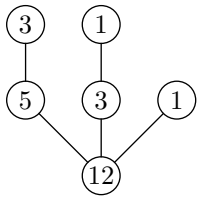
December 22, 2006 at 21:08

1. Overview. This is a solution to the LANDSCAP SPOJ problem. It is also my first `CWEB` program.

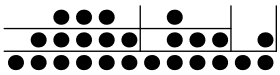
2. A 'landscape' is an array of N 'elevations', where $1 \le N \le 1000$. We must level the landscape so that $\le L$ 'peaks' remain, where $1 \le L \le 25$. A peak is a constant region of the landscape that is delimited on both sides by a smaller elevation or by the edge of the landscape.

For the specification of the input/output format, an example, and other details please see the problem statement at SPOJ.

3. In the first phase the program transforms the landscape into a rooted and weighed tree. For example



comes from the following partition of the sample input:



Each node corresponds to a rectangular region of the landscape. Its weight is the area of the rectangular region. Its children are the rectangular regions that directly sit on it. We choose the height of the rectangular regions greedily maximal.

If we denote by $A(n)$ the maximum number of nodes that can be obtained from a landscape of length n we have

$$A(n) = \max_{n_1+\cdots+n_d \le n-d} A(n_1) + \cdots + A(n_d) + 1 \qquad \text{for some } d \ge 0$$

We can now prove by induction that $A(n) \le n$.

For the running time we have the following recurrence.

$$B(n) = B(n_1) + \cdots + B(n_d) + O(n)$$

We can now prove that $B(n) = O(n^2)$. Here is a sketch of a proof that assumes $d > 0$.

$$\begin{aligned} B(n) &= B(n_1) + \cdots + B(n_d) + n \\ &\le Cn_1^2 + \cdots + Cn_d^2 + Cn \\ &\le C(n^2 - 2nd - d^2 - d(d-1) + n) \\ &= C(n^2 + (n+d)(1-2d)) \\ &\le Cn^2 \end{aligned}$$

(Is there a tighter bound?)

4. The second phase finds the answer by processing the tree. An equivalent problem is the following: What is the minimum weight we need to cut from the tree so that only $\le L$ leafs remain? In fact, we will solve the (roughly) opposite problem: What is the maximum weight of a sub-tree that has exactly l leafs? This problem is hard so we generalize to make it easier: What is the maximum weight of a sub-tree of the tree rooted in x that has exactly l leafs, provided it may use only the first c children of x? Let's denote this quantity by $m(x, l, c)$. In the following formulæ the weight of x is denoted by w, the c-th child of x is denoted by y, and the number of children of y is denoted by d.

$$m(x, 0, c) = 0$$

$$m(x, l, c) = \max\left(w + m(y, l, d), \max_{0<k\le l}\big(m(x, k, c-1) + m(y, l-k, d)\big)\right) \qquad \text{for } c > 0 \text{ and } l > 0$$

$$m(x, l, 0) = \begin{cases} 0, & \text{for } l = 0 \\ w, & \text{for } l = 1 \\ -\infty, & \text{otherwise} \end{cases}$$

In general, for a node x we will denote its weight by w_x and the number of its children by d_x.

5. How many m-values are there? The node x can take N values; the number of allowed leafs can take $L + 1$ values, from 0 to L; the children limit c is nonnegative and $\le d_x$.

$$\sum_{x=0}^{N-1}\sum_{l=0}^{L}\sum_{c=0}^{d_x} 1 = (L+1)\sum_{x=0}^{N-1}(d_x + 1) = (L+1)(2N-1)$$

(The number of edges in a tree with N nodes is $N - 1$.) So there are $(L + 1)(2N - 1)$ values to compute. Each value is computed in $\le L$ steps, therefore all are computed in $O(NL^2)$ steps.

6. The overall structure of the code reflects the two phase process—first construct a tree out of the landscape and then process the tree.

```
#define max_N 1024      /* conservative strict upper limit for N */
#define max_L 32        /* conservative strict upper limit for L */
⟨ Includes 16 ⟩
using namespace std;
⟨ Preprocessor definitions ⟩
int land[max_N];        /* the landscape */
int N;      /* landscape length */
int L;      /* maximum number of peaks */
int *m[max_N][max_L];       /* the values m(x, l, c) presented above */
⟨ Tree data 7 ⟩
⟨ List operations 15 ⟩;
⟨ Tree construction 8 ⟩
int main( )
{
  int tests; cin >> tests;
  while (tests--) {
    ⟨ Local variables 10 ⟩;
    ⟨ Read input and construct tree 9 ⟩;
    ⟨ Allocate memory for m 13 ⟩;
    ⟨ Compute m 11 ⟩;
    ⟨ Print solution 12 ⟩;
    ⟨ Cleanup 14 ⟩;
  }
}
```

7. Tree construction. The tree can be represented by an array of lists with children. (A matrix representation would require $\Omega(N^2)$ space while this approach uses $\Theta(N)$ space.)

```
⟨ Tree data 7 ⟩ ≡
  struct Node {
    Node(Node *n, int c) : n(n), c(c) { }
    int c;      /* child */
    Node *n;    /* next */
  };
  Node *children[max_N];    /* children[x] is a linked list with the children of x */
  int d[max_N];    /* number of children (d[x] is the length of the list children[x]) */
  int w[max_N];    /* node weights */
  int nodes_cnt;    /* number of nodes */
```

This code is used in section 6.

8. The construction is done recursively by a function that processes the interval $[a..b)$ of *land*.

```
#define infty 0x7fffffff    /* almost infinity */
⟨ Tree construction 8 ⟩ ≡
  void make_tree(int a, int b)
  {
    int i, j;
    int n = nodes_cnt++;    /* current node */
    int m = infty;
    for (i = a; i < b; ++i) m = min(m, land[i]);
    for (i = a; i < b; ++i) land[i] -= m;
    w[n] = m * (b - a);
    j = a;
    while (true) {
      for (i = j; i < b ∧ ¬land[i]; ++i) ;
      if (i ≡ b) return;
      for (j = i; j < b ∧ land[j]; ++j) ;
      children[n] = add(children[n], nodes_cnt); ++d[n];
      make_tree(i, j);
    }
  }
```

This code is used in section 6.

9. The tree construction is done immediately after reading the input.

```
⟨ Read input and construct tree 9 ⟩ ≡
  cin ≫ N ≫ L;
  for (int i = 0; i < N; ++i) cin ≫ land[i];
  make_tree(0, N);
```

This code is used in section 6.

10. Tree processing. The values m can be computed by dynamic programming. Note that below we use -1 instead of $-\infty$; that should be small enough. We also use that all children of x are $> x$.

```
⟨ Local variables 10 ⟩ ≡
  int x;      /* current node */
  Node *y;      /* current child */
  int l;      /* limit on number of leafs for the first c children */
  int k;      /* limit on number of leafs for the first c − 1 children */
  int c;      /* current number of considered children */
```

This code is used in section 6.

11.

```
⟨ Compute m 11 ⟩ ≡
  for (x = 0; x < nodes_cnt; ++x) {
    m[x][0][0] = 0; m[x][1][0] = w[x];
    for (l = 2; l ≤ L; ++l) m[x][l][0] = −1;
  }
  for (x = nodes_cnt − 1; x ≥ 0; −−x) {
    y = children[x];
    for (c = 1; c ≤ d[x]; ++c) {
      m[x][0][c] = 0;
      for (l = 1; l ≤ L; ++l) {
        m[x][l][c] = w[x] + m[y→c][l][d[y→c]];
        for (k = 1; k ≤ l; ++k) m[x][l][c] = max(m[x][l][c], m[x][k][c − 1] + m[y→c][l − k][d[y→c]]);
      }
      y = y→n;
    }
  }
```

This code is used in section 6.

12. The minimum weight that needs to be cut form the tree to obtain a tree with $\le L$ leafs is obtained by subtracting from the total weight the maximal weight of a tree with 0, 1, ..., or L leafs.

```
⟨ Print solution 12 ⟩ ≡
  int total_weight = 0;
  int max_weight = −1;
  for (x = 0; x < nodes_cnt; ++x) total_weight += w[x];
  for (l = 0; l ≤ L; ++l) max_weight = max(max_weight, m[0][l][d[0]]);
  cout ≪ (total_weight − max_weight) ≪ endl;
```

This code is used in section 6.

13. Memory management and list operations. We need to allocate memory for m after we construct the tree (so that we know how big the individual arrays should be).

```
⟨ Allocate memory for m 13 ⟩ ≡
  for (l = 0; l ≤ L; ++l) {
    for (x = 0; x < nodes_cnt; ++x) m[x][l] = (int *) malloc((d[x] + 1) * sizeof(int));
  }
```

This code is used in section 6.

14. At the end we must free this memory plus the memory used for representing the adjacency lists. Also, we reset all the other data structures that are expected to be reset when starting to solve a test case.

```
⟨ Cleanup 14 ⟩ ≡
  for (x = 0; x < nodes_cnt; ++x) {
    del(children[x]); children[x] = Λ;
    for (l = 0; l ≤ L; ++l) free(m[x][l]);
  }
  nodes_cnt = 0;
  memset(d, 0, sizeof (d));
  memset(children, 0, sizeof (children));
```

This code is used in section 6.

15. The only two list operations are inserting an element and deleting the whole lost. The brevity of these two functions is an argument that list operations are easy.

```
⟨ List operations 15 ⟩ ≡
  Node *add(Node *p, int c)
  {
    return new Node(p, c);
  }
  void del(Node *p)
  {
    Node *n;
    while (p) {
      n = p->n;
      delete p;
      p = n;
    }
  }
```

This code is used in section 6.

16. Loose ends.

⟨ Includes 16 ⟩ ≡
```
#include <iostream>
#include <vector>
#include <algorithm>      /* for min */
#include <cstdlib>
```
This code is used in section 6.

a: 8.
add: 8, 15.
b: 8.
c: 7, 10, 15.
children: 7, 8, 11, 14.
cin: 6, 9.
cout: 12.
d: 7.
del: 14, 15.
endl: 12.
free: 14.
i: 8, 9.
infty: 8.
j: 8.
k: 10.
L: 6.
l: 10.
land: 6, 8, 9.
m: 6, 8.
main: 6.
make_tree: 8, 9.
malloc: 13.
max: 11, 12.
max_L: 6.
max_N: 6, 7.
max_weight: 12.
memset: 14.
min: 8, 16.
N: 6.
n: 7, 8, 15.
Node: 7, 10, 15.
nodes_cnt: 7, 8, 11, 12, 13, 14.
p: 15.
std: 6.
tests: 6.
total_weight: 12.
true: 8.
w: 7.
x: 10.
y: 10.

⟨ Allocate memory for m 13 ⟩ Used in section 6.
⟨ Cleanup 14 ⟩ Used in section 6.
⟨ Compute m 11 ⟩ Used in section 6.
⟨ Includes 16 ⟩ Used in section 6.
⟨ List operations 15 ⟩ Used in section 6.
⟨ Local variables 10 ⟩ Used in section 6.
⟨ Print solution 12 ⟩ Used in section 6.
⟨ Read input and construct tree 9 ⟩ Used in section 6.
⟨ Tree construction 8 ⟩ Used in section 6.
⟨ Tree data 7 ⟩ Used in section 6.

LANDSCAP

	Section	Page
Overview	1	1
Tree construction	7	3
Tree processing	10	4
Memory management and list operations	13	5
Loose ends	16	6