریتم عقبگرد همانند جست و جوی عمقی است، با این تفاوت که فرزندان یک گره فقط هنگامی ملاقات می شوند که گره امید بخش باشد و در آن گره حلی وجود نداشته باشد.

الگوریتم ۵-۱: **الگوریتم عقبگرد برای مسئله n وزیر**

```
void queens ( index i)
    {
        index j;
        if ( promising(i))
           if ( i == n)
                cout << col [1] through col [n];
           else
                for ( j = 1 ; j <= n ; j++ ) {
```

```
col [ i +1 ] = j;
        queens ( i + 1);
      }
}
bool promising  ( index i )
{
     index k ;
     bool  switch;
     k = 1;
     switch  = true ;
      while ( k < i && switch ) {
        if (col [i] == col[k] || abs(col[i] – col[k] == i-k)
            switch = false;
        k++;
       }
    return switch;
  }
```

۵-۳ استفاده از الگوریتم مونت کارلو برای برآورد کردن کارایی یک الگوریتم عقبگرد

- الگوریتم های مونت کارلو، احتمالی هستند، یعنی دستور اجرایی بعدی گاه به طور تصادفی تعیین می شوند.
- در الگوریتم قطعی چنین چیزی رخ نمی دهد.
- همه ی الگوریتم هایی که تا کنون بحث شدند، قطعی هستند.
- الگوریتم مونت کارلو مقدار مورد انتظار یک متغیر تصادفی را که روی یک فضای ساده تعریف می شود ، با استفاده از مقدار میانگین آن روی نمونه تصادفی از فضای ساده بر آورد می کند.
- تضمینی وجود ندارد که این برآورد به مقدار مورد انتظار واقعی نزدیک باشد، ولی احتمال نزدیک شدن آن، با افزایش زمان در دسترس برای الگوریتم، افزایش می یابد.

الگوریتم۵ -۲ : **برآورد مونت کارلو**

```
int estimate ()
{
   node v  ;
   int m, mprod , numnodes;
   v = root of state space tree;
   numnodes  = 1;
   m = 1;
   mprod = 1;
while ( m != 0)
{
```

```
        t = number of children  of  v ;
        mprod  - mprod * m;
        numnodes  = numnodes  + mprod  * t;
        m = number of promising children of v;
        if ( m != 0)
            v = randomly selected promising
                child of v ;
       }
        return numnodes;
  }
```

الگوریتم ۵ -۳: **برآورد مونت کارلو برای الگوریتم ۵ -۱ (الگوریتم عقبگرد برای مسئله n وزیر)**

```
int ostimate _ n_ queens (int n)
{
    index i , j , col [1..n];
    int m , mprod , numnodes ;
    set _of_ index  prom _children;
    i = 0;
    numnodes =1 ;
    m = 1;
   mprod  = 1 ;
```

```
        while  ( m != 0 && i != n ) {
        mprod = mprod * m ;
        numnodes  = numnodes + mprod * n;
        i ++;
        m = 0 ;
        prom_childern  = Ø;
        for ( j = 1 ; j <= n ; j++;) {
              col [i]  = j ;
              if ( promising(i)) {
m++;
         prom_children = prom _ children U {i};
               }
            }
            if ( m != 0)  {
               j = random selection from prom _ children ;
               col [i];
             }
        }
        return numnodes;
    }
```

الگوریتم ۵-۴: **الگوریتم عقبگرد برای مسئله حاصل جمع زیر مجموعه ها**

مسئله : تعیین همه ی ترکیبات اعداد صحیح موجود در یک مجموعه n عدد صحیح ، به طوری که حاصل جمع آن ها مساوی مقدار معین W شود.

```
void sum _of _subsets ( index i ,
                        int weight , int total)
  {
     if (promising(i))
         if  ( weight == W )
            cout << include [1] through include [i];
          else  {
               include [i +1] = “yes” ;
     sum_ of_subsets ( i +1 , weight + w [i +1],
     total – w [ i + 1 ]);
       }
    }
    bool promising ( index i);
   {
   return (weight + total >= W) &&( weight == W ||
    weight + w [ i + 1 ]  <= W );
    }
```

۵-۵ رنگ آمیزی گراف

- مسئله رنگ آمیزی m ، عبارت از یافتن همه ی راه ها ی ممکن برای رنگ آمیزی یک گراف بدون جهت ، با استفاده از حداکثر m رنگ متفاوت است، به طوری که هیچ دو راس مجاوری هم رنگ نباشند.

الگوریتم۵-۵: **الگوریتم عقبگرد برای مسئله رنگ آمیزی m**

```
void m_coloring (index i )
    {
        int color ;
        if ( promising (i))
            if ( i == n)
                cout << vcolor[1] through vcolor [n];
            else
                for (color = 1; color <=m; color ++)  {
       vcolor [ i + 1] = color ;
       m_coloring (i + 1);
      }
  }
  bool promising ( index i )
   {
      index j ;
      bool switch ;
```

```
    switch = true;
  j = 1;
  while ( j < i  && switch ) {
      if ( W [i] [j] && vcolor [i] == vcolor[j])
           switch = false ;
      j ++;
    }
    return  switch ;
 }
```

الگوریتم ۵-۶: **الگوریتم عقبگرد برای مسئله مدارهای هامیلتونی**

```
void hamiltonian  ( index i )
{
    index j ;
    if ( promising (i)
      if ( i == n - 1)
        cout << vindex [0] through  vindex [ n-1];
      else
        for  ( j = 2 ; j <= n ; j ++) {
       vindex [ i +1] = j ;
       hamiltonian ( i +1 );
    }
```

```
    }
    bool promising ( index i )
    {
      index j ;
      bool switch ;
      if( i == n -1 &&! W [vindex [ n-1 ]] [ vindex [0]])
           switch = false ;
    else {
        switch = true ;
        j = 1;
while ( j < i && switch ) {
         if (vindex [i] == vindex [j])
              switch = false ;
         j++;
        }
      }
     return switch;
  }
```

۵-۷ مسئله کوله پشتی صفر و یک

- در این مسئله مجموعه ای از قطعات داریم که هر یک دارای وزن و ارزش معین است.
- اوزان و ارزش ها اعداد مثبتی هستند.
- دزدی درنظر دارد قطعاتی که می دزدد درون یک کوله پشتی قرار دهد و اگر وزن کل قطعات قرار داده شده در آن کوله پشتی از یک عدد صحیح مثبتW فراتر رود، کوله پشتی پاره می شود.

الگوریتم ۵-۷: **الگوریتم عقبگرد برای مسئله کوله پشتی صفر و یک**

```
void knapsack ( index i ,
                int profit , int weight)
  {
    if ( weight <= W &&  profit > maxprofit ) {
       maxprofit = profit ;
       numbest  = i ;
       bestset = include;
    }
    if ( promising (i)) {
     include [ i + 1 ] = “yes”;
     knapsack ( i + 1, profit + p [ i + 1] , weight +
      w [ i +1 ]);
     include [ i +1] = “ no”;
     knapsachk ( i +1 , profit , weight );
    }
```

```
}
bool promising ( index i )
{
    index j , k ;
int totweight ;
float bound;
if ( weight >=  W)
       return  false ;
{
     j = i +1 ;
     bound = profit ;
     totweight = weight ;
     while ( j <= n && totweight + w[j] <= W) {
     totweight  = totweight + W [j];
  bound = bound + p[j];
  j++;
 }
 k = j;
 if ( k <= n)
  bound = bound + (W – totweight) * p [k]/w[k];
 return bound > max profit ;
```

```
    }
}
```

۵-۷-۱ مقایسه الگوریتم برنامه نویسی پویا و الگوریتم عقبگرد برای مسئله کوله پشتی صفر و یک

- تعداد عناصری که توسط الگوریتم برنامه نویسی پویا برای مسئله کوله پشتی صفر و یک محاسبه می شود دربدترین حالت به $O(\text{minimum}(2^n, nW))$ تعلق دارد.
- در بدترین حالت ، الگوریتم عقبگرد $\theta(2^n)$ گره را چک می کند.
- الگوریتم عقبگرد معمولا کارایی بیشتری نسبت به الگوریتم برنامه نویسی پویا دارد.
- هورورویتز و شانی ، روش تقسیم و حل را با روش برنامه نویسی پویا ترکیب کرده الگوریتمی برای مسئله کوله پشتی صفر و یک بسط داده اند که دربدترین حالت به $O(2^{n/2})$ تعلق دارد.

فصل ششم:

راهبرد شاخه و حد

- راهبرد شاخه و حد ازآن لحاظ به عقبگرد شباهت دارد که درآن، برپا حل مسئله از درخت فضای حالت استفاده می شود.
- تفاوت این روش با عقبگرد، اولا ما را به پیمایش خاصی ازدرخت محدود نمی کند وثانیا فقط برای مسائل بهینه سازی به کار می رود.
- الگوریتم شاخه و حد ، در هر گره عددی (حدی) را محاسبه می کند تا تعیین شود که آیا این گره امید بخش هست یا خیر.
- اگر آن حد بهتر از مقدار بهترین حلی که تاکنون یافته شده، نباشد، گره غیر امید بخش است. در غیر این صورت ، امید بخش است.
- علاوه براستفاده از حد، می توان حدود گره های امید بخش را مقایسه کرد و فرزندان گرهی با بهترین حد را ملاقات نمود. بدین ترتیب می توان سریع تر از آن که گره ها را در یک ترتیب از پیش تعیین شده ملاحظه کرد، به حل بهینه دست یافت.
- این روش را بهترین جست و جو با هرس کردن شاخه و حد می گویند.
- پیاده سازی این روش، شکل اصلاح شده ی ساده ای از یک روش دیگر موسوم به جست و جوی عرضی هرس کردن شاخه و حد است.

الگوریتم ۶-۱: **الگوریتم جست و جوی عرضی با هرس کردن شاخه و حد برای مسئله کوله پشتی صفر و یک**

```
void knapsack ( int n ,
                const int p [ ] , const int w [ ],
                int W ,
                int & maxprofit )
  {
      queue _of _ node Q;
      node  u , v ;
      intialize (Q);
   v.level = 0 ; v.profit = 0 ; v.weight = 0 ;
  maxprofit = 0 ;
  enqueue (Q , v);
  while (!empty (Q))  {
    dequeue ( Q , v );
    u.level  = v.level + 1;
    u.weight = v. weight + w [ u.level];
    u. profit = v.profit + p [ u.level];
    if (  u.weight <= W && u.profit > maxprofit)
       maxprofit = u.profit;
 if ( bound (u) > maxprofit )
        enqueue (Q, u) ;
```

```
        u.weight = v. weight;
        u. profit = v.profit;
        if ( bound (u) > maxprofit )
            enqueue (Q , u);
      }
  }
  float bound ( node u)
  }
   index j, k ;
   int  totweight;
   float  result ;
   if ( u.weight >= W)
        return 0;
   else  {
         result = u.profit;
         j = u.level + 1;
         totweight  = u.weight;
         while( j <= n && totweight +w [j] <= W) {
totweight = totweight + w [j];
 result  = result + p [j];
 j++;
```

```
    }
    k = j ;
    if ( k <= n )
      result = result +( W – totweight ) * p [k] / w [k];
    return result;
  }
}
```

الگوریتم ۶-۲: **بهترین جست و جو با هرس کردن شاخه و حد برای مسئله کوله پشتی صفر و یک**

```
void  knapsack ( int n ,
                 const int p [ ] , const int w [ ],
                 int W ,
                 int & maxproit)
  {
      priority _ queue_of _node PQ;
      node u , v ;
      initialize (PQ) ;
v.level = 0 ; v.profit = 0 ; v.weight = 0 ;
      maxprofit  = 0 ;
      v.bound = bound (v);
      insert (PQ , v);
```

```
        while (! Empty(PQ))  {
              remove (PQ , v);
              if( v.bound > maxprofit )   {
                  u.level = v.level + 1;
                  u.weight = v.weight + w [ u.level ];
                  u. profit = v.profit + p [ u.level ] ;
if ( u.weight <= W && u.profit  > maxprofit  )
            maxprofit  = u.profit;
     u.bound  = bound (u);
     if ( u.bound  > maxprofit )
            insert (PQ , u);
            u.weight = v.weight ;
            u. profit = v.profit ;
            u.bound =  bound (u) ;
            if ( u.bound > maxprofit )
                insert (PQ <u );
          }
       }
     }
```

۶-۲ مسئله فروشنده دوره گرد

- هدف از این مسئله یافتن کوتاه ترین مسیر در یک گراف جهت دار با شروع از یک راس مفروض است ، مشروط بر آن که هر راس فقط یک بار ملاقات شود. چنین مسیری را یک تور می گویند.

الگوریتم ۶-۳: **الگوریتم بهترین جستجو با هرس کردن شاخه و حد برای مسئله فروشنده دوره گرد**

```
void travel2 ( int n,
               const number W [ ] [ ] ,
               ordered-set & opttour ,
               number & minlength )
    {
      priority _ queue _ of _ node PQ;
      node u , v ;
      initialize  ( PQ ) ;
       v.level = 0 ;
  v.path  = [1];
  v.bound  = bound  (v) ;
  minlength  = ∞ ;
  insert (PQ , v ) ;
  while ( ! Empty (PQ))  {
      remove( PQ, v );
      if  ( v.bound < minlength )  {
        u.level    = v.level + 1;
```

```
        for (all i such that 2 <= i <= n &&  i is not in v.path){
        u.path = v.path ;
        put i at the end of  u.path;
        if  ( u.level == n – 2 )   {
              put  index of only vertex
              put in u.path at the end of path;
              put 1 at the end of  u.path;
              if ( length (u) < minlength )  {
                  minlength = length (u);
      opttour  = u.path ;
               }
            }
else
{
            u.bound = bound (u);
            if ( u.bound  < minlength )
                 insert (PQ , u );
          }
        }
      }
    }
```

۶-۳ استنباط فرضیه ای (تشخیص بیماری)

- یک مسئله مهم در هوش مصنوعی و سیستم های خبره، تعیین محتمل ترین توضیح برای برخی یافته ها است.
- برا ی مثال، در پزشکی می خواهیم محتمل ترین مجموعه از بیماری ها را که از یک سری علائم قابل نتیجه گیری است ، تعیین کنیم.
- فرایند تعیین محتمل ترین توضیح برای یک مجموعه یافته ها را **استنباط از طریق فرضیه** می نامیم.
- **شبکه باور** استانداردی برای نشان دادن روابط احتمالی ، نظیر روابط میان بیماری و نشانه ها به شمار می رود.

الگوریتم ۶-۴ : **الگوریتم بهترین جست و جو با هرس کردن شاخه و حد برای استنباط فرضیه ای (الگوریتم کوپر)**

```
void cooper ( int n ,
        Bayesian_network_of_n_diseases BN,
      set _ of _symptoms S ,
        set_ of _indices & best , float & pbest )
{
    priority_queue_of_node PQ ;
    node u , v ;
    v.level = 0 ;
   v.D = Ø ;
   best = Ø ;
   pbest =  p (Ø | S );
```

```
v.bound  = bound (v);
insert  (PQ , v );
while  ( ! Empty (PQ))  {
        remove  (PQ , v);
        if ( v.bound > pbest )  {
            u.level  = v.level + 1;
            u.D = v.D;
   put u.level in u.D;
   if ( p ( u.D  | S ) > pbest)  {
       best =  u.D;
       pbest = p ( u.D  | S );
      }
      u.bound = bound(u);
      if ( u.bouond  > pbest )
         insert (PQ, u);
      u.D = v.D;
      u.bound = bound (u);
       if ( u.bound > pbest )
                insert (PQ , u );
     }
   }
```

```
}
int bound ( node u )
{
    if (u.level  == n )
        return 0 ;
    else
        return p(u.D  | p (S);
 }
```

فصل هفتم:

مقدمه ای بر پیچیدگی محاسباتی، مسئله مرتب سازی

۷- ۱ پیچیدگی محاسباتی

- پیچیدگی محاسباتی عبارت از مطالعه تمام الگوهای امکان پذیر برای حل یک مسئله مفروض است.
- در تحلیل پیچیدگی محاسباتی کوشش می کنیم تا حد پایینی کارایی همه ی الگوریتم ها را برای یک مسئله مفروض به دست آوریم.
- تحلیل پیچیدگی محاسباتی را با مطالعه مسئله مرتب سازی معرفی می کنیم.
- این انتخاب دو دلیل دارد:

۱- چند الگوریتم ابداع شده اند که که مسئله را حل می کنند.

۲- مسئله مرتب سازی یکی از معدود مسائلی است که در بسط الگوریتم هایی با پیچیدگی زمانی نزدیک به حد پایینی برای آن موفق بوده ایم.

۷-۲ مرتب سازی درجی و مرتب سازی انتخابی

- الگوریتم مرتب سازی درجی الگوریتمی است که مرتب سازی را با درج رکوردها در یک آرایه مرتب شده ی موجود مرتب سازی می کند.

الگوریتم ۷-۱: **مرتب سازی درجی**

```
void insertionsort  ( int n , keytype S[ ] )
   {
       index i , j ;
       keytype  x ;
       for ( i = 2 ; i <= n ; i ++) {
            x = S [i];
            j = i + 1;
while ( j >0 && S [i] > x )  {
      S [ j + 1 ] = S [j];
      j - - ;
    }
    S [ j + 1 ] = x ;
  }
}
```

تحلیل پیچیدگی زمانی تعداد مقایسه های کلید ها در الگوریتم مرتب سازی درجی در بدترین حالت

عمل اصلی : مقایسه S [j] با x.

اندازه ورودی : n ، تعداد کلید هایی که باید مرتب شوند.

$$W(n) = n(n-1)/2$$

تحلیل پیچیدگی زمانی تعداد مقایسه های کلید ها در الگوریتم مرتب سازی درجی در حالت میانگین

عمل اصلی : مقایسه S [j] با x.

اندازه ورودی : n ، تعداد کلید هایی که باید مرتب شوند.

$$A(n) = n^2/4$$

تحلیل استفاده از فضای اضافی برای الگوریتم ۷-۱ (مرتب سازی درجی)

- تنها فضایی که با n افزایش می یابد، اندازه ی آرایه ورودی S است. پس ،این الگوریتم یک مرتب سازی درجا است و فضای اضافی به $\theta(1)$ تعلق دارد.

جدول ۷-۱ : خلاصه تحلیل مرتب سازی تعویضی ، درجی و انتخابی

الگوریتم	مقایسه کلید	انتساب رکورد ها	استفاده ازفضای اضافی
تعویضی	$T(n) = n^2 / 2$	$W(n) = 3n^2/2$ $A(n) = 3n^2/2$	درجا
درجی	$W(n) = n^2 / 2$ $A(n) = n^2 / 4$	$W(n) = n^2 / 2$ $A(n) = n^2 / 4$	درجا
انتخابی	$T(n) = n^2 / 2$	$T(n)= 3n$	درجا

الگوریتم ۷-۲: **مرتب سازی انتخابی**

```
void selectionsort ( int n , keytype S [ ] )
 {
        index i , j , smallest ;
        for ( i = 1; i <= n-1 ; i ++ )
            if ( S [j] < S [ smallest])
                  smallest = j ;
             exchange S [i] and S [smsllest ] ;
           }
     }
```

قضیه ۷-۱

- هر الگوریتمی که n کلید متمایز را فقط از طریق مقایسه کلید ها انجام دهد و پس از هر بار مقایسه ، حد اکثر یک وارونگی را حذف کند، باید در بدترین حالت حداقل $n(n-1)/4$ مقایسه روی کلید ها انجام دهد.

الگوریتم مرتب سازی تعویضی

```
void  exchangesort ( int n , keytype S [ ] )
{
    index i , j ;
    for ( i = 1 ; i <= n -1 ; i ++ )
       if  ( S [j] < S [i] )
            exchange S [i] and S [j];
 }
```

۴-۷ نگاهی دوباره به مرتب سازی ادغامی

- پیچیدگی زمانی تعداد مقایسه های کلید ها در مرتب سازی ادغامی در بدترین حالت، وقتی که n توانی از 2 است، به طور کلی به $\theta(n \lg n)$ تعلق دارد:

$$W(n) = n \lg n - (n - 1)$$

- پیچیدگی زمانی تعداد مقایسه های رکورد ها در مرتب سازی ادغامی در حالت معمول تقریبا به صورت زیر است:

$$T(n) = 2n \lg n$$

بهبود بخشیدن به مرتب سازی ادغامی

- می توانیم مرتب سازی ادغامی را به سه شیوه بهبود ببخشیم:

۱- نسخه ای ازمرتب سازی ادغامی به روش برنامه نویسی پویا

۲- نسخه پیوندی

۳- الگوریتم ادغامی پیچیده تر

الگوریتم ۷-۳: مرتب سازی ادغامی ۳ (نسخه برنامه نویسی پویا)

```
void mergesort ( int n , keytype S [ ] )
{
    int m ;
    index  low , mid , high , size ;
    m = 2 ^ [ lg n] ;
    size = 1 ;
    repeat (lgm times )  {
        for ( low = 1; low <= m -2 * size + 1 ;
            low = low + 2 * size )   {
mid = low + size -1 ;
    high = minimun ( low + 2 * size – 1 , n );
    merge3 ( low , mid , high , S );
  }
  size = 2 * size ;
```

```
    }
  }
```

الگوریتم ۷-۴ : **مرتب سازی ادغامی ۴ (نسخه پیوندی)**

```
void mergesort4(int low,indexhigh, index&mergedlist)
  {
      index mid , list 1 , list 2;
      if ( low == high ) {
              mergedlist  = low ;
              S [ mergedlist ].link = 0 ;
       }
       else {
mid  = ⌊ (low + high ) / 2⌋;
       mergesort4 ( low , mid , list1 );
       mergesort4 ( mid + 1, high , list 2);
       mergesort4 ( list1 , list 2, mergedlist );
     }
}
void merge4(index list1,indexlist2,index& mergedlist)
{
       index lastsorted ;
```

```
if S [ list 1].key  < S [list 2 ] .key {
        mergedlist = list1;
        list1 = S [list 2 ].link;
    }
    else  {
          mergedlist = list 2;
          list 2 = S [list 2 ] .link;
    }
   lastsorted = mergedlist;
while ( list1 != 0 && list2 != 0 )
    if S [list 1].key < S [list 2].key   {
   S [ lastsorted ].link = list 1 ;
     lastsorted = list1;
     list 1 = S [ list 1] .link;
}
 else  {
    S [ lastsorted ].link = list 2 ;
     lastsorted = list2;
list 2 = S [ list 2] .link;
 }
 if ( list 1 ==0)
```

```
            S [ lastsorted ].link = list 2 ;
        else
            S [ lastsorted ] .link = list 1 ;
    }
```

تحلیل استفاده از فضای اضافی برای الگوریتم ۷-۴(مرتب سازی ادغامی ۴)

- در هر حالت استفاده از فضای اضافی به$\theta (n)$ پیوند تعلق دارد.
- منظور از $\theta (n)$ پیوند این است که تعداد پیوند ها به $\theta (n)$ تعلق دارد.

۷-۵ نگاهی دوباره به مرتب سازی سریع

```
void quicksort ( index low , index high )
  {
      index pivotpoint ;
      if ( high > low ) {
          partition ( low , high , pivotpoint ) ;
          quicksort ( low , high , pivotpoint – 1);
          quicksort (pivotpoint + 1 , high );
      }
  }
```

روش های بهبود بخشیدن به الگوریتم مرتب سازی سریع

- استفاده از فضای اضافی در مرتب سازی سریع را می توان به پنج شیوه کاهش داد:

۱- در روال quicksort ، تعیین می کنیم کدام زیر آرایه کوچک تر است و همواره آن را در پشته قرار می دهیم و دیگری را نگه می داریم.

- در این نسخه استفاده از فضای اضافی در بدترین حالت به $\theta(\lg n)$ اندیس تعلق دارد.

۲- نسخه ای از partition وجود دارد که تعداد انتساب های رکورد ها را به طرزی چشمگیر کاهش می دهد.

- برای این نسخه، پیچیدگی زمانی تعداد انتساب های انجام شده توسط مرتب سازی سریع در حالت میانگین عبارت است از:

$$A(n) = 0.69 (n + 1) \lg n$$

۳- مقدار زیادی از اعمال pop و push بی مورد است. می توانیم با نوشتن quicksort به شیوه تکرار و دستکاری پشته در روال، ازعملیات بیهوده پرهیز کنیم، یعنی به جای استفاده از پشته و بازگشتی ، پشته را خودمان می سازیم.

۴- الگوریتم های بازگشتی مثل مرتب سازی سریع را می توان با تعیین یک مقدار آستانه که در آن الگوریتم به جای تقسیم بیشتر نمونه ، یک الگوریتم تکراری را فراخوانی می کند ، بهبود بخشید.

۵- انتخاب میانه S [low] ، S [⌊ low + high)/2⌋ ، S [high] برای نقطه ی محوری است.

۶-۷ مرتب سازی heap

- درخت دودویی کامل یک درخت دودویی است که واجد شرایط زیر باشد:
  - همه ی گره های داخلی دو فرزند داشته باشند.
  - همه ی برگ ها دارای عمق d باشند.
- درخت دودویی اساسا کامل یک درخت دودویی است که:
  - تا عمق $(n - 1)$ یک درخت دودویی کامل باشد.

- گره هایی با عمق d تا حد امکان در طرف چپ واقع شوند.
- heap یک درخت دودویی اساس کامل است به طوری که:
- مقادیر نگهداری شده در گره های آن از یک مجموعه مرتب باشند.
- مقادیر نگهداری شده درهر گره بزرگ تر یا مساوی مقادیر نگهداری شده در فرزندان آن باشد.

۷-۶-۱ پیاده سازی مرتب سازی heap

ساختمان داده هایheap

```
struct heap
  {
     keytype  S [1..n];
     int heapsize ;
  };
  void  siftdown ( heap& H , index i )
  {
    index parent, largerchild; keytype siftkey;
    bool spotfound;
    siftkey = H.S[i];
    parent = i ; spotfound = false;
    while (2* parent <= H.hepsize &&! spotfound)  {
       if ( 2 * parent < H.heapsize && H.S [ 2*parent]
```

```
            largerchild  = 2 * parent + 1;
        else
            largerchild = 2 * parent ;
if ( siftkey < H.S[largerchild ])  {
        H.S [parent] = H.S[largerchild ];
        parent = largerhild ;
      }
       else
          spotfound  = true;
    }
     H.S[parent] = siftkey;
  }
    keytype  keyout ;
    keyout = H.S [1];
    H.S[1] = H.S[heapsize];
    H.heapsize =  H.heapsize -1;
    siftdown (H , 1 );
    return keyout;
  }
  void  removekeys  ( int n,
                      heap& H
```

```
                    keytype S[ ] )
{
   index i ;
   for ( i = n; i >= 1 ; i --)
        S [i] = root (H);
}
void makeheap ( int n,
                    heap& H)
{
      index i ;
      H.heapsize = n ;
    for ( i = ⌊ n/2⌋ ; i >= 1 ; i --)
      siftdown  (H,i);
}
```

الگوریتم ۷-۵: **مرتب سازی heap**

```
void  heapsort ( int n , heap& H)
{
    makeheap  ( n, H );
    removekeys ( n, H , H.S );
 }
```

۷-۷مقایسه مرتب سازی ادغامی، مرتب سازی سریع ومرتب سازی heap

- مرتب سازی سریع معمولا بر مرتب سازی heap ارجحیت دارد.
- مرتب سازی سریع معمولا بر مرتب سازی ادغامی ترجیح داده می شود.

۷-۸-۱ درخت های تصمیم گیری برای الگوهای مرتب سازی

- **لم ۷-۱:** متناظر با هر الگوریتم قطعی برای مرتب سازی n کلید متمایز، یک درخت تصمیم گیری دودویی ، معتبر و هرس شده با دقیقا n! برگ وجود دارد.
- **لم ۷-۲:** تعداد مقایسه های انجام شده توسط یک درخت تصمیم گیری در بدترین حالت برابر با عمق درخت است.
- **لم ۷-۳:** اگر m تعداد برگ ها در یک درخت دودویی و d عمق آن باشد، داریم:

$$d >= \lceil \lg m \rceil$$

قضیه ۷-۲

- هر الگوریتم قطعی که n کلید متمایز را فقط با مقایسه کلید ها مرتب سازی می کند، باید در بدترین حالت حد اقل ⌈ lg (n!) ⌉ مقایسه کلید ها را انجام دهد.
- **لم ۷-۴:** به ازای هر عدد مثبت و صحیح n، داریم:

$$\lg(n!) >= n \lg n - 1.45\, n$$

قضیه ۷-۳

هر الگوریتم قطعی که n کلید متمایز را فقط با مقایسه ی کلید ها انجام می دهد باید در بدترین حالت حداقل ⌈n lg n – 1.45n⌉ مقایسه کلید ها را انجام دهد.

۷-۸-۲حدود پایینی برای رفتار در حالت میانگین

- **لم ۷-۵:** متنا ظر با هر درخت تصمیم گیری دودویی معتبر هرس شده برای مرتب سازی n کلید متمایز، یک درخت ۲- تصمیم گیری معتبر هرس شده وجود دارد که حداقل به اندازه درخت اول کارایی دارد.
- **لم ۷-۶:** هر الگوریتم قطعی که n کلید متمایز را فقط با مقایسه کلید ها مرتب سازی می کند، حداقل تعداد محاسبه کلیدهای که انجام می دهد به طور میانگین برابر است با:

$$mimEPL(n!) / n!$$

- **لم ۷-۷:** هر درخت دودویی۲- که دارای m برگ باشد و EPL آن برابر minEPL(m) باشد، باید همه ی برگ های آن در پایین ترین سطح باشند.
- **لم ۷-۸:** هر درخت ۲- که دارای m برگ وEpl آن برابر minEPL(m) باشد ، باید 2^d – m برگ در سطح d – 1 و 2m – 2 ^d برگ درسطح d داشته باشد و برگ دیگری نداشته باشد، d عمق درخت است.

- **لم ۷-۹:** به ازای هر درخت ۲- که m برگ و EPL آن برابربا minEPL(m) باشد، عمق d عبارت است از:

$$d = \lceil \lg m \rceil$$

- **لم ۷-۱۰:** به ازای همه ی مقادیر $m >= 1$ داریم :

$$\text{minEPL}(M) >= m \lfloor \lg m \rfloor$$

قضیه ۷-۴

- هر الگوریتم قطعی که n کلید متمایز را تنها با مقایسه کلیدها مرتب سازی کند، به طور میانگین باید حداقل این تعداد مقایسه ها را انجام دهد:

$$\lfloor n \lg n - 1.45n \rfloor$$

۷-۹ مرتب سازی از طریق توزیع (مرتب سازی مبنایی)

الگوریتم ۷-۶: **مرتب سازی مبنایی**

```
void radixsort ( node_pointer& mastrrlist;
                 int  numdigits)
 {
     index i ;
     node_pointer  list [0..9]
     for ( i = 1 ; i <= numdigits ; i ++)  {
      disteibute (i);
      coalesce ;
     }
  }
   void distrebute ( index i )
  {
      index  j ;
      node_pointer  p ;
      for ( j = 0 ; j <= 9 ; j++)
            list [j] = NULL;
    p = masterlist ;
    while ( p != NULL)
 {
```

```
        j = value of the digit (from the right)in p -> key;
        link p to the end of list [j];
        p = p ->  link ;
      }
  }
  void  coalesce()
{
   index j;
   masterlist  = NULL;
   for ( j = 0 ; j <= 9 ; j++)
     link the nodes in list [j] to the end of masterlist;
 }
```