Dynamic Programming

برنامه‌ریزی پویا

طراحی الگوریتم‌ها – جلسه دوازدهم

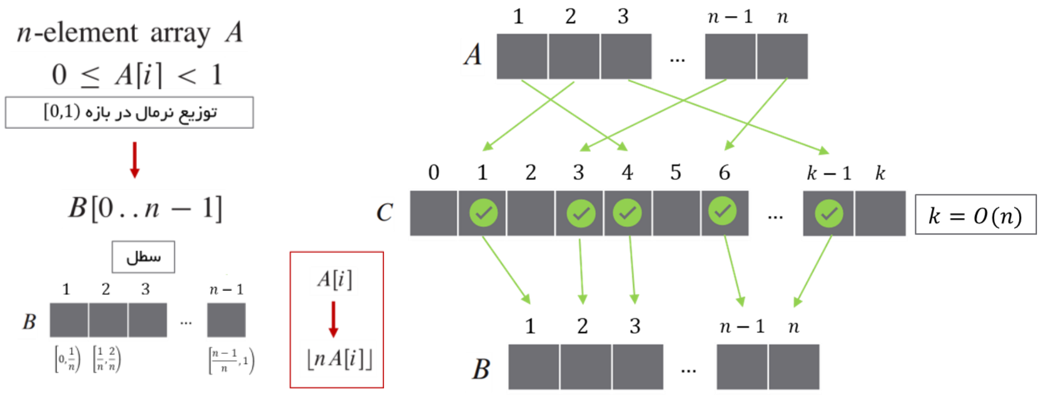
Introduction to Algorithm

استاد: جوانمردی

۱۳۹۹/۸/۱۲

مرور جلسه قبل

مرتب‌سازی سطلی BUCKET SORT



مرتب‌سازی خطی RADIX SORT

اطلاعات مازاد: اعداد دارای d رقم و هر رقم k مقدار متفاوت

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

```
RADIX-SORT(A, d)
for i = 1 to d
    use a stable sort to sort array A on digit i
```

- زمان اجرا: $\theta(d(n+k))$
- اگر $k = O(n)$ و $d = O(1)$ باشد زمان اجرای radix sort $\theta(n)$ خواهد بود

تحلیل زمانی BUCKET SORT

```
BUCKET-SORT(A)
let B[0..n-1] be a new array
n = A.length
for i = 0 to n-1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊nA[i]⌋]
for i = 0 to n-1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n-1] together in order
```

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

n_i متغیر تصادفی نشانگر تعداد المان‌ها در سطل B_i

$$= \Theta(n) + \sum_{i=0}^{n-1} O\left(2 - \frac{1}{n}\right) = \Theta(n)$$

تحلیل زمانی RADIX SORT

- اگر n عدد b بیتی داشته باشیم، برای هر عدد مثبت دلخواه $r \le b$ خواهیم داشت:

زمان اجرایی radix sort برای این اعداد $\theta((b/r)(n+2^r))$ خواهد بود

به شرطی که مرتب‌سازی پایدار استفاده شده $\theta(n+k)$ باشد

برای اعداد n و b تعیین $r \le b$ بگونه‌ای که زمان اجرای $(b/r)(n+2^r)$ را کمینه کند

حالت اول: $b < \lfloor \lg n \rfloor$ ← برای $r = b$ خواهیم داشت: $\Theta(n)$

حالت دوم: $b \ge \lfloor \lg n \rfloor$ ← برای $r = \lfloor \lg n \rfloor$ خواهیم داشت: $\Theta(bn/\lg n)$

درصورتی که $b = O(\lg n)$ باشد با انتخاب $r = \lfloor \lg n \rfloor$ زمان اجرای radix sort برابر $O(n)$

فصل ۱۵ کتاب

- برنامه‌نویسی پویا
  - مسئله برش میله
  - ضرب زنجیره‌ای ماتریس
  - المان‌های برنامه‌نویسی پویا
  - طولانی‌تر زیر رشته مشترک
  - درخت دودویی جستجو بهینه

برنامه نویسی پویا Dynamic Programming

- یک روش برای حل مسئله با ترکیب حل زیرمسئله‌ها
- مشابه روش‌های تقسیم و حل

روش تقسیم و حل	روش برنامه‌نویسی پویا
تقسیم مسئله به زیرمسئله‌های مجزا و حل بازگشتی زیرمسئله‌ها ترکیب جواب‌ها و حل مسئله اصلی	تقسیم مسئله به زیرمسئله‌هایی با هم‌پوشانی حل هر زیرمسئله فقط یکبار و ذخیره‌ی نتیجه استفاده از نتایج قبلی برای حل مسئله بالاتر

- استفاده برنامه‌نویسی پویا در مسئله‌های بهینه سازی
- بهینه سازی: راه حل‌های متعددی برای حل مسئله دارد و ما جواب بهینه را میخواهیم
- هر جواب یک مقداری دارد و جواب بهینه ماکزیمم یا مینیمم مقدار را داشته باشد

مراحل برنامه‌نویسی پویا

1. مشخص کردن ساختار یک جواب بهینه
2. تعیین مقدار برای یک جواب بهینه بصورت بازگشتی
3. محاسبه مقدار یک جواب بهینه، معمولا بصورت پایین به بالا
4. جواب بهینه را از روی اطلاعات محاسبه شده بدست بیاور

پایه‌های برنامه‌نویسی پویا

اگر فقط مقدار بهینه را میخواهیم و نه خود جواب بهینه، میتوان از مرحله چهارم صرف نظر کرد
برای بدست آوردن جواب بهینه میبایست برخی اطلاعات مازاد در مراحل قبلی نگهداری شوند

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

مسئله برش میله Rod Cutting

- برش یک میله بلند به قطعات کوچکتر به گونه ای که قیمت فروش آن بیشنیه شود!

- ورودی‌ها:
  - طول میله اولیه n
  - قیمت فروش بر حسب طول p_i
  - قیمت برش رایگان

- خروجی‌ها:
  - بیشترین درآمد قابل حصول r_n
  - طول برش‌های مورد نیاز

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

جدول قیمت فروش بر حسب طول

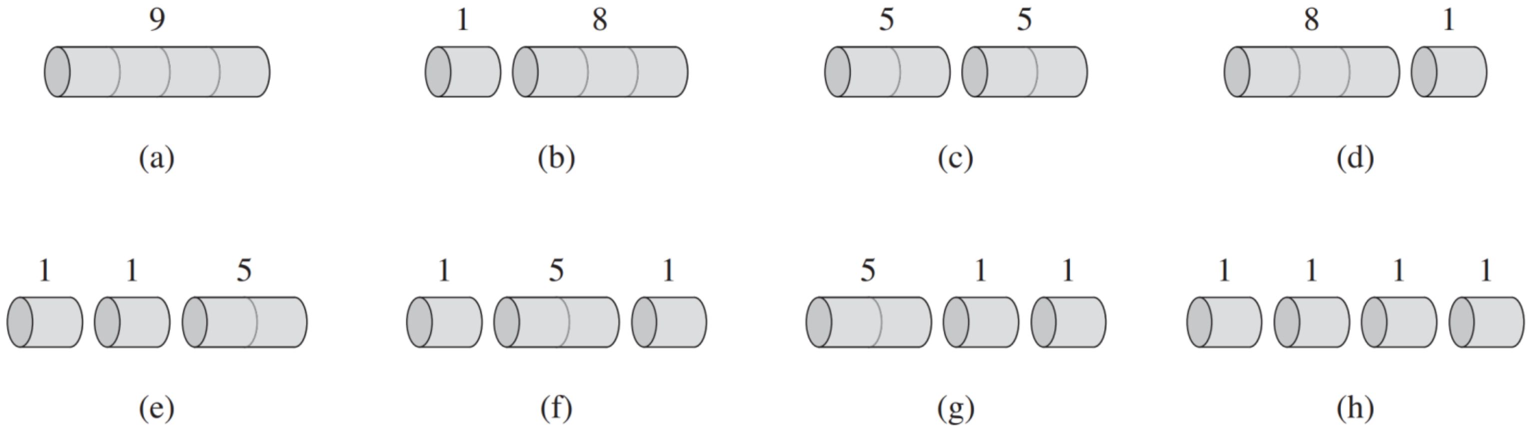
مثال

$$n = 4 \longrightarrow p_2 + p_2 = 5 + 5 = 10$$

مسئله برش میله Rod Cutting

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

جدول قیمت فروش بر حسب طول



حالت‌های ممکن برای برش میله به طول ۴

- تعداد حالات ممکن برای برش یک میله به طول n؟

فرمول

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k (b)^{n-k}$$

partition function $e^{\pi\sqrt{2n/3}}/4n\sqrt{3}$

جواب بهینه با مسئله برش میله

- اگر جواب بهینه میله را به k قسمت تقسیم کند، در این صورت برای برخی $1 \le k \le n$ خواهیم داشت:

$$n = i_1 + i_2 + \cdots + i_k$$

بگونه‌ییکه در آمد حاصله بیشینه باشد:

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$$

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

جدول قیمت فروش بر حسب طول

$$\begin{aligned}
r_1 &= 1 && \text{from solution } 1 = 1 \quad \text{(no cuts)}, \\
r_2 &= 5 && \text{from solution } 2 = 2 \quad \text{(no cuts)}, \\
r_3 &= 8 && \text{from solution } 3 = 3 \quad \text{(no cuts)}, \\
r_4 &= 10 && \text{from solution } 4 = 2 + 2, \\
r_5 &= 13 && \text{from solution } 5 = 2 + 3, \\
r_6 &= 17 && \text{from solution } 6 = 6 \quad \text{(no cuts)}, \\
r_7 &= 18 && \text{from solution } 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3, \\
r_8 &= 22 && \text{from solution } 8 = 2 + 6, \\
r_9 &= 25 && \text{from solution } 9 = 3 + 6, \\
r_{10} &= 30 && \text{from solution } 10 = 10 \quad \text{(no cuts)}.
\end{aligned}$$

به دست آوردن ساختار جواب بهینه

- بیشترین در آمد حاصل از برش یک میله به طول n (r_n)
با فرض داشتن بیشترین درآمد حاصل از برش میله‌های کوچکتر ($r_1, r_2, \ldots, r_{n-1}$):

$$r_n = \max\left(\underbrace{p_n}, \underbrace{r_1 + r_{n-1}}, \underbrace{r_2 + r_{n-2}}, \ldots, \underbrace{r_{n-1} + r_1}\right)$$

بدون برش

اولین برش با طول ۱

اولین برش با طول ۲

اولین برش با طول n-1

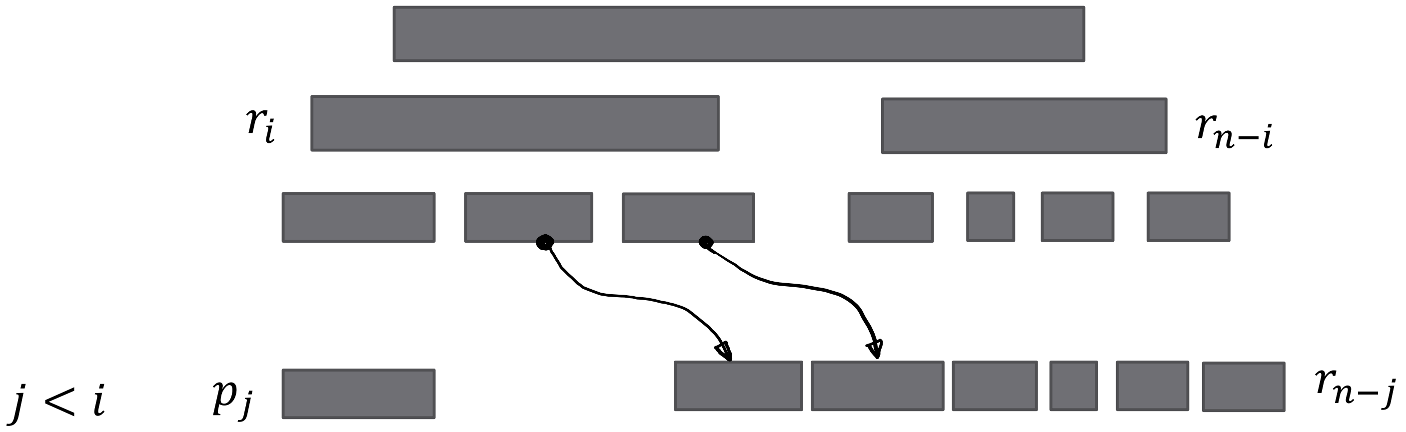
در ادامه، برش طرفین میله بصورت بهینه!

زیرساختار بهینه:
جواب بهینه مسئله را می‌توان از جواب بهینه زیرمسئله‌ها به دست آورد

ساختار کلی جواب بهینه

- بیشترین در آمد حاصل از برش یک میله به طول n (r_n)
با فرض داشتن بیشترین درآمد حاصل از برش میله‌های کوچکتر ($r_1, r_2, \dots, r_{n-1}$):

$$r_n = \max\left(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1\right)$$



کلی حالت‌های تکراری خواهیم داشت.. برش فقط یک طرف در ادامه همه حالات را ایجاد میکند!

کمی ساده‌تر

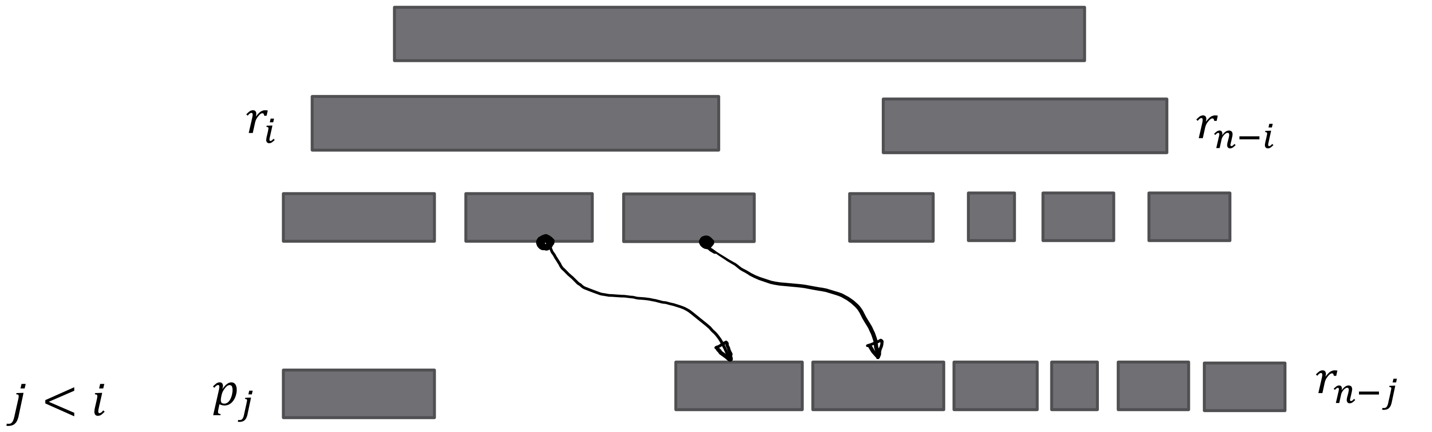
ساختار کلی جواب بهینه

$$r_n = \max_{1<i<n} (p_i + r_{n-i}))$$

ساختار کلی جواب بهینه

- بیشترین در آمد حاصل از برش یک میله به طول n (r_n)
با فرض داشتن بیشترین درآمد حاصل از برش میله‌های کوچکتر ($r_1, r_2, \ldots, r_{n-1}$):

$$r_n = \max\left(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\right)$$



کلی حالت‌های تکراری خواهیم داشت.. برش فقط یک طرف در ادامه همه حالات را ایجاد میکند!

کمی ساده‌تر

ساختار کلی جواب بهینه

$$r_n = \max_{1<i<n}\left(p_i + r_{n-i})\right)$$

حل بالا به پایین مسئله برش میله

- حل بالا به پایین مسئله بصورت بازگشتی

Recursive top-down approach

ساختار کلی جواب بهینه

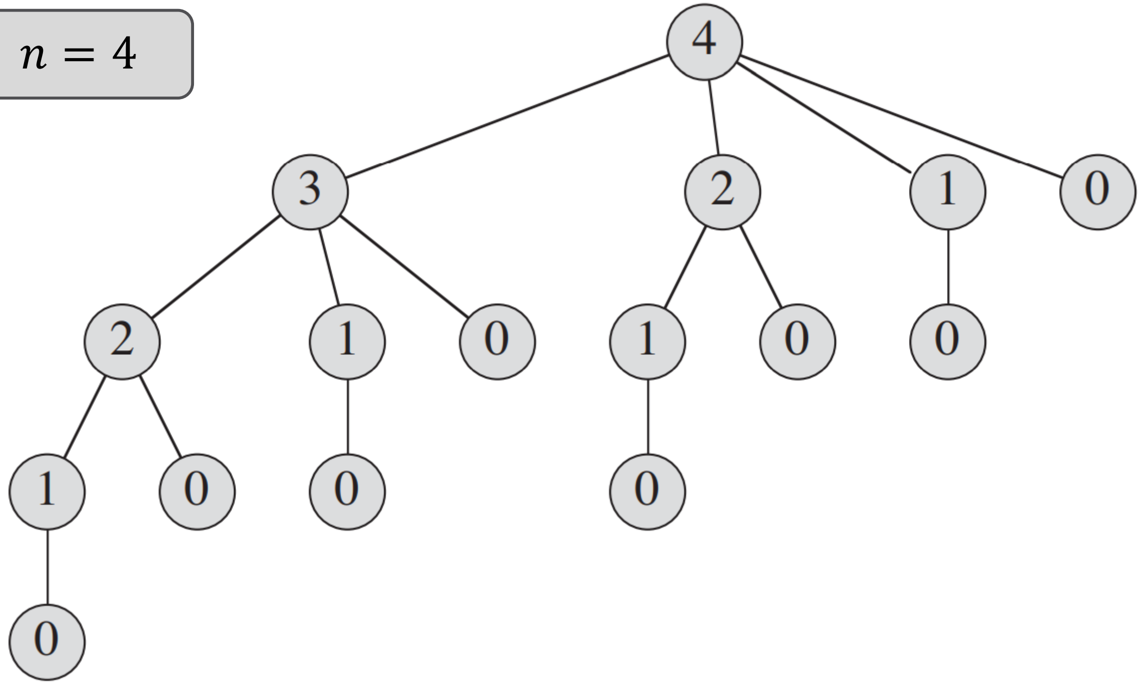
$$r_n = \max_{1<i<n} (p_i + r_{n-i}))$$

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

بسیار زمان‌بر است! چرا؟؟

تحلیل زمان اجرا حل بالا به پایین مسئله برش میله

$n = 4$




```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

$$\rightarrow$$

$$T(n) = 2^n$$

برنامه نویسی پویا و مسئله برش میله

- ایده اصلی: کاری کنیم که هر زیرمسئله فقط یکبار حل شود!
- اگر در ادامه به زیرمسئله تکراری نیاز داریم از جواب قبلی استفاده کنیم نه محاسبه مجدد
- نیاز به حافظه اضافی برای نگهداری جواب‌های محاسبه شده (time-memory trade-off)

- روش‌های پیاده‌سازی

۱. حل بالا به پایین با حفظ کردن

ساختار کلی مشابه روش قبل

قبل از حل هر تابع بازگشتی چک میکند
که آیا قبلا محاسبه شده یا نه

اگر محاسبه شده فقط استفاده میکند
در غیر اینصورت محاسبه و ذخیره میکند

۲. حل پایین به بالا

حل مسئله از کوچک به بزرگ
شروع از کوچکترین زیرمسئله

برای حل هر زیرمسئله مقادیر کوچکتر آن
قبلا محاسبه و ذخیره شده

حل مسئله بزرگتر با مقادیر کوچکتر

برنامه نویسی پویا و مسئله برش میله

۱. حل بالا به پایین با حفظ کردن

ساختار کلی مشابه روش قبل

قبل از حل هر تابع بازگشتی چک میکند
که آیا قبلا محاسبه شده یا نه

اگر محاسبه شده فقط استفاده میکند
در غیر اینصورت محاسبه و ذخیره میکند

MEMOIZED-CUT-ROD(p, n)

```
let r[0..n] be a new array
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

MEMOIZED-CUT-ROD-AUX(p, n, r)

```
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

برنامه نویسی پویا و مسئله برش میله

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

۲. حل پایین به بالا

حل مسئله از کوچک به بزرگ
شروع از کوچکترین زیرمسئله

برای حل هر زیرمسئله مقادیر کوچکتر آن قبلا محاسبه و ذخیره شده

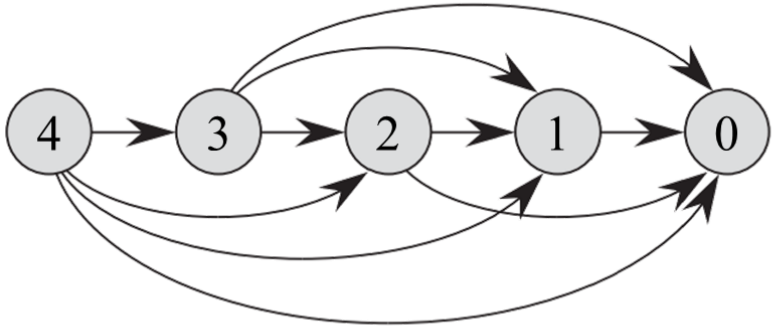
حل مسئله بزرگتر با مقادیر کوچکتر

- تحلیل زمانی هر دو روش: به دلیل حلقه تو در تو $\Theta(n^2)$
- روش پایین به بالا دارای ثابت کوچکتر و در نتیجه سریع‌تر

درخت زیرمسئله‌ها

- برای حل مسائل بصورت برنامه‌نویسی پویا باید رابطه بین زیرمسئله‌ها را بدانیم
- درخت زیرمسئله‌ها میتواند اطلاعات کافی در این زمینه به ما بدهد

درخت زیرمسئله‌ها برای $n = 4$



- یال جهتدار از گره x به گره y یعنی:
  حل بهینه x نیازمند جواب بهینه y
- این گراف فشرده شده گراف حل بالا به پایین بازگشتی
- در روش پایین به بالا، گره y که از x به آن یالی وجود دارد باید قبل از حل گره x حتما حل شده باشد

Topological sort Sect. 21

بازسازی جواب بهینه

- تا الان: محاسبه بیشترین درآمد ممکن!
- اما روش بدست آوردن بیشترین درآمد؟ ← لیستی از طول قطعات
- نیازمند ذخیره‌سازی نه تنها مقدار بهینه
بلکه انتخاب بهینه!
- انتخاب بهینه:

طول قطعه حاصل از اولین برش در حالت بهینه

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        if q < p[i] + r[j - i]
            q = p[i] + r[j - i]
            s[j] = i
    r[j] = q
return r and s
```

بازسازی جواب بهینه

- تا الان: محاسبه بیشترین درآمد ممکن!
- اما روش بدست آوردن بیشترین درآمد؟ ← لیستی از طول قطعات

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]
4      n = n - s[n]
```

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10